UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS DA MATEMÁTICA E DA NATUREZA
INSTITUTO DE MATEMÁTICA
NÚCLEO DE COMPUTAÇÃO ELETRÔNICA

RAYMUNDO THEODORO CARVALHO DE OLIVEIRA

ALGORITMO PARA MINIMIZAR ERRO EM INTEGRAÇÃO NUMÉRICA

RIO DE JANEIRO

2006

RAYMUNDO THEODORO CARVALHO DE OLIVEIRA

ALGORITMO PARA MINIMIZAR ERRO EM INTEGRAÇÃO NUMÉRICA

Dissertação apresentada ao Instituto de Matemática da Universidade Federal do Rio de Janeiro como requisito parcial para obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Algoritmos, Métodos Numéricos e Robótica.

Orientador: Professor Doutor Mauro Antonio Rincon

RIO DE JANEIRO

2006

RAYMUNDO THEODORO CARVALHO DE OLIVEIRA

ALGORITMO PARA MINIMIZAR ERRO EM INTEGRAÇÃO NUMÉRICA

Dissertação apresentada ao Núcleo de Computação Eletrônica do Instituto de Matemática da Universidade Federal do Rio de Janeiro, como requisito parcial para obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Algoritmos, Métodos Numéricos e Robótica.

Aprovado em

BANCA EXAMINADORA

_____
Prof. Dr. Mauro Antonio Rincon
Doutor em Matemática - UFRJ
Orientador

_____
Prof. Dr. Carlos Alberto Nunes Cosenza
Doutor em Ciência da Engenharia – DSC/UFRJ

_____
Prof. Dr. José Márcio Machado
Doutor em Engenharia Elétrica – IBILCE/UNESP

_____
Prof. Dr. Adilson Elias Xavier
Doutor em Ciência da Computação - COPPE/UFRJ

RIO DE JANEIRO

2006

RESUMO

É comum, em problemas numéricos, o aparecimento de somatórios de número elevado de elementos. Essas parcelas vão sendo somadas num acumulador, ao qual é adicionado cada novo elemento. Com o acúmulo de parcelas somadas e o crescimento do valor do acumulador, cada nova parcela perde precisão, por ser adicionada a um valor de grandeza muito superior ao seu. Em decorrência, o somatório perde precisão. Esse tipo de somatório aparece, entre outros exemplos, no caso de cálculo numérico de integrais definidas de uma função, pois o valor da integral é o limite do somatório do produto da função por Δx, quando $\Delta x \rightarrow 0$, isto é, quando o número de parcelas tende ao infinito. Na busca de se obter a integral com maior precisão, trata-se de aumentar o número dessas parcelas, cuja soma tende ao valor da integral. Entretanto, se inicialmente o aumento do número de parcelas aumenta a precisão da integral calculada, com o aumento maior dessas parcelas, elas vão perdendo precisão ao serem acumuladas no somatório. O algoritmo proposto supera essa perda de precisão decorrente da soma de parcelas de grandezas muito diferentes, viabilizando o cálculo dos somatórios com precisão excepcionalmente elevada, deixando essa busca da precisão de ser limitada pelos erros decorrentes da soma de parcelas de grandezas muito diferentes. São apresentados exemplos práticos do cálculo de integrais, inclusive integrais duplas cujo cálculo analítico é hoje desconhecido, viabilizando-se sua estimativa com a precisão que se desejar, sendo as limitações somente as dos métodos numéricos aplicados.

Palavras-chave: algoritmo. Ponto flutuante, Somatório, Integrais.

ABSTRACT

In numerical analysis, it is very common to use the result of the addition of many values. As the numbers are being added in an accumulator, the value of the accumulator increases and so each new value is added to a number that is much bigger than itself. When it is necessary to add values that have different orders, one being much bigger than the other, the small one loses precision and so the result also loses its precision. For example, this kind of addition occurs when an integral of a function is being evaluated numerically, as the value of the integral is the limit, when $\Delta x \rightarrow 0$, of the summation of function multiplied by Δx. When we try to improve the calculation of the integral, we increase the number (n) of divisions of the interval of integration and so it is increased the number of parcels to be added. At the beginning, when n increases, the precision also increases; when n increases much more, the precision decreases and the error increases, due to the fact that small numbers are being added to big ones. This work proposes an algorithm to overcome the losing of precision, allowing the increasing the precision by the increasing of n, avoiding the addition of small numbers to big ones. Practical examples are presented, including the calculation, with big precision, of double integrals whose analytical solutions are not known. The limits we face are those of the numerical methods in itself.

Keywords: algorithm. Floating point. Summation. Integral.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A precisão no cálculo é crítica em alguns tipos de problema: cálculo de raízes de polinômios de grau elevado, sistemas lineares mal condicionados, integrais definidas, entre outros. Por outro lado, usando-se computadores, esses cálculos são feitos com números reais representados em ponto flutuante. A própria representação em ponto flutuante já vai implicar em limite para a precisão com que se vai operar, tendo em vista que se necessita representar o número real num número finito de bits, reservados para representá-los.

Dessa forma, erros estarão presentes nos cálculos, podendo levar a resultados inaceitáveis, tal a diferença com os resultados teóricos. Em decorrência, desenvolveram-se algoritmos que buscam minimizar os erros ocasionados por essa perda de precisão.

Nesta dissertação, após serem apresentados exemplos, onde a precisão de cálculo é crítica, e ser analisada em detalhe a utilização do ponto flutuante, é aprofundado o estudo do particular caso das integrais definidas, calculadas numericamente, a partir da divisão do intervalo de integração em n partes.

É conhecido que a precisão do cálculo numérico dessas integrais aumenta com o número de divisões e que, com os sucessivos aumentos de n, a partir de certo ponto a precisão não mais aumenta e, ao contrário, diminui.

Mostra-se que a razão dessa perda de precisão está no erro crescente, com n, do cálculo do somatório que leva ao valor da integral. Quanto maior o número de parcelas, a tendência é a de aumentar o erro decorrente das somas das parcelas desse somatório.

De fato, confrontam-se duas tendências: a do algoritmo, que leva à diminuição do erro, e a do somatório, que leva ao aumento do erro. A primeira tendência é dominante para ***n*** pequeno e a segunda para ***n*** maior.

É proposto um algoritmo para minimizar o erro desse somatório, minimizando-se, em decorrência, o erro do cálculo da integral, que pode, dessa maneira, ser calculada com precisão muito maior do que quando do uso dos algoritmos convencionais.

Diversos exemplos são oferecidos, sendo comparados os resultados com e sem a utilização do algoritmo proposto.

Esse algoritmo pode ser usado sempre que se esteja buscando calcular somatórios de número elevado de parcelas, sendo a integral definida somente uma de suas potenciais aplicações[1].

1 A formatação dessa Dissertação se orienta pelas normas da ABNT sobre informação e documentação, destacando-se: NBR 6023: 2002 Referências e NBR 14724: 2002 Apresentação de trabalhos acadêmicos.

2 A PROPAGAÇÃO DO ERRO

São apresentados, a seguir, alguns exemplos onde um pequeno erro na precisão leva a resultados absurdos, diante do esperado teoricamente. Daí a importância do cuidado com a precisão.

2.1 SÉRIE DE TAYLOR

Seja o cálculo de e^{-4} por desenvolvimento em Série de Taylor.

Tem-se que: $e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + ...$

Assim, $e^{-x} = 1 - x + x^2/2! - x^3/3! + x^4/4! - ...$

Logo, $e^{-4} = 1 - 4 + 16/2! - 64/3! + 256/4! - ...$

O resultado correto, com dez dígitos significativos, é: $e^{-4} \approx 0{,}01831563889$.

Desenvolvendo essa série até o fatorial de 14, tem-se: $e^{-4} \approx$ **0,018970917**. Esse erro é da ordem de 4 %.

Nesse ponto, foi-se ao limite da precisão da HP 28S.

Alterando-se ligeiramente o algoritmo de cálculo, para $e^{-x} = 1/e^x$, e calculando-se e^x , tem-se: $e^4 = 1 + 4 + 16/2! + 64/3! + 256/4! + ...$

Indo até o termo em 14! , tem-se: $e^4 \approx 54{,}597061$, logo $e^{-4} \approx 0{,}018316004$.

Este erro é da ordem de 0,002 %.

Como se vê, usando-se a mesma precisão e com os mesmos números de termos, a alteração do algoritmo melhorou sensivelmente o resultado obtido.

2.2 POLINÔMIO DE WILKINSON

Outro exemplo, onde uma pequena alteração muda sensivelmente o resultado, pode ser obtido pelo chamado Polinômio de Wilkinson (Skeel et all, pág 23).

$P(z) = (z-1)(z-2)(z-3)....(z-20) = z^{20} - 210\ z^{19} + ... + 20!$

Evidentemente, esse polinômio tem 1, 2, ... 19, 20 como suas raízes, todas reais. Esse polinômio é profundamente sensível a mudanças em seus coeficientes. Se alteramos o coeficiente - 210 para $-210 + 10^{-7}$, as raízes se alteram e passamos a ter, além de 10 raízes reais, outras 10 raízes complexo conjugadas.

O gráfico a seguir ilustra essa afirmação mostrando as 10 raízes reais e as 10 complexas.

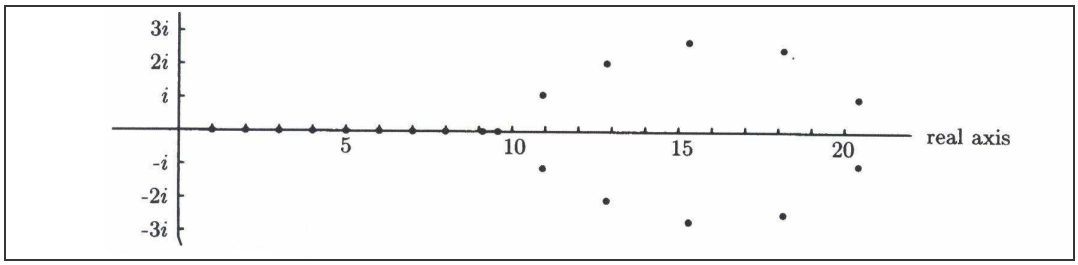

Gráfico 1 – Polinômio de Wilkinson

2.3 FÓRMULA DE BHASKARA

Vejamos, a seguir, um exemplo bem familiar do cálculo das raízes reais de uma equação do segundo grau: $a\ x^2 + b\ x + c = 0$.

As raízes são dadas pela conhecida Fórmula de Bhaskara: $\frac{(-b \pm \sqrt{b^2 - 4ac})}{2a}$.

Tomemos como exemplo a equação $\mathbf{x^2 - 10\ x + 0{,}001 = 0.}$

Sabe-se que uma das importantes razões de perda de precisão em cálculos vem da subtração de elementos da mesma ordem de grandeza. Isso é fácil de ser observado, subtraindo-se dois elementos de grandeza semelhantes e que tenham margem pequena de erro.

Exemplos: (235,7 ± 0,2) – (232,4 ± 0,2) . Ambos os números têm erro relativo da ordem de grandeza de 0,1% . Ao subtrairmos temos: 3,3 ± 0,4 , sendo o erro relativo superior a 10%.

Dessa forma, sempre que possível, deve-se evitar subtração de números de grandeza semelhantes.

Tendo em vista esta preocupação, retornemos ao já citado cálculo das raízes da equação $\mathbf{x^2 - 10\,x + 0{,}001 = 0}$.

Reparemos que $\sqrt{b^2 - 4ac}$ é aproximadamente igual a *b*. Sendo *b* negativo e igual a -10, - *b* será muito próximo de $\sqrt{b^2 - 4ac}$, e a subtração trará perda de precisão.

O que fazer ?

Neste caso, sendo *b* negativo, mantém-se para a primeira raiz a fórmula já vista $\frac{(-b + \sqrt{b^2 - 4ac})}{2a}$ e altera-se a segunda fórmula, multiplicando-se o numerador e o denominador da fórmula $\frac{(-b - \sqrt{b^2 - 4ac})}{2a}$ por $(-b + \sqrt{b^2 - 4ac})$, obtendo-se a fórmula $x_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}}$, evitando-se a subtração de valores próximos.

No caso em que *b* for positivo, faz-se a troca da primeira fórmula para se obter $x_1 = \frac{2c}{-b-\sqrt{b^2-4ac}}$, e mantém-se a segunda $x_2 = \frac{(-b-\sqrt{b^2-4ac})}{2a}$.

Dessa forma, minimiza-se o erro decorrente de *b* ser próximo a $\sqrt{b^2-4ac}$.

Voltemos ao exemplo numérico citado acima.

$x^2 - 10\,x + 0{,}001 = 0$.

Com nove casas decimais tem-se: $x_1 = 9{,}999899999$ e $x_2 = 0{,}000100001$.

Façamos o cálculo com três dígitos. Usando-se a fórmula usual tem-se:

$x_1 = (10+\sqrt{100-0{,}004})/2 = (10+\sqrt{100})/2 = 10{,}0$

$x_2 = (10-\sqrt{100-0{,}004})/2 = (10-\sqrt{100})/2 = 0{,}0$

Alterando-se a segunda fórmula, tem-se:

$x_2 = 0{,}002/(10+\sqrt{100-0{,}004}) = 0{,}0001$

Como se vê, houve sensível melhora na precisão, fruto de alteração do algoritmo.

2.4 MATRIZES MAL CONDICIONADAS

O exemplo a seguir se refere a Sistemas Lineares, com matrizes mal condicionadas.

Seja o sistema abaixo:

$5{,}00\,x_1 + 2{,}00\,x_2 = 10{,}00$

$10{,}00x_1 + 4{,}00\,x_2 = 20{,}00$

Trata-se de um sistema singular, onde o determinante vale zero, sendo indeterminado.

Vamos introduzir uma pequena alteração num dos coeficientes.

$5{,}01\ x_1 + 2{,}00\ x_2 = 10{,}00$

$10{,}00x_1 + 4{,}00\ x_2 = 20{,}00$

Neste caso, o determinante já não vale zero, embora muito pequeno quando comparado aos coeficientes.

O sistema é possível determinado e sua solução é $x_1 = 0$ e $x_2 = 5{,}00$.

Da mesma forma, vamos fazer outra pequena alteração num dos coeficientes.

$5{,}00\ x_1 + 2{,}01\ x_2 = 10{,}00$

$10{,}00x_1 + 4{,}00\ x_2 = 20{,}00$

Neste novo sistema possível determinado a solução é $x_1 = 2{,}00$ e $x_2 = 0{,}00$.

Sendo o sistema mal condicionado, uma pequena alteração, ou um pequeno erro de precisão, leva a resultados completamente diferentes.

2.5 O SINAL DE UMA INTEGRAL DEFINIDA

Finalmente, há ainda um exemplo expressivo onde sabemos que o resultado da integral é necessariamente positivo e chega-se a um resultado negativo, fruto da falta de precisão.

Trata-se de calcular a integral $E_n = \int_0^1 x^n e^{x-1} dx$, onde n = 1,2 ..

Integrando-se por partes, tem-se:

$dv = e^{x-1}dx$, logo $v = e^{x-1}$

$u = x^n$, logo $du = nx^{n-1}dx$

Assim, $\int_0^1 x^n e^{x-1}dx = e^{x-1}x^n]_0^1 - \int_0^1 nx^{n-1}e^{x-1}dx = 1 - nE_{n-1}$.

Logo, $E_n = 1 - nE_{n-1}$, n = 2,3,....

Porém, $E_1 = \int_0^1 x^1 e^{x-1}dx = xe^{x-1}]_0^1 - e^{x-1}]_0^1 = \frac{1}{e}$

Partindo de E_1 com dez dígitos, teremos a seguinte seqüência desenvolvida abaixo.

Tabela 1 – A mudança do sinal da integral

E_1	0,3678794412
E_2	0,2642411176
E_3	0,2072766472
E_4	0,1708934112
E_5	0,1455329440
E_6	0,1268023360
E_7	0,1123836480
E_8	0,100930816
E_9	0,091622656
E_{10}	0,08377344
E_{11}	0,078849216
E_{12}	0,05809408
E_{13}	0,24477696
E_{14}	-2,42687744

Porém, a função $x^n e^{x-1}$ é sempre positiva e sua integral nunca poderia ser negativa.

A razão está na falta de precisão do valor inicial E_1, tomado com dez dígitos, pois as demais contas foram exatas.

A fórmula $E_n = 1 - nE_{n-1}$ vai fazer com que o erro de E_1 se propague aos demais elementos.

Sendo e = 2,71828182845904523536..., o valor de $E_1 = 1/e$ será:

E_1 = 0,367879441171442321595..

Entretanto, tomou-se E_1 = 0,3678794412, com erro $0{,}2855768.10^{-10}$.

No cálculo de E_2, o erro fica multiplicado por 2. No cálculo de E_3 o erro de E_2 fica multiplicado por 3, logo o erro de E_1 fica multiplicado por 2.3=3!.

No cálculo de E_{14} o erro de E_1 fica multiplicado por 14! = 87178291200.

Assim, erro de $E_{14} = 0{,}2855768.\ 10^{-10}\ .\ 8{,}71782912.10^{10} \approx 2{,}4896097$ que é o que torna E_{14} negativo e valendo -2,42687744.

Seu valor é $\mathbf{E_{14} \approx -2{,}42687744 + 2{,}4896097 = 0{,}062732}$....

Vamos escolher outro algoritmo buscando minimizar o erro.

Seja: $E_n = 1 - nE_{n-1}$, logo $E_{n-1} = (1 - E_n)/n$.

Assim, se partirmos de um erro em E_n, esse erro será dividido por n, e não multiplicado, como no algoritmo anterior.

Por outro lado, $E_n = \int_0^1 x^n e^{x-1} dx < \int_0^1 x^n dx = \frac{x^{n+1}}{n+1}\Big]_0^1 = \frac{1}{n+1}$

Dessa forma, quando $n \to \infty$, $E_n \to 0$, pois $E_n < \frac{1}{n+1}$.

Tomando-se n suficientemente grande, tem-se E_n tão pequeno quanto se deseje. Sendo o valor de E_n necessariamente positivo, o erro de se tomar $E_n = \frac{1}{n+1}$ é menor que $\frac{1}{n+1}$.

Seja **n** = 20.

$E_{20} \approx \frac{1}{21} \approx 0{,}04761904762$ $\qquad$ $e_{20} \approx 0{,}04761904762$

$E_{19} = \frac{1 - E_{20}}{20} = 0{,}0476190476$ e seu erro $e_{19} = e_{20} / 20 \approx 0{,}0023809524$

$E_{18} = \frac{1 - E_{19}}{19} = 0{,}05012531328$ $\qquad$ $e_{18} = e_{19} / 19 \approx 0{,}0001253132$

$E_{17} = \frac{1 - E_{18}}{18} = 0{,}052770816$ $\qquad$ $e_{17} = e_{18} / 18 \approx 0{,}00000696184907$

$E_{16} = \frac{1 - E_{17}}{17} = 0{,}0557193637$ $\qquad$ $e_{16} = e_{17} / 17 \approx 0{,}000000409520533$

$E_{15} = \frac{1 - E_{16}}{16} = 0{,}0590175398$ $\qquad$ $e_{15} = e_{16} / 16 \approx 0{,}0000000255950333$

$E_{14} = \frac{1 - E_{15}}{15} = 0{,}06273216402$ $\qquad$ $e_{14} = e_{15} / 15 \approx 0{,}000000001706335556$

Este resultado confere com o visto anteriormente, para o valor de E_{14}.

O objetivo dessas observações iniciais é o de mostrar que a precisão pode ser crítica em determinadas aplicações ou algoritmos.

Quando se trabalha com computadores e eles usam ponto flutuante, o que é a norma, a precisão fica limitada pelo número de dígitos disponíveis para representar os números reais. Em decorrência, a precisão fica ameaçada, podendo levar, como visto, a resultados inesperados.

Daí a importância do desenvolvimento de algoritmos que minimizem a existência e propagação de erros.

3 O PROBLEMA ANALISADO

É muito comum, na resolução numérica de um problema, calcular o somatório de um número muito grande de elementos.

Os exemplos a seguir ilustram alguns casos onde são utilizadas tais rotinas:

- Cálculo da soma dos elementos de um vetor de ordem elevada;
- Estimativa do valor de uma integral, calculada numericamente;
- Cálculo do produto interno de dois vetores de ordem elevada.

O cálculo numérico do valor de uma integral é bem emblemático do somatório suscitado. A precisão desse cálculo tende a aumentar com o número de divisões do intervalo de integração e, conseqüentemente, com o número de parcelas somadas. Cada parcela representa o cálculo da função a ser integrada, nos pontos decorrentes da divisão do intervalo de integração.

Entretanto, a partir de um certo ponto, em vez de aumentar, a precisão do erro diminui com o aumento do número de divisões. Por quê ?

A causa dessa perda de precisão, e a maneira de a superar, é do que trata este trabalho.

Resumidamente, a razão da perda de precisão se deve à maneira como é armazenado um número real no computador: ponto flutuante (floating point).

