Métodos Numéricos de Zero Real: Teoria, Implementação e Comparação

Gabriel dos Santos Schmitz
Vitor Augusto Salata de Souza
RA 2487438 & 2461200, Engenharia de Computação,
gabrielzschmitz@protonmail.com
vitsou@alunos.utfpr.edu.br

I. Introdução

A busca por zeros de funções é um problema fundamental na matemática aplicada, engenharia e computação científica. Métodos numéricos desempenham um papel essencial nessa tarefa, pois muitas funções não possuem soluções analíticas exatas, tornando necessário o uso de aproximações iterativas.

Neste artigo, exploraremos cinco dos principais algoritmos para encontrar zeros reais de funções: *Bisseção, Newton-Raphson, Falsa Posição, Ponto Fixo e Secante*. Cada um desses métodos possui vantagens e limitações, variando em critérios como taxa de convergência, robustez e eficiência computacional.[1]

Além de analisar o funcionamento teórico de cada abordagem, desenvolveremos um programa que permitirá visualizar esses algoritmos em ação. Isso possibilitará uma compreensão mais intuitiva dos diferentes comportamentos e estratégias utilizadas para aproximar as raízes de uma função.

Por fim, realizaremos uma comparação detalhada do desempenho de cada método, avaliando sua eficácia em diferentes cenários. Com essa abordagem, buscamos oferecer um guia prático e acessível para estudantes, pesquisadores e profissionais que desejam compreender e aplicar algoritmos de busca de raízes de forma eficiente.[2]

II. Fundamentos Teóricos

Nesta seção, são apresentados os métodos numéricos implementados. Cada método é descrito em termos de conceito, funcionamento, convergência e vantagens/desvantagens.[2] [1]

A. *Método da Bisseção*

O método da bisseção é um algoritmo de busca de raízes que divide repetidamente um intervalo ao meio e seleciona o subintervalo que contém a raiz.[3] [4]

a.1) Funcionamento

Dada uma função contínua $f(x)$ e um intervalo $[a, b]$ onde $f(a) \cdot f(b) < 0$, o método calcula o ponto médio $c = \frac{a+b}{2}$. Se $f(c) = 0$, c é a raiz. Caso contrário, o intervalo é reduzido para $[a, c]$ ou $[c, b]$, dependendo do sinal de $f(c)$.

a.2) Convergência

O método converge linearmente, garantindo uma raiz desde que $f(x)$ seja contínua no intervalo e haja uma mudança de sinal. O erro absoluto é reduzido pela metade a cada iteração, e o número de iterações necessárias para atingir uma precisão ϵ é dado por:

$$n \geq \frac{\log(b-a) - \log(\epsilon)}{\log(2)}.$$

a.3) Vantagens e Desvantagens

- **Vantagens**: Simplicidade e garantia de convergência.
- **Desvantagens**: Convergência lenta comparada a outros métodos.

B. *Método de Newton-Raphson*

O método de Newton-Raphson é um algoritmo iterativo que usa a derivada da função para aproximar a raiz.[4] [5]

b.1) Funcionamento

Dada uma função $f(x)$ e sua derivada $f'(x)$, o método começa com uma estimativa inicial x_0. A cada iteração, a aproximação é atualizada por:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

O processo repete até que $|x_{n+1} - x_n|$ seja menor que uma tolerância pré-definida.

b.2) Convergência

O método converge quadraticamente, desde que a estimativa inicial seja próxima da raiz e $f'(x) \neq 0$.

b.3) Vantagens e Desvantagens

- **Vantagens**: Convergência rápida.
- **Desvantagens**: Requer o cálculo da derivada e pode divergir se a estimativa inicial for inadequada.

C. *Método da Falsa Posição (Regula Falsi)*

O método da falsa posição é uma variação do método da bisseção que usa uma aproximação linear para estimar a raiz.[3]

c.1) Funcionamento

Dada uma função $f(x)$ e um intervalo $[a, b]$ onde $f(a) \cdot f(b) < 0$, o método calcula a interseção da reta que liga $(a, f(a))$ e $(b, f(b))$ com o eixo x:

$$c = \frac{a \cdot f(b) - b \cdot f(a)}{f(b) - f(a)}.$$

Se $f(c) = 0$, c é a raiz. Caso contrário, o intervalo é reduzido para $[a, c]$ ou $[c, b]$, dependendo do sinal de $f(c)$.

c.2) Convergência

Mais rápida que a bisseção, mas ainda linear.

c.3) Vantagens e Desvantagens

- **Vantagens**: Mais eficiente que a bisseção em muitos casos.
- **Desvantagens**: Pode ser lento se a função for muito não-linear.

