UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
DEPARTAMENTO DE CIÊNCIAS EXATAS E NATURAIS
BACHARELADO INTERDISCIPLINAR EM CIÊNCIA E TECNOLOGIA

GUILHERME DE FRANÇA VASCONCELOS

AVALIAÇÃO DA APLICAÇÃO DE JULIA NO ENSINO DE CÁLCULO NUMÉRICO:
UM ESTUDO DE CASO DAS EQUAÇÕES DIFERENCIAIS ORDINÁRIAS

PAU DOS FERROS
2025

GUILHERME DE FRANÇA VASCONCELOS

AVALIAÇÃO DA APLICAÇÃO DE JULIA NO ENSINO DE CÁLCULO NUMÉRICO: UM ESTUDO DE CASO DAS EQUAÇÕES DIFERENCIAIS ORDINÁRIAS

Monografia apresentada à Universidade Federal Rural do Semi-Árido como requisito parcial para obtenção do título de Bacharel Interdisciplinar em Ciência e Tecnologia.

Orientador: Bruno Francisco Xavier, Prof. Dr.

PAU DOS FERROS

2025

V331a Vasconcelos, Guilherme de França.
Avaliação da aplicação de Julia no ensino de Cálculo Numérico: um estudo de caso das Equações Diferenciais Ordinárias / Guilherme de França Vasconcelos. - 2025.
59 f. : il.

Orientador: Bruno Francisco Xavier.
Monografia (graduação) - Universidade Federal Rural do Semi-árido, Curso de Ciência e Tecnologia, 2025.

1. Linguagem Julia;. 2. Métodos numéricos. 3. Equações diferenciais ordinárias. I. Xavier, Bruno Francisco, orient. II. Título.

Ficha catalográfica elaborada por sistema gerador automático em conformidade com AACR2 e os dados fornecidos pelo) autor(a).
Biblioteca Campus Pau dos Ferros
Bibliotecária: Erlanda Maria Lopes da Silva
CRB: 15/966

GUILHERME DE FRANÇA VASCONCELOS

AVALIAÇÃO DA APLICAÇÃO DE JULIA NO ENSINO DE CÁLCULO NUMÉRICO:
UM ESTUDO DE CASO DAS EQUAÇÕES DIFERENCIAIS ORDINÁRIAS

Monografia apresentada à Universidade Federal Rural do Semi-Árido como requisito parcial para obtenção do título de Bacharel Interdisciplinar em Ciência e Tecnologia.

Defendida em: 22 / 12 / 2025

BANCA EXAMINADORA

Bruno Francisco Xavier, Prof. Dr. (UFERSA)
Presidente

Francisco Carlos Gurgel da Silva Segundo, Prof. Dr. (UFERSA)
Membro Examinador

Reudismam Rolim de Sousa, Prof. Dr. (UFERSA)
Membro Examinador

AGRADECIMENTOS

Agradeço primeiramente à minha família, aos meus pais, José Vasconcelos da Silva e Maria Dantas de França Silva, e ao meu irmão, Gustavo de França Vasconcelos, por todo apoio, paciência e incentivo em todos os momentos.

Agradeço especialmente ao meu orientador, Prof. Dr. Bruno Francisco Xavier, pelo conhecimento compartilhado, pela orientação dedicada, pela paciência e conselhos fundamentais para a realização deste trabalho.

Agradeço à banca examinadora, pelo interesse e disponibilidade.

Por fim, agradeço a todos que, de alguma forma, contribuíram para a concretização deste trabalho.

“It gets easier. Every day it gets a little easier. But you gotta do it every day – that’s the hard part. But it does get easier”

Raphael Bob-Waksberg (Bojack Horseman)

RESUMO

Este trabalho investiga o uso da linguagem Julia e de seu ecossistema científico como ferramenta de apoio ao ensino e aprendizagem de métodos numéricos aplicados à solução de Equações Diferenciais Ordinárias (EDO). A partir de uma abordagem experimental, foram implementados os principais métodos trabalhados em disciplinas de Cálculo Numérico como Euler, Heun e Runge-Kutta; afim de comparar sua precisão em diferentes circunstâncias. Além disso, alguns sistemas físicos e químicos clássicos como o pêndulo simples e o decaimento do Carbono-14, foram utilizados como estudo de caso para análise e visualização utilizando os métodos estudados e o pacote especializado `DiferentialEquations.jl`. Além dos experimentos computacionais, o trabalho examina o potencial pedagógico de Julia no contexto da disciplina de Cálculo Numérico. A linguagem apresenta vantagens relevantes, como sintaxe expressiva, proximidade com a notação matemática e suporte nativo a Unicode. O ambiente interativo Pluto.jl mostrou-se especialmente adequado para demonstrações em aula, promovendo experimentação, visualização imediata e aprendizagem ativa. Os resultados indicam que Julia pode favorecer tanto a implementação de algoritmos quanto a compreensão conceitual por parte dos estudantes, representando uma alternativa promissora para o ensino de métodos numéricos e para a integração entre a parte teórica e prática no estudo de EDOs.

Palavras-chave: linguagem Julia; métodos numéricos; equações diferenciais ordinárias

ABSTRACT

This work investigates the use of the Julia language and its scientific ecosystem as a support tool for the teaching and learning of numerical methods applied to the solution of Ordinary Differential Equations (ODE). Using an experimental approach, the main methods addressed in Numerical Analysis courses, such as Euler, Heun, and Runge-Kutta, were implemented to compare their accuracy under different circumstances. Furthermore, classical physical and chemical systems, such as the simple pendulum and Carbon-14 decay, were used as case studies for analysis and visualization using the studied methods and the specialized package `DifferentialEquations.jl`. In addition to the computational experiments, this work examines Julia's pedagogical potential within the context of the Numerical Analysis discipline. The language presents relevant advantages, such as expressive syntax, closeness to mathematical notation, and native Unicode support. The Pluto.jl interactive environment proved particularly suitable for classroom demonstrations, promoting experimentation, immediate visualization, and active learning. The results indicate that Julia facilitates both the implementation of algorithms and conceptual understanding by students, representing a promising alternative for teaching numerical methods and for integrating theory and practice in the study of ODEs.

Keywords: Julia language; numerical methods; ordinary differential equations.

LISTA DE FIGURAS

Figura 1 – Gráfico das funções seno e cosseno de 0 até 2π 22
Figura 2 – Método de Euler aplicado com diferentes números de passos 29
Figura 3 – Método de Heun aplicado com diferentes números de passos 30
Figura 4 – Decaimento do carbono-14 usando métodos numéricos 37
Figura 5 – Configuração do sistema Pêndulo Simples. 38
Figura 6 – Simulação utilizando o método de Runge-Kutta com número de passos diferentes 39
Figura 7 – Utilização de um slider para alterar o número de passos 45
Figura 8 – Comparação entre a solução real e a numérica usando o método de passo adaptativo Tsit5. 50
Figura 9 – Evolução temporal Pêndulo Simples utilizando o método Tsit5. 51

LISTA DE TABELAS

Tabela 1 – Comparação entre Euler (h = 0,5) e a solução analítica. 27
Tabela 2 – Comparação entre Euler (h = 0,25) e a solução analítica. 27
Tabela 3 – Comparação entre Euler (h = 0,1) e a solução analítica. 28
Tabela 4 – Comparação entre Heun e a solução analítica. 30
Tabela 5 – Comparação entre RK4 (h = 0,5) e a solução analítica. 34
Tabela 6 – Comparação entre os erros relativos 36
Tabela 7 – Comparação entre o valor analítico e as aproximações para o decaimento do carbono-14 usando DifferentialEquations.jl 49
Tabela 8 – Comparação entre o valor analítico e aproximado para o decaimento do carbono-14 utilizando o método de passo adaptativo `Tsit5` 49
Tabela 9 – Gasto de tempo e a alocação de memória para a solução dos problemas . . . 55

SUMÁRIO

1 INTRODUÇÃO 11

1.1 Justificativa 12

1.2 Objetivos 12

1.2.1 Objetivo geral 12

1.2.2 Objetivos específicos 12

1.3 Metodologia 13

2 FUNDAMENTAÇÃO TEÓRICA 14

2.1 Linguagem de Programação Julia 14

2.1.1 Características 14

2.1.2 O Ambiente Pluto.jl 15

2.1.3 Estruturas básicas 16

2.1.4 Estruturas de dados fundamentais 19

2.1.5 Gráficos 21

2.1.6 Mais sobre Julia 22

2.2 Equações Diferenciais Ordinárias 22

2.2.1 Classificações 23

2.2.2 Solução de EDOs 24

2.3 Métodos Numéricos 25

2.3.1 Análise de erros 26

2.3.2 Método de Euler 26

2.3.3 Método de Heun 28

2.3.4 Métodos de Runge-Kutta 31

2.3.5 Sistemas Dinâmicos de Segunda Ordem 34

3 ESTUDOS DE CASO 36

3.1 Decaimento do carbono-14 36

3.2 Pêndulo simples 37

4 IMPLEMENTAÇÃO EM JULIA 40

4.1 Método de Euler 40

4.2 Método de Heun 42

4.3 Método de Runge-Kutta 4 43

4.4 Sistemas de Segunda Ordem 43

5 SOLUÇÃO EFICIENTE DE EQUAÇÕES DIFERENCIAIS EM JULIA 46

5.1 O Ecossistema 46

5.1.1 Problema do Carbono-14 com DifferentialEquations.jl 48

5.1.2 Pêndulo Simples com DifferentialEquations.jl 50

6 RESULTADOS 52

6.1 Perspectivas de Uso da Linguagem Julia na Disciplina de Cálculo Numérico 52

6.1.1 Perspectivas por Conteúdo da Ementa do Componente Curricular 52

6.1.2 Considerações Finais sobre as Perspectivas Pedagógicas 56

7 CONCLUSÕES E TRABALHOS FUTUROS 57

REFERÊNCIAS 58

1 INTRODUÇÃO

A modelagem matemática de fenômenos físicos, biológicos e de engenharia depende, fundamentalmente, da capacidade de descrever como as grandezas variam em relação ao tempo ou ao espaço. Essa descrição é feita por meio de Equações Diferenciais (EDs), sendo utilizada em áreas como a dinâmica de populações, reações químicas, análise de circuitos elétricos e sistemas mecânicos (Zill, 2016).

A teoria das equações diferenciais fornece métodos analíticos para a resolução de uma variedade de problemas. No entanto, a grande maioria dos problemas reais apresenta características que dificultam a utilização desses métodos. Nesse contexto, os métodos numéricos se tornam essenciais, pois permitem obter soluções aproximadas (Chapra, 2013).

Em cursos de engenharia e ciências exatas, o componente curricular Cálculo Numérico é responsável por introduzir esses métodos. Contudo, o processo de ensino-aprendizagem enfrenta desafios significativos. Entre eles, está a persistência de metodologias de ensino tradicionais; muitas vezes caracterizadas por uma abordagem passiva, na qual há uma supervalorização de conteúdos e pouca contribuição dos alunos no planejamento de atividades (Cruz, 2024). A ausência de metodologias ativas, como a resolução de problemas reais e a modelagem matemática, contribui para que os discentes não percebam a aplicabilidade dos conceitos, o que é um fator crítico para a desmotivação (Gama; Gomes; Pires, 2018).

Amaral, Leite e Silva (2013) ressaltam a importância da inclusão de ferramentas computacionais como apoio à aprendizagem dos alunos, pois permitem criar situações de aprendizagem estimulantes. Os autores mencionam ainda que, mesmo quando tecnologias são introduzidas, elas são frequentemente subutilizadas ou aplicadas de maneira simplificada, sem explorar suas potencialidades para fomentar uma aprendizagem significativa. Mota (2012) reforça que, sem o uso adequado de ferramentas computacionais, a aprendizagem de métodos numéricos também se torna desestimulante.

A implementação prática dos métodos numéricos requer habilidades de programação que muitos alunos ainda não dominam plenamente. O uso de linguagens como C ou Fortran pode representar uma barreira adicional devido à complexidade de sintaxe e falta de ambientes matemáticos adequados para visualização imediata (Finger *et al.*, 2021).

Dessa forma, a superação desses desafios exige uma reformulação curricular que privilegie a interdisciplinaridade, a atualização das ferramentas tecnológicas (como o uso de softwares livres) e a adoção de estratégias pedagógicas que valorizem a visualização gráfica e a simulação,

permitindo ao aluno construir o conhecimento de forma ativa e contextualizada (Azevedo; Silva, 2025; Sassano *et al.*, 2021; Silva, 2021).

Nesse contexto, a linguagem Julia apresenta um conjunto de características que a tornam particularmente adequada ao ensino e aprendizagem de métodos numéricos, especialmente quando integrada ao ecossistema Pluto.jl, que é um ambiente de programação projetado para o ensino e que permite a criação de notebooks interativos.

1.1 Justificativa

A justificativa deste trabalho reside na necessidade de modernizar o ensino de métodos numéricos, adotando ferramentas que reduzam a barreira de entrada para a programação e permitam que o aluno aprenda os conceitos básicos ao mesmo tempo em que compreende a utilização de solucionadores profissionais. A adoção de Julia, aliada a ambientes interativos, pode enriquecer o processo de ensino-aprendizagem de métodos numéricos, especificamente na resolução de EDOs.

1.2 Objetivos

1.2.1 Objetivo geral

Este trabalho tem como objetivo discutir qualitativamente o potencial da linguagem Julia e de seu ecossistema, como ferramenta para o ensino-aprendizagem da solução numérica de EDOs.

1.2.2 Objetivos específicos

- Apresentar os conceitos fundamentais de EDOs e dos métodos numéricos clássicos utilizados para sua solução;
- Desenvolver exemplos práticos de resolução de EDOs utilizando Julia, demonstrando o passo a passo para a execução dos métodos numéricos;
- Avaliar, por meio de estudos de caso e visualizações gráficas, como a interatividade e a visualização de resultados em Julia pode contribuir para a compreensão conceitual dos alunos em cursos de Cálculo Numérico.

1.3 Metodologia

Este trabalho caracteriza-se como um relato de experiência de natureza exploratória, no qual se investigou a utilização da linguagem Julia como ferramenta de apoio ao ensino-aprendizagem de métodos numéricos para a solução de EDOs.

Após uma breve revisão do assunto (Capítulo 2), foram realizados estudos de caso seguidos de implementações em Julia para análise didática.

Na primeira etapa, foram solucionados problemas de valor inicial de fenômenos físico-químicos clássicos, incluindo EDOs lineares e não lineares de segunda ordem, usando métodos numéricos.