De fato, algumas propriedades dos números reais não permanecem válidas quando se está representando o número real em ponto flutuante. Propriedades como a comutativa, a distributiva e a associativa exigem precisão infinita nas operações. Isso

não acontece quando se está operando com ponto flutuante. Em decorrência erros se acumulam, levando, muitas vezes, a resultados inesperados.

Desenvolvemos, a seguir, as características das operações com ponto flutuante, mostrando como são armazenados os números e, em decorrência, a perda de propriedades que os números reais possuem.

4 ARITMÉTICA DE PONTO FLUTUANTE

O nome ponto flutuante, ou vírgula flutuante, vem do fato de que um número real pode ter a posição de sua vírgula alterada, desde que corrigida pela potência da base.

Assim, na base 10 (dez), tem-se:

$43{,}15 = 4{,}315 \,.\, 10^{1} = 0{,}4315.10^{2} = 0{,}04315.10^{3} = 431{,}5.10^{-1} = 4315.10^{-2}$ etc...

A forma que tem 1 dígito ≠ zero antes da vírgula é chamada de forma normalizada. No caso visto, a forma normalizada é: $4{,}315.10^{1}$.

Alguns autores consideram normalizada a forma que começa por 0 (zero) e tem, após a vírgula, 1 dígito ≠ zero. No caso seria a forma $0{,}4315.10^{2}$.

O inconveniente dessa forma será apresentado ao ser discutida a representação, em binário, do ponto flutuante.

4.1 PROPRIEDADES DA ARITMÉTICA DE PONTO FLUTUANTE

A aritmética de ponto flutuante é, ainda, pouco dominada por muitos que utilizam ponto flutuante freqüentemente. Segue a expressão de Goldberg (1991, p.1):

> A aritmética de ponto flutuante é considerada um assunto esotérico por muitas pessoas. Isto é bastante surpreendente porque ponto flutuante é de uso muito generalizado em sistemas de computadores. Quase todas as linguagens têm um tipo de dados em ponto flutuante; computadores, dos PC's aos supercomputadores, têm aceleradores de ponto flutuante; a maior parte dos compiladores são chamados a compilar algoritmos que usam ponto flutuante de vez em quando e, virtualmente, todos os sistemas operacionais devem responder a exceções causadas por ponto flutuante, tais como overflow.

De uma maneira geral, as propriedades dos números reais exigem que os números tenham infinita precisão, o conjunto de números reais seja infinito e ilimitado, tanto no que se refere a números arbitrariamente grandes, isto é, de módulo superior a qualquer número proposto, quanto a números arbitrariamente pequenos, isto é, de módulo inferior a qualquer número diferente de zero que seja sugerido. E mais, entre quaisquer dois números reais, por exemplo, entre 3,1 e 3,2, há infinitos números.

Essas características não estão presentes quando se está trabalhando com números em ponto flutuante, caso em que o número de números reais é finito e determinado pelo número de bits reservados para representar os reais.

Em decorrência de se terem finitos bits para se representar um número real e, em conseqüência, havendo um número finito de números reais representáveis, a aritmética de ponto flutuante não comporta algumas propriedades dos números reais.

Por exemplo, a aritmética de ponto flutuante não é associativa, isto é, ***(a+b)+c*** não é necessariamente igual a ***a+(b+c)***, nem ***(a.b).c*** é obrigatoriamente igual a ***a.(b.c)***. Da mesma forma, a aritmética de ponto flutuante não é distributiva, isto é, ***a.(b+c)*** não é necessariamente igual a ***a.b + a.c***.

O fundamental é lembrar que a ordem em que as operações são realizadas pode influir no resultado.

Vejamos alguns exemplos, na base dez, trabalhando-se com três dígitos.

Propriedade Associativa

A = 143 B = 18,4 C = 13,4

$(A + B) + C = (1{,}43 \cdot 10^2 + 1{,}84 \cdot 10^1) + 1{,}34 \cdot 10^1 = 1{,}61 \cdot 10^2 + 1{,}34 \cdot 10^1 =$

$= 1{,}74 \cdot 10^2$

$A + (B + C) = 1{,}43 \cdot 10^2 + (1{,}84 \cdot 10^1 + 1{,}34 \cdot 10^1) = 1{,}43 \cdot 10^2 + 3{,}18 \cdot 10^1 =$

$= 1{,}75 \cdot 10^2$

Neste caso, $(A + B) + C \neq A + (B + C)$, logo, não é associativa.

$(A \times B) \times C = (1{,}43.10^2 \times 1{,}84.10^1) \times 1{,}34.10^1 = 2{,}63.10^3 \times 1{,}34.10^1 = 3{,}52.10^4$

$A \times (B \times C) = 1{,}43.10^2 \times (1{,}84.10^1 \times 1{,}34.10^1) = 1{,}43.10^2 \times 2{,}47.10^2 = 3{,}53.10^4$

$(A \times B) \times C \neq A \times (B \times C)$, logo não é associativa.

$A \times (B + C) = 1{,}43.10^2 \times (1{,}84.10^1 + 1{,}34.10^1) = 1{,}43.10^2 \times 3{,}18.10^1 = 4{,}55.10^3$

$A \times B + A \times C = 1{,}43.10^2 \times 1{,}84.10^1 + 1{,}43.10^2 \times 1{,}34.10^1 =$

$= 2{,}63 \cdot 10^3 + 1{,}92 \cdot 10^3 = 4{,}55.10^3$

Isto é, $A \times (B + C) = A \times B + A \times C$.

A propriedade distributiva valeu neste caso.

Vejamos agora com A = 43,2 B = 92,4 C = 13,2

$A \times (B + C) = 4{,}32.10^1 \times (9{,}24.10^1 + 1{,}32.10^1) = 4{,}32.10^1 \times 1{,}06.10^2 = 4{,}58.10^3$

$A \times B + A \times C = 4{,}32.10^1 \times 9{,}24.10^1 + 4{,}32.10^1 \times 1{,}32.10^1 = 3{,}99.10^2 + 5{,}70.10^2 =$

$= 4{,}56 \cdot 10^3$

$A \times (B + C) \neq A \times B + A \times C$, logo, a propriedade distributiva não é válida.

O exemplo a seguir, trabalhando com três dígitos, mostra, mais uma vez, que a ordem pode alterar o resultado:

$0{,}132 + (572 - 572) = 0{,}132$

$(0{,}132 + 572) - 572 = 0$

Como conseqüência, se verá que, no caso da soma de uma parcela a outra de ordem muito superior, a parcela menor, em módulo, perde precisão, alterando a precisão da nova soma. No exemplo recém visto, **0,132 + 572** continua a valer **572**. Assim, **572** não é alterado ao ser somado a **0,132**, considerando-se operação com somente três dígitos.

No caso em que a diferença entre as parcelas não seja tão expressiva, pode haver, ainda assim, perda de precisão: **1,32 + 572 = 573,** perdendo-se a parte fracionária **0,32**.

4.1.1 Perda de precisão nas operações de soma

Os números reais são, como vimos, representados num número finito de dígitos. Quando somamos dois números, é comum ter-se um resultado que não cabe mais no número de dígitos disponíveis.

Se o resultado exigir mais dígitos que os disponíveis, haverá arredondamento do resultado, com perda de precisão.

Vejamos, primeiramente, em decimal, como esses fatos ocorrem, supondo que se está operando com três dígitos significativos.

São apresentados, a seguir, alguns números e sua forma normalizada:

$235 = 2{,}35 \cdot 10^2$; $740 = 7{,}40 \cdot 10^2$; $45{,}8 = 4{,}58 \cdot 10^1$; $9{,}12 = 9{,}12 \cdot 10^0$;

$8{,}20 = 8{,}20 \cdot 10^0$; $0{,}187 = 1{,}87 \cdot 10^{-1}$; $0{,}0754 = 7{,}54 \cdot 10^{-2}$;

$0{,}00253 = 2{,}53 \cdot 10^{-3}$ etc.

Acompanhemos a operação de soma entre dois números reais, normalizados, com número limitado de dígitos, no caso, três dígitos.

$$\mathbf{S = 2{,}43 \cdot 10^{2} + 1{,}32 \cdot 10^{1}}$$

Ao fazermos uma operação de soma, tem-se que desnormalizar o menor número, de modo que ele passe a ter o expoente do maior número, antes de ser feita a operação de soma.

$$\mathbf{S = 2{,}43 \cdot 10^{2} + 0{,}132 \cdot 10^{2} = 2{,}562 \cdot 10^{2} = 2{,}56 \cdot 10^{2}}$$

Como se viu, houve perda de precisão no menor número. Somar $\mathbf{1{,}32 \cdot 10^{1}}$ ou somar $\mathbf{1{,}3 \cdot 10^{1}}$ levaria ao mesmo resultado.

Isso também é válido quando se opera com binários, desenvolvido no Apêndice E.

4.1.2 Somatório de grande número de parcelas

Sabendo-se que na aritmética de ponto flutuante não são válidas as propriedades distributiva e associativa, vejamos seu reflexo no caso do cálculo do somatório de número elevado de parcelas.

Para simplificar, em nossa análise do somatório, vamos considerar que se estão somando valores positivos. No cálculo dessas somas, as parcelas vão sendo acumuladas, em um campo, em número crescente. Esse campo, onde estão sendo acumuladas essas parcelas, tende a crescer, com a adição dessas novas parcelas. Quando se tem acumulado um número expressivo de parcelas já somadas, a nova parcela a ser adicionada tende a ser muito inferior ao somatório, ao qual ela está sendo adicionada. Assim, tem-se a soma de um valor a outro de ordem muito superior. Quando isso acontece, tem-se perda de precisão desse valor, como foi visto no exemplo citado, onde **1,32 + 572 = 573** , quando se perdeu a parte fracionária **0,32**.

Além disso, tendo crescido a soma já acumulada, para cada novo valor adicionado, a perda de precisão é crescente.

O objetivo central desta monografia é exatamente o de desenvolver um algoritmo que minimize essa perda de precisão do somatório.

Em primeiro lugar, apresenta-se como se dá o armazenamento de números reais em ponto flutuante. Em seguida, será mostrada a perda de precisão ocorrida quando se somam valores de ordem de grandeza diferentes, acompanhada de exemplos emblemáticos, com ênfase no cálculo numérico de integrais definidas.

Finalmente, o algoritmo sugerido é exposto, comparando-se os resultados com os obtidos pelos métodos tradicionais.

4.2 NÚMEROS REAIS EM PONTO FLUTUANTE

Vejamos, num breve resumo, como é armazenado o número real, contrapondo-o à maneira como é armazenado o número inteiro. Neste caso, números inteiros, dados **n** bits consecutivos, pode-se armazenar qualquer número de **000...00** a **111...111**, cada um contendo **n** dígitos, num total de $\mathbf{2^n}$ possibilidades para representação exata de números inteiros na base binária. Assim, podem ser armazenados números inteiros não negativos, de **0** (zero) a $\mathbf{2^n - 1}$.

Havendo **4** dígitos, os números variariam de **0** (zero) a $\mathbf{15 = 2^4 - 1}$, num total de **16** números inteiros não negativos.

Vamos tomar **32** bits e ver as possibilidades. Haveria $\mathbf{2^{32}}$ números inteiros, de **0** (zero) a $\mathbf{2^{32} - 1 = 4.294.967.295}$. Isso nos permite representar números inteiros não negativos até pouco mais de $\mathbf{4.10^9}$ **(4 bilhões).** Na representação de números negativos, há

várias possibilidades, entre as quais a de se considerar **1** bit para o sinal, sobrando-se **31** bits para complementar a representação dos números.

Entretanto, na representação de números maiores que $\mathbf{4.10^9}$ (**4** bilhões), números fracionários ou números com expoentes negativos, não teríamos como fazê-lo, com a representação sugerida. Daí a idéia do ponto flutuante (floating point).

Como já afirmado, com ponto flutuante, cada número real pode ter diferentes representações, alterando-se a posição da vírgula e corrigindo-se com a potência da base. Assim, a vírgula (o ponto) flutua.

Recordemos, inicialmente na base dez, onde um número qualquer, por exemplo, o número **245**, pode ter diferentes representações.

Assim, $\mathbf{245 = 24{,}5\ .10^1 = 2{,}45.10^2 = 0{,}245.10^3 = 2450.10^{-1} = 24500.10^{-2}}$ etc...

Da mesma forma, $\mathbf{0{,}312 = 3{,}12.10^{-1} = 0{,}0312.10^1}$ ...

Como se viu, a vírgula (o ponto) pode flutuar, sendo sua posição corrigida pela potência da base, no caso, base **10** (dez).

Excluído o real **0** (zero), a ser visto adiante, das diferentes formas possíveis, uma delas passou a ser chamada forma normalizada, que é a que tem um dígito diferente de zero antes da vírgula. Nos casos citados teríamos: $\mathbf{2{,}45.10^2}$ e $\mathbf{3{,}12.10^{-1}}$.

O que foi exemplificado para a base **10** (dez) valeria para outra base. No caso dos computadores binários, a base **2** (dois).

Assim o número **3,5** , que em binário se representa por **11,1** , seria $\mathbf{1{,}11.2^1}$ na base dois, normalizado. Não há necessidade de se escrever a base **2**, por ser ela assumida pelo computador. Além disso, o próprio expoente será representado na base **2**.

O número **- 4,25**, representado em binário por **- 100,01**, na forma normalizada seria representado por - 1,0001.22; lembrando que o expoente **2**, também deverá ser escrito na base dois, de uma forma a ser apresentada adiante.

Dessa maneira, para se representar um número real, necessita-se de **1** (um) bit para o sinal do número (**0 ou 1**), alguns bits para se representar o expoente (um número inteiro), e alguns bits para se representar a mantissa, isto é, o corpo do número, excluído o sinal e o expoente.

No exemplo dado: **- 4,25 = - 1,0001 . 2^2**, tem-se: sinal negativo (bit **1**), expoente (**2**), mantissa (**1,0001**). Pode-se observar que, à exclusão do **0** (zero), todos os números reais, na base dois em forma normalizada, começarão por **1,** (um vírgula). Assim, não há necessidade de representar o **1,** (um vírgula), pois isto será assumido pelo computador, em seu próprio projeto.

Tem-se, portanto, que representar o sinal com um bit (**0** para o positivo e **1** para o negativo, por convenção), alguns bits para o expoente (inteiro positivo ou negativo) e alguns bits para a mantissa, sem a necessidade de se representar o **1,** (um vírgula).

Em 1985, chegou-se a um acordo, definindo-se padrões para a representação de números reais, bem como para resultados computacionais, em ponto flutuante, com precisão simples (single) e dupla (double), transformado em resolução (IEEE 754: Standard for Binary Floating-Point Arithmetic - http://grouper.ieee.org/groups/754)

Por essa resolução, para precisão simples (single), reservou-se **1** (um) bit para o sinal, **8** bits para o expoente e **23** bits para a mantissa. Para precisão dupla (double), a mesma resolução reservou **1** (um) bit para o sinal, **11** para o expoente e **52** para a mantissa.

No interior dos 32 bits da precisão simples, vejamos como são representados os expoentes, todos inteiros, para que seja garantida a possibilidade de expoente negativo.

No caso da precisão simples, dispondo de **8** bits, os expoentes podem variar de **00000000** a **11111111**, isto é, de **0** a **255**. Tomou-se o valor **01111111** (**127**) como sendo o expoente **0** (zero). O número **01111110** (**126**) passou a ser o expoente **-1** (menos um), enquanto **10000000** (**128**) o expoente **+1** (mais um). Dessa forma, o valor verdadeiro do expoente será o número escrito em binário menos **127** (bias).

Reservaram-se os expoentes **00000000** e **11111111** para usos especiais a serem vistos adiante. Dessa forma, o menor expoente **00000001** passou a representar o expoente: **1 – 127 = = – 126**. O expoente **11111110** a ser o maior expoente: **254 – 127 = +127**. Os expoentes variam, portanto, de **– 126 a +127**.

Como se observa, a gama de variação de números é muito superior ao que se conseguiria na representação de números inteiros, onde se ia de **0** a **4.294.967.295**. Em ponto flutuante, os números normalizados positivos variam, em ordem de grandeza, de $\mathbf{2^{-126}}$ **(** $\mathbf{1{,}17.10^{-38}}$ **)** a $\mathbf{2^{127}}$ **(** $\mathbf{1\ ,7\ .10^{38}}$ **)**, além dos números negativos, com igual ordem de variação. Pode-se usar números muito elevados ($\mathbf{10^{38}}$) e muito próximos a zero ($\mathbf{10^{-38}}$), além do zero.

As mantissas são, todas elas, normalizadas, sem necessidade de se representar o **1,** (um vírgula), como dito acima.

Vejamos alguns exemplos:

8,5 → 1000,1 → $\mathbf{1{,}0001.2^3}$ **→ 0 10000010 00010000000000000000000**

-10,25→-1010,01→ $\mathbf{-1{,}01001.2^3}$ **→ 1 10000010 01001000000000000000000**

Dessa forma, o número real é representado por **24** bits: os **23** bits expostos e um bit escondido, o que antecede a vírgula.

O expoente **00000000** tem uma função especial: ele é utilizado para se obter valores mais próximos do zero do que seria conseguido com os números normalizados, vistos até agora.

Para números normalizados, o menor número positivo seria $\mathbf{1{,}0 \cdot 2^{-126} \approx 1{,}17.10^{-38}}$.

Usando o expoente **00000000** o número deixa de ser normalizado e começa por **0,** (zero vírgula) e não mais por **1,** (um vírgula), embora o expoente permaneça sendo **-126**.

Assim, pode-se obter o valor $\mathbf{0{,}00000000000000000000001 \cdot 2^{-126} = 2^{-23}.2^{-126} = 2^{-149}}$. Este processo recebe o nome de underflow gradual, permitindo que sejam representados números muito mais próximos de zero do que seria o caso com números normalizados.

O número real **0** (zero) pode ter duas representações:

0 00000000 00000000000000000000000 → + zero

1 00000000 00000000000000000000000 → – zero

Embora com representações diferentes, eles devem ser considerados iguais em uma comparação direta.

O expoente **11111111** é usado para representar **+** ∞ **, –** ∞ ou indeterminado e servindo, também, para indicar que uma representação binária não é um número válido. (NAN – not a number). Vejam-se detalhes em: Hollasch, Steve. IEEE Standard 754 Floating Point Numbers-2005 - http://stevehollasch.com/cgindex/coding/ieeefloat.html.

É fundamental lembrar, como dito acima, que no caso da representação em dupla precisão, double, reservam-se **11** bits para o expoente, **52** bits para a mantissa, além do bit que marca o sinal do número, num total de **64** bits. Dessa forma, o zero do expoente será: **01111111111 = 1023** (bias).

Da mesma maneira que em precisão simples, o expoente **1022 (01111111110)** representará **–1** e **1024 (10000000000)** representará **+1**. Assim, **00000000001 (1)** representa o expoente **–1022 (1 – 1023)** ao passo que **11111111110 (2046)** representa o expoente **+ 1023 (2046 – 1023)**.

4.2.1 Arredondamento ou truncamento?

O fundamental do que foi visto é a consciência de que há um número limitado de bits para serem usados na representação dos números reais. Dessa forma, em geral, ao se tentar representar números reais, fazem-se arredondamentos, por não se terem infinitos dígitos.

Assim, das operações matemáticas, os resultados serão, em geral, aproximados, por não caberem no limitado número de bits reservados.

Alguns exemplos ajudam a esclarecer. O número decimal **0,8** , oito décimos, ao ser escrito em binário, se torna uma dízima e deve ser escrito:

0,1100110011001100110011001100110011001100110011 ...

Ao ser representado como real com precisão simples, o número 0,8 se torna:

1,10011001100110011001101 . 2^{-1}. Claramente, ele já não vale **0,8**, sendo uma aproximação.

Isso também acontece na base dez, quando se tenta escrever o número **1/3**, que vale **0,333333...** . No caso de um número limitado de dígitos, o resultado será forçosamente diferente de **1/3**.

No caso acima, **0,8** , ao final dos **23** bits da mantissa tinha-se: **..1100**, em vermelho representado o **23º** bit, e, em seguida, fora da precisão dos **23** bits, continuaria **110011001100 ...**

O que acontece ao ser o número limitado aos **23** bits: arredondamento ou truncamento? Tudo se faz da mesma maneira que na base dez: se o que sobra é maior que **0,5** aproxima-se o dígito anterior para cima; se menor que **0,5**, trunca-se, mantendo-se o dígito anterior.

Exemplos: **4,82 → 5** **4,52 → 5** **4,48 → 4**

No exemplo do número **0,8**, acima citado, após a 23^0 bit da mantissa, viria **1100...**, logo o restante é maior que **0,5**, daí haver aproximação para cima:

$$1,10011001100110011001101 \cdot 2^{-1}$$

A diferença entre o valor representado de **0,8** (arredondado para cima) e o verdadeiro valor **0,8** pode ser calculada, dando:

$$\varepsilon = (1,0 - 0,11001100...).2^{-23}.2^{-1} = (1,0 - 0,8).2^{-24} = 0,2 \cdot 2^{-24} = 1,192 \cdot 10^{-8}$$

Vejamos o número **0,7**, que em binário é representado por:

$$0,10110011001100... = 1,01100110011001100110011.2^{-1}$$

Depois do vigésimo terceiro bit, em vermelho, foi abandonado **0,0011...**, que é menor que **0,5**, isto é, metade da última unidade que entrou. Assim, essa parte é abandonada, ficando somente:

0,7 → 1,01100110011001100110011. 2^{-1} que é representado em precisão simples da maneira a seguir: **0 01111110 01100110011001100110011**

Este número é ligeiramente inferior a **0,7**, por ter sido abandonada uma parte dele.

