

Minicurso de C/C++ para Matemática

Pedro H A Konzen

15 de novembro de 2024

Conteúdo

| | |
|--|-----------|
| Licença | 2 |
| 1 Sobre a Linguagem | 2 |
| 1.1 Instalação e Execução | 2 |
| 1.1.1 IDE | 3 |
| 1.2 Olá, mundo! | 3 |
| 2 Elementos da Linguagem | 4 |
| 2.1 Tipos de Dados Básicos | 4 |
| 2.2 Operações Aritméticas Elementares | 6 |
| 2.3 Funções e Constantes Elementares | 7 |
| 2.4 Operadores de Comparação Elementares | 8 |
| 2.5 Operadores Lógicos Elementares | 9 |
| 2.6 Arranjos | 11 |
| 3 Elementos da Programação Estruturada | 12 |
| 3.1 Métodos/Funções | 12 |
| 3.2 Ramificação | 15 |
| 3.3 Repetição | 17 |
| 3.3.1 <code>while</code> | 17 |
| 3.4 <code>do ... while</code> | 19 |
| 3.4.1 <code>for</code> | 20 |
| 4 Elementos da Computação Matricial | 21 |

| | | |
|----------|---|-----------|
| 4.1 | Vetores | 21 |
| 4.1.1 | Operações com Vetores | 22 |
| 4.2 | Matrizes | 25 |
| 4.3 | Operações Matriciais | 27 |
| 4.4 | Sistemas Lineares | 30 |
| 5 | Elementos da Orientação-a-Objetos | 33 |
| | Notas ³⁴ Referências ³⁵ | |

Licença

Este texto é disponibilizado sob a Licença Atribuição-CompartilhaIgual 4.0 Internacional Creative Commons. Para visualizar uma cópia desta licença, visite

http://creativecommons.org/licenses/by-sa/4.0/deed.pt_BR

ou mande uma carta para Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

1 Sobre a Linguagem

C e C++ são **linguagens** de programação **compiladas** de propósito geral. A primeira é **estruturada e procedural**, tendo sido criada em 1972 por Dennis Ritchie¹. A segunda foi inicialmente desenvolvida por Bjarne Stroustrup² como uma extensão da primeira. Em sua mais recente especificação, a **linguagem C++ se caracteriza por ser multi-paradigma (imperativa, orientada a objetos e genérica)**.

1.1 Instalação e Execução

Códigos C/C++ precisam ser compilados antes de serem executados. De forma simplificada, o **compilador** é um programa que interpreta e converte o código em um programa executável em computador. Há vários compiladores

¹Dennis Ritchie, 1941-2011, cientista da computação estadunidense. Fonte: [Wikipédia](#).

²Bjarne Stroustrup, 1950, cientista da computação dinamarquês. Fonte: [Wikipédia](#).

gratuitos disponíveis na web. Ao longo deste minicurso, usaremos a coleção de compiladores [GNU GCC](#) instalados em sistema operacional [Linux](#).

1.1.1 IDE

Usar um **ambiente integrado de desenvolvimento** (IDE, em inglês, *integrated development environment*) é a melhor forma de capturar o melhor das linguagens C/C++. Algumas alternativas são:

- [Eclipse](#)
- [GNU Emacs](#)
- [VS Code](#)

1.2 Olá, mundo!

Vamos **implementar** nosso primeiro programa C/C++. Em geral, são três passos: 1. escrever; 2. compilar; 3. executar.

1. **Escrever o código.**

Em seu IDE preferido, digite o código:

Código 1: ola.cc

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Olá, mundo!\n");
6     return 0;
7 }
```

2. **Compilar.**

Para compilá-lo, digite no terminal de seu sistema operacional

```
1 $ gcc ola.cc -o ola.x
```

3. **Executar.**

Terminada a compilação, o arquivo executável `ola.x` é criado. Para executá-lo, digite

```
1 $ ./ola.x
```

2 Elementos da Linguagem

2.1 Tipos de Dados Básicos

Na linguagem C/C++, **dados** são alocados em **variáveis** com tipos declarados³.

Exemplo 2.1.1. Consideramos o seguinte código.

Código 2: dados.cc

```
1 /* dados.cc
2    Exemplo de alocação de variáveis.
3 */
4 #include <stdio.h>
5
6 int main()
7 {
8     // var inteira
9     int i = 1;
10    // var pto flutuante
11    double x;
12
13    x = 2.5;
14    char s[6] = "i + x";
15    double y = i + x;
16    printf("%s = %f\n", s, y);
17    return 0;
18 }
```

Na linha 9, é alocada uma **variável do tipo inteira** com **identificador** `i` e **valor** 1. Na linha 11, é alocada uma **variável do tipo ponto flutuante**

³Consulte [Wikipedia: C data type](#) para uma lista dos tipos de dados disponíveis na linguagem

(64 *bits*) com identificador `x`.

Na linha 14, é alocada uma **variável do tipo *string***⁴. Na linha 15, alocamos uma nova variável `y`.

Observação 2.1.1. (**Comentários e Continuação de Linha.**) Códigos C++ admitem **comentários** e **continuação de linha** como no seguinte exemplo acima. Comentários em linha podem ser feitos com `//` e de múltiplas linhas com `* ... */`. Linhas de instruções muito compridas podem ser quebradas em múltiplas linhas com a instrução de continuação de linha `\`.

Observação 2.1.2. (**Notação científica.**) Podemos usar **notação científica** em C++. Por exemplo 5.2×10^{-2} é digitado da seguinte forma `5.2e-2`.

Código 3: `notacaoCientifica.cpp`

```
1 #include <stdio.h>
2
3 int main()
4 {
5
6     int i = -51;
7     double x = 5.2e-2;
8
9     // inteiro
10    printf("inteiro: %d\n", i);
11    // fixada
12    printf("fixada: %f\n", x);
13    // notação científica
14    printf("científica: %e\n", x);
15    return 0;
16 }
```

Exercício 2.1.1. Antes de implementar, diga qual o valor de `x` após as seguintes instruções.

