UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

CURSO DE MESTRADO EM INFORMÁTICA

UM PACOTE DE ARITMETICA DE MÚLTIPLA PRECISÃO

MARIO AUGUSTO BESSA DE FIGUEIRÊDO

CAMPINA GRANDE - PB

AGOSTO - 1989

MARIO AUGUSTO BESSA DE FIGUEIRÊDO

UM PACOTE DE ARITMÉTICA DE MÚLTIPLA PRECISÃO

Dissertação apresentada ao curso de MESTRADO EM INFORMÁTICA da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do Grau de Mestre.

ÁREA DE CONCENTRAÇÃO: CIÊNCIA DA COMPUTAÇÃO

MARIO TOYOTARO HATTORI

Orientador

CAMPINA GRANDE - PB

AGOSTO - 1989

F475p Figueiredo, Mario Augusto Bessa de
Um pacote de aritmetica de multipla precisao / Mario Augusto Bessa de Figueiredo. - Campina Grande, 1989.
95 f.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Aritmetica - 2. Aritmetica em Multipla Precisao 3. Dissertacao I. Hattori, Mario Toyotaro, M.Sc. II. Universidade Federal da Paraiba - Campina Grande (PB) III. Título

CDU 519.612(043)

UM PACOTE DE ARITMÉTICA DE MÚLTIPLA PRECISÃO

MARIO AUGUSTO BESSA DE FIGUEIREDO

Dissertação aprovada em 30/08/89

MARIO TOYOTARO HATTORI - M.Sc

- Presidente -

GREGÓRIO MARANGUAPE DA CUNHA - Dr

- Examinador -

CAMILO DE LÉLIS GONDIM MEDEIROS - M.Sc

- Examinador -

CAMPINA GRANDE

AGOSTO - 1989

A G R A D E C I M E N T O S

Agradeço a todos os funcionários do Departamento de Sistemas e Computação que colaboraram direta ou indiretamente para a realização deste trabalho.

Agradeço aos colegas do mestrado pela amizade e pela ajuda que me deram.

Agradeço principalmente ao meu orientador, prof. Mário Toyotaro Hattori pelo incentivo, ajuda e compreensão durante todas as fases do trabalho.

A Márcia, Mariana, Adriano, Alcier e Adriana pelo incentivo e apoio constantes.

RESUMO

Neste trabalho nos propomos a desenvolver um pacote de software de aritmética de múltipla precisão para servir de ferramenta em um ambiente de computação numérica utilizando microcomputadores compatíveis com IBM-PC. O pacote então será usado para implementar uma calculadora científica on-line, em que a precisão é flexível, permitindo ao usuário mudá-la até o limite imposto pela memória do computador.

ABSTRACT

Our purpose in this work is to develop a package of multiple precision arithmetic software to be used as a tool in an enviroment of numerical computation using IBM PC-compatible microcomputers. The package then will be used to implement an on-line scientific calculator, in which the precision is flexible, allowing to the user to change it up to the memory capacity.

CONTEÚDO

CAPÍTULO I

INTRODUÇAO

Em computação numérica uma medida tão importante quanto o tempo de execução é a precisão dos resultados. Erros computacionais normalmente são causados por diferenças entre o sistema de números reais e sua realização física em forma de circuitos eletrônicos. Em matemática, o sistema de numeros reais é infinito e continuo, não havendo imposição de limites na precisão e grandeza de números em um cálculo. Um computador por outro lado, só é capaz de manipular precisão e grandeza finitas não importando quão grande seja sua palavra. Ele não pode operar com sistemas de números reais, mas sim com uma aproximação finita embora representativa. Isso significa que o computador impõe limites inferior e superior na magnitude e precisão dos números que podem ser representados.

A segunda limitação é a possibilidade de apenas um número finito de valores poder ser representado entre os limites inferior e superior. Em lugar de valores contínuos, o computador só é capaz de representar uma série de valores discretos que são separados por pequenos intervalos. A maior parte dos erros computacionais ocorre devido a esses limites, o que nos sugere que façamos por software uma expansão de tais limites impostos pelo hardware, criando o que se conhece como **aritmética de múltipla precisão.**

No capítulo II mostraremos como os números reais podem ser representados no computador usando o modelo de ponto flutuante. Trataremos da aritmética com os números representados nesse modelo fazendo comparações com o sistema de números reais e mostraremos como o arredondamento e o truncamento são feitos.

Através de um exemplo, vamos mostrar como a limitação de precisão pode comprometer todo um cálculo e então introduziremos o conceito de dígitos de guarda, que junto com a aritmética de múltipla precisão, são ferramentas que nos auxiliam a resolver tais problemas para obter resultados com melhor precisão.

De posse do conceito de múltipla precisão e de como armazenar os números, o capítulo III apresenta as técnicas para aproximar uma função f(x), faz um resumo de aproximação polinomial, mostra porque utilizamos tal aproximação e apresenta os três métodos mais utilizados: interpolação, quadrados mínimos e aproximação racional. Cada método é descrito com algum detalhe.

Estamos agora em condições de implementar um pacote de aritmética de múltipla precisão. No capítulo IV são apresentados todos os algoritmos e estruturas de dados utilizados nesse pacote. É feita uma comparação entre os possíveis algoritmos para implementação das funções e as dificuladades encontradas na implementação de cada um. Esse pacote servirá de base para a implementação de uma calculadora científica que será mostrada no capítulo 5.

CAPÍTULO II

ARITMÉTICA EM PONTO FLUTUANTE

2.1 Introdução

Vamos começar este capítulo com uma pequena história da evolução do número, mostrando as dificuldades que existiam antes do sistema posicional, um dos triunfos da mente humana. Nesse sistema podem ser efetuados cálculos e análises que em outros sistemas seriam muito difíceis ou mesmo impossíveis.

Em seguida vamos mostrar como este sistema é representado no computador através do modelo de ponto flutuante. Depois vamos formalizar matematicamente o sistema de números em ponto flutuante, mostrando como as leis da álgebra se comportam nesse sistema.

Iremos também mostrar a aritmética com truncamento e com arredondamento para esses números e como as operações aritméticas de adição, subtração, multiplicação e divisão são realizadas introduzindo o conceito de **overflow e underflow.**

Finalmente, vamos mostrar a necessidade de múltipla precisão para os cálculos científicos e como os dígitos de guarda podem melhorar essa precisão.

2.2 Sistema Posicional

Segundo [BRUM63] os símbolos que utilizamos hoje para representar os números foram desenvolvidos primeiramente na índia. A data exata dessa aparição é incerta, aproximadamente 600 A.D. Hoje escrevemos o numeral 10 para representar o número de dedos que possuimos nas duas mãos. Os romanos e a maioria das pessoas na Europa Meridional utilizavam o símbolo X. Os egípcios utilizavam o símbolo ∩. Na antiga Babilônia o símbolo < era usado e os matemáticos gregos usavam o símbolo ι (a letra iota do seu alfabeto).

O homem hoje tem o mesmo número de dedos que tinha a 10.000 anos atrás. Números não mudam, mas podemos usar diferentes símbolos para representar um número. A maneira pela qual a aritmética é feita está diretamente ligada à maneira de se representar os números. Dessa forma, para representar números, usamos comumente um sistema posicional com base 10 (sistema decimal) e que possui dez símbolos diferentes, e a magnitude com que cada símbolo a contribui ao valor do número depende da posição de a, isto é, se a estiver na n-ésima posição à esquerda do ponto decimal, então o valor contribuído será de $a \cdot 10^{n-1}$, ou se estiver na n-ésima posição à direita do ponto decimal, o valor contribuído será de $a \cdot 10^{-n}$.

Todo número tem uma única representação da forma $\ldots + a_n b^n + a_{n-1} b^{n-1} + \ldots + a_1 b + a_0 + a_{-1} b^{-1} + a_{-2} b^{-2} + \ldots + a_{-m} b^{-m} + \ldots$ onde os coeficientes a_i's são os dígitos no sistema posicional de

base b, isto é, inteiros positivos tais que $0 \leq a_i \leq b-1$ e b, a base do sistema, é qualquer número inteiro maior ou igual a 2.

Uma das maiores vantagens do sistema posicional é de permitir usar regras simples e gerais para as operações aritméticas. Quanto menor a base, mais simples são essas regras. Por essa razão, entre outras, a maioria dos computadores operam na base 2, o sistema binário.

2.3 Representação de Números decimais em Ponto Flutuante

O computador não usa o sistema de números reais e sim um modelo aproximado denominado **sistema de números de ponto flutuante** [FORS70].

Um número em ponto flutuante pode ser representado na forma

$$x = m \cdot b^e \qquad (2.1)$$

onde

b - é a base do sistema de números, $b \geq 2$;

e - expoente. Um inteiro qualquer;

m - fração ou mantissa, $\frac{1}{b} \leq |m| < 1$.

A representação interna de um número em ponto flutuante no computador normalmente tem a forma

onde

s - é o sinal do número armazenado;

característica - expoente mais excesso. Este excesso varia de acordo com o hardware;

mantissa - é a fração m de (2.1) sem sinal.

O uso do expoente com excesso tem vantagem sobre o uso puro e simples do expoente, pois além de não necessitar da representação de seu sinal, facilita as operações aritméticas e lógicas.

Quanto mais próximo o dígito estiver do ponto decimal da mantissa, mais esse dígito é significativo. Um número em ponto flutuante está normalizado se o dígito mais significativo for diferente de zero.

O uso da representação em ponto flutuante aumenta o intervalo dos números que podem ser acomodados em um registrador [MANO76]. Por esta razão, na maioria dos computadores as operações de ponto flutuante são realizadas a nível de hardware podendo ser a nível de software em outros. Nas linguagens de alto nível quando declaramos que uma variável é do tipo real, estamos dizendo que queremos que o valor dessa variável seja representado em ponto flutuante.

2.4 Sistema de Números em Ponto Flutuante

Representamos o sistema de números em ponto flutuante por FP(b, p, a), onde

b - é a base do sistema de números;

p - é a precisão e indica o número de dígitos na base b contidos na mantissa;

a - especifica os detalhes de como a aritmética será executada, podendo ser substituído por vários símbolos, tais como c para aritmética com truncamento, r para arredondamento, etc.

FP(b, p, a) é composto por um conjunto S de números que chamaremos de números em ponto flutuante e uma definição das quatro operações aritméticas de adição, subtração, multiplicação e divisão de ponto flutuante para elementos de S. Esse conjunto S depende de b e p mas não de a. Quando b e p não são fixados pelo contexto, escrevemos S(b, p) ao invés de S. O conjunto S(b, p) contém zero e todos os números da forma (2.1) cujo valor absoluto pode ser expresso na base b usando no máximo p dígitos. Isto é

$$|m| = b^{-p} \cdot M$$

onde

M - é um inteiro contido no intervalo $b^{p-1} \leq M < b^{p}$.

Considerando que todo número em ponto flutuante é um número real, podemos executar as operações aritméticas de adição, subtração, multiplicação e divisão com os elementos de S vistos como números reais.

Dessa forma, para x e y $\in$ S podemos obter x + y, x - y, x * y e x / y. No entanto, é possível que essas operações produzam números que não estejam em S. Por exemplo, precisamos de 2p dígitos para representar o produto de dois números com p dígitos cada um, e a divisão pode produzir um resultado exigindo um número infinito de dígitos. Mesmo que os resultados da aritmética de ponto flutuante sempre estejam em S, as operações em ponto flutuante podem produzir resultados diferentes das produzidas pelas operações aritméticas no campo dos números reais. Portanto, para aritmética de ponto flutuante utilizaremos os operadores $\oplus$, $\ominus$, $\otimes$ e $\oslash$ para representarem as operações aritméticas de adição, subtração, multiplicação e divisão, respectivamente.

Na definição de FP(b, p, a) foram omitidos vários detalhes da representação de números em ponto flutuante:

a) Limites do intervalo de definição do expoente, que sempre existem, mas são particulares a cada máquina;
b) Variações na representação de como os números negativos são representados; neste trabalho assumimos que a representação da mantissa usa o sinal e magnitude;
c) Variações no ponto da mantissa, que neste trabalho assumimos estar à esquerda da mantissa.

2.5 Aritmética com Truncamento para Números em Ponto Flutuante

Representamos com FP(b, p, c) o sistema de números em ponto flutuante onde as operações $\oplus$, $\ominus$, $\otimes$ e $\oslash$ são realizadas em aritmética com truncamento.

Seja x qualquer número real, então $\overline{x}$ denota x truncado para p dígitos na base b. Mais especificamente, para qualquer número real x, seja T o conjunto de todos os números y em S(b, p) com $|y| \leq |x|$.

Vamos usar um exemplo para tornar claro. Seja FP(10, 8, c), então $\overline{.123456789} = .12345678$ e $\overline{-.123456789} = -.12345678$.

Para efetuar as operações aritméticas em FP(b, p, c), primeiro executamos as operações no sistema de números reais e então truncamos o resultado para p dígitos na base b. Portanto

$$x \oplus y = \overline{x + y};$$

$$x \ominus y = \overline{x - y};$$

$$x \otimes y = \overline{x * y};$$

$$x \oslash y = \overline{x / y}.$$

2.6 Aritmética com Arredondamento para Números em Ponto Flutuante

Representamos por FP(b, p, r) o sistema de números em ponto flutuante onde as operações $\oplus$, $\ominus$, $\otimes$ e $\oslash$ são realizadas com arredondamento.

Existem várias maneiras de se arredondar um número em ponto flutuante e a que vamos descrever aqui é considerada a mais simples. As outras regras além de mais complicadas são raramente implementadas em computadores [STER74].

Seja x um número real qualquer, então $\bar{x}^{\circ}$ denota x arredondado para p dígitos na base b.

Vamos exemplificar. Seja FP(10, 8, r), então $.123456789^{\circ}$ = .12345679, $.123456785^{\circ}$ = .12345679, $.123456783^{\circ}$= .12345678.

Para efetuar as operações aritméticas em FP(b, p, r), primeiro executamos as operações no sistema de números reais e então arredondamos o resultado para p dígitos na base b. Portanto

$$x \oplus y = \overline{x + y}^{\circ}$$
$$x \ominus y = \overline{x - y}^{\circ}$$
$$x \otimes y = \overline{x * y}^{\circ}$$
$$x \oslash y = \overline{x / y}^{\circ}$$

2.7 Leis da Álgebra

As manipulações algébricas de fórmulas estão baseadas na validade de poucas leis fundamentais. Especificamente, dispomos do fato de que os números reais formam um campo, isso significa que a soma e o produto de números reais estão definidos e que os seis axiomas seguintes são satisfeitos por quaisquer números reais a, b e c:

1) Fechamento: O produto $a \cdot b$ e a soma $a + b$ de números reais são também números reais;

2) Comutatividade: $a + b = b + a$,

$$a \cdot b = b \cdot a;$$

3) Associatividade: $(a + b) + c = a + (b + c)$,

$$(a \cdot b) \cdot c = a \cdot (b \cdot c);$$

4) Distributividade: $a \cdot (b + c) = a \cdot b + a \cdot c$;

5) Existem os números 0 e 1 tais que

$a + 0 = 0 + a = a$ (elemento neutro da adição),

$a \cdot 1 = 1 \cdot a = a$ (elemento neutro da multiplicação)

para todo a;

6) Para qualquer número real a existe um número real -a tal que $a + (-a) = (-a) + (a) = 0$ e se $a \neq 0$, existe um número real a^{-1} tal que $a \cdot a^{-1} = a^{-1} \cdot a = 1$.