A implementação seguiu duas etapas: uma versão semelhante ao que um aluno faria com lápis e papel e outra criando uma função específica. Além disso, foram utilizados gráficos para a visualização das soluções.

Finalmente, foi analisado o potencial pedagógico de Julia no ensino-aprendizagem de cálculo numérico, considerando critérios como legibilidade e expressividade do código, possibilidade de experimentação interativa, eficiência computacional e integração com recursos de visualização e interpretação de resultados.

Os resultados obtidos foram analisados de forma qualitativa, destacando evidências observadas durante a implementação e a experimentação dos métodos, sem a pretensão de generalização estatística, mas com foco na identificação das vantagens da linguagem Julia no contexto do ensino de métodos numéricos.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Linguagem de Programação Julia

Julia[1] é uma linguagem de programação *open-source*, de alto nível, de propósito geral, criada com foco em computação científica e numérica de alto desempenho. O projeto foi iniciado em 2009 por um grupo de pesquisadores do MIT (*Massachusetts Institute of Technology*) formado por Jeff Bezanson, Stefan Karpinski, Viral B. Shah e Alan Edelman.

A primeira versão de Julia foi lançada em 2012 com a proposta central de oferecer uma linguagem com a simplicidade e produtividade de linguagens de alto nível (como MATLAB ou Python) ao mesmo tempo em que entrega desempenho próximo ao C/Fortran.

Esta seção explora os fundamentos da linguagem Julia, analisando suas características, seu posicionamento frente a outras linguagens consolidadas, suas áreas de aplicação e as estruturas básicas.

2.1.1 Características

O desempenho e a flexibilidade de Julia não são acidentais, mas sim o resultado de um conjunto de decisões de design e escolha de recursos que funcionam bem entre si.

O código é compilado automaticamente para código nativo eficiente. O compilador LLVM, o mesmo utilizado por projetos como Clang (C/C++) e Swift, garante a geração de um código altamente otimizado por meio da compilação *Just In Time* (JIT) (Bezanson *et al.*, 2015).

A linguagem é dinamicamente tipada, ou seja, as variáveis não precisam ter seus tipos declarados. No entanto, pode-se adicionar anotações de tipo a fim de evitar erros e, mais importante, fornecer informações ao compilador para que ele possa gerar um código de máquina mais especializado e eficiente.

Outra característica é o uso de macros. Eles permitem a manipulação do código antes de sua execução, permitindo gerar código repetitivo automaticamente, por exemplo. Julia também permite a chamada de funções de bibliotecas em C ou Python, dando acesso a todo o ecossistema dessas linguagens.

Uma característica diferencial desta linguagem é o despacho múltiplo. Refere-se ao mecanismo pelo qual a implementação específica (método) de uma função genérica é escolhida

[1] https://julialang.org/

em tempo de execução com base nos tipos de todos os seus argumentos, e não apenas do primeiro ou de uma única instância de objeto.

Apesar de ser uma linguagem relativamente jovem, Julia já foi adotada em diversas áreas da indústria e também no ensino devido ao seu alto desempenho e a existência de uma vasta gama de pacotes científicos disponíveis. Destaca-se o uso em *machine learning*, *deep learning*, programação probabilística, resolução de equações diferenciais e otimização matemática.

Na área acadêmica, Julia é cada vez mais adotada em cursos de graduação e pós-graduação em matemática, física, engenharia e ciência de dados, graças à sua sintaxe acessível e desempenho comparável a linguagens tradicionais de baixo nível (Language, 2025).

Julia pode ser usada em projetos de simulação de fenômenos físicos, análise de sistemas dinâmicos, processamento de sinais, bioinformática e modelagem estatística (Phillips, 2023). Projetos deste tipo, e muitos outros, podem ser acessados em julialang.org/learning/classes.

Outro ponto de destaque é o uso de Julia em supercomputadores. Projetos como o Celeste.jl, voltado para análise de dados astronômicos, demonstraram a capacidade da linguagem em atingir desempenho em escala petascale, algo antes restrito a códigos em C/Fortran. Foram analisados 178 terabytes de dados de imagens do espaço, catalogando 188 milhões de estrelas e galáxias em 14,6 minutos (JuliaHub, 2023). Isso consolidou Julia como uma ferramenta de confiança em contextos de computação de alto desempenho.

2.1.2 O Ambiente Pluto.jl

Para o desenvolvimento dos exemplos, este trabalho utiliza o pacote **Pluto.jl**. É um ambiente de *notebooks* projetado especificamente para Julia, que se diferencia dos ambientes tradicionais (como Jupyter e Google Colab) por ser reativo. Em um notebook reativo, a ordem de execução das células não é determinada pela ordem em que o usuário as executa manualmente, mas sim pela dependência entre as variáveis. Se uma variável x é alterada em uma célula, todas as outras células que dependem de x são atualizadas automaticamente, similarmente ao comportamento de uma planilha eletrônica.

O Pluto é um pacote padrão da linguagem Julia e pode ser instalado através do gerenciador de pacotes. A instalação pode ser feita pressionando a tecla `]` no REPL para entrar no modo de pacotes e em seguida executando o comando: `add Pluto`. Após instalado, basta voltar ao modo normal do REPL e iniciar o Pluto com os comandos abaixo:

O Julia iniciará um servidor local e abrirá automaticamente o navegador padrão do sistema

Listagem 1 – Inicialização do servidor Pluto.

```
import Pluto
Pluto.run()
```

na interface do Pluto, onde é possível criar novos notebooks ou abrir arquivos existentes. Todo o desenvolvimento dos métodos numéricos apresentados nos capítulos seguintes foi realizado utilizando esta ferramenta, aproveitando sua capacidade de integração com visualizações gráficas interativas.

2.1.3 Estruturas básicas

A linguagem Julia foi projetada para possuir sintaxe similar à outras linguagens de alto nível, porém com características únicas (Benzanson, 2015).

Variáveis

As variáveis são criadas por atribuição. Os nomes são *case-sensitive* (difere maiúsculas de minusculas) e a tipagem é dinâmica; ou seja, Julia atribui automaticamente o tipo de variável. Apesar dessa característica, é possível fazer anotações de tipo.

Listagem 2 – Sintaxe de declaração de variáveis em Julia

```
# A inferência de tipo entra em ação. 'numero' será do tipo Int64.
num = 42

# Variável de ponto flutuante.
pi_aproximado = 3.14

# Strings são delimitadas por aspas duplas.
frase = "Julia é uma linguagem rápida"

j = 1+2im       # Número complexo
j = 1+2im       # Número complexo
quarto = 1//4   # Número racional

# Anotações de tipo. Força 'idade' a ser um inteiro de 8 bits
idade::Int8 = 22

# Constantes são declaradas com 'const'.
const g = 9.81

x = 2 + 2       # Operações matemáticas.
```

Controle de Fluxo

Estruturas de controle permitem que partes do código sejam avaliadas ou não, dependendo do valor de uma expressão booleana. A sintaxe do `if-elseif-else` é similar a de outras linguagens, porém terminando com a palavra-chave `end`. Julia também possui um operador ternário para uma sintaxe mais limpa. Além disso, os operadores lógicos de conjunção e disjunção funcionam em curto-circuito, isto é, eles não necessariamente avaliam o segundo argumento.

Listagem 3 – Sintaxe de controle de fluxo em Julia

```
idade = 21
if idade >= 18
    println("Maior de idade")
elseif idade >= 12
    println("Adolescente")
else
    println("Criança")
end
# Operador ternário
idade >= 18 ? println("Maior de idade") : println("Menor de idade")
println(idade >= 18 ? "Maior de idade" : "Menor de idade")
```

Laços de repetição

Em Julia temos duas estruturas de repetição: o laço `for` e o laço `while`.

Listagem 4 – Laços em Julia

```
# Exemplo de laço com 'while'
i = 1
while i <= 3
    println(i)
    global i += 1
end

# Exemplo de laço com 'for'
for i = 1:3
    println(i)
end

# Exemplo de laço com 'for' usando sintaxe de conjunto
for i in [2, 4, 6]
    println(i)
end
```

Funções

As funções são objetos que mapeiam uma tupla de valores como argumento para retornar

um valor. As funções podem ser definidas de diferentes formas. Elas não são funções matemáticas puras, porque elas podem alterar e ser afetadas pelo estado global do programa. Julia também permite a declaração de funções anônimas muito usadas como argumentos para funções de ordem superior.

Listagem 5 – Sintaxe de declaração de função em Julia

```
# Sintaxe padrão
function soma(x, y)
    return x + y
end

# Sintaxe compacta, semelhante ao que vemos nos livros de matemática
soma(x, y) = x + y

# Exemplo de despacho múltiplo
processar(dado::Int) = println("Processando um inteiro: $dado")
processar(dado::String) = println("Processando uma string: $dado")

processar(10)        # Saída: Processando um inteiro: 10
processar("teste")   # Saída: Processando uma string: teste

# Anotações de tipo para funções
function produto(x, y)::Int8
    return x * y
end

# A função 'map' recebe uma função anônima como argumento
map(x -> x^2, [-1 0 2])  # Saída: 1 0 4
```

Julia também permite escrever funções com argumentos opcionais ou com um número arbitrário deles. Além disso, as funções podem ser combinadas por composição ou encadeamento (operador *pipe*) delas juntas. As listagens 5 e 6 mostram exemplos básicos de funções em Julia.

Listagem 6 – Composição de funções em Julia

```
# Composição de funções: soma 1 e extrai a raiz quadrada
(sqrt ∘ (x-> x+1))(3)           # Saída: 2.0
# Operador de broadcasting '.': a função é aplicada a cada elemento do vetor
(sqrt ∘ (x-> x+1)).([0 3 8])    # Saída:  1.0  2.0  3.0

# Operador pipe: passa o vetor para 'sum', o resultado é passado para 'sqrt'
[1 3 5] |> sum |> sqrt   # Saída: 3.0
```

Praticamente, todas as linguagens de programação mais usadas possuem controle de fluxo, laços, funções e variáveis como recursos básicas. No entanto, Julia traz recursos interessantes para as funções que não se vê comumente no estudo de programação.

Em relação às estruturas de dados, Julia não deixa a desejar. Ela possui estruturas fundamentais para o que ela se propõe a resolver/fazer.

2.1.4 Estruturas de dados fundamentais

Tuplas

As tuplas são coleções ordenadas e imutáveis de elementos. Uma vez criada, uma tupla não pode ser modificada.

Listagem 7 – Tuplas em Julia

```
# Criação de uma tupla. Pode conter tipos diferentes.
coordenadas = (12.8, -5.2, "Ponto A")

# Acesso aos elementos, a indexação começa em 1.
latitude = coordenadas[1]  # 12.8
```

Essa estrutura é eficiente e frequentemente usada para retornar múltiplos valores de uma função. Por exemplo, a declaração da função `f(x,y) = (x+y,x*y)` significa que `f` vai retornar a soma e o produto em forma de tupla.

As tuplas também podem ser nomeadas em relação aos seus elementos internos.

Listagem 8 – Nomeação de tuplas em Julia

```
# Criação de uma tupla nomeada
ponto = (x=2, y=1+2)   # Saída: (x=2, y=3)

# Acesso aos elementos
ponto.x   # Saída: 2
ponto.y   # Saída: 3

ponto.x = 0   # Erro: não podemos modificar uma tupla
```

Também podemos criar *structs*, caso a intenção seja criar uma estrutura com campos mutáveis. Embora imutável por padrão, a palavra-chave *mutable struct* a torna mutável.

Arrays

Os arrays (vetores e matrizes) são coleções ordenadas e mutáveis. No próximo trecho de código, declaramos arrays , acessamos os campos e modificamos também. É importante destacar que a indexação em Julia começa em 1 e isso a distancia da maioria das linguagens,

mas a aproxima da sintaxe natural da matemática.

Listagem 9 – Criação de Arrays

```
# Vetor (Array de 1 dimensão): usamos vírgula
inteiros_impares = [1, 3, 5, 7, 9]

# Acessar o elemento por meio do índice
primeiro_impar = inteiros_impares[1]

# Modificar o elemento de índice 3 (o valor 5 para 6)
inteiros_impares[3] = 6

# Matriz (Array de 2 dimensões): usamos espaço
matriz = [1 3; 5 7] # Cria uma matriz 2x2.

# Acessa o elemento na segunda linha, primeira coluna (valor 5)
elemento_2_1 = matriz[2, 1]
```

Dicionários

Há também os dicionários. Eles são coleções de pares chave-valor que são eficientes para buscar, inserir e deletar valores baseados em uma chave única, pois são implementados usando *hash*. A sintaxe `chave => valor` cria um objeto `Pair`.

Listagem 10 – Criação de Dicionários

```
# Criação de um dicionário. As chaves podem ser de tipos diferentes.
dados_aluno = Dict(
    "nome" => "José",
    "id" => 3301,
    "cursos" => ["Cálculo", "Circuitos", "Algoritmos"]
)

# Acessando um valor através de sua chave.
nome_aluno = dados_aluno["nome"] # Retorna "José"

# Adicionando um novo par chave-valor.
dados_aluno["semestre"] = 3

# Verificando se uma chave existe.
haskey(dados_aluno, "id") # Retorna true
```

Além das estruturas apresentadas, Julia também possui estruturas de dados no estilo de conjuntos e pilhas.

Os exemplos de estruturas básicas apresentados nesta seção foram baseados na documentação oficial da linguagem. Para aprofundamento, é altamente recomendada a leitura da documentação oficial do Julia (docs.julialang.org) e a utilização dos cursos interativos dispo-

níveis na JuliaAcademy (juliaacademy.com), que oferecem desde conteúdos introdutórios até materiais avançados sobre a linguagem e seu ecossistema.

2.1.5 Gráficos

Para a geração de gráficos, pode-se utilizar a biblioteca `Plots`. Para isso, após a instalação, deve-se incluí-la no cabeçalho.

O trecho de código a seguir irá plotar os gráficos das funções seno e cosseno no intervalo de $[0, 2\pi]$. Os gráficos aparecem em uma mesma janela devido a sintaxe com ponto de exclamação. O parâmetro `label` é opcional.

Listagem 11 – Gráfico 2D

```
using Plots

x = 0:0.1:2pi
plot(x, sin.(x), label="Seno")
plot!(x, cos.(x),label="Cosseno")
```

Basicamente é necessário passar o vetor de abscissas e o vetor de ordenadas do plano cartesiano, acrescidos de parâmetros opcionais.