```
1 int x = 1;
2 int y = x;
3 y = 0;
```

⁴Um arranjo de `char` (caracteres).

Justifique seu resposta e verifique-a.

Exercício 2.1.2. Implemente um código em que a(o) usuária(o) entra com valores para as variáveis `x` e `y`. Então, os valores das variáveis são permutados entre si. Dica: a entrada de dados por usuária(o) pode ser feita com o método C/C++ `scanf` da biblioteca `stdio.h`. Por exemplo,

```
1 double x;  
2 scanf("%lf", &x);
```

faz a leitura de um `double` (long float) e o armazena na variável `x`.

2.2 Operações Aritméticas Elementares

Os operadores aritméticos elementares são⁵:

`*`, `/`, `%` **multiplicação, divisão, módulo**

`+`, `-` **adição, subtração**

Exemplo 2.2.1. Qual é o valor impresso pelo seguinte código?

```
1 #include <stdio.h>  
2  
3 int main()  
4 {  
5     printf("%f\n", 2+17%9/2*2-1 );  
6     return 0;  
7 }
```

Observamos que as operações `*`, `/` e `%` têm precedência maior que as operações `+` e `-`. Operações de mesma precedência seguem a ordem da esquerda para direita, conforme escritas na linha de comando. **Usa-se parênteses para alterar a precedência entre as operações**, por exemplo

```
1 printf("%f\n", (2+17)%9/2*2-1 );
```

imprime o resultado `-1`. Sim, pois a **divisão inteira** está sendo usada. Para computar a divisão em ponto flutuante, um dos operandos deve ser `double`

⁵Em ordem de precedência.

. Para tanto, podemos fazer um **casting** `double((2+17)\%9)/2*2-1` ou, simplesmente, `(2+17)\%9/2.*2-1`.

Observação 2.2.1. (**Precedência das Operações.**) Consulte mais informações sobre a precedência de operadores em [Wikipedia:Operators in C and C++](#).

Exercício 2.2.1. Escreva um programa para computar o vértice da parábola

$$ax^2 + bx + c = 0, \quad (1)$$

para $a = 2$, $b = -2$ e $c = 4$.

O operador `%` módulo computa o **resto** da divisão inteira, por exemplo, `5\%2` é igual a 1.

Exercício 2.2.2. Use C/C++ para computar os inteiros não negativos q e r tais que

$$25 = q \cdot 3 + r, \quad (2)$$

sendo r o menor possível.

2.3 Funções e Constantes Elementares

A biblioteca C/C++ `math.h` disponibiliza várias funções e constantes elementares.

Exemplo 2.3.1. O seguinte código, imprime os valores de π , $\sqrt{2}$ e $\ln e$.

Código 4: `mat.cc`

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main()
5 {
6     printf("pi = %.9e\n", M_PI);
7     printf("2^(1/2) = %.5f\n", sqrt(2.));
8     printf("log(e) = %f\n", log(M_E));
9     return 0;
10 }
```

Observação 2.3.1. (**Compilação e Linkagem.**) A compilação de um código C/C++ envolve a linkagem de bibliotecas. A `stdio.h` é linkada de forma automática na compilação. Já, `math.h` precisa ser explicitamente linkada com

```
1 $ gcc foo.cc -lm
```

Observação 2.3.2. (**Logaritmo Natural.**) Notamos que `log` é a função logaritmo natural, i.e. $\ln(x) = \log_e(x)$. A implementação C/C++ para o logaritmo de base 10 é `log10(x)`.

Exercício 2.3.1. Compute

a) $\sin\left(\frac{\pi}{4}\right)$

b) $\log_3(\pi)$

c) $e^{\log_2(\pi)}$

d) $\sqrt[3]{-27}$

Exercício 2.3.2. Compute as raízes do seguinte polinômio quadrático

$$p(x) = 2x^2 - 2x - 4 \quad (3)$$

usando a fórmula de Bhaskara¹.

2.4 Operadores de Comparação Elementares

Os operadores de comparação elementares são

`==` **igual a**

`!=` **diferente de**

`>` **maior que**

`<` **menor que**

`>=` **maior ou igual que**

<= menor ou igual que

Estes operadores retornam os **valores lógicos** **true** (verdadeiro, 1) ou **false** (falso, 0).

Por exemplo, temos

Código 5: opComp.cc

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int x = 2;
6     bool res = x + x == 5;
7     printf("2 + 2 == 5? %d", res);
8 }
```

Exercício 2.4.1. Considere a circunferência de equação

$$c : (x - 1)^2 + (y + 1)^2 = 1. \quad (4)$$

Escreva um código em que a(o) usuária(o) entra com as coordenadas de um ponto $P = (x, y)$ e o código verifica se P pertence ao disco determinado por c .

Exercício 2.4.2. Antes de implementar, diga qual é o valor lógico da instrução `sqrt(3) == 3`. Justifique sua resposta e verifique!

2.5 Operadores Lógicos Elementares

Os operadores lógicos elementares são:

&& e lógico

|| ou lógico

! não lógico

Exemplo 2.5.1. (**Tabela Booleana do &&**.) A tabela booleana² do **e lógico** é

| A | B | A && B |
|-------|-------|--------|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

O seguinte código, monta essa tabela booleana, verifique!

