Métodos Numéricos para Engenharia

MÓDULO 5- FONTES DE ERROS – REPRESENTAÇÃO BINÁRIA

PROFESSOR LUCIANO NEVES DA FONSECA

Fontes de erros em Métodos Numéricos

- Erros de Truncamento
- Erros de Arredondamento

Erros de Truncamento

- Erros de truncamento são causados por aproximações sucessivas usadas no esquema de cálculo de uma fórmula matemática.
- São erros associados ao truncamento de um processo infinito, que será resolvido por um número finito de passo.
- A maior parte dos algoritmos utilizados em métodos numéricos são iterativos e alcançam a precisão esperada após um certo número de iterações sucessivas, de modo que a resposta é um truncamento de uma série infinita.
- Como exemplo, podemos usar o método de Newton Raphson para calcular a raiz de uma equação f(x)=0, e interromper o cálculo após n iterações. O cálculo exato ocorreria com infinitas iterações. Como paramos com n iterações, assumimos um erro por 'truncar' a série infinita precocemente.

Exemplo: Série de Taylor para encontra neper

- A Série de Taylor da função $f(x) = e^x$ em torno do ponto $x = 0$ é dada por:

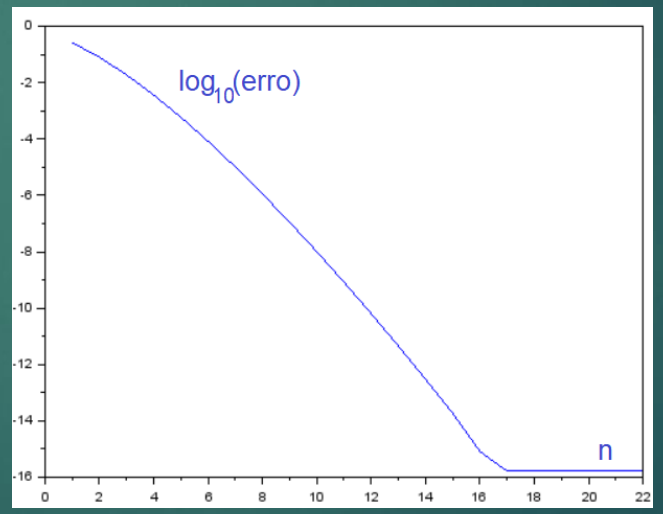
$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots \ldots$$

Para encontrarmos um aproximação do número de neper, podemos avaliar a função $f(x)$ no ponto x=1.
Ao escolhermos um N de termos da série, assumimos um certo erro de truncamento
Para o cálculo do erro utilizaremos como valor exato a constante %e do scilab

$$e = 1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \cdots \ldots$$

```
--> exemplo_truncamento(1,22)
 1       1.00000000000000000      6e-01
 2       2.00000000000000000      3e-01
 3       2.50000000000000000      8e-02
 4       2.66666666666666652      2e-02
 5       2.70833333333333304      4e-03
 6       2.71666666666666634      6e-04
 7       2.71805555555555545      8e-05
 8       2.71825396825396837      1e-05
 9       2.71827876984127004      1e-06
10       2.71828152557319225      1e-07
11       2.71828180114638451      1e-08
12       2.71828182619849290      8e-10
13       2.71828182828616871      6e-11
14       2.71828182844675936      5e-12
15       2.71828182845823019      3e-13
16       2.71828182845899491      2e-14
17       2.71828182845904287      8e-16
18       2.71828182845904553      2e-16
19       2.71828182845904553      2e-16
20       2.71828182845904553      2e-16
21       2.71828182845904553      2e-16
22       2.71828182845904553      2e-16
23       2.71828182845904553      2e-16
Valor exato     2.71828182845904509
```

```
1  function exemplo_truncamento(N)
2      neper=0
3      for n=0:N
4          neper=neper+1/factorial(n)
5          erro_rel(n+1) = abs((neper-%e)/%e)
6          printf("%2d\t%.17f\t%.0e\n",n+1,neper,erro_rel(n+1))
7      end
8      plot(log10(erro_rel))
9      printf("Valor exato\t%.17f\n",%e)
10 endfunction
```



Notar que o erro de truncamento diminui a cada iteração. No entanto, a partir da iteração 17, este erro para de diminuir. Neste ponto, o erro de truncamento não é mais importante. O que está controlando o resultado é o erro de arredondamento.

Erros de Arredondamento

- Computadores mantêm um número fixo de algarismos significativos durante o cálculo, o que causa erros de arredondamento.
- Erros de arredondamento são a soma das incertezas associadas à representação do sistema de numeração do computador.
- Isso se deve ao fato de números reais serem representados em computadores com um número limitado de bits.
- Para estudarmos os erros de arredondamento, devemos primeiro revisar como os números inteiros e reais são armazenados em um computador

- Conversão de decimal/binário e binário/decimal (inteiros)

Caso se queira transformar o número decimal 'a' (exemplo a=74) em binário b devemos:

- Primeiro escolhemos o número n de bits, por exemplo n= 8 bits
- Então formamos o vetor de potência de 2 , $p = [2^{n-1}, 2^{n-2}, \dots, 2^2, 2^1, 2^0]$, com no exemplo n=8, temos:

$$p = [2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0] = [128,64,32,16,8,4,2,1]$$

- Então calculamos o vetor de razões r, que é formado pela parte inteira da divisão $a./p$

$$\mathrm{r} = int(a./p) = \mathrm{int}\left(\left[\frac{74}{128}, \frac{74}{64}, \frac{74}{32}, \frac{74}{16}, \frac{74}{8}, \frac{74}{4}, \frac{74}{2}, \frac{74}{1}\right]\right) = [0, 1, 2, 4, 9, 18, 37, 74]$$

