Условие задачи

Найти машинное ε, число разрядов в мантиссе, максимальную и минимальную степени e_{max}, e_{min} при вычислениях с обычной и двойной точностью. Сравнить друг с другом четыре числа: 1, $1 + \frac{\varepsilon}{2}$, $1 + \varepsilon$ и $1 + \varepsilon + \frac{\varepsilon}{2}$, объяснить результат. Будут ли равны $(1 + 10^{-16}) + 10^{-16}$ и $1 + (10^{-16} + 10^{-16})$. Почему?

Найдем машинный эпсилон

Что такое машинный ε? Это такой ε, что $1 + \varepsilon \neq 1 \wedge 1 + \frac{\varepsilon}{2} = 1$. То есть это минимальная разница в числе, которое может различать компьютер. Полагаю, что вместо $1 + \frac{\varepsilon}{2} = 1$ можно было взять условие $1 + \frac{\varepsilon}{1.5} = 1$ или $1 + \frac{\varepsilon}{1.1} = 1$.

Решение перебрать самое очевидное.

```
    float16: finfo.eps = 9.7656250e-04  |  machine_epsilon = 9.7656250e-04
    float32: finfo.eps = 1.1920929e-07  |  machine_epsilon = 1.1920929e-07
    float64: finfo.eps = 2.2204460e-16  |  machine_epsilon = 2.2204460e-16
longdouble: finfo.eps = 1.0842022e-19  |  machine_epsilon = 1.0842022e-19
```

Проверим гипотезу, что можно было взять условие $1 + \frac{\varepsilon}{1.5} = 1$.

```
    float16: finfo.eps = 9.7656250e-04  |  machine_epsilon = 4.8828125e-04
    float32: finfo.eps = 1.1920929e-07  |  machine_epsilon = 5.9604645e-08
    float64: finfo.eps = 2.2204460e-16  |  machine_epsilon = 1.1102230e-16
longdouble: finfo.eps = 1.0842022e-19  |  machine_epsilon = 5.4210109e-20
```

numpy берет макрос __DBL_EPSILON__ из C. gcc считает __DBL_EPSILON__ следующим образом https://github.com/gcc-mirror/gcc/blob/38666cbccff5114e2f23930fae180f03c385eb45/gcc/c-family/c-cppbuiltin.cc#L279-L288.

Найдем число разрядов в мантиссе

Мантисса - это часть числа с плавающей точкой, содержащая его значащие цифры. Тогда число можно записать как:

$$x = \mu \cdot b^e,$$

где

$\mu = \pm\left(\gamma_1 b^{-1} + \gamma_2 b^{-2} + \cdots + \gamma_t b^{-t}\right)$ — мантисса,

t - разрядность (точность) мантиссы,

b - система счисления,

e - порядок.

Кажется, что t - число, такое что $\gamma_{t+1} b^{-t-1} < \varepsilon$.

```
    float16: finfo.precision =  3  |  precision =   3
    float32: finfo.precision =  6  |  precision =   7
    float64: finfo.precision = 15  |  precision =  15
 longdouble: finfo.precision = 18  |  precision =  19
```

Но так как считали в десятично системе счисления, было получено не число разрядов в мантиссе, а точность мантиссы, то есть количество десятичных цифр, которые можно хранить без потерь.

Как получить число разрядов?

```
    float16: finfo.nmant = 11  |  bits = 10
    float32: finfo.nmant = 24  |  bits = 24
    float64: finfo.nmant = 53  |  bits = 50
 longdouble: finfo.nmant = 64  |  bits = 64
```

Есть отличия... Попробуем в обратную сторону - вычислим точность, используя функцию в numpy для вычисления количества разрядов мантиссы.

```
    float16: finfo.precision =  3  |  precision_finfo.nmant =   3
    float32: finfo.precision =  6  |  precision_finfo.nmant =   6
    float64: finfo.precision = 15  |  precision_finfo.nmant =  15
 longdouble: finfo.precision = 18  |  precision_finfo.nmant =  18
```

Получается, что $precision = \lfloor bits \cdot \log_{10} 2 \rfloor$.

Тогда $bits \approx \lceil precision \cdot \log_2 10 \rceil$.

```
    float16: finfo.nmant = 11  |  bits =  10
    float32: finfo.nmant = 24  |  bits =  24
    float64: finfo.nmant = 53  |  bits =  50
 longdouble: finfo.nmant = 64  |  bits =  64
```

По-прежнему отличие на пару битов. В чем дело? Попробуем сразу высчитывать количество разрядов, а не десятичную точность, чтобы избавиться от приближений.

```
    float16: finfo.nmant = 11  |  bits =  11
    float32: finfo.nmant = 24  |  bits =  24
    float64: finfo.nmant = 53  |  bits =  53
 longdouble: finfo.nmant = 64  |  bits =  64
```

Вычисленное число разрядов с фактическим совпало. 😎

Найдем максимальную и минимальную степени

e_{max}, e_{min} при вычислениях с обычной и двойной точностью

Что такое $emin$, $emax$?