```

1 #include <stdio.h>
2
3 int main()
4 {
5     bool T = true;
6     bool F = false;
7     printf("A    | B    | A && B\n");
8     printf("%d    | %d    | %d\n", T, T, T&&T);
9     printf("%d    | %d    | %d\n", T, F, T&&F);
10    printf("%d    | %d    | %d\n", F, T, F&&T);
11    printf("%d    | %d    | %d\n", F, F, F&&F);
12 }
```

Exercício 2.5.1. Construa as tabelas booleanas do operador `||` e do `!`.

Exercício 2.5.2. Escreva um código para verificar as seguintes comparações

- a) $1.4 \leq \sqrt{2} < 1.5$.
- b) $|x| < 1$, $x = \sin(\pi/3)$.
- c) $|x| > \frac{1}{2}$, $x = \cos(\pi * 2)$.

Exercício 2.5.3. Considere um retângulo $r : ABDC$ de vértices $A = (1, 1)$ e $D = (2, 3)$. Crie um código em que a(o) usuária(o) informa as coordenadas de um ponto $P = (x, y)$ e o código verifica cada um dos seguintes itens:

1. $P \in r$.
2. $P \in \partial r$.
3. $P \notin \bar{r}$.

Exercício 2.5.4. Implemente uma instrução para computar o operador `xor` (ou exclusivo). Dadas duas afirmações A e B , $A \text{ xor } B$ é `true` no caso de uma, e somente uma, das afirmações ser `true`, caso contrário é `false`.

2.6 Arranjos

Um **arranjo**⁶ é uma sequência de dados do mesmo tipo. Os elementos dos arranjos são indexados⁷ e mutáveis (podemos ser alterados por nova atribuição).

Exemplo 2.6.1. No código abaixo, alocamos o ponto $P = (2, 3)$ e o vetor $v = (2.5, \pi, -1.)$ como arranjos.

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main()
5 {
6     // P = (2, 3)
7     int P[2] = {2, 3};
8     printf("P = (%d, %d)\n", P[0], P[1]);
9
10    double v[3];
11    v[0] = 2.5;
12    v[1] = M_PI;
13    v[2] = -1.;
14    printf("v = (%lf, %lf, %lf)\n", v[0], v[1], v[2]);
15
16    return 0;
17 }
```

Exercício 2.6.1. Escreva um código em que a(o) usuária(o) entra com um ponto $P = (x, y)$ e o programa informe se P pertence ao disco determinado pela circunferência de equação $(x - 1)^2 + y^2 = 4$. Use de um arranjo para alocar o ponto P .

⁶Em inglês, *array*

⁷O índice é um inteiro não negativo, sendo o primeiro elemento indexado por 0 (zero).

Exercício 2.6.2. Considere os vetores

$$\mathbf{v} = (-1., 2., 1.) \quad (5)$$

$$\mathbf{w} = (1., -3., 2.). \quad (6)$$

Faça um código que aloca os vetores como arranjos e imprime o vetor soma $\mathbf{v} + \mathbf{w}$.

Exercício 2.6.3. Considere a matriz

$$A = \begin{vmatrix} 1. & -2. \\ 3. & 3. \end{vmatrix}. \quad (7)$$

Faça um código que aloca a matriz como um arranjo bidimensional (um arranjo de arranjos) e compute seu determinante.

3 Elementos da Programação Estruturada

C/C++ são linguagens procedurais⁸ e contém instruções para a **programação estruturada**. Neste paradigma de programação, as computações são organizadas em sequências de blocos computacionais e, um bloco inicia sua computação somente após o bloco anterior tiver terminado. Contam com estruturas de **ramificação** (seleção de blocos), **repetição** de blocos e definição de **funções/métodos** (sub-blocos computacionais).

3.1 Métodos/Funções

Um **método** (ou **função**) é um subprograma (ou subbloco computacional) que pode ser chamado/executado em qualquer parte do programa principal. Todo código C/C++ inicia-se no método `main()`, consulte o Código 1. A sintaxe de definição de um método é

```
1 typeOut foo(typeIn0 x0, typeIn1 x1, ..., typeInN
  x2)
2 {
3   typeOut out;
4   statment0;
```

⁸C++ também é orientada-a-objetos.

```
5  statement1;  
6  ...;  
7  statementN;  
8  return out;  
9 }
```

Aqui, `typeOut` denota o tipo da saída, `foo` denota o identificador/nome do método, `typeIn0 x1`, `typeIn1 x2`, ..., `typeInN x3` são os tipos e identificadores dos parâmetros de entrada⁹. O escopo do método é delimitado entre chaves e pode conter qualquer instrução (*statement*) C/C++. O método é encerrado¹⁰ quando terminado seu escopo ou ao encontrar a instrução `return`. Esta instrução, também, permite o retorno de um dado do mesmo tipo da saída do método.

Exemplo 3.1.1. Vamos considerar a função

$$f(x) = 2x - 3. \quad (8)$$

a) No código abaixo, o método f computa a função e imprime seu valor¹¹.

Código 6: method.cc

```
1 #include <stdio.h>  
2  
3 void f(double x)  
4 {  
5     double y = 2.*x - 3.;  
6     printf("f(%lf) = %lf\n", x, y);  
7 }  
8  
9 int main()  
10 {  
11     f(0.);  
12     double x = -1.;  
13     f(x);  
14     double y = 2.;  
15     f(y);
```

⁹Parâmetros de entrada são opcionais

¹⁰No encerramento do método o código retorna ao programa principal.

