

MC-202
Curso de C - Parte 2

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

2º semestre/2020

Como calcular a raiz quadrada de um número?

Dado um número x queremos calcular $y = \sqrt{x}$

Como calcular a raiz quadrada de um número?

Dado um número x queremos calcular $y = \sqrt{x}$

Método Babilônico (ou de Heron¹):

¹Matemático grego do século I

Como calcular a raiz quadrada de um número?

Dado um número x queremos calcular $y = \sqrt{x}$

Método Babilônico (ou de Heron¹):

1. Seja y_1 uma estimativa para $y = \sqrt{x}$

¹Matemático grego do século I

Como calcular a raiz quadrada de um número?

Dado um número x queremos calcular $y = \sqrt{x}$

Método Babilônico (ou de Heron¹):

1. Seja y_1 uma estimativa para $y = \sqrt{x}$
 - Por exemplo, $y_1 = x$

¹Matemático grego do século I

Como calcular a raiz quadrada de um número?

Dado um número x queremos calcular $y = \sqrt{x}$

Método Babilônico (ou de Heron¹):

1. Seja y_1 uma estimativa para $y = \sqrt{x}$
 - Por exemplo, $y_1 = x$
 - Quanto melhor a estimativa, mais rápido o algoritmo

¹Matemático grego do século I

Como calcular a raiz quadrada de um número?

Dado um número x queremos calcular $y = \sqrt{x}$

Método Babilônico (ou de Heron¹):

1. Seja y_1 uma estimativa para $y = \sqrt{x}$
 - Por exemplo, $y_1 = x$
 - Quanto melhor a estimativa, mais rápido o algoritmo
2. Faça $y_n = \frac{1}{2} \left(y_{n-1} + \frac{x}{y_{n-1}} \right)$

¹Matemático grego do século I

Como calcular a raiz quadrada de um número?

Dado um número x queremos calcular $y = \sqrt{x}$

Método Babilônico (ou de Heron¹):

1. Seja y_1 uma estimativa para $y = \sqrt{x}$
 - Por exemplo, $y_1 = x$
 - Quanto melhor a estimativa, mais rápido o algoritmo
2. Faça $y_n = \frac{1}{2} \left(y_{n-1} + \frac{x}{y_{n-1}} \right)$
3. Se $|y_n - y_{n-1}|$ for “grande”, volte para 2

¹Matemático grego do século I

Como calcular a raiz quadrada de um número?

Dado um número x queremos calcular $y = \sqrt{x}$

Método Babilônico (ou de Heron¹):

1. Seja y_1 uma estimativa para $y = \sqrt{x}$
 - Por exemplo, $y_1 = x$
 - Quanto melhor a estimativa, mais rápido o algoritmo
2. Faça $y_n = \frac{1}{2} \left(y_{n-1} + \frac{x}{y_{n-1}} \right)$
3. Se $|y_n - y_{n-1}|$ for “grande”, volte para 2
4. Devolva y_n

¹Matemático grego do século I

Código em Python

```
1 ERRO = 1e-12
2
3
4 def square_root(x):
5     y = x
6     erro_pequeno = False
7     while not erro_pequeno:
8         anterior = y
9         y = (y + x / y) / 2
10        if abs(anterior - y) <= ERRO:
11            erro_pequeno = True
12    return y
13
14
15 print("Entre com o numero:")
16 x = float(input())
17 print("Raiz quadrada:", square_root(x))
```

Código em Python

```
1 ERRO = 1e-12
2
3
4 def square_root(x):
5     y = x
6     erro_pequeno = False
7     while not erro_pequeno:
8         anterior = y
9         y = (y + x / y) / 2
10        if abs(anterior - y) <= ERRO:
11            erro_pequeno = True
12    return y
13
14
15 print("Entre com o numero:")
16 x = float(input())
17 print("Raiz quadrada:", square_root(x))
```

Traduzindo para C:

Código em Python

```
1 ERRO = 1e-12
2
3
4 def square_root(x):
5     y = x
6     erro_pequeno = False
7     while not erro_pequeno:
8         anterior = y
9         y = (y + x / y) / 2
10        if abs(anterior - y) <= ERRO:
11            erro_pequeno = True
12    return y
13
14
15 print("Entre com o numero:")
16 x = float(input())
17 print("Raiz quadrada:", square_root(x))
```

Traduzindo para C:

- Como representar números reais em C?

Código em Python

```
1 ERRO = 1e-12
2
3
4 def square_root(x):
5     y = x
6     erro_pequeno = False
7     while not erro_pequeno:
8         anterior = y
9         y = (y + x / y) / 2
10        if abs(anterior - y) <= ERRO:
11            erro_pequeno = True
12    return y
13
14
15 print("Entre com o numero:")
16 x = float(input())
17 print("Raiz quadrada:", square_root(x))
```

Traduzindo para C:

- Como representar números reais em C?
- Como ler e escrever tais números?