- Se r for par, teremos um bit 0 em b, e se for ímpar teremos um bit 1 em b, então número binário b será:

$$\mathrm{b} = [0, 1, 0, 0, 1, 0, 1, 0] = 01001010$$

- Para transformarmos o binário b no decimal a2, fazemos:

$$\mathrm{a2} = \mathrm{b} * \mathrm{p}' = [0, 1, 0, 0, 1, 0, 1, 0] * [128,64,32,16,8,4,2,1]' = 74$$

- Algoritmo para a conversão de decimal/binário e binário/decimal (inteiros)

```
1  function b=dec2binario1(a,n)
2    p= 2^(n-1:-1:0)
3    r=int(a./p)
4    b = modulo(r,2)
5    if((a>(2^n)-1)|(a<0)) b=[] end
6  endfunction
```

$a = 74$

$n=8$

$p = [2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0] = [128,64,32,16,8,4,2,1]$

$r = int(a./p) = \text{int}\left(\left[\frac{74}{128}, \frac{74}{64}, \frac{74}{32}, \frac{74}{16}, \frac{74}{8}, \frac{74}{4}, \frac{74}{2}, \frac{74}{1}\right]\right) = [0, 1, 2, 4, 9, 18, 37, 74]$

$b = [0, 1, 0, 0, 1, 0, 1, 0]$

```
--> a=74;

--> b=dec2binario1(a,8)
 b  =

   0.   1.   0.   0.   1.   0.   1.   0.

--> b=dec2binario1(a,10)
 b  =

   0.   0.   0.   1.   0.   0.   1.   0.   1.   0.

--> b=dec2binario1(a,6)
 b  =

    []
```

```
1  function a=binario2dec1(b)
2      p= 2^(length(b)-1:-1:0);
3      a= b*p';
4  endfunction
```

$a = b * p' = [0, 1, 0, 0, 1, 0, 1, 0] * [128,64,32,16,8,4,2,1]'=74$

```
--> b=[1 0 0 1 0 1 0];

--> a=binario2dec1(b)
 a  =

   74.

--> b=[0 0 0 1 0 0 1 0 1 0];

--> a=binario2dec1(b)
 a  =

   74.
```

Números negativos – utilizar 1 bit para o sinal

Simplesmente converter valor absoluto do inteiro a, porém com n-1 bits

O Primeiro bit será utilizado para o sinal, 0 ser for positivo, 1 se for negativo

```
--> b=dec2binario2(74,8)
 b  =

   0.   1.   0.   0.   1.   0.   1.   0.

--> a=binario2dec2(b)
 a  =

   74.

--> b=dec2binario2(-74,8)
 b  =

   1.   1.   0.   0.   1.   0.   1.   0.

--> a=binario2dec2(b)
 a  =

  -74.
```

```
1  function b=dec2binario2(a,n)
2    b=dec2binario1(abs(a),n-1)
3    if (a>=0) then  b=[0 b] // acrescentar o bit de sinal
4    else b=[1 b]  end
5    if( a<(-2^(n-1)-1) | a>(2^(n-1)-1) ) b=[] end
6  endfunction
```

```
1  function a=binario2dec2(b)
2    a=binario2dec1(b(2:length(b)))
3    a= (-1)^b(1)*a;
4  endfunction
```

```
--> b=[0 0 0 0 0 0 0 0];

--> a=binario2dec2(b)
 a  =

   0.

--> b=[1 0 0 0 0 0 0 0];

--> a=binario2dec2(b)
 a  =

   0.
```

Notar que temos duas representações para o zero.

0000 0000
1000 0000

Soma de números Binários

$$
\begin{array}{l}
0+0+V0=\ 0\ (+0)\\
0+0+V1=\ 1\ (+0)\\
0+1+\mathrm{V0}=\ 1\ (+0)\\
0+1+\mathrm{V1}=\ 0\ (+1)\\
1+1+\mathrm{V0}=\ 0\ (+1)\\
1+1+\mathrm{V1}=\ 1\ (+1)
\end{array}
$$

```
--> a1=63;

--> a2=5;

--> a3=a1+a2
 a3  =

   68.

--> b1=dec2binario1(a1,8)
 b1  =

   0.   0.   1.   1.   1.   1.   1.   1.

--> b2=dec2binario1(a2,6)
 b2  =

   0.   0.   0.   1.   0.   1.

--> b3=SomaBinaria(b1,b2,%t);
  [00111111]
+ [00000101]
= [01000100]

--> a=binario2dec3(b3)
 a  =

   68.
```

$$
\begin{array}{rr}
 & 00111111\\
+ & \quad 000101\\
\hline
 & ?
\end{array}
$$

$$
\begin{array}{rr}
 & 00111111\\
+ & 00000101\\
\hline
 & 01000100
\end{array}
$$

$$
\begin{array}{l}
0+0=0\\
0+1=1\\
1+1=10\\
1+1+1=11
\end{array}
$$

```
function b3=SomaBinaria(b1,b2,prt)
    [b1,b2]=Compatibiliza_Tamanhos(b1,b2)
    n=length(b1)
    vl=0
    for(k=n:-1:1)
        b3(1,k)=b1(k)+b2(k)+vl
        vl=0
        if b3(k)==2 then
            b3(k)=0
            vl=1
        elseif b3(k)==3 then
            b3(k)=1
            vl=1
        end
    end
    if (prt) then
        printf(" %s\n+ %s\n= %s \n",...
        print_bl(b1),print_bl(b2),print_bl(b3))
    end
endfunction
```

```
function [b1,b2]=Compatibiliza_Tamanhos(b1,b2)
    n1=length(b1)
    n2=length(b2)
    if (n1>n2) then //length(b1)<>length(b2)
        b2=[zeros(1:n1-n2) b2]
    elseif (n2>n1)
        b1=[zeros(1:n2-n1) b1]
    end
endfunction
```