b^{emin-1}, b^{emax} - пороги значений, где b - система счислений. $emin$, $emax$ - степени этих порогов.

```
    float16: finfo.minexp =   -14  |  emin =    -24 | emin_tiny =    -14
    float32: finfo.minexp =  -126  |  emin =   -149 | emin_tiny =   -126
    float64: finfo.minexp = -1022  |  emin =  -1074 | emin_tiny =  -1022
 longdouble: finfo.minexp =-16382  |  emin =  -1074 | emin_tiny =  -1075
```

Использования нуля в условии не приближает к нахождению $emin$, хотя программа работает правильно, если ищу $emin$, зная машинный нуль (numpy.finfo.tiny). Это может быть связано с тем, что операции производятся не в $data_type$, а во встроенном $float$.

```
    float16: finfo.minexp =    -14  |  emin =    -25
    float32: finfo.minexp =   -126  |  emin =   -150
    float64: finfo.minexp =  -1022  |  emin =  -1075
 longdouble: finfo.minexp = -16382  |  emin = -16446
```

Вычисление в нужном типе данных $data_type$ не помогает. Это связано с тем, что нужно искать минимальную экспоненту $emin$ для нормализованных чисел. А я делю до 0 (всех нулей в битовом представлении), затрагивая субнормальные числа.

Если экспонента числа $= 0$, но мантисса $\neq 0$, число является субнормальным.

```
False
True
```

Найдем $emin$.

```
    float16: finfo.minexp =    -14  |  emin =    -14
    float32: finfo.minexp =   -126  |  emin =   -126
    float64: finfo.minexp =  -1022  |  emin =  -1022
 longdouble: finfo.minexp = -16382  |  emin = -16382
```

Лирическое отступление. В типах данных до longdouble

```
1 бит знаковый | биты экспоненты | биты мантиссы
```

В типе longdouble:

```
1 бит знаковый | биты экспоненты | бит нормальности | биты мантиссы + биты сдвига
```

longdouble на UNIX имеет 80 значащих битов, а может иметь 112. Но помещается этот тип

данных в 128 бит (битами сдвига).

Я хотел это все учесть, обрезав до 80 или 112 бит, добавив бит нормальности в кортеж, который возвращает divide_binary_str.

Но! число 999 должно представляться так (теоретически):

```
 0 011111111111100 1
11110011100000000000000000000000000000000000000000000000000000000
```

но numpy его представляет как:

```
 0 000000000000000 0
11111110011010101001101100110100100000000010001111100111000000
```

Я теперь не понимаю, почему $emin$ посчиталось правильно, ибо нормализовано ли число в 128-битном представление должно проверяться иначе от проверки для float16/32/64, проверив просто бит нормальности (a. k. a. integer bit).

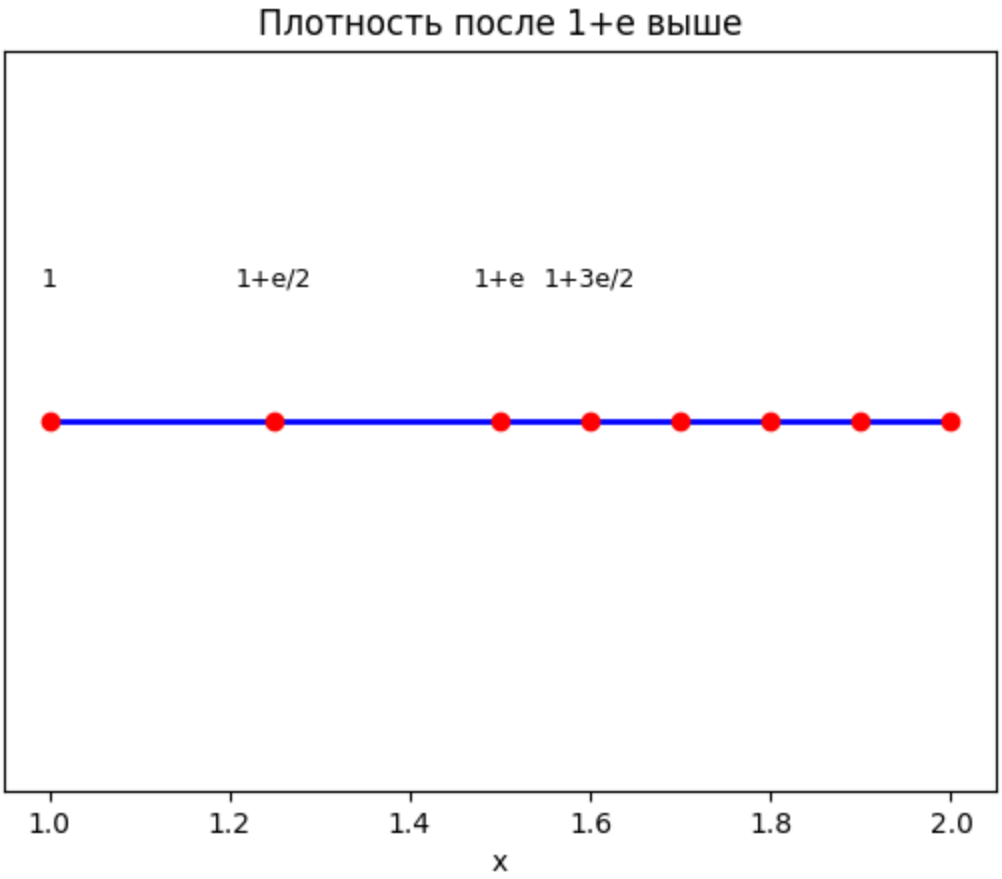
```
     float16: finfo.maxexp =     16  |  emin =     16
     float32: finfo.maxexp =    128  |  emin =    128
     float64: finfo.maxexp =   1024  |  emin =   1024
  longdouble: finfo.maxexp =  16384  |  emin =  16384
/tmp/ipykernel_242446/1175179095.py:5: RuntimeWarning: overflow encountered in scalar
multiply
  value *= dtype(2)
```