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples
- Em geral, usa 32 bits

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples
- Em geral, usa 32 bits
- A leitura e a escrita é feita com `%f`

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples
- Em geral, usa 32 bits
- A leitura e a escrita é feita com `%f`
 - Ou usando notação científica: `%e`

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples
- Em geral, usa 32 bits
- A leitura e a escrita é feita com `%f`
 - Ou usando notação científica: `%e`
 - Ou o mais curto dos dois: `%g`

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples
- Em geral, usa 32 bits
- A leitura e a escrita é feita com `%f`
 - Ou usando notação científica: `%e`
 - Ou o mais curto dos dois: `%g`

Temos também o tipo `double`

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples
- Em geral, usa 32 bits
- A leitura e a escrita é feita com `%f`
 - Ou usando notação científica: `%e`
 - Ou o mais curto dos dois: `%g`

Temos também o tipo `double`

- Número de ponto flutuante de precisão dupla

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples
- Em geral, usa 32 bits
- A leitura e a escrita é feita com `%f`
 - Ou usando notação científica: `%e`
 - Ou o mais curto dos dois: `%g`

Temos também o tipo `double`

- Número de ponto flutuante de precisão dupla
- Em geral, usa 64 bits

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples
- Em geral, usa 32 bits
- A leitura e a escrita é feita com `%f`
 - Ou usando notação científica: `%e`
 - Ou o mais curto dos dois: `%g`

Temos também o tipo `double`

- Número de ponto flutuante de precisão dupla
- Em geral, usa 64 bits
 - Maior precisão, mas é mais lento e gasta mais memória

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples
- Em geral, usa 32 bits
- A leitura e a escrita é feita com `%f`
 - Ou usando notação científica: `%e`
 - Ou o mais curto dos dois: `%g`

Temos também o tipo `double`

- Número de ponto flutuante de precisão dupla
- Em geral, usa 64 bits
 - Maior precisão, mas é mais lento e gasta mais memória
 - É o mais usado em geral

O tipo `float` e o tipo `double`

Em C, um número real é representado usando o tipo `float`

- Número de ponto flutuante de precisão simples
- Em geral, usa 32 bits
- A leitura e a escrita é feita com `%f`
 - Ou usando notação científica: `%e`
 - Ou o mais curto dos dois: `%g`

Temos também o tipo `double`

- Número de ponto flutuante de precisão dupla
- Em geral, usa 64 bits
 - Maior precisão, mas é mais lento e gasta mais memória
 - É o mais usado em geral
- A leitura/impressão é feita com `%lf`, `%le` ou `%lg`

Código em C

```
1 #include <stdio.h>
2 #include <math.h>
3 #define ERRO 1e-12
4
5 double square_root(double x) {
6     double y = x, anterior;
7     do {
8         anterior = y;
9         y = (y + x / y) / 2;
10    } while (fabs(anterior - y) > ERRO);
11    return y;
12 }
13
14 int main() {
15     double x;
16     printf("Entre com o numero:\n");
17     scanf("%lf", &x);
18     printf("Raiz quadrada: %lf\n", square_root(x));
19     return 0;
20 }
```

Código em C

```
1 #include <stdio.h>
2 #include <math.h>
3 #define ERRO 1e-12
4
5 double square_root(double x) {
6     double y = x, anterior;
7     do {
8         anterior = y;
9         y = (y + x / y) / 2;
10    } while (fabs(anterior - y) > ERRO);
11    return y;
12 }
13
14 int main() {
15     double x;
16     printf("Entre com o numero:\n");
17     scanf("%lf", &x);
18     printf("Raiz quadrada: %lf\n", square_root(x));
19     return 0;
20 }
```

A biblioteca `math.h` contém várias funções matemáticas

Código em C

```
1 #include <stdio.h>
2 #include <math.h>
3 #define ERRO 1e-12
4
5 double square_root(double x) {
6     double y = x, anterior;
7     do {
8         anterior = y;
9         y = (y + x / y) / 2;
10    } while (fabs(anterior - y) > ERRO);
11    return y;
12 }
13
14 int main() {
15     double x;
16     printf("Entre com o numero:\n");
17     scanf("%lf", &x);
18     printf("Raiz quadrada: %lf\n", square_root(x));
19     return 0;
20 }
```

A biblioteca `math.h` contém várias funções matemáticas

- `fabs` devolve o valor absoluto de um número `double`

Manual de fabs

Execute `man fabs` no terminal para ver a documentação

Name

`fabs, fabsf, fabsl` - absolute value of floating-point number

Synopsis

```
#include <math.h>
```

```
double fabs(double x);  
float fabsf(float x);  
long double fabsl(long double x);
```

Link with `-lm`.

