UNIVERSIDADE ESTADUAL DO CEARÁ

FACULDADE DE EDUCAÇÃO, CIÊNCIAS E LETRAS DO SERTÃO CENTRAL

PROGRAMA DE PÓS-GRADUAÇÃO EM MATEMÁTICA

MESTRADO PROFISSIONAL EM MATEMÁTICA

GUSTAVO NOGUEIRA LEITE

PYTHON E FORTRAN COMO FERRAMENTA NA RESOLUÇÃO DE PROBLEMAS DE MATEMÁTICA E FÍSICA

QUIXADÁ – CEARÁ

2020

GUSTAVO NOGUEIRA LEITE

PYTHON E FORTRAN COMO FERRAMENTA NA RESOLUÇÃO DE PROBLEMAS DE MATEMÁTICA E FÍSICA

Dissertação apresentada ao Curso de Mestrado Profissional em Matemática do Programa de Pós-Graduação em Matemática do Faculdade de Educação, Ciências e Letras do Sertão Central da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Matemática. Área de Concentração: Matemática

Orientador: Prof. Dr. Jobson de Queiroz Oliveira

Co-Orientador: Prof. Dr. Antonio Joel Ramiro de Castro

QUIXADÁ – CEARÁ

2020

Dados Internacionais de Catalogação na Publicação

Universidade Estadual do Ceará

Sistema de Bibliotecas

Leite, Gustavo Nogueira .
Python e fortran como ferramenta na resolução de problemas de matemática e física [recurso eletrônico] / Gustavo Nogueira Leite. - 2020
Um arquivo no formato PDF do trabalho acadêmico com 151 folhas.

Dissertação (mestrado profissional) - Universidade Estadual do Ceará, Centro de Ciências e Tecnologia, Mestrado Profissional em Matemática em Rede Nacional, Quixadá, 2020.
Área de concentração: Matemática em Rede Nacional..
Orientação: Prof. Dr. Jobson de Queiroz Oliveira.
Coorientação: Prof. Dr. Antonio Joel Ramiro de Castro.

1. Python . 2. Linguagens de Programação. 3. Matemática. 4. Física. 5. Fortran. I. Título.

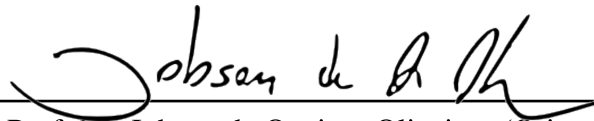
GUSTAVO NOGUEIRA LEITE

PYTHON E FORTRAN COMO FERRAMENTA NA RESOLUÇÃO DE PROBLEMAS DE MATEMÁTICA E FÍSICA

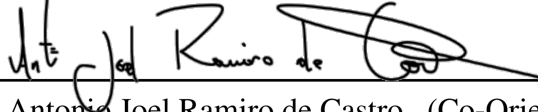
Dissertação apresentada ao Curso de Mestrado Profissional em Matemática do Programa de Pós-Graduação em Matemática do Faculdade de Educação, Ciências e Letras do Sertão Central da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Matemática. Área de Concentração: Matemática

Aprovada em: 20 de Dezembro de 2020

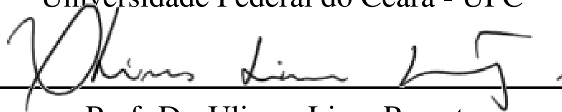
BANCA EXAMINADORA



Prof. Dr. Jobson de Queiroz Oliveira (Orientador)
Universidade Estadual do Ceará UECE



Prof. Dr. Antonio Joel Ramiro de Castro (Co-Orientador)
Universidade Federal do Ceará - UFC



Prof. Dr. Ulisses Lima Parente
Universidade Estadual do Ceará UECE

Prof. Dr. Otávio Paulino Lavor
Universidade Federal Rural do Semi-Árido - UFERSA

AGRADECIMENTOS

Se você participou dessa jornada, de modo direto ou indireto, sinta-se agradecido.

“ This statement is true, I guess. ”

“ Estaremos sempre sob o mesmo céu. ”

(...)

RESUMO

O uso das linguagens de programação tem aumentado enormemente em diversas áreas. Por essa razão neste trabalho é feita uma contextualização histórica da linguagem de programação Python e Fortran motivados pela aplicabilidade tanto em Matemática quanto em Física. Também é dedicada uma parte deste trabalho para a manipulação básica de cada uma das linguagens. Foram escolhidos cinco problemas, três problemas de Matemática e dois problemas de Física para serem resolvidos de maneira algorítmica, apresentando a solução desses problemas utilizando as duas linguagens em questão. Esse estudo é significativo devido ao grande emprego de recursos computacionais nas diversas áreas da Matemática, tanto em setores acadêmicos quanto em setores comerciais. Por este motivo, a motivação deste trabalho é aplicar as linguagens de computação como instrumento para cálculos matemáticos, conhecer as linguagens Fortran e Python e suas usabilidades.

Palavras-chave: Python. Fortran. Linguagens de Programação. Matemática. Física.

ABSTRACT

The use of programming languages has increased enormously in several areas. For this reason, in this work, a historical contextualization of Python and Fortran programming language is done, motivated by applicability in Mathematics and Physics. Part of this work is also dedicated to basic manipulation from each languages. Five problems, three Mathematics problems and two Physics problems were chosen to be solved in an algorithmic way, presenting the solution of these problems using two languages in question. This study is significant due to large use of computational resources in various areas of Mathematics, both in academic and commercial sectors. For this reason, the motivation of this work is to apply computer languages as an instrument to mathematical calculations, know Fortran and Python languages and their usability.

Keywords: Python. Fortran. Programming languages. Math, Physic.

LISTA DE FIGURAS

Figura 1 – Onda senoidal. 34
Figura 2 – Multiplots. 35
Figura 3 – Usando grids. 36
Figura 4 – Gráfico da interpolação de um conjunto de dados extraídos da função $f(x) = \sin(x)/x$. 62
Figura 5 – Aproximando função $f(x) = e^x$ pelo método de *spline* cúbico. 71
Figura 6 – Área sob uma curva 77
Figura 7 – Divisão dos intervalos 83
Figura 8 – Soluções aproximadas. 101
Figura 9 – Solução da equação $y' = y$, $y(0) = 1$ usando método de Euler. 102
Figura 10 – Solução da equação $y' = y$, $y(0) = 1$ usando método de Euler com regra trapezoidal. 103
Figura 11 – Comparação dos métodos de Euler, Euler com regra trapezoidal e Runge-Kutta na solução da equação $y' = y - t^2 + 1$, $y(0) = 0.5$. 109
Figura 12 – Movimento em uma dimensão. 127
Figura 13 – Movimento de projétil em uma dimensão com resistência. 129
Figura 14 – Diagrama de corpo livre para um projétil. 129
Figura 15 – Trajetória de um projétil com e sem resistência. 133
Figura 16 – Um pêndulo de massa m, sob a ação da força da gravidade. 134
Figura 17 – Pêndulo com amortecimento. 137
Figura 18 – Energia cinética, potencial e total. 138
Figura 19 – Pêndulo com um torque externo. 138
Figura 20 – Posição para amplitude igual a 1,2. 139
Figura 21 – Velocidades para amplitude igual a 1,2. 139
Figura 22 – Energia para amplitude igual a 1,2. 139

LISTA DE TABELAS

Tabela 1 – Operadores Aritméticos. 20
Tabela 2 – Operadores de comparação. 20
Tabela 3 – Operadores lógicos. 20
Tabela 4 – Manipulação de strings. 22
Tabela 5 – Funções para Listas. 24
Tabela 6 – Palavras chaves da linguagem. 39
Tabela 7 – Operadores aritméticos. 43
Tabela 8 – Operadores relacionais. 43
Tabela 9 – Algumas funções matemáticas 55
Tabela 10 – Valores para alguns pontos de $\ln x$ 60
Tabela 11 – Esquema de diferenças divididas para um conjunto de pontos 64
Tabela 12 – Tabela Romberg 89

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Tipo e operação com inteiros. 19
Código-fonte 2 – Tipo e operação com flutuantes. 19
Código-fonte 3 – Variáveis e operação com string. 21
Código-fonte 4 – Definição e exemplo com Lista. 22
Código-fonte 5 – Definição e exemplo com Tuplas. 25
Código-fonte 6 – Definição e exemplo de sintaxe dos condicionais. 26
Código-fonte 7 – Definição e exemplo de sintaxe do laço for. 27
Código-fonte 8 – Definição e exemplo de sintaxe do laço while. 28
Código-fonte 9 – Definição e exemplo de sintaxe de funções. 29
Código-fonte 10 – Exemplo de uso da biblioteca math. 30
Código-fonte 11 – Arrays em Numpy. 31
Código-fonte 12 – Gráfico onda senoidal usando matplotlib 33
Código-fonte 13 – Usando subplot. 35
Código-fonte 14 – Usando grid. 36
Código-fonte 15 – Programa principal 37
Código-fonte 16 – Soma dois números. 37
Código-fonte 17 – Maior valor armazenado em inteiro de quatro bytes. 39
Código-fonte 18 – Maior número que pode ser armazenado pelo tipo de dado inteiro. 39
Código-fonte 19 – Dado tipo real. 40
Código-fonte 20 – Exemplo character 41
Código-fonte 21 – Sintaxe if... 44
Código-fonte 22 – Exemplo usando if. 44
Código-fonte 23 – Sintaxe if... else. 45
Código-fonte 24 – Exemplo usando if... else. 45
Código-fonte 25 – Sintaxe de if... else if... else. 46
Código-fonte 26 – Exemplo usando if... else if... else. 46
Código-fonte 27 – Sintaxe de if... aninhada. 47
Código-fonte 28 – Exemplo usando if... aninhado. 47
Código-fonte 29 – Sintaxe de do loop. 48
Código-fonte 30 – Fatorial dos números de 1 a 10. 49
Código-fonte 31 – Declaração de parada. 50

Código-fonte 32 – Declaração de array. 50
Código-fonte 33 – Declaração de array bidimensional. 51
Código-fonte 34 – Adição de arrays em Fortrab 51
Código-fonte 35 – *Slicing* arrays em Fortran 52
Código-fonte 36 – Cáculo area de um círculo. 53
Código-fonte 37 – Usando sub-rotina. 54
Código-fonte 38 – Interpolação de Lagrange 58
Código-fonte 39 – Interpolação de Lagrange com múltiplos argumentos 59
Código-fonte 40 – Interpolação usando Polinômio de Lagrange 60
Código-fonte 41 – Interpolação usando Polinômio de Lagrange 61
Código-fonte 42 – Interpolação usando Método de Newton 63
Código-fonte 43 – Polinômio interpolador passando pelos pontos dados. 64
Código-fonte 44 – Interpolação usando Método de *spline* cúbico 68
Código-fonte 45 – Interpolação de Lagrange 71
Código-fonte 46 – Interpolação de Lagrange com múltiplos argumentos 72
Código-fonte 47 – Interpolação usando método de Newton 73
Código-fonte 48 – Interpolação usando Método de *spline* cúbico 74
Código-fonte 49 – Integração de uma função usando o método dos trapézios 78
Código-fonte 50 – Integração de uma função usando o método de Simpson 82
Código-fonte 51 – Integração de uma função usando o método dos Trapézio Adaptativo 84
Código-fonte 52 – Integração de uma função usando o método de Simpson Adaptativo 86
Código-fonte 53 – Integração de uma função usando o método de romberg 89
Código-fonte 54 – Integração de uma função usando o método dos trapézios 91
Código-fonte 55 – Integração de uma função usando o método dos trapézios Adaptivo 92
Código-fonte 56 – Integração de uma função usando o método de Simpson 93
Código-fonte 57 – Integração de uma função usando o método de Simpson Adaptivo 94
Código-fonte 58 – Integração de uma função usando o método de romberg 96
Código-fonte 59 – Método de Euler. 101
Código-fonte 60 – Método de Euler usando a regra trapezoidal. 102
Código-fonte 61 – Método de Runge-Kutta de terceira ordem. 107
Código-fonte 62 – Método de Runge-Kutta de terceira ordem. 108
Código-fonte 63 – Método de Runge-Kutta de terceira ordem. 109

Código-fonte 64 – Método de Runge-Kutta para um sistema de k equações. 113
Código-fonte 65 – Método de Runge-Kutta para um sistema de k equações usando a biblioteca numpy. 115
Código-fonte 66 – Método de Euler. 115
Código-fonte 67 – Método de Euler usando a regra trapezoidal. 116
Código-fonte 68 – Método de Runge-Kutta de terceira ordem. 117
Código-fonte 69 – Método de Runge-Kutta de quarta ordem. 118
Código-fonte 70 – Comparação dos métodos de Euler, Euler com trapezoidal e Runge-Kutta na solução da equação $y' = y - t^2 + 1, y(0) = 0.5$. 119
Código-fonte 71 – Método de Runge-Kutta para um sistema de k equações. 122
Código-fonte 72 – Simular movimento de projétil em uma dimensão. 126
Código-fonte 73 – Movimento de projétil em uma dimensão com resistência. 128
Código-fonte 74 – Trajetória de um projétil com e sem resistência. 131
Código-fonte 75 – Movimento de um pêndulo sob ação da gravidade. 135
Código-fonte 76 – Simular movimento de projétil em uma dimensão. 140
Código-fonte 77 – Trajetória de um projétil com e sem resistência. 141
Código-fonte 78 – Movimento de um pêndulo sob ação da gravidade. 144

SUMÁRIO

1 INTRODUÇÃO 15

2 NOÇÕES BÁSICAS DE PROGRAMAÇÃO 18

2.1 Noções Python 18

2.1.1 Variáveis 18

2.1.2 String 21

2.1.3 Listas 22

2.1.4 Tuplas 24

2.1.5 Condicionais e Laços 25

2.1.6 Funções 28

2.1.7 Bibliotecas 30

2.2 Noções Fortran 36

2.2.1 Sintaxe básica 36

2.2.2 Tipos de dados 38

2.2.3 Variáveis 41

2.2.4 Operadores 43

2.2.5 Condicionais 44

2.2.6 Loops 48

2.2.7 Arrays 50

2.2.8 Procedimentos 52

2.2.9 Funções intrínsecas 55

3 INTERPOLAÇÃO 56

3.1 Polinômios de Lagrange 56

3.2 Diferenças divididas de Newton 62

3.3 Interpolação por *Spline* Cúbico 65

3.4 Implementação em Fortran 71

4 INTEGRAÇÃO DE FUNÇÕES 77

4.1 O Método dos Trapézios 77

4.2 O Método de Simpson 79

4.3 Quadratura Adaptativa 82

4.4 Método de Romberg 87

4.5 Implementação em Fortran 91

5 EQUAÇÕES DIFERENCIAIS ORDINÁRIAS 99
5.1 Método de Euler 99
5.2 Métodos de Runge-Kutta 103
5.3 Sistema de equações diferenciais 112
5.4 Implementação em Fortran 115
6 PROPOSTA DE ATIVIDADE NO ENSINO MÉDIO 125
6.1 Movimento de projétil 125
6.2 Pêndulo 133
6.3 Implementação em Fortran 140
7 CONSIDERAÇÕES FINAIS 146
REFERÊNCIAS 149

1 INTRODUÇÃO

A Linguagem de Programação (LP) é uma mecanismo utilizado em computação para escrever programas - algoritmo -, isto é, conjuntos de instruções sequenciadas a serem seguidas pelo computador para realizar um determinado processo. Dito de outro modo, a LP é a comunicação entre o indivíduo que deseja resolver um determinado problema e o computador escolhido para ajudá-lo na solução. Sendo assim, a LP deve fazer a ligação entre o pensamento humano - por muitas vezes não muito bem estruturado - e a precisão exigida para o processamento da máquina.

Desenvolver e implementar um algoritmo faz necessário descrevê-lo de forma que o computador possa executá-lo. Dessa forma, a LP precisa integrar, suportar a definição de ações - rotinas - e prover meios para especificar operações básicas de computação, além de permitir que usuários especifiquem como estas rotinas devem ser sequenciados para resolver um problema. Uma linguagem de programação pode ser entendida como uma notação usada para especificar algoritmos com precisão.

As primeiras LP conhecidas eram códigos de máquina complicados que eram inseridos manualmente nas primeiras máquinas de computação. Uma das LP mais antigas, o FORTRAN foi desenvolvido por uma equipe de programadores da IBM liderada por John Backus (IBM, 2011), e foi publicado pela primeira vez em 1957. O nome FORTRAN é um acrônimo para FORmula TRANslation, porque foi projetado para permitir a tradução fácil da matemática fórmulas em código.

Muitas vezes referida como uma linguagem científica, FORTRAN foi a primeira linguagem de alto nível, usando o primeiro compilador já desenvolvido. Antes do desenvolvimento do FORTRAN, os programadores de computador eram obrigados a programar em código de máquina, o que era uma tarefa extremamente difícil e demorada, sem mencionar a terrível tarefa de depurar o código. O objetivo durante seu design era criar uma linguagem de programação que fosse: simples de aprender, adequada para uma ampla variedade de aplicações, independente de máquina, e permitiria que expressões matemáticas complexas fossem declaradas de forma semelhante à notação algébrica regular e ainda quase tão eficiente na execução quanto a linguagem assembly. Como FORTRAN era muito mais fácil de codificar, os programadores foram capazes de escrever programas mais rápido do que antes, enquanto a eficiência de execução foi pouco reduzida, isso permitiu que eles se concentrassem mais nos aspectos de resolução de um problema e menos na codificação .

Desde sua criação em 1954 e seu lançamento comercial em 1957 como o progenitor do software, Fortran se tornou o primeiro padrão de linguagem de computador, "ajudou a abrir as portas para a computação moderna"e pode muito bem ser o produto de software mais influente em história. O Fortran liberou os computadores do domínio exclusivo dos programadores e os abriu para quase todo mundo. Ainda está em uso mais de 50 anos após sua criação.

Pela primeira vez, FORTRAN tornou o código compreensível para pessoas com experiência em outras áreas além da computação, abrindo a programação para matemáticos e cientistas. Alguém que sabia álgebra no ensino médio, mas nada sobre computadores, provavelmente poderia descobrir as declarações do Fortran. Fortran iniciou o processo de abstração do software do hardware em que era executado. Os programas de linguagem de máquina anteriores tinham que ser escritos para um computador específico, enquanto um programa Fortran poderia ser executado em qualquer sistema com um compilador Fortran.

A linguagem de programação Python foi criada em 1991 por Guido Van Rossumem, no Instituto Nacional de Pesquisa para Matemática e Ciência da Computação da Holanda (CWI) cujo público alvo originalmente eram físicos e engenheiros (ROSSUM, 1995), com a finalidade de ser uma linguagem simples e de fácil compreensão. Além de simples, Python é uma linguagem muito poderosa, de alto nível que possui um modelo de desenvolvimento aberto, comunitário e gerenciado pela *Python Software Foudation (PSF)*, sem fins lucrativos. Conta com bibliotecas poderosas desenvolvidas em comunidades de colaboradores. Pode ser usada para desenvolver e administrar grandes sistemas.

A linguagem Python se destaca pela facilidade de programação e versatilidade. Python é uma linguagem de uso geral, que pode ser utilizada para diversas aplicações, como o tratamento de dados científicos, no caso de interesse desta dissertação em soluções numéricas. Apresenta uma sintaxe simples, tornando os programas mais legíveis, o que também facilita o aprendizado da linguagem.

A ciência da computação costumava ser uma área reservada apenas ao ensino superior, mas agora está se tornando amplamente adotada nos currículos das escolas primárias e secundárias. Muitos países pela Europa estão começando a introduzir a codificação como uma habilidade básica ao lado da leitura, escrita e aritmética.

Portanto, levando em conta todo o exposto, o presente trabalho tem a proposta de relacionar o ensino de matemática e física com a utilização da linguagem de programação Python e Fortran através de resolução de problemas, pois é cada vez mais recorrente o acesso a tecnologias e as transformações que estas possibilitam à sociedade. Nesse contexto as práticas

de ensino podem tentar usufruir desses recursos tecnológicos para dar um maior dinamismo e tornar a aprendizagem mais prazerosa e significativa.

Por fim, o trabalho está organizado da seguinte forma: no capítulo 02, apresentamos uma introdução as linguagens de programação Python e FORTRAN. Nos capítulos seguintes resolvemos alguns problemas de análise numérica, sendo eles, o problema de interpolação polinomial no capítulo 03, integração numérica no capítulo 04, equações diferenciais ordinárias no capítulo 05 e dois problemas de física para serem aplicados no ensino médio, movimento em uma e duas dimensões, com e sem atrito, e o pendulo no capítulo 06. Terminamos apresentando, no capítulo 07, algumas considerações finais sobre este trabalho.

2 NOÇÕES BÁSICAS DE PROGRAMAÇÃO

Programação de computador é o processo de projetar e construir um programa de computador executável para realizar um resultado de computação próprio ou para realizar uma tarefa específica. A programação envolve tarefas como: análise, geração de algoritmos, precisão dos algoritmos de criação de perfil e consumo de recursos, e a implementação de algoritmos em uma linguagem de programação escolhida (comumente referida como codificação). O código-fonte de um programa é escrito em uma ou mais linguagens inteligíveis para os programadores, ao invés do código de máquina, que é executado diretamente pela unidade central de processamento. O objetivo da programação é encontrar uma sequência de instruções que automatizará o desempenho de uma tarefa (que pode ser tão complexa quanto um sistema operacional) em um computador, geralmente para resolver um determinado problema.

2.1 Noções Python

Python usa tipagem dinâmica, isto é, o tipo de uma variável é inferido pelo interpretador em tempo de execução. Quando uma variável é criada através de atribuição, o interpretador define um tipo para a variável, com as operações que podem ser aplicadas. Para uma descrição mais completa a respeito das operações e funções básicas de Python, consultar (BORGES, 2014). Todas noções básicas de programação em Python discutidas nessa dissertação são facilmente encontradas em manuais e livros disponíveis na literatura (MENEZES, 2010; TELECOMUNICAÇOES; TUTORIAL; GRUPO, 2011; BEAZLEY; JONES, 2013; LABAKI; WOISKI, v. 2, 2003).

2.1.1 Variáveis

Variáveis são pequenos espaços reservados de memória, utilizados para armazenar, atribuir e manipular dados. Os tipos (ou classe) de dados básicos são: tipo `inteiro`, armazena números inteiros; tipo `float`, flutuante, armazena números em formato decimal; e tipo `string`, armazena um conjunto de caracteres. Cada variável pode armazenar apenas um tipo de dado por vez.

Observação a ser feita é que, em Python, ao contrário de outras linguagens, não é preciso declarar o tipo de cada variável no início do programa. Quando se faz uma atribuição de valor, automaticamente a variável se torna do tipo do valor armazenado. Alguns exemplos apresentados a seguir:

Código-fonte 1 – Tipo e operação com inteiros.

```
#Define valores. Somente inteiros
a = 5; b = 2

print("a + b = ", a + b)
print("a - b = ", a - b)
print("a * b = ", a * b)
print("a // b = ", a // b)

#-------------------------------------
a + b =  7
a - b =  3
a * b =  10
a // b =  2
```

Fonte: Elaborado pelo autor.

As variáveis `a` e `b` se tornam variáveis do `tipo(classe)` inteiro, `'int'`, como especificado na linha 2 na Saída do programa Tipo e operação com inteiros, código 1. Deve-se frisar, o modo como são definidos as operações básicas de soma, subtração, multiplicação e divisão("//"), nesse caso aplicados aos inteiros.

Código-fonte 2 – Tipo e operação com flutuantes.

```
#Define valores. Somente flutuantes
c = 5.0; d = 2.0

print("c + d = ", c + d)
print("c - d = ", c - d)
print("c * d = ", c * d)
print("c / d = ", c / d)

#---------------------------------------
c + d =  7.0
c - d =  3.0
c * d =  10.0
c / d =  2.5
```

Fonte: Elaborado pelo autor.

As variáveis `c` e `d` se tornam variáveis do tipo(classe) flutuante, ‘`flot`’, como especificado na linha 2 na Saída do programa Tipo e operação com flutuantes, código 2. Deve-se frisar, o modo como são definidos as operações básicas de soma, subtração, multiplicação e divisão("`/`"), nesse caso aplicados aos flutuantes

Tabela 1 – Operadores Aritméticos.

Operador	**Descrição**	**Exemplo**
+	soma	5 + 5 = 10
-	subtração	7 - 2 = 5
*	multiplicação	2 * 2 = 4
/	divisão	4 / 2 = 2
%	resto da divisão	10 % 3 = 1
**	potenciação	4**2 = 16

Fonte: Elaborado pelo autor.

Tabela 2 – Operadores de comparação.

Operador	**Descrição**	**Exemplo**
<	menor que	a < 10
<=	menor ou igual	b <= 5
>	maior	c > 4
>=	maior ou igual	d >= 2
==	igual	e == 1
!=	diferente	f != 16

Fonte: Elaborado pelo autor.

Tabela 3 – Operadores lógicos.

Operador	**Descrição**	**Exemplo**
Not	NÃO	**not** a
And	E	(a<=10) **and** (c=8)
Or	OU	(a<=15) **or** (c=18)

Fonte: Elaborado pelo autor

Os tipos numéricos básicos utilizados em Python, são números inteiros (`int`), números decimais (`float`) e, por sua vez, números longos (`long`) e números complexos (`complex`). Com isso, a linguagem também possui seus operadores aritméticos, lógicos, de comparação, como descritos nas tabelas 1, 2 e 3.

2.1.2 String

A atribuição de valor para uma variável pode ser conseguido utilizando o comando `input(...)`, que solicita ao usuário o valor a ser atribuído à variável. O comando `input(...)`, sempre vai retornar uma `string`. Nesse caso, para retornar dados do tipo inteiro ou `float`, é preciso converter o tipo do valor lido. Para isso, utiliza-se o `int (string)` para converter para o tipo inteiro, ou `float (string)` para converter para o tipo `float`. Como mostra os exemplos a seguir

Código-fonte 3 – Variáveis e operação com string.

```
1  nome=input("Inserir seu nome: ")
2  print("O nome inserido foi: ",nome)
3
4  num=int(input("Adicione um número: "))
5  print("O número inserido foi: ", num)
6
7  altura=float(input("Inserir sua altura (m): "))
8  print("A altura inserida foi:", altura,"m.")
9  #-----------------------------------------------
10 Inserir seu nome: Fulano de Beltrano
11 O nome inserido foi:  Fulano de Beltrano
12
13 Adicione um número: 125
14 O número inserido foi:  125
15
16 Inserir sua altura: 1.72
17 A altura inserida foi: 1.72 m.
```

Fonte: Elaborado pelo autor.

Os nomes das variáveis em Python devem ser iniciados com uma letra, porém podem possuir outros tipos de caracteres, como números e símbolos. O símbolo sublinha (_), ou "`underline`", também é aceito no início de nomes de variáveis.

Uma `string` nada mais é do que uma sequência de caracteres simples. Em Python, `strings` são utilizadas com aspas simples (‘...’) ou aspas duplas ("..."), como escrita nos exemplos 1, 2 e 3 acima. A saber, para exibir uma *string*, utiliza-se do comando `print()`. Ou seja, os argumentos da função `print()`, como feitos nos exemplos acima, que são escritas em

aspas simples ou duplas trata-se de `strings`. Outro exemplo básico e usual nos cursos iniciais de programação é escrever o primeiro código para mostrar a frase "`Hello, World!`", que se faz em Python apenas com o comando `print("Hello, World!")`.

A manipulação de `strings` se faz mediante diferentes métodos, ou funções. A seguir são listadas algumas dessas funções de manipulação:

Tabela 4 – Manipulação de strings.

Método	Descrição	Exemplo

`len()`	Retorna o tamanho da string.	`teste = ‘‘Apostila de Python’’`
		`len(teste)`
		`18`

`split()`	Transforma a string em uma lista, utilizando os espaços como referência.	`m = "cana de açúcar"`
		`m.split()`
		`['cana', 'de', 'açúcar']`

Fonte: Elaborado pelo autor

As `strings` são abreviadas pela sigla "`str`", já os `inteiros` são abreviados pela sigla "`int`"(abreviação do inglês "integer"). As funções como `type()` e `print()` são tipos embutidos no Python como vimos nos exemplos acima.

2.1.3 Listas

Na linguagem Python, uma **Lista** é uma coleção de valores indexada, identificada, onde cada valor tem sua posição identificada por um índice; a saber, o índice 0 denota primeiro item na lista, o índice 1 denota o segundo item, e assim por diante. A notação usada para designar uma lista, são os colchetes, (`[ ]`), e por sua vez os elementos devem ser postos e separados por virgulas.

Um exemplo de lista cujos elementos postos entre colchetes e os itens adicionados são separados por vírgula, é mostrado no código a seguir

Código-fonte 4 – Definição e exemplo com Lista.

```
1 nomeLista = [ elemento1 , elemento2 , ..., elementoN]
2
3 L = [3 , 'abacate' , 9.7 , [5 , 6 ] , "Python" , (3 , 'j')]
4 print(L[2])
5 print(L[3])
6 print(L[3][1])
7
```

```
8  frutas = ['Manga', 'Laranja', 'Abacaxi', 'Mamao', 'Goiaba']
9  print(type(frutas))
10 print(frutas[4])
11 
12 #---------------------------------
13 9.7
14 [5, 6]
15 6
16 type 'list'
17 Goiaba
```

Fonte: Elaborado pelo autor

Na linha 1 do código acima temos a definição de lista, onde `nomeLista` trata-se da variável na qual armazenamos a informação de todo conteúdo que fica a direita da igualdade. Os elementos da lista, `elemento1` até o `elementoN`, são separados por vírgulas.

Da linha 3 até a linha 6, temos o exemplo 1 de lista. Neste, `L` é a variável que armazenamos a informação contida na lista. Note que uma lista pode ter elementos inteiros, `string`, como também uma "sub-lista"e assim por diante, desde que sejam separadas por virgulas. Na linha 4, pede-se para que seja impressa, pela função `print()`, o elemento [2] da lista `L`, representado por `L[2]`. O mesmo se repete para a linha 5. Vejamos agora que há uma "sub-lista"em `L`, localizado na posição [3], e com isso, a linha 6, pede para que seja impressa na tela o conteúdo da sub-lista que fica na posição [1].