Uma consequência desses axiomas é que não existe divisão por zero, isto é, se $a \cdot b = 0$, então um dos fatores a ou b deve ser zero. Uma outra consequência desses axiomas é a lei do cancelamento:

se $a \cdot b = a \cdot c$ e $a \neq 0$, então $b = c$.

Definiremos a subtração como:

$$a - b = a + (-b)$$

e se $b \neq 0$, definiremos a divisão por

$$\frac{a}{b} = a \cdot b^{-1}.$$

Duas consequências dessas definições junto com os axiomas de comutatividade e associatividade são:

$$(a + b) - b = a \qquad (2.2)$$

e

$$a \cdot \left(\frac{b}{a} \right) = b. \qquad (2.3)$$

O problema agora é saber se esses axiomas são válidos no sistema de números em ponto flutuante. Uma vez que sua validade pode depender de detalhes de como a aritmética é executada, estudaremos apenas o sistema específico FP(b, p, c).

O teorema seguinte é uma consequência imediata das definições e validades do axioma da comutatividade das operações com números reais.

Teorema 2.1: Em FP(b, p, c), a soma e o produto de dois números são também números em ponto flutuante. Também, para quaisquer a e b em S(b, p) temos

$$a \oplus b = b \oplus a,$$

$$a \otimes b = b \otimes a,$$

$$a \oplus 0 = 0 \oplus a = a,$$

$$a \otimes 1 = 1 \otimes a = a,$$

$$a \oplus (-a) = (-a) \oplus a = 0.$$

Todos os outros axiomas, ou seja, os da associatividade, da distributividade e do cancelamento, além de (2.2) e (2.3) falham em FP(b, p, c). Prova matemática em [STER74].

2.8 Desigualdades em Sistemas de Números em Ponto Flutuante

As leis fundamentais para manipulação de desigualdades no sistema de números reais são:

. Se $a < b$, então para todo $c \Rightarrow a + c < b + c$;

. Se $a < b$ e $c < d \Rightarrow a + c < b + d$;

. Se $b < c$ e $a > 0 \Rightarrow a \cdot b < a \cdot c$.

O ideal seria que essas leis também fossem aplicáveis a sistemas de números em ponto flutuante FP(b, p, c).

Primeiro, observamos que se x e y são dois números reais quaisquer com $x < y$, então $\bar{x} \leq \bar{y}$. Podemos ter $\bar{x} = \bar{y}$ mesmo quando $x < y$. Isso ocorre se os primeiros p dígitos de x e y forem iguais.

Teorema 2.2: Em FP(b, p, c) temos

. Se $a < b$, então para todo $c \Rightarrow a \oplus c \leq b \oplus c$;

. Se $a < b$ e $c < d \Rightarrow a \oplus c \leq b \oplus d$;

. Se $b < c$ e $a > 0 \Rightarrow a \otimes b \leq a \otimes c$.

Infelizmente, essas relações que são desigualdades estritas

no sistema de números reais, foram enfraquecidas pelo $\leq$ em FP(b, p, c). Prova matemática em [STER74].

2.9 Operações Aritméticas de números em Ponto Flutuante

Dados dois números x e y escritos na forma (2.1), iremos mostrar como são realizadas as operações aritméticas no sistema de números em ponto flutuante segundo [HAMA78].

2.9.1 Adição e Subtração

1. Escolher o número com menor expoente entre x e y e deslocar sua mantissa para a direita um número de dígitos igual a diferença absoluta entre os respectivos expoentes;
2. Colocar o expoente do resultado igual ao maior expoente entre x e y;
3. Executar adição/subtração das mantissas obedecendo a ordem (x + y) e (x - y) e determinar o sinal do resultado;
4. Normalizar o valor do resultado, se necessário;
5. Arredondar o valor do resultado, se necessário;
6. Verificar se houve overflow (explicado em 2.10).

2.9.2 Multiplicação

1. Colocar o expoente do resultado igual à soma dos expoentes de x e y;
2. Multiplicar as mantissas e determinar o sinal do resultado;

3. Normalizar o valor do resultado, se necessário;
4. Arredondar o valor do resultado, se necessário;
5. Verificar se houve **overflow** ou **underflow** (explicado em 2.10).

2.9.3 Divisão

1. Colocar o expoente do resultado igual a diferença dos dois expoentes, do dividendo e do divisor;
2. Dividir as mantissas obedecendo a ordem (x / y) e determinar o sinal do resultado;
3. Normalizar o resultado, se necessário;
4. Arredondar o resultado, se necessáario;
5. Verificar se houve overflow ou underflow (explicado a seguir).

2.10 Overflow e Underflow em Ponto Flutuante

Até este ponto assumimos que um número em ponto flutuante está escrito na forma (2.1). Em geral, o expoente está restrito ao intervalo

$$e_* \leq e \leq e^* . \qquad (2.4)$$

Para uma máquina que armazena o expoente com excesso, usualmente temos

$$e_* = -(e^* + 1).$$

Mas se a máquina mantém expoentes negativos como complemento de um ou sinal e magnitude, temos

$$e_* = -e^*.$$

Restringindo o intervalo do expoente, restringimos o intervalo dos números em ponto flutuante que podemos representar.

Usamos L para representar o maior número positivo em ponto flutuante que podemos representar sujeito a (2.4). De modo similar usaremos l para representar o menor número positivo normalizado em ponto flutuante que podemos representar sujeito a (2.4). Então

$$L = b^{e^*} \cdot (1 - b^{-p}) < b^{e^*}$$

e

$$l = b^{e_*} \cdot b^{-1} = b^{e_* - 1}.$$

Isso mostra uma pequena assimetria em nosso sistema de números em ponto flutuante: Existem alguns números pequenos cuja recíproca não pode ser representado porque eles são maiores do que L.

O fato dos limites e^* e e_* serem inerentes à máquina, sugere que devessem ser incluídos na definição de números em ponto flutuante. Dessa forma, ao invés de S(b, p) deveríamos ter S(b, p, e_*, e^*) que contém zero e todos os números em S(b, p) que podem ser escritos na forma (2.1). Mas continuaremos com a abordagem anterior para tratar o problema de overflow e underflow separado

dos problemas relacionados com erros de arredondamento e outras anomalias pertinentes à aritmética de ponto flutuante. Ocorrerá overflow se tentarmos produzir um número com valor absoluto maior do que L, e underflow se diferente de zero com valor absoluto menor do que 1. Em ambos os casos ocorre transbordamento de expoente.

Existem algumas maneiras de lidar com transbordamento de expoente. Elas podem envolver questões do que o hardware faz e do que o compilador faz. O que o compilador é capaz de fazer é claramente alguma extensão do que o hardware faz.

2.10.1 Arranjos G - zero

Uma das primeiras abordagens para o tratamento do transbordamento de expoente foi chamada de arranjos G-zero. Este arranjo funciona da seguinte maneira:

a) Se houver underflow, o resultado se torna zero;
b) Se houver overflow, o resultado terá o sinal correto e o valor igual a G. G é um número em ponto flutuante diferente de infinito, este arranjo é conhecido como arranjo padrão.

2.10.2 Arranjos ∞ - zero

a) Se houver underflow, o resultado se torna zero;
b) Se houver overflow, o resultado será um genuíno infinito.

Neste caso precisa haver alguma forma de representar ∞ como no INTEL 80387.

Existem vantagens e desvantagens no uso de ambos os arranjos. Para a computação onde esses arranjos não dão bons resultados, existem outras formas de contornar o problema. Para mais detalhes ver [STER74].

2.11 Introdução à Múltipla Precisão

Vamos mostrar como a falta de precisão pode influenciar o resultado de uma computação. Seja o sistema linear

$$\begin{aligned} 37639840\,x - 46099201\,y &= 0, \\ 29180479\,x - 35738642\,y &= -1, \end{aligned} \qquad (2.5)$$

resolvido por um programa que reconhecidamente é de boa qualidade num equipamento com 16 dígitos decimais de precisão fornece a seguinte resposta:

$$x = -42587641.592475,$$
$$y = -34772663.750032,$$

embora a solução correta seja

$$x = 46099201,$$
$$y = 37639840.$$

O exemplo acima mostra bem a necessidade de uma precisão maior. O ideal seria se pudéssemos variar a precisão e ir substituindo o resultado no sistema linear (2.5) até conseguir a resposta exata ou aceitável.

Como foi visto na seção 2.3, podemos representar um número qualquer em ponto flutuante por

$$x = m \cdot b^e$$

onde

b - é a base do sistema de números, $b \geq 2$;

e - expoente, um inteiro qualquer;

m - fração ou mantissa, $\frac{1}{b} \leq |m| < 1$.

Quando fixamos o expoente e a quantidade de digitos da mantissa, estamos restringindo o conjunto de números reais que podemos representar. Ora, é bem sabido que o sistema de números reais é infinito e já que não podemos representar todos os seus elementos, é bom que possamos representar o maior número deles.

No exemplo anterior encontramos uma resposta absurda porque no conjunto de números que poderíamos representar, usando aquela precisão, perdíamos bastante digitos em virtude de arredondamento e/ou truncamento. Se aumentarmos a precisao para 25 digitos, por exemplo, já obtemos uma resposta bem satisfatória:

x = 46099201.0004,

y = 37639840.0003.

A idéia de múltipla precisão é esta, variar o expoente e a quantidade de dígitos da mantissa até que os números representados por essa combinação atendam às nossas necessidades.

2.12 Algoritmos Básicos em Aritmética de Múltipla Precisão

Por simplicidade, vamos assumir que estamos trabalhando com números inteiros e positivos ao invés de números da forma (2.1). Embora a mantissa deva satisfazer $1/b \leq m < 1$, as operações podem ser realizadas em inteiro com as mantissas, uma vez que o ponto decimal não é representado.

Vamos apresentar os algoritmos para realizar as quatro operações aritméticas. Vamos assumir que estamos trabalhando no sistema posicional de base b. Os algoritmos foram sugeridos por [KNUT69].

2.12.1 Adição de Números em Múltipla Precisão

Dados dois números inteiros com n dígitos cada um $u_1u_2...u_n$ e $v_1v_2...v_n$, este algoritmo obtém a soma desses números $w_0w_1w_2...w_n$ = $u_1u_2...u_n$ + $v_1v_2...v_n$. w_0 indica o carry, que sempre será 0 ou 1.

```
procedure A
  begin
    j := n  {indica os dígitos de u, v}
    k := 0  {carry}
    while (j ≥ 1) do
      w_j := (u_j + v_j + k) mod b
      k := ⌊(u_j + v_j + k) / b⌋  {k é igual a 0 ou 1,
                                   dependendo da ocorrência ou
                                   não de um carry}
      j := j - 1
    end_while
    w_o = k
  end.
```

2.12.2 Subtração de Números em Múltipla Precisão

Dados dois numeros inteiros positivos com n dígitos cada um $u_1u_2...u_n$ e $v_1v_2...v_n$, este algoritmo obtém a diferença desses números $w_1w_2...w_n = u_1u_2...u_n - v_1v_2...v_n$.

```
procedure S
  begin
    j := n  {indica os dígitos de u, v}
    k := 0  {carry}
    while (j ≥ 1) do
      w_j := (u_j - v_j + k) mod b
      k := ⌊(u_j - v_j + k) / b⌋  {k é igual a -1 ou 0,
                                   dependendo da ocorrência de um
```

```
                                    empréstimo ou não, isto é, se
                                    u_j - v_j + k < 0 ou não)
        j := j - 1
      end_while
    end.
```

2.12.3 Multiplicação de Números em Múltipla Precisão

Dados dois numeros inteiros positivos $u_1u_2...u_n$ e $v_1v_2...v_m$, este algoritmo obtém o produto desses números $w_1w_2...w_{m+n}$ = $u_1u_2...u_n * v_1v_2...v_m$.

```
  procedure M
    begin
      w_{m+1} := w_{m+2} := ... := w_{m+n} := 0
      j := m
      while (j ≥ 1) do
        if (v_j ≠ 0) then
          i := n
          k := 0
          while (i ≥ 1) do
            t := u_i * v_j + w_{i+j} + k
            w_{i+j} := t mod b
            k := ⌊ t / b ⌋ (o carry k sempre estará no intervalo
                                0 ≤ k < b)
            i := i - 1
          end_while
          w_j := k
```

```
        end_if
        j := j -1
      end_while
    end.
```

2.12.4 Divisão de Números em Múltipla Precisão

Dados dois numeros inteiros positivos $u_1u_2...u_{m+n}$ e $v_1v_2...v_n$ onde $n > 1$ e $v_1 \neq 0$, este algoritmo obtém o quociente $q_0q_1...q_m$ e o resto $r_1r_2...r_n$ da divisão de $u_1u_2...u_n$ por $v_1v_2...v_m$.

```
procedure D
  begin
    d := ⌊ b / (v1 + 1) ⌋ {d pode ser qualquer valor desde
                           que v1 ≥ ⌊ b / 2 ⌋}
    uo u1 u2...um+n := u1 u2...um+n * d
    v1v2...vn := v1v2...vn * d {introduzimos um novo dígito
                                uo à esquerda de u1. Se d = 1
                                tudo o que precisamos fazer é
                                uo = 0}
    j := 0
    while (j ≤ m) do {este ciclo indica a divisão de
                      uj uj+1 uj+2...uj+n por v1v2...vn}
      if (uj = v1) then
        q̂ := b - 1
      else
        q̂ := ⌊ (uj b + uj+1) / v1 ⌋
```

```
        end_if
        while (v_2 q̂ > (u_j b + u_{j+1} - q̂ v_1) b + u_{j+2}) do
          q̂ := q̂ - 1
        end_while
        u_j u_{j+1} u_{j+2} ... u_{j+n} = u_j u_{j+1} u_{j+2} ... u_{j+n} - q̂ * v_1 v_2 ... v_n
        q_j := q̂
        if (u_j u_{j+1} u_{j+2} ... u_{j+n} < 0) then
          u_j u_{j+1} u_{j+2} ... u_{j+n} := u_j u_{j+1} u_{j+2} ... u_{j+n} + 0v_1 v_2 ... v_n
        end_if
      end_while (q_0 q_1 ... q_m é o quociente, o resto é obtido de
                 u_{m+1} ... u_{m+n} / d)
    end.
```

2.13 Dígitos de Guarda

Como já foi visto, FP(b, p, a) representa o sistema de números em ponto flutuante onde p indica a precisão, ou seja, o número de dígitos na base b contidos na mantissa. Embora existam p dígitos, é importante usar dígitos extras, chamados de **dígitos de guarda**, durante as operações aritméticas com os números em ponto flutuante. Isso permite aumentar ainda mais a precisão dos resultados e é sobre esses dígitos de guarda que serão realizadas as operações de arredondamento ou de truncamento.

Não existe uma relação entre p e a quantidade de dígitos de guarda. A prática nos mostrou que dois dígitos de guarda são suficientes para se obter os resultados desejados. A justificativa para isso é que se temos p dígitos na mantissa mais

dois dígitos de guarda, o (p+2)-ésimo dígito, ou o segundo dígito de guarda, pode estar sujeito a erros devido aos métodos numéricos, mas o primeiro dígito de guarda estará correto. E é sobre este dígito que será feito o arredondamento ou truncamento.