Feature Test Macro Requirements for `glibc` (see `feature_test_macros(7)`):

```
fabsf(), fabsl():  
_BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 600 || _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L;  
or cc -std=c99
```

Description

The `fabs()` functions return the absolute value of the floating-point number `x`.

Return Value

These functions return the absolute value of `x`.

If `x` is a NaN, a NaN is returned.

If `x` is `-0`, `+0` is returned.

If `x` is negative infinity or positive infinity, positive infinity is returned.

...

Um cuidado!

Em C, as funções

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`
 - por exemplo, escreva `(double) x`

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`
 - por exemplo, escreva `(double) x`
 - é o que chamamos de *casting*

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`
 - por exemplo, escreva `(double) x`
 - é o que chamamos de *casting*
 - `1` é convertido para `1.0`

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`
 - por exemplo, escreva `(double) x`
 - é o que chamamos de *casting*
 - `1` é convertido para `1.0`
- valor `double` pode ser convertido para `int`

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`
 - por exemplo, escreva `(double) x`
 - é o que chamamos de *casting*
 - `1` é convertido para `1.0`
- valor `double` pode ser convertido para `int`
 - `1.0` é convertido para `1`

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`
 - por exemplo, escreva `(double) x`
 - é o que chamamos de *casting*
 - `1` é convertido para `1.0`
- valor `double` pode ser convertido para `int`
 - `1.0` é convertido para `1`
 - `1.937` é convertido para `1`

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`
 - por exemplo, escreva `(double) x`
 - é o que chamamos de *casting*
 - `1` é convertido para `1.0`
- valor `double` pode ser convertido para `int`
 - `1.0` é convertido para `1`
 - `1.937` é convertido para `1`

Temos duas funções diferentes que calculam valor absoluto:

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`
 - por exemplo, escreva `(double) x`
 - é o que chamamos de *casting*
 - `1` é convertido para `1.0`
- valor `double` pode ser convertido para `int`
 - `1.0` é convertido para `1`
 - `1.937` é convertido para `1`

Temos duas funções diferentes que calculam valor absoluto:

- `int abs(int x)` e `double fabs(double x)`

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`
 - por exemplo, escreva `(double) x`
 - é o que chamamos de *casting*
 - `1` é convertido para `1.0`
- valor `double` pode ser convertido para `int`
 - `1.0` é convertido para `1`
 - `1.937` é convertido para `1`

Temos duas funções diferentes que calculam valor absoluto:

- `int abs(int x)` e `double fabs(double x)`
- Mas, `abs(-7.9)` é `7` e `fabs(-3)` é `3.0`. Por que?

Um cuidado!

Em C, as funções

- recebem parâmetros de um tipo específico
- devolvem resultados de um tipo específico

E tipos podem ser convertidos:

- valor `int` pode ser convertido para `double`
 - por exemplo, escreva `(double) x`
 - é o que chamamos de *casting*
 - `1` é convertido para `1.0`
- valor `double` pode ser convertido para `int`
 - `1.0` é convertido para `1`
 - `1.937` é convertido para `1`

Temos duas funções diferentes que calculam valor absoluto:

- `int abs(int x)` e `double fabs(double x)`
- Mas, `abs(-7.9)` é `7` e `fabs(-3)` é `3.0`. Por que?
 - os valores são convertidos automaticamente

Divisão inteira e divisão real

Python

C

Divisão inteira e divisão real

	Python	C
6 / 4	1.5	1

Divisão inteira e divisão real

	Python	C
$6 / 4$	1.5	1
$6.0 / 4.0$	1.5	1.5
$6.0 / 4$	1.5	1.5
$6 / 4.0$	1.5	1.5

Divisão inteira e divisão real

	Python	C
<code>6 / 4</code>	1.5	1
<code>6.0 / 4.0</code>	1.5	1.5
<code>6.0 / 4</code>	1.5	1.5
<code>6 / 4.0</code>	1.5	1.5
<code>6 // 4</code>	1	

Divisão inteira e divisão real

	Python	C
$6 / 4$	1.5	1
$6.0 / 4.0$	1.5	1.5
$6.0 / 4$	1.5	1.5
$6 / 4.0$	1.5	1.5
$6 // 4$	1	
$6.0 // 4.0$	1.0	
$6.0 // 4$	1.0	
$6 // 4.0$	1.0	

Divisão inteira e divisão real

	Python	C
$6 / 4$	1.5	1
$6.0 / 4.0$	1.5	1.5
$6.0 / 4$	1.5	1.5
$6 / 4.0$	1.5	1.5
$6 // 4$	1	
$6.0 // 4.0$	1.0	Op. não existe
$6.0 // 4$	1.0	
$6 // 4.0$	1.0	

