Computação Científica em Linguagem C

Um Livro Colaborativo

30 de julho de 2018

Organizadores

Esequia Sauter - UFRGS

Fabio Souto de Azevedo - UFRGS

Pedro Henrique de Almeida Konzen - UFRGS

Colaboradores

Este material é fruto da escrita colaborativa. Veja a lista de colaboradores em:

https://github.com/reamat/ComputacaoCientifica/graphs/contributors

Para saber mais como participar, visite o site oficial do projeto:

https://www.ufrgs.br/reamat/ComputacaoCientifica

ou comece agora mesmo visitando nosso repositório GitHub:

https://github.com/reamat/ComputacaoCientifica

Licença

Nota dos organizadores

Este livro surgiu como notas de aula em uma disciplina de Computação Científica da Pós-Graduação em Matemática Aplicada da Universidade Federal do Rio Grande do Sul no ano de 2017.

O sucesso do projeto depende da colaboração! Edite você mesmo o livro, dê sugestões ou nos avise de erros e imprecisões. Toda a colaboração é bem vinda. Saiba mais visitando o site oficial do projeto:

https://www.ufrgs.br/reamat

Nós preparamos uma série de ações para ajudá-lo a participar. Em primeiro lugar, o acesso irrestrito ao livro pode se dar através do site oficial do projeto.

PDF

Além disso, o livro está escrito em código LaTeX disponível em repositório GitHub público.

Nada disso estaria completo sem uma licença apropriada à colaboração. Por isso, escolhemos disponibilizar o material do livro sob licença Creative Commons Atribuição-Compartilh
Ou seja, você pode copiar, redistribuir, alterar e construir um novo material para qualquer uso, inclusive comercial. Leia a licença para maiores informações.

Desejamos-lhe ótimas colaborações!

Prefácio

Este livro busca introduzir a linguagem de programação C no contexto de computação científica. Alguns livros de programação em linguagem C, tais como [2, 3, 6, 7], estão focados no básico da linguagem, outros, tais como [1, 8], estão focados nos métodos numéricos. O excelente livro [4] faz as duas coisas, no entanto, ele não apresenta as bibliotecas para computação científica. Aqui, o interesse é trabalhar a linguagem de programação C, dando foco aos itens de interesse para resolver problemas em computação científica, entre eles as bibliotecas GSL e LAPACK.

Sumário

Capítulo 1

Introdução à programação científica em linguagem C

1.1 Linguagem C

Linguagem de programação é um conjunto de regras sintáticas e semânticas usadas para construir um programa de computador. Um programa é uma sequência de instruções que podem ser interpretada por um computador ou convertida em linguagem de máquina. As linguagens de programação são classificadas quanto ao nível de abstração:

- Linguagens de programação de baixo nível são aquelas cujos símbolos são uma representação direta do código de máquina. Exemplo: Assembly.
- Linguagem de programação de médio nível são aquelas que possuem símbolos que são um representação direta do código de máquina, mas também símbolos complexos que são convertidos por um compilador. Exemplo: C, C++
- Linguagem de programação de alto nível são aquelas composta de símbolos mais complexos, inteligível pelo ser humano e não-executável diretamente pela máquina. Exemplo: Pascal, Fortran, Java.

Este livro de programação e computação científica está desenvolvido em linguagem de programação C. Essa é uma linguagem de programação compilada, isto é, precisa um compilador que converte o programa para um código em linguagem de máquina. O sistema operacional Linux já possui o compiador GCC instalado por padrão, mas os usuários do sistema operacional Windows deverão baixar e instalar. No caso do Linux, o código poderá ser escrito em qualquer programa que edite texto, tais como Gedit ou Kile.

1.2 Hello, world!

Uma maneira eficaz para aprender uma linguagem de programação é fazendo programas nela. Portanto, vamos ao nosso primeiro programa.

Exemplo 1.2.1. Faça um programa que escreva as palavras Hello, world!

Abra o programa Gedit e numa janela limpa escreva o seguinte código:

```
#include <stdio.h>

main()
{
printf("hello, world!\n");
}
```

Salve-o com o nome hello_world.c. Para compilar o programa, abra um terminal, vá até a pasta onde está salvo o arquivo e digite a seguinte linha de comando:

```
gcc hello_world.c
```

Finalmente, para executar o programa, digite a seguinte linha de comando:

```
./a.out
```

Comentários sobre o programa do exemplo 1.2.1:

- O comando `#include <stdio.h>` que aparece no início do código inclui o arquivo `stdio.h` ao código. Esse arquivo é um pedaço de código chamado de biblioteca padrão de entrada e saída (standard input/output library) permite o uso do comando `printf`. Usamos bibliotecas quando precisamos usar funções que não são nativas da linguagem C.
- O `main()` indica onde começa a execução do programa. A chave `{` indica o início do `main()` e a chave `}` indica o fim.
- A instrução `printf("hello, world!\n");` imprime o texto *hello, world!* na tela. Toda instrução deve ser concluída com ponto e vírgula `;`. O texto a ser impresso deve estar entre aspas `" "`. O comando `\n` imprime linha nova, isto é, a texto a seguir será impresso na próxima linha.
- Quando o código estiver com erros de sintaxe, a compilação feita pela linha de comando

  ```
  gcc hello_world.c
  ```

imprime os erros indicando a linha onde ele está. Por exemplo, tente compilar o programa

```
#include <stdio.h>

main()
{
  printf("hello, world!\n")
}
```

O compilador retorna o seguinte erro:

```
gcc hello_world.c
hello_world.c: In function 'main':
hello_world.c:5:1: error: expected ';' before '}' token
 }
```

- O nome `a.out` para o executável é o padrão. Podemos nomear o executável compilando com a linha

  ```
  gcc hello_world.c -o exec.out
  ```

  Nesse caso, execute com a linha

  ```
  ./exec.out
  ```

1.3 Inserção de comentários

Os códigos necessitam de explicações, lembretes e comentários que ajudam na sua leitura. No entanto, esses textos devem ser ignorado pelo compilador, pois interessam apenas aos programadores. Para isso, usaremos o comando \\ para comentar a parte final de uma linha e `/*  */` para comentar um trecho inteiro. Observe o exemplo:

```
#include <stdio.h>

main()
{
  printf("h\n");
  /*printf("e\n");
  printf("l\n");
```

```
  printf("l\n");*/
  printf("o\n");
  printf("world\n"); // O hello ficou na vertical e o world na horizontal
}
```

1.4 Variáveis

Um variável é o nome que damos a uma posição da memória. Os tipos de variáveis mais comuns são:

- `double` : um número racional armazenado em um sistema de ponte flutuante com 64 bits, sendo 53 bits para a mantissa, 11 para o expoente e 1 para o sinal. O menor número positivo tipo `double` é da ordem de 10^{-308} e o maior da ordem de 10^{308}.
- `float` : um número racional armazenado em um sistema de ponte flutuante com 32 bits, sendo 24 bits para a mantissa, 7 para o expoente e 1 para o sinal.
- `int` : um número inteiro armazenado com 16 bits. O tipo `int` abrange número entre -32768 e 32767.
- `long int` : um número inteiro armazenado com 32 bits. O tipo `long int` abrange número entre -2147483648 e 2147483647.
- `unsigned int`: um inteiro positivo entre 0 e 65535.
- Existem outras variações tais como `short int`, `unsigned long int`, `short int` e `unsigned short int`.
- `char` : uma variável que armazena um único caractere (1 byte de 8 bits).

As variáveis precisam ser declaradas antes de ser usadas e devem receber um nome diferentes das palavras-chave da linguamgem C. O símbolo `=` é usado para atribuir valores a uma variável.

Exemplo 1.4.1. Escreva um programa que imprime a número $\pi \approx 3.14159265358979323846264$

```
#include <stdio.h>

main()
{
  //Declaração de variáveis
```

```
  double pi1;
  float pi2;

  //atribuição de valores
  pi1=3.14159265358979323846264338327950288419 71;
  pi2=3.14159265358979323846264338327950288419 71;


  printf("pi1=%.20f\n",pi1);
  printf("pi2=%.20f\n",pi2);
}
```

Comentários sobre o programa do exemplo 1.4.1:

- Observe que a variável `pi1` confere com o número π em até 16 dígitos significativos, enquanto `pi2` confere no máximo 8 dígitos. Isso é devido a precisão do sistema de ponto flutuante tipos `double` e `float`.
- A parte do texto no comando `printf` continua entre aspas, mas a variável está sem aspas e depois da vírgula. Para imprimir o valor de uma variável é necessário informar o tipo de variável: `%f` e `%e` para variáveis `double` ou `float`, `%d` para variáveis do tipo `int` e `%c` para `char`. Para indicar o número de casas decimais de uma variável `float` ou `double` usamos `%.12f` (doze casas).
- Usamos aspas simples para atribuir valor para variavél tipo `char`. Por exemplo `var='A'`.
- Podemos atribuir um valor para a variável ao mesmo tempo que declaramos. Por exemplo `char var='A'`.

1.5 Scanf

Assim como podemos imprimir uma variável, também podemos ler. Usamos o comando `scanf` para essa utilidade.

Exemplo 1.5.1. Escreva um programa que lê um número entre 0 e 9 e depois imprima-o.

```
  #include <stdio.h>

main()
```

```
{
  int x;
  printf("digite um número inteiro entre 0 e 9\n");
  scanf("%d",&x);

  printf("x=%d\n",x);
}
```

Observe para usar o comando `scanf("%d",&x);` precisamos, além de dizer o tipo inteiro `%d`, também passar a referência na memória onde está armazenado o valor do inteiro usando `&x` em vez de colocar simplesmente o valor `x`. Faremos uma discussão mais apurada sobre o símbolo `&` durante o curso. O comando `getchar();` é uma alternativa na leitura de um caractere.

1.6 A biblioteca math.h

A biblioteca `math.h` permite usar funções matemáticas básicas, tais como senos, cossenos, exponenciais, logarítmos, etc. Para usá-la, é necessário adicionar a linha `#include <math.h>` no cabeçalho e compilar o programa com a linha `gcc programa.c -lm`.

Exemplo 1.6.1. Implemente um programa para testar as funções seno, cosseno, tangente, etc.

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

main (void)
{
  double x = 8.62;

  printf("Biblioteca math.h \n\n");

  printf("Valor aproximado para baixo de %f é %f\n",x, floor(x) );
  printf("Valor aproximado para cima de %f é %f\n", x, ceil(x));

  printf("Raiz quadrada de %f é %f\n",x,sqrt(x));
  printf("%.2lf ao quadrado é %.2f \n",x,pow(x,2));

  printf("Valor de  seno de %.2f = %.2f \n",x,sin(x));
```

```
  printf("Valor de  cosseno de %.2f = %.2f \n",x,cos(x));
  printf("Valor de  tangente de %.2f = %.2f \n",x,tan(x));

  printf("Logaritmo natural de %.2f = %.2f \n",x,log(x));
  printf("Logaritmo de %.2f  na base 10 = %.2f \n",x,log10(x));
  printf("Exponencial de %.2f = %e \n",x,exp(x));

  printf("O valor aproximado de pi é %e \n",M_PI);
  printf("O valor aproximado de pi/2 é %e \n",M_PI_2);

  printf("O módulo de -3.2 é %f \n",fabs(-3.2));
  printf("O módulo de -3 é %d \n",abs(-3));
}
```

1.7 Operações entre inteiros e reais

Como esperado, usamos os símbolos `+`, `-`, `/` e `*` para somar, subtrair, dividir e multiplicar, respectivamente. No caso de operação entre dois inteiros, ainda existe mais um símbolo, o `%`, que serve para pegar o resto da divisão inteira (aritmética modular). A operação entre dois números inteiros, resulta em um número inteiro e a operação entre dois números reais, resulta em um número real. No entanto, a operação entre um real e um inteiro, resulta em um real.

Exemplo 1.7.1. Escreva um programa que lê dois números inteiros, tome o resto da divisão do primeiro pelo segundo e imprima o resultado.

```
 #include <stdio.h>

main()
{
  int a,b, resultado;
  printf("digite um número inteiro\n");
  scanf("%d",&a);
  printf("digite outro número inteiro\n");
  scanf("%d",&b);
  resultado=a%b;
  printf("O resto da divisão do primeiro número pelo segundo é=%d\n",resultado);
}
```

Exemplo 1.7.2. Escreva um programa que lê dois flot's e imprime o resultado da soma de 1 e 10^{-8}.

```
#include <stdio.h>

main()
{
  float a,b, resultado;
  a=1;
  b=1e-8;
  resultado=a+b;
  printf("a=%f, b=%f, resultado=%f\n",a,b,resultado);
}
```

Observe no exemplo 1.7.2 a soma de $1+10^{-8}$ resultou em 1. Isso se deve a precisão de oito dígitos de um `float`.

A linguagem de programação C é chamada de linguagem imperativa, pois ela é executada em sequência de ações. Para entender isso, vamos estudar o seguinte programa:

```
#include <stdio.h>

main()
{
  double x,y,z;
  x=2;
  y=2;
  z=x+y;
  x=1;
  y=1;
  printf("x=%f, y=%f, z=%f\n",x,y,z);
}
```

Os valores impressos são $x = 1$,$y = 1$ e $z = 4$. Mas uma das linhas faz $z = x + y$, então, por que o valores impressos não são $x = 1$,$y = 1$ e $z = 2$ ou $x = 2$,$y = 2$ e $z = 4$? Para entender, interprete o programa instrução por instrução, sequencialmente. Primeiro, defininos três variáveis tipo `double`, x, y e z. Depois atribuímos o valor 2 para x, 2 para y e atribuímos o valor $x + y$ para z. Nesse ponto do programa, $z = 4$. Seguindo, atribuímos o valor 1 para x, substituindo o antigo valor. Da mesma forma, atribuímos o valor 1 para y. Nesse ponto do programa, $x = 1$, $y = 1$ e $z = 4$, pois não atribuímos novo valor para z.

1.8 Operadores relacionais

São operadores que comparam dois números, tendo como resposta duas possibilidades: VERDADE (1) ou FALSO (0). Para testar se dois números são iguais, usamos o símbolo `==`. Observe o programa abaixo:

```
#include <stdio.h>

main()
{
  double x,y,z;
  x=y=2;
  z=1;
  printf("O resultado de %f == %f é : %d\n",x,y,x==y);
  printf("O resultado de %f == %f é : %d\n",x,z,x==z);
}
```

Aqui, testamos se 2 é igual a 2 e obtemos como resposta 1, isto é, VERDADE. Também testamos se 2 é igual a 1 e a reposta foi 0, isto é, FALSO. Os operadores lógicos são:

Operador	Nome	Significado
`a==b`	Igual	a é igual a b?
`a>b`	maior	a é maior que b?
`a>=b`	maior ou igual	a é maior ou igual a b?
`a<b`	menor	a é menor que b?
`a<=b`	menor ou igual	a é menor ou igual a b?
`a!=b`	diferente	a é diferente a b?

1.9 Exercícios

E 1.9.1. Escreva versões similares ao código do exemplo 1.2.1 usando a instrução `printf` várias vezes e introduzindo comentários. Use os comandos

```
\7    \a    \b    \n    \r    \t    \v    \\    \'    \"    \?    %%
```

e descubra a função de cada um deles. Produza propositalmente alguns erros de sintaxe e observe a resposta do compilador. Por exemplo, execute o código

```
#include <stdio.h>

main(){
  printf("\t matemática \v e \n \t computação\n \t científica\n");
  printf("\n\n\n");
  printf("\v \"linguagem\" \v programação \n programa\n");
}
```

E 1.9.2. Escreva versões similares ao código do exemplo 1.4.1, definindo variáveis de vários tipos e imprimindo-as. Use os formatos de leitura e escrita

```
%d    %i    %o    %x
```

para inteiros e

```
%f    %e
```

para reais.

E 1.9.3. Escreva um programa que lê três caracteres um inteiro e dois double's e depois imprime todos eles.

E 1.9.4. Escreva um programa que some dois double's, $10^8 + 10^{-8}$. Discuta o resultado.

E 1.9.5. Abaixo temos três programas que divide 1 por 2. Execute-os e discuta os resultados.

Programa 1

```
#include <stdio.h>

main()
{
  double a,b, resultado;
  a=1;
  b=2;
  resultado=a/b;
  printf("a=%f, b=%f, resultado=%f\n",a,b,resultado);
}
```

Programa 2

```
#include <stdio.h>
```

```
main()
{
  double resultado=1/2;
  printf("%f\n",resultado);
}
```

Programa 3

```
#include <stdio.h>

main()
{
  double resultado=1.0/2.0;
  printf("%f\n",resultado);
}
```

E 1.9.6. Escreva um programa que calcula a área e o perímetro de um círculo de raio r.

E 1.9.7. Escreva um programa que lê dois números inteiros a e b e imprime o resultado do teste $a > b$.

E 1.9.8. Estude o comportamento assintótico de cada uma das expressões abaixo:

a) $\frac{(1+x)-1}{x}$, $x = 10^{-12}$, $x = 10^{-13}$, $x = 10^{-14}$, $x = 10^{-15}$, $x = 10^{-16}$, ...

b) $\left(1 + \frac{1}{x}\right)^x$, $x = 10^{12}$, $x = 10^{13}$, $x = 10^{14}$, $x = 10^{15}$, $x = 10^{16}$, $x = 10^{17}$,...

Para cada um dos itens acima, escreva um programa para estudar o comportamento numérico. Use variável `double` no primeiro programa, depois use `float` para comparações. Explique o motivo da discrepância entre os resultados esperado e o numérico. Para entender melhor esse fenômeno, leia o capítulo 2 do livro https://www.ufrgs.br/numerico/, especialmente a seção sobre cancelamento catastrófico.

E 1.9.9. Considere as expressões:

$$\frac{\exp(1/\mu)}{1+\exp(1/\mu)}$$

e

$$\frac{1}{\exp(-1/\mu)+1}$$

com $\mu > 0$. Verifique que elas são idênticas como funções reais. Teste no computador cada uma delas para $\mu = 0{,}1$, $\mu = 0{,}01$ e $\mu = 0{,}001$. Qual dessas expressões é mais adequada quando μ é um número pequeno? Por quê?

E 1.9.10. Use uma identidade trigonométrica adequada para mostrar que:

$$\frac{1-\cos(x)}{x^2} = \frac{1}{2}\left(\frac{\operatorname{sen}(x/2)}{x/2}\right)^2.$$

Analise o desempenho destas duas expressões no computador quando x vale 10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}, 10^{-9}, 10^{-200} e 0. Discuta o resultado. **Dica:** Para $|x| < 10^{-5}$, $f(x)$ pode ser aproximada por $1/2 - x^2/24$ com erro de truncamento inferior a 10^{-22}.

Capítulo 2

Testes, condições e laços

2.1 if-else

O `if-else` permite escolher em quais circunstância se deve executar uma instrução. Sua sintaxe é:

```
if (condição)
instrução 1
else
instrução 2
```

Exemplo 2.1.1. Escreva um programa que lê dois números e imprime qual é o maior

```
#include <stdio.h>

main()
{
  float x,y;
  printf("Entre com o valor de x\n");
  scanf("%f", &x);
  printf("Entre com o valor de y\n");
  scanf("%f", &y);
  if (x>=y)
    printf("x é maior ou igual a y\n");
  else
    printf("x menor que y\n");
}
```

Exemplo 2.1.2. Escreva um programa que lê dois números inteiros e apresente-os em ordem decrescente.

```
 #include <stdio.h>

main()
{
  int x,y,aux;
  printf("Entre com o valor de x\n");
  scanf("%d", &x);
  printf("Entre com o valor de y\n");
  scanf("%d", &y);
  if (x<y)
    {
      aux=x;
      x=y;
      y=aux;
    }

  printf("%d  %d\n",x,y);
}
```

Observe que é importante a identação do bloco de instrução internos ao if para maior legibilidade do código.

2.2 Operadores Lógicos

Interliga duas condições e funcionam da mesma forma que os operadores lógicos aritméticos. Por exemplo, no caso de $a < x < b$, fazemos `x>a && x<b`, onde `&&` é o operador lógico **e**. O operador lógico **ou** é o `||` e o operador lógico **não** é o `!`.

Exemplo 2.2.1. Escreva um programa que leia um número real e responda se ele está entre 0 e 1 ou entre 3 e 5.

```
#include <stdio.h>

main()
{
  float x;
  printf("Entre com o valor de x\n");
  scanf("%f", &x);
  if ((x>=0 && x<=1)||(x>=3 && x<=5))
    printf("o número está ou entre 0 e 1 ou entre 3 e 5\n");
  else
```

```
    printf("o número não está entre 0 e 1 e não está entre 3 e 5\n");
}
```

2.3 switch

O `switch` é uma alternativa ao `if-else` quando o número de possibilidades é grande. Sua sintaxe é

```
switch (expressão)
case valor 1: instrução 1;
case valor 2: instrução 2;
case valor 3: instrução 3;
```

Exemplo 2.3.1. Escreva um programa que lê um caractere e imprime solteiro para S ou s, casado para C ou c, etc.

```
#include <stdio.h>

main()
{
  char estado_civil;
  printf("Entre com seu estado civil\n");
  scanf("%c", &estado_civil);
  switch (estado_civil)
    {
    case 's':
    case 'S': printf("solteiro(a)\n"); break;
    case 'C':
    case 'c': printf("casado(a)\n"); break;
    case 'V':
    case 'v': printf("viúvo(a)\n");break;
    case 'D':
    case 'd': printf("divorciado\n");break;
    default: printf("não informou/outro\n"); break;
    }
}
```

2.4 While

O `while` permite fazer um loop enquanto uma expressão for verdadeira (ou diferente de zero). Sua sintaxe é:

```
while (expressão)
instrução
```

Essa estrutura só faz sentido se a expressão é recalculada a cada iteração e para quando chegar no valor zero. Vamos estudar o exemplo abaixo.

Exemplo 2.4.1. Escreva um código que imprime os números entre 0 e 10.

```
#include <stdio.h>

main()
{
  int i=0;
  while (i<=10)
  {
    printf("i=%d\n",i);
    i=i+1;
  }
}
```

Vamos discutir o código do exemplo 2.4.1. A primeira instrução é uma atribuição de valor 0 para o inteiro `i`. O `while` testa a expressão `(i<=10)`: 0 é menor ou igual a 10? Como o teste tem resposta 1 (VERDADE), o programa executa as instruções dentro do bloco `while`. A primeira instrução imprime o número 0 e a segunda atribui para `i` o valor de `i+1`, ou seja, agora `i` tem valor $0 + 1 = 1$. O `while` repete a pergunta: 1 é menor ou igual a 10? A resposto é verdadeira, então o programa imprime o número 1 e incrementa `i` para 2. Quando o valor de `i` chegar em 11, o `while` faz a pergunta: 11 é menor ou igual a 10? A resposta é 0 (FALSO) e o programa encerra o loop.

No contexto de loops, um operador importante para incrementar o valor de uma variável é o operador incremento `++`. Observe uma possibilidade de programa para o exemplo 2.4.1:

```
#include <stdio.h>

main()
{
  int i=0;
  while (i<=10)
    {
    printf("i=%d\n",i);
    i++;
```

```
  }
}
```

O operador `++i` pode ser usado antes da variável (pré-fixado) ou depois `i++` (pós-fixado), sedo que em ambos os casos a variável é incrementada em uma unidade. No entanto, quando o valor da variável é usado numa expressão, o efeito é diferente: `i++` primeiro usa o valor antigo depois incrementa e; `++i` primeiro incrementa e depois usa o valor atualizado. Observe os dois programas abaixo:

Programa 1:

```
#include <stdio.h>

main()
{
  int x,i=0;
  while (i<=10)
  {
    printf("i=%d\n",++i);
  }
}
```

Programa 2:

```
#include <stdio.h>

main()
{
  int x,i=0;
  while (i<=10)
  {
    printf("i=%d\n",i++);
  }
}
```

O programa 1 imprime os número entre 1 e 11 e o programa 2 imprime os números entre 0 e 10. O operador análogo que diminui em uma unidade o valor da variável é o operador decremento `--`.

Observe que o `while` primeiro testa, depois executa. Para inverter essa sequência, usamos o `do-while`. Sua sintaxe é:

```
do
instrução
while (expressão)
```

Uma versão para o programa do exemplo 2.4.1 é:

```
 #include <stdio.h>

main()
{
  int i=0;
  do
  {
    printf("i=%d\n",i++);
  }
  while (i<=10);
}
```

Exemplo 2.4.2. Escreva um programa que some os primeiros termos da série

$$\sum_{i=0}^{\infty} \frac{1}{2^i}.$$

```
#include <stdio.h>

main()
{
  double enesimo=1,soma=0,epsilon=1e-10;
  int controle=3;
  soma=enesimo;
  while (controle)
  {
    enesimo*=1.0/2.0;
    soma+=enesimo;
    if (enesimo<epsilon) controle--;
    printf("controle=%d, enesimo=%e, soma=%f\n",controle,enesimo,soma);
  }
}
```

No exemplo 2.4.2 `enesimo*=1.0/2.0` tem a mesma funcionalidade de `enesimo=enesimo*1.0/2.0` assim como `soma+=enesimo` funciona como `soma=soma+enesimo`.

2.5 For

A `for` também é usado para construir loops. Sua sintaxe é

```
for (expressão 1;expressão 2; expressão 3)
instrução
```

onde `expressão 1` configura o início do loop, `expressão 2` o fim e `expressão 3` o incremento. Vamos estudar um exemplo.

Exemplo 2.5.1. Reescreva o código do exemplo 2.4.1 usando `for`.

```
#include <stdio.h>

main()
{
  int i;
  for (i=0;i<=10;i++)
  {
    printf("i=%d\n",i);
  }
}
```

Observe no código do exemplo 2.4.2 o uso do `for (i=0;i<=10;i++)`. Aqui, `i` recebe 0 no início do loop com a expressão `i=0`, o laço de repetição segue enquanto a expressão `i<=10` for verdadeira e é incrementada em uma unidade a cada iteração pela expressão `i++`.

Exemplo 2.5.2. Some os 20 primeiros termos da série

$$\sum_{i=1}^{\infty} \frac{1}{i^2}$$

```
#include <stdio.h>

main()
{
  double enesimo,soma=0;
  int i;
  for (i=1;i<=100;i++)
  {
    enesimo=1.0/(i*i);
    soma+=enesimo;
    printf("i=%d, enesimo=%e, soma=%f\n",i,enesimo,soma);
  }
}
```

2.6 Exercícios

E 2.6.1. Implemente um programa que indique se o inteiro lido é zero ou não.

E 2.6.2. Implemente um programa que retorna quantos segundos tem x horas.

E 2.6.3. Implemente um programa que retorna quantos segundos tem x horas. Cuide que enviar uma mensagem de erro se x for negativo.

E 2.6.4. Escreva um programa para testar se um ano é bissexto ou não.

E 2.6.5. Escreva um programa que verifique quantos dias tem um dado mês (exemplo, janeiro tem 31 dias).

E 2.6.6. Escreva um programa que leia uma data e verifique se ela é válida ou não.

E 2.6.7. Escreva um programa que imprime os números entre 0 e 20 em ordem decrescente. Faça versões que incluam `for`, `while` e `--`.

E 2.6.8. Reescreva o código do exemplo 2.5.2 e calcule a soma até que n-ésimo termo ficar menor que 10^{-10}.

E 2.6.9. Estude o comportamento dessas duas sequências:

$$\begin{cases} x_0 = \dfrac{1}{3} \\ \\ x_n = \dfrac{x_{n-1}+1}{4}, \qquad n = 1,2,... \end{cases}$$

e

$$\begin{cases} x_0 = \dfrac{1}{3} \\ \\ x_n = 4x_{n-1} - 1, \qquad n = 1,2,... \end{cases}$$

Verfique que ambas as sequências são constantes: $\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, ...$ Depois implemente um código e estude o comportamento numérico dessas sequências. Para entender melhor o fenômeno numérico, leia o capítulo 2 do livro https://www.ufrgs.br/numerico/, especialmente a seção intitulada “mais exemplos de cancelamento catastrófico”.

E 2.6.10. A sequência

$$\begin{cases} x_0 = a, \qquad a > 0 \\ \\ x_n = \dfrac{1}{2}\left(x_{n-1} + \dfrac{b}{x_{n-1}}\right), \qquad b > 0 \text{ e } n = 1,2,... \end{cases}$$

converge para $\sqrt{b}$. Implemente um código para calcular $\sqrt{2}$ e use como critério de parada a expressão:

$$\frac{|x_n - x_{n-1}|}{|x_n|} < 10^{-10}.$$

E 2.6.11. Conclua que a função $f(x) = x - \cos(x)$ tem uma única raíz real. A sequência

$$\begin{cases} x_0 = a, \\ \\ x_n = \cos(x_{n-1}), \qquad n = 1,2,... \end{cases}$$

converge para a raíz da função f. Implemente um código para calcular a raíz da função f com 8 dígitos significativos corretos. Use diferentes valores de a.

Capítulo 3

Funções

3.1 Funções

Começamos com o seguinte exemplo:

Exemplo 3.1.1. Implemente um programa para estudar a expressão

$$\left(1+\frac{1}{x}\right)^x$$

para $x = 10^2$, $x = 10^6$ e $x = 10^{10}$.

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

main (void)
  {
  double x = 1e2,y;
  y=1+1/x;
  y=pow(y,x);
  printf("(1+1/%e)^%e=%f\n",x,x,y);

  x=1e6;
  y=1+1/x;
  y=pow(y,x);
  printf("(1+1/%e)^%e=%f\n",x,x,y);

  x=1e10;
  y=1+1/x;
  y=pow(y,x);
```

```
  printf("(1+1/%e)^%e=%f\n",x,x,y);
}
```

No solução do exemplo 3.1.1, as linhas 7, 12 e 17 são as mesmas, assim como 8, 13 e 18 e 9, 14 e 19. Para deixar o código mais sucinto e legível, convém escrever essas linhas uma única vez e chamar sua execução cada vez que desejarmos. Observe uma versão alternativa para o código do exemplo 3.1.1:

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

Euler_seq(double x)
{
  double y;
  y=1+1/x;
  y=pow(y,x);
  printf("(1+1/%e)^%e=%f\n",x,x,y);
}

main (void)
{
  Euler_seq(1e2);
  Euler_seq(1e6);
  Euler_seq(1e10);
}
```

Nessa última versão, introduzimos a função `Euler_seq(double x)` antes do ambiente `main` que entra um parâmetro `x` do tipo `double` e executa as linhas de antes estavam repetidas. Dentro do ambiente `main` nós chamamos a função pelo nome e dizemos qual é o `double` que vamos passar.

As funções também podem retornar um valor. Observe uma outra versão para o exemplo 3.1.1:

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

double Euler_seq(double x)
{
  double y;
  y=1+1/x;
  return y=pow(y,x);
}
```

```
main (void)
{
  double z=1e2;
  printf("(1+1/%e)^%e=%f\n",z,z,Euler_seq(z));
  z=1e6;
  printf("(1+1/%e)^%e=%f\n",z,z,Euler_seq(z));
  z=1e10;
  printf("(1+1/%e)^%e=%f\n",z,z,Euler_seq(z));
}
```

Seguem algumas observações sobre funções:

- O nome da variável enviada para uma função não precisa ser o mesmo nome do parâmetro de entrada no cabeçalho. Observe a última versão do código do exemplo 3.1.1, onde a variável enviada é `z` e a variável de entrada da funçao é `x`. Mas observe que elas são do mesmo tipo.
- O cabeçalho não pode ser seguido de ponto-e-vírgula (;).
- Não se pode definir função dentro de outra função. No entanto, pode-se invocar uma função dentro de outra função.
- Depois da instrução `return` pode se colocar qualquer expressão válida em C, inclusive deixar sem nada (tipo `void`), desde que fique coerente com o cabeçalho.
- Sempre que não fique indicado o tipo de retorno, o C assume o padrão `int`.
- As variáveis definidas dentro das funções são chamadas de variáveis locais e não podem ser chamadas de fora. Na última versão do código do exemplo 3.1.1, `y` é uma variável local do tipo `double`.