¹¹`void` é a instrução para “no type”.

```
16 return 0;
17 }
```

- b) Nesta versão do código, o método `f` retorna o valor computado da função `f` e é o método principal `main` que imprime o resultado.

```
1 #include <stdio.h>
2
3 double f(double x)
4 {
5     return 2.*x - 3.;
6 }
7
8 int main()
9 {
10    double y = f(0.);
11    printf("f(%lf) = %lf\n", 0., y);
12    printf("f(%lf) = %lf\n", -1., f(-1.));
13    double z = 2.;
14    printf("f(%lf) = %lf\n", z, f(z));
15    return 0;
16 }
```

Exercício 3.1.1. Implemente uma função para computar as raízes de um polinômio de grau 1 $p(x) = ax + b$. Assuma que $a \neq 0$.

Exercício 3.1.2. Implemente uma função para computar as raízes reais de um polinômio de grau 2 $p(x) = ax^2 + bx + c$. Assuma que p tenha raízes reais.

Exercício 3.1.3. Considerando vetores em \mathbb{R}^3

$$x = (x_1, x_2, x_3), \quad (9)$$

$$y = (y_1, y_2, y_3), \quad (10)$$

implemente um código que contenha:

- a) função para computação do vetor soma $\mathbf{x} + \mathbf{y}$.
- b) função para computação do produto escalar $\mathbf{x} \cdot \mathbf{y}$.

Exercício 3.1.4. Implemente uma função que computa o determinante de matrizes reais 2×2 .

Exercício 3.1.5. Implemente uma função que computa a multiplicação matrix-vetor Ax , com A 2×2 e x um vetor coluna de dois elementos.

Exercício 3.1.6. (Recursividade) Implemente uma função recursiva para computar o fatorial de um número natural n , i.e. $n!$.

3.2 Ramificação

Uma estrutura de ramificação é uma instrução para a tomada de decisões durante a execução de um programa. Nas linguagens C/C++ usa-se a sintaxe

```
1 if (condition0) {  
2     block0;  
3 } else if (condition1) {  
4     block1;  
5 } else {  
6     block2;  
7 }
```

A instrução `if` permite a execução do bloco computacional `block0` somente no caso de a `condition0` seja `true` (verdadeira). A instrução `else if` somente é verificada quando `condition0 == false`. Neste caso, o `block1` é executado somente se `condition1 == true`. Senão, `block2` é executado.

Exemplo 3.2.1. Os seguintes códigos computam os zeros da função

$$f(x) = ax + b, \quad (11)$$

para parâmetros informados por usuária(o).

a) Caso restrito a raiz real única.

```
1 #include <stdio.h>  
2  
3 int main()  
4 {  
5     double a,b;  
6     printf("a = ");
```

```
7  scanf("%lf", &a);
8  printf("b = ");
9  scanf("%lf", &b);
10
11  if (a != 0.) {
12      double x = -b/a;
13      printf("x = %lf\n", x);
14  }
15
16  return 0;
17 }
```

b) Caso de raiz real única ou múltiplas.

```
1  #include <stdio.h>
2
3  int main()
4  {
5      double a,b;
6      printf("a = ");
7      scanf("%lf", &a);
8      printf("b = ");
9      scanf("%lf", &b);
10
11     if (a != 0.) {
12         double x = -b/a;
13         printf("x = %lf\n", x);
14     } else if ((a == 0.) && (b == 0.)) {
15         printf("Todo x real é zero da função.\n");
16     }
17
18     return 0;
19 }
```

c) Caso de raiz real única, ou múltiplas ou nenhuma.

```
1  #include <stdio.h>
2
3  int main()
```



```
4 {
5     double a,b;
6     printf("a = ");
7     scanf("%lf", &a);
8     printf("b = ");
9     scanf("%lf", &b);
10
11     if (a != 0.) {
12         double x = -b/a;
13         printf("x = %lf\n", x);
14     }
15
16     return 0;
17 }
```

Exercício 3.2.1. Implemente um código que contenha uma função que recebe dois números n e m e imprime o maior deles.

Exercício 3.2.2. Implemente um código que contenha uma função que recebe os coeficientes de um polinômio

$$p(x) = ax^2 + bx + c \quad (12)$$

e classifique-o como um polinômio de grau 0, 1 ou 2.

Exercício 3.2.3. Implemente um código que contenha uma função para a computação das raízes de um polinômio de segundo grau.

3.3 Repetição

Estruturas de repetição são instruções que permitem a execução repetida de um bloco computacional. São três instruções disponíveis `while`, `do ... while` e `for`.

3.3.1 while

A sintaxe da instrução `while` é

```
1 while (condition) {
2     block
```

```
3 }
```

Isto é, enquanto (**while**) a expressão `condition == true`, o bloco computacional `block` é repetidamente executado. Ao final de cada execução, a condição é novamente verificada. Quando `condition == false`, `block` não é executado e o código segue para a primeira instrução após o escopo do **while**.

Exemplo 3.3.1. O seguinte código computa a soma dos 10 primeiros termos da progressão geométrica

$$a_i = 2^{-i}, \quad (13)$$

para $i = 0, 1, 2, \dots$

Código 7: while.cc

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main()
5 {
6     int i = 0;
7     double s = 0.;
8     while (i < 10) {
9         s = s + pow(0.5, double(i));
10        i += 1;
11    }
12    printf("s = %lf\n", s);
13    return 0;
14 }
```

Observação 3.3.1. As instruções de controle **break**, **continue** são bastante úteis em várias situações. A primeira, encerra as repetições e, a segunda, pula para uma nova repetição.

Exercício 3.3.1. Use **while** para imprimir os dez primeiros números ímpares.

Exercício 3.3.2. Crie uma função para a computação da soma de dois vetores $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, com dado $n \geq 0$.

Exercício 3.3.3. Use a instrução `while` para escreva uma função que retorne o n -ésimo termo da função de Fibonacci³, $n \geq 1$.

3.4 do ... while

Diferentemente da instrução `while`, a `do ... while` verifica a condição de repetição ao final do escopo do seu bloco computacional.

Exemplo 3.4.1. O seguinte código computa a soma dos 10 primeiros termos da progressão geométrica

$$a_i = 2^{-i}, \quad (14)$$

para $i = 0, 1, 2, \dots$

Código 8: doWhile.cc

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main()
5 {
6     int i = 0;
7     double s;
8     do {
9         s += pow(0.5, double(i));
10        i += 1;
11    } while (i < 10);
12    printf("s = %lf\n", s);
13    return 0;
14 }
```