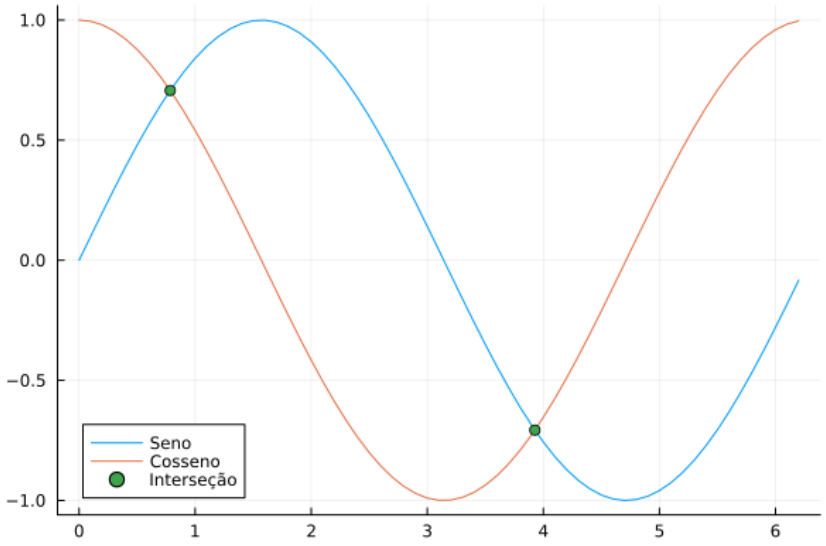
Quanto à plotagem de pontos discretos, como é o caso das soluções numéricas, usaremos a função `scatter`.

Listagem 12 – Plotagem de Pontos

```
using Plots

x = 0:0.1:2pi
plot(x, sin.(x), label="Seno")
plot!(x, cos.(x),label="Cosseno")

xs = [pi/4, 5pi/4]
scatter!(xs, sin.(xs), label="Interseção")

savefig("plot_seno_cosseno.png")
```

A função `savefig` irá salvar um arquivo de imagem do último gráfico renderizado na pasta de trabalho. No nosso caso, o resultado final será

Figura 1 – Gráfico das funções seno e cosseno de 0 até 2π



Fonte: Autoria própria (2025)

2.1.6 Mais sobre Julia

Em sua biblioteca padrão, Julia conta com, por exemplo, matrizes esparsas, *multi-threading*, computação distribuída, álgebra linear, estatística e interface para chamar funções em C. Ademais, outros pacotes podem ser adicionados como plotagem de gráficos, grafos e solução numérica de equações diferenciais.

Um recurso poderoso da linguagem é a metaprogramação que permite que o seu código escreva ou manipule outros códigos (ou a si mesmo) em tempo de execução.

2.2 Equações Diferenciais Ordinárias

As Equações Diferenciais (ED) são fundamentais na modelagem de fenômenos contínuos em ciência, engenharia e matemática aplicada. Modelar esses fenômenos significa obter uma equação que relaciona a taxa de variação de suas propriedades, ou seja, uma relação envolvendo derivadas.

Segundo Zill (2016), uma equação diferencial é uma equação que contém derivadas (ou diferenciais) de uma ou mais funções desconhecidas, chamadas de variáveis dependentes, em relação a uma ou mais variáveis independentes. Elas podem ser classificadas de acordo com o tipo, ordem e linearidade.

Quando uma equação diferencial envolve somente derivadas simples de uma ou mais

funções desconhecidas em relação a uma única variável independente, temos uma equação diferencial ordinária (EDO). Por outro lado, quando contém derivadas parciais de uma ou mais funções em relação a duas ou mais variáveis independentes, trata-se de uma equação diferencial parcial (EDP). Neste trabalho, as equações diferenciais abordadas são ordinárias e, majoritariamente, apresentam o tempo como variável independente. Um exemplo de EDO de primeira ordem é dado por:

$$\frac{dy}{dt} + 4y = e^t, \tag{2.1}$$

onde y é a função desconhecida (ou variável dependente) e t a variável independente.

Diversas notações podem ser empregadas para representar as derivadas. A notação usada na equação (2.1) é a notação de Leibniz e é uma das mais comuns. Outras possibilidades incluem a notação de linha y', y'', y''', $\ldots$, $y^{(n)}$ e a notação de Newton conhecida como "sujeira de mosca" $\dot{y}$, $\ddot{y}$, $\dddot{y}$, $\ddot{\dddot{y}}$. Assim, a equação (2.1) pode ser expressa nas notações alternativas da forma: $y' + 4y = e^t$ e $\dot{y} + 4y = e^t$, respectivamente.

2.2.1 Classificações

A classificação das EDOs pode ser feita seguindo diferentes critérios, como ordem, linearidade, homogeneidade etc.

Quanto à ordem

A ordem de uma equação diferencial ordinária (ou parcial) é determinada pela ordem da derivada de maior ordem da equação (Boyce; DiPrima, 2005). Por exemplo:

$$\frac{d^2y}{dt^2} + 2\frac{dy}{dt} + 4y = e^t \tag{2.2}$$

corresponde a uma equação diferencial de segunda ordem, enquanto a equação (2.1) é de primeira ordem.

Quanto à linearidade

A equação diferencial ordinária

$$a_n(t)\frac{d^n x(t)}{dt^n} + a_{n-1}(t)\frac{d^{n-1}x(t)}{dt^{n-1}} + \cdots + a_0(t)x(t) = F(t) \tag{2.3}$$

é linear se os termos $x(t)$, $dx(t)/dt$, $d^n x(t)/d^n t$ são escritos como combinação linear, ponderados pelos coeficientes $a_i (i = 0, 1, 2, 3, ..., n)$.

Já os sistemas não-lineares são aqueles que não podem ser escritos na forma de uma combinação linear. Dessa forma, os coeficientes a_0, a_1, ..., a_n só podem depender, no máximo, da variável independente t. Termos como *sen x* ou e^x não podem aparecer como função da variável dependente e nem de suas derivadas em uma equação linear (Monteiro, 2002; Chapra; Canale, 2015).

Quanto à homogeneidade

Equações diferenciais lineares podem ainda ser classificadas em homogêneas ou não homogêneas. Um sistema é dito homogêneo quando $F(t) = 0$ na Equação 1.3. Caso contrário, é não homogêneo.

2.2.2 Solução de EDOs

Nosso objetivo é encontrar uma função desconhecida a partir de uma relação entre suas derivadas. Segundo Boyce e DiPrima (2005), a solução de uma equação diferencial é uma função ϕ tal que suas derivadas existem e satisfazem a equação diferencial em um certo intervalo.

Uma vez definido o que constitui uma solução para uma EDO, é fundamental distinguir as duas formas pelas quais essa solução pode ser obtida: a via analítica e a via numérica. Uma solução analítica consiste em uma expressão matemática de forma fechada que satisfaz a EDO. Por exemplo, para uma EDO simples, a solução pode ser $y(t) = e^{-t} + C$. A presença da constante de integração C indica que uma única EDO pode gerar uma família infinita de soluções, cada uma correspondendo a um valor diferente para C.

Para isolar uma única curva de solução que represente um fenômeno físico específico, é necessário impor restrições adicionais. A mais comum é a especificação do estado do sistema em um momento inicial. Essa restrição é chamada de condição inicial. A combinação da equação diferencial com uma condição inicial define um Problema de Valor Inicial (PVI), que, para uma EDO de primeira ordem, é formalmente expresso como:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0 \tag{2.4}$$

Onde a equação diferencial descreve a dinâmica do sistema e (t_0, y_0) é a condição inicial

que especifica o valor da função solução $y(t)$ no instante inicial t_0.

A existência e a unicidade da solução de um PVI são garantidas sob determinadas condições, como a continuidade de $f(t,y)$ em um retângulo contendo (t_0,y_0) e a satisfação da condição de Lipschitz em relação a y (Boyce e DiPrima, 2006).

Apesar da grande importância, os métodos analíticos apresentam limitações e não se aplicam à maioria dos sistemas não lineares ou de alta complexidade. Parte considerável das EDOs que modelam sistemas reais apresentam coeficientes variáveis ou estão acopladas em sistemas complexos que não admitem soluções em forma fechada.

É neste ponto que a abordagem numérica se torna necessária. Em vez de buscar uma função exata para $y(t)$, o objetivo de um método numérico é gerar uma tabela de valores selecionados da variável independente e os valores correspondentes da variável dependente. Ao plotar os gráficos, é possível visualizar o comportamento aproximado da solução verdadeira em um intervalo pretendido (Boyce e DiPrima, 2006).

Na próxima seção será debatido os fundamentos dessa abordagem, bem como os principais métodos.

2.3 Métodos Numéricos

Como mencionado anteriormente, a principal razão para o emprego de métodos numéricos é que nem sempre é possível resolver PVIs de forma analítica ou é muito custoso.

O objetivo de todo método numérico é, a partir de um ponto conhecido (t_i,y_i), determinar o próximo ponto (t_{i+1},y_{i+1}) de forma eficiente e acurada. Isso é feito aplicando uma fórmula de iteração que se baseia na discretização da variável independente (normalmente o tempo), substituindo derivadas por diferenças finitas e gerando assim uma sequência de aproximações da solução exata.

Existem diversas maneiras de se construir essa fórmula, resultando em uma vasta família de métodos numéricos, cada um com suas próprias características de precisão, custo computacional e estabilidade.

Para fins didáticos e de fundamentação, a abordagem mais clara é a progressiva, partindo do método mais simples e intuitivo para os mais robustos e amplamente utilizados em aplicações práticas.

Para efeito comparativo, será utilizado como exemplo o PVI dado por

$$y' = t - 2y + 1, \quad y(0) = 1 \tag{2.5}$$

cuja solução analítica é $y(t) = \frac{1}{4}(2t + 3e^{-2t} + 1)$

2.3.1 Análise de erros

Antes de adentrar nos métodos propriamente ditos, é salutar discutir os erros gerados, pois os métodos numéricos inevitavelmente envolvem aproximações e, consequentemente, erros. A análise de erros é essencial para avaliar a precisão, a estabilidade e a consistência de um método numérico.

As deduções aqui apresentadas podem ser vistas com mais detalhes em Burden & Faires (2016) e em Chapra & Canale (2015).

Erro local de truncamento

O erro local de truncamento está associado à aproximação feita em um único passo do método, assumindo que o valor inicial é exato.

Erro global

O erro global resulta da acumulação dos erros locais em todos os passos do processo numérico. Diferentemente do erro local, que considera uma única iteração, o erro global mede a diferença entre a solução exata e a solução numérica após várias iterações sucessivas.

2.3.2 Método de Euler

A ideia central do método consiste em utilizar a inclinação da solução no ponto inicial (x_0, y_0) para estimar o valor da função no próximo ponto. Como a derivada da solução no ponto inicial é conhecida ($y'(x_0) = f(x_0, y_0)$), é possível aproximar a curva solução por uma reta tangente que passa pelo ponto inicial (Ruggiero; Lopes, 1996).

A equação da reta tangente é dada por:

$$r_0(x) = y(x_0) + (x - x_0)y'(x_0) \tag{2.6}$$

Escolhendo um passo $h = x_{i+1} - x_i$, temos:

$$y_1 = r_0(x_1) \approx y_0 + hf(x_0, y_0) \tag{2.7}$$

Assim, partindo de (x_0, y_0), obtêm-se o próximo ponto aproximado da solução (x_1, y_1). Repetindo o mesmo raciocínio para (x_2, y_2) e assim sucessivamente, chega-se à fórmula iterativa do método

de Euler:

$$y_{i+1} = y_i + hf(x_i, y_i) \tag{2.8}$$

onde $i = 0, 1, 2, 3, \ldots$.

Iremos resolver o PVI e comparar com a solução analítica. Os dados do nosso problema fornecem $x_0 = 0$ e $y_0 = 1$ (valor inicial no PVI). O termo h é escolhido livremente e determina quão acurada será a nossa solução.

Construiremos três tabelas no intervalo $[0, 2]$ usando passos $h = 0,5$, $h = 0,25$ e $h = 0,1$. O erro será dado por

$$erro = \frac{|y_i - y(ti)|}{y(ti)}$$

Tabela 1 – Comparação entre Euler (h = 0,5) e a solução analítica.

t_i	y_i (Euler)	$y(t_i)$ (Analítica)	Erro %
0,00	1,000000	1,000000	0,00%
0,50	0,500000	0,775900	35,56%
1,00	0,750000	0,851501	11,92%
1,50	1,000000	1,037340	3,60%
2,00	1,250000	1,263737	1,09%

Fonte: Autoria própria (2025)

Essa abordagem fornece uma sequência de aproximações da solução exata no intervalo considerado, com espaçamento h. Quanto mais pontos subdividimos o intervalo (h menor), melhor fica nossa solução. As Tabelas 2 e 3 foram produzidas usando o método de Euler com $h = 0,25$ e $h = 0,1$, respectivamente.

Tabela 2 – Comparação entre Euler (h = 0,25) e a solução analítica.

t_i	y_i (Euler)	$y(t_i)$ (Analítica)	Erro %
0,00	1,000000	1,000000	0,00%
0,50	0,687500	0,775900	11,39%
1,00	0,796875	0,851501	6,42%
1,50	1,011719	1,037340	2,47%
2,00	1,252930	1,263737	0,85%

Fonte: Autoria própria (2025)

Nota-se que o erro foi reduzido pela metade quando o número de passos foi dobrado (de 4 para 8).

Tabela 3 – Comparação entre Euler (h = 0,1) e a solução analítica.

t_i	y_i (Euler)	$y(t_i)$ (Analítica)	Erro %
0,00	1,000000	1,000000	0,00%
0,50	0,745760	0,775900	3,89%
1,00	0,830531	0,851501	2,46%
1,50	1,002688	1,037340	1,10%
2,00	1,258647	1,263737	0,40%

Fonte: Autoria própria (2025)

O erro local do método de Euler é da ordem de $O(h^2)$, indicando que ele tende a zero quadraticamente à medida que o passo h diminui. Já o erro global é proporcional a $O(h)$, o que classifica Euler como um método de primeira ordem. Isso significa que, para reduzir o erro pela metade, é necessário dobrar o número de passos (reduzir h pela metade), tornando-o computacionalmente muito custoso para se obter alta precisão.

O método é particularmente útil para obter uma primeira aproximação ou para sistemas relativamente simples, em que o comportamento da solução não exige métodos mais robustos.

A Figura 2 mostra mais claramente a influência do passo h por meio de três gráficos.