Números negativos – Complemento 2

Simplesmente converter valor absoluto do inteiro a, porém com n-1 bits

Trocar o valor de todos os bits e somar 1

```
--> b=dec2binario3(74,8)
 b  =

   0.    1.    0.    0.    1.    0.    1.    0.

--> a=binario2dec3(b)
 a  =

   74.

--> b=dec2binario3(-74,8)
 b  =

   1.    0.    1.    1.    0.    1.    1.    0.

--> a=binario2dec3(b)
 a  =

  -74.
```

```
1  function b=dec2binario3(a,n)
2    b=dec2binario1(abs(a),n)
3    if(a<0) then  b=comple2(b)  end
4    if( a<(-2^(n-1)) | a>(2^(n-1)-1) ) then b=[] end
5  endfunction
```

```
1  function a=binario2dec3(b)
2    sinal = b(1)
3    if(sinal) then  b=comple2(b) end
4    a= (-1)^sinal * binario2dec1(b) ;
5  endfunction
```

```
1  function b=comple2(b)
2        b=bitcmp(b,1)   // inverter bits
3        b=SomaBinaria(b,[0 1],%f)  // somar 1
4  endfunction
```

Subtração de números Binários

Com complemento 2, o mesmo algoritmo de soma pode ser utilizado na subtração.
Basta somar o número negativo.
Para compatibilizar tamanhos, deve-se completar número, se for negativo, com 1´s e não com 0´s

$$A - B = A + (-B)$$

```
--> a1=63;

--> a2=-5;

--> a3=a1+a2
 a3  =

   58.

--> b1=dec2binario3(a1,8)
 b1  =

   0.   0.   1.   1.   1.   1.   1.   1.

--> b2=dec2binario3(a2,6)
 b2  =

   1.   1.   1.   0.   1.   1.

--> b3=SomaBinaria(b1,b2,%t);
  [00111111]
+ [11111011]
= [00111010]

--> a=binario2dec1(b3)
 a  =

   58.
```

$$\begin{array}{rll} & 00111111 & (63) \\ + & 111011 & (-5) \\ \hline & ? & \end{array}$$

$$\begin{array}{rll} & 00111111 & (63) \\ + & 11111011 & (-5) \\ \hline & 00111010 & (58) \end{array}$$

$flag = 1 \Rightarrow imprime\ resultados$
$flag = 0 \Rightarrow não\ imprime\ resultados$

$0 + 0 + V0 = 0\ (+0)$
$0 + 0 + V1 = 1\ (+0)$
$0 + 1 + \mathrm{V0} = 1\ (+0)$
$0 + 1 + \mathrm{V1} = 0\ (+1)$
$1 + 1 + \mathrm{V0} = 0 (+1)$
$1 + 1 + \mathrm{V1} = 1 (+1)$

- # Conversão de números reais: decimal/binário e binário/decimal

Caso se queira transformar o número decimal 'a' (exemplo a=74.1) em binário b devemos:

- Primeiro escolhemos n bits para a parte inteira, e m bits para depois da vírgula (n=8, m=4)
- Então formamos o vetor de potência de 2 , $p = [2^{n-1}, 2^{n-2}, \dots, 2^2, 2^1, 2^0, 2^{-1}, 2^{-2} \dots 2^{-m}]$,
- com no exemplo n=8 e m=4, temos:

$$p = [2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}] = \left[128,64,32,16,8,4,2,1,\frac{1}{2},\frac{1}{4},\frac{1}{8},\frac{1}{16}\right]$$

- Então calculamos o vetor de razões r, que é formado pela parte inteira da divisão $a./p$

$$\mathrm{r} = int(a./p) = \mathrm{int}\left(\left[\frac{74.1}{128},\frac{74.1}{64},\frac{74.1}{32},\frac{74.1}{16},\frac{74.1}{8},\frac{74.1}{4},\frac{74.1}{2},\frac{74.1}{1},\frac{74.1}{1/2},\frac{74.1}{1/4},\frac{74.1}{1/8},\frac{74.1}{1/16}\right]\right) = [0,1,2,4,9,18,37,74,148,296,592,1185]$$

- Se r for par, teremos um bit 0 em b, e se for ímpar teremos um bit 1 em b, então número binário b será:

$$\mathrm{b} = [0,1,0,0,1,0,1,0,0,0,0,1] = 01001010.0001$$

- Para transformarmos o binário b no decimal a2, fazemos:

$$\mathrm{a2} = \mathrm{bp}' = [0,1,0,0,1,0,1,0,0,0,0,1] * \left[128,64,32,16,8,4,2,1,\frac{1}{2},\frac{1}{4},\frac{1}{8},\frac{1}{16}\right]' = 74.0625$$

- Algoritmo para a conversão de decimal/binário e binário/decimal (reais)

```
function b=dec2binario4(a,n,m) //real positivo
  p= 2^(n-1:-1:-m);
  r=int(a./p)
  b = modulo(r,2)
endfunction
```

```
function a=binario2dec4(b,n,m)//real positivo
   p= 2^(n-1:-1:-m);
   a= b*p';
endfunction
```

```
--> b=dec2binario4(74.0625,8,4);

--> print_b2(b,8,4)
 ans  =

  "[01001010.0001]"

--> a=binario2dec4(b,8,4)
 a  =

   74.0625
```

```
--> b=dec2binario4(74.5,8,4);

--> print_b2(b,8,4)
 ans  =

  "[01001010.1000]"

--> a=binario2dec4(b,8,4)
 a  =

   74.5
```

```
--> b=dec2binario4(74.75,8,4);

--> print_b2(b,8,4)
 ans  =

  "[01001010.1100]"

--> a=binario2dec4(b,8,4)
 a  =

   74.75
```

```
--> b=dec2binario4(74.33,8,4);

--> print_b2(b,8,4)
 ans  =

  "[01001010.0101]"

--> a=binario2dec4(b,8,4)
 a  =

   74.3125
```

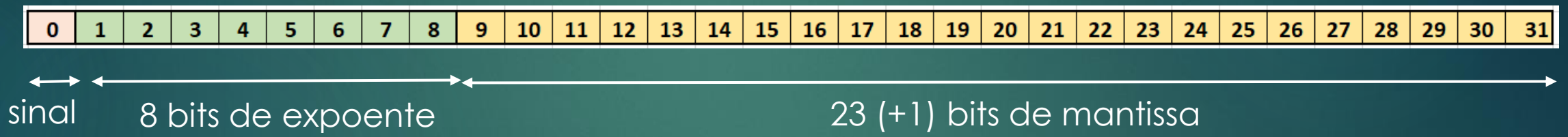
```
--> b=dec2binario4(74.33,8,15);

--> print_b2(b,8,15)
 ans  =

  "[01001010.010101000111101]"

--> a=binario2dec4(b,8,15)
 a  =

   74.329987
```

```
--> b=dec2binario4(74.0,8,4);

--> print_b2(b,8,4)
 ans  =

  "[01001010.0000]"

--> a=binario2dec4(b,8,4)
 a  =

   74.
```

Ponto Flutuante Binário

- Os números reais são normalmente armazenados na notação de ponto flutuante.