Exercício 3.4.1. Uma aplicação do Método Babilônico¹² para a aproximação da solução da equação $x^2 - 2 = 0$, consiste na iteração

$$x_0 = 1, \quad (15)$$

$$x_{i+1} = \frac{x_i}{2} + \frac{1}{x_i}, \quad i = 0, 1, 2, \dots \quad (16)$$

¹²Matemática Babilônica, matemática desenvolvida na Mesopotâmia, desde os Sumérios até a queda da Babilônia em 539 a.C.. Fonte: [Wikipédia](#).

Faça um código com `while` para computar aproximação x_i , tal que $|x_i - x_{i-1}| < 10^{-5}$.

3.4.1 for

A estrutura `for` tem a sintaxe

```
1 for (init; condition; iter) {  
2     block;  
3 }
```

onde, `init` é a instrução de inicialização, `condition` é o critério de parada, `iter` é a instrução do iterador.

Exemplo 3.4.2. O seguinte código computa a soma dos 10 primeiros termos da progressão geométrica

$$a_i = 2^{-i}, \quad (17)$$

para $i = 0, 1, 2, \dots$

```
1 #include <stdio.h>  
2 #include <math.h>  
3  
4 int main()  
5 {  
6     double s = 0;  
7     for (int i=0; i<10; ++i) {  
8         s += pow(2., double(-i));  
9     }  
10    printf("s = %lf\n", s);  
11    return 0;  
12 }
```

Exercício 3.4.2. Use a instrução `for` para escreva uma função que retorne o n -ésimo termo da função de Fibonacci⁴, $n \geq 1$.

Exercício 3.4.3. Implemente uma função para computar o produto escalar de dois vetores de n elementos. Use a instrução de repetição `for` e assuma que os vetores estão alocados como um arranjo `double`.

Exercício 3.4.4. Implemente uma função para computar a multiplicação de

uma matriz A $n \times n$ por um vetor coluna x de n elementos. Use a instrução `for` e assumo que o vetor e a matriz estejam alocadas como arranjos `double`.

Exercício 3.4.5. Implemente uma função para computar a multiplicação de uma matriz A $n \times m$ por uma matriz B de $m \times n$. Use a instrução `for` e assumo que as matrizes estão alocadas como arranjos `double`.

4 Elementos da Computação Matricial

GSL (GNU Scientific Library) é uma biblioteca de métodos numéricos para **C/C++**. É um software livre sobre a *GNU General Public License* e está disponível em

<https://www.gnu.org/software/gsl/>.

A biblioteca fornece uma grande número de rotinas matemáticas para várias áreas da análise numérica. Aqui, vamos nos concentrar em uma rápida introdução à computação matricial.

4.1 Vetores

A alocação de um vetor no **GSL** segue o estilo de `malloc` (alocação de memória) e `free` (liberação de memória). O **suporte a vetores** requer a importação da biblioteca `gsl_vector.h`.

Exemplo 4.1.1. No seguinte código, alocamos e imprimimos o seguinte vetor

$$\mathbf{v} = (\sqrt{2}, 1, 3.5, \pi). \quad (18)$$

Código 9: vector.cc

```
1 #include <stdio.h>
2
3 // GSL const e funs matemáticas
4 #include <gsl/gsl_math.h>
5 // GSL vetores
6 #include <gsl/gsl_vector.h>
7
8 int main() {
```

```
9 // alocação de memória
10 gsl_vector *v = gsl_vector_alloc(4);
11
12 // atribuição
13 gsl_vector_set(v, 0, sqrt(2.));
14 gsl_vector_set(v, 1, 1.);
15 gsl_vector_set(v, 2, 3.5);
16 gsl_vector_set(v, 3, M_PI);
17
18 // acesso e impressão
19 for (int i=0; i<4; ++i) {
20     printf("v_%d = %g\n", i, gsl_vector_get(v, i))
21 ;
22 }
23 // liberação de memória
24 gsl_vector_free(v);
25 }
```

A compilação desse código requer a linkagem com a biblioteca [GSL](#):

```
1 $ gcc vetor.cc -lgsl -lgslcblas -lm
```

Observação 4.1.1. (**Inicialização.**) Alternativamente, a alocação com o método

```
1 gsl_vector *gsl_vector_calloc(size_t n)
```

cria um vetor e inicializa todos os seus elementos como zero. Outros métodos de inicialização estão disponíveis, consulte [GSL Docs: Initializing vector elements](#).

4.1.1 Operações com Vetores

Operações básicas envolvendo vetores do [GSL](#) estão disponíveis com os seguintes métodos¹³:

- `int gsl_vector_add(gsl_vector *a, const gsl_vector *b)`

¹³Mais detalhes, consulte [GNU Docs: Vector operations](#).

Computa a adição vetorial $\mathbf{a} + \mathbf{b}$ e o resultado é armazenado no vetor \mathbf{a} .

- `int gsl_vector_sub(gsl_vector *a, const gsl_vector *b)`

Computa a subtração vetorial $\mathbf{a} - \mathbf{b}$ e o resultado é armazenado no vetor \mathbf{a} .

