Capítulo 3

Erros, Precisão Numérica e Ponto Flutuante

No capítulo anterior introduzimos o conceito de *variável* em programação. Uma variável é basicamente um nome usado para se referir a algum conteúdo na memória do computador. Além disso, vimos que ela contém também informações sobre o *tipo de dados* e, associado a esse tipo, há um conjunto de regras de como os dados devem ser manipulados. Por exemplo, vimos que uma divisão entre dois números reais (`REAL`, `DOUBLE PRECISION`) não é o mesmo que uma divisão entre inteiros (`INTEGER`).

Na prática, variáveis podem ser muito mais complexas, mas nesse capítulo descreveremos apenas como variáveis inteiras e reais são representadas pelo computador, e discutiremos as conseqüências dessa representação para o cálculo numérico.

3.1 Representação

Computadores *não* armazenam números com precisão infinita; usam, ao contrário, alguma aproximação que pode ser reduzida em um número fixo de *bits* (*binary digits*, em inglês) ou *bytes* (grupos de 8 bits). Quase todos os computadores permitem ao usuário uma escolha entre diferentes *representações* ou *tipos de dados*. Tipos de dados podem variar no número de bits utilizados, mas, mais fundamentalmente, na forma como o número é representado. Exemplos importantes são representações de inteiros e de ponto-flutuante (*floating-point*).

3.1.1 Representação de inteiros

Em computação há dois tipos de inteiros, os inteiros positivos (*unsigned*, sem sinal) e os inteiros negativos e positivos (*signed*, com sinal).

Inteiros positivos são simplesmente o número escrito em binário. Nesse caso, dados t dígitos binários, b_i $(i = 1,\, t-1)$, a conversão entre o que está representado no computador e o sistema decimal é feita através da expressão

$$b_{t-1}2^{t-1} + b_{t-2}2^{t-2} + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0\,. \qquad (3.1)$$

O máximo inteiro positivo que pode ser representado é, portanto, $2^t - 1$. Por exemplo, com 8 bits (ou seja, 8 dígitos binários), só é possível representar os inteiros entre 0 a 255:

Decimal	Representação na memória
0	00000000
1	00000001
2	00000010
255	11111111

Os tipos mais comuns de inteiro aceitam negativos e são feitos dividindo-se os números representáveis aproximadamente à metade, com os positivos no começo e os negativos no final. Dessa forma, o maior valor negativo representável é -2^{t-1} e o maior positivo representável é $2^{t-1} - 1$. Por exemplo, com 8 bits, só é possível representar os números inteiros entre -128 e $+127$.

Decimal	Representação na memória
0	00000000
1	00000001
127	01111111
-128	10000000
-127	10000001
-126	10000010
-2	11111110
-1	11111111

Em Fortran os inteiros são em geral com sinal, mas existem algumas implementações que permitem a declaração de inteiros sem sinal. Tipicamente, um `INTEGER` é representado com 32 bits (4 palavras de 1 byte) e um `INTEGER*8` é representado com 64 bits. Assim, o maior valor de um `INTEGER` é $2^{31} - 1 = 2.147.483.647$ e o maior valor de um `INTEGER*8` é $2^{63} - 1 = 9.223.372.036.854.775.807$.

Vale lembrar que uma representação inteira é sempre exata. Expressões aritméticas envolvendo inteiros são também exatas, desde que

1. a resposta não esteja fora do intervalo representável, e,

2. divisão seja interpretada como uma *divisão inteira* que produz um resultado inteiro, desconsiderando qualquer resto.