Exemplo 3.1.2. Escreva um programa que lê dois números inteiros positivos e imprime o menor entre eles. Use a estrutura de função para retornar o menor.

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

int menor(int a,int b)
{
  if (a<=b) return a;
  else return b;
```

```
}

main (void)
{
  int x,y;
  printf("Entre com dois números inteiros positivos\n");
  scanf(" %d  %d",&x,&y);
  printf("O menor número entre eles é o %d\n",menor(x,y));
}
```

3.2 Problemas

Exemplo 3.2.1. Implemente um programa que aproxime o valor da integral de $f(x)$ no intervalo $[a,b]$ usando o método de Simpson composto, dado por

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{h}{3}\left[f(x_1)+2\sum_{i=1}^{n-1} f(x_{2i+1})+4\sum_{i=1}^{n} f(x_{2i})+f(x_{2n+1})\right]\\ &= \frac{h}{3}\left[f(x_1)+4f(x_2)+2f(x_3)+4f(x_4)+2f(x_5)+4f(x_6)+\cdots+2f(x_{2n-1})+4f(x_{2n})+\right.\end{aligned}$$

onde

$$\begin{aligned} h &= \frac{b-a}{2n}\\ x_i &= a+(i-1)h, \qquad i=1,2,\cdots,2n+1. \end{aligned}$$

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

double f(double x)
{
  return exp(-x);
}

double Simpson(double a,double b, int n)
{
  double x,h,soma;
  int i;
  x=a;
  h=(b-a)/(2.0*n);
  soma=f(x);
```

```
  for (i=1;i<=n;i++)
  {
    x+=h;
    soma+=4*f(x);
    x+=h;
    soma+=2*f(x);
  }
  soma-=f(x);
  soma*=h/3;
  return soma;
}

main (void)
{
  printf("Integral=%f\n",Simpson(0,1,20));
}
```

Nós ainda não estudamos estruturas de vetores para armazenar a malha $x_1 = a$, $x_2 = a + h$, $x_3 = a + 2h$, $\cdots$. Por isso, no código acima, calculamos dentro do `for` usando a relação de recorrência $x_i = x_{i-1} + h$. Observe que calculamos $f(x_1) = f(a)$ antes do `for`. No `for`, nós corremos $i = 1,...,n$ e a cada iteração calculamos $4f(x_i)$ e $2f(x_i + h)$. Isso significa que, na última iteração, calculamos $4f(x_{2n})$ e $2f(x_{2n+1}) = 2f(b)$. Mas $f(b)$ deveria ser somado uma vez só, por isso, precisamos diminuir $f(b)$ depois de encerrado o `for`.

No código da solução do exemplo 3.2.1, nós calculamos uma aproximação para $\int_a^b f(x)dx$, mas não se preocupamos com a precisão do cálculo. Vamos introduzir o seguinte critério de parada: se o valor absoluto da diferença entre as aproximações com n e $n + 1$ pontos na malha for menor que 10^{-8}, então apresenta o resultado dado pela malha com $n + 1$ pontos. Façamos isso introduzindo uma nova função, como segue abaixo:

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

double f(double x)
{
  return exp(-x);
}

double Simpson(double a,double b, int n)
{
```

```
  double x,h,soma;
  int i;
  x=a;
  h=(b-a)/(2.0*n);
  soma=f(x);
  for (i=1;i<=n;i++)
  {
    x+=h;
    soma+=4*f(x);
    x+=h;
    soma+=2*f(x);
  }
  soma-=f(x);
  soma*=h/3;
  return soma;
}

double Integral(double a, double b,double tolerancia)
{
  int n=1;
  double I_antigo=0,I_atual=2*tolerancia;
  while ((fabs(I_antigo-I_atual)>=tolerancia))
  {
    I_antigo=Simpson(a,b,n);
    I_atual=Simpson(a,b,n+1);
    n++;
  }
  return I_atual;
}

main (void)
{
printf("Integral=%.12f\n",Integral(0,1,1e-8));
}
```

Nessa versão do código, os valores `I_antigo=0` e `I_atual=2*tolerancia` são para que o `while` execute a primeira vez. Observe que, se `I_antigo=I_atual=0`, o programa não entra no `while`. Uma alternativa para implementar o critério de parada é inserir uma variável de controle que, quando a tolerância é atingida, o programa executa mais algumas iterações. Isso dá uma segurança maior para o resultado obtido, evitando que, por exemplo, `I_atual` e `I_antigo` fiquem p?oximos, mas

longe do valor da integral. Veja a próxima versão do código:

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

double f(double x)
{
  return exp(-x);
}

double Simpson(double a,double b, int n)
{
  double x,h,soma;
  int i;
  x=a;
  h=(b-a)/(2.0*n);
  soma=f(x);
  for (i=1;i<=n;i++)
  {
    x+=h;
    soma+=4*f(x);
    x+=h;
    soma+=2*f(x);
   }
  soma-=f(x);
  soma*=h/3;
  return soma;
}

double Integral(double a, double b,double tolerancia)
{
  int n=1,controle=2;
  double I_antigo,I_atual;
  while (controle)
  {
    I_antigo=Simpson(a,b,n);
    I_atual=Simpson(a,b,n+1);
    if((fabs(I_antigo-I_atual)<tolerancia)) controle--;
    else controle=2;
    n++;
  }
```

```
  return I_atual;
}

main (void)
{
  printf("Integral=%.12f\n",Integral(0,1,1e-8));
}
```

É importante validar a rotina antes de considerar o problema resolvido. Para isso, aproxime a integral de várias funções que conhecemos o valor exato da integral e compare.

3.3 Protótipo

Observe que, em todos os programas que implementamos até agora, a função `main` é a última. Nós podemos deixar a função `main` em cima, desde que definimos protótipos das funções. Observe um código anternativo para o exemplo 6.2.2.

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

double f(double x);
double df(double x);
void Newton(double x0, double tol, double *xn, int *k);

void main(void)
{
  double x0=2,xn,tol=1e-10;
  int k;
  Newton(x0,tol,&xn,&k);
  printf("Sol=%f, iterações=%d\n",xn,k);
}
double f(double x)
{
  return cos(x)-x;
}
double df(double x)
{
  return -sin(x)-1.;
}
void Newton(double x0, double tol, double *xn, int *k)
```

```
{
  double dif;
  *k=0;
  do
  {
  (*k)++;
  *xn=x0-f(x0)/df(x0);
  dif=fabs(*xn-x0);
  x0=*xn;
  }while (dif>tol);
}
```

3.4 Exercícios

E 3.4.1. Escreva um código com as seguintes funções:

- Calcula o máximo entre dois inteiros
- Testa se dois inteiros são iguais e retorna 0 ou 1.
- Entra dois `double` e testa se um é o quadrado do outro.
- Entra um `double` r e calcula o comprimento da circunferência de raio r.

E 3.4.2. Escreva uma versão para os códigos dos exercícios 2.6.4, 2.6.5 e 2.6.6 usando a estrutura de função.

E 3.4.3. Escreva uma versão para os códigos dos exercícios 2.6, 2.6.11 usando a estrutura de função.

E 3.4.4. O método de Newton para calcular as raízes de uma função suave $f(x) = 0$ é dado pela sequência

$$\begin{cases} x_0 = a, \\ \\ x_n = x_{n-1} - \dfrac{f(x_{n-1})}{f'(x_{n-1})}, \qquad n = 1,2,... \end{cases},$$

desde que a seja escolhido suficientemente próximo da raíz. Implemente uma código da seguinte forma:

- Com uma função para definir $f(x) = e^{-x^2} - x$.
- Com uma função que entre o chute inicial a e a tolerância e retorne a raíz. Coloque uma mensagens de erro no caso de não convergência.

Teste seu programa com várias funções, tais como $f(x) = \cos(10x) - e^{-x}$, $f(x) = x^4 - 4x^2 + 4$.

E 3.4.5. Repita o exercício 3.4.4 usando o método das Secantes para calcular as raízes da função suave $f(x) = 0$. A sequência é

$$\begin{cases} x_0 = a, \\ \\ x_2 = b, \qquad b \neq a \\ \\ x_{n+1} = x_n - f(x_n)\dfrac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}, \qquad n = 1,2,... \end{cases} .$$

Observe que aqui você precisará de dois chutes iniciais.

E 3.4.6. Implemente um programa que aproxime o valor da integral de $f(x)$ no intervalo $[a,b]$. Faça da seguinte forma:

- implemente uma função para definir $f(x)$;
- implemente o método de Simpson simples dado por

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

E 3.4.7. Implemente um código similar ao dado no exemplo 3.2.1 para aproximar $\int_a^b f(x)dx$ usando a regra dos trapézios

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(x_1) + f(x_{n+1})\right] + h\sum_{i=2}^{n} f(x_i),$$

onde

$$\begin{aligned} h &= \frac{b-a}{n} \\ x_i &= a + (i-1)h, \qquad i = 1,2,\cdots,n+1. \end{aligned}$$

E 3.4.8. Implemente um código similar ao dado no exemplo 3.2.1 para aproximar $\int_a^b f(x)dx$ usando a regra de Boole

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{2h}{45}\left(7f(x_1)+32f(x_2)+12f(x_3)+32f(x_4)+14f(x_5)+32f(x_6)+12f(x_7)+32\right.\\ &+ \left.14f(x_{4n-3})+32f(x_{4n-2})+12f(x_{4n-1})+32f(x_{4n})+7f(x_{4n+1})\right)\end{aligned}$$

onde

$$\begin{aligned}h &= \frac{b-a}{4n}\\ x_i &= a+(i-1)h, \qquad i=1,2,\cdots,4n+1.\end{aligned}$$

E 3.4.9. A quadrature de Gauss-Legendre com dois pontos aproxima a integral da seguinte forma

$$\int_{-1}^{1} f(x)\,dx = f\left(-\frac{\sqrt{3}}{3}\right)+f\left(\frac{\sqrt{3}}{3}\right).$$

Fazendo a mudança de variável $u=\frac{2x-a-b}{b-a}$, temos:

$$\begin{aligned}\int_a^b f(x)\,dx &= \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}u+\frac{a+b}{2}\right)du\\ &\approx \frac{b-a}{2}f\left(-\frac{b-a}{2}\frac{\sqrt{3}}{3}+\frac{a+b}{2}\right)+\frac{b-a}{2}f\left(\frac{b-a}{2}\frac{\sqrt{3}}{3}+\frac{a+b}{2}\right).\end{aligned}$$

Implemente um código que calcule uma aproximação da integral de $f(x)$ no intervalo $[a,b]$ usando a quadratura de Gauss-Legendre com dois pontos.

E 3.4.10. Implemente a seguinte regra para aproximar a integral de $f(x)$ no intervalo $[a,b]$

- Calcule a integral Gauss-Legendre com dois pontos introduzida no exercício 3.4.9.
- Divida o intervalo $[a,b]$ em duas partes e use a quadratura do item anterior em cada parte.
- Compare as integrais dos dois itens anteriores.
- Divida o intervalo $[a,b]$ em quatro partes e use a mesma quadratura em cada parte.
- Compare as integrais com duas e quatro divisões.

- Repita até a convergência com tolerância de 10^{-8}.

E 3.4.11. Implemente o método da bisseção para calcular a raíz de uma função.

O método da bisseção explora o fato de que uma função contínua $f : [a, b] \to \mathbb{R}$ com $f(a) \cdot f(b) < 0$ tem um zero no intervalo (a, b) (veja o teorema de Bolzano). Assim, a ideia para aproximar o zero de uma tal função $f(x)$ é tomar, como primeira aproximação, o ponto médio do intervalo $[a, b]$, isto é:

$$x^{(0)} = \frac{(a+b)}{2}.$$

Pode ocorrer de $f(x^{(0)}) = 0$ e, neste caso, o zero de $f(x)$ é $x^* = x^{(0)}$. Caso contrário, se $f(a) \cdot f(x^{(0)}) < 0$, então $x^* \in (a, x^{(0)})$. Neste caso, tomamos como segunda aproximação do zero de $f(x)$ o ponto médio do intervalo $[a, x^{(0)}]$, isto é, $x^{(1)} = (a + x^{(0)})/2$. Noutro caso, temos $f(x^{(0)}) \cdot f(b) < 0$ e, então, tomamos $x^{(1)} = (x^{(0)} + b)/2$. Repetimos este procedimento até alcançar a tolerância

$$\frac{|b^{(n)} - a^{(n)}|}{2} < TOL.$$

Teste usando algumas funções, tais como $f(x) = \cos(x) - x$ no intervalo [0,2] ou $f(x) = e^{-x} - x$ no intervalo [0,1].

Capítulo 4

Vetores e matrizes

4.1 Vetores

Vetor (ou array) é uma lista de elementos do mesmo tipo que podem ser acessados individualmente com o mesmo nome de variável. Definimos um vetor dizendo o tipo, o nome da variável e o número de elementos: `tipo nome[nº elementos];`.

Exemplo 4.1.1. Alguns testes de inicialização de vetores.

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

main (void)
{
  int i;
  double x[3]={1.5,2,3};
  int y[]={1,2,5};
  int z[8]={1,2,5};

  for (i=0;i<3;i++)     printf("x[%d] = %f\n",i,x[i]);
  printf("\n");
  for (i=0;i<3;i++)     printf("y[%d] = %d\n",i,y[i]);
  printf("\n");
  for (i=0;i<8;i++)     printf("z[%d] = %d\n",i,z[i]);
}
```

Exemplo 4.1.2. Implemente um programa que armazena em um vetor os 10 primeiros números da sequência de Fibonacci e imprime a soma do sétimo com o décimo.

```
 #include <stdio.h>

main (void)
{
  int i,x[10];
  x[0]=1;
  x[1]=1;
  for (i=0;i<8;i++)
  {
    x[i+2]=x[i+1]+x[i];
  }
  printf("soma do sétimo com o décimo = %d\n",x[6]+x[9]);
}
```

No exemplo 4.1.2, definimos um vetor com 10 posições fazendo `int x[10];`. Depois, acessamos a posição `i` do vetor fazendo `x[i]`. Observe que as posições do vetor começamos em `0` e, portanto, a sétima posição é acessada por `x[6]` e a décima por `x[9]`.

Exemplo 4.1.3. O método iterativo de Jacobi para calcular a solução do sistema linear

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= y_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= y_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= y_n \end{aligned} \tag{4.1}$$

é dado pela recursão

$$\begin{aligned} x_1^{(k+1)} &= \frac{y_1 - \left(a_{12}x_2^{(k)} + \cdots + a_{1n}x_n^{(k)}\right)}{a_{11}} \\ x_2^{(k+1)} &= \frac{y_2 - \left(a_{21}x_1^{(k)} + a_{23}x_3^{(k)} + \cdots + a_{2n}x_n^{(k)}\right)}{a_{22}} \\ &\vdots \\ x_n^{(k+1)} &= \frac{y_2 - \left(a_{n1}x_1^{(k)} + \cdots + a_{n,n-2}x_{n-2}^{(k)} + a_{n,n-1}x_{n-1}^{(k)}\right)}{a_{nn}} \end{aligned}$$

onde x_1^0, x_2^0, $\cdots$, x_n^0 é um chute inicial. A iteração converge para a solução quando a matriz do sistema é estritamente diagonal dominante.

Implemente um código para resolver o sistema

$$\begin{aligned} 3x + y + z &= 1 \\ -x - 4y + 2z &= 2 \\ -2x + 2y + 5z &= 3 \end{aligned}$$

com o método de Jacobi.

Primeiro, montamos a iteração:

$$\begin{aligned} x^{k+1} &= \frac{1 - y^k - z^k}{3} \\ y^{k+1} &= \frac{-2 - x^k + 2z^k}{4} \\ z^{k+1} &= \frac{3 + 2x^k - 2y^k}{5} \end{aligned}$$

Vamos usar o chute inicial $x^0 = 0$, $y^0 = 0$ e $z^0 = 0$.

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

main (void)
{
  double norma2,tolerancia=1e-3,x_antigo[3],x_atual[3];
  int i,controle=3;
  //chute inicial
  for (i=0;i<3;i++)x_antigo[i]=0;

  //iteração
  while (controle)
  {
    x_atual[0]=(1-x_antigo[1]-x_antigo[2])/3;
    x_atual[1]=(-2-x_antigo[0]+2*x_antigo[2])/4;
    x_atual[2]=(3+2*x_antigo[0]-2*x_antigo[1])/5;
    norma2=0;
    for (i=0;i<3;i++) norma2+=(x_atual[i]-x_antigo[i])*(x_atual[i]-x_antigo[i]);
    if (norma2<tolerancia) controle--;
    else controle=3;
    for (i=0;i<3;i++) x_antigo[i]=x_atual[i];
    printf("x=%f, y=%f, z=%f\n",x_atual[0],x_atual[1],x_atual[2]);
  }
}
```

Nessa versão de código usamos o critério de parada norma em l_2 da diferença menor que uma tolerância, isto é,

$$||(x^{k+1}-x^k,y^{k+1}-y^k,z^{k+1}-z^k)||_2 = \sqrt{(x^{k+1}-x^k)^2+(y^{k+1}-y^k)^2+(z^{k+1}-z^k)^2} < TOL.$$

Também usamos o dispositivo de controle, isto é, depois que atingimos a tolerança caminhamos mais três passos.

Agora vamos fazer um vesão um pouco mais elaborada. Vamos criar uma função que calcula norma l_2 de vetores e outra que faz uma iteração. Para isso, temos que passar um vetor como parâmetro para a função. Observe o código:

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

double norma_dif_2(double x[3],double y[3])
{
  int i;
  double norma2=0;
  for (i=0;i<3;i++) norma2+=(x[i]-y[i])*(x[i]-y[i]);
  return sqrt(norma2);
}
void Iteracao(double x_antigo[3],double x_atual[3])
{
    x_atual[0]=(1-x_antigo[1]-x_antigo[2])/3.;
    x_atual[1]=(-2-x_antigo[0]+2*x_antigo[2])/4.;
    x_atual[2]=(3+2*x_antigo[0]-2*x_antigo[1])/5.;
}
void Jacobi(double tolerancia,double x_atual[3])
{
  double x_antigo[3];
  int i,controle=3;
  //chute inicial
  for (i=0;i<3;i++)x_antigo[i]=0;

  //iteração
  while (controle)
  {
    Iteracao(x_antigo,x_atual);
    if (norma_dif_2(x_antigo,x_atual)<tolerancia) controle--;
    else controle=3;
    for (i=0;i<3;i++) x_antigo[i]=x_atual[i];
    //printf("x=%f, y=%f, z=%f\n",x_atual[0],x_atual[1],x_atual[2]);
```

```
  }
  return;
}

main (void)
{
  int i;
  double tolerancia=1e-3,solucao[3];
  Jacobi(tolerancia,solucao);
  for (i=0;i<3;i++)     printf("x[%d] = %f\n",i,solucao[i]);
}
```

Esse último código passa vetores como parâmetros para funções. A função `norma_dif_2` passa dois vetores e retorna `double`. A `Iteracao` passa dois vetores e retorna `void`, onde usamos um vetor para levar a informação para dentro da função e o outro vetor para retirar o resultado: `x_antigo` leva a informação e `x_atual` entra com qualquer coisa e é calculado dentro da função. Já a função `Jacobi` entra o valor da tolerância e passa um vetor para colocar a solução.

Os códigos que implementamos resolve apenas um único sistema 3x3. É interessante pensar em rotinas mais gerais, que resolvem um sistema $Ax = b$ de dimensão nxn.

4.2 Matrizes

Matrizes são vetores de vetores, isto é, cada posição de um vetor pode ser um vetor. Para definir uma matriz fazemos `tipo nome[nº linhas][nº colunas]`, como no exemplo:

Exemplo 4.2.1. Alguns testes de inicialização de matrizes.

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

main (void)
{
  int i,j;
  double x[2][2];// não inicializado
  int y[3][3]={{1,2,5},{1,0,1},{0,1,0}};
  x[0][0]=1.1;
  x[0][1]=2;
  x[1][0]=3;
```

```
  x[1][1]=4;
  for (i=0;i<2;i++)
  {
    for (j=0;j<2;j++)
    {
    printf("x[%d][%d] = %f    ",i,j,x[i][j]);
    }
  printf("\n");
  }

  printf("\ny[%d][%d] = %d\n",1,2,y[1][2]);
}
```

Exemplo 4.2.2. Vamos refazer o exemplo 4.1.3 usando estrutura de matrizes

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

main (void)
{
  double vetor[3]={1,2,3},matriz[3][3]={{3,1,1},{-1,-4,2},{-2,2,5}};
  double norma2,tolerancia=1e-3,x_antigo[3],x_atual[3];
  int i,controle=3;
  //chute inicial
  for (i=0;i<3;i++)x_antigo[i]=0;

  //iteração
  while (controle)
  {
    x_atual[0]=(vetor[0]-matriz[0][1]*x_antigo[1]-matriz[0][2]*x_antigo[2])/matriz[0]
    x_atual[1]=(vetor[1]-matriz[1][0]*x_antigo[0]-matriz[1][2]*x_antigo[2])/matriz[1]
    x_atual[2]=(vetor[2]-matriz[2][0]*x_antigo[0]-matriz[2][1]*x_antigo[1])/matriz[2]
    norma2=0;
    for (i=0;i<3;i++) norma2+=(x_atual[i]-x_antigo[i])*(x_atual[i]-x_antigo[i]);
    if (norma2<tolerancia) controle--;
    else controle=3;
    for (i=0;i<3;i++) x_antigo[i]=x_atual[i];
    printf("x=%f, y=%f, z=%f\n",x_atual[0],x_atual[1],x_atual[2]);
  }
}
```

Agora, vamos fazer o mesmo código outra vez generalizando as rotinas

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

#define N 3 /* dimensão do sistema*/

double norma_2(double x[N])
{
  int i;
  for (i=0;i<N;i++)
  {
    norma2+=x[i]*x[i];
  }
  return sqrt(norma2);
}
void Iteracao(double x_antigo[N],double x_atual[N],double matriz[N][N], double
{
  double aux;
  int i,j;
  for (i=0;i<N;i++)
  {
    aux=0;
    for (j=0;j<i;j++)
    aux+=matriz[i][j]*x_antigo[j];
    for (j=i+1;j<N;j++)
    aux+=matriz[i][j]*x_antigo[j];
  x_atual[i]=(vetor[i]-aux)/matriz[i][i];
  }
}
void Jacobi(double tolerancia,double x_atual[N])
{
  double x_antigo[N],dif[N];
  double matriz[N][N]={{3,1,1},{-1,-4,2},{-2,2,5}};
  double vetor[N]={1,2,3};
  int i,controle=3;
  //chute inicial
  for (i=0;i<N;i++)x_antigo[i]=0;

  //iteração
  while (controle)
  {
```

```
    Iteracao(x_antigo,x_atual,matriz,vetor);
    for (i=0;i<N;i++) dif[i]=x_atual[i]-x_antigo[i];
    if (norma_2(dif)<tolerancia) controle--;
    else controle=3;
    for (i=0;i<N;i++) x_antigo[i]=x_atual[i];
  }
  return;
}

main (void)
{
  int i;
  double tolerancia=1e-3,solucao[N];
  Jacobi(tolerancia,solucao);
  for (i=0;i<N;i++)     printf("x[%d] = %f\n",i,solucao[i]);
}
```

No último código usamos uma diretiva `#define N 3` que diz para o compilador trocar todos os N do código por 3. Observe que a constante N é bastante usada no código e a diretiva `# define` desobriga de passá-la como parâmetro em todas as funções que ela aparece.

4.3 Problemas

Exemplo 4.3.1. Resolva numericamente o problema de valor de contorno

$$\begin{aligned} -u_{xx} &= 100(x-1)^2, \quad 0 < x < 1, \\ u(0) &= 0, \\ u(1) &= 0. \end{aligned}$$

usando a fórmula de diferenças finitas central de ordem 2 para discretizar a derivada. Compute o erro absoluto médio definido por

$$E := \frac{1}{N}\sum_{i=1}^{N} |u(x_i) - u_i| \,,$$

onde x_i é o i-ésimo ponto da malha, $i = 1, 2, \ldots, N$ e N é o número de pontos da mesma, u_i é a solução aproximada e $u(x_i)$ é a solução analítica, dada por

$$u(x) = -\frac{25(x-1)^4}{3} - \frac{25x}{3} + \frac{25}{3}.$$

Solução: Primeiro, vamos montar a discretização do problema que está definido no domínio [0,1]. A malha é dada por:

$$x_i = (i-1)h, \quad i = 1, 2, \ldots, N,$$

com $h = 1/(N-1)$.

Aplicando na equação diferencial o método de diferenças finitas central de ordem 2 para aproximar a derivada segunda, obtemos as seguintes $N-2$ equações:

$$-\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} = 100(x_i - 1)^2, \quad i = 2, \ldots, N-1.$$

As equações de contorno levam as condições:

$$u_1 = u_N = 0.$$

Ou seja, obtemos o seguinte sistema linear $N \times N$:

$$\begin{aligned} u_1 &= 0, && (4.2)\\ -\frac{1}{h^2}\left(u_{i-1} - 2u_i + u_{i+1}\right) &= 100(x_i - 1)^2, \quad i = 2, \ldots, N-1, && (4.3)\\ u_N &= 0. && (4.4) \end{aligned}$$

Observamos que este é um sistema linear $N \times N$, o qual pode ser escrito na forma matricial $A\underline{u} = b$, cujos matriz de coeficientes é

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix},$$

o vetor das incógnitas e o vetor dos termos constantes são

$$\underline{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_N \end{bmatrix} \quad \text{e} \quad b = \begin{bmatrix} 0 \\ -100h^2(x_2 - 1)^2 \\ -100h^2(x_3 - 1)^2 \\ \vdots \\ 0 \end{bmatrix}.$$

Vamos aplicar o método de Jacobi para resolver esse problema. Partimos do exemplo 4.2.2 e introduzimos a matriz oriunda da discretização:

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

#define N 101 /* dimensão do sistema*/

double norma_2(double x[N])
{
  int i;
  double norma2=x[0]*x[0]/(2*(N-1));
  for (i=1;i<N-1;i++)
  {
    norma2+=x[i]*x[i]/(N-1);
  }
  norma2+=x[N-1]*x[N-1]/(2*(N-1));
  return sqrt(norma2);
}
void Iteracao(double u_antigo[N],double u_atual[N],double matriz[N][N], double vetor[
{
  double aux;
  int i,j;
  for (i=0;i<N;i++)
  {
    aux=0;
    for (j=0;j<i;j++)
    aux+=matriz[i][j]*u_antigo[j];
    for (j=i+1;j<N;j++)
    aux+=matriz[i][j]*u_antigo[j];
    u_atual[i]=(vetor[i]-aux)/matriz[i][i];
  }
}
void Jacobi(double tolerancia,double u_atual[N],double vetor[N],double matriz[N][N],d
{

  double u_antigo[N],dif[N];
  int i,controle=3;

  //chute inicial
  for (i=0;i<N;i++)u_antigo[i]=0;

  //iteração
```

```
  while (controle)
  {
    Iteracao(u_antigo,u_atual,matriz,vetor);
    for (i=0;i<N;i++) dif[i]=u_atual[i]-u_antigo[i];
    if (norma_2(dif)<tolerancia) controle--;
    else controle=3;
    for (i=0;i<N;i++) u_antigo[i]=u_atual[i];
  }
  return;
}

double solucao_analitica(double x)
{
return -25.*(x-1.)*(x-1.)*(x-1.)*(x-1.)/3-25.*x/3+25./3.;
}

main (void)
{
  int i,j;
  double tolerancia=1e-5,solucao[N];
  double matriz[N][N];
  double vetor[N],x[N];
  //malha
  double h=1./(N-1);
  for (i=0;i<N;i++) x[i]=i*h;


  //matriz e vetor
  for (i=0;i<N;i++) for (j=0;j<N;j++) matriz[i][j]=0;
  matriz[0][0]=1;
  vetor[0]=0;
  for (i=1;i<N-1;i++)
  {
    matriz[i][i]=-2;
    matriz[i][i+1]=1;
    matriz[i][i-1]=1;
    vetor[i]=-100*h*h*(x[i]-1)*(x[i]-1);
  }
  matriz[N-1][N-1]=1;
  vetor[N-1]=0;
```

```
  Jacobi(tolerancia,solucao,vetor,matriz,x);
  //erro médio
  double erro=0;
  for (i=0;i<N;i++) erro+=fabs(solucao[i]-solucao_analitica(x[i]));
  erro/=N;
  printf("erro médio = %f\n",erro);
  //for (i=0;i<N;i++)     printf("u_%d=%f, u(x[%d]) = %f\n",i,solucao[i],i,
  solucao_analitica(x[i]));
}
```

4.4 Exercícios

E 4.4.1. Implemente um código para resolver o sistema

$$\begin{aligned} 5x_1 + x_2 + x_3 - x_4 &= 1 \\ -x_1 - 4x_2 + 2x_4 &= -2 \\ -2x_1 + 2x_2 - 5x_3 - x_4 &= -5 \\ x_2 - x_3 + 5x_4 &= -5 \end{aligned}$$

com o método de Jacobi.

E 4.4.2. Implemente um código para resolver o sistema

$$\begin{aligned} 5x_1 + x_2 + x_3 - x_4 &= 1 \\ -x_1 - 4x_2 + 2x_4 &= -2 \\ -2x_1 + 2x_2 - 5x_3 - x_4 &= -5 \\ x_2 - x_3 + 5x_4 &= -5 \end{aligned}$$

com o método de Gauss-Seidel.

A iteração de Gauss-Seidel para resolver o sistema 4.1 é

$$\begin{aligned} x_1^{(k+1)} &= \frac{y_1 - \left(a_{12}x_2^{(k)} + \cdots + a_{1n}x_n^{(k)}\right)}{a_{11}} \\ x_2^{(k+1)} &= \frac{y_2 - \left(a_{21}x_1^{(k+1)} + a_{23}x_3^{(k)} + \cdots + a_{2n}x_n^{(k)}\right)}{a_{22}} \\ &\vdots \\ x_n^{(k+1)} &= \frac{y_2 - \left(a_{n1}x_1^{(k+1)} + \cdots + a_{n(n-1)}x_{n-1}^{(k+1)}\right)}{a_{nn}} \end{aligned}$$

onde x_1^0, x_2^0, $\cdots$, x_n^0 é um chute inicial.