Сравнить друг с другом четыре числа: 1, $1 + \frac{\varepsilon}{2}$, $1 + \varepsilon$ и $1 + \varepsilon + \frac{\varepsilon}{2}$, объяснить результат

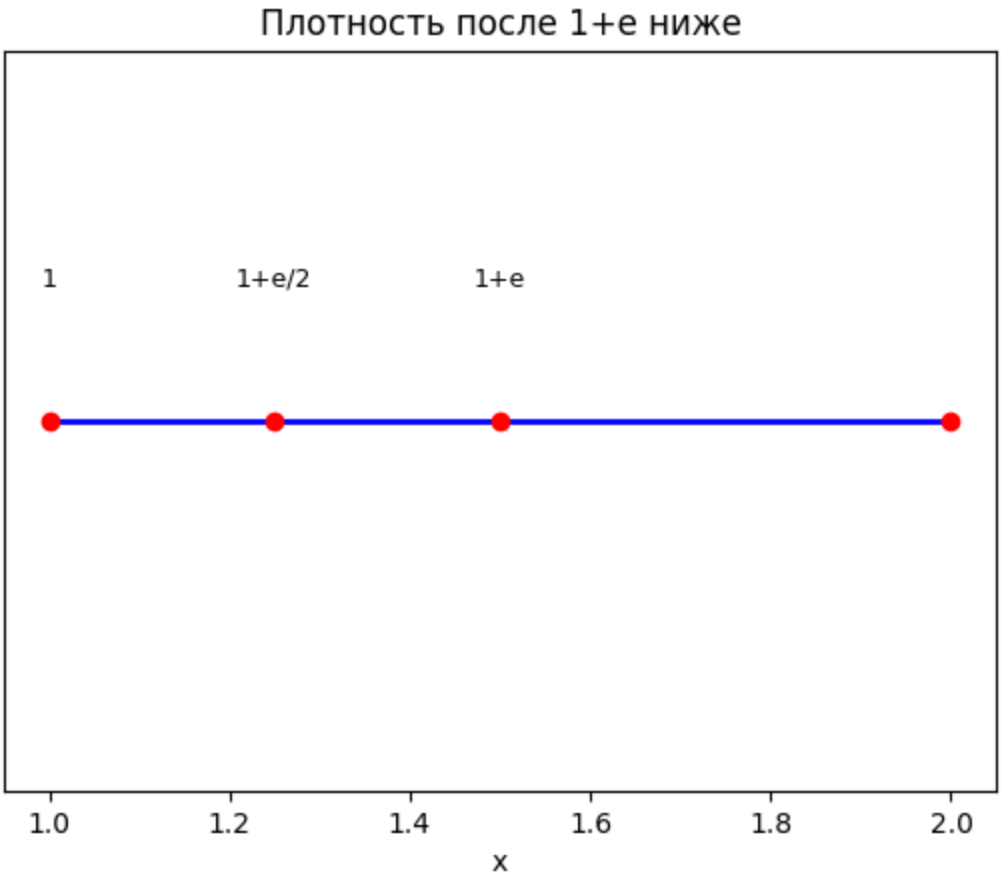
```
1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0
```

1 и $1 + \frac{\varepsilon}{2}$ по определению машинного эпсилона неотличимы, а $1 + \varepsilon$ и $1 + \varepsilon + \frac{\varepsilon}{2}$ по определению машинного эпсилона отличимы от 1.

Но почему тогда $1 + \varepsilon$ и $1 + \varepsilon + \frac{\varepsilon}{2}$ отличимы друг от друга? По определению машинного эпсилона это минимальная разница между 1 и следующим представимым числом, отличным от 1. То есть это Δ между 1 и следующим числом. Но Δ между $1 + \varepsilon$ и следующим представимым числом не факт, что равна ε. Видимо, это Δ меньше, чем Δ для единицы.



На самом деле я неправ. Плотность уменьшется с ростом числа. Числа различимы просто из-за округления.



Будут ли равны $(1 + 10^{-16}) + 10^{-16}$ и $1 + (10^{-16} + 10^{-16})$

```
1.0 1.0
1.0 1.0
1.0 1.0000000000000002
1.0000000000000001999 1.0000000000000002
```

Дело в том, что во всех типах, кроме longdouble $10^{-16} < e$, поэтому при сложении получается единица. Для float64 сумма двух десяток больше эпсилона.

В последнем случае для longdouble $1 + 10^{-16}$ - слишком мало, чтобы изменить последний бит мантиссы. При еще одном добавлении 10^{-16} по-прежнему число слишком мало, чтобы изменить последний бит.

