Cálculo Numérico

Notas de aula

Prof. Yuri Dumaresq Sobral

Departamento de Matemática
Universidade de Brasília

2025

O que é Cálculo Numérico?

O curso de *Cálculo Numérico* é uma introdução ao estudo da *Análise Numérica*, que é a área da Matemática que se interessa em *desenvolver* e *caracterizar* algoritmos eficientes para computar precisamente quantidades matemáticas (contas aritméticas, funções, integrais, etc...)

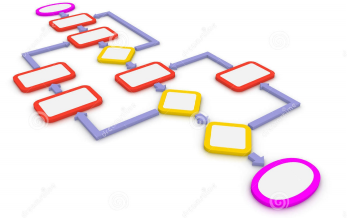
Temos três palavras *cruciais* devem ser bem definidas para entendermos o escopo da *Análise Numérica*!

Algoritmo

Um algoritmo é um procedimento que descreve, de maneira clara, sem ambiguidade, uma sequência finita de instruções/passos/comandos que devem ser executados em uma ordem específica.

Existem várias maneiras de representar um algoritmo:

- Diagrama de balões: forma de representação gráfica



https://thumbs.dreamstime.com/z/algorithm-27863175.jpg

Algoritmo

- Pseudocódigo: representação a partir de uma *linguagem* computacional simplificada

Exemplo:

Faça para i variando de 1 a 100
    Se i for par, escreva 'Par'
    Senão, escreva 'Ímpar'
Termine

Vamos usar principalmente esta opção para representar os algoritmos que vamos ver no curso de Cálculo Numérico.

Eficiência

A eficiência é uma medida de quão rapidamente um algoritmo produz um determinado resultado. A eficiência de algoritmos, em geral, é medida em termos da quantidade de operações aritméticas que o código executa.

A unidade de medida de esforço computacional é o flop, que é o esforço computacional envolvido em uma operação do tipo:

$$soma = soma + x(i) * x(i),$$

isto é: uma operação de soma, uma operação de produto e uma gestão de índices.

Eficiência

Exemplo:

Faça para i variando de 1 a N

$$soma = soma + x(i) * x(i)$$

Termine

Este algoritmo requer (exatamente) N flops para ser executado.

Observação: Raramente nos interessamos pela quantidade exata de flops de un algoritmo, e sim por sua ordem de grandeza. Às vezes, é impossível determinar exatamente quantos flops um determinado algoritmo necessita para ser executado.

Eficiência

Definição: Dizemos que $f(x) = \mathcal{O}\Big(g(x)\Big)$ quando, para $x \to \infty$, temos que

$$\lim_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| = M > 0,$$

isto é, $|f(x)| < M \cdot |g(x)|$ quando $x \to \infty$.

Exemplo: $f(x) = 6x^4 - 2x^3 + 5$ é $\mathcal{O}(x^4)$

Exemplo: O método para multiplicar matrizes $N \times N$ é $\mathcal{O}(N^3)$

Eficiência

Observação: Note que falar de ordem de um algoritmo não é necessariamente falar da velocidade com que um programa baseado neste algoritmo vai completar sua tarefa.

• Algoritmos bons em programas ruins levam a desempenhos modestos!

• Saber codificar bem um bom algoritmo é essencial se o objetivo é alcançar resultados mais rapidamente.

Precisão

Quando falamos de precisão nos referimos ao desejo de se avaliar uma quantidade matemática pelo computador corretamente, isto é, queremos que ela seja o mais correta possível.

Temos dois aspectos sobre a precisão que precisam ser discutidos quando falamos em calcular uma quantidade matemática no computador: convergência e erros sistemáticos.

Precisão

1. Convergência: Quando aplicamos um determinado algoritmo para resolver um problema matemático, precisamos ter *certeza* de que ele nos levará à sua solução *correta*, isto é, que o algoritmo convergirá para a solução do problema.

Mas, em geral, esta solução é desconhecida!

O ponto central da Análise Numérica é a convergência dos algoritmos: como demonstrar, a priori, que um algoritmo vai nos levar onde queremos chegar? Veremos alguns exemplos ao longo do curso!