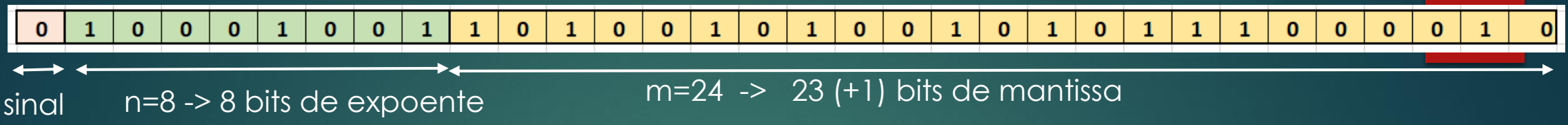
$$x = (-1)^{Sinal}\ (2^{Expoente-Bias})(1.matissa)$$



- Nesta representação (n,m)=(8,24) temos:
- 1 bit para o sinal
- temos 8 bits para o expoente
- 24 bits para a mantissa.
- Este é o padrão IEEE 754 – 32 bits

Conversão de Ponto Flutuante Binário para Decimal

$e = [0\,1\,0\,0\,0\,1\,0\,0\,1\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,0\,0\,0\,1\,0]$



sinal = [0]

$E = [1000\ 1001]$

$E_d = [1000\ 1001]_2 = 137_{10}$

$M = [1\ 101\ 0010\ 1001\ 0101\ 1100\ 0010]$ (+1) bits de mantissa

$M_d = [1.10100101001010111000010]_2 = 1.645195245742797851562_{10}$

$Bias = (2^{n-1} - 1) = 127$

```
1  function a=float2dec(e,n,m)
2      sinal= e(1)
3      E    = e(2:n+1)
4      Ed = binario2dec1(E)
5      M    = [1 e(n+2:n+m)] // + 1 bit
6      Md = binario2dec4(M,1,m-1)
7      Bias = 2^(n-1)-1
8      a = (-1)^sinal * (Md) * 2^(Ed-Bias)
9  endfunction
```

$$a = (-1)^{sinal}\,(2^{E_d - Bias})(M_d) = (-1)^0\,(2^{10})(1.645195245742797851562) = 1684.679931640625$$

```
--> e=[0 1 0 0 0 1 0 0 1 1 0 1 0 0 1 0 1 0 0 1 0 1 0 1 1 1 0 0 0 0 1 0];

--> printfloat(e,8,24)
[0] [10001001] [1.10100101001010111000010]

--> a=float2dec(e,8,24)
 a  =

   1684.679931640625
```

Conversão de decimal para Ponto Flutuante Binário

$a = 1684.679931640625$

$n = 8 \qquad m = 24$

$sinal = 0$ (0 se positivo , 1 se negativo)
$E_d = floor(\log_2|a|) = 10$
$Bias = 2^{n-1} - 1 = 127$
$E = E_d + Bias = 10 + 127 = 137_{10} = [1000\ 1001]_2$
$M_d = |a|\ 2^{-Ed} = |1684.679931640625|2^{-10} = 1.645195245742797851562 5$
M = [1.101 0010 1001 0101 1100 0010]

e = $[sinal\ \ E\ \ M(2:m)]$
e = [0 **1 0 0 0 1 0 0 1** 1 0 1 0 0 1 0 1 0 0 1 0 1 0 1 1 1 0 0 0 0 1 0]

```
--> a=1684.679931640625;

--> e=dec2float(a,8,24,%t);
[0] [10001001] [1.10100101001010111000010]
```

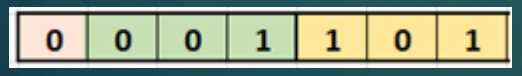
```
1  function e=dec2float(a,n,m,prt)
2      sinal = 0;
3      if (a<0) then sinal = 1; end
4      Bias = 2^(n-1) -1;
5      Ed = floor(log2(abs(a)))+Bias;
6      E = dec2binario1(Ed,n);
7      Md= abs(a) * 2^(-(Ed-Bias));
8      M = dec2binario4(Md,1,m-1);
9      e = [sinal E M(2:m)]
10     if (Ed<0) then
11         e= [sinal zeros(1:n+m-1) ]
12         printf("underflow\n")
13     end
14     if (Ed>2^Bias+1) then
15         e= [sinal ones(1:n+m-1) ]
16         printf("overflow\n")
17     end
18     if (prt) then printfloat(e,n,m) end
19 endfunction
```

Exemplo – float com n = 3 (expoente) e m=4 (mantissa)

		Expoente							
		[000] = -3	[001] = -2	[010] = -1	[011]=0	[100] = 1	[101] = 2	[110] = 3	[111] = 4
Mantissa	[1.000] = 1.000	0.125	0.25	0.5	1	2	4	8	16
	[1.001] = 1.125	0.140625	0.28125	0.5625	1.125	2.25	4.5	9	18
	[1.010] = 1.250	0.15625	0.3125	0.625	1.25	2.5	5	10	20
	[1.011] = 1.375	0.171875	0.34375	0.6875	1.375	2.75	5.5	11	22
	[1.100] = 1.500	0.1875	0.375	0.75	1.5	3	6	12	24
	[1.101] = 1.625	0.203125	0.40625	0.8125	1.625	3.25	6.5	13	26
	[1.110] = 1.750	0.21875	0.4375	0.875	1.75	3.5	7	14	28
	[1.111] = 1.875	0.234375	0.46875	0.9375	1.875	3.75	7.5	15	30

```
--> tabela_floating_point(3,4)
```

