Relatório de Matemática Computacional

Cálculo de erros envolvendo raiz quadrada;
Cálculo de erros de exponenciação pelo método de *Bailey;*
Cálculo de erros de exponenciação pelo LUT(Look Up Table);

Autores:
Guilherme Frare Clemente RA:124349
Marcos Vinicius de Oliveira RA:134408

Matemática Computacional - 6900/01 - 2024
Universidade Estadual de Maringá - PR

Introdução

A matemática computacional desempenha um papel fundamental na resolução de uma ampla gama de problemas em diversas áreas, desde engenharia e ciência da computação até finanças e física. Uma parte essencial dessa disciplina é a análise dos erros inerentes às operações numéricas, uma vez que os computadores representam números em formato finito e realizam cálculos aproximados.

Neste contexto, o presente relatório tem como objetivo investigar e analisar os erros associados a três problemas comuns de matemática computacional: cálculo da raiz quadrada, exponenciação pelo método de ***Bailey*** e exponenciação pelo método ***LUT (Look Up Table)***. Cada um desses problemas apresenta desafios distintos e requer abordagens específicas para lidar com os erros numéricos.

Ao longo deste relatório, serão apresentadas as formulações matemáticas dos erros associados a cada problema, os algoritmos implementados em Python para resolver esses problemas e uma análise detalhada dos resultados obtidos. Por meio da análise dos erros em diferentes cenários e intervalos de valores de entrada, buscamos compreender melhor as limitações dos métodos computacionais empregados e identificar estratégias para melhorar a precisão e a eficiência desses métodos.

Descrição do problema

- **Cálculo de erros na raiz quadrada:** O cálculo da raiz quadrada de um número envolve estimar a raiz quadrada de um valor *x*. Neste problema, investigamos o erro associado entre a raiz quadrada calculada e a raiz quadrada exata fornecida pela biblioteca `math.sqrt`. Porém, é necessário entender como é calculada essa raiz quadrada:

Primeiramente, tendo como início $\sqrt{x}$:

Temos que x é calculado da seguinte maneira

$$x = (1 + f) * 2^k$$

A partir disso, aplicamos raiz em ambos os lados:

$$\sqrt{x} = \sqrt{(1 + f)} * \sqrt{2}^k$$

Com isso precisamos calcular ambos os valores da igualdade, para assim conseguirmos calcular o valor da $\sqrt{x}$:

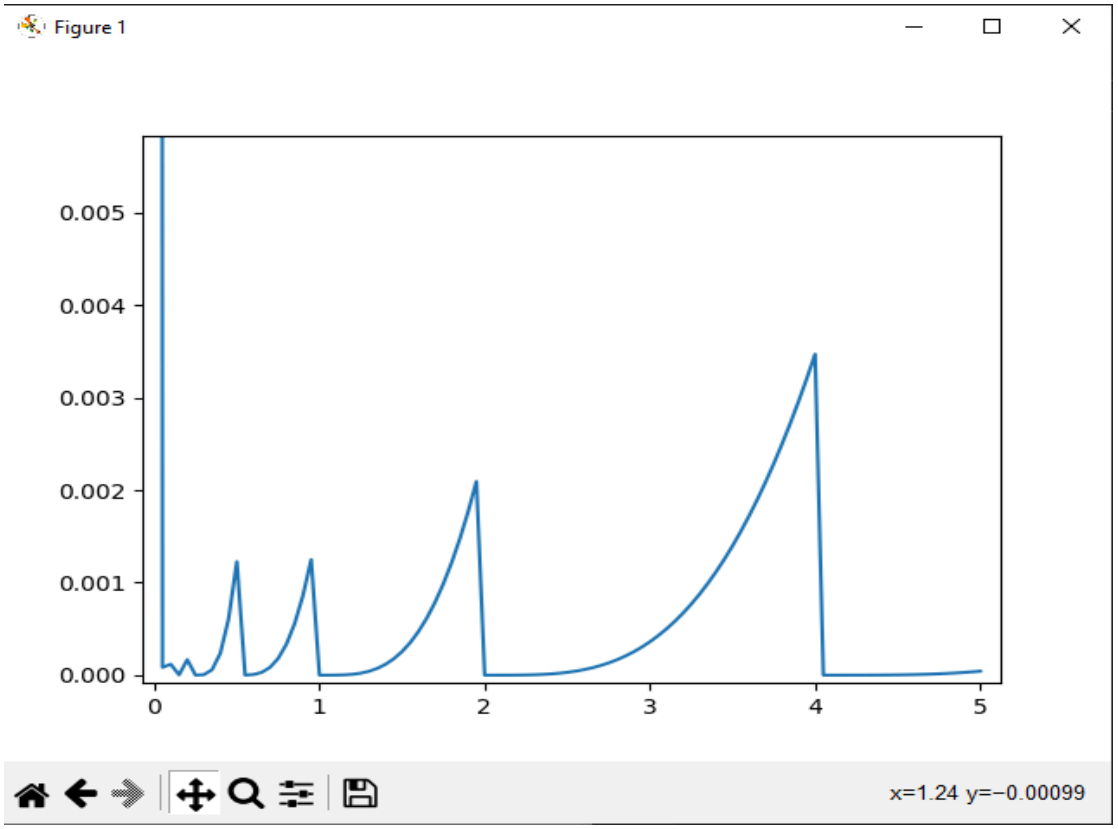
- Calcular $\sqrt{2^k}$:
  - $1/\sqrt{2^k}$, se k < 0
  - 1, se k = 0
  - $\sqrt{2}$, se k = 1
  - $2^{k/2}$, se k é par
  - $\sqrt{2} * 2^{(k-1)/2}$, se k é ímpar

- Calcular $\sqrt{(1 + f)} \cong 1 + f/2 * (1 - (1/4 + 2f))$

Após calcularmos esses dois valores, basta multiplicarmos e com isso obtemos a nossa raiz quadrada. Lembrando que quanto maior for o número utilizado, maior será o erro quando comparada com a raiz quadrada padrão da linguagem Python, o erro estará mais próximo da parte inteira à medida que o valor de x é aumentado.

Agora que temos o valor da $\sqrt{x}$. Basta compararmos com o `math.sqrt` do Python e verificar o erro (basta subtrair as 2 raízes quadradas para obtermos os erros de cada valor indicado).

Para um melhor entendimento, foi comparado utilizando um intervalo de número que vão de -5 até 5, aumentando numa taxa de 0.05, e com isso obtivemos esse gráfico como resposta



Observamos que, em geral, o erro aumenta conforme o número de entradas aumenta. Isso é esperado, pois números maiores têm uma raiz quadrada que está mais distante de ser exata e, portanto, a discrepância entre a raiz calculada e a verdadeira raiz é maior. Para números pequenos, o erro tende a ser relativamente baixo. Isso pode ser atribuído à capacidade do algoritmo de calcular com mais precisão as raízes quadradas de números menores, onde a magnitude do erro absoluto é menor. Ou seja, por exemplo:

Com $x = 2.2$

$\sqrt{x} = 1.483240652917497$

$\text{math.sqrt}(x) = 1.4832396974191326$

Podemos ver que os valores começam a ficar diferentes a partir da quinta casa decimal na parte fracionada, ou seja, um erro de valor 10^{-5}, agora para um valor grande por exemplo:

Com x = 100.2

$\sqrt{x}$ = 10.013101400730816

math.sqrt(x) = 10.00999500499375

Podemos ver que os valores começaram a ficar diferentes a partir segunda casa decimal na parte fracionada, ou seja, o erro foi bem maior (10^{-2}) quando comparado com o x = 2.2, o que faz sentido sobre tudo que foi comentado, quanto maior o valor de x, maior o erro

Por fim, é importante entender que o algoritmo implementado para calcular a raiz quadrada utiliza o método de aproximação de Newton, combinado com manipulações de ponto flutuante para lidar com números grandes. O processo é dividido em dois passos: calcular a raiz da mantissa normalizada e ajustar para o expoente.