Divisão inteira e divisão real

	Python	C
$6 / 4$	1.5	1
$6.0 / 4.0$	1.5	1.5
$6.0 / 4$	1.5	1.5
$6 / 4.0$	1.5	1.5
$6 // 4$	1	
$6.0 // 4.0$	1.0	Op. não existe
$6.0 // 4$	1.0	
$6 // 4.0$	1.0	
$6 \% 4$	2	2

Divisão inteira e divisão real

	Python	C
$6 / 4$	1.5	1
$6.0 / 4.0$	1.5	1.5
$6.0 / 4$	1.5	1.5
$6 / 4.0$	1.5	1.5
$6 // 4$	1	
$6.0 // 4.0$	1.0	Op. não existe
$6.0 // 4$	1.0	
$6 // 4.0$	1.0	
$6 \% 4$	2	2
$6.0 \% 4.0$	2.0	erro de compilação
$6.0 \% 4$	2.0	erro de compilação
$6 \% 4.0$	2.0	erro de compilação

Divisão inteira e divisão real

	Python	C
$6 / 4$	1.5	1
$6.0 / 4.0$	1.5	1.5
$6.0 / 4$	1.5	1.5
$6 / 4.0$	1.5	1.5
$6 // 4$	1	
$6.0 // 4.0$	1.0	Op. não existe
$6.0 // 4$	1.0	
$6 // 4.0$	1.0	
$6 \% 4$	2	2
$6.0 \% 4.0$	2.0	erro de compilação
$6.0 \% 4$	2.0	erro de compilação
$6 \% 4.0$	2.0	erro de compilação

Se necessário, fazemos *casting*:

Divisão inteira e divisão real

	Python	C
$6 / 4$	1.5	1
$6.0 / 4.0$	1.5	1.5
$6.0 / 4$	1.5	1.5
$6 / 4.0$	1.5	1.5
$6 // 4$	1	
$6.0 // 4.0$	1.0	Op. não existe
$6.0 // 4$	1.0	
$6 // 4.0$	1.0	
$6 \% 4$	2	2
$6.0 \% 4.0$	2.0	erro de compilação
$6.0 \% 4$	2.0	erro de compilação
$6 \% 4.0$	2.0	erro de compilação

Se necessário, fazemos *casting*:

- Se x vale 6 e y vale 4, então $(double)x / y$ é 1.5

Código em C

```
1 #include <stdio.h>
2 #include <math.h>
3 #define ERRO 1e-12
4
5 double square_root(double x) {
6     double y = x, anterior;
7     do {
8         anterior = y;
9         y = (y + x / y) / 2;
10    } while (fabs(anterior - y) > ERRO);
11    return y;
12 }
13
14 int main() {
15     double x;
16     printf("Entre com o numero:\n");
17     scanf("%lf", &x);
18     printf("Raiz quadrada: %lf\n", square_root(x));
19     return 0;
20 }
```

Código em C

```
1 #include <stdio.h>
2 #include <math.h>
3 #define ERRO 1e-12
4
5 double square_root(double x) {
6     double y = x, anterior;
7     do {
8         anterior = y;
9         y = (y + x / y) / 2;
10    } while (fabs(anterior - y) > ERRO);
11    return y;
12 }
13
14 int main() {
15     double x;
16     printf("Entre com o numero:\n");
17     scanf("%lf", &x);
18     printf("Raiz quadrada: %lf\n", square_root(x));
19     return 0;
20 }
```

A diretiva **#define** cria uma macro

Código em C

```
1 #include <stdio.h>
2 #include <math.h>
3 #define ERRO 1e-12
4
5 double square_root(double x) {
6     double y = x, anterior;
7     do {
8         anterior = y;
9         y = (y + x / y) / 2;
10    } while (fabs(anterior - y) > ERRO);
11    return y;
12 }
13
14 int main() {
15     double x;
16     printf("Entre com o numero:\n");
17     scanf("%lf", &x);
18     printf("Raiz quadrada: %lf\n", square_root(x));
19     return 0;
20 }
```

A diretiva **#define** cria uma macro

- Onde aparecer a palavra **ERRO**, substitua por **1e-12**

Código em C

```
1 #include <stdio.h>
2 #include <math.h>
3 #define ERRO 1e-12
4
5 double square_root(double x) {
6     double y = x, anterior;
7     do {
8         anterior = y;
9         y = (y + x / y) / 2;
10    } while (fabs(anterior - y) > ERRO);
11    return y;
12 }
13
14 int main() {
15     double x;
16     printf("Entre com o numero:\n");
17     scanf("%lf", &x);
18     printf("Raiz quadrada: %lf\n", square_root(x));
19     return 0;
20 }
```