2.3.3 Método de Heun

A principal limitação do método de Euler é sua baixa acurácia, uma consequência direta de ser um método de primeira ordem, cujo erro global é proporcional ao tamanho do passo ($O(h)$). Essa imprecisão origina-se da suposição de que a inclinação no início do intervalo, $f(x_i, y_i)$, é uma representação adequada da inclinação média ao longo de todo o intervalo $[x_i, x_{i+1}]$. Quando a solução real não é linear, essa premissa falha e introduz um erro de truncamento considerável a cada passo.

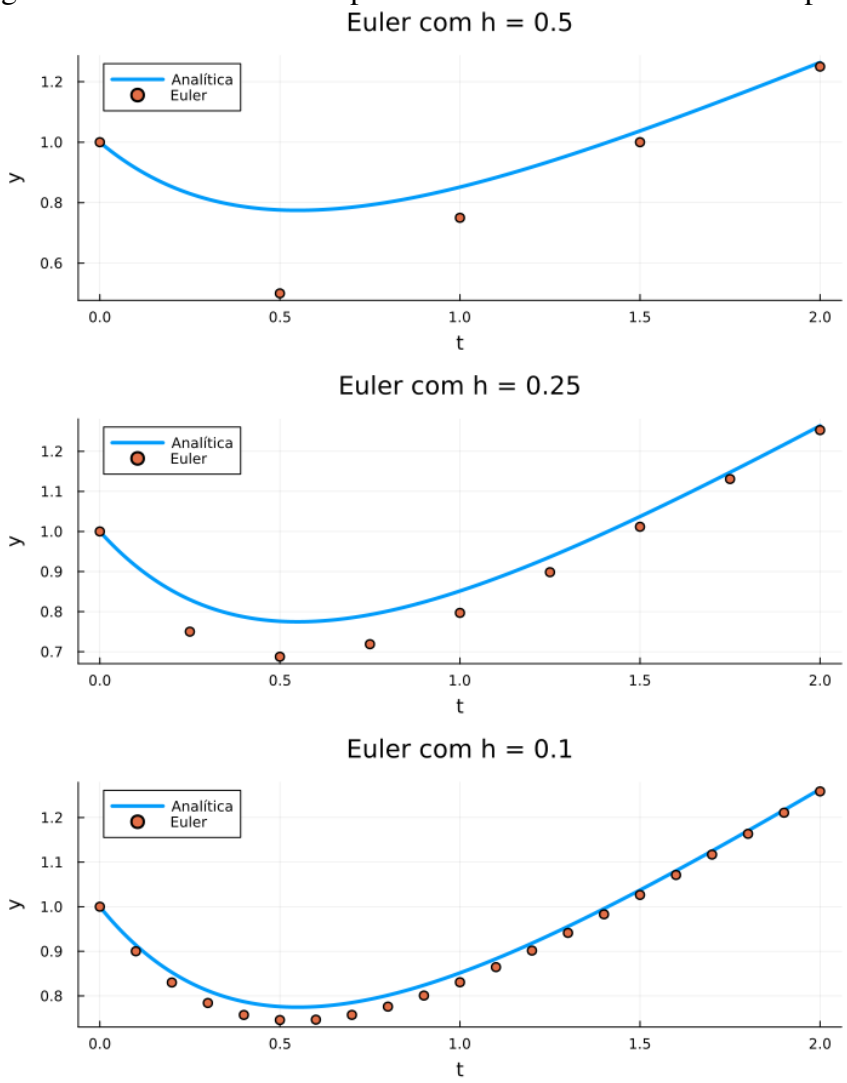
Uma abordagem para contornar esse problema é melhorar a estimativa de inclinação, utilizando a média aritmética entre a inclinação inicial e uma estimativa da inclinação no final do intervalo. Essa lógica é a base do Método de Euler Aprimorado, também conhecido como Método de Heun.

Primeiro, o método "prediz" um valor provisório para y_{i+1} usando o Método de Euler padrão. Este valor é uma estimativa inicial, que denota-se por y^*_{i+1}:

$$y^*_{i+1} = y_i + h \cdot f(t_i, y_i) \tag{2.9}$$

Esta equação é chamada de *equação preditora*, e serve para estimar a inclinação da solução no

Figura 2 – Método de Euler aplicado com diferentes números de passos



Fonte: Autoria própria (2025)

final do intervalo, t_{i+1}:

$$y'_{i+1} = f(t_{i+1}, y^*_{i+1}) \tag{2.10}$$

Combinando as duas inclinações obtém-se a *equação corretora*

$$y_{i+1} = y_i + \frac{h}{2} \cdot (K_1 + K_2) \tag{2.11}$$

onde $K_1 = f(t_i, y_i)$ e $K_2 = f(t_{i+1}, y^*_{i+1})$

Enquanto o Método de Euler pode ser interpretado como a aproximação da integral da EDO pela Regra do Retângulo (usando a altura $f(x_i, y_i)$), o Método de Heun é análogo à Regra do Trapézio para integração numérica. Ao usar a média das inclinações nas extremidades do intervalo, ele aproxima a área sob a curva $f(x,y)$ com um trapézio, o que é significativamente mais preciso (Burden; Faires, 2011).

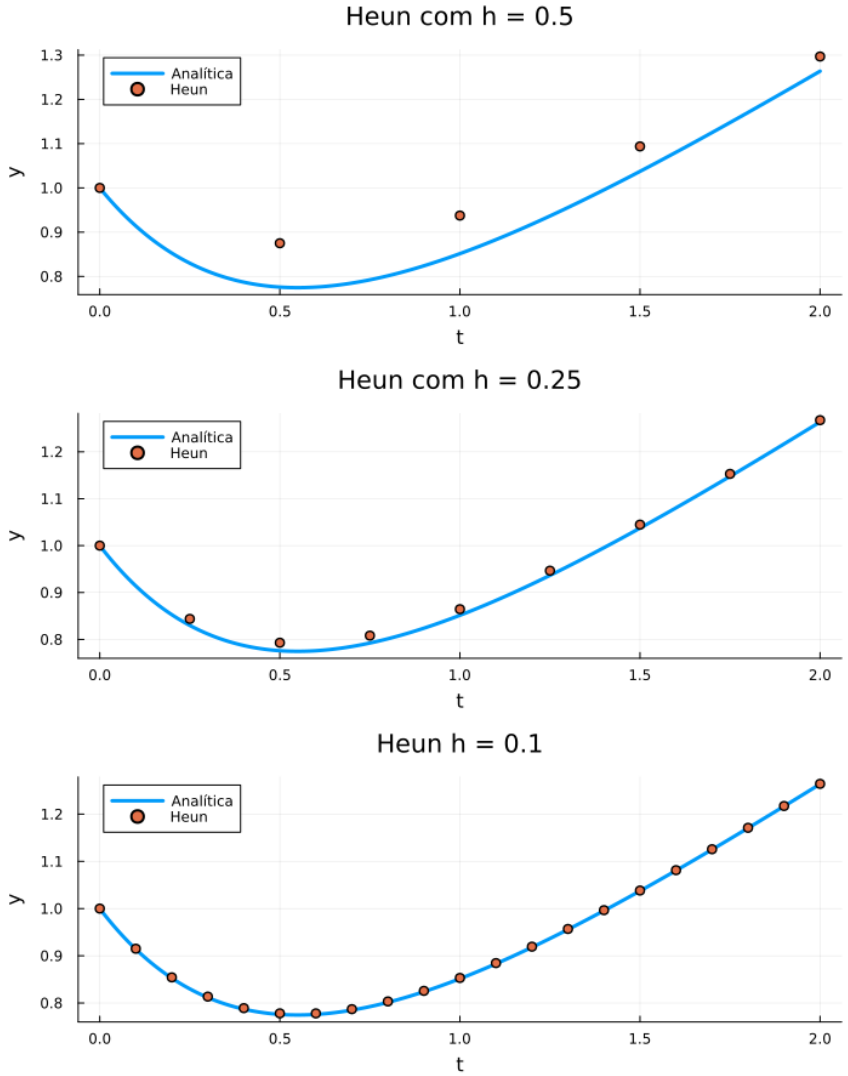
Para efeito comparativo, vamos resolver PVI da seção anterior com os mesmos parâmetros (Tabela 4). A velocidade de convergência pode ser analisada na Figura 3.

Tabela 4 – Comparação entre Heun e a solução analítica.

t_i	$h = 0,5$		$h = 0,25$		$h = 0,1$		$y(t_i)$ (Analítica)
0,00	1,000000	0,00%	1,000000	0,00%	1,000000	0,00%	1,000000
0,50	0,875000	12,77%	0,792969	2,20%	0,778055	0,28%	0,775900
1,00	0,937500	10,10%	0,864441	1,52%	0,853086	0,19%	0,851501
1,50	1,093750	5,44%	1,044703	0,71%	1,038218	0,08%	1,037340
2,00	1,296875	2,62%	1,267462	0,29%	1,264169	0,03%	1,263737

Fonte: Autoria própria (2025)

Figura 3 – Método de Heun aplicado com diferentes números de passos



Fonte: Autoria própria (2025)

A consequência desta melhoria é um aumento na acurácia do método. O erro de truncamento local do método é da ordem de $O(h^3)$. Isso implica que o erro global acumulado é da

ordem de $O(h^2)$, tornando o Método de Heun um método de segunda ordem. Na prática, isso significa que se o passo h for reduzido pela metade, o erro global é reduzido por um fator de (2^2) e não pela metade como no Euler padrão.

2.3.4 Métodos de Runge-Kutta

A busca por métodos de ordem superior, que convergem mais rapidamente para a solução real, levou ao desenvolvimento da família de métodos de Runge-Kutta (RK). A ideia central, desenvolvida pelos matemáticos alemães Carl Runge (1895) e Martin Kutta (1901), é que em vez de usar a inclinação $f(t,y)$ apenas no ponto inicial para projetar a solução, pode-se obter uma estimativa muito mais precisa da "inclinação média" ao longo do intervalo através de múltiplas avaliações da função f em pontos intermediários.

Esses métodos alcançam a precisão de uma abordagem por Série de Taylor sem a necessidade explícita de se calcular as derivadas de $y(t)$ $(y',y'',y''',\dots)$ o que, em geral, é um processo complicado e demorado (Burden; Faires, 2011).

O método de Runge-Kutta (RK) é uma família de métodos com muitas variações, porém todas podem ser postas da seguinte maneira:

$$y_{i+1} = y_i + \phi h, \tag{2.12}$$

onde ϕ é chamada de função de incremento e h é o passo. A função ϕ pode ser interpretada como a inclinação do intervalo, sendo escrita da seguinte forma:

$$\phi = a_1 k_1 + a_2 k_2 + \cdots + a_n k_n, \tag{2.13}$$

onde os a's são constantes e os k's são

$$k_1 = f(t_i, y_i) \tag{2.16a}$$

$$k_2 = f(t_i + p_1 h, y_i + q_{11} k_1 h) \tag{2.16b}$$

$$k_3 = f(t_i + p_2 h, y_i + q_{21} k_1 h + q_{22} k_2 h) \tag{2.16c}$$

$$\vdots$$

$$k_n = f(t_i + p_{n-1} h, y_i + q_{n-1,1} k_1 h + q_{n-1,2} k_2 h + \cdots + q_{n-1,n-1} k_{n-1} h) \tag{2.16d}$$

em que os p's e os q's são constantes. Observando as equações, é possível notar que k_1 aparece em k_2 que, por sua vez, aparece em k_3. Isso caracteriza uma relação de ocorrência à qual torna os métodos RK computacionalmente eficientes (Chapra; Canale, 2015).

A depender do número de termos da função incremento (especificado por n), é possível deduzir vários tipos de métodos de Runge-Kutta.

2.3.4.1 Rungge-Kutta de primeira ordem

Se definirmos $n = 1$, temos a partir das Equações 2.12 e 2.13:

$$y_{i+1} = y_i + a_1 f(t_i, y_i)h \tag{2.14}$$

Nota-se, portanto, que o método de Runge-Kutta de primeira ordem com $n = 1$ é o método de Euler. Os valores para as constantes a, p e q são calculados igualando a Equação 2.14 à expansão em série de Taylor truncada na ordem correspondente. Dessa forma, para os tipos de ordem mais baixa, o número de termos n representa a ordem do método (Chapra; Canale, 2015).

2.3.4.2 Rungge-Kutta de segunda ordem

Definindo $n = 2$, pode-se escrever uma versão de segunda ordem da Equação 2.12:

$$y_{i+1} = y_i + h[a_1 f(t_i, y_i) + a_2 f(t_i + p_1 h, y_i + q_{11} k_1 h)] \tag{2.15}$$

Igualando esta expressão à expansão em série de Taylor até os termos de ordem 2, obtêm-se um sistema de equações para determinar o valor das constantes a_1, a_2, p_1 e q_{11}:

$$\begin{cases} a_1 + a_2 = 1 \\ a_2 p_1 = \frac{1}{2} \\ a_2 q_{11} = \frac{1}{2}. \end{cases} \tag{2.16}$$

A dedução desse sistema pode ser visto com mais detalhes em Ruggiero e Lopes (1996).

Nota-se que se trata de um sistema de três equações e quatro incógnitas, então deve-se escolher o valor para uma das incógnitas e assim determinar as outras três. Escolhendo $a_2 = 0,5$. por exemplo, $a_1 = 0,5$ e $p_1 = q_{11} = 1$; Realizando as substituições:

$$y_{i+1} = y_i + h\left(\frac{1}{2}k_1 + \frac{1}{2}k_2\right) \tag{2.17}$$

onde

$$\begin{aligned} k_1 &= f(t_i, y_i) \\ k_2 &= f(t_i + h,\ y_i + k_1 h) \end{aligned} \tag{2.18}$$

Observa-se que k_1 neste caso, é a inclinação no início do intervalo e k_2 é a inclinação no fim do intervalo. Portanto, tem-se que o método RK de segunda ordem é na realidade o Método de Heun, onde k_1 atua como o "preditor", e k_2 atua como o "corretor", usando a média das inclinações (Chapra; Canale, 2015).

A medida que aumentamos a ordem, a solução se torna mais precisa, porém o custo computacional cresce. Por essa razão, o método RK de terceira ordem não é popularizado por não apresentar um bom custo-benefício. O método padrão que traz o melhor dos dois mundos é o RK de quarta ordem apresentado a seguir.