- **Cálculo de erros na exponenciação pelo método de *Bailey*:** Neste problema, investigamos o erro associado à exponenciação de um número usando o método de ***Bailey***, que utiliza uma fórmula específica para calcular a parte fracionária de uma exponenciação *(float)* com precisão. Para esse método, temos algumas fórmulas para representar o e^x:

  Temos que $e^x = 2^n * (e^r)^{256}$.

Ou seja, para obtermos o valor do e^x é necessário encontrar o valor de n e e^r, para isso, ***Bailey*** fornece o seguinte suporte:

- Calcular o n:

  $n = ceil * ((x / ln(2)) - 1/2)$

  Onde **Ceil** é uma função da biblioteca **math** do Python.

- Calcular o r:

  $r = (x - n * ln(2)) / 256$

- Calcular e^r:

  $e^r = 1 + r * (1 + r * (1/2 + r * (1/6 + 1/24 * r)))$

Após esses cálculos, precisamos entender as situações que o 2^n pode adotar:

Para 2^n, **se** $n >= 0$, será feito (na linguagem Python) um shift de bit para a esquerda, representado por $1 << n$.

Para 2^n, **se** n < 0, o shift de bit ocorrerá de uma forma diferente, representado por $1/(1 << -n)$

Após o cálculo, obtivemos o valor da parte fracionária da exponencial, pois o método de ***Bailey*** só se aplica para a parte fracionária de uma exponencial, para a parte inteira é realizado um algoritmo recursivo (chamado também de algoritmo recursivo de exponenciação inteira). Por exemplo:

$$e^{20.8} = e^{20} * e^{0.8}$$

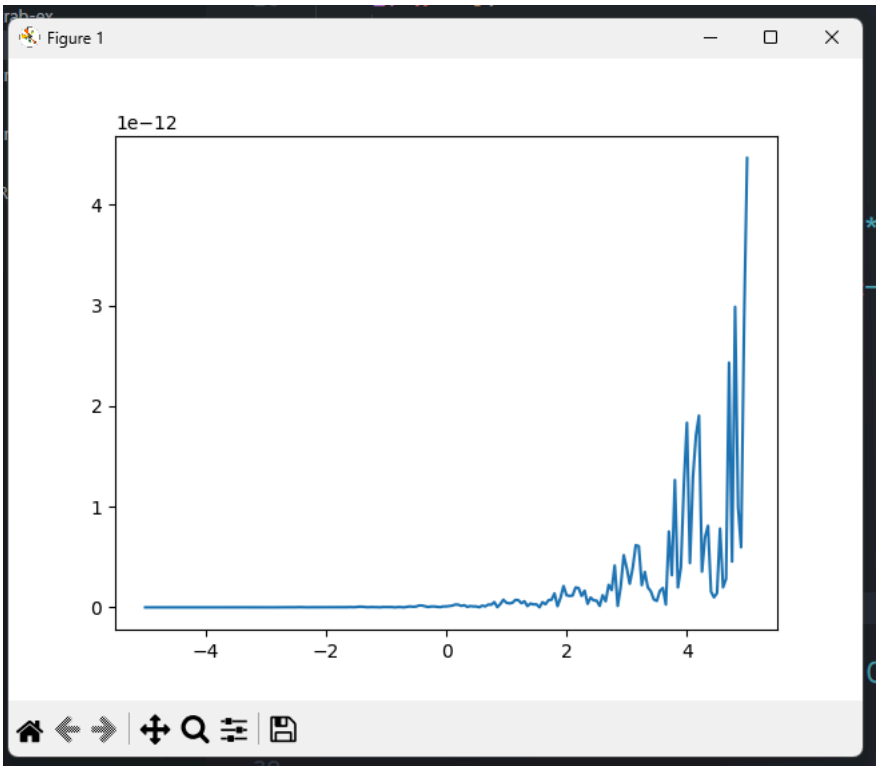
Ou seja, e^{20} será resolvido pelo algoritmo recursivo que será mostrado abaixo, enquanto $e^{0.8}$ é resolvido pelo método de ***Bailey*** que foi mostrado o passo a passo anteriormente, agora abaixo é destacado o algoritmo recursivo:

Para x^k, temos:

- $1/x^{-k}$, se k < 0
- 1, se k = 0
- x, se k = 1
- $(x^{k/2})^2$, se k é par
- $x^{k-1} * x$, se k é ímpar

Após o cálculo da parte inteira, basta multiplicarmos a exponencial obtida pelo algoritmo recursivo e o outro obtido pelo método de ***Bailey***, e assim teremos o valor do e^x.