D. Método do Ponto Fixo

O método do ponto fixo transforma o problema de encontrar a raiz de $f(x) = 0$ em encontrar um ponto fixo $g(x) = x$.[4]

d.1) Funcionamento

Reescreva $f(x) = 0$ como $x = g(x)$. Escolha uma estimativa inicial x_0 e itere:

$$x_{n+1} = g(x_n).$$

O processo repete até que $|x_{n+1} - x_n|$ seja menor que uma tolerância pré-definida.

d.2) Convergência

Depende da escolha de $g(x)$. A convergência é garantida se $|g'(x)| < 1$ em um intervalo contendo a raiz.

d.3) Vantagens e Desvantagens

- **Vantagens**: Simplicidade e flexibilidade na escolha de $g(x)$.
- **Desvantagens**: Pode divergir se $g(x)$ não for adequada.

E. Método da Secante

O método da secante é uma variação do método de Newton que não requer o cálculo da derivada.[6]

e.1) Funcionamento

Dada uma função $f(x)$ e duas estimativas iniciais x_0 e x_1, o método calcula a próxima aproximação usando a fórmula:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

O processo repete até que $|x_{n+1} - x_n|$ seja menor que uma tolerância pré-definida.

e.2) Convergência

Superlinear (ordem ≈ 1.61), mas não tão rápida quanto o método de Newton.

e.3) Vantagens e Desvantagens

- **Vantagens**: Não precisa do cálculo da derivada.
- **Desvantagens**: Pode divergir se as estimativas iniciais não forem adequadas.

F. Comparação Geral dos Métodos

A Tabela 2 resume as características principais dos métodos discutidos.

Método	Tipo Converg.	Deriv. Necessária?	Garantia Converg.	Veloc. Converg.
Bisseção	Linear	Não	Sim	Lenta
Newton-Raphson	Quadrática	Sim	Não	Rápida
Falsa Posição	Linear	Não	Sim	Moderada
Ponto Fixo	Dep. $g(x)$	Não	Dep. $g(x)$	Variável
Secante	Superlinear	Não	Não	Moderada

TABLE 2. Comparação dos métodos numéricos.

III. Implementação

A implementação dos métodos de busca de raízes foi desenvolvida em Python, utilizando bibliotecas como `ttkbootstrap` para a interface gráfica e `matplotlib` para a visualização dos métodos em ação. O código está estruturado em três arquivos principais: `app.py`, `ui.py` e `functions.py`.

A. Visão Geral

A estrutura do código é organizada da seguinte forma:

- **app.py**: Ponto de entrada da aplicação, responsável por inicializar a interface gráfica.
- **ui.py**: Contém a implementação da interface gráfica e gerencia a interação do usuário.
- **functions.py**: Implementa os métodos numéricos de busca de raízes e fornece suporte para os cálculos.

B. Interface

A interface gráfica do aplicativo foi implementada no arquivo `ui.py`, utilizando a biblioteca `ttkbootstrap` para um design moderno e intuitivo. A interface permite ao usuário inserir uma função $f(x)$, definir um intervalo ou chute inicial, escolher um método de busca de raízes e visualizar os resultados tanto numericamente quanto graficamente.

O código é estruturado em três classes principais:

- **RootFinderUI**: Gerencia a interface principal, incluindo campos de entrada, botões de ação e a exibição dos resultados.
- **PlotManager**: Responsável por gerar e atualizar os gráficos das funções e dos métodos utilizados.

- **ThemeManager**: Permite alternar entre temas claros e escuros para melhor experiência visual.

b.1) RootFinderUI: Estrutura da Interface

A classe `RootFinderUI` define os elementos principais da interface e gerencia as interações do usuário. A inicialização ocorre no método `__init__`, onde são criados os seguintes componentes:

- Campos de entrada para a função $f(x)$, $g(x)$ (caso necessário), intervalo $[a, b]$ ou chute inicial e tolerância.
- Um menu suspenso para selecionar o método numérico desejado.
- Botões para executar a busca da raiz, copiar os resultados, salvar o gráfico e alternar o tema.
- Um rótulo para exibir os resultados numéricos.
- Uma área gráfica onde a função e a raiz encontrada são plotadas.

