Лабораторная работа по float

Моргулёв Илья

Октябрь 10, 2023

1 Введение.

Цель работы: ознакомление с хранением чисел типа float в памяти компьютера и изучение закономерностей с этим связанных.

Оборудование: Ноутбук, программы для воспроизведения, компиляции и дебаггинга кода, онлайн компиляторы и строители графиков.

2 Теоретические сведения.

Float - формат чисел с плавающей точкой. В связи с чем возникают особенности: хранение с помощью мантисы позволяет с высокой степнью точности считать многие вещественные числа, вместе с чем возникают некоторые трудности: переполнение мантисы, приближённое число и прочее.

Union - это оператор в языке SQL, применяется для объединения двух наборов строк, возвращаемых SQL-запросами. Оба запроса должны возвращать одинаковое число столбцов, и столбцы с одинаковым порядковым номером должны иметь совместимые типы данных. (пример: см. Задание 1)

IEEE 754-2008 - это правило, по которому компьютер высчитывает число и хранит его.

$$Number_{10} = (-1)^s \cdot 2^{E-127} \cdot (1 + \frac{m}{2^n})$$

Где S - бит знака (0 - положительное; 1 - отрицательное), E - смещённая экспонента двоичного числа, m - остаток мантисы двоичного двоичного нормализованного числа.

3 Ход работы

1. **Задание 0:** Вывести на экран unsigned int в двоичной системе счисления (то есть так, как этот unsigned int хранится в памяти компьютера), используя побитовые операции.

   Используемый код:

```
#include <iostream>
using namespace std;

int main(){
    unsigned int n,k,b;
    int a =0;
    cin >> n;
    k=1;
```

```
    k<<=31;
    for (int i=0;i<32;i++){
        b=n&k;
        if (b == 0) cout << 0;
        else cout << 1;
        k>>=1;
        if (a==3) {cout << " "; a=0;}
        else a++;
    }
}
```

Программа роботает, выводит число в том виде, как оно хранится в компьютере, перепишем функцию *main* в вызываемую функцию *binary*.

2. **Задание 1:** С помощью union и функции из предыдущего пункта вывести на экран представление float в памяти компьютера. Проверьте, что $IEEE754-2008$ (стандарт) действительно работает.

   Используемый код:

```
#include <iostream>
union flo_int{
    unsigned int u;
    float f;
};
void binary(unsigned int n){
    unsigned int k,b;
    int a =0;
    k=1;
    k<<=31;
    for (int i=0;i<32;i++){
        b=n&k;
        if (b == 0) std::cout << 0;
        else std::cout << 1;
        k>>=1;
        if (a==3) {std::cout << " "; a=0;}
        else a++;
    }
}
int main(){
    flo_int n_lab_rat;
    std::cin >> n_lab_rat.f;
    binary(n_lab_rat.u);
}
```

Убедимся в соответствии (или несоответсвии):

ввод	вывод	$IEEE754-2008$
5	0100 0000 1010 0000 0000 0000 0000 0000	Соответствует
12.5	0100 0001 0100 1000 0000 0000 0000 0000	Соответствует
37.5	0100 0010 0001 0110 0000 0000 0000 0000	Соответствует
101.567	0100 0010 1100 1011 0010 0010 0100 1110	Соответствует
123.5	0100 0010 1111 0111 0000 0000 0000 0000	Соответствует
12345.23456	0100 0110 0100 0000 1110 0100 1111 0000	Сооветствует
-123.456	0100 0110 0100 0000 1110 0100 1111 0000	Соответствует

3. **Задание 2:** Воспроизвести переполнение мантиссы. Проще и нагляднее всего – в цикле выводить десятичное и двоичное представление степеней десятки.

   Используемый код:

```
#include <iostream>
union flo_int{
    unsigned int u;
    float f;
};
void binary(unsigned int n){
    unsigned int k,b;
    int a =0;
    k=1;
    k<<=31;
    for (int i=0;i<32;i++){
        b=n&k;
        if (b == 0) std::cout << 0;
        else std::cout << 1;
        k>>=1;
        if (a==3) {std::cout << " "; a=0;}
        else a++;
    }
}
int main(){
    std::cout << std::fixed;
    std::cout.precision(2);
    flo_int n_lab_rat;
    float k =1.0;
    for (int i=0;i<39;i++){
        n_lab_rat.f=k*10;
        k*=10;
        std::cout << k/10 << " // ";
        binary(n_lab_rat.u);
        std::cout << std::endl;
    }
}
```

Проанализируем выходные данные, занесённые в таблицу:

1.00	0100 0001 0010 0000 0000 0000 0000 0000
10.00	0100 0010 1100 1000 0000 0000 0000 0000
100.00	0100 0100 0111 1010 0000 0000 0000 0000
1000.00	0100 0110 0001 1100 0100 0000 0000 0000
10000.00	0100 0111 1100 0011 0101 0000 0000 0000
100000.00	0100 1001 0111 0100 0010 0100 0000 0000
1000000.00	0100 1011 0001 1000 1001 0110 1000 0000
10000000.00	0100 1100 1011 1110 1011 1100 0010 0000
100000000.00	0100 1110 0110 1110 0110 1011 0010 1000
1000000000.00	0101 0000 0001 0101 0000 0010 1111 1001
10000000000.00	0101 0001 1011 1010 0100 0011 1011 0111
99999997952.00	0101 0011 0110 1000 1101 0100 1010 0101
999999995904.00	0101 0101 0001 0001 1000 0100 1110 0111
9999999827968.00	0101 0110 1011 0101 1110 0110 0010 0001
100000000376832.00	0101 1000 0110 0011 0101 1111 1010 1001
1000000054099968.00	0101 1010 0000 1110 0001 1011 1100 1010
10000000272564224.00	0101 1011 1011 0001 1010 0010 1011 1100
99999998430674944.00	0101 1101 0101 1110 0000 1011 0110 1011
999999984306749440.00	0101 1111 0000 1010 1100 0111 0010 0011
9999999980506447872.00	0110 0000 1010 1101 0111 1000 1110 1100

Как мы видим, из-за особенностей записи числа типа $float$ в памяти компьютера возникает ситуация, что в мантиссе возникают значащие единицы, которые привлдят в последствии умножения на 10 к тому, что число не совсем уж и умножается на 10. Пример - 10000000000.00− > 99999997952.00 - умножение на 10 выглядит немного странно, согласитесь. В следствие того, что единица в бинарном разлодении будто "вылазиет"из мантиссы это и названно переполнением мантисы.

4. **Задание 3:** Создать бесконнечный цикл.

Используемый код:

```
#include <iostream>

int main(){
    std::cout << std::fixed;
    std::cout.precision(2);
    float f;
    int k=0;
    system("color 02");
    std::cin >> f;
    for (f;f<10000000000;f++){
        std::cout << f << " _-_//_-_ ";
        k++;
        if (k==5) {k=0; std::cout << std::endl;}

    }
}
```

На значении f, равном 16777216.00 , расстояние между соседними $floatf$ будет больше единицы, отчего после цикл не сможет "перепрыгнуть"с одной иттерации цикла на другую. Это происходит из-за того, что не все числа могут быть закодированны в мантисе. Простой пример - число 0.2 , переводя его в $float$ получим следующую последовательность шагов:

номер шага	текущее число	пояснение
0	0.2	non
1	0.4	non
2	0.8	non
3	1.6	->выделяем целую часть
3	0.6	non
4	1.2	->выделяем целую часть
4	0.2	non

Как мы видим цикл записи числа 0.2 перешёл в бесконечный цикл в связи с особенностями типа $float$. Что же тогда запишется в память компьютера? Число, приблизительно равное 0.2 . В связи с этим, в довесок, появляются проблемы с сравнением чисел, но нас волнует число 16777216.00 . Так это число выглядит в памяти компьютера: **0100 1011 1000 0000 0000 0000 0000 0000** . Однако следующее за ним число - 16777217.00 имеет такой же двоичный код в памяти компьютера, следовательно прибавляя единицу к 16777216.00 двоичный код не меняется и компьютер не видит разницы оставаясь на той же иттерации. Выражаясь неформально: компилятор не смог "перепрыгнуть"с одной иттерации на другую, так как с увличением числа типа $float$ между числами появлюятся некоторые зазоры, которые как раз и могут привести к подобному результату.