Código em C

```
1 #include <stdio.h>
2 #include <math.h>
3 #define ERRO 1e-12
4
5 double square_root(double x) {
6     double y = x, anterior;
7     do {
8         anterior = y;
9         y = (y + x / y) / 2;
10    } while (fabs(anterior - y) > ERRO);
11    return y;
12 }
13
14 int main() {
15     double x;
16     printf("Entre com o numero:\n");
17     scanf("%lf", &x);
18     printf("Raiz quadrada: %lf\n", square_root(x));
19     return 0;
20 }
```

Estamos usando `do ... while` que não existe em Python

Código em C

```
1 #include <stdio.h>
2 #include <math.h>
3 #define ERRO 1e-12
4
5 double square_root(double x) {
6     double y = x, anterior;
7     do {
8         anterior = y;
9         y = (y + x / y) / 2;
10    } while (fabs(anterior - y) > ERRO);
11    return y;
12 }
13
14 int main() {
15     double x;
16     printf("Entre com o numero:\n");
17     scanf("%lf", &x);
18     printf("Raiz quadrada: %lf\n", square_root(x));
19     return 0;
20 }
```

Estamos usando `do ... while` que não existe em Python

- Calculamos `y` e `anterior` antes de testar a condição

Multiplicação de Matrizes Reais

Dadas duas matrizes A e B em $\mathbb{R}^{n \times n}$, calcular $C = A \times B$

Multiplicação de Matrizes Reais

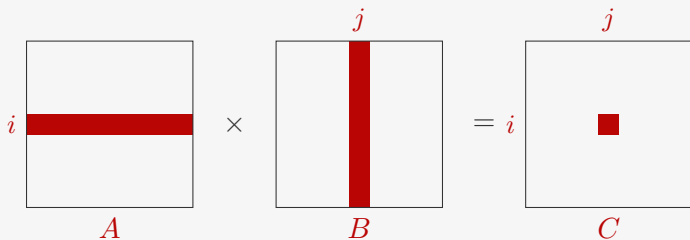
Dadas duas matrizes A e B em $\mathbb{R}^{n \times n}$, calcular $C = A \times B$

Relembrando...

Multiplicação de Matrizes Reais

Dadas duas matrizes A e B em $\mathbb{R}^{n \times n}$, calcular $C = A \times B$

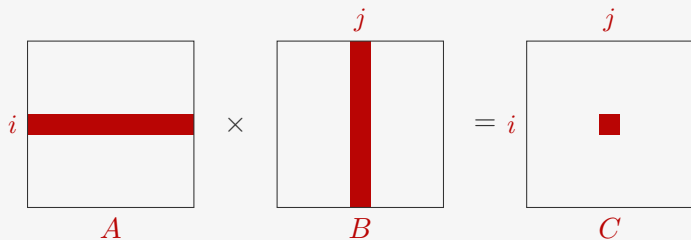
Relembrando...



Multiplicação de Matrizes Reais

Dadas duas matrizes A e B em $\mathbb{R}^{n \times n}$, calcular $C = A \times B$

Relembrando...



C_{ij} é o produto interno da linha i de A com a coluna j de B

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$$

Primeiro em Python...

Uma boa forma de programar é pensar nas pequenas tarefas

Primeiro em Python...

Uma boa forma de programar é pensar nas pequenas tarefas

```
1 n = int(input())
2 A = le_matriz_quadrada(n)
3 B = le_matriz_quadrada(n)
4 C = multiplica_quadradas(A, B, n)
5 imprime_matriz_quadrada(C, n)
```

Primeiro em Python...

Uma boa forma de programar é pensar nas pequenas tarefas

```
1 n = int(input())
2 A = le_matriz_quadrada(n)
3 B = le_matriz_quadrada(n)
4 C = multiplica_quadradas(A, B, n)
5 imprime_matriz_quadrada(C, n)
```

Basta então criar as três funções que faltam

Funcões

```
1 def le_matriz_quadrada(n):
2     M = []
3     for i in range(n):
4         M.append([])
5         for j in range(n):
6             M[i].append(float(input()))
7     return M
8
9 def multiplica_quadradas(A, B, n):
10    C = [[0 for i in range(n)] for j in range(n)]
11    for i in range(n):
12        for j in range(n):
13            for k in range(n):
14                C[i][j] += A[i][k] * B[k][j]
15    return C
16
17 def imprime_matriz_quadrada(M, n):
18    for i in range(n):
19        for j in range(n):
20            print(M[i][j], end=' ')
21    print("")
```

Funcões

```
1 def le_matriz_quadrada(n):
2     M = []
3     for i in range(n):
4         M.append([])
5         for j in range(n):
6             M[i].append(float(input()))
7     return M
8
9 def multiplica_quadradas(A, B, n):
10    C = [[0 for i in range(n)] for j in range(n)]
11    for i in range(n):
12        for j in range(n):
13            for k in range(n):
14                C[i][j] += A[i][k] * B[k][j]
15    return C
16
17 def imprime_matriz_quadrada(M, n):
18    for i in range(n):
19        for j in range(n):
20            print(M[i][j], end=' ')
21    print("")
```

Em C, não há *list comprehension*!