2.3.4.3 Rungge-Kutta de quarta ordem

Embora os métodos RK2 sejam superiores ao de Euler, o método mais amplamente utilizado é o Método de Runge-Kutta de ordem 4 (RK4). Este método alcança uma precisão de quarta ordem (erro global $O(h^4)$) utilizando quatro avaliações da função f por passo, oferecendo equilíbrio entre acurácia e custo computacional.

O algoritmo do RK4 é consideravelmente mais preciso porque sua derivação corresponde à expansão da Série de Taylor até o termo de ordem h^4. Assim como nos métodos analisados anteriormente, existem inúmeras versões para o RK4. No entanto, existe uma forma mais utilizada que é chamada de método RK4 clássico:

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h \tag{2.19}$$

onde

$$\begin{aligned} k_1 &= f(t_i, y_i) \\ k_2 &= f\left(t_i + \frac{1}{2}h,\ y_i + \frac{1}{2}k_1 h\right) \\ k_3 &= f\left(t_i + \frac{1}{2}h,\ y_i + \frac{1}{2}k_2 h\right) \\ k_4 &= f(t_i + h,\ y_i + k_3 h) \end{aligned} \tag{2.20}$$

Pode-se interpretar os k's da seguinte forma:

- k_1: É a inclinação no início do intervalo (a mesma do Método de Euler).
- k_2: É uma estimativa da inclinação no ponto médio do intervalo.
- k_3: É uma segunda estimativa da inclinação no ponto médio.
- k_4: É uma estimativa da inclinação no final do intervalo.

A fórmula final (Equação 2.19) é uma média ponderada dessas inclinações. Percebe-se que as inclinações do ponto médio recebem o dobro do peso, de forma análoga à Regra 1/3 de Simpson para integração numérica (Chapra; Canale, 2015).

A superioridade do RK4 é notável; se o passo h for reduzido pela metade, o erro global é reduzido por um fator de 16. Isso permite que o RK4 alcance alta precisão com um passo h maior, tornando-o o método de passo único de escolha para boa parte dos problemas.

Isso pode ser exemplificado resolvendo o PVI da Equação 2.5 para $h = 0,5$

Tabela 5 – Comparação entre RK4 (h = 0,5) e a solução analítica.

t_i	y_i (RK4)	$y(t_i)$ (Analítica)	Erro %
0,00	1,000000	1,000000	0,00%
0,50	0,781250	0,775900	0,69%
1,00	0,855469	0,851501	0,47%
1,50	1,039551	1,037340	0,21%
2,00	1,264832	1,263737	0,09%

Fonte: Autoria própria (2025)

2.3.5 Sistemas Dinâmicos de Segunda Ordem

Até este ponto, os métodos numéricos foram aplicados a uma EDO de primeira ordem. No entanto, a grande maioria dos problemas físicos de interesse na engenharia - como vibrações mecânicas, circuitos RLC e dinâmica de fluidos - é modelada por EDOs de segunda ordem ou superior (Boyce; DiPrima, 2005).

2.3.5.1 Redução de Ordem

Os métodos discutidos até aqui são formulados para resolver equações de primeira ordem; portanto, é necessário transformar a EDO de segunda ordem em um sistema de equações de primeira ordem. Considere a equação geral:

$$\ddot{y} = f(t, y, \dot{y}), \tag{2.21}$$

onde as condições iniciais são $y(0) = a_1$, $\dot{y}(0) = a_2$.

Define-se o vetor de estado como:

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} y \\ \dot{y} \end{bmatrix}. \tag{2.22}$$

(Boyce; DiPrima, 2005)

Realizando a derivada de u, obtém-se o sistema equivalente de primeira ordem:

$$\begin{cases} \dot{u}_1 = u_2, \\ \dot{u}_2 = f(t, u_1, u_2). \end{cases} \tag{2.23}$$

Ou, de forma compacta,

$$\dot{u} = \begin{bmatrix} u_2 \\ f(t, u_1, u_2) \end{bmatrix}. \tag{2.24}$$

Com o vetor de condição inicial $u(0) = [a_1, a_2]$ (Burden; Faires, 2011).

No próximo capítulo, iremos abordar equações diferenciais, com diferentes ordens, de problemas reais. Dessa maneira, podemos analisar o custo-benefício de cada método.

3 ESTUDOS DE CASO

Foram selecionados dois problemas envolvendo equações diferenciais. O objetivo não é mostrar como obter essas equações, apenas resolvê-las usando os métodos numéricos abordados na seção anterior.

3.1 Decaimento do carbono-14

O decaimento radioativo é um processo de primeira ordem no qual a taxa de decaimento de uma substância ($\frac{dN}{dt}$) é diretamente proporcional à quantidade de substância presente (N) (Zill, 2016). Isso é descrito pelo PVI:

$$\frac{dN}{dt} = -\lambda N, \quad N(t_0) = N_0 \tag{3.1}$$

onde λ é a constante de decaimento, que pode ser calculada usando o tempo de meia-vida ($t_{1/2}$) da substância através da relação

$$\lambda = \frac{\ln 2}{t_{1/2}}. \tag{3.2}$$

A solução analítica exata para o PVI (Equação 3.1) é,

$$N(t) = N_0\, e^{-\lambda t} \tag{3.3}$$

Para o Carbono-14, sabe-se que o tempo de meia-vida é de aproximadamente 5730 anos (Zill, 2016). Portanto, $\lambda \approx 1,2097 \times 10^{-4}$ anos^{-1}.

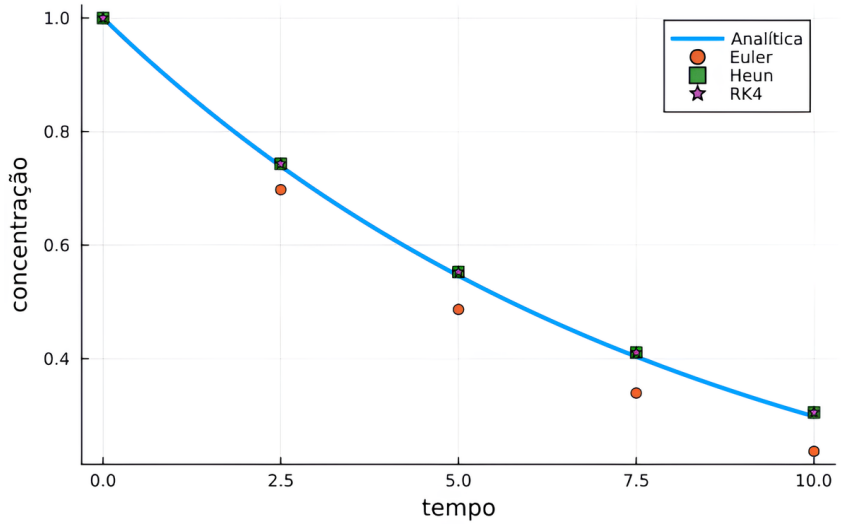
Considerando $N_0 = 1$ e um período de 10000 anos subdivididos em 4 intervalos, a Figura 4 mostra a solução do nosso PVI usando os métodos de Euler, Heun e RK4. Nota-se a superioridade do método de Heun sobre o de Euler. Para uma análise mais acurada entre Heun e RK4, observe a Tabela 6.

Tabela 6 – Comparação entre os erros relativos

t_i	Euler	Heun	RK4
0,0	0,00%	0,00%	0,00%
2,50	5,608%	0,579%	0,003%
5,0	10,902%	1,162%	0,005%
7,5	15,899%	1,748%	0,008%
10,0	20,616%	2,337%	0,011%

Fonte: Autoria própria (2025)

Figura 4 – Decaimento do carbono-14 usando métodos numéricos



Fonte: Autoria própria (2025)

3.2 Pêndulo simples

Até este ponto, os métodos numéricos foram aplicados a uma EDO de primeira ordem. No entanto, a grande maioria dos problemas físicos de interesse na engenharia - como vibrações mecânicas, circuitos RLC e dinâmica de fluidos - é modelada por EDOs de segunda ordem ou superior (Boyce; DiPrima, 2005).

O pêndulo simples é um sistema clássico da mecânica composto por uma massa puntiforme presa a uma haste ideal (inextensível e de massa desprezível), que oscila sob a ação da gravidade.

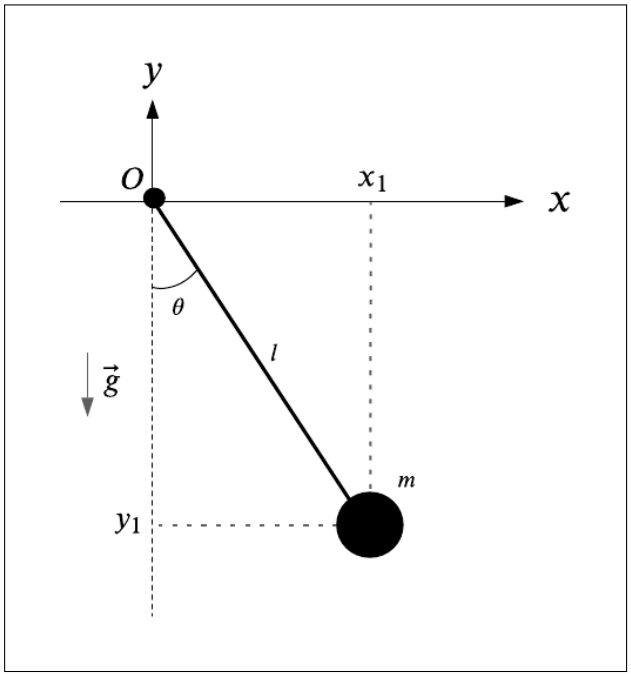
O movimento de um pêndulo simples é considerado periódico, e o tempo necessário para completar uma oscilação completa é chamado de período (T). O modelo é amplamente utilizado em livros didáticos para a introdução e compreensão de diversos conceitos da física, como o Movimento Harmônico Simples (MHS) e força peso, e também foi historicamente usado na determinação experimental da aceleração da gravidade (Moreira *et al.*, 2019).

A equação que descreve o comportamento do pêndulo simples é:

$$\ddot{\theta} + \frac{g}{l}\sin\theta = 0, \tag{3.4}$$

onde g é a aceleração da gravidade, l é o comprimento da haste e θ é o ângulo que a haste faz com a vertical (Figura 5). Nota-se que se trata de uma equação não-linear; portanto, não admite solução em termos de funções elementares. Uma abordagem comum para contornar esse problema é considerar a aproximação para pequenos ângulos ($\sin\theta \approx \theta$), o que torna a expressão

Figura 5 – Configuração do sistema Pêndulo Simples.



Fonte: Autoria própria (2025).

uma equação diferencial linear de segunda ordem que descreve o movimento harmônico simples:

$$\ddot{\theta} + \frac{g}{l}\theta = 0 \tag{3.5}$$

A solução é dada por

$$\theta(t) = a\cos\left(\sqrt{\frac{g}{l}}.t\right), \tag{3.6}$$

onde a é a amplitude angular inicial (Baker; Blackburn, 2005).

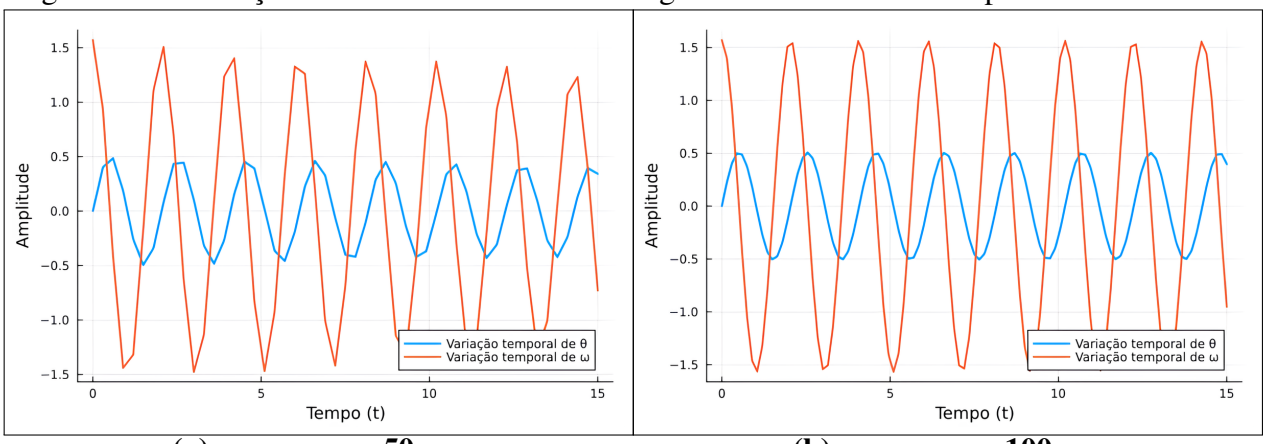
Agora que os métodos numéricos já foram apresentados, não é mais necessário realizar aproximações lineares. Isso possibilita que o comportamento do sistema seja analisado por completo.

Por se tratar de uma EDO de segunda ordem, é necessário transformar o problema em um sistema de EDOs de primeira ordem. Para isso, será utilizada a notação apresentada na Seção 2.3.5. Assumindo que $y_1 = \theta$ e $y_2 = \dot{\theta}$ tem-se:

$$\begin{cases} \dot{y_1} = y_2 \\ \dot{y_2} = -\frac{g}{l}\sin(\theta). \end{cases} \tag{3.7}$$

A Figura 6 apresenta duas simulações realizadas para esse sistema utilizando o método Runge-Kutta. Ambas mostram a evolução temporal do ângulo e da velocidade angular, porém com diferentes números de passos. Observa-se que, ao utilizar apenas n = 50 passos, a curva

Figura 6 – Simulação utilizando o método de Runge-Kutta com número de passos diferentes



(a) `n_passos` **= 50** **(b)** `n_passos` **= 100**

Fonte: Autoria própria (2025)

apresenta irregularidades perceptíveis, especialmente nas regiões de maior variação angular. Isso ocorre porque, com um largura de passo maior, o método não consegue acompanhar com precisão a curvatura da trajetória real, produzindo uma aproximação menos suave. Quando o número de passos é aumentado para n = 100, a solução torna-se significativamente mais suave e visualmente mais próxima do comportamento esperado.

Além disso, percebe-se que o método RK4 não conserva a energia mecânica do sistema, mesmo quando o número de passos é aumentado. Isso se manifesta como um leve aumento ou diminuição da amplitude ao longo do tempo. Métodos específicos, como integradores simpléticos, preservam melhor a energia em sistemas oscilatórios (SCIML, 2025).