5. **Задание 4:** График π. Найдём 4-5 иттерационных формул для числа π, посчитаем, построим график зависимости точности от количества иттераций. Найдём ошибку подсчёта.

1.1: Формула Лейбница.

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1} \tag{1}$$

Используемый код:

```
#include <iostream>
#include <math.h>
#include <fstream>
using namespace std;
int main() {
    float n;
    float pi;
    ofstream pi_res("1.csv", ios::out);
    pi=0;
    std::cout << "Approx Leibnica's series\n";
    std::cout << "\nEnter the number of iterations: ";
    std::cin >> n;
    std::cout << "Please wait. Running..." << "\n";
        for(long long int i = 0; i < n; i++){
            if (i%2 == 0) pi+=4.0/((2.0*i) + 1.0);
            else pi+=-4.0/((2.0*i) + 1.0);
        pi_res << "[" << i << ", " << pi << "], ";
       }
   return 0;
}
```

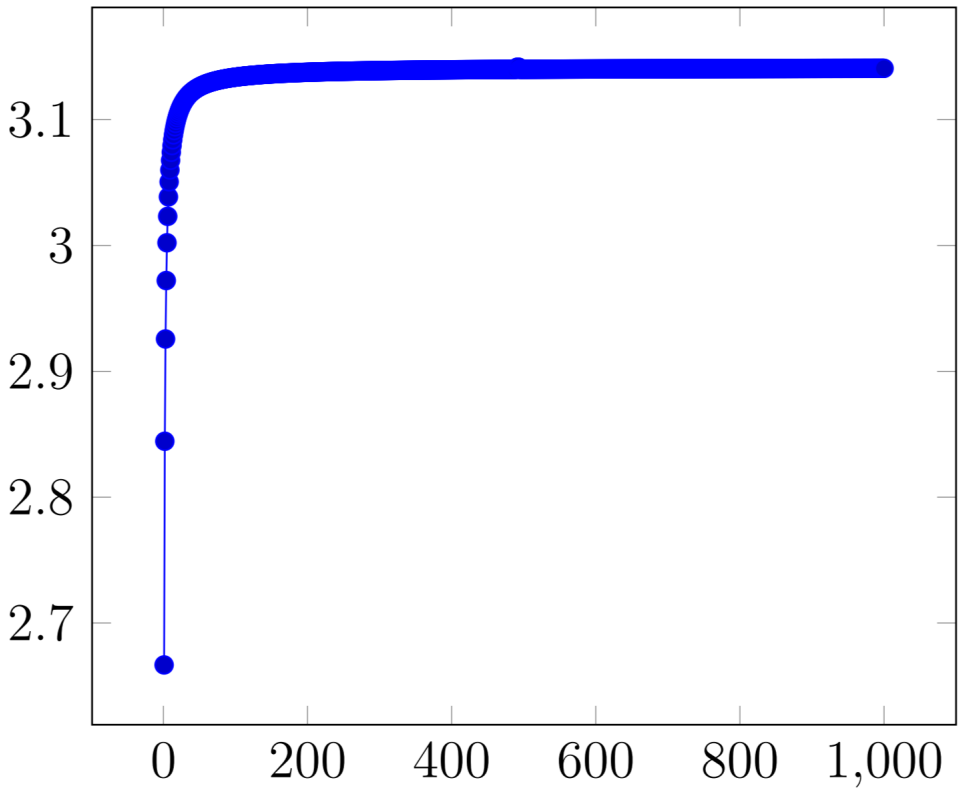

Рис. 1: График зависимости числа π от числа итераций (n)

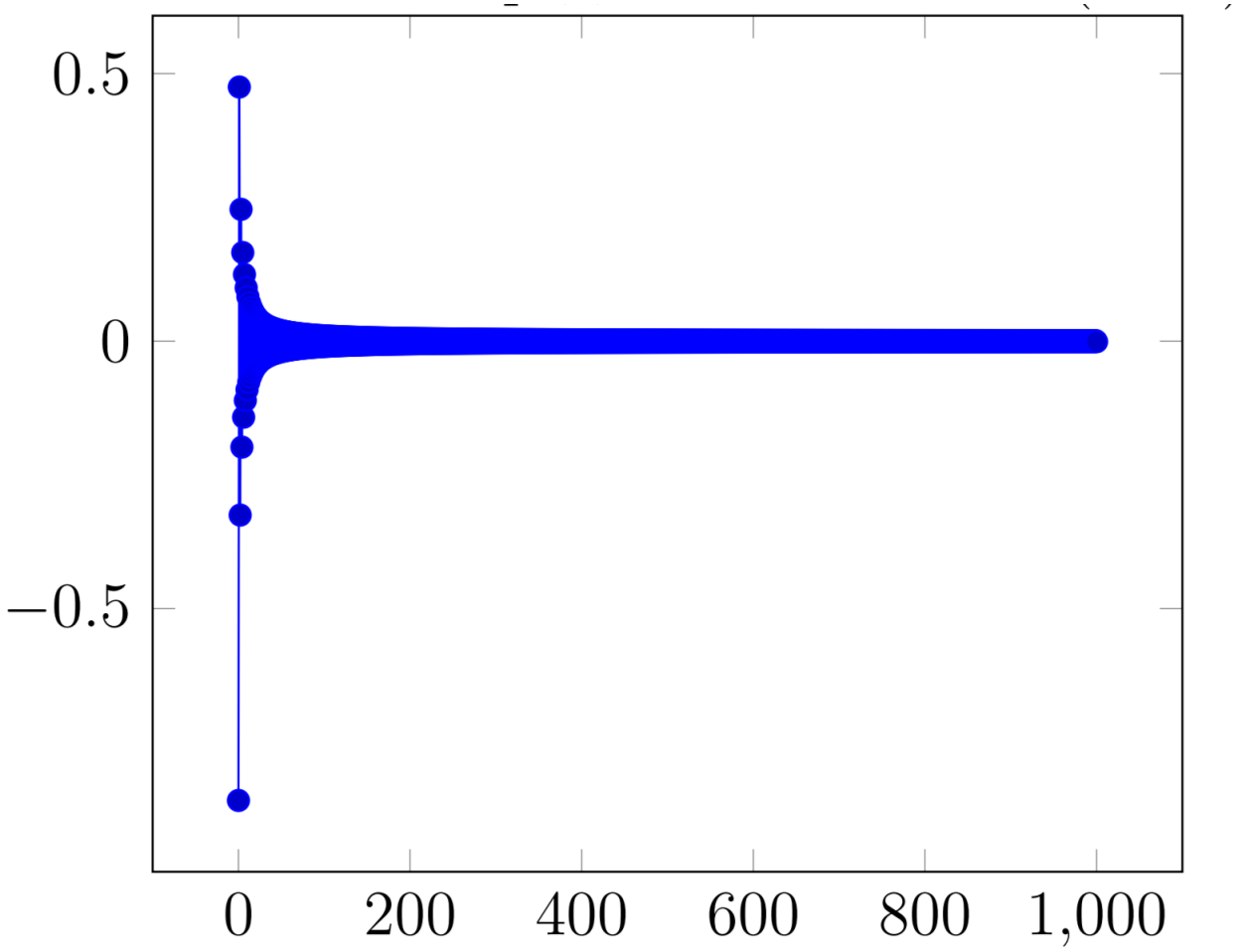

Рис. 2: Разности истинного значения числа π и полученного нами значения в формуле в зависимости от номера итерации

Рассмотрим участок времени от 0 до 60 итераций, рассмотрим, как ведёт себя разница между полученным нами значением и истинным значением:

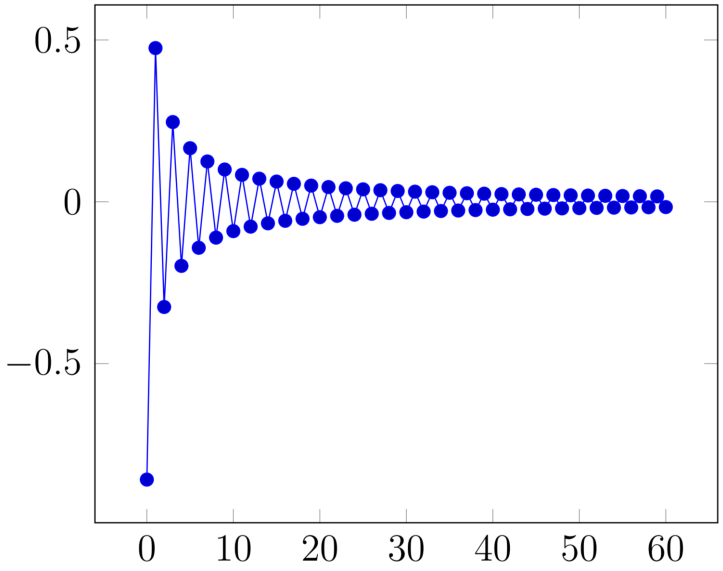

Рис. 3: разница между полученным нами значением и истинным значением в зависимости от номера итерации

Как мы видим, последовательность полученных нами значений, сходится к истинному значению с двух сторон. Рассмотрим участок от 500 до 1000: Как мы видим по графику

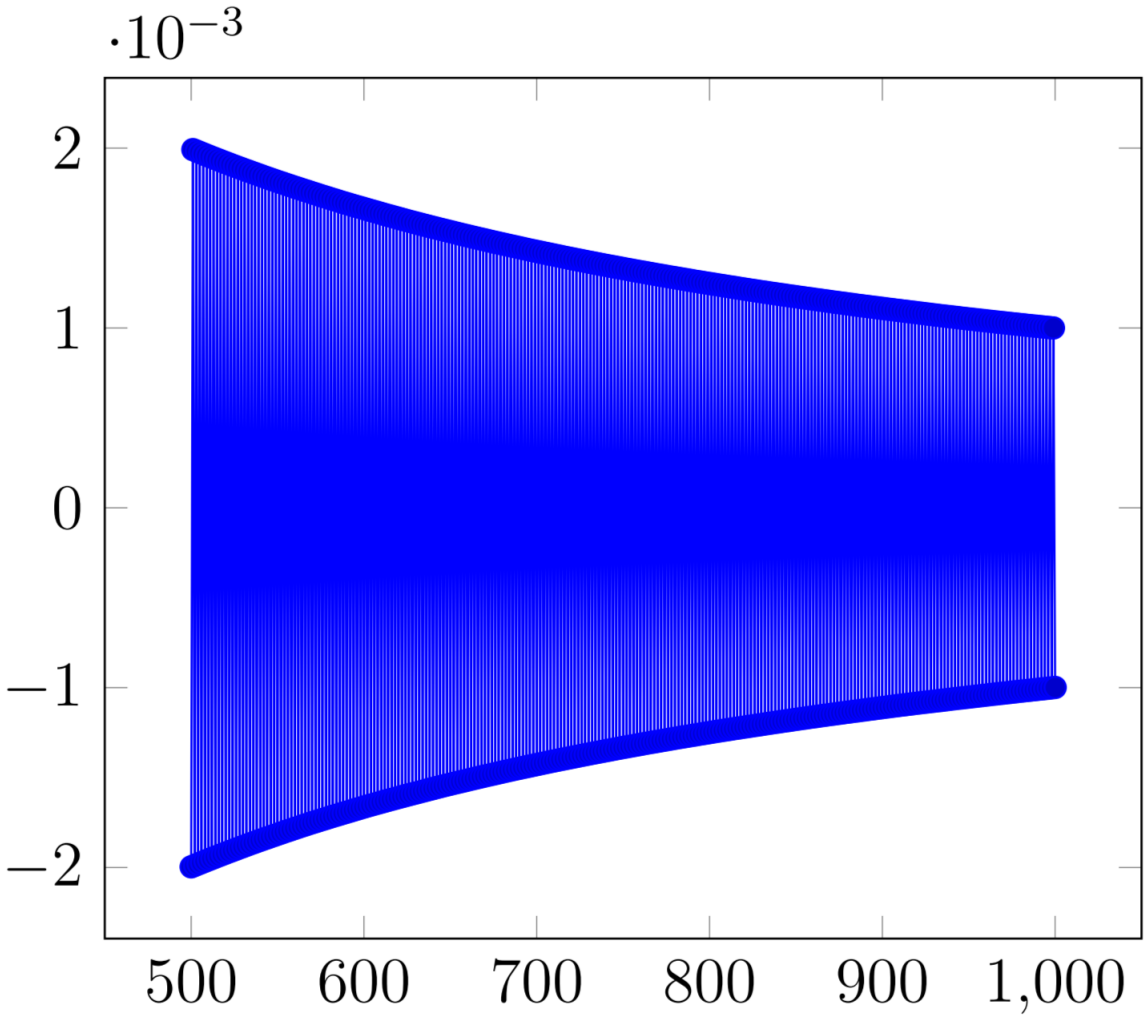

Рис. 4: разница между полученным нами значением и истинным значением в зависимости от номера итерации

(и что было вполне очевидно) - разность между истинным значением и полученным нами отличается на тысячные доли и колеблется вокруг истинного значения.

Следствие: при измерении числа π с помощью формулы (1) точность составляет 0.0001.

1.2: Формула Валлиса.

$$\pi = 2 \cdot \prod_{i=0}^{\infty} \frac{4i^2}{4i^2 - 1} \tag{2}$$

Используемый код:

```
#include <iostream>
#include <math.h>
#include <fstream>
using namespace std;
int main() {
    float n, i, j;
    float pi;
    ofstream pi_res("1.csv", ios::out);
    while (1){
        pi=1.0;
        std::cout << "PI through the Vallie's series\n";
        std::cout << "\nEnter the number of iterations: ";
        std::cin >> n;
        std::cout << "Please wait. Running..." << "\n";
        for (i=0;i<n;i++){
            j=4.0*pow(i,2);
            pi *= j/(j-1.0);
            pi_res << "[" << i << "," << pi << "], ";
        }
    }
   return 0;
}
```

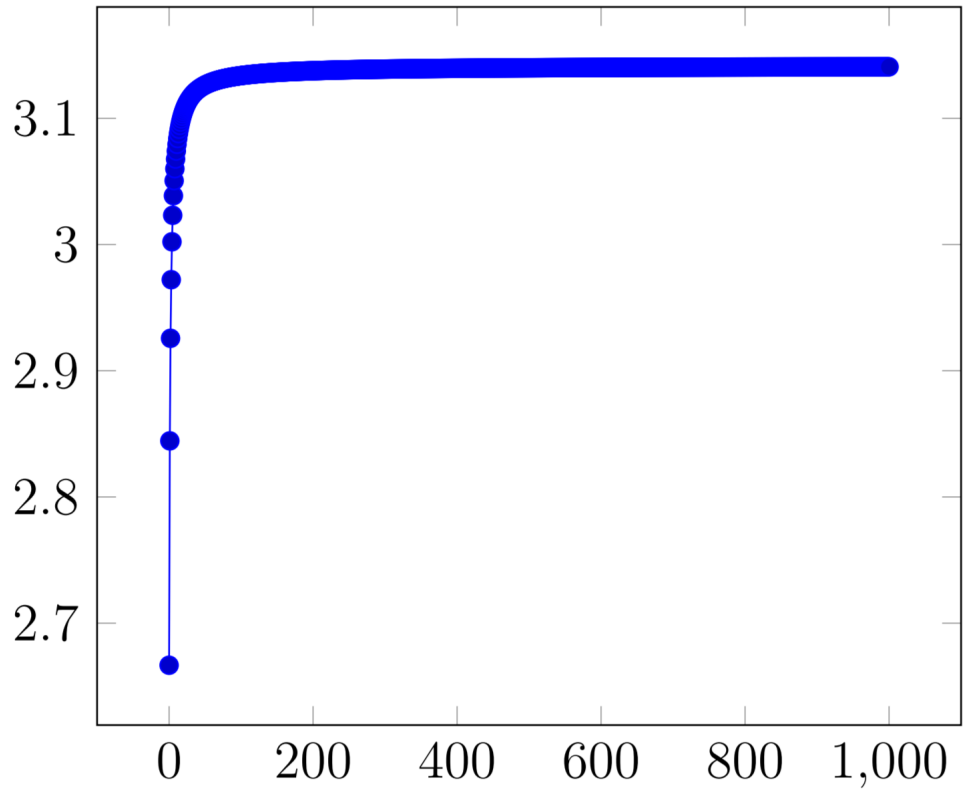

Рис. 5: График зависимости числа π от числа итераций (n)

