Logaritmo Natural com Tabel de busca
Matemática Computacional

Felipe Gabriel Comin Scheffel[1]

[1]Universidade Estadual de Maringá (UEM)

{ra117306}@uem.br

Abstract. *This paper aims to present an implementation of the natural logarithm estimation problem, solved using the lookup table (LUT) strategy, analyzing aspects such as accuracy and performance.*

Resumo. *Este trabalho tem como objetivo principal apresentar uma implementação do problema da estimativa do logaritmo natural de um número, resolvido utilizando a estratégia da tabela de busca (LUT), analisando aspectos como precisão e performance.*

1. Introdução

A estratégia de lookup table (LUT) é uma técnica de pré-cálculo e armazenamento em tabela dos valores de uma função matemática ou de uma operação complexa em um conjunto limitado de entradas possíveis, permitindo que se obtenha estimativas precisas para qualquer outro número através da interpolação linear entre os valores da tabela mais próximos. Como a tabela de busca deve utilizar um espaço limitado, uma possibilidade de entrada são os chamados ”nice numbers”: um conjunto de números pré-determinados que são considerados ”bonitos”ou ”atraentes”por por terem propriedades matemáticas interessantes. O uso conjunto dessas estratégias pode ser uma opção eficiente para estimar o valor do logaritmo natural de um número.

2. Metodologia

Para este trabalho, foi implementada a estimativa do logaritmo natural utilizando uma tabela de busca com ”nice numbers”pré definidos, de forma que a partir das soluções armazenadas seja possível obter novas soluções aplicando o método da interpolação linear. Também é possível comparar a implementação realizada com a operação da biblioteca padrão, a fim de verificar o erro.

2.1. Fundamentação Teórica

A interpolação linear para o logaritmo natural de x com função

$$f(x) = \ln x; x > 0$$

com forma invariante

$$y = \ln \frac{x}{x} = 0$$

Assim, para $\frac{1}{x} = \frac{1}{kx}$ na forma invariante, tem-se:

$$y = \ln \frac{x}{kx} = \ln \frac{x}{x} \cdot \frac{1}{k}$$
$$y = \ln \frac{x}{x} + \ln \frac{1}{k}$$

O que nos dá a invariante de recorrência:

$$y_{j+1} = y_j - \ln x$$

Portanto, partindo de valores conhecidos de $\ln x$, é possível realizar uma aproximação para outra solução da função.

Esses valores já conhecidos serão armazenados em uma estrutura finita chamada tabela de busca ou lookup table (LUT), que será percorrida do topo para baixo até que não existam mais elementos a serem utilizados. A tabela funciona como um dicionário, onde os valores de k os quais tem um resultado conhecido são as chaves e o resultado da função ($\ln k$) é o valor sendo indexado.

Buscando manter a precisão, a aproximação será feita sempre utilizando o argumento mais reduzido possível. Assim, o primeiro elemento será aquele imediatamente maior que x. A partir de então, serão buscados os elementos tal que $k \cdot x < 1$. Para cada elemento encontrado, o invariante terá o valor do resultado ($\ln k$) subtraído, promovendo a aproximação.

Por fim, quando toda a aproximação pelos elementos da tabela haver sido finalizada, ocorre a recuperação de resíduo, levando em conta o último elemento utilizado e o valor do invariante. Ou seja:

$\ln x = y_j - |1 - x_j|$,

onde j é a última iteração realizada.

Quanto a escolha dos valores na LUT, entra o conceito dos ”nice numbers”, ou seja, números ”bonitos”, que tem uma propriedade matemática relevante. Essa estratégia de escolher números com base em uma característica é amplamente utilizada na codificação de bibliotecas de matemática por empresas como a Intel.

Nesse caso, a propriedade relevante é de que todos os números podem ser representados com potências de 2, formato 2^n sendo $n > 0$, ou no formato $2^n + 1$, para $n < 0$. Assim, as operações de multiplicação podem ser realizadas como um bitshift e, se necessário, uma adição, melhorando a precisão ao remover uma operação que seria realizada em cada aproximação.

Sendo assim, os números escolhidos para a tabela são os seguintes, permitindo a aproximação de até $\ln 255$:

2.2. Implementação

Nessa seção, será comentado sobre o processo de desenvolvimento do script, além de serem apresentados trechos relevantes de código e instruções para sua utilização. Nos

	k	$\ln k$
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

trechos de código, serão omitidos comentários e validações a fim de apresentar a lógica principal de forma mais limpa.