A diferença entre o valor representado de **0,7** (arredondado para baixo) e o verdadeiro valor **0,7** pode ser calculada, dando:

$$\varepsilon = (1,0 - 1,00110011...).2^{-23}.2^{-1} = (-0,2).2^{-24} = -0,2 \cdot 2^{-24} = -1,192 \cdot 10^{-8}$$

Da mesma forma, o número **0,6 = 0,10011001100110011... =**

= 1,00110011001100110011001100110011... 2^{-1} . Onde o **1** é o **23º** bit.

Logo, limitando-se a mantissa aos **23** bits, tem-se:

$0,6 \approx 1,00110011001100110011010 \cdot 2^{-1}$, que é ligeiramente maior que **0,6**.

A diferença entre **0,6** e o **0,6** representado é igual a:

$$\varepsilon = 10,0000....2^{-23}.2^{-1} - 0,1001100110011...2^{-23}2^{-1} = (2 - 0,6).2^{-24} = 2,3841858.10^{-8}$$

Esses três valores do erro podem ser observados da maneira abaixo, usando um programa em Delphi.

Criam-se três variáveis reais, A, B e C, de precisão simples (single). São atribuídas às três variáveis os valores **A : = 0.7, B : = 0.8 e C := 0.6**. Imprimem-se as três variáveis, obtendo-se **A = 0.6999999881, B = 0.8000000119 e C = 0,6000000238** . Assim, A era um pouco abaixo de **0,7**, B um pouco acima de **0,8** e C um pouco acima de **0,6**, sendo as diferenças exatamente as previstas acima.

A →0.6999999881 – 0,7 = – 0,0000000119

B →0.8000000119 – 0,8 = 0,0000000119

C →0,6000000238 – 0,6 = 0,0000000238

4.2.2 Sobrando exatamente metade da unidade

E se for o caso de se ter exatamente **4,5**, sobrando metade da unidade onde termina o número? Faz-se, neste caso, a aproximação para número par mais próximo. Assim,

2,5 → 2 **1,5 → 2** **3,5 → 4** **4,5 → 4** .

Em binário: **5,5 → 101,1 → 110 (6)** **6,5 → 110,5 → 110 (6)** .

Podemos observar isso em binário tomando-se os seguintes exemplos: **A = 0,5, B = 0,8 , C = 0,7 e D = 0,6.**

Vejamos a representação de cada um em ponto flutuante, precisão simples.

0,5 → 1,00000000000000000000000 . 2^{-1}

0,8 → 1,10011001100110011001101 . 2^{-1}

0,7 → 1,01100110011001100110011 . 2^{-1}

0,6 → 1,00110011001100110011010 . 2^{-1}

Procuremos, para cada um desses números, a menor potência de dois que ainda altera esses números, ao ser a eles somados. Isto é, qual o maior **n** inteiro, tal que 2^{-n} ainda altera um desses números ao ser a ele somado.

Seja para **A = 0,5**.

$0,5 + 2^{-n} > 0,5$, sendo $0,5 + 2^{-(n+1)} = 0,5$

0,5 → 1,00000000000000000000000 . 2^{-1} (o bit em vermelho é o **23^{o}** bit da mantissa).

Se somarmos a ele $\varepsilon = 0{,}00000000000000000000001 \cdot 2^{-1} = 2^{-24} \cdot 2^{-1} = 2^{-25}$, a soma $0{,}5 + \varepsilon = 1{,}00000000000000000000001 \cdot 2^{-1}$.

Ao ser arredondada para **23** bits, haveria simplesmente perda do **1**, para arredondar para o par mais próximo. Assim, $0{,}5 + 2^{-25} = 0{,}5$, não alterando o valor de **0,5**. O menor que alteraria seria 2^{-24}, por atingir diretamente o **23º** bit do **0,5**. O mesmo ocorre com **D = 0,6** .

No caso de **B = 0,8** tem-se: **0,8 → 1,10011001100110011001101 . 2^{-1}** .

Se somarmos a ele $\varepsilon = 0{,}00000000000000000000001 \cdot 2^{-1} = 2^{-24} \cdot 2^{-1} = 2^{-25}$, a soma $0{,}8 + \varepsilon = 1{,}10011001100110011001110 \cdot 2^{-1} > 0{,}8$, alterando seu valor, pelo fato de a aproximação se dar para o par mais próximo. O mesmo ocorre com **C = 0,7** .

No APÊNDICE A, é apresentado um programa onde esse resultado foi observado.

Na tabela a seguir, apresenta-se, para cada valor de A, a maior potência de **2**(dois) que não altera seu valor ao ser a ele somado (**H**) e a menor potência de **2**(dois) que altera seu valor ao ser a ele somado (**H1**).

Tabela 2 - Maior potência de 2 que não altera (H) e Menor que altera (H1)

A	1,0	0,9	0,8	0,7	0,6	0,5	0,4
H	$5{,}960.10^{-8}$	$2{,}980.10^{-8}$	$1{,}490.10^{-8}$	$1{,}490.10^{-8}$	$2{,}980.10^{-8}$	$2{,}980.10^{-8}$	$7{,}451.10^{-9}$
H1	$1{,}192.10^{-7}$	$5{,}960.10^{-8}$	$2{,}980.10^{-8}$	$2{,}980.10^{-8}$	$5{,}960.10^{-8}$	$5{,}960.10^{-8}$	$1{,}490.10^{-8}$

Lembra-se que:

$7{,}451.10^{-9} \approx 2^{-27}$, $1{,}490 .10^{-8} \approx 2^{-26}$, $2{,}980 . 10^{-8} \approx 2^{-25}$, $5{,}960 .10^{-8} \approx 2^{-24}$

É importante ressaltar que esta maneira de operar é a default, podendo ser utilizadas maneiras alternativas de aproximação, se devidamente explicitadas por opções disponíveis em alguns chips. Ainda Goldberg (1991, p. 36):

> O IEEE Standard tem diversos flags e modos de operação. Como discutido acima, há um flag de status para cada um dos cinco casos de exceção: underflow, overflow, divisão por zero, operação inválida ou inexata. Há quatro modos de arredondamento: arredondar para o mais próximo, arredondar em direção ao +∞, arredondar em direção ao zero e arredondar em direção a -∞. É fortemente recomendado que haja um bit que habilite um modo para cada uma das cinco exceções.

4.3 IEEE-754: UM ACORDO

Até 1985, cada fabricante de computador tinha seu próprio sistema de ponto flutuante, definindo o número de bits do expoente e da mantissa, o sinal do expoente e as regras de arredondamento.

Em decorrência, rodar-se um programa em diferentes computadores podia levar a resultados diferentes.

Em 1985, chegou-se a um acordo sobre a padronização do ponto flutuante: a norma IEEE-754.

William Kahan (1989) apresenta um histórico da própria construção dessa norma IEEE-754, mostrando as dificuldades na busca da padronização, pois os diversos fornecedores já tinham suas máquinas e gostariam de impor suas soluções. O caso particular do underflow gradual é muito interessante por destacar as resistências que duraram anos.

A leitura da própria norma IEEE-754, bem como a apresentação feita no artigo da Intel (The art of assembly language programming) ajudam a superar freqüentes dúvidas

quanto à normalização, o bit escondido (hidden bit), os falsos números (NAN – not a number), a representação de $+\infty$, $-\infty$ e indeterminação.

Da mesma forma, Steve Hollasch (2005), em IEEE Standard 754 Floating Point Numbers, e Raymundo de Oliveira (2005), em Noções de ponto flutuante, têm sido uma fonte de esclarecimento da norma, numa linguagem didática.

Alguns textos ajudam a esclarecer pontos polêmicos, muitos dos quais custaram a ser equacionados. Cito os seguintes trabalhos, detalhados na seção Referências:

- BUSH, Bruce M. (1996). The perils of floating point.
- DAWSON, Bruce. Comparing floating point numbers.
- FUNDAMENTALS OF COMPUTER SCIENCE II. IEEE Floating point representation of real numbers.
- GOLDBERG, David (1991). What every computer scientist should know about floating point arithmetic.
- HOLLASCH, Steve (2005). IEEE Standard 754 Floating point numbers.
- IEEE 754 Standard for binary floating point arithmetic.
- INTEL (2000). Itanium processor floating point software assistance and floating point exception handling.
- INTEL. The art of assembly language programming.
- JUFFA, Norbert; BEEBE, Nelson H. F. . Bibliography of material on floating point arithmetic.
- KAHAM, W. (1997). Lecture notes on the status of IEEE Standard 754 for binary floating point arithmetic.
- KAHAN, William (1989). IEEE Standard 754 for binary floating point arithmetic.

- KHATIB, Jamil. Introduction to floating point calculations and IEEE 754 Standard.
- OLIVEIRA, Raymundo. Noções de ponto flutuante.
- PITTSBURGH Supercomputing Center. The IEEE Standard for floating point arithmetic.
- PYTHON TUTORIAL. Floating Point Arithmetic: issues and limitations.
- WIKIPEDIA, the free encyclopedia. Floating point.

4.4 UM COMPUTADOR SIMULADO

Para tornar mais claro o aparecimento dos erros de aproximação, vamos criar um hipotético computador que tenha **1** (um) dígito para o sinal, **4** dígitos para o expoente e **7** dígitos para a mantissa (excluído o **1,**) , num total de **12** dígitos, no lugar dos **32** dígitos do número com precisão simples. Os expoentes **0000** e **1111** tem funções especiais já vistas, sendo **0111 = 7,** o zero do expoente.

O menor expoente será menos **6**, representado por **0001 = 1 (1-7 = -6)** e o maior expoente será **+7** , representado por **1110 = 14 (14 – 7 = 7)**.

Vamos fazer a soma **35,5 + 7,25** .

35,5 = 100011,10 = 1,0001110 . 2^5

7,25 = 111,01000 = 1,1101000 . 2^2 = 0,0011101000 . 2^5

1,0001110 . 2^5 + 0,0011101000 . 2^5 = 1,0101011 .2^5 = 101010,11 = 42,75 (exato)

Vejamos, em seguida, o cálculo de **67,5 + 1,75**

67,5 = 1000011,1 = 1,0000111 . 2^6

$1{,}75 = 1{,}1100000 = 1{,}1100000 \cdot 2^{0} = 0{,}0000011100000 \cdot 2^{6}$

$1{,}0000111 \cdot 2^{6} + 0{,}0000011100000 \cdot 2^{6} = 1{,}00010101 \cdot 2^{6} \approx 1{,}0001010 \cdot 2^{6} = 69{,}0$

Houve perda de precisão, pois o resultado seria **69,25**.

Um caso mais grave se veria com a soma **124 + 0,1875** .

$124 = 1111100{,}0 = 1{,}1111000 \cdot 2^{6}$

$0{,}1875 = 0{,}00111000 = 1{,}1100000 \cdot 2^{-3} = 0{,}00000000111 \cdot 2^{6}$

$1{,}1111000 \cdot 2^{6} + 0{,}00000000111 \cdot 2^{6} = 1{,}11110000111 \cdot 2^{6} \approx 1{,}1111000 \cdot 2^{6} = 124$

O número **0,1875** não teve qualquer efeito ao ser somado a **124**.

O que se está ressaltando é que, ao se somar a um número outro de grandeza muito inferior, este segundo perde precisão, podendo mesmo se anular, como indicado acima.

O que é válido para este computador simulado, vale para qualquer outro, desde que consideradas as grandezas envolvidas. Claro que a perda de precisão somente aparecerá para diferenças muito mais expressivas, se houver maior precisão de cálculo.

5 O SOMATÓRIO NO CÁLCULO DA INTEGRAL

Como já afirmado, é muito comum ter-se que calcular um somatório com elevado número de parcelas. Uma rotina típica para cálculo do somatório dos **n** elementos de um vetor **Y** é apresentado a seguir.

Soma := 0.0;

For i := 1 to n do

Soma := Soma + Y[i];

Vamos admitir, por simplicidade, que se trata de somar números positivos, da mesma ordem de grandeza, isto é, não havendo grandes diferenças entre esses números.

No início do loop, a Soma vale zero. Em seguida, a Soma começa a acumular valores, ou seja: **Y[1]+Y[2]+Y[3]+Y[4]+...+Y[K]**

A esta altura, já se está pretendendo que o campo **Soma** valha a soma dos **K**$^{\textbf{ésimos}}$ primeiros valores. Observemos que, neste ponto, para **K** elevado, tem-se o valor do campo **Soma**, já elevado, que vai ser somado ao próximo valor de **Y**, isto é:

Soma =Soma + Y[K+1];

Trata-se, exatamente, do que foi colocado anteriormente: está sendo feita a soma de um valor elevado (**Soma**) com outro valor de ordem muito inferior (**Y[K+1]**). A tendência é a de haver perda de precisão do valor **Y[K+1]** e, em conseqüência, perda de precisão da soma total, isto é, do valor calculado para o campo **Soma**.

Sendo a ordem de grandeza de A muito maior que a de B, ao se somar A + B, o valor de B é desnormalizado, sendo aumentado seu expoente para que fique igual ao de A.

Com isso, ao serem somados, perdem-se os últimos bits de B, tendo em vista que ocorre arredondamento porque nem todos os bits da soma caberiam no resultado, isto é, na mantissa, cujo número de bits é limitado.

Para mostrar a perda de precisão de uma soma, em condições como a citada acima, vamos tomar como exemplo o cálculo numérico de uma integral, onde o somatório aparece.

Pelo **Método $1/3$ de Simpson**, para cálculo aproximado de integrais definidas, chega-se à fórmula (BURDEN; FAIRES, 2003; CURTIS; WHEATLEY, 1989) a seguir:

$$\int_a^b Y(x)dx \approx \frac{h}{3}(Y[a] + 4Y[a+h] + 2Y[a+2h] + ... + 2Y[a+(n-2)h] + 4Y[a+(n-1)h] + Y[b])$$

Onde **h = (b-a)/n** , sendo **n**, par, o número de divisões do intervalo de integração **[a,b].**

Observe-se que, tirando o primeiro e o último valores de **Y**, os **Y** relativos a índices ímpares são multiplicados por **4** e os relativos a índices pares são multiplicados por **2**.

Na programação (ver listagem no Apêndice B), foram construídas duas somas: **Soma4** e **Soma2**, onde em **Soma4** estão sendo somados os **Y's** com índices ímpares e em **Soma2**, os **Y's** com índices pares, acrescida da metade de **Y[n]**.

Lembremo-nos que, conforme se aumente o número **n** de divisões, ambas as somas terão elevado número de parcelas a serem somadas: **Soma4** e **Soma2** terão, cada uma, **n/2** parcelas.

O valor da integral, calculada numericamente, será:

I ≈ (Y[0] + 4.0 * Soma4 + 2.0 * Soma2) * h / 3.0

Algumas observações precisam ser feitas.

Em primeiro lugar, no Método de Simpson, também chamado de Método das Parábolas, divide-se o intervalo **[a,b]**, de integração, num número **n** de partes iguais, sendo **n**, obrigatoriamente, um número par, obtendo-se os pontos **(x_0, x_1, ..., x_i, ..., x_{n-1}, x_n)**, onde **$a = x_0$** e **$b = x_n$**. Forma-se uma tabela com **n+1** pontos, **[x_i , $Y(x_i)$], i** variando de **0** a **n**.

A partir do primeiro ponto, **[x_0,$Y(x_0)$]**, constrói-se uma parábola passando por três pontos **consecutivos, isto é, por dois intervalos, cobrindo os pontos [x_0,$Y(x_0)$], [x_1,$Y(x_1)$] e [x_2,$Y(x_2)$].** Calcula-se a integral dessa parábola, de **x_0** a **x_2** . Toma-se, em seguida, o último ponto **[x_2,$Y(x_2)$]** e constrói-se outra parábola passando pelos pontos **[x_2,$Y(x_2)$], [x_3,$Y(x_3)$], [x_4,$Y(x_4)$],** calculando a integral da segunda parábola de **x_2** a **x_4**.

Repetem-se esses cálculos, até a última parábola, com os pontos **[x_{n-2},$Y(x_{n-2})$], [x_{n-1},$Y(x_{n-1})$] e [x_n,$Y(x_n)$].**

O valor da integral será aproximadamente a soma dos valores da **n/2** integrais das parábolas. Chega-se à fórmula apresentada acima.

Evidentemente, o valor exato da integral será diferente do valor aproximado, só coincidindo em casos particulares, até porque foram integradas parábolas e não a função **Y(x)** original.

Entretanto, quanto maior o número de divisões, mais as respectivas parábolas tendem a se aproximar dos valores da função original **Y(x).**

5.1 PERDA DE PRECISÃO COM O AUMENTO DE n

Vamos tomar uma integral conhecida, para verificarmos a queda do erro, com o aumento de **n**.

Seja a integral $\int_0^1 \frac{1}{(x+1)^2} dx$ cuja resolução analítica é: $[\frac{-1}{1+x}]_0^1 = -0,5 + 1 = 0,5$.

Sabendo que o resultado exato é **0,5**, vamos calcular esta integral numericamente, portanto aproximadamente, usando precisão simples (variáveis single), e verificar os erros que ocorrem, em especial a redução do erro com o aumento de **n**, isto é, do número de divisões do intervalo **[0,1].**

Usando os algoritmos que a bibliografia oferece, vamos calcular a integral definida apresentada de duas maneiras, dependendo da maneira como a variável independente **x** será calculada, sabendo-se que se pretende que ela varie de **0** a **1**. No cálculo de $\mathbf{I_1}$, $\mathbf{x_0 = a}$ e $\mathbf{x_i}$ é calculado como sendo $\mathbf{x_{i-1} + h}$, onde **h = (b-a)/n**. No cálculo de $\mathbf{I_2}$, $\mathbf{x_0 = a}$ e $\mathbf{x_i}$ vale **a + i*h**. Assim, em $\mathbf{I_1}$, $\mathbf{x_i = x_{i-1} + h}$, enquanto em $\mathbf{I_2}$, $\mathbf{x_i = a + i.h}$.

A grande diferença é que, em $\mathbf{I_1}$, $\mathbf{x_i = a + \sum_{k=1}^{i} h}$, para $\mathbf{1 \leq i \leq n}$.

Dessa forma, cada $\mathbf{x_i}$, $\mathbf{1 \leq i \leq n}$, será calculado a partir de um somatório de **h**. Com o aumento do número de parcelas, vai-se chegando a um ponto em que se somará **h** a um somatório grande de **h's** anteriores, isto é, se somará **h** a um número bem maior que **h**, perdendo-se precisão no resultado. Portanto, para se calcular $\mathbf{x_i}$, utilizando-se a fórmula $\mathbf{x_i = a + \sum_{k=1}^{i} h}$, para $\mathbf{i \geq 1}$ (usado no cálculo de $\mathbf{I_1}$), vai ficando cada vez mais distante de $\mathbf{x_i}$ calculado por $\mathbf{x_i = a + i*h}$ (usado no cálculo de $\mathbf{I_2}$). Esta segunda fórmula garante uma precisão maior no cálculo de cada $\mathbf{x_i}$, sendo evitada a perda de precisão decorrente do somatório.

O objetivo desse cálculo de $\mathbf{x_i}$ pelos dois métodos é o de mostrar o acúmulo de erros devido ao somatório.

A tabela abaixo resume os valores encontrados.

Tabela 3 – Erros da integral e do limite de integração em função de n

n	10^4	10^5	10^6	10^7	10^8	10^9
b em I_1	**1,00005352**	**1,00099015**	**1,00903893**	**1,06476748**	**0,25000000**	**0,03125000**
I_1	**0,500000417**	**0,499861538**	**0,498677611**	**0,491950542**	**0,335544318**	**0,033554431**
b em I_2	**1,00000000**	**1,00000000**	**1,00000000**	**1,00000000**	**1,00000000**	**1,00000000**
I_2	**0,499999702**	**0,499998927**	**0,499967158**	**0,489437610**	**0,335544318**	**0,033554431**

Pode-se observar que o valor de **b** usado no cálculo de $\mathbf{I_1}$ se afasta do valor exato, **1,00000000**, na medida em que cresce **n**. E mais, quando **n** passa de $\mathbf{10^7}$ para $\mathbf{10^8}$, o valor de **b** cai para **0,25** e, adiante, quando **n** passa de $\mathbf{10^8}$ para $\mathbf{10^9}$, há outra queda para **0,03125.**

Vamos entender o que houve.