4 IMPLEMENTAÇÃO EM JULIA

Os códigos em Julia partem de exemplos para a implementação de funções. Como se trata de equações iterativas, é necessário usar um laço para calcular ponto a ponto os valores da função. O laço `for` foi escolhido e o PVI da Equação 2.5 foi utilizado como exemplo.

4.1 Método de Euler

A implementação inicia-se com um problema em que se sabe o passo h.

Listagem 13 – Implementação Método de Euler

```
1  f(t, y) = t - 2y + 1       # Função f(t,y)
2
3  x0, y0 = 0, 1              # Condição inicial
4
5  h = 0.5                    # Tamanho do passo
6
7  # Vetores que armazenarão o resultado
8  ts = x0:h:2           # tempo variando de 0 a 2
9  ys = similar(ts)      # vetor do mesmo tipo e tamanho de ts
10 len = length(ts)      # tamanho dos vetores
11 ys[1] = y0            # setando a coordenada inicial
12
13 # Euler
14 for i in 1:len-1
15 ys[i+1] = ys[i] + h*f(ts[i], ys[i])
16 end
17
18 ts, ys  # Saída: (0.0:0.5:2.0, [1.0, 0.5, 0.75, 1.0, 1.25])
```

Embora os comentários no trecho de código da Listagem 13 sejam autoexplicativos, vale destacar que a notação `i:p:j` cria um vetor partindo de `i` até `j` com intervalos igualmente espaçados por `p`. Caso o `p` fosse omitido, Julia iria considerar um salto de uma unidade. O tamanho do vetor é necessário para saber quantos pontos iremos usar.

Não existe uma maneira única de implementar um algoritmo, a não ser que ele seja trivial. Neste contexto, nem sempre é interessante passar como parâmetro o valor de h. Muitas vezes, deseja-se calcular passando uma quantidade de pontos que dividirá o intervalo em subintervalos iguais, e o h seria calculado pelo algoritmo.

A Listagem 14 mostra uma segunda alternativa. Destaca-se neste código a linha 17. Como `n` vale 4, isso significa que estamos gerando um vetor da seguinte forma: `[a + 0*h, a + 1*h, a + 2*h, a + 3*h, a + 4*h]`. Ou seja, `ts` vai de 0 até 2. Esse laço dentro do vetor é

bastante útil e não é visto na maioria das linguagens de programação.

Listagem 14 – Implementação Método de Euler

```
f(t, y) = t - 2y + 1      # Função f(t,y)

x0, y0 = 0, 1             # Condição inicial

a, b = x0, 2              # Intervalo

n = 4                     # Número de subintervalos

h = (b - a)/n             # Calculo de h

# Vetores que armazenarão o resultado
ts = [a + i*h for i in 0:n]    # tempo variando de 0 a 2
ys = similar(ts)               # vetor do mesmo tipo e tamanho de ts
ys[1] = y0                     # setando a coordenada inicial

# Euler
for i in 1:n
ys[i+1] = ys[i] + h*f(ts[i], ys[i])
end

ts, ys   # Saída: ([0.0, 0.5, 1.0, 1.5, 2.0], [1.0, 0.5, 0.75, 1.0, 1.25])
```

Para melhorar a manutenção e reutilização do código, deve-se escrever funções. A função deve retornar uma tupla de vetores contendo os valores $(t, y(t))$. Tomando a segunda abordagem como inspiração (Listagem 14), a função deve receber como parâmetros a função do PVI, as condições iniciais, o intervalo e o número de subintervalos.

Listagem 15 – Implementação Método de Euler com função

```
function euler(f, y0, a, b, n)
h = (b - a)/n
ts = [a + i*h for i in 0:n]
ys = similar(ts)
ys[1] = y0

for i in 1:n
      ys[i+1] = ys[i] + h*f(ts[i],ys[i])
end

return ts, ys
end
```

Praticamente, o código na Listagem 14 foi encapsulado em uma função chamada `euler`. Um exemplo de uso das funções segue abaixo:

Listagem 16 – Exemplo de uso Método de Euler

```
f(t, y) = t - 2y + 1   # função considerada no PVI

ts, ys = euler(f, 1.0, 0.0, 2.0, 4)
# Saída: ([0.0, 0.5, 1.0, 1.5, 2.0], [1.0, 0.5, 0.75, 1.0, 1.25])
```

4.2 Método de Heun

Tomando como base a Listagem 14, pode-se obter a solução com Heun facilmente. É somente necessário modificar os cálculos dentro do laço para refletir as equações preditora e corretora.

Listagem 17 – Implementação Método de Heun

```
# Heun
for i in 1:n
    k1 = f(ts[i], ys[i])
    k2 = f(ts[i+1], ys[i] + h*k1)
    ys[i+1] = ys[i] + h*(k1 + k2)/2
end

ts, ys  # Saída: (0.0:0.5:2.0, [1.0, 0.875, 0.9375, 1.09375, 1.296875])
```

De forma análoga ao que foi feito com o método de Euler, o Método de Heun pode ser implementado como uma função. A estrutura do código reflete diretamente as etapas de "predição" e "correção" do algoritmo.

Listagem 18 – Implementação Método de Heun com função

```
function heun(f, y0, a, b, n)
    h = (b-a)/n                 # cálculo do passo h
    ts = [a + i*h for i in 0:n] # discretização da variável independente
    ys = similar(ts)            # vetor solução do tamanho de ts
    ys[1] = y0                  # setando a coordenada inicial

    for i in 1:n
        t = ts[i]
        k1 = f(t, ys[i])
        k2 = f(t + h, ys[i] + h*k1)
        ys[i+1] = ys[i] + h*(k1 + k2)/2
    end
    return ts, ys
end
```

É possível notar que a implementação do Método de Heun é mais complexa, pois o laço `for` agora exige duas avaliações da função f (linhas 9 e 10) para cada iteração. O algoritmo se torna computacionalmente mais custoso, dobrando o número de cálculos da EDO. No entanto, os resultados obtidos demonstram um ganho considerável de precisão obtido com essa complexidade adicional.

4.3 Método de Runge-Kutta 4

Para obter o método, mais uma vez, basta modificar apenas a parte interna do laço. Desta vez, a implementação será feita diretamente em uma função.

Listagem 19 – Implementação Método de Runge-Kutta 4

```
1  function rk4(f, y0, a, b, n)
2      h = (b-a)/n
3      ts = [a + i*h for i in 0:n]
4      ys = similar(ts)
5      ys[1] = y0
6
7      for i in 1:n
8          t = ts[i]
9          k1 = f(t, ys[i])
10         k2 = f(t+h/2, ys[i] + h*k1/2)
11         k3 = f(t+h/2, ys[i] + h*k2/2)
12         k4 = f(t+h, ys[i] + h*k3)
13         ys[i+1] = ys[i] + h*(k1 + 2*k2+2*k3+k4)/6
14     end
15     return ts, ys
16 end
```

Note que, por simplicidade, os erros dos métodos não estão sendo tratados em Julia.

4.4 Sistemas de Segunda Ordem

As funções para resolver EDOs possuem como parâmetro o valor da função y no ponto inicial (`y0`). Todavia, para um sistema de segunda ordem deve-se trabalhar com vetores (veja a Seção 2.3.5.1). Logo, as funções devem ser ligeiramente modificadas; definindo o vetor `ys = similar(ts)`. Agora, `ys` deve ser um *array* de vetores:

Listagem 20 – Definição de um *array* de vetores

```
1  ys = Vector{Vector{Float64}}(undef, n+1)
```

Onde `undef` é uma constante que indica a Julia que você quer alocar o espaço de memória para o array, mas não quer preencher o espaço com valores iniciais e `n+1` é o tamanho.

Outra modificação que pode ser realizada é usar a função `push`, que insere um ou mais itens no final de uma coleção. Dessa maneira, evita-se a preocupação com os tamanhos.

Listagem 21 – Implementação Método de RK4 usando `push`

```
function rk4(f, y0, a, b, n)
    h = (b-a)/n
    t = a      # tempo inicial
    y = y0     # valor inicial
    ts = [t]   # vetor com a marcação do tempo
    ys = [y0] # vetor com os resultados

    for i in 1:n
        k1 = f(t, ys[i])
        k2 = f(t+h/2, ys[i] + h*k1/2)
        k3 = f(t+h/2, ys[i] + h*k2/2)
        k4 = f(t+h, ys[i] + h*k3)
        y = y + h*(k1 + 2*k2+2*k3+k4)/6
        t = t + h
        push!(ts, t)
        push!(ys, y)
    end
    return ts, ys
end
```

A solução do pêndulo simples é mostrada na Listagem 22.

A fim de observar o fenômeno descrito na Figura 6 de forma reativa. Será adicionado um *slider* ao notebook Pluto, que será associado ao número de passos utilizado no método.

A Figura 7 mostra uma barra acima do gráfico que pode ser arrastada, alterando o gráfico instantaneamente.

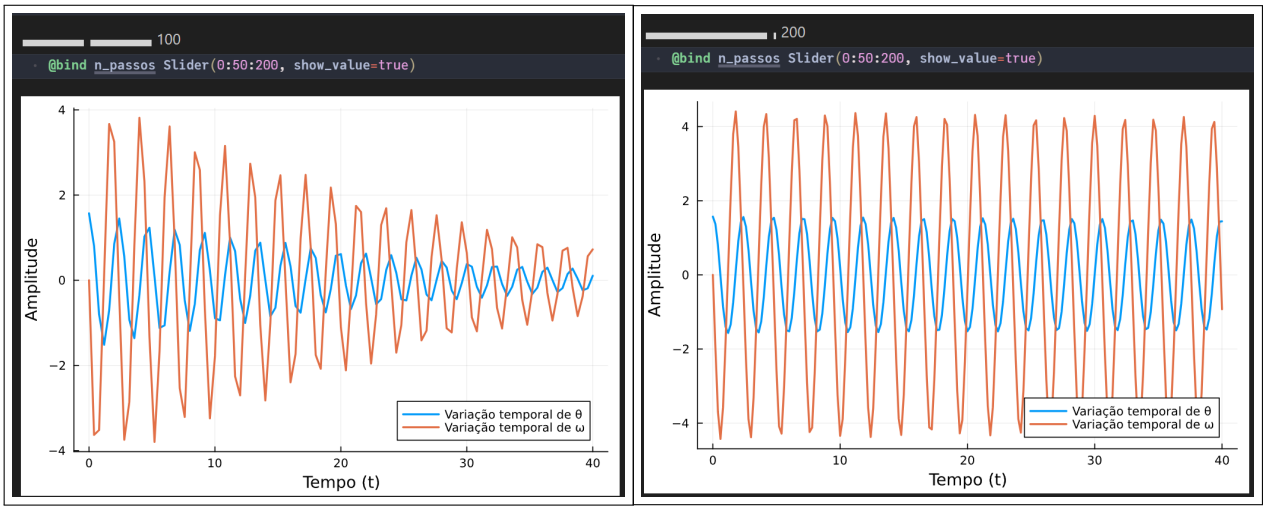
Listagem 22 – PVI do pêndulo simples em Julia

```julia
function pendulum(t, Y, g = 9.81, L = 1.0)
    y1 = Y[1] # Angulo theta
    y2 = Y[2] # Velocidade angular d(theta)/dt

    dy1_dt = y2
    dy2_dt = - (g/L) * sin(y1)

    # Retorna o vetor de derivadas
    return [dy1_dt, dy2_dt]
end

# Valor inicial
y0 = [pi/2, 0.0]
# Parâmetros de Simulação
a, b = 0.0, 40.0 # Tempo inicial e final
n = 2000         # Número de passos

# Aplicando RK4
ts, ys = rk4(pendulum, y0, a, b, n)
# Extraindo os resultados
theta = [y[1] for y in ys]
omega = [y[2] for y in ys]
```

Listagem 23 – Criação de um *slider*

```julia
using PlutoUI
@bind n_passos Slider(0:50:200, show_value=true)
```

Figura 7 – Utilização de um slider para alterar o número de passos



Fonte: Autoria própria (2025).

5 SOLUÇÃO EFICIENTE DE EQUAÇÕES DIFERENCIAIS EM JULIA

Neste capítulo, será apresentada a ferramenta mais amplamente utilizada para a resolução de EDOs no ecossistema Julia: o pacote `DifferentialEquations.jl`.

5.1 O Ecossistema

O pacote `DifferentialEquations.jl` é um ecossistema performático e rico em recursos projetado pela SciML (Scientific Machine Learning) para resolver equações diferenciais na linguagem de programação Julia. Ele é construído sobre a fundação de bibliotecas anteriores, mas modernizado utilizando os recursos de Julia, como *multiple dispatch* e *metaprogramming* (Rackauckas; Nie, 2017).

Segundo Rackauckas e Nie (2017), o objetivo principal do DifferentialEquations.jl é oferecer uma interface de usuário unificada (API) para resolver e analisar várias formas de equações diferenciais, mantendo altos níveis de desempenho e funcionalidade. Entre os principais tipos de equações que o pacote cobre estão as Equações Estocásticas (EDEs), Ordinárias (EDOs), com Atraso (EDRs), Algébricas (EDAs) e Parciais (EDPs). Neste trabalho, o interesse principal está nos recursos oferecidos para solução de EDOs.

Os *solvers* para EDOs nativos em Julia estão localizados principalmente em pacotes como `OrdinaryDiffEq.jl`, que fazem parte do ecossistema `DiferentialEquations.jl`. Entre os principais algoritmos estão:

- `Tsit5()`: Uma escolha padrão e eficiente para problemas não rígidos.
- `vern7()`: Recomendado para problemas não rígidos que exigem alta precisão.
- `AutoTsit5(Rosenbrock23())`: Um algoritmo que se adapta bem tanto a equações rígidas quanto não rígidas, sendo uma boa escolha inicial quando o tipo de equação é desconhecido.
- `Rodas4() ou Rodas5()`: Recomendados para equações rígidas de pequeno porte que utilizam tipos definidos em Julia ou que possuem eventos.
- `QNDF()`: Adequado para equações rígidas de grande porte.

A maior contribuição desta API, e o que a diferencia fundamentalmente das implementações manuais vistas anteriormente, é a filosofia de separação entre o problema e o solucionador.

Nas seções anteriores, o problema (a EDO) e o algoritmo (o laço `for`) estavam postos de tal forma que a troca de método (de `euler` para `rk4`, por exemplo) exigia a reescrita de toda a

lógica da função.