Foi desenvolvida a implementação utilizando a linguagem C, compilado no Windows com Clang. A linguagem foi escolhida visto a necessidade de realizar operações a nível de bits na memória, o que é facilitado pelas funções nativas de bitshift, union e struct bitfields.

Também foi implementado um programa básico em python com pandas e matplotlib para renderizar os gráficos dos resultados.

O programa completo está disponível no github, em *schelip/ln-nice-numbers*.

2.2.1. Uso

O programa tem duas opções de uso: número único e geração de arquivo.

Ao passar o argumento `--x` e um número x, será realizado o cálculo de $\ln x$ será apresentado seu resultado e erro no console.

Ao passar o argumento `--upper` e um número n, será realizado o cálculo para todos os valores entre 1 e n, inclusos. O resultado e o erro serão salvos em um arquivo csv, que pode ser renderizado com um gráfico utilizando o script python, rodando o comando `py plot.py <nome_arquivo>`.

2.2.2. Funcionamento

A tabela é construída de maneira ”manual”em uma estrtura especializada. A estrutura funciona como um dicionário ordenado, onde as chaves e valores poderão ser extraídos em listas separadas e serem acessadas por um índice em comum.

```
#define ARRAY_SIZE(a) sizeof(a) / sizeof(a[0])

typedef struct {
    float key;
```

```
    float value;
} key_value_pair_t;

typedef struct {
    key_value_pair_t *data;
    size_t size;
} ordered_dict_t;

ordered_dict_t create_nice_numbers_lookup_table() {
    key_value_pair_t custom[] = {
        {256.0f, logf(256.0f)},
        {16.0f, logf(16.0f)},
        {4.0f, logf(4.0f)},
        {2.0f, logf(2.0f)},
        {3.0f / 2.0f, logf(3.0f / 2.0f)},
        {5.0f / 4.0f, logf(5.0f / 4.0f)},
        {9.0f / 8.0f, logf(9.0f / 8.0f)},
        {17.0f / 16.0f, logf(17.0f / 16.0f)},
        {33.0f / 32.0f, logf(33.0f / 32.0f)},
        {65.0f / 64.0f, logf(65.0f / 64.0f)},
        {129.0f / 128.0f, logf(129.0f / 128.0f)}
    };
    ordered_dict_t result;
    result.size = ARRAY_SIZE(custom);
    result.data = malloc(result.size * sizeof(
       key_value_pair_t));
    if (!result.data) {
        fprintf(stderr, "Failed␣to␣allocate␣memory␣for␣
           lookup␣table\n");
        exit(1);
    }
    memcpy(result.data, custom, result.size * sizeof(
       key_value_pair_t));
    return result;
}
```

Para obter o elemento inicial, é calculada qual a menor potência de 2 maior que x, utilizando bitshift. Então, caso ela não esteja na tabela, a potência sobe de grau até encontrar um elemento.

```
float get_k_0(float x, float* keys, int num_keys, size_t*
   idx_k) {
    int k = 0;
    while ((1 << k) <= (int)x) k++;
    k = (float)(1 << k);
    *idx_k = -1;
    while (1) {
        for (size_t i = 0; i < num_keys; i++)
```

```
            if (k == keys[i]) {
                *idx_k = i;
                break;
            }
        if (*idx_k != -1) break;
        else k <<= 1;
    }
    return k;
}
```

Para a multiplicação, foi criada uma função que verifica se k é da forma 2^n ou $2^n + 1$, e se for o segundo caso, é realizada uma subtração em k para obter o expoente. Então, é realizada a multiplicação com bitshift utilizando a função `ldexp(float, int)`[1], e, no segundo caso, é realizada uma soma adicional de x no resultado para compensar a subtração do expoente.

```
float nice_multiply(float k, float x) {
    float_bits_t fi_k, fi_x;
    fi_k.f = k;

    int i = (int)k; // obtm o inteiro correspondente a k
    float result;

    if (k == (float)i) { // k    da forma 2^n e n > 0
        int n = ((fi_k.i >> 23) & 0xff) - 127;
        result = ldexp(x, n); // multiplica x por 2^n
           usando bitshift
    } else { // k    da forma 2^n + 1 e n < 0
        fi_k.f -= 1.0;
        int n = ((fi_k.i >> 23) & 0xff) - 127;
        result = ldexp(x, n); // multiplica x por 2^n
           usando bitshift
        result += x; // adiciona x mais uma vez
    }

    return result;
}
```