No cálculo do valor de **x** pelo primeiro método, isto é, $\mathbf{x_i = a + \sum_{k=1}^{i} h}$, era de se imaginar que quando $\mathbf{i = n}$, $\mathbf{x_i}$ fosse igual a **b**, isto é, igual a **1,0**. Isso não ocorre, como se viu. A razão está no acúmulo de erros de precisão das somas, por se estar somando número de ordem de grandeza muito diferente, isto é, somando-se **h** a um somatório de **h's**.

Vejamos, em detalhes, o que se passou. Quando **n** passa de $\mathbf{10^7}$ para $\mathbf{10^8}$, **h** cai de $\mathbf{10^{-7}}$ a $\mathbf{10^{-8}}$, pois $\mathbf{h = (b-a) / n = (1-0) / n = n^{-1}}$.

Este será o valor a ser somado à soma dos **h's** anteriores.

Quando $\mathbf{x_i}$ chega a **0,25**, isto é, a soma dos **h's** chega a **0,25**, a esse $\mathbf{x_i}$ vai ser somado mais um **h**.

Vejamos a representação, em binário, de **0,25** como número real com precisão simples:

0,25 → 0,01 = 1,000....000.2^{-2}, na forma normalizada.

Lembremo-nos de que há **23** zeros após o **1,** (um vírgula).

Qual será o maior **ε** tal que **0,25 + ε = 0,25**, isto é, qual o maior **ε** que somado a **0,25** não consegue alterar o **0,25**, exatamente por ser tão pequeno que não chega a alterar o último de seus bits ?

Já se viu que, quando são somados dois números, o menor é desnormalizado, se necessário, para que passe a ter o mesmo expoente do maior.

Assim, o **ε** desnormalizado, não altera o último bit de **0,25**.

0,25 **0 01111101 00000000000000000000000 = 1,0.2^{-2}**

ε **0 01111101 00000000000000000000010000...**

0,25 = 1,0.2^{-2}

$\varepsilon = 2^{-24} . 2^{-2} = 2^{-26} \approx 1{,}49.10^{-8}$

Se tivermos qualquer **$\varepsilon > 2^{-26}$** , então, **0,25 + ε > 0,25**, o que alteraria o valor de $\mathbf{x_i}$ que continuaria a crescer, em direção a **1,0**, como esperado.

No caso, quando **$n = 10^8$** , isto é, **$h = 10^{-8}$, $h < 2^{-26} \approx 1{,}49.10^{-8}$**. Dessa forma, a soma de **h** não altera o valor de **$x_i = 0{,}25$**.

Por outro lado, podemos mostrar que, enquanto $\mathbf{x_i}$ não chegava a **0,25**, ele ia crescendo com a soma do $\mathbf{h = 10^{-8}}$.

Em primeiro lugar, calculemos o maior **x = M**, inferior a **0,25**, representável com precisão simples.

Tomemos $\mathbf{0,25 = 1,0 \,.\, 2^{-2}}$ e sua representação:

0,25 → 0 01111101 00000000000000000000000

Trata-se do menor $\mathbf{x_i}$ com expoente **-2 (01111101).** Para se conseguir um número menor que **0,25**, vai ser necessário reduzir o expoente **-2 (01111101)** para **-3 (01111100).** Busquemos o maior número normalizado com expoente **-3**.

M → 0 01111100 11111111111111111111111

$$\mathbf{M = 1,111..11 \,.\, 2^{-3} = (2 - 2^{-23}) \,.\, 2^{-3} = 2^{-2} - 2^{-26} = 0,25 - 2^{-26} \approx 2 - 1,49 \,.\, 10^{-8}}$$

Este é o maior número, inferior a **0,25** que pode ser representado como real com precisão simples.

Qual o menor $\boldsymbol{\delta} > \mathbf{0}$ que ainda alteraria esse valor se a ele somado ?

M → 0 01111100 11111111111111111111111

δ → 0 01111100 00000000000000000000010000...

$$\boldsymbol{\delta} = \mathbf{2^{-24} \,.\, 2^{-3} = 2^{-27} \approx 7,45.10^{-9} < 10^{-8} = h.}$$

Logo, somando-se **h**, ele altera **M**, chegando a **0,25**.

Vale a pena fazer uma breve recordação de como se passa o arredondamento, no resultado de uma operação.

Apresentamos dois casos:

0,25 e **M = 0,25 – 2^{-26} < 0,25.**

0,25 → 0 01111101 00000000000000000000000

ε → 0 01111101 00000000000000000000000100000...

M → 0 01111100 11111111111111111111111

δ → 0 01111100 00000000000000000000000100000...

No primeiro caso dissemos que o bit **1** do **ε** não alteraria o último bit **0** do **0,25**. No segundo caso afirmamos que o bit **1** do **δ** alteraria o último bit, no caso bit **1**, do **M**.

Qual a diferença ?

Pensemos inicialmente, na base **10**(dez). O arredondamento se faz olhando-se o valor a ser abandonado. Se for **> 0,5**, soma-se **1** à última casa e se for **< 0,5**, mantem-se a última casa. Assim, **14,52** se arredonda para **15** e **14,48** para **14**.

Se for exatamente **0,5** , o que deve ser feito ?

Recordando o que já foi colocado, na convenção adotada pela norma IEEE Standard 754, de 1985, afirma-se que tudo depende do último dígito anterior ao que será abandonado. Se ele for par, permanece par. Se ele for ímpar, soma-se **1** e ele vira par.

Na base dez, por exemplo, se tivermos que arredondar **1,5** para uma casa, ele fica sendo **2**. Se tivermos que arredondar **2,5** para uma casa, ele também fica sendo **2**. No caso de **3,5** ou **4,5**, o arredondamento levaria a **4**.

Na base dois, essa regra é mantida.

01,1 → 10 10,1 → 10 11,1→ 100 100,1 → 100

Recordemos o que foi visto acima, com **0,25** e **M**, imediatamente abaixo de **0,25**.

0,25 → 0 01111101 00000000000000000000000

ε → 0 01111101 00000000000000000000000100000...

M → 0 01111100 11111111111111111111111

δ → 0 01111100 00000000000000000000000100000...

Nesses casos, tem-se: **0,25 + ε = 0,25** e **M + δ = 0,25 > M**.

Dessa forma, o que se quer frisar é que, quando $\mathbf{n = 10^8}$, isto é, $\mathbf{h = 10^{-8}}$, a soma **0,25 + h** é igual a 0,25. (**0,25 + h = 0,25**)

A mesma análise pode ser feita para a razão de $\mathbf{x_i}$ ter se mantido em **0,03125**, quando se tomou $\mathbf{n = 10^9}$ ou $\mathbf{h = 10^{-9}}$.

0,03125 = 2^{-5} → 0 01111010 00000000000000000000000

ε → 0 01111010 00000000000000000000000010000...

Este é o maior ε que somado a **0,03125** não altera seu valor. Ou seja:

0,03125 + ε = 0,03125 .

$\varepsilon = 2^{-24} \cdot 2^{-5} = 2^{-29} = 1{,}86 \cdot 10^{-9}$.

Dessa forma, quando se soma $\mathbf{h = 10^{-9}}$ **a 0,03125** , sendo $\mathbf{10^{-9} < 1{,}86.10^{-9}}$, o valor **0,03125** não se altera, permanecendo **0,03125**, por mais que se some $\mathbf{10^{-9}}$.

5.2 QUEDA BRUSCA DO VALOR DA INTEGRAL

Outro ponto interessante a ser observado é que há queda do valor da integral da proximidade de **0,5** para **0,335544318**, tanto em $\mathbf{I_1}$, quanto em $\mathbf{I_2}$, ao se passar de $\mathbf{n = 10^7}$ para $\mathbf{n = 10^8}$.

O que terá acontecido?

Lembremo-nos que, no cálculo da integral, estão sendo feitas duas somas: **Soma4** e **Soma2**. Em **Soma4**, estão sendo somados os valores da função ($\mathbf{y = 1/(1+x)^2}$) que serão posteriormente multiplicados por **4**, dentro da fórmula de integração de Simpson. Em **Soma2**, estão sendo somados os valores da função ($\mathbf{y = 1/(1+x)^2}$) que serão posteriormente multiplicados por **2**. Em ambos os casos, se está somando o valor de $\mathbf{y = 1/(1+x)^2}$, **x** variando de **0** a **1**. A variação de $\mathbf{1/(1+x)^2}$ é dada a seguir.

$\mathbf{x = 0}$	$\mathbf{y = 1 / (1+x)^2 = 1}$
$\mathbf{0 < x \leq \sqrt{2} - 1}$	$\mathbf{1 > y = 1 / (1+x)^2 \geq 0{,}5}$
$\mathbf{\sqrt{2} - 1 < x \leq 1}$	$\mathbf{0{,}5 > y = 1 / (1+x)^2 \geq 0{,}25}$
$\mathbf{1 < x}$	$\mathbf{0{,}25 > y = 1 / (1+x)^2}$

Para se entender a razão porque o valor da integral cai de **0,5** para **0,335544318**, tanto em $\mathbf{I_1}$, quanto em $\mathbf{I_2}$, foi feita uma pequena alteração no programa para cálculo da

integral, de modo a se determinar em que ponto a **Soma**, usada para cálculo da integral, não é mais alterada pela soma da função $\mathbf{Y(x_i)}$.

Por outro lado, observou-se que, quando $\mathbf{n = 10^8}$, o valor de $\mathbf{Y(x_i)}$ que não altera a **Soma** foi **0,63999999** para $\mathbf{I_1}$ e **0,53798878** para $\mathbf{I_2}$.

Estando entre **0,5** e **1,0**, para qualquer dos dois valores, tanto **0,63999999** quanto **0,53798878**, sua representação em binário será: $\mathbf{Y(x_i) = 0,1abcde...}$ onde **a,b,c,d,e,..** são **0** ou **1**, claro que com valores diferentes para os dois.

Sua representação normalizada sendo: $\mathbf{Y(x_i) = 1,abcde... \,.\, 2^{-1}}$ **.(a)**

Vejamos, a partir de que somatório, a parcela $\mathbf{1, abcde... \,.\, 2^{-1}}$ já não acrescenta nada à **Soma**, isto é, não altera o último bit da **Soma**. Em algum momento, a representação do valor da **Soma** muda de expoente, passando de **k-1** para **k**. Nesse ponto, a **Soma** seria representada por: **Soma** → $\mathbf{1, 0000...000.2^k}$. É então, nesse ponto, que o primeiro bit de $\mathbf{Y(x_i)}$ desnormalizado, já não pode alterar o último bit da **Soma**.

Para tentarmos somar $\mathbf{Y(x_i)}$ à **Soma**, teríamos que desnormalizá-lo, para que assumisse o expoente **k**, ficando: $\mathbf{0,00000000000000000000000001abcde....2^k}$. (chamo a atenção de que há **24** bits **0**, para se ter a certeza de que no arredondamento não seria alterado o último bit **0**).

Assim, $\mathbf{Y(x_i)}$ valeria: $\mathbf{1,abcde...2^{-25+k} = 1,abcde...2^{-1}}$.(veja (a), acima)

Logo, $\mathbf{k = 24}$.

O valor que a soma assumiria seria: $\mathbf{1,000.....2^{24} = 2^{24}}$.

Com isso, a integral que é dada por $\mathbf{I \approx (Y[0]+4.0*Soma4+2.0*Soma2)*h/3.0}$,

Onde **Y[0]** é desprezível diante de **Soma4** e de **Soma2**. Essas duas somas tendem a $\mathbf{2^{24}}$, como visto acima.

Assim, $\mathbf{I \approx 6*2^{24}*10^{-8}/3.0 = 0{,}33554431}$, que é valor encontrado, tanto para $\mathbf{I_1}$, quanto para $\mathbf{I_2}$.

Repetindo-se o mesmo cálculo para $\mathbf{n = 10^9}$, chega-se ao valor $\mathbf{I \approx\ 0{,}033554431}$, como obtido.

Observe-se a perda de precisão, com o aumento de **n**.

É fundamental lembrar que o Método de Simpson, sob o ponto de vista teórico, previa que haveria aumento da precisão com aumento de **n**.

De fato, dada a integral $\int_a^b$ **f(x)dx,** o erro teórico (**e**) cometido, quando se aplica o Método de Simpson é conhecido e dado por : $\mathbf{e = (b-a)^5 f^{iv}(\varepsilon) / (180n^4)}$ (Burden,R et al. (2003) e Gerald (1989)).

O erro cometido é inversamente proporcional à quarta potência de **n**. Assim, ao se dobrar **n**, o erro fica aproximadamente **16** vezes menor, como indicado em seqüência.

Tabela 4 – Erro da integral em função de n (precisão simples)

n	**2**	**4**	**8**	**16**	**32**
I	**0,50463**	**0,500418**	**0,500030**	**0,5000020**	**0,50000012**
Erro	**0,00463**	**0,000418**	**0,000030**	**0,0000020**	**0,00000012**
razão		$\mathbf{e_2/e_4 = 11}$	$\mathbf{e_4/e_8 = 14}$	$\mathbf{e_8/e_{16} = 15}$	$\mathbf{e_{16}/e_{32} = 17}$

Tabela 5 – Erro da integral em função de n (precisão dupla)

n	**2**	**4**	**8**	**16**	**32**
I	**0,5046296**	**0,5004176**	**0,50002988**	**0,50000194**	**0,500000123**
Erro	**0,0046296**	**0,0004176**	**0,00002988**	**0,00000194**	**0,000000123**
Razão		e_2/e_4 **=11**	e_4/e_8 **=14**	e_8/e_{16} **= 15,4**	E_{16}/e_{32} **= 15,8**

Dobrando-se **n**, o erro tende a cair **16** vezes, aproximadamente.

Entretanto, como vimos, quando o número de divisões continua a crescer, o erro volta a crescer. Nesse caso, o crescimento do erro não se deve ao Método de Simpson, em si, mas à perda de precisão ao se somar números de ordem de grandeza muito diferentes, como visto.

5.3 ALGORITMO DE KAHAN

O erro em somatórios é analisado e sugestões para minimizá-lo estão presentes em Knuth (1981), bem como em Goldberg (1991). Em ambos é sugerida, para aumento da precisão, a utilização do Algoritmo Kahan para Somatório (Kahan, 1989).

A idéia central do Algoritmo de Kahan é a de buscar recuperar a perda de precisão que ocorre quando se somam valores de ordem de grandeza muito diferentes.

Suponhamos que se deseja calcular a soma **n** dos elementos do vetor X. Assim, $\mathbf{S = \sum X(i)}$, **i** variando de **1** a **n**.

A Seguir o algoritmo de Kahan:

```
S = X[1];
C = 0;
for j = 2 to N {
Y = X[j] - C;
T = S + Y;
C = (T - S) - Y;
S = T;
}
```

Observe-se que se está buscando colocar a soma em S. Inicialmente colocou-se em S o primeiro valor X[1].

O valor seguinte é colocado em Y (Y = X[j] - C;).

No loop, soma-se a S o valor seguinte, sendo guardado em T. (T = S + Y;) .

Neste ponto pode ter havido perda da precisão, com o arredondamento ocorrido nos últimos bits de Y.

Tenta-se recuperar esta perda com a expressão (C = (T – S) – Y;).

Esse valor C, que corresponde ao que se perdeu, é acrescido ao próximo valor a ser adicionado. (Y = X[j] – C;).

6 ALGORITMO PROPOSTO

Apresento a seguir, o algoritmo proposto, visando o aumento da precisão das integrais calculadas numericamente.

Tendo em vista que a origem do erro está no fato de serem adicionadas grandezas de ordem diferente, perdendo-se precisão do elemento de menor módulo, o algoritmo proposto vai minimizar as operações de soma com números de ordem de grandeza diferentes.

A idéia central do algoritmo sugerido é a de buscar-se, em todas as operações, adicionarem-se parcelas de mesma ordem.

Vamos admitir, para exemplificar, que se têm os valores seguintes:

a, b, c, d, e, f, g, h, i, j, k, ... que precisam ser somados para se obter o valor **Soma**.

Soma = a + b + c + d + e + f + g + h + i + j + k

Já se viu que, se forem somados nessa seqüência, haverá, forçosamente, a soma de um elemento com a soma parcial anterior a ele. Com isso, haverá perda de precisão da nova parcela que entra.

O algoritmo sugerido segue a seguinte lógica, após ser criado um vetor **Soma** com **n** elementos. Exagerando, vamos tomar **n = 101**. Os elementos são **Soma(0), Soma(1), ...Soma(100).** A dimensão necessária do vetor **Soma** será discutida mais tarde.

1. **Criam-se um vetor Soma e os campos i e j;**

2. **Inicialmente, zeram-se os componentes do vetor Soma e o campo i;**

3. **Soma-se 1 ao campo i e entra-se com o $i^{ésimo}$ elemento, que passa a ocupar a posição Soma(0). Se todos já tiverem entrado, ir para a etapa 5. Faz-se j = 1;**

4. É questionado se o elemento seguinte, Soma(j), está ocupado

a. Se estiver vazio coloca-se o elemento Soma(j-1) na posição Soma(j), zera-se o elemento Soma(j-1) e volta-se à etapa 3;

b. Se estiver cheio, soma-se o elemento Soma(j-1) ao elemento Soma(j), zera-se o elemento Soma(j-1), adiciona-se 1 a j e retorna-se à etapa 4;

5. Calcula-se a soma dos elementos Soma(j), j variando de 1 a 100, que será a soma total dos elementos de entrada.

O que se está fazendo é entrar com **a**, na posição **Soma(0),** pergunta-se se **Soma(1)** está ocupada, o que não ocorre e desloca **Soma(0)** para **Soma(1),** zerando **Soma(0).**

Em seguida entra-se com **b** em **Soma(0).** Pergunta-se se **Soma(1)** está ocupada e verifica que está (aliás, contendo o elemento **a**). Soma **b** ao conteúdo de **Soma(1)** que passa a valer **a+b**.

Pergunta-se, em seguida, se **Soma(2)** está ocupada, o que não ocorre. Desloca-se **Soma(1)** para **Soma(2),** que passa a conter **a + b**, zerando-se **Soma(1).**

Em seguida entra-se com **c** em **Soma(0).** Pergunta-se se **Soma(1)** está ocupada e verifica-se que não está. Coloca-se **c** em **Soma(1).**

Em seguida entra-se com **d** em **Soma(0).** Pergunta-se se **Soma(1)** está ocupada e verifica que está (aliás, contendo o elemento **c**). Soma-se **d** ao conteúdo de **Soma(1)** que passa a valer **c+d**.

Em seguida pergunta-se se **Soma(2)** está ocupada e verifica-se que de fato está, contendo **a + b**. Adiciona-se **Soma(1),** que está contendo **c+d** ao conteúdo de **Soma(2),** que contem **a+b**, ficando com **a+b+c+d**.

Em seguida pergunta-se de **Soma(3)** está ocupada e verifica-se que não está. Desloca-se **Soma(2)** para **Soma(3),** que passa a conter **a+b+c+d**, zerando-se **Soma(2).**

Em seguida entra-se com **e** em **Soma(0)**. Pergunta-se se **Soma(1)** está ocupada e verifica-se que não está. Coloca-se **e** em **Soma(1),** zerando-se **Soma(0).**

Em seguida entra-se com **f** em **Soma(0).** Pergunta-se se **Soma(1)** está ocupada e verifica-se que está (aliás, contendo o elemento **e**). Soma-se **f** ao conteúdo de **Soma(1)** que passa a valer **e+f**, zerando-se **Soma(0).**

Pergunta-se, em seguida, se **Soma(2)** está ocupada, o que não ocorre. Desloca-se **Soma(1)** para **Soma(2),** que passa a conter **e + f**, e zera-se **Soma(1).**

Em seguida entra-se com **g** em **Soma(0).** Pergunta-se se **Soma(1)** está ocupada e verifica-se que não está. Coloca **g** em **Soma(1).**

Em seguida entra-se com **h** em **Soma(0).** Pergunta-se se **Soma(1)** está ocupada e verifica que está (aliás, contendo o elemento **g**). Soma **h** ao conteúdo de **Soma(1)** que passa a valer **g+h**, zerando-se **Soma(0).**

Em seguida pergunta-se de **Soma(2)** está ocupada e verifica-se que está, contendo **e + f**. Adiciona-se **Soma(1),** que está contendo **g+h** ao conteúdo de **Soma(2),** que contem **e+f**, ficando com **e+f+g+h**.

Em seguida pergunta-se de **Soma(3)** está ocupada e verifica-se que está, contendo **a+b+c+d**. Adiciona-se **Soma(2)** a **Soma(3),** que passa a conter **a+b+c+d+e+f+g+h** e zera-se **Soma(2).**

Em seguida pergunta de **Soma(4)** está ocupada e verifica-se que não está. Desloca-se **Soma(3)** para **Soma(4),** que passa a conter **a+b+c+d+e+f+g+h** e zera-se **Soma(3).**

Em seguida entra-se com **i** em **Soma(0).** Pergunta-se se **Soma(1)** está ocupada e verifica-se que não está. Coloca **i** em **Soma(1)** e zera-se **Soma(0).**

Esta seqüência de operações se repete até serem processados todos os elementos.

