Análise Comparativa de Erros em Métodos Computacionais para Cálculo de e^x e Raiz Quadrada

Eduardo Emílio dos Santos[1], **Vitor Felipe de Souza Siqueira**[1]

[1]Departamento de Informática (DIN) – Universidade Estadual de Maringá (UEM)

ra124501@uem.br, ra122907@uem.br

Abstract. *In this report, we explore computational mathematics concepts focusing on accuracy, precision, and error in the context of square root and exponential function approximations. We implement and analyze three algorithms: a square root approximation method, Bailey's method for calculating e^x, and the Nice Number method for e^x calculation. Our objective is to evaluate each algorithm's performance in terms of mathematical accuracy and computational efficiency, by analyzing the error margins through graphical representations.*

Resumo. *Neste relatório, exploramos conceitos de matemática computacional com foco em acurácia, precisão e erro no contexto de aproximações da raiz quadrada e da função exponencial. Implementamos e analisamos três algoritmos: um método de aproximação da raiz quadrada, o método de Bailey para cálculo de e^x, e o método Nice Numbers para cálculo de e^x. Nosso objetivo é avaliar o desempenho de cada algoritmo em termos de precisão matemática e eficiência computacional, analisando as margens de erro por meio de representações gráficas.*

1. Introdução

A matemática computacional desempenha um papel fundamental na Ciência da Computação, especialmente no desenvolvimento de algoritmos eficientes para cálculos numéricos. Neste relatório, exploramos a análise de algoritmos para a aproximação da raiz quadrada e o cálculo de funções exponenciais, duas operações essenciais em diversas áreas da computação e discutidas em sala de aula com o Prof. Dr. Airton Marco Polidório. Comparando esses algoritmos com as funções da biblioteca Numpy, da linguagem de programação Python, investigamos sua precisão e eficiência.

2. Objetivo

Nosso objetivo é implementar e analisar três algoritmos distintos: um para a aproximação da raiz quadrada, o método de Bailey para e^x, e o método Nice Numbers para e^x, comparando-os com as implementações da biblioteca Numpy. Buscamos avaliar o erro associado a cada algoritmo e, consequentemente, como se comporta os conceitos de precisão e eficácia.

3. Construção de Funções Personalizadas

Para a análise dos algoritmos de aproximação, foi necessário construir funções personalizadas para operações matemáticas básicas, a fim de evitar o uso de funções de bibliotecas que poderiam introduzir suas próprias aproximações e otimizações. Abaixo estão as descrições dessas funções e suas implementações.

3.1. Função Pow

A função `pow` implementa a exponenciação por meio de uma abordagem recursiva. A exponenciação é definida matematicamente pela relação de recorrência:

$$x^y = \begin{cases} 1, & \text{se } y = 0 \\ x \cdot x^{y-1}, & \text{se } y > 0 \text{ e ímpar} \\ (x^{y/2})^2, & \text{se } y > 0 \text{ e par} \end{cases}$$

Esta função utiliza divisão e conquista para dividir o problema em subproblemas menores, minimizando o número total de multiplicações necessárias.

```
def pow(x: float, y: int):
    if y == 0:
        return 1
    elif y == 1:
        return x
    elif y < 0:
        return 1 / pow(x, -y)
    elif y % 2 == 0:
        half_pow = pow(x, y // 2)
        return half_pow * half_pow
    else:
        return x * pow(x, y - 1)
```

3.2. Função Log Base 2

A função `log_base_2` calcula o logaritmo de um número na base 2. A implementação se baseia na ideia de que o logaritmo na base 2 é o número de vezes que um número pode ser dividido por 2 antes de chegar a 1. Para valores menores que 1, o processo é invertido, e o expoente é negativo.

```
def log_base_2(x):
    exponent = 0
    if x >= 1:
        while x >= 2:
            x /= 2
            exponent += 1
    else:
        while x < 1:
            x *= 2
            exponent -= 1
    return exponent
```

3.3. Método de Horner para Exponenciação Fracionária

O método de Horner é uma técnica numérica eficiente para avaliação de polinômios e foi adaptado neste trabalho para aproximar a exponenciação fracionária, mais especificamente, a exponencial de um número r que surge no cálculo de e^x pelo método de Bailey.