Por fim, segue a função do loop principal, que realiza a aproximação da invariante e, no fim, a recuperação do resíduo.

```
float nice_multiply(float k, float x) {
    float_bits_t fi_k, fi_x;
    fi_k.f = k;

    int i = (int)k; // obtm o inteiro correspondente a k
```

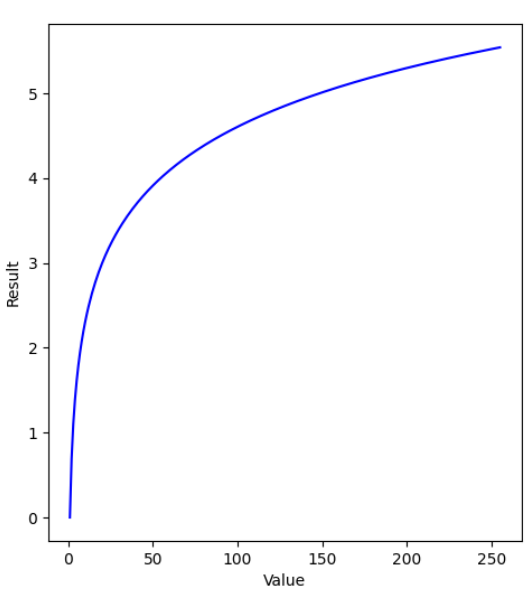
[1] A função `ldexp` não utiliza nenhuma operação, realizando apenas bitshifts a partir do padrão IEEE-754, como pode ser visto no código fonte.

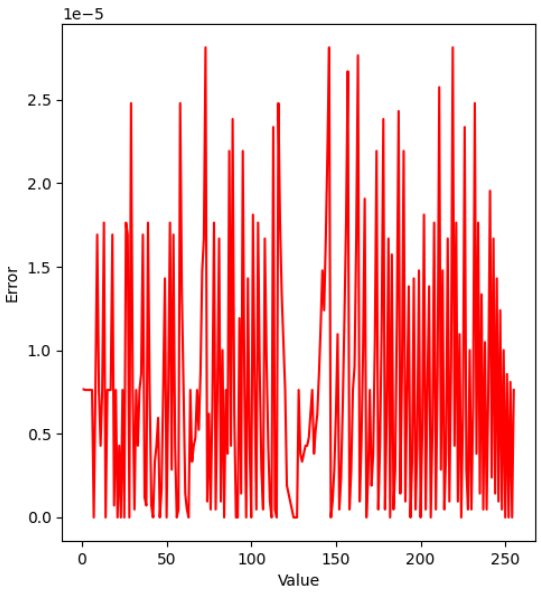
```
    float result;

    if (k == (float)i) { // k    da forma 2^n e n > 0
        int n = ((fi_k.i >> 23) & 0xff) - 127;
        result = ldexp(x, n); // multiplica x por 2^n
           usando bitshift
    } else { // k    da forma 2^n + 1 e n < 0
        fi_k.f -= 1.0;
        int n = ((fi_k.i >> 23) & 0xff) - 127;
        result = ldexp(x, n); // multiplica x por 2^n
           usando bitshift
        result += x; // adiciona x mais uma vez
    }

    return result;
}
```

3. Discussão

Nesse tópico, serão apresentados os resultados obtidos, assim como será apresentada uma análise do erro.

Seguem os gráficos do resultado e do erro para os valores de 0 a 255:





Seguem o resultados e cada iteração para a aproximação de $\ln 54$:

NR-SQRT

```
Iter: 0
K: 4.0000000000
```

```
Xj: 0.84375000000000000000, Yj: 4.15888309478759765625

Iter: 1
K: 1.1250000000
Xj: 0.94921875000000000000, Yj: 4.04110002517700195312

Iter: 2
K: 1.0312500000
Xj: 0.97888183593750000000, Yj: 4.01032829284667968750

Iter: 3
K: 1.0156250000
Xj: 0.99417686462402343750, Yj: 3.99482417106628417969

ln_nice_numbers(54.000000) = 3.989001 (error: 1.69277e-05)
```

Percebe-se que a aproximação foi bem sucedida analisando a forma do gráfico de resultados, que segue o mesmo formato da função logaritma natural para números positivos. Além disso, o erro não passou do grau de $10^{-}5$, preciso suficiente pra maioria das aplicações.

4. Conclusão

No decorrer das atividade, foi possível desenvolver a implementação de um script que realize o cálculo aproximado do logaritmo natural utilizando uma LUT com entrada de nice numbers. Com isso, também foi possível realizar uma análise sobre o erro e seus impactos em métodos de estimação e no contexto geral da computação.