Com isso, foi feito, em Python, comparações entre a exponencial obtida por ***Bailey*** e a exponencial básica fornecida pelo python (`math.exp(x)`). Foram testados intervalos de número entre -5 a 5, avançando com uma taxa de 0.05, e com isso foi gerado um gráfico com os erros obtidos (o erro é calculado subtraindo a exponencial de ***Bailey*** pela exponencial fornecida pelo Python):



Com esse gráfico, conforme o valor de x aumenta, o erro também tende a aumentar. Isso é evidenciado pelos erros ficando gradualmente maiores à medida que x aumenta, conforme observado nas últimas linhas da saída.

Para alguns valores específicos de x, como 0 e 1, o erro é reportado como zero. Isso ocorre porque o cálculo da exponenciação nesses casos é trivial e o método de Bailey é capaz de reproduzir exatamente o valor da função exponencial nativa.

Para valores negativos de x, especialmente quando x se torna muito negativo, os erros tendem a crescer. Isso é comum em métodos de exponenciação, onde valores muito grandes ou muito pequenos podem resultar em perda de precisão devido a limitações numéricas.

Em resumo, os resultados indicam que o método de Bailey é eficaz para calcular a exponenciação em uma ampla gama de valores de x, produzindo resultados com erros geralmente pequenos e aceitáveis em comparação com a função exponencial nativa do Python.

Por fim, é importante entender que o algoritmo implementado calcula a exponenciação dividindo o valor de x em sua parte inteira e fracionária. A parte inteira é tratada pela recursão exponencial convencional, enquanto a parte fracionária é utilizada para calcular os coeficientes de acordo com a fórmula de ***Bailey.***

- **Cálculo de erros na exponenciação pelo método *LUT(Look Up Table):*** Neste problema, investigamos o erro associado à exponenciação de um número usando o método ***LUT***, que consiste em utilizar uma tabela pré definida para valores de exponenciação com base em potências de 2.

  Para esse problema, é preciso primeiramente calcular a tabela ***LUT***, abaixo é apresentado a tabela que foi utilizada para a realização de testes (a tabela também está presente no algoritmo exTabela.py):

Tabela *LUT*

Valores	e^k	$k = ln(e^k)$
2^8	256	5.5452
2^4	16	2.7726
2^2	4	1.3863
2^1	2	0.6931
$2^{-1} + 1$	$3/2$	0.4055
$2^{-2} + 1$	$5/4$	0.2231
$2^{-3} + 1$	$9/8$	0.1178
$2^{-4} + 1$	$17/16$	0.0606
$2^{-5} + 1$	$33/32$	0.0308
$2^{-6} + 1$	$65/64$	0.0155
$2^{-7} + 1$	$129/128$	0.0078

A partir dessa tabela é aplicado um pseudocódigo (presente também no exTabela.py) para calcular o e^x apenas para a parte fracionária, pois, da mesma forma que o método de ***Bailey*** é utilizado apenas para a parte fracionária, o método ***LUT*** funciona da mesma maneira, ou seja, a parte inteira da exponencial será calculada pelo algoritmo recursivo de exponenciação inteira.