Precisão

2. Erros sistemáticos

Existem 4 tipos de erros sistemáticos em cálculo numérico:

- modelo matemático inadequado para o problema estudado;
- erros nos parâmetros que alimentam o modelo matemático;
- aproximações usadas para resolver o problema matemático;
- erros de aproximação de aritmética computacional.

Os dois primeiros itens dizem respeito a problemas anteriores ao cálculo numérico e, portanto, não estão ao nosso alcance.

Às vezes, quando nos deparamos com um resultado obtido via computador que está claramente *errado*, precisamos ter certeza de que estamos resolvendo o problema correto antes de *culparmos* o método.

Precisão

O terceiro item é, de fato, um dos focos da análise numérica:

— Se não conseguimos resolver o *problema cheio*, que simplificações podemos fazer para encontrar uma resposta?

— Se der certo encontrarmos uma resposta, quão próxima da solução verdadeira do problema ela está?

Normalmente, esta abordagem é comum em problemas mais complicados, muitas vezes não-lineares. Não serão o nosso foco, mas talvez veremos alguns problemas deste tipo ao longo do nosso curso.

Precisão

O quarto ponto é, de fato, um problema de *engenharia de computação*! E, portanto, está fora de nosso alcance.

Mas, para controlarmos os erros que podemos cometer ao usarmos o computador, é essencial que conheçamos um pouco melhor como um computador faz contas.

Um computador, a *calculadora* que vamos utilizar neste curso, opera de uma maneira significativamente diferente de uma calculadora padrão e, por isto, vamos estudar um pouco de aritmética computacional.

Introdução à aritmética computacional

Quando trabalhamos com cálculo numérico, nosso objeto de trabalho mais essencial são os números e precisamos saber como o computador representa e manipula estes números.

Existem dois sistemas principais de representação de números em máquinas:

i. Representação por ponto fixo, que dá origem à aritmética de ponto fixo.

Nesta representação, os números são representados de forma igualmente espaçada, todos separados de seus *vizinhos* por uma quantidade Δ fixa.

Introdução à aritmética computacional

Este sistema tende a gerar sistemas computacionais mais rápidos e eficientes em processamento de operações.

Exemplo: são usados com frequência em *calculadoras* portáteis.

Porém... Um computador, por mais *potente* que seja, só pode representar uma quantidade finita de números (todos racionais)! Então...

O sistema de representação por ponto fixo NÃO é uma boa idéia se quisermos representar muitos números!

Introdução à aritmética computacional

Precisamos de uma sistema melhor!

ii. Representação por ponto flutuante, que dá origem à aritmética de ponto flutuante.

Neste sistema, existe uma distribuição não-uniforme dos números ao longo da reta real.

Existem vários *padrões* para a representação de números por ponto flutuante. Vamos discutir um exemplo destes para termos uma noção de como eles funcionam.

Introdução à aritmética computacional

Normalmente, os números nos sistema de ponto flutuante são representados como

$$\pm(1 + 0.d_1 d_2 d_3 \ldots d_t) \cdot \beta^e$$

onde:

- $d_1 \neq 0$ e $0 \leq d_i < \beta$, com $d_i \in \mathbb{Z}$
- β é a base do sistema
- t é a precisão do sistema
- e é o expoente do sistema
- Os números $d_1 d_2 d_3 \ldots d_t$ são chamados de *mantissa*

Introdução à aritmética computacional

Vamos dar uma olhada no sistema IEEE 754 de 32 bits, que é um dos padrões mais frequentemente usados atualmente. Neste padrão, temos:

- $\beta = 2$

- Os números são guardados em *words*, que consistem numa sequência de 32bits (números 0 ou 1) da seguinte forma:

0	10000100	10100000000000000000000
(sinal)	(expoente)	(mantissa)

Introdução à aritmética computacional

0	10000100	10100000000000000000000
(sinal)	(expoente)	(mantissa)

- (sinal): 1 bit para $+$ (0) ou $-$ (1);

- (expoente): 8 bits para determinar o expoente via números binários (base 2). Neste campo, o menor valor permitido é 00000000 ($= 0_{10}$) e o maior é 11111111 ($= 255_{10}$). O expoente é calculado como sendo este número menos 127 para permitir representar números pequenos e grandes;

- (mantissa): 23 bits para determinar a mantissa via números binários fracionais (base 2^{-1}).