Dessa forma, se temos $0.d_1d_2...d_p$, p dígitos na mantissa e $d_{p+1} \geq 5$, então $0.d_1d_2...d_p$ será aumentado de $1 \cdot b^{-p}$. E se $d_{p+1} < 5$ então $0.d_1d_2...d_p$ será o resultado.

Vamos ilustrar o que foi visto com um exemplo. Seja FP(10, 5, r) com dois dígitos de guarda e queremos calcular $y = \frac{1}{x}$ com $x > 0$. Utilizaremos o processo iterativo de Newton sugerido po [MARO87] para aproximar y:

$$y_{n+1} = y_n (2 - x.y_n) \qquad ; \text{para } n = 0, 1, 2,...$$

O valor de y_o sugerido por [MARO87] é o seguinte. Suponha que x seja escrito na forma

$$x = x_1 \cdot 2^m$$

onde

$$\frac{1}{2} \leq x_1 < 1,$$

m é um inteiro.

$y_o = 2^{-m}$ será então a primeira aproximação para y. Seja $x = 2$, então $y_o = \frac{1}{4}$

iteração	valor de y	dígito de guarda
0	0.25	00
1	0.375	00
2	0.46875	00
3	0.49804	68
4	0.49999	23
5	0.49999	99

Sem usarmos os dígitos de guarda os resultados da quarta e quinta iteração seriam iguais, dessa forma concluiríamos que 1 / 2 seria 0.49999. Ao passo que, usando os dígitos de guarda o resultado seria arredondado para 0.5, pois o primeiro dígito de guarda é 9 e é maior que 5. Isso mostra que podemos aumentar a precisão usando tais dígitos.

CAPÍTULO III

COMPUTAÇÃO DE FUNÇÕES ELEMENTARES

3.1 Introdução

Neste capítulo vamos mostrar como podemos aproximar valores de funções através de processos numéricos. Trataremos apenas da aproximação polinomial, já que esta é de longe a mais usada, porque envolve apenas as quatro operações aritméticas se considerarmos a potenciação (inteira) como sequência de multiplicações. Utilizaremos três métodos: interpolação, quadrados mínimos e aproximação racional, que é o tipo de função mais geral que pode ser avaliada diretamente em um computador. Finalmente, vamos mostrar como ficam esses métodos de aproximação de funções quando se utiliza aritmética de múltipla precisão.

3.2 Aproximação

Em análise numérica, procura-se resolver problemas matemáticos por processos numéricos. Claramente há necessidade de aproximar quantidades não aritméticas por quantidades aritméticas e descobrir os erros associados a tais aproximações é o tema central dessa análise. Há situações em que existirão vários métodos possíveis para se obter uma aproximação desejada. A escolha de um método depende do critério a ser usado para julgar tal aproximação.

Suponha que é dado um valor $x > 0$ e desejamos calcular $\sqrt{x}$. Algumas possibilidades são as seguintes:

1. Usar o método clássico aprendido pela maioria das pessoas na escola de segundo grau;
2. Procurar x na tabela de raiz quadrada, e se x estiver entre dois argumentos nesta tabela, fazer interpolação para encontrar $\sqrt{x}$;
3. Usar qualquer um dos processos iterativos para computar $\sqrt{x}$.

Naturalmente assumiremos que todos esses métodos conduzem a um resultado razoável de $\sqrt{x}$. A questão básica é: qual o erro que podemos tolerar no resultado? Esta questão não somente reconhece a ocorrência de um erro, mas a importância de se estar apto a estimá-lo. Somente quando se tem métodos onde é possivel estimar ou limitar um erro de aproximação podemos comparar esses métodos baseados na magnitude desses erros.

A questão que surge é de qual erro estimar, se o absoluto ou o relativo? Outra questão é de como limitar esse erro, se para todo x em algum intervalo ou algum erro médio. Essas questões necessitam ser respondidas na prática, mas nosso objetivo é mostrar que em uma aproximação, primeiramente devemos alcançar algum grau desejado de precisão e implicitamente assumir que essa precisão pode ser estimada. Além desses objetivos, devemos levar em conta a eficiência de um método para obter uma aproximação.

3.3 Classes de Funções de Aproximação

Muitas das aproximações feitas em análise numérica consistem em aproximar uma função f(x) por alguma combinação, a maioria das vezes linear, de funções pertencentes a alguma classe particular. O exemplo mais familiar dessa aproximação é usar os n primeiros termos da série de Taylor. Outro exemplo familiar é a regra do trapézio em que f(x) é aproximada por uma sequência de segmentos de reta. Para cada segmento, a aproximação é um polinômio de grau 1. Generalizando, podemos considerar um polinômio de grau n em vez de 1. Uma outra classe sugerida pela importância das funções é a classe de funções trigonométricas sin nx + cos nx, n = 0, 1, ... Existem outras classes de funções, sendo a exponencial um exemplo útil em alguns casos particulares. Para aplicações comuns em geral, aproximações polinomiais são as mais importantes segundo [RALS65].

3.4 Tipos de Aproximações

Seja f(x) uma função que desejamos aproximar usando a classe de funções $\{ g_n(x) \}$, n = 0, 1, ... Suponha que aproximamos f(x) por uma combinação linear

$$f(x) = a_0 g_0(x) + a_1 g_1(x) + \ldots + a_m g_m(x) \qquad (3.1)$$

onde a_i, i = 0, 1, ..., m são constantes. Chamamos (3.1) uma aproximação do tipo linear de f(x). Uma aproximação do tipo racional de f(x) é da forma

$$f(x) = \frac{a_0 g_0(x) + a_1 g_1(x) + \ldots + a_m g_m(x)}{b_0 g_0(x) + b_1 g_1(x) + \ldots + b_k g_k(x)} \qquad (3.2)$$

onde a_i, b_j, $i = 0, 1, \ldots m$ e $j = 0, 1, \ldots, k$ são constantes. A questão crucial do problema de aproximação é o critério usado na escolha das constantes em (3.1) e (3.2). Três métodos baseados em diferentes critérios são importantes:

1. **Interpolação** : As constantes são escolhidas de modo que nos pontos x_i, $i = 1, \ldots, p$ a aproximação e suas primeiras derivadas r_i (onde r_i é um inteiro maior do que zero) coincidem com f(x), exceto pelo erro de arredondamento inevitável.

2. **Quadrados Mínimos**: O objetivo é minimizar a integral do quadrado da diferença de f(x) e sua aproximação no intervalo [a, b], ou mais comumente, minimizar a soma dos quadrados dos erros nesse mesmo intervalo.

3. **Aproximação Racional**: O objetivo nessa aproximação é minimizar a máxima magnitude da diferença entre f(x) e sua aproximação no intervalo [a, b].

3.5 Aproximação Polinomial

Uma aproximação polinomial é do tipo (3.1), onde cada $g_i(x)$ é um polinômio. A aproximação polinomial se justifica pelo fato de que um computador pode executar diretamente as operações aritméticas envolvidas na avaliação de um polinômio, desde que considere a potenciação inteira como multiplicações de fatores iguais. Dessa forma, se usarmos funções que não sejam polinômios essas funções devem ser avaliadas usando aproximação polinomial ou racional, que é o tipo de função mais geral que pode ser avaliada diretamente em um computador.

Uma propriedade que as potências de x e as funções trigonométricas (bem como as funções exponenciais) têm em comum é que uma aproximação usando qualquer uma dessas classes, mudam seus coeficientes, mas não suas formas se a origem do sistema de coordenadas for mudado. Dessa forma, se $p(x)$ é um polinômio ou função racional, $p(x + \alpha)$ também o é, e se $t(x)$ é uma aproximação linear ou racional usando seno e cosseno, $t(x + \alpha)$ também o é. Uma aproximação usando potência de x tem uma grande vantagem, pois se a escala da variável x for mudada em relação aos coeficientes, sua forma de aproximação continua. Dessa forma, $p(kx)$ é ainda um polinômio em x e do mesmo grau de $p(x)$. Isto não é verdade para aproximações usando seno e cosseno. Em geral, para k não inteiro $\sin(nkx)$ não é membro da classe $\sin(nx)$.

Outra vantagem dos polinômios é que em geral são fáceis de serem manipulados, em particular diferenciados e integrados.

O teorema da aproximação de Weierstrass nos garante que se f(x) for uma função continua em um intervalo finito [a, b], então, dado qualquer $\epsilon > 0$, exixte um n e um polinômio $p_n(x)$ tal que

$$|f(x) - p_n(x)| < \epsilon$$

para todo x em [a, b]. Prova em [RALS65].

Esse teorema nos garante que podemos encontrar um polinômio que aproxime f(x) e que tenha um erro arbitrariamente pequeno. As vantagens computacionais e analíticas citadas anteriormente, fortemente recomendam o uso da **aproximação polinomial.**

3.6 Interpolação

Suponha que se tenha uma função f(x) que é conhecida em um conjunto de pontos. Esses pontos são chamados de pontos tabulares. O objetivo da interpolação é estimar valores dessa função em pontos não tabulares e limitar o erro entre a estimativa e o valor verdadeiro.

A necessidade de se interpolar surge em várias situações, como por exemplo:

a) Quando são conhecidos apenas os valores numéricos da função para um conjunto de pontos e há necessidade de calcular o valor da função em um ponto não tabelado ou obter o valor de sua integral;

b) Quando a função em estudo tem expressão analítica complicada que torna as operações como a diferenciação e a integração difíceis ou mesmo impossíveis de serem realizadas.

Nossa abordagem será aproximar **f(x)** por uma função **g(x)** que, nos pontos tabulares, tem um mesmo valor de **f(x)** e pertence à classe de funções polinomiais.

3.6.1 Interpolação Polinomial

Dados os pontos $(x_0, f(x_0))$, $(x_1, f(x_1))$, ..., $(x_n, f(x_n))$, portanto n + 1 pontos, queremos aproximar **f(x)** por um polinômio de grau $\leq n$, p_n tal que

$$f(x_k) = p_n(x_k),\ k = 0, 1, \ldots, n.$$

A existência e unicidade de $p_n(x)$ é estabelecida como se segue.

Representemos $p_n(x)$ por $a_o + a_1(x) + a_2x^2 + \ldots + a_nx^n$. Para obter $p_n(x)$, basta obter os coeficientes $a_0, a_1, \ldots, a_n$.

Da condição $f(x_k) = p_n(x_k)$, $k = 0, 1, \ldots, n$ resulta o seguinte sistema linear

$$
\begin{aligned}
a_0 + a_1 x_0 + a_2 x_0^2 + \ldots + a_n x_0^n &= f(x_0) \\
a_0 + a_1 x_1 + a_2 x_1^2 + \ldots + a_n x_1^n &= f(x_1) \\
\vdots \qquad\quad \vdots \qquad\qquad\qquad\qquad &\quad \vdots \\
a_0 + a_1 x_n + a_2 x_n^2 + \ldots + a_n x_n^n &= f(x_n)
\end{aligned}
$$

com n + 1 equações e n + 1 incógnitas a_0, a_1, ..., a_n . A matriz **A** dos coeficientes é

$$
A = \begin{bmatrix}
1 & x_0 & x_0^2 & \ldots & x_0^n \\
1 & x_1 & x_1^2 & \ldots & x_1^n \\
 & & \vdots & & \\
1 & x_n & x_n^2 & \ldots & x_n^n
\end{bmatrix}
$$

que é uma matriz de **Vandermonde** e, portanto, desde que x_0, x_1, ..., x_n sejam pontos distintos, temos que **det A ≠ 0**, o que implica que o sistema linear admite solução única. Portanto, existe um único polinômio $p_n(x)$ tal que $f(x_k) = p_n(x_k)$, k = 0, 1, ..., n desde que $x_k \neq x_j$ para k ≠ j.

3.6.2 Formas de se obter $p_n(x)$ - Forma de Newton

Existem várias formas do polinômio interpolador. Uma delas é a forma de Newton,

$$
p_n(x) = d_0 + d_1(x - x_0) + \ldots + d_n(x - x_0)(x - x_1)\ldots(x - x_{n-1})
$$

em que precisamos obter os valores de d_0, d_1, ..., d_n.

I. Operador de diferenças divididas.

Seja f(x) uma função tabelada em n + 1 pontos distintos x_0, x_1, ..., x_n . O operador de diferenças divididas é definido por:

$$f[x_0] = f(x_0)$$

$$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0} = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

$$\vdots \qquad\qquad \vdots$$

$$f[x_0, x_1, \ldots, x_n] = \frac{f[x_1, x_2, \ldots, x_n] - f[x_0, x_1, \ldots, x_{n-1}]}{x_n - x_0}$$

Dizemos que $f[x_0, x_1, \ldots, x_k]$ é a diferença dividida de ordem k da função f(x) sobre os k + 1 pontos $x_0, x_1, \ldots, x_k$.

TABELA DAS DIFERENÇAS DIVIDIDAS

Dada uma função f(x) e conhecidos os valores que f(x) assume nos pontos distintos $x_0, x_1, \ldots, x_n$, podemos construir a tabela:

x	ordem 0	ordem 1	ordem 2	...	ordem n
x_0	$f[x_0]$				
		$f[x_0, x_1]$			
x_1	$f[x_1]$		$f[x_0, x_1, x_2]$		
		$f[x_1, x_2]$			$f[x_0, x_1, \ldots, x_n]$
x_2	$f[x_2]$		$f[x_1, x_2, x_3]$		
		$f[x_2, x_3]$			
x_3	$f[x_3]$				
			$f[x_{n-2}, x_{n-1}, x_n]$		
		$f[x_{n-1}, x_n]$			
x_n	$f[x_n]$				

II. Forma de Newton para o Polinômio Interpolador

Seja $f(x)$ contínua e com tantas derivadas quantas forem necessárias num intervalo $[a, b]$. E sejam $a = x_0 < x_1 < \ldots < x_n = b$, $n + 1$ pontos.

Para construirmos o polinômio $p_n(x)$ que interpola $f(x)$ em x_0, x_1, ..., x_n iniciamos obtendo $p_0(x)$ que interpola $f(x)$ em $x = x_0$. E assim sucessivamente, construiremos $p_k(x)$ que interpola $f(x)$ em x_0, x_1, ..., x_k, $k = 0, 1, \ldots, n$ sendo que $p_{k+1}(x) = p_k(x) + p_{k+1}(x)$, $k = 0, 1, \ldots, n$.

Seja $p_0(x)$ o polinômio de grau zero que interpola $f(x)$ em $x = x_0$. Então, $p_0(x) = f(x_0) = f[x_0]$.

Temos, para todo $x \in [a, b]$, $x \neq x_0$

$$f[x_0,x] = \frac{f[x] - f[x_0]}{x - x_0} = \frac{f(x) - f(x_0)}{x - x_0}$$

$$(x - x_0) \cdot f[x_0, x] = f(x) - f(x_0)$$

$$f(x) = \underbrace{f(x_0)}_{p_0(x)} + \underbrace{(x - x_0) \cdot f[x_0, x]}_{E_0(x)}$$

$$E_0(x) = f(x) - p_0(x) = (x - x_0) \cdot f[x_0, x].$$

Note que $E_0(x)$ é o erro ao se aproximar $f(x)$ por $p_0(x)$. Seja agora construir $p_1(x)$, o polinômio de grau um que interpola $f(x)$ em x_0 e x_1.