Algoritmo (pseudocódigo):

1. Entrada: x, LUT.
2. Definir $j = 1,\ x1 = x,\ y1 = 1$.
3. Pesquisar no LUT o maior valor $k(kmáx)$ de tal forma que $xj - kmáx >= 0$ e faça : $xj + 1 = xj - max(kLUT)$, onde $k <= xj$.
4. Multiplicar o valor yj pelo valor e^x correspondente e fazer:
$$yj + 1 = yj * e^x.$$
5. $j = j + 1$.
6. Retornar ao passo 3 até terminar a tabela ou $x = 0$, ao terminar, fazer: $x = j$. Junto com a recuperação do resíduo y', ou seja: $e^x \cong y' = (1 + Xn - 1) + Yn$.

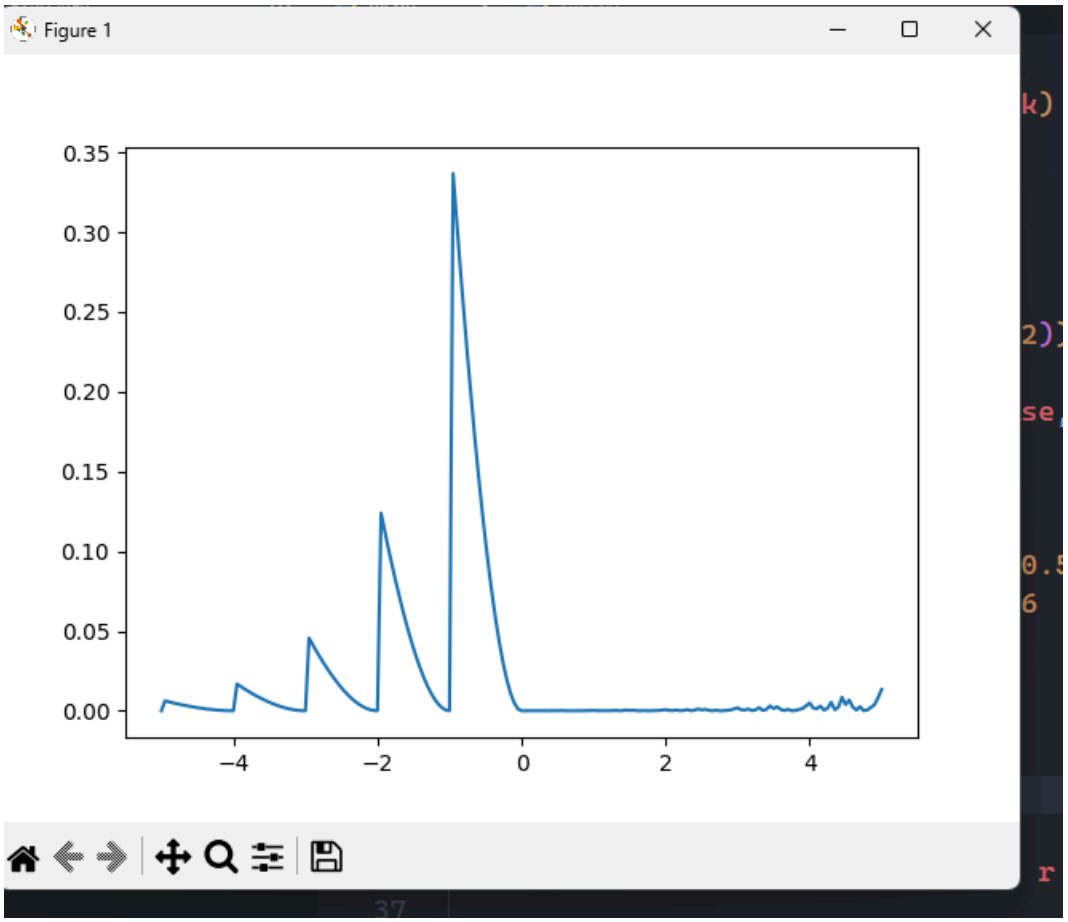
Com esse algoritmo, conseguimos calcular o valor do e^x, onde x representa apenas a parte fracionária do valor total de x. Após isso aplicamos o mesmo algoritmo recursivo para calcular a e^x da parte inteira de x, definido como:

Para x^k, temos:

- $1/x^{-k}$, se k < 0
- 1, se k = 0
- x, se k = 1
- $(x^{k/2})^2$, se k é par
- $x^{k-1} * x$, se k é ímpar

Concomitantemente, basta multiplicarmos os valores de ambos e^x (parte inteira e parte fracionária) e obtemos o valor do e^x utilizando o método ***LUT (Look Up Table).*** Dessa forma, é calculado os erros, que basta subtrair o e^xcalculado pelo método ***LUT*** com o e^x calculado pela função nativa do Python (**math.exp()**), e com isso conseguimos plotar em um gráfico os erros obtidos.