Na linha 8 temos o exemplo 2, onde criamos a variável do tipo lista chamada '`frutas`' contendo cinco nomes como os seus itens. Como já visto antes a função `type()` (Linha 9) traz o tipo de variável. Observe que na linha 10 imprimimos um item da lista acessando o índice [4].

Outra característica das listas em Python é que elas são mutáveis, podendo ser alteradas depois de terem sido criadas. Dito de outro modo, podemos adicionar, remover e até mesmo alterar os itens de uma lista, bem como fazer as operações básicas como soma(concatenação) de listas, por exemplo, definindo as variáveis

`a = [0,1,2]`

`b = [3,4,5]`

`c = a + b`

`print(c)` $\to [0,1,2,3,4,5];$

multiplicação(repetição) de listas,

```
L = [1,2]
R = L * 4
print(R) → [1, 2, 1, 2, 1, 2, 1, 2];
```

fatiamento etc. Resumidos na tabela (5) estão algumas funções básicas de manipulação de listas.

Tabela 5 – Funções para Listas.

Função	Descrição	Exemplo
`len()`	retorna o tamanho da lista.	`L = [1, 2, 3, 4]` `len(L)` → 4
`min()`	retorna o menor valor da lista.	`L = [10, 40, 30, 20]` `min(L)` → 10
`max()`	retorna o maior valor da lista.	`L = [10, 40, 30, 20]` `max(L)` → 40
`sum()`	retorna soma dos elementos da lista.	`L = [10, 30, 20]` `sum(L)` → 60

`append()`	adiciona um novo valor na no final da lista.	`L = [1, 2, 3]`
		`L.append(100)`
		L → [1, 2, 3, 100]

`sort()`	ordena em ordem crescente	L = [3, 5, 2, 4, 1, 0] `L.sort()` L → [0, 1, 2, 3, 4, 5]

Fonte: Elaborado pelo autor

2.1.4 Tuplas

Assim como nas lista, uma tupla é uma coleção de valores indexada, identificada, onde cada valor tem sua posição identificada por um índice; a saber, análogo ao anterior, o índice 0 denota primeiro item na tupla, o índice 1 denota o segundo item, e assim por diante. A notação usada para designar uma tupla, são os parenteses, `(  )`, e por sua vez os elementos devem ser postos e separados por virgulas. A diferença entre Listas e Tuplas, é que nesta última são imutáveis, ou seja, seus elementos não podem ser alterados.

Uma tupla é imutável, ou seja, após ser criada, ela não pode ser alterada. Vejamos como define-se e se usa no código abaixo

Código-fonte 5 – Definição e exemplo com Tuplas.

```
1  nomeTupla = ( valor1, valor2, ..., valorN )
2
3  naipesBaralho = ('Copas','Espadas','Paus','Ouros')
4  print(naipesBaralho)
5
6  t = (10,20,30,40,50)
7  a,b,c,d,e = t
8  print("a=",a,"b=",b)
9  print("d+e=",d+e)
10 #-------------------------------
11 ('Copas','Espadas','Paus','Ouros')
12 a= 10 b= 20
13 d+e= 90
```

Fonte: Elaborado pelo autor.

Na linha 1 do código acima temos a definição de tupla, onde `nomeTupla` trata-se da variável na qual armazenamos a informação de todo conteúdo que fica a direita da igualdade. Os elementos da tupla, `valor1` até o `valorN`, são separados por vírgulas.

Na linha 3 temos o exemplo 1 de tupla. Neste, `naipesBaralho` é a variável que armazenamos a informação contida na tupla. Na linha 6 é dado o exemplo 2 de tupla, `T` é a variável que armazena a informação da tupla. Vejamos que na linha 7, está respectivamente associando-se para cada elemento da tupla as novas variáveis `a`,`b`, `c`, `d` e `e`. Em seguida pede-se para que seja impressa, pela função `print()` os valores das respectivas variáveis.

2.1.5 Condicionais e Laços

Facilmente se encontra em programas escritos em Python que certos conjuntos de instruções sejam executados de forma condicional, para casos em que se quer validar entradas de dados respeitando certas condições. Para esses casos condicionais chamamos de estrutura de decisão. Essas permitem modicar o curso do fluxo de execução de um programa, a depender do valor (Verdadeiro/Falso)(True/false) de um teste lógico. Em Python as estruturas de decisão são as seguintes:

- `if` (se): é usado para decidir se determinado trecho do programa deve ou não ser executado. Está associado a uma condição, e será executado se o valor da condição for verdadeiro.

- `if..else` (se..senão): é usado se determinado trecho de código uma condição for verdadeira e se a outra condição for falsa.
- `if..elif..else` (se..senão..senão se): usado quando há diversas condições, cada uma associada a um trecho de código, via comando `elif`.

A sintaxe é mostrada no código a seguir juntamente com um exemplo de aplicação adaptado de (BORGES, 2014):

Código-fonte 6 – Definição e exemplo de sintaxe dos condicionais.

```
# Sintaxe
if <condição>:
    <bloco de código>
elif <condição>:
    <bloco de código>
elif <condição>:
    <bloco de código>
else:
    <bloco de código>

# Exemplos de Sinatxe condicionais
temp = int(input("Entre com a temperatura: "))
if temp < 0:
    print("Congelando...")
elif temp <= 20:
    print("Frio")
elif temp <= 25:
    print("Normal")
elif temp <= 35:
    print("Quente")
else:
    print("Muito quente!")
#-----------------------------------------
Entre com a temperatura: 23
Normal
```

Fonte: Elaborado pelo autor.

Com relação as sintaxes, escritas da linha 1 à linha 9 temos

- <condição>: sentença que possa ser avaliada como verdadeira ou falsa.

- <bloco de código>: sequência de linhas de comando.
- As clausulas `elif` e `else` são opcionais e podem existir vários `elifs` para o mesmo `if`, porém apenas um `else` ao final.

No exemplo, iniciado na linha 12 à linha 22, é pedido ao usuário que "Entre com a temperatura:", o valor de entrada é convertido para inteiro, quando dado a temperatura ao programa, "23" é a entrada digitada, inicia a execução dos condicionais do algoritmo e, "Normal" é a resposta do programa.

Laços de repetição, também chamados de `loop`, são usadas para repetir um bloco do código enquanto dada condição for verdadeira. Cada repetição do bloco é chamada de iteração. Há dois laços de repetição em Python: o laço `for`, usado para executar um número fixo de iterações, e o laço indeterminado `while`, usado quando não se sabe as iterações necessárias antes do início do laço. Para a repetição do laço `for`, é preciso que a coleção iterada contenha mais itens. Por outro lado, o laço `while` verifica a condição lógica antes de cada iteração e só continua se essa condição for verdadeira.

Código-fonte 7 – Definição e exemplo de sintaxe do laço for.

```
1  # Sintaxe laço 'for':
2  for <referência> in <sequência>:
3      <bloco de código>
4      continue
5      break
6  else:
7      <bloco de código>
8
9  # Exemplo laço 'for':  Soma de 0 a 99
10 s = 0
11 for x in range(1, 100):
12     s = s + x
13 print s
14 #----------------
15 4950
```

Fonte: Elaborado pelo autor.

Da linha 1 à 7, mostra a estrutura de aplicação do laço `for`. Por outro lado, da linha 9 à 15 é dado um exemplo para o laço `for`, para fazer a soma de todos os elementos compreendidos entre

0 e 99 dado que a iteração seja verdadeira. A função `range(m, n, p)`, muito usada em laços por retornar uma lista de inteiros, varrendo valores entre `m` e `n`, em passos de comprimento `p` do laço.

Código-fonte 8 – Definição e exemplo de sintaxe do laço while.

```
1  # Sintaxe laço 'while':
2  while <condição>:
3      < bloco de código>
4      continue
5      break
6  else:
7      < bloco de código>
8
9  # Exemplo laço 'while': Soma de 0 a 99
10 soma = 0
11 x = 1
12 while x < 100:
13     soma = soma + x
14     x = x + 1
15 print s
```

Fonte: Elaborado pelo autor.

Da linha Da linha 1 à 7, mostra a estrutura de aplicação do laço `while`, o código dentro do laço `while` é repetido enquanto a condição estiver sendo avaliada como verdadeira. Assim como no exemplo anterior aplicado ao laço `for`, o exemplo proposto ao laço `while` se propõe a somar todos os elementos compreendidos entre o mesmo intervalo, e isso é feito enquanto a condição se manter verdadeira e não não há como determinar quantas iterações vão ocorrer sem sequência a seguir. A saída do programa Definição e exemplo de sintaxe do laço `while` é análoga. Ambas sintaxes e exemplos foram adaptados (BORGES, 2014).

2.1.6 Funções

Em Python uma função é uma rotina de instruções que calcula um ou mais resultados, podendo receber parâmetros pré-determinados. Vimos acima algumas funções prontas da linguagem, como o `print()`, `input()`, `type()` etc. como mostrados na tabela (5).

Funções são definidas pelo uso da palavra-chave `def`, conforme a sintaxe e exemplos

a seguir:

Código-fonte 9 – Definição e exemplo de sintaxe de funções.

```
#Sintaxe de 'função':
def func(parametro1, parametro2,...):
    <bloco de código>
    return valor

# Função para imprimir o maior entre 2 valores
def maior(x,y):
    if x>y:
        return x
    return y

maior(4, 7)
7

def fatorial(n):
    p = 1
    for i in range(1, n + 1):
        p *= i
    return p

for i in range(1, 6):
    print(i, '->', fatorial(i))

# Saída fatorial
1 -> 1
2 -> 2
3 -> 6
4 -> 24
5 -> 120
```

Fonte: Elaborado pelo autor.

Da linha 1 à 4 representa a sintaxe, forma estrutural de definir uma função em Python. o fator `func()` é a função que pode receber os parâmetros $1, 2, \ldots, N$ se assim forem declarados. O <bloco de código> na sintaxe pode ser melhor visualizado no exemplo para imprimir o maior entre 2 valores e também em fatorial.

2.1.7 Bibliotecas

Python também conta com um enorme conjunto de funções prontas para serem aplicadas em algoritmos, agrupamento de módulos, bibliotecas, que simplificam a usabilidade evitando a reescrição desnecessária de comandos básico por já estarem agrupados nesses módulos(bibliotecas) de acesso livre.

O uso desses módulos são feitos chamando-os (importando-os) no início do script com comando `import <nome_do_módulo>`. Como também é possível importar do módulo apenas a função desejada. Para isso, utilizamos o comando `from <nome_do_módulo> import função`, e a função estará disponível para utilização. Por exemplo, uso de funções matemáticas numéricas básicas da biblioteca **math** que contém representações logarítmicas, exponenciais, hiperbólicas, trigonométricas, conversões angulares etc,

Código-fonte 10 – Exemplo de uso da biblioteca math.

```
1  # Exemplo de uso da bilioteca math'
2  import math
3  x = 5
4  math.sqrt(x)
5  #2.2360679774997898
6  math.sin(x)
7  #-0.95892427466313845
8  math.log(x)
9  #1.6094379124341003
10 math.cos(x)
11 #0.28366218546322625
```

Fonte: Elaborado pelo autor.

As constantes π e e, exponencial, são definidos, podem ser usadas pelo nome quando o módulo é importado. Abaixo, estão listadas algumas funções desse módulo:

- **math.factorial(x)** - Retorna o valor de x fatorial. Se x negativo, retorna um erro.
- **math.modf(x)** - Retorna o valor inteiro e o valor fracionário de x.
- **math.exp(x)** - Retorna e exponencial de x.
- **math.log(x,base)** - Retorna o log de x na base pedida.
- **math.log1p(x)** - Retorna o log natural de x.
- **math.sqrt(x)** - Retorna a raiz quadrada de x.

- **math.degrees(x)** - Converte o ângulo x de radianos para graus.
- **math.radians(x)** - Converte o ângulo x de graus para radianos.

NumPy, abreviação de *Numerical Python*, bibliotecas Python para computação numérica, é um módulo de manipulação numérica. É um software de código aberto, *open source*, dedicado a criação e manipulação de matrizes, matrizes multidimensionais. Fornece uma variedade de funções de alto nível para lidar com matrizes complexas.

NumPy, trata-se de uma biblioteca que inclui uma classe chamada de `array`, esses são os objetos de trabalho e manipulação desta biblioteca, ou seja, pode-se pensar que esses `arrays` são arranjos variáveis com número arbitrário de elementos e homogêneos, os elementos são do mesmo tipo, semelhante às listas da linguagem Python como discutidos nas seções anteriores; As classes matriciais são também objetos de manipulação desta biblioteca, assim como as várias funções auxiliares do pacote. Outra abordagem interessante desta biblioteca é a vetorização desses `arrays` e matrizes. Essas, `arrays` e matrizes, podem ser unidimensionais (1D), bidimensionais (2D), tridimensionais (3D) ou N-dimensionais.

Exemplo da classe de `arrays` (BORGES, 2014) é posto a seguir

Código-fonte 11 – Arrays em Numpy.

```
1  import numpy as np
2  
3  # Criando arranjos
4  a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
5  
6  # Arranjo criado a partir de um intervalo
7  b = np.arange(0., 4.5, .5)
8  
9  # Arranjo de 1's de dimensão 2x3
10 c = np.ones((2, 3))
11 
12 # Arranjos podem gerar novos arranjos
13 d = np.round(numpy.cos(z), 1)
14 
15 # Multiplicando cada elemento por um escalar
16 d1 = 5 * d
17 
18 # Somando arranjos elemento por elemento
19 e = b + d
```

```
20
21 # Redimensionando o arranjo
22 b.shape = 3, 3
23
24 # Arranjo transposto
25 b.transpose()
```

Fonte: Elaborado pelo autor.

a saída completa é mostrado na listagem abaixo

```
[0 1 2 3 4 5 6 7 8]
[ 0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  ]
[[ 1.  1.  1.]
[ 1.  1.  1.]]
[ 1.  0.9 0.5 0.1 -0.4 -0.8 -1.  -0.9 -0.7]
[ 0.  2.5 5.  7.5 10.  12.5 15.  17.5 20.  ]
[ 1.  1.4 1.5 1.6 1.6 1.7 2.  2.6 3.3]
[[ 0.  0.5 1.  ]
[ 1.5 2.  2.5]
[ 3.  3.5 4.  ]]
[[ 0.  1.5 3.  ]
[ 0.5 2.  3.5]
[ 1.  2.5 4.  ]]
```

Matplotlib é um dos pacotes Python mais populares usados para visualização de dados. É uma biblioteca de plataforma cruzada para fazer gráficos 2D de dados em `arrays`. Matplotlib é escrito em Python e usa NumPy, a extensão matemática numérica do Python. Ele fornece uma API orientada a objetos que ajuda a incorporar gráficos em aplicativos usando kits de ferramentas Python GUI, como PyQt, WxPythonotTkinter. Ele pode ser usado em shells Python e IPython, notebook Jupyter e servidores de aplicativos da web também.

Matplotlib tem uma interface procedural chamada Pylab, que foi projetada para se assemelhar ao MATLAB, uma linguagem de programação proprietária desenvolvida pela MathWorks. Matplotlib junto com NumPy pode ser considerado o equivalente de código aberto do MATLAB. Foi originalmente escrito por John D. Hunter em 2003.

Vamos exibir um gráfico de linha simples de ângulo em radianos vs. seu valor de seno em Matplotlib. Para começar, o módulo Pyplot do pacote Matplotlib é importado, com um `alias`[1] plt por uma questão de convenção.

```
1 import matplotlib.pyplot as plt
```

Em seguida, precisamos de uma matriz de números para plotar. Várias funções de `array` são definidas na biblioteca NumPy que é importada com o apelido `np`.

```
2 import numpy as np
```

Agora obtemos o objeto `ndarray` de ângulos entre 0 e 2π usando a função `arange` da biblioteca NumPy.

```
4 x = np.arange (0, math.pi * 2, 0,05)
```

O objeto `ndarray` serve como valores no eixo x do gráfico. Os valores de seno correspondentes dos ângulos em x a serem exibidos no eixo y são obtidos pela seguinte declaração:

```
5 y = np.sin(x)
```

Os valores dos dois `arrays` são plotados usando a função `plot()`.

```
6 plt.plot(x, y)
```

Podemos definir o título do gráfico e os rótulos dos eixos x e y.

```
7 plt.xlabel("Ângulo")
8 plt.ylabel("Seno")
9 plt.title("Onda senoidal")
```

A janela do visualizador de Plot é chamada pela função `show()`:

```
10 plt.show ()
```

O programa completo segue no código 12:

Código-fonte 12 – Gráfico onda senoidal usando matplotlib

```
1 from matplotlib import pyplot as plt
2 import numpy as np
3 import math #para definição de pi
4 x = np.arange(0, math.pi*2, 0.05)
```

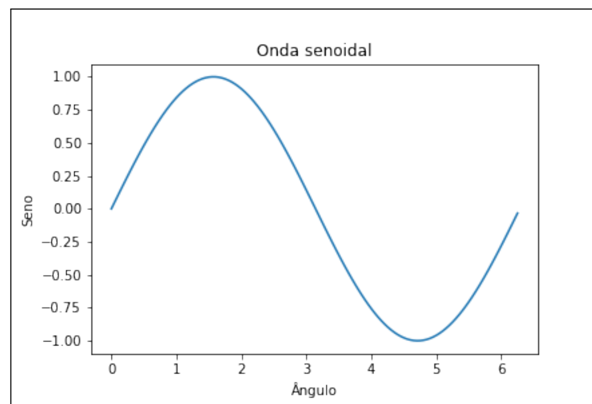
[1] O termo `alias`, que significa segundo nome ou apelido é utilizado para abreviar comandos complexos que são usados repetidamente.

```
5  y = np.sin(x)
6  plt.plot(x,y)
7  plt.xlabel("Ângulo")
8  plt.ylabel("Seno")
9  plt.title("Onda senoidal")
10 plt.show()
```

Fonte: Elaborado pelo autor.

Quando o código 12 acima é executada, o gráfico na imagem (1) é exibido.

Figura 1 – Onda senoidal.



Fonte: Elaborado pelo autor

A função `subplot()` retorna o objeto de eixos em uma determinada posição da grade. A chamada desta função é:

```
1  plt.subplot(nrows, ncols, index)
```

Na figura, a função cria e retorna um objeto Axes[2], no índice de posição de uma grade de `nrows` por `ncol`. Os índices vão de 1 a `nrows * ncols`, incrementando na ordem da linha principal.

[2] O objeto Axes é a região da imagem com o espaço de dados. Uma determinada figura pode conter muitos eixos, mas um determinado objeto Axes só pode estar em uma figura. Os eixos contém dois objetos de eixo.

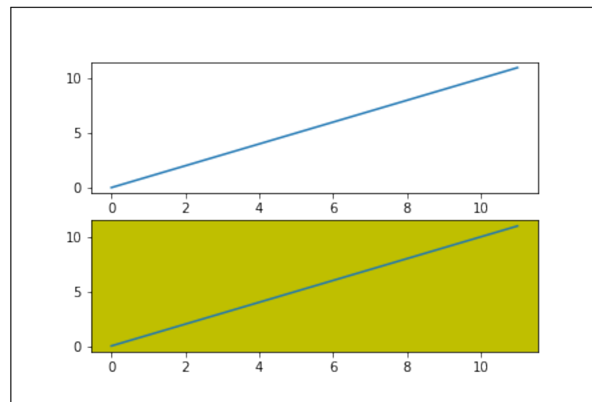
Código-fonte 13 – Usando subplot.

```
import matplotlib.pyplot as plt

# plot uma linha, criando implicitamente a subplot(1, 1, 1)
plt.subplot(2, 1, 1)
plt.plot(range(12))

# cria o segundo subplot com fundo amarelo
plt.subplot(2 1, 2, facecolor='y')
plt.plot(range(12))
```

Fonte: Elaborado pelo autor.

O código 13 acima gera a imagem (2) seguinte.

Figura 2 – Multiplots.



Fonte: Elaborado pelo autor

A função `grid()` do objeto de eixos define a visibilidade da grade dentro da figura como ligada ou desligada.

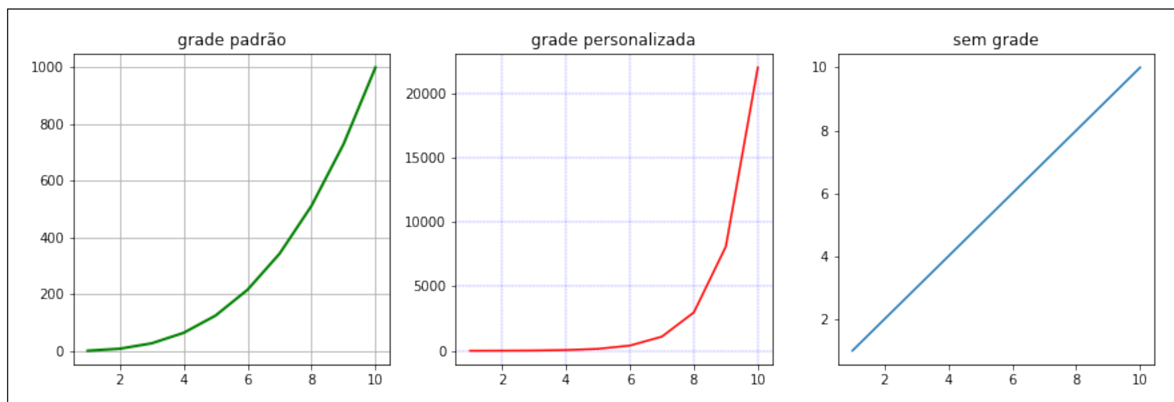
Código-fonte 14 – Usando grid.

```
import matplotlib.pyplot as plt
import numpy as np
fig, eixos = plt.subplots (1,3, figsize = (12,4))
x = np.arange(1,11)
eixos[0].plot(x, x ** 3, 'g', lw = 2)
eixos[0].grid (True)
eixos[0].set_title('grade padrão')
eixos[1].plot(x, np.exp (x), 'r')
eixos[1].grid(color = 'b', ls = '-.', lw = 0,25)
eixos[1].set_title('grade personalizada')
eixos[2].plot(x, x)
eixos[2].set_title('sem grade')
fig.tight_layout()
plt.show()
```

Fonte: Elaborado pelo autor.

O código 14 acima gera a imagem (3) seguinte.

Figura 3 – Usando grids.



Fonte: Elaborado pelo autor

2.2 Noções Fortran

2.2.1 Sintaxe básica

Um programa Fortran é composto de uma coleção de unidades de programa, como um programa principal, módulos e subprogramas ou procedimentos externos.

Cada programa contém um programa principal e pode ou não conter outras unidades de programa. A sintaxe do programa principal é a seguinte:

Código-fonte 15 – Programa principal

```
program nomePrograma
    implicit none
    ! declarações de declaração de tipo
    ! declarações executáveis
end program nomePrograma
```

Fonte: Elaborado pelo autor.

Vamos escrever um programa que adiciona dois números e imprime o resultado.

Código-fonte 16 – Soma dois números.

```
program somaNumeros

    ! Este programa simples adiciona dois números
    implicit none

    ! Declarações de tipo
    real :: a, b, result
    ! Declarações de variáveis
    a = 12.0
    b = 15.0
    resultado = a + b
    print *, 'O total é ', resultado
end program somaNumeros
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, ele produz o seguinte resultado `O total é 27.0000000`.

Notemos que:

- Todos os programas FORTRAN comaçam com a palavra-chave `program` e termina com a palavra-chave `end program` seguida pelo nome do programa.
- A instrução `implicit none` permite que o compilador verifique se todos os tipos de

variáveis foram declarados corretamente. Devemos sempre usar `implicit none` no início de cada programa.

- Os comentário em FORTRAN são iniciados com o ponto de exclamação (`!`). Todos os caracteres após serão ignorados pelo compilador.
- O comando `print*,` exibe dados na tela.
- O recuo das linhas de código é uma boa prática para manter um programa legível.
- FORTRAN permite letras maiúsculas e minusculas mas não faz distinção entre elas, exceto em `strings`

O conjunto de caracteres básicos de FORTRAN contém:

- as letras A...Z e a...z.
- os dígitos 0...9.
- o caractere sublinhado _.
- os caracteres especiais `=:`+espaço em branco`-*/()[],.$'!"%&;<>?`

Um identificador é um nome usado para identificar uma variável, procedimento ou qualquer outro item definido pelo usuário. Um nome em FORTRAN deve seguir as seguintes regras:

- Não pode ter mais de 31 caracteres
- Deve ser composto de caracteres alfanuméricos (todas as letras do alfabeto e os dígitos 0 a 9) e sublinhados (_)
- O primeiro caractere de um nome deve ser uma letra.]
- FORTRAN não diferencia maiúsculas e minúsculas.

Palavras-chaves são palavras especiais, reservadas para a linguagem. Essas palavras reservadas não podem ser usadas como identificadores ou nomes.

A tabela a seguir lista as palavras-chaves de FORTRAN.

2.2.2 Tipos de dados

FORTRAN fornece cinco tipos de dados intrínsecos, no entanto podemos criar nossos tipos de dados. Os cinco tipos são:

- Integer
- Real
- Complex
- Logical
- Character

Tabela 6 – Palavras chaves da linguagem.

allocatable	allocate	assign	assignment	backspace
block data	call	case	character	close
common	complex	contains	continue	cycle
data	deallocate	default	do	double precision
endfile	else	else if	elsewhere	end block data
end do	end function	end if	end interface	end module
end program	end select	end subroutine	end type	end where
entry	equivalence	exit	external	format
function	go to	if	implicit	in
inout	inquire	integer	intent	interface
intrinsic	kind	len	logical	module
namelist	nullify	only	open	operator
optional	out	parameter	pause	pointer
print	private	program	public	read
real	recursive	result	return	rewind
save	select case	stop	subroutine	target
then	type	type()	use	Where
While	Write			

Fonte: (TUTORIALS POINT, 2021)

Os tipo `integer` podem conter apenas valores inteiro. O exemplo a seguir extrai o maior valor que pode ser armazenado em um número inteiro de quatro bytes.

Código-fonte 17 – Maior valor armazenado em inteiro de quatro bytes.

```
program tipoInteiro
    implicit none

    integer :: maiorNumero
    print *, huge(maiorNumero)
end program tipoInteiro
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte resultado

`2147483647`

A função `huge()` fornece o maior número que pode ser armazenado pelo tipo de dado especificado. Podemos também especificador o numero de bytes usando o especificador `kind`.

Código-fonte 18 – Maior número que pode ser armazenado pelo tipo de dado inteiro.

```
program tipoInteiro
    implicit none
```

```
3
4     ! inteiro de dois bytes
5     integer ( kind = 2 ) :: int2
6
7     ! inteiro de quatro bytes
8     integer ( kind = 4 ) :: int4
9
10    ! inteiro de oito bytes
11    integer ( kind = 8 ) :: int8
12
13    ! inteiro de dezesseis bytes
14    integer ( kind = 16 ) :: int16
15
16    print*, huge(int2)
17    print*, huge(int4)
18    print*, huge(int8)
19    print*, huge(int16)
20 end program tipoInteiro
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte resultado

```
32767
2147483647
9223372036854775807
170141183460469231731687303715884105727
2147483647.
```

O tipo `real` armazena os números de ponto flutuante, como 2.0, 3.1415, -100, 876.

FORTRAN 90/95 (esse que estamos estudando) oferece controle sobre a precisão dos tipos de dados reais e inteiros por meio do especificador `kind`.

O exemplo a seguir mostra o uso de dados `real type`.

Código-fonte 19 – Dado tipo real.

```
1 programa numerosReais
2     implicit none
3
4     ! variáveis reais
5     real :: p, q, realResultado
```

```
6       ! variáveis inteiras
7       integer :: i, j, intResultado
8
9       ! atribuição de valores
10      p = 2.0
11      q = 3.0
12      i = 2
13      j = 3
14      ! divisão de ponto flutuante
15      realResultado = p/q
16      intResultado = i/j
17
18      print*, realResultado
19      print*, intResultado
20  end program numerosReais
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte resultado

```
0.666666687
0
```

O tipo `complex` é usado para armazenar números complexos. Um número complexo possui duas partes, a parte real e a parte imaginaria. Não discutiremos aqui esse tipo.

Existem apenas dois valores lógicos: `.true.` e `.false.`.

O `character type` armazena caracteres e strings. O comprimento da string pode ser especificado pelo especificador `len`. Por exmeplo:

Código-fonte 20 – Exemplo character

```
1  character ( len = 40 ) :: nome
2  nome = "FORTRAN é legal"
```

Fonte: Elaborado pelo autor.

2.2.3 Variáveis

Uma variável é um nome dado a uma área de armazenamento que nossos programas podem manipular. Cada variável deve ter um tipo específico, que determina o tamanho da

memória, a faixa de valores que podem ser armazenados e o conjuntos de operações que podem ser aplicadas.

Variáveis são declaradas no inicio de um programa (ou subprograma). A sintaxe para declarações de variáveis é a seguinte:

```
1 tipo da variável :: nome da variável
```

Por exemplo:

```
1 integer :: total
2 real :: media
3 character (len = 80) :: menssagem
```

Depois, podemos atribuir valores a essas variáveis, como:

```
1 total=1000
2 media = 7.0
3 menssagem = "Fortran é legal"
```

As constantes referem-se aos valores fixos que o programa não pode alterar durante a execução. Esses valores fixos são também chamados de literais.

Constantes podem ser de qualquer tipos de dados básicos, como uma constante inteira, uma constante flutuante, um constante de caracteres ou literal de `strings`. Existem apenas duas constantes lógicas: `.true.` e `.false.`. As constantes são tratadas como variáveis comuns, exceto que seus valores não podem ser modificados após seu definição.

Existem dois tipos de constantes:

- Constantes literais.
- Constantes nomeadas.

Uma constante literal tem um valor, mas não tem um nome. Uma constante nomeada tem um valor e um nome também. Assim, `1234`, `3.1415926` e `.true.` são exemplos de constantes literais do tipo, `integer`, `real` e `logical` respectivamente.