O que ocorre quando se tenta atribuir a uma variável um inteiro fora do intervalo representável? O resultado pode ser um *overflow* ou um *rollover*. Por exemplo, se tentarmos compilar um código em Fortran que contenha a linha:

```
PRINT*, HUGE(i)+10
```

provavelmente teremos um erro de compilação (`arithmetic overflow`). Entretanto, o código abaixo não resultará em erro de compilação, e provavelmente o que o programa fará durante a execução é um *rollover*, ou seja, algo parecido com o que ocorre com um odômetro de carro quando ele atinge a máxima quilometragem. O código abaixo retorna o valor -2147483639.

```
i = HUGE(i)
PRINT*, i+10
```

Não considerar o tamanho dos números inteiros é um erro comum e com resultados em geral muito graves. Por exemplo, o código abaixo vai retornar um número negativo para o quadrado de um inteiro:

```
i = 50000
PRINT*, i**2
```

3.1.2 Representação de ponto-flutuante

Para representar números reais com o maior intervalo possível adota-se a chamada *representação de ponto-flutuante*. Nesta representação os números são armazenados com um número (aproximadamente) fixo de algarismos significativos e são escalados para cima ou para baixo usando-se um expoente. O termo ponto-flutuante refere-se ao fato de o ponto decimal (ou no caso de computadores, ponto binário) poder ”flutuar”, isto é, poder ser colocado em qualquer lugar relativo aos algarismos significativos do número. Isso é equivalente à chamada notação científica, que representa um número como a multiplicação de outro número e uma potência de 10.

Em ponto-flutuante, um número é representado internamente através de um bit de sinal, S (interpretado como mais ou menos), um expoente inteiro exato, E, e uma mantissa binária, M. Juntos, eles representam o número

$$S \times M \times 2^{E-e} \tag{3.2}$$

onde e é um viés do expoente, uma constante inteira fixa que depende da máquina usada e da representação implementada. Outra maneira de se escrever a representação de um numero real x com m dígitos binários na mantissa é

$$x = \pm .b_1 b_2 b_3 \dots b_m \times 2^{E-e} \tag{3.3}$$

Note que o valor de b_1 é sempre não nulo. Tal representação, dita normalizada, maximiza o número de algarismos significativos e portanto a precisão do número representado.

Vejamos dois exemplos de números binários normalizados usando-se $m = 8$:

- $x_1 = 0.11100110 \times 2^2$.
  O correspondente na base 10 desde número é dado por:

$$x_1 = \left[2^{-1} + 2^{-2} + 2^{-3} + 2^{-6} + 2^{-7}\right] \times 2^2 = 3.59375.$$

- $x_2 = 0.11100111 \times 2^2$. O correspondente na base 10 desde número é dado por:

$$x_2 = \left[2^{-1} + 2^{-2} + 2^{-3} + 2^{-6} + 2^{-7} + 2^{-8}\right] \times 2^2 = 3.609375.$$

Note que, no sistema de representação usado, x_1 e x_2 são dois números consecutivos, isto é, não se pode representar nenhum outro número real entre x_1 e x_2. Por exemplo, o decimal 3.6 não teria representação exata nesse sistema. Essa é a origem de um tipo de erro em cálculo chamado *erro de representação*; esse erro é inerente ao sistema de representação adotado.

Em um computador moderno, os 32 bits correspondentes a uma variável tipo `REAL` são divididos em três partes: 1 bit para o sinal, 8 bits para o expoente e 23 bits para a mantissa, da seguinte forma:

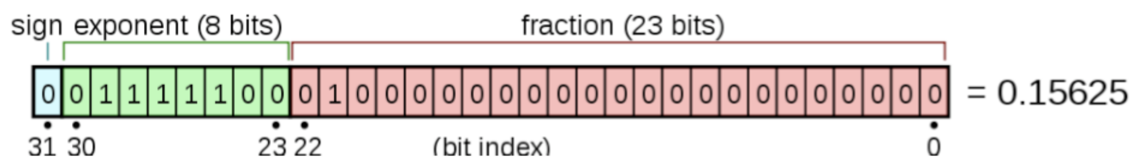


Figura 3.1: Distribuições dos bits em uma representação de ponto-flutuante

Nestes computadores, o termo *precisão simples* (single precision) corresponde a números com 32 bits (ou 4 bytes) e o termo *precisão dupla* (double precision) denota números com 64bits (ou 8 bytes). A tabela abaixo elenca algumas propriedades dos números de 32 e 64 bits.