Lembrando que, da mesma forma que os outros tópicos mostrados anteriormente, problema foi investigado com intervalos de valores de entrada (de -5 a 5 com um avanço de 0.05) e os erros foram plotados em um gráfico para uma análise visual dos resultados:



Conforme mostrado no gráfico, os erros estão aumentando à medida que os números se afastam de zero em ambas as direções. Isso é esperado, já que os métodos de aproximação tendem a ser menos precisos para valores distantes do ponto de referência.Os erros são muito pequenos (em notação científica) para números próximos de zero. Isso sugere que sua implementação está funcionando bem para valores próximos de zero.

Como mencionado anteriormente, os erros são maiores à medida que os números se afastam de zero. Isso é típico em métodos de aproximação.

No geral, a implementação está produzindo resultados razoáveis, especialmente para valores próximos de zero. Por fim é importante entender que o algoritmo implementado divide o valor de x em sua parte inteira e fracionária. A parte inteira é

tratada pela recursão exponencial convencional, enquanto a parte fracionária é utilizada para buscar na tabela ***LUT*** e calcular o resultado.

Conclusão

Este relatório explorou os erros associados a três problemas comuns de matemática computacional: o cálculo da raiz quadrada, a exponenciação pelo método de Bailey e a exponenciação pelo método LUT (Look Up Table). Cada problema foi abordado com uma análise detalhada dos métodos utilizados, implementações em Python e uma investigação dos resultados obtidos.

Para o cálculo da raiz quadrada, observamos que o erro aumenta à medida que os números se afastam de zero. No entanto, para valores próximos de zero, o erro é relativamente baixo, indicando uma boa precisão. Isso é atribuído à capacidade do algoritmo de calcular com mais precisão as raízes quadradas de números menores. A implementação utiliza o método de aproximação de Newton combinado com manipulações de ponto flutuante para lidar com números grandes.

No caso da exponenciação pelo método de Bailey, constatamos que o erro também aumenta conforme os valores de entrada se distanciam de zero. O método de Bailey produz resultados com erros geralmente pequenos e aceitáveis em comparação com a função exponencial nativa do Python. O algoritmo implementado divide o valor de x em sua parte inteira e fracionária, tratando-as separadamente para calcular o resultado final.

Por fim, ao analisar a exponenciação pelo método LUT, verificamos que os erros também aumentam à medida que os números se afastam de zero. No entanto, os erros são razoáveis para valores próximos de zero e aumentam gradualmente à medida que os números se afastam. A implementação divide o valor de x em sua parte inteira e fracionária, utilizando uma tabela LUT pré-definida para calcular a exponenciação com base em potências de 2.

Em resumo, cada método apresentou suas próprias vantagens e limitações em termos de precisão e eficiência computacional. Os resultados obtidos fornecem insights valiosos sobre as estratégias para lidar com erros numéricos em operações matemáticas comuns e podem orientar futuros desenvolvimentos em matemática computacional e algoritmos numéricos.

Referências

1. **Hamming, Richard W.** ***Numerical Methods for Scientists and Engineers*****. Courier Corporation, 2012.**
2. **Press, William H., et al.** ***Numerical Recipes: The Art of Scientific Computing*****. Cambridge University Press, 2007.**
3. **Cormen, Thomas H., et al.** ***Introduction to Algorithms*****. MIT Press, 2009.**
4. ***Foundations of Computational Mathematics*** **(Revista). Springer.**
5. ***Journal of Computational and Applied Mathematics*** **(Revista). Elsevier.**