А $2 * 10^{-16}$ достаточно большое, чтобы изменить последний бит мантиссы.

Условие задачи

Рассчитайте сумму следующего ряда: $1 - \frac{1}{2} + \frac{1}{3} - \ldots +$ Чему равна эта сумма при $n \to \infty$? Как ведет себя относительная погрешность при достаточно больших n (точное решение =

$ln(2)$)? Почему?

Ряд сходится условно по признаку Лейбница.

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots = \sum_{n=1}^{\infty} (-1)^{n+1} a_n, \quad a_n = \frac{1}{n}$$

$$S_n = \sum_{k=1}^{n} (-1)^{k+1} a_k$$

$$R_n = S - S_n = \sum_{k=n+1}^{\infty} (-1)^{k+1} a_k$$

$$|R_n| = |a_{n+1} - a_{n+2} + a_{n+3} - \ldots| \leq a_{n+1}$$

Так как

$$|a_{n+1} - (a_{n+2} - a_{n+3}) > 0 - \ldots| \leq a_{n+1}$$

Заметим, что ряд является рядом тейлора для $\ln(1+x)$, где $x = 1$. Поэтому сумма ряда равняется $ln2$.

Определение относительной погрешности:

$$\frac{|S - S_n|}{S}$$

$$\frac{|S - S_n|}{S} \leq \frac{1}{(n+1)} \cdot \frac{1}{S} = \frac{1}{(n+1)} \cdot \frac{1}{\ln 2}$$

При $n \to \infty$:

$$\frac{1}{(n+1)} \cdot \frac{1}{S} = 0$$

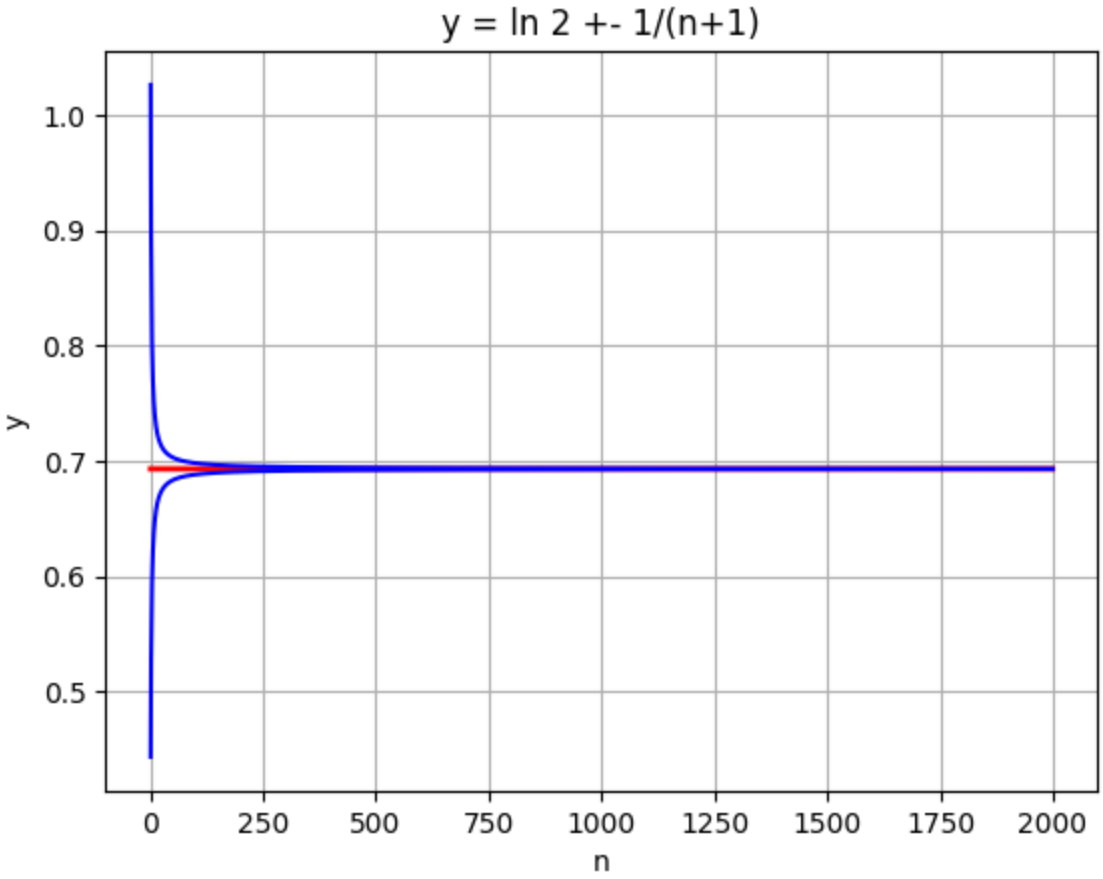
Как рассчитать сумму для любого n?