E 4.4.3. Um sistema tridiagonal é um sistema de equações lineares cuja matriz associada é tridiagonal, conforme a seguir:

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}.$$

A solução do sistema tridiagonal acima pode ser obtido pelo algoritmo de Thomas dado por:

$$c'_i = \begin{cases} \frac{c_i}{b_i}, & i = 1 \\ \frac{c_i}{b_i - a_i c'_{i-1}}, & i = 2, 3, \ldots, n-1 \end{cases}$$

e

$$d'_i = \begin{cases} \frac{d_i}{b_i}, & i = 1 \\ \frac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}}, & i = 2, 3, \ldots, n. \end{cases}$$

Finalmente a solução final é obtida por substituição reversa:

$$\begin{aligned} x_n &= d'_n \\ x_i &= d'_i - c'_i x_{i+1}, \qquad i = n-1, n-2, \ldots, 1. \end{aligned}$$

Implemente um código para resolver sistemas tridiagonais e teste resolvendo o

problema

$$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 4 \\ 4 \\ 0 \\ 0 \\ 2 \end{bmatrix}.$$

E 4.4.4. Refaça o exercício 4.4.1 usando a estrutura de matrizes.

E 4.4.5. Escreva um programa que lê as entradas de uma matriz diagonal estritamente dominante A e um vetor B e imprime a solução do sistema $Ax = B$. Use o método iterativo de Gauss-Seidel.

E 4.4.6. Resolva numericamente o problema de valor de contorno para a equação de calor no estado estacionário

$$\begin{cases} -u_{xx} = 32, \ \ 0 < x < 1. \\ u(0) = 5 \\ u(1) = 10 \end{cases}$$

usando o método de diferenças finitas.

E 4.4.7. Resolva numericamente o problema de valor de contorno para a equação de calor no estado estacionário

$$\begin{cases} -u_{xx} = 200e^{-(x-1)^2}, \ \ 0 < x < 2. \\ u(0) = 120 \\ u(2) = 100 \end{cases}$$

usando o método de diferenças finitas.

E 4.4.8. Resolva numericamente o problema de valor de contorno para a equação de calor no estado estacionário

$$\begin{cases} -u_{xx} = 200e^{-(x-1)^2}, \ \ 0 < x < 2. \\ u'(0) = 0 \\ u(2) = 100 \end{cases}$$

usando o método de diferenças finitas.

Capítulo 5

Ponteiros

5.1 Endereços

Uma variável é armazenada na memória do computador com três informações básicas:

- O tipo está relacionado ao tamanho em bytes da posição - `double` tem 64 bits (ou 8 bytes);
- O endereço se refere a posição da memória onde está localizada - dado por um hexadecimal;
- O valor da variável que é armazenada nesse endereço.

O nome da variável é a forma fácil do programador se comunicar com a máquina. O compilador se encarrega de converter o nome da variável em posição da memória ou valor atribuído. Para chamar o valor atribuído, basta usar o próprio nome da variável. Agora, para obter o endereço de memória da variável, usa-se o operador `&`.

Exemplo 5.1.1. Escreva um programa que lê um inteiro e imprime o seu valor e seu endereço na memória.

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

int main(void)
{
  int x;
  printf("digite o valor de x\n\n");
  scanf("%d",&x);
```

```
  printf("O valor de x é %d\n",x);
  printf("O endereço de x na memória %p\n",&x);
}
```

Na linha `int x;` nós definimos uma variável com o nome x do tipo `int`. Nesse momento, foi alocado um espaço na memória para armazenar x, que tem um certo endereço. Na linha `scanf("%d",&x)` foi atribuído um certo valor para x.

Observamos que as variável podem ocupar vários bytes de memória e, nesse caso, o endereço da variável é aquele do primeiro byte.

5.2 Ponteiros

Ponteiros são variáveis que ocupam uma posição da memória mas estão preparados para armazenar endereços de outras variáveis. Usamos um asterístico para definir uma variável do tipo ponteiro. Por exemplo, um ponteiro apontado para um `double` por ser definido por `double *p;`.

Exemplo 5.2.1. Escreva um programa que lê um inteiro e armazena o endereço num ponteiro.

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

int main(void)
{
  int x;
  int *y=NULL;
  printf("digite o valor de x\n\n");
  scanf("%d",&x);
  y=&x;
  printf("O valor de x é %d\n",x);
  printf("O endereço de x na memória %p\n",y);
}
```

Observe o detalhe na linha `y=&x;`, onde o ponteiro `y` recebe o endereço da variável `x`. A linha `int *y=NULL;` pode ser substituída por `int *y;`. A diferença é que na primeira dissemos explicitamente que o ponteiro `int` não está apontado para nenhuma variável e, na segunda não demos carga inicial. É sempre indicado dar carga inicial em ponteiros para produzir códigos mais seguros.

Se definirmos uma variável do tipo ponteiro para `double` com o nome `p`, então `*p` é o valor apontado pelo ponteiro. Observe uma versão do código do exemplo 5.2.1 que produz o mesmo resultado:

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

int main(void)
{
  int x;
  int *y=NULL;
  printf("digite o valor de x\n\n");
  scanf("%d",&x);
  y=&x;
  printf("O valor de x é %d\n",*y);
  printf("O endereço de x na memória %p\n",y);
}
```

A linha `printf("O valor de x é %d\n",*y);` imprime o valor apontado pelo ponteiro `y`, que é o valor de `x`.

O ponteiro é uma variável como qualquer outra, o que permite a definição de um ponteiro que aponta outro ponteiro.

Exemplo 5.2.2. Escreva um programa define um `double`, depois um ponteiro para esse `double`, depois um ponteiro para o primeiro ponteiro.

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

int main(void)
{
  double x,*y,**z;
  x=2.3;
  y=&x;
  z=&y;
  printf("O valor de x é %f\n",x);
  printf("O endereço de x é %p\n",&x);
  printf("O valor de y é %p\n",y);
  printf("O endereço de y é %p\n",&y);
  printf("O valor apontado por y é %f\n",*y);
  printf("O valor de z é %p\n",z);
  printf("O endereço de z é %p\n",&z);
  printf("O valor apontado por z é %p\n",*z);
  printf("O valor apontado pela variável que está apontada por z é %f\n",**z);
}
```

A instrução que atribui um valor para a uma variável através do ponteiro também é funciona.

Exemplo 5.2.3. Escreva uma programa que define uma variável sem atribuir valor, depois define um ponteiro apontado para essa variável e, finalmente, atribui valor para a variável usando o ponteiro.

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

int main(void)
{
  double x;
  double *y=&x;
  *y=2.4;
  printf("O valor de x é %f\n",x);
}
```

A necessidade de indicar o tipo de variável para onde o ponteiro aponta é devido ao tamanho de cada tipo de variável, que pode ocupar vários bytes. O ponteiro aponta para o primeiro byte e a informação do tipo diz o tamanho da variável em bytes.

O endereço de um vetor `v` é dado pelo endereço do seu primeiro elemento `&v[0]`, que também pode ser chamado por `v`. Naturalmente, cada elemento também tem um endereço, sequencialmente a partir do primeiro.

Exemplo 5.2.4. Escreva um programa que define um vetor e imprime seu endereço.

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

int main(void)
{
  double x[3]={2.3,3.1,-1.2};
  printf("O primeiro elemento é %f\n",x[0]);
  printf("Esse é o endereço do primeiro elemento do vetor %p\n",&x[0]);
  printf("Esse é o endereço do vetor %p\n",x);
  printf("Esse é o endereço do segundo elemento do vetor %p\n",&x[1]);
  printf("Esse é o endereço do terceiro elemento do vetor %p\n",&x[2]);
}
```

5.3 Incremento e decremento de ponteiros

Um ponteiro avança/recua o tamanho do tipo de variável para cada unidade de incremento/decremento. A diferença entre dois ponteiros diz a distância entre bytes entre eles. A comparação entre ponteiros diz quem está mais a direita.

Exemplo 5.3.1. Escreva um código que incremente/decremente ponteiros para vetores de `double` e imprime os valores, endereços, diferenças, etc.

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

main (void)
{
  double v[3]={1.1,2.2,3.3}, *pv=v,*pv2;
  printf("O endereço do vetor v é %p\n",v);//hexadecimal
  printf("O endereço do vetor v é %ld\n",(long int) v);//decimal
  printf("O ponteiro *pv aponta para o endereço %p\n",pv);//hexadecimal
  printf("O ponteiro *pv aponta para o endereço %ld\n",(long int) pv);//decimal
  printf("O endereço de *pv é %ld\n",(long int) &pv);//decimal
  printf("O valor apontado por *pv é %f\n",*pv);
  pv++;
  printf("O endereço de *pv é %ld\n",(long int) &pv);//decimal
  printf("O ponteiro *pv aponta para o endereço %ld\n",(long int) pv);//decimal
  printf("O valor apontado por *pv é %f\n",*pv);
  printf("O ponteiro caminhou %ld bytes\n",sizeof(double));
  pv2=pv+2;
  printf("O valor apontado por *pv2 é %f\n",*pv2);
  printf("A diferença entre pv2 e pv é %ld double\n",pv2-pv);
  pv2--;
  printf("O valor apontado por *pv2 é %f\n",*pv2);
  if (pv2<pv) printf("pv2 está à esquerda de pv\n");
  else printf("pv2 está à direita de pv\n");
}
```

5.4 Ponteiros - acesso aos elementos de um vetor

Podemos usar ponteiros para acessar os elementos de um vetor.

Exemplo 5.4.1. Declare um vetor com quatro números e acesse o terceiro via notação de vetor e via ponteiro.

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

main (void)
{
  double a[4]={1,2,3,4},*p=a;
  printf("O terceiro número é %f\n",a[2]);//acesso normal
  printf("O terceiro número é %f\n",*(a+2));//acesso via endereço
  printf("O terceiro número é %f\n",*(p+2));//acesso via ponteiro
  printf("O terceiro número é %f\n",p[2]);//acesso via ponteiro
}
```

A última notação `p[2]` pode parecer a mais estranha, pois usamos a notação de vetor para acessar o valor dois doubles a direita de onde aponta p.

5.5 Exercícios

E 5.5.1. Escreva um programa para calcular o limite da sequência

$$\begin{cases} x_0 = \dfrac{1}{3} \\ \\ x_n = \dfrac{x_{n-1}+1}{4}, \qquad n = 1,2,... \end{cases}$$

Siga os seguintes passos:

- Calcule x_n em termos de x_{n-1} e calcule a diferença $|x_n - x_{n-1}|$. Execute até que a diferença seja menor que 10^{-10}.
- Defina ponteiros apontados para x_n e x_{n-1} e, quando for atualizar a iteração, não troque os valores de x_n e x_{n-1} de posição na memória, mas apenas inverta os ponteiros.

E 5.5.2. Digite um programa que lê dois números tipo float e imprima o maior. Faça uma função que entra dois ponteiros apontados para float e compare os valores apontados.

E 5.5.3. Sem implementar, escreva o que o programa abaixo vai escrever na tela. Depois implemente e teste.

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

main (void)
{
  double a=0.9,*b=&a,**c=&b;
  printf("%f, %p, %p\n",a,b,c);
  printf("%p, %p, %p\n",&a,&b,&c);
  printf("%f, %p, %f\n",*b,*c,**c);
}
```

Naturalmente, você não saberá qual o número que será impresso em um endereço, pois o computador vai automaticamente alocar uma posição da memória, mas certamente você poderá comparar os endereços entre si.

Capítulo 6

Passagem de parâmetros

6.1 Passagem de vetores para função

Exemplo 6.1.1. Escreva um programa com uma função que recebe um vetor e devolve outro vetor contento mínimo, máximo e norma.

```
#include <stdio.h>
#include <math.h>
#define N 4
void teste(double x[N], double saida[3])
{
  double max=x[0], norma=0, min=x[0];
  int i;
  for (i=0;i<N;i++)
  {
    norma+=x[i]*x[i];
    if (max<x[i]) max=x[i];
    if (min>x[i]) min=x[i];
  }
saida[0]=min;
saida[1]=max;
saida[2]=sqrt(norma);
return;
}


main (void)
{
  double a[N]={1,-2,3,4},saida[3];
```

```
  teste(a,saida);
  printf("mínimo=%f, máximo=%f, norma=%f\n",saida[0],saida[1],saida[2]);
}
```

Quando passamos um vetor de N posições para dentro de uma função, não estamos levando N valores, mas um único valor que é o endereço de memória dos vetores. Veja como chamamos a linha `teste(a,saida);`. Aqui `a` e `saida` são endereços de memória dos vetores. Naturalmente, a protótipo `void teste(double x[N], double saida[3])` também funciona da forma `void teste(double *x, double *saida)`, pois, de fato, estamos passando um ponteiro.

```
#include <stdio.h>
#include <math.h>
#define N 4
void teste(double *x, double *saida)
{
  double max=*x, norma=0, min=*x;
  int i;
  for (i=0;i<N;i++)
  {
    norma+=*(x+i)**(x+i);
    if (max<*(x+i)) max=*(x+i);
    if (min>*(x+i)) min=*(x+i);
  }
*saida=min;
saida++;
*saida=max;
saida++;
*saida=sqrt(norma);
return;
}
main (void)
{
  double a[N]={1,-2,3,4},saida[3];
  teste(a,saida);
  printf("mínimo=%f, máximo=%f, norma=%f\n",saida[0],saida[1],saida[2]);
}
```

Uma versão mista do código também funciona.

Uma grande vantagem dos ponteiros é evitar a copia um vetor inteiro de uma posição da memória para outra, mas apenas trocar um ponteiro que aponta para

o tal vetor. Isso reduz significantemente o custo computacional de um processo iterativo.

Exemplo 6.1.2. Vamos refazer o exemplo 4.3.1, que resolve um problema de contorno, melhorando a performance do processo iterativo com uso de ponteiros

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

#define N 101 /* dimensão do sistema*/

double norma_2(double x[N])
{
  int i;
  double norma2=x[0]*x[0]/(2*(N-1));
  for (i=1;i<N-1;i++)
  {
    norma2+=x[i]*x[i]/(N-1);
  }
  norma2+=x[N-1]*x[N-1]/(2*(N-1));
  return sqrt(norma2);
}
void Iteracao(double u_antigo[N],double u_atual[N],double matriz[N][N], double vetor[
{
  double aux;
  int i,j;
  for (i=0;i<N;i++)
  {
    aux=0;
    for (j=0;j<i;j++)
    aux+=matriz[i][j]*u_antigo[j];
    for (j=i+1;j<N;j++)
    aux+=matriz[i][j]*u_antigo[j];
    u_atual[i]=(vetor[i]-aux)/matriz[i][i];
  }
}
void Jacobi(double tolerancia,double u_atual[N],double vetor[N],double matriz[N][N],d
{
  double u_antigo[N],dif[N];
  int i,controle=3;
  double *pu_antigo=u_antigo,*pu_atual=u_atual,*pswap;
```

```
  //chute inicial
  for (i=0;i<N;i++)u_antigo[i]=0;

  //iteração
  while (controle)
  {
    Iteracao(pu_antigo,pu_atual,matriz,vetor);
    for (i=0;i<N;i++) dif[i]=u_atual[i]-u_antigo[i];
    if (norma_2(dif)<tolerancia) controle--;
    else controle=3;
    pswap=pu_atual;
    pu_atual=pu_antigo;
    pu_antigo=pswap;
  }
  return;
}

double solucao_analitica(double x)
{
return -25.*(x-1.)*(x-1.)*(x-1.)*(x-1.)/3-25.*x/3+25./3.;
}

main (void)
{
  int i,j;
  double tolerancia=1e-10,solucao[N];
  double matriz[N][N];
  double vetor[N],x[N];
  //malha
  double h=1./(N-1);
  for (i=0;i<N;i++) x[i]=i*h;


  //matriz e vetor
  for (i=0;i<N;i++) for (j=0;j<N;j++) matriz[i][j]=0;
  matriz[0][0]=1;
  vetor[0]=0;
  for (i=1;i<N-1;i++)
  {
```

```
    matriz[i][i]=-2;
    matriz[i][i+1]=1;
    matriz[i][i-1]=1;
    vetor[i]=-100*h*h*(x[i]-1)*(x[i]-1);
  }
  matriz[N-1][N-1]=1;
  vetor[N-1]=0;

  Jacobi(tolerancia,solucao,vetor,matriz,x);
  //erro médio
  double erro=0;
  for (i=0;i<N;i++) erro+=fabs(solucao[i]-solucao_analitica(x[i]));
  erro/=N;
  printf("erro médio = %f\n",erro);
//  for (i=0;i<N;i++)     printf("u_%d=%f, u(x[%d]) = %f\n",i,solucao[i],i,solucao_an
}
```

6.2 Passagem de parâmetros

Para entender como funciona a passagem de parâmetros, vamos trabalhar com um exemplo.

Exemplo 6.2.1. Implemente um programa que troca os valores de duas variáveis entre si.

Uma solução simples para o exemplo acima é implementar a troca direto no main.

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

void main(void)
{
  double a=2,b=3,tmp;
  printf("a=%f, b=%f\n",a,b);
  tmp=a;
  a=b;
  b=tmp;
  printf("a=%f, b=%f\n",a,b);
}
```

No entanto, suponha que gostaríamos de separar um função para fazer a troca. Então, a função entrar `a` e `b` e devolver com valores trocados. Teste o código abaixo:

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

void troca(double a,double b)
{
  double tmp=a;
  a=b;
  b=tmp;
}

void main(void)
{
  double a=2,b=3;
  printf("a=%f, b=%f\n",a,b);
  troca(a,b);
  printf("a=%f, b=%f\n",a,b);
}
```

Apesar de compilar e rodar sem erros, o código não faz o que queremos. Isso se deve ao fato que quando executamos um função, o programa separa uma parte da memória para o trabalho, faz cópia das valores do parâmetros de entrada, inicia variáveis locais, e, no fim da rotina, apaga toda a parte da memória usada para esse fim. Na prática, a função `troca` criou um espaço na memória, copiou os valores de `a` e `b` e trocou eles de lugar. Mas os valores de `a` e `b` nas posições originais não se alteraram. Isso de resolve se passarmos uma referência para a função e não um valor.

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

void troca(double *a,double *b)
{
  double tmp=*a;
  *a=*b;
  *b=tmp;
}

void main(void)
```

```
{
  double a=2,b=3;
  printf("a=%f, b=%f\n",a,b);
  troca(&a,&b);
  printf("a=%f, b=%f\n",a,b);
}
```

Agora, a função troca criou um ambiente e levou os endereços de `a` e `b`. Com os endereços, pudemos acessar os valores de `a` e `b` que estão fora desse ambiente e trocá-los de posição.

Em resumo, quando queremos apenas usar um parâmetro dentro de uma função, podemos passar o valor. Mas se queremos atualizar seu valor na passagem pela função, deveremos passar a referência.

Vamos refazer o mesmo exemplo de troca usando vetor de duas posições

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

void troca(double v[2])
{
  double tmp=v[0];
  v[0]=v[1];
  v[1]=tmp;
}

void main(void)
{
  double v[2]={2,3};
  printf("a=%f, b=%f\n",v[0],v[1]);
  troca(v);
  printf("a=%f, b=%f\n",v[0],v[1]);
}
```

Agora, nós passamos um vetor e o atualizamos, que parece contradizer aquilo que acabamos de dizer. Mas não se engane, quando usamos vetor, automaticamente o função já passa apenas a referência. Veja uma forma alternativa abaixo:

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

void troca(double *v)
{
```

```
  double tmp=*v;
  *v=*(v+1);
  *(v+1)=tmp;
}

void main(void)
{
  double v[2]={2,3};
  printf("a=%f, b=%f\n",v[0],v[1]);
  troca(v);
  printf("a=%f, b=%f\n",v[0],v[1]);
}
```

Exemplo 6.2.2. Implemente uma rotina para calcular a raíz de uma função pelo método de Newton passando o chute inicial e retornando a solução e o número de iterações necessárias.

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

double f(double x)
{
  return cos(x)-x;
}
double df(double x)
{
  return -sin(x)-1.;
}

void Newton(double x0, double tol, double *xn, int *k)
{
  double dif;
  *k=0;
  do
  {
  (*k)++;
  *xn=x0-f(x0)/df(x0);
  dif=fabs(*xn-x0);
  x0=*xn;
  }while (dif>tol);
}
```

```
void main(void)
{
  double x0=2,xn,tol=1e-10;
  int k;
  Newton(x0,tol,&xn,&k);
  printf("Sol=%f, iterações=%d\n",xn,k);
}
```

6.3 Problemas

Você deve ter observado que todos os exercícios na aula 8 resultam em um sistema tridiagonal. O algorítmo de Thomas é um método eficiente para resolvê-los.

Exemplo 6.3.1. Implemente o método de Thomas para resolver um sistema $Ax = b$ de dimensão N.

O algoritmo TDMA para resolver o sistema

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}.$$

é dado por

$$c'_i = \begin{cases} \frac{c_i}{b_i}, & i = 1 \\ \frac{c_i}{b_i - a_i c'_{i-1}}, & i = 2, 3, \ldots, n-1 \end{cases},$$

$$d'_i = \begin{cases} \frac{d_i}{b_i}, & i = 1 \\ \frac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}}, & i = 2, 3, \ldots, n. \end{cases}$$

e

$$\begin{aligned} x_n &= d'_n \\ x_i &= d'_i - c'_i x_{i+1}, \qquad i = n-1, n-2, \ldots, 1. \end{aligned}$$

```
#include <stdio.h>
#include <math.h>
#define N 3

//Método de Thomas para resolver Ax=y
//x entra o vetor y e saí a solução x (N posições, i=0,....,N-1)
//a entra a subdiagonal à esquerda de A (N-1 posições, i=1,...,N-1)
//b entra a diagonal  (N posições, i=0,....,N-1)
//c entra a subdiagonal à direita de A (N-1 posições, i=0,...,N-2)
void Thomas(double x[], const double a[], const double b[], double c[])
{
  int i;

  /*calculo de c[0]' e d[0]'*/
  c[0] = c[0] / b[0];
  x[0] = x[0] / b[0];


  /* laço para calcular c' e d' */
  for (i = 1; i < N; i++)
  {
    double aux = 1.0/ (b[i] - a[i] * c[i - 1]);
    c[i] = c[i] * aux;
    x[i] = (x[i] - a[i] * x[i - 1]) * aux;
  }

  /* Calculando a solução */
  for (i = N - 1; i >= 0; i--)
    x[i] = x[i] - c[i] * x[i + 1];
}


main (void)
{
  double x[N],a[N],b[N],c[N];
  a[0]=0;c[N-1]=0;

  x[0]=4;x[1]=9;x[2]=11;
  a[1]=1;a[2]=1;
  b[0]=3;b[1]=3;b[2]=3;
```

```
  c[0]=1;c[1]=1;
  Thomas(x,a,b,c);
  printf("%f, %f, %f\n",x[0],x[1],x[2]);
}
```

Exemplo 6.3.2. Use o algoritmo TDMA para resolver o PVC

$$\begin{cases} -u_{xx} + u_x = 200e^{-(x-1)^2}, \ \ 0 < x < 1. \\ 15u(0) + u'(0) = 500 \\ 10u(1) + u'(1) = 1 \end{cases}$$

Uma versão discreta oriunda do método de diferenças finitas é

$$\begin{aligned} &15u_1 + \frac{u_2 - u_1}{h} = 500 \\ &\frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} + \frac{u_{i+1} - u_{i-1}}{2h} = 200e^{-(x_i-1)^2}, \qquad i = 2,3,...,N-1 \\ &10u_N + \frac{u_N - u_{N-1}}{h} = 1 \end{aligned}$$

onde $x_i = h(i-1)$, $i = 1,2,...,N$, $h = \frac{1}{N-1}$. Isto é,

$$\begin{aligned} &u_1\left(15 - \frac{1}{h}\right) + u_2\left(\frac{1}{h}\right) = 500 \\ &u_{i-1}\left(-\frac{1}{h^2} - \frac{1}{2h}\right) + u_i\left(\frac{2}{h^2}\right) + u_{i+1}\left(-\frac{1}{h^2} + \frac{1}{2h}\right) = 200e^{-(x_i-1)^2}, \qquad i = 2,3,...,N-1 \\ &u_{N-1}\left(-\frac{1}{h}\right) + u_N\left(10 + \frac{1}{h}\right) = 1 \end{aligned}$$

ou ainda

$$\begin{bmatrix} 15 - \frac{1}{h} & \frac{1}{h} & 0 & 0 & 0 & \cdots & 0 \\ -\frac{1}{h^2} - \frac{1}{2h} & \frac{2}{h^2} & -\frac{1}{h^2} + \frac{1}{2h} & 0 & 0 & \cdots & 0 \\ 0 & -\frac{1}{h^2} - \frac{1}{2h} & \frac{2}{h^2} & -\frac{1}{h^2} + \frac{1}{2h} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots \\ 0 & \cdots & 0 & 0 & -\frac{1}{h^2} - \frac{1}{2h} & \frac{2}{h^2} & -\frac{1}{h^2} + \frac{1}{2h} \\ 0 & \cdots & 0 & 0 & 0 & -\frac{1}{h} & 10 + \frac{1}{h} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix} = \begin{bmatrix} 500 \\ 200e^{-(x_2-1)^2} \\ 200e^{-(x_3-1)^2} \\ \vdots \\ 200e^{-(x_{N-1}-1)^2} \\ 1 \end{bmatrix}$$

```
#include <stdio.h>
#include <math.h>
#define N 11

//Método de Thomas para resolver Ax=y
//x entra o vetor y e saí a solução x (N posições, i=0,....,N-1)
//a entra a subdiagonal à esquerda de A (N-1 posições, i=1,...,N-1)
//b entra a diagonal  (N posições, i=0,....,N-1)
//c entra a subdiagonal à direita de A (N-1 posições, i=0,...,N-2)
void Thomas(double x[], const double a[], const double b[], double c[])
{
  int i;

  /*calculo de c[0]' e d[0]'*/
  c[0] = c[0] / b[0];
  x[0] = x[0] / b[0];


  /* laço para calcular c' e d' */
  for (i = 1; i < N; i++)
  {
    double aux = 1.0/ (b[i] - a[i] * c[i - 1]);
    c[i] = c[i] * aux;
    x[i] = (x[i] - a[i] * x[i - 1]) * aux;
  }

  /* Calculando a solução */
  for (i = N - 1; i >= 0; i--)
    x[i] = x[i] - c[i] * x[i + 1];
}


main (void)
{
  double x[N],a[N],b[N],c[N];
  a[0]=0;c[N-1]=0;

  //malha
  int i;
  double h=1./(N-1);
```

```
  double p[N];
  for (i=0;i<N;i++) p[i]=i*h;

  //sistema
  b[0]=15-1/(h);
  c[0]=1/(h);
  x[0]=500;
  for (i=1;i<N-1;i++)
  {
  b[i]=2/(h*h);
  a[i]=-1/(h*h)-1/(2*h);
  c[i]=-1/(h*h)+1/(2*h);
  x[i]=200*exp(-(p[i]-1)*(p[i]-1));
  }
  b[N-1]=10+1/h;
  a[N-1]=-1/h;
  x[N-1]=1;
  Thomas(x,a,b,c);
  for (i=0;i<N;i++) printf("%f\n",x[i]);
}
```

Com o objetivo de resolver alguns problemas de contorno não lineares, vamos implementar o método de Newton para sistema.

Exemplo 6.3.3. Implemente um código para calcular raízes do sistema não linear

$$\begin{aligned} x_1^2 &= \cos(x_1 x_2) + 1 \\ \operatorname{sen}(x_2) &= 2\cos(x_1). \end{aligned}$$

Calcule uma aproximação para a raíz que fica próxima ao ponto $x_1 = 1{,}5$ e $x_2 = 0{,}5$.