Temos

$$f[x_0, x_1, x] = f[x_1, x_0, x] = \frac{f[x_0, x] - f[x_1, x_0]}{x - x_1} =$$

$$\frac{\dfrac{f(x) - f(x_0)}{x - x_0} - f[x_1, x_0]}{(x - x_1)} = \frac{f(x) - f(x_0) - (x - x_0) \cdot f[x_1, x_0]}{(x - x_1) \cdot (x - x_0)}$$

$$f[x_0, x_1, x] = \frac{f(x) - f(x_0) - (x - x_0) \cdot f[x_1, x_0]}{(x - x_0) \cdot (x - x_1)}$$

$$f(x) = \underbrace{f(x_0) + (x - x_0).f[x_1, x_0]}_{p_1(x)} + \underbrace{(x - x_0).(x - x_1).f[x_0, x_1, x]}_{E_1(x)}.$$

Assim

$$p_1(x) = f(x_0) + (x - x_0) \cdot f[x_0, x_1]$$

$$E_1(x) = (x - x_0) \cdot (x - x_1) \cdot f[x_0, x_1, x].$$

Aplicando sucessivamente o mesmo raciocínio, teremos a forma de Newton para o polinômio de grau menor ou igual a n que interpola f(x) em x_0, x_1, ..., x_n:

$$p_n(x)=f(x_0)+(x-x_0).f[x_0,x_1]+...+(x-x_0)...(x-x_{n-1}).f[x_0,...,x_n].$$

Exemplo: Obter o polinômio $p_2(x)$ que interpola f(x) nos pontos dados abaixo:

x	-1	0	2
f(x)	4	1	1

$$p_2(x)=f(x_0)+(x - x_0).f[x_0, x_1]+(x - x_0).(x - x_1).f[x_0, x_1, x_2]$$

ordem

x	0	1	2
-1	4		
		-3	
0	1		2/3
		-1	
2	-1		

$$p_2(x) = 4 + (x + 1) . (-3) + (x + 1) . (x - 0) . 2/3 \qquad (1)$$

$$p_2(x) = \frac{2}{3} x^2 - \frac{7}{3} x + 1 \qquad (2)$$

na prática a forma (1) é mantida.

3.6.3 Estudo do Erro na Interpolação

Ao se aproximar uma função f(x) por um polinômio interpolador de grau $\leq$ n, comete-se um erro, ou seja, $E_n(x) = f(x) - p_n(x)$ para

todo $x \in [x_0, x_n]$. O estudo do erro é importante para sabermos quão próximo $p_n(x)$ está de $f(x)$.

Teorema 3.1: Sejam $x_0 < x_1 < \ldots < x_n$ n+1 pontos. Seja $f(x)$ com derivadas até a ordem n+1 para todo $x \in [x_0, x_n]$. Seja $p_n(x)$ o polinômio interpolador de $f(x)$ nos pontos $x_0, x_1, \ldots, x_n$. Então, em qualquer ponto $x \in [x_0, x_n]$, o erro é dado por $E_n(x) = f(x) - p_n(x) = (x - x_0) \cdot (x - x_1) \ldots (x - x_n) \cdot \frac{f^{n+1}(\xi_x)}{(n+1)!}$ onde $\xi_x \in (x_0, x_n)$. Prova matemática em [RUGG88].

Exemplo: Seja o problema de se obter ln(3.7) por interpolação, onde ln(x) está tabelado abaixo:

x	1	2	3	4
ln(x)	0	0.6931	1.0986	1.3863

com $x = 3.7 \in (3,4)$ escolhemos $x_0 = 3$ e $x_1 = 4$. Pela fórmula de Newton, $p_1(x) = f(x_0) + (x - x_0) \cdot f[x_0, x_1]$. Substituindo os valores tabelados temos

$$p_1(x) = 1.0986 + (x - 3) \cdot (0.2877)$$

$$p_1(3.7) = 1.3000.$$

Dado que com quatro casas decimais ln(3.7) = 1.3083, o erro cometido é $E_1(3.7) = 8.3 \times 10^{-3}$. Queremos mostrar com esse exemplo que ξ_x que aparece na expressão do erro do teorema 3.1 é realmente uma função de x.

$$E_1(x) = (x - x_0).(x - x_1).\frac{f''(\xi_x)}{2} \quad , \xi_x \in [x_0, x_1]$$

$$\text{Para } x = 3.7,\ E_1(3.7) = (3.7 - 3)\ .\ (3.7 - 4).\frac{f''(\xi_x)}{2} = 8.3\times10^{-3}$$

$$\text{Agora } f''(x) = -\frac{1}{x^2}\ , \text{ então, } (0.7)\ .\ (-0.3)\ .\ \left(-\frac{1}{2\xi_x^2}\right) = 8.3\times10^{-3}$$

$$\Rightarrow\ \xi_x' = -3.5578 \quad e \quad \xi_x'' = 3.5578\ .\ \text{Mas } \xi_x \in (3, 4), \text{ então}$$

$$\xi_x = 3.5578.$$

3.6.4 Limitante para o Erro

A fórmula para o erro

$$E_n(x) = f(x) - p_n(x) = (x - x_0)\ .\ (x - x_1)\ \ldots\ (x - x_n)\ .\ \frac{f^{n+1}(\xi_x)}{(n+1)!}$$

onde $\xi_x \in (x_0, x_n)$ tem uso limitado na prática, uma vez que serão raras as situações em que conheceremos $f^{n+1}(x)$, e o ponto ξ_x nunca é conhecido.

A importância da fórmula exata para $E_n(x)$ é teórica e pode ser usada na obtenção das estimativas de erro para as fórmulas de interpolação, diferenciação e integração numérica.

Veremos a seguir dois corolários do teorema anterior, que relacionam o erro com um limitante de $f^{n+1}(x)$.

Corolário 1: Com as hipóteses do teorema 3.1 e se $f^{n+1}(x)$ for contínua em $I = [x_0, x_n]$ podemos escrever a seguinte relação:

$$|E_n(x)| = |f(x) - p_n(x)| \leq |(x - x_0).(x - x_1)...(x - x_n)| \cdot \frac{M_{n+1}}{(n+1)!}$$

onde $M_{n+1} = \max_{x \in I} |f^{n+1}(x)|$.

Corolário 2: Se além das hipóteses anteriores os pontos forem igualmente espacejados, ou seja

$$x - x_0 = x_2 - x_1 = ... = x_n - x_{n-1} = h$$

então

$$|f(x) - p_n(x)| < \frac{h^{n+1} \cdot M_{n+1}}{4.(n+1)} .$$

Exemplo: Seja $f(x) = e^x + x - 1$ tabelada abaixo. Obter $f(0.7)$ por interpolação e fazer uma análise do erro cometido:

x	0	0.5	1	1.5	2.0
f(x)	0	1.1487	2.7183	4.9811	8.3890

$$p_1(x) = f(x_0) + (x - x_0).f[x_0, x_1]$$

$x = 0.7 \in (0.5, 1)$, então $x_0 = 0.5$ e $x_1 = 1$

e então

$$p_1(x) = 1.1487 + (x - 0.5) \cdot (3.1392)$$

$$p_1(0.7) = 1.7765.$$

Nesse caso, temos condições de calcular o valor do erro dado por $|E_1(0.7)| = |f(0.7) - p_1(0.7)| = 0.0628$.

Os corolários 1 e 2 fornecem os seguintes limites para o erro:

a) Corolário 1 em x = 0.7

$$|E_1(0.7)| \leq |(0.7 - 0.5) \cdot (0.7 - 1)| \cdot \frac{M_2}{2},$$

onde

$$M_2 = \max_{x \in [0.5,1]} |f''(x)| = e^1 = 2.7183$$

então

$$|E_1(0.7)| \leq 0.0815.$$

b) Corolário 2 em x = 0.7

$$|E_1(x)| \leq \frac{h^2}{8} \cdot M_2 = \frac{0.5^2}{8} \cdot 2.7183 = 0.0850.$$

3.6.5 Estimativa para o Erro

Se a função f(x) for dada na forma tabular, o valor absoluto do erro $|E_n(x)|$ só pode ser estimado, porque nesse caso, não é possível calcular M_{n+1}; mas se construirmos a tabela de diferenças divididas até a ordem n+1, podemos usar o maior valor em módulo

das diferenças de ordem n+1 como uma aproximação para $\frac{M_{n+1}}{(n+1)!}$ no intervalo $[x_0, x_n]$. Nesse caso, dizemos que

$$|E_n(x)| \cong |(x - x_0) \cdot (x - x_1) \ldots (x - x_n)| \cdot \max|\text{dif. divididas}|$$

3.6.6 Sobre o Grau do Polinômio Interpolador

A tabela de diferenças divididas junto com a relação entre a diferença dividida de ordem k e a k-ésima derivada,podem nos auxiliar na escolha do grau do polinômio que usaremos para interpolar uma função f(x).

Deve-se em primeiro lugar, construir a tabela de diferenças divididas. Em seguida, examinar as diferenças divididas da função na vizinhança do ponto de interesse. Se nesta vizinhança as diferenças divididas de ordem n são praticamente constantes ou se as diferenças de ordem n + 1 variam em torno de zero, podemos concluir que um polinômio de grau n será o que melhor aproximará a função na região considerada da tabela.

Por exemplo, consideremos $f(x) = \sqrt{x}$ tabelada abaixo com quatro casas decimais :

x	0	1.01	1.02	1.03	1.04	1.05
f(x)	0	1.005	1.01	1.0149	1.0198	1.0247

	ordem		
x	0	1	2
1.0	1		
		0.5	
1.01	1.005		0
		0.5	
1.02	1.01		-0.5
		0.49	
1.03	1.0149		0
		0.49	
1.04	1.0198		0
		0.49	
1.05	1.0247		

Assim, no intervalo [1, 1.05] dizemos que um polinômio de grau 1 é uma boa aproximação para $\sqrt{x}$.

3.7 Quadrados Mínimos

Vimos na seção anterior como escolher as constantes de (3.1) usando a interpolação. A interpolação não é aconselhável quando:

a) É preciso obter um valor aproximado da função em algum ponto fora do intervalo tabelado, ou seja, quando se quer extrapolar;

b) Os valores são resultados de algum experimento, porque nesse caso, esses valores poderão conter erros inerentes que, em geral, não são previsíveis.

Surge então a necessidade de se ajustar a essas funções tabeladas uma função que seja uma boa aproximação para os valores tabelados e que nos permita extrapolar com certa margem de segurança. Trataremos apenas do caso discreto, uma vez que o

contínuo é análogo.

3.7.1 Caso Discreto

O problema do ajuste de curvas no caso em que temos uma tabela de pontos $(x_1, f(x_1))$, $(x_2, f(x_2))$, ..., $(x_m, f(x_m))$ com x_1, x_2, ..., x_n pertencentes a um intervalo [a, b], consiste em **escolhermos** n funções $g_1(x)$, $g_2(x)$, ..., $g_n(x)$, contínuas em [a, b], e obter n constantes α_1, α_2, ..., α_n tais que a função $\delta(x) = \alpha_1 g_1(x) + \alpha_2 g_2(x) + \ldots + \alpha_n g_n(x)$, se aproxime ao máximo de f(x).

Dizemos que este é um modelo matemático linear porque os coeficientes a determinar α_1, α_2, ..., α_n aparecem linearmente, embora as funções $g_1(x)$, $g_2(x)$, ..., $g_n(x)$ possam ser funções não lineares em x.

A escolha das funções $g_1(x)$, $g_2(x)$, ..., $g_n(x)$ pode ser feita observando o gráfico dos pontos tabelados ou baseando-se em fundamentos teóricos do experimento.

3.7.2 O Método dos Quadrados Mínimos

Sejam dados os pontos $(x_1, f(x_1))$, $(x_2, f(x_2))$, ..., $(x_m, f(x_m))$ e as n funções $g_1(x)$, $g_2(x)$, ..., $g_n(x)$ **escolhidas** de alguma forma.

Supomos que o número de pontos m, tabelados, é sempre maior ou igual a n, o número de funções escolhidas ou o número de

coeficientes α_i's a se determinar.

Seja $d_k = f(x_k) - \delta(x_k)$ o desvio em x_k. O método dos quadrados mínimos consiste em escolher os α_i's de tal forma que a soma dos quadrados dos desvios seja mínima. É claro, para que a soma $\sum_{k=1}^{m} d_k^2 = \sum_{k=1}^{m} [f(x_k) - \delta(x_k)]^2$ seja mínima, cada parcela $[f(x_k)-\delta(x_k)]^2$ deve ser pequena, ou seja, cada desvio $f(x_k) - \delta(x_k)$ é pequeno.

Portanto, pelo critério dos quadrados mínimos, os coeficientes α_i's, que fazem com que $\delta(x)$ se aproxime ao máximo de $f(x)$, minimizam a função

$$F(\alpha_1, \alpha_2, \ldots, \alpha_n) = \sum_{k=1}^{m} [f(x_k) - \delta(x_k)]^2 =$$

$$\sum_{k=1}^{m} [f(x_k) - \alpha_1 g_1(x_k) + \alpha_2 g_2(x_k) - \ldots - \alpha_n g_n(x_k)]^2 \qquad (3.3)$$

Observamos que se o modelo ajustar exatamente aos dados, o mínimo da função acima será zero e, portanto, a interpolação é um caso especial dos quadrados mínimos.

Diferenciando (3.3) em relação a α_1, α_2, ..., α_n e igualando as derivadas a zero, obtemos o sistema

$$\left[\sum_{k=1}^{m} g_1(x_k).g_1(x_k)\right]\alpha_1 + \ldots + \left[\sum_{k=1}^{m} g_n(x_k).g_1(x_k)\right]\alpha_n = \sum_{k=1}^{m} f(x_k).g_1(x_k)$$

$$\left[\sum_{k=1}^{m} g_1(x_k).g_2(x_k)\right]\alpha_1 + \ldots + \left[\sum_{k=1}^{m} g_n(x_k).g_2(x_k)\right]\alpha_n = \sum_{k=1}^{m} f(x_k).g_2(x_k)$$

$$\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots$$

$$\left[\sum_{k=1}^{m} g_1(x_k).g_n(x_k)\right]\alpha_1 + \ldots + \left[\sum_{k=1}^{m} g_n(x_k).g_n(x_k)\right]\alpha_n = \sum_{k=1}^{m} f(x_k).g_n(x_k)$$

que é um sistema linear com n equações e n incógnitas α_i's. Essas equações são chamadas **equações normais** [RUGG88].

Exemplo: Para função tabelada abaixo

x	-1.0	-1.75	-0.6	-0.5	-0.3	0.0	0.2	0.4	0.5	0.7	1.0
f(x)	2.05	-1.153	0.45	0.4	0.5	0.0	0.2	0.6	0.512	1.2	2.05

deve ser ajustada por uma parábola passando pela origem, ou seja, $f(x) \cong \delta(x) = \alpha.x^2$ (neste caso temos apenas a função $g(x) = x^2$).