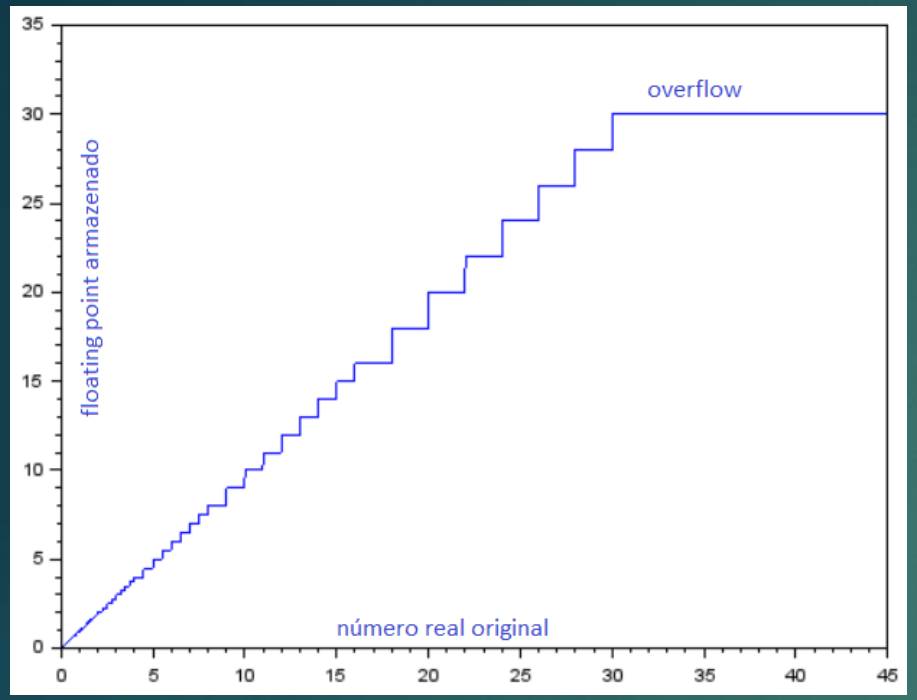
```
Nan     -3.         -2.         -1.         0.          1.          2.          3.          4.
1.       0.125       0.25        0.5        1.          2.          4.          8.          16.
1.125    0.140625    0.28125     0.5625     1.125       2.25        4.5         9.          18.
1.25     0.15625     0.3125      0.625      1.25        2.5         5.          10.         20.
1.375    0.171875    0.34375     0.6875     1.375       2.75        5.5         11.         22.
1.5      0.1875      0.375       0.75       1.5         3.          6.          12.         24.
1.625    0.203125    0.40625     0.8125     1.625       3.25        6.5         13.         26.
1.75     0.21875     0.4375      0.875      1.75        3.5         7.          14.         28.
1.875    0.234375    0.46875     0.9375     1.875       3.75        7.5         15.         30.
```



3 (+1) bits de mantissa

sinal 3 bits de expoente

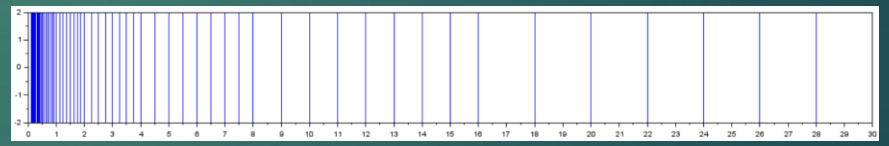
```
--> grafico_floating_point(3,4)
```




```
--> tabela_floating_point(3,4)
```

Nan	-3.	-2.	-1.	0.	1.	2.	3.	4.
1.	0.125	0.25	0.5	1.	2.	4.	8.	16.
1.125	0.140625	0.28125	0.5625	1.125	2.25	4.5	9.	18.
1.25	0.15625	0.3125	0.625	1.25	2.5	5.	10.	20.
1.375	0.171875	0.34375	0.6875	1.375	2.75	5.5	11.	22.
1.5	0.1875	0.375	0.75	1.5	3.	6.	12.	24.
1.625	0.203125	0.40625	0.8125	1.625	3.25	6.5	13.	26.
1.75	0.21875	0.4375	0.875	1.75	3.5	7.	14.	28.
1.875	0.234375	0.46875	0.9375	1.875	3.75	7.5	15.	30.

```
--> eixo_floating_point(3,4)
```