As constantes nomeadas devem ser declaradas no inicio de um programa ou procedimento, assim como uma declaração de tipo de variável, indicando seu nome e seu tipo. As constantes nomeadas são declaradas com o atributo `parameter`. Por exemplo:

```
1 real, parameter :: PI = 3.1415926
```

2.2.4 Operadores

Um operador é um símbolo que informa ao compilador para executar operações matemáticas ou lógicas específicas. FORTRAN oferece os seguintes tipo de operadores:

- Operadores aritméticos
- Operadores relacionais
- Operadores lógicos

A tabela 7 mostra todos os operadores aritméticos. Suponhamos que a variável A tenha o valor 5 e a variável B tenho a o valor 3, então:

Tabela 7 – Operadores aritméticos.

Operador	Descrição	Exemplo
+	Operador de adição	A + B $\rightarrow$ 8
-	Operador de subtração	A - B $\rightarrow$ 2
*	Operador de multiplicação	A * B $\rightarrow$ 15
/	Operador de divisão	A/B $\rightarrow$ 1
**	Operador de exponenciação	A**B $\rightarrow$ 125

Fonte: Elaborado pelo autor

A tabela 8 mostra todos os operadores relacionais suportados em FORTRAN.

Tabela 8 – Operadores relacionais.

Operador	Descrição	Exemplo
==	Verifica se os valores de dois operandos são iguais ou não	A == B não é verdadeiro
/=	Verifica se os valores de dois operados são iguais ou não, sendo a negação do operador anterior	A /= B é verdadeiro
>	Verifica se o valor do operando esquerdo é maior que o valor do operando direito, se sim, a condição retorna verdadeiro	A > B é verdadeiro
<	Verifica se o valor do operando esquerdo é menor que o valor do operando direito, se sim, a condição retorna verdadeiro	A < B não é verdadeiro
>=	Verifica se o valor do operando esquerdo é maior ou igual ao valor do operando direito, se sim, a condição retorna verdadeiro	A >= B é verdadeiro
<=	Verifica se o valor do operando esquerdo é menor ou igual ao valor do operando direito, se sim, a condição retorna verdadeiro	A <= B não é verdadeiro

Fonte: Elaborado pelo autor

2.2.5 Condicionais

As estruturas condicionais requerem que o programador especifique uma ou mais condições a serem avaliadas ou testadas pelo programa, junto com uma instrução ou instruções a serem executadas, se a condição for determinada como verdadeira, e opcionalmente, outras instruções a serem executadas se a condição é determinada como falsa.

Uma instrução `if... then` consiste em uma expressão lógica seguida por uma ou mais instruções e terminada por uma instrução `end if`. A sintaxe básica de uma instrução `if... then` é:

Código-fonte 21 – Sintaxe if... .

```
if (expressão lógica) then
    intrução
end if
```

Fonte: Elaborado pelo autor.

Código-fonte 22 – Exemplo usando if.

```
program exemploIF
    implicit none

    ! declaração de variaveis
    integer :: a = 10
    ! verificação lógica usando a instrução if
    if ( a < 20 ) then
        ! se a condição for verdadeira
        ! imprime na tela a seguinte mensagem
        print *, "a é menor que 20"
    end if

    print *, "Valor de a é", a
end program exemploIF
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte resultado

`a é menor que 20`

`Valor de a é 10`

Uma instrução `if... then` pode ser seguida por uma instrução `else` opcional, que é executada quando a expressão lógica é falsa. A sintaxe básica de uma instrução `if... then.. else` é:

Código-fonte 23 – Sintaxe if... else.

```
if (expressão lógica) then
    instruções
else
    outras intruções
end if
```

Fonte: Elaborado pelo autor.

Código-fonte 24 – Exemplo usando if... else.

```
program exemploIfElse
    implicit none

    ! declaração de variáveis
    integer :: a = 100
    !  verificação lógica usando a instrução if
    if ( a < 20 ) then
        ! se a condição for verdadeira
        ! imprime na tela a seguinte mensagem
        print*, "a é menor que 20"
    else
        print*, "a não é menor que 20"
    end if

    print *, "Valor de a é", a
end program exemploIfElse
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte resultado

`a não é menor que 20`

`Valor de a é 100`

Uma construção de instrução `if` pode ter uma ou mais construções `else-if` opci-

onais. Quando a condição `if` falha, o `else-if` imediatamente seguido é executado. Quando o `else-if` também falha, a instrução `else-if` seguinte (se houver) é executada e assim por diante. O `else` opcional é colocado no final e é executado quando nenhuma das instruções acima for verdadeira.

- Todas as intruções `else` são opcionais.
- `else-if` pode ser usado uma ou mais vezes.
- `else` deve sempre ser colocado no final da construção e deve aparecer apenas uma vez.

A sintaxe de uma instrução `if... else if... else` é:

Código-fonte 25 – Sintaxe de if... else if... else.

```
if ( expressão lógica 1) then
    ! bloco 1
else if ( expressão lógica 2) then
    ! bloco 2
else if ( expressão lógica 3) then
    ! bloco 3
else
    ! bloco 4
end if
```

Fonte: Elaborado pelo autor.

Código-fonte 26 – Exemplo usando if... else if... else.

```
program exemploIfElseIfElse
    implicit none

    ! declaração de variáveis
    integer :: a = 100
    ! verificação lógica usando a instrução if
    if ( a ==  10 ) then
        ! se a condição for verdadeira
        ! imprime na tela a seguinte mensagem
        print *, "Valor de a é 10"
    else if ( a == 20 ) then
        ! condição if else if é verdadeira
        print*, "Valor de a é 20"
    else if ( a ==  30 ) then
```

```
15          ! condição if else if é verdadeira
16          print*, "Valor de a é 30"
17      else
18          ! se nenhumas das condições é verdadeira
19          print*, "Não corresponde a nenhum dos valores"
20      end if
21
22      print*, "O valor exato de a é", a
23  end program exemploIfElseIfElse
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte resultado

```
Não corresponde a nenhum dos valores
Valor exato de a é 100
```

Podemos usar uma instrução `if` ou `else if` dentro de outra instrução `if` ou `else if`. A sintaxe de uma instrução `if` aninhada é a seguinte:

Código-fonte 27 – Sintaxe de if... aninhada.

```
1  if ( expressão lógica 1) then
2      ! É executada quando a expressão lógica 1 é verdadeira
3      if (expressão lógica 2) then
4          ! É executada quando a expressão lógica 2 é verdadeira
5      end if
6  end if
```

Fonte: Elaborado pelo autor.

Código-fonte 28 – Exemplo usando if... aninhado.

```
1  program exemploIfAninhado
2      implicit none
3
4      ! declaração de variavéis
5      integer :: a = 100, b = 200
6      ! verificação lógica usando a instrução if
7      if ( a ==  100 ) then
8          ! if for verdadeiro, verifica a condição seguinte
9          if ( b == 200 ) then
```

```
10              ! if interno for verdadeiro
11              print *, "Valor de a é 100 e o valor de b é 200"
12          end if
13      end if
14
15      print *, "Valor exato de a é", a
16      print *, "Valor exato de b é", b
17  end program exemploIfAninhado
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte resultado

```
Valor de a é 100 e o valor de b é 200
Valor exato de a é 100
Valor exato de b é 200
```

2.2.6 Loops

Pode existir uma situação em que precisemos executar um bloco de código várias vezes. Em geral, as instruções são executadas sequencialmente: a primeira instrução em uma função é executada primeiro, seguida pela segunda e assim por diante. Uma instrução de `loop` nos permite executar uma instrução ou grupo de instruções várias vezes.

A construção `do loop` permite que uma instrução ou série de instruções, seja executada iterativamente, enquanto uma determinada condição for verdadeira. A forma geral do `do loop` é:

Código-fonte 29 – Sintaxe de do loop.

```
1  do var = start, stop [, step]
2      ! instruções
3  end do
```

Fonte: Elaborado pelo autor.

Onde

- a variável `var` deve ser um inteiro.
- `start` é o valor inicial
- `stop` é o valor final
- `step` é o incrimento, se for omitido, a variável é aumentada de um

O programa a seguir calcula o fatorial dos números de 1 a 10.

Código-fonte 30 – Fatorial dos números de 1 a 10.

```
program fatorial
    implicit none

    ! define variáveis
    integer :: fat = 1
    integer :: i
    ! computar fatorial
    do i = 1, 10
        fat = fat * i
        ! imprime os valores
        print*, i, fat
    end do
end program fatorial
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte resultado

```
 1          1
 2          2
 3          6
 4         24
 5        120
 6        720
 7       5040
 8      40320
 9     362880
10    3628800
```

As instruções de controle de loop alteram a execução de sua sequência normal. Quando a execução deixa um escopo, todos os objetos automáticos que foram criados nesse escopo são destruídos. Se desejamos que a execução do programa termine, podemos inserir uma instrução de parada.

Código-fonte 31 – Declaração de parada.

```
program exemploStop
    implicit none

    integer :: i
    do i = 1, 10
        if ( i == 5) then
            stop
        end if
        print*, i
    end do
end program exemploStop
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte

```
1
2
3
4
```

2.2.7 Arrays

Os `arrays` podem armazenar uma coleção sequencial de tamanho fixo de elementos do mesmo tipo. Um `array` é usada para armazenar uma coleção de dados, mas geralmente é mais útil pensar em um `array` como uma coleção de variáveis do mesmo tipo. Todos os `array` consiste, em locais de memória contíguos. O endereço mais baixo corresponde ao primeiro elemento e o endereço mais alto ao último elemento.

Os `arrays` podem ser unidimensionais (como vetores), bidimensionais (como matrizes) e FORTRAN permite `arrays` com até 7 dimensões.

Os `arrays` são declarados com o atributo `dimension`. Por exemplo, para declarar um `array` unidimensional chamado vetor, de números reais contendo 5 elementos, escrevemos:

Código-fonte 32 – Declaração de array.

```
real, dimension(5) :: vetor
```

Fonte: Elaborado pelo autor.

Os elementos individuais dos `arrays` são referenciados especificando seus índices. O primeiro elemento de um `array` possui índice um. O vetor declarado no exemplo acima contem cinco variáveis reais: vetor(1), vetor(2), vetor(3), vetor(4) e vetor(5).

Para criar um `array` bidimensional 5x5 de inteiros, podemos escrever:

Código-fonte 33 – Declaração de array bidimensional.

```
integer, dimension(5, 5) :: matriz
```

Fonte: Elaborado pelo autor.

Podemos atribuir valores a membros individuais, como:

```
vetor(1) = 2.0
```

ou usando um `loop`

```
do i = 1, 5
    vetor(i) = i * 2.0
end do
```

Devido a seus objetivos computacionais, as operações matemáticas com `arrays` são diretas no Fortran. As operações com `array` da mesma forma e tamanho são muito semelhantes à álgebra de matrizes/vetores. Em vez de percorrer todos os índices com `loops`, pode-se escrever adição (e subtração):

Código-fonte 34 – Adição de arrays em Fortrab

```
real, dimension(2,3) :: A, B, C
real, dimension(5,6,3) :: D
A     = 3.     ! Atribuição de valor único a toda o array
B     = 5.     ! Escrita equivalente para atribuição
C     = A + B ! Todos os elementos de C agora têm valor 8.
D     = A + B ! O compilador gerará um erro. As formas e
               ! dimensões não são as mesmas
```

Fonte: Elaborado pelo autor.

“Fatiar” `arrays` é uma operação válida em fortran, como podemos ver no código 35 abaixo.

Código-fonte 35 – *Slicing* arrays em Fortran

```
integer :: i, j
real, dimension(3, 2) :: Mat = 0.
real, dimension(3)    :: Vec1 = 0., Vec2 = 0., Vec3 = 0.
i = 0
j = 0
do i = 1,3
  do j = 1,2
    Mat(i,j) = i+j
  end do
end do
Vec1 = Mat(:,1)
Vec2 = Mat(:,2)
Vec3 = Mat(1:2,1) + Mat(2:3,2)
```

Fonte: Elaborado pelo autor.

Fortran 90/95 fornece vários procedimentos para manipulação de `array`, mas por fugirem ao propósito elementar dessas notas, não veremos aqui.

2.2.8 Procedimentos

Um procedimento é um grupo de instruções que executam uma tarefa bem definida e podem ser chamados de seu programa. Informações (ou dados) são passados para o programa de chamada, para o procedimento como argumentos. Existem dois tipos de procedimentos -

- Funções
- Sub-rotinas

Uma função é um procedimento que retorna um único valor. Uma função não deve modificar seus argumentos. A quantidade retornada é conhecida como valor da função e é denotada pelo nome da função. A sintaxe de um função é:

```
function nome(arg1, arg2, ...)
    [ declarações, incluindo aquelas para os argumentos ]
    [ intruções ]
end function nome
```

O exemplo a seguir mostra uma função chamada `areaCirculo`. Ela calcula a área de um círculo com raio r.

Código-fonte 36 – Cáculo area de um círculo.

```
function calculaArea(r)
    ! função que calcula a area de um circulo
    ! com raio r
    implicit none

    real, parameter :: PI = 3.1415926
    real :: calculaArea
    real :: r

    calcula area = PI * r**2
end function calculaArea

program areaCirculo
    implicit none

    real :: area
    real :: raio = 2.0

    area = calculaArea(r)
    print*, "A área de um círculo com raio ", r, " é ", area.
end program areaCirculo
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte

`A área de um círculo com raio 2.0 é 12.5663710.`

Notemos que devemos especificar `implicit none` tanto no programa principal como no procedimento.

Uma sub-rotina não retorna um valor, mas pode modificar seus argumentos. Sua sintaxe é:

```
subroutine nome(arg1, arg2, ...)
    [ declarações, incluindo aquelas para os argumentos ]
    [ intruções ]
end subroutine nome
```

Precisamos chamar uma sub-rotina usando a instrução `call`. O exemplo a seguir mostra a definição e o uso de uma sub-rotina swap que altera os valores de seus argumentos.

Código-fonte 37 – Usando sub-rotina.

```
subroutine swap(x, y)
    implicit none

    real :: x, y, temp
    temp = x
    x = y
    y = temp

end subroutine swap

program main
    implicit none

    real :: a, b
    a = 2.0
    b = 3.0
    print*, "Antes de chamar swap"
    print*, "a = ", a
    print*, "b = ", b

    call swap(a, b)
    print*, "Depois de chamar swap"
    print*, "a = ", a
    print*, "b = ", b

end program main
```

Fonte: Elaborado pelo autor.

Quando compilamos e executamos o programa acima, produz o seguinte

```
Antes de chamar swap
a = 2.00000000
b = 3.00000000
Depois de chamar swap
a = 3.00000000
b = 2.00000000
```

2.2.9 Funções intrínsecas

As funções intrínsecas são algumas funções comuns e importantes fornecidas como parte da linguagem Fortran. Já vimos algumas dessas funções nas seções anteriores. Na tabela 9, fornecemos breves descrições de algumas dessas funções.

Tabela 9 – Algumas funções matemáticas

Função	Descrição
`abs(x)`	Retorna o valor absoluto de x
`acos(x)`	Retorna o arco cosseno no intervalo $(0, \pi)$, em radianos.
`asin(x)`	Retorna o arco seno no intervalo $(-\pi/2, \pi/2)$, em radianos
`atan(x)`	Retorna o arco tangente no intervalo $(-\pi/2, \pi/2)$, em radianos
`cos(x)`	Retorna o cosseno de x, x em radianos.
`cosh(x)`	Retorna o cosseno hiperbólico de x.
`exp(x)`	Retorna o exponencial de x.
`log(x)`	Retorna o logaritmo natural de x.
`log10(x)`	Retorna o logaritmo comum (base 10) de x.
`sin(x)`	Retorna o seno de x, x em radianos.
`sinh(x)`	Retorna o seno hiperbólico de x.
`sqrt(x)`	Retorna a raiz quadrada de x.
`tan(x)`	Retorna a tangente de x, x em radianos.
`tanh(x)`	Retorna a tangente hiperbólica de x.

Fonte: Adaptado de (TUTORIALS POINT, 2021)

O Fortran foi originalmente desenvolvido por uma equipe da IBM em 1957 para cálculos científicos. Desenvolvimentos posteriores o transformaram em uma linguagem de programação de alto nível. Nestas notas, aprendemos os conceitos básicos do Fortran e seu código de programação.

3 INTERPOLAÇÃO

Nesse capítulo, trataremos sobre interpolação de um polinômio de grau k a $k+1$ pontos dados. A interpolação polinomial será utilizada como uma maneira de "aproximar" uma função, ou seja, dado $f : [a,b] \to \mathbb{R}$ uma função e uma partição de seu domínio

$$a = x_0 < x_1 < \cdots < x_{k-1} < x_k = b$$

e conhecidos $k+1$ valores discretos correspondentes a esses pontos da partição

$$f(x_i) = y_i, \quad i = 0, 1, \cdots, k,$$

abordamos o problema de construir o polinômio $p_m(x)$ de menor grau que interpola todos os pontos de (x_i, y_i), ou seja,

$$p_m(x) = y_i, \quad i = 0, 1, \cdots, k,$$

Assumiremos que $a < b$ e que $k \geqslant 1$.

Dados $k+1$ pontos, podemos interpolar um polinômio de grau k, que é único (por que?). A pergunta é: quanto se perde ao se trocar a função f pelo polinômio interpolador? Para $k = 1$ o polinômio interpolador é a função de primeiro grau que passa pelo pontos $(x_0, f(x_0))$ e $(x_1, f(x_1))$. Para quaisquer valores de k, a função se anula em todos os pontos $x_0, x_1, \cdots, x_k$ da partição.

3.1 Polinômios de Lagrange

Considere o polinômio

$$(x - x_0) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_k),$$

que tem grau k e se anula em $x_0, \cdots, x_{i-1}, x_{i+1}, \cdots, x_k$. Temos que

$$L_i(x) = \frac{(x - x_0) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_k)}{(x_i - x_0) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_k)}$$

vale 1 em x_i e zero nos demais pontos $x_0, \cdots, x_{i-1}, x_{i+1}, \cdots, x_k$. A soma desses polinômios é um polinômio de grau k, logo

$$c_0 L_0(x) + c_1 L_1(x) + \cdots + c_k L_k(x)$$

é um polinômio de grau k, que vale c_0 em x_0, c_1 em x_1, $\cdots$, c_k em x_k. Portanto, basta tomar os $c_i's$ iguais a $f(x_i)$:

$$p(x) = f(x_0)L_0(x) + f(x_1)L_1(x) + \cdots + f(x_k)L_k(x).$$

Os polinômios L_i são conhecidos como polinômio de Lagrange.

Para mostrar a unicidade, suponhamos que P_k e Q_k são dois polinômios de grau menor ou igual a k interpoladores de f nos pontos da partição dada. Então o polinômio

$$R_k(x) = P_k(x) - Q_k(x)$$

anula-se, pelo menos, nos pontos $x_i, i = 0, 1, \cdots, k$. Como R_k é um polinômio de grau menor ou igual a k, ele só pode ter $k+1$ se for identicamente nulo. Logo, $P_k(x) = Q_k(x)$, para todo o x.

As expressões

$$P_k(x) = \sum_{i=0}^{k} f(x_i)L_i(x) \quad \text{e} \quad L_i(x) = \prod_{j=0, j\neq i}^{k} \frac{x - x_j}{x_i - x_j} \tag{3.1}$$

definem a fórmula de Lagrange para calcular o polinômio interpolador de f nos pontos

$$a = x_0 < x_1 < \cdots < x_{k-1} < x_k = b \tag{3.2}$$

No caso particular $k = 1$, com 2 pontos de interpolação, (x_0, y_0) e (x_1, y_1), o polinômio interpolador se reduz a

$$p_1(x) = \sum_{i=0}^{1} f(x_i)l_i(x) = f(x_0)L_0(x) + f(x_1)L_1(x) \quad \text{onde}$$
$$L_0(x) = \prod_{j=0, j\neq 0}^{1} \frac{x - x_j}{x_i - x_j} = \frac{x - x_1}{x_0 - x_1},$$
$$L_1(x) = \prod_{j=0, j\neq 1}^{1} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_1 - x_0}$$

ou seja

$$p_1(x) = \frac{x - x_1}{x_0 - x_1} y_0 + \frac{x - x_0}{x_1 - x_0} y_1 \tag{3.3}$$

uma reta passando pelos pontos.

No caso $k = 2$, o polinômio interpolador corresponde a uma parábola definido pelos três pontos x_0, y_0, (x_1, y_1) e (x_2, y_2):

$$p_2(x) = \sum_{i=0}^{2} f(x_i) l_i(x) = f(x_0) L_0(x) + f(x_1) L_1(x) + f(x_2) L_2(x) \quad \text{onde}$$

$$L_0(x) = \prod_{j=0, j\neq 0}^{2} \frac{x - x_j}{x_i - x_j} = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)},$$

$$L_1(x) = \prod_{j=0, j\neq 1}^{2} \frac{x - x_j}{x_i - x_j} = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)},$$

$$L_2(x) = \prod_{j=0, j\neq 2}^{2} \frac{x - x_j}{x_i - x_j} = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)},$$

ou seja

$$p_2(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_2 \tag{3.4}$$

A equação (3.1) é codificado no código (38). Ao receber k pontos, nos *arrays* x e y, a função Lagrange retorna o valor $p_k(x_i)$.

Código-fonte 38 – Interpolação de Lagrange

```
def lagrange(x, y, k, xi):
    """
    Calcula o Polinomio de Lagrange de k pontos no ponto xi
    x[] - coordenadas x dos pontos
    y[] - coordenadas y dos pontos
    k   - numero de pontos
    xi  - argumento
    """

    yi = 0
    for i in range(1, k+1):
        p = 1
        for j in range(1, k+1):
            if (j != i):
                p *= (xi - x[j])/(x[i] - x[j])
        yi += p * y[i]

    return yi
```

Fonte: Elaborado pelo autor.

A variante `lagrange1`, apresentada no código (39), recebe um array x_i de k_i argumentos de interpolação, para os quais ele retorna os valores $P_k(x_i)$ por meio do array y_i.

Código-fonte 39 – Interpolação de Lagrange com múltiplos argumentos

```
def lagrange1(x, y, k, xi, yi, ki):
    """
    Calcula o polinomio de Lagrange de ni pontos
    x[] - coordenadas x dos pontos
    y[] - coordenadas y do spontos
    k - numero de pontos
    xi[] - argumentos da interpolacao
    yi[] - saida (pk(xi))
    ki - numero de pontos de interpolacao
    """

    yf = [0]*k

    # calcula valores constantes
    for i in range(k):
        p = 1
        for j in range(k):
            if (j != i ):
                p *= (x[i] - x[j])
        yf[i] = y[i] / p

    # loop nos ki pontos
    for l in range(ki):
        xk = xi[l]
        yk = 0e0
        for i in range(k):
            p = 1e0
      for j in range(k):
        if (j != i):
          p *= (xk - x[j])
      yk += p * yf[i]

    yi[l] = yk
```

Fonte: Elaborado pelo autor.

Como exemplo de uso do código (38), seja dada a seguinte tabela

Tabela 10 – Valores para alguns pontos de $\ln x$

x	.40	.50	.70	.80
ln$x$	-.916291	-.693147	-.356675	-.223144

Fonte: Adaptado de (CHAPMAN, 2018)

onde vamos calcular o valor estimado de $\ln 0.60$ usando o código (38), obtendo a aproximação $\ln 0.60 \approx -0.509975$. O valor real é $\ln 0.60 = -0.510826$, um erro de 0.000850. O código vemos abaixo (40)

Código-fonte 40 – Interpolação usando Polinômio de Lagrange

```
import math

# def lagrange(...)

k = 4                         # numero de pontos
x = [0]*k
y = [0]*k

x = [.4, .5, .7, .8]
y = [-.916291, -.693147, -.356675, -.223144]

xi = .60

yi = lagrange(x, y, k, xi)    # valor estimado
a = math.log(xi)              # valor correto

print(yi)                     # -0.5099754999999999
print(a)                      # -0.5108256237659907
print(abs(yi-a))              #  0.0008501237659908067
```

Fonte: Elaborado pelo autor.

No programa do código (41), temos a interpolação de um conjunto de pontos de dados extraídos da função $f(x) = \sin x / x$ e avaliados para vários argumentos e a plotagem do gráfico

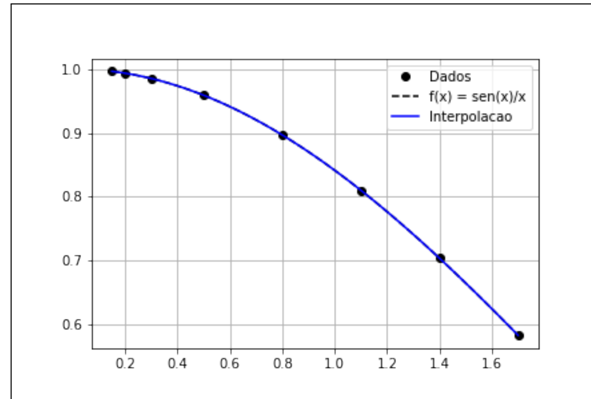
Código-fonte 41 – Interpolação usando Polinômio de Lagrange

```
import numpy as np
import matplotlib.pyplot as plt     # bibliotecas p/ plot

def lagrange1(...): ...

k = 8                               # numero de pontos
y = [0]*k
x = [.15, .2, .3, .5, .8, 1.1, 1.4, 1.7]
for i in range(k):
    y[i] = np.sin(x[i])/x[i]        # f(x) = sen(x)/x

# numero de pontos de interpolacao
ki = 100
h = abs(x[k-1]-x[0])/(ki-1)
xi = [0]*ki
for i in range(ki):
    xi[i] = x[0]+i*h

yi = [0]*ki
lagrange1(x, y, k, xi, yi, ki)

xa = np.linspace(x[0], x[k-1], 100)
ya = np.sin(xa)/xa

plt.plot(x, y, "ko", label="Dados")
plt.plot(xa, ya, "k--", label="f(x) = sen(x)/x")
plt.plot(xi, yi, "b-", label="Interpolacao")

plt.grid(True)
plt.legend()
plt.show()
```

Fonte: Elaborado pelo autor.

Na figura 4, temos a plotagem dos dados do código e sua concordância (41), usamos as bibliotecas numpy e matplotlib.

Figura 4 – Gráfico da interpolação de um conjunto de dados extraídos da função $f(x) = \sin(x)/x$.



Fonte: Elaborado pelo autor

3.2 Diferenças divididas de Newton

Suponha que $P_n(x)$ seja o enésimo polinômio de Lagrange que concorda com a função f nos pontos $x_0, x_1, \cdots, x_n$. Embora este polinômio seja único, existem representações algébricas alternativas que são úteis em certas situações. O método das diferenças divididas de Newton consiste em construir o polinômio interpolador da forma

$$P_n(x) = a_0 + a_1(x - x_0) + \cdots + a_n(x - x_0)\cdots(x - x_{n-1}), \tag{3.5}$$

para constantes apropriadas $a_0, a_1, \cdots, a_n$. Para determinar a primeira dessas constantes a_0, observemos que $P_n(x)$ passa pelo ponto $(x_0, f(x_0))$, daí

$$a_0 = P_n(x_0) = f(x_0)$$

Da mesma forma, quando $P_n(n)$ é avaliado em x_1 os únicos termos diferentes de zero em $P_n(x_1)$ são os termos constantes e lineares,

$$f(x_0) + a_1(x_1 - x_0) = P_n(x_1) = f(x_1)$$

daí

$$a_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \tag{3.6}$$

A diferença dividida de ordem zero da função f em relação a x_i, denotada por $f[x_i]$, é simplesmente o valor de f em x_i

$$f[x_i] = f(x_i)$$

As diferenças divididas restantes são definidas recursivamente; a diferença dividida de ordem 1 de f em relação a x_i e x_{i+1} é denotado por $f[x_i, x_{i+1}]$ e definido como

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i} \tag{3.7}$$

A diferença dividida de ordem 2, $f[x_i, x_{i+1}, x_{i+2}]$, é definida como

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}$$

Similarmente, após a $(k-1)$-ésima diferença dividida

$$f[x_i, x_{i+1}, x_{i+2}, \cdots, x_{i+k-1}] \quad \text{e} \quad f[x_{i+1}, x_{i+2}, \cdots, x_{i+k-1}, x_{i+k}]$$

determinamos a k-ésima diferença dividida em relação a $x_i, x_{i+1}, x_{i+2}, \cdots, x_{i+k}$

$$f[x_i, x_{i+1}, \cdots, x_{i+k-1}, x_{i+k}] = \frac{f[x_{i+1}, x_{i+2}, \cdots, x_{i+k}] - f[x_i, x_{i+1}, \cdots, x_{i+k-1}]}{x_{i+k} - x_i} \tag{3.8}$$

O processo termina com a n-ésima diferença dividida

$$f[x_0, x_1, \cdots, x_n] = \frac{f[x_1, x_2, \cdots, x_n] - f[x_0, x_1, \cdots, x_{n-1}]}{x_n - x_0} \tag{3.9}$$

Uma inspeção cuidadosa dos coeficientes obtidos nos mostra que

$$a_k = f[x_0, x_1, x_2, \cdots, x_k],$$

para cada $k = 0, 1, \cdots, n$. Isto nos permite esquematizar o método conforme apresentado na tabela (11) abaixo

O método de Newton (3.8 e 3.9) é codificado em (42). Ao receber n pontos de dados nas `listas` x e y, a função `metodoNewton` retorna os coeficientos do polinômio de interpolação.

Código-fonte 42 – Interpolação usando Método de Newton

```
1  def metodoNewton(x, y, n, p):
2      """
3      x[] - coordenadas x de pontos de dados
4      y[] - coordenadas y de pontos de dados
5      n - número de pontos de dados
6      p[] - coeficientes do polinômio de interpolação
7      """
8  
9      F = [[0] * n for _ in range(n)]
10     for i in range(n):
```

```
11            F[i][0] = y[i]
12
13        for i in range(1,n):
14            for j in range(1, i+1):
15                F[i][j] = (F[i][j-1]-F[i-1][j-1])/(x[i]-x[i-j])
16
17        for i in range(n):
18            p[i] = F[i][i]
```

Fonte: Elaborado pelo autor.

Tabela 11 – Esquema de diferenças divididas para um conjunto de pontos

x_i	$f[x_i]$	$f[x_{i-1},x_i]$	$f[x_{i-2},x_{i-1},x_i]$	$f[x_{i-3},x_{i-2},x_{i-1},x_i]$
x_0	$f[x_0]$			
		$f[x_0,x_1]=\dfrac{f[x_1]-f[x_0]}{x_1-x_0}$		
x_1	$f[x_1]$		$f[x_0,x_1,x_2]=\dfrac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0}$	
		$f[x_1,x_2]=\dfrac{f[x_2]-f[x_1]}{x_2-x_1}$		$f[x_0,x_1,x_2,x_3]=\dfrac{f[x_1,x_2,x_3]-f[x_0,x_1,x_2]}{x_3-x_0}$
x_2	$f[x_2]$		$f[x_1,x_2,x_3]=\dfrac{f[x_2,x_3]-f[x_1,x_2]}{x_3-x_1}$	
		$f[x_2,x_3]=\dfrac{f[x_3]-f[x_2]}{x_3-x_2}$		$f[x_1,x_2,x_3,x_4]=\dfrac{f[x_2,x_3,x_4]-f[x_1,x_2,x_3]}{x_4-x_1}$
x_3	$f[x_3]$		$f[x_2,x_3,x_4]=\dfrac{f[x_3,x_4]-f[x_2,x_3]}{x_4-x_2}$	
		$f[x_3,x_4]=\dfrac{f[x_4]-f[x_3]}{x_4-x_3}$		
x_4	$f[x_4]$			

Fonte: Adaptado de (BEU, 2014)

Como exemplo do método de diferenças divididas vamos encontrar o polinômio que passe pelos pontos $(1,1)$, $(2,1)$, $(3,2)$, $(4,3)$, $(5,5)$, $(6,8)$, $(7,13)$, $(8,21)$.

Código-fonte 43 – Polinômio interpolador passando pelos pontos dados.