Рассмотрим разность между истинным значением числа π и полученным нами с помощью формулы (2):

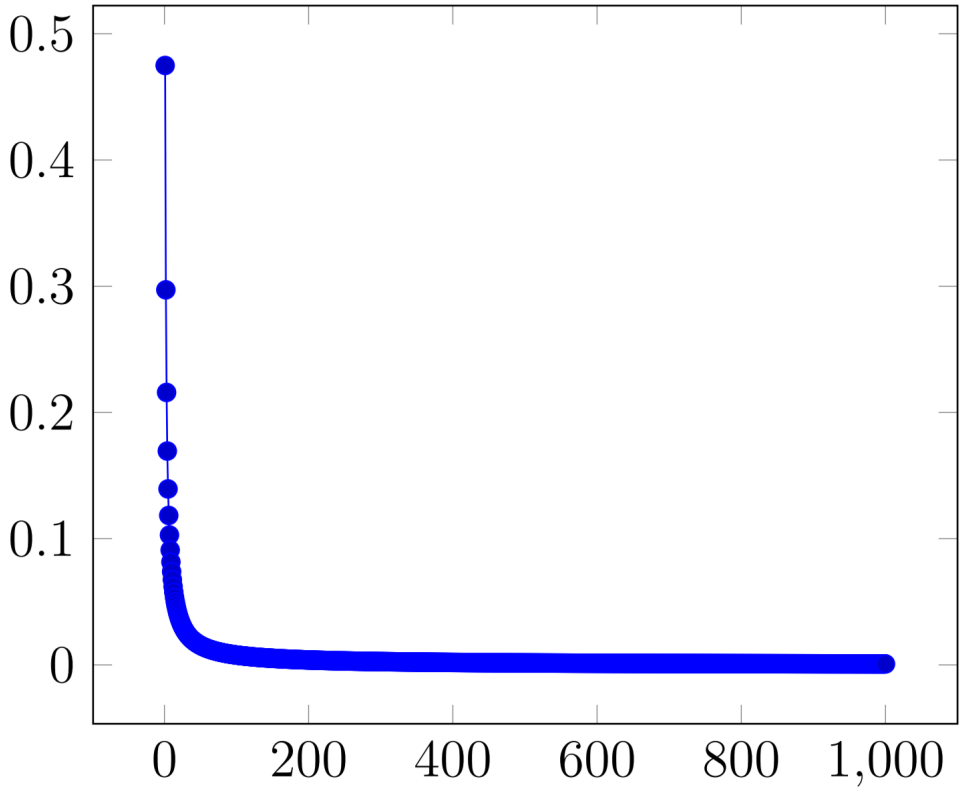

Рис. 6: Разность между истинным значением числа π и полученным нами в зависимости от номера итерации.

Рассмотрим промежуток от 0 до 100 итераций и посмотрим как на нём ведёт себя разность между истиным значением числа π и полученным нами:

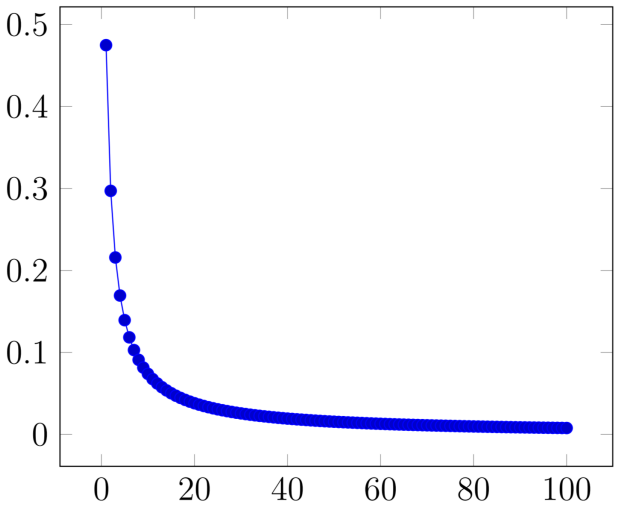

Рис. 7: Разность между истинным значением числа π и полученным нами в зависимости от номера итерации.

Как мы видим на графике разность стремится к нулю.

Рассмотрим промежуток от 400 до 1000 итераций:

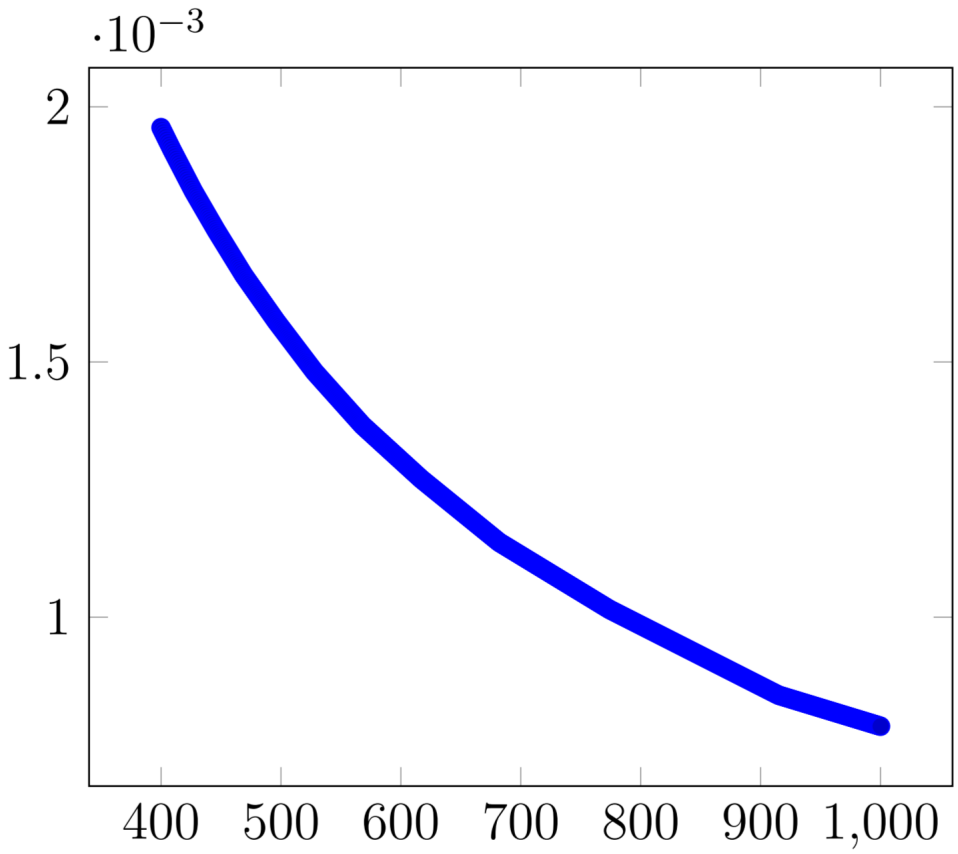

Рис. 8: Разность между истинным значением числа π и полученным нами в зависимости от номера итерации.