O trecho de código abaixo demonstra a criação do campo de entrada para a função $f(x)$:

```
ttk.Label(input_frame, text="f(x):").grid(row=0,
    column=0, sticky="w")
self.f_entry = ttk.Entry(input_frame, width=40)
self.f_entry.grid(row=0, column=1, columnspan=3)
self.f_entry.insert(0, "sin(x) + cos(x) + 1")
```

A entrada padrão para a função é $f(x) = \sin(x) + \cos(x) + 1$, podendo ser alterada pelo usuário.

b.2) Seleção de Método e Ajuste de Parâmetros

Os métodos disponíveis são carregados automaticamente a partir da classe `RootFinderMethods`, permitindo que novos algoritmos sejam adicionados sem necessidade de modificar a interface:

```
self.methods = self._get_root_finding_methods()
self.method_combobox = ttk.Combobox(
    input_frame, textvariable=self.method_var,
        values=list(self.methods.keys())
)
```

A função `_get_root_finding_methods` percorre os métodos definidos na classe `RootFinderMethods` e os adiciona à interface permitindo selecionar funções a ignorar.

b.3) Execução do Cálculo e Exibição dos Resultados

Ao pressionar o botão "Solve", o método `solve` é chamado, realizando as seguintes operações:

1) Obtém os valores inseridos pelo usuário.
2) Converte a função de string para código executável usando `eval`.
3) Chama o método apropriado de `RootFinderMethods`.
4) Exibe a raiz encontrada e o número de iterações.
5) Atualiza o gráfico com a função e a raiz.

O código abaixo ilustra como a solução é processada:

```
root, iterations, computation_time = self.
    function_solver.solve(
    f_str, a, b, tol, method_name, g_str
)
result_text = f"Root: {root:.6f}\nIterations: {
    iterations}\nTime: {computation_time:.6f} s"
self.result_label.config(text=result_text)
```

Se um erro for encontrado (como ausência de mudanças de sinal no método da bisseção), uma mensagem de erro é exibida ao usuário.

b.4) Gerenciamento da Visualização Gráfica

A classe `PlotManager` é responsável por criar e atualizar os gráficos. O método `update_plot` traça a função no intervalo fornecido e destaca a raiz encontrada:

```
x = np.linspace(a, b, 400)
y = [self._eval_function(f_str, xi) for xi in x]
self.ax.plot(x, y, label=f"f(x) = {f_str}")
self.ax.scatter([root], [0], color="red", label="
    Root")
self.ax.axhline(0, color="black", linewidth=0.5)
```

Além disso, um botão permite salvar o gráfico gerado para análise posterior.

b.5) Personalização da Aparência

Para melhorar a usabilidade, a classe `ThemeManager` gerencia os esquemas de cores do aplicativo. Dois temas estão disponíveis: um claro e um escuro, alternáveis pelo usuário.

O código abaixo ilustra a alternância de temas:

```
def toggle_theme(self):
    self.current_theme = "minty" if self.
        current_theme == "darkly" else "darkly"
    self.style.theme_use(self.current_theme)
```

b.6) Resumo

A interface do aplicativo proporciona uma experiência interativa e intuitiva para explorar diferentes métodos numéricos de busca de raízes. Os usuários podem visualizar o comportamento das funções e comparar a eficiência dos algoritmos de forma dinâmica.

C. Funções

Os métodos numéricos utilizados para encontrar raízes de funções foram implementados no arquivo `functions.py`. A implementação está organizada na classe `RootFinderMethods`, que contém cada método como uma função estática, permitindo chamadas diretas sem necessidade de instanciar objetos.

Além disso, o módulo inclui a classe `FunctionSolver`, responsável por interpretar a função fornecida pelo usuário, selecionar o método adequado e gerenciar a execução dos cálculos. Essa organização modular facilita a reutilização e a extensão do código, permitindo adicionar novos métodos sem impactar a estrutura principal.

A seguir, são detalhadas as implementações de cada método, explicando suas operações e como os parâmetros de entrada são utilizados para conduzir o processo iterativo de busca da raiz.

c.1) Bisseção

A função `bisection` recebe como entrada uma função f, um intervalo inicial $[a, b]$, uma tolerância e um número máximo de iterações. O código inicia verificando se $f(a)$ e $f(b)$ possuem sinais opostos. Caso contrário, uma exceção é levantada.