Impressão

```
1 #define MAX 1000
2
3 void imprime_matriz_quadrada(double M[][MAX], int n) {
4     int i, j;
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < n; j++)
7             printf("%lf ", M[i][j]);
8         printf("\n");
9     }
10 }
```

Impressão

```
1 #define MAX 1000
2
3 void imprime_matriz_quadrada(double M[][MAX], int n) {
4     int i, j;
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < n; j++)
7             printf("%lf ", M[i][j]);
8         printf("\n");
9     }
10 }
```

Matrizes têm sempre um tamanho definido:

Impressão

```
1 #define MAX 1000
2
3 void imprime_matriz_quadrada(double M[][MAX], int n) {
4     int i, j;
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < n; j++)
7             printf("%lf ", M[i][j]);
8         printf("\n");
9     }
10 }
```

Matrizes têm sempre um tamanho definido:

- Estamos usando um `#define` para esse tamanho

Impressão

```
1 #define MAX 1000
2
3 void imprime_matriz_quadrada(double M[][MAX], int n) {
4     int i, j;
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < n; j++)
7             printf("%lf ", M[i][j]);
8         printf("\n");
9     }
10 }
```

Matrizes têm sempre um tamanho definido:

- Estamos usando um `#define` para esse tamanho
- E temos que passar o número de colunas para a função

Impressão

```
1 #define MAX 1000
2
3 void imprime_matriz_quadrada(double M[][MAX], int n) {
4     int i, j;
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < n; j++)
7             printf("%lf ", M[i][j]);
8         printf("\n");
9     }
10 }
```

Matrizes têm sempre um tamanho definido:

- Estamos usando um `#define` para esse tamanho
- E temos que passar o número de colunas para a função
 - Passar o número de linhas é opcional

Impressão

```
1 #define MAX 1000
2
3 void imprime_matriz_quadrada(double M[][MAX], int n) {
4     int i, j;
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < n; j++)
7             printf("%lf ", M[i][j]);
8         printf("\n");
9     }
10 }
```

Matrizes têm sempre um tamanho definido:

- Estamos usando um `#define` para esse tamanho
- E temos que passar o número de colunas para a função
 - Passar o número de linhas é opcional

Note que um `for` usa `{` e `}`, mas o outro não...

Impressão

```
1 #define MAX 1000
2
3 void imprime_matriz_quadrada(double M[][MAX], int n) {
4     int i, j;
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < n; j++)
7             printf("%lf ", M[i][j]);
8         printf("\n");
9     }
10 }
```

Matrizes têm sempre um tamanho definido:

- Estamos usando um `#define` para esse tamanho
- E temos que passar o número de colunas para a função
 - Passar o número de linhas é opcional

Note que um `for` usa `{` e `}`, mas o outro não...

- E se não usarmos `{` e `}` no primeiro `for`?

Leitura

Não podemos devolver matrizes...

Leitura

Não podemos devolver matrizes...

- Mas podemos passá-las como parâmetro e modificá-las

Leitura

Não podemos devolver matrizes...

- Mas podemos passá-las como parâmetro e modificá-las
- O mesmo que fizemos para vetores

Leitura

Não podemos devolver matrizes...

- Mas podemos passá-las como parâmetro e modificá-las
- O mesmo que fizemos para vetores

```
1 void le_matriz_quadrada(double M[][MAX], int n) {  
2     int i, j;  
3     for (i = 0; i < n; i++)  
4         for (j = 0; j < n; j++)  
5             scanf("%lf", &M[i][j]);  
6 }
```


Leitura

Não podemos devolver matrizes...

- Mas podemos passá-las como parâmetro e modificá-las
- O mesmo que fizemos para vetores

```
1 void le_matriz_quadrada(double M[][MAX], int n) {
2     int i, j;
3     for (i = 0; i < n; i++)
4         for (j = 0; j < n; j++)
5             scanf("%lf", &M[i][j]);
6 }
```

Note que ambos os `for`s não usam `{` e `}`

Leitura

Não podemos devolver matrizes...

- Mas podemos passá-las como parâmetro e modificá-las
- O mesmo que fizemos para vetores

```
1 void le_matriz_quadrada(double M[][MAX], int n) {
2     int i, j;
3     for (i = 0; i < n; i++)
4         for (j = 0; j < n; j++)
5             scanf("%lf", &M[i][j]);
6 }
```

Note que ambos os `for`s não usam `{` e `}`

- As linhas 4 e 5 correspondem a um único comando!