Как мы видим разность стремится к нулю справа в тысячных долях.

Следствие: в данном случае погрешность составляет 0.0008.

1.3: Формула Виета.

$$\pi = 2 \cdot \prod_{i=2}^{\infty} \frac{1}{\cos \frac{\pi}{2^i}} \tag{3}$$

Используемый код:

```
#include <iostream>
#include <math.h>
#include <fstream>
using namespace std;
int main() {
    float n, i, j;
    float pi;
    double PI =acos(-1.0);
    ofstream pi_res("1.csv", ios::out);
    while (1){
        pi=1.0;
        std::cout << "PI through the Viett series\n";
        std::cout << "\nEnter the number of iterations: ";
        std::cin >> n;
        std::cout << "Please wait. Running..." << "\n";
        for (i=2;i<n;i++){
            j=1.0*pow(2,i);
            pi *= 1/cos(PI/j);
            pi_res << "[" << i << "," << pi << "], ";
        }
    }
    return 0;

}
```

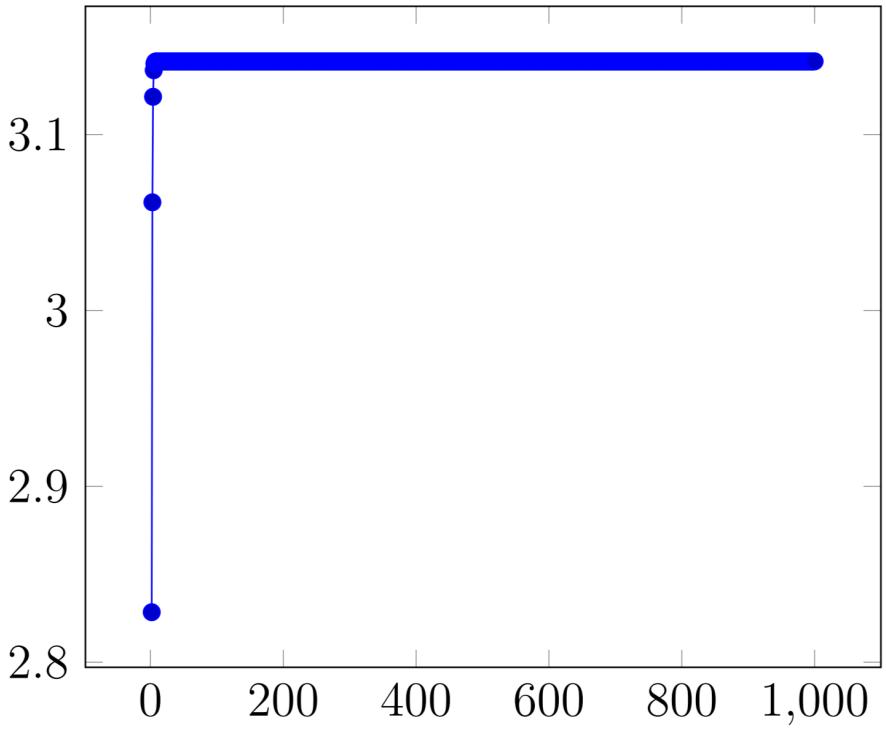

Рис. 9: График зависимости числа π от числа итераций (n)

Рассмотрим разность полученных значений с истинным в полном диапазоне итераций:

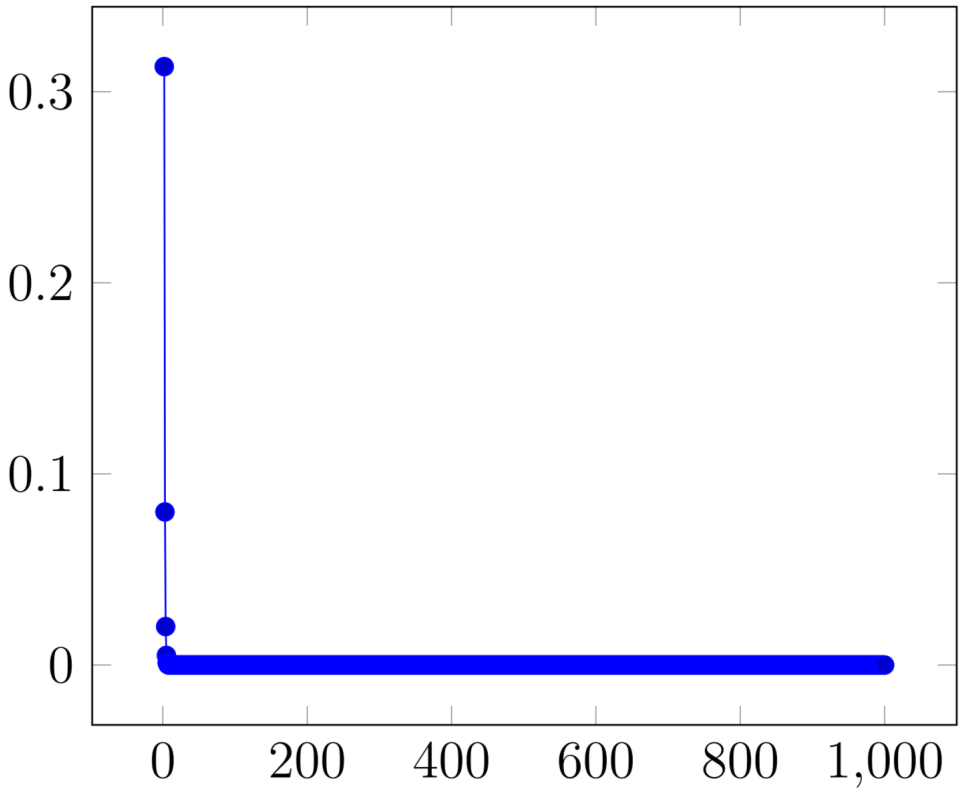

Рис. 10: Разность между истинным значением числа π и полученным нами в зависимости от номера итерации.

Рассмотрим поближе участок в диапазоне 2-30 итераций:

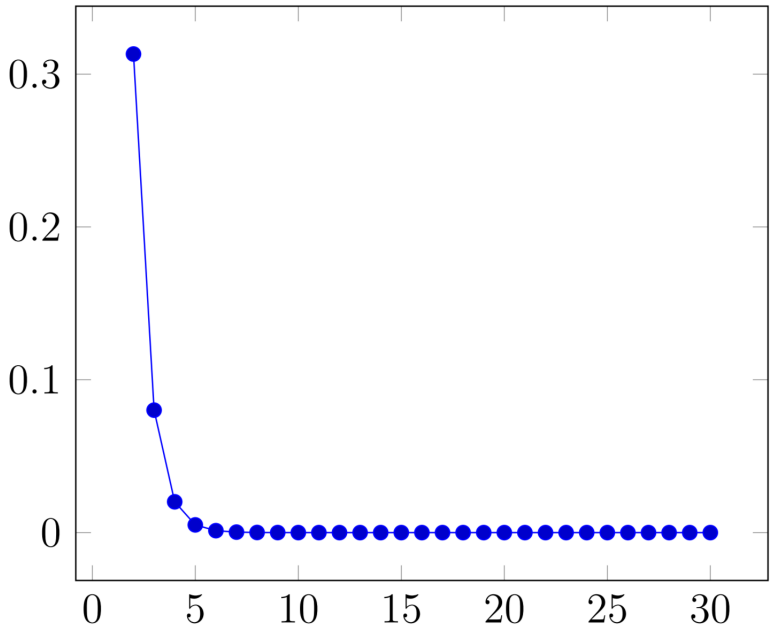

Рис. 11: Разность между истинным значением числа π и полученным нами в зависимости от номера итерации.