O loop principal divide o intervalo ao meio repetidamente até que a raiz seja encontrada dentro da tolerância ou o número máximo de iterações seja atingido. A cada iteração:

1) O ponto médio c do intervalo é calculado.
2) Se $f(c)$ for zero, a raiz foi encontrada.
3) Caso contrário, o intervalo é reduzido dependendo do sinal de $f(c)$.

```
@staticmethod
def bisection(f, a, b, tol=1e-6, max_iter=100):
    if f(a) * f(b) >= 0:
        raise ValueError("The function must have
            opposite signs at the endpoints.")
    iterations = 0
    while (b - a) / 2 > tol and iterations <
        max_iter:
        c = (a + b) / 2
        if f(c) == 0:
            break
        elif f(c) * f(a) < 0:
            b = c
        else:
            a = c
        iterations += 1
    return (a + b) / 2, iterations
```

c.2) Newton-Raphson

A função `newton_raphson` recebe uma função f, um chute inicial x_0, uma tolerância e um número máximo de iterações. A cada iteração:

1) Calcula-se a derivada numérica de f no ponto atual x_0.
2) Se a derivada for muito pequena, o algoritmo para para evitar divisões por valores próximos de zero.
3) O próximo valor de x_0 é calculado usando a equação $x_1 = x_0 - f(x_0)/f'(x_0)$.

O processo continua até que o valor de $f(x)$ esteja dentro da tolerância ou que o número máximo de iterações seja atingido.

```
@staticmethod
def newton_raphson(f, x0, tol=1e-6, max_iter=100):
    iterations = 0
    while abs(f(x0)) > tol and iterations < max_iter
        :
        df = RootFinderMethods.numerical_derivative(
            f, x0)
        if abs(df) < 1e-10:
            raise ValueError(
                "Derivative is too close to zero.
                    Choose a better interval or
                    initial guess."
            )
        x0 = x0 - f(x0) / df
        iterations += 1
    return x0, iterations
```

c.3) Falsa Posição

A função `false_position` segue a mesma estrutura da bisseção, mas em vez de dividir o intervalo ao meio, calcula um novo ponto c através de interpolação linear:

$$c = b - \frac{f(b) \cdot (b - a)}{f(b) - f(a)}$$

Esse novo valor é usado para reduzir o intervalo da mesma forma que na bisseção.

```
@staticmethod
def false_position(f, a, b, tol=1e-6, max_iter=100):
    if f(a) * f(b) >= 0:
        raise ValueError("The function must have
            opposite signs at the endpoints.")
    iterations = 0
    c = a
    while abs(b - a) > tol and iterations < max_iter
        :
        c = b - f(b) * (b - a) / (f(b) - f(a))
        if abs(f(c)) < tol:
            break
        if f(c) * f(a) < 0:
            b = c
        else:
            a = c
        iterations += 1
    return c, iterations
```

c.4) Ponto Fixo

A função `fixed_point` recebe como entrada a função iterativa $g(x)$, um chute inicial x_0, uma tolerância e um número máximo de iterações. A cada iteração, a função $g(x)$ é aplicada para gerar um novo valor até que a diferença entre duas iterações consecutivas seja menor do que a tolerância especificada.

```
@staticmethod
def fixed_point(_, g, x0, tol=1e-6, max_iter=100):
    iterations = 0
    x1 = x0
    while iterations < max_iter:
        x1 = g(x0)  # Aplica a fun  o iterativa g(
            x)
        if abs(x1 - x0) < tol:
            return x1, iterations
        x0 = x1
        iterations += 1
    raise ValueError("Fixed-point iteration did not
        converge.")
```

O código segue os seguintes passos:

- A variável `iterations` inicia em zero e serve como contador de iterações.
- O loop `while` continua até que o número máximo de iterações seja atingido.
- A cada iteração, `x1` recebe o valor gerado por `g(x0)`, atualizando a estimativa da raiz.
- O critério de parada verifica se $|x_1 - x_0|$ é menor do que a tolerância `tol`, indicando convergência.
- Caso o critério de parada seja atendido, a função retorna o valor de `x1` e o número de iterações realizadas.