Temos pois que resolver apenas a equação

$$\left[\sum_{k=1}^{11} g(x_k).g(x_k)\right].\alpha = \sum_{k=1}^{11} f(x_k).g(x_k)$$

$$\left[\sum_{k=1}^{11} g(x_k)^2\right].\alpha = \sum_{k=1}^{11} f(x_k).g(x_k)$$

$$\left[\sum_{k=1}^{11} (x_k^2)^2\right].\alpha = \sum_{k=1}^{11} x_k^2 .f(x_k)$$

$$2.8464\ \alpha = 5.8756 \Rightarrow \alpha \cong 2.0642.$$

Então $\delta(x) = 2.0642\ x^2$ é a parábola que melhor se aproxima, no sentido dos quadrados mínimos, da função tabelada.

3.8 Aproximação Racional

Uma outra maneira de se avaliar uma função matemática - trigonométrica, exponencial, logarítmica, etc - em um computador seria armazenar uma tabela da função na memória e usar interpolação para avaliar a função nos pontos não tabulados. Mas essa técnica seria extremamente consumidora de memória e também não teria nenhuma vantagem em velocidade e em precisão sobre a aproximação racional segundo [CODY80].

Como vimos na equação (3.2), uma função racional é a função mais geral que pode ser avaliada diretamente em um computador. Mas por que usar função racional ao invés de polinomial (que na verdade é apenas um caso particular de aproximação racional)?

A situação geral é a seguinte: uma computação deve ser executada muitas, talvez milhões, de vezes para avaliar uma função matemática . Geralmente, em toda a computação é desejável limitar o erro no resultado. Para limitar esse erro sem um conhecimento a priori dos números envolvidos, será com certeza o pior caso. Dessa forma a propriedade do erro dessa aproximação de maior importância será o maior erro em magnitude no intervalo. Portanto, o maior objetivo de uma computação para aproximação de uma função é minimizar o máximo erro.

Nossa meta nessa seção será desenvolver uma técnica para gerar uma aproximação racional que, entre todas as funções racionais com o mesmo grau do polinômio no numerador e denominador, cometa o mínimo erro máximo.

Se uma aproximação for avaliada milhões de vezes, uma outra meta da aproximação é alcançar a máxima velocidade. A primeira meta acima, minimizar o erro máximo, é a maior razão para a escolha da aproximação racional em vez da aproximação polinomial. Para uma computação qualquer, a aproximação racional obtém menor erro máximo que a aproximação polinomial na maioria das funções aproximadas em um computador. Esse resultado é empírico, e é mostrado com detalhes em [RALS65].

3.8.1 A Aproximação de Padé

Nossa tentativa de obter uma aproximação da forma

$$R_{mk}(x) = \frac{P_m(x)}{Q_k(x)} \qquad (3.4)$$

onde $P_m(x)$ e $Q_k(x)$ são polinômios de grau no máximo m e k respectivamente, e N = m + k o índice de $R_{mk}(x)$, será dados m e k, escolher $P_m(x)$ e $Q_k(x)$ de tal forma que $P_m(x) / Q_k(x)$ seja igual a f(x) em x = 0 e tenham as mesmas derivadas nesse ponto. No caso de k = 0, a aproximação é apenas a série de Maclaurin para f(x). Será assumido que a série de Maclaurin para f(x) existe para x próximo de zero. Existem duas razões para a escolha de x = 0: (1) a manipulação se torna bem mais simples do que para outro x

qualquer; (2) o intervalo sobre o qual desejamos aproximar a maioria das funções conterá o zero, e quando não, uma simples mudança de variável pode ser usada para fazer o intervalo conter o zero. Assumiremos também que $P_m(x)$ e $Q_k(x)$ não têm fatores comuns. Sejam

$$P_m(x) = \sum_{j=0}^{m} a_j.x^j \quad e$$

$$Q_k(x) = \sum_{j=0}^{k} b_j.x^j$$

onde $b_o = 1$. É permitido tornar o termo constante em $Q_k(x)$ igual a 1 porque: (1) o termo constante não pode ser zero se a aproximação deve existir em $x = 0$; (2) o valor de $R_{mk}(x)$ não muda se o numerador e denominador forem divididos pela mesma constante.

Seja $\sum_{j=0}^{\infty} c_j.x^j$ a série de Maclaurin para f(x). Então consideramos a diferença

$$f(x) - \frac{P_m(x)}{Q_k(x)} = \frac{\left[\sum_{j=0}^{\infty} c_j.x^j\right].\left[\sum_{j=0}^{k} b_j.x^j\right] - \sum_{j=0}^{m} a_j.x^j}{\sum_{j=0}^{k} b_j.x^j} \qquad (3.5)$$

Uma vez que temos N + 1 constantes - m + 1 a_j's e k b_j's - a nossa disposição, faremos $f(x) - R_{mk}(x)$ e suas primeiras N derivadas iguais a zero em $x = 0$. Alcançaremos isso se o numerador

do lado direito de (3.5) é tal que suas primeiras potências são de grau N+1. Dessa forma escreveremos

$$\left[\sum_{j=0}^{\infty} c_j . x^j\right] . \left[\sum_{j=0}^{k} b_j . x^j\right] - \sum_{j=0}^{m} a_j . x^j = \sum_{j=0}^{\infty} d_{N+1} . x^j \qquad (3.6)$$

O desaparecimento dos coeficientes nas primeiras N+1 potências de x no lado direito de (3.6) é equivalente às equações

$$\sum_{j=0}^{k} c_{N-s-j} . b_j = 0 \ , \begin{cases} s = 0, 1, \ldots, N - m - 1 \\ c_j = 0 \ , \ \text{se } j < 0 \ , \end{cases} \qquad (3.7)$$

$$a_r = \sum_{j=0}^{r} c_{r-j} . b_j \ , \begin{cases} r = 0, 1, \ldots \\ b_j = 0 \ , \text{se } j > k \end{cases} \qquad (3.8)$$

Quando esse conjunto de N + 1 equações lineares tiver uma solução, tem-se a aproximação da forma de (3.4) [RALS65]. Uma boa estimativa do erro em (3.4) pode ser dada pelo primeiro termo do lado direito de (3.6), que é $d_{N+1}.x^{n+1}$ com d_{N+1} dado por

$$\sum_{j=0}^{k} c_{N-1-j} . b_j .$$

Essa aproximação do erro ilustra o fato que, em comum com a aproximação por série de Maclaurin, a aproximação racional do tipo (3.4) tem erros pequenos próximos do zero e maiores afastando-se.

Isso é o que se poderia esperar devido a exigência feita, pela qual a aproximação coincidiria com f(x) e suas primeiras N derivadas em x = 0. Isso significa na prática que se x = 0 não for o ponto médio do intervalo pelo qual estamos aproximando, então faremos a mudança de variável de modo que o zero venha a ser o ponto médio desse intervalo. Na prática, para m = k ou m = k + 1, $R_{mk}(x)$ tem o mínimo erro máximo para um dado N. Detalhes em [RALS65].

3.8.2 Exemplo

Vamos considerar o problema de aproximar e^x no intervalo $(-\infty,\infty)$. Primeiro devemos considerar que o intervalo é infinito. Não podemos esperar aproximar uma função num intervalo infinito com bom comportamento do erro nesse intervalo. Nosso primeiro passo será converter o problema para aproximar e^x em algum intervalo finito.

Para todo $x \in (-\infty,\infty)$, escrevemos

$$x.\log_{10} e = X + F \qquad (3.9)$$

onde X é um inteiro e F uma fração tal que $0 \leq F < 1$. De (3.9) obtemos

$$10^{x.\log_{10} e} = e^x = 10^X.10^F = 10^X.e^{F.\ln 10}.$$

Como X é um inteiro, a influência de 10^x pode ser considerada apenas adicionando x ao expoente do número em ponto flutuante que usa base 10. Dessa forma, transpomos o problema para aproximar e^x no intervalo [0, ln 10] ≅ [0, 2.3026]. Como o zero não é o ponto médio do intervalo, primeiro faremos uma mudança de variável para conseguir o novo intervalo [-1.1513, 1.1513]. E então reduzimos o problema original, para encontrar uma aproximação para e^x no intervalo [-1.1513, 1.1513]. Para ilustrar, tomemos N = 4. Da série de Maclaurin para e^x obtemos $c_0 = 1$, $c_1 = 1$, $c_2 = 1/2$, $c_3 = 1/6$ e $c_4 = 1/24$. Usando as equações (3.7) e (3.8) obtemos

$$\begin{cases} \frac{1}{24} + \frac{1}{6} b_1 + \frac{1}{2} b_2 = 0 \\ \frac{1}{6} + \frac{1}{2} b_1 + b_2 = 0 \end{cases}$$

$$\begin{cases} a_0 = 1 \\ a_1 = 1 + b_1 \\ a_2 = \frac{1}{2} + b_1 + b_2 \end{cases}$$

Resolvendo as primeiras duas equações para b_1 e b_2 e usando as ultimas três para calcular a_0, a_1 e a_2, obtemos

$$b_1 = -1/2, \; b_2 = 1/12$$

$$a_0 = 1, \; a_1 = 1/2 \text{ e } a_2 = 1/12$$

então

$$R_{2,2}(x) = \frac{12 + 6x + x^2}{12 - 6x + x^2}$$

De modo semelhante podemos variar m e k para encontrar os outros elementos da tabela de Padé para N = 4

$$R_{4,0}(x) = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4$$

$$R_{3,1}(x) = \frac{24 + 18x + 6x^2 + x^3}{24 - 6x}$$

$$R_{1,3}(x) = \frac{24 + 6x}{24 - 18x + 6x^2 - x^3}$$

$$R_{0,4}(x) = \frac{1}{1 - x + \frac{1}{2}x^2 - \frac{1}{6}x^3 + \frac{1}{24}x^4}$$

Na tabela 3.1 abaixo, vimos que no intervalo [-1.1513, 1.1513] $R_{3,1}(x)$ tem o mínimo máximo erro -0.00882, o que contraria as previsões de que m = k ou m = k + 1 seria a melhor estimativa. No entanto, no intervalo [-0.75, 0.75], $R_{2,2}(x)$ é certamente a melhor aproximação. Note também que nos extremos dos intervalos os erros crescem mais rapidamente. Isso sugere que quando possível, aproximemos funções em pequenos intervalos.

Tabela 3.1: Erros na aproximação de Padé para e^x - R(m,k)

x \ (m,k)	(4,0)	(3,1)	(2,2)	(1,3)	(0,4)
-1.15	-.01401	.00226	-.00096	.00088	-.00208
-1.00	-.00712	.00121	-.00054	.00053	-.00135
- .75	-.00175	.00033	-.00016	.00018	-.00050
- .50	-.000240	.000049	-.000027	.000032	-.000104
- .25	-.0000078	.0000018	-.0000011	.0000014	-.0000052
.25	.0000086	-.0000023	.0000018	-.0000029	.0000129
.50	.000284	-.000088	.000073	-.000134	.000653
.75	.00225	-.00079	.00072	-.00147	.00783
1.00	.00995	-.00394	.00400	-.00899	.05162
1.15	.02059	-.00882	.00950	-.02274	.13381

Em geral se tem duas alternativas quando uma aproximação para um dado N não for suficientemente boa em todo intervalo: (1) desdobrar o intervalo em um ou mais subintervalos e usar a aproximação adequada em cada subintervalo; (2) usar sempre um valor de N grande para conseguir maior precisão. A desvantagem disso é que se necessitam de mais operações e, consequentemente mais tempo será necessário para se obter o resultado.

Na aproximação de Padé, o erro cresce rapidamente à medida que se afasta do ponto médio do intervalo de aproximação.

3.9 Considerações Gerais

Para o uso em aritmética de múltipla precisão a melhor aproximação seria a racional com pequenos intervalos. O maior problema, e que a torna inviável, é que como a precisão é variável isso implicaria em calcular os coeficientes de $P_m(x)$ e $Q_k(x)$ em (3.4) a cada nova precisão escolhida bem como os graus m e k, e só depois disso é que poderíamos computar a aproximação. Para cada nova precisão novos cálculos dos coeficientes de $P_m(x)$, $Q_k(x)$ e os graus m e k são necessários, isso acarretaria um tempo muito grande para computar uma aproximação.

CAPÍTULO IV

UM PACOTE DE SUBPROGRAMAS EM ARITMÉTICA DE MÚLTIPLA PRECISÃO

4.1 Introdução

Neste capítulo será descrito um pacote de subprogamas para executar operações aritméticas em ponto flutuante utilizando múltipla precisão. O pacote também é capaz de avaliar funções elementares, trigonométricas e suas respectivas inversas.

Os algoritmos das operações aritméticas foram baseados nos algoritmos para realizar as quatro operações com inteiros utilizando o conceito de múltipla precisão e dígito de guarda vistos no capítulo 2.

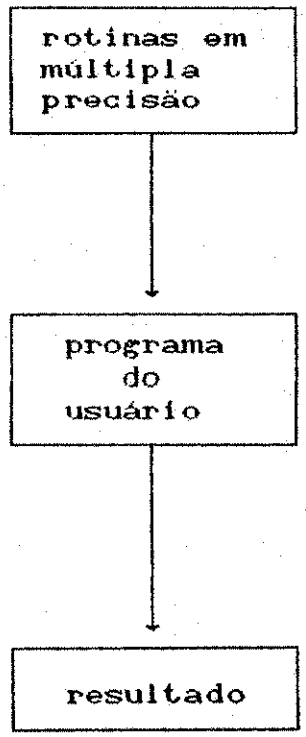
Para a avaliação das funções, utilizamos séries e o processo iterativo de Newton porque não pudemos usar aproximação racional devido a precisão ser variável. Alguns algoritmos que os capítulos 2 e 3 sugeriram que fossem implementados tiveram que ser alterados ou substituídos por outros.

Este pacote servirá de base para a construção de uma calculadora científica on-line para ambientes de microcomputadoes compatíveis com IBM-PC.

4.2 Aspectos do Pacote

O pacote é uma coleção de subrotinas em Fortran 77 para executar aritmética de ponto flutuante utilizando múltipla precisão. A precisão pode ser variada dinamicamente, ou seja, o limite para os números será a própria memória do computador. As rotinas foram compiladas separadamente, portanto o usuário poderá escvrever seus programas normalmente e quando quiser fazer alguma operação aritmética em precisão múltipla poderá fazê-la utilizando as subrotinas do pacote.

Exemplo: C = A + B ⇒ call MPSOMA(A, B, C, P) onde A, B, C são números em múltipla precisão com precisão P.



Um número em múltipla precisão é representado por um vetor do tipo inteiro de no mínimo p+4 posições. Onde p é a precisão fornecida pelo usuário. A primeira posição é utilizada para o sinal do número - positivo ou negativo - a segunda, para guardar o expoente do número, da terceira à p+2-ésima posição, para fração normalizada do número, e as duas últimas posições são reservadas para os dígitos de guarda. E é sobre o primeiro dígito de guarda que é feito o arredondamento ou truncamento.