A utilização de bits conforme a tabela acima permite a representação de números com os seguintes limites:

Precisão	simples	simples	dupla	dupla
Relação ao zero	mais distante	mais próximo	mais distante	mais próximo
Cálculo	$\pm(1-2^{-24})\times 2^{128}$	$\pm(1-2^{-24})\times 2^{-126}$	$\pm(1-2^{-53})\times 2^{1024}$	$\pm(1-2^{-53})\times 2^{-1022}$
Valor	$\pm 3,40\times 10^{38}$	$\pm 1,17\times 10^{-38}$	$\pm 1,79\times 10^{308}$	$\pm 2,22\times 10^{-308}$

Além dos números reais, o sistema de ponto-flutuante reservou alguns números especiais: +0.0, -0.0, $+\infty$, $-\infty$, e `NaN` (*not a number*, na sigla em inglês). Esses números especiais permitem que o processador lide com as chamadas exceções de ponto flutuante. Por exemplo:

```
x = HUGE(x)
PRINT*, x*2. !retorna +infinito
x = -1.
PRINT*, SQRT(x) retorna NaN
```

A ocorrência de `NaN` durante a execução é potencialmente grave, pois ela pode se propagar a outras variáveis. Isso ocorre porque:

- `NaN`*número = `NaN`

- `NaN`¡ número = `.FALSE.`

- `NaN` ¿ número = `.FALSE.`

- `NaN` == `NaN` = `.FALSE.`

Uma fonte interessante de informação sobre ponto-flutuantes pode ser encontrada na Wikipedia[1] sob os termos *IEEE 754-1985* ou *IEEE floating point arithmetic*.

3.2 Erros numéricos devido ao uso de computador digital

Os erros de representação, oriundo do fato de que nem todo número real pode ser representado exatamente no sistema binário, já foram descritos acima. Veremos agora os erros de arredondamento e truncamento.

[1] *http://www.wikipedia.org*

3.2.1 Erros de arredondamento

A aritmética entre dois números na representação de ponto-flutuante frequentemente não é exata, mesmo que os operandos possam ser representados exatamente. Para compreender isso é necessário considerar a maneira como dois números são somados. Isso é feito em dois passos: primeiro desloca-se para a direita a mantissa do menor número, dividindo-a por 2, enquanto que o expoente é aumentado, até que os dois operandos tenham o mesmo expoente, quando então os números podem ser somados. Esse procedimento descarta os algarismos menos significativos do menor operando, o que pode resultar em uma adição inexata.

Por exemplo, para somar os seguintes números binários normalizados em um sistema de ponto-flutuante que tem $m = 8$:

$$x_1 = .11100110 \times 2^5$$
$$x_2 = .11001111 \times 2^2$$

Desloca-se a mantissa de x_2:

$$x_2 = .01100111\mathbf{1} \times 2^3$$
$$x_2 = .00110011\mathbf{11} \times 2^4$$
$$x_2 = .00011001\mathbf{111} \times 2^5$$

e então faz-se a soma. Note que algarismos marcados em negrito serão descartados no processo!

O menor número que adicionado ao 1.0 produz um número diferente de 1.0 é chamado de precisão da máquina, ϵ_m, e é fornecido pela função intrínseca do Fortran `EPSILON`. Em outras palavras, o ϵ_m é a precisão relativa (ver abaixo) na qual números de ponto-flutuante podem ser representados, e corresponde a uma mudança do último dígito da mantissa. Praticamente toda operação aritmética em ponto-flutuante introduz um erro fracional de ao menos ϵ_m. Esse tipo de erro é chamado de erro de arredondamento (*roundoff*).