- Se o método não convergir dentro do limite de iterações, a função lança um erro informando que a iteração falhou.

c.5) Secante

A função `secant` aproxima a derivada de $f(x)$ através da equação da reta que passa pelos pontos $(x_{n-1}, f(x_{n-1}))$ e $(x_n, f(x_n))$:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

Dessa forma, a secante pode ser vista como uma versão do método de Newton-Raphson que evita a necessidade de calcular a derivada. O algoritmo é repetido até que o critério de convergência seja atendido.

```
@staticmethod
def secant(f, a, b, tol=1e-6, max_iter=100):
    iterations = 0
    c = b
    while abs(b - a) > tol and iterations < max_iter
        :
        c = b - f(b) * (b - a) / (f(b) - f(a))
        if abs(f(c)) < tol:
            break
        a = b
        b = c
        iterations += 1
    return c, iterations
```

c.6) Resumo

Cada método implementado apresenta características distintas em relação à convergência e eficiência. As funções são organizadas dentro da classe `RootFinderMethods` e podem ser facilmente chamadas a partir do restante da aplicação.

D. Aplicativo Final

O código principal está no arquivo `app.py`, que inicializa a interface gráfica e permite a execução dos métodos. A Figura 1 apresenta uma visualização do aplicativo em funcionamento.

Fig. 1. Interface do aplicativo para comparação dos métodos numéricos.

IV. Resultados

Nesta seção, serão apresentados os resultados dos métodos numéricos aplicados à função $\mathbf{sin(x) + cos(x) + 1}$ com uma tolerância de **0,000001**.

A. *Resultados Gerados Por Cada Método*

*a.1) **Método da Bisseção***

Através da utilização do método da bisseção, partindo dos valores iniciais $[1.1, 3.9]$, chegamos à raiz 3,141592, gerando o seguinte gráfico:

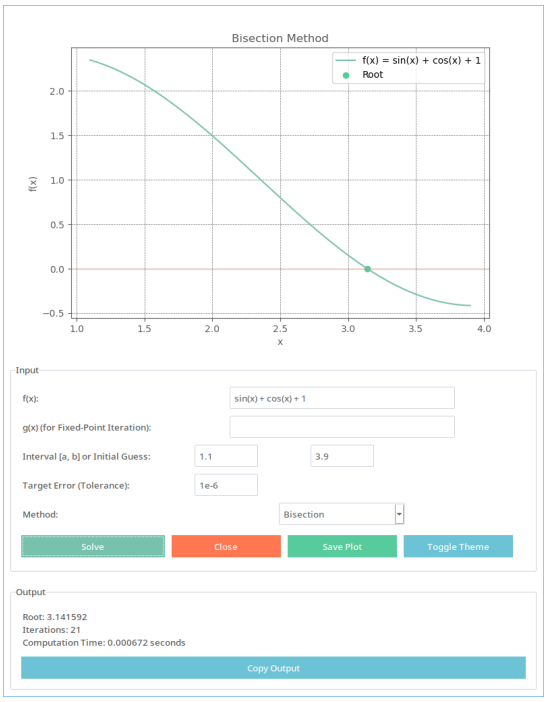

Fig. 2. Gráfico da bisseção $[1.1, 3.9]$.

a.2) Método de Newton-Raphson

Utilizando o método de Newton-Raphson com uma estimativa inicial de $[1.1, 3.9]$, encontramos a raiz 3,141593 em 8 iterações. O gráfico abaixo ilustra a convergência do método:

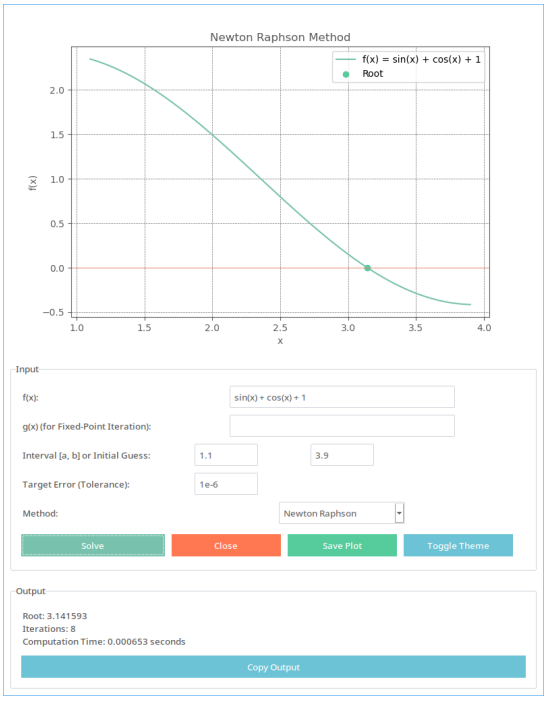

Fig. 3. Gráfico newton-raphson $[1.1, 3.9]$.