Na figura abaixo está representado um número em múltipla precisão:

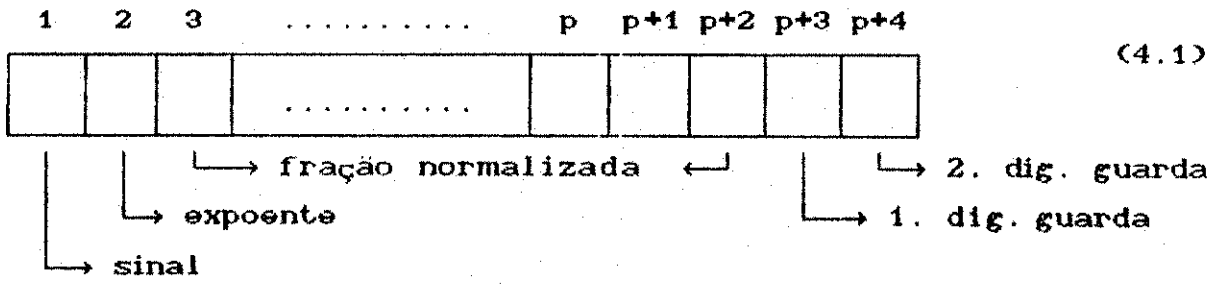


fig 4.1 - Um número em m últipla precisão

O pacote é quase completamente independente de máquina. Foi escolhido o FORTRAN 77 por ser largamente utilizado em computação numérica e também por gerar programas objetos eficientes. Uma outra razão para sua escolha é que precisávamos de um ambiente de múltipla precisão para o projeto MICRO-BITAN: Biblioteca Transportável de Algoritmos Numéricos para Microcomputadores que está escrito em Fortran 77, e em que a transportabilidade é um

dos objetivos almejados.

4.3 Conteúdo do Pacote

Na versão atual, o pacote possui rotinas para realizar:

a) Conversão de números inteiros, reais e strings para múltipla precisão e vice-versa;
b) As quatro operações aritméticas: adição, subtração, multiplicação e divisão;
c) Recíproca de x, $\left(\frac{1}{x} \right)$;
d) Funções elementares: log, ln, n!, e^x, x^y e $\sqrt{x}$;
e) Funções trigonométricas: sin, cos e tan;
f) Funções trigonométricas inversas: arcsin, arccos e arctan;

Não implementamos outras funções, porque algumas podem ser obtidas dessas já implementadas, tais como combinação, arranjo e permutação. E outras funções poderão, dependendo da necessidade, ser implementadas a partir das quatro operações aritméticas e das funções já disponíveis.

4.4 Algoritmos Utilizados

O objetivo dessa fase, foi achar sempre o algoritmo mais rápido. Alguns algoritmos que os capítulos 2 e 3 sugeriram que fossem implementados tiveram que ser alterados ou substituídos por outros devido ao tempo de resposta ou mesmo por serem inviáveis, como é o caso da aproximação racional para computar funções. Por

exemplo, a implementação do algoritmo de [KNUT69] para a divisão (2.12.4) de números em múltipla precisão não é boa comparada à sugerida em [MARO87].

4.4.1 Operações aritméticas

Os algoritmos abaixo foram baseados nos algoritmos das operações aritméticas com inteiros utilizando os conceitos de múltipla precisão vistos no capítulo 2 onde um número é representado na forma (4.1).

4.4.1.1 Adição

Foi adaptado o algoritmo de adição de números inteiros em múltipla precisão (2.12.1) para se trabalhar com números sob a forma (4.1):

```
procedure MPSOMA(A, B, R, P, ERRO)
(R = A + B, onde A, B, R são números em múltipla precisão com
precisão P e ERRO é a variável que verifica se houve
overflow)
  begin
    N := P + 4
    R := 0
    IGUALAR_EXPOENTE(A, B)
    (escolhe-se o número com menor expoente e desloca-se sua
    mantissa para a direita um número igual a diferença entre
    os expoentes)
```

```
R(1) := SINAL(A, B) {atribui sinal ao resultado}
R(2) := A(2) {atribui ao expoente do resultado o expoente
              de A}
T := POSIÇÃO_A_DIREITA(A, B)
{retorna a posição mais a direita de A, B diferente de
zero}
CARRY := 0
for I := N to 3 step -1 do
  R(I) := A(I) + B(I) + CARRY
  if (R(I) > 9) then
    R(I) := R(I) - 10
    CARRY := 1
  else
    CARRY := 0
  end_if
end_for
if (overflow) then
  ERRO := 1
else
  ERRO := 0
end_if
NORMALIZAR(R) {normaliza R se R(3) = 0}
ARREDONDAR(R) {arredonda R se R(N - 1) ≥ 5}
end.
```

4.4.1.2 Subtração

Foi adaptado o algoritmo de subtração de números inteiros em

múltipla precisão (2.12.2) para se trabalhar com números sob a forma (4.1):

```
procedure MPSUB(A, B, R, P)
{R = A - B, onde A, B, R são números em múltipla precisão com
precisão P}
  begin
    N := P + 4
    if (A < B) then
      TROCA(A, B)
    end_if
    R := 0
    IGUALAR_EXPOENTE(A, B)
    {escolhe-se o número com menor expoente e desloca-se sua
    mantissa para a direita o número igual a diferença entre
    os expoentes}
    R(1) := SINAL(A, B) {atribui sinal ao resultado}
    R(2) := A(2) {atribui ao expoente do resultado o expoente
                  de A}
    T := POSIÇÃO_À_DIREITA(A, B)
    {retorna a posição mais a direita de A, B diferente de
    zero}
    CARRY := 0
    for I := N to 3 step -1 do
      if (A(I) ≥ B(I) + CARRY) then
        R(I) := A(I) - (B(I) + CARRY)
        CARRY := 0
      else
```

```
            R(I) := (A(I) +10 ) - (B(I) + CARRY)
            CARRY := 1
          end_if
        end_for
        NORMALIZAR(R) {normaliza R se R(3) = 0}
        ARREDONDAR(R) {arredonda R se R(N - 1) ≥ 5}
      end.
```

4.4.1.3 Multiplicação

Este algoritmo também foi adaptado do algoritmo de multiplicação de números inteiros em múltipla precisão (2.12.3) para se trabalhar com números sob a forma (4.1):

```
procedure MPMULT(A, B, R, P, ERRO)
{R = A * B, onde A, B, R são números em múltipla precisão com
precisão P e ERRO verifica se hove overflow ou underflow}
  begin
    M := P + 4
    if (A > B) then
      TROCA(A, B)
    end_if
    R := 0
    R(1) := A(1) * B(1) {atribui sinal ao resultado}
    R(2) := A(2) + B(2) {expoente do resultado igual ao
                         expoente de A + B}
    if (overflow) or (underflow) then
      ERRO := 1
```

```
    return
  else
    ERRO := 0
    T1 := POSIÇÃO_À_DIREITA(A)
   (retorna a posição mais a direita de A diferente de
    zero)
    T2 := POSIÇÃO_À_DIREITA(B)
   (retorna a posição mais a direita de B diferente de
    zero)
    T := T1 + T2 - 2
   (número de dígitos necessário para o resultado)
    for I := T1 to 3 step -1 do
      N := M
      CARRY := 0
      for J :=  T2 to 3 step -1 do
        R(N) := R(N) + A(I) * B(J) + CARRY
        if (R(N) > 9) then
          R(N) := mod(R(N), 10)
          CARRY := int(R(N) / 10)
        else
          CARRY := 0
        end_if
        N := N - 1
      end_for
    end_for
    NORMALIZAR(R) (Normaliza R se R(3) = 0)
    ARREDONDAR(R) (arredonda R se R(M - 1) ≥ 5)
  end_if
```

end.

4.4.1.4 Divisão

Primeiramente fizemos a modificação no algoritmo (2.12.4) para que um número em múltipla precisão fosse armazenado em vetor, e quando implementamos no computador, seu desempenho não foi muito bom. Em vista disso, usamos a sugestão de [MAR087]

$$\frac{Y}{X} = Y \cdot \frac{1}{X}$$

porque o algoritmo para calcular 1 / X é bastante rápido. Como a multiplicação também ficou rápida, conseguimos uma resposta para Y / X bem melhor do que a anterior. Sendo X um número em múltipla precisão na forma (4.1), o algoritmo para a obtenção de 1 / X é o seguinte. Seja

$$y = \frac{1}{x}$$

então resolvemos a equação $f(y) = x - \frac{1}{y}$ pelo método de Newton,

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$$

ou

$$y_{n+1} = y_n.(2 - x.y_n)$$

onde y_{n+1} são as sucessivas aproximações para y.

Fazendo $x = x_1.10^m$, onde $0 \leq x_1 < 1$, podemos usar $y_o = 10^{-m}$ como o valor inicial da iteração sugerido por [MARO87]. Este processo iterativo converge muito rapidamente, dobrando o número de dígitos corretos a cada iteração. Dessa forma, conseguimos obter um algoritmo para divisão usando apenas as operações aritméticas de multiplicação e subtração explicadas anteriormente. Mesmo com essas mudanças, o algoritmo para a divisão ficou mais lento comparado com as outras operações aritméticas, adição, subtração e multiplicação. Isso se deve ao fato de outros processos não utilizarem aproximações sucessivas e sim processos diretos.

4.4.2 Funções Elementares

4.4.2.1 Raiz Quadrada - sqrt(x)

Tentamos vários algoritmos, mas o que melhor se comportou foi o sugerido por [BREN78]. O algoritmo é o seguinte

Seja $y = x^{-1/2}$ (4.2)

fazendo

$$x = x_1.10^m$$

onde

$$0 \leq x_1 < 1 \quad \text{e} \quad m \text{ um inteiro,}$$

então

$$y = x_1^{-1/2} \cdot 10^{-m/2}$$

chamando

$$z = x_1^{-1/2}$$

e resolvendo a equação $f(z) = z - x_i^{-1/2}$ pelo método de Newton obtém-se a relação

$$z_{n+1} = z_n \cdot (3 - x_i \cdot z_n^2) \times 0.5$$

e adotamos o valor inicial $z_0 = 0.5$ sugerido por [MAR087].

De posse posse de $x^{-1/2}$, efetuamos a multiplicação por $10^{-m/2}$. Ora se m for par, apenas multiplicamos por 10 m/2 vezes. E se m for impar decompomos $10^{-m/2}$ em $10^{-(m-1)/2} \times 10^{-1/2}$.

Como (m-1) / 2 é um inteiro, fazemos a multiplicação por 10 (m-1) / 2 vezes. E $10^{-1/2}$ é calculado novamente pela equação (4.2), mas nesse caso m = 0. Uma vez calculado, $10^{-1/2}$ é uma constante para o sistema naquela precisão. Não sendo mais necessário calculá-lo desde que não mude a precisão.

Portanto, com $10^{-m/2}$ e $x_i^{-1/2}$ calculados, podemos então calcular $y = x_i^{-1/2} \times 10^{-m/2} = x^{-1/2}$.

Agora basta multiplicar y por x que teremos $\sqrt{x}$. O algoritmo parece gastar bastante tempo. Mas se observarmos com atenção, conseguimos obter a raiz quadrada sem usar a operação de divisão, o que o torna relativamente rápido.

4.4.2.2 Potenciação geral

Implementamos apenas o caso de y ser inteiro, pois caso

contrário podemos obter x^y calculando

$$x^y = e^{y.\ln(x)}$$

com a utilização de ln(x) e e^x, disponíveis no pacote.

Para y inteiro calculamos o produto de y fatores iguais a x, x . x ... x. Um algoritmo elegante sugerido por [MARO87] é

```
function POWER(B:número, E:integer): número
  begin
    if (E < 0) then
      POWER := -1 / POWER(B, E)
    else
      if (E = 0) then
        POWER := 1
      else
        if (E = 1) then
          POWER := B
        else
          if ((E mod 2) = 0) then
            POWER := (POWER(B, E div 2))^2
          else
            POWER := (POWER(B, E div 2))^2 * b
          end_if
        end_if
      end_if
    end_if
```

end.

Esse algoritmo é mais lento que o anterior para números em múltipla precisão não muito grande. A recursividade, calcular o quadrado e o resto da divisão de um número por 2, comprometeram muito esse algoritmo. Outra razão bastante forte para a escolha de x . x ... x y vezes para aproximar x^y, foi que nos algoritmos para aproximar funções utilizando séries, e que utilizam x^y, o valor de y é pequeno.

4.4.2.3 Fatorial - fat(x)

O algoritmo é a própria definição de fatorial

$$n! = n \times (n-1) \times \ldots \times 1,$$

$$0! = 1,$$

onde n é um número inteiro em multipla precisão da forma (4.1).

Como o algoritmo só tem as operações aritméticas de multiplicação e subtração, o tempo de resposta ficou bom.

4.4.2.4 Exponencial - exp(x)

Neste algoritmo primeiro fazemos a decomposição de x em

$$x = n + f$$

onde

n é um inteiro,

f um número tal que $0 \leq f < 1$,

portanto $e^x = e^n \times e^f$, mas $e^n = e \cdot e \ldots e$ n vezes e o valor de e e e^f são dados pelas séries

$$e = 1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

$$e^f = 1 + f + \frac{f^2}{2!} + \frac{f^3}{3!} + \ldots$$

Decompomos e^x em $e^n \cdot e^f$ porque uma vez calculado, e se torna constante para uma determinada precisão, não sendo mais calculado. Portanto, só temos que calcular e^f.

4.4.2.5 Logaritmo Neperiano - ln(x)

Primeiramente tentamos avaliar ln(x) pelo método iterativo de Newton sugerido por [BREN78] usando e^x, ou seja, sendo

$$\ln(x) = a$$

então

$e^a = x$, onde a e x são dois números quaisquer e $x > 0$.

Usando o processo iterativo para aproximar a função $y = e^a - x$, obtemos

$$y_{n+1} = y_n - 1 + \frac{x}{e^{y_n}}.$$

O tempo de resposta não foi bom, pois e^x, explicada em (4.4.2.4), utiliza uma série que usa divisão. Então partimos para uma série que convergisse diretamente para ln(x) sem usar o processo iterativo de Newton.

Para ln(x) fizemos $x = x_1.10^m$, onde $0 \leq x_1 < 1$ e m um inteiro. Portanto

$$\ln(x) = \ln(x_1) + m \cdot \ln(10). \qquad (4.3)$$

Para aproximar $\ln(x_1)$ e ln(10) usamos a série conhecida

$$\ln(z) = 2 \cdot \left[\left(\frac{z-1}{z+1}\right) + \frac{1}{3}\left(\frac{z-1}{z+1}\right)^3 + \frac{1}{5}\left(\frac{z-1}{z+1}\right)^5 + \ldots \right] \qquad (4.4)$$

A série acima tem a vantagem de que uma vez calculado, o termo $\left(\frac{x-1}{x+1}\right)$ se torna uma constante naquela precisão. E os fatores $\frac{1}{3}$, $\frac{1}{5}$, $\frac{1}{7}$ são facilmente computados. Outra vantagem da decomposição (4.3) é que ln(10) também se torna uma constante. O tempo de resposta do segundo algoritmo foi bem melhor que o do primeiro.

4.4.2.6 Logaritmo Decimal - log(x)

Como em ln(x), tentamos calcular log(x) pelo processo iterativo de Newton sugerido por [BREN78]. Ou seja

$$y_{n+1} = y_n - 1 + \frac{x}{10^{y_n}}$$

seriam as aproximações para log(x). O problema que encontramos foi na hora de calcular 10^{y_n}, pois se y_n não fosse inteiro, precisaríamos de ln(x) para calculá-lo. Ora como iríamos precisar de ln(x) para calcular log(x), então simplesmente usamos a identidade bem conhecida

$$\log(x) = \frac{\ln(x)}{\ln(10)}$$

que foi mais rápido. Além do mais ln(10) é uma constante para uma determinada precisão.