```
1  n = 8
2  x = [1,2,3,4,5,6,7,8]
3  y = [1, 1, 2, 3, 5, 8, 13, 21]
4  p=[0]*n
5
6  metodoNewton(x1, y1, n, p)
7  print(p)
```

Fonte: Elaborado pelo autor.

Ao executar o código acima, temos a saída:

`[1, 0.0, 0.5, -0.16666666666666666, 0.08333333333333333,-0.025, 0.006944444444444445, -0.0015873015873015873]` que representa o polinômio

$$
\begin{gathered}
P(x) = 1 + \frac{(x-1)(x-2)}{2} - \frac{(x-1)(x-2)(x-3)}{6} \\
+ \frac{(x-1)(x-2)(x-3)(x-4)}{12} - \frac{(x-1)(x-2)(x-3)(x-4)(x-5)}{40} \\
+ \frac{(x-1)(x-2)(x-3)(x-4)(x-5)(x-6)}{144} \\
- \frac{(x-1)(x-2)(x-3)(x-4)(x-5)(x-6)(x-7)}{630}
\end{gathered}
$$

.

3.3 Interpolação por *Spline* Cúbico

Suponha que uma função f definida em um intervalo $[a,b]$ seja suficientemente suave e o intervalo seja subdividido pelos pontos $a \leq x_0 < x_1 < \cdots < x_{n-1} < x_n \leq b$. Uma por *spline* cúbico interpolador S de f é uma função que satisfaz as seguintes condições:

1. $S(x) \in C^2[a,b]$,
2. $S(x)$ é um polinômio de grau 3, em cada subintervalo $[x_i, x_{i+1}]$.
3. $S(x_i) = f(x_i)$ para cada $i = 0, 1, \cdots, n$, ou seja,

$$
S(x) = \begin{cases} S_1(x), & x_0 \leq x \leq x_1 \\ \quad \cdots & \\ S_i(x), & x_{i-1} \leq x \leq x_i \\ \quad \cdots & \\ S_n(x), & x_{n-1} \leq x \leq x_n \end{cases}
$$

onde $S_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ $(d_i \neq 0)$, $i = 1, \cdots, n$.

Para determinar $S(x)$ precisamo determinar a_i, b_i, c_i, d_i para cada i, de tal modo:

- $S_i(x_{i-1}) = f(x_{i-1})$ e $S_i(x_i) = f(x_i)$ para cada $i = 1, \cdots, n$.
- $S'_i(x_i) = S'_{i+1}(x_i)$ pra cada $i = 1, \cdots, n-1$.
- $S''_i(x_i) = S''_{i+1}(x_i)$ pra cada $i = 1, \cdots, n-1$.

Podemos ver que há $n + n + (n-1) + (n-1) = 4n - 2$ condições, mas precisamos determinar $4n$ coeficientes então adicionamos duas condições de contorno para resolver este problema.

1. $S''(x_0) = S''(x_n) = 0$, ou

2. $S'(x_0) = f'(x_0)$ e $S'(x_n) = f'(x_n)$

As condições dadas acimas são suficientes para nossos objetivos.

Existem vários métodos que podem ser usados para encontrar a função *spline* $S(x)$ de acordo com suas condições correspondentes. Uma vez que existem $4n$ coeficientes para determinar com $4n$ condições, podemos facilmente inserir os valores que conhecemos nas $4n$ condições e então resolver o sistema de equações. Observemos que todas as equações são lineares em relação aos coeficientes, então isso é viável e os computadores podem fazer isso muito bem.

Os coeficientes a_i, b_i, c_i e d_i das *splines* podem ser determinados, em princípio, a partir das condições de continuidade e interpolação nos números x_i:

$$S(x_i) = f(x_i) \tag{3.10}$$

A segunda derivada do *spline*, $S_i''(x)$, é uma função linear definida no intervalo $[x_{i-1}, x_i]$ pela equação

$$\frac{S_i''(x) - S''(x_{i-1})}{x - x_{i-1}} = \frac{S''(x_i) - S''(x_{i-1})}{x_i - x_{i-1}}, \tag{3.11}$$

Os termos $x_i - x_{i-1}$ são usados repetidamente neste desenvolvimento, por isso é conveniente introduzir a notação mais simples

$$h_i = x_i - x_{i-1}$$

para $i = 1, \cdots, n-1$. Da equação (3.11), temos que

$$S_i''(x) = \frac{(x - x_{i-1})S''(x_i) + (x_i - x)S''(x_{i-1})}{h_i} \tag{3.12}$$

Integrando a Equação 3.12 duas vezes, obtém-se sucessivamente a primeira derivada e a própria *spline*:

$$S_i'(x) = \frac{(x - x_{i-1})^2 S''(x_i) - (x_i - x)^2 S''(x_i)}{2h_i} + C_i^1, \tag{3.13}$$

$$S_i(x) = \frac{(x - x_{i-1})^3 S''(x_i) + (x_i - x)^3 S''(x_{i-1})}{6h_i} + C_i^1 x + C_i^2. \tag{3.14}$$

As constantes de integração C_i^1 e C_i^2 podem ser determinadas usando

$$S_i(x_{i-1}) = f(x_{i-1}), \qquad S_i(x_i) = f(x_i)$$

resultando no sistema:

$$\begin{cases} \frac{h_i^2}{6} S''(x_{i-1}) + C_i^1 x_{i-1} + C_i^2 = f(x_{i-1}) \\ \frac{h_i^2}{6} S''(x_i) + C_i^1 (x_i) + C_i^2 = f(x_i) \end{cases}$$

Resolvendo esse sistema para C_i^1 e C_i^2 resulta

$$C_i^1 = \frac{f(x_i) - f(x_{i-1})}{h_i} - \frac{h_i}{6}[S''(x_i) - S''(x_{i-1})], \quad (3.15)$$

$$C_i^2 = \frac{x_i f(x_{i-1}) - x_{i-1} f(x_i)}{h_i} + \frac{h_i}{6}[x_{i-1} S''(x_i) - x_i S''(x_{i-1})] \quad (3.16)$$

Após a substituição das constantes de integração de volta na equação (3.14) obtém-se os coeficientes polinomiais da *spline* $S_i(x)$ em termos das segundas derivadas $S''(x_i)$ nos pontos de interpolação:

$$\begin{cases} d_i &= \dfrac{S''(x_i) - S''(x_{i-1})}{6h_i}, \\ c_i &= \dfrac{x_i S''(x_{i-1}) - x_{i-1} S''(x_i)}{2h_i}, \\ b_i &= \dfrac{x_{i-1}^2 S''(x_i) - x_i^2 S''(x_{i-1})}{2h_i} + \dfrac{f(x_i) - f(x_{i-1})}{h_i} - d_i h_i^2, \\ a_i &= \dfrac{x_i^3 S''(x_{i-1}) - x_{i-1}^3 S''(x_i)}{6h_i} + \dfrac{x_i f(x_{i-1}) - x_{i-1} f(x_i)}{h_i} - \dfrac{c_i h_i^2}{3}. \end{cases} \quad (3.17)$$

Para a definição completa das *splines*, ainda é necessário determinar os valores $S''(x_i)$ das segundas derivadas e isso pode ser feito usando a continuidade da primeira derivada,

$$S_i'(x_i) = S_{i+1}'(x_i)$$

Usando as equações (3.13) e (3.15), resulta em,

$$\frac{h_i}{6} S''(x_{i-1}) + \frac{h_i + h_{i+1}}{3} S''(x_i) + \frac{h_{i+1}}{6} S''(x_{i+1}) = \frac{f(x_{i+1}) - f(x_i)}{h_{i+1}} - \frac{f(x_i) - f(x_{i-1})}{h_i}. \quad (3.18)$$

O sistema de equações (3.18) compreende $n-1$ equações para as $n+1$ derivadas $S''(x_i)$ e, consequentemente, para sua determinação inequívoca, duas equações adicionais precisam ser fornecidas. Usando

$$S''(x_0) = S''(x_n) = 0$$

ficamos com o seguinte sistema linear descrito pela equação vetorial $A\mathbf{x} = \mathbf{b}$, onde A é a matriz $(n+1) \times (n+1)$

$$A = \begin{bmatrix} 1 & 0 & 0 & \cdots & & 0 \\ h_1 & 2(h_1 + h_2) & h_2 & & & \\ 0 & h_2 & 2(h_2 + h_3) & h_3 & & \vdots \\ \vdots & & & \ddots & & \\ & & & h_{n-1} & 2(h_{n-1} + h_n) & h_n \\ 0 & & \cdots & 0 & 0 & 1 \end{bmatrix} \quad (3.19)$$

e **b** e **x** são os vetores

$$\mathbf{b} = 6 \begin{bmatrix} 0 \\ \frac{f(x_2)-f(x_1)}{h_2} - \frac{f(x_1)-f(x_0)}{h_1} \\ \vdots \\ \frac{f(x_n)-f(x_{n-1})}{h_n} - \frac{f(x_{n-1})-f(x_{n-2})}{h_{n-1}} \\ 0 \end{bmatrix} \quad \text{e} \quad \mathbf{x} = \begin{bmatrix} S''(x_0) \\ S''(x_1) \\ \vdots \\ S''(x_n) \end{bmatrix}$$

Essencialmente, a implementação da interpolação de *spline* cúbica envolve:

1. Avaliação da segunda derivada $S''(x_i)$ resolvendo o sistema acima.
2. Cálculo dos coeficientes *spline* a_i, b_i, c_i e d_i usando as equações (3.17),
3. Avaliação do interpolante para um determinado conjunto de argumentos.

O código abaixo implementa essas ideias. A entrada é representada pelos `array` `x` e `y`, transmitindo as `n` coordenadas dos pontos interpolados. Para resolver o sistema (3.19) para as segundas derivadas das *splines* nos nós de interpolação, usamos um método não abordado nesse trabalho.

Código-fonte 44 – Interpolação usando Método de *spline* cúbico

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def splineMetodo(x, y, n, a, b, c, d):
5    """
6    Calcula os coeficientes das splines cúbicas para n pontos de dados
7    x[] - coordenadas x dos pontos
8    y[] - coordenadas y dos pontos
9    n   - número de pontos
10   a[], b[], c[], d[] -
11     cooeficientes da spline cúbica
12   """
13
14   h = [0]*(n+1)
15   bb = [0]*(n+1)
16
17   # usado para resolver sistema
18   # método não abordado nesse trabalho
19   l = [0]*(n+1)
```

```
20    u = [0]*(n+1)
21    z = [0]*(n+1)
22
23    S = [0]*(n+1)
24
25    for i in range(1,n+1):
26      h[i] = x[i]-x[i-1]
27
28    for i in range(1, n):
29      btmp1 = (y[i+1]-y[i])/h[i+1]
30      btmp2 = (y[i]-y[i-1])/h[i]
31      bb[i] =6*(btmp1-btmp2)
32
33    l[0] = 1
34    u[0] = 0
35    z[0] = 0
36
37    for i in range(1, n):
38      l[i] = 2*(x[i+1]-x[i-1])-h[i]*u[i-1]
39      u[i] = h[i+1]/l[i]
40      z[i] = (bb[i]-h[i]*z[i-1])/l[i]
41
42    l[n] = 1
43    z[n] = 0
44    S[n] = 0
45
46    for i in range(n, 0, -1):
47      S[i-1] = z[i-1] - u[i]*S[i]
48
49      d[i-1] = (S[i] - S[i-1])/(6*h[i])
50      c[i-1] = (x[i]*S[i-1]-x[i-1]*S[i])/(2*h[i])
51      b[i-1] = (x[i-1]**2*S[i]-x[i]**2*S[i-1])/(2*h[i])+\
52             (y[i]-y[i-1])/h[i]-d[i-1]*h[i]*h[i]
53      a[i-1] = (x[i]**3*S[i-1]-x[i-1]**3*S[i])/(6*h[i])+\
54             (x[i]*y[i-1]-x[i-1]*y[i])/h[i] - (c[i-1]*h[i]**2)/3.
55
56
57  def polinomio(a, b, c, d, p, x):
58    return a[p]+x*(b[p]+x*(c[p]+x*d[p]))
```

```python

x = [0, 1, 2, 3]
y = np.exp(x)
n = 3

a = [0]*n
b = [0]*n
c = [0]*n
d = [0]*n

splineMetodo(x, y, n, a, b, c, d)
print("a = ", a)
print("b = ", b)
print("c = ", c)
print("d = ", d)

x1=np.arange(0, 3, .1)
plt.plot(x1, np.exp(x1))

xx1=np.arange(0,1.1,.1)
xx2=np.arange(1,2.1,.1)
xx3=np.arange(2,3,.1)

plt.plot(xx1, polinomio(a, b, c, d, 0, xx1))
plt.plot(xx2, polinomio(a, b, c, d, 1, xx2))
plt.plot(xx3, polinomio(a, b, c, d, 2, xx3))
plt.legend([r"$y = e^x$", "[0, 1]","[1, 2]","[2, 3]"])

plt.grid(True)
plt.show()
```

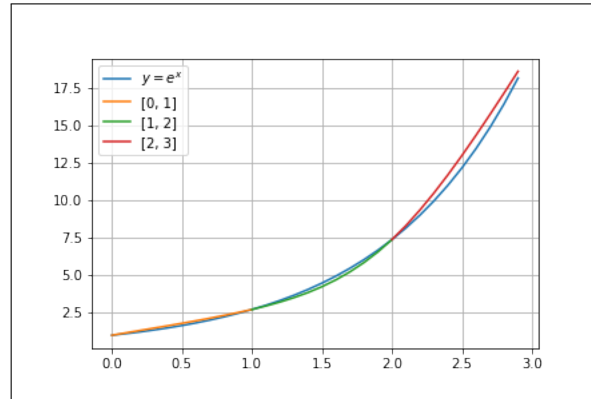
Fonte: Elaborado pelo autor.

Aproximando a exponencial $f(x) = e^x$, usando os pontos $(0,1)$, $(1,e)$, $(2,e^2)$ e $(3,e^3)$ usando método *spline*, obtemos:

$$S(x) = \begin{cases} 1.0 + 1.60833x - 0.16492x^2 + 0.27487x^3 & \text{para} \quad x \in [0,1] \\ -0.63312 + 6.13864x - 4.51070x^2 1.72346x^3 & \text{para} \quad x \in [1,2] \\ 28.63663 - 37.83076x + 17.49020x^2 - 1.94336x^3 & \text{para} \quad x \in [2,3] \end{cases}$$

A *spline* e sua concordância com $f(x) = e^x$ são mostrados na figura (5)

Figura 5 – Aproximando função $f(x) = e^x$ pelo método de *spline* cúbico.



Fonte: Elaborado pelo autor

3.4 Implementação em Fortran

Os códigos seguintes mostram a implementações FORTRAN equivalentes das funções Python desenvolvidas durante esse capítulo. As rotinas correspondentes têm nomes, parâmetros e funcionalidades idênticos.

Código-fonte 45 – Interpolação de Lagrange

```
function lagrange(x, y, k, xi)
    ! Calcula o Polinomio de Interpolacao de Lagrange
    ! de k pontos no ponto xi
    ! x[] - coordenadas x dos pontos
    ! y[] - coordenadas y dos pontos
    ! k   - numero de pontos
    ! xi  - argumento

    implicit none
    real ( kind = 8 ), dimension(k) :: x, y
    integer :: k, i, j
    real ( kind = 8 ) :: xi, yi, lagrange, p

    yi = 0
    do i = 1, k
        p = 1.e0
        do j = 1, k
                if ( j /= i ) then
                        p = p*(xi - x(j))/(x(i) - x(j))
```

```
20                end if
21          end do
22
23          yi = yi + p * y(i)
24      end do
25
26      lagrange = yi
27
28  end function lagrange
```

Fonte: Elaborado pelo autor.

Código-fonte 46 – Interpolação de Lagrange com múltiplos argumentos

```
1   subroutine lagrange1(x, y, k, xi, yi, ki)
2       ! Calcula o polinomio de Lagrange de ni pontos
3       ! x[] - coordenadas x dos pontos
4       ! y[] - coordenadas y do spontos
5       ! k - numero de pontos
6       ! xi[] - argumentos da interpolacao
7       ! yi[] - saida (pk(xi))
8       ! ki - numero de pontos de interpolacao
9
10      implicit none
11
12      integer :: k, ki
13      real ( kind = 8 ), dimension(k) :: x, y, xi, yi
14
15      real ( kind = 8 ), dimension(k) :: yf
16      real ( kind = 8 ) :: p, xk, yk
17      integer :: i, j , l
18
19      ! calcula valores constantes
20      do i = 1, k
21          p = 1.e0
22          do j = 1, k
23              if ( j /= i ) then
24                  p = p * (x(i)-x(j))
25              end if
```

```
26          end do
27          yf(i) = y(i)/p
28      end do
29
30      ! loop nos ki pontos
31      do l = 1, ki
32          xk = xi(l)
33          yk = 0.e0
34          do i = 1, k
35              p = 1.e0
36              do j = 1, k
37                  if ( j /= i ) then
38                      p = p * (xk - x(j))
39                  end if
40              end do
41              yk = yk + p * yf(i)
42
43          end do
44          yi(l) = yk
45      end do
46
47  end subroutine
```

Fonte: Elaborado pelo autor.

Código-fonte 47 – Interpolação usando método de Newton

```
1   subroutine metodoNewton(x, y, n, p)
2       implicit none
3
4       integer :: n
5
6       real ( kind = 8 ), dimension(n) :: x, y
7       real ( kind = 8 ), dimension(n) :: p
8       real ( kind = 8 ), dimension(0:n-1, 0:n-1) :: F
9
10      integer :: i, j
11
12      do i = 0, n-1
```

```
13          F(i, 0) = y(i+1)
14      end do
15
16      do i = 1, n-1
17          do j = 1, i
18              F(i, j) = (F(i, j-1) - F(i-1, j-1))/(x(i+1)-x(i-j+1))
19          end do
20      end do
21
22      do i = 0, n-1
23          p(i+1) = F(i, i)
24      end do
25
26  end subroutine metodoNewton
27
28  program main
29      implicit none
30
31      integer, parameter :: n = 8
32      integer :: i
33
34      real ( kind = 8 ), dimension(n) :: x, y, p
35
36      x = (/1, 2, 3, 4, 5, 6, 7, 8/)
37      y = (/1, 1, 2, 3, 5, 8, 13, 21/)
38      p = (/0, 0, 0 ,0 ,0 ,0 ,0 ,0/)
39
40      call metodoNewton(x, y, n, p)
41
42      do i = 1, n
43          print*, p(i)
44      end do
45
46  end program main
```

Fonte: Elaborado pelo autor.

Código-fonte 48 – Interpolação usando Método de *spline* cúbico

```
subroutine splineMetodo(x, y, n, a, b, c, d)
    ! Calcula os coeficientes das splines cúbicas para n pontos de
      dados
    ! x[] - coordenadas x dos pontos
    ! y[] - coordenadas y dos pontos
    ! n   - número de pontos
    ! a[], b[], c[], d[] -
    !   cooeficientes da spline cúbica

    implicit none

    integer :: n
    real ( kind = 8 ), dimension(0:n) :: x, y
    real ( kind = 8 ), dimension(n) :: a, b, c, d

    real ( kind = 8 ), dimension(0:n) :: h, bb

    ! usado para resolver o sistema
    ! método não abordado nesse trabalho
    real ( kind = 8 ), dimension(0:n) :: l, z, u, S

    integer :: i
    do i = 1, n
      h(i) = x(i)-x(i-1)
    end do

    do i = 1, n-1
      bb(i) = 6*((y(i+1)-y(i))/h(i+1)-(y(i)-y(i-1))/h(i))
    end do

    l(0) = 1.e0
    u(0) = 0.e0
    z(0) = 0.e0
    do i = 1, n-1
      l(i) = 2*(x(i+1)-x(i-1))-h(i)*u(i-1)
      u(i) = h(i+1)/l(i)
      z(i) = (bb(i)-h(i)*z(i-1))/l(i)
    end do
```

```
38
39      l(n) = 1.e0
40      z(n) = 0.e0
41      S(n) = 0.e0
42      do i = n, 1, -1
43        S(i-1) = z(i-1) - u(i)*S(i)
44
45        d(i) = (S(i) - S(i-1))/(6*h(i))
46        c(i) = (x(i)*S(i-1)-x(i-1)*S(i))/(2*h(i))
47        b(i) = (x(i-1)*x(i-1)*S(i)-x(i)*x(i)*S(i-1))/(2*h(i))+&
48             &(y(i)-y(i-1))/h(i)-d(i)*h(i)*h(i)
49        a(i) = (x(i)*x(i)*x(i)*S(i-1)-x(i-1)*x(i-1)*x(i-1)*S(i))/(6*h(i
           ))&
50             &+(x(i)*y(i-1)-x(i-1)*y(i))/h(i)-(c(i)*h(i)*h(i))/3.e0
51      end do
52  end subroutine splineMetodo
53
54  program main
55      implicit none
56
57      integer, parameter :: n = 3
58      real ( kind = 8 ), dimension(n) :: a, b, c, d
59      real ( kind = 8 ), dimension(0:n) :: x, y
60      integer :: i
61
62      x = (/0.e0, 1.e0, 2.e0, 3.e0/)
63      do i = 0, n
64        y(i) = exp(x(i))
65      end do
66
67      call splineMetodo(x, y, n, a, b, c, d)
68
69      do i = 1, n
70        print*, a(i), b(i), c(i), d(i)
71      end do
72  end program main
```

Fonte: Elaborado pelo autor.

4 INTEGRAÇÃO DE FUNÇÕES

Para obter o valor da integral definida

$$\int_a^b f(x)\,dx,$$

basta achar uma função $F(x)$ tal que $F'(x) = f(x)$|, de forma que

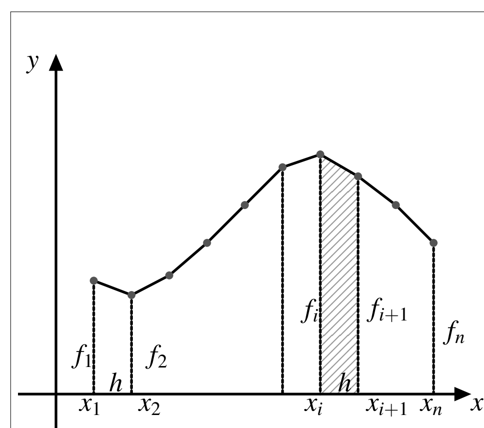
$$\int_a^b f(x)\,dx = F(b) - F(a).$$

Se $f(x)$ for desconhecida, exceto em alguns pontos ou não for integrável, ou no caso em que $f(x)$ é conhecido, mas a forma de $f(x)$ é tão complicada que é muito difícil integrá-la e encontrar um valor, podemos tentar encontrar o valor numérico da integral até um grau desejado de precisão. Este problema é denominado integração numérica ou quadratura.

4.1 O Método dos Trapézios

Partindo da interpretação geométrica da integral definida, como a área delimitada pelo gráfico de uma função f, o eixo x e pelas retas verticais $x = a$ e $x = b$, a abordagem algorítmica mais simples é substituir o gráfico de f por uma linha poligonal (como mostrado na figura 6) e somar as áreas trapezoidais assim formadas.

Figura 6 – Área sob uma curva



Fonte: Elaborado pelo autor.

Dividimos o intervalo $[a, b]$ em intervalos de tamanhos iguais, denotado por $x_i = a, x_1, \cdots, x_{n-1}, x_n = b$ de modo que

$$x_i = x_0 + ih, \quad i = 0, 1, \cdots, n,$$

onde

$$h = (b-a)/n.$$

Substituindo as integrais de cada um dos n subintervalos (x_i, x_{i+1}) pela correspondente área trapezoidal, a integral será, aproximadamente

$$\int_b^a f(x)dx \approx \frac{h}{2}(f_0+f_1)+\cdots+\frac{h}{2}(f_i+f_{i+1})+\cdots+\frac{h}{2}(f_{n-1}+f_n).$$

Colocando h em evidência e organizando, ficamos

$$\int_b^a f(x)dx \approx h\left[\frac{f_0}{2}+\sum_{i=1}^{n-1} f_i+\frac{f_n}{2}\right].$$

Nosso resultado será tanto mais preciso quanto menor for o tamanho de h.

O valor da integral de uma determinada função f num intervalo $[a,b]$ pela fórmula do trapézio pode ser dado de acordo com o seguinte implementação em python3.

Código-fonte 49 – Integração de uma função usando o método dos trapézios

```
def regraTrapz(func, a, b, n):
    """
    Calcula a integral da funcao func no intervalo
    [a,b] usando o metodo dos trapezios com n pontos
    """
    h = (b-a)/n
    soma = (func(a)+func(b))/2
    for i in range(1,n):
        soma += func(a+i*h)

    return h*soma
```

Fonte: Elaborado pelo autor.

Para exemplificar o uso do método, ilustremos com um exemplo cujo resultado é bem conhecido.

$$\int_0^1 \frac{1}{1+x^2}dx = \frac{\pi}{4}. \tag{4.1}$$

Como ainda não falamos em estimativas de erro para o método, nossa escolha em relação ao tamanho dos intervalos da partição e ao número de algarismos significativos será arbitrária.

Dividindo o intervalo $[0,1]$ em 10 intervalos iguais, e usando a implementação acima, obtemos que

$$\pi \approx 3.1399, \quad \text{com 5 algarismos significativos},$$

valor à distância de aproximadamente 1.6×10^{-3} do valor verdadeiro.

4.2 O Método de Simpson

Antes, aproximamos a função f, em cada subintervalo, por uma reta coincidindo com a função nos extremos. Agora, consideraremos polinômios quadráticos como forma de aproximar a função.

Com o objetivo de calcular a integral definida

$$\int_b^a f(x)dx,$$

começamos dividindo o intervalo $[a,b]$ em n subintervalos de tamanhos

$$h = (b-a)/n,$$

de tal modo que

$$x_i = a + ih, \quad i = 0, 1, \cdots, n.$$

Assumindo que os valores $f_i \equiv f(x_i)$ são conhecidos, aproximamos a função f por um polinômio de grau menor ou igual a n, usando interpolação polinomial de Lagrange, tal que $P_n(x_i) = f_i$

$$P_{n-1}(x) = \sum_{i=0}^{n} \frac{\prod_{j \neq i}^{n}(x - x_j)}{\prod_{j \neq i}^{n}(x_i - x_j)} f_i. \tag{4.2}$$

Fazendo a mudança de variável $t = (x-a)/h$, podemos escrever $x = a + th$, assim a expressão acima fica

$$\begin{aligned}
\prod_{j \neq i}^{n}(x - x_j) &= h^{n-1} \prod_{j \neq i}^{n}(t - j), \\
\prod_{j \neq i}(x_i - x_j) &= h^{n-1} \prod_{j \neq i}^{n}(i - j) = (-1)^{n-i} h^{n-1} \prod_{j=0}^{i-1}(i - j) \prod_{j=i+1}^{n}(j - i) \\
&= (-1)^{n-i} h^{n-1} i!(n-i)!.
\end{aligned}$$

O polinômio toma a forma

$$P_n(x) = \sum_{i=0}^{n} \frac{\prod_{i\neq j}^{n}(t-j)}{(-1)^{n-i} i!(n-i)!} f_i, \tag{4.3}$$

e obtemos a aproximação da integral:

$$\int_b^a f(x)dx \approx \int_b^a P_n(x)dx = \sum_{i=0}^{n} A_i f_i, \tag{4.4}$$

onde os coeficientes A_i são dados por (e usando a mudança de variável):

$$A_i = \int_b^a \frac{\prod_{i\neq j}^{n}(t-j)}{(-1)^{n-i} i!(n-i)!} = \frac{h}{(-1)^{n-i} i!(n-i)!} \int_0^n \prod_{j\neq i}^{n}(t-j)dt. \tag{4.5}$$

Notemos que os coeficientes (ou pesos) A_i dependem somente de n; em particular, não dependem da função f a ser integrada nem do intervalo de integração.