Dessa forma se evita somar um elemento à soma de elementos. Um elemento só é somado a outro elemento, formando a soma de dois elementos. Essa soma de dois elementos será somada à soma de outros dois elementos, formando a soma de quatro elementos. Essa soma de quatro elementos será somada á soma de outros quatro elementos, formando a soma de oito elementos e assim por diante.

É interessante observar que, quando se tem um (2^0) elemento, chega-se ao nível **Soma(1).** Havendo dois (2^1) elementos, chega-se ao **Soma(2).** Com quatro (2^2) elementos, chega-se ao **Soma(3).** Com oito (2^3) elementos, chega-se ao **Soma(4).** Com dezesseis (2^4) elementos chega-se ao **Soma(5).**

Assim, com 2^n elementos, chega-se ao nível **Soma(n+1).** Dessa forma, tendo-se o número de elementos a serem somados menor que 2^n, basta tomar um vetor Soma com **n** elementos, além do **Soma(0)** que foi um simples instrumento para facilitar a programação, não permanecendo ocupado por qualquer elemento.

Para tornar ainda mais claro: se houver **20** elementos a serem somados, estando **20** entre 2^4 e 2^5, basta tomar o vetor **Soma** até **Soma(5).** Se tivermos exatamente **32** elementos, ($32 = 2^5$) teremos que usar o **Soma(6).** Isso define o tamanho mínimo necessário da dimensão do vetor **Soma**, onde estão sendo acumuladas as somas parciais.

Assim, se tivermos **n** parcelas a serem somadas, o vetor Soma variará de **Soma(0)** a **Soma(N),** onde $N > \log(n)/\log(2)$ ou $N > \log_2(n)$.

6.1 O ALGORITMO NO CÁLCULO NUMÉRICO DE INTEGRAIS

O resultado apresentado na Tabela 6 mostra o enorme ganho em precisão obtido pelo algoritmo sugerido, cuja listagem do programa está no APÊNDICE C. A tabela apresenta o aumento da precisão, isto é, a queda vertiginosa da perda de precisão no cálculo da integral apresentada, inclusive quando se estão somando bilhões de elementos, caso em que a perda de precisão dos métodos tradicionais é enorme, como vimos anteriormente.

Claro que o algoritmo não está imune à perda de precisão, até porque essa perda é inerente à própria maneira como os números são armazenados em ponto flutuante, limitando o número de dígitos a serem utilizados, na escrita de um número.

Voltando-se à integral sugerida:

$$I = \int_0^1 \frac{1}{(x+1)^2} dx = 0,5000000$$

A tabela abaixo, apresenta os erros ocorridos no cálculo numérico da integral acima, pelo Método de Simpson, considerando três maneiras de implementar esse método: $\mathbf{I_1}$, $\mathbf{I_2}$ e $\mathbf{I_3}$.

As implementações $\mathbf{I_1}$ e $\mathbf{I_2}$ já foram definidas anteriormente, onde o Método de Simpson é aplicado seguindo-se diretamente a fórmula já citada, sendo os somatórios calculados seguindo a maneira tradicional de fazê-lo, somando-se cada novo termo ao total existente até então. A implementação $\mathbf{I_3}$, embora aplicando também o Método de Simpson, utiliza o algoritmo apresentado no cálculo dos somatórios. Conhecendo-se o resultado da integral, que é exatamente **0,5**, seguem, na tabela, os erros obtidos em cada caso.

Tabela 6 - Módulo dos erros das diversas implementações

n	Single			Double		
	e(I1)	e(I2)	e(I3)	e(I1)	E(I2)	e(I3)
1,00E+04	4,17E-07	2,98E-07	0,00E+00	1,72E-14	1,11E-16	0,00E+00
1,00E+05	1,38E-04	1,07E-06	0,00E+00	7,12E-14	2,00E-15	0,00E+00
1,00E+06	1,32E-03	3,28E-05	0,00E+00	5,85E-13	2,11E-15	0,00E+00
1,00E+07	8,05E-03	1,06E-02	0,00E+00	1,67E-11	2,63E-14	0,00E+00
1,00E+08	1,64E-01	1,64E-01	0,00E+00	1,28E-10	9,02E-14	0,00E+00
1,00E+09	4,66E-01	4,66E-01	5,96E-08	6,82E-10	2,16E-13	0,00E+00

Recordo que se trata da integral já referida: $\int_0^1 \frac{1}{(x+1)^2} dx$ cujo resolução analítica é: $[\frac{-1}{1+x}]_0^1 = -0{,}5 + 1 = 0{,}5$.

Enfatizamos que, na tabela acima, está sendo apresentado não o resultado da integral, mas o erro, isto é, a diferença entre o verdadeiro valor **0,5** e o valor calculado.

As colunas com **e(I1)** e **e(I2)** apresentam os erros nos cálculos já indicados anteriormente, onde, em **I1**, cada novo $\mathbf{x_i}$ era dado pela fórmula $\mathbf{x_i = x_{i-1} + h}$, sendo que, em **I2**, $\mathbf{x_i = a + i*h}$, evitando o erro das sucessivas somas ao se adicionar **h** a $\mathbf{x_{i-1}}$, como já analisado.

Na coluna **e(I3),** foi feito o cálculo da integral, numericamente, usando-se o algoritmo recém descrito. O aumento da precisão é indiscutível, ficando ainda mais nítido quando cresce o número de divisões.

No caso referente à precisão simples, vamos observar as três últimas linhas da tabela, acrescidas agora do próprio valor da integral calculada, lembrando que o valor correto é **0,5**.

Tabela 7 – Integral e erro em três implementações (precisão simples)

n	I1	E(I1)	I2	e(I2)	I3	e(I3)
1,00E+07	0,49195	8,05E-03	0,4894	1,06E-02	0,50000	0,00E+00
1,00E+08	0,3355	1,64E-01	0,3355	1,64E-01	0,50000	0,00E+00
1,00E+09	0,03355	4,66E-01	0,03355	4,66E-01	0,50000	5,96E-08

Já foi observada a perda expressiva de precisão, quando **n** aumenta de $\mathbf{10^7}$ para $\mathbf{10^8}$, caindo o valor para **0,3355**; e quando vai de $\mathbf{10^8}$ para $\mathbf{10^9}$, cai para **0,03355**, valores esses já explicados.

O que se quer observar agora é que, ainda nesses casos, o erro do método pelo algoritmo sugerido ou é zero, ou muito pequeno, **5,96 E-08**, quando $\mathbf{n = 10^9}$.

Nesse caso, valendo a integral calculada **0,4999999404**, ela será representada da seguinte maneira: **0,011111.....,** sendo ligeiramente abaixo de **0,5**, que é **0,100000....**Dessa forma, a representação normalizada será: $\mathbf{1,1111....2^{-2}}$ sendo, portanto, o expoente representado em **8** bits, o binário equivalente a **125**, pois **125 – 127 = -2**.

A representação interna será: **0 01111101 1111.....**

O último bit, o vigésimo terceiro bit da mantissa, logo após o (1,) que não é representado, vale: $\mathbf{e = 2^{-23} \cdot 2^{-2} = 2^{-25} = 2{,}98.10^{-8}}$ .Dessa forma, o erro está no penúltimo bit da mantissa, o vigésimo segundo bit, demonstrando a grande precisão atingida pelo algoritmo. Lembrando que os métodos tradicionais levavam a um valor de **0,03355** , no lugar do **0,5** correto.

Mesmo no caso da precisão dupla, mostrado na tabela acima, observa-se que o erro de arredondamento, com **15** casas decimais, não chega a aparecer. O resultado é rigorosamente exato. Foi testado, ainda, um segundo exemplo, com a integral:

$$\int_{o}^{\pi} sen\ (x)\,dx = [-\cos(x)]_{0}^{\pi} = 1 + 1 = 2$$

Foi dividido o intervalo **[0,π]** em **n** partes e calculada a integral, numericamente.

A tabela a seguir mostra os resultados alcançados, indicando somente os erros.

Tabela 8 – Erros pela aplicação de cada um dos três métodos

n	Precisão Simples			Precisão Dupla		
	I1	I2	I3	I1	I2	I3
1,00E+04	2,56E-05	2,38E-07	0,00E+00	3,02E-13	3,33E-15	4,44E-16
1,00E+05	2,64E-03	1,03E-05	0,00E+00	3,70E-13	1,47E-14	0,00E+00
1,00E+06	1,94E-02	1,93E-04	0,00E+00	7,98E-12	1,09E-14	0,00E+00
1,00E+07	6,39E-02	1,56E-02	0,00E+00	4,14E-10	9,59E-14	0,00E+00
1,00E+08	9,46E-01	9,46E-01	1,19E-07	3,56E-09	4,02E-13	0,00E+00
1,00E+09	1,99E+00	1,89E+00	0,00E+00	5,39E-08	2,63E-12	0,00E+00

Mais uma vez, vamos observar as três últimas linhas no caso de precisão simples.

Tabela 9 – Valor da integral e erro em cada um dos três métodos

n	I1	E(I1)	I2	e(I2)	I3	e(I3)
1,00E+07	2,0639	6,39E-02	2,0156	1,56E-02	2,0000	0,00E+00
1,00E+08	1,054	9,46E-01	1,054	9,46E-01	2,0000	1,19E-07
1,00E+09	0,00659	1,993E+00	0,105	1,895E+00	2,0000	0,00E+00

É nítida a queda da precisão nas implementações que utilizam o cálculo dos somatórios da maneira tradicional, cuja razão já foi explicada. Ainda nesses casos, é desprezível o erro obtido quando se aplica o algoritmo sugerido, quando existe erro.

O método não está isento de erro, pois algum arredondamento pode se dar, em especial no somatório final, feito após a entrada da última parcela. Vejamos o caso emblemático,quando $\mathbf{n = 10^4}$, precisão dupla, onde $\mathbf{e(I3) = 4{,}44\ 10^{-16}}$ **.**

Neste caso, precisão dupla, estão sendo utilizados **52** bits para a mantissa, sem incluir o (**1,**), que está escondido.

Sendo o valor da integral calculada **2,000000000000000444**, houve um erro de $\mathbf{4{,}44 \cdot 10^{-16}}$ **.**

Vejamos o valor do último bit, isto é, do $\mathbf{52^o}$ bit da mantissa do número **2**. Seja $\boldsymbol{\varepsilon}$ este valor. $\boldsymbol{\varepsilon = 2^{-52} \cdot 2^1 = 2^{-51} = 4{,}44.10^{-16}}$ **.** O erro se deu, somente, no último bit, no $\mathbf{52^o}$ bit do número real representado em precisão dupla.

6.2 ACELERANDO O ALGORITMO

Como se viu, dado um vetor **Y(i),** quando se busca calcular a soma $\mathbf{S = \sum_1^n Y(i)}$, o algoritmo cria um vetor **Soma**, inicialmente zerado. Em seguida, calcula-se:

- Soma(1) =Y(1);
- Soma(1) = Soma(1)+Y(2)=Y(1)+Y(2);
- Guarda-se Soma(1) em Soma(2), zerando Soma(1);
- Calcula-se um novo Soma(1)=Y(3);
- Soma(1)= Soma(1)+Y(4)=Y(3)+Y(4);
- Soma(2)=Soma(2)+Soma(1)=Y(1)+Y(2)+Y(3)+Y(4);
- Guarda-se
- Soma(2) em Soma(3), zerando Soma(2);
- etc...

Algumas observações:

Sempre se vai somar um elemento a outro ou a soma de dois elementos à soma de dois elementos ou a soma de quatro elementos à soma de quatro elementos etc... Essa é a razão por que minimiza-se a perda de precisão no somatório, evitando-se soma elementos de ordem de grandeza muito diferentes.

Entretanto, como se estão sendo somados dois (2^1) elementos, quatro (2^2) elementos, oito (2^3) elementos etc, esta seqüência pode ser simulada por um ninho de loops.

```
I := 0;
   SOMA [3] := 0.0;
FOR I3 := 0 to 1 do     Begin
   SOMA [2] := 0.0;
FOR I2 := 2*I3 to 2*I3+1 do   Begin
   SOMA [1] := 0.0;
FOR I1 := 2*I2 to 2*I2+1 do              Begin
   INC(I);
   SOMA [1] := SOMA [1]+ Y(I);
                                          end;
   SOMA [2] := SOMA [2]+ SOMA [1];
                                    end;
   SOMA [3] := SOMA [3]+ SOMA [2];
                              end;
\
```

Usando estes três loops, **Y(1)** será somado a Y(2), Y(3) a Y(4), Y(1) + Y(2) a Y(3) + Y(4), Y(5) a Y(6), Y(7) a Y(8), Y(5)+Y(6) a Y(7)+Y(8) e finalmente,

Y(1)+Y(2)+Y(3)+Y(4) a Y(5)+Y(6)+Y(7)+Y(8), dando em **Soma[3]** a soma dos oito elementos.

Esta rotina evita alguns testes feitos anteriormente, como o de perguntar se um campo estava zerado ou não, visto que a seqüência será sempre exatamente a mesma: o primeiro somado ao segundo, o terceiro ao quarto, os dois primeiros aos dois seguintes etc...

6.3 COMPARANDO OS ALGORITMOS

Vamos a seguir comparar os tempos dos dois algoritmos: o algoritmo de Kahan, já citado, e o algoritmo acelerado. Para tanto, vamos calcular a integral $\int_0^1 \frac{1}{(x+1)^2} dx$, já referida, pelos dois métodos.

A tabela a seguir compara os tempos gastos pelos dois algoritmos, resolvendo a integral acima, sendo **n**, coluna da esquerda, o número de divisões do intervalo **[0, 1]**, sendo T1 – tempo do algoritmo de Kahan e T2 – tempo do algoritmo proposto acelerado. Foi utilizado um computador Pentium 4, com 2.6 Mhz de clock e uma memória ram de 1 Gbytes.

Tabela 10 – Comparação de tempos entre os dois algoritmos

n	**T1**	**T2**
2^{30} = 1073741824	2min 10seg	1min 59seg
2^{31} = 2147483648	4min 22seg	3min 58seg
2^{32} = 4294967296	8min 45seg	7min 56seg
2^{33} = 8589934592	17min 17seg	15min 52seg

Como se observa, o algoritmo que se está propondo é **9%** mais rápido que o de Kahan.

7 A INTEGRAL DUPLA

Finalmente, apresento um terceiro exemplo, neste caso tratando-se de integral dupla.

Foi desenvolvido um programa em Delphi, listagem no APÊNDICE D, para cálculo da seguinte integral:

$$\int_0^2 \int_{y^2}^{2+y} (4x + 2y)\,dxdy$$

Esta integral é facilmente calculada, considerando-se y constante e integrando-se:

$$I_x(y) = \int_{y^2}^{2+y} (4x + 2y)\,dx$$

$$I_x(y) = [2x^2 + 2xy]_{y^2}^{2+y} = \{2*(2+y)^2 + 2*(2+y)*y\} - \{2y^4 + 2y^2.y\} =$$
$$= 2(4+4y+y^2) + 4y + 2y^2 - 2y^4 - 2y^3 = -2y^4 - 2y^3 + 4y^2 + 12y + 8$$

$$I = \int_0^2 (-2y^4 - 2y^3 + 4y^2 + 12y + 8)dy = [-\frac{2}{5}y^5 - \frac{2}{4}y^4 + \frac{4}{3}y^3 + 6y^2 + 8y]_0^2 =$$
$$= -\frac{64}{5} - \frac{32}{4} + \frac{32}{3} + 24 + 16 = 40 - \frac{768 + 480 - 640}{60} = 40 - 10{,}133333... = 29{,}86666...$$

Vejamos o desenvolvimento seguido, ao ser feita a integral em **x**.

$$I_x(y) = \int_{y^2}^{2+y} (4x + 2y)\,dx$$

Chega-se a uma função de $\mathbf{I_x(y)}$.

A Integral Dupla será

$$I = \int_0^2 I_x(y)\,dy$$

Tentemos calcular esta integral **I** pelo Método de Simpson, já apresentado.

h_y = (2-0)/n, onde **n** é o número de partes em que se está dividindo o intervalo [0,2].

$$I = \frac{h_y}{3}(I_x(y_0) + 4I_x(y_1) + 2I_x(y_2) + 4I_x(y_3) + ... + 2I_x(y_{n-2}) + 4I_x(y_{n-1}) + I_x(y_n))$$

Sendo $y_0 = 0$, $y_n = 2$ e $y_i = y_0 + i * h_y$.

Cada parcela deste somatório é formado por uma integral:

$$I_x(y) = \int_{y^2}^{2+y} (4x + 2y)\,dx$$

Assim,

$$I_x(0) = \int_{0^2}^{2+0} (4x + 2*0)\,dx = \int_0^2 4x\,dx$$

.....

$$I_x(y_i) = \int_{y_i^2}^{2+y_i} (4x + 2y_i)\,dx$$

.....

$$I_x(2) = \int_{2^2}^{2+2} (4x + 2*2)dx = \int_4^4 (4x+4)dx = 0$$

Cada uma dessas integrais é, da mesma forma, calculada pelo Método de Simpson.

$$\int_{y_i^2}^{2+y_i} f(x)dx = \frac{h_x}{3}(f(x_0) + 4f(x_1) + 2f(x_2) + ... + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n))$$

Calculadas estas integrais, elas são aplicadas na fórmula:

$$I = \frac{h_y}{3}(I_x(y_0) + 4I_x(y_1) + 2I_x(y_2) + 4I_x(y_3) + ... + 2I_x(y_{n-2}) + 4I_x(y_{n-1}) + I_x(y_n))$$

Com isso, tem-se a integral dupla desejada. Tomou-se a integral citada, por ter resultado conhecido, de modo a poder-se comparar com os resultados numéricos, chegando-se aos resultados abaixo, onde **IA** representa a Integral dupla calculada sem uso do algoritmo, reservado para cálculo de **IN**.

As duas tabelas abaixo mostram a variação do valor da integral como função de **n.**

Tabela 11 - Integral e erros dobrando-se n (precisão simples)

n	IA	e(IA)	IN	e(IN)
1000	29,8666706	3,81E-06	29,8666668	0,00E+00
2000	29,8666573	9,54E-06	29,8666649	1,91E-06
4000	29,8666859	1,91E-05	29,8666668	0,00E+00
8000	29,8667011	3,43E-05	29,8666687	1,91E-06
16000	29,8666000	6,68E-05	29,8666687	1,91E-06

Tabela 12 - Integral e erros multiplicando-se n por 10 (precisão simples)

n	IA	e(IA)	IN	e(IN)
10	29,8658142	8,53E-04	29,865814	8,53E-04
100	29,8666687	1,91E-06	29,8666687	1,91E-06
1.000	29,8666706	3,81E-06	29,8666668	0,00E+00
10.000	29,8667068	4,01E-05	29,8666687	1,91E-06
100.000	29,8667221	5,53E-05	29,8666668	0,00E+00
200.000	29,8670101	3.43E-04	29,8666649	1,91E-06
600.000	29,8642578	2,41E-03	29,8666668	0,00E+00

Como era de se esperar, quando **n** cresce, em **IA**, o erro cai até certo ponto e depois começa a crescer. Isso não ocorre com **IN**, onde o erro não cresce, mesmo com **n** da ordem de **600.000**.

Observe-se que, com **n = 100** já se atinge a precisão da máquina, pois:

29,8666... = 11101,110... = 1,1101110...2^4

Logo o último bit valerá $\mathbf{2^{-23}.2^4 = 2^{-19} = 1{,}91.10^{-6}}$. Usando-se o algoritmo, este é o erro que ocorre, quando ocorre.

Limitamo-nos a **600000**, pois, tratando-se de integral dupla, será necessário um cálculo de **600000 * 600000** funções, isto é: $\mathbf{360.10^9 = 360}$ bilhões e o tempo de máquina foi de **24** horas, aproximadamente. Para **1** milhão de divisões, teríamos uns **4** dias de máquina.

Outro exemplo:

$$I = \int_0^{\pi}\int_0^{\pi} \cos(x.y)\,dxdy$$

O gráfico dessa superfície, **Z(x.y) = cos(x.y)**, é mostrado abaixo.