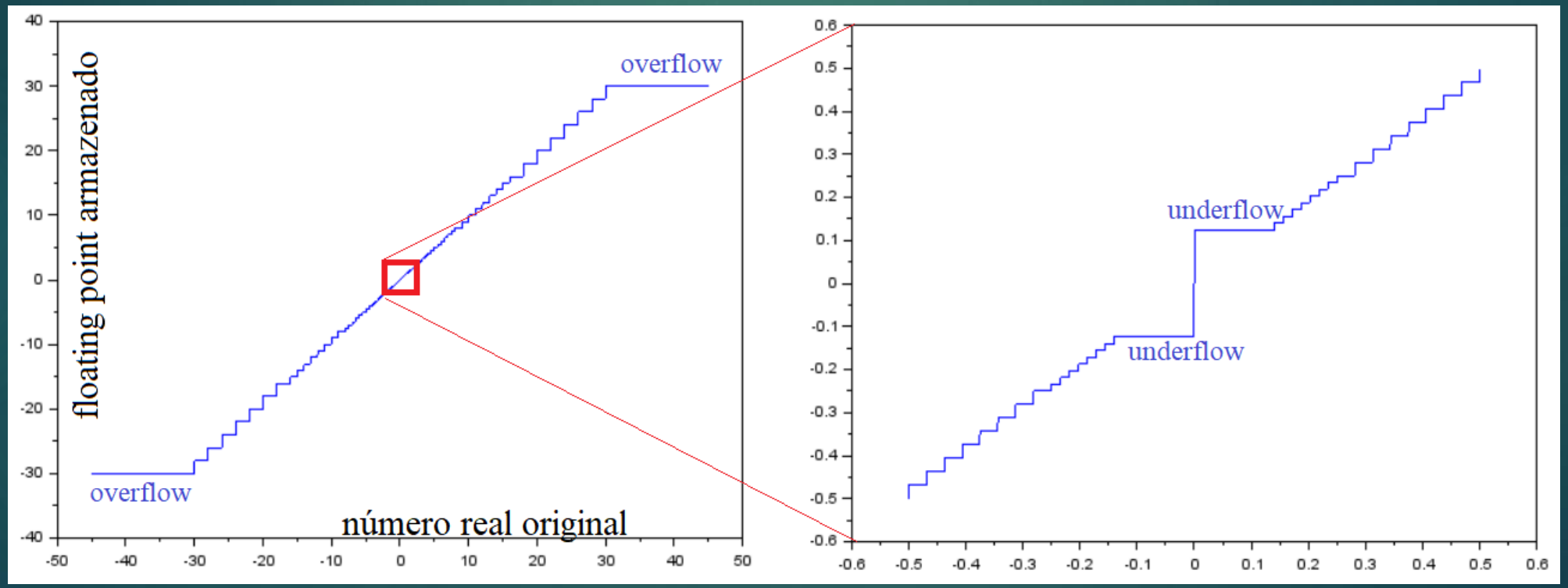


- Vemos que a resolução não é linear.
- Perdemos resolução nos números (com módulo) maiores, pois a resolução é inversamente proporcional ao expoente.
- O maior número representável (30), que depende de n
- Todos os número maiores que 30 serão armazenados em 30(overflow)
- Todos os números menores que -30 serão armazenados -30 (overflow)

```
--> tabela_floating_point(3,4)
```

Nan	-3.	-2.	-1.	0.	1.	2.	3.	4.
1.	0.125	0.25	0.5	1.	2.	4.	8.	16.
1.125	0.140625	0.28125	0.5625	1.125	2.25	4.5	9.	18.
1.25	0.15625	0.3125	0.625	1.25	2.5	5.	10.	20.
1.375	0.171875	0.34375	0.6875	1.375	2.75	5.5	11.	22.
1.5	0.1875	0.375	0.75	1.5	3.	6.	12.	24.
1.625	0.203125	0.40625	0.8125	1.625	3.25	6.5	13.	26.
1.75	0.21875	0.4375	0.875	1.75	3.5	7.	14.	28.
1.875	0.234375	0.46875	0.9375	1.875	3.75	7.5	15.	30.

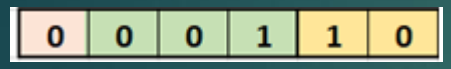
- Todos os números entre -0.125 e zero, serão armazenados em -0.125(underflow)
- Todos os números entre 0 e 0.125 serão armazenados em 0.125 (underflow.



```
--> tabela_floating_point(3,3)
   Nan   -3.      -2.      -1.      0.      1.      2.      3.      4.
   1.    0.125    0.25     0.5      1.      2.      4.      8.      16.
   1.25  0.15625  0.3125   0.625    1.25    2.5     5.      10.     20.
   1.5   0.1875   0.375    0.75     1.5     3.      6.      12.     24.
   1.75  0.21875  0.4375   0.875    1.75    3.5     7.      14.     28.
```

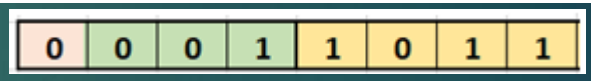
```
--> tabela_floating_point(3,4)
   Nan   -3.      -2.      -1.      0.      1.      2.      3.      4.
   1.    0.125    0.25     0.5      1.      2.      4.      8.      16.
   1.125 0.140625 0.28125  0.5625   1.125   2.25    4.5     9.      18.
   1.25  0.15625  0.3125   0.625    1.25    2.5     5.      10.     20.
   1.375 0.171875 0.34375  0.6875   1.375   2.75    5.5     11.     22.
   1.5   0.1875   0.375    0.75     1.5     3.      6.      12.     24.
   1.625 0.203125 0.40625  0.8125   1.625   3.25    6.5     13.     26.
   1.75  0.21875  0.4375   0.875    1.75    3.5     7.      14.     28.
   1.875 0.234375 0.46875  0.9375   1.875   3.75    7.5     15.     30.
```

```
--> tabela_floating_point(3,5)
   Nan   -3.      -2.      -1.      0.      1.      2.      3.      4.
   1.    0.125    0.25     0.5      1.      2.      4.      8.      16.
   1.0625 0.1328125 0.265625 0.53125 1.0625 2.125  4.25    8.5     17.
   1.125 0.140625 0.28125  0.5625   1.125   2.25    4.5     9.      18.
   1.1875 0.1484375 0.296875 0.59375 1.1875 2.375  4.75    9.5     19.
   1.25  0.15625  0.3125   0.625    1.25    2.5     5.      10.     20.
   1.3125 0.1640625 0.328125 0.65625 1.3125 2.625  5.25    10.5    21.
   1.375 0.171875 0.34375  0.6875   1.375   2.75    5.5     11.     22.
   1.4375 0.1796875 0.359375 0.71875 1.4375 2.875  5.75    11.5    23.
   1.5   0.1875   0.375    0.75     1.5     3.      6.      12.     24.
   1.5625 0.1953125 0.390625 0.78125 1.5625 3.125  6.25    12.5    25.
   1.625 0.203125 0.40625  0.8125   1.625   3.25    6.5     13.     26.
   1.6875 0.2109375 0.421875 0.84375 1.6875 3.375  6.75    13.5    27.
   1.75  0.21875  0.4375   0.875    1.75    3.5     7.      14.     28.
   1.8125 0.2265625 0.453125 0.90625 1.8125 3.625  7.25    14.5    29.
   1.875 0.234375 0.46875  0.9375   1.875   3.75    7.5     15.     30.
   1.9375 0.2421875 0.484375 0.96875 1.9375 3.875  7.75    15.5    31.
```