Multiplicação

```
1 void multiplica_quadradas(double A[][MAX], double B[][MAX],
2                           double C[][MAX], int n) {
3     int i, j, k;
4     for (i = 0; i < n; i++)
5         for (j = 0; j < n; j++) {
6             C[i][j] = 0;
7             for (k = 0; k < n; k++)
8                 C[i][j] += A[i][k] * B[k][j];
9         }
10 }
```

Multiplicação

```
1 void multiplica_quadradas(double A[][MAX], double B[][MAX],
2                           double C[][MAX], int n) {
3     int i, j, k;
4     for (i = 0; i < n; i++)
5         for (j = 0; j < n; j++) {
6             C[i][j] = 0;
7             for (k = 0; k < n; k++)
8                 C[i][j] += A[i][k] * B[k][j];
9         }
10 }
```

C é passada como parâmetro para ser alterada

Multiplicação

```
1 void multiplica_quadradas(double A[][MAX], double B[][MAX],
2                           double C[][MAX], int n) {
3     int i, j, k;
4     for (i = 0; i < n; i++)
5         for (j = 0; j < n; j++) {
6             C[i][j] = 0;
7             for (k = 0; k < n; k++)
8                 C[i][j] += A[i][k] * B[k][j];
9         }
10 }
```

C é passada como parâmetro para ser alterada

O **for** da linha 4 não precisa de **{** e **}**

Multiplicação

```
1 void multiplica_quadradas(double A[][MAX], double B[][MAX],
2                           double C[][MAX], int n) {
3     int i, j, k;
4     for (i = 0; i < n; i++)
5         for (j = 0; j < n; j++) {
6             C[i][j] = 0;
7             for (k = 0; k < n; k++)
8                 C[i][j] += A[i][k] * B[k][j];
9         }
10 }
```

C é passada como parâmetro para ser alterada

O **for** da linha **4** não precisa de **{** e **}**

- Tem uma única expressão dentro dele, o **for** de **5-9**

Multiplicação

```
1 void multiplica_quadradas(double A[][MAX], double B[][MAX],
2                           double C[][MAX], int n) {
3     int i, j, k;
4     for (i = 0; i < n; i++)
5         for (j = 0; j < n; j++) {
6             C[i][j] = 0;
7             for (k = 0; k < n; k++)
8                 C[i][j] += A[i][k] * B[k][j];
9         }
10 }
```

C é passada como parâmetro para ser alterada

O **for** da linha **4** não precisa de **{** e **}**

- Tem uma única expressão dentro dele, o **for** de **5-9**
- O **{** e **}** pode ser omitido para encurtar o código

Multiplicação

```
1 void multiplica_quadradas(double A[][MAX], double B[][MAX],
2                           double C[][MAX], int n) {
3     int i, j, k;
4     for (i = 0; i < n; i++)
5         for (j = 0; j < n; j++) {
6             C[i][j] = 0;
7             for (k = 0; k < n; k++)
8                 C[i][j] += A[i][k] * B[k][j];
9         }
10 }
```

C é passada como parâmetro para ser alterada

O **for** da linha 4 não precisa de **{** e **}**

- Tem uma única expressão dentro dele, o **for** de 5-9
- O **{** e **}** pode ser omitido para encurtar o código
- Ou pode ser colocado para deixar explícito

Multiplicação

```
1 void multiplica_quadradas(double A[][MAX], double B[][MAX],
2                           double C[][MAX], int n) {
3     int i, j, k;
4     for (i = 0; i < n; i++)
5         for (j = 0; j < n; j++) {
6             C[i][j] = 0;
7             for (k = 0; k < n; k++)
8                 C[i][j] += A[i][k] * B[k][j];
9         }
10 }
```

C é passada como parâmetro para ser alterada

O **for** da linha 4 não precisa de **{** e **}**

- Tem uma única expressão dentro dele, o **for** de 5-9
- O **{** e **}** pode ser omitido para encurtar o código
- Ou pode ser colocado para deixar explícito
 - Faça como te deixar mais confortável!

Multiplicação

```
1 void multiplica_quadradas(double A[][MAX], double B[][MAX],
2                           double C[][MAX], int n) {
3     int i, j, k;
4     for (i = 0; i < n; i++)
5         for (j = 0; j < n; j++) {
6             C[i][j] = 0;
7             for (k = 0; k < n; k++)
8                 C[i][j] += A[i][k] * B[k][j];
9         }
10 }
```

C é passada como parâmetro para ser alterada

O **for** da linha 4 não precisa de **{** e **}**

- Tem uma única expressão dentro dele, o **for** de 5-9
- O **{** e **}** pode ser omitido para encurtar o código
- Ou pode ser colocado para deixar explícito
 - Faça como te deixar mais confortável!
 - E cuidado para a indentação incorreta não te confundir!