O método iterativo de Newton-Raphson para encontrar as raízes de $F(x) = 0$,

$$F(x) = \begin{bmatrix} f_1(x_1,x_2,\ldots,x_n) \\ f_2(x_1,x_2,\ldots,x_n) \\ \vdots \\ f_n(x_1,x_2,\ldots,x_n) \end{bmatrix},$$

é dado por

$$\begin{cases} x^{(0)} &= \text{dado inicial} \\ x^{(k+1)} &= x^{(k)} - J_F^{-1}\left(x^{(k)}\right) F(x^{(k)}), \quad k \geq 0 \end{cases}$$

onde a matriz Jacobina J_F é

$$J_F = \frac{\partial(f_1,f_2,\ldots,f_n)}{\partial(x_1,x_2,\ldots,x_n)} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \ldots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \ldots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \ldots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}.$$

Especificamente no problema do exemplo 6.3.3, temos

$$F(x) = \begin{bmatrix} x_1^2 - \cos(x_1 x_2) - 1 \\ \operatorname{sen}(x_2) - 2\cos(x_1) \end{bmatrix}.$$

e

$$J_F(x) = \begin{bmatrix} 2x_1 + x_2 \operatorname{sen}(x_1 x_2) & x_1 \operatorname{sen}(x_1 x_2) \\ 2\operatorname{sen}(x_1) & \cos(x_2) \end{bmatrix}.$$

Observamos que na prática não invertemos a matriz Jacobina, mas calculamos a solução do sistema $J_F\delta = F$:

$$\begin{cases} x^{(0)} = \begin{bmatrix} 1{,}5 \\ 0{,}5 \end{bmatrix} \\ J_F(x^{(k)})\delta^{(k)} = F(x^{(k)}) \quad \text{(Calcula-se a solução } \delta^{(k)} \text{ do sistema)} \\ x^{(k+1)} = x^{(k)} - \delta^{(k)}, \ k \geq 0 \end{cases}$$

```
#include <stdio.h>
#include <math.h>
#define N 2

//Método de Thomas para resolver Ax=y
//x entra o vetor y e saí a solução x (N posições, i=0,....,N-1)
//a entra a subdiagonal à esquerda de A (N-1 posições, i=1,...,N-1)
```

```
//b entra a diagonal  (N posições, i=0,....,N-1)
//c entra a subdiagonal à direita de A (N-1 posições, i=0,...,N-2)
void Thomas(double x[], const double a[], const double b[], double c[])
{
  int i;

  /*calculo de c[0]' e d[0]'*/
  c[0] = c[0] / b[0];
  x[0] = x[0] / b[0];


  /* laço para calcular c' e d' */
  for (i = 1; i < N; i++)
  {
    double aux = 1.0/ (b[i] - a[i] * c[i - 1]);
    c[i] = c[i] * aux;
    x[i] = (x[i] - a[i] * x[i - 1]) * aux;
  }

  /* Calculando a solução */
  for (i = N - 1; i >= 0; i--)
    x[i] = x[i] - c[i] * x[i + 1];
}
//Função
//x é o ponto de entrada
//f é o vetor com o resultado de f(x)
void F(double *x,double *f)
{
  f[0]=x[0]*x[0]-cos(x[0]*x[1])-1.0;
  f[1]=sin(x[1])-2*cos(x[0]);
}
//Jacobiana
//x é o ponto de entrada
//a sai a subdiagonal à esquerda de A (N-1 posições, i=1,...,N-1)
//b sai a diagonal  (N posições, i=0,....,N-1)
//c sai a subdiagonal à direita de A (N-1 posições, i=0,...,N-2)
void J(double *x,double *a,double *b,double *c)
{
  b[0]=2*x[0]+x[1]*sin(x[0]*x[1]);
  a[0]=0;
```

```
  c[0]=x[0]*sin(x[0]*x[1]);
  a[1]=2*sin(x[0]);
  c[1]=0;
  b[1]=cos(x[1]);
}
//norma da diferença entre dois vetores
double norma_dif(double x[N],double y[N])
{
  int i;
  double norma2=0;
  for (i=0;i<N;i++)
  {
    norma2+=(x[i]-y[i])*(x[i]-y[i]);
  }
  return sqrt(norma2);
}
main (void)
{
  int i;
  double x0[N],x[N],d[N],a[N],b[N],c[N],tol=1e-10,*px0=x0,*px=x,*paux=NULL;
  x0[0]=1.5;
  x0[1]=0.5;
  do
  {
    F(px0,d);
    J(px0,a,b,c);
    Thomas(d,a,b,c);
    for (i=0;i<N;i++) *(px+i)=*(px0+i)-d[i];
    for (i=0;i<N;i++) printf("%.12f  ",*(px+i));
    printf("\n");
    paux=px0;
    px0=px;
    px=paux;
  }while(norma_dif(px0,px)>tol);
}
```

Exemplo 6.3.4. Use o método de diferenças finitas e o método de Newton para

resolver o problema de valor de contono não-linear

$$\begin{cases} -u_{xx} + u_x = e^{-u^2}, \ \ 0 < x < 1. \\ u'(0) + 3u(0) = 10 \\ u(1) = 5 \end{cases}$$

Uma versão discreta oriunda do método de diferenças finitas é

$$\begin{aligned} &3u_1 + \frac{u_2 - u_1}{h} = 10 \\ &\frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} + \frac{u_{i+1} - u_{i-1}}{2h} = e^{-u_i^2}, \qquad i = 2,3,...,N-1 \\ &u_N = 5 \end{aligned}$$

onde $x_i = h(i-1)$, $i = 1,2,...,N$, $h = \frac{1}{N-1}$. Podemos colocar na forma $F(u) = 0$, onde $u = [u_1 \ \ u_2 \ \cdots \ u_N]^T$ e

$$F(u) = \begin{cases} 3u_1 + \frac{u_2 - u_1}{h} - 10 \\ \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} + \frac{u_{i+1} - u_{i-1}}{2h} - e^{-u_i^2}, \qquad i = 2,3,...,N-1 \\ u_N - 5 \end{cases}$$

A jacobina desse sistema não linear é

$$J_F(u) = \begin{bmatrix} 3 - \frac{1}{h} & \frac{1}{h} & 0 & 0 & 0 & \cdots & 0 \\ -\frac{1}{h^2} - \frac{1}{2h} & \frac{2}{h^2} + 2u_i e^{-u_i^2} & -\frac{1}{h^2} + \frac{1}{2h} & 0 & 0 & \cdots & 0 \\ 0 & -\frac{1}{h^2} - \frac{1}{2h} & \frac{2}{h^2} + 2u_i e^{-u_i^2} & -\frac{1}{h^2} + \frac{1}{2h} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots \\ 0 & \cdots & 0 & 0 & -\frac{1}{h^2} - \frac{1}{2h} & \frac{2}{h^2} + 2u_i e^{-u_i^2} & -\frac{1}{h^2} + \frac{1}{2h} \\ 0 & \cdots & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Vamos implementar um código usando o método de Thomas para resolver os sistemas tridiagonais.

```
#include <stdio.h>
#include <math.h>
#define N 11
```

```
#define h (1./(N-1))

//Método de Thomas para resolver Ax=y
//x entra o vetor y e saí a solução x (N posições, i=0,....,N-1)
//a entra a subdiagonal à esquerda de A (N-1 posições, i=1,...,N-1)
//b entra a diagonal  (N posições, i=0,....,N-1)
//c entra a subdiagonal à direita de A (N-1 posições, i=0,...,N-2)
void Thomas(double x[], const double a[], const double b[], double c[])
{
  int i;

  /*calculo de c[0]' e d[0]'*/
  c[0] = c[0] / b[0];
  x[0] = x[0] / b[0];


  /* laço para calcular c' e d' */
  for (i = 1; i < N; i++)
  {
    double aux = 1.0/ (b[i] - a[i] * c[i - 1]);
    c[i] = c[i] * aux;
    x[i] = (x[i] - a[i] * x[i - 1]) * aux;
  }

  /* Calculando a solução */
  for (i = N - 1; i >= 0; i--)
    x[i] = x[i] - c[i] * x[i + 1];
}
//Função
//x é o ponto de entrada
//f é o vetor com o resultado de f(x)
void F(double *x,double *f)
{
  int i;
  f[0]=3*x[0]+(x[1]-x[0])/h-10;
  for (i=1;i<N-1;i++) f[i]=(-x[i+1]+2*x[i]-x[i-1])/(h*h)+(x[i+1]-x[i-1])/(2*h)-e
  f[N-1]=x[N-1]-5;
}
//Jacobiana
//x é o ponto de entrada
```

```
//a sai a subdiagonal à esquerda de A (N-1 posições, i=1,...,N-1)
//b sai a diagonal  (N posições, i=0,....,N-1)
//c sai a subdiagonal à direita de A (N-1 posições, i=0,...,N-2)
void J(double *x,double *a,double *b,double *c)
{
  int i;
  a[0]=0;
  b[0]=3.-1./h;
  c[0]=1./h;
  for (i=1;i<N-1;i++)
  {
    a[i]=-1./(h*h)-1./(2*h);
    b[i]=2./(h*h)+2*x[i]*exp(-x[i]*x[i]);
    c[i]=-1./(h*h)+1./(2*h);
  }
  a[N-1]=0;
  b[N-1]=1.;
  c[N-1]=0;
}
//norma da diferença entre dois vetores
double norma_max(double x[N])
{
  int i;
  double norm=fabs(x[0]);
  for (i=0;i<N;i++) if(norm<fabs(x[i])) norm=fabs(x[i]);
  return norm;
}
void main (void)
{
  int i;
  double x0[N],x[N],d[N],a[N],b[N],c[N],tol=1e-10,*px0=x0,*px=x,*paux=NULL;
  for (i=0;i<N;i++) x0[i]=0;
  do
  {
    F(px0,d);
    J(px0,a,b,c);
    Thomas(d,a,b,c);
    for (i=0;i<N;i++) *(px+i)=*(px0+i)-d[i];
    for (i=0;i<N;i++) printf("%.12f  ",*(px+i));
    printf("\n");
```

```
    paux=px0;
    px0=px;
    px=paux;
  }while(norma_max(d)>tol);
  for (i=0;i<N;i++) printf("%.12f  ",*(px+i));
}
```

6.4 Passagem de parâmetros na linha de comando

A função `main` pode receber strings que são passadas na linha de comando. Usamos a linha

```
main(int argc,char *argv[]),
```

onde `argc` é um inteiro que indica a quantidade de argumentos passados e `argv` é um vetor contendo todas as strings.

Exemplo 6.4.1. Vamos implementar um código que entra com um nome e um sobrenome na linha de comando e imprime-os.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
printf("número de entradas=%d\n",argc);
printf("executável=%s\n",argv[0]);
printf("nome=%s\n",argv[1]);
printf("sobrenome=%s\n",argv[2]);
}
```

Esse programa é executado com uma linha de comando da forma

```
./a.out Artur Avila
```

e o resultado é

```
número de entradas=3
executável=./a.out
nome=Artur
sobrenome=Avila
```

Observações sobre o exemplo 6.4.1:

1. O números de entradas é 3 e não 2 (`argc` tem valor 3).

2. `argv[0]` armazena o executável.

3. O valor `argc` não entra explicitamente na linha de comando, mas é calculado na execução.

Algumas vezes entramos com strings contendo números e desejamos convertê-los para inteiros. Nesse caso, usamos a função `int atoi(char *str)` da biblioteca `<stdlib>`.

Exemplo 6.4.2. Implemente um programa que entre com dois números inteiros e retorna a soma.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
int i,soma;
for (i=1;i<argc;i++) soma+=atoi(argv[i]);
printf("soma=%d\n",soma);
}
```

Se desejamos entrar com números racionais, trocamos `atoi` por `atof`. Observe a versão do código do exemplo 6.4.2 usando números racionais:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
int i;
double soma;
for (i=1;i<argc;i++) soma+=atof(argv[i]);
printf("soma=%f\n",soma);
}
```

Exemplo 6.4.3. Implemente uma rotina para calcular a raíz de uma função pelo método de Newton passando o chute inicial e a tolerância na linha de comando e retornando a solução e o número de iterações necessárias (refazendo o exemplo 6.2.2, mas agora passando informações na linha de comando).

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

```
#include <stdlib.h>

double f(double x)
{
  return cos(x)-x;
}
double df(double x)
{
  return -sin(x)-1.;
}

void Newton(double x0, double tol, double *xn, int *k)
{
  double dif;
  *k=0;
  do
  {
  (*k)++;
  *xn=x0-f(x0)/df(x0);
  dif=fabs(*xn-x0);
  x0=*xn;
  }while (dif>tol);
}


int main(int argc, char **argv)
{
  double x0,xn,tol;
  x0=atof(argv[1]);
  tol=atof(argv[2]);
  int k;
  Newton(x0,tol,&xn,&k);
  printf("Sol=%f, iterações=%d\n",xn,k);
}
```

Experimente alguns chutes iniciais e tolerâncias tais como

```
./a.out 1 1e-10
```

Nesse último programa usamos `**argv` em vez de `*argv[]`. De fato, as notações são equivalentes. Como discutido anteriormente, `**` representa ponteiro de ponteiro. Especificamente aqui, temos um ponteiro que aponta para a lista de argumentos e cada um dos argumentos são strings representadas por seus ponteiros.

6.5 Recursão

Quando uma função chama ela mesma.

Exemplo 6.5.1. Escreva uma função que imprime os n primeiros números em ordem crescente.

Primeiro vamos resolver o problema usando um laço de repetição

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

int main(int argc, char **argv)
{
  int i,N;
  N=atoi(argv[1]);
  for(i=0;i<=N;i++) printf("%d\n",i);
}
```

Agora, vamos trocar o `for` por uma função recursiva.

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

void imprime(int i)
{

 if(i<0) return;
 imprime(i-1);
 printf("%d\n",i);
}

int main(int argc, char **argv)
{
  int i,N;
  N=atoi(argv[1]);
  imprime(N);
}
```

Observe no último programa que a função imprime chama ela mesma. Observe que a primeira linha dentro da função recursiva é o critério de parada. Isso é fundamental, pois caso contrário, a função seria executada até acabar a memória do computador. Vamos tentar entender o que faz a função `imprime(3)`:

- Quando chamamos `imprime(3)`, temos `i=3`, logo a função não para no primeiro `if`.
- Então a função chama `imprime(2)` antes de imprimir.
- O `imprime(2)` se repete os passos do `imprime(3)` e chama o `imprime(1)`, também antes de imprimir.
- Depois `imprime(0)` e `imprime(-1)`.
- `imprime(-1)` termina na primeira linha, pois `-1<0`.
- Ainda dentro do `imprime(0)`, a função imprime `printf("%d\n",0);`.
- Uma vez que fecha o `imprime(0)`, vai para o término da função `imprime(1)` e escreve `printf("%d\n",1);`.
- Sucessivamente, o as funções `imprime(2)` e `imprime(3)` terminam com `printf("%d\n",2);` e `printf("%d\n",3);`

6.6 Passagem de ponteiro para função

É possível enviar para uma funçao o endereço de outra função. O próprio nome da função é o endereço da função.

Exemplo 6.6.1. Implemente uma função que eleva outra função ao quadrado.

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

double f(double x)
{
  return exp(-x);
}
double g(double x)
{
  return sin(x);
}


double Quad(double (*fun)(double),double x)
{
 return ((*fun)(x))*((*fun)(x));
```

```
}


void main (void)
{
double x=2;
printf("f(%f)=%f\n",x,f(x));
printf("g(%f)=%f\n",x,g(x));
printf("(f(%f))^2=%f\n",x,Quad(f,x));
printf("(g(%f))^2=%f\n",x,Quad(g,x));
}
```

Exemplo 6.6.2. Faça um código para calcular

$$\int_0^1 f(t)dt,$$

onde

$$f(t) = \int_0^t \text{sen}\,(t-\tau)e^{-\tau^2}d\tau.$$

Vamos começar construindo uma rotina que imprime os pesos e abscissas de uma quadratura.

```
#include <stdio.h>
#include <math.h>
#define DEBUG
void boole(double a,double b,int N,double *nodes, double *weights)
{
  int i;
  for (i=0;i<N;i++)
  {
    nodes[i]=a+ (b-a)*i/(N-1.0);
  }
  weights[0]=(b-a)*28/90.0/(N-1.0);
  for (i=1;i<N;i+=4)
  {
    weights[i]=4.0*16.0*(b-a)/45.0/(N-1.0);
    weights[i+1]=4.0*12.0*(b-a)/90.0/(N-1.0);
    weights[i+2]=4.0*16.0*(b-a)/45.0/(N-1.0);
    weights[i+3]=4.0*7.0*(b-a)/45.0/(N-1.0);
  }
  weights[N-1]=28*(b-a)/90.0/(N-1.0);
```

```
}


void main (void)
{
  int i,N=13;
  if (N%4!=1)
  {
    printf("o número de pontos da quadratura deve ser da forma 4N+1\n");
    return;
  }
  double a=0,b=1,x[N],w[N];
  boole(a,b,N,x,w);
#ifdef DEBUG
  for (i=0;i<N;i++)
  {
  printf("x[%d]=%f,w[%d]=%f\n",i,x[i],i,w[i]);
  }
#endif
}
```

Agora vamos integrar algumas funções usando alocamento dinâmico de memória.

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

void boole(double a,double b,int N,double *nodes, double *weights)
{
  int i;
  for (i=0;i<N;i++)
  {
    nodes[i]=a+(b-a)*i/(N-1.0);
  }
  weights[0]=(b-a)*28/90.0/(N-1.0);
  for (i=1;i<N;i+=4)
  {
    weights[i]=4.0*16.0*(b-a)/45.0/(N-1.0);
    weights[i+1]=4.0*12.0*(b-a)/90.0/(N-1.0);
    weights[i+2]=4.0*16.0*(b-a)/45.0/(N-1.0);
    weights[i+3]=4.0*7.0*(b-a)/45.0/(N-1.0);
  }
```

```
  weights[N-1]=28*(b-a)/90.0/(N-1.0);
}


void main (void)
{
  int i,N=13;
  if ((N%4)!=1)
  {
    printf("o número de pontos da quadratura deve ser da forma 4N+1\n");
    return;
  }
  double a=0,b=1,*x,*w;
  x=malloc(N*sizeof(double));
  w=malloc(N*sizeof(double));
  if ((x==NULL)||(w==NULL))
  {
    printf("erro ao alocar memória\n");
    return;
  }
  boole(a,b,N,x,w);
//integral de 1 no intervalo [0,1];
  double I=0;
  for (i=0;i<N;i++) I+=w[i];
  printf("A integral de 1 é %f\n",I);

//integral de x no intervalo [0,1];
  I=0;
  for (i=0;i<N;i++) I+=w[i]*x[i];
  printf("A integral de x é %f\n",I);

//integral de x^2 no intervalo [0,1];
  I=0;
  for (i=0;i<N;i++) I+=w[i]*x[i]*x[i];
  printf("A integral de x^2 é %f\n",I);

//integral de x^3 no intervalo [0,1];
  I=0;
  for (i=0;i<N;i++) I+=w[i]*x[i]*x[i]*x[i];
  printf("A integral de x^3 é %f\n",I);
```

```
}
```

Ocorre que a integral está com um número fixo de pontos. Então, vamos implementar rotinas que refinam até a convergência.

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

void boole(double a,double b,int N,double *nodes, double *weights)
{
  int i;
  for (i=0;i<N;i++)
  {
    nodes[i]=a+(b-a)*i/(N-1.0);
  }
  weights[0]=(b-a)*28/90.0/(N-1.0);
  for (i=1;i<N;i+=4)
  {
    weights[i]=4.0*16.0*(b-a)/45.0/(N-1.0);
    weights[i+1]=4.0*12.0*(b-a)/90.0/(N-1.0);
    weights[i+2]=4.0*16.0*(b-a)/45.0/(N-1.0);
    weights[i+3]=4.0*7.0*(b-a)/45.0/(N-1.0);
  }
  weights[N-1]=28*(b-a)/90.0/(N-1.0);
}
double f(double x)
{
  return exp(-x);
}

void main (void)
{
  int i,N=5;
  if ((N%4)!=1)
  {
    printf("o número de pontos da quadratura deve ser da forma 4N+1\n");
    return;
  }
  double a=0,b=1,*x,*w,tol=1e-10;
  x=malloc(N*sizeof(double));
  w=malloc(N*sizeof(double));
```

```
  if ((x==NULL)||(w==NULL))
  {
    printf("erro ao alocar memória\n");
    return;
  }
  boole(a,b,N,x,w);
//integral de 1 no intervalo [0,1];
  int cont=0;
  double I1=0,I2=0,erro;
  for (i=0;i<N;i++) I1+=w[i]*f(x[i]);
  do
  {
  cont++;
  N=2*N-1;
  x=realloc(x,N*sizeof(double));
  w=realloc(w,N*sizeof(double));
  boole(a,b,N,x,w);
  I2=0;
  for (i=0;i<N;i++) I2+=w[i]*f(x[i]);
  erro=fabs(I2-I1);
  printf("erro=%.12f\n",erro);
  I1=I2;
  }while (erro>tol);
  printf("A integral de f(x) é %f e foi calculado com %d iterações\n",I2,cont);
}
```

Agora, vamos resolver o problema proposto, começando pela implementação de $f(t)$:

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

void boole(double a,double b,int N,double *nodes, double *weights)
{
  int i;
  for (i=0;i<N;i++)
  {
    nodes[i]=a+(b-a)*i/(N-1.0);
  }
  weights[0]=(b-a)*28/90.0/(N-1.0);
  for (i=1;i<N;i+=4)
```

```
  {
    weights[i]=4.0*16.0*(b-a)/45.0/(N-1.0);
    weights[i+1]=4.0*12.0*(b-a)/90.0/(N-1.0);
    weights[i+2]=4.0*16.0*(b-a)/45.0/(N-1.0);
    weights[i+3]=4.0*7.0*(b-a)/45.0/(N-1.0);
  }
  weights[N-1]=28*(b-a)/90.0/(N-1.0);
}
double integrando(double tau, double *t)
{
  return sin(*t-tau)*exp(-tau*tau);
}

double f(double t)
{
  int i,N=5;
  if ((N%4)!=1)
  {
    printf("o número de pontos da quadratura deve ser da forma 4N+1\n");
    return;
  }
  double a=0,b=1,*x,*w,tol=1e-10;
  x=malloc(N*sizeof(double));
  w=malloc(N*sizeof(double));
  if ((x==NULL)||(w==NULL))
  {
    printf("erro ao alocar memória\n");
    return;
  }
  boole(a,b,N,x,w);
  int cont=0;
  double I1=0,I2=0,erro;
  for (i=0;i<N;i++) I1+=w[i]*integrando(x[i],&t);
  do
  {
  cont++;
  N=2*N-1;
  x=realloc(x,N*sizeof(double));
  w=realloc(w,N*sizeof(double));
  boole(a,b,N,x,w);
```

```
  I2=0;
  for (i=0;i<N;i++) I2+=w[i]*integrando(x[i],&t);
  erro=fabs(I2-I1);
  I1=I2;
  }while (erro>tol);
  return I2;
}

void main (void)
{
  printf("f(1)=%f\n",f(1));
}
```

Agora, vamos fazer a última integral:

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

void boole(double a,double b,int N,double *nodes, double *weights)
{
  int i;
  for (i=0;i<N;i++)
  {
    nodes[i]=a+(b-a)*i/(N-1.0);
  }
  weights[0]=(b-a)*28/90.0/(N-1.0);
  for (i=1;i<N;i+=4)
  {
    weights[i]=4.0*16.0*(b-a)/45.0/(N-1.0);
    weights[i+1]=4.0*12.0*(b-a)/90.0/(N-1.0);
    weights[i+2]=4.0*16.0*(b-a)/45.0/(N-1.0);
    weights[i+3]=4.0*7.0*(b-a)/45.0/(N-1.0);
  }
  weights[N-1]=28*(b-a)/90.0/(N-1.0);
}
double integrando(double tau, double *t)
{
  return sin(*t-tau)*exp(-tau*tau);
}

double f(double t)
```

```
{
  int i,N=5;
  if ((N%4)!=1)
  {
    printf("o número de pontos da quadratura deve ser da forma 4N+1\n");
    return;
  }
  double a=0,b=t,*x,*w,tol=1e-10;
  x=malloc(N*sizeof(double));
  w=malloc(N*sizeof(double));
  if ((x==NULL)||(w==NULL))
  {
    printf("erro ao alocar memória\n");
    return;
  }
  boole(a,b,N,x,w);
  int cont=0;
  double I1=0,I2=0,erro;
  for (i=0;i<N;i++) I1+=w[i]*integrando(x[i],&t);
  do
  {
  cont++;
  N=2*N-1;
  x=realloc(x,N*sizeof(double));
  w=realloc(w,N*sizeof(double));
  boole(a,b,N,x,w);
  I2=0;
  for (i=0;i<N;i++) I2+=w[i]*integrando(x[i],&t);
  erro=fabs(I2-I1);
  I1=I2;
  }while (erro>tol);
  free(x);
  free(w);
  return I2;
}

void main (void)
{
   int i,N=5;
  if ((N%4)!=1)
```

```
  {
    printf("o número de pontos da quadratura deve ser da forma 4N+1\n");
    return;
  }
  double a=0,b=1,*x,*w,tol=1e-10;
  x=malloc(N*sizeof(double));
  w=malloc(N*sizeof(double));
  if ((x==NULL)||(w==NULL))
  {
    printf("erro ao alocar memória\n");
    return;
  }
  boole(a,b,N,x,w);
  int cont=0;
  double I1=0,I2=0,erro;
  for (i=0;i<N;i++) I1+=w[i]*f(x[i]);
  do
  {
  cont++;
  N=2*N-1;
  x=realloc(x,N*sizeof(double));
  w=realloc(w,N*sizeof(double));
  boole(a,b,N,x,w);
  I2=0;
  for (i=0;i<N;i++) I2+=w[i]*f(x[i]);
  erro=fabs(I2-I1);
  I1=I2;
  }while (erro>tol);
  free(x);
  free(w);
  printf("O valor da integral é %f\n",I2);
}
```

Observe que existem várias linhas do código repetida. Isso pode ser melhorado quando usamos ponteiro para função. Vamos calcular $f(1)$ com essa ferramenta:

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

double Boole(double (*f)(double, double),double a,double b, int N,double *param)
{
```

```
  int i;
  double h=(b-a)/(4*N),Int=0,t=param[0];

  Int=14./45.*(*f)(a,t);
  for(i=0;i<N;i++)
  {
    Int+=64./45.*(*f)(a+h,t);
    Int+=24./45.*(*f)(a+2*h,t);
    Int+=64./45.*(*f)(a+3*h,t);
    Int+=28./45.*(*f)(a+4*h,t);
    a+=4*h;
  }
  Int-=14./45.*(*f)(a,t);
return h*Int;
}
double integrando(double tau, double t)
{
  return sin(t-tau)*exp(-tau*tau);
}
void main(void)
{
double param[1]={1};
printf("f(1)=%f\n",Boole(integrando,0,1,5,param) );
}
```

Agora, vamos resolver o problema proposto:

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

double Boole(double (*f)(double, double),double a,double b, int N,double *param)
{
  int i;
  double h=(b-a)/(4*N),Int=0,t=param[0];

  Int=14./45.*(*f)(a,t);
  for(i=0;i<N;i++)
  {
    Int+=64./45.*(*f)(a+h,t);
    Int+=24./45.*(*f)(a+2*h,t);
    Int+=64./45.*(*f)(a+3*h,t);
```

```
    Int+=28./45.*(*f)(a+4*h,t);
    a+=4*h;
  }
  Int-=14./45.*(*f)(a,t);
return h*Int;
}
double integrando(double tau, double t)
{
  return sin(t-tau)*exp(-tau*tau);
}
double f(double t,double tau)
{
  int N=5;
  double I1=0,I2=0,erro=0,a=0,b=t,tol=1e-10,param[1]={t};
  I1=Boole(integrando,a,b,N,param);
  int cont=0;
  do
  {
  cont++;
  N=2*N-1;
  I2=Boole(integrando,a,b,N,param);
  erro=fabs(I2-I1);
  I1=I2;
  }while (erro>tol);
  return I2;
}

void main(void)
{
  int N=5;
  double I1=0,I2=0,erro=0,a=0,b=1,tol=1e-10,param[1]={0};
  I1=Boole(f,a,b,N,param);
  int cont=0;
  do
  {
  cont++;
  N=2*N-1;
  I2=Boole(f,a,b,N,param);
  erro=fabs(I2-I1);
  I1=I2;
```

```
  }while (erro>tol);
  printf("O valor da integral é %f\n",I2);
}
```

6.7 Exercícios

E 6.7.1. Refaça os exercícios 4.4.6, 4.4.7 e 4.4.8 usando troca de ponteiros na parte iterativa.

E 6.7.2. Implemente uma função que entra um vetor com N posições e devolve o máximo e o mínimo valor.

E 6.7.3. Use a estrutura trabalhada no exemplo 6.2.2 para refazer os exercícios 3.4.4 e 3.4.5

E 6.7.4. A eliminação gaussiana (ou escalonamento), é um método para resolver sistemas lineares que consiste em manipular o sistema através de determinadas operações elementares, transformando-o em um sistema triangular. A solução pode ser obtida via substituição regressiva.

1. multiplicação de um linha por uma constante não nula.
2. substituição de uma linha por ela mesma somada a um múltiplo de outra linha.
3. permutação de duas linhas.

a) Implemente um programa que resolve sistema lineares por eliminação gaussiana.

b) Implemente um programa que resolve sistema lineares por eliminação gaussiana com pivotamento parcial. A eliminação gaussiana com pivotamento parcial consiste em fazer uma permutação de linhas para escolher o maior pivô (em módulo) a cada passo.

Para ajudar o programador, segue um sistema linear 3x3 retirado do livro colaborativo www.ufrgs.br/numerico

$$\begin{aligned} x + y + z &= 1 \\ 2x + y - z &= 0 \\ 2x + 2y + z &= 1 \end{aligned}$$

A solução via eliminação gaussiana com pivotamento parcial segue os seguintes passos:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & 0 \\ 2 & 2 & 1 & 1 \end{bmatrix} \sim \begin{bmatrix} 2 & 1 & -1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 1 & 1 \end{bmatrix}$$

$$\sim \begin{bmatrix} 2 & 1 & -1 & 0 \\ 0 & 1/2 & 3/2 & 1 \\ 0 & 1 & 2 & 1 \end{bmatrix}$$

$$\sim \begin{bmatrix} 2 & 1 & -1 & 0 \\ 0 & 1 & 2 & 1 \\ 0 & 1/2 & 3/2 & 1 \end{bmatrix}$$

$$\sim \begin{bmatrix} 2 & 1 & -1 & 0 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & 1/2 & 1/2 \end{bmatrix}$$

Por substituição regressiva, temos $1/2z = 1/2$, ou seja, $z = 1$. Substituímos na segunda equação e temos $y + 2z = 1$, ou seja, $y = -1$ e, finalmente $2x + y - z = 0$, resultando em $x = 1$.

E 6.7.5. As aproximações de segunda ordem para a derivada primeira são obtidas pelas seguintes fórmulas

a) Progressiva: $f'(x_0) \approx \frac{1}{2h}\left[-3f(x_0) + 4f(x_0 + h) - f(x_0 + 2h)\right]$

b) Central: $f'(x_0) \approx \frac{1}{2h}\left[f(x_0 + h) - f(x_0 - h)\right]$

c) Regressiva: $f'(x_0) \approx \frac{1}{2h}\left[f(x_0 - 2h) - 4f(x_0 - h) + 3f(x_0)\right]$

Refaça o exemplo 6.3.2 usando diferenças de segunda ordem para todas as derivadas e o algorítmo de eliminação Gaussiana para resolver o sistema linear.

E 6.7.6. Encontre uma aproximação numérica para o seguinte problema não-linear de três equações e três incógnitas:

$$\begin{aligned} 2x_1 - x_2 &= \cos(x_1) \\ -x_1 + 2x_2 - x_3 &= \cos(x_2) \\ -x_2 + x_3 &= \cos(x_3) \end{aligned}$$

Partindo das seguintes aproximações iniciais:

a) $x^{(0)} = [1,\ 1,\ 1]^T$

b) $x^{(0)} = [-0{,}5,\ -2,\ -3]^T$

c) $x^{(0)} = [-2,\ -3,\ -4]^T$

d) $x^{(0)} = [0,\ 0,\ 0]^T$

E 6.7.7. Considere o seguinte sistema de equações não-lineares:

$$\begin{aligned} x_1 - x_2 &= 0 \\ -x_{j-1} + 5(x_j + x_j^3) - x_{j+1} &= 10\exp(-j/3),\ \ 2 \leq j \leq 10 \\ x_{11} &= 1 \end{aligned} \tag{6.1}$$

Implemente um código para encontrar a única solução deste problema pelo método de Newton.

E 6.7.8. Resolva numericamente o problema de valor de contorno para a equação de calor no estado estacionário com um termo não linear de radiação

$$\begin{cases} -u_{xx} = 100 - \frac{u^4}{10000},\ \ 0 < x < 2. \\ u(0) = 0 \\ u(2) = 10 \end{cases}$$

usando o método de diferenças finitas.

E 6.7.9. Resolva numericamente o problema de valor de contorno para a equação de calor no estado estacionário com um termo não linear de radiação e um termo de convecção:

$$\begin{cases} -u_{xx} + 3u_x = 100 - \frac{u^4}{10000},\ \ 0 < x < 2. \\ u'(0) = 0 \\ u(2) = 10 \end{cases}$$

usando o método de diferenças finitas.

E 6.7.10. Resolva numericamente o problema de valor de contorno

$$\begin{cases} -u'' + 2u' = e^{-x} - \frac{u^2}{100}, \ \ 1 < x < 4. \\ u'(1) + u(1) = 2 \\ u'(4) = -1 \end{cases}$$

usando o método de diferenças finitas.

E 6.7.11. Implemente o cálculo da fatorial de n de várias formas:

1. Entre com n no `scanf` e use um `for`.
2. Entre com n na linha de comando
3. Construa uma função recursiva $n! = n(n-1)!$ (não use `for`).

E 6.7.12. Repita o exercício anterior para x^n, $n \geq 0$.

E 6.7.13. O método de Runge-Kutta quarta ordem clássico para resolver o PVI

$$\begin{aligned} u' &= f(t,u) \\ u(0) &= u_0 \end{aligned}$$

é dado pela recursão

$$\begin{aligned} k_1 &= hf\left(t^{(n)},u^{(n)}\right) \\ k_2 &= hf\left(t^{(n)} + h/2,u^{(n)} + k_1/2\right) \\ k_3 &= hf\left(t^{(n)} + h/2,u^{(n)} + k_2/2\right) \\ k_4 &= hf\left(t^{(n)} + h,u^{(n)} + k_3\right) \\ u^{(n+1)} &= u^{(n)} + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \end{aligned}$$

Implemente um código em C com a algorítmo acima para resolva o sistema para o oscilador não linear de Van der Pol dado por

$$u''(t) - \alpha(A - u(t)^2)u'(t) + w_0^2 u(t) = 0 \tag{6.2}$$

onde A, α e w_0 são constantes positivas. Encontre a frequência e a amplitude de oscilações quando $w_0 = 1$, $\alpha = .1$ e $A = 10$ (teste diversas condições iniciais).