A série de Taylor para a exponencial é reescrita utilizando o método de Horner para otimizar as operações de multiplicação.

A implementação de Horner para exponenciação fracionária é dada por:

```
def horner(x):
    result = 1
    for i in range(10, 0, -1):
        result = 1 + x * result / i
    return result
```

Esta função é utilizada para calcular e^r onde r é o valor fracionário obtido após a redução de x no método de Bailey. Ao invés de calcular e^r diretamente, que poderia resultar em uma profunda recursão com a função `pow`, utilizamos o método de Horner para uma aproximação eficiente e robusta. A função `horner` é chamada com o valor de r e um grau predeterminado que define a precisão da aproximação do polinômio.

Cada uma dessas funções foi projetada para garantir que os algoritmos de aproximação pudessem ser avaliados sem influência externa.

4. Metodologia

A metodologia analisa detalhadamente os algoritmos para aproximação da raiz quadrada e cálculo de e^x, destacando os fundamentos matemáticos e suas implementações em Python. Além disso, algumas constantes como $\ln(2)$, $\sqrt{2}$ e e foram introduzidas como constantes no começo do projeto.

4.1. Método para Aproximação da Raiz Quadrada

Este método envolve várias etapas matemáticas para decompor o número x e calcular sua raiz quadrada aproximada.

4.1.1. Decomposição de x em e e f

Inicialmente, o número x é decomposto em duas partes distintas, e e f, onde:

- $e = \lfloor \log_2(x) \rfloor$, representando a maior potência de 2 não superior a x.
- $f = \frac{x}{2^e} - 1$, a parte fracionária após subtrair a base da potência de 2 mais significativa de x.

A implementação dessa decomposição é realizada pela função `calcula_e_f(x)`:

```
def calcula_e_f(x):
    e = log_base_2(x)
    f = x / pow(2, e) - 1
    return e, f
```

4.1.2. Cálculo de $\sqrt{2^e}$

Dependendo do valor de e, a raiz quadrada de 2^e é calculada diferentemente:

- Para $e = 0$, $\sqrt{2^e} = 1$.
- Para $e = 1$, $\sqrt{2^e} = \sqrt{2}$.
- Para e par, $\sqrt{2^e} = 2^{e/2}$.
- Para e ímpar, $\sqrt{2^e} = 2^{(e-1)/2} \cdot \sqrt{2}$.

A função `sqrt_2e(e)` realiza este cálculo:

```
def sqrt_2e(e):
    if e == 0:
        return 1
    elif e == 1:
        return sqrt2
    elif e % 2 == 0:
        return pow(2, e // 2)
    else:
        return pow(2, (e - 1) // 2) * sqrt2
```

4.1.3. Cálculo da Raiz Quadrada Aproximada de x

A raiz quadrada aproximada de x é então dada por:

$$\sqrt{x} \approx \sqrt{2^e} \cdot (1 + f/2)$$

Essa aproximação combina os resultados anteriores para obter uma estimativa da raiz quadrada de x. O algoritmo que propõe essa solução é:

```
def raiz_calculada(x):
    e, f = calcula_e_f(x)
    sqrt_2e_val = sqrt_2e(e)
    return sqrt_2e_val * (1 + f / 2)
```

4.2. Algoritmo Bailey para e^x

O algoritmo de Bailey para o cálculo de e^x envolve a redução do argumento x para aproximar e^x com precisão.

4.2.1. Redução de x e Cálculo de e^x

A abordagem inicia com a redução de x usando técnicas aditivas e multiplicativas para simplificar o cálculo de e^x. A redução é seguida pelo cálculo de e^x com base nos valores reduzidos de x.