Напомню, что:

$$R_n = S - S_n$$

Тогда $S_n = S - R_n = \ln 2 - R_n \leq \ln 2 - \frac{1}{n+1}$

График частичных сумм ряда, вычисленных через остаточный член.


y = ln 2 +- 1/(n+1)
1.0
0.9
0.8
0.7
0.6
0.5
y
0
250
500
750
1000
1250
1500
1750
2000
n

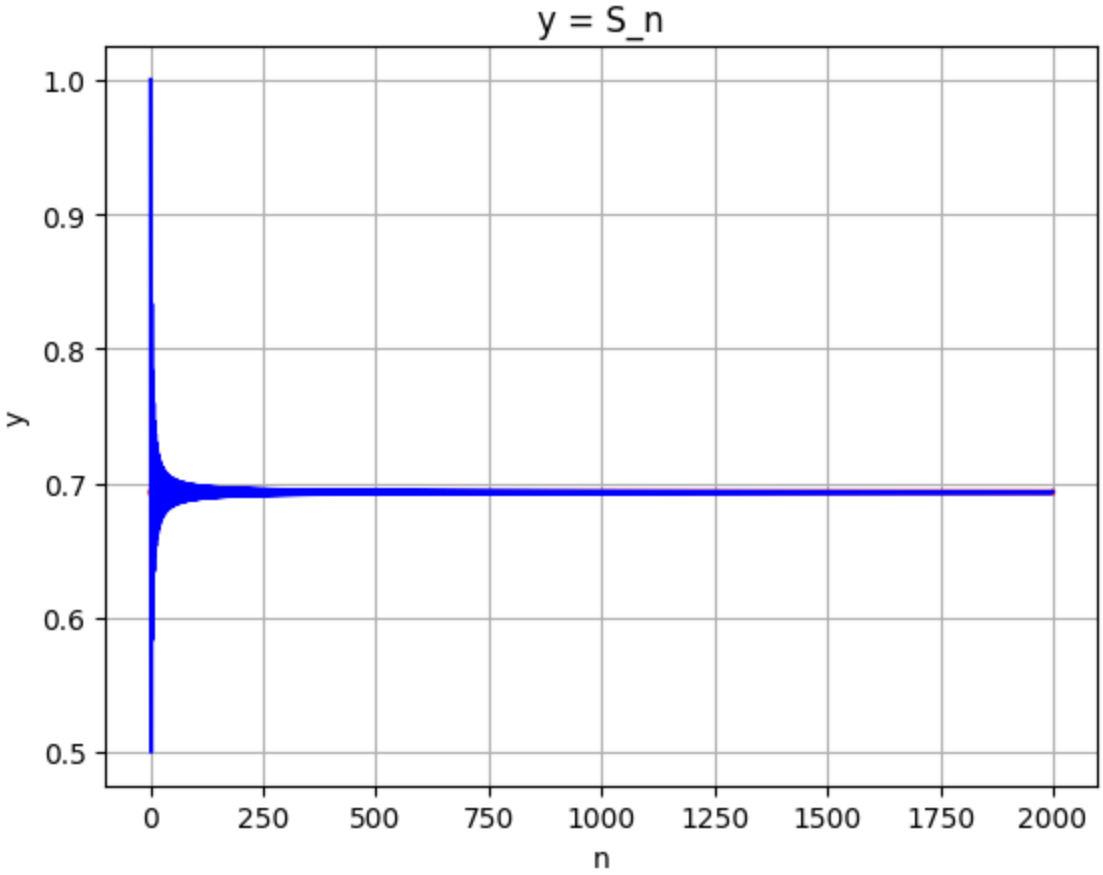
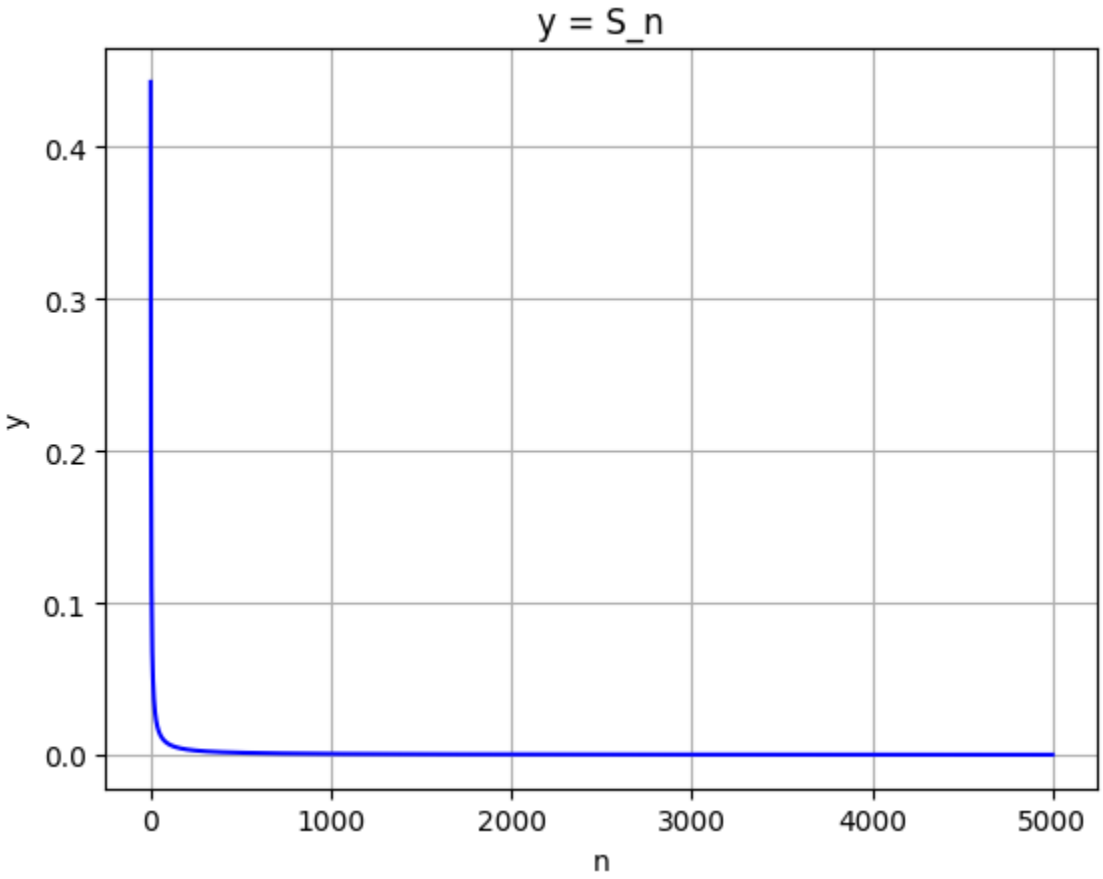
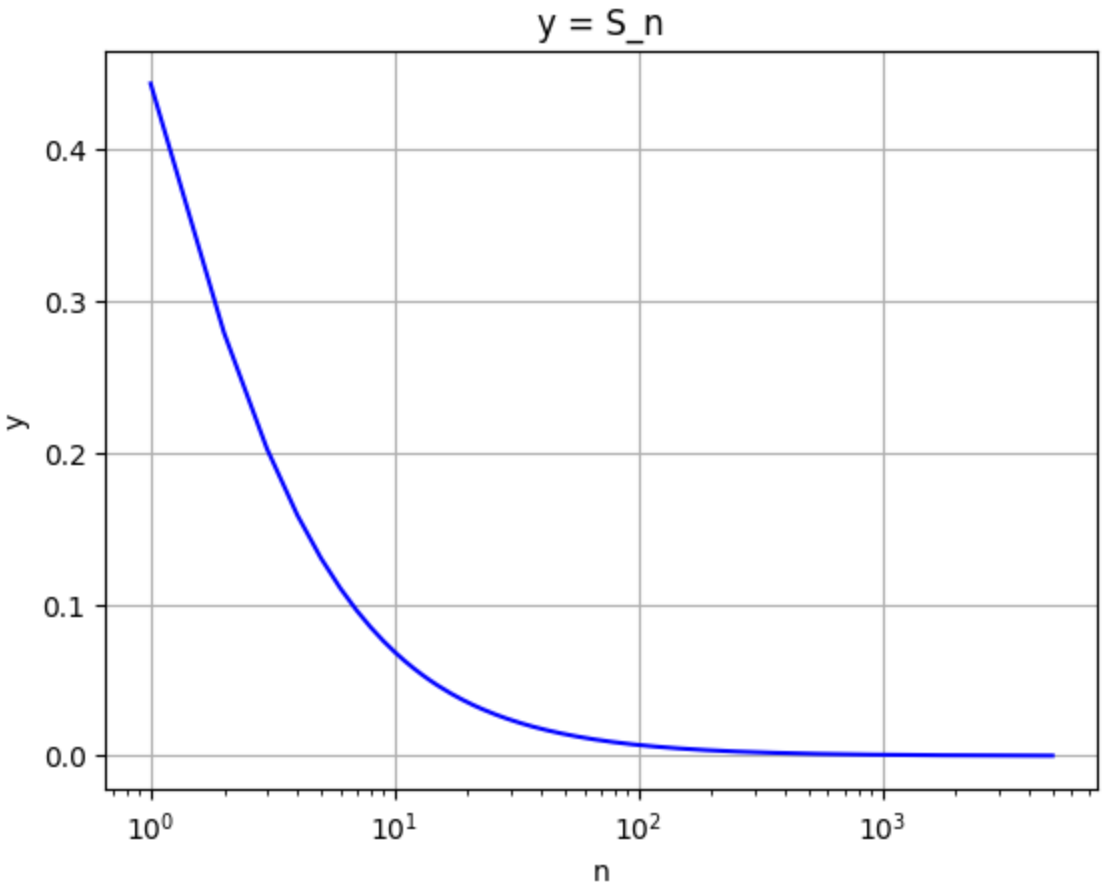