A API do `DifferentialEquations.jl` resolve este problema abstraindo o processo e padronizando-o em três etapas:

1. **Definir um problema**: O usuário define o problema em termos puramente matemáticos: a função f, a condição inicial u_0, o intervalo de tempo t_{span} e os parâmetros p. Isso é encapsulado em um objeto, como um `ODEProblem`.
2. **Resolver o problema**: O usuário solicita ao pacote que resolva este objeto `ODEProblem`, passando o nome de um algoritmo (como `Euler()`, `Heun()`, `RK4()` ou métodos avançados) como argumento para a função `solve`.
3. **Plotar a solução**: Utilização de ferramentas para análise gráfica como `Plots` para visualização e análise dos resultados.

Essa abstração torna o uso de solucionadores complexos substancialmente mais simples. Não é mais necessário lidar com a complexidade da execução do algoritmo (como as múltiplas avaliações da função do RK4); o foco do usuário é transferido para a modelagem correta do problema (a definição do `ODEProblem`).

O comando `solve` aceita diversas palavras-chave (`kwargs`) que controlam o comportamento da integração, algumas estão especificadas no Quadro 1.

Quadro 1 – Principais argumentos da função `solve` e descrições.

Categoria	**Argumentos Chave**	**Descrição**
Controle de Passos	`adaptive` (padrão `true`)	Ativa o dimensionamento de tempo adaptativo.
	`abstol`, `reltol`	Tolerância absoluta e relativa na estimativa de erro local. Padrões para EDOs são 10^{-6} e 10^{-3} respectivamente.
	`dt`, `dtmax`, `dtmin`	Define o tamanho do passo inicial (`dt`) e os limites máximo/mínimo do passo adaptativo.
Controle de Saída	`saveat`	Especifica tempos específicos para salvar a solução (se usado, `save_everystep` e `dense` são `false` por padrão).
	`dense` (padrão `true`)	Denota se deve salvar peças extras necessárias para saída densa (contínua via interpolação de alta ordem).
	`save_everystep` `save_idxs`	Salva o resultado em cada passo de tempo do solver. Salva apenas os índices especificados dos componentes da solução
Diversos	`maxiters`	Número máximo de iterações antes de parar (padrão 10^5)
	`callback` (padrão `true`)	Especifica uma função de retorno (callback) para manipulação de eventos e descontinuidades
	`alg_hints`	Fornece dicas de alto nível ao seletor de algoritmo.

Fonte: Elaborado pelo autor com base na documentação SciML.

5.1.1 Problema do Carbono-14 com DifferentialEquations.jl

Para demonstrar a simplicidade da abordagem, o problema do decaimento do Carbono-14 será resolvido novamente. A primeira etapa é carregar o pacote e definir o `ODEProblem`. Como já foi dito, o pacote DifferentialEquations.jl abrange uma grande variedade de solucionadores e por isso é bastante extenso. Como o foco é voltado ao fluxo de trabalho de EDOs, no exemplo será carregado apenas o módulo `OrdinaryDiffEq`.

A função da EDO e os parâmetros são exatamente os mesmos que foram usados nas implementações manuais.

Listagem 24 – Definição do `ODEProblem` para o Carbono-14

```
import OrdinaryDiffEq as ODE
using Plots

t½ = 5730
p = log(2) / t½
intervalo = (0.0,10000.0)
N0 = 1.0 # Condição Inicial
num_passos = 4
parametros = p

edo = (N,p,t) -> -p*N
prob = ODEProblem(edo, N0, intervalo, parametros)
```

Com o `prob` definido, segue-se para a segunda etapa: a chamada da função. Para permitir uma comparação direta, pode-se "forçar" o pacote a usar os mesmos métodos implementados manualmente. Para isso, basta usar a opção `adatative=false` (para desligar o passo adaptativo) e `dt=2500` (para definir o mesmo passo $h = 2500$ utilizado na implementação manual).

Listagem 25 – Solução de um `ODEProblem`

```
h = 2500.0
sol_euler = ODE.solve(prob, ODE.Euler(), dt=h, adaptive=false)
sol_heun  = ODE.solve(prob, ODE.Heun(), dt=h, adaptive=false)
sol_rk4   = ODE.solve(prob, ODE.RK4(), dt=h, adaptive=false)
```

Percebe-se que os valores obtidos na Tabela 7 são idênticos aos da Tabela 6. Porém, foram necessárias apenas três linhas de código para implementar todos os métodos. Isso mostra que o pacote está executando os mesmos algoritmos, mas de forma encapsulada, abstraindo a

Tabela 7 – Comparação entre o valor analítico e as aproximações para o decaimento do carbono-14 usando DifferentialEquations.jl

t	$N_{analtico}$	N_{solve_euler}	N_{solve_heun}	N_{solve_rk4}
0	1	1	1	1
2500	0,73903	0,69758	0,743309	0,739047
5000	0,54616	0,486618	0,552508	0,546191
7500	0,40363	0,339455	0,410684	0,403661
10000	0,29829	0,236797	0,305265	0,298325

Fonte: Autoria própria (2025)

complexidade da implementação.

No entanto, a verdadeira vantagem não é apenas replicar métodos simples, mas acessar algoritmos profissionais que seriam muito complexos de implementar manualmente.

Por padrão, a função `solve` utiliza métodos de passo adaptativo; ou seja, o passo `h` é ajustado automaticamente para que o erro permaneça abaixo de uma tolerância pré-definida (por padrão, `reltol=1e-3` e `abstol=1e-6`).

O solucionador padrão escolhido para EDOs simples é o `Tsit5` - um método de quinta ordem. Para utiliza-lo, basta chamar a função `solve` passando o método como argumento.

Listagem 26 – Solução de um `ODEProblem` com o *solver* Tsit5

```
1 sol_Tsit5 = ODE.solve(prob, ODE.Tsit5())
```

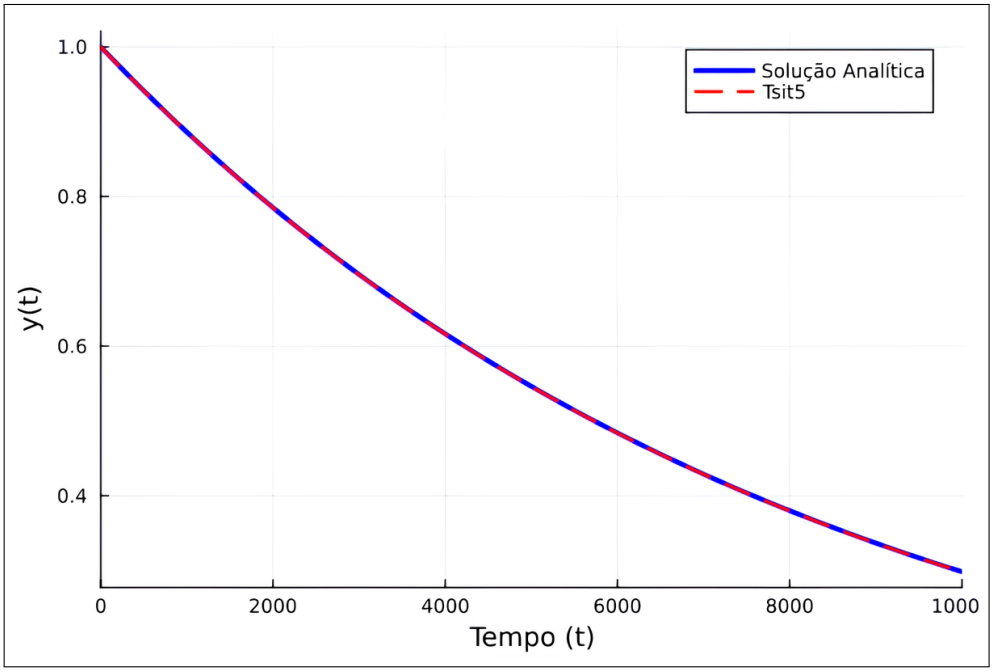
Tabela 8 – Comparação entre o valor analítico e aproximado para o decaimento do carbono-14 utilizando o método de passo adaptativo `Tsit5`

t	$N_{analtico}$	N_{Tsit5}	Erro %
0	1	1	0
2500	0,7390274338	0,7390274430	$1{,}23 \cdot 10^{-6}$
5000	0,5461615479	0,5461616155	$1{,}24 \cdot 10^{-5}$
7500	0,4036283672	0,4036284368	$1{,}73 \cdot 10^{-5}$
10000	0,2982924364	0,2982925832	$4{,}92 \cdot 10^{-5}$

Fonte: Autoria própria (2025)

A Figura 8 e a Tabela 8 mostram que a solução obtida pelo método padrão sobrepõe-se à analítica e mantém um erro relativo na ordem de $10^{-5}\%$.

Figura 8 – Comparação entre a solução real e a numérica usando o método de passo adaptativo Tsit5.



Fonte: Autoria própria (2025)

5.1.2 Pêndulo Simples com DifferentialEquations.jl

De forma similar ao problema de decaimento do Carbono-14, serão utilizados os mesmos parâmetros da implementação manual. Com isso, a Listagem 27 mostra a definição do ODEProblem e a Listagem 28, mostra como chamar o solucionador.

Listagem 27 – Definição do `ODEProblem` para o Pêndulo Simples

```
import OrdinaryDiffEq as ODE
using Plots

function pendulosimples(u, du, p )
    du[1] = u[2]
    du[2] = -(g / L) * sin(u[1])
    return [du[1], du[2]]
end
L = 1.0 ; g = 9.81
p = [L, g]
u0 = [pi/2, 0.0]
intervalo = (0.0, 40.0) # Tempo inicial e final
prob = ODE.ODEProblem(pendulosimples, u0, intervalo, p)
```

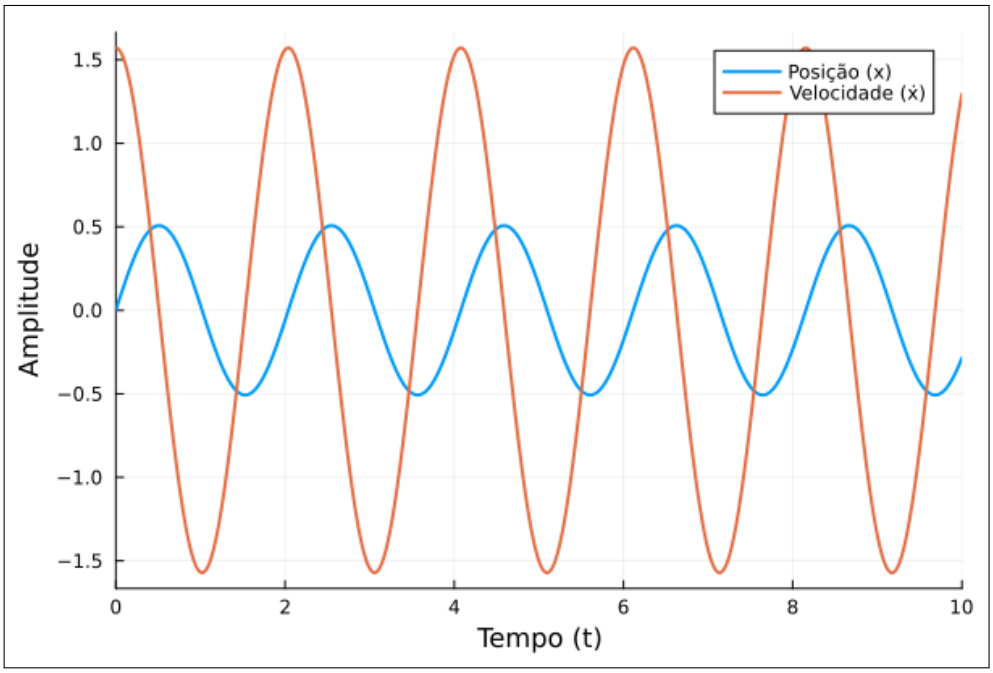
A Figura 9 mostra a evolução temporal do ângulo e da velocidade angular. Utilizando o

Listagem 28 – Chamada de solucionador para o Pêndulo Simples

```
1 sol = ODE.solve(prob, ODE.Tsit5())
```

solucionador de passo adaptativo `Tsit5`, é possível notar um aumento considerável na precisão; evidenciado pela suavidade das curvas.

Figura 9 – Evolução temporal Pêndulo Simples utilizando o método Tsit5.



Fonte: Autoria própria (2025)

6 RESULTADOS

6.1 Perspectivas de Uso da Linguagem Julia na Disciplina de Cálculo Numérico

Com base na investigação realizada até aqui, esta seção discute como Julia pode ser incorporada à disciplina de Cálculo Numérico, seguindo a estrutura da ementa tradicional: sistemas lineares, solução de equações não lineares, interpolação, integração numérica e por fim equações diferenciais ordinárias que foi o foco principal do projeto.

6.1.1 Perspectivas por Conteúdo da Ementa do Componente Curricular

A seguir, discute-se como Julia e seu ecossistema podem ser utilizados em cada tema da ementa de Cálculo Numérico, enfatizando atividades que podem ser aplicadas em aula e experimentos que reforcem a compreensão conceitual.

6.1.1.1 Sistemas Lineares: Métodos Diretos e Iterativos

Julia fornece, no módulo padrão `LinearAlgebra`, implementações robustas de operações matriciais, como decomposições LU, QR e SVD. Além disso, é possível demonstrar graficamente a convergência de métodos iterativos, como Jacobi ou Gauss-Seidel.

O professor pode, por exemplo, utilizar um notebook Pluto para criar *sliders* que alteram elementos da matriz e observar em tempo real como o vetor solução se aproxima (ou diverge) do resultado esperado, dependendo do condicionamento da matriz.

6.1.1.2 Zeros de Funções e Raízes

Métodos como Bisseção e Newton-Raphson são essencialmente geométricos. A capacidade de plotagem rápida do Julia permite que o aluno visualize a função $f(x)$ e as retas tangentes (no caso de Newton) sendo traçadas iterativamente. A reatividade dos notebooks permite que o aluno altere o "chute inicial"(x_0) e perceba instantaneamente como isso afeta a convergência, reforçando a compreensão geométrica do método.

6.1.1.3 Interpolação e Ajuste de Curvas

Este tópico é particularmente beneficiado pela abordagem visual e interativa. Em um ambiente `Pluto.jl`, é possível configurar pontos de dados experimentais interativos que podem

ser “arrastados” pelo cursor. O aluno pode visualizar, simultaneamente, as curvas geradas por um Polinômio Interpolador de Lagrange e por Splines Cúbicas se ajustando instantaneamente às novas posições dos pontos.