- `int gsl_vector_mul(gsl_vector *a, const gsl_vector *b)`

Computa a multiplicação elemento-a-elemento $\mathbf{a} * \mathbf{b}$ e armazena o resultado no vetor \mathbf{a} .

- `int gsl_vector_div(gsl_vector *a, const gsl_vector *b)`

Computa a divisão elemento-a-elemento \mathbf{a} / \mathbf{b} e armazena o resultado no vetor \mathbf{a} .

- `int gsl_vector_scale(gsl_vector *a, const double x)`

Computa a multiplicação por escalar $\mathbf{x} * \mathbf{a}$ e armazena o resultado no vetor \mathbf{a} .

- `int gsl_vector_add_constant(gsl_vector *a, const double x)`

Re-computa o vetor \mathbf{a} somando o escalar \mathbf{x} a cada um de seus elementos.

- `double gsl_vector_sum(const gsl_vector *a)`

Retorna a soma dos elementos do vetor \mathbf{a} .

Observação 4.1.2. (Suporte BLAS.) O **GSL** também fornece suporte a biblioteca BLAS (*Basic Linear Algebra Subprograms*) pelo `gsl_blas.h`. Operações vetoriais estão disponíveis no nível 1 da biblioteca. Para mais informações e acesso à documentação sobre os métodos disponíveis, consulte

<https://www.gnu.org/software/gsl/doc/html/blas.html>.

Exemplo 4.1.2. No código abaixo, computamos $\mathbf{w} = \alpha \mathbf{u} + \mathbf{v}$ para o escalar

$\alpha = 2$ e os vetores

$$\begin{aligned}\mathbf{u} &= (1, -2, 0.5), \\ \mathbf{v} &= (2, 1, -1.5).\end{aligned}\tag{19}$$

Código 10: axpy.cc

```
1 #include <stdio.h>
2
3 // GSL vetores
4 #include <gsl/gsl_vector.h>
5 // GSL BLAS
6 #include <gsl/gsl_blas.h>
7
8 int main() {
9
10     // alpha
11     double alpha = 2.;
12
13     // u
14     gsl_vector *u = gsl_vector_alloc(3);
15     gsl_vector_set(u, 0, 1.);
16     gsl_vector_set(u, 1, -2.);
17     gsl_vector_set(u, 2, 0.5);
18
19     // v
20     gsl_vector *v = gsl_vector_alloc(3);
21     gsl_vector_set(v, 0, 2.);
22     gsl_vector_set(v, 1, 1.);
23     gsl_vector_set(v, 2, -1.5);
24
25     // w = alpha*u + v
26     // alloc w
27     gsl_vector *w = gsl_vector_alloc(3);
28     // copy w = v
29     gsl_vector_memcpy(w, v);
30     // w = alpha*u + w
31     gsl_blas_daxpy(alpha, u, w);
32
33     // imprime
```



```

34  for (int i=0; i<3; ++i) {
35      printf("w_%d = %g\n", i, gsl_vector_get(w, i))
36  };
37  }
38  // liberação de memória
39  gsl_vector_free(v);
40 }

```

Exercício 4.1.1. Faça um código para computar o produto escalar $\mathbf{x} \cdot \mathbf{y}$ dos vetores

$$\mathbf{x} = (1.2, \ln(2), 4), \quad (20)$$

$$\mathbf{y} = (\pi^2, \sqrt{3}, e). \quad (21)$$

- Crie sua própria função `double dot(const gsl_vector *x, const gsl_vector *y)` que recebe os vetores e retorna o produto escalar deles.
- Use o método BLAS `gsl_blas_dsdot`.

Exercício 4.1.2. Faça um código para computar a norma L^2 do vetor

$$\mathbf{x} = (1.2, \log_{10}^2(2), 0.5). \quad (22)$$

- Crie sua própria função `double norm2(const gsl_vector *x)` que recebe o vetor e retorna sua norma.
- Use o método BLAS `gsl_blas_dnrm2`.

4.2 Matrizes

Assim como vetores, a alocação de matrizes no `GSL` segue o estilo `malloc` e `free`. O `suporte a matrizes` requer a importação da biblioteca `gsl_matrix.h`.

Exemplo 4.2.1. No seguinte código, alocamos e imprimimos a matriz

$$A = \begin{bmatrix} 1.5 & -1 & \pi & 2.3 \\ \sqrt[3]{25} & 2.1 & -3.5 & 3 \\ \log 15 & 0 & 2.5^3 & -1.7 \end{bmatrix} \quad (23)$$

Código 11: matriz.cc

```
1 #include <stdio.h>
2
3 // GSL
4 #include <gsl/gsl_math.h>
5 #include <gsl/gsl_matrix.h>
6
7 int main()
8 {
9     // alocação
10    gsl_matrix *A = gsl_matrix_alloc(3, 4);
11
12    // população
13    gsl_matrix_set(A, 0, 0, 1.5);
14    gsl_matrix_set(A, 0, 1, -1.);
15    gsl_matrix_set(A, 0, 2, M_PI);
16    gsl_matrix_set(A, 0, 3, 2.3);
17
18    gsl_matrix_set(A, 1, 0, cbrt(25.));
19    gsl_matrix_set(A, 1, 1, 2.1);
20    gsl_matrix_set(A, 1, 2, -3.5);
21    gsl_matrix_set(A, 1, 3, 3.);
22
23    gsl_matrix_set(A, 2, 0, log10(15.));
24    gsl_matrix_set(A, 2, 1, 0.);
25    gsl_matrix_set(A, 2, 2, pow(2.5, 3.));
26    gsl_matrix_set(A, 2, 3, -1.7);
27
28    // imprime
29    printf("A = \n");
30    for (int i=0; i<A->size1; ++i) {
31        for (int j=0; j<A->size2; ++j)
32            printf("%g ", gsl_matrix_get(A, i, j));
33        printf("\n");
34    }
35
36    gsl_matrix_free(A);
```

```
37 return 0;  
38 }
```

Observação 4.2.1. (**Inicialização de Matrizes.**) Matrizes podem ser inicializadas com todos os seus elementos nulos usando

```
1 gsl_matrix *gsl_matrix_calloc(size_t n1, size_t n2  
  )
```

Outros métodos de inicialização também estão disponíveis, consulte

<https://www.gnu.org/software/gsl/doc/html/vectors.html#initializing-matrix-elements>.