A implementação deste algoritmo é exemplificada pela função `bailey_e_x(x)`:

```
def bailey_e_x(x):
    n = np.ceil((x - ln2 / 2) / ln2)
    r = (x - n * ln2) / 256
    e_elevado_r = horner(r)
    return pow(2, n) * pow(e_elevado_r, 256)
```

4.3. Algoritmo Nice Number para e^x

O algoritmo Nice Number aproveita uma Tabela de Consulta (LUT) para otimizar o cálculo de e^x, realizando iterações baseadas em valores pré-definidos e^x.

4.3.1. Construção da LUT e Processo Iterativo

A LUT contém valores precalculados de e^k para diferentes potências de 2, facilitando o cálculo iterativo de e^x até a convergência. O intervalo que vai até -53 foi calculado e é o limite que a linguagem Python conseguiu realizar as operações da LUT com precisão.

```
def criar_LUT():
    lut = {2**(-i): np.exp(2**(-i)) for i in range(53)}
    return lut
```

Este processo é implementado pela função `calculo_ex_usando_lut(x, lut):`

```
def calculo_ex_usando_lut(x, lut):
    y = 1
    while x > 0:
        keys = [k for k in lut.keys() if k <= x]
        if keys:
            max_k = max(keys)
            x -= max_k
            y *= lut[max_k]
        else:
            break
    if x < min(lut.keys()):
        y *= (1 + x)

    return y
```

5. Resultados

Os gráficos a seguir ilustram a comparação dos erros entre os algoritmos implementados e as operações realizadas pela biblioteca Numpy. A análise foca na diferença de precisão, demonstrando como cada método se comporta em relação às funções otimizadas da Numpy para cálculos de raiz quadrada e exponenciação.

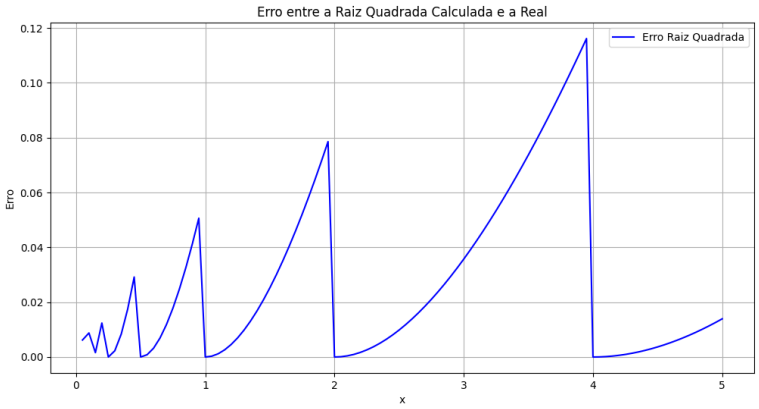

Figura 1. Comparação do erro na aproximação da raiz quadrada.

O erro na aproximação da raiz quadrada aumenta com o valor de x, o que é consistente com as limitações da aproximação linear da série de Taylor em torno de 1 para $\sqrt{1+f}$. Esse aumento é mais significativo para valores maiores de x, onde a parcela fracionária f se torna maior, e a aproximação linear se desvia mais do valor real. A função `sqrt_2e` calcula a raiz quadrada de 2^e de maneira exata quando e é par e uma aproximação quando e é ímpar.

Figura 2. Erro no cálculo de e^x pelo método de Bailey.

Para o cálculo de e^x usando o método de Bailey, o algoritmo é muito preciso para a maioria do intervalo, exceto nos extremos onde os picos de erro são notáveis. Esses picos podem ser atribuídos à precisão finita e aos erros de arredondamento que se acumulam durante o cálculo, especialmente ao lidar com potências muito altas. O uso da função `horner` para calcular e^r ajuda a minimizar esses erros, pois é uma técnica eficiente para avaliar polinômios, reduzindo o número de operações e potencialmente os erros de arredondamento.