Se n=1, então

$$A_0 = \frac{h}{(-1)^{1-0} 0!(1-0)!} \int_0^1 \prod_{j\neq 0}^{1}(t-j)dt = -h\int_0^1 (t-1)dt = \frac{h}{2},$$
$$A_1 = \frac{h}{(-1)^{1-1} 1!(1-1)!} \int_0^1 \prod_{j\neq 1}^{1}(t-j)dt = h\int_0^1 t dt = \frac{h}{2},$$

e obtemos

$$\int_{x_1}^{x_2} f(x)dx \approx \frac{h}{2}(f_1+f_2). \tag{4.6}$$

Dividindo o intervalo $[a,b]$ em n subintervalo de igual comprimento $\frac{b-a}{n}$, decompomos a integral numa soma de n parcelas,

$$\int_b^a f(x)dx = \int_a^{x_1} f(x)dx + \cdots + \int_{x_{n-1}}^{b} f(x)dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx \tag{4.7}$$

A cada uma dessas parcelas da soma 4.7 podemos aplicar a fórmula 4.6, isto é,

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{f(x_i)+f(x_{i+1})}{2} h.$$

Assim, obtêm-se

$$\int_b^a f(x)dx \approx \sum_{i=0}^{n-1} \frac{f(x_i)+f(x_{i+1})}{2} h. \tag{4.8}$$

O somatório 4.8 pode ser representado na forma

$$\int_b^a f(x)dx \approx h\left[\frac{f(a)}{2} + \sum_{i=1}^{n-1} f(x_i) + \frac{f(b)}{2}\right], \tag{4.9}$$

obtido anteriormente de maneira puramente geométrica.

Para $n = 2$, os correspondentes coeficientes são dados por

$$A_0 = \frac{h}{(-1)^{2-0}0!(2-0)!}\int_0^2 \prod_{j\neq 0}^{2}(t-j)dt = \frac{h}{2}\int_0^2 (t-1)(t-2)dt = \frac{h}{3},$$
$$A_1 = \frac{h}{(-1)^{2-1}1!(2-1)!}\int_0^2 \prod_{j\neq 1}^{2}(t-j)dt = -h\int_0^2 t(t-2)dt = \frac{4h}{3},$$
$$A_2 = \frac{h}{(-1)^{2-2}2!(2-2)!}\int_0^2 \prod_{j\neq 2}^{2}(t-j)dt = \frac{h}{2}\int_0^2 t(t-1)dt = \frac{h}{3},$$

resultando

$$\int_a^b f(x)dx \approx \frac{h}{3}(f_0 + 4f_1 + f_2). \tag{4.10}$$

Tal como se fez para a regra dos trapézios (ver seção 4.1), subdividimos o intervalo $[a,b]$ em n partes. Dado que a regra de Simpson utiliza 3 pontos, o número $n \geqslant 2$ deverá ser par.

Em cada um dos subintervalos

$$[x_i, x_{i+2}] = [a + ih, a + (i+2)h]$$

é aplicada a equação 4.10. Tem-se

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{h}{3}[(f_0 + 4f_1 + f_2) + (f_2 + 4f_3 + f_4) + \cdots + (f_{n-2} + 4f_{n-1} + f_n)]\\ &= \frac{h}{3}[f_0 + f_n + 4(f_1 + f_3 + \cdots + f_{n-1}) + 2(f_2 + f_4 + \cdots + f_{n-2})].\end{aligned}$$

Fazendo

$$\sigma_1 = \sum_{i=1}^{n/2} f(x_{2i-1}), \quad \sigma_2 = \sum_{i=1}^{n/2-1} f(x_{2i}), \tag{4.11}$$

obtemos a fórmula da Regra de Simpson:

$$\int_a^b f(x)dx \approx \frac{h}{3}(f_0 + 4\sigma_2 + 2\sigma_1 + f_n).$$

Usando as fórmulas 4.11 e 4.2, implementamos o código 50 abaixo. Os argumentos da função `regraSimpson` são: `func` recebe o nome da função que será integrada, enquanto `a` e `b` são os limites de integração. Se o número de pontos `n` for ímpar, ele é automaticamente corrigido para o valor par mais alto, de modo que a fórmula de Simpson possa ser aplicada corretamente, como descrito no texto.

Código-fonte 50 – Integração de uma função usando o método de Simpson

```
def regraSimpson(func, a, b, n):
    """
    Calcula a integral da funcao func no intervalo
    [a,b] usando o metodo de Simpson com n (par) pontos
    """

    if (n % 2 == 1):                    #verifica se n e par
        n += 1

    h = (b-a)/n
    s1 = s2 = 0e0
    for i in range(1,n//2):             #soma em indice par
        s1 += func(a+2*i*h)

    for i in range(1,n//2+1):           #soma em indice impar
      s2 += func(a+(2*i-1)*h)

    return (h/3)*(func(a)+4*s2+2*s1+func(b))
```

Fonte: Elaborado pelo autor.

Para comparar com a regra do Trapézio, calculemos a mesma integral 4.1, usando a mesma divisão de intervalos. Usaremos 9 algarismos significativos. Obtemos

$$\pi = 4\int_0^1 \frac{1}{1+x^2}dx \approx 3.14159261,$$

com erro de 4.0×10^{-8}, resultado bem melhor que o obtido anteriormente.

4.3 Quadratura Adaptativa

As implementações `regraTrapz` e `regraSimpson` apresentadas na seção anterior além de exigirem o número de pontos de integração, não fornecem uma estimativa de erro para a integral. Pode-se implementar um procedimento automático de quadratura, no qual todos os subintervalos de $[a,b]$ são continuamente subdivididos até que uma precisão suficiente seja alcançada. No entanto essa abordagem não é necessário em regiões onde a função é suave.

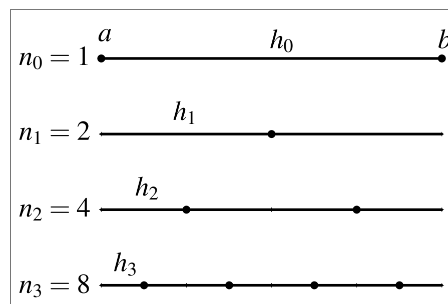
Uma alternativa é a *quadratura adaptativa*. A quadratura adaptativa é uma técnica em que o intervalo $[a,b]$ é dividido em n subintervalos $[x_i, x_{i+1}]$, para $i = 0, 1, \cdots, n$ e uma regra de quadratura, como a regra trapezoidal, é usada em cada subintervalo para calcular

$$I_f(f) = \int_{x_i}^{x_{n+1}} f(x)dx.$$

No entanto, na quadratura adaptativa, um subintervalo $[x_i, x_{i+1}]$ é subdividido se for determinado que a regra da quadratura não calculou $I_i(f)$ com precisão suficiente.

Vamos estabelecer uma relação recursiva entre as aproximações consecutivas. De acordo com a Figura 7, que mostra as grades de integração para os primeiros quatro estágios do processo de redução pela metade, a avaliação do integrando é necessária somente nos novos pontos de integração (indicados por círculos) que dividem os subintervalos anteriores. Os valores da função já empregada são reutilizados implicitamente pela aplicação da relação de recorrência entre duas aproximações consecutivas da integral.

Figura 7 – Divisão dos intervalos



Fonte: Adaptado de (BEU, 2014)

As três primeiras aproximações produzidas pela regra trapezoidal são

$$\begin{aligned}
T_0 &= h_0 \left[\frac{f(a)}{2} + \frac{f(b)}{2} \right], \quad h_0 = b - a, \\
T_1 &= h_1 \left[\frac{f(a)}{2} + f(a+h_1) + \frac{f(b)}{2} \right], \quad h_1 = \frac{h_0}{2}, \\
T_2 &= h_2 \left[\frac{f(a)}{2} + f(a+h_2) + f(a+2h_2) + f(a+3h_2) + \frac{f(b)}{2} \right], \quad h_2 = \frac{h_2}{2}.
\end{aligned} \tag{4.12}$$

A sequência acima pode ser escrita como

$$
\begin{aligned}
T_0 &= \frac{h_0}{2}[f(a)+f(b)], \quad h_0 = b-a, \quad n_0 = 1, \\
T_k &= \frac{1}{2}\left[T_{k-1} + h_{k-1}\sum_{i=1}^{n_{k-1}} f\left(a+\left(i-\frac{1}{2}\right)h_{k-1}\right)\right], \\
h_k &= h_{k-1}/2, \quad n_k = 2n_{k-1}, \quad k = 1,2,\cdots,
\end{aligned}
\tag{4.13}
$$

onde $h_k = (b-a)/n_k$ é o espaçamento após a etapa de divisão k, com $n_k = 2^k$ representando o número correspondente de subintervalos ou, equivalentemente, o número de novos pontos de integração relevantes para a próxima aproximação da integral.

O processo recursivo (4.13) é continuado até que duas aproximações consecutivas se tornem indistinguíveis para uma dada precisão, isto é, até que sua diferença relativa, $|T_k - T_{k-1}/|T_k|$ seja menor que uma tolerância pré-definida ε. Para evitar a divisão na situação particular quando $T_k = 0$, o critério de convergência pode ser convenientemente formulado como

$$
|T_k - T_{k-1}| \leqslant \varepsilon |T_k|. \tag{4.14}
$$

A função `integralTrapzAdaptive` (código 51) implementa o algoritmo adaptativo descrito com base na regra trapezoidal.

Código-fonte 51 – Integração de uma função usando o método dos Trapézio Adaptativo

```
1  def integralTrapzAdaptive(func, a, b, eps = 1e-6):
2    """
3    Calcula a integral da funcao func no intervalo [a,b]
4    com precisao eps usando o metodo trapezoidal adaptativo
5    """
6
7    kmax = 30   # numero maximo de interacoes
8    h = b-a
9    n = 1
10   t0 = 0.5*h*(func(a) + func(b))    #aproximacao inicial
11
12   for k in range(1, kmax+1):
13     sumf = 0e0
14     for i in range(1,n+1):
15       sumf += func(a+(i-0.5)*h)
```

```
16
17      t = 0.5*(t0 + h*sumf)        #nova aproximacao
18      if (k > 1):                  #checando a convergencia
19        if (abs(t-t0) <= eps*abs(t)):
20          break
21        if (abs(t) <= eps and abs(t) <= abs(t-t0)):
22          break
23
24      h *= 0.5
25      n *= 2
26      t0 = t
27
28    if (k >= kmax):
29      print("Maximo de interacoes excedido!")
30
31    return t
```

Fonte: Elaborado pelo autor.

A partir das três primeiras aproximações da regra trapezoidal (4.12), aproximações consecutivas da fórmula de Simpson podem ser expressas da seguinte forma

$$\begin{aligned} S_1 &= \frac{h_1}{3}[f(a)+4f(a+h_1)+f(b)] = \frac{4T_1-T_0}{3}, \\ S_2 &= \frac{h_2}{3}[f(a)+4f(a+h_2)+2f(a+h_2)+4f(a+3h_2)+f(b)] = \frac{4T_2-T_1}{3}. \end{aligned} \tag{4.15}$$

Daí, obtemos a relação

$$S_k = \frac{4T_k - T_{k-1}}{3}. \tag{4.16}$$

Assim, segue que o algoritmo recursivo com base na regra de Simpson pode ser descrito ainda em termos de aproximações trapezoidais:

$$\begin{aligned} T_0 &= \frac{h_0}{2}[f(a)+f(b)], \quad h_0 = b-a, \quad n_0 = 1, \\ T_k &= \frac{1}{2}\left[T_{k-1} + h_{k-1}\sum_{i=1}^{n_{k-1}} f\left(a + \left(i-\frac{1}{2}\right)h_{k-1}\right)\right], \\ S_k &= \frac{4T_k - T_{k-1}}{3}, \quad h_k = h_{k-1}/2, \quad n_k = 2n_{k-1}, \quad k = 1,2,\cdots. \end{aligned} \tag{4.17}$$

Código-fonte 52 – Integração de uma função usando o método de Simpson Adaptativo

```
def integralSimpsonAdaptive(func, a, b, eps = 1e-6):
  """
  Calcula a integral da funcao func no intervalo [a, b] com
  precisao eps usando o metodo de Simpson adaptativo
  """

  kmax = 30                     # numero maximo de interacoes
  h = b - a
  n = 1
  s0 = t0 = 0.5*h*(func(a) + func(b))   # aproximacao inicial

  for k in range(1,kmax+1):
    sumf = 0e0
    for i in range(1,n+1):
      sumf += func(a+(i-0.5)*h)

    t = 0.5*(t0 + h*sumf)
    s = (4*t - t0)/3            # nova aproximacao

    if (k > 1):
      if (abs(s-s0) <= eps*abs(s)):
        break
      if (abs(s) <= eps and abs(s) <= abs(s-s0)):
        break

    h *= 0.5
    n *= 2
    s0 = s
    t0 = t

  if (k >= kmax):
    print("Maximo de interacoes excedido!")

  return s
```

Fonte: Elaborado pelo autor.

Os argumentos, variáveis e constantes da função `integralSimpsonAdaptive`, im-

plementando no código 52 mantêm os mesmos significados que foram atribuídas na função `integralTrapzAdaptive`, no código 51.

4.4 Método de Romberg

Nesta seção, ilustraremos como o método de Richardson aplicada aos resultados da regra trapezoidal pode ser usada para obter aproximações de alta precisão com pouco custo computacional.

Na seção 4.3 encontramos a seguinte relação

$$S_k = \frac{4T_k - T_{k-1}}{3}$$

entre as aproximações T_k da regra trapezoidal (4.12) e as aproximações correspondentes S_k dadas pela fórmula de Simpson (4.15). A fim de generalizar esse resultado, renomeamos as aproximações trapezoidais como $R_{k,0}$:

$$\begin{aligned}
R_{0,0} &= h_0\left[\frac{f(a)}{2} + \frac{f(b)}{2}\right], \quad h_0 = b-a, \\
R_{1,0} &= h_1\left[\frac{f(a)}{2} + f(a+h_1) + \frac{f(b)}{2}\right], \quad h_1 = \frac{h_0}{2}, \\
R_{2,0} &= h_2\left[\frac{f(a)}{2} + f(a+h_2) + f(a+2h_2) + f(a+3h_2) + \frac{f(b)}{2}\right], \quad h_2 = \frac{h_1}{2}.
\end{aligned} \tag{4.18}$$

Reescrevendo a (4.13) em termos dessa mudança, ficamos

$$R_{k,0} = \frac{1}{2}\left[R_{k-1,0} + h_{k-1}\sum_{i=1}^{n_{k-1}} f\left(a + \left(i - \frac{1}{2}\right)h_{k-1}\right)\right], \quad k = 1, 2, \cdots \tag{4.19}$$

e as duas primeiras aproximações produzidas pela fórmula de Simpson são:

$$\begin{aligned}
R_{1,1} &= \frac{h_1}{3}[f(a) + 4f(a+h_1) + f(b)] = \frac{4R_{1,0} - R_{0,0}}{3}, \\
R_{2,1} &= \frac{h_2}{3}[f(a) + 4f(a+h_2) + 2f(a+h_2) + 4f(a+3h_2) + f(b)] \\
&= \frac{4R_{2,0} - R_{1,0}}{3}.
\end{aligned} \tag{4.20}$$

Então, a relação geral (4.16) entre as aproximações baseadas na fórmula de Simpson e a regra trapezoidal assume a forma:

$$R_{k,1} = \frac{4R_{k,0} - R_{k-1,0}}{3} = \frac{4R_{k,0} - R_{k-1,0}}{4-1}, \quad k = 1, 2, \cdots. \tag{4.21}$$

Semelhante ao que foi feito para obter a equação (4.10), tomamos $n=4$ (conhecida como Regra de Boole - 5 pontos), temos

$$A_0 = \frac{h}{(-1)^{4-0}0!(4-0)!}\int_0^4 \prod_{j\neq 0}^{4}(t-j)dt = \frac{h}{24}\int_0^4 (t-1)(t-2)(t-3)(t-4)dt = \frac{14h}{45},$$

$$A_1 = \frac{h}{(-1)^{4-1}1!(4-1)!}\int_0^4 \prod_{j\neq 1}^{4}(t-j)dt = \frac{-h}{6}\int_0^4 t(t-2)(t-3)(t-4)dt = \frac{64h}{45},$$

$$A_2 = \frac{h}{(-1)^{4-2}2!(4-2)!}\int_0^4 \prod_{j\neq 2}^{4}(t-j)dt = \frac{h}{4}\int_0^4 t(t-1)(t-3)(t-4)dt = \frac{8h}{15},$$

$$A_2 = \frac{h}{(-1)^{4-3}3!(4-3)!}\int_0^4 \prod_{j\neq 3}^{4}(t-j)dt = \frac{-h}{6}\int_0^4 t(t-1)(t-2)(t-4)dt = \frac{64h}{45},$$

$$A_2 = \frac{h}{(-1)^{4-4}4!(4-4)!}\int_0^4 \prod_{j\neq 4}^{4}(t-j)dt = \frac{h}{24}\int_0^4 t(t-1)(t-2)(t-3)dt = \frac{14h}{45},$$

resultando

$$\int_a^b f(x)dx \approx \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4). \tag{4.22}$$

Esta primeira aproximação com base na fórmula de Boole pode ser expressa da seguinte forma (semelhamte ao processo feito na equação (4.12))

$$R_{2,2} = \frac{2h_2}{45}[7f(a) + 32f(a+h_2) + 12f(a+2h_2) + 32f(a+3h_2) + 7f(b)] \tag{4.23}$$

e podemos relacionar com a equação (4.20)

$$R_{2,2} = \frac{16R_{2,1} - R_{1,1}}{15} = \frac{4^2R_{2,1} - R_{1,1}}{4^2 - 1}. \tag{4.24}$$

Pode-se provar [veja (PETERSDORFF, 1993)] que vale a relação de recorrência abaixo:

$$R_{k,j}\frac{4^jR_{k,j-1} - R_{k-1,j-1}}{4^j - 1}, \quad j = 1,2,\cdots, \quad k = 1,2,\cdots, \tag{4.25}$$

As estimativas computadas $R_{k,j}$ podem ser convenientemente organizadas em uma tabela

A coluna j contém as aproximações baseadas na fórmula de *Newton-Cotes* de ordem 2^j para números sucessivamente duplicados de subintervalos (tamanhos de subintervalos reduzidos à metade).

A avaliação da integral pode ser alcançada completando a tabela 12 recursivamente de cima para baixo, determinando para cada linha k as aproximações com base em todas as fórmulas

Tabela 12 – Tabela Romberg

n_k	Trapezoidal	Simpson	Boole	...
1	$R_{0,0}$			
2	$R_{1,0}$	$R_{1,1}$		
2^2	$R_{2,0}$	$R_{2,1}$	$R_{2,2}$	
$\vdots$	$\vdots$	$\vdots$		$\ddots$

Fonte: Adaptado de (BEU, 2014)

de *Newton-Cotes* aplicáveis na respectiva malha de $2k$ pontos de integração. Especificamente, os elementos da matriz são avaliados na ordem $R+0,0$, $(R_{1,0}, R_{1,1})$, $(R_{2,0}, R_{2,1}, R_{2,2})$, $\cdots$.

A essência do método de *Romberg* consiste, portanto, em preencher recursivamente a matriz de aproximações $R_{k,j}$ da integral e pode ser lançada sob a forma:

$$R_{0,0} = \frac{h_0}{2}[f(a)+f(b)], \quad h_0 = b-a, \quad n_0 = 1, \tag{4.26}$$

$$R_{k,0} = \frac{1}{2}\left[R_{k-1,0} + h_{k-1}\sum_{i=1}^{n_{k-1}} f(a+(i-1/2)k_{k-1})\right], \tag{4.27}$$

$$R_{k,j} = \frac{4^j R_{k,j-1} - R_{k-1,j-1}}{4^j - 1}, \quad j = 1,2,\cdots,k,\ h_k = \frac{h_{k-1}}{2}, n_k = 2n_{k-1}. \tag{4.28}$$

O processo iterativo deve ser continuado até que a diferença relativa entre duas aproximações diagonais consecutivas se torne menor do que uma tolerância predefinida ε:

$$|R_{k,k} - R_{k-1,k-1}| \leq \varepsilon |R_{k,k}|. \tag{4.29}$$

Código-fonte 53 – Integração de uma função usando o método de romberg

```
1  import math
2
3  def rombergMetodo(func, a, b, eps = 1e-6):
4      """
5      Integra a função Func no intervalo [a, b] com eps de precisão
6      relativa usando o método adaptativo de Romberg
7      """
8
9      kmax = 30
10     r1 = [0]*(kmax+1)
11     r2 = [0]*(kmax+1)
```

```
12
13      h = b - a
14    n = 1
15
16      # aproximação inicial
17      r1[0] = 0.5*h*(func(a) + func(b))
18      for k in range(1,kmax+1):
19          sumf = 0e0
20          for i in range(1,n+1):
21              sumf += func(a+(i-0.5)*h)
22
23          # fórmula trapezoidal
24          r2[0] = 0.5*(r1[0] + h*sumf)
25          f = 1e0
26          for j in range(1,k+1):
27              f *=4
28              r2[j] = (f*r2[j-1] - r1[j-1])/(f-1)
29
30          # verificação de convergência
31          if (k > 1):
32              if (abs(r2[k]-r1[k-1]) <= eps*abs(r2[k])):
33                  break
34
35              if (abs(r2[k]) <= eps and
36                  abs(r2[k]) <= abs(r2[k]-r1[k-1])):
37                  break
38
39          h *= 0.5
40          n *=2
41          for j in range(0,k+1):
42              r1[j] = r2[j]
43
44      if (k >= kmax):
45          print("rombergMetodo: número máximo de interações excedido!")
46          k-=1
47
48      return r2[k]
49
50  def func(x):
```

```
51      return x**3 * math.exp(-x)
52
53  a = 0e0
54  b = 1e0
55  eps = 1e-10
56
57  print("romberg = ", rombergMetodo(func, a, b, eps))
```

Fonte: Elaborado pelo autor.

O método de *Romberg*, nas equações (4.26) - (4.29) é implementado no código abaixo. Os parâmetros das funções, `integralTrapzAdaptive` e `integralSimpsonAdaptive`, são semelhantes as funções anteriores.

4.5 Implementação em Fortran

Os códigos seguintes mostram a implementações Fortran equivalentes das funções Python desenvolvidas durante o texto. As rotinas correspondentes têm nomes, parâmetros e funcionalidades idênticos.

Código-fonte 54 – Integração de uma função usando o método dos trapézios

```
1   function regraTrapz(a, b, n)
2
3       !Calcula a integral da funcao func no intervalo
4       ![a,b] usando o metodo dos trapezios com n pontos
5
6       implicit none
7
8       real ( kind = 8 ) :: a, b
9       integer :: n
10
11      real ( kind = 8 ) :: func
12      real ( kind = 8 ) :: regraTrapz
13
14      real ( kind = 8 ) :: h, soma!, fa, fb, fi
15      integer :: i
16
```

```
17    h = (b-a)/(n*1.0)
18    soma = (func(a)+func(b))/2
19
20    do i = 1,n-1
21        soma = soma + func(a+i*h)
22    end do
23    regraTrapz = h*soma
24
25 end function regraTrapz
```

Fonte: Elaborado pelo autor.

Código-fonte 55 – Integração de uma função usando o método dos trapézios Adaptivo

```
1  function integralTrapzAdaptive(a, b)
2
3    !Calcula a integral da funcao func no intervalo [a,b]
4    !com precisao eps usando o metodo trapezoidal adaptativo
5
6    implicit none
7
8    real ( kind = 8 ) :: a, b, eps
9    real (kind = 8 ) :: integralTrapzAdaptive
10   integer :: kmax = 30, n, k, i
11   real ( kind = 8 ) :: h, t0, sumf , t
12   real ( kind = 8 ) :: func
13
14   eps = 1.e-10
15
16   h = b-a
17   n = 1
18   t0 = 0.5*h*(func(a) + func(b))            !aproximacao inicial
19
20   do k = 1, kmax
21     sumf = 0.e0
22
23     do  i = 1, n
24       sumf = sumf + func(a+(i-0.5)*h)
25     end do
```

```
26
27      t = 0.5*(t0 + h*sumf)                     !nova aproximacao
28
29      if ( k > 1 ) then                         !checando convergencia
30        if ( abs(t-t0) <= eps*abs(t)) then
31          exit
32        end if
33        if ( abs(t) <= eps .and. abs(t) <= abs(t-t0)) then
34          exit
35        end if
36      end if
37
38      h = h*0.5
39      n = n*2
40      t0 = t
41    end do
42
43    if ( k >= kmax ) then
44      print*, "Maximo de interacoes excedido!"
45    end if
46
47    integralTrapzAdaptive = t
48
49  end function integralTrapzAdaptive
```

Fonte: Elaborado pelo autor.

Código-fonte 56 – Integração de uma função usando o método de Simpson

```
1  function regraSimpson(a, b, n)
2
3      !Calcula a integral da funcao func no intervalo
4      ![a,b] usando o metodo de Simpson com n (par) pontos
5
6      implicit none
7
8      real ( kind = 8 ) :: a, b
9      integer :: n
10
```

```
    real ( kind = 8 ) :: h, s1, s2, func
    real ( kind = 8 ) :: regraSimpson
    integer :: i

    if ( mod(n, 2) == 1 ) then
        n = n+1
    end if

    h = (b-a)/n
    s1 = 0.e0
    s2 = 0.e0

    n = n/2

    do i = 1, n-1              !soma em indice par
        s1 = s1 + func(a+2*i*h)
    end do

    do i = 1, n                !soma em indice impar
        s2 = s2 + func(a+(2*i-1)*h)
    end do

    regraSimpson = (h/3)*(func(a)+4*s2+2*s1+func(b))

end function regraSimpson
```

Fonte: Elaborado pelo autor.

Código-fonte 57 – Integração de uma função usando o método de Simpson Adaptivo

```
function integralSimpsonAdaptive(a, b)

    !Calcula a integral da funcao func no intervalo [a, b] com
    !precisao eps usando o metodo de Simpson adaptativo

    implicit none

    real ( kind = 8 ) :: a, b, eps
    real ( kind = 8 ) :: integralSimpsonAdaptive
```

```
10      integer :: kmax = 30
11      integer :: n, i, k
12      real ( kind = 8 ) :: h, s0, t0, sumf
13      real ( kind = 8 ) :: t, s
14      real ( kind = 8 ) :: func
15
16      eps = 1.e-6
17
18      h = b-a
19      n = 1
20      s0 = 0.5*h*(func(a) + func(b))
21      t0 = s0
22
23      do k = 1, kmax
24          sumf = 0.e0
25
26          do i = 1, n
27              sumf = sumf + func(a+(i-0.5)*h)
28          end do
29
30          t = 0.5*(t0 + h*sumf)
31          s = (4*t - t0)/3
32
33          if ( k > 1 ) then
34              if ( abs(s-s0) <= eps*abs(s) ) then
35                  exit
36              end if
37
38              if (abs(s) <= eps .and. abs(s) <= abs(s-s0)) then
39                  exit
40              end if
41          end if
42
43          h = h * 0.5
44          n = n * 2
45          s0 = s
46          t0 = t
47      end do
48
```

```
49      if (k >= kmax) then
50          print*, "Maximo de interacoes excedido!"
51      end if
52
53      integralSimpsonAdaptive = s
54
55  end function integralSimpsonAdaptive
```

Fonte: Elaborado pelo autor.

Código-fonte 58 – Integração de uma função usando o método de romberg

```
1   function rombergMetodo(a, b, eps)
2
3       ! Integra a função Func no intervalo [a, b] com eps
4       ! de precisão usando o método adaptativo de Romberg
5
6       implicit none
7
8       real ( kind = 8 ) :: a, b, eps
9       integer, parameter :: kmax = 30
10
11      real ( kind = 8 ), dimension(kmax+1) :: r1, r2
12      real ( kind = 8 ) :: h
13      integer :: n
14      integer :: i, j, k
15      real ( kind = 8 ) :: sumf, f, rombergMetodo
16      real ( kind = 8 ) :: func
17
18      h = b - a
19      n = 1
20
21      r1(1) = 0.5*h*(func(a)+func(b))
22      do k = 2, kmax+1
23        sumf = 0.e0
24        do i = 1, n
25          sumf = sumf + func(a+(i-0.5)*h)
26        end do
27
```

```
28        r2(1) = 0.5 * (r1(1)+h*sumf)
29        f = 1.e0
30        do j = 2, k+1
31          f = f * 4
32          r2(j) = (f*r2(j-1)-r1(j-1))/(f-1)
33        end do
34
35        ! verificação de convergência
36
37        if ( k > 1 ) then
38          if ( abs(r2(k)-r1(k-1)) <= eps*abs(r2(k))) then
39            exit
40          end if
41
42          if (abs(r2(k)) <= eps .and. &
43                  &abs(r2(k)) <= abs(r2(k)-r1(k-1))) then
44            exit
45          end if
46        end if
47
48        h = h * 0.5
49        n = n * 2
50        do j = 1, k+1
51          r1(j) = r2(j)
52        end do
53      end do
54
55      if ( k >= kmax) then
56        print*, "rombergMetodo: número máximo de interações excedido!"
57      end if
58
59      rombergMetodo = r2(k+1)
60
61  end function rombergMetodo
62
63  function func(x)
64
65      implicit none
66
```

```
67     real ( kind = 8 ) :: x, func
68     func = x*x*x * exp(-x)
69
70 end function
71
72 program main
73
74     implicit none
75
76     real ( kind = 8 ) :: a, b, eps
77     real ( kind = 8 ) :: romberg
78     real ( kind = 8 ) :: rombergMetodo
79
80     a = 0.e0
81     b = 1.e0
82     eps = 1.e-10
83
84     romberg = rombergMetodo(a, b, eps)
85     print *, romberg
86
87 end program
```

Fonte: Elaborado pelo autor.

5 EQUAÇÕES DIFERENCIAIS ORDINÁRIAS

As primeiras equações diferenciais são tão antigas quanto o cálculo diferencial. As equações diferenciais estão entre as ferramentas matemáticas mais importantes usadas na produção de modelos nas ciências físicas, biologia e engenharia.

A importância particular dos métodos numéricos para resolver equações diferenciais ordinárias (EDOs) ou seus sistemas é devido ao fato de que muitas das leis da natureza são convenientemente expressas na forma diferencial. As equações clássicas de movimento de partículas, as equações de difusão de massa ou transporte de calor ou a equação de onda de Schrödinger são apenas alguns exemplos que ilustram a extraordinária diversidade de fenômenos físicos essencialmente diferentes, que são modelados por meio de equações diferenciais.