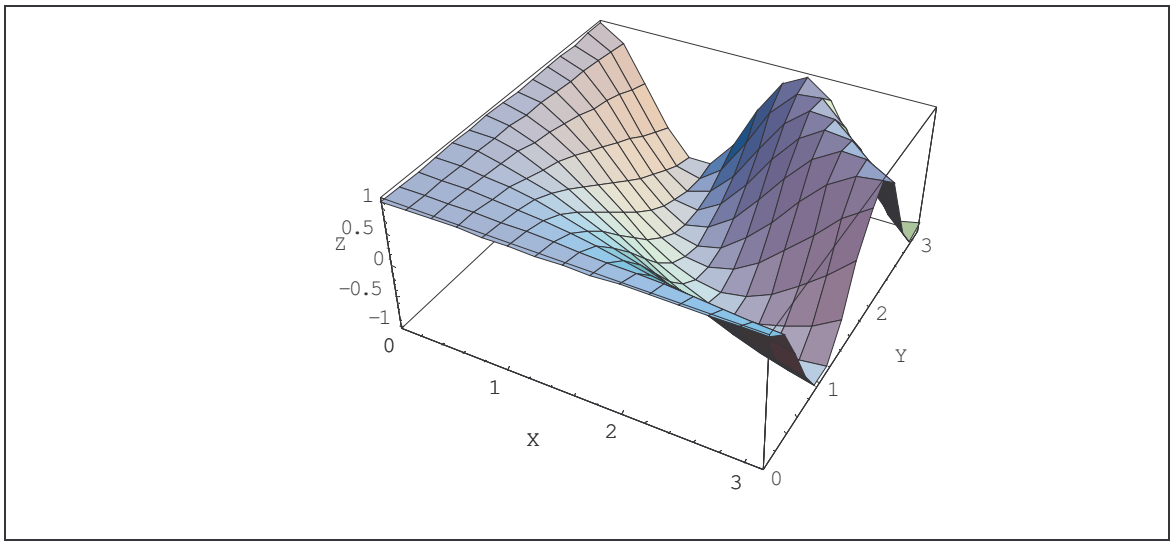

Gráfico 2 - **Gráfico de cos(x.y) $0 \le x \le \pi$, $0 \le y \le \pi$**

Trata-se de calcular a integral. Vamos buscar resolvê-la analiticamente.

$$\int_0^{\pi} \cos(x.y)dx = [\frac{sen(x.y)}{y}]_0^{\pi} = \frac{sen(\pi.y)}{y}$$

É preciso calcular a integral

$$I = \int_0^{\pi} \frac{sen\ (\pi\ .\ y\ )}{y} dy$$

Para calcular estas integrais, torna-se necessário um breve estudo.

As integrais a seguir recebem os nomes de **SinIntegral(z)** e **CosIntegral(z).**

$$\text{SinIntegral(x)} = \int_0^{x} \frac{sen\ (t)}{t}\, dt$$

$$\text{CosIntegral(x)} = -\int_x^{\infty} \frac{\cos(t)}{t}\, dt$$

O software Mathematica tem estas funções disponíveis.

Temos que calcular

$$I = \int_0^{\pi} \frac{sen\ (\pi\ .\ y\ )}{y} dy$$

Fazemos **z = π.y** . Logo **dy = dz / π** .

Se **y = 0 → z = 0**.

Se **y = π → z = π^2**

$$I = \int_0^{\pi} \frac{sen\ (\pi . y)}{y} dy = \int_0^{\pi^2} \frac{sen\ (z)}{\frac{z}{\pi}} \frac{dz}{\pi}$$

$$I = \int_0^{\pi^2} \frac{sen\ (z)}{z} dz$$

I = SinIntegral(π^2) = 1,6647491833.....

Calculando-se numericamente, chegamos ao seguinte resultado:

Tabela 13 – SinIntegral (π^2) para diferentes valores de **n**

n	**I_1**	**e_1**	**I_2**	**E_2**
4096	**1,664749622**	**$4{,}768.10^{-7}$**	**1,664749146**	**0,0**
8192	**1,664749146**	**0,0**	**1,664749146**	**0,0**
16384	**1,664750695**	**$1{,}550.10^{-6}$**	**1,664749146**	**0,0**

Outro exemplo:

$$I = \int_{-\pi/2}^{\pi/2} \int_{-\pi/2}^{\pi/2} \cos(\ x . y\ ) dxdy$$

O gráfico de **cos (x*y)** é mostrado a seguir.

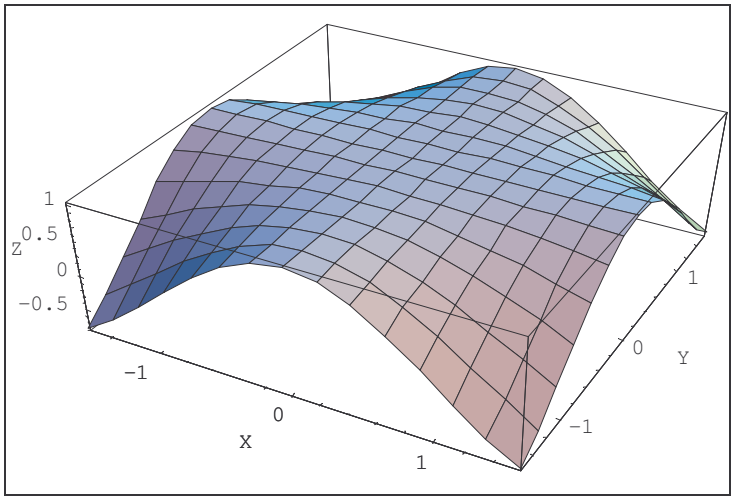

Gráfico 3 - **cos (x.y), $-\pi/2 \leq x \leq \pi/2$, $-\pi/2 \leq y \leq \pi/2$**

Calculando-se analiticamente, tem-se:

$$I_x(y) = \int_{-\pi/2}^{\pi/2} \cos(x.y)dx = [\frac{sen(x.y)}{y}]_{-\pi/2}^{\pi/2}$$

$I_x(y) = (sen(y.\pi/2) - sen(-y.\pi/2))/y = 2sen(y.\pi/2)/y$

$$I = \int_{-\pi/2}^{\pi/2} \frac{2\,sen\,(y.\pi/2)}{y} dy$$

Fazendo **z = πy/2** **dz = π/2 . dy** **y = 2z/ π**

y = - π/2 → **z = - π^2/4**

y = π/2 → **z = π^2/4**

$$I = 2\int_{-\pi^2/4}^{\pi^2/4} \frac{sen\,(z)}{\frac{2z}{\pi}} \frac{2}{\pi} dz = 4\int_{0}^{\pi^2/4} \frac{sen\,(z)}{z} dz$$

I = 4,0 . SinIntegral (π^2/4) = 4,0 . 1,77049467725571

I = 7,0819787090228418101844

Vejamos, na tabela a seguir, o cálculo desta integral, numericamente, pelo Método de Simpson.

Tabela 14 - **cos (x.y), $-\pi/2 \leq x,y \leq \pi/2$** , Precisão simples

n	I_1	e_1	I_2	e_2
30	7.0819865E+00	0,0000078	7.0819860E+00	0,0000073
300	7.0819778E+00	0,000000909	7.0819788E+00	0,00000009
3000	7.081984996E+00	0,00000629	7.0819787979E+00	0,0000000889
100000	7.082067966E+00	0,000089	7.0819783211E+00	0,0000003879
100	7.081978321E+00	4.76837E-07	7.081978798E+00	0,000000E+00
1000	7.081980705E+00	1.90735E-06	7.081979275E+00	4.768371E-07
1024	7.081975937E+00	2.86102E-06	7.081978321E+00	4.768371E-07
2048	7.081980705E+00	1.90735E-06	7.081978798E+00	0,000000E+00
4096	7.081980705E+00	1.90735E-06	7.081978321E+00	4.768372E-07
8192	7.081983566E+00	4.76837E-06	7.081978798E+00	0,000000E+00
16384	7.081994057E+00	1.52588E-05	7.081978798E+00	0,000000E+00

Tabela 15 - **cos (x.y), $-\pi/2 \leq x,y \leq \pi/2$,** Precisão dupla

n	I_1	E_1	I_2	e_2
16384	7.0819787088	1.91977E-10	7.0819787090	0.00000000E+00
32768	7.0819787095	4.30239E-10	7.08197870902284	8.88178E-16
65536	7.0819787090	1.04485E-11	7.08197870902284	0.00000000E+00
131072	7.0819787090	2.09548E-11	7.08197870902284	0.00000000E+00
262144	7.0819787090	2.16716E-12	7.08197870902284	8.88178420E-16
524288	7.0819787090	8.34888E-13	7.08197870902284	8.88178420E-16

Outro exemplo:

$$I = \int_{-\pi/2}^{\pi/2}\int_{-\pi/2}^{\pi/2} \cos(x+y)dxdy$$

Essa superfície é traçada abaixo, vista de ângulos diferentes.

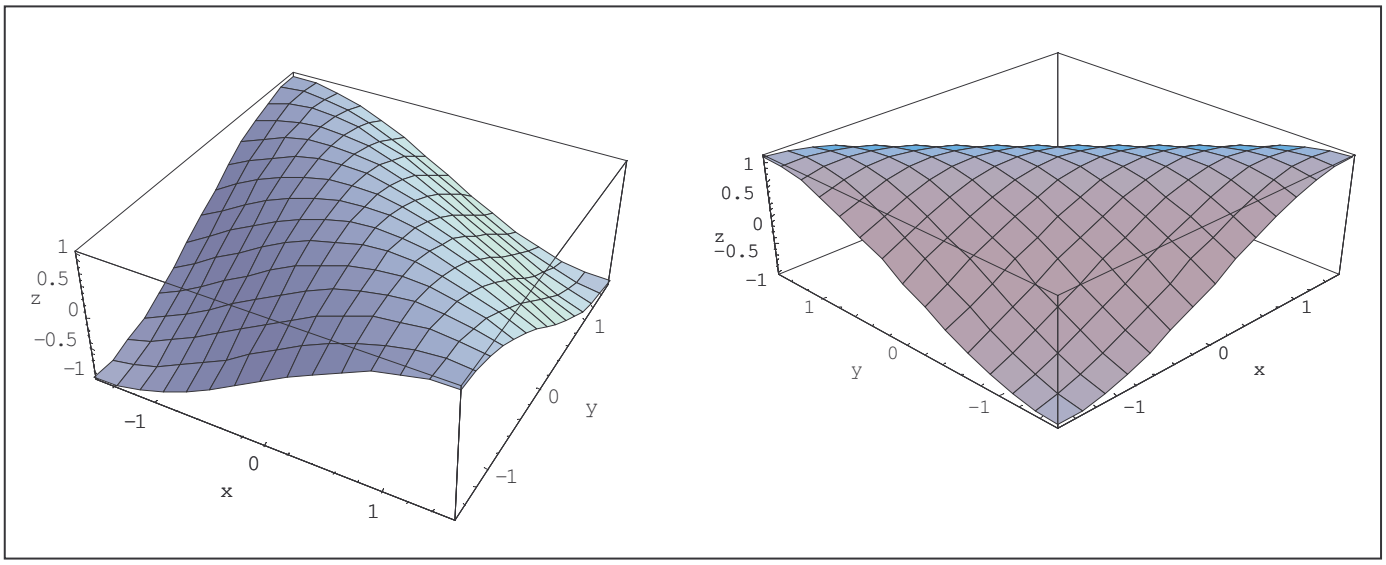

Gráfico 4 - **cos (x+y), $-\pi/2 \le x \le \pi/2$, $-\pi/2 \le y \le \pi/2$**

Esta integral é facilmente calculada.

$$I = \int_{-\pi/2}^{\pi/2}\int_{-\pi/2}^{\pi/2} \cos(x+y)\,dxdy = \int_{-\pi/2}^{\pi/2} [sen\ (x+y)]_{-\pi/2}^{\pi/2}$$

$$\int_{-\pi/2}^{\pi/2} (sen\,(\frac{\pi}{2}+y) - sen\,(\frac{-\pi}{2}+y))\,dy = \int_{-\pi/2}^{\pi/2} (\cos(y)+\cos(y))\,dy = 4$$

I = 4

A tabela seguinte mostra esta integral calculada pelo Método de Simpson, onde I_1 e I_2 representam, respectivamente, a integral calculada sem e com o algoritmo proposto.

Tabela 16 – Integral de **cos (x+y), $-\pi/2 \leq x, y \leq \pi/2$** , Precisão simples (single).

n	I_1	e_1	I_2	e_2
8192	**3,9999948**	**5,25E-06**	**4,000000000**	**0,0000000**
16384	**4,0000043**	**4,29E-06**	**4,000000000**	**0,0000000**
32768	**4,0000100**	**1,00E-05**	**4,000000000**	**0,0000000**
65536	**4,0000176**	**1,76E-05**	**4,000000048**	**4,8E-07**
131072	**4,0000572**	**5,72E-05**	**4,000000000**	**0,0000000**
262144	**4,0001526**	**1,53E-04**	**3,999999762**	**2,4E-07**

Mesmo sem termos atingido n muito elevado, da ordem de $\mathbf{10^9}$, como fez-se em integrais simples, ainda assim, observa-se o crescimento do erro com o aumento de n.

O erro observado com o algoritmo proposto, quando ocorre, está aparecendo no último bit, isto é, no $\mathbf{23^0}$ bit após a vírgula na mantissa.

Vejamos quando o resultado obtido é ligeiramente acima de **4**, no caso **4,000000048**. Sua representação será: **1,0000.... .2^2** . Qual o valor do $\mathbf{23^0}$ bit após a vírgula?

$\mathbf{2^{-23} \cdot 2^2 = 2^{-21} = 4{,}8.10^{-7}}$ que é o valor do erro.

No caso em que o valor é ligeiramente abaixo de **4,0**, isto é, **3,999999762**, sua representação será: **1,11111.... 2^1**.

O valor do $\mathbf{23^0}$ bit após a vírgula será: $\mathbf{2^{-23} \cdot 2^1 = 2^{-22} = 2{,}4.10^{-7}}$, que é o erro presente.

O que se pretende observar é a queda do erro, o aumento da precisão do cálculo quando se utiliza o algoritmo proposto.

Não se está afirmando que sempre o erro aparecerá no último bit e, sim, que há aumento expressivo na precisão do cálculo.

8 CONCLUSÃO

O fato de que, com o uso dos computadores, não se trabalha com números exatos pode levar a erros surpreendentemente grandes. Isso pode se tornar particularmente grave porque, nos algoritmos matemáticos, na imensa maioria dos casos, opera-se com número reais aproximados. E mais, mesmo quando se entra com números exatos, logo que é feita uma operação matemática, o resultado deverá ser aproximado, salvo em casos muito particulares.

Isso se deve ao fato de se operar com números em ponto flutuante (floating point), caso em que, o número real deve ser representado num número finito de bits. Em decorrência, haverá, necessariamente, um número finito de números reais representáveis. Como conseqüência, as propriedades dos números reais, tais como distributiva e associativa, não são válidas para os números reais representados em ponto flutuante. Assim, o resultado das operações matemáticas depende da ordem em que as operações são feitas.

Por outro lado, como se necessita armazenar o número real em número definido de bits, alguns bits são reservados para representarem o expoente, outros bits para a representação do corpo do número, responsável por sua precisão, chamado de mantissa, além do próprio sinal do número, representado por um bit.

Com exceção do zero e do underflow gradual, os números são representados internamente em binário, na forma normalizada, isto é, com um bit diferente de zero, antes da vírgula (ou do ponto).

Quando se necessita somar números de grandezas diferentes, isto é, números que têm expoentes diferentes, é necessário desnormalizar o menor número, de forma que ele seja escrito com o expoente do maior número. Ao ser isso feito, havendo um número limitado de bits, haverá perda de precisão no resultado da operação de soma.

No caso de ser calculado um somatório de número elevado de parcelas, na medida em que as parcelas vão sendo acumuladas, a nova parcela que chega para ser somada, será acrescentada a um número de ordem de grandeza mais elevada que a desta

parcela. Em decorrência, a soma perde precisão. Em geral, quanto maior o número de parcelas, maior a perda de precisão do somatório.

O objetivo deste trabalho é, exatamente, o de apresentar um algoritmo que minimiza a perda de precisão do somatório, no caso de número elevado de parcelas.

O artifício central do algoritmo é o de evitar a soma de parcelas de grandezas diferentes, minimizando os momentos em que se soma uma parcela a uma soma de parcelas. Isso se consegue, somando-se uma parcela a uma parcela, a soma de duas parcelas à soma de duas parcelas, a soma de 2^n parcelas à soma de 2^n parcelas. Dessa forma, consegue-se somar todas as parcelas, obtendo-se a soma total, minimizando as oportunidades em que sejam somadas parcelas de ordem diferentes.

Tomou-se, como emblemático da necessidade de somatórios, o cálculo numérico de integrais, onde os somatórios necessariamente aparecem.

É sobejamente conhecido na literatura da área numérica, o Método (1/3) de Simpson, ou das parábolas. Nesse método, o intervalo de integração é dividido num número **n**, par, de partes, e por esses **n+1** pontos da função a ser integrada, passam-se parábolas. Essas parábolas integradas dão, aproximadamente, o valor da integral da função desejada.

Aumentando-se o número **n** de divisões, os pontos representam melhor a função original e, em decorrência, é menor o erro do cálculo numérico efetuado.

Demonstra-se, e a literatura é amplamente conhecida, que com o aumento de **n**, o erro cai proporcionalmente à quarta potência de **n**. Dessa forma, dobrando-se **n**, o erro cai aproximadamente $\mathbf{2^4 = 16}$ vezes.

Observa-se que, continuando-se a aumentar **n**, após um certo ponto, o erro começa a aumentar e não a cair, como se esperaria do método. A razão está, exatamente, no aumento do número de parcelas.

Para superar essa limitação, foi desenvolvido um algoritmo que minimiza o erro decorrente do somatório de grande número de parcelas.

Aplicou-se o algoritmo desenvolvido no cálculo numérico dessas integrais, sendo comparado com o método de cálculo direto do somatório, sem o algoritmo. As tabelas apresentadas mostram que os resultados foram amplamente favoráveis ao algoritmo.

Na medida em que o número de divisões do intervalo de integração cresce, cresce o erro do método direto, enquanto não se chega a perceber a queda da precisão do cálculo usando-se o algoritmo, mesmo quando é aplicado a dezenas de bilhões de divisões.

Dessa forma, o algoritmo viabiliza o cálculo numérico da integral com precisão crescente, tendo em vista que a precisão não cai com o aumento de **n**.

O mesmo algoritmo foi aplicado a Integral Dupla, ocorrendo o mesmo fato, isto é, enquanto cai a precisão do cálculo sem o uso do método, ela aumenta permanentemente, com o algoritmo sugerido, até mesmo para número de divisões da ordem de centenas de milhares, casos em que a queda de precisão do método direto é já considerável.

O algoritmo foi acelerado, levando-se me conta a lógica embutida na sistemática do próprio algoritmo, evitando-se a necessidade de **IF's** que aparecem na lógica do algoritmo. Com isso, sendo **n** uma potência de dois, o método pode ser muito acelerado, como indicaram os resultados, também tabelados. Em vez de serem feitos **IF's,** foram desenvolvidos loops dentro de loops, aumentando-se mais de duas vezes a velocidade do algoritmo.

Observação final a ser feita é a de que, havendo um somatório a ser calculado, com número muito elevado de parcelas e que precisa ser avaliado com precisão muito crítica, recomenda-se o algoritmo apresentado.

Em decorrência, como trabalho complementar, sugiro a implementação deste algoritmo nos casos em que a limitação é fruto da falta de precisão do somatório, ocasionando a dificuldade em aumentar o número de parcelas. Essa limitação, o algoritmo supera.

Adicionalmente, nesta monografia foi apresentado, de maneira sistemática, como funciona o ponto flutuante, com ênfase na norma do IEEE, aprovada em 1985. Este ponto foi enfatizado, pelo fato de que algumas sutilezas de seu funcionamento são ainda muito desconhecidas, mesmo de usuários de Análise Numérica.

Em síntese, a aplicação do algoritmo proposto viabiliza o aumento da precisão de cálculo, toda vez que os erros em somatórios são a limitação encontrada. Isto é exatamente o que acontece quando se calcula numericamente integrais definidas.

REFERÊNCIAS

ABNT. *NBR 6023. Referências – Elaboração*. Rio de Janeiro, ago. 2002.

——— *NBR 14724 – Trabalhos acadêmicos – Apresentação*. Rio de Janeiro, ago. 2002.

BURDEN, Richard; FAIRES, Douglas. *Análise numérica*. São Paulo: Thomson, 2003.

BUSH, Bruce M.. *The perils of floating point.* [S.l.]: Lahey Computer Systems, 1996.

CURTIS, Gerald; WHEATLEY, Patrick. *Applied numerical analysis*. New York: Addison Wesley, 1992.

DAWSON, Bruce. *Comparing floating point numbers.* Disponível em: <http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm>. Acesso em 31jan.2006.

GOLDBERG, David. *What every computer scientist should know about floating point arithmetic* . Mountain View: Association for Computing Machinery, 1991.

HOLLASCH, Steve. *IEEE Standard 754 floating point numbers.* Disponível em: <http://support.intel.com/products/processor/pentium4/product-brief/index.htm#fpu>. Acesso em 31jan2006.

IEEE 754. *Standard for binary floating point arithmetic*. Disponível em: <http://grouper.ieee.org/groups/754/>. Acesso em 31jan.2006.

IEEE. Fundamentals of computer science II. *Floating point representation of real numbers*. Disponível em: <http://www.math.grin.edu/~stone/courses/fundamentals/IEEE-reals.html>. Acesso em 31jan.2006.

INTEL. *Itanium Processor floating point software assistance and floating point exception handling*. 2000.

INTEL. *The art of assembly language programming* . Disponível em: <http://www.arl.wustl.edu/~lockwood/class/cs306/books/artofasm/Chapter_14/CH14->. Acesso em 2fev.2006.

JUFFA, Norbert; BEEBE Nelson H. F.. *Bibliography of material on floating point arithmetic*. Disponível em: <http://liinwww.ira.uka.de/bibliography/Math/fparith.html>. Acesso em 31jan.2006.