4.3 Operações Matriciais

O **GSL** conta com os seguintes **métodos de operações matriciais**¹⁴:

- `int gsl_matrix_add(gsl_matrix *a, const gsl_matrix *b)`

Computa a adição matricial $a + b$ e armazena o resultado em a .

- `int gsl_matrix_sub(gsl_matrix *a, const gsl_matrix *b)`

Computa a subtração matricial $a - b$ e armazena o resultado em a .

- `int gsl_matrix_mul_elements(gsl_matrix *a, const gsl_matrix *b)`

Computa a multiplicação elemento-a-elemento $a*b$ e armazena o resultado em a .

- `int gsl_matrix_div_elements(gsl_matrix *a, const gsl_matrix *b)`

Computa a divisão elemento-a-elemento a/b e armazena o resultado em a .

- `int gsl_matrix_scale(gsl_matrix *a, const double x)`

¹⁴Consulte a lista completa em [GSL Docs: Matrix operations](#).

Computa a multiplicação por escalar $\mathbf{x} \cdot \mathbf{a}$ e armazena o resultado em \mathbf{a} .

- `int gsl_matrix_add_constant(gsl_matrix *a, const double x)`

Re-computa a matriz \mathbf{a} somando \mathbf{x} a cada um de seus elementos.

Observação 4.3.1. (**Operações Matrix-Vetor e Matriz-Matriz.**) Na **GSL**, **operações matrix-vetor** estão disponíveis no suporte **BLAS de nível 2**. Já, operações matriz-matriz, no suporte **BLAS de nível 3**. Consulte a lista completa de métodos em

<https://www.gnu.org/software/gsl/doc/html/blas.html#blas-support>.

Exemplo 4.3.1. O seguinte código verifica se $\mathbf{x} = (-1, 1, -2)$ é solução do sistema linear

$$\begin{aligned}x_1 - x_2 + 2x_3 &= -6 \\2x_1 + x_2 - x_3 &= 1 \\-x_1 + x_2 + x_3 &= 0\end{aligned}\tag{24}$$

Código 12: sisLin.cc

```
1 #include <stdio.h>
2
3 // GSL
4 #include <gsl/gsl_math.h>
5 #include <gsl/gsl_vector.h>
6 #include <gsl/gsl_matrix.h>
7 #include <gsl/gsl_blas.h>
8
9 int main()
10 {
11     // matriz dos coefs
12     gsl_matrix *A = gsl_matrix_alloc(3, 3);
13
14     gsl_matrix_set(A, 0, 0, 1.);
15     gsl_matrix_set(A, 0, 1, -1.);
16     gsl_matrix_set(A, 0, 2, 2.);
17 }
```

```
18  gsl_matrix_set(A, 1, 0, 2.);
19  gsl_matrix_set(A, 1, 1, 1.);
20  gsl_matrix_set(A, 1, 2, -1.);
21
22  gsl_matrix_set(A, 2, 0, -1.);
23  gsl_matrix_set(A, 2, 1, 1.);
24  gsl_matrix_set(A, 2, 2, 1.);
25
26  // vetor dos termos consts
27  gsl_vector *b = gsl_vector_alloc(3);
28
29  gsl_vector_set(b, 0, -6.);
30  gsl_vector_set(b, 1, 1.);
31  gsl_vector_set(b, 2, 0.);
32
33  // vetor solução ?
34  gsl_vector *x = gsl_vector_alloc(3);
35
36  gsl_vector_set(x, 0, -1.);
37  gsl_vector_set(x, 1, 1.);
38  gsl_vector_set(x, 2, -2.);
39
40  // verificação
41  //  $y = Ax$ 
42  gsl_vector *y = gsl_vector_alloc(3);
43  gsl_blas_dgemv(CblasNoTrans, 1., A, x, 0., y);
44
45  //  $y - b$ 
46  gsl_vector_sub(y, b);
47
48  if (gsl_blas_dnrm2(y) < 1e-14)
49      printf("x é solução do sistema.\n");
50  else
51      printf("x não é solução do sistema.\n");
52
53  gsl_matrix_free(A);
54  gsl_vector_free(b);
55  gsl_vector_free(x);
```

```
56  gsl_vector_free(y);  
57  
58  return 0;  
59 }
```

Exercício 4.3.1. Crie uma função para computar a norma de Frobenius de uma matriz A . Teste seu código com a matriz

$$A = \begin{bmatrix} -4 & -3 & -2 \\ -1 & 0 & 1 \\ 2 & 3 & 4 \end{bmatrix}. \quad (25)$$

Exercício 4.3.2. Faça um código para verificar se a matriz

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 3 & 1 \\ 1 & 2 & 0 \end{bmatrix} \quad (26)$$

é inversa de

$$B = \begin{bmatrix} 1 & 0 & 0 \\ -0.5 & 0 & 0.5 \\ 0.5 & 1 & -1.5 \end{bmatrix}. \quad (27)$$

4.4 Sistemas Lineares

A **GSL** tem **suporte à álgebra linear** pelo módulo `gsl_linalg.h`. Métodos para a decomposição LU, QR, Cholesky entre tantos outros métodos estão disponíveis. Consulte a lista completa em

<https://www.gnu.org/software/gsl/doc/html/linalg.html#linear-algebra>.