a.3) Método da Falsa Posição

Com o método da falsa posição, partindo do intervalo $[1.1, 3.9]$, obtivemos a raiz 3,141593 em 7 iterações. O gráfico a seguir mostra o comportamento do método:

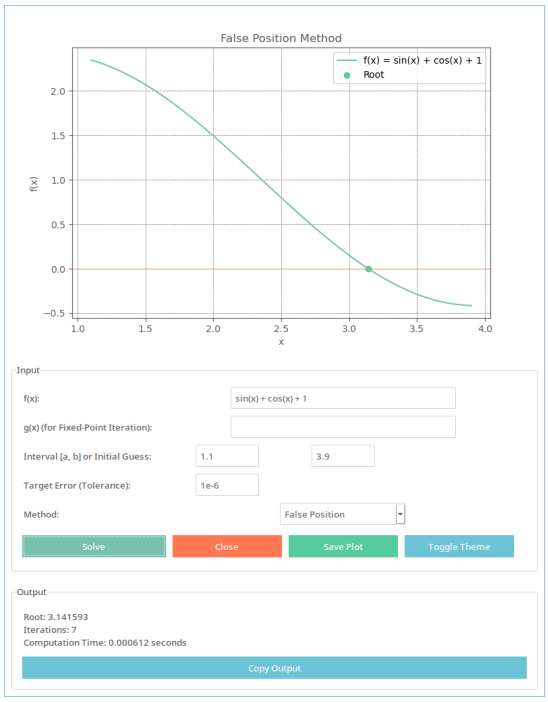

Fig. 4. Gráfico da falsa posição $[1.1, 3.9]$.

a.4) Método do Ponto Fixo

Aplicando o método do ponto fixo com a função $g(x) = x - 0.5(sin(x) + cos(x) + 1)$ e uma estimativa inicial de 1.1, o método convergiu para a raiz 1,570796 em 20 iterações. O gráfico abaixo ilustra a convergência:

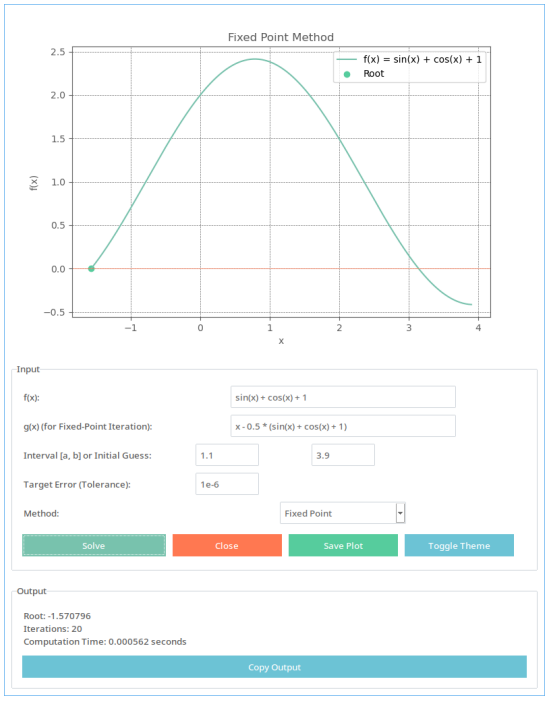

Fig. 5. Gráfico do ponto fixo $[1.1, 3.9]$.

a.5) Método da Secante

Por fim, utilizando o método da secante com as estimativas iniciais 1.1 e 3.9, encontramos a raiz 3,141593 em 6 iterações.

O gráfico a seguir mostra a convergência do método:

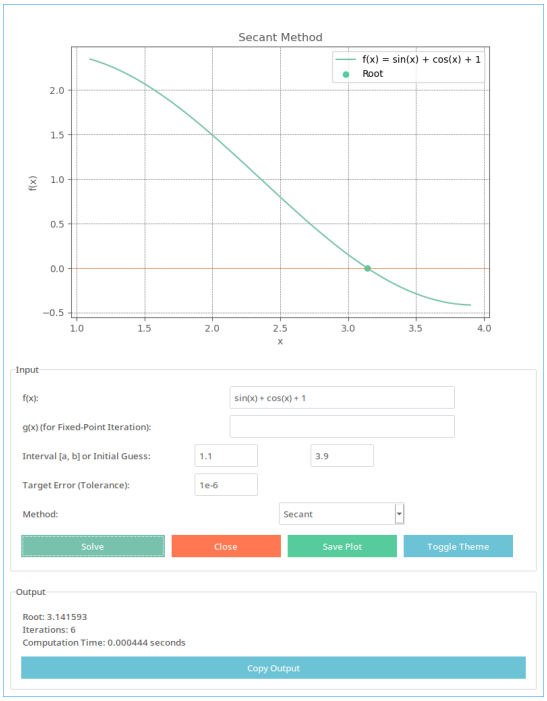

Fig. 6. Gráfico da secante $[1.1, 3.9]$.