KAHAN, William. *IEEE Standard 754 for binary floating point arithmetic*. Disponível em: <http://cch.loria.fr/documentation/IEEE754/>. Acesso em 31jan.2006.

———. *Kahan summation algorithm* . Disponível em: <http://en.wikipedia.org/wiki/Kahan_summation_algorithm>. Acesso em 31jan.2006.

———. *Lecture notes on the status of IEEE Standard 754 for binary floating point arithmetic.* Berkeley : Un. of California, 1997.

KHATIB, Jamil. *Introduction to floating point calculations and IEEE 754 Standard*. Disponível em: <http://www.geocities.com/SiliconValley/Pines/6639/docs/fp_summary.html>. Acesso em 31jan.2006.

OLIVEIRA, Raymundo. *Noções de ponto flutuante*. Disponível em : <http://raymundo.oliveira.eng.br>. Acesso em 28nov.2005.

PITTSBURGH Supercomputing Center. *The IEEE Standard for floating point arithmetic*. Disponível em: <http://www.psc.edu/general/software/packages/ieee/ieee.html>. Acesso em 2fev.2006.

PYTHON TUTORIAL. *Floating point arithmetic: issues and limitations*. Disponível em: <http://docs.python.org/tut/node16.html>. Acesso em 31jan.2006.

SKEEL, R.D. e KEIPER,J.B. *Elementary numerical computing with mathematica*. Illinois: Mc Graw Hill,1993

WIKIPEDIA, the free encyclopedia . *Floating point*. Acesso em 31jan.2006.

WOLFRAM, Stephen, *The Mathematica book*. Champaign: Cambridge Univ. Press, 1996.

ZACHARY, Joseph L. *Introduction to scientific programming*: c*omputational problem solving using: C (Floating-point number tutorial).* Disponível em: <http://www.cs.utah.edu/~zachary/isp/resources.html>. Acesso em 11abril2006.

APÊNDICE A - Potência de dois que altera e a que não altera um número

PROGRAMA FONTE

```
unit menor_unit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    btnMenor: TButton;
    edtH1: TEdit;
    Label1: TLabel;
    edtEntrada: TEdit;
    Label2: TLabel;
    edtB: TEdit;
    Label4: TLabel;
    edtB1: TEdit;
    Label5: TLabel;
    edtH2: TEdit;
    Label6: TLabel;
    edte: TEdit;
    Label7: TLabel;
    edtMensagem: TEdit;
    edtH: TEdit;
    Label9: TLabel;
    procedure btnMenorClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.btnMenorClick(Sender: TObject);
var
A,B,B1,H,e,H1,H2,M:single;
I:Integer;
QG:string;
arquivo:textfile;
begin
AssignFile(arquivo,'C:\Tese mestrado\Menor 2\dadosmenor.txt');
{$I-}
reset(arquivo);
{$I+}
I := IOResult;
```

```
If I = 2 then Rewrite(ARQUIVO)
         else APPEND(ARQUIVO);
e := 1024.*1024.*8. ;
e:= 1.0/e;
QG := FloatToStrF(e,ffExponent,18,3);
edte.text := QG;
A := StrToFloat(edtEntrada.text);
H:=0.5;
repeat
H := H/2.0;
B := A + H;
until B = A;
QG := FloatToStrF(H,ffExponent,11,3);
edtH.text := QG;
QG := FloatToStrF(B,ffExponent,18,3);
edtB.text := QG;
H1 := H*1.5;
B1 := A + H1;
QG := FloatToStrF(B1,ffExponent,18,3);
edtB1.text := QG;
IF B1 = A then Begin
edtMensagem.text := 'Tou certo ';
H1 := H*(2.0 -1.0*e);
H2 := H*2.0;   end
else begin
edtMensagem.text := 'Tou errado ';
H1 := H;
H2 := H*(1.0 + 1.0*e);
     end;

{Writeln(arquivo,' Valor de A ',A:15:10,' Valor de B ',B:15:10,
' Maior valor de H1 que não altera A -->',H1); }

{Writeln(arquivo,' Valor de A ',A:15:10,' Valor de B ',B:15:10,
' Maior valor de H que não altera A -->',H);}
QG := FloatToStrF(H1,ffExponent,11,3);
edtH1.text := QG;
QG := FloatToStrF(H2,ffExponent,11,3);
edtH2.text := QG;
Writeln(arquivo);
CloseFile(arquivo);

end;

end.
```

APÊNDICE B - Integral por três métodos

PROGRAMA FONTE

```
unit Integral;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    edtNumDiv: TEdit;
    edtIntegral: TEdit;
    Button1: TButton;
    Label3: TLabel;
    edtErro: TEdit;
    Label4: TLabel;
    Label5: TLabel;
    edtVINTEGRAL: TEdit;
    edtVERRO: TEdit;
    edtIntegral2: TEdit;
    edtErro2: TEdit;
    Label6: TLabel;
    Label7: TLabel;
    edtExp: TEdit;
    Label8: TLabel;
    EdtX1: TEdit;
    EdtX2: TEdit;
    EdtX3: TEdit;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

function FUNCAO(X:single):single;
begin
     result := 1/((1.0+X)*(1.0+X)); { RAY RAY }
end;
procedure TForm1.Button1Click(Sender: TObject);
var
I,IQG,IQG2,N,NN,NS2:int64;
VI,NINT,J:integer;
```

```
X,X2,VVX,Y,Y2,H,SOMA,SOMA2,SOMA4,SOMA_2,SOMA2_2,SOMA4_2, SOMA44, XQG,
SOMA44_2,XQG2, YQG,YQG2,
ERRO,ERRO2,VERRO,VS2,VS4,VSOMA,PI:single;
VSOMA2,VSOMA4,VX: ARRAY[0..100] OF single;
QG,VOLTA,TEMPOINICIO,TEMPOFIM:string;
arquivo:textfile;
DateTime:TDateTime;
begin
DateTime:= Time;
TEMPOINICIO := TimeToStr(DateTime);
AssignFile(arquivo,'C:\TESE MESTRADO\resumo_t\resumo.txt');
{$I-}
reset(arquivo);
{$I+}
I := IOResult;
IF I = 2 then Rewrite (arquivo)
else append (arquivo);

NINT := StrToInt(edtNumDiv.text);
N := INT64(NINT)*1000;
NN := N;
IQG := 0; IQG2 :=0;
XQG:=1.44;XQG2 := 1.55; YQG:=2.44;YQG2:=2.55;
{ PI:=3.14159265358979323846264338327795; }
J := 0;
repeat
INC(J);
N := N*10;
 NS2 := N div 2;
H := (1.0 - 0.0)/N; X := 0.0;
SOMA := FUNCAO(0); SOMA2 := 0.0; SOMA4 := 0.0;
SOMA_2 := FUNCAO(0); SOMA2_2 := 0.0; SOMA4_2 := 0.0;
FOR VI := 0 TO 100 do BEGIN
VSOMA2[VI] := 0.0;
VSOMA4[VI] := 0.0;
VX[VI] := 0.0;  END;
VSOMA2[1] := SOMA/2.0;
I := 0;
repeat

INC(I);
    X := X + H; X2 := (2*I - 1)*H;Y := FUNCAO(X);Y2 := FUNCAO(X2);
    SOMA44 := SOMA4 + Y;  SOMA44_2 := SOMA4_2 + Y2;
    IF (SOMA4 = SOMA44) and (IQG = 0) then begin
    IQG := 1;
    XQG := X;
    YQG := Y;
                                        end;
                                        SOMA4 := SOMA44;
    IF (SOMA44_2 = SOMA4_2) and (IQG2 = 0) then begin
    IQG2 := 1;
    XQG2 := X2;
    YQG2 := Y2;
                                        end;
                                        SOMA4_2 := SOMA44_2;

    VX[0]:=H;
    VOLTA := 'S'; VI := 0;
    repeat
    INC(VI);
    IF VX[VI] = 0 THEN BEGIN
```

```
                        VX[VI] := VX[VI-1];
                        VX[VI-1] := 0;
                        VOLTA := 'N';
                        END
                   ELSE BEGIN
                        VX[VI] := VX[VI]+ VX[VI-1];
                        VX[VI-1] := 0;
                        END;
until VOLTA = 'N';
VVX := 0.0;
FOR VI := 1 to 100 do
VVX := VVX + VX[VI];
VSOMA4[0] := FUNCAO(VVX);
VOLTA := 'S'; VI := 0;
repeat
INC(VI);
IF VSOMA4[VI] = 0 THEN BEGIN
                        VSOMA4[VI] := VSOMA4[VI-1];
                        VSOMA4[VI-1] := 0;
                        VOLTA := 'N';
                        END
                   ELSE BEGIN
                        VSOMA4[VI] := VSOMA4[VI]+ VSOMA4[VI-1];
                        VSOMA4[VI-1] := 0;
                        END;
until VOLTA = 'N';

X := X + H; X2 := 2*I*H; Y := FUNCAO(X); Y2 := FUNCAO(X2);
SOMA2 := SOMA2 + Y; SOMA2_2 := SOMA2_2 + Y2;
VX[0]:=H;
VOLTA := 'S'; VI := 0;
repeat
INC(VI);
IF VX[VI] = 0 THEN BEGIN
                        VX[VI] := VX[VI-1];
                        VX[VI-1] := 0;
                        VOLTA := 'N';
                        END
                   ELSE BEGIN
                        VX[VI] := VX[VI]+ VX[VI-1];
                        VX[VI-1] := 0;
                        END;
until VOLTA = 'N';
VVX := 0.0;
FOR VI := 1 to 100 do
VVX := VVX + VX[VI];
VSOMA2[0] := FUNCAO(VVX);
VOLTA := 'S'; VI := 0;
repeat
INC(VI);
IF VSOMA2[VI] = 0 THEN BEGIN
                        VSOMA2[VI] := VSOMA2[VI-1];
                        VSOMA2[VI-1] := 0;
                        VOLTA := 'N';
                        END
                   ELSE BEGIN
                        VSOMA2[VI] := VSOMA2[VI]+ VSOMA2[VI-1];
                        VSOMA2[VI-1] := 0;
                        END;
until VOLTA = 'N';
```

```
 until I = NS2;

VSOMA2[1] := VSOMA2[1]-FUNCAO(VVX)/2.0;
SOMA := (SOMA + 4.0*SOMA4 + 2.0*SOMA2 - Y)*H/3.0;
SOMA_2 := (SOMA_2 + 4.0*SOMA4_2 + 2.0*SOMA2_2 - Y2)*H/3.0;
VS4 := 0.0; VS2 := 0.0;
FOR VI := 1 to 100 do BEGIN
VS4 := VS4 + VSOMA4[VI];
VS2 := VS2 + VSOMA2[VI];
                    END;
VSOMA := ( 4.0 * VS4 + 2.0 * VS2 )*H/3.0;
writeln(arquivo,' Data -> 11/02/06 ',' Reais single *',' Final de X --
> ',X:20:15);
ERRO := 0.5 - SOMA;
ERRO2 := 0.5 - SOMA_2;
VERRO := 0.5 - VSOMA;
writeln(arquivo,' Número de Divisões --> ',N:20);
writeln(arquivo,' Integral 1 --> ',SOMA,' Erro 1 --> ',ERRO);
writeln(arquivo,' Integral 2 --> ',SOMA_2,' Erro 2 --> ',ERRO2);
writeln(arquivo,' Integral 3 --> ',VSOMA,' Erro 3 --> ',VERRO);
writeln(arquivo,' XQG = ',XQG:20:15,' YQG = ',YQG:20:15);
writeln(arquivo,' XQG2 = ',XQG2:20:15,' YQG2 = ',YQG2:20:15);
until J = 3;
DateTime:= Time;
TEMPOFIM := TimeToStr(DateTime);
writeln(arquivo,'Inicio --> ',TEMPOINICIO,'  Fim --> ',TEMPOFIM);
writeln(arquivo);
closefile(arquivo);


end;

end.
```

APÊNDICE C - Cálculo de integral definida usando o algoritmo proposto

PROGRAMA FONTE

```
unit Integral;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    edtNumDiv: TEdit;
    Button1: TButton;
    Label4: TLabel;
    Label5: TLabel;
    edtVINTEGRAL: TEdit;
    edtVERRO: TEdit;
    edtverdadeiro: TEdit;
    Label2: TLabel;
    edtA: TEdit;
    Label3: TLabel;
    edtB: TEdit;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

function FUNCAO(X:single):single; {acelera}
begin
     result := 1/((1.0+X)*(1.0+X));
end;
procedure TForm1.Button1Click(Sender: TObject);
var
I,N,NS2,NN:int64;
VI,NINT,J,JJ,I1,I2,I3,I4,I5,I6,I7,I8,I9,I10,I11,
I12,I13,I14,I15,I16,I17,I18,I19,I20,I21,
I22,I23,I24,I25,I26,I27,I28,I29,I30,I31,I32,I33:integer;
X2,H,VERRO,VS2,VS4,VSOMA,PI,A,B,Verdadeiro:single;
VSOMA2,VSOMA4: ARRAY[0..100] OF single;
VOLTA,TEMPOINICIO1,TEMPOFIM1,TEMPOINICIO2,TEMPOFIM2:string;
arquivo:textfile;
DateTime:TDateTime;
begin
```

```
AssignFile(arquivo,'C:\tese mestrado\acelera\acelera 2\resumo.txt');
{$I-}
reset(arquivo);
{$I+}
I := IOResult;
IF I = 2 then Rewrite (arquivo)
else append (arquivo);


NINT := StrToInt(edtNumDiv.text);
N := INT64(NINT)*4096;
N := 2147483648;
N := 4294967296;
NN := N;
A := StrToFloat(edtA.text);
B := StrToFloat(edtB.text);
Verdadeiro := StrToFloat(edtVerdadeiro.text);
{ PI:=3.14159265358979323846264338327795; }  {


DateTime:= Time;
TEMPOINICIO1 := TimeToStr(DateTime);
J := 0;
repeat
INC(J);

 NS2 := N div 2;
H := (B - A)/N;
FOR VI := 0 TO 100 do BEGIN
VSOMA2[VI] := 0.0;
VSOMA4[VI] := 0.0;
END;
VSOMA2[1] := FUNCAO(A)/2.0;
I := 0;
repeat

INC(I);
    X2 := A + (2*I - 1)*H;
    VSOMA4[0] := FUNCAO(X2);
    VOLTA := 'S'; VI := 0;
    repeat
    INC(VI);
    IF VSOMA4[VI] = 0 THEN BEGIN
                           VSOMA4[VI] := VSOMA4[VI-1];
                           VSOMA4[VI-1] := 0.0;
                           VOLTA := 'N';
                           END
                      ELSE BEGIN
                           VSOMA4[VI] := VSOMA4[VI]+ VSOMA4[VI-1];
                           VSOMA4[VI-1] := 0.0;
                           END;
    until VOLTA = 'N';

    X2 := A + 2*I*H;
    VSOMA2[0] := FUNCAO(X2);
    VOLTA := 'S'; VI := 0;
    repeat
    INC(VI);
    IF VSOMA2[VI] = 0 THEN BEGIN
                           VSOMA2[VI] := VSOMA2[VI-1];
```

```
                          VSOMA2[VI-1] := 0.0;
                          VOLTA := 'N';
                          END
                     ELSE BEGIN
                          VSOMA2[VI] := VSOMA2[VI]+ VSOMA2[VI-1];
                          VSOMA2[VI-1] := 0.0;
                          END;
    until VOLTA = 'N';

 until I = NS2;

VSOMA2[1] := VSOMA2[1]-FUNCAO(B)/2.0;
VS4 := 0.0; VS2 := 0.0;
FOR VI := 1 to 100 do BEGIN
VS4 := VS4 + VSOMA4[VI];
VS2 := VS2 + VSOMA2[VI];
                    END;
VSOMA := ( 4.0 * VS4 + 2.0 * VS2 )*H/3.0;
VERRO := Verdadeiro - VSOMA;
writeln(arquivo,' Número de Divisões --> ',N:20);
writeln(arquivo,' Integral  --> ',VSOMA,' Erro  --> ',VERRO);
until J = 1;

DateTime:= Time;
TEMPOFIM1 := TimeToStr(DateTime);
writeln(arquivo,'Inicio --> ',TEMPOINICIO1,'  Fim --> ',TEMPOFIM1);
writeln(arquivo); }

DateTime:= Time;
TEMPOINICIO2 := TimeToStr(DateTime);
N := NN;
J := 0;
repeat
INC(J);
{ N := N*5; }
 NS2 := N div 2;
H := (B - A)/N;
FOR VI := 0 TO 100 do BEGIN
VSOMA2[VI] := 0.0;
VSOMA4[VI] := 0.0;
                    END;
I :=0;
{    VSOMA4[33]:=0.0;
    VSOMA2[33]:=0.0;
FOR I33 := 0 to 1 do                          BEGIN }
 {   VSOMA4[32] := 0.0;
    VSOMA2[32] := 0.0;
FOR I32 := 2*I33 to 2*I33+1 do                Begin }
   VSOMA4[31] := 0.0;
    VSOMA2[31] := 0.0;
FOR I31 := {2*I32 to 2*I32+1 } 0 to 1  do          Begin
    VSOMA4[30] := 0.0;
    VSOMA2[30] := 0.0;
FOR I30 := 2*I31 to 2*I31+1  do               Begin
    VSOMA4[29] := 0.0;
    VSOMA2[29] := 0.0;
FOR I29 := 2*I30 to 2*I30+1   do            Begin
    VSOMA4[28] := 0.0;
    VSOMA2[28] := 0.0;
FOR I28 := 2*I29 to 2*I29+1 do              Begin
    VSOMA4[27] := 0.0;
```

```
    VSOMA2[27] := 0.0;
FOR I27 := 2*I28 to 2*I28+1 do             Begin
    VSOMA4[26] := 0.0;
    VSOMA2[26] := 0.0;
FOR I26 := 2*I27 to 2*I27+1 do           Begin
    VSOMA4[25] := 0.0;
    VSOMA2[25] := 0.0;
FOR I25 := 2*I26 to 2*I26+1 do         Begin
    VSOMA4[24] := 0.0;
    VSOMA2[24] := 0.0;
FOR I24 := 2*I25 to 2*I25+1 do       Begin
    VSOMA4[23] := 0.0;
    VSOMA2[23] := 0.0;
FOR I23 := 2*I24 to 2*I24+1 do     Begin
    VSOMA4[22] := 0.0;
    VSOMA2[22] := 0.0;
FOR I22 := 2*I23 to 2*I23+1 do Begin
    VSOMA4[21]:=0.0;
    VSOMA2[21]:=0.0;
FOR I21 := 2*I22 to 2*I22+1 do                           Begin
    VSOMA4[20] := 0.0;
    VSOMA2[20] := 0.0;
FOR I20 := 2*I21 to 2*I21+1 do               Begin
    VSOMA4[19] := 0.0;
    VSOMA2[19] := 0.0;
FOR I19 := 2*I20 to 2*I20+1 do              Begin
    VSOMA4[18] := 0.0;
    VSOMA2[18] := 0.0;
FOR I18 := 2*I19 to 2*I19+1 do              Begin
    VSOMA4[17] := 0.0;
    VSOMA2[17] := 0.0;
FOR I17 := 2*I18 to 2*I18+1 do             Begin
    VSOMA4[16] := 0.0;
    VSOMA2[16] := 0.0;
FOR I16 := 2*I17 to 2*I17+1 do           Begin
    VSOMA4[15] := 0.0;
    VSOMA2[15] := 0.0;
FOR I15 := 2*I16 to 2*I16+1 do         Begin
    VSOMA4[14] := 0.0;
    VSOMA2[14] := 0.0;
FOR I14 := 2*I15 to 2*I15+1 do       Begin
    VSOMA4[13] := 0.0;
    VSOMA2[13] := 0.0;
FOR I13 := 2*I14 to 2*I14+1 do     Begin
    VSOMA4[12] := 0.0;
    VSOMA2[12] := 0.0;
FOR I12 := 2*I13 to 2*I13+1 do   Begin
    VSOMA4[11]:=0.0;
    VSOMA2[11]:=0.0;
FOR I11 := 2*I12 to 2*I12+1 do                           Begin
    VSOMA4[10] := 0.0;
    VSOMA2[10] := 0.0;
FOR I10 := 2*I11 to 2*I11+1 do               Begin
    VSOMA4[9] := 0.0;
    VSOMA2[9] := 0.0;
FOR I9 := 2*I10 to 2*I10+1 do              Begin
    VSOMA4[8] := 0.0;
    VSOMA2[8] := 0.0;
FOR I8 := 2*I9 to 2*I9+1 do              Begin
    VSOMA4[7] := 0.0;
    VSOMA2[7] := 0.0;
```

```
FOR I7 := 2*I8 to 2*I8+1 do             Begin
    VSOMA4[6] := 0.0;
    VSOMA2[6] := 0.0;
FOR I6 := 2*I7 to 2*I7+1 do           Begin
    VSOMA4[5] := 0.0;
    VSOMA2[5] := 0.0;
FOR I5 := 2*I6 to 2*I6+1 do         Begin
    VSOMA4[4] := 0.0;
    VSOMA2[4] := 0.0;
FOR I4 := 2*I5 to 2*I5+1 do       Begin
    VSOMA4[3] := 0.0;
    VSOMA2[3] := 0.0;
FOR I3 := 2*I4 to 2*I4+1 do     Begin
    VSOMA4[2] := 0.0;
    VSOMA2[2] := 0.0;
FOR I2 := 2*I3 to 2*I3+1 do   Begin
    VSOMA4[1] := 0.0;
    VSOMA2[1] := 0.0;
FOR I1 := 2*I2 to 2*I2+1 do Begin
    INC(I);
    X2 := A + (2*I - 1)*H;
    VSOMA4[1] := VSOMA4[1]+ FUNCAO(X2);
    X2 := A + 2*I*H;
    VSOMA2[1] := VSOMA2[1]+ FUNCAO(X2);
                           end;
    VSOMA4[2] := VSOMA4[2]+ VSOMA4[1];
    VSOMA2[2] := VSOMA2[2]+ VSOMA2[1];
                              end;
    VSOMA4[3] := VSOMA4[3]+ VSOMA4[2];
    VSOMA2[3] := VSOMA2[3]+ VSOMA2[2];
                                 end;
    VSOMA4[4] := VSOMA4[4]+ VSOMA4[3];
    VSOMA2[4] := VSOMA2[4]+ VSOMA2[3];
                                   end;
    VSOMA4[5] := VSOMA4[5]+ VSOMA4[4];
    VSOMA2[5] := VSOMA2[5]+ VSOMA2[4];
                                   end;
    VSOMA4[6] := VSOMA4[6]+ VSOMA4[5];
    VSOMA2[6] := VSOMA2[6]+ VSOMA2[5];
                                   end;
    VSOMA4[7] := VSOMA4[7]+ VSOMA4[6];
    VSOMA2[7] := VSOMA2[7]+ VSOMA2[6];
                                   end;
    VSOMA4[8] := VSOMA4[8]+ VSOMA4[7];
    VSOMA2[8] := VSOMA2[8]+ VSOMA2[7];
                                   end;
    VSOMA4[9] := VSOMA4[9]+ VSOMA4[8];
    VSOMA2[9] := VSOMA2[9]+ VSOMA2[8];
                                   end;
    VSOMA4[10] := VSOMA4[10]+ VSOMA4[9];
    VSOMA2[10] := VSOMA2[10]+ VSOMA2[9];
                                   end;
    VSOMA4[11] := VSOMA4[11]+ VSOMA4[10];
    VSOMA2[11] := VSOMA2[11]+ VSOMA2[10];
                                   end;
    VSOMA4[12] := VSOMA4[12]+ VSOMA4[11];
    VSOMA2[12] := VSOMA2[12]+ VSOMA2[11];
                                end;
    VSOMA4[13] := VSOMA4[13]+ VSOMA4[12];
    VSOMA2[13] := VSOMA2[13]+ VSOMA2[12];
                                   end;
```

```
 VSOMA4[14] := VSOMA4[14]+ VSOMA4[13];
 VSOMA2[14] := VSOMA2[14]+ VSOMA2[13];
                                  end;
 VSOMA4[15] := VSOMA4[15]+ VSOMA4[14];
 VSOMA2[15] := VSOMA2[15]+ VSOMA2[14];
                                  end;
 VSOMA4[16] := VSOMA4[16]+ VSOMA4[15];
 VSOMA2[16] := VSOMA2[16]+ VSOMA2[15];
                                  end;
 VSOMA4[17] := VSOMA4[17]+ VSOMA4[16];
 VSOMA2[17] := VSOMA2[17]+ VSOMA2[16];
                                  end;
 VSOMA4[18] := VSOMA4[18]+ VSOMA4[17];
 VSOMA2[18] := VSOMA2[18]+ VSOMA2[17];
                                  end;
 VSOMA4[19] := VSOMA4[19]+ VSOMA4[18];
 VSOMA2[19] := VSOMA2[19]+ VSOMA2[18];
                                  end;
 VSOMA4[20] := VSOMA4[20]+ VSOMA4[19];
 VSOMA2[20] := VSOMA2[20]+ VSOMA2[19];
                                  end;
 VSOMA4[21] := VSOMA4[21]+ VSOMA4[20];
 VSOMA2[21] := VSOMA2[21]+ VSOMA2[20];
                                  end;
 VSOMA4[22] := VSOMA4[22]+ VSOMA4[21];
 VSOMA2[22] := VSOMA2[22]+ VSOMA2[21];
                               end;
 VSOMA4[23] := VSOMA4[23]+ VSOMA4[22];
 VSOMA2[23] := VSOMA2[23]+ VSOMA2[22];
                                  end;
 VSOMA4[24] := VSOMA4[24]+ VSOMA4[23];
 VSOMA2[24] := VSOMA2[24]+ VSOMA2[23];
                                  end;
 VSOMA4[25] := VSOMA4[25]+ VSOMA4[24];
 VSOMA2[25] := VSOMA2[25]+ VSOMA2[24];
                                  end;
 VSOMA4[26] := VSOMA4[26]+ VSOMA4[25];
 VSOMA2[26] := VSOMA2[26]+ VSOMA2[25];
                                  end;
 VSOMA4[27] := VSOMA4[27]+ VSOMA4[26];
 VSOMA2[27] := VSOMA2[27]+ VSOMA2[26];
                                  end;
 VSOMA4[28] := VSOMA4[28]+ VSOMA4[27];
 VSOMA2[28] := VSOMA2[28]+ VSOMA2[27];
                                  end;
 VSOMA4[29] := VSOMA4[29]+ VSOMA4[28];
 VSOMA2[29] := VSOMA2[29]+ VSOMA2[28];
                                  end;
 VSOMA4[30] := VSOMA4[30]+ VSOMA4[29];
 VSOMA2[30] := VSOMA2[30]+ VSOMA2[29];
                                  end;
 VSOMA4[31] := VSOMA4[31]+ VSOMA4[30];
 VSOMA2[31] := VSOMA2[31]+ VSOMA2[30];
                                  end;
{ VSOMA4[32] := VSOMA4[32]+ VSOMA4[31];
 VSOMA2[32] := VSOMA2[32]+ VSOMA2[31];
                                  end; }
{ VSOMA4[33] := VSOMA4[33]+ VSOMA4[32];
 VSOMA2[33] := VSOMA2[33]+ VSOMA2[32];
                                  end; }
```

```
{VSOMA2[1] := VSOMA2[1]+ FUNCAO(A)/2.0 - FUNCAO(B)/2.0; }
VS4 := 0.0; VS2 := 0.0;
{
FOR VI := 1 to 100 do BEGIN
VS4 := VS4 + VSOMA4[VI];
VS2 := VS2 + VSOMA2[VI];
                    END;}
VS4 := {VSOMA4[33];} VSOMA4[29];
VS2 := {VSOMA2[33]} VSOMA2[29]+ FUNCAO(A)/2.0 - FUNCAO(B)/2.0;
VSOMA := ( 4.0 * VS4 + 2.0 * VS2 )*H/3.0;
VERRO := Verdadeiro - VSOMA;
writeln(arquivo,' Número de Divisões --> ',N:20);
writeln(arquivo,' Integral  --> ',VSOMA,' Erro  --> ',VERRO);
writeln(arquivo,' VSOMA4 e VSOMA2 ');
{FOR JJ := 1 to 21 do
writeln(arquivo,' VSOMA4[',JJ:2,']= ',VSOMA4[JJ]:20:5);
FOR JJ := 1 to 21 do
writeln(arquivo,' VSOMA2[',JJ:2,']= ',VSOMA2[JJ]:20:5); }
until J = 1;
DateTime:= Time;
TEMPOFIM2 := TimeToStr(DateTime);
writeln(arquivo,'Inicio --> ',TEMPOINICIO2,'  Fim --> ',TEMPOFIM2);

writeln(arquivo);

closefile(arquivo);


end;

end.
```