4.4.3 Funções Trigonométricas

Na verdade bastaria ser implementado uma função trigonométrica para que todas as outras pudessem ser obtidas. Mas como na matemática é muito frequente o uso de todas, implementamos seno, cosseno e tangente e suas respectivas inversas mostradas mais adiante. As funções trigonométricas são calculadas apenas em radianos e todos os valores de seus argumentos são convertidos para o primeiro quadrante, ou seja, maior ou igual a zero e menor ou igual $\Pi / 2$.

4.4.3.1 Função Seno - sin(x)

O valor de sin(x) é calculado pela série de Taylor

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, \quad 0 < x \leq \Pi/2$$

caso

$$\frac{\Pi}{2} < x \leq \Pi \quad \text{então } \sin(x) = \sin(\Pi - x);$$

$$\Pi < x \leq \frac{3\Pi}{2} \quad \text{então } \sin(x) = -\sin(x - \Pi);$$

$$\frac{3\Pi}{2} < x \leq 2\Pi \quad \text{então } \sin(x) = -\sin(2\Pi - x).$$

E se $x > 2\Pi$ então fazemos

$$a = \frac{x}{2\Pi}$$

sendo b a parte decimal de a, reduzimos x para $x = b \cdot 2\Pi$.

4.4.3.2 Função cosseno - cos(x)

Primeiramente tentamos calcular cos(x) pela identidade

$$\sin^2(x) + \cos^2(x) = 1$$

então

$$\cos(x) = \sqrt{1 - \sin^2(x)}$$

mas o tempo de resposta não ficou bom, pois além da série para calcular sin(x), precisaríamos do processo de Newton para calcular $\sqrt{x}$. Então decidimos também usar a série de Taylor para calcular cos(x)

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, \quad 0 < x \leq \Pi/2$$

caso

$$\frac{\Pi}{2} < x \leq \Pi \quad \text{então } \cos(x) = -\cos(\Pi - x);$$

$$\Pi < x \leq \frac{3\Pi}{2} \quad \text{então } \cos(x) = -\cos(x - \Pi);$$

$$\frac{3\Pi}{2} < x \leq 2\Pi \quad \text{então } \cos(x) = \cos(2\Pi - x).$$

E se $x > 2\Pi$ então fazemos

$$a = \frac{x}{2\Pi}$$

sendo b a parte decimal de a, reduzimos x para $x = b \cdot 2\Pi$.

4.4.3.3 Função Tangente - tan(x)

Para calcular tan(x), simplesmente usamos a igualdade

$$\tan(x) = \sin(x) \cdot [(1 - \sin^2(x))^{-1/2}]$$

onde sin(x) utiliza o algoritmo explicado em (4.4.3.1).

4.4.4 Funções Trigonométricas Inversas

4.4.4.1 Função Arco-tangente - arctan(x)

O valor de arctan(x) é calculado pela bem conhecida série de Taylor

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots, \quad |x| \leq 1/2$$

e caso $|x| \geq 1/2$ então usamos a igualdade

$$\arctan(x) = 2 \cdot \arctan\left[\frac{x}{1 + (1 + x^2)^{1/2}}\right]$$

para reduzir x.

4.4.4.2 Função arco-seno - arcsin(x)

O valor de arcsin(x) é calculado em função de arctan(x) pela igualdade

$$\arcsin(x) = \arctan\left[x \cdot (1 - x^2)^{-1/2}\right], \quad |x| < 1.$$

4.4.4.3 Função arco-cosseno - arccos(x)

O valor de arccos(x) é calculado em função de arcsin(x) pela igualdade

$$\arccos(x) = \frac{\Pi}{2} - \arcsin(x), \quad |x| < 1.$$

O valor de π usado para calcular as funções trigonométricas foi obtido pelo método de Brent-Salamin usando média aritmética-geométrica:

```
function PI: number
  var a, b, t, x, tempa: number
  begin
    a := 1
    b := sqrt(0.5)
    t := 0.25
    x := 1
    while (a - b) > ε do
      tempa := a
      a := (a + b) / 2
      b := sqrt(tempa * b)
      t := t - x * (a - tempa)^2
      x := 2 * x
    end_while
    PI := (a + b)^2 / (4 * t)
  end
```

CAPÍTULO V

CALCULADORA ON-LINE (CALCON)

5.1 Introdução

Neste capítulo descreveremos a **CALCON**, uma calculadora científica cuja precisão pode ser alterada dinamicamente. Essa calculadora é baseada no pacote aritmético descrito no capítulo anterior. A calculadora é capaz de executar as quatro operações aritméticas, computar funções elementares, trigonométricas, suas respectivas inversas, as principais constantes matemáticas e físicas podem ser referenciadas simbolicamente e aceitar expressões envolvendo as operações, constantes e funções acima.

Veremos quais os requisitos da calculadora, e como é feita a análise léxica e sintática para uma expressão válida.

5.2 Requisitos exigidos

Como requisitos, a calculadora deve ser capaz de

a) executar as quatro operações aritméticas: adição, subtração, multiplicação e divisão;

b) computar funções elementares: x^y, $\sqrt{x}$, log(x), ln(x), n!, e^x;

c) computar funções trigonométricas: sin(x), cos(x) e tan(x);

d) computar funções trigonométricas inversas: arcsin(x), arccos(x) e arctan(x);

e) aceitar algumas constantes: PI, PLANK, velocidade da luz, etc;

f) aceitar e avaliar expressões que envolvam as operações aritméticas, funções e constantes acima;

g) antes de avaliar uma expressão, aceitar mudança de precisão;

h) ter algum tipo de memória.

5.2.1 Alteração da Precisão

Como foi visto no capítulo 2 podemos obter resultados errados em um cálculo por falta de precisão. CALCON nos dá a facilidade de aumentar a precisão até que se obtenha o grau desejado de precisão. Isso é feito informando a precisão quando a calculadora a solicita. Porém como um número em múltipla precisão foi armazenado sob forma de vetor de inteiros, existe uma precisão máxima permitida (tamanho desse vetor). Para que a máxima precisão possa ser alterada, no programa CALCON.FOR o valor da variável L deve ser alterada, recompilar o programa e executar a ligação. Dessa forma, se a escolha da precisão for maior que a precisão máxima permitida será emitida a mensagen 'PRECISAO ASSUMIDA DE # DIGITOS', onde # é a precisão máxima.

5.2.2 Referência simbólica a constantes

Implementamos as constantes mais usadas na matemática, física e engenharia segundo [ABRA68]. Essas constantes são referenciadas simbolicamente por

PI - valor de pi;

C - velocidade da luz no vácuo;

BOHR0 - raio de Bohr;

BOHRM - magnéton de Bohr;

BOLTZ - constante de Boltzmann;

CE - comprimento de onda do elétron;

CP - comprimento de onda do próton;

CR1 - primeira contante de radiação ($2\pi HC^2$);

CR2 - segunda constante de radiação;

E - carga elementar;

F - constante de Faraday;

G - constante gravitacional;

GAMA - constante Gama;

GAS - constante Gas;

H - constante de Plank;

MAGN - magnéton nuclear;

MOMP - momento do próton;

ME - massa do elétron;

MN - massa do neutron;

MP - massa do próton;

NA - constante de Avogrado;

RE - raio do elétron;

RY - constante de Rydberg;

SBOLT - constante de Stefan-Boltzmann;

VNOR - volume normal de gas perfeito;

WIEN - constante de deslocamento de Wien;

ZEE - constante de Zeeman;

GO - aceleração da gravidade a nível do mar;

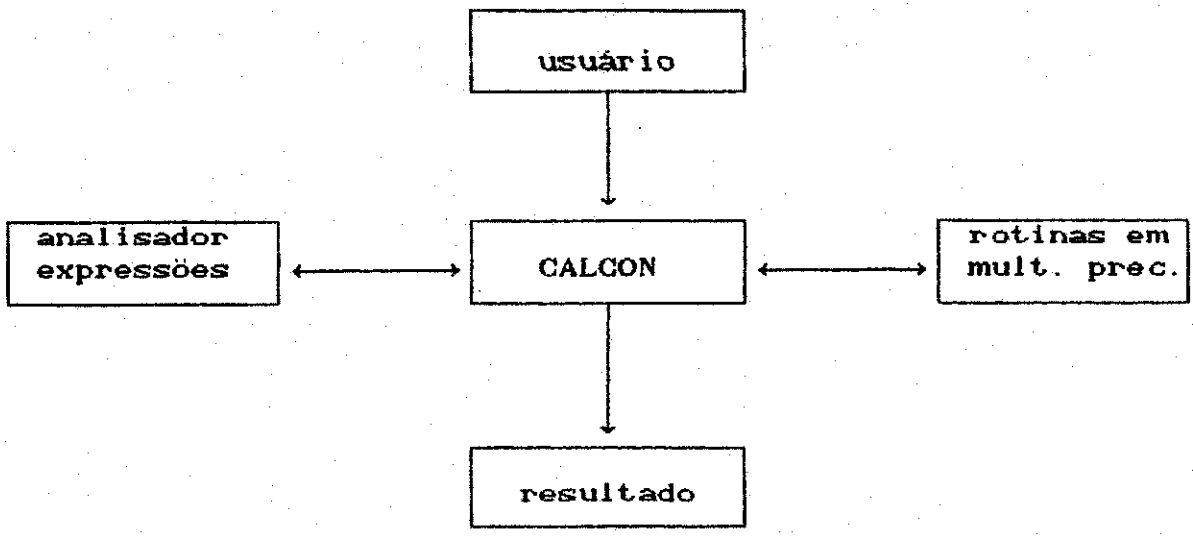
P0 - pressão atmosférica a nível do mar.

5.3 A organização da calculadora

A calculadora CALCON é uma interface simples que permite a um usuário ter acesso limitado às rotinas do pacote de múltipla precisão descrito no capítulo anterior.

Expressões aritméticas são o meio de comunicação utilizado pela CALCON.

A organizaçao da calculadora é exibida no diagrama a seguir



Após a ativação e algum protocolo inicial, a calculadora aceita uma expressão satisfazendo regras gramaticais simples, faz a análise da validade da expressão, aciona as rotinas de múltipla precisão necessárias para a avaliação da expressão e finalmente

exibe o resultado dessa avaliação no vídeo.

5.4 A gramática usada nas expressões

A calculadora reconhece as expressões aritméticas que satisfazem a seguinte BNF.

Metassimbolo	Significado
=	definido por
|	alternativamente
	final do esquema
[x]	0 ou 1 instância de x
{x}	0 ou mais instâncias de x
(x|y)	um argumento: ou x ou y
"xyz"	o símbolo terminal xyz

ExpressãoSimples = [Sinal]Termo{TermoOperadorAdição}

Sinal = "+"|"-"

Termo = Fator{FatorOperadorMultiplicação}

Fator = ConstanteSemSinal|DesignadorFunção|"("ExpressãoSimples")"

ConstanteSemSinal = NúmeroSemSinal|IdentificadorConstante

NúmeroSemSinal = InteiroSemSinal|RealSemSinal

InteiroSemSinal = SequênciaDígito

SequênciaDígito = Dígito{Dígito}

Dígito = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

RealSemSinal = InteiroSemSinal"."SequênciaDígito"e" FatorEscalar
InteiroSemSinal"e"FatorEscalar

FatorEscalar = [Sinal]InteiroSemSinal

IdentificadorConstante = Identificador

Identificador = Letra{Letra|Digito}

Letra = "A"|"B"|"C"|"D"| ... |"Z"|"a"|"b"|"c"|"d"| ... |"z"|

DesignadorFunção = IdentificadorFunção[ListaParâmetroReal]

IdentificadorFunção = Identificador

ListaParâmetroReal = ExpressãoSimples

OperadorMultiplicação = "*"|"**"|"/"

OperadorAdição = "+"|"-"

Se a expressão for muito grande (mais de uma linha no terminal), a calculadora aceita o desdobramento em sub-expressões. O último resultado fica sempre armazenado em uma variável e pode ser recuperado através do símbolo =.

Exemplo: Seja a expressão

1.234234234 * 8657648654 + sin(cos(pi / sqrt(2))) - c * sin(0.234) + exp(7.875) / gama.

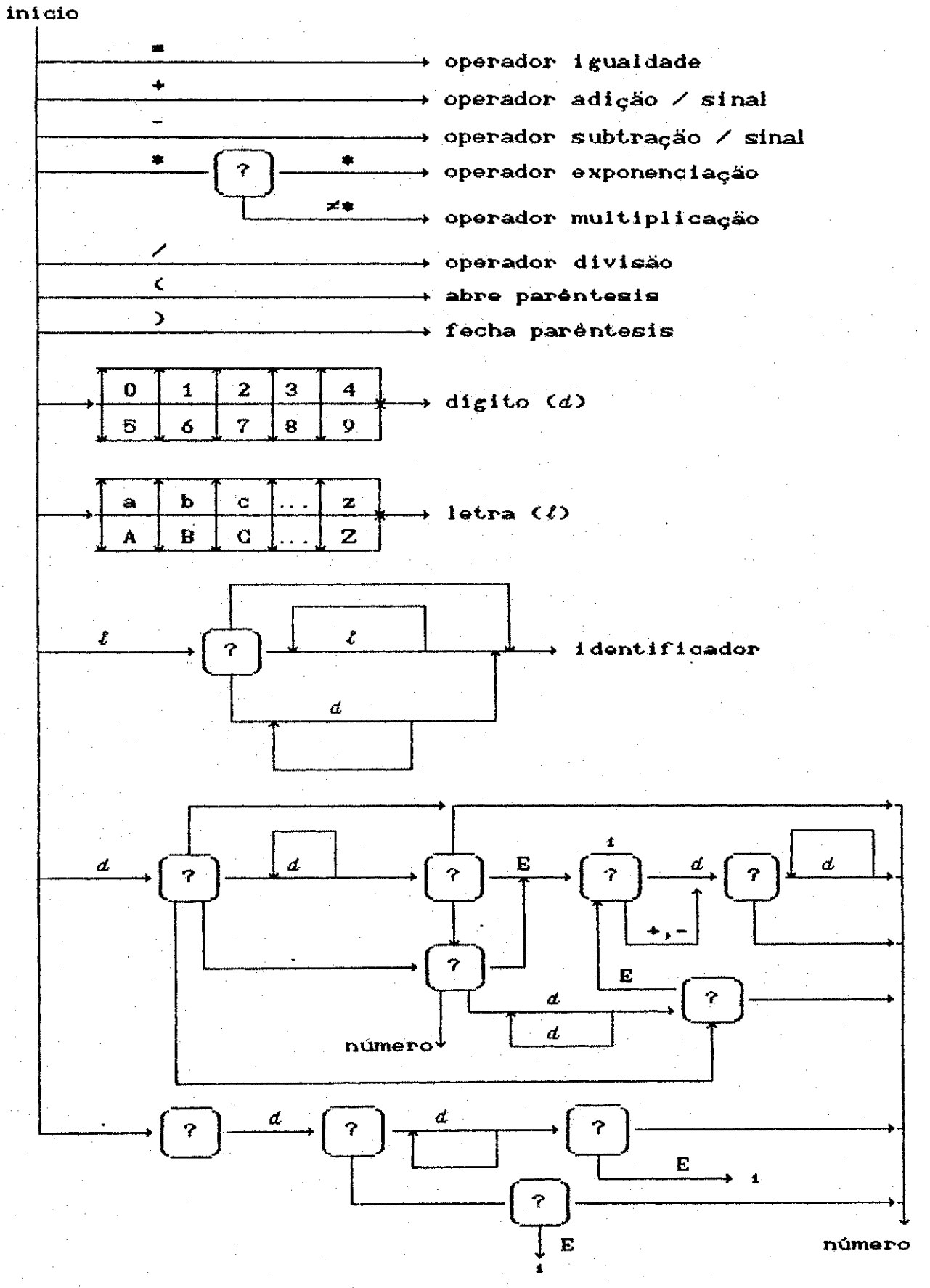
Podemos, por exemplo, entrar com duas expressões da forma

(1) 1.234234234 * 8657648654 + sin (cos(pi / sqrt(2))) - c

(2) = * sin(0.234) + exp(7.875) / gama.