B. *Análise e Comparação*

Dessa forma, é realizada uma análise comparativa dos métodos numéricos aplicados à função $f(x) = sin(x) + cos(x) + 1$, considerando o tempo de computação, o número de iterações e o erro relativo. O gráfico resume os dados obtidos.[2] [1].

Observando de forma geral, é notório como o modo de Newton-Raphson definitivamente é um dos métodos mais eficientes, devido a seu tempo de execução satisfatório e baixo erro relativo. Diferentemente do método da bisseção, que gastou muito tempo de execução e um número muito alto de iterações.[6] [4].

C. *Limitações*

Embora os métodos numéricos sejam ferramentas poderosas para encontrar raízes de funções, cada um deles possui limitações que devem ser consideradas ao escolher a abordagem mais adequada para um problema específico. Alguns métodos exigem um número muito elevado de iterações para atingir a precisão desejada (bisseção), enquanto outros necessitam de valores iniciais adequados para garantir a convergência (Newton e secante). Além disso, funções altamente não lineares podem ser um empecilho para determinados métodos (posição falsa e bisseção), e uma má escolha da função $g(x)$ pode ser um grande problema para o método do ponto fixo.[7] [8].

Assim, a junção de todas essas limitações acaba restringindo o desempenho do software, especialmente quando utilizado por alguém que não domina os conhecimentos necessários de cálculo numérico, o que pode resultar em resultados imprecisos.[9] [3].

V. Conclusão

Através dos resultados apresentados nesta seção, foi possível comparar a eficiência e a precisão dos métodos numéricos estudados, considerando o *tempo de computação*, o *número de iterações* e o *erro relativo médio*. A fundamentação teórica para esses métodos pode ser encontrada em [3] e [2], que abordam os princípios matemáticos subjacentes.

Em termos de tempo de computação, o método de Ponto Fixo foi o mais eficiente, com um tempo médio de 0.000501 segundos, seguido de perto pelo método Secante, com 0.000496 segundos. Por outro lado, o método Bisseção apresentou o maior tempo médio de execução, com 0.001047 segundos, embora ainda dentro de limites aceitáveis para a maioria das aplicações. Esses tempos são ilustrados no gráfico de *Tempo de Computação Médio* mostrado na Figura 7.

Quanto ao número de iterações, o método de Ponto Fixo também apresentou um número elevado de iterações (20), sendo superado apenas pelo método Bisseção, que alcançou 21 iterações. Já os métodos Secante e Newton-Raphson foram os mais rápidos, com uma média de 6 e 8 iterações, respectivamente. Esse desempenho pode ser observado no gráfico de *Número de Iterações* da Figura 7.[10].

Em relação ao erro relativo, os métodos Newton-Raphson e Secante se destacaram pela precisão, com erros relativos muito baixos ($5.747333e-12$ e $4.001674e-08$, respectivamente). Em contrapartida, o método de Ponto Fixo apresentou um erro relativamente alto de $1.500000e+00$, indicando que, apesar de ser eficiente em termos de tempo, pode não ser adequado para problemas que exijam alta precisão. O gráfico de *Erro Relativo* na Figura 7 ilustra claramente essas diferenças.

Portanto, pode-se concluir que os métodos Secante e Newton-Raphson são, em geral, os mais eficientes e precisos, especialmente para problemas que exigem alta precisão em um número reduzido de iterações. No entanto, a escolha do método ideal depende das características do problema em questão, como a precisão desejada e os recursos computacionais disponíveis.[9] [6].