y = S_n
1.0
0.9
0.8
0.7
0.6
0.5
y
0
250
500
750
1000
1250
1500
1750
2000
n

График относительной погрешности





Чтобы достичь точности 10^{-p}

$$\frac{|\ln 2 - S_n|}{\ln 2} \leq 10^{-p}$$

Мы знаем, что

$$\frac{|\ln 2 - S_n|}{\ln 2} \leq \frac{1}{(n+1)} \cdot \frac{1}{\ln 2}$$

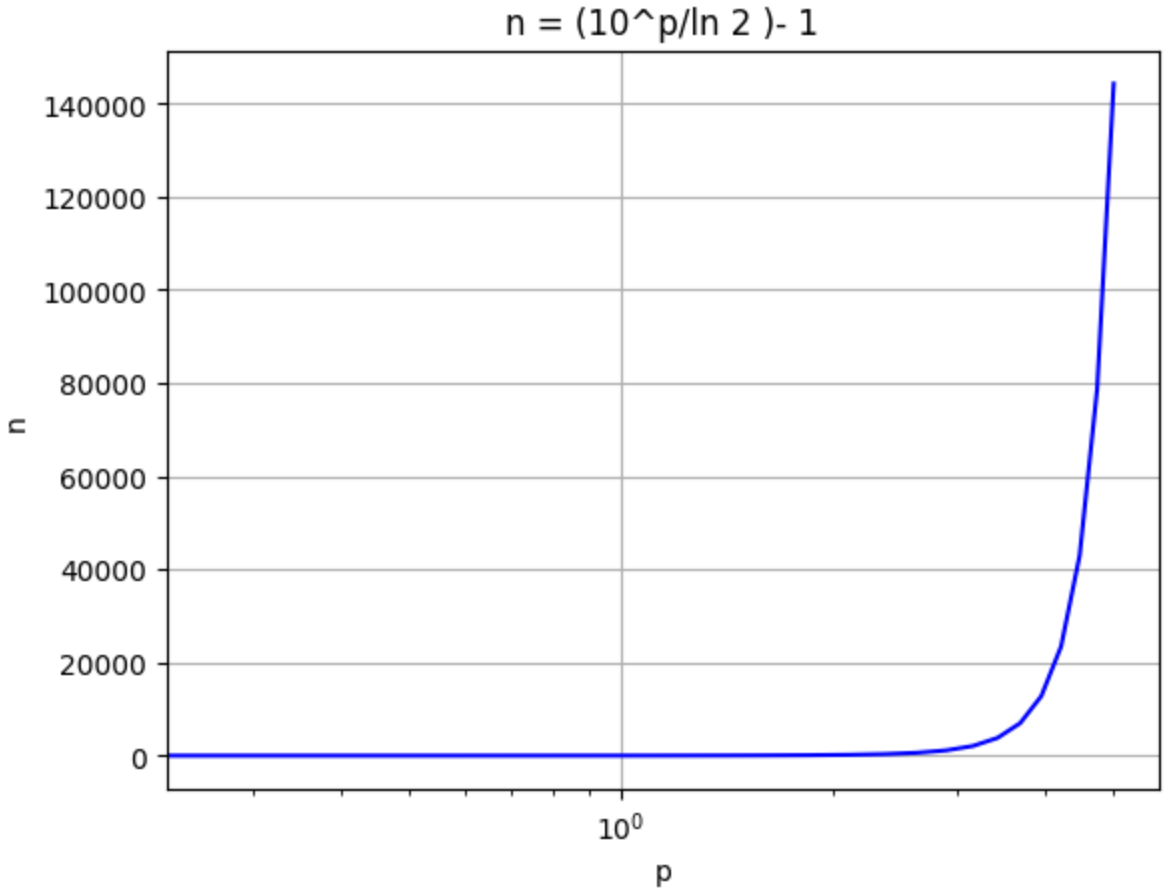
Тогда

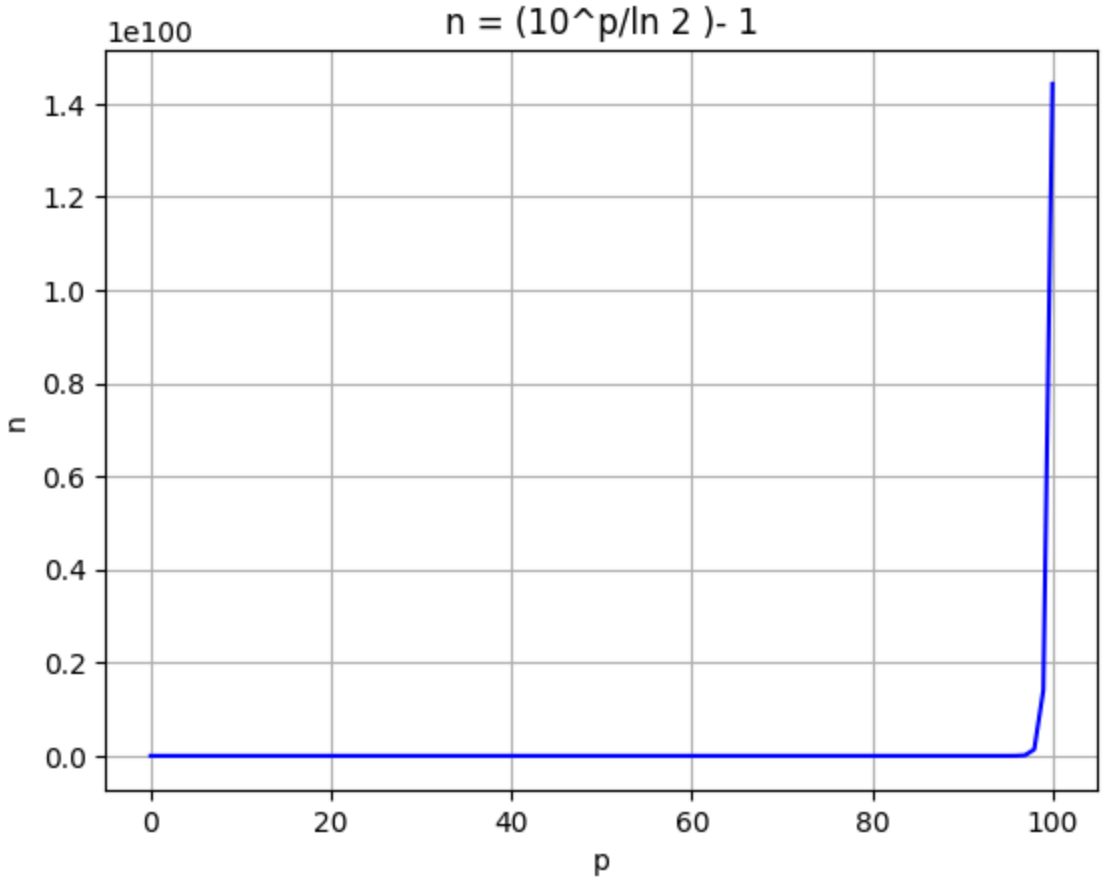
$$\frac{|\ln 2 - S_n|}{\ln 2} \leq \frac{1}{(n+1)} \cdot \frac{1}{\ln 2} \leq 10^{-p}$$