A análise tradicional resolve os casos triviais e pode produzir uma visão inestimável para a solução dos mais complexos. Mas o ponto principal é que esses casos devem ser tratados numericamente.

Infelizmente, não existe um método que solucione todas as equações diferencias. Cada equação tem uma característica própria, e um método que funciona bem em uma que pode funcionar mal ou mesmo falhar em outra.

O que faremos é desenvolver algumas ideias gerais sobre a solução numérica de equações diferenciais e implementar essas ideias em um código que funcione bem para uma ampla variedade de problemas. Os problemas que iremos considerar serão de dois tipos: problemas com condição inicial e problemas com condições de fronteira.

5.1 Método de Euler

O método de Euler é a técnica de aproximação mais elementar para resolver problemas de valor inicial. Embora raramente seja usado na prática, a simplicidade de sua derivação pode ser usada para ilustrar as técnicas envolvidas na construção de algumas das técnicas mais avançadas, sem a álgebra complicada que acompanha essas construções. O objetivo do método de Euler é obter aproximações para o problema de valor inicial

$$\frac{dy}{dt} = f(t,y) \quad a \leqslant t \leqslant b, \quad y(a) = y_0. \tag{5.1}$$

Uma aproximação contínua para a solução $y(t)$ não será obtida; em vez disso, aproximações para y serão geradas em vários valores, chamados pontos de malha, no intervalo

$[a,b]$. Uma vez que a solução aproximada é obtida nos pontos, a solução aproximada em outros pontos do intervalo pode ser encontrada por interpolação. Primeiro fazemos a estipulação de que os pontos da malha são igualmente distribuídos ao longo do intervalo $[a,b]$. Esta condição é garantida escolhendo um número inteiro positivo N e selecionando os pontos da malha

$$t_i = a + ih, \quad \text{para } i = 0,\ 1,\ 2,\ \cdots,\ N. \tag{5.2}$$

A distância comum entre os pontos $h = (b-a)/N = t_{i+1} - t_i$ é chamada de tamanho do passo.

Assumimos que a solução existe, é única, finita e expansível em uma série de Taylor nas proximidades do ponto inicial a:

$$y(t) = y(a) + \frac{t-a}{1!}y'(a) + \frac{(t-a)^2}{2!}y''(a) + \cdots. \tag{5.3}$$

A expansão de Taylor (5.3) fornece um procedimento para avaliar a solução $y(t)$ em determinados pontos da malha $t_0, t_1, \cdots, t_m, \cdots$ como uma sequência de soluções do problema inicial. Especificamente, propagar a solução de t_m para t_{m+1} é equivalente a resolver a equação diferencial (5.1) sujeita à condição inicial

$$y(t_m) = y_m,$$

que é realizado com base na expansão

$$y_{m+1} = y_m + hy'_m + \frac{h^2}{2}y''_m + \cdots, \tag{5.4}$$

onde $y_m \equiv t(t_m)$, $y'_m \equiv y'(t_m)$, $y'' \equiv y''(t_m)$ e assim sucessivamente.

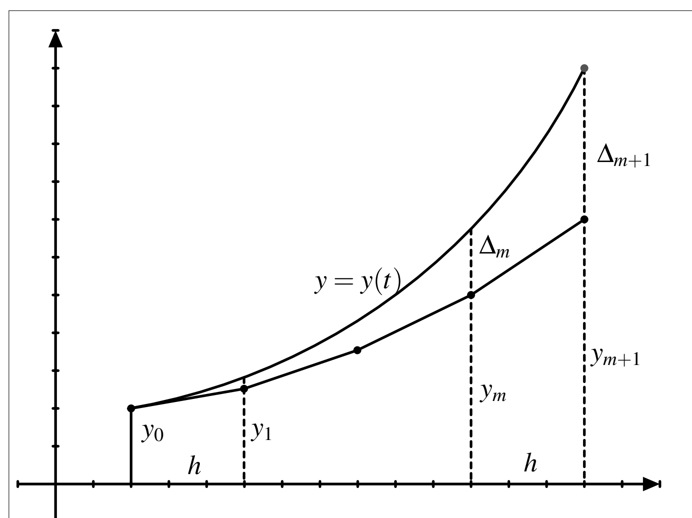
Tomando a aproximação linear da série de Taylor, obtém assim, o método de Euler

$$y_{m+1} = y_m + hf(t_m, y_m), \quad m = 0, 1, 2, \cdots. \tag{5.5}$$

Vemos na figura 8 mostrando que a aplicação repetida do método de Euler equivale a substituir a curva $y = y(t)$ pela função linear por partes conectando os pontos (t_0, y_0), (t_1, y_1), (t_2, y_2) , $\cdots$. O segmento inicial representa a reta tangente à solução exata que passa pelo ponto inicial (t_0, y_0). Em geral, a solução propagada y_{m+1} pode ser considerada como a extrapolação em linha reta da solução particular da equação passando pelo ponto anterior (t_m, y_m), no entanto, não necessariamente satisfazendo a condição inicial $y(t_0) = y_0$.

Uma maneira prática de superar a baixa precisão do método de Euler é aumentar a ordem da aproximação aplicando uma variação do método, que opera com duas estimativas de

Figura 8 – Soluções aproximadas.



Fonte: Elaborado pelo autor

solução em cada etapa de propagação. Primeiro, começando do valor y_m, calculamos um valor estimado inicial $\overline{y}_{m+1}$ através do método de Euler

$$\overline{y}_{m+1} = y_m + hf(t_m, y_m), \quad m = 0, 1, 2, \cdots . \tag{5.6}$$

Em seguida melhoramos a estimativa inicial usando a regra trapezoidal,

$$y_{m+1} = y_m + \frac{h}{2}[f(t_m, y_m) + f(t_{m+1}, \overline{y}_{m+1})]. \tag{5.7}$$

O método de Euler e suas variantes é implementado nas funções `Euler1` e `Euler2`, no código abaixo.

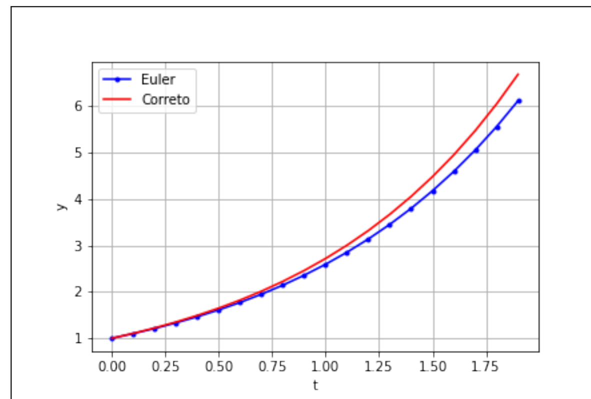
Código-fonte 59 – Método de Euler.

```
1  def euler(funcao, y0, a, b, n):
2
3      """
4      Aproxima a solucao de y'= f (y, t) pelo metodo de Euler.
5      ------------------------------------------------
6      funcao: Lado direito da equacao diferencial y'= f (t, y),
7              y(t0) = y0
8      y0: Valor inicial y(t0) = y0 onde t0 e a entrada no
9          indice 0 no array t
10     a, b: Intervalo onde sera feita a aproximacao
11     n: Quantidade de pontos entre a e b
12     ------------------------------------------------
```

```
13        y: Aproximacao y[n] da solucao y(tn) calculada pelo
14           metodo de Euler.
15        """
16
17        y = [0]*n
18        y[0] = y0
19
20        h = (b-a)/n
21        for i in range(n-1):
22            y[i+1] = y[i] + funcao(y[i],a+i*h)*h
23
24        return y
```

Fonte: Elaborado pelo autor.

Figura 9 – Solução da equação $y' = y$, $y(0) = 1$ usando método de Euler.



Fonte: Elaborado pelo autor

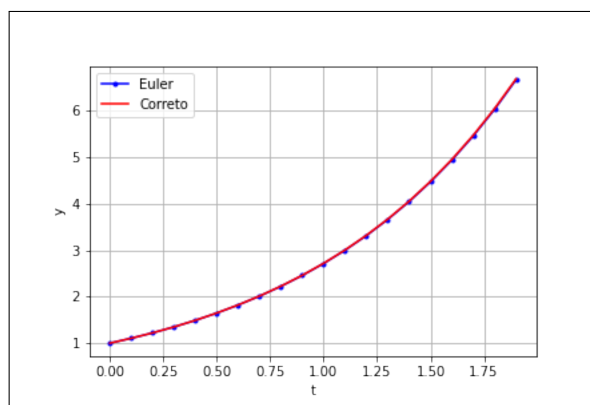
Código-fonte 60 – Método de Euler usando a regra trapezoidal.

```
1   def euler2(fun, y0, a, b, n):
2
3       """
4       Mesmos parametros da funcao euler anterior
5       """
6
7       y = [0]*n
8       y2 = [0]*n
9
```

```
10      y[0] = y0
11      y2[0] = y0
12
13      h = (b-a)/n
14
15      for i in range(n-1):
16          ti = a+i*h
17
18          y2[i+1] = y[i] + fun(y[i],ti)*h
19          y[i+1] = y[i] + (fun(y[i], ti) + fun(y2[i+1], ti+h))*h/2
20
21      return y
```

Fonte: Elaborado pelo autor.

Figura 10 – Solução da equação $y' = y$, $y(0) = 1$ usando método de Euler com regra trapezoidal.



Fonte: Elaborado pelo autor

5.2 Métodos de Runge-Kutta

Os métodos de Runge-Kutta são uma família de métodos que generalizam o método de Euler com a regra trapezoidal. Os métodos usam várias avaliações de f entre cada etapa de uma forma inteligente que leva a uma maior precisão.

Se a função f, definida na equação (5.1) é um número suficiente de vezes diferenciável, a solução exata $y(t)$ pode ser expandida em uma série de Taylor sobre o ponto $t = a$ de acordo com a equação (5.3). As sucessivas derivadas que ocorre na série, pode ser calculada

observando que

$$\begin{aligned} y' &= f(t,y), \\ y'' &= \frac{d}{dt}f(t,y) = f_t + f_y y' = f_t + f_y f, \\ y''' &= \frac{d}{dt}(f_t + f_y f) = f_{tt} + f_{ty} f + f_y(f_t + t_y f) + f(f_{yt} + f_{yy} f), \\ &= f_{tt} + 2f_{ty} f + f_t f_y + f_y^2 f + t_{yy} f^2. \end{aligned}$$

É claro que, em geral, a dificuldade de calcular as derivadas torna-se cada vez mais difícil com a ordem crescente da derivada. Para evitar a dificuldade de calcular as derivadas de ordem superior expressas em termos das derivadas parciais de f, substituiremos as derivadas pelo número necessário de valores funcionais de f em pontos intermediários.

Na série de Taylor, dado na equação (5.1), desconsiderando as derivadas de ordem maiores que 2, teremos

$$\begin{aligned} y_{m+1} &= y_m + hy'_m + \frac{h^2}{2} y''_m, \\ &= y_m + hf + \frac{h^2}{2}(f_t + f f_y). \end{aligned} \tag{5.8}$$

Pelo teorema do valor médio

$$y_{m+1} = y_m + hf(c, y_m(c)).$$

Aqui, usamos a média ponderada de $f(t_m, y_m)$ e $f(t_m + \alpha h, y_m + \beta h f(t_m, y_m))$ como substituto de $f(t_m, y_m)$, ou seja

$$y_{m+1} = y_m + hf(c, y_m(c)) = y_m + h(Af(t_m, y_m) + Bf(t_m + \alpha h, y_m + \beta h f(t_m, y_m))).$$

Escrevendo

$$\begin{aligned} k_1 &= hf(t_m, y_m), \\ k_2 &= hf(t_m + \alpha h, y_m + \beta k_1), \end{aligned}$$

ficamos

$$y_{m+1} = y_m + hf(c, y_m(c)) = y_m + Ak_1 + Bk_2.$$

Aplicando a fórmula de Taylor para funções de duas variáveis em k_2

$$k_2 = h(f + \alpha h f_t + \beta k_1 f_y + \cdots),$$

daí

$$y_{m+1} = y_m + Ahf + Bh(f + \alpha h f_t + \beta k_1 f_y + \cdots),$$
$$y_{m+1} = y_m + h(A+B)f + h^2 B(\alpha f_t + \beta f f_y) + \cdots,$$
$$y_{m+1} = y_m + h(A+B)f + \frac{h^2}{2}(2\alpha B f_t + 2\beta B f f_y) + \cdots. \tag{5.9}$$

Comparando a equação (5.9) com a equação (5.8), tiramos que

$$A + B = 1, \quad 2\alpha B = 1, \quad 2\beta B = 1.$$

Como temos quatro incógnitas e apenas três equações, não podemos determinar A, B, α, β unicamente. Mas, podemos tomar uma solução particular

$$A = B = \frac{1}{2}, \quad \alpha = \beta = 1$$

obtemos a solução

$$y_{m+1} = y_m + \frac{1}{2}(k_1 + k_2), \tag{5.10}$$
$$k_1 = hf(t_m, y_m),$$
$$k_2 = hf(t_m + h, y_m + k_1),$$

ou seja, o método de Euler com regra trapezoidal.

De maneira geral, representando y_{m+1} como combinação linear temos

$$y_{m+1} = y_m + \sum_{i=1}^{p} w_i k_i, \tag{5.11}$$

onde w_i são os pesos e k_i toma valores proporcionais a $f(t,y)$ com

$$k_i = hf(\varepsilon_i, \eta_i),$$

e

$$\varepsilon_i = t_m + \alpha_i h,$$
$$\eta_i = y_m + \sum_{j=i}^{i-1} \beta_{ij} k_j,$$

sendo α_i, β_{ij} e w_i parâmetros que devem ser determinados de forma a maximizar a concordância entre a combinação linear (5.11), e a série de Taylor (5.3). Colocando

$$\alpha_1 = 0, \quad \beta_{11} = 0, \quad \varepsilon_1 = t_m, \quad \eta_1 = y_m, \tag{5.12}$$

temos então

$$\begin{aligned} k_1 &= hf(t_m, y_m), \\ k_2 &= hf(t_m + \alpha_2 h, y_m + \beta_{21} k_1), \\ k_3 &= hf(t_m + \alpha_3 h, y_m + \beta_{31} k_1 + \beta_{32} k_2), \\ &\vdots \end{aligned} \tag{5.13}$$

vemos que várias quantidades são calculadas de uma maneira direta com base apenas nas quantidades já disponíveis.

Para $p = 1$, obtemos a fórmula do método de Euler. Para $p = 2$, o método de Euler com regra trapezoidal.

Para $p = 3$, temos a relação

$$y_{m+1} = y_m + h(w_1 k_1 + w_2 k_2 + w_3 k_3). \tag{5.14}$$

E combinando as equações (5.13) e (5.14), obtemos

$$\begin{aligned} w_1 + w_2 + w_3 &= 1, \\ \alpha_2 w_2 + \alpha_3 w_3 &= \frac{1}{2}, \\ w_3 \beta_{32} \alpha_2 &= \frac{1}{6}, \\ w_2 \alpha_2^2 + w_3 \alpha_3^2 &= \frac{1}{3}, \end{aligned} \tag{5.15}$$

um sistema de 4 equações e 6 incógnitas. Tomamos a seguintes solução

$$w_1 = \frac{1}{6}, \quad w_2 = \frac{4}{6}, \quad w_3 = \frac{1}{6}$$
$$\alpha_2 = \frac{1}{2}, \quad \alpha_3 = 1$$
$$\beta_{21} = \frac{1}{2}, \quad \beta_{31} = -1, \quad \beta_{32} = 2.$$

Com esses valores, as equações formam o método de Runge-Kutta de ordem 3

$$\begin{aligned} y_{m+1} &= y_m + \frac{1}{6}(k_1 + 4k_2 + k_3), \\ k_1 &= hf(t_m, y_m), \\ k_2 &= hf(t_m + \frac{h}{2}, y_m + \frac{k1}{2}), \\ k_3 &= hf(t_m + h, y_m - k_1 + 2k_2). \end{aligned} \tag{5.16}$$

A implementação desse método encontra-se no código 61 abaixo.

Código-fonte 61 – Método de Runge-Kutta de terceira ordem.

```
1  def rungeKutta3(funcao, y0, a, b, n):
2      """
3      Semelhante as funções euler anteriores
4      """
5
6      y = [0]*n
7      y[0] = y0
8      h = (b-a)/n
9      for i in range(n-1):
10         ti = a+i*h
11
12         k1 = h*funcao(ti, y[i])
13         k2 = h*funcao(ti+h/2, y[i]+k1/2)
14         k3 = h*funcao(ti+h, y[i]-k1+2*k2)
15
16         y[i+1] = y[i] + (k1+4*k2+k3)/6
17
18     return y
```

Fonte: Elaborado pelo autor.

A variante mais importante e amplamente usada é o método Runge-Kutta de quarta ordem ($p = 4$) Seguindo o mesmo raciocínio para a determinação dos coeficientes α_i, β_{ij} e w_i como no caso $p = 2$, resulta um sistema não linear de 10 equações com 13 incógnitas que não possuem uma solução única. Vários conjuntos de coeficientes estão em uso, sendo preferidos aqueles que apresentam simplicidade ou favorecem o uso de armazenamento eficiente. A variante comumente empregada do método Runge-Kutta de quarta ordem é definida pelas relações abaixo, e cuja implementação encontra-se no código 62.

$$
\begin{aligned}
y_{m+1} &= y_m + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\
k_1 &= hf(t_m, y_m), \\
k_2 &= hf(t_m + \frac{h}{2}, y_m + \frac{k1}{2}), \\
k_3 &= hf(t_m + h\frac{h}{2}, y_m \frac{k_2}{2}), \\
k_4 &= hf(t_m + h, y_m + k_3).
\end{aligned}
\tag{5.17}
$$

Código-fonte 62 – Método de Runge-Kutta de terceira ordem.

```
def rungeKutta4(funcao, y0, a, b, n):

    """
    Parâmetros semelhantes aos anteriores
    """

    y = [0]*n
    y[0] = y0

    h = (b-a)/n

    for i in range(n-1):
        ti = a+i*h

        k1 = h*funcao(ti, y[i])
        k2 = h*funcao(ti+h/2, y[i]+k1/2)
        k3 = h*funcao(ti+h/2, y[i]+k2/2)
        k4 = h*funcao(ti+h, y[i]+k3)

        y[i+1] = y[i] + (k1+2*k2+2*k3+k4)/6

    return y
```

Fonte: Elaborado pelo autor.

Como exemplo, consideremos o problema de valor inicial

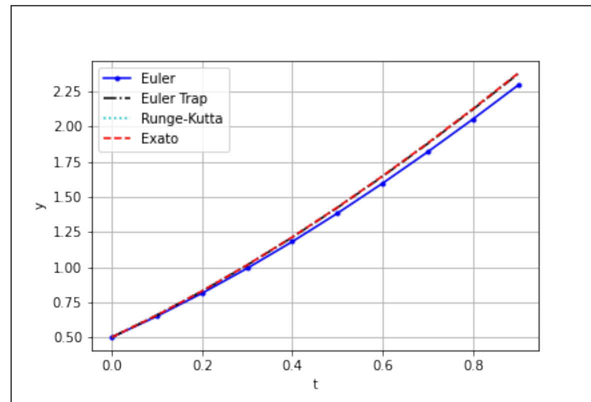
$$\begin{cases} y' = y - t^2 + 1, \quad 0 \leq t \leq 1, \\ y(0) = 0.5, \end{cases}$$

cuja solução é

$$y(t) = (1+t)^2 - \frac{1}{2}e^t \; .$$

Obteremos uma aproximação de $y(1)$, aplicando os métodos de Euler, Euler com regra Trapezoidal e Runge-Kutta, com passo $h = 0.1$ e compararemos os respectivos erros na figura 11.

Figura 11 – Comparação dos métodos de Euler, Euler com regra trapezoidal e Runge-Kutta na solução da equação $y' = y - t^2 + 1$, $y(0) = 0.5$.



Fonte: Elaborado pelo autor

Código-fonte 63 – Método de Runge-Kutta de terceira ordem.

```
import numpy as np
import matplotlib.pyplot as plt

#def euler(...)

#def eulerTrap(...):

#def rungeKutta4(...):

# valor inicial
y0 = 0.5
def fun(t, y):
  # y' = f(y, t) = y-t^2+1
  return y-t**2+1

def solucao(t):
  return 1+2*t+t**2-np.exp(t)/2

a = 0
b = 1

# h = (b-a)/n = 1/10 = 0.1
n = 10

```

```
25 yEuler = euler(fun, y0, a, b, n)
26 yEulerTrap = eulerTrap(fun, y0, a, b, n)
27 yRunge4 = rungeKutta4(fun, y0, a, b, n)
28
29 t = [a+i*(b-a)/n for i in range(n)]
30 yExato = [solucao(i) for i in t]
31
32 plt.plot(t, yEuler, 'b.-')
33 plt.plot(t, yEulerTrap, 'k-.')
34 plt.plot(t, yRunge4, 'c:')
35 plt.plot(t, yExato, 'r--')
36
37 plt.legend(['Euler','Euler Trap', 'Runge-Kutta', 'Exato'])
38
39 plt.grid(True)
40 plt.show()
41
42 print("{:>3} {:>6} {:>6} {:>6}".format("i","euler","exato","erro"))
43 for i in range(n):
44   erro = abs(yEuler[i]-yCorreto[i])
45   print("{} {:.5f} {:.5f} {:.5e}".format(t[i],yEuler[i],yCorreto[i],
       erro))
46
47 print("\n{:>3} {:>6} {:>6} {:>6}".format("i","euler trap","exato","
     erro"))
48 for i in range(n):
49   erro = abs(yEulerTrap[i]-yCorreto[i])
50   print("{} {:.5f} {:.5f} {:.5e}".format(t[i],yEulerTrap[i],yCorreto[
       i],erro))
51
52 print("\n{:>3} {:>6} {:>6} {:>6}".format("i","runge-kutta","exato","
     erro"))
53 for i in range(n):
54   erro = abs(yRunge4[i]-yCorreto[i])
55   print("{} {:.5f} {:.5f} {:.5e}".format(t[i],yRunge4[i],yCorreto[i],
       erro))
56
57 # saida
58 i       euler      exato       erro
```

```
0.0   0.50000   0.50000   0.00000e+00
0.1   0.57500   0.65741   8.24145e-02
0.2   0.65194   0.82930   1.77361e-01
0.3   0.72944   1.01507   2.85635e-01
0.4   0.80623   1.21409   4.07860e-01
0.5   0.88123   1.42564   5.44412e-01
0.6   0.95357   1.64894   6.95369e-01
0.7   1.02264   1.88312   8.60482e-01
0.8   1.08806   2.12723   1.03917e+00
0.9   1.14967   2.38020   1.23052e+00

i   euler trap   exato     erro
0.0   0.50000   0.50000   0.00000e+00
0.1   0.65700   0.65741   4.14541e-04
0.2   0.82844   0.82930   8.63621e-04
0.3   1.01372   1.01507   1.34992e-03
0.4   1.21221   1.21409   1.87631e-03
0.5   1.42319   1.42564   2.44583e-03
0.6   1.64588   1.64894   3.06174e-03
0.7   1.87940   1.88312   3.72751e-03
0.8   2.12278   2.12723   4.44680e-03
0.9   2.37497   2.38020   5.22352e-03

i   runge-kutta   exato     erro
0.0   0.50000   0.50000   0.00000e+00
0.1   0.65741   0.65741   1.65962e-07
0.2   0.82930   0.82930   3.44923e-07
0.3   1.01507   1.01507   5.37779e-07
0.4   1.21409   1.21409   7.45476e-07
0.5   1.42564   1.42564   9.69002e-07
0.6   1.64894   1.64894   1.20939e-06
0.7   1.88312   1.88312   1.46771e-06
0.8   2.12723   2.12723   1.74508e-06
0.9   2.38020   2.38020   2.04264e-06
```

Fonte: Elaborado pelo autor.

5.3 Sistema de equações diferenciais

Até agora, nos concentramos em como resolver uma única equação diferencial de primeira ordem. Na prática, duas ou mais dessas equações, acopladas, são necessárias para modelar um problema, e talvez até equações de ordem superior. Nesta seção, veremos como os métodos que desenvolvemos acima podem ser facilmente adaptados para lidar com sistemas de equações e equações de ordem superior.

Muitos problemas práticos envolvem não uma, mas duas ou mais equações diferenciais. Por exemplo, muitos processos evoluem no espaço tridimensional, com equações diferenciais separadas em cada dimensão espacial.

Considerando, por exemplo as três equações com condições iniciais

$$x' = xy + \cos z, \quad x(0) = x_0, \tag{5.18}$$

$$y' = 2 - t^2 + z^2 y, \quad y(0) = y_0, \tag{5.19}$$

$$z' = \sin t - x + y, \quad z(0) = z_0\ . \tag{5.20}$$

Se nós introduzirmos os vetores $\vec{\boldsymbol{x}} = (x, y, z)$, $\vec{\boldsymbol{x}}_0 = (x_0, y_0, z_0)$ e os vetores de funções $\vec{\boldsymbol{f}}(t, \vec{\boldsymbol{x}}) = (f_1(t, \vec{\boldsymbol{x}}, f_2(t, \vec{\boldsymbol{x}}), f_3(t, \vec{\boldsymbol{x}})))$ definido por

$$
\begin{aligned}
x' &= f_1(t, \vec{\boldsymbol{x}}) = f_1(t, x, y, z) = xy + \cos z, \\
y' &= f_2(t, \vec{\boldsymbol{x}}) = f_2(t, x, y, z) = 2 - t^2 + z^2 y, \\
z' &= f_3(t, \vec{\boldsymbol{x}}) = f_3(t, x, y, z) = \sin t - x + y,
\end{aligned}
$$

podemos escrever as equações (5.18-5.20) simplesmente como

$$\vec{\boldsymbol{x}} = \vec{\boldsymbol{f}}(t, \vec{\boldsymbol{x}}), \quad \vec{\boldsymbol{x}}(0) = \vec{\boldsymbol{x}}_0.$$

Assim, o método de Euler, o método de Euler com regra trapezoidal e os métodos de Runge-Kutta generalizam naturalmente para sistemas de equações diferenciais.

Com isso, as fórmulas anteriores se escrevem:

Método de Euler

$$\vec{\boldsymbol{x}}_{m+1} = \vec{\boldsymbol{x}}_m + hf(t_m, \vec{\boldsymbol{x}}_m). \tag{5.21}$$

Método de Euler com regra trapezoidal

$$
\begin{aligned}
\vec{\boldsymbol{x}}_{m+1} &= \vec{\boldsymbol{x}}_m + \frac{1}{2}(\vec{\boldsymbol{k}}_1 + \vec{\boldsymbol{k}}_2), \\
\vec{\boldsymbol{k}}_1 &= hf(t_m, \vec{\boldsymbol{x}}_m), \\
\vec{\boldsymbol{k}}_2 &= hf(t_m + h, \vec{\boldsymbol{x}}_m + \vec{\boldsymbol{k}}_1).
\end{aligned} \tag{5.22}
$$

Método de Runge-Kutta de terceira ordem

$$\begin{aligned}
\vec{x}_{m+1} &= \vec{x}_m + \frac{1}{6}(\vec{k}_1 + 4\vec{k}_2 + \vec{k}_3),\\
\vec{k}_1 &= hf(t_m, \vec{x}_m),\\
\vec{k}_2 &= hf(t_m + \frac{h}{2}, \vec{x}_m + \frac{k\vec{k}_1}{2}),\\
\vec{k}_3 &= hf(t_m + h, \vec{x}_m - \vec{k}_1 + 2\vec{k}_2).
\end{aligned} \tag{5.23}$$

Método de Runge-Kutta de quarta ordem

$$\begin{aligned}
\vec{x}_{m+1} &= \vec{x}_m + \frac{1}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4),\\
\vec{k}_1 &= hf(t_m, \vec{x}_m),\\
\vec{k}_2 &= hf(t_m + \frac{h}{2}, \vec{x}_m + \frac{\vec{k}_1}{2}),\\
\vec{k}_3 &= hf(t_m + h\frac{h}{2}, \vec{x}_m \frac{\vec{k}_2}{2}),\\
\vec{k}_4 &= hf(t_m + h, \vec{x}_m + \vec{k}_3).
\end{aligned} \tag{5.24}$$

Código-fonte 64 – Método de Runge-Kutta para um sistema de k equações.

```
1  def sumVec(vl, n):
2      """
3      vl: lista de vetores de tamanho n
4      retorna a soma dessa lista de vetores
5      """
6  
7      v = [0]*n
8      for l in vl:
9          for i in range(n):
10             v[i] += l[i]
11     return v
12 
13 def multVecEsc(escalar, vetor):
14     """
15     Multiplica um escalar por um vetor
16     """
17 
18     n = len(vetor)
19     resultado = [0]*n
```

```
20      for i in range(n):
21          resultado[i] = escalar*vetor[i]
22      return resultado
23
24  def rungeKuttaVec(funcao, y0, a, b, n, k):
25      """
26      k: sistema de k equações diferenciais
27      """
28
29      y = [[0]*k]*n
30      y[0] = y0
31      h = (b-a)/n
32
33      for i in range(n-1):
34          ti = a+i*h
35
36          vk1 = multVecEsc(h, funcao(ti, y[i]))
37          vk2 = multVecEsc(h, funcao(ti+h/2, sumVec([y[i], multVecEsc
                (1/2, vk1)], k)))
38          vk3 = multVecEsc(h, funcao(ti+h/2, sumVec([y[i], multVecEsc
                (1/2, vk2)], k)))
39          vk4 = multVecEsc(h, funcao(ti+h, sumVec([y[i], vk3], k)))
40
41          y[i+1] = sumVec([y[i], multVecEsc(1/6, sumVec([vk1,
                multVecEsc(2, vk2), multVecEsc(2, vk3), vk4], k))], k)
42
43    return y
```

Fonte: Elaborado pelo autor.

Usando a blilioteca numpy, a implementação é quase direta, como podemos ver no código 65 abaixo.