А теперь рассмотрим участок от 700 итерации до 1000-ной:

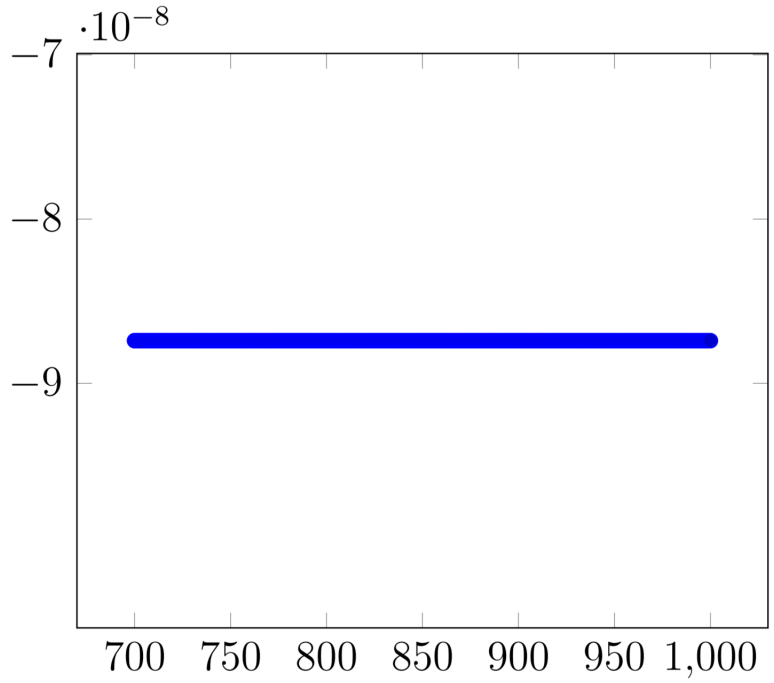

Рис. 12: Разность между истинным значением числа π и полученным нами в зависимости от номера итерации.

Заметим странную вещь: по графику видно, что начиная с некоторого элемента значения числа π, полученного нами, не меняются. Разберёмся - это ошибка вычислений, баг или особенность формулы? Рассмотрим некоторый элемент с индексом i:

$$a_i = \lim_{i\to\infty} \cos\frac{\pi}{2^i} = \cos\lim_{i\to\infty}\frac{\pi}{2^i} = \cos 0 = 1$$

А как известно при домножении на 1 числа не меняются, эт этого наше значение π и не меняется. Вывод - это не ошибка, а особенность формулы, которую мы использовали для вычислений.

Следствие: При 1000 сомножителях точность достигает $1 \cdot 10^7$

1.4: **Преобразование Мандхава.**

$$\pi = 2\sqrt{3} \cdot \sum_{i=0}^{\infty} \frac{(-1)^i}{3^i \cdot (2i+1)} \tag{4}$$

Используемый код:

```
#include <iostream>
#include <math.h>
#include <fstream>
using namespace std;
int main() {
    float n, i, j;
    float pi;
    double PI =acos(-1.0);
    ofstream pi_res("1.csv", ios::out);
    while (1){
        pi=0.0;
        std::cout << "PI through the Mandha's series\n";
        std::cout << "\nEnter the number of iterations: ";
        std::cin >> n;
        std::cout << "Please wait. Running..." << "\n";
        for (i=0;i<n;i++){
            if (i%2==0) j=2*sqrt(3)/(pow(3,i)*(2*i+1));
            else j=(2*sqrt(3)/(pow(3,i)*(2*i+1)))*(-1);
            pi+=j;
            pi_res << "[" << i << ", " << pi << "], ";
    }
   return 0;
}
```

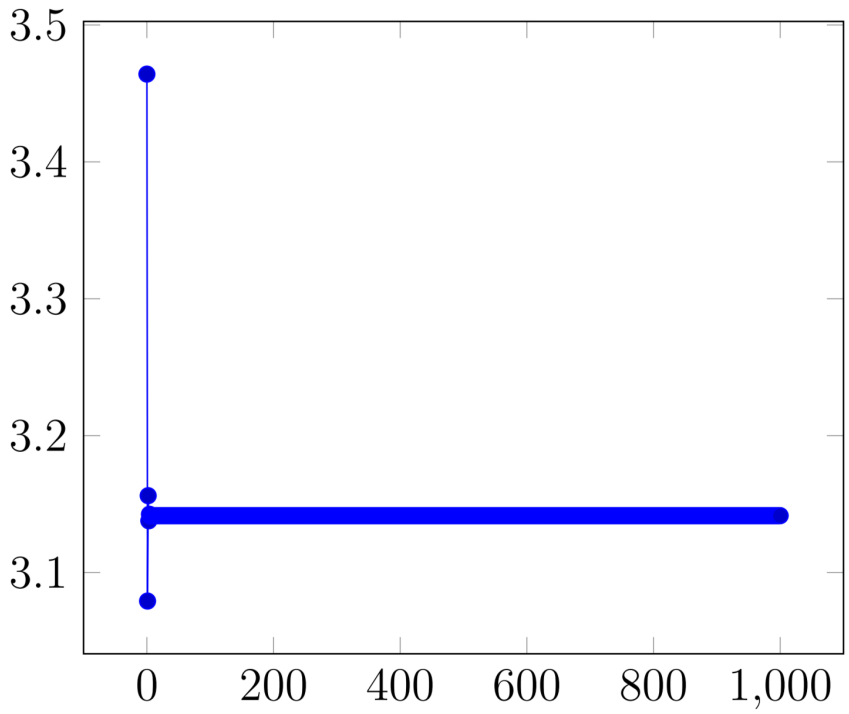

Рис. 13: График зависимости числа π от числа итераций (n)

Разность с истинным значением π для всех итераций на графике:

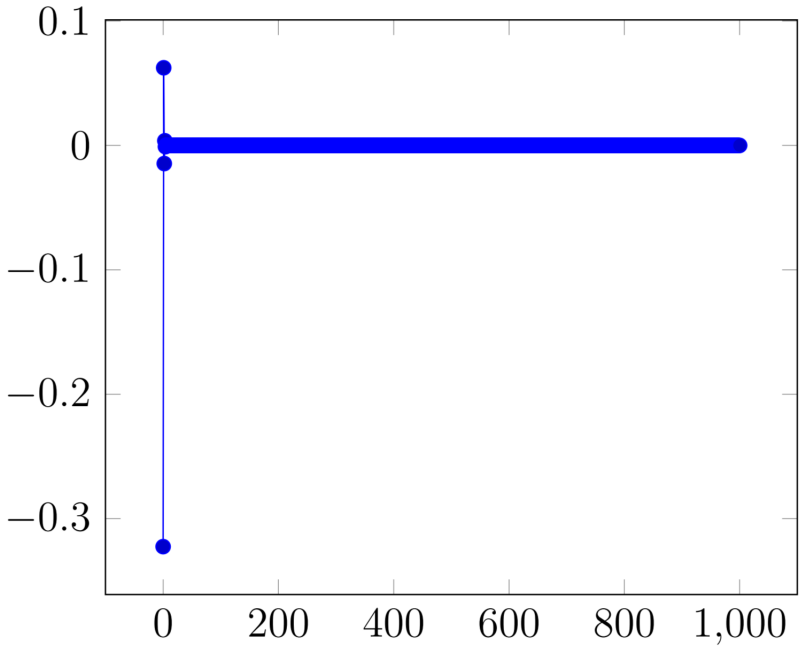

Рис. 14: Разность с истинным значнием в зависимости от числа итераций (n)

Как мы видим, опять значения в какой-то момент перестают меняться. Рассмотрим малый промежуток, число итераций меньше 30:

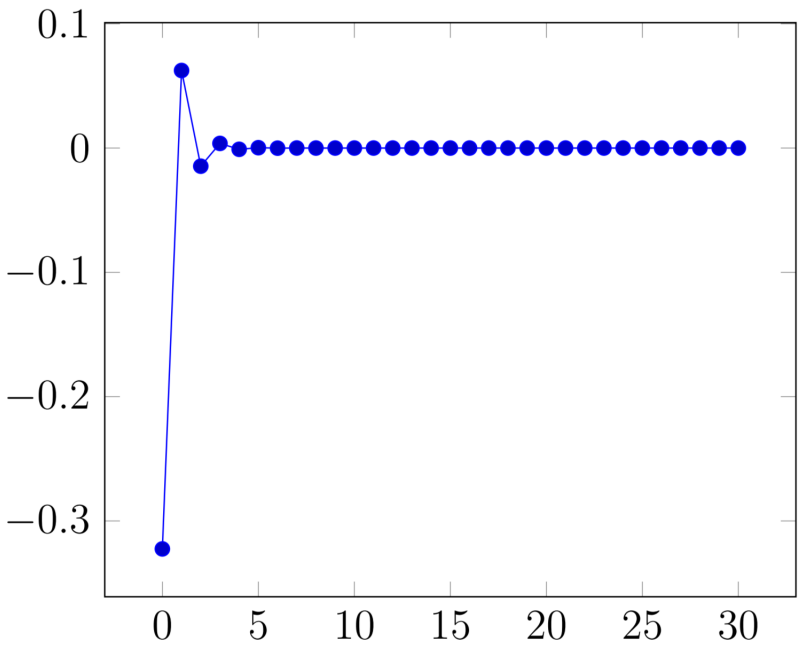

Рис. 15: Разность с истинным значнием в зависимости от числа итераций (n)

Теперь рассмотрим большое число итераций, ведь опять начиная с некоторого слагаемого сумма не меняется:

$$a_i = \lim_{i \to \infty} \frac{(-1)^i}{3^i \cdot (2i+1)} = 0 \Longrightarrow a_i \to 0$$

Это значит, что начиная с некоторой итерации a_i будет равно 0, следовательно сумма перестанет меняться.

Следовательно: Для данного способа нахождения π погрешность измерения составляет около $1 * 10^{-6}$. (для итераций $\to$ 1000)

6. **Задание 5:** Рассчет времени обработки. Рассчитаем время используя стандартную функцию для измерения времени: Используемый фрагмент кода:

```
#include <chrono>
double get_time() {
return std::chrono::duration_cast<std::chrono::microseconds>
(std::chrono::steady_clock::now().time_since_epoch()).count()/1e6;
}
```

По полученным данным составим графики зависимости. Для каждой формулы, описанной выше, оценим время, за которая эта формула доходит до 10 знака после запятой в числе π.

Время для формулы Лейбница:

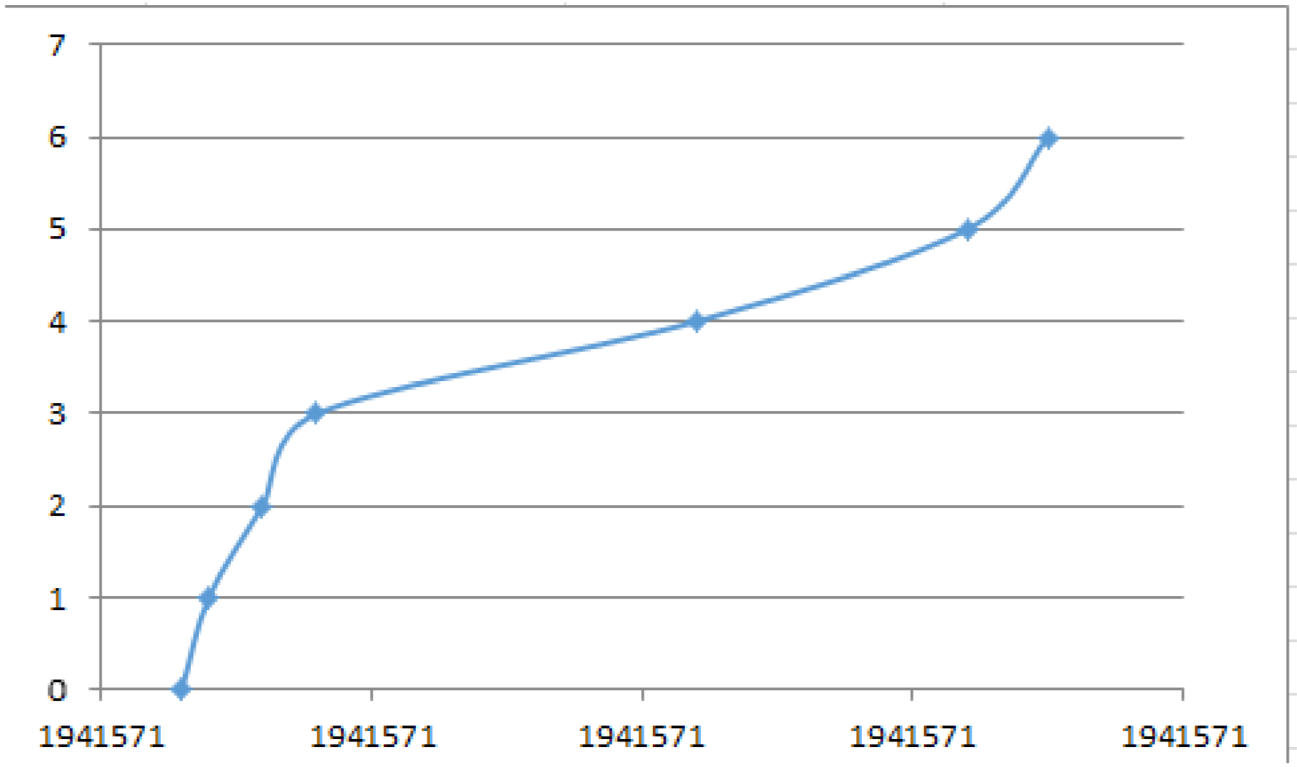

Рис. 16: Время рассчёта 10 знакп после запятой в числе π

Время для формулы Валлиса:

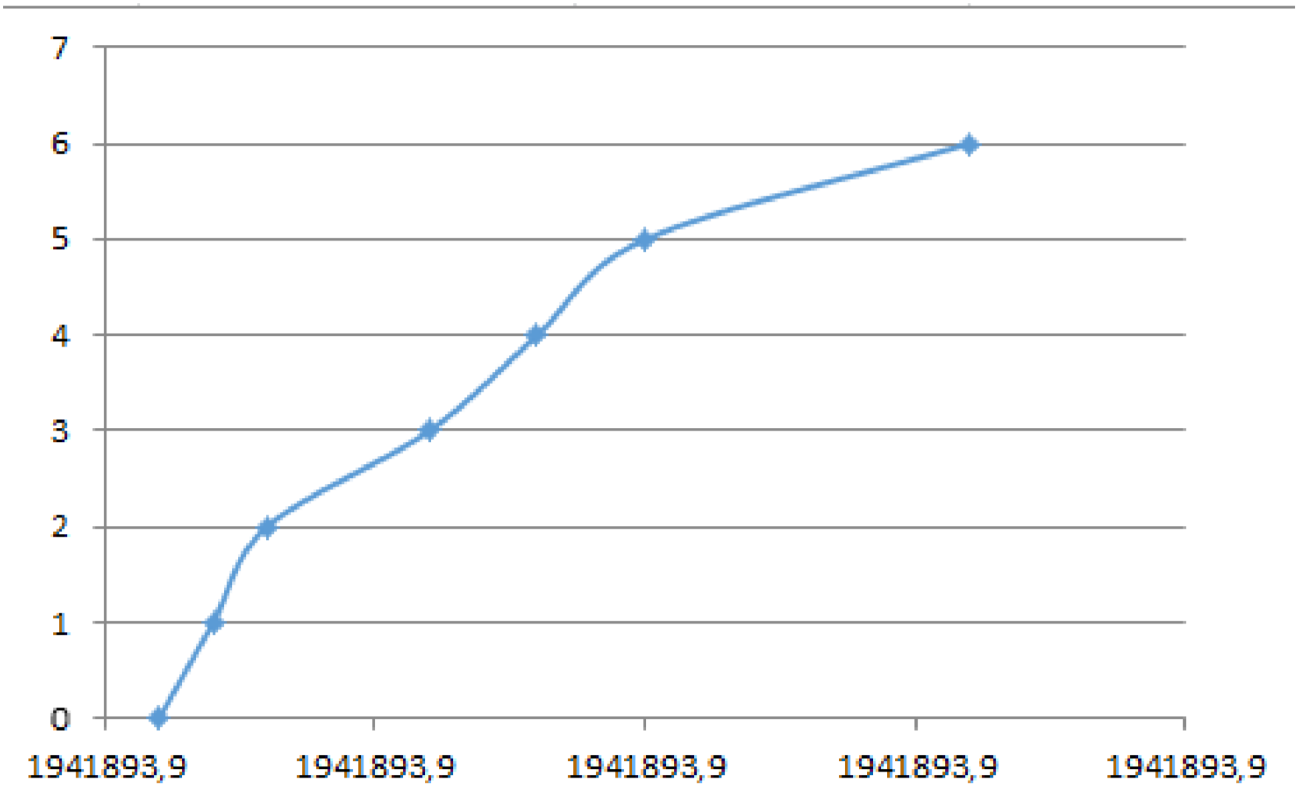

Рис. 17: Время рассчёта 10 знакп после запятой в числе π

Время для формулы Виетта:

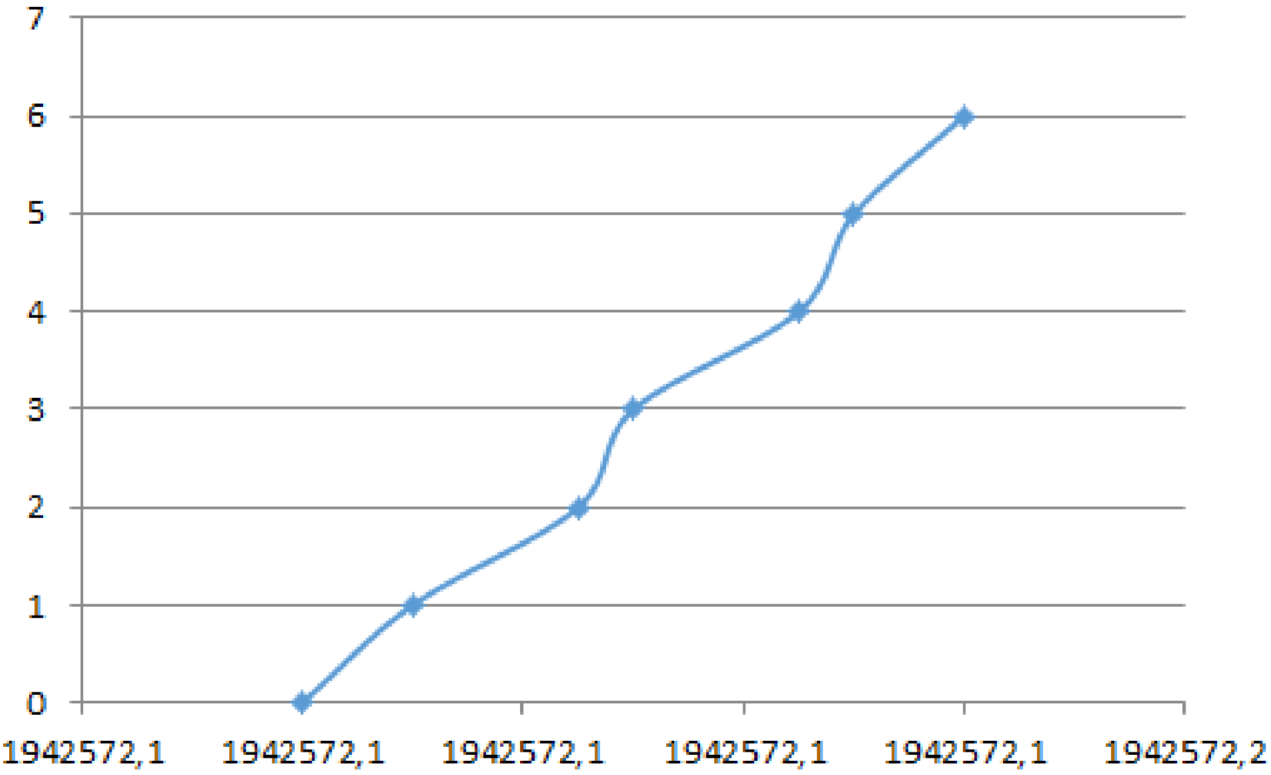

Рис. 18: Время рассчёта 10 знакп после запятой в числе π

Время для формулы Мандхава:

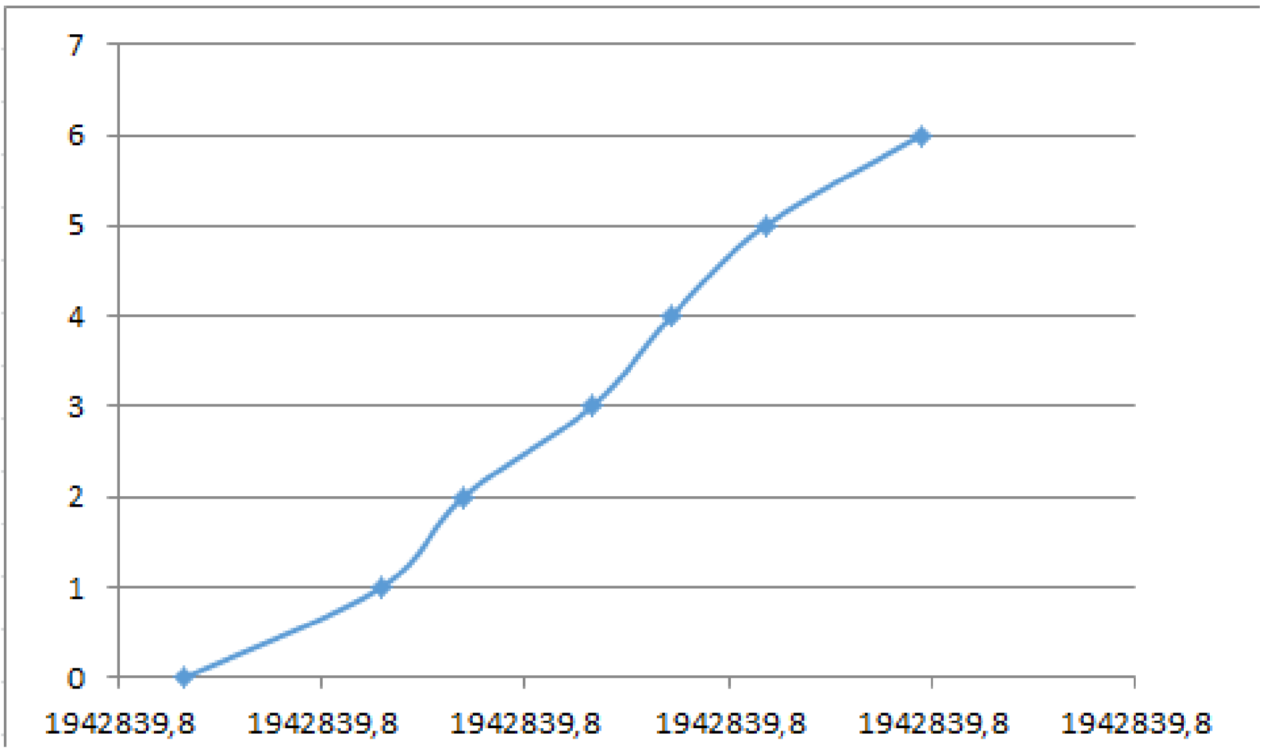

Рис. 19: Время рассчёта 10 знакп после запятой в числе π

4 Вывод

Мы исследвали то, как данные типа $float$ хранятся в памяти компьютера и обнаружили несколько интересных и важных в дальнейшем особенностей и ошибок. Научились визуализировать данные с помощью $Python$ и научились анализировать числа с плавающей точкой.