APÊNDICE D – Integral dupla usando o algoritmo

PROGRAMA FONTE

```
unit Integral;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    edtNumDiv: TEdit;
    Button1: TButton;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    edtIntegral2: TEdit;
    edtErro2: TEdit;
    edtIntegral1: TEdit;
    edtErro1: TEdit;
    edtReal: TEdit;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

function SIMPSON1(Y:single;N:integer) :single;
var
  HX,SOMAX,SOMA2X,SOMA4X,X,X0,PI :single;

  I:integer;

function FXY(X,Y :single):single;
 begin
  FXY := COS(x+y)
 end;

begin
  PI := 3.14159265358979323846264338327950288419 7;
  SOMA2X := 0; SOMA4X := 0;
  X0 := -PI/2.0;
  HX := (PI / 2.0 - X0)/N; SOMAX := FXY(X0,Y);
```

```
  for I := 1 to N div 2 do
    begin
      X := X0 + (2*I-1)*HX;
      SOMA4X := SOMA4X + FXY(X,Y);
      X := X0 + 2*I*HX;
      SOMA2X := SOMA2X + FXY(X,Y);
    end;

  result := ( SOMAX + 4.0 * SOMA4X + 2.0 * SOMA2X - FXY(X,Y) ) * HX /
3.0;
end;
 function SIMPSON2(Y:single;N:integer) :single;
var
  HX,X,X0,VS2,VS4,PI :single;
  VSOMAX2,VSOMAX4:ARRAY[0..100] OF single;
  I,VI:integer;
  VOLTA:STRING;
function FXY(X,Y :single):single;
 begin
  FXY := COS(x+y)
 end;

begin
  FOR I := 0 to 100 do begin
   VSOMAX2[I] := 0.0;
   VSOMAX4[I] := 0.0;   end;
  PI := 3.14159265358979323846264338327950288419 7;
  X0 := -PI / 2.0;
  HX := (PI / 2.0 - X0)/N; VSOMAX2[1] := FXY(X0,Y)/2.0;
  for I := 1 to N div 2 do
    begin
      X := X0 + (2*I-1)*HX;
      VSOMAX4[0]:= FXY(X,Y);
      VOLTA := 'S'; VI := 0;
      repeat
      INC(VI);
      IF VSOMAX4[VI] = 0.0 then Begin
                VSOMAX4[VI]:=VSOMAX4[VI-1];
                VSOMAX4[VI-1] := 0.0;
                VOLTA := 'N';     end
                          else Begin
                VSOMAX4[VI]:=VSOMAX4[VI]+VSOMAX4[VI-1];
                VSOMAX4[VI-1]:=0.0;
                                  end;
      until VOLTA = 'N';

      X := X0 + 2*I*HX;
      VSOMAX2[0]:= FXY(X,Y);
      VOLTA := 'S'; VI := 0;
      repeat
      INC(VI);
      IF VSOMAX2[VI] = 0.0 then Begin
                VSOMAX2[VI]:=VSOMAX2[VI-1];
                VSOMAX2[VI-1] := 0.0;
                VOLTA := 'N';     end
                          else Begin
                VSOMAX2[VI]:=VSOMAX2[VI]+VSOMAX2[VI-1];
                VSOMAX2[VI-1]:=0.0;
                                  end;
      until VOLTA = 'N';
```

```
    end;
VSOMAX2[1] := VSOMAX2[1]-FXY(X,Y)/2.0;
 VS2:=0.0; VS4:=0.0;
 FOR VI := 1 to 100 do BEGIN
 VS4 := VS4 + VSOMAX4[VI];
 VS2 := VS2 + VSOMAX2[VI];
                        end;
 result := (4.0 * VS4 + 2.0* VS2)*HX/3.0;

end;



procedure TForm1.Button1Click(Sender: TObject);
var

  HY,SOMAY,SOMA2Y,SOMA4Y,Y,Verdadeiro,Erro1,VSOMA,erro2,
  VS2,VS4,PI,YO :single;
  VSOMAY2,VSOMAY4:ARRAY[0..100] of single;
  VI,I,NUM  :integer;
  N,J,NS2:INT64;
  QG,VOLTA :string;
  arquivo:textfile;
begin
  AssignFile(arquivo,'C:\Tese Mestrado\Int
Dupla\SENXY\dadosdupla.txt');
  {$I-}
  reset(arquivo);
  {$I+}
  I := IOResult;
  IF I = 2 then Rewrite(arquivo)
           else append(arquivo);
  N:=StrToInt(edtNumDiv.text);
  PI := 3.14159265358979323846264338327950288419 7;
  {PI = 3.14159265358979323846264338327950288419 7}
  {Int COS = 7.08197870902284181018440448 8254 }
  YO := -PI/2.0;
  NUM:=0;
  repeat
  INC(NUM);
  N := N*2;
  SOMA2Y := 0.0; SOMA4Y := 0.0;
  For VI := 0 To 100 do Begin
  VSOMAY2[VI] := 0.0;
  VSOMAY4[VI] := 0.0;   end;


  NS2 := N DIV 2;
  HY := ( PI / 2.0 - YO ) / N;
  Y := YO;

  SOMAY := SIMPSON1(Y,N);
  VSOMAY2[1] := SIMPSON2(Y,N) / 2.0;
  J := 0;
  repeat
  INC(J);
      Y := YO + (2*J -1)*HY;
      SOMA4Y := SOMA4Y + SIMPSON1(Y,N); VSOMAY4[0] := SIMPSON2(Y,N);
      VOLTA := 'S'; VI :=0;
      repeat
```

```
      INC(VI);
      IF VSOMAY4[VI] = 0.0 Then Begin
                VSOMAY4[VI] := VSOMAY4[VI-1];
                VSOMAY4[VI-1] := 0.0;
                VOLTA := 'N';   end
                           Else Begin
                VSOMAY4[VI] := VSOMAY4[VI]+VSOMAY4[VI-1];
                VSOMAY4[VI-1] := 0.0;
                                end;
      until VOLTA = 'N';
      Y := YO + 2*J*HY;
      SOMA2Y := SOMA2Y + SIMPSON1(Y,N); VSOMAY2[0] := SIMPSON2(Y,N);
      VOLTA := 'S'; VI :=0;
      repeat
      INC(VI);
      IF VSOMAY2[VI] = 0.0 Then Begin
                VSOMAY2[VI] := VSOMAY2[VI-1];
                VSOMAY2[VI-1] := 0.0;
                VOLTA := 'N';   end
                           Else Begin
                VSOMAY2[VI] := VSOMAY2[VI]+VSOMAY2[VI-1];
                VSOMAY2[VI-1] := 0.0;
                                end;
      until VOLTA = 'N';

  until J = NS2;
  VSOMAY2[1]:= VSOMAY2[1]-SIMPSON2(Y,N)/2.0;
  SOMAY := (SOMAY + 4.0*SOMA4Y + 2.0* SOMA2Y- SIMPSON1(Y,N))*HY/3.0;
  VS4:=0.0; VS2:=0.0;
  FOR VI := 1 to 100 do begin
  VS4 := VS4 + VSOMAY4[VI];
  VS2 := VS2 + VSOMAY2[VI];
                        end;
  VSOMA := (4.0 * VS4 + 2.0*VS2) * HY / 3.0;
  Verdadeiro := StrToFloat(edtReal.text);
  Erro1 := Somay - Verdadeiro;
  Erro2 := VSOMA - Verdadeiro;
  QG := FloatToStrF(SOMAY,ffFixed,35,25);
  edtIntegral1.text := QG;
  QG := FloatToStrF(VSOMA,ffFixed,35,25);
  edtIntegral2.text := QG;
  QG := FloatToStrF(Erro1,ffFixed,35,25);
  edtErro1.text := QG;
  QG := FloatToStrF(Erro2,ffFixed,35,25);
  edtErro2.text := QG;
  writeln(arquivo,'Valor verdadeiro --> ??????? ',' Divisões --> ',
  N:15,' REAIS single ');
  writeln(arquivo,' Integral pelo Método I ',Somay,' erro -->
',erro1);
  writeln(arquivo,' Integral pelo Método II ',VSOMA,'  erro -->
',erro2);
  writeln(arquivo);
  until Num = 3;
  closefile(arquivo);
end; { fim da INTDUPLA }


end.
```

APÊNDICE E – Razão da perda de precisão nos somatórios

No caso em estudo, em especial quando se refere à integração numérica, nos somatórios estão sendo somados valores da função em pontos muito próximos uns dos outros. Assim, trata-se da soma de valores da mesma ordem de grandeza. É isto que está sendo confrontado com a soma de valores com ordens de grandeza muito diferentes.

A afirmação é a de que, neste caso, quando as ordens de grandezas são muito diferentes, perde-se precisão do número de menor ordem de grandeza e, em conseqüência, perde-se precisão na soma.

Vejamos como isto acontece com variáveis de precisão simples, as que têm 23 bits reservados para a mantissa.

Sejam dois números, A e B, positivos, da mesma ordem de grandeza, valores próximos um do outro e que se deseja somar ao valor A o valor de B.

Suas representações serão:

A → 1, - - - .....- - . 2^p

B → 1, - - - .....- - . 2^p

onde - - -....- - indicam os 23 bits após a vírgula.

Tendo havido arredondamento, o erro máximo de cada um será a metade da última unidade: $\boldsymbol{\varepsilon} = 0{,}5 \,.\, 2^{-23}.2^p = 2^{-24+p}$ (ULP – units in the last place).

Dessa forma, o erro relativo máximo vale: $2^{-24+p}/(1{,}0.2^p) = 2^{-24}$.

Em um número grande de parcelas, aproximadamente 50% serão erros para mais e 50% para menos.

Havendo $\boldsymbol{n}$ termos a serem somados, haverá $\boldsymbol{n/2}$ somas, das quais, aproximadamente, a metade será arredondada para cima e metade para baixo, sendo esses erros, em média, a metade do erro máximo, $\boldsymbol{\varepsilon}$, apresentado.

Independente dos erros das parcelas A e B, a soma A + B será, também, arredondada. Necessariamente, $A + B \geq 10{,}.....x\ 2^p$, logo $A + B \geq 1{,}0....\ x\ 2^{p+1}$.

O erro máximo desse arredondamento será: $\boldsymbol{\sigma} = 0{,}5 \,.\, 2^{-23}.2^{p+1} = 2^{-24+p+1}$. O erro relativo máximo vale: $2^{-24+p+1}/(1{,}0.2^{p+1}) = 2^{-24}$. Assim, o erro relativo máximo de cada soma continua valendo 2^{-24}, que era o erro máximo de cada parcela.

Isso continua valendo para a soma das somas, a soma das somas das somas, etc... Dessa forma, ao final, a integral terá aproximadamente esse mesmo erro relativo máximo.

Vejamos o que acontece quando se somam valores de ordem de grandeza diferentes.

Seja A → 1, - - - ....- - -. 2^p e B → 1, - - - ....- - -. 2^q onde p >> q.

Para ser feita a soma, é necessário desnormalizar B.

A → 1, - - - ....- - - . 2^p

B → 0,00..01- - - ....- - - . 2^p, sendo inseridos p-(q+1) zeros, antes do 1.

Ao ser executada a soma, sobrarão 23 bits após a vírgula, havendo arredondamento.

Se houver mais de 23 zeros, isto é, se p – q – 1 > 23 (p – q > 24), B será simplesmente ignorado, sendo que A + B = A.

Assim, num somatório, se a soma parcial (representada aqui pelo A → 1, - - - ....- - -.2^p) é tal que as novas parcelas (representada aqui pelo B → 1, - - - ....- - -.2^q) são tais que p – q > 24, então o erro da soma vai sendo acrescido pela soma de todas essas parcelas "abandonadas", crescendo o erro relativo de A, isto é, da soma.

Se não for esse o caso, isto é, não houver mais de 23 zeros, então serão perdidos os bits menos significativos de B, havendo arredondamento no que sobrar, para ser somado ao A.

Se p – q ≤ 24, serão abandonados (p – q) bits, com arredondamento na posição anterior ao bit mais significativo abandonado, isto é, na última posição de A. Esse arredondamento corresponde a um valor máximo de 0,5 x 2^{-23+p} , isto é, à metade da posição correspondente ao último bit de A.

Entretanto, como não se está somando um número da ordem de grandeza de A, ao ser feita a soma, não é alterado necessariamente o expoente de A, que permanece p e não p+1, como ocorria anteriormente. Assim, o erro relativo de A, isto é, da soma, cresce a cada nova parcela.

Todas estas observações visam mostrar a importância de se somarem parcelas da mesma ordem de grandeza, para evitar o crescimento do erro acumulado.

É exatamente por isso que se observou a brusca queda do valor calculado nas integrais pelos métodos tradicionais, o que foi evitado pelo algoritmo proposto.

Oliveira, Raymundo de

Algoritmo para minimizar erro em integração numérica / Raymundo Theodoro Carvalho de Oliveira. – Rio de Janeiro: UFRJ / Instituto de Matemática, 2006.

100 f.: il., 21 cm.

Dissertação (mestrado) – UFRJ / Instituto de Matemática, 2006.

Inclui bibliografia.

1. Título.I. Algoritmo.II. Ponto Flutuante. III. Somatório. IV. Integrais.