Observação:

- Embora o oscilador seja de segunda ordem, você pode fazer uma mudança de variável e escrevê-lo com um sistema de primeira ordem.
- Entre com os parâmetros tempo final e o número de intervalos na linha de comando.

E 6.7.14. Resolva o modelo simplificado de FitzHugh-Nagumo para o potencial elétrico sobre a membrana de um neurônio:

$$\begin{aligned} \frac{dV}{dt} &= V - V^3/3 - W + I \\ \frac{dW}{dt} &= 0{,}08(V + 0{,}7 - 0{,}8W) \end{aligned}$$

onde I é a corrente de excitação.

- Encontre o único estado estacionário (V_0,W_0) com $I = 0$.
- Resolva numericamente o sistema com condições iniciais dadas por (V_0,W_0) e $I = e^{-t/200}$.

Capítulo 7

Strings

7.1 Strings

Strings são vetores de `char`.

Exemplo 7.1.1. Alguns testes com strings.

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


main (void)
{
  char nome[7]="Artur";
  char sobrenome[10]={'A','v','i','l','a'};

  printf("O nome do matemático é %s %s\n",nome,sobrenome);
  printf("\n");
  int i;
  for(i=0;i<5;i++) printf("%c\n",nome[i]);
  printf("\n");
  for(i=0;i<5;i++) printf("%c\n",sobrenome[i]);
}
```

Observe que usamos `%s` para imprimir um string e `%c` para imprimir um caractere.

Exemplo 7.1.2. Faça um programa que lê nome e o sobrenome e imprime-os.

```
#include <stdio.h>
```

```
#include <math.h>


main (void)
{
  char nome[20],sobrenome[20];

  printf("Digite seu nome\n");
  scanf("%s",nome);
  printf("Digite seu sobrenome\n");
  scanf("%s",sobrenome);
  printf("\n%s",nome);
  printf(" %s\n",sobrenome);
}
```

Existem várias função para manipulação de strings, tais como

- `strlen(string)` conta o número de caracteres e retorna um inteiro positivo.
- `strcpy(string1,string2)` copia string2 para string1.
- `strcat(string1,string2)` coloca string2 imediatamente após string1.
- `strcmp(string1,string2)` compara as strings alfabeticamente e retorna verdadeiro ou falso (1 ou 0).

Essas funções não são nativas em C e precisam da biblioteca `string.h` para funcionar.

Exemplo 7.1.3. Faça um programa que lê nome e o sobrenome e imprime o número de letras de cada um.

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

main (void)
{
  char nome[20],sobrenome[20];

  printf("Digite seu nome\n");
  scanf("%s",nome);
  printf("Digite seu sobrenome\n");
```

```
  scanf("%s",sobrenome);
  printf("\n%s",nome);
  printf(" %s\n",sobrenome);
  printf("Seu nome tem %d letras\n",(int) strlen(nome));
  printf("Seu sobrenome tem %d letras\n",(int) strlen(sobrenome));
}
```

Exemplo 7.1.4. Escreva um código que lê uma string e inverte os caracteres.

```
#include <stdio.h>
#include <string.h>

char *strinv(char *s)
{
  int i,j,k=strlen(s);
  char aux;
  for (i=0,j=k-1;i<j;i++,j--)
    {
    aux=s[i];
    s[i]=s[j];
    s[j]=aux;
    }
  return s;
}

int main(void)
{
  char nome[6]="Artur";
  strinv(nome);
  printf("O nome invertido é %s\n",nome);
}
```

Exemplo 7.1.5. Escreva um programa que lê a idade de um gato e escreve-o por extenso. Suponha que o gato viva no máximo 39 anos.

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

int unidades(char idade[],char resposta[])
{
```

```
  if (idade[0]=='0') {strcpy(resposta,"zero"); return 1;}
  if (idade[0]=='1') {strcpy(resposta,"um"); return 1;}
  if (idade[0]=='2') {strcpy(resposta,"dois"); return 1;}
  if (idade[0]=='3') {strcpy(resposta,"três"); return 1;}
  if (idade[0]=='4') {strcpy(resposta,"quatro"); return 1;}
  if (idade[0]=='5') {strcpy(resposta,"cinco"); return 1;}
  if (idade[0]=='6') {strcpy(resposta,"seis"); return 1;}
  if (idade[0]=='7') {strcpy(resposta,"sete"); return 1;}
  if (idade[0]=='8') {strcpy(resposta,"oito"); return 1;}
  if (idade[0]=='9') {strcpy(resposta,"nove"); return 1;}
  return 0;
}
int dezenas(char idade[],char resposta[])
{
  if (strcmp(idade,"10")==0) {strcpy(resposta,"dez"); return 1;}
  if (strcmp(idade,"11")==0) {strcpy(resposta,"onze"); return 1;}
  if (strcmp(idade,"12")==0) {strcpy(resposta,"doze"); return 1;}
  if (strcmp(idade,"13")==0) {strcpy(resposta,"treze"); return 1;}
  if (strcmp(idade,"14")==0) {strcpy(resposta,"quatorze"); return 1;}
  if (strcmp(idade,"15")==0) {strcpy(resposta,"quize"); return 1;}
  if (strcmp(idade,"16")==0) {strcpy(resposta,"dezesseis"); return 1;}
  if (strcmp(idade,"17")==0) {strcpy(resposta,"dezessete"); return 1;}
  if (strcmp(idade,"18")==0) {strcpy(resposta,"dezoito"); return 1;}
  if (strcmp(idade,"19")==0) {strcpy(resposta,"dezenove"); return 1;}
  if (idade[0]=='0') return 0;
  if (idade[0]=='2')
  {
    if (strcmp(idade,"20")==0) {strcpy(resposta,"vinte"); return 1;}
    char unidade[2]={idade[1]},resp[10];
    strcpy(resposta,"vinte e ");
    if (unidades(unidade,resp)==1)
    {
    strcat(resposta,resp);
    return 1;
    }
  }
  if (idade[0]=='3')
  {
    if (strcmp(idade,"30")==0) {strcpy(resposta,"trinta"); return 1;}
    char unidade[2]={idade[1]},resp[10];
```

```
    strcpy(resposta,"trinta e ");
    if (unidades(unidade,resp)==1)
    {
    strcat(resposta,resp);
    return 1;
    }
  }




  return 0;
}


int main (void)
{
  char idade[4]="",resposta[30]="";

  printf("Digite a idade do gato\n");
  scanf("%s",idade);
  if (strlen(idade)==1)
  {
    if (unidades(idade,resposta)==1)
    {
      if ((idade[0]=='1')||(idade[0]=='0')) {printf("A idade é %s ano\n",resposta); r
      else {printf("A idade é %s anos\n",resposta); return 0;}
    }
    else
    {
    printf("Idade inválida\n");
    return 0;
    }
  }
  if (strlen(idade)==2)
  {
    if (dezenas(idade,resposta)==1)
    {
    printf("A idade é %s anos\n",resposta);
    return 0;
```

```
    }
    else
    {
    printf("Idade inválida\n");
    return 0;
    }
  }
printf("Idade inválida\n");
}
```

7.2 Exercícios

E 7.2.1. Escreva um programa que lê uma string `s` e conta quantas vezes aparece um caractere `ch`: `int strcount(char *s, char ch)`.

E 7.2.2. Escreva um programa que lê a idade de uma pessoa e escreve-o por extenso. Suponha que uma pessoa viva no máximo 129 anos.

E 7.2.3. Escreva um programa que lê dois números correspondente ao valor de um cheque (reais e centavos) e escreve-o por extenso. Suponha que o valor máximo de um cheque seja R9999,99.

Capítulo 8

Arquivos

8.1 Escrevendo em arquivo o conteúdo da tela

A forma mais simples de colocar o conteúdo da tela em arquivo é usar o comando >> na linha de comando.

Exemplo 8.1.1. Escreva um programa que escreva Avila na tela e execute salvando o conteúdo em arquivo.

```
#include <stdio.h>

int main(void)
{
  char nome[6]="Avila";
  printf("%s\n",nome);
}
```

Execute com a linha

```
./a.out >> arquivo.txt
```

Agora, abra o arquivo `arquivo.txt` e olhe o conteúdo.

8.2 Abertura e Fechamento de arquivos

Até agora, os programas terminavam de ser executados e os resultados não eram salvos definitivamente em arquivo. Os arquivos são interessantes para fornecer dados de entrada para o programa e gravar resultados parciais e finais.

As operações mais comuns em arquivos são: abertura, leitura, escrita e fechamento. Abertura consiste em associar uma variável lógica ao nome do arquivo,

leitura consiste em colocar em memória o conteúdo do arquivo, escrita é o caminho inverso da leitura e fechamento é a liberação da variável que associa o arquivo.

Em `C` não existe um tipo de variável nativa para arquivos. O tipo de variável usada é `FILE` da biblioteca `<stdio.h>`. Na prática, declaramos um ponteiro para o tipo de variável `FILE` tal como fazemos com outras variáveis:

```
int x;
double *y;
char ch;
FILE *fp;
```

O comando para a abertura de um arquivo é o `fopen`, que entra com a string contendo o nome do arquivo, outra string contendo os modos de abertura e retorna um ponteiro para `FILE`:

```
FILE *fp;

fp=fopen(string com nome, string com modo de abertura);
```

Os modos de abertura são os seguintes:

- "r"(read): abertura de arquivo para leitura;
- "w"(write): abertura de arquivo para escrita;
- "a"(append): abertura de arquivo para acrescentar;

Além desses três modos, podemos usar um modo combinado:

- "r+"(read): abertura de arquivo para leitura e escrita. Se o arquivo não existir, ele não é criado e, se ele já existir, os dados sobrescreverão os antigos;
- "w+"(write): abertura de arquivo para leitura e escrita. Se o arquivo não existir, ele é criado e, se ele já existir, será apagado e recriado;
- "a+"(append): abertura de arquivo para leitura e escrita. Se o arquivo não existir, ele é criado e, se ele já existir, os dados serão armazenados a partir do fim do arquivo.

Além disso, os arquivos podem ser abertos como binário ou texto. Os arquivos texto são aqueles que possuem símbolos interpretados por nós. Os arquivos binários em códigos não necessariamente legíveis e estão preparados para ser interpretados por um computador. Na prática, se você está interessado em salvar dados de simulação, vai preferir arquivo binário. Usamos a letra `t` para configurar

o arquivo como texto e a letra `b` para binário. Assim, `rb` é o modo de abertura para arquivo binário que permite leitura. Analogamente, `wb`, `ab`, `r+b`, `w+b` e `a+b`.

Depois que você não precisa mais do arquivo aberto, você deverá fechá-lo. Para isso, você usará a função `fclose(arquivo)`, que entrará um ponteiro para o arquivo e retornará um inteiro. A função `fcloseall(arquivo)` fecha todos arquivos abertos.

Exemplo 8.2.1. Implemente um código para abrir um arquivo binário no modo `rb`.

```
#include <stdio.h>

int main(void)
{
  char nome[9]="nome.dat";
  FILE *arq;

  arq=fopen(nome,"r");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    fclose(arq);
    }
}
```

Observe que, se você não possui um arquivo com o nome `nome.dat` na mesma pasta onde está seu código em C, esse programa retorna `Não conseguiu abrir o arquivo`.

Exemplo 8.2.2. Implemente um código para abrir um arquivo binário no modo `w+b`. Adicione o nome do arquivo em linha de comando.

```
#include <stdio.h>

int main(int argc,char **argv)
{
  char *nome;
  nome=argv[1];
  FILE *arq;

  arq=fopen(nome,"w+b");
```

```
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    fclose(arq);
    }
}
```

8.3 Leitura e escrita de arquivos texto

Vamos começar com o comando `fputc`, que entra um caractere e um ponteiro para arquivo e retorna um inteiro. Esse comando escreve o caractere no arquivo.

Exemplo 8.3.1. Escreva o sobrenome Avila em um arquivo de texto.

```
#include <stdio.h>

int main(void)
{
  FILE *arq;

  arq=fopen("nome.txt","wt");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    fputc('A',arq);
    fputc('v',arq);
    fputc('i',arq);
    fputc('l',arq);
    fputc('a',arq);

    fclose(arq);
    }
}
```

Vá até a pasta onde está nome.txt a abre a arquivo com o gedit. Para ler um caractere de um arquivo, usamos comando `fgetc` que entra um ponteiro para arquivo e retorna um inteiro relacionado ao caractere pelo tabela ASCII. O retorno

EOF (End-of-File), indica que não havia caractere no arquivo para ser lido. A constante EOF vale -1.

Exemplo 8.3.2. Escreva um programa para ler o arquivo gerado pela rotina anterior e imprime o resultado na tela.

```
#include <stdio.h>

int main(void)
{
  FILE *arq;
  int ch;

  arq=fopen("nome.txt","rt");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    while ((ch=fgetc(arq))!=EOF)
      {
      printf("Caractere na tabela ASCII=%d\n",ch);
      putchar(ch);
      printf("\n");
      }
    fclose(arq);
    }
}
```

8.4 Leitura e escrita de arquivos binários

No modo binário, não existe noção de linha no arquivo, mas os arquivos são lidos e escritos em blocos. Nesse modo é possível ler e escrever um vetor inteiro de uma só vez. Essas operações para acessar um arquivo binário são chamadas de acesso direto.

Para ler e escrever um arquivo binário, usamos as funções `fread` e `fwrite`, respectivamente. Eles têm a seguinte sintaxe:

```
int fwrite(const void *ptr, int size, int n, FILE *arq)
int fread(const void *ptr, int size, int n, FILE *arq)
```

onde

- `ptr` é um ponteiro de qualquer tipo apontando para o lugar da memória onde será lido ou escrito.
- `size` é o tamanho em bytes de cada um dos elementos que queremos ler ou escrever.
- `n` é a quantidade de elementos.
- `arq` é o ponteiro para o arquivo que estamos lendo ou escrevendo.

Essas duas funções retorno um inteiro contendo o número de elementos escritos/-lidos com sucesso.

Exemplo 8.4.1. Implemente um código para escrever e ler o nome Artur.

```
#include <stdio.h>

int main(void)
{
  FILE *arq;
  char str[6]="Artur";
  char vamos_ler_aqui[6];


//escrevendo
  arq=fopen("nome.dat","w+b");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    fwrite(str,sizeof(char),5,arq);
    fclose(arq);
    }

//lendo
  arq=fopen("nome.dat","rb");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    fread(vamos_ler_aqui,sizeof(char),5,arq);
```

```
    fclose(arq);
    }
  printf("Nome=%s\n",vamos_ler_aqui);
}
```

Exemplo 8.4.2. Implemente um código que escreve os dez primeiros números da sequência de Fibonacci em arquivo.

```
#include <stdio.h>

int main(void)
{
  FILE *arq;
  int v[10]={0,1};
  int i;
  for (i=0;i<8;i++) v[i+2]=v[i+1]+v[i];
  arq=fopen("saida.dat","w+b");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    fwrite(v,sizeof(int),10,arq);
    printf("Gravou! Vamos fechar o arquivo\n");
    fclose(arq);
    }
}
```

Exemplo 8.4.3. Implemente um código que lê o arquivo do exemplo anterior.

```
#include <stdio.h>

int main(void)
{
  FILE *arq;
  int v[10];
  int i;
  arq=fopen("saida.dat","rb");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
```

```
    {
    printf("Arquivo aberto com sucesso\n");
    fread(v,sizeof(int),10,arq);
    printf("Leu! Vamos fechar o arquivo\n");
    fclose(arq);
    }
  for (i=0;i<10;i++) printf("v[%i]=%d\n",i,v[i]);
}
```

Em vez de ler/escrever bloco, você pode ler/escrever um de cada vez. Observe uma versão do código 8.4.3.

```
#include <stdio.h>

int main(void)
{
  FILE *arq;
  int v[10];
  int i;
  arq=fopen("saida.dat","rb");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    for (i=0;i<10;i++) fread(v+i,sizeof(int),1,arq);
    printf("Leu! Vamos fechar o arquivo\n");
    fclose(arq);
    }
  for (i=0;i<10;i++) printf("v[%i]=%d\n",i,v[i]);
}
```

ou ainda,

```
#include <stdio.h>

int main(void)
{
  FILE *arq;
  int v[10];
  int i;
  arq=fopen("saida.dat","rb");
```

```
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    for (i=0;i<10;i++) fread(&v[i],sizeof(int),1,arq);
    printf("Leu! Vamos fechar o arquivo\n");
    fclose(arq);
    }
  for (i=0;i<10;i++) printf("v[%i]=%d\n",i,v[i]);
}
```

O retorno das funções `fread` e `fwrite` são inteiro contendo o número de itens lidos com sucesso. Observe:

```
#include <stdio.h>

int main(void)
{
  FILE *arq;
  int v[10];
  int i;
  arq=fopen("saida.dat","rb");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    printf("Leu %zu inteiros\n",fread(v,sizeof(int),10,arq));
    fclose(arq);
    }
  for (i=0;i<10;i++) printf("v[%i]=%d\n",i,v[i]);
}
```

As vezes queremos ler um arquivo e não sabemos o tamanho de antemão. Nesse caso, identificamos o fim do arquivo com a função `feof`, que entra um ponteiro para arquivo e retorna um valor lógico indicando se está ou não no fim do arquivo.

Exemplo 8.4.4. Implemente um código que entre na linha de comando alguns números, escreva todos eles em arquivo binário. Em seguida, faça um programa que lê o arquivo criado e escreva todos os números na tela.

Programa que escreve em arquivo alguns números digitados na linha de comando:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  double v[20];
  FILE *arq;
  int i=0;
  while (argv[i+1]!=NULL)
    {
    v[i]=atof(argv[i+1]);
    i++;
    }
  arq=fopen("saida.dat","w+b");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    printf("Escreveu %zu números\n",fwrite(v,sizeof(double),argc-1,arq));
    fclose(arq);
    printf("Arquivo fechado com sucesso\n");
    }
}
```

Programa que lê o arquivo anterior e escreve na tela

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  double v[20],*p=v;
  FILE *arq;
  int i,ch,total=0;
  arq=fopen("saida.dat","rb");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    while ((ch=fread(p,sizeof(double),1,arq))!=0)
```

```
    {
    total+=ch;
    p++;
    }
    fclose(arq);
    printf("Arquivo fechado com sucesso\n");
    }
printf("total=%d\n",total);
for (i=0;i<total;i++) printf("%f\n",v[i]);
}
```

ou, equivalentemente,

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  double v[20],*p=v;
  FILE *arq;
  int i,ch,total=0;
  arq=fopen("saida.dat","rb");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    //while ((ch=fread(p,sizeof(double),1,arq))!=0)
    do
    {
    ch=fread(p,sizeof(double),1,arq);
    total+=ch;
    p++;
    }while(feof(arq)==0);
    fclose(arq);
    printf("Arquivo fechado com sucesso\n");
    }
printf("total=%d\n",total);
for (i=0;i<total;i++) printf("%f\n",v[i]);
}
```

8.5 Acesso direto e acesso sequencial

Até o momento nós abrimos os arquivos e lemos sequencialmente a partir do início, isto é, fizemos acesso sequencial. Agora, pretendemos ler uma posição do arquivo sem lê-lo desde o início, isto é, pretendemos acessar diretamente um elemento. Os comandos úteis para se posicionarmos ao longo do arquivo são: `ftell`, que entra um ponteiro para arquivo e retorna a posição e `rewind`, que entra um arquivo e o posiciona o início.

Exemplo 8.5.1. Reescreva um código que lê o arquivo gerado no exemplo 8.4.4, depois lê o primeiro número novamente.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  double v[20],*p=v,first;
  FILE *arq;
  int i,ch;
  long int total;
  arq=fopen("saida.dat","rb");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    printf("%ld\n",total=ftell(arq));
    while ((ch=fread(p,sizeof(double),1,arq))!=0)
    {
    p++;
    }
    printf("%ld\n",total=ftell(arq));
    rewind(arq);
    printf("%ld\n",ftell(arq));
    fread(&first,sizeof(double),1,arq);
    printf("%ld\n",ftell(arq));
    fclose(arq);
    printf("Arquivo fechado com sucesso\n");
    }
  printf("%ld\n",total/=8);
  for (i=0;i<total;i++) printf("%f\n",v[i]);
```

```
  printf("first=%f\n",first);
}
```

A função usada para posicionamento dentro do arquivo é a

```
int fseek(FILE *arq,long int salto, int origem)
```

onde

- `FILE` é um ponteiro para arquivo.
- `salto` é o número de bytes que desejamos andar (positivo para frente e negativo para trás).
- `origem` é o local de onde queremos saltar, podemos assumir os seguintes valores:

  `SEEK_SET` salto realizado a partir da origem do arquivo.

  `SEEK_CUR` salto realizado a partir da posição atual.

  `SEEK_END` salto realizado a partir da fim do arquivo.

Exemplo 8.5.2. Reescreva um código que lê o arquivo gerado no exemplo 8.4.4, depois lê somente os três últimos.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  double v[20],*p=v,last[3];
  FILE *arq;
  int i,ch;
  long int total;
  arq=fopen("saida.dat","rb");
  if (arq==NULL)
    printf("Não conseguiu abrir o arquivo\n");
  else
    {
    printf("Arquivo aberto com sucesso\n");
    printf("%ld\n",total=ftell(arq));
    while ((ch=fread(p,sizeof(double),1,arq))!=0)
    {
    p++;
```

```
    }
    printf("%ld\n",total=ftell(arq));
    fseek(arq,-24,SEEK_END);
    printf("onde estou? %ld\n",ftell(arq));
    fread(last,sizeof(double),3,arq);
    fclose(arq);
    printf("Arquivo fechado com sucesso\n");
    }
  printf("%ld\n",total/=8);
  for (i=0;i<total;i++) printf("%f\n",v[i]);
  printf("\n");
  for (i=0;i<3;i++) printf("%f\n",last[i]);
}
```

Exemplo 8.5.3. Vamos resolver o exemplo 4.3.1 novamente imprimindo a solução em arquivo. Vamos usar o mesmo código, ampliando apenas as linhas que salvam em arquivo binário.

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

#define N 101 /* dimensão do sistema*/

double norma_2(double x[N])
{
  int i;
  double norma2=x[0]*x[0]/(2*(N-1));
  for (i=1;i<N-1;i++)
  {
    norma2+=x[i]*x[i]/(N-1);
  }
  norma2+=x[N-1]*x[N-1]/(2*(N-1));
  return sqrt(norma2);
}
void Iteracao(double u_antigo[N],double u_atual[N],double matriz[N][N], double
{
  double aux;
  int i,j;
  for (i=0;i<N;i++)
  {
    aux=0;
```

```
    for (j=0;j<i;j++)
    aux+=matriz[i][j]*u_antigo[j];
    for (j=i+1;j<N;j++)
    aux+=matriz[i][j]*u_antigo[j];
    u_atual[i]=(vetor[i]-aux)/matriz[i][i];
  }
}
void Jacobi(double tolerancia,double u_atual[N],double vetor[N],double matriz[N][N],d
{

  double u_antigo[N],dif[N];
  int i,controle=3;

  //chute inicial
  for (i=0;i<N;i++)u_antigo[i]=0;

  //iteração
  while (controle)
  {
    Iteracao(u_antigo,u_atual,matriz,vetor);
    for (i=0;i<N;i++) dif[i]=u_atual[i]-u_antigo[i];
    if (norma_2(dif)<tolerancia) controle--;
    else controle=3;
    for (i=0;i<N;i++) u_antigo[i]=u_atual[i];
  }
  return;
}

double solucao_analitica(double x)
{
return -25.*(x-1.)*(x-1.)*(x-1.)*(x-1.)/3-25.*x/3+25./3.;
}

main (void)
{
  int i,j;
  double tolerancia=1e-5,solucao[N],sol_anal[N];
  double matriz[N][N];
  double vetor[N],x[N];
  //malha
```

```
  double h=1./(N-1);
  for (i=0;i<N;i++)
    {
    x[i]=i*h;
    sol_anal[i]=solucao_analitica(x[i]);
    }
  //matriz e vetor
  for (i=0;i<N;i++) for (j=0;j<N;j++) matriz[i][j]=0;
  matriz[0][0]=1;
  vetor[0]=0;
  for (i=1;i<N-1;i++)
  {
    matriz[i][i]=-2;
    matriz[i][i+1]=1;
    matriz[i][i-1]=1;
    vetor[i]=-100*h*h*(x[i]-1)*(x[i]-1);
  }
  matriz[N-1][N-1]=1;
  vetor[N-1]=0;

  Jacobi(tolerancia,solucao,vetor,matriz,x);
  //erro médio
  double erro=0;
  for (i=0;i<N;i++) erro+=fabs(solucao[i]-sol_anal[i]);
  erro/=N;
  printf("erro médio = %f\n",erro);
  for (i=0;i<N;i++)     printf("u_%d=%f, u(x[%d]) = %f\n",i,solucao[i],i, sol_a
  FILE *arq;
  arq=fopen("saida.dat","w+b");
  if (arq==NULL)
    printf("Erro ao abrir arquivo\n");
  else
    {
    printf("Arquivo aberto\n");
    fwrite(solucao,sizeof(double),N,arq);
    fwrite(sol_anal,sizeof(double),N,arq);
    printf("Gravou\n");
    fclose(arq);
    }
}
```

Observe um código em scilab para ler e fazer o gráfico das duas soluções:

```
N=101;
x=linspace(0,1,N);
arq=mopen('/home/pasta/subpasta/subsubpasta/saida.dat','rb')
Snum=mget(N,'d',arq)
Sanal=mget(N,'d',arq)
mclose(arq)
plot(x,Snum)
plot(x,Sanal,'red')
```

8.6 Exercícios

E 8.6.1. Escreva um programa que copia o conteúdo de um arquivo para outro arquivo. Faça versões que lê e escreve arquivo de texto e arquivo binário.

E 8.6.2. Abra o gedit e escreva o seguinte arquivo:

```
Artur Avila
O Instituto
A PPGMAp
O DMPA
```

Salve o arquivo com um nome de sua preferência. Agora escreve um programa para ler o arquivo e escrever na tela o seguinte resultado:

```
linha 1: Artur Avila
linha 2: O Instituto
linha 3: A PPGMAp
linha 4: O DMPA
```

E 8.6.3. Faça um programa que lê um arquivo binário e escreve o menor e o maior número na tela.

E 8.6.4. Faça um programa que entra uma ou duas strings na linha de comando, sendo uma delas sempre o nome de um arquivo binário que será lido. O código vai somar os números lidos. Se entrar duas strings, uma será o nome do arquivo e a outra terá três opções: `-p` soma apenas os pares, `-i` soma apenas os ímpares e `-t` soma todos. Se entrar apenas uma string a resposta do programa será a mesma de `-t`.

E 8.6.5. Resolva novamente o exercício 6.7.8 usando a seguinte estratégia:

- Resolva primeiro o problema linear

$$\begin{cases} -u_{xx} = 100, \;\; 0 < x < 2. \\ u(0) = 0 \\ u(2) = 10 \end{cases}$$

e salve a solução em arquivo binário.

- Leia a solução do item anterior e use-a como chute inicial para o problema não linear

$$\begin{cases} -u_{xx} = 100 - \frac{u^4}{10000}, \;\; 0 < x < 2. \\ u(0) = 0 \\ u(2) = 10 \end{cases}$$

Salve a solução em arquivo e use-a para fazer gráfico no scilab.

E 8.6.6. Resolva novamente os exercícios 6.7.13 e 6.7.14, imprima a solução em arquivo e faça o gráfico no scilab.

E 8.6.7. (Ajuste linear - mínimos quadrados)
Dada m funções $\{f_1(x), f_2(x), \ldots, f_m(x)\}$ e um conjunto de n pares ordenados $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, o problema de mínimos quadrados linear consiste em calcular os coeficientes a_1, a_2, ..., a_m tais que a função dada por

$$f(x) = \sum_{j=1}^{m} a_j f_j(x) = a_1 f_1(x) + a_2 f_2(x) + \ldots + a_m f_m(x) \tag{8.1}$$

minimiza o resíduo

$$R = \sum_{i=1}^{n} \left[f(x_i) - y_i \right]^2 . \tag{8.2}$$

Aqui, a minimização é feita por todas as possíveis escolhas dos coeficientes a_1, a_2, ..., a_m. Esse problema é equivalente a encontrar a solução o sistema $Ma = w$, onde a matriz M é dada por:

$$M = \begin{bmatrix} \sum_{i=1}^{n} f_1(x_i)^2 & \sum_{i=1}^{n} f_2(x_i) f_1(x_i) & \cdots & \sum_{i=1}^{n} f_m(x_i) f_1(x_i) \\ \sum_{i=1}^{n} f_1(x_i) f_2(x_i) & \sum_{i=1}^{n} f_2(x_i)^2 & \cdots & \sum_{i=1}^{n} f_m(x_i) f_2(x_i) \\ \sum_{i=1}^{n} f_1(x_i) f_3(x_i) & \sum_{i=1}^{n} f_2(x_i) f_3(x_i) & \cdots & \sum_{i=1}^{n} f_m(x_i) f_3(x_i) \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} f_1(x_i) f_m(x_i) & \sum_{i=1}^{n} f_2(x_i) f_m(x_i) & \cdots & \sum_{i=1}^{n} f_m(x_i)^2 \end{bmatrix} .$$

e os vetores a e w são dados por

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \qquad \text{e} \qquad w = \begin{bmatrix} \sum_{i=1}^{n} f_1(x_i)y_i \\ \sum_{i=1}^{n} f_2(x_i)y_i \\ \sum_{i=1}^{n} f_3(x_i)y_i \\ \vdots \\ \sum_{i=1}^{n} f_m(x_i)y_i \end{bmatrix}.$$

Mais simplificadamente, observamos que $M = V^T V$ e $w = V^T y$, onde a matriz V é dada por:

$$V = \begin{bmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_m(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_m(x_2) \\ f_1(x_3) & f_2(x_3) & \cdots & f_m(x_3) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_n) & f_2(x_n) & \cdots & f_m(x_n) \end{bmatrix} \tag{8.3}$$

e y é o vetor coluna $y = (y_1, y_2, \ldots, y_N)$. Assim, o problema de ajuste se reduz a resolver o sistema linear $Ma = w$, ou $V^T V a = V^T y$.

Implemente um código para calcular os coeficientes de uma reta que melhor se ajusta a N pontos no plano. Imprime o resultado em arquivo e faça gráfico da solução no scilab.

Capítulo 9

Estruturas

9.1 Estruturas

Estrutura é uma forma de agrupar um ou mais tipos de variáveis, definindo um novo tipo. Declaramos uma estrutura com a palavra `struct`. Acessamos os membros de uma estrutura usando um ponto.

Exemplo 9.1.1. Defina um estrutura que suporta uma data no formato dia/-mês/ano.