A função main

```
1 int main() {
2     int n;
3     double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
4     scanf("%d", &n);
5     le_matriz_quadrada(A, n);
6     le_matriz_quadrada(B, n);
7     multiplica_quadradas(A, B, C, n);
8     imprime_matriz_quadrada(C, n);
9     return 0;
10 }
```

A função main

```
1 int main() {
2     int n;
3     double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
4     scanf("%d", &n);
5     le_matriz_quadrada(A, n);
6     le_matriz_quadrada(B, n);
7     multiplica_quadradas(A, B, C, n);
8     imprime_matriz_quadrada(C, n);
9     return 0;
10 }
```

Note que como é feita a declaração das matrizes:

A função main

```
1 int main() {
2     int n;
3     double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
4     scanf("%d", &n);
5     le_matriz_quadrada(A, n);
6     le_matriz_quadrada(B, n);
7     multiplica_quadradas(A, B, C, n);
8     imprime_matriz_quadrada(C, n);
9     return 0;
10 }
```

Note que como é feita a declaração das matrizes:

- `double A[MAX][MAX];` declara uma matriz

A função main

```
1 int main() {
2     int n;
3     double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
4     scanf("%d", &n);
5     le_matriz_quadrada(A, n);
6     le_matriz_quadrada(B, n);
7     multiplica_quadradas(A, B, C, n);
8     imprime_matriz_quadrada(C, n);
9     return 0;
10 }
```

Note que como é feita a declaração das matrizes:

- `double A[MAX][MAX];` declara uma matriz
 - `MAX × MAX`

A função main

```
1 int main() {
2     int n;
3     double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
4     scanf("%d", &n);
5     le_matriz_quadrada(A, n);
6     le_matriz_quadrada(B, n);
7     multiplica_quadradas(A, B, C, n);
8     imprime_matriz_quadrada(C, n);
9     return 0;
10 }
```

Note que como é feita a declaração das matrizes:

- `double A[MAX][MAX];` declara uma matriz
 - `MAX` \times `MAX`
 - de `doubles`

A função main

```
1 int main() {
2     int n;
3     double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
4     scanf("%d", &n);
5     le_matriz_quadrada(A, n);
6     le_matriz_quadrada(B, n);
7     multiplica_quadradas(A, B, C, n);
8     imprime_matriz_quadrada(C, n);
9     return 0;
10 }
```

Note que como é feita a declaração das matrizes:

- `double A[MAX][MAX];` declara uma matriz
 - `MAX` \times `MAX`
 - de `doubles`
- Números de linhas e colunas podem ser diferentes

A função main

```
1 int main() {
2     int n;
3     double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
4     scanf("%d", &n);
5     le_matriz_quadrada(A, n);
6     le_matriz_quadrada(B, n);
7     multiplica_quadradas(A, B, C, n);
8     imprime_matriz_quadrada(C, n);
9     return 0;
10 }
```

Note que como é feita a declaração das matrizes:

- `double A[MAX][MAX];` declara uma matriz
 - `MAX` \times `MAX`
 - de `doubles`
- Números de linhas e colunas podem ser diferentes
 - 10 linhas e 3 colunas: `int matriz[10][3];`

A função main

```
1 int main() {
2     int n;
3     double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
4     scanf("%d", &n);
5     le_matriz_quadrada(A, n);
6     le_matriz_quadrada(B, n);
7     multiplica_quadradas(A, B, C, n);
8     imprime_matriz_quadrada(C, n);
9     return 0;
10 }
```

Note que como é feita a declaração das matrizes:

- `double A[MAX][MAX];` declara uma matriz
 - `MAX` \times `MAX`
 - de `doubles`
- Números de linhas e colunas podem ser diferentes
 - 10 linhas e 3 colunas: `int matriz[10][3];`
- Podemos também declarar matrizes multidimensionais

A função main

```
1 int main() {
2     int n;
3     double A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
4     scanf("%d", &n);
5     le_matriz_quadrada(A, n);
6     le_matriz_quadrada(B, n);
7     multiplica_quadradas(A, B, C, n);
8     imprime_matriz_quadrada(C, n);
9     return 0;
10 }
```

Note que como é feita a declaração das matrizes:

- `double A[MAX][MAX];` declara uma matriz
 - `MAX` \times `MAX`
 - de `doubles`
- Números de linhas e colunas podem ser diferentes
 - 10 linhas e 3 colunas: `int matriz[10][3];`
- Podemos também declarar matrizes multidimensionais
 - `double M[10][5][7];`

Exercício

Dado um tempo t em segundos, converta para a representação horas-minutos-segundos.

Exemplo: $123456s$ é $34h17m36s$

Exercício

Dada uma aplicação financeira com:

- depósito inicial ini reais,
- depósitos mensais de $mensal$ reais,
- juros mensais de j por cento ao mês
- e um número de meses t ,

calcule o valor final da aplicação.