Código-fonte 65 – Método de Runge-Kutta para um sistema de k equações usando a biblioteca numpy.

```
import numpy as np

def rungeKuttaVec(funcao, y0, a, b, n, k):

  y = np.zeros((n, k))

  y[0] = y0

  h = (b-a)/n

  for i in range(n-1):
    ti = a+i*h

    k1 = h*funcao(ti, y[i])
    k2 = h*funcao(ti+h/2, y[i]+k1/2)
    k3 = h*funcao(ti+h/2, y[i]+k2/2)
    k4 = h*funcao(ti+h, y[i]+k3)

    y[i+1] = y[i] + (k1+2*k2+2*k3+k4)/6

  return y
```

Fonte: Elaborado pelo autor.

5.4 Implementação em Fortran

Os códigos seguintes mostram a implementações FORTRAN equivalentes das funções Python desenvolvidas durante esse capítulo. As rotinas correspondentes têm nomes, parâmetros e funcionalidades idênticos.

Código-fonte 66 – Método de Euler.

```
subroutine euler(funcao, y0, a, b, n, y)

    !Aproxima a solucao de y'= f (t, y) pelo metodo de Euler.
```

```
4     !
5     ! Parametros
6     ! ------------------------------------------------
7     ! funcao: funcao
8     !    Lado direito da equacao diferencial y'= f (t, y),
9     !    y(t0) = y0
10    ! y0: numero
11    !    Valor inicial y(t0) = y0 onde t0 e a entrada no
12    !    indice 0 no array t
13    ! a, b: numeros
14    !    Intervalo onde sera feita a aproximacao
15    ! n: numero inteiro
16    !    Quantidade de pontos entre a e b
17    !
18    ! Retorno
19    ! ------------------------------------------------
20    ! y: array
21    !     Aproximacao y[n] da solucao y(tn) calculada pelo
22    !     metodo de Euler.
23
24    implicit none
25
26    real ( kind = 8 ) :: funcao
27    real ( kind = 8 ) :: y0, a, b, h, ti
28    integer :: n, i
29    real ( kind = 8 ), dimension(n) :: y
30
31    y(1) = y0
32    h = (b-a)/n
33
34    do i = 1, n-1
35      ti = a+(i-1)*h
36      y(i+1) = y(i) + funcao(ti, y(i))*h
37    end do
38
39 end subroutine euler
```

Fonte: Elaborado pelo autor.

Código-fonte 67 – Método de Euler usando a regra trapezoidal.

```
subroutine euler2(funcao, y0, a, b, n, y)

    ! Aproxima a solucao de y'= f (t, y) pelo metodo de Euler
    ! com trapezoidal.

    implicit none

    real ( kind = 8 ) :: funcao
    real ( kind = 8 ) :: y0, a, b, h, ti
    integer :: n, i
    real ( kind = 8 ), dimension(n) :: y, y2

    y(1) = y0
    y2(1) = y0

    h = (b-a)/n

    do i = 1, n-1
      ti = a+(i-1)*h
      y2(i+1) = y(i) + funcao(ti, y(i))*h
      y(i+1) = y(i) + (funcao(ti, y(i))+funcao(ti+h, y2(i+1)))*h/2
    end do

end subroutine euler2
```

Fonte: Elaborado pelo autor.

Código-fonte 68 – Método de Runge-Kutta de terceira ordem.

```
subroutine rungeKutta3(funcao, y0, a, b, n, y)

    ! Aproxima a solução de y '= f (y, t) pelo método de Runge-Kutta
    ! de terceira ordem.

    implicit none

    real ( kind = 8 ) :: funcao
```

```
    real ( kind = 8 ) :: y0, a, b, h, ti
    integer :: n, i
    real ( kind = 8 ), dimension(n) :: y
    real ( kind = 8 ) :: k1, k2, k3

    y(1) = y0

    h = (b-a)/n

    do i = 1, n-1
        ti = a+(i-1)*h

        k1 = h*funcao(ti, y(i))
        k2 = h*funcao(ti+h/2, y(i)+k1/2)
        k3 = h*funcao(ti+h, y(i)-k1+2*k2)

        y(i+1) = y(i) + (k1+4*k2+k3)/6
    end do

end subroutine rungeKutta3
```

Fonte: Elaborado pelo autor.

Código-fonte 69 – Método de Runge-Kutta de quarta ordem.

```
subroutine rungeKutta4(funcao, y0, a, b, n, y)

    ! Aproxima a solução de y '= f (y, t) pelo método de Runge-Kutta
    ! de quarta ordem.

    implicit none

    real ( kind = 8 ) :: funcao
    real ( kind = 8 ) :: y0, a, b, h, ti
    integer :: n, i
    real ( kind = 8 ), dimension(n) :: y
    real ( kind = 8 ) :: k1, k2, k3, k4

    y(1) = y0
```

```
15
16      h = (b-a)/n
17
18      do i = 1, n-1
19          ti = a+(i-1)*h
20
21          k1 = h*funcao(ti, y(i))
22          k2 = h*funcao(ti+h/2, y(i)+k1/2)
23          k3 = h*funcao(ti+h/2, y(i)+k2/2)
24          k4 = h*funcao(ti+h, y(i)+k3)
25
26          y(i+1) = y(i) + (k1+2*k2+2*k3+k4)/6
27      end do
28
29  end subroutine rungeKutta4
```

Fonte: Elaborado pelo autor.

Código-fonte 70 – Comparação dos métodos de Euler, Euler com trapezoidal e Runge-Kutta na solução da equação $y' = y - t^2 + 1$, $y(0) = 0.5$.

```
1   ! subroutine euler(...)
2   ! subroutine eulerTrap(...)
3   ! subroutine rungeKutta4(...)
4
5   function fun(t, y)
6
7     implicit none
8
9     real ( kind = 8 ) :: t, y, fun
10
11    fun = y - t*t+1
12
13  end function fun
14
15  function solucao(t)
16
17    implicit none
18
```

```
19    real ( kind = 8 ) :: t, exp, solucao
20
21    solucao = 1+2*t+t*t-exp(t)/2
22
23  end function solucao
24
25  program main
26
27    implicit none
28
29    real ( kind = 8 ) :: a, b, y0, fun, solucao
30    integer, parameter:: n = 10
31    real ( kind = 8 ), dimension(n) :: yEuler, yEulerTrap, yRunge4,
         yCorreto
32    integer :: i
33    real ( kind = 8 ) :: h, ti, er
34
35    external fun, solucao
36
37    a = 0.e0
38    b = 1.e0
39    y0 = 0.5e0
40    h = (b-a)/n
41
42    call euler(fun, y0, a, b, n, yEuler)
43    call eulerTrap(fun, y0, a, b, n, yEulerTrap)
44    call rungeKutta4(fun, y0, a, b, n, yRunge4)
45
46    do i = 1, n
47      ti = a+(i-1)*h
48      yCorreto(i) = solucao(ti)
49    end do
50
51    print 100
52    100 format (3x, "i", 5x, "euler", 5x, "exato", 5x, "erro")
53    do i = 1, n
54      er = abs(yEuler(i)-yCorreto(i))
55      print 200, "!", i, yEuler(i), yCorreto(i), er
56      200 format(1x, a1, 2x, i3, 5x, f7.5, 5x, f7.5, 5x, e10.5)
```

```
57      end do
58
59      print 110
60      110 format (3x, "i", 5x, "euler trap", 5x, "exato", 5x, "erro")
61      do i = 1, n
62        er = abs(yEulerTrap(i)-yCorreto(i))
63        print 210, "!", i, yEulerTrap(i), yCorreto(i), er
64        210 format(1x, a1, 2x, i3, 5x, f7.5, 5x, f7.5, 5x, e10.5)
65      end do
66
67      print 120
68      120 format (3x, "i", 5x, "runge-kutta", 5x, "exato", 5x, "erro")
69      do i = 1, n
70        er = abs(yRunge4(i)-yCorreto(i))
71        print 220, "!", i, yRunge4(i), yCorreto(i), er
72        220 format(1x, a1, 2x, i3, 5x, f7.5, 5x, f7.5, 5x, e10.5)
73      end do
74
75    end program main
76
77    ! saida
78    ! i      euler      exato      erro
79    !    1      0.50000      0.50000      .00000E+00
80    !    2      0.65000      0.65741      .74145E-02
81    !    3      0.81400      0.82930      .15299E-01
82    !    4      0.99140      1.01507      .23671E-01
83    !    5      1.18154      1.21409      .32548E-01
84    !    6      1.38369      1.42564      .41945E-01
85    !    7      1.59706      1.64894      .51877E-01
86    !    8      1.82077      1.88312      .62354E-01
87    !    9      2.05385      2.12723      .73383E-01
88    !   10      2.29523      2.38020      .84967E-01
89    ! i      euler trap      exato      erro
90    !    1      0.50000      0.50000      .00000E+00
91    !    2      0.65700      0.65741      .41454E-03
92    !    3      0.82844      0.82930      .86362E-03
93    !    4      1.01372      1.01507      .13499E-02
94    !    5      1.21221      1.21409      .18763E-02
95    !    6      1.42319      1.42564      .24458E-02
```

```
96  !     7      1.64588      1.64894      .30617E-02
97  !     8      1.87940      1.88312      .37275E-02
98  !     9      2.12278      2.12723      .44468E-02
99  !    10      2.37497      2.38020      .52235E-02
100 ! i       runge-kutta       exato      erro
101 !     1      0.50000      0.50000      .00000E+00
102 !     2      0.65741      0.65741      .16596E-06
103 !     3      0.82930      0.82930      .34492E-06
104 !     4      1.01507      1.01507      .53778E-06
105 !     5      1.21409      1.21409      .74548E-06
106 !     6      1.42564      1.42564      .96900E-06
107 !     7      1.64894      1.64894      .12094E-05
108 !     8      1.88312      1.88312      .14677E-05
109 !     9      2.12723      2.12723      .17451E-05
110 !    10      2.38020      2.38020      .20426E-05
```

Fonte: Elaborado pelo autor.

Código-fonte 71 – Método de Runge-Kutta para um sistema de k equações.

```
1  subroutine rungeKuttaMetodo(y0, a, b, n, k, s)
2      ! y0 : valor inicial
3      ! y' = f(t, y), a <= t <= b
4      ! n : pontos entre a e b
5      ! k : equações
6      ! s : solução
7
8      implicit none
9
10     integer :: n, k
11     real ( kind = 8 ), dimension(n, k) :: s
12     real ( kind = 8 ), dimension(k) :: y0
13     real ( kind = 8 ) :: a, b, h
14     real ( kind = 8 ) :: ti
15     real ( kind = 8 ), dimension(k) :: k1, k2, k3, k4
16     integer :: i
17
18     h = (b-a)/n
19     s(1,:) = y0(:)
```

```fortran

    do i = 1, n-1
      ti = a+(i-1)*h

      call func(ti, s(i,:), k, k1)
      call func(ti+h/2.e0, s(i,:)+k1*h/2.e0, k, k2)
      call func(ti+h/2.e0, s(i,:)+k2*h/2.e0, k, k3)
      call func(ti+h, s(i,:)+k3*h, k, k4)

      s(i+1,:) = s(i,:) + (k1+2.e0*k2+2.e0*k3+k4)*h/6.e0
    end do

end subroutine rungeKuttaMetodo

subroutine func(t, x, k, outv)
    ! k equações
    ! y' = f(t, y)
    implicit none

    integer :: k
    real ( kind = 8 ), dimension(k) :: x, outv
    real ( kind = 8 ) :: t

    outv(1) = x(2)
    outv(2) = -x(1)

end subroutine func

program main
    implicit none

    integer, parameter :: n = 100, k = 2

    real ( kind = 8 ) :: a, b
    real ( kind = 8 ), dimension(n, k) :: solucao
    real ( kind = 8 ), dimension(k) :: y0
    integer :: i, j

    a = 0.e0
```

```
59    b = 10.e0
60    y0 = (/1.e0, 1.e0/)
61
62    call rungeKuttaMetodo(y0, a, b, n, k, solucao)
63
64    do i = 1, n
65      do j = 1, k
66        write(*,"(i3,i3)", advance="no") i, j
67        write(*,"(es15.7)", advance="no") solucao(i, j)
68      end do
69      print*,""
70    end do
71
72 end program main
```

Fonte: Elaborado pelo autor.

6 PROPOSTA DE ATIVIDADE NO ENSINO MÉDIO

A ideia de uma simulação é programar as leis da física no computador e, em seguida, deixar o computador calcular o que acontece em função do tempo, passo a passo, no futuro. Essas leis geralmente serão as leis do movimento de Newton, e nosso objetivo será prever o movimento de um ou mais objetos sujeitos a várias forças. As simulações nos permitem fazer isso para quaisquer forças e quaisquer condições iniciais, mesmo quando não existe uma fórmula explícita para o movimento.

6.1 Movimento de projétil

Vamos simular o movimento de um projétil, primeiro em uma dimensão e depois em duas dimensões. Quando o movimento é puramente vertical, o estado do projétil é definido por sua posição, y, e sua velocidade, v_y. Essas quantidades são relacionadas por

$$v_y = \frac{dy}{dt} \approx \frac{\Delta y}{\Delta t} = \frac{y_{\text{final}} - y_{\text{inicial}}}{\Delta t}. \tag{6.1}$$

Em uma simulação de computador, já sabemos o valor atual de y e queremos prever o valor futuro. Então, vamos resolver esta equação para y_{final}

$$y_{\text{final}} \approx y_{\text{inicial}} + v_y \Delta t. \tag{6.2}$$

Da mesma forma, podemos prever o valor futuro de v_y se conhecermos o valor atual, bem como a aceleração:

$$a_y = \frac{v_y}{dt} \approx \frac{\Delta v_y}{\Delta t} = \frac{v_{y_{\text{final}}} - v_{y_{\text{inicial}}}}{\Delta t} \;, \tag{6.3}$$

$$v_{y_{\text{final}}} \approx v_{y_{\text{inicial}}} + a_y \Delta t. \tag{6.4}$$

Essas equações são válidas para qualquer objeto em movimento. Para um projétil se movendo perto da superfície da terra sem resistência do ar, $a_y = -g$ (tomando a direção $+y$ para cima). Em geral, a_y é dado pela segunda lei de Newton

$$a_y = \frac{\sum F_y}{m}, \tag{6.5}$$

onde m é a massa do objeto e as várias forças podem depender de y, v_y ou ambos.

Em uma simulação de computador de movimento unidimensional, a ideia é começar com o estado da partícula em $t = 0$, então usar as equações (6.2) a (6.5) para calcular y e v_y em

$t = \Delta t$, depois repetir o cálculo para o próximo intervalo de tempo e o próximo, e assim por diante. Felizmente, os computadores não se importam em fazer cálculos repetitivos.

Como regra geral, os computadores não podem lidar com o movimento contínuo. Portanto, para fazer qualquer simulação de movimento físico real parecer realista, teremos que "dividir" o movimento contínuo verdadeiro em muitos quadros separados por intervalos de tempo muito pequenos. A situação não é diferente de assistir a um filme em que percebemos um movimento contínuo quando, na realidade, estamos sendo expostos a uma sequência rápida de imagens estáticas.

Com isso, podemos usar o seguinte código para simular o movimento do projétil em uma dimensão sem resistência do ar:

Código-fonte 72 – Simular movimento de projétil em uma dimensão.

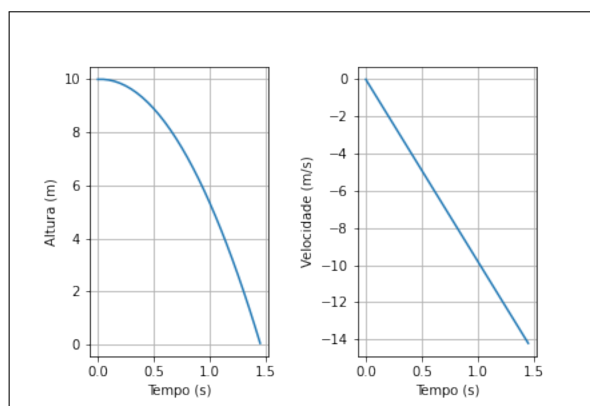
```
1  import matplotlib.pyplot as plt
2  
3  g = 9.8 # m/s2. ACeleração da gravidade.
4  y = 10 # m. Posição inicial.
5  dt = 0.05
6  vy = 0 # m/s. Partindo do repouso
7  t = 0 #s. Tempo inicial
8  
9  ay = -g
10 tempo = []          # guardar as posições e velocidades
11 posicao=[]                # e tempo para plotagem dos
12 velocidade=[]             # graficos
13 while y > 0:
14     tempo.append(t)
15     posicao.append(y)
16     velocidade.append(vy)
17     y += vy * dt          # usando vy antigo para calcular o novo y
18     vy += ay * dt         # usando ay para calular novo vy
19     t += dt
20 
21 print("Posição")
22 for p in posicao:
23     print(p)
24 
25 print("Velocidade")
```

```
26 for vi in velocidade:
27     print(vi)
28
29 plt.subplot(1,2,1)
30 plt.plot(tempo, posicao)
31 plt.xlabel("Tempo (s)")
32 plt.ylabel("Altura (m)")
33 plt.grid(True)
34
35 plt.subplot(1,2,2)
36 plt.plot(tempo, velocidade)
37 plt.xlabel("Tempo (s)")
38 plt.ylabel("Velocidade (m/s)")
39 plt.grid(True)
40
41 plt.show()
```

Fonte: Elaborado pelo autor.

A figura (12) mostra o gráfico do movimento. O primeiro tempo x posição e o segundo, tempo x velocidade.

Figura 12 – Movimento em uma dimensão.



Fonte: Elaborado pelo autor

Não há muito sentido em escrever uma simulação de computador quando você pode calcular a resposta exata com tanta facilidade. Então, vamos tornar o problema mais difícil adicionando alguma resistência ao ar. Em velocidades normais, a força da resistência do ar é aproximadamente proporcional a velocidade do projétil. Isso ocorre porque um projétil mais

rápido não apenas colide com mais moléculas de ar por unidade de tempo, mas também confere mais impulso a cada molécula que atinge. Então, podemos escrever a magnitude de a força de resistência como

$$F = cv, \tag{6.6}$$

para alguma constante c que depende do tamanho e forma do objeto e densidade do ar. A direção da força dessa força é sempre oposto a direção de v. Nesse caso, a Lei de Newton, na equação (6.5), pode ser expresso como:

$$\frac{\Delta v_{\mathrm{y}}}{\Delta t} = -g + \frac{c}{m} v_{\mathrm{y}}. \tag{6.7}$$

Código-fonte 73 – Movimento de projétil em uma dimensão com resistência.

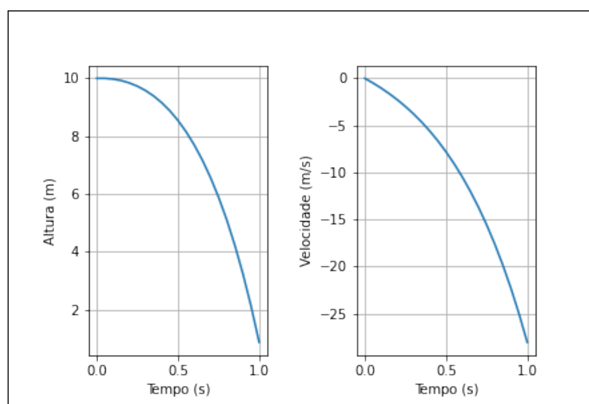
```
# (...) Dados iguais ao anterior

c = 2
m = 1
res = c/m

# (...)
while y > 0:
    # (...)
    vy += (ay + res * vy) * dt        # res (resistencia)
    t += dt
```

Fonte: Elaborado pelo autor.

Simular o movimento do projétil é apenas um pouco mais difícil em duas dimensões do que em uma. Para fazer isso, precisaremos de uma variável x para cada variável y, e cerca de duas vezes mais linhas de código para inicializar essas variáveis e atualizá-las dentro do *loop* de simulação.

Considere agora um objeto esférico lançado com uma velocidade v formando um ângulo θ com o solo horizontal. Na ausência de resistência do ar, a trajetória seguida por esse projétil é conhecida como uma parábola. Isso decorre de escrever a lei de Newton separadamente

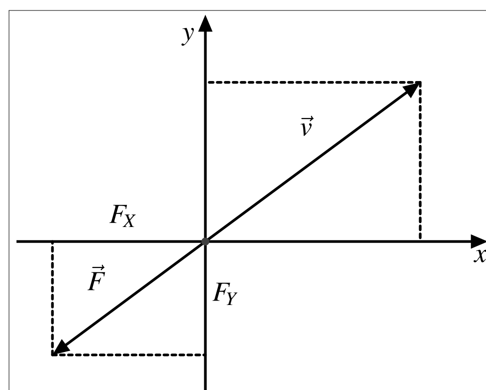
Figura 13 – Movimento de projétil em uma dimensão com resistência.



Fonte: Elaborado pelo autor.

para as coordenadas horizontal e vertical. O primeiro, linearmente com o tempo, enquanto o último varia quadraticamente. Portanto, quando o tempo é eliminado, ficamos com uma equação quadrática que dá origem a uma trajetória parabólica. Vamos ver como a trajetória muda quando a resistência do ar não é mais negligenciada. No caso de uma força resistiva que cresce linearmente com a velocidade, podemos ainda separar o movimento entre as coordenadas horizontal e vertical.

Figura 14 – Diagrama de corpo livre para um projétil.



Fonte: Elaborado pelo autor

Pelo diagrama de corpo livre, na imagem (14) a lei de Newton para as coordenadas horizontais e verticais torna-se

$$\frac{\Delta v_x}{\Delta t} = -\frac{c}{m} v_x \ ,$$
$$\frac{\Delta v_y}{\Delta t} = -g + \frac{c}{m} v_y.$$

Imaginemos agora que estamos em uma situação em que a resistência do ar depende

quadraticamente da velocidade. Agora as equações que descrevem a evolução temporal das componentes da velocidade não estão mais desacopladas. Em outras palavras, a equação para a componente horizontal v_x depende da componente vertical v_y e vice versa.

A direção de $\vec{F}$ é oposta a direção de $\vec{v}$, então podemos escrever a equação (6.6) em forma vetorial $\vec{F} = -cv\vec{v}$ e as componentes de $\vec{F}$ são

$$F_x = -cvv_x \quad \text{e} \quad F_y = -cvv_y,$$

onde $v = \sqrt{v_x^2 + v_y^2}$.

Os componentes de aceleração a_x e a_y estão mudando constantemente conforme os componentes de velocidade mudam. Mas, após um intervalo de tempo Δt suficientemente curto, podemos considerar a aceleração como essencialmente constante. Se conhecermos as coordenadas e as componentes da velocidade em algum horário, podemos encontrar essas quantidades no tempo $t + \Delta t$ usando as fórmulas para aceleração constante. Durante um intervalo de tempo Δt, a aceleração média da componente x é $a_x = \dfrac{\Delta v_x}{\Delta t}$ e a velocidade v_x muda $\Delta v_x = a_x \Delta t$. Similarmente, v_y muda por uma quantidade $\Delta v_y = a_y \Delta t$. Logo as componentes horizontal e vertical da velocidade no fim do intervalo é

$$v_x + \Delta v_x = v_x + a_x \Delta t \quad \text{e} \quad v_y + \Delta v_y = v_y + a_y \Delta t. \tag{6.8}$$

Enquanto isto está acontecendo, o projétil está se movendo, ou seja, as coordenadas estão mudando. A velocidade média na direção x durante o intervalo de tempo Δt é a média de v_x (no início do intervalo) e $v_x + \Delta v_x$ (no fim do intervalo), ou $v_x + \dfrac{\Delta v_x}{2}$. A coordenada x muda de

$$\Delta x = \left(v_x + \frac{\Delta v_x}{2} \right) \Delta t = v_x \Delta t + \frac{1}{2} a_x (\Delta t)^2,$$

e de igual modo para y.

Se tivermos um projétil lançado em um ângulo θ em relação ao solo, a componente horizontal da velocidade é

$$v_x = v \cos\theta,$$

e a componente vertical da velocidade

$$v_y = v \mathrm{sen}\theta.$$

A figura (15) mostra a trajetória de um projétil com e sem resistência do ar. Escolhemos o valor $c = 0.0012$ para a constante. Velocidade inicial de $50 m/s$ e um ângulo de $35°$ em

relação ao eixo x. Podemos ver que tanto o alcance da bola de beisebol quanto a altura máxima alcançada são substancialmente menores do que o cálculo sem resistência. Calcular a trajetória de um projétil e ignorar a resistência do ar é bastante irreal. A resistência do ar é uma parte importante de lançamento.

Código-fonte 74 – Trajetória de um projétil com e sem resistência.

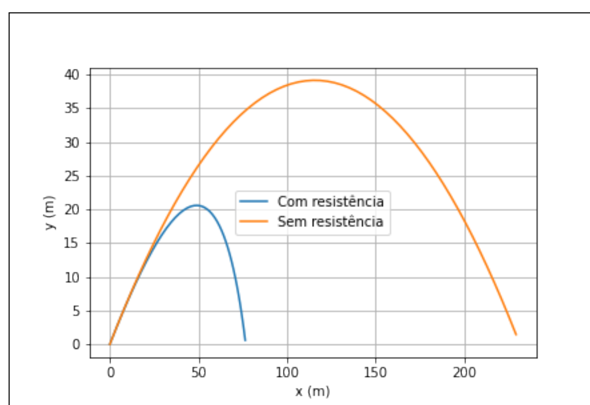
```
1  import math
2  import matplotlib.pyplot as plt
3
4  def trajetoriaProjetil(x, y, v, theta, t, m, c,
5      tempo, posicao, velocidade):
6
7      """
8      x,y          : posição inicial
9      v            : velocidade inicial
10     theta        : ângulo inicial
11     t            : tempo inicial
12     m            : massa do projétil
13     c            : constante de resistência
14     tempo        : lista  para guardar os tempo para plotagem
15     posicao      : posição sucessivas do projétil
16                  : posição[0] -> componente horizontal
17                  : posição[1] -> componente vertical
18     velocidade   : lista com velocidades sucessivas do projétil
19                  : velocidade[0] -> componente horizontal
20                  : velocidade[1] -> componente vertical
21     """
22
23     vx = v*math.cos(theta*math.pi/180)
24     vy = v*math.sin(theta*math.pi/180)
25     res = c/m
26
27     posicaox = posicao[0]
28     posicaoy = posicao[1]
29     velocidadex = velocidade[0]
30     velocidadey = velocidade[1]
31
32     dt = 0.1
```

```
33      while y >= 0:
34          ax = -res*v*vx
35          ay = -g - res*v*vy
36
37          tempo.append(t)
38          posicaox.append(x)
39          posicaoy.append(y)
40          velocidadex.append(vx)
41          velocidadey.append(vy)
42
43          vx += ax*dt
44          vy += ay*dt
45          x += (vx + ax*dt/2)*dt
46          y += (vy + ay*dt/2)*dt
47          t += dt
48
49  g = 9.8                 # m/s2. ACeleração da gravidade.
50  x = 0
51  y = 0                   # m. Posição inicial.
52  v = 50                  # m/s. velocidade
53  theta = 35              # 35*pi/180 radianos
54  t = 0                   # s. Tempo inicial
55  m = 0.145               # kg
56
57  tempo = []
58  posicaoComRes = [[], []]
59  velocidadeComRes = [[], []]
60
61  posicaoSemRes = [[], []]
62  velocidadeSemRes = [[], []]
63
64  # calculo da trajetoria sem resistencia
65  trajetoriaProjetil(x, y, v, theta, t, m, 0, tempo,
66    posicaoSemRes, velocidadeSemRes)
67
68  # calculo da trajetoria com resistencia
69  trajetoriaProjetil(x, y, v, theta, t, m, 0.0012, tempo,
70    posicaoComRes, velocidadeComRes)
71
```

```
72 plt.plot(posicaoComRes[0], posicaoComRes[1])
73 plt.plot(posicaoSemRes[0], posicaoSemRes[1])
74 plt.xlabel("x (m)")
75 plt.ylabel("y (m)")
76 plt.legend(["Com resistência", "Sem resistência"])
77 plt.grid(True)
78
79 plt.show()
```

Fonte: Elaborado pelo autor.

Figura 15 – Trajetória de um projétil com e sem resistência.



Fonte: Elaborado pelo autor

6.2 Pêndulo

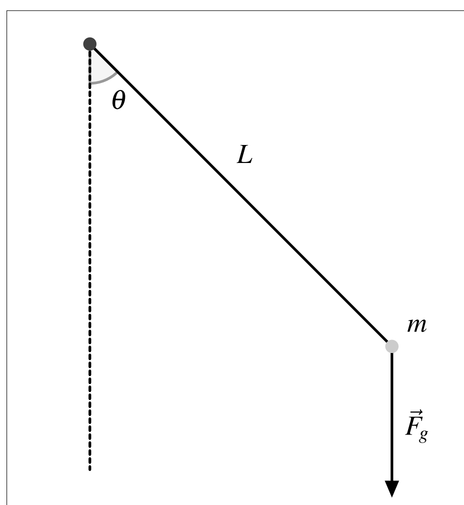
Agora exploraremos o comportamento de um pêndulo. Não há melhor exemplo de sistema que parece simples à primeira vista, mas que acaba por conter camadas intrincadas de complexidade.

A figura mostra a configuração básica: um pivô fixo, uma barra sem massa de comprimento L e um corpo com massa m no final, oscilando no plano da página. É mais fácil analisar o movimento em termos de torque e aceleração angular. Lembremos a versão angular da segunda lei de Newton

$$\sum \tau = I\alpha. \tag{6.9}$$

No lado esquerdo desta equação está a soma de todos os torques atuando no objeto. No lado direito, I é a inércia rotacional do objeto, mL^2 para o pêndulo. A aceleração angular α é

Figura 16 – Um pêndulo de massa m, sob a ação da força da gravidade.



Fonte: Elaborado pelo autor

definida análogo à aceleração comum

$$\alpha = \frac{\Delta\omega}{\Delta t} \quad \text{e} \quad \omega = \frac{\Delta\theta}{\Delta t}, \tag{6.10}$$

onde ω é a velocidade angular.