```
#include <stdio.h>

struct data
{
  int dia;
  char *mes;
  int ano;
}d;

void main(void)
{
  d.dia=1;
  d.mes="janeiro";
  d.ano=2018;
  printf("A data é %d de %s de %d\n",d.dia,d.mes,d.ano);
}
```

No exemplo 9.1.1, nós definimos uma estrutura para armazenar a data da forma `struct data`. Três variáveis existem dentro da estrutura, sendo duas inteiras e

uma ponteiro para `char`. Observe que `data` é o nome da estrutura e `d` é uma variável do tipo `data`. Assim, dentro do `main`, acessamos as variáveis internas da estrutura através da variavél `d` usando o ponto: `d.dia`, `d.mes` e `d.ano`.

Um estrutura admite uma carga inicial, veja outra versão do exemplo 9.1.1:

```
#include <stdio.h>

struct data
{
  int dia;
  char *mes;
  int ano;
}d={1,"janeiro",2018};

void main(void)
{
  printf("A data é %d de %s de %d\n",d.dia,d.mes,d.ano);
  d.dia=5;
  d.mes="fevereiro";
  d.ano=2017;
  printf("A data é %d de %s de %d\n",d.dia,d.mes,d.ano);
}
```

A ordem da carga inicial é a mesma ordem da definição da estrutura. Alternativamente, podemos definir a variável do tipo `data` no curso do programa:

```
#include <stdio.h>

struct data
{
  int dia;
  char *mes;
  int ano;
};

void main(void)
{
  struct data d={1,"janeiro",2018};
  printf("A data é %d de %s de %d\n",d.dia,d.mes,d.ano);
  d.dia=5;
  d.mes="fevereiro";
  d.ano=2017;
```

```
  printf("A data é %d de %s de %d\n",d.dia,d.mes,d.ano);
}
```

Em linguagem C também admite-se estrutura dentro de estrutura.

9.2 Passagem de estrutura para funções

A passagem de estrutura para função se faz de forma similar a outro tipo de variável.

Exemplo 9.2.1. Implemente um programa que valide uma data e imprime o resultado.

```
#include <stdio.h>

struct data
{
int dia;
int mes;
int ano;
};

int bissexto(int ano);
int Max_mes(struct data d);
int testa_data(struct data d);

int main(int argc,char **argv)
{
struct data d;
d.dia=atoi(argv[1]);
d.mes=atoi(argv[2]);
d.ano=atoi(argv[3]);
if (testa_data(d)==1) printf("%d / %d / %d é uma data válida\n",d.dia,d.mes,d.ar
}

int bissexto(int ano)
{
if ((ano%4)==0)
{
if ((ano%100)!=0) return 1;
}
```

```
if ((ano%400)==0) return 1;

return 0;
}
int Max_mes(struct data d)
{
 int bi=bissexto(d.ano);
 switch (d.mes)
{
case 1: return 31; break;
case 2:
{
if (bi==0) return 28;
else  return 29;
break;
}
case 3: return 31; break;
case 4: return 30; break;
case 5: return 31; break;
case 6: return 30; break;
case 7: return 31; break;
case 8: return 31; break;
case 9: return 30; break;
case 10:return 31; break;
case 11:return 30; break;
case 12:return 31; break;
default:return 0; break;
}

}

int testa_data(struct data d)
{
  int max_mes=Max_mes(d);
  if (max_mes==0)
  {
    printf("Escolha um mes entre 1 e 12\n");
    return 0;
  }
  else
```

```
  {
    if (d.dia<=max_mes)
    {
      return 1;
    }
    else
    {
      printf("data incorreta: o mes %d tem no máximo %d dias\n",d.mes,max_mes);
      return 0;
    }
  }
}
```

A código acima tem três função, duas delas passam uma estrutura e retornam um inteiro. A estrutura tem três membros que são variáveis do tipo `int`. A função `bissexto` passa a variável `ano` e retorna 1 se é bissexto e 0 caso contrário. A função `Max_mes` calcula quandos dias tem um mês, levando em conta se ele é bissexto. Essa função retorna 0 se o a o número de dias do mês estiver errado ou se o número do mês não estiver entre 1 e 12.

Também podemos passar um endereço de memória de uma estrutura para uma função. Nesse caso, para pegar o valor de uma variável dentro de uma estrutura através do ponteiro, podemos usar `(*p).nome` ou `p->nome`.

Exemplo 9.2.2. Vamos refazer o exemplo 9.1.1 usando ponteiro para estrutura.

```
#include <stdio.h>

struct data
{
  int dia;
  char *mes;
  int ano;
};

void Escreve(struct data *p)
{
  printf("dia= %d\n",p->dia);
  printf("mes= %s\n",p->mes);
  printf("ano= %d\n\n",p->ano);
}
```

```
void main(void)
{

  struct data d;
  d.dia=1;
  d.mes="janeiro";
  d.ano=2018;
  Escreve(&d);
}
```

Também podemos fazer as seguintes operações entre estruturas:

- `x.m` é o valor de `m` dentro da estrutura `x`.
- `&x.m` é o endereço de `m` dentro da estrutura `x`.
- `&x` é o endereço da estrutura `x`.
- `(*p).m` ou `p->m` é o valor de `m` dentro da estrutura `x` dado que `p` é um ponteiro para a estrutura `x`.
- `x=y` atribui a `x` todo o conteúdo de `y` dado que `x` e `y` têm as mesmas estruturas.

Exemplo 9.2.3. Refaça o exemplo 9.2.1 passando ponteiro para estrutura para as funções.

```
#include <stdio.h>

struct data
{
int dia;
int mes;
int ano;
};

int bissexto(int ano);
int Max_mes(struct data *d);
int testa_data(struct data *d);

int main(int argc,char **argv)
{
if (argc<4)
{
```

```
  printf("Entre com uma data na linha de comando\n");
  return 0;
}
struct data d;
d.dia=atoi(argv[1]);
d.mes=atoi(argv[2]);
d.ano=atoi(argv[3]);
if (testa_data(&d)==1) printf("%d / %d / %d é uma data válida\n",d.dia,d.mes,d.a
}

int bissexto(int ano)
{
if ((ano%4)==0)
{
if ((ano%100)!=0) return 1;
}
if ((ano%400)==0) return 1;

return 0;
}
int Max_mes(struct data *d)
{
 int bi=bissexto(d->ano);
 switch (d->mes)
{
case 1: return 31; break;
case 2:
{
if (bi==0) return 28;
else  return 29;
break;
}
case 3: return 31; break;
case 4: return 30; break;
case 5: return 31; break;
case 6: return 30; break;
case 7: return 31; break;
case 8: return 31; break;
case 9: return 30; break;
case 10:return 31; break;
```

```
case 11:return 30; break;
case 12:return 31; break;
default:return 0; break;
}

}

int testa_data(struct data *d)
{
  int max_mes=Max_mes(d);
  if (max_mes==0)
  {
    printf("Escolha um mes entre 1 e 12\n");
    return 0;
  }
  else
  {
    if (d->dia<=max_mes)
    {
      return 1;
    }
    else
    {
      printf("data incorreta: o mes %d tem no máximo %d dias\n",d->mes,max_mes);
      return 0;
      }
    }
}
```

9.3 Problemas

Exemplo 9.3.1. Seja

$$F(x) = \int_0^x t^4 e^{-50(t-2)^2} dt.$$

Faça um código para calcular $F(x)$, salve seus valores em arquivo e visualize a função no scilab.

Vamos integrar usando a regra de Boole

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{2h}{45}\left(7f(x_1)+32f(x_2)+12f(x_3)+32f(x_4)+14f(x_5)+32f(x_6)+12f(x_7)+32f(x_8)+\right.\\ &+ \left.14f(x_{4n-3})+32f(x_{4n-2})+12f(x_{4n-1})+32f(x_{4n})+7f(x_{4n+1})\right)\end{aligned}$$

onde

$$\begin{aligned} h &= \frac{b-a}{4n} \\ x_i &= a+(i-1)h, \qquad i = 1,2,\cdots,4n+1. \end{aligned}$$

Vamos resolver o problema por partes. Primeiro, vamos implementar uma rotina que integra uma função pelo método de Boole.

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

double f(double t)
{
return t*t*t*t*exp(-50*(t-2)*(t-2));
}

void main(void)
{
  int N=10,i;
  double a=0,b=4,h=(b-a)/(4*N),Int=0;

  Int=14./45.*h*f(a);
  for(i=0;i<N;i++)
  {
    Int+=64./45.*h*f(a+h);
    Int+=24./45.*h*f(a+2*h);
    Int+=64./45.*h*f(a+3*h);
    Int+=28./45.*h*f(a+4*h);
    a+=4*h;
  }
  Int-=14./45.*h*f(a);
printf("%f\n",Int);
}
```

Agora, vamos usar uma estrutura com a,b, e N.

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

struct INT
{
double a;
```

```
double b;
int    N;
};

double f(double t)
{
return t*t*t*t*exp(-50*(t-2)*(t-2));
}

double Boole(struct INT *I)
{
  int i,N=I->N;
  double a=I->a,b=I->b,h=(b-a)/(4*N),Int=0;

  Int=14./45.*f(a);
  for(i=0;i<N;i++)
  {
    Int+=64./45.*f(a+h);
    Int+=24./45.*f(a+2*h);
    Int+=64./45.*f(a+3*h);
    Int+=28./45.*f(a+4*h);
    a+=4*h;
  }
  Int-=14./45.*f(a);
return h*Int;
}

void main(void)
{
  struct INT I;
  I.a=0;
  I.b=4;
  I.N=10;

printf("%f\n",Boole(&I));
}
```

Agora vamos fazer a primeira versão do código que resolve o problema acima. Vamos calcular $F(x)$ para onze pontos no intervalo [0,4].

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

```
struct INT
{
double a;
double b;
int    N;
};

double f(double t)
{
return t*t*t*t*exp(-50*(t-2)*(t-2));
}

double Boole(struct INT *I)
{
  int i,N=I->N;
  double a=I->a,b=I->b,h=(b-a)/(4*N),Int=0;

  Int=14./45.*f(a);
  for(i=0;i<N;i++)
  {
    Int+=64./45.*f(a+h);
    Int+=24./45.*f(a+2*h);
    Int+=64./45.*f(a+3*h);
    Int+=28./45.*f(a+4*h);
    a+=4*h;
  }
  Int-=14./45.*f(a);
return h*Int;
}

void main(void)
{
  struct INT I;
  I.a=0;
  I.N=10;
  int i,M=10;
  for(i=0;i<=M;i++)
  {
  I.b=4.*i/M;
```

```
  printf("%f\n",Boole(&I));
  }
}
```

Veja, até agora calculamos as integrais sem preocupação com a convergência. Então, vamos implementar um código que calcule as integrais até que o erro fica menor que uma tolerância relativa de 10^{-8}.

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

struct INT
{
double a;
double b;
int    N;
};

double f(double t)
{
return t*t*t*t*exp(-50*(t-2)*(t-2));
}

double Boole(struct INT *I)
{
  int i,N=I->N;
  double a=I->a,b=I->b,h=(b-a)/(4*N),Int=0;

  Int=14./45.*f(a);
  for(i=0;i<N;i++)
  {
    Int+=64./45.*f(a+h);
    Int+=24./45.*f(a+2*h);
    Int+=64./45.*f(a+3*h);
    Int+=28./45.*f(a+4*h);
    a+=4*h;
  }
  Int-=14./45.*f(a);
return h*Int;
}
double Integral(struct INT *I)
{
```

```
  double I1,I2,aux;
  I1=Boole(I);
  do
  {
    (I->N)*=2;
    I2=Boole(I);
    aux=fabs(I2-I1);
    I1=I2;
  }while ((aux/fabs(I2))>1e-8);
  return I2;
}

double F(double x)
{
  struct INT I;
  I.a=0;
  I.N=10;
  I.b=x;
  return Integral(&I);
}
void main(void)
{
  int i,M=1000;
  double res;
  FILE *arq;
  arq=fopen("saida.dat","w+b");
  if (arq==NULL) printf("Não foi possível abrir o arquivo");
  else
  {
    printf("arquivo aberto com sucesso");
    for(i=0;i<=M;i++)
    {
      res=F(4.*i/M);
      fwrite(&res,sizeof(double),1,arq);
      printf("%f\n",res);
    }
    fclose(arq);
  }
}
```

Agora vamos imprimir o resultado em arquivo e abrir no scilab:

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

struct INT
{
double a;
double b;
int    N;
};

double f(double t)
{
return t*t*t*t*exp(-50*(t-2)*(t-2));
}

double Boole(struct INT *I)
{
  int i,N=I->N;
  double a=I->a,b=I->b,h=(b-a)/(4*N),Int=0;

  Int=14./45.*f(a);
  for(i=0;i<N;i++)
  {
    Int+=64./45.*f(a+h);
    Int+=24./45.*f(a+2*h);
    Int+=64./45.*f(a+3*h);
    Int+=28./45.*f(a+4*h);
    a+=4*h;
  }
  Int-=14./45.*f(a);
return h*Int;
}
double Integral(struct INT *I)
{
  double I1,I2,aux;
  I1=Boole(I);
  do
  {
    (I->N)*=2;
    I2=Boole(I);
```

```
    aux=fabs(I2-I1);
    I1=I2;
  }while ((aux/fabs(I2))>1e-8);
  return I2;
}

double F(double x)
{
  struct INT I;
  I.a=0;
  I.N=10;
  I.b=x;
  return Integral(&I);
}
void main(void)
{
  int i,M=1000;
  double res;
  FILE *arq;
  arq=fopen("saida.dat","w+b");
  if (arq==NULL) printf("Não foi possível abrir o arquivo");
  else
  {
    printf("arquivo aberto com sucesso");
    for(i=0;i<=M;i++)
    {
      res=F(4.*i/M);
      fwrite(&res,sizeof(double),1,arq);
      printf("%f\n",res);
    }
    fclose(arq);
  }
}
```

Abrimos no scilab com os seguintes comandos:

```
M=1000;
x=linspace(0,4,M);
arq=mopen('home/fulano/pasta1/pasta2/saida.dat','rb')
Snum=mget(M,'d',arq)
mclose(arq)
plot(x,Snum)
```

Comentários sobre os códigos:

- O ponto na linha `res=F(4.*i/M);` é importante. Substituir a linha por `res=F(4*i/M);` não produz o resultado esperado.
- As linhas `return Integral(&I);` e `I1=Boole(I);` passam um ponteiro para a mesma estrutura `INT`, mas a primera usa `&I` e a segunda `I`. Procure entender essa diferença na notação.

Observe que a rotina de integração vai aumentando o número de pontos até a convergência. Uma alternativa mais eficiente é implementar um esquema auto-adaptativo com refinamento local, isto é, adicionar pontos apenas nas regiões onde a integral ainda não convergiu.

9.4 Exercícios

E 9.4.1. Faça uma nova versão do exercício 6.7.13.

- Use o método de Adams-Bashforth terceira ordem:

$$u^{(n+3)} = u^{(n+2)} + \frac{h}{12}\left[23f\left(t^{(n+2)},u(t^{(n+2)})\right) - 16f\left(t^{(n+1)},u(t^{(n+1)})\right) + 5f\left(t^{(n)},u(t^{(n)})\right)\right] \tag{9.1}$$

- Use o método de Kunge-Kutta terceira ordem para calcular as três primeiras iterações:

$$\begin{aligned} k_1 &= hf\left(t^{(n)},u^{(n)}\right) \\ k_2 &= hf\left(t^{(n)}+h/2,u^{(n)}+k_1/2\right) \\ k_3 &= hf\left(t^{(n)}+h,u^{(n)}-k1+2k_2\right) \\ u^{(n+1)} &= u^{(n)} + \frac{k_1+4k_2+k_3}{6} \end{aligned}$$

- Faça uma rotina geral para sistemas de equações diferenciais ordinárias.

Teste usando vários PVIs:

1.

$$\begin{aligned} u' &= u + e^{-t}, \\ u(0) &= 1 \end{aligned}$$

2.

$$\begin{aligned} u' &= -2u + v, \\ v' &= u - 2v - e^{-t^2}, \\ u(0) &= 1 \\ v(0) &= 0. \end{aligned}$$

3.

$$\begin{aligned} u' &= -2u + v - 2w, \\ v' &= u - 2v + w, \\ w' &= u + v - w + e^{-t}, \\ u(0) &= 1 \\ v(0) &= 0 \\ w(0) &= -1. \end{aligned}$$

E 9.4.2. Implemente um algorítmo para calcular a seguinte integral dupla:

$$F(x) = \int_0^1 \int_0^1 \frac{1}{\mu} e^{-\frac{1}{\mu}|s-x|} Q(s) d\mu ds$$

onde $Q(s) = s(1-s)$.

Dicas:

- Embora a integral está bem definida, tem singularidades quando $x = s$ e $\mu = 0$. Logo, um abordagem ingênua produz resultado com bastante erro.
- Use o seguinte truque para remover a singularidade:

$$F(x) = \int_0^1 \int_0^1 \frac{1}{\mu} e^{-\frac{1}{\mu}|s-x|} (Q(s) - Q(x)) d\mu ds + Q(x) \int_0^1 \int_0^1 \frac{1}{\mu} e^{-\frac{1}{\mu}|s-x|} d\mu ds.$$

A primeira integral fica numericamente bem comportada e a segunda podemos integral analiticamente:

$$\begin{aligned} \int_0^1 \int_0^1 \frac{1}{\mu} e^{-\frac{1}{\mu}|s-x|} d\mu ds &= \int_0^1 \int_0^1 \frac{1}{\mu} e^{-\frac{1}{\mu}|s-x|} ds d\mu \\ &= \int_0^1 \left[\int_0^x \frac{1}{\mu} e^{-\frac{1}{\mu}(x-s)} ds + \int_x^1 \frac{1}{\mu} e^{-\frac{1}{\mu}(s-x)} ds \right] d\mu \\ &= \int_0^1 \left[e^{-\frac{1}{\mu}(x-s)} \Big|_0^x - e^{-\frac{1}{\mu}(s-x)} \Big|_x^1 \right] d\mu \\ &= \int_0^1 \left[1 - e^{-\frac{x}{\mu}} - e^{-\frac{1}{\mu}(1-x)} + 1 \right] d\mu \\ &= 2 - \int_0^1 \left[e^{-\frac{x}{\mu}} + e^{-\frac{(1-x)}{\mu}} \right] d\mu. \end{aligned}$$

Capítulo 10

Alocação dinâmica de memória

10.1 Alocação dinâmica de memória

O objetivo é definir um ponteiro no início do programa e alocar/liberar memória durante a execução. Até agora, nossos programas alocavam memória no momento que definíamos as variáveis. Por exemplo, `double x[N];` define uma variável do tipo `double`, como o nome `x` e aloca $8 * N$ bytes para armazená-la. Suponha que nós não sabemos de antemão quantos bytes serão necessários, pois o número de bytes será calculado na primeira parte do código? Nesse caso, usamos as funções da biblioteca `<stdlib.h>` para alocar/liberar memória durante a execução do programa. As funções

```
void *malloc(size_t numero_de_bytes)
```

e

```
void *calloc(size_t numero_de_elementos, size_t tamanho_de_cada_elementos)
```

permitem alocar memória dinamicamente. Aqui, `size_t` é o tipo `unsigned int` definido dentro da biblioteca `<stdlib.h>`, `numero_de_bytes` é o número de bytes, `numero_de_elementos` é o número de elementos e `tamanho_de_cada_elementos` é o tamanho de cada elementos em bytes. Ambas funções retornam um ponteiro para a posição de memória onde está sendo alocado ou `NULL` no caso de falha de alocamento. A diferença básica é que a segunda função já coloca um carga inicial nula, enquanto a primeira não atribui valor algum. A função `void free(void *p)` entra o ponteiro e libera a memória alocada.

Exemplo 10.1.1. Faça um programa que armazena os N primeiros números da sequência de Fibonacci. Leia N com a função `scanf`.

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  int N,i;
  int *FIB;
  printf("Entre com o número de termos\n");
  scanf("%d",&N);
  FIB=malloc(N*sizeof(int));
  if (FIB==NULL) printf("Não alocou memória corretamente\n");
  else
  {
    printf("Memória alocada\n");
    FIB[0]=0;
    FIB[1]=1;
    for (i=2;i<N;i++)
    {
      FIB[i]=FIB[i-1]+FIB[i-2];
    }
  }
  for (i=0;i<N;i++) printf("%d\n",FIB[i]);
  free(FIB);
  printf("Memória liberada\n");
}
```

A função

```
void *realloc(void *p, size_t novo_tamanho_em_bytes)
```

permite realocar a memória que já foi alocada no ponteiro `p` com um novo tamanho `novo_tamanho_em_bytes`.

Exemplo 10.1.2. Faça um programa que armazena os N primeiros números da sequência de Fibonacci. Depois, use o realloc para armazenar novamente os N números de trás para frente.

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
```

```
  int N,i;
  int *FIB;
  printf("Entre com o número de termos\n");
  scanf("%d",&N);
  FIB=malloc(N*sizeof(int));
  if (FIB==NULL) printf("Não alocou memória corretamente\n");
  else
  {
    printf("Memória alocada\n");
    FIB[0]=0;
    FIB[1]=1;
    for (i=2;i<N;i++)
    {
      FIB[i]=FIB[i-1]+FIB[i-2];
    }
    for (i=0;i<N;i++) printf("%d\n",FIB[i]);
    FIB=realloc(FIB,2*N*sizeof(double));
    for (i=N;i<2*N;i++)
    {
      FIB[i]=FIB[2*N-i-1];
    }
    for (i=0;i<2*N;i++) printf("%d\n",FIB[i]);
  }
  free(FIB);
  printf("Memória liberada\n");
}
```

Exemplo 10.1.3. Voltamos ao exemplo 6.3.2: resolver o PVC

$$\begin{cases} -u_{xx} + u_x = 200e^{-(x-1)^2}, \ \ 0 < x < 1. \\ 15u(0) + u'(0) = 500 \\ 10u(1) + u'(1) = 1 \end{cases}$$

usando o método de Thomas.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define N 11
```

```
//Método de Thomas para resolver Ax=y
//x entra o vetor y e saí a solução x (N posições, i=0,....,N-1)
//a entra a subdiagonal à esquerda de A (N-1 posições, i=1,...,N-1)
//b entra a diagonal  (N posições, i=0,....,N-1)
//c entra a subdiagonal à direita de A (N-1 posições, i=0,...,N-2)
void Thomas(double x[], const double a[], const double b[], double c[])
{
  int i;

  /*calculo de c[0]' e d[0]'*/
  c[0] = c[0] / b[0];
  x[0] = x[0] / b[0];


  /* laço para calcular c' e d' */
  for (i = 1; i < N; i++)
  {
    double aux = 1.0/ (b[i] - a[i] * c[i - 1]);
    c[i] = c[i] * aux;
    x[i] = (x[i] - a[i] * x[i - 1]) * aux;
  }

  /* Calculando a solução */
  for (i = N - 1; i >= 0; i--)
    x[i] = x[i] - c[i] * x[i + 1];
}


void main (void)
{
  double *x,*a,*b,*c;
  if ((x=malloc(N*sizeof(double)))==NULL)
  {
  printf("Não alocou memória");
  return;
  }
  if ((a=malloc(N*sizeof(double)))==NULL)
  {
  printf("Não alocou memória");
  return;
```

```
  }
  if ((b=malloc(N*sizeof(double)))==NULL)
  {
  printf("Não alocou memória");
  return;
  }
  if ((c=malloc(N*sizeof(double)))==NULL)
  {
  printf("Não alocou memória");
  return;
  }
  a[0]=0;c[N-1]=0;

  //malha
  int i;
  double h=1./(N-1);
  double p[N];
  for (i=0;i<N;i++) p[i]=i*h;

  //sistema
  b[0]=15-1/(h);
  c[0]=1/(h);
  x[0]=500;
  for (i=1;i<N-1;i++)
  {
  b[i]=2/(h*h);
  a[i]=-1/(h*h)-1/(2*h);
  c[i]=-1/(h*h)+1/(2*h);
  x[i]=200*exp(-(p[i]-1)*(p[i]-1));
  }
  b[N-1]=10+1/h;
  a[N-1]=-1/h;
  x[N-1]=1;
  Thomas(x,a,b,c);
  free(a);
  free(b);
  free(c);
  for (i=0;i<N;i++) printf("%f\n",x[i]);
  free(x);
}
```

Exemplo 10.1.4. Implemente um código que aloque dinamicamente espaço para uma matriz $N \times N$ com as entradas $A_{i,j} = i^2 + j^2$.

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

void preencher(double **A,int N)
{
  int i,j;
  for (i=0;i<N;i++)
  {
    for (j=0;j<N;j++)
    {
      A[i][j]=i*i+j*j;
    }
  }
}

void main (void)
{
  int i,j,N=3;
  double **A;
  if ((A=malloc(N*sizeof(double)))==NULL)
  {
  printf("Não alocou memória");
  return;
  }
  for (i=0;i<N;i++)
  {
    if ((A[i]=malloc(N*sizeof(double)))==NULL)
    {
    printf("Não alocou memória");
    return;
    }
  }
  preencher(A,N);
  for (i=0;i<N;i++)
  {
    for (j=0;j<N;j++)
    {
```

```
      printf(" %f ",A[i][j]);
    }
  printf("\n");
  }
}
```

Naturalmente, você pode armazenar toda a informação de uma matriz em um vetor. Na prática, basta a estrutura de vetor para resolver a maioria dos problemas matriciais. Observe uma versão do código do exemplo 10.1.4 usando apenas vetor:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define N 3

int m(int i, int j)
{
  return i+N*j;
}

void preencher(double *A)
{
  int i,j;
  for (i=0;i<N;i++)
  {
    for (j=0;j<N;j++)
    {
      A[m(i,j)]=i*i+j*j;
    }
  }
}

void main (void)
{
  int i,j;
  double *A;
  if ((A=malloc(N*N*sizeof(double)))==NULL)
  {
  printf("Não alocou memória");
  return;
  }
```

```
  preencher(A);
  for (i=0;i<N;i++)
  {
    for (j=0;j<N;j++)
    {
      printf(" %f ",A[m(i,j)]);
    }
  printf("\n");
  }
}
```

10.2 Exercícios

E 10.2.1. Implemente um programa de com as seguintes características:

- lê N números inteiros na linha de comando, a_1, a_2, ... a_N.
- Calcule a soma $S = a_1 + a_2 + \cdots + a_N$.
- Faça o seguinte teste:

  Se a divisão de S por 3 tiver resto 0, então aloque dinamicamente memória para o vetor de $2N$ posições com as entradas $V_i = a_i$, $i = 1,2,...,N$ e $V_i = a_{2N-i}$, $i = N,N+1,...,2N$.

  Se a divisão de S por 3 tiver resto 1, então aloque dinamicamente memória para o vetor de N^2 posições com as entradas $V_{(i+N(j-1))} = a_i + a_j$, $i = 1,2,...,N$ e $j = 1,2,...,N$.

  Se a divisão de S por 3 tiver resto 2, então aloque dinamicamente memória para o vetor de N posições com as entradas $V_i = a_i$, $i = 1,2,...,N$.

E 10.2.2. Considere o seguinte problema

$$\begin{aligned} -\frac{dI_1}{dy} + I_1 &= \frac{1}{8}(I_1 + I_2), \qquad 0 < y < 1 \\ \frac{dI_2}{dy} + I_2 &= \frac{1}{8}(I_1 + I_2), \qquad 0 < y < 1 \\ I_1(1) &= 1 \\ I_2(0) &= 1 \end{aligned}$$

Observações:

- Embora o problema possua solução analítica, vamos resolvê-lo numericamente.
- Esse problema não é um problema de valor inicial, portanto, os métodos de Runge-Kutta, Adams, BDF's não se aplicam.
- Discretize o domínio y, $y_k = k/N$, $k = 0, \cdots, N$, e resolva o problema

$$\begin{aligned} \frac{dI_1}{dy} - I_1 &= -S^{k+1/2}, \qquad y_k < y < y_{k+1} \\ \frac{dI_2}{dy} + I_2 &= S^{k+1/2}, \qquad y_k < y < y_{k+1} \end{aligned}$$

supondo que $S^{k+1/2}$ é um valor conhecido no intervalo $[y_k, y_{k+1}]$. Assim:

$$\begin{aligned} \frac{d(I_1 e^{-y})}{dy} &= -e^{-y} S^{k+1/2}, \qquad y_k < y < y_{k+1} \\ \frac{d(I_2 e^{y})}{dy} &= e^{y} S^{k+1/2}, \qquad y_k < y < y_{k+1}. \end{aligned}$$

Agora, integramos no intervalo $[y_k, y_{k+1}]$:

$$\begin{aligned} I_1 e^{-y}\Big|_{y_k}^{y_{k+1}} &= S^{k+1/2}\, e^{-y}\Big|_{y_k}^{y_{k+1}} \\ I_2 e^{y}\Big|_{y_k}^{y_{k+1}} &= S^{k+1/2} e^{y}\Big|_{y_k}^{y_{k+1}}. \end{aligned}$$

Assim, temos:

$$\begin{aligned} I_1^{k+1} e^{-y_{k+1}} - I_1^k e^{-y_k} &= S^{k+1/2}\left(e^{-y_{k+1}} - e^{-y_k}\right) \\ I_2^{k+1} e^{y_{k+1}} - I_2^k e^{y_k} &= S^{k+1/2}\left(e^{y_{k+1}} - e^{y_k}\right). \end{aligned}$$

Finalmente, temos o seguinte algoritmo:

$$\begin{aligned} I_1^k &= I_1^{k+1} e^{y_k - y_{k+1}} + S^{k+1/2}\left(1 - e^{y_k - y_{k+1}}\right), \qquad k = N-1, N-2, \cdots, 1, 0, \\ I_2^{k+1} &= I_2^k e^{y_k - y_{k+1}} + S^{k+1/2}\left(1 - e^{y_k - y_{k+1}}\right), \qquad k = 0, 1, 2, \cdots, N-1, \\ I_1^N &= 1, \\ I_2^0 &= 1. \end{aligned}$$

onde a primeira aproximação $S^{k+1/2}$ é zero e, depois de calcular I_1^k e I_2^k, calculamos uma nova aproximação

$$S^{k+1/2} = \frac{1}{16}\left(I_1^k + I_2^k\right) + \frac{1}{16}\left(I_1^{k+1} + I_2^{k+1}\right).$$

Atualizamos $S^{k+1/2}$ até o processo iterativo converjir.

E 10.2.3. Resolva o seguinte problema de transporte

$$\begin{aligned}
-\frac{dI_1}{dy} + I_1 &= \frac{1}{24}\left(I_1 + 2I_2 + 2I_3 + I_4\right), \qquad 0 < y < 1 \\
-\frac{1}{3}\frac{dI_2}{dy} + I_2 &= \frac{1}{24}\left(I_1 + 2I_2 + 2I_3 + I_4\right), \qquad 0 < y < 1 \\
\frac{1}{3}\frac{dI_3}{dy} + I_3 &= \frac{1}{24}\left(I_1 + 2I_2 + 2I_3 + I_4\right), \qquad 0 < y < 1 \\
\frac{dI_4}{dy} + I_4 &= \frac{1}{24}\left(I_1 + 2I_2 + 2I_3 + I_4\right), \qquad 0 < y < 1 \\
I_1(1) &= 0.5 \\
I_2(1) &= 0.5 \\
I_3(0) &= 0.5 \\
I_4(0) &= 0.5
\end{aligned}$$

Observação: Use o algoritmo introduzida do exercício 10.2.2.

Capítulo 11

Macros e pré-processamento

11.1 Macros

São diretivas do pré-processamento que permitem substituir porções de código antes da compilação. As instruções do pré-processamento são seguidas do operador `#`.

a) `#define cod exp` substitui o `cod` por `verb`.

b) `#include <cod.h>` inclui o código da biblioteca `cod.h`. Essa sintaxe vale para as biliotecas padrão.

c) `#include "home/fulano/aula/cod.h"` inclui o código da biblioteca `cod.h`. Essa sintaxe vale para bibliotecas próprias salva com o nome `cod.h` na pasta `home/fulano/aula/`.