Introdução à aritmética computacional

0	10000100	10100000000000000000000
(sinal)	(expoente)	(mantissa)

Vamos calcular este número:

- (sinal) $= 0$: $+$ (número positivo)
- (expoente) $= 10000100$:

$$\begin{aligned} 10000100_2 &= \Big\{0\cdot 2^0 + 0\cdot 2^1 + 1\cdot 2^2 + 0\cdot 2^3 + 0\cdot 2^4 + 0\cdot 2^5 + \\ &\quad +0\cdot 2^6 + 1\cdot 2^7\Big\}_{10} \\ &= \{2^2 + 2^7\}_{10} = \{4+128\}_{10} = 132_{10} \end{aligned}$$

Portanto: $e = 132 - 127 = 5$

Introdução à aritmética computacional

0	10000100	10100000000000000000000
(sinal)	(expoente)	(mantissa)

- (mantissa) = 10100000000000000000000:

$$
\begin{aligned}
10100000000000000000000_{\frac{1}{2}} &= \Big\{1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + \\
&\quad + 0 \cdot 2^{-4} + \cdots + 0 \cdot 2^{-23}\Big\}_{10} \\
&= \left\{\frac{1}{2} + \frac{1}{8}\right\}_{10} = 0,625
\end{aligned}
$$

Introdução à aritmética computacional

0	10000100	10100000000000000000000
(sinal)	(expoente)	(mantissa)

Portanto:

$\{01000010010100000000000000000000\}_{IEEE754} =$

$= \{+\ (1 + 0,625) \cdot 2^{5}\}_{10} = 52_{10}$ (ufa!)

Mas o que é realmente importante de toda esta conta?

Que agora sabemos qual é o próximo número possível que pode ser representado por este computador!

Introdução à aritmética computacional

O número seguinte a $\{01000010010100000000000000000000\}_{IEEE754}$ é:

$\{01000010010100000000000000000001\}_{IEEE754} =$

52,000003814697265625

Portanto, se buscarmos uma solução para um problema que esteja ENTRE estes dois números, temos um PROBLEMA: o computador não será capaz de representar este número! Este número não existe no computador!

O sistema físico do computador (*hardware*) encontrará uma maneira de aproximar (erro!) este número buscado para um possível.

Erros da aritmética computacional

Existem duas maneiras padrão de aproximar números:

Truncamento: Retiram-se as casas decimais que diferem do número possível imediatamente menor.

Arredondamento: Leva-se o número ao número possível mais próximo, seja ele maior ou menor.

Estas operações (que não podemos controlar!) levam a erros que são inerentes ao sistema de representação numérica de um computador. Temos que levar isto em consideração na hora de pensar sobre a solução de um problema numérico!

Erros da aritmética computacional

Podemos quantificar estes erros de duas maneiras. Seja x o valor exato que queiramos computar numericamente e seja x^* sua aproximação computada.

Erro absoluto: $E_a = |x - x^*|$

Erro relativo: $E_r = \dfrac{|x - x^*|}{|x|}$, se $x \neq 0$. Não definido para $x = 0$.

Se $E_r < 5 \cdot 10^{-t}$, dizemos que x^* aproxima x com $t - 1$ algarismos significativos.

Erros da aritmética computacional

Além dos erros de representação de números, temos também um problema de geração de erros quando operamos números com precisão finita, isto é, com um número finito de dígitos.

Normalmente, o sistema de 32 bits discutido anteriormente trabalha com 7 algarismos significativos.

Detalhes deste tipo de erros fogem ao nosso escopo, mas vamos mostrar onde este tipo de problemas podem aparecer.