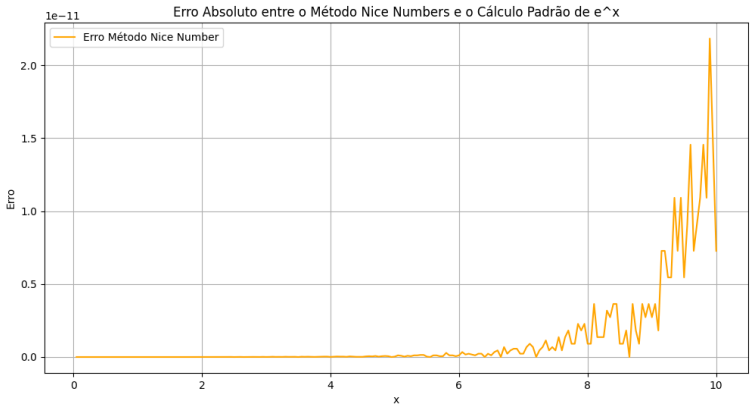

Figura 3. Erro no cálculo de e^x pelo método Nice Numbers.

O erro associado ao método Nice Number é baixo para valores menores de x, e cresce com o aumento de x. Isso é esperado, pois a LUT fornece uma cobertura adequada para valores menores, mas não possui entradas suficientes para valores maiores, o que torna a aproximação menos precisa. À medida que x aumenta, a chance de um valor específico de x não corresponder exatamente a uma entrada na LUT aumenta, resultando em erros maiores.

6. Conclusão

Neste trabalho, exploramos e implementamos três algoritmos distintos para a aproximação da raiz quadrada e o cálculo da função exponencial e^x, evidenciando suas peculiaridades e aplicabilidades. A análise comparativa com as funções da biblioteca Numpy permitiu avaliar a precisão e eficiência computacional desses métodos sob diferentes condições.

O método de aproximação da raiz quadrada demonstrou ser eficaz para valores menores de x, com limitações na precisão para valores maiores. O método de Bailey para o cálculo de e^x se destacou pela sua alta precisão em grande parte do intervalo considerado, ressaltando a eficiência do método de Horner na minimização dos erros de arredondamento e na otimização das operações matemáticas. Por sua vez, o método Nice Number, através do uso de uma Tabela de Consulta, forneceu resultados precisos para valores menores de x, evidenciando a importância de estratégias eficientes de pré-cálculo e acesso rápido a dados.

Concluímos que a seleção de um algoritmo deve considerar não somente a precisão desejada, mas também as características específicas do problema, como o intervalo de valores a serem calculados e os recursos computacionais disponíveis. Este estudo contribui para o entendimento dos métodos computacionais aplicados à matemática, destacando a relevância de técnicas otimizadas para o cálculo numérico em variadas aplicações da Ciência da Computação.

Referências

[1] Airton Marco Polidório. *Cálculo da Raiz Quadrada*. Notas de aula para a disciplina de Matemática Computacional. Universidade Estadual de Maringá. 2022.

[2] Airton Marco Polidório. *Ponto Flutuante*. Notas de aula para a disciplina de Matemática Computacional. Universidade Estadual de Maringá. 2022.
[3] Airton Marco Polidório. *A Representação Interna dos Números*. Notas de aula para a disciplina de Matemática Computacional. Universidade Estadual de Maringá. 2022.
[4] Airton Marco Polidório. z=x^y. Notas de aula para a disciplina de Matemática Computacional. Universidade Estadual de Maringá. 2022.
[5] Airton Marco Polidório. z=$x^y XPadrãoIEE-754$. Notas de aula para a disciplina de Matemática Computacional. Universidade Estadual de Maringá. 2022.
[6] Airton Marco Polidório. *Funções Elementares Transcendentais*. Notas de aula para a disciplina de Matemática Computacional. Universidade Estadual de Maringá. 2022.
[7] José Carlos Polidório. "Arredondamento para o Par". Em: *Revista Brasileira de Computação* 15.4 (dez de 2000). Discussão sobre métodos de arredondamento em sistemas de ponto flutuante, pp. 34–45.