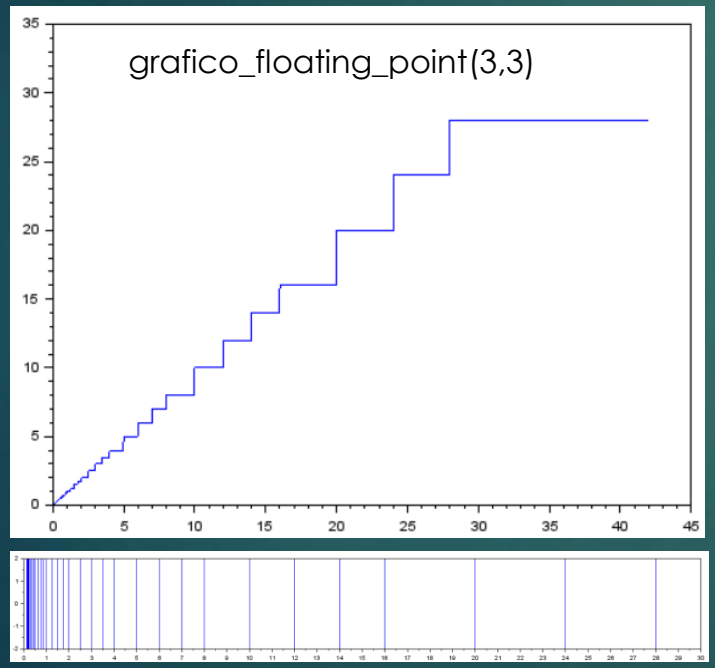


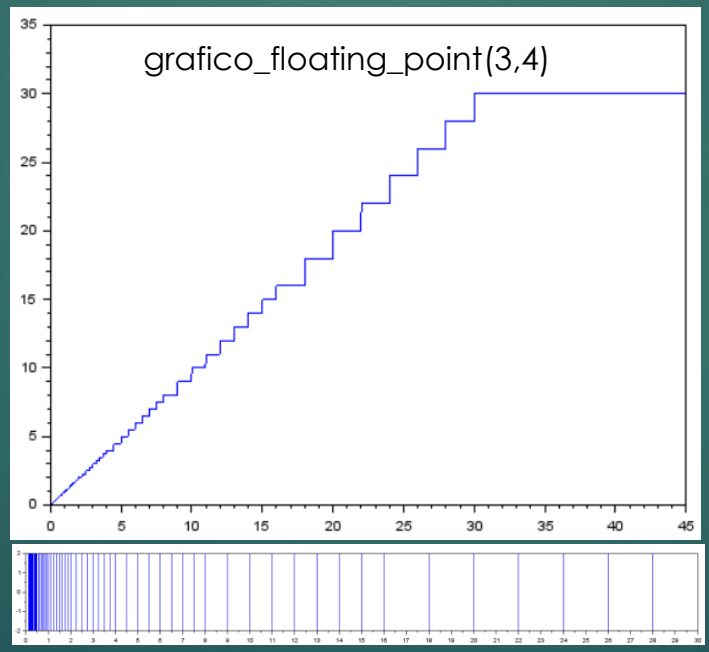
n = 3 e m=3

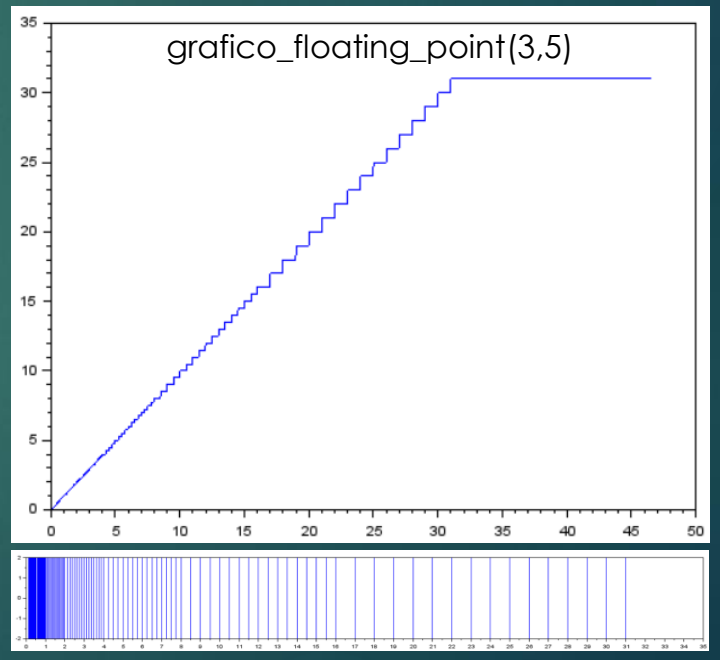
n = 3 e m=4



n = 3 e m=5





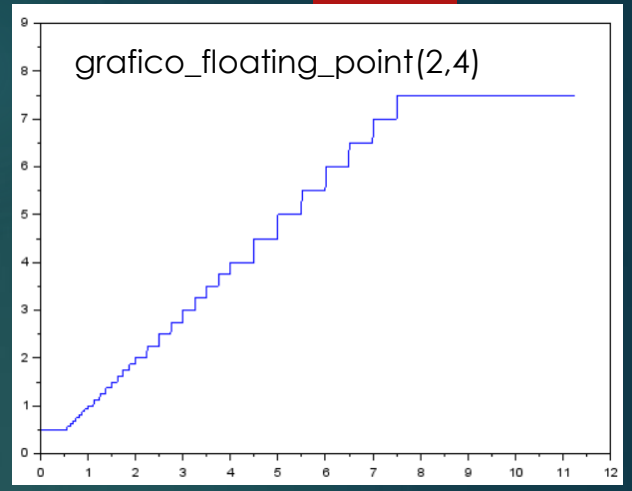


- Ao variarmos m (mantissa),, vemos claramente que a resolução aumenta com m.
- No entanto, o valor máximo (~30) parece ser definido pelo expoente n