Erros de arredondamento se acumulam quando uma longa série de cálculos é feita. Se, para se obter um resultado, foram feitas N operações aritméticas, em geral o *mínimo* erro de arredondamento será da ordem de $\sqrt{N}\epsilon_m$ caso os erros ocorram de forma aleatória (a $\sqrt{N}$ vem do caminho aleatório). Se houver uma regularidade no sinal de ϵ_m, então o erro total será da ordem de $N\epsilon_m$.

Além disso, algumas ocasiões especialmente problemáticas favorecerão erros de arredondamento em uma simples operação. Em geral elas estão associadas à subtração de dois números muito próximos, dando um resultado cujos algarismos significativos são aqueles poucos algarismos em que os números diferiam.

Por exemplo, considere a expressão familiar para a solução da equação quadrática (Fórmula de Bhaskara)

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{3.4}$$

A adição do numerador torna-se problemática sempre que $b > 0$ e $|4ac| \ll b^2$, pois nesse caso haverá uma subtração de dois números muito próximos. A solução para esse problema em particular é simples, e está mostrada no programa `cap03_bhaskara.f90`.

Vamos denotar $\oplus$, $\ominus$, $\otimes$ e $\oslash$ as operações em ponto-flutuante. Em geral:

$$x \oplus y \neq x + y$$
$$x \otimes y \neq x \times y$$

Da mesma forma a associatividade e a distributiva também não são sempre válidas:

$$x \otimes (y + z) \neq x \otimes y + x \otimes z$$

3.2.2 Erros de truncamento

Erros de arredondamento são uma característica do hardware do computador. Existem um outro tipo diferente de erro que é uma característica do algorítimo usado e que é independente do hardware. Em geral ocorrem porquê muitos métodos computam *aproximações discretas* para uma quantidade contínua desejada.

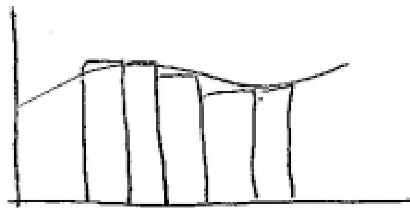
Exemplos:

- Calcular números que são o resultado de séries infinitas. Vimos anteriormente o exemplo de π, que pode ser calculado a partir da série:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots .$$

  O programa `cap03_pi.f90` ilustra a implementação numérica deste exemplo em particular.

- Aproximação de áreas por retângulos:



A discrepância entre o resultado correto e a resposta é chamado de erro de truncamento. Tal erro persistiria mesmo em um computador perfeito hipotético, com uma representação infinitamente acurada. A minimização de erros de truncamento corresponde ao cerne do campo da análise numérica!

3.3 Perda de precisão numérica

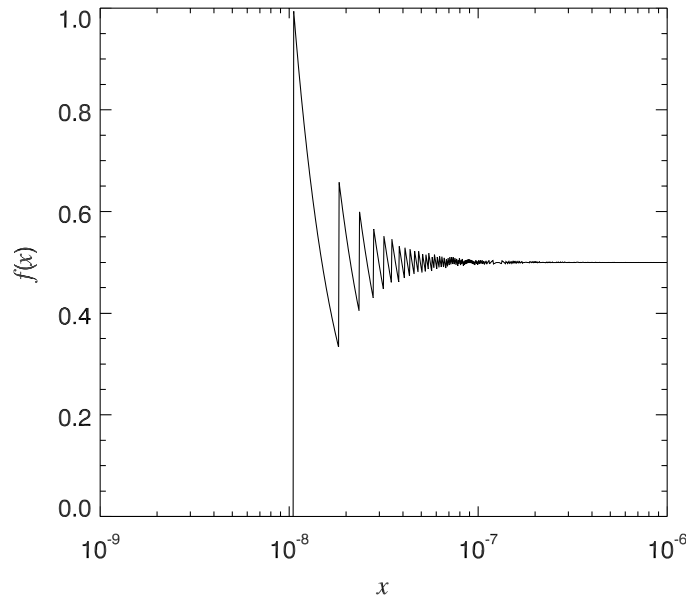
Em geral, quando um problema numérico é resolvido por um computador, os três tipos de erro descritos acima estão presentes e concorrem para a chamada *perda de precisão numérica*. Dependendo da magnitude desta perda de precisão, os resultados podem tornar-se completamente errados e sem sentido.