5.5 Análise léxica

A análise léxica foi implementada usando o autômato finito da figura a seguir com base na BNF citada em 5.4:


início
=
operador igualdade
+
operador adição / sinal
-
operador subtração / sinal
*
?
*
operador exponenciação
≠*
operador multiplicação
/
operador divisão
(
abre parêntesis
)
fecha parêntesis
0 1 2 3 4
5 6 7 8 9
dígito (d)
a b c ... z
A B C ... Z
letra (l)
l
?
l
d
identificador
d
?
d
?
E
1
?
d
?
d
+,-
?
E
?
d
d
número
?
d
?
d
?
E
1
?
E
1
número

Este autômato reconhece os símbolos:

a) Operadores Aritméticos: +, -, *, /, **;

b) Símbolos Especiais: =, (,);

c) Funções elementares, trigonométricas e suas inversas citadas em 5.2;

d) Constantes citadas em 5.2.2;

e) Números: inteiros, reais e na forma exponencial.

5.6 Análise Sintática

Para fazer a análise sintática das expressões lidas, comparamos os tokens, fornecidos pelo analisador léxico, com a **bnf** das expressões aritméticas. Caso os tokens não estejam de acordo com a gramática, uma mensagem de erro é emitida no vídeo e o sistema fica esperando uma outra expressão.

5.7 Avaliação das expressões

Os tokens são lidos e armazenados em duas pilhas. Pilha 1, pilha de operandos e Pilha 2, pilha de operadores. Quando for encontrado um), a prioridade de um operador de fora da pilha for menor do que a prioridade do operador de dentro da pilha ou ainda quando terminar a expressão lida, as pilhas são desempilhadas e os resultados obtidos são empilhados novamente até que a base da pilha 1 conterá apenas um elemento, que é o resultado da expressão. Essa implementação foi sugerida por [HORO84].

As prioridades dos operadores dentro e fora da pilha estão relacionadas abaixo:

operador	prioridade	
	dentro	fora
**	3	4
*	2	2
/	2	2
+	1	1
-	1	1
(	0	4
função	0	4
constante	0	4

5.8 Utilização da calculadora

CALCON é uma calculadora científica cuja precisão pode ser alterada dinamicamente. O usuário usa a calculadora da seguinte forma: o sistema pede do usuário a precisão que deseja trabalhar, o usuário digita um número inteiro seguido de <enter>, em seguida entra com uma expressão aritmética e tecla novamente <enter>. É feita uma análise léxica e sintática na expressão lida, e caso haja algum erro a calculadora informa e fica esperando uma nova expressão. Para mudar de precisão basta digitar <enter> na leitura da expressão que em seguida lhe será pedido uma nova precisão. Para sair da calculadora basta o usuário digitar <enter> na leitura da precisão. Para repetir a última expressao basta teclar F3.

CAPÍTULO VI

CONCLUSÃO

6.1 Ambiente ON-LINE

O maior problema encontrado na implementação do pacote de múltipla precisão foi o tempo de execução dos algoritmos. Alguns algoritmos utilizados não tiveram bom desempenho. Precisávamos de respostas rápidas devido ao ambiente da calculadora ser on-line. Os algoritmos sugeridos por [BREN78] são adequados para um ambiente de processamento em lote onde a eficiência não tem um impacto tão acentuado sobre o usuário. Por isso alguns algoritmos sugeridos nos capítulos 2 e 3 tiveram que ser modificados ou mesmo substituídos para que pudessem ter um desempenho (tempo de resposta) adequado a um ambiente on-line.

6.2 Linguagem de implementação

Precisávamos de uma linguagem que gerasse código executável eficiente para compensar o tempo gasto pelos algoritmos. A escolha do FORTRAN-77 se deveu à necessidade de inserir o pacote de múltipla precisão no software do projeto MICRO-BITAN (Biblioteca Transportável de Algoritmos Numéricos para Microcomputadores compatíveis com IBM-PC) desenvolvido em FORTRAN. Porém pagamos um preço alto por isso. Tivemos que utilizar uma estrutura estática, vetor, para representar um número em múltipla precisão ao invés de usar alocação dinâmica. E quando precisamos de uma precisão maior

do que o tamanho do vetor, temos que editar o programa de instalação e mudar esse tamanho, recompilar e ligar com o resto do pacote. Uma outra desvantagem do FORTRAN é que quando definimos um vetor do tipo inteiro para representar um número em múltipla precisão, cada posição desse vetor gasta 32 bits, quando na verdade bastariam 4 bits, pois cada posição do vetor precisa aramazenar apenas um dígito. Dessa forma desperdiçamos memória. A maior vantagem da alocação dinâmica seria o fato de podermos realmente operar com precisão infinita com inteiros. Quanto ao gasto de memória, não seria tão diferente do FORTRAN pois para cada dígito, iríamos precisar de um apontador para o próximo dígito. O ideal seria que pudéssemos ter um vetor do tipo dígito onde cada posição pudesse armazenar somente um dígito e que o seu tamanho variasse em tempo de execução.

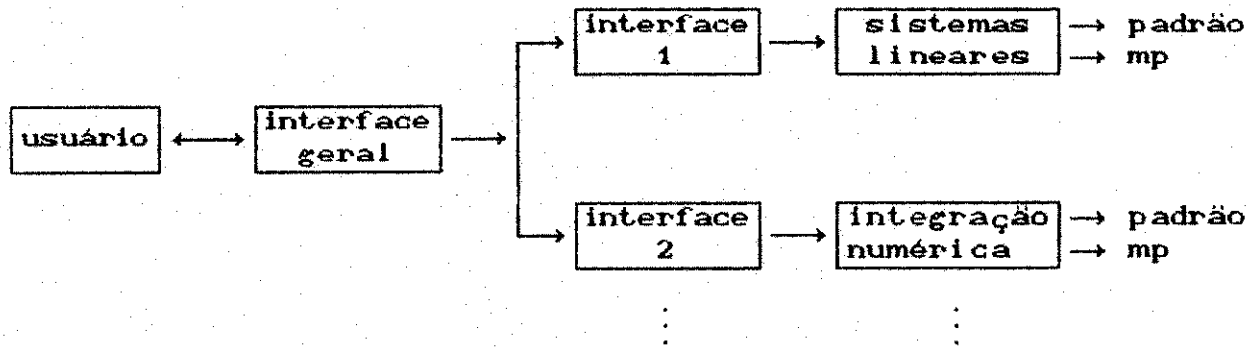
6.3 Escolha dos algoritmos

Um outro fator que contribuiu para um tempo de resposta alto foi o fato de não podermos utilizar aproximação racional, ideal para aproximar funções. O requisito de precisão variável fez com que utilizássemos séries, pois calcular os polinômios aproximadores a cada nova precisão seria muito dispendioso. Dessa forma tivemos que substituir processos finitos (aproximação racional) por processos infinitos (séries), e isto custou tempo.

6.4 Pacote educacional de análise numérica

As rotinas do pacote de múltipla precisão podem ser usadas

como base na construção de um tutor de análise numérica. O usuário aprenderia de uma forma amigável que a precisão influencia os resultados de um cálculo. Poderia escolher algumas áreas de interesse da análise numérica, por exemplo, resolução de sistemas lineares e integração numérica. A solução de problemas dessas duas áreas poderia ser obtida com a implementação de métodos conhecidos utilizando a aritmética padrão e as rotinas do pacote. Com as implementações usando múltipla precisão poderíamos observar a influência da precisão. O diagrama abaixo mostra a estrutura desse tutor



(mp) - aritmética de múltipla precisão

APÊNDICE A

TEMPOS E VALORES DE ALGUMAS FUNCOES DA CALCULADORA

FUNÇAO	VALOR	TEMPO
1 / 7	0.142857142857142857142857142857142857142857142857	0''80
$\sqrt{2}$	1.414213562373095048801688724209698078569671875376	0'26''
10^{100}	10E100	0'1''
50!	30414093201713378043612608166064768844377641568961000000000000000	0'1''
e	2.718281828459045235360287471352662497757247093	2'45''
ln(2)	0.693147180559945309417232121458176568075500134360255	2'14''
sin(0.5)	0.479425538604203000273287935215571388081803367940 6	1'15''
cos(0.5)	0.877582561890372716116281582603829651991645197109 74	1'05''
tan(0.5)	0.546302489843790513255179465780285383297551720179 79	1'29''
arctan(0.5)	0.463647609000806116214256231461214402028537054286 12	1'59''
arcsin(0.5)	0.523598775598298873077107230546583814032861566562 52	2'28''
arccos(0.5)	1.047197551196597746154214461093167628065723133125	2'30''
π	3.141592653589793238462643383279502884197169399375 1	2'20''

obs: foi utilizado um computador PC-AT 80286 com 10 MHz de clock.

a precisão é de 50 dígitos.

APÊNDICE B

DESCRIÇÃO DAS PRINCIPAIS ROTINAS DO PACOTE

ROTINA	DESCRIÇÃO
CALCON	programa principal
ANASIN	faz a análise sintática da expressão lida
CONCAT	concatena os caracteres para formação dos tokens
DESEM1	desempilha as pilhas de operadores e de operandos até encontrar um (realizando as operações contidas na expressão
DESEM2	desempilha as pilhas de operadores e de operandos quando a prioridade do operador de fora da pilha é menor ou igual a prioridade do operador de dentro da pilha ou quando termina a expressão lida
EMPCON	empilha token constante
EMPESP	empilha token especial: (, -, +, *, /, **
EMPNUM	empilha token numérico
IGEXP	iguala o expoente do menor número em múltipla precisão ao expoente do maior
IGUAL1	obtém um número em múltipla precisão da pilha
IGUAL2	coloca um número em múltipla na pilha
IGUAL3	atribui valor a uma variável em múltipla precisão

ROTINA	DESCRIÇÃO
MPACOS	calcula arco-cosseno de um número em múltipla precisão
MPASIN	calcula arco-seno de número em múltipla precisão
MPATAN	calcula arco-tangente de um número em múltipla precisão
MPCIM	converte um número inteiro em múltipla precisão
MPCOMP	compara dois números em múltipla precisão
MPCOS	calcula o cosseno de um número em múltipla precisão
MPCRM	converte um número real em múltipla precisão
MPCSM	converte um número em caracter para múltipla precisão
MPDEC	obtém a parte decimal de um número em múltipla precisão
MPDIV	divide dois números em múltipla precisão
MPEXP	calcula e^x
MPEXP1	calcula x^y
MPEXP2	calcula e
MPEXP3	calcula e^a, onde a é um número em múltipla precisão positivo e menor que 1
MPFAT	calcula o fatorial de número em múltipla precisão

ROTINA	DESCRIÇAO
MPINT	obtém a parte inteira de número em múltipla precisão
MPLOG	calcula o logaritmo decimal de um número em múltipla precisão
MPLN	calcula o logaritmo neperiano de um número em múltipla precisão
MPLN1	calcula o logaritmo neperiano de 10 ou de um número em múltipla precisão < 1
MPMULT	multiplica dois números em múltipla precisão
MPOUT	mostra o resultado no vídeo da expressaó lida
MPPI	calcula o valor de PI
MPREC	calcula o inverso de um número em múltipla precisão
MPRND	arredonda um número em múltipla precisão se o primeiro digito de guarda for ≥ 5
MPRQ	calcula a raiz quadrada de um número em múltipla precisão
MPRQ1	calcula o inverso da raiz quadrada de um numero em múltipla precisão
MPRQ2	calcula o inverso da raiz quadrada de um número em múltipla precisão < 1
MPSET	define um número em múltipla precisão com todos digitos iguais a k

ROTINA	DESCRIÇAO
MPSIN	calcula o seno de um número em múltipla precisão
MPSOMA	soma dois números em múltipla precisão
MPSOM2	soma 1 ao número em múltipla precisão em virtude do arredondamento
MPSUB	subtrai dois números em múltipla precisão
MPTAN	calcula a tangente de um número em múltipla precisão
MPTRUN	trunca um número em múltipla precisão
NORMAL	normaliza um número em múltipla precisão
POSDIG	retorna a posicão do último dígito de um número em múltipla precisão
POSSIG	retorna a posição do dígito menos significativo de um número em múltipla precisão
PRIIN	retorna a prioridade do operador dentro da pilha de operdores
PRIOUT	retorna a prioridade do operador fora da pilha de operadores
TOKEN	retorna o próximo token da expressaö lida
VAL	retorna o valor decimal de um número em múltipla precisão

REFERÊNCIAS BIBLIOGRÁFICAS

[ABRA68] ABRAMOWITZ, M. **Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables**, Dover Publications, Inc, New York, 1968.

[BREN78] BRENT, Richard P. **A Fortran Multiple Precision Arithmetic Package**, ACM Transactions On Mathematical Software, vol. 4, n. 1, Março, 1978, pags. 57-70.

[BRUM63] BRUMFIEL, Charles F. **Arithmetic: Concepts and Skills**, Addison-Wesley Publishing Company, London, 1963.

[CODY80] CODY Jr., William J. & WAITE, William. **Software Manual for Elementary Functions**, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[FORS70] FORSYTHE, George E. **Pitfall in Computation, or Why a Math Book Isn't Enough**, American Mathematical Monthly, 1970, pags. 931-956.

[HAMA78] HAMACHER, V. C., VRANESIC, Z. G. & ZAKI, S. G. **Computer Organization**, McGraw-Hill, Tokyo, 1978.

[HORO84] HOROWITZ, E. & SAHNI, S. **Fundamentos de Estruturas de Dados**, Campus, Rio de Janeiro, 1984.

[KNUT69] KNUTH, Donald E. **The Art of Computer Programming**, vol. 2, Addison-Wesley, Reading, Mass., 1969.

[MANO76] MANO, M. Morris. **Computer Systems Architecture**, Prentice-Hall, Inc., Englewod Cliffs, 1976.

[MARO87] MARON, I. A. & DEMINOWICH, B. P. **Computational Mathematics**, Mir Publishers, Moscow, 1987.

[RALS65] RALSTON, A. **A First Course in Numerical Analysis**, McGraw-Hill, Tokyo, 1965.

[RUGG88] RUGGIERO, Marcia A. G. & LOPES, Vera Lúcia da Rocha **Cálculo Numérico: Aspectos Teóricos e Computacionais**, McGraw-Hill, São Paulo, 1988.

[STER74] STERBENZ, Pat H. **Floating Point Computation** Prentice-Hall, Inc., Englewood Cliffs, N.J., 1974