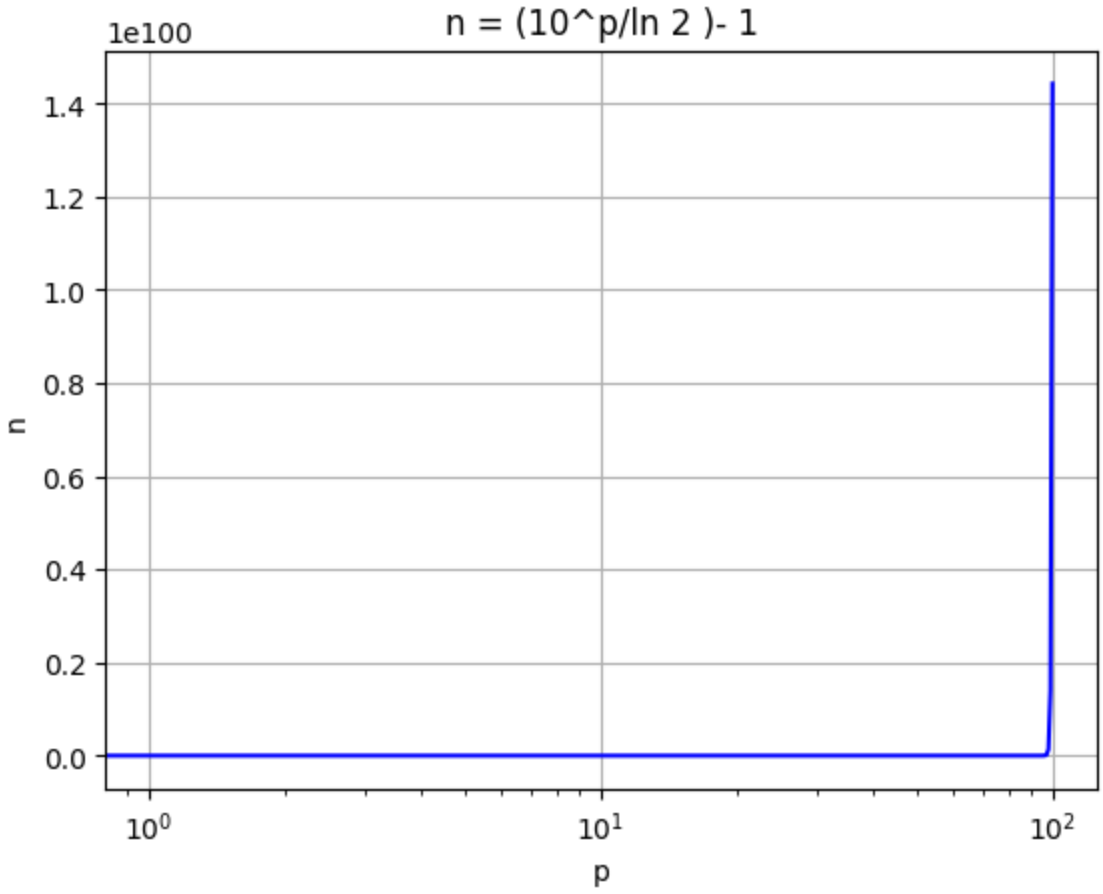
Для $p = const$

$$n \geq \frac{10^p}{\ln 2} - 1$$

Чтобы получить точность p, нужно вычислить $\approx 10^p$ точек.


n = (10^p/ln 2)- 1
140000
120000
100000
80000
60000
40000
20000
0
n
10^0
p


n = (10^p/ln 2)- 1
1e100
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
n
0
20
40
60
80
100
p


n = (10^p/ln 2)- 1
1e100
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
n
10^0
10^1
10^2
p