Essa interatividade permite demonstrar fenômenos complexos, como o Fenômeno de Runge (oscilações excessivas nas bordas do intervalo ao utilizar polinômios de grau elevado). Ao aumentar o grau do polinômio por meio de um controle deslizante, o aluno observa a degradação da aproximação nas extremidades, constatando o conceito teórico de que um maior grau não implica necessariamente uma melhor aproximação.

6.1.1.4 Integração Numérica

A compreensão das regras de integração, como a Regra dos Trapézios e a Regra de Simpson, depende fundamentalmente da visualização da área sob a curva e da forma como a função é aproximada em cada subintervalo. Por meio da biblioteca `Plots.jl`, é fácil desenhar os trapézios ou as parábolas sobre a função integrada. Um *slider* pode ser configurado para alterar o número de subintervalos N da partição. O aluno observa visualmente a redução da área de erro (a diferença entre a área do trapézio e a área real da função) conforme N aumenta.

6.1.1.5 Solução de Equações Diferenciais Ordinárias

Conforme explorado neste trabalho, a linguagem Julia com o pacote `DifferentialEquations.jl` permite abordar uma grande variedade de métodos para solução de EDOs, como por exemplo:

- métodos manuais implementados do zero (Euler, Heun, RK2, RK4);
- métodos de passo adaptativo e com controle de erro (Tsit5, Vern7);
- métodos implícitos para problemas rígidos;
- integradores geométricos, como os métodos simpléticos - úteis para sistemas oscilatórios.

A seguir, serão discutidos alguns pontos importantes observados durante o trabalho acerca da linguagem, levando em consideração alguns critérios.

6.1.1.6 Legibilidade e Expressividade do Código

Um dos principais desafios no ensino de métodos numéricos é a tradução da notação matemática apresentada na teoria para o código. As implementações realizadas neste trabalho demonstram que Julia oferece um alto grau de isomorfismo entre a equação e o código.

Observando a implementação do Método de Euler (Listagem 15), a linha central do algoritmo:

```
y[i+1] = y[i] + h*f(y[i], p, t[i])
```

é quase uma tradução perfeita da expressão matemática $y_{i+1} = y_i + h \cdot f(t_i, y_i)$. O mesmo pode ser observado para o método de Runge-Kutta (Listagem 19).

Esse aspecto diminui a distância entre a equação e o código, pois ambos estão utilizando a mesma notação. Tornando assim o código mais legível.

6.1.1.7 Possibilidade de Experimentação Interativa

A experimentação é um componente essencial no aprendizado de métodos numéricos, pois permite observar na prática efeitos como erro, convergência, sensibilidade e dependência de parâmetros. A linguagem Julia permitiu a realização de diversos testes sem a necessidade de alterar a estrutura dos algoritmos de forma significativa. Foi possível, por exemplo, explorar o uso do passo fixo e do passo adaptativo facilmente, permitindo comparar graus de suavidade e precisão entre as implementações manuais e as fornecidas pelo pacote `DifferentialEquations`.

A resolução de exemplos deste trabalho adotou uma abordagem progressiva, na qual os problemas se tornaram cada vez mais complexos. Foi possível perceber que a linguagem não introduziu saltos de complexidade tão significativos entre os problemas, mesmo quando a natureza do problema foi alterada (de 1ª para 2ª ordem). Para o sistema pêndulo simples, por exemplo, a mesma função (`rk4_sistema`) foi utilizada para resolver as equações pelo método Runge Kutta de quarta ordem. Isso foi possível devido ao despacho múltiplo, uma das características fundamentais da linguagem.

Estes aspectos permitem que o aluno explore diferentes sistemas, reutilize o código empregado e receba *feedbacks* que validam ou não sua implementação. Isso favorece o teste de hipóteses criadas pelos alunos, promovendo uma aprendizagem exploratória.

6.1.1.8 Visualização e Interpretação de Resultados

O processo de plotagem de gráficos em Julia assim como em outras linguagens mostrou-se simples e eficiente. O pacote `Plots.jl` permitiu visualizar as soluções dos problemas sem muita complicação e com baixo custo de codificação.

Uma vantagem importante apresentada, foi a utilização de notebooks reativos com o Pluto. A possibilidade de vizualizar mudanças nos gráficos em tempo real quando se altera um parâmetro do problema pode facilitar consideravelmente a compreenção dos alunos.

A visualização gráfica é fundamental para que o aluno compreenda as características do sistema estudado, em vez de se ater a compreensão abstrata de uma tabela de números. Esse quesito foi bem atendido pela linguagem.

6.1.1.9 Eficiência Computacional no Contexto Didático

Embora o ensino de métodos numéricos não tenha como foco primário a otimização de desempenho, a eficiência computacional também é importante no ambiente de aprendizagem. Isso acontece porque, em muitos problemas, é necessário realizar uma quantidade massiva de simulações para que se possa compreender toda a dinâmica do sistema.

A linguagem Julia, por utilizar compilação *Just-in-Time* (JIT) via LLVM, demonstrou um desempenho satisfatório. As implementações manuais utilizando RK4 foram executadas de forma fluída, mesmo com um número elevado de passos; a Tabela 9 exibe o gasto de tempo e a alocação de memória para a solução dos problemas.

Tabela 9 – Gasto de tempo e a alocação de memória para a solução dos problemas

Problema	**Método**	**Implementação**	**Tempo Médio**	**Alocações**	**Memória**
Decaimento C-14	Euler	Manual	141.98 ns	5	224 bytes
		Solver	1.91 μs	91	3.66 KiB
	Heun	Manual	374.75 ns	5	224 bytes
		Solver	2.13 μs	107	4.17 KiB
	RK4	Manual	2.44 μs	151	4.78 KiB
		Solver	4.10 μs	203	7.17 KiB
Pêndulo Simples	RK4	Manual	1.09 ms	52009	1.63 MiB
	Tsit5	Solver	406.40 μs	25421	921.53 KiB

Fonte: Autoria própria (2025)

Nota-se que no Carbono-14 a implementação manual é muito mais rápida que o solver. Isso acontece porque, em problemas simples, o custo "fixo" da biblioteca (configurações, verificação de tipos) é maior que o cálculo em si. Portanto, para problemas simples, a implementação manual é suficientemente eficiente e leve. Já para o pêndulo simples, é possível observar que o Solver `Tsit5` é cerca de duas vezes mais rápido que o RK4 manual e utiliza metade da memória.

Além da vantagem de não precisar escrever manualmente o algoritmo de solução, a biblioteca se demonstra mais otimizada e apresenta melhor precisão.

6.1.2 Considerações Finais sobre as Perspectivas Pedagógicas

A exposição dos conteúdos anteriores mostra que Julia oferece um ambiente coerente para lidar com todos os tópicos da ementa de Cálculo Numérico e pode representar uma vantagem significativa no ensino.

Do ponto de vista pedagógico, destacam-se os seguintes benefícios:

- **Menor distância entre teoria e código**: A sintaxe expressiva reduz o conflito entre fórmula e implementação.
- **Feedback imediato**: A reatividade do Pluto.jl favorece aprendizagem exploratória e baseada em hipóteses.
- **Alto potencial visual**: Gráficos e animações fortalecem a intuição matemática.
- **Reuso natural de código**: Métodos implementados para um capítulo são reaproveitados em capítulos seguintes.
- **Redução do estresse computacional**: Código curto, legível e semelhante à notação matemática minimiza frustração com detalhes de sintaxe.

Esses fatores estão diretamente alinhados com desafios didáticos da disciplina, como a organização do pensamento algorítmico, a visualização de erros e a compreensão da convergência.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho investigou o uso da linguagem de programação Julia como ferramenta de apoio ao ensino-aprendizagem da solução numérica de EDOs. Para isso, foram apresentados os fundamentos teóricos das EDOs e dos principais métodos numéricos clássicos, bem como suas implementações computacionais na linguagem Julia.

A análise dos métodos de Euler, Heun e Runge-Kutta permitiu demonstrar as diferenças de precisão entre métodos de ordens diferentes. Os estudos de caso do decaimento do Carbono-14 e do pêndulo simples mostraram como a escolha do método numérico e do tamanho do passo influencia diretamente a qualidade da solução obtida. Em particular, foi observado que, embora métodos de alta ordem como o Runge-Kutta de quarta ordem apresentem elevada precisão, eles não garantem a conservação de propriedades físicas, como a energia mecânica em sistemas oscilatórios.

No contexto educacional, a linguagem Julia demonstrou-se adequada principalmente por sua sintaxe simples, proximidade com a notação matemática e bom desempenho computacional. O uso de pacotes especializados, como o DifferentialEquations.jl, possibilita uma transição natural entre a implementação manual dos algoritmos e o emprego de solucionadores profissionais, favorecendo a integração entre teoria e prática.

Ressalta-se que este estudo possui caráter exploratório, não contemplando uma avaliação quantitativa do impacto pedagógico junto a estudantes. Como trabalhos futuros, sugere-se a realização de uma pesquisa tendo como participantes os alunos de Cálculo Numérico; afim de obter dados quantitivos que evidenciem os beneficios da adoção da linguagem Julia. Além disso, pode-se realizar estudos comparativos com outras linguagens utilizadas no ensino, bem como a investigação de métodos numéricos mais avançados, como integradores simpléticos e métodos adaptativos, aplicados a sistemas dinâmicos mais complexos.

Conclui-se, portanto, que a linguagem Julia constitui uma alternativa promissora para o ensino de métodos numéricos aplicados à solução de EDOs, contribuindo para a modernização do ensino de Cálculo Numérico e para uma melhor articulação entre os aspectos teóricos e computacionais da disciplina.

REFERÊNCIAS

AMARAL, T. R. d.; LEITE, N. M. G.; SILVA, A. O. d. O ensino de calculo numÉrico utilizando o scilab. In: **Anais...** Canoas: ULBRA, 2013. p. 1–9.

AZEVEDO, D. d. S. N.; SILVA, F. d. S. A integração do python no ensino do cálculo: Concepções acadêmicas a partir de uma revisão de literatura. In: **Anais...** Manaus: Sociedade Brasileira de Educação Matemática, 2025. p. 1–12.

BAKER, G.; BLACKBURN, J. **The Pendulum: A Case Study in Physics**. OUP Oxford, 2005. ISBN 9780198567547. Disponível em: https://books.google.com.br/books?id=t4ISDAAAQBAJ.

BEZANSON, J. *et al.* **Julia: A Fresh Approach to Numerical Computing**. 2015. Disponível em: https://arxiv.org/abs/1411.1607.

BOYCE, W. E.; DIPRIMA, R. C. **Elementary Differential Equations and Boundary Value Problems/**. 8th. ed. United States of America:: Wiley,, 2005. Includes index.

BURDEN, R. L.; FAIRES, J. D. **Numerical Analysis**. 9th. ed. Boston, MA, USA: Brooks/Cole, Cengage Learning, 2011. ISBN 978-0538735643.

CHAPRA, S. C. **Métodos Numéricos Aplicados com MATLAB® para Engenheiros e Cientistas - 3.ed.** [S.l.]: AMGH Editora, 2013.

CHAPRA, S. C.; CANALE, R. P. **Numerical methods for engineers**. 7. ed. [S.l.]: Mcgraw-Hill Education, Cop, 2015.

CRUZ, E. C. **UM PERCURSO DE ENSINO EM MÉTODOS NUMÉRICOS: ESTUDO DE CASO EM UMA TURMA DE LICENCIATURA EM MATEMÁTICA**. 276 p. Tese (Tese (Doutorado em Didática de Ciências e Tecnologias)) — UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO, Vila Real, 2024.

FINGER, A. F. *et al.* Avaliação de usabilidade do sofemn: Software de apoio ao ensino de métodos numéricos. In: **Anais...** Online: CBIE, 2021. p. 103–112.

GAMA, C. L. G.; GOMES, M. d. N.; PIRES, L. A. Da teoria à prática: problematização e metodologias diferenciadas no cálculo numérico. **Ensino em Re-Vista**, v. 25, n. 1, p. 234–255, jan./abr. 2018. Disponível em: https://seer.ufu.br/index.php/emrevista/article/view/41375.

JULIAHUB. **Celeste: Parallel Supercomputing for Astronomy with Julia**. 2023. https://juliahub.com/case-studies/celeste. Acesso em: 10 dez. 2025.

LANGUAGE, J. **Julia Documentation**. 2025. https://docs.julialang.org/. Acesso em: 24 nov. 2025.

MONTEIRO, L. **Sistemas Dinâmicos**. Editora Livraria da Física, 2002. ISBN 9788588325081. Disponível em: https://books.google.com.br/books?id=w0eYcHddMq0C.

MOREIRA, J. P. d. S. *et al.* Estudo e modelagem de um pÊndulo simples atravÉs de equaÇÕes diferenciais e anÁlise de vÍdeo assistida por computador. **Qualif**, Qualif - REVISTA ACADÊMICA - ENSINO DE CIÊNCIAS E TECNOLOGIAS IFSP – CAMPUS CUBATÃO, 2019.

MOTA, R. P. B. Ensino de cálculo numérico através de rotinas didáticas e scilab no ambiente web. In: **Anais...** Águas de Lindóia: CNMAC, 2012. p. 1027–1033.

PHILLIPS, L. **Practical Julia**. 1. ed. San Francisco: No Starch Press, 2023. ISBN 9781718502765.

RACKAUCKAS, C.; NIE, Q. Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia. **Journal of Open Research Software**, Ubiquity Press, v. 5, n. 1, p. 15, 2017.

RUGGIERO, M. A. G.; LOPES, V. L. d. R. **Cálculo Numérico: aspectos teóricos e computacionais**. 2. ed. São Paulo: Pearson Makron Books, 1996. ISBN 978-8534602044.

SASSANO, F. *et al.* O uso de ferramentas computacionais no processo de ensino e aprendizagem de matemática na ead. **Revisata Brasileira de Aprendizagem Aberta e a Distância**, v. 20, n. 1, p. 1–21, maio 2021. Disponível em: https://abed.emnuvens.com.br/RBAAD/article/view/551.

SCIML. **DiffEqDocs**. 2025. https:docs.sciml.ai/DiffEqDocs. Acesso em: 14 dez. 2025.

SILVA, I. S. B. **DESENVOLVIMENTO DE UMA TOOLBOX NO MATLAB PARA O AUXÍLIO DO PROCESSO DE ENSINO-APRENDIZAGEM DOS MÉTODOS NUMÉRICOS**. Pau dos Ferros, 2021. 61 p.

ZILL, D. G. **Equações diferenciais com aplicações em modelagem**. [S.l.]: Cengage Learning, 2016.