Exemplo 11.1.1. Implemente uma biblioteca com uma função que troca `x` e `y`.

Salvamos um arquivo com nome `lib.h` com as seguintes linhas:

```
//lib.h
void troca(double *a,double *b)
{
double aux;
aux=*a;
*a=*b;
*b=aux;
}
```

Depois, salvamos um arquivo com o nome teste.c com as linhas:

```
#include <stdio.h>
#include <stdlib.h>

#include "lib.h"

#define Mult(x,y) (x)*(y)

void main (int argc,char **argv)
{
  if (argc!=3)
  {
    printf("Digite dois números\n");
    return;
  }
  double x=atof(argv[1]);
  double y=atof(argv[2]);
  printf("x=%f, y=%f,  xy=%f\n",x,y,Mult(x,y));
  troca(&x,&y);
  printf("x=%f, y=%f,  xy=%f\n",x,y,Mult(x,y));
}
```

Exemplo 11.1.2. Várias implementações da função mínimos entre dois números.

Primeiro implementamos um a biblioteca com três tipos de função mínimo:

```
//salva como lib.h
double Min1(double a,double b)
{
if (a<b) return a;
else return b;
}

void Min2(double *a,double b)
{
if (b<*a) *a=b;
}

double Min3(double a,double b)
{
return a<b?a:b;
}
```

Depois, implementamos um código que inclui a primeira biblioteca:

```
#include <stdio.h>
#include <stdlib.h>

#include "lib.h"

#define Min4(a,b) ((a)<(b))?(a):(b)

void main (int argc,char **argv)
{
  double x=2,y=3;
  printf("Min1(%f,%f)=%f\n",x,y,Min1(x,y));
  printf("Min3(%f,%f)=%f\n",x,y,Min3(x,y));
  printf("Min4(%f,%f)=%f\n",x,y,Min4(x,y));
  Min2(&x,y);
  printf("Min4=%f\n",x);
}
```

Observe a funcionamento da linha `return a<b?a:b;` na função `Min3`. Essa é uma forma de escrever a condição

```
if (a<b) return a;
else return b;
```

A expressão `a<b?a:b` devolve o valor do teste: Se `(a<b)` devolve `a`, senão devolve `b`. O macro `#define Min4(a,b) ((a)<(b))?(a):(b)` também calcula o mínimo entre `a` e `b`. Um macro está escrito em uma única linha que inicia com `#`. Se a expressão for muito grande e você precisar de duas linhas, use o operador continuidade \:

```
#define Min4(a,b) ((a)<(b))?\
                        (a):\
                        (b)
```

11.2 Compilação condicional

Quando não queremos compilar um pedaço de código, podemos usar a opção comentar `//` ou `/* */`. Mas as vezes, queremos mais de uma versão do código, compilando pedaços segundo uma condição que escolhemos. Usamos a sintaxe

```
#if condicao_1
  instrucao_1
#elif condicao_2
```

```
  instrucao_2
#else
  instrucao_padrao
#endif
```

Exemplo 11.2.1. Faça um programa com dois `main()`, um curto para fazer DEBUG e outro longo com um código que some n números.

```
#include <stdio.h>
#include <stdlib.h>

#define DEBUG 1

#if (DEBUG==1)
void main (int argc,char **argv)
{
  double soma=0;
  int i;
  for (i=1;i<argc;i++) soma+=atof(argv[i]);
  printf("A soma é =%f\n",soma);
}

#else
void main (int argc,char **argv)
{
  printf("Entrou com %d números\n ",argc-1);
}
#endif
```

Também podemos usar as diretivas `#ifdef`, `#ifndef` e `#undef` para escolher um pedaço de código na compilação. A diretiva `#ifdef` verifica se um determinado símbolo está definido, `#ifndef` verifica que um determinado símbolo não está definido e `#undef` retira a definição de um símbolo.

Exemplo 11.2.2. Faça um programa que soma n números de duas formas:

- Se ligar a condição linha de comando, os números são lidos da linha de comando.
- senão, os números são lidos da tela com `scanf`.

```
#include <stdio.h>
#include <stdlib.h>
```

```
//#define DEBUG

#ifdef DEBUG
void main (int argc,char **argv)
{
#endif
#ifndef DEBUG
void main (void)
{
#endif
  double soma=0;
  int i;
#ifndef DEBUG
  int n;
  printf("Digite quantos números você vai somar\n");
  scanf("%d",&n);
  float v[n];
  printf("Digite os números\n");
  for (i=0;i<n;i++) scanf("%f",&v[i]);
  double argc=n+1;
#endif
#ifdef DEBUG
  float v[argc-1];
  for (i=0;i<argc-1;i++) v[i]=atof(argv[i+1]);
#endif
  for (i=0;i<argc-1;i++) soma+=v[i];
  printf("A soma é =%f\n",soma);
}
```

O operador `defined()` pode ser usado no teste da compilação condicional: `#ifdef DEBUG` é o mesmo que `#if defined(DEBUG)`. É possível trabalhar com mais de um símbolo, por exemplo, `#if defined(DEBUG1) && !defined(DEBUG2)`, que compila quando `DEBUG1` e `DEBUG2` não está.

11.3 assert()

É uma macro da biblioteca `assert.h` que entra um valor lógico, 0 ou 1. No caso de verdadeiro, não acontece nada. No caso falso, a função imprime o nome do arquivo fonte, a linha do arquivo contendo a chamada para a macro, o nome

da função que contém a chamada e o texto da expressão que foi avaliada.

Exemplo 11.3.1. Implemente um código que lê n números na linha de comando e imprime a soma. No caso de não entrar número, abortar a execução.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void main (int argc,char **argv)
{
  assert(argc-1);
  double soma=0;
  int i;
  for (i=1;i<argc;i++) soma+=atof(argv[i]);
  printf("A soma é =%f\n",soma);
}
```

11.4 Exercícios

E 11.4.1. Calcule as raízes da função $f(x) = x - \cos(x)$ usando os métodos de Newton e Secantes. Use `#ifdef - #endif` para programar as duas opções. Veja os algorítmos nos exercícios 3.4.4 e 3.4.5.

Capítulo 12

Variáveis globais, divisão do código e outros aspectos

12.1 Divisão do código em vários arquivos

É possível quebrar um código em vários arquivos. Basta compilar todos os códigos juntos.

Exemplo 12.1.1. Implemente um código resolver novamente o exemplo 6.6.2, que consiste em calcular a integral

$$\int_0^1 f(t)dt,$$

onde

$$f(t) = \int_0^t \text{sen}\,(t-\tau)e^{-\tau^2}d\tau.$$

Use um arquivo para o main e outro com as funções.

O primeiro arquivo foi salvo como `main.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
double Boole(double (*f)(double, double),double a,double b, int N,double *param);
double integrando(double tau, double t);
double f(double t,double tau);

void main(void)
{
  int N=5;
  double I1=0,I2=0,erro=0,a=0,b=1,tol=1e-10,param[1]={0};
```

```
  I1=Boole(f,a,b,N,param);
  int cont=0;
  do
  {
  cont++;
  N=2*N-1;
  I2=Boole(f,a,b,N,param);
  erro=fabs(I2-I1);
  I1=I2;
  }while (erro>tol);
  printf("O valor da integral é %f\n",I2);
}
```

O segundo arquivo foi salvo como `func.c`

```
#include <math.h>
double Boole(double (*f)(double, double),double a,double b, int N,double *param)
{
  int i;
  double h=(b-a)/(4*N),Int=0,t=param[0];

  Int=14./45.*(*f)(a,t);
  for(i=0;i<N;i++)
  {
    Int+=64./45.*(*f)(a+h,t);
    Int+=24./45.*(*f)(a+2*h,t);
    Int+=64./45.*(*f)(a+3*h,t);
    Int+=28./45.*(*f)(a+4*h,t);
    a+=4*h;
  }
  Int-=14./45.*(*f)(a,t);
return h*Int;
}
double integrando(double tau, double t)
{
  return sin(t-tau)*exp(-tau*tau);
}
double f(double t,double tau)
{
  int N=5;
  double I1=0,I2=0,erro=0,a=0,b=t,tol=1e-10,param[1]={t};
  I1=Boole(integrando,a,b,N,param);
```

```
  int cont=0;
  do
  {
  cont++;
  N=2*N-1;
  I2=Boole(integrando,a,b,N,param);
  erro=fabs(I2-I1);
  I1=I2;
  }while (erro>tol);
  return I2;
}
```

Compilamos com a linha de comando

```
gcc main.c func.c -lm
```

12.2 Makefile

Imagine que dividimos o trabalho em diversos arquivos e estamos usando várias bibliotecas externas que precisam aparecer explicitamente na linha de comando para compilação. Provavelmente, estaremos interessados em simplificar o uso da linha de comando que compila o trabalho, ou seja, construir um Makefile. Na mesma pasta onde salvamos os arquivos do exemplo 12.1.1, vamos salvar um arquivo com o nome `Makefile` (sem extensão) com o seguinte conteúdo:

```
main: main.c func.c
gcc main.c func.c -lm -oprog
```

Agora, na linha de comando, digite apenas

```
make
```

Execute o programa com a linha

```
./prog
```

12.3 Variáveis globais

As vezes estamos interessados em trabalhar com variáveis globais, que podem ser enxergadas por qualquer função do código sem precisar ser passada como parâmetro.

Exemplo 12.3.1. Implemente um código com duas funções, uma que ordena os pontos de um vetor de n posições e outra que coloca o valor da soma de todos os elementos de vetor de n posições na posição $n+1$. Defina um vetor como variável global.

```
#include <stdio.h>
#include <stdlib.h>

double *x;
int N;

void troca(double *a,double *b)
{
  double aux=*a;
  *a=*b;
  *b=aux;
}

void ordena()
{
  int i,ordena=1;

  do
  {
    for (ordena=i=0;i<N-1;i++)
      if (x[i]>x[i+1])
      {
      troca(&x[i],&x[i+1]);
      ordena=1;
      }
  }while (ordena);
}

void coloca_soma_na_ultima_posicao()
{
  double soma=0;
  int i;
  for (i=0;i<N;i++) soma+=x[i];
  N++;
  x=realloc(x,N*sizeof(double));
  x[N-1]=soma;
```

```
}

void main(int argc,char **argv)
{
  N=argc-1;
  if (N==0)
  {
  printf("Entre com alguns números na linha de comando");
  return;
  }
  int i;
  x=malloc(N*sizeof(double));
  for (i=0;i<N;i++)
  {
  x[i]=atof(argv[i+1]);
  printf("x[%d]=%f\n",i,x[i]);
  }
  ordena();
  for (i=0;i<N;i++) printf("x[%d]=%f\n",i,x[i]);
  coloca_soma_na_ultima_posicao();
  for (i=0;i<N;i++) printf("x[%d]=%f\n",i,x[i]);
  ordena();
  for (i=0;i<N;i++) printf("x[%d]=%f\n",i,x[i]);
  coloca_soma_na_ultima_posicao();
  for (i=0;i<N;i++) printf("x[%d]=%f\n",i,x[i]);
}
```

12.4 time

Uma forma de calcular o tempo de execução do programa é usar `time` na linha de comando:

```
time ./a.out
```

No entanto, se estamos interessados em estimar o tempo de processamento de um pedaço do código, podemos usar as funções da biblioteca `time.h`. A bilioteca possui os tipos específicos de variáveis para trabalhar com tempo, a saber, `time_t` e `clock_t`. Descrevemos duas das função aqui:

- `clock_t clock(void)` retorna o tempo de processamento desde o início do programa em uma unidade de processamento. Basta dividir por `CLOCKS_PER_SEC` para obter o valor em segundos.

- `time_t time(time_t *timer)` salva a hora atual no formato `time_t`.
- `double difftime(time_t time1, time_t time2)` retorna em segundos a diferença entre time1 e time2.

Exemplo 12.4.1. Implemente um código que imprime o tempo para calcular cem mil vezes logarítmo de um número.

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

void main(void)
{
  time_t tempo;
  tempo=time(NULL);
  double x;
  int i;
  for (i=0;i<100000;i++)
  {
    x=log(i+1);
    printf("%f\n",x);
  }
  printf("%d s\n",(int) (time(NULL)-tempo));
}
```

Alternativamente,

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

int main(void)
{
  int i;
  double x;
  time_t inicio_t, fim_t;

  time(&inicio_t);
  for (i=0;i<100000;i++)
  {
    x=log(i+1);
```

```
    printf("%f\n",x);
  }
  time(&fim_t);
  printf("Passaram-se %.0lf segundos\n", (double)difftime(fim_t, inicio_t));
  return 0;
}
```

12.5 Constante

Quando definimos uma constante, significa que não desejamos que a variável troque de valor ao longo da rotina. Nesse caso, usamos a palabra-chave `const` antes do tipo de variável. Observe o código abaixo:

```
#include <stdio.h>

void main(void)
{
  const int x=2;
  printf("%d\n",x);
  x=3;
  printf("%d\n",x);
}
```

que dá um erro de compilação

```
teste.c: In function ?main?:
teste.c:7:3: error: assignment of read-only variable ?x?
   x=3;
```

12.6 Exercícios

E 12.6.1. Resolva o exemplo 9.3.1 novamente, mas use uma variável global para salvar os valores de $F(x)$.

Capítulo 13

Bibliotecas de computação científica

13.1 GSL - quadraturas

A GSL (GNU Scientific Library) é uma biblioteca livre para C e C++ com milhares de rotinas matemáticas envolvendo funções especiais, mínimos quadrados, integração numéricas, autovalores e autovetores, etc. A página do projeto https://www.gnu.org/software/gsl/ descreve a documentação sobre instalação e uso do pacote e o manual https://www.gnu.org/software/gsl/doc/html/index.html detalha o uso das rotinas.

Vamos começar trabalhando com as rotinas de integração numérica: https://www.gnu.org/sof
O uso das rotinas de integração exigem a inclusão do pacote `gsl_integration.h`.

Uma das rotinas mais simplesde integração é a `gsl_integration_qng`. O nome da rotina tem um significado:

- `q` → quadrature routine
- `n` → non-adaptive integrator
- `g` → general integrand

A sintaxe dessa rotina é:

```
int gsl_integration_qng(const gsl_function * f, double a, double b, double epsa
double * result, double * abserr, size_t * neval)
```

onde

- `*f` é um ponteiro para função do tipo gsl (`gsl_function`). Essa função aceita passagem de parâmetros.

- `a` é o limite inferior de integração;
- `b` é o limite superior de integração;
- `epsabs` é a tolerância absoluta;
- `epsrel` é o tolerância relativa;
- `result` é o resultado da integral
- `abserr` estimativa do erro absoluto;
- `neval` é o número de vezes que a função foi calculada.

O esquema de integração é Gauss-Konrad com número fixo de pontos: 10, 21, 43 ou 87. Se a integral apresentar estimativa de erro grande, escolha outra rotina de integração.

Exemplo 13.1.1. Calcule a integral

$$\int_0^1 e^{-x^2} dx$$

usando a rotina `qng` da biblioteca gsl.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f(double x, void *params)
{
return exp(-x*x);
}

void main()
{
double params,result,abserr;
size_t neval;
gsl_function F;
F.function=&f;
F.params=&params;
gsl_integration_qng(&F, 0, 1, 1e-10, 1e-10,&result, &abserr, &neval);
printf("resultado=%f, erro=%e, n_calculos=%zu\n",result,abserr,neval);
}
```

Compile o código com a linha

```
gcc teste.c -lm -lgsl -lgslcblas
```

Uma função para ser integrada no gsl deve ter a formato `double f(double x, void *params)`, onde `x` é a variável de integração e `*params` é um ponteiro para uma lista de parâmetros da função. A definição da funçao do tipo gsl se dá pela linha `gsl_function F;`. Em seguida, usa-se a linha `F.function=&f;` para configurar a função gsl `F`, associando ao ponteiro para a função `f`. A linha `F.params=&params;` atribui o vetor `params` a lista de parâmetro da função `F`. No caso acima, nós definimos `double params;`, mas não atribuímos valor algum, pois a função não tem parâmetros.

Exemplo 13.1.2. Calcule a integral

$$\int_0^1 e^{-\lambda x^2} dx$$

usando $\lambda = .5$ e $\lambda = 2$.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f(double x, void *params)
{
double lambda=*((double *) params);
return exp(-lambda*x*x);
}

void main()
{
double params,result,abserr;
size_t neval;
gsl_function F;
F.function=&f;
F.params=&params;

params=.5;
gsl_integration_qng(&F, 0, 1, 1e-10, 1e-10,&result, &abserr, &neval);
printf("resultado=%f, erro=%f, n_calculos=%zu\n",result,abserr,neval);

params=2;
```

```
gsl_integration_qng(&F, 0, 1, 1e-10, 1e-10,&result, &abserr, &neval);
printf("resultado=%f, erro=%f, n_calculos=%zu\n",result,abserr,neval);
}
```

A rotina autoadaptativa mais simples é a `gsl_integration_qag`:

- `q` → quadrature routine
- `a` → adaptive integrator
- `g` → general integrand

Ela consiste em dividir o intervalo de integração ao meio e, depois, cada um dos subintervalos ao meio outra vez, até a convergência. A sintaxe é

```
int gsl_integration_qag(const gsl_function * f, double a, double b, double epsabs, do
size_t limit, int key, gsl_integration_workspace * workspace, double * result, double
```

onde

- `*f` é um ponteiro para função do tipo gsl (`gsl_function`). Essa função aceita passagem de parâmetros.
- `a` é o limite inferior de integração;
- `b` é o limite superior de integração;
- `epsabs` é a tolerância absoluta;
- `epsrel` é o tolerância relativa;
- `limit` é o máximo de subintervalos;
- `key` é a regra de integração

  ```
  GSL_INTEG_GAUSS15 (key = 1);
  GSL_INTEG_GAUSS21 (key = 2);
  GSL_INTEG_GAUSS31 (key = 3);
  GSL_INTEG_GAUSS41 (key = 4);
  GSL_INTEG_GAUSS51 (key = 5);
  GSL_INTEG_GAUSS61 (key = 6);
  ```

- `* workspace` é o espaço na memória para dividir o intervalo e recalcular a integral;

- `result` é o resultado da integral
- `abserr` estimativa do erro absoluto;

Exemplo 13.1.3. Calcule a integral

$$\int_0^1 \frac{1}{\mu} e^{-\frac{r}{\mu}} d\mu$$

para $r = 0.5$ e $r = 1$.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f(double mu, void *params)
{
double r=*((double *) params);
return exp(-r/mu)/mu;
}

void main()
{
double params,result,abserr;

gsl_integration_workspace * w = gsl_integration_workspace_alloc (1000);
gsl_function F;
F.function=&f;
F.params=&params;

params=.5;
gsl_integration_qag(&F, 0, 1, 1e-10, 1e-10, 1000,6,w ,&result,&abserr);
printf("resultado=%f, erro=%e \n",result,abserr);

params=1;
gsl_integration_qag(&F, 0, 1, 1e-10, 1e-10, 1000,6,w ,&result,&abserr);
printf("resultado=%f, erro=%e \n",result,abserr);

gsl_integration_workspace_free(w);
}
```

A função `gsl_integration_qags` é adequada para integrandos com singularidades e `gsl_integration_qagp` é adequada para integrandos com singularidades

conhecidas. Na sintaxe da primeira função não passa a regra de integração e, na segunda, passa os pontos onde a função é singular junto com os limites de integração. Observe a resolução do exemplo 13.1.3 com essas funções.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f(double mu, void *params)
{
double r=*((double *) params);
return exp(-r/mu)/mu;
}

void main()
{
double params,result,abserr,pontos[2]={0,1};

gsl_integration_workspace * w = gsl_integration_workspace_alloc (1000);
gsl_function F;
F.function=&f;
F.params=&params;

params=.5;
gsl_integration_qags(&F, 0, 1, 0, 1e-10, 1000,w ,&result,&abserr);
printf("resultado=%f, erro=%e \n",result,abserr);

params=1;
gsl_integration_qagp(&F,pontos,2, 0, 1e-10, 1000,w ,&result,&abserr);
printf("resultado=%f, erro=%e \n",result,abserr);

gsl_integration_workspace_free(w);
}
```

Agora, estudar as rotinas do gsl que aplicam a quadratura de Gauss-Legendre. As quatro funções têm as seguintes sintaxes:

```
gsl_integration_glfixed_table * gsl_integration_glfixed_table_alloc(size_t n)
```

e

```
void gsl_integration_glfixed_table_free(gsl_integration_glfixed_table * t)
```

A primeira função calcula as abscissas e pesos da quadratura de Gauss-Legendre com n pontos. A segunda libera o espaço alocado na memória. A terceira função tem a sintaxe

```
double gsl_integration_glfixed(const gsl_function * f, double a, double b, const
gsl_integration_glfixed_table * t)
```

onde o retorno da função é o valor da integral no intervalo $[a,b]$ usando os pesos e abscissas salvos na tabela `t`. A última função tem sintaxe

```
int gsl_integration_glfixed_point(double a, double b, size_t i, double * xi, dou
gsl_integration_glfixed_table * t)
```

onde

- `i`, `i=0,1,2,...n-1` é o índice que indica o i-ésimo ponto;
- `xi` é a i-ésima abscissa;
- `wi` é o i-ésima peso;
- `gsl_integration_glfixed_table * t` é a tabela com os pesos e abscissas.

Exemplo 13.1.4. Calcule a integral

$$\int_0^1 e^{-\lambda x^2} dx$$

usando a quadratura de Gauss-Legendre.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f(double mu, void *params)
{
double r=*((double *) params);
return exp(-r/mu)/mu;
}

void main()
{
  int N=200;
  gsl_integration_glfixed_table * table;
  table = gsl_integration_glfixed_table_alloc (N);
```

```
  double result,params;

  gsl_function F;
  F.function=&f;
  F.params=&params;

  params=.5;
  result=gsl_integration_glfixed(&F, 0, 1, table);
  printf("O valor da integral é %f\n",result);

  gsl_integration_glfixed_table_free(table);
}
```

Alternativamente, podemos salvar os pesos e abscissas em vetores integrar usando o somatório

$$\sum_{i=0}^{n-1} w_i f(x_i).$$

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f(double mu, void *params)
{
double r=*((double *) params);
return exp(-r/mu)/mu;
}

void main()
{
  int N=200,flag;
  gsl_integration_glfixed_table * table;
  table = gsl_integration_glfixed_table_alloc (N);
  double result=0,a[N],p[N],params=.5;
  size_t i;
  for (i=0; i<N;i++)
  {
    flag = gsl_integration_glfixed_point (0, 1, i, &a[i], &p[i], table);
    printf("%zu %f %f\n", i, a[i], p[i]);
  }
  for (i=0; i<N;i++) result+=p[i]*f(a[i],&params);
```

```
  printf("O valor da integral é %f\n",result);

  gsl_integration_glfixed_table_free(table);
}
```

13.2 Problema de transporte

Exemplo 13.2.1. Resolva o problema dado pela equação estacionária do transporte unidimensional com espalhamento isotrópico:

$$\begin{aligned} \mu\frac{\partial I}{\partial y} + \lambda I = \frac{\sigma}{2}\int_{-1}^{1} I(y,\mu)d\mu + Q(y,\mu), \quad & y \in (0,L),\ \mu \in [-1,1], \\ I(0,\mu) = B_0, & \mu > 0, \\ I(L,\mu) = B_L, & \mu < 0. \end{aligned}$$

Nesse modelo, $I(y,\mu)$ é uma intensidade radiativa,

$$\mathcal{I}(y) = \frac{1}{2}\int_{-1}^{1} I(y,\mu)d\mu$$

é o fluxo escalar, λ é o coeficiente de absorção, σ é o coeficiente de espalhamento, $Q(y,\mu)$ é uma fonte, B_0 e B_L são os fluxos de entrada na fronteira, $y \in [0,L]$ é a variável espacial e $\mu \in [-1,1]$ é o cosseno do ângulo entre o raio e o eixo y.

Começamos discretizando à variável angular μ com um método chamado de Ordenadas Discretas. Escolhemos a quadratura numérica de Gauss-Legendre com um número par de abscissas μ_i e pesos ω_i e discretizamos a equação de forma a transformar a integral em um somatório. Este procedimento transforma o problema em um sistema de EDO's na variável y:

$$\begin{aligned} \mu_i\frac{dI_i}{dy} + \lambda I_i = \frac{\sigma}{2}\sum_{j=-M}^{M} \omega_j I_j(y) + Q_i, \quad & y \in (0,L), \\ I_i(0) = B_0, & \mu_i > 0, \\ I_i(L) = B_L, & \mu_i < 0, \end{aligned}$$

onde $i = -M,...,M$. Agora, separaramos o sistema de EDO's em dois: um para $\mu_i < 0$ e outro para $\mu_i > 0$. Portanto,

$$\begin{aligned} -\mu_i\frac{dI_{-i}}{dy} + \lambda I_{-i} = \frac{\sigma}{2}\sum_{j=1}^{M} \omega_j (I_{-j} + I_j) + Q_{-i}, \quad & i = 1,2,...,M, \\ I_{-i}(L) = B_L, & i = 1,2,...,M, \end{aligned}$$

e

$$\mu_i \frac{dI_i}{dy} + \lambda I_i = \frac{\sigma}{2} \sum_{j=1}^{M} \omega_j (I_{-j} + I_j) + Q_i, \quad i = 1,2,...,M,$$
$$I_i(0) = B_0, \qquad i = 1,2,...,M.$$

Agora, vamos discretizar à variável espacial y. Dividimos o intervalo $[0,L]$ em N subintervalos de tamanho $h = \frac{L}{N}$, e construímos a malha $y_k = (k-1)h$, $k = 1,2,...,N+1$. Suponhamos que em um intervalo da malha $[y_k,y_{k+1}]$ o lado direito das equações seja constante e conhecido. Vamos denotar a aproximação para o lado direito da equação de $S_i^{k+1/2}$. Logo, neste intervalo temos:

$$\frac{dI_{-i}}{dy} - \frac{\lambda}{\mu_i} I_{-i} = -\frac{1}{\mu_i} S_{-i}^{k+1/2}, \quad i = 1,2,...,M, \quad y_k \leq y \leq y_{k+1},$$
$$\frac{dI_i}{dy} + \frac{\lambda}{\mu_i} I_i = \frac{1}{\mu_i} S_i^{k+1/2}, \qquad i = 1,2,...,M, \quad y_k \leq y \leq y_{k+1}.$$

As duas equações podem ser resolvidas pelo método do fator integrante:

$$e^{-\frac{\lambda}{\mu_i} y_{k+1}} I_{-i}^{k+1} - e^{-\frac{\lambda}{\mu_i} y_k} I_{-i}^{k} = \frac{S_{-i}^{k+1/2}}{\lambda} \left(e^{-\frac{\lambda}{\mu_i} y_{k+1}} - e^{-\frac{\lambda}{\mu_i} y_k} \right), \quad i = 1,2,...,M, \quad k = 1,2,...,N,$$
$$e^{\frac{\lambda}{\mu_i} y_{k+1}} I_{i}^{k+1} - e^{\frac{\lambda}{\mu_i} y_k} I_{i}^{k} = \frac{S_{i}^{k+1/2}}{\lambda} \left(e^{\frac{\lambda}{\mu_i} y_{k+1}} - e^{\frac{\lambda}{\mu_i} y_k} \right), \qquad i = 1,2,...,M, \quad k = 1,2,...,N.$$

As condições de contorno nos dão:

$$I_{-i}^{N+1} = B_L, \quad i = 1,2,...,M,$$
$$I_i^1 = B_0, \quad i = 1,2,...,M.$$

Logo, podemos obter I_{-i}^k e I_i^{k+1} de maneira recursiva, ou seja,

$$I_{-i}^{k} = e^{\frac{\lambda}{\mu_i}(y_k - y_{k+1})} I_{-i}^{k+1} + \frac{S_{-i}^{k+1/2}}{\lambda} \left(1 - e^{\frac{\lambda}{\mu_i}(y_k - y_{k+1})} \right), \quad i = 1,2,...,M, \quad k = N,N-1,...,1,$$
$$I_{i}^{k+1} = e^{\frac{\lambda}{\mu_i}(y_k - y_{k+1})} I_{i}^{k} + \frac{S_{i}^{k+1/2}}{\lambda} \left(1 - e^{\frac{\lambda}{\mu_i}(y_k - y_{k+1})} \right), \qquad i = 1,2,...,M, \quad k = 1,2,...,N.$$

Damos um chute inicial para a solução e calculamos $S_i^{k+1/2}$ com as seguintes médias:

$$S_{\pm i}^{k+1/2} = \frac{1}{2} \left[\frac{\sigma}{2} \sum_{j=1}^{M} \omega_j (I_{-j}^k + I_j^k) + Q_{\pm i}^k + \frac{\sigma}{2} \sum_{j=1}^{M} \omega_j (I_{-j}^{k+1} + I_j^{k+1}) + Q_{\pm i}^{k+1} \right].$$

A solução aproximada é obtida quando houver convergência do processo iterativo: Calculamos $S_{\pm i}^{k+1/2} = 0$, depois I_{-i}^{N+1}, I_{-i}^{N}, I_{-i}^{N-1}, $\cdots$, I_{-i}^{1} e I_{i}^{1},I_{i}^{2}, $\cdots$, I_{i}^{N+1}. Depois recalculamos $S_{\pm i}^{k+1/2}$ e $I_{\pm i}^{k}$.

Para implementar esse algoritmo, fazemos algumas considerações:

- É necessário entender o algoritmo antes de começar a programar. Embora seja uma questão óbvia, a ansiedade pode nos trair.
- É importante pensar na estrutura do código antes de programar: tipo das principais variáveis, formas de armazenamento (ponteiros, matrizes), variáveis globais, estruturas, funções, pacotes, etc.
- Não é indicado programar tudo de uma só vez. Faça por pedaços e valide cada função independentemente.
- Compile o código seguidamente para não acumular erros de compilação.