Pelo diagrama podemos ver que o torque devido a gravidade é

$$\tau_g - LF_g \text{sen}\theta = -Lmg\text{sen}\theta, \tag{6.11}$$

onde o sinal de menos indica que o torque é negativo (sentido horário) quando θ é positivo e vice-versa. Se não houver atrito ou outro torque atuando, então a lei de Newton (equação 6.9) diz

$$-Lmg\text{sen}\theta = mL^2\alpha \quad \text{ou} \quad \alpha = -\frac{g}{L}\text{sen}\theta. \tag{6.12}$$

Como a massa m foi cancelada da equação (6.12), não haverá necessidade de especificar uma massa em sua simulação de pêndulo. Podemos pensar que precisamos especificar um comprimento L e também um valor de g (dependendo de qual planeta o pêndulo vive), mas na verdade não precisa, porque ainda temos a liberdade de escolher nossas unidades para medir distância e tempo. Nossa liberdade de escolher unidades significa que quaisquer que sejam os valores reais de L e g, podemos simplesmente especificar que estamos usando unidades nas quais ambas são iguais a 1. Essas unidades são chamadas de unidades naturais, porque são naturais para a física que estamos estudando. Uma vantagem de usar unidades naturais é que a equação (6.12) se torna simplesmente $\alpha = -\text{sen}\theta$. Mas a vantagem real é que nossa simulação será aplicável a qualquer pêndulo em qualquer planeta. O preço é que, para aplicar nossos resultados

a um pêndulo específico, podemos precisar convertê-los de unidades naturais em unidades mais convencionais após executar a simulação.

Precisamos expressar a posição do pêndulo em coordenadas retangulares. Considerando que a origem está no pivô e as direções x e y para a direita e para cima, respectivamente, as fórmulas para x e y em termos de θ é

$$x = L\text{sen}\theta, \qquad y = -L\cos\theta.$$

A energia potencial do pêndulo pode ser modelada a partir da equação básica

$$E_p = mgh,$$

onde g é a aceleração da gravidade e h é a altura. Costumamos usar essa equação para modelar objetos em queda livre. No entanto, o pêndulo é restringido pela haste ou corda e não está em queda livre. Portanto, devemos expressar a altura em termos de θ, o ângulo e L, o comprimento do pêndulo. Assim, $h = L(1-\cos\theta)$. Logo, a energia potencial pode ser descrita por

$$E_p = mgL(1-\cos\theta).$$

A energia cinética é dado por $E_c = \frac{1}{2}mv^2$ mas $v = L\omega$, logo $E_c = \frac{1}{2}mL^2\omega^2$. Com isso, implementamos o código 75 abaixo.

Código-fonte 75 – Movimento de um pêndulo sob ação da gravidade.

```
1  import math
2  import matplotlib.pyplot as plt
3
4  theta = 0
5  omega = 10*math.pi/180 #inicial em radianos
6  t = 0
7  tmax = 10
8
9  dt = 0.01
10
11 tempo = [t]
12 posicaox = [math.sin(theta)]
13 posicaoy = [-math.cos(theta)]
14 angulo = [theta]
15
```

```
16 energiaP = [1-math.cos(theta)]
17 energiaC = [omega**2/2]
18 energiaTotal = [energiaP[-1]+energiaC[-1]]
19
20 while t <= tmax:
21   alpha = -math.sin(theta)
22   omega += alpha*dt
23   theta += omega*dt
24   t += dt
25
26   posicaox.append(math.sin(theta))
27   posicaoy.append(-math.cos(theta))
28   angulo.append(theta)
29
30   energiaP.append(1-math.cos(theta))
31   energiaC.append(omega**2/2)
32   energiaTotal.append(energiaP[-1]+energiaC[-1])
33
34   tempo.append(t)
35
36 plt.subplot(1,2,1)
37 plt.plot(tempo, posicaox)
38 plt.xlabel("t")
39 plt.ylabel("x")
40 plt.grid(True)
41
42 plt.subplot(1,2,2)
43 plt.plot(tempo, posicaoy)
44 plt.xlabel("t")
45 plt.ylabel("y")
46 plt.grid(True)
47
48 plt.show()
49
50 # grafico de energia cinetica, potencial e total
51 plt.plot(tempo,  energiaC)
52 plt.plot(tempo, energiaP)
53 plt.plot(tempo, energiaTotal)
54 plt.xlabel("tempo")
```

```
55 plt.ylabel("energia")
56 plt.legend(["Energia cinética","energia potencial","Energia Total"])
57 plt.grid(True)
58 plt.show()
```

Fonte: Elaborado pelo autor.

Agora vamos adicionar mais algumas complicações: fricção para desacelerar o pêndulo e um torque externo periódico que adiciona energia continuamente ao sistema. Uma maneira simples de adicionar atrito é subtrair um termo proporcional a ω do lado direito da equação (6.12) para a aceleração angular, ou seja

$$\alpha = -\frac{g}{L}\text{sen}\theta - c\omega. \tag{6.13}$$

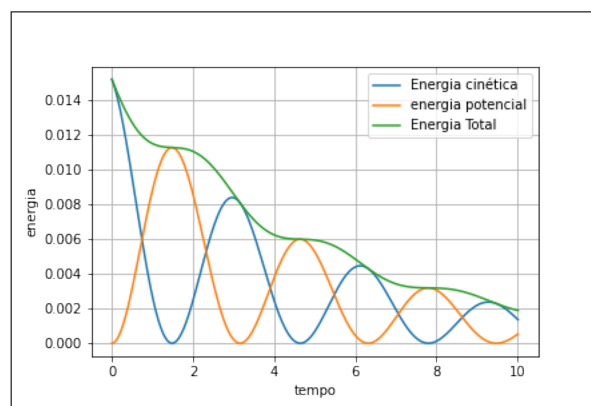
Uma força resistiva linear (ou torque) desta forma é chamada de amortecimento e o coeficiente c é chamado de constante de amortecimento.

No código (75), basta alterar a linha 21 com algum valor de c.

```
1 alpha = -math.sin(theta) - c*omega
```

Os graficos de energia tomam a forma agora como nas imagens 17 e 18.

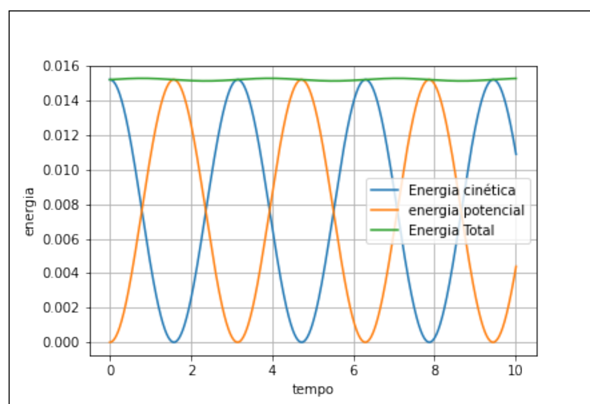
Figura 17 – Pêndulo com amortecimento.



Fonte: Elaborado pelo autor

O amortecimento remove a energia do pêndulo, então o movimento "morre". Mas podemos manter o movimento, aplicando continuamente um torque externo ao pêndulo. Uma maneira simples, porém interessante de fazer isso é adicionar um termo à equação (6.13) que

Figura 18 – Energia cinética, potencial e total.



Fonte: Elaborado pelo autor

seja senoidal no tempo, a saber

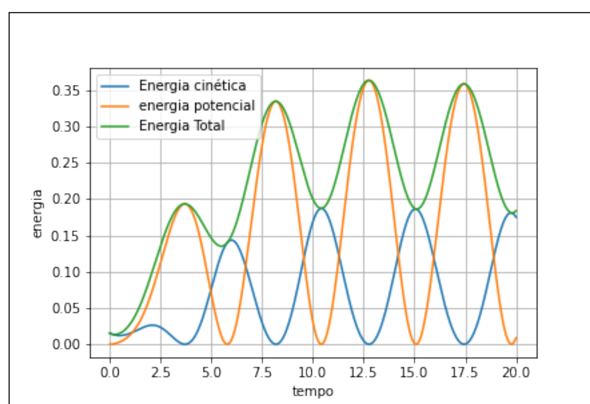
$$\alpha = -\frac{g}{L}\text{sen}\theta - c\omega + A\text{sen}(ft). \tag{6.14}$$

Este termo adicional representaria uma força de torção suave para a frente e para trás aplicada no pivô, que não é afetada pela posição e velocidade do pêndulo. As constantes A e f representam a amplitude e a frequência angular desse torque.

Usando a constante de amortecimento igual a $0,5$, a amplitude igual a $0,5$ e a frequência igual a 2/3, e mais uma vez modificando a linha 21, vemos que, após um comportamento "transitório" inicial, o movimento se torna periódico, como podemos vê na imagem 19.

```
alpha = -math.sin(theta) - c*omega + A*math.sin(f*t)
```

Figura 19 – Pêndulo com um torque externo.

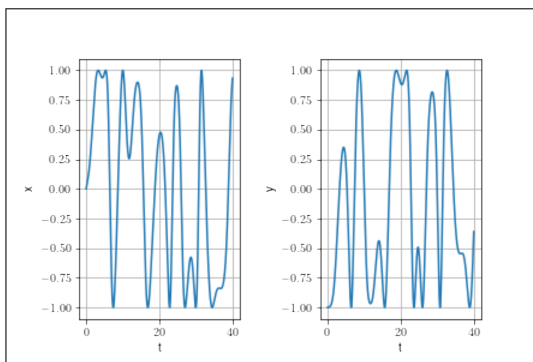


Fonte: Elaborado pelo autor.

Agora, aumentando a amplitude para 1,2, o pêndulo pode oscilar sobre o pivô e o movimento se torna muito mais complexo. Em alguns casos, como vimos nesse exemplo,

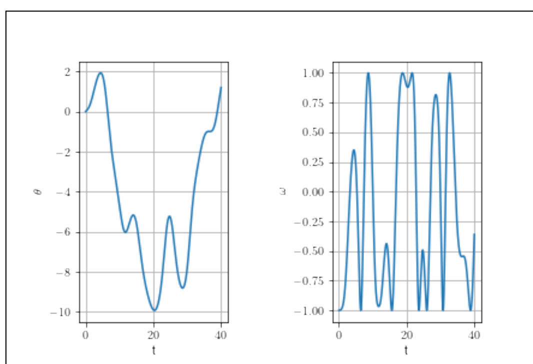
o movimento nunca se estabiliza nem se torna periódico. Como o movimento parece tão imprevisível, a palavra "caos" muitas vezes vem à mente.

Figura 20 – Posição para amplitude igual a 1,2.



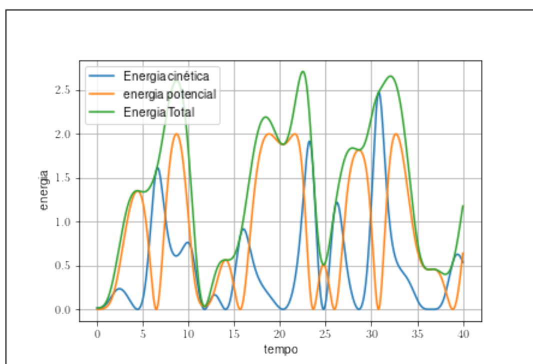
Fonte: Elaborado pelo autor.

Figura 21 – Velocidades para amplitude igual a 1,2.



Fonte: Elaborado pelo autor.

Figura 22 – Energia para amplitude igual a 1,2.



Fonte: Elaborado pelo autor.

Caos é, na verdade, um termo técnico em dinâmica, usado para descrever um

comportamento imprevisível na prática, porque mesmo uma pequena mudança na condição inicial resulta em uma mudança exponencialmente crescente no movimento.

6.3 Implementação em Fortran

Os códigos seguintes mostram as implementações em FORTRAN equivalentes das funções Python desenvolvidas durante esse capítulo. As rotinas correspondentes têm nomes, parâmetros e funcionalidades idênticos.

Código-fonte 76 – Simular movimento de projétil em uma dimensão.

```
1  program main
2
3    implicit none
4
5    real ( kind = 8 ) :: gravidade , y, vy, t, ay
6    real ( kind = 8 ) :: dt
7    integer , parameter :: N = 100
8    integer :: i
9
10   real ( kind = 8 ), dimension(N) :: tempo
11   real ( kind = 8 ), dimension(N) :: posicao
12   real ( kind = 8 ), dimension(N) :: velocidade
13
14   gravidade = 9.8
15   y = 10.e0
16   vy = 0.e0
17   t = 0e0
18
19   ay = -gravidade
20
21   dt = 1.e0/N
22
23   do i=1, N
24     tempo(i) = t
25     posicao(i) = y
26     velocidade(i) = vy
27
```

```fortran
28       y = y + vy * dt
29       vy = vy + ay * dt
30       t = t + dt
31     end do
32
33     do i=1, N
34       print*, i, tempo(i), posicao(i),velocidade(i)
35     end do
36
37   end program main
```

Fonte: Elaborado pelo autor.

Código-fonte 77 – Trajetória de um projétil com e sem resistência.

```fortran
1    subroutine trajetoriaProjetil(x, y, v, theta, t, m, g, c, tempo,
         posicao, velocidade, N)
2
3      ! x,y        : posicao inicial
4      ! v        : velocidade inicial
5      ! theta      : ângulo inicial
6      ! t        : tempo inicial
7      ! m        : massa do projétil
8      ! g        : aceleração da gravidade
9      ! c        : constantede resistência
10     ! tempo      : lista  para guardar os tempo para plotagem
11     ! posicao   : posicao sucessivas do projétil
12     !        : posicao(1) -> componente horizontal
13     !        : posicao(2) -> componente vertical
14     ! velocidade : lista com velocidades sucessivas do projétil
15     !        : velocidade(1) -> componente horizontal
16     !        : velocidade(2) -> componente vertical
17
18     implicit none
19
20     real ( kind = 8 ), parameter :: PI = 3.1415926
21
22     real ( kind = 8 ) :: x, y, v, theta
23     real ( kind = 8 ) :: t, m, c, dt, vx, vy
```

```
24  real ( kind = 8 ) :: res
25  real ( kind = 8 ) :: ax, ay, g
26  integer :: N, i
27
28  real ( kind = 8 ), dimension(N, N) :: posicao
29  real ( kind = 8 ), dimension(N, N) :: velocidade
30  real ( kind = 8 ), dimension(N) :: tempo
31
32  real ( kind = 8 ), dimension(N) :: posicaox
33  real ( kind = 8 ), dimension(N) :: posicaoy
34
35  real ( kind = 8 ), dimension(N) :: velocidadex
36  real ( kind = 8 ), dimension(N) :: velocidadey
37
38  vx = v*cos(theta*PI/180)
39  vy = v*sin(theta*PI/180)
40
41  res = c/m
42  dt = 1.e0/N
43
44  do i = 1, N
45    ax = -res * V * vx
46    ay = -g - res * v * vy
47
48    tempo(i) = t
49    posicaox(i) = x
50    posicaoy(i) = y
51    velocidadex(i) = vx
52    velocidadey(i) = vy
53
54    vx = vx + ax * dt
55    vy = vy + ay * dt
56
57    x = x + (vx + ax * dt/2) * dt
58    y = y + (vy + ay * dt/2) * dt
59
60    t = t + dt
61  end do
62
```

```
63      do i = 1, N
64        posicao(1, i) = posicaox(i)
65        posicao(2, i) = posicaoy(i)
66
67        velocidade(1, i) = velocidadex(i)
68        velocidade(2, i) = velocidadey(i)
69      end do
70
71    end subroutine trajetoriaProjetil
72
73
74    program main
75
76      implicit none
77
78      real ( kind = 8 ) :: g, x, y, v, theta, t, m
79      real ( kind = 8 ) :: cComRes, cSemRes
80
81      integer, parameter :: N = 100
82      integer :: i
83
84      real ( kind = 8 ), dimension(N, N) :: posicaoSemRes
85      real ( kind = 8 ), dimension(N, N) :: velocidadeSemRes
86
87        real ( kind = 8 ), dimension(N, N) :: posicaoComRes
88      real ( kind = 8 ), dimension(N, N) :: velocidadeComRes
89
90        real ( kind = 8 ), dimension(N) :: tempo
91
92      g = 9.8e0
93      x = 0.e0
94      y = 0.e0
95      v = 50.e0
96      theta = 35.e0
97      t =0.e0
98
99      m = 0.145
100     cComRes = 0.0012
101     cSemRes = 0.e0
```

```
102
103    call trajetoriaProjetil(x, y, v, theta, t, m, g, cSemRes, &
104      tempo, posicaoSemRes, velocidadeSemRes, N)
105
106    call trajetoriaProjetil(x, y, v, theta, t, m, g, cComRes, &
107      tempo, posicaoComRes, velocidadeComRes, N)
108
109    do i = 1, N
110      print*, tempo(i), posicaoSemRes(1,i), posicaoSemRes(2, i)
111    end do
112
113  end program main
```

Fonte: Elaborado pelo autor.

Código-fonte 78 – Movimento de um pêndulo sob ação da gravidade.

```
1   program main
2
3     implicit none
4
5     real ( kind = 8 ), parameter :: PI = 3.1415926
6     integer, parameter :: N = 100
7
8     real ( kind = 8 ) :: theta, omega, t,alpha
9     real ( kind = 8 ) :: c, A, f
10    real ( kind = 8 ) :: dt
11    integer :: i
12
13    real ( kind = 8 ), dimension(N) :: tempo
14    real ( kind = 8 ), dimension(N) :: posicaox
15    real ( kind = 8 ), dimension(N) :: posicaoy
16
17      real ( kind = 8 ), dimension(N) :: angulo
18    real ( kind = 8 ), dimension(N) :: velocidade
19
20      real ( kind = 8 ), dimension(N) :: energiaP
21    real ( kind = 8 ), dimension(N) :: energiaC
22    real ( kind = 8 ), dimension(N) :: energiaTotal
```

```
23
24    theta = 0.e0
25    omega = 10.e0 * PI / 180
26    t = 0.e0
27    c = 0.5
28    A = 1.2
29    f = 2.e0/3
30
31    dt = 1.e0/N
32
33    do i = 1, N
34      alpha = -sin(theta) - c * omega + A * sin(f * t)
35      omega = omega + alpha * dt
36      theta = theta + omega * dt
37      t = t + dt
38
39      posicaox(i) = sin(theta)
40      posicaoy(i) = -cos(theta)
41
42          angulo(i) = theta
43      velocidade(i) = omega
44
45      energiaP(i) = 1 - cos(theta)
46      energiaC(i) = omega * omega / 2
47      energiaTotal(i) = energiaP(i) + energiaC(i)
48
49      tempo(i) = t
50    end do
51
52    do i = 1, N
53      print*, i, posicaox(i), energiaTotal(i)
54    end do
55
56  end program main
```

Fonte: Elaborado pelo autor.

7 CONSIDERAÇÕES FINAIS

A matemática computacional e a computação científica estão preocupadas com a computação, ou seja, o procedimento de realmente produzir números e gráficos como soluções para problemas definidos por modelos matemáticos. Com a introdução dos computadores em meados da década de 1940, valeu a pena construir métodos numéricos, mesmo que a implementação exigisse milhões de operações aritméticas básicas. Em pouco tempo, a computação científica tornou-se uma terceira ferramenta científica como um complemento às ferramentas clássicas de teoria e experimentos.

Para a equação quadrática $x^2 + ax + b = 0$ existe uma fórmula simples para sua solução, $x = -\frac{a}{2} \pm \sqrt{\frac{a^2}{4} - b}$, que pode ser avaliada para a e b dados. Para equações de grau superior contendo os termos x^3 e x^4, existem fórmulas explícitas para as soluções. No entanto, para equações de grau superior, não existem tais fórmulas gerais. Assim, para equações polinomiais gerais $x^n + an - 1x^{n-1} + \cdots + a_0 = 0$, a solução x pode ser encontrada por meios analíticos se $n \leq 4$, mas algum procedimento numérico deve ser usado para avaliar as expressões matemáticas envolvidas. Para $n \geq 5$, uma solução aproximada deve ser encontrada aplicando um método numérico diretamente à equação.

Dados $n+1$ valores de função $y_i = f(x_i)$, o problema de interpolação é encontrar um polinômio $p(x) = a_0 + a_1 x + \cdots + a_n + x^n$ tal que $p(x_i) = f(x_i)$. A solução para este problema é única e pode ser definida por um sistema algébrico de equações lineares. Pode parecer que o problema de interpolação não é grande coisa, devemos apenas resolver o sistema. No entanto, para n grande, esse sistema não é muito fácil de resolver. Diferentes fórmulas para avaliar $p(x)$ foram desenvolvidas.

Com a introdução do cálculo diferencial e integral veio o problema de calcular o valor de integrais definidas. Apenas para funções especiais $f(x)$ é possível encontrar uma função primitiva $F(x)$ como $dF/dx = f$ por meios analíticos. Às vezes, a fórmula explicita de $f(x)$ nem mesmo é conhecida, mas seus valores são conhecidos em certos pontos discretos. Portanto, é necessário construir métodos numéricos para encontrar valores precisos da integral definida.

Se um corpo localizado em x_0 quando $t = 0$ move-se com velocidade constante v, o modelo matemático pode ser descrito como $\frac{dx}{dt} = v$, $x(0) = x_0$. Este é um problema de valor inicial envolvendo uma equação diferencial e uma condição inicial. Aqui estamos procurando uma solução na forma de uma função $x(t)$, e é facilmente encontrada como $x(t) = x_0 + vt$, que é uma expressão explícita que pode ser avaliada para qualquer valor de t. No entanto, quando os

coeficientes dependem de t, em geral não existe uma forma explícita para a solução. É ainda pior quando a equação diferencial não é linear. Nem fica claro se existe uma solução e, se existe, como encontrá-lo por meios analíticos. No entanto, existem métodos numéricos muitos eficazes baseadas na discretização, onde as derivadas são substituídas por fórmulas de diferenças finitas de modo que a solução possa ser calculada com uma precisão muito alta. Euler provavelmente foi o primeiro a usar um método de diferenças finitas para resolver uma equação diferencial. Para sistemas de equações diferenciais ordinárias, uma longa série de variações no principio básico foi posteriormente desenvolvido com os métodos de Runge-Kutta como um dos mais importantes.

Imaginemos que uma pessoa veio do passado e devemos contar a ela as vantagens da alfabetização. Teremos uma grande empreitada - ele confia plenamente em sua memória e não teve convívio com livros. Queremos explicar o mundo das ideias que podem ser encontradas nos livros, mas tudo o que ele consegue ver são símbolos engraçados. Sabemos que a leitura enriqueceu nossa vida, e queremos dizer isso. Mas temos medo que ele ria.

Agora imaginemos que somos enviado para o futuro, um futuro no qual os computadores abrem a porta para a matemática como os livros abrem a porta para a alfabetização. Nosso guia diz que existe uma fértil experiência humana em matemática, um conhecimento tão gratificante quanto qualquer outra proporcionado pelos livros, uma vivência que nos foi negada. Somos informado de uma bela paisagem matemática, mas tudo o que podemos ver são símbolos engraçados. Nosso guia tenta nos mostrar que seremos uma pessoa melhor e rimos.

Estamos no prelúdio de uma revolução na matemática - como pensamos sobre ela, como a praticamos e como a aprendemos. A revolução está centrada no computador como uma ferramenta matemática, uma ferramenta que está se tornando reconhecida como fundamental para a matemática, assim como os livros são para a alfabetização. A revolução matemática pode, no devido tempo, se tornar tão importante quanto a revolução do computador que a precedeu e a trouxe à existência.

Como em toda revolução, as pessoas estão tomando partido - algumas gostariam de ver uma exploração completa do papel do computador na matemática humana, enquanto outras acham que usar um computador na matemática é trapaça. Consideramos que desenvolver uma pesquisa que alia os conhecimentos de uma linguagem de programação, e computação em geral, com conhecimentos matemáticos é uma grande avanço para desenvolvermos habilidades e resolvermos problemas que combinam percepções de uma ou mais disciplinas das ciências naturais. Isso fornece uma combinação única de habilidades e conhecimentos teóricos e aplicados.

Essas habilidades pode nos mudar mais profundamente do que imaginamos. Ideias matemáticas podem chegar a igualar palavras em riqueza de expressão, fornecendo uma nova linguagem humana que não favorece nenhum território e não requer tradução.

REFERÊNCIAS

ARACHNOID.COM. **Computer Math, Exploring a new frontier beyond the realm of human calculation**. 2008. Disponível em: https://arachnoid.com/lutusp/computermath.html. Acesso em: 3 dez 2020.

ARAÚJO, A. **Matemática Computacional**. Coimbra: Universidade de Coimbra, 2011.

BEAZLEY, D.; JONES, B. K. **Python Cookbook: Recipes for Mastering Python 3** . [S.l.]: "O'Reilly Media, Inc.", 2013.

BEU, T. **Introduction to Numerical Programming:** A Practical Guide for Scientists and Engineers Using Python and C/C++. [S.l.]: Taylor & Francis, 2014. (Series in Computational Physics).

BORGES, L. E. **Python para desenvolvedores: aborda Python 3.3**. [S.l.]: Novatec, 2014.

BOSE, S. K. **Numerical Methods of Mathematics Implemented in Fortran**. [S.l.]: Springer, 2019.

BURDEN, R.; FAIRES, J. **Numerical Analysis**. [S.l.]: Brooks/Cole, Cengage Learning, 2011.

CANTAO, L. A. P. **Cálculo Numérico e Computacional - cnc**. São Paulo: UNESP, 2010.

CHAPMAN, S. J. **Fortran for scientists and engineers**. [S.l.]: McGraw-Hill Education, 2018.

DEMIDOVICH, B.; MARON, I. Computational mathematics (translated by g. yankovsky)**Mir, Moskva**, v. 1, 1987.

DEPARTAMENT OF PHYSICS, UNIVERSITY IN ST. LOUIS. Disponível em: https://web.physics.wustl.edu/~wimd/topic01.pdf. Acesso em: 2 dez 2020.

DICKINSON SCHOLAR. **VPython for Introductory Mechanics:** Complete Version. 2019. Disponível em: https://scholar.dickinson.edu/vpythonphysics/?utm_source=scholar.dickinson.edu%2Fvpythonphysics%2F1&utm_medium=PDF&utm_campaign=PDFCoverPages. Acesso em: 12 dez 2020.

DOMINGUES, R. O.; SIQUEIRA, A. S.**Teoria, Implementaçao e Comparaçao de Métodos de Integraçao Numérica**. [S.l.: s.n.], 2016.

FOURIER.ENG.HMC.EDU. Disponível em: http://fourier.eng.hmc.edu/e176/lectures/ch6/ch6.html. Acesso em: 20 nov 2020.

GRAÇA, M.; LIMA, P. **Apontamentos de Matemática Computacional**. Lisboa: [s.n.], 2016.

GUSTAFSSON, B.**Scientific Computing from a Historical Perspective** [S.l.]: Springer, 2018. v. 17.

HÜROL, S. **Numerical Methods for Solving Systems of Ordinary Differential Equations** . 2013. 135f. Tese (Doutorado em Matemática) – Eastern Mediterranean University, 2013.

IBM. 2011. Disponível em: https://www.ibm.com/ibm/history/ibm100/us/en/icons/fortran/. Acesso em: 6 nov 2020.

KANSCHAT, G.**Numerical Analysis of Ordinary Differential Equations**. 2018. Disponível em: https://www.mathsim.eu/~gkanscha/notes/ode.pdf. Acesso em: 12 dez 2020.

LABAKI, J.; WOISKI, E. Introdução a python–módulo a. **Grupo Python, UNESP-Ilha Solteira**, v. 2, 2003.

MENEZES, N. N. C. **Introdução a programação com Python**. São Paulo: Novatec, 2010.

PETERSDORFF, T. von. A short proof for romberg integration. **The American Mathematical Monthly**, v. 100, n. 8, p. 783–785, 1993. Disponível em: http://www.jstor.org/stable/2324787. Acesso em: 12 dez 2020.

RAY, S. S. **Numerical analysis with algorithms and programming**. [S.l.]: CRC Press, 2018.

ROSSUM, G. van. **Python Tutorial**. 1995. Disponível em: https://ir.cwi.nl/pub/5007/05007D.pdf.

SOLOMON, J. **Numerical algorithms:** methods for computer vision, machine learning, and graphics. [S.l.]: CRC press, 2015.

STOER, J.; BULIRSCH, R. **Introduction to numerical analysis**. [S.l.]: Springer Science & Business Media, 2013. v. 12.

TELECOMUNICAÇOES, C. d. E. de; TUTORIAL, P. de E.; GRUPO, P. Tutorial de introduçao ao python. 2011.

TRINITY COLLEGE DUBLIN, THE UNIVERSITY OF DUBLIN. 2016. Disponível em: https://www.tcd.ie/Physics/people/Charles.Patterson/teaching/PY2050_CP/New_Scripts/Projectile_Motion.pdf. Acesso em: 5 nov 2020.

TUTORIALS POINT. **Matplotlib Tutorial**. 2020. Disponível em: https://www.tutorialspoint.com/matplotlib/index.htm. Acesso em: 17 dez 2020.

TUTORIALS POINT. **Fortran Tutorial**. 2021. Disponível em: https://www.tutorialspoint.com/fortran/index.htm. Acesso em: 24 nov 2020.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. **Recursos Educacionais Abertos de Matemática**. 2020. Disponível em: https://www.ufrgs.br/reamat/CalculoNumerico/index.html. Acesso em: 7 nov 2020.

UNIVERSITETET I OSLO. **Numerical Solution of Differential Equations**. 2008. Disponível em: https://www.uio.no/studier/emner/matnat/math/MAT-INF1100/h08/kompendiet/differential-eqs.pdf. Acesso em: 9 nov 2020.

WEBER STATE UNIVERSITY. **Python Manual**. 2018. Disponível em: http://physics.weber.edu/schroeder/scicomp/PythonManual.pdf. Acesso em: 21 dez 2020.

WIKIPEDIA. **Divided differences** . 2015. Disponível em: https://en.wikipedia.org/wiki/Divided_differences. Acesso em: dez 2020.

WIKIVERSITY. **Cubic Spline Interpolation**. 2020. Disponível em: https://en.wikiversity.org/wiki/Cubic_Spline_Interpolation. Acesso em: dez 2020.