Exemplo 4.4.1. No seguinte código, computamos a solução do sistema

$$\begin{aligned} x_1 - x_2 + 2x_3 &= -6 \\ 2x_1 + x_2 - x_3 &= 1 \\ -x_1 + x_2 + x_3 &= 0 \end{aligned} \quad (28)$$

pelo método LU.

Código 13: lu.cc

```
1 #include <stdio.h>
2
3 // GSL
4 #include <gsl/gsl_vector.h>
5 #include <gsl/gsl_matrix.h>
6 #include <gsl/gsl_linalg.h>
7
8 int main()
9 {
10     // matriz dos coefs
11     gsl_matrix *A = gsl_matrix_alloc(3, 3);
12
13     gsl_matrix_set(A, 0, 0, 1.);
14     gsl_matrix_set(A, 0, 1, -1.);
15     gsl_matrix_set(A, 0, 2, 2.);
16
17     gsl_matrix_set(A, 1, 0, 2.);
18     gsl_matrix_set(A, 1, 1, 1.);
19     gsl_matrix_set(A, 1, 2, -1.);
20
21     gsl_matrix_set(A, 2, 0, -1.);
22     gsl_matrix_set(A, 2, 1, 1.);
23     gsl_matrix_set(A, 2, 2, 1.);
24
25     // vetor dos termos consts
26     gsl_vector *b = gsl_vector_alloc(3);
27
28     gsl_vector_set(b, 0, -6.);
29     gsl_vector_set(b, 1, 1.);
30     gsl_vector_set(b, 2, 0.);
31
32     // decomposição LU
33     // PA = LU
34     gsl_permutation *p = gsl_permutation_alloc(3);
35     int signum;
36     gsl_linalg_LU_decomp(A, p, &signum);
```

```
37
38 // solução
39 gsl_vector *x = gsl_vector_alloc(3);
40 gsl_linalg_LU_solve(A, p, b, x);
41
42 // imprime a solução
43 for (int i=0; i<3; ++i)
44     printf("x_%d = %g\n", i, gsl_vector_get(x, i))
45 ;
46 gsl_matrix_free(A);
47 gsl_vector_free(b);
48 gsl_permutation_free(p);
49 gsl_vector_free(x);
50
51 return 0;
52 }
```

Exercício 4.4.1. Crie sua própria função para a computação da solução de sistemas triangulares inferiores. Verifique seu código para o sistema

$$\begin{aligned} -x_1 &= 2 \\ -3x_1 + 2x_2 &= -8 \\ -x_1 + x_2 - x_3 &= 0 \end{aligned} \tag{29}$$

Exercício 4.4.2. Crie sua própria função para a computação da solução de um sistemas triangulares superiores. Verifique seu código para o sistema

$$\begin{aligned} 2x_1 - x_2 + 2x_3 &= 7 \\ 2x_2 - x_3 &= -3 \\ 3x_3 &= 3 \end{aligned} \tag{30}$$

Exercício 4.4.3. Faça um código para resolver o sistema linear

$$\begin{aligned} x_1 - x_2 + 2x_3 &= -6 \\ 2x_1 + x_2 - x_3 &= 1 \\ -x_1 + x_2 + x_3 &= 0 \end{aligned} \tag{31}$$

na sua forma matricial $A\mathbf{x} = \mathbf{b}$.

- a) Use `int gsl_linalg_LU_decomp(gsl_matrix *A, gsl_permutation *p, int *signum)` para computar a decomposição $A = LU$, onde A é a matriz de coeficientes do sistema.
- b) Use sua função criada no Exercício 4.4.1 para resolver $Ly = b$, onde b o vetor dos termos constantes do sistema.
- c) Use sua função criada no Exercício 4.4.2 para resolver $Ux = y$.

Exercício 4.4.4. Faça um código para computar a solução do sistema

$$\begin{aligned} 2x_1 - x_2 &= 0 \\ x_{i-1} - 6x_i + 4x_{i+1} &= \sin(\pi i / [2(n-1)]) \\ x_{n-1} + x_n &= 1 \end{aligned} \tag{32}$$

para $i = 2, 3, \dots, n-1, n \geq 3$. Dica: use o método `gsl_linalg_solve_tridiag`.

5 Elementos da Orientação-a-Objetos

Em construção

Notas

¹Bhaskara Akaria, 1114 - 1185, matemático e astrônomo indiano. Fonte: [Wikipédia: Bhaskara II](#).

²George Boole, 1815 - 1864, matemático britânico. Fonte: [Wikipédia: George Boole](#).

³Leonardo Fibonacci, 1170 - 1250, matemático italiano. Fonte: [Wikipédia: Leonardo Fibonacci](#).

⁴Leonardo Fibonacci, 1170 - 1250, matemático italiano. Fonte: [Wikipédia: Leonardo Fibonacci](#).

Referências

- [1] The GNU C Reference Manual, <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html>, Novembro, 2021.
- [2] GNU Scientific Library, <https://www.gnu.org/software/gsl/doc/html/index.html>, Novembro, 2021.
- [3] Soffner, R.. Algoritmos e Programação em Linguagem C. Editora Saraiva, 2013. ISBN: 978-8502207530. [Sabi+UFRGS](#).
- [4] Backes, A.. Linguagem C: Completa e Descomplicada. LTC, 2. ed., 2018. ISBN: 978-8535291063. <https://bit.ly/46Vk7Wv>.