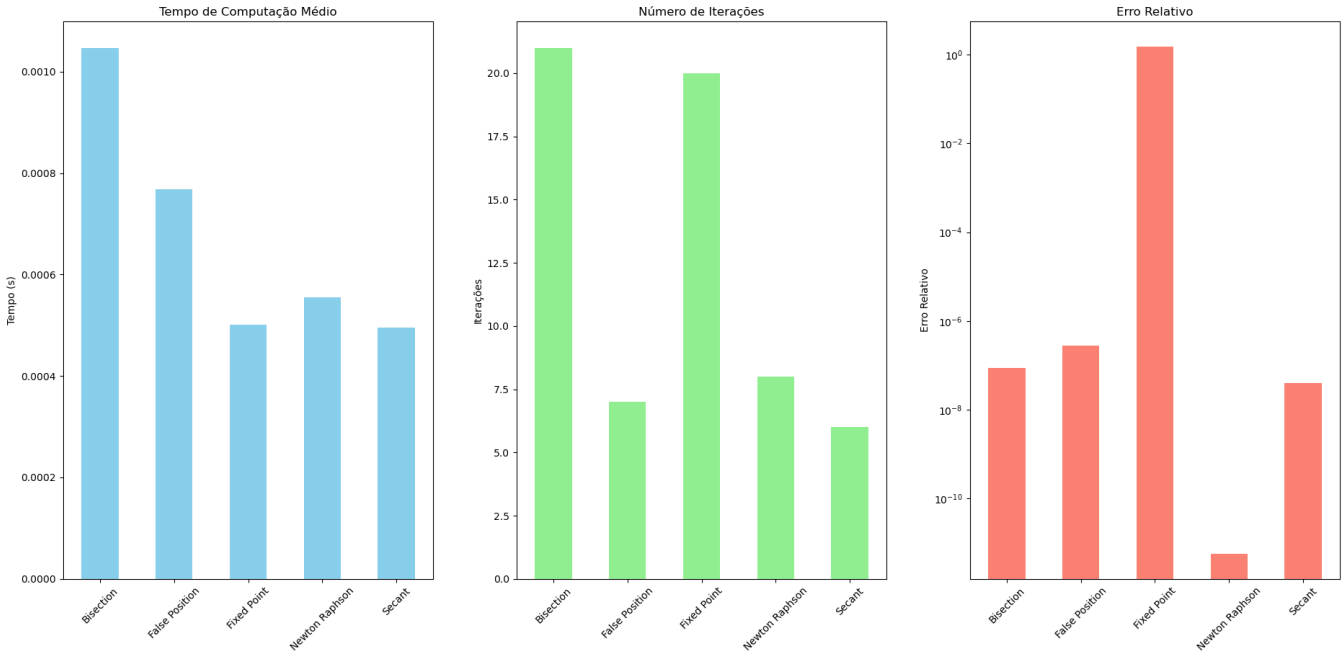

Fig. 7. Gráfico comparativo [1.1, 3.9].

REFERENCES

[1] C. H. Asano and E. Colli. *Cálculo Numérico: Fundamentos e Aplicações*. Departamento de Matemática Aplicada, IME-USP, 2009.

[2] M. A. G. Ruggiero and V. L. da R. Lopes. *Cálculo Numérico: Aspectos teóricos e computacionais*. 2nd ed. São Paulo: Pearson, 1996.

[3] R. G. Bartle. *Elementos de Análise Real*. Rio de Janeiro: Editora Campus LTDA, 1983.

[4] C. N. Moreira and M. A. P. Cabral. *Curso de Análise Real*. Departamento de Matemática Aplicada, IM-UFRJ, 2011.

[5] R. G. Bartle and D. R. Sherbert. *Introduction to Real Analysis*. 4th ed. John Wiley & Sons, 2010.

[6] R. De Quadros and A. L. De Bortoli. *Fundamentos de Cálculo Numérico para Engenheiros*. Porto Alegre: Universidade Federal de Pelotas, 2009.

[7] Jussara Maria Marins. "Localização de Zeros Reais de Polinômios Intervalares". Orientador: Prof. Dalcidio Moraes Claudio. PhD thesis. Porto Alegre: Universidade Federal do Rio Grande do Sul, 1996.

[8] Diomar Cesar Lobão. *Introdução aos Métodos Numéricos*. Trabalho original preparado por: Prof. Ionildo José Sanches e Prof. Diógenes Lago Furlan. Acesso em: [data de acesso]. Volta Redonda, RJ: Universidade Federal Fluminense, [s.d.] URL: http://www.professores.uff.br/diomar_cesar_lobao.

[9] H. Anton, I. Bivens, and S. Davis. *Cálculo*. 10th ed. Porto Alegre: Bookman, 2014.

[10] W. Rudin. *Principles of Mathematical Analysis*. 3rd ed. McGraw-HILL International Book Company, 1976.