Começamos implemetando uma função que calcula pesos e abscissas para a quadratura de Gauss-Legendre:

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>
#define M 10

double *x,*w;

double GL()
{
  size_t i,flag;
  gsl_integration_glfixed_table * table;
  table = gsl_integration_glfixed_table_alloc (M);
  for (i=0; i<M;i++)
  {
    flag = gsl_integration_glfixed_point (-1, 1, i, &x[i], &w[i], table);
    //printf("flag=%zu, %zu %f %f\n",flag, i, x[i], w[i]);
  }
  gsl_integration_glfixed_table_free(table);
}
double teste(double x)
{
  return x*x;
}
```

```
void main()
{
  int i;
  if (((x=malloc(sizeof(double)*M))==NULL)||((w=malloc(sizeof(double)*M))==NULL))
  {
  printf("Erro ao alocar memória");
  return;
  }
  GL(x,w);
  double result=0;
  for (i=0; i<M;i++) result+=w[i]*teste(x[i]);
  printf("O valor da integral é %f\n",result);
}
```

Depois de testar essa rotina integrando várias funções, vamos um passo adiante. Definimos duas variáveis globais para salvar pesos e abscissas e alocamos espaço para salvar a solução numa variável tipo ponteiro para ponteiro.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>
#define M 10
#define N 10

double *x,*w;

double GL()
{
  size_t i,flag;
  gsl_integration_glfixed_table * table;
  table = gsl_integration_glfixed_table_alloc (2*M);
  for (i=0; i<2*M;i++)
  {
    flag = gsl_integration_glfixed_point (-1, 1, i, &x[i], &w[i], table);
    //printf("flag=%zu, %zu %f %f\n",flag, i, x[i], w[i]);
  }
  gsl_integration_glfixed_table_free(table);
}

void main()
{
  int i;
```

```
  double **I;
  if (((x=malloc(sizeof(double)*2*M))==NULL)||((w=malloc(sizeof(double)*2*M))==N
  {
  printf("Erro ao alocar memória");
  return;
  }
  if ((I = calloc(N+1,sizeof(double)))==NULL)
  {
  printf("Erro ao alocar memória");
  return;
  }
  for (i=0;i<N+1;i++) if ((I[i] = calloc(2*M,sizeof(double)))==NULL)

  {
  printf("Erro ao alocar memória");
  return;
  }
  GL(x,w);
}
```

Agora, implementamos a primeira versão do código que resolve o problema:

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>
#define M 1000
#define N 2000

const double L=1;
const double B0=1;
const double BL=1;
const double sigma=1;
const double lambda=1.0;
double *mu,*w,*y;

double GL()
{
  size_t i,flag;
  gsl_integration_glfixed_table * table;
  table = gsl_integration_glfixed_table_alloc (2*M);
  for (i=M; i<2*M;i++)
  {
```

```
    flag = gsl_integration_glfixed_point (-1, 1, i, &mu[i-M], &w[i-M], table);
    //printf("flag=%zu, %zu %f %f\n",flag, i, mu[i], w[i]);
  }
  gsl_integration_glfixed_table_free(table);
}
double Q(double x)
{
 return exp(-x);
}
void main(void)
{

  int i,k,n;
  double **I,*II;
  if (((mu=malloc(sizeof(double)*M))==NULL)||((w=malloc(sizeof(double)*M))==NULL))
  {
  printf("Erro ao alocar memória");
  return;
  }
  GL();
  if ((y=malloc(sizeof(double)*(N+1)))==NULL)
  {
  printf("Erro ao alocar memória");
  return;
  }
  if ((II=malloc(sizeof(double)*(N+1)))==NULL)
  {
  printf("Erro ao alocar memória");
  return;
  }
  if ((I = calloc(N+1,sizeof(double)))==NULL)
  {
  printf("Erro ao alocar memória");
  return;
  }
  for (i=0;i<N+1;i++) if ((I[i] = calloc(2*M,sizeof(double)))==NULL)
  {
  printf("Erro ao alocar memória");
  return;
  }
```

```
//malha espacial
  double h = L/N;
  for (i=0;i<N+1;i++) y[i] = i*h;
//Contorno
  for (i=0;i<M;i++)
  {
    I[0][i] = B0;

    I[N][M+i] =BL;
  }
  //iteracao
  double Exp,S=0;
  for (n=0;n<200;n++)
  {
    //Fluxo Escalar
    for(k=0;k<N+1;k++)
    {
      II[k]=0;
      for(i=0;i<M;i++)
      {
        II[k] += w[i]*(I[k][i]+I[k][M+i]);
      }
      if (k%200==0) printf("II(%d)=%f\n",k,II[k]);
    }

    //Recorrência
    for (i=0;i<M;i++)
    {
      for(k=0;k<N;k++)
      {
        S = 1/2.0*(sigma/2.0*(II[k]+II[k+1])+Q(y[k])+Q(y[k+1]));
        Exp = exp(lambda/mu[i]*(y[k]-y[k+1]));

        I[k+1][i] = S/lambda*(1-Exp)+I[k][i]*Exp;
      }

      for(k=N-1;k>-1;k--)
      {
        S = 1/2.0*(sigma/2.0*(II[k]+II[k+1])+Q(y[k])+Q(y[k+1]));
        Exp = exp(lambda/mu[i]*(y[k]-y[k+1]));
```

```
        I[k][M+i] = S/lambda*(1-Exp)+I[k+1][M+i]*Exp;

      }

    }
  }
}
```

Para validar usamos resultados de artigos científicos, que fornecem dados tais como dessa tabela:

Tabela 13.1: Valores calculados para $\int_{-1}^{1} I(y,\mu)d\mu$ quando $Q(y) = e^{-y}$, $\lambda' = \sigma' = 1$ e $B_0 = B_L = 1$, com $L = 1$.

y	0.0	0.2	0.4	0.6	0.8	1.0
GFD_{800}	3.514742	4.193467	4.307001	4.162773	3.820960	3.196349
DD & Gauss-Legendre	3.514748	4.193470	4.307004	4.162776	3.820963	3.196353
DD & Clenshaw-Curtis	3.514748	4.193471	4.307005	4.162777	3.820963	3.196354

Naturalmente, o código pode ser melhorado. Por exemplo, podemos estabelecer um critério de convergência para o processo iterativo.

13.3 GSL - matrizes e vetores

Para utilizar os pacotes de algebra linear, é necessário armazenar os vetores e matrizes no formato gsl. Portanto, começamos introduzindo as funções do gsl para manipulação de vetores e matrizes que estão nos pacotes `gsl_vector.h` e `gsl_vector.h`, respectivamente.

- `gsl_vector * gsl_vector_alloc(size_t n)` aloca um vetor de tamanho n e retorna um ponteira para essa estrutura;
- `gsl_vector * gsl_vector_calloc(size_t n)` aloca um vetor de tamanho n com todas posições nulas;
- `void gsl_vector_free(gsl_vector * v)` libera espaço na memória;
- `double gsl_vector_get(const gsl_vector * v, const size_t i)` retorna o valor de $v[i]$;

- `void gsl_vector_set(gsl_vector * v, const size_t i, double x)` atribui $v[i] = x$;
- `void gsl_vector_set_all(gsl_vector * v, double x)` atribui $v[i] = x\ \forall i$;
- `void gsl_vector_set_zero(gsl_vector * v)` atribui $v[i] = 0\ \forall i$.
- `int gsl_vector_memcpy(gsl_vector * dest, const gsl_vector * src)` copia o o vetor `src` para o vetor `dest`;
- `gsl_vector_add(gsl_vector * a, const gsl_vector * b)` $a_i \leftarrow a_i + b_i\ \forall i$;
- `gsl_vector_sub(gsl_vector * a, const gsl_vector * b)` $a_i \leftarrow a_i - b_i\ \forall i$;
- `gsl_vector_div(gsl_vector * a, const gsl_vector * b)` $a_i \leftarrow a_i / b_i\ \forall i$;
- `gsl_vector_mul(gsl_vector * a, const gsl_vector * b)` $a_i \leftarrow a_i * b_i\ \forall i$;
- `double gsl_vector_max(const gsl_vector * v)` retorna o máximo de v.
- `gsl_matrix * gsl_matrix_alloc(size_t n1, size_t n2)` aloca matriz $n1 \times n2$;
- `gsl_matrix * gsl_matrix_calloc(size_t n1, size_t n2)` aloca matriz $n1 \times n2$ e atribui zero para todas as posições;
- `void gsl_matrix_free(gsl_matrix * m)` libera memoria;
- `double gsl_matrix_get(const gsl_matrix * m, const size_t i, const size_t j)` retorna $m[i,j]$;
- `void gsl_matrix_set(gsl_matrix * m, const size_t i, const size_t j, double` atribui $m[i,j] = x$;
- $\vdots$

A lista acima mostra apenas algumas funções das dezenas que existem. As demais podem ser acessadas na documentação do gsl https://www.gnu.org/software/gsl/doc/html/vecto

Exemplo 13.3.1. Vamos definir dois vetores e duas matrizes e fazer algumas operações.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
```

```
void main(void)
{
  int i,j;
  gsl_vector *v,*w;
  gsl_matrix *A,*B;
  v=gsl_vector_alloc(3);
  w=gsl_vector_calloc(3);
  A=gsl_matrix_alloc(3,3);
  B=gsl_matrix_calloc(3,3);
  gsl_vector_set_zero(v);
  for (i=0;i<3;i++) printf("v[%d]=%f\n",i,gsl_vector_get(v,i));
  printf("\n");
  for (i=0;i<3;i++) printf("v[%d]=%f\n",i,gsl_vector_get(v,i));
  printf("\n");
  for (i=0;i<3;i++) for (j=0;j<3;j++) gsl_matrix_set(A, i,j, i*j);
  for (i=0;i<3;i++) for (j=0;j<3;j++) gsl_matrix_set(B, i,j, exp(-i*j));
  for (i=0;i<3;i++)
  {
    for (j=0;j<3;j++)
    {
      printf("A[%d,%d]=%f  ",i,j,gsl_matrix_get(A,i,j));
    }
    printf("\n");
  }
  printf("\n");
  for (i=0;i<3;i++)
  {
    for (j=0;j<3;j++)
    {
      printf("B[%d,%d]=%f  ",i,j,gsl_matrix_get(B,i,j));
    }
    printf("\n");
  }
  gsl_matrix_add(A, B);
  for (i=0;i<3;i++)
  {
    for (j=0;j<3;j++)
    {
      printf("A[%d,%d] + B[%d,%d]=%f  ",i,j,i,j,gsl_matrix_get(A,i,j));
```

```
    }
    printf("\n");
  }
}
```

Exemplo 13.3.2. Vamos voltar ao exemplo 4.1.3 e resolver o sistema linear

$$\begin{aligned} 3x + y + z &= 1 \\ -x - 4y + 2z &= 2 \\ -2x + 2y + 5z &= 3 \end{aligned}$$

usando o método de Jacobi. No entanto, vamos utilizar a estrura de vetores do gsl.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_vector.h>

double norma_dif_2(gsl_vector *x,gsl_vector *y)
{
  int i;
  gsl_vector_sub(x, y);
  double norma2=0;
  for (i=0;i<3;i++) norma2+=(gsl_vector_get(x,i))*(gsl_vector_get(x,i));
  return sqrt(norma2);
}
void Iteracao(gsl_vector *x_antigo,gsl_vector *x_atual)
{
  gsl_vector_set(x_atual,0,(1-gsl_vector_get(x_antigo,1)-gsl_vector_get(x_antigo
  gsl_vector_set(x_atual,1,(-2-gsl_vector_get(x_antigo,0)+2*gsl_vector_get(x_ant
  gsl_vector_set(x_atual,2,(3+2*gsl_vector_get(x_antigo,0)-2*gsl_vector_get(x_an
}

void Jacobi(double tolerancia,gsl_vector *x_atual)
{
  gsl_vector *x_antigo=gsl_vector_alloc(3);
  int i,controle=3;
  //chute inicial
  gsl_vector_set_zero(x_antigo);
//iteração
  while (controle)
```

```
  {
    Iteracao(x_antigo,x_atual);
    if (norma_dif_2(x_antigo,x_atual)<tolerancia) controle--;
    else controle=3;
    gsl_vector_memcpy(x_antigo,x_atual);
    printf("x=%f, y=%f, z=%f\n",gsl_vector_get(x_atual,0),gsl_vector_get(x_atual,1),g
  }
  gsl_vector_free(x_antigo);
  return;
}
main (void)
{
  int i;
  gsl_vector *solucao;
  solucao = gsl_vector_alloc(3);
  double tolerancia=1e-3;
  Jacobi(tolerancia,solucao);
  for (i=0;i<3;i++)     printf("x[%d] = %f\n",i,gsl_vector_get(solucao,i));
  gsl_vector_free(solucao);
}
```

13.4 GSL - álgebra linear

O gsl resolve problemas em álgebra linear por diversos métodos. Aqui nós vamos trabalhar apenas com a decomposição QR e sistemas tridiagonais. O pacote para álgebra linear é `gsl_linalg.h`.

- `int gsl_linalg_QR_decomp(gsl_matrix * A, gsl_vector * tau)`: calcula a decomposição QR da matriz A.
- `int gsl_linalg_QR_solve(const gsl_matrix * QR, const gsl_vector * tau, const gsl_` resolve o $Ax = b$ usando a decomposição QR.

Exemplo 13.4.1. Implemente um código para resolver o sistema

$$\begin{aligned} 3x + y + z &= 1 \\ -x - 4y + 2z &= 2 \\ -2x + 2y + 5z &= 3 \end{aligned}$$

usando decomposição QR da biblioteca gsl.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_linalg.h>

main (void)
{
  gsl_vector *tau=gsl_vector_alloc(3);
  gsl_vector *b=gsl_vector_alloc(3);
  gsl_vector *x=gsl_vector_alloc(3);
  gsl_matrix *A=gsl_matrix_alloc(3,3);;
  gsl_matrix_set(A,0,0,3);
  gsl_matrix_set(A,0,1,1);
  gsl_matrix_set(A,0,2,1);
  gsl_matrix_set(A,1,0,-1);
  gsl_matrix_set(A,1,1,-4);
  gsl_matrix_set(A,1,2,2);
  gsl_matrix_set(A,2,0,-2);
  gsl_matrix_set(A,2,1,2);
  gsl_matrix_set(A,2,2,5);
  gsl_vector_set(b,0,1);
  gsl_vector_set(b,1,2);
  gsl_vector_set(b,2,3);
  int i,j;
  for (i=0;i<3;i++)
  {
    for (j=0;j<3;j++) printf("A[%d,%d] = %f ",i,j,gsl_matrix_get(A,i,j));
  printf("\n");
  }
  for (i=0;i<3;i++) printf("b[%d] = %f\n",i,gsl_vector_get(b,i));
  gsl_linalg_QR_decomp(A, tau);
  gsl_linalg_QR_solve(A, tau,b,x);
  for (i=0;i<3;i++) printf("solucao[%d] = %f\n",i,gsl_vector_get(x,i));
}
```

O comando

- `int gsl_linalg_QR_lssolve(const gsl_matrix * QR, const gsl_vector * tau, co` resolve sistema impossível (quando possui mais linhas independes que colunas) pelo critério de mínimos quadrados.

Exemplo 13.4.2. Implemente um código para calcular os coeficientes de uma reta que melhor se ajusta aos pontos: (0,1),(1,0.5), (2, − 0.2), (3, − 0.7).

Supondo que a reta tenha equação $y = ax + b$, temos

$$\begin{aligned} b &= 1 && (13.1)\\ a+b &= 0.5 && (13.2)\\ 2a+b &= -0.2 && (13.3)\\ 3a+b &= -0.7 && (13.4) \end{aligned}$$

um sistema que pode ser resolvido pelo critério dos mínimos quadrados. Na forma matricial, temos

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 \\ 0.5 \\ -0.2 \\ -0.7 \end{bmatrix}.$$

Vamos resolver usando a decomposição QR.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_linalg.h>

main (void)
{
  gsl_vector *tau=gsl_vector_alloc(2);
  gsl_vector *b=gsl_vector_alloc(4);
  gsl_vector *x=gsl_vector_alloc(2);
  gsl_vector *residual=gsl_vector_alloc(4);
  gsl_matrix *A=gsl_matrix_alloc(4,2);;
  gsl_matrix_set(A,0,0,0);
  gsl_matrix_set(A,0,1,1);
  gsl_matrix_set(A,1,0,1);
  gsl_matrix_set(A,1,1,1);
  gsl_matrix_set(A,2,0,2);
  gsl_matrix_set(A,2,1,1);
  gsl_matrix_set(A,3,0,3);
  gsl_matrix_set(A,3,1,1);
```

```
  gsl_vector_set(b,0,1);
  gsl_vector_set(b,1,0.5);
  gsl_vector_set(b,2,-0.2);
  gsl_vector_set(b,3,-0.7);

  int i,j;
  for (i=0;i<4;i++)
  {
    for (j=0;j<2;j++) printf("A[%d,%d] = %f ",i,j,gsl_matrix_get(A,i,j));
  printf("\n");
  }
  for (i=0;i<4;i++) printf("b[%d] = %f\n",i,gsl_vector_get(b,i));
  gsl_linalg_QR_decomp(A, tau);
  gsl_linalg_QR_lssolve(A, tau,b,x,residual);
  for (i=0;i<2;i++) printf("solucao[%d] = %f\n",i,gsl_vector_get(x,i));
  for (i=0;i<4;i++) printf("residuo[%d] = %f\n",i,gsl_vector_get(residual,i));
}
```

A função

- `int gsl_linalg_solve_tridiag(const gsl_vector * diag, const gsl_vector * e,` resolve o sistema tridiagonal

$$\begin{bmatrix} d_0 & e_0 & 0 & 0 & \cdots & 0 \\ f_0 & d_1 & e_1 & 0 & \cdots & 0 \\ 0 & f_1 & d_2 & e_2 & \cdots & 0 \\ \vdots & & & \vdots & \ddots & \\ 0 & \cdots & 0 & f_{n-2} & d_{n-1} & e_{n-1} \\ 0 & \cdots & 0 & 0 & f_{n-1} & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}.$$

Exemplo 13.4.3. Vamos resolver novamente o problema do exemplo 6.3.2 usando a rotina gsl para sistemas tridiagonais.

O PVC do problema

$$\begin{cases} -u_{xx} + u_x = 200e^{-(x-1)^2}, \ \ 0 < x < 1. \\ 15u(0) + u'(0) = 500 \\ 10u(1) + u'(1) = 1 \end{cases}$$

possui a seguinte versão discreta

$$\begin{bmatrix} 15-\frac{1}{h} & \frac{1}{h} & 0 & 0 & 0 & \cdots & 0 \\ -\frac{1}{h^2}-\frac{1}{2h} & \frac{2}{h^2} & -\frac{1}{h^2}+\frac{1}{2h} & 0 & 0 & \cdots & 0 \\ 0 & -\frac{1}{h^2}-\frac{1}{2h} & \frac{2}{h^2} & -\frac{1}{h^2}+\frac{1}{2h} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots \\ 0 & \cdots & 0 & 0 & -\frac{1}{h^2}-\frac{1}{2h} & \frac{2}{h^2} & -\frac{1}{h^2}+\frac{1}{2h} \\ 0 & \cdots & 0 & 0 & 0 & -\frac{1}{h} & 10+\frac{1}{h} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix} = \begin{bmatrix} 500 \\ 200e^{-(x_2-1)^2} \\ 200e^{-(x_3-1)^2} \\ \vdots \\ 200e^{-(x_{N-1}-1} \\ 1 \end{bmatrix}$$

onde $x_i = h(i-1)$, $i = 1,2,...,N$, $h = \frac{1}{N-1}$.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_linalg.h>
#define N 10

main (void)
{
  gsl_vector *d=gsl_vector_alloc(N);
  gsl_vector *f=gsl_vector_alloc(N-1);
  gsl_vector *e=gsl_vector_alloc(N-1);
  gsl_vector *sol=gsl_vector_alloc(N);
  gsl_vector *b=gsl_vector_alloc(N);
  double h=1./(N-1);
  int i,j;
  //malha
  gsl_vector *x=gsl_vector_alloc(N);
  for (i=0;i<N;i++) gsl_vector_set(x,i,i*h);
  for (i=0;i<N;i++) printf("x[%d] = %f\n",i,gsl_vector_get(x,i));

  //sistema
  gsl_vector_set(d,0,15-1/h);
  gsl_vector_set(e,0,1/h);
  gsl_vector_set(b,0,500);
  for (i=1;i<N-1;i++)
  {
    gsl_vector_set(d,i,2/(h*h));
```

```
    gsl_vector_set(e,i,-1/(h*h)+1/(2*h));
    gsl_vector_set(f,i-1,-1/(h*h)-1/(2*h));
    gsl_vector_set(b,0,200*exp(-(gsl_vector_get(x,i)-1)*(gsl_vector_get(x,i)-1)
  }
  gsl_vector_set(d,N-1,10+1/h);
  gsl_vector_set(f,N-2,-1/h);
  gsl_linalg_solve_tridiag(d,e,f,b,sol);
  for (i=0;i<N;i++) printf("solucao[%d] = %f\n",i,gsl_vector_get(sol,i));
}
```

13.5 Exercícios

E 13.5.1. Implemente um código para resolver o problema do exemplo 6.6.2 usando as rotinas do gsl.

E 13.5.2. Implemente um código para resolver o problema do exercício 9.4.2 usando as rotinas do gsl.

E 13.5.3. Resolva o problema dado pela equação estacionária do transporte unidimensional com espalhamento isotrópico e condições de contorno semi-reflectivas:

$$\begin{aligned}
\mu\frac{\partial I}{\partial y}+\lambda I &= \frac{\sigma}{2}\int_{-1}^{1} I(y,\mu)d\mu + Q(y,\mu), && y\in(0,L),\ \mu\in[-1,1],\\
I(0,\mu) &= \rho_0(\mu)I(0,-\mu)+(1-\rho_0(\mu))B_0(\mu), && \mu>0,\\
I(L,\mu) &= \rho_L(\mu)I(L,-\mu)+(1-\rho_L(\mu))B_L(\mu), && \mu<0.
\end{aligned}$$

Aqui, ρ_0 e ρ_L são índices de reflexão. Use as quadraturas de Boole e Gauss-Legendre.

Dica: Use a mesma discretização do exemplo 13.2.1 adicionado a discretização

$$\begin{aligned}
I_{-i}^{N+1} &= \rho_{L_{-i}}I_i^{N+1}+(1-\rho_{L_{-i}})B_L, && i=1,2,...,M,\\
I_i^1 &= \rho_{0_i}I_{-i}^1+(1-\rho_{0_i})B_0, && i=1,2,...,M.
\end{aligned}$$

na fonteira.

E 13.5.4. Refaça os exercícios 4.4.1 e 4.4.3 usando as estruturas do gsl para vetores.

E 13.5.5. Resolva o problema do exercício 6.7.7 usando o método de Newton e as rotinas da bilbioteca gsl.

E 13.5.6. Resolva o problema do exercício 6.7.8 usando o método de diferenças finitas, o método de Newton e as rotinas da bilbioteca gsl.

E 13.5.7. Resolva numericamente o problema dado pela equação do calor evolutiva

$$\begin{aligned} \frac{\partial u}{\partial t} &= \mu \frac{\partial^2 u}{\partial x^2}, \qquad 0 < x < 1,\, t > 0 \\ u(x,0) &= 10e^{-x} \\ u(0,t) &= 1 \\ u(1,t) &= 2 \end{aligned}$$

Dica: Use o método de Crank-Nicolson para a discretização temporal:

$$\begin{aligned} \frac{u_i^{k+1} - u_i^k}{h_t} &= \mu \frac{u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1}}{2h_x^2} + \mu \frac{u_{i+1}^k - 2u_i^k + u_{i-1}^k}{2h_x^2} \\ u_i^0 &= 10e^{-x_i} \\ u_0^k &= 1 \\ u_N^k &= 2, \end{aligned}$$

onde

- $u_i^k \approx u(x_i,t_k)$ é uma aproximação para a solução no tempo t_k e ponto x_i.
- x_i é uma malha espacial.
- h_t é incremento temporal.
- h_x é o espaçamento da malha espacial.

Observe que teremos que resolver um sistema linear a cada passo de tempo:

$$\begin{aligned} -\frac{\mu h_t}{2h_x^2} u_{i+1}^{k+1} + \left(1 + \frac{\mu h_t}{h_x^2}\right) u_i^{k+1} - \frac{\mu h_t}{2h_x^2} u_{i-1}^{k+1} &= u_i^k + \frac{\mu h_t}{2h_x^2}\left(u_{i+1}^k - 2u_i^k + u_{i-1}^k\right) \\ u_i^0 &= 10e^{-x_i} \\ u_0^k &= 1 \\ u_N^k &= 2, \end{aligned}$$

E 13.5.8. Resolva numericamente o problema dado pela equação do calor evolutiva

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{\partial}{\partial x}\left(\mu(x)\frac{\partial u}{\partial x}\right), \qquad 0 < x < 1,\ t > 0 \\ u(x,0) &= 10e^{-x} \\ u(0,t) &= 1 \\ u(1,t) &= 2 \end{aligned}$$

onde $\mu(x) = 0.5 - 0.2x$. Use a seguinte discretização:

$$\begin{aligned} \frac{u_i^{k+1} - u_i^k}{h_t} &= \frac{\mu_{i+1/2}\left(u_{i+1}^k - u_i^k\right) - \mu_{i-1/2}\left(u_i^k - u_{i-1}^k\right)}{2h_x^2} + \\ &+ \frac{\mu_{i+1/2}\left(u_{i+1}^{k+1} - u_i^{k+1}\right) - \mu_{i-1/2}\left(u_i^{k+1} - u_{i-1}^{k+1}\right)}{2h_x^2} \\ u_i^0 &= 10e^{-x_i} \\ u_0^k &= 1 \\ u_N^k &= 2, \end{aligned}$$

onde

- $u_i^k \approx u(x_i,t_k)$ é uma aproximação para a solução no tempo t_k e ponto x_i.
- x_i é uma malha espacial.
- h_t é incremento temporal.
- h_x é o espaçamento da malha espacial.
- $\mu_{i+1/2} = \frac{\mu(x_{i+1})+\mu(x_i)}{2}$ é o valor aproximado da difusão no intervalo $[x_i,x_{i+1}]$.

Capítulo 14

Introdução à programação orientada a objetos

Neste capítulo discutiremos os objetos e as classes de objetos. Objetos são semelhantes às estruturas estudadas no capítulo 9 e provêm muito mais possibilidades. Além de variáveis internas, os objetos possuem funções internas, chamadas de métodos.

14.1 Classes e objetos

Uma classe de objetos é estrutura formada por variáveis internas e funções. Vejamos um exemplo de uma classe contendo três variáveis internas e um método. Veja o seguinte exemplo:

```
#include <iostream>

class minha_classe_de_objetos{
public:
double x,y;
int n;

void multiplica(void);
};


void minha_classe_de_objetos::multiplica(void){
x=y*n;
}
```

```
int main(int argn, char** argc){
minha_classe_de_objetos objeto;

objeto.x=10;
objeto.y=10;
objeto.n=3;

std::cout <<"objeto.x = " << objeto.x << std::endl;
std::cout <<"objeto.y = " << objeto.y << std::endl;
std::cout <<"objeto.n = " << objeto.n << std::endl;

std::cout <<"objeto.multipla()" << std::endl;

objeto.multiplica();

std::cout <<"objeto.x = " << objeto.x << std::endl;
        return 0;
}
```

Aqui foi declarada a classe chamada de "`minha_classe_de_objetos`". Dentro desta classe, há três variáveis públicas (depois discutiremos este conceito) e um método. As variáveis públicas dentro de uma objeto são acessadas da mesma forma como em uma estrutura. Os métodos precisam ser declarados dentro da classe e depois definidos indicando o nome da classe conforme exemplo.

14.2 Variáveis e métodos privados

Em C++, é possível criar variáveis e métodos privados, que são acessíveis apenas internamente. Veja o seguinte exemplo:

```
#include <iostream>

class minha_classe_de_objetos{
public:
double x,y;

void multiplica(void);
void grava_n(int);
int le_n(void);
```

```
private:

int n;
};


void minha_classe_de_objetos::multiplica(void){
x=y*n;
}


void minha_classe_de_objetos::grava_n(int k){
n=k;
}

int minha_classe_de_objetos::le_n(void){
return n;
}




int main(int argn, char** argc){
minha_classe_de_objetos objeto;

objeto.x=10;
objeto.y=10;
objeto.grava_n(3);

std::cout <<"objeto.x = " << objeto.x << std::endl;
std::cout <<"objeto.y = " << objeto.y << std::endl;
std::cout <<"objeto.n = " << objeto.le_n() << std::endl;

std::cout <<"objeto.multipla()" << std::endl;

objeto.multiplica();

std::cout <<"objeto.x = " << objeto.x << std::endl;
return 0;
}
```

Neste exemplo, foram criados dois métodos: `grava_n()` e `le_n()`. Estes métodos são públicos e permitem o acesso à variável privada n. Um tentativa de acessar diretamente a variável, por exemplo, $objeto.n = 2$ geraria um erro de compilação.

14.3 Método construtor

É possível definir dentro de uma classe, um método a ser executado automaticamente cada vez um objeto daquela classe é criado. Este método é chamando de construtor e recebe o mesmo nome da classe. Veja o exemplo:

```
 #include <iostream>

class minha_classe_de_objetos{
public:
double x,y;
minha_classe_de_objetos(void);

private:
int n;
};


minha_classe_de_objetos::minha_classe_de_objetos(void){
std::cout<< "Um objeto da classe minha_classe_de_objetos foi criado."<<std::endl
}




int main(int argn, char** argc){
minha_classe_de_objetos objeto;


return 0;
}
```

O método construtor pode receber parâmetros, veja o exemplo abaixo:

```
  #include <iostream>

class minha_classe_de_objetos{
```

```
public:
double x,y;
minha_classe_de_objetos(void);
minha_classe_de_objetos(int);

private:
int n;
};


minha_classe_de_objetos::minha_classe_de_objetos(void){
std::cout<< "Um objeto da classe minha_classe_de_objetos foi criado."<<std::endl;
}

minha_classe_de_objetos::minha_classe_de_objetos(int j){
std::cout<< "Um objeto da classe minha_classe_de_objetos foi criado e recebeu j = "<<
}



int main(int argn, char** argc){
minha_classe_de_objetos objeto(2);


return 0;
}
```

Observe que a classe acima admite dois construtores distindos, o construtor padrão `minha_classe_de_objetos(void)` e o construtor `minha_classe_de_objetos(int)`.

14.4 Ponteiros para objetos

Assim como qualquer outro tipo de variável, é possivel criar ponteiros e arrays de classes de objetos. Veja o exemplo:

```
#include <iostream>

class minha_classe_de_objetos{
public:
double x,y;
minha_classe_de_objetos(void);
```

```
minha_classe_de_objetos(int);

private:
int n;
};


minha_classe_de_objetos::minha_classe_de_objetos(void){
std::cout<< "Um objeto da classe minha_classe_de_objetos foi criado."<<std::endl
}

minha_classe_de_objetos::minha_classe_de_objetos(int j){
std::cout<< "Um objeto da classe minha_classe_de_objetos foi criado e recebeu j
}



int main(int argn, char** argc){
    minha_classe_de_objetos* ponteiro_para_objeto;

ponteiro_para_objeto=new minha_classe_de_objetos(1);

//É possível acessar o objeto das seguintes formas:
ponteiro_para_objeto->x=1;
(*ponteiro_para_objeto).x=1;
ponteiro_para_objeto[0].x=1;
return 0;
}
```

No exemplo, criou-se um ponteiro chamado `ponteiro_para_objeto` para classe do tipo `minha_classe_de_objetos`. Observe que o objeto deve ser criado usando o comando `new`.

Referências Bibliográficas

[1] R.L. Burden and J.D. Faires. *Análise Numérica*. Cengage Learning, 8 edition, 2013.

[2] L. Damas. *Linguagem C*. LTC, Rio de Janeiro, 2007.

[3] B. W. Kernigham and D. M. Ritchie. *The C Programming Language*. Englewood Cliffs: Prentice Hall, 1998.

[4] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C : the Art of Scientific Computing*. Cambridge University Press, 2ed edition, 1992.

[5] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C++ : the Art of Scientific Computing*. Cambridge University Press, 2ed edition, 2002.

[6] H. Schildt. *C : completo e total*. Makron Books, São Paulo, 3ed edition, 1997.

[7] E. L. F. Senne. *Primeiro curso de programação em C*. Visual Books, Florianopolis, 2006.

[8] Todos os Colaboradores. Cálculo numérico - um livro colaborativo - versão com scilab. disponível em https://www.ufrgs.br/numerico/livro/main.html, Novembro 2016.