Como exemplo, considere a função

$$f(x) = \frac{1 - \cos(x)}{x^2} .$$

É possível mostrar (ver abaixo) que

$$\lim_{x \to 0} f(x) = \frac{1}{2} .$$



Entretanto, se calcularmos o valor de $f(x)$ *em dupla precisão*, obteremos o resultado mostrado na Figura 3.3.

Qual a origem do problema? Sabemos que $\lim_{x\to 0}\cos(x) = 1$. Assim, para valores pequenos de x a expressão $1 - \cos(x)$ tende a zero e começa a perder precisão, finalmente retornando o valor 0 quando $1 - \cos(x) < \epsilon_m$. Solução? Calcular $f(x)$ a partir de uma expansão em série. A expansão em série de $\cos(x)$ é bem conhecida:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \dots .$$

A partir dela, podemos escrever $f(x)$ como

$$f(x) = \frac{1}{2} - \frac{x^2}{4!} + \dots .$$

Da expressão acima é evidente que $\lim_{x\to 0} f(x) = 1/2$. Para implementar corretamente uma função numérica que calcula $f(x)$, temos que usar ora a expansão acima, para valores pequenos de x, ora $f(x)$ propriamente dita, para valores maiores do argumento. O programa `cap03_precisao.f90` mostra uma possível implementação.

3.4 Erro Absoluto e Erro Relativo

Define-se erro absoluto, e_x, como:

$$e_x = x - \bar{x} \tag{3.5}$$

onde $\bar{x}$ é o valor verdadeiro da grandeza e x seu valor aproximado. O erro relativo, ϵ_x, é definido como

$$\epsilon_x = \frac{|e_x|}{\bar{x}} = \frac{|x - \bar{x}|}{\bar{x}} . \tag{3.6}$$

Deve-se notar que o valor do erro absoluto pode ser pequeno enquanto que o erro relativo é grande. Por exemplo:

$$\bar{x} = 0.00052$$
$$x = 0.00061$$
$$e_x = 0.00009$$
$$\epsilon_x = \frac{|e_x|}{\bar{x}} = 0.17 = 17\%$$

3.4.1 IFs envolvendo número reais

Se números reais são representados de forma aproximada por um computador, devemos ter cuidado com expressões do tipo:

```
REAL :: x, y
...
IF (x == y) THEN ...
```

Pode ocorrer a situação em que, *matematicamente*, `x` e `y` devem ser iguais um ao outro, mas que devido a erros de truncamento, arredondamento ou representação eles não mais sejam exatamente iguais. Por exemplo, considere o código:

```
REAL :: x,y
...
x = 2.
...
y = SQRT(x)
...
IF (y*y == x) THEN
...
```

A expressão lógica dentro do `IF` é matematicamente verdadeira, mas provavelmente será avaliada como falsa pelo computador.

A solução para esse problema é rescrever a expressão lógica de forma que a comparação entre dois reais x e y seja feita de seguinte forma:

$$\frac{|x-y|}{x} < \epsilon \tag{3.7}$$

onde ϵ é uma precisão previamente escolhida, que depende do problema em questão. Pode-se, por exemplo, escolher ϵ como um múltiplo da precisão da máquina, ϵ_m. Uma solução para o problema acima seria:

```
REAL :: x,y
...
x = 2.
...
y = SQRT(x)
....
IF (ABS(y*y-x)/x < 10.*EPSILON(x)) THEN
....
```

3.5 Exercícios

1.