n = 2 e m=4

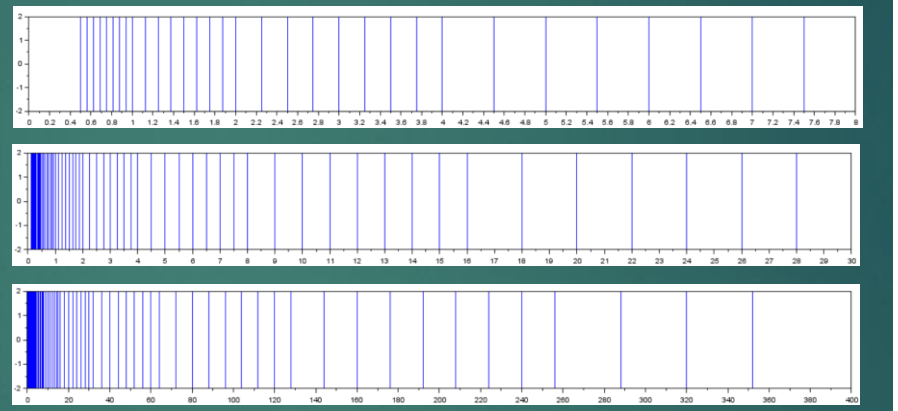
```
--> tabela_floating_point(2,4)
   Nan     -1.        0.        1.        2.
   1.       0.5       1.        2.        4.
   1.125    0.5625    1.125     2.25      4.5
   1.25     0.625     1.25      2.5       5.
   1.375    0.6875    1.375     2.75      5.5
   1.5      0.75      1.5       3.        6.
   1.625    0.8125    1.625     3.25      6.5
   1.75     0.875     1.75      3.5       7.
   1.875    0.9375    1.875     3.75      7.5
```

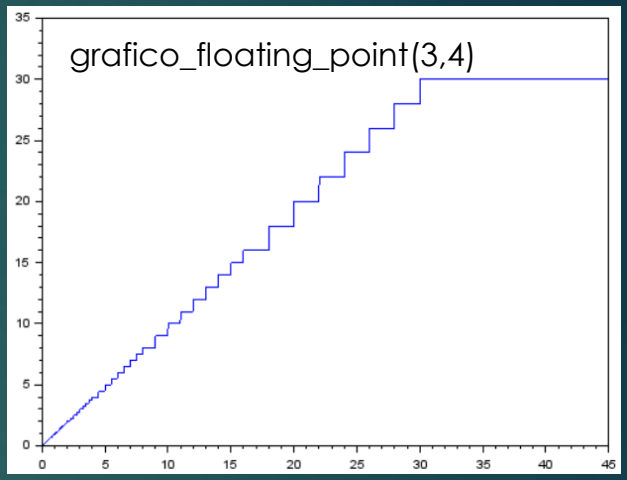
- Ao variarmos n (expoente), mantendo m (mantissa) constante, vemos um aumento no número que pode ser representado aumenta (7.5 , 30 e 480).
- No entanto, a resolução parece não depender do expoente
- Notar o underflow visível no gráfico n=2, m=4



n = 3 e m=4

```
--> tabela_floating_point(3,4)
   Nan     -3.        -2.        -1.        0.        1.        2.        3.        4.
   1.       0.125     0.25       0.5        1.        2.        4.        8.        16.
   1.125    0.140625  0.28125    0.5625     1.125     2.25      4.5       9.        18.
   1.25     0.15625   0.3125     0.625      1.25      2.5       5.        10.       20.
   1.375    0.171875  0.34375    0.6875     1.375     2.75      5.5       11.       22.
   1.5      0.1875    0.375      0.75       1.5       3.        6.        12.       24.
   1.625    0.203125  0.40625    0.8125     1.625     3.25      6.5       13.       26.
   1.75     0.21875   0.4375     0.875      1.75      3.5       7.        14.       28.
   1.875    0.234375  0.46875    0.9375     1.875     3.75      7.5       15.       30.
```





n = 4 e m=4

```
--> tabela_floating_point(4,4)
 Nan    -7.          -6.          -5.          -4.          -3.          -2.          -1.          0.        1.        2.        3.        4.        5.        6.        7.        8.
 1.      0.0078125    0.015625     0.03125      0.0625       0.125        0.25         0.5          1.        2.        4.        8.        16.       32.       64.       128.      256.
 1.125   0.008789062  0.017578125  0.03515625   0.0703125    0.140625     0.28125      0.5625       1.125     2.25      4.5       9.        18.       36.       72.       144.      288.
 1.25    0.009765625  0.01953125   0.0390625    0.078125     0.15625      0.3125       0.625        1.25      2.5       5.        10.       20.       40.       80.       160.      320.
 1.375   0.010742188  0.021484375  0.04296875   0.0859375    0.171875     0.34375      0.6875       1.375     2.75      5.5       11.       22.       44.       88.       176.      352.
 1.5     0.01171875   0.0234375    0.046875     0.09375      0.1875       0.375        0.75         1.5       3.        6.        12.       24.       48.       96.       192.      384.
 1.625   0.012695312  0.025390625  0.05078125   0.1015625    0.203125     0.40625      0.8125       1.625     3.25      6.5       13.       26.       52.       104.      208.      416.
 1.75    0.013671875  0.02734375   0.0546875    0.109375     0.21875      0.4375       0.875        1.75      3.5       7.        14.       28.       56.       112.      224.      448.
 1.875   0.014648438  0.029296875  0.05859375   0.1171875    0.234375     0.46875      0.9375       1.875     3.75      7.5       15.       30.       60.       120.      240.      480.
```