Exemplo 1 Considere que tenhamos um sistema com 5 algarismos significativos e que queiramos calcular o valor de $\pi + \sqrt{2}$. Então:

$$[\pi]^* = 0.31416 \cdot 10^1 \quad \text{e} \quad [\sqrt{2}]^* = 0.14142 \cdot 10^1$$

Erros da aritmética computacional

Assim, teremos:

$$\pi+\sqrt{2} = \left[[\pi]^* + [\sqrt{2}]^*\right]^* = [0.31416{\cdot}10^1+0.14142{\cdot}10^1]^* = 0.45558{\cdot}10^1$$

Os erros envolvidos nestas operações são:

$$E_a = |0.45558 \cdot 10^1 - 4.5558062159\ldots| = 0.00000621596\ldots$$

$$E_r = \frac{|0.45558 \cdot 10^1 - 4.5558062159\ldots|}{|4.5558062159\ldots|} = 0.00000136\ldots = 1.36{\cdot}10^{-6}$$

Conclusão: estas operações preservaram os 5 algarismos significativos!

Erros da aritmética computacional

Porém, isto nem sempre acontece.

Exemplo 2 Considere que tenhamos um sistema com 4 algarismos significativos, que funcione com arredondamento, e que queiramos calcular o valor de $x - y$ para $x = 0.54617$ e $y = 0.54601$. Então:

$$x - y = [[x]^* - [y]^*]^* = [0.5462 \cdot 10^0 - 0.5460 \cdot 10^0]^* = 0.0002 \cdot 10^0$$

O erro relativo envolvido nesta operação é:

$$E_r = \frac{|0.0002 \cdot 10^0 - 0.00016|}{|0.00016|} = 0.25 = 2.5 \cdot 10^{-1}$$

Conclusão: perdemos TODOS os algarismos significativos nestas operações. Note que o erro relativo é de 25%, muito alto!

Erros da aritmética computacional

Problema: Se este valor for um passo intermediário de uma grande conta, estaremos propagando um erro de 25% nos passos seguintes, que irá contaminaro resultado final!

Esta perda significativa de precisão que o computador sofre devido a operações de ponto flutuante com precisão finita é chamada de CANCELAMENTO CATASTRÓFICO.

A evitar: operações com resultados que dão próximos de zero, operações com número de ordem de grandeza muito diferentes, números foram dos limites extremos do sistema de representação, etc.

Para o sistema de 32 bits, o menor número é próximo de 10^{-38}, (underflow), e o maior de 10^{36}, (overflow).

Erros da aritmética computacional - exemplos reais

- *The Patriot and the Scud*: em 25 de feveriro de 1991, na guerra do Iraque, um míssil de defesa Patriot americano não conseguiu interceptar um míssil Scud iraquiano, causando a morte de 28 soldados americanos. A causa do *erro* foi o erro acumulado de uma conversão de um fator de conversão de 10^{-1} (dízima periódica em binário) para um sistema de 24 bits. O míssil errou o alvo por cerca de 500m.
- *Foguete Ariane V*: em 4 de junho de 1996, um foguete Ariane V explodiu 36s após seu lançamento. O motivo foi a conversão da velocidade horizontal do foguete, guardada como um número de ponto flutuante de 64 bits, para um inteiro (com sinal) de 16 bits. O número resultou ser maior que 32768, o maior valor que podia ser guardado em 16 bits, e a conversão falhou.
- *Outros*: Bolsa de valores de Vancouver, etc... https://www-users.cse.umn.edu/∼arnold/455.f96/disasters.html , https://web.ma.utexas.edu/users/arbogast/misc/disasters.html

Erros da aritmética computacional

Exemplo final: Suponha que queiramos resolver numericamente o PVI abaixo.

$$\frac{dy}{dt} = 2y - \mathrm{e}^{-t}, \quad y(0) = \frac{1}{3}$$

A solução exata é dada por $y(t) = \frac{1}{3}\mathrm{e}^{-t}$, isto é, $y(t) \to 0$ quando $t \to \infty$. Porém... A condição inicial terá inevitavelmente um erro de representação δ, tal que $[y]^*(0) = \frac{1}{3} + \delta$. Então, a solução encontrada pelo computador será:

$$[y]^*(t) = \frac{1}{3}\mathrm{e}^{-t} + \delta \mathrm{e}^{2t}$$

Note que $[y]^*(t) \to \pm\infty$ quando $t \to \infty$, dependendo do sinal de δ. Comportamento totalmente errado.