Лабораторная работа №1
Хранение переменных в С++
Изучение особенностей переменной float

Информатика, 1-й семестр

Дудин Иван Юрьевич Б03-304

30 сентября 2023г.

1. Введение

Цель:

Изучение и оценка стандартного формата чисел с плавающей точкой в языке программирования С++. Получение фундаментального понимания того, что происходит с вещественными числами при выполнении операции над ними в языке программирования С++.

Метод изучения:

Экспериментальное выполнение поставленных в процессе работы заданий в компиляторе языка программирования С++ “CodeBlocks”. Анализ открытых источников и форумов, посвященных языку программирования, из интернета, с целью получения новой информации и ее последующего использования при выполнении лабораторной работы. Визуальное отображение результатов эксперимента в форме графиков в прямоугольной системе координат, построенных в компиляторе “PyCharm” языка программирования Python.

2. Основная часть

Задание 1: Перевод числа из десятичной системы счисления в двоичную

Основная задача: преобразование числа формата *unsigned int* в *binary* при помощи побитового сдвига числа *char*

Описание работы: изначально, было необходимо написать код, который переводит число из десятеричной системы в двоичную с помощью математических операций. Однако данный код неоднократно был мною прописан в контестах №1-3, поэтому моя цель состояла в написании такого кода, который либо вовсе не использует, либо выполняет только основные математические операции (сложение или умножение), но при этом всем выводит двоичную запись любого заданного пользователем десятеричного числа.

Рисунок 1: Первая версия перевода в двоичную систему

```
#include <iostream>
using namespace std;
union un {
    int i;
    float f;};

void binary(unsigned int n)
{
    for (int i=1; i<=32; i++)
        {if ((n<<1)>>1==n)
            {cout<<0;}
        else
            {cout<<1;}
        n=n<<1; }
}
int main()
{
    un t;
    cin >> t.f;
    binary(t.i);
    cout << endl;
}
```

Рисунок 2: Вторая версия перевода в двоичную систему

```
#include <iostream>
using namespace std;
int main()
{
    unsigned int s;
    cin>>s;
    for(int i=0;i<32;i++)
    {
        if (s==((s<<1)>>1))
            cout<<"0";
        else
            cout<<"1";
        s=s<<1;
    }
}
```

Оба кода выполняют корректный перевод в двоичную запись. Первая версия содержит в себе функцию перевода числа и строку *union*, из-за чего она более удобна для написания дальнейшей программы.

Вторая же версия не содержит себе ничего из этого, но при этом все она более компактна.

Таблица 1: Проверка первого задания на практике

Введенное значение	Полученный результат	Ожидаемый результат	
0	000000000000000000000000 00000000	000000000000000000000000 00000000	+
1	000000000000000000000000 00000001	000000000000000000000000 00000001	+
343	000000000000000000000001 01010111	000000000000000000000001 01010111	+
235467	000000000000001110010111 11001011	000000000000001110010111 11001011	+
f	000000000000000000000000 00000000	NaN	±
1000000000	001110111001101011001010 00000000	001110111001101011001010 00000000	+
-7	111111111111111111111111 11111001	111111111111111111111111 11111001	+
4294967296	111111111111111111111111 11111111	NaN	-
-1	111111111111111111111111 11111111	111111111111111111111111 11111111	+

Большинство результатов оказались ожидаемыми, однако некоторые сработали не так, как планировались. При введении символа в числовую переменную мною ожидалось ошибка компилятора, однако он обнулил эту переменную. При введении числа, большего чем допустимого значения *unsigned int,* я ожидал ошибку компилятора, однако он считал его числом выше порога на “-1” и поэтому вывел двоичный код для числа “-1”. Таким образом можно сделать вывод, что при вводе отрицательного числа, программа прибавляет слева к двоичной записи числа “1”, обнуляя последующие знаки. После чего из данного значения вычитает модуль отрицательного числа и выводит его на экран. Аналогично работает с теми числами, которые больше порогового значения *unsgined int,* только вычитается не просто число, а модуль разности максимального значения “4294967295”(2^{32}-1) и введенного числа.

Задание 2: Переполнение мантиссы

Основная задача: воспроизвести переполнение мантиссы и оценить полученные результаты

Описание работы: на основе перевода числа в двоичную запись были изучены результаты для степеней числа 10. Пример кода используемого кода изображен на рисунке 3

В рисунке 4 представлены результаты для цикла до сороковой степени числа 10.

Были получены весьма любопытные результаты:

До значения 10^{10} результаты были предсказуемы и являются корректными, после чего со значениями начинается происходить что-то непонятное. Значение переменной *float* становится не равной степени числа 10, но все еще является сильно приближенно к ней. Это связано с дискретностью диапазона float – в нем представимы далеко не все вещественные числа. Даже те числа, которые в десятичном разложении выглядят «нормально» - степени десятки или та же 0.2 – могут быть непредставимы во float. А значит, все вычисления у вас будут идти с какой-то погрешностью, потому что заранее предсказать, где там обрежется какой хвост у числа, практически невозможно. Погрешность невелика, но она может накапливаться и полностью менять расчет при больших количествах операций.

С двоичной записью числа вопросов практически не возникает. При значении выше порога программа преобразует число в разность, однако если из порогового значение *a* вычесть число $k >> 2a$, их разность превзойдет даже отрицательный порог двоичной записи, и компилятор, ожидаемо, выведет 0.

Также стоит отметить, что при значение больше чем 10^{38} переменная вида *unsigned int* достигает своего порога и не считывает переменную вовсе.

Рисунок 3: Пример кода для изучения свойств мантиссы

```
#include <iostream>
using namespace std;
union un {
    int i;
    float f;
};

void binary(unsigned int n)
{
for (int i=1; i<=32; i++) {
     if ((n<<1)>>1==n)
{
    cout<<0;
}
else
{
    cout<<1;
}
 n=n<<1; }
}
int main()
{
    cout << fixed;
    cout.precision(2);
    un t,l;
    l.f=1.0;
    int n=1;
    for(int i=0;i<40;i++){
        l.f=l.f*10;
        cout<<(l.f);
        cout<<endl;
        binary(l.f);
        cout<<endl;
        cout<<endl;
    }
}
```

Рисунок 4: Результаты 2-го задания для чисел от 10 до 10^{40}

10.00

00000000000000000000000000001010

100.00

00000000000000000000000001100100

1000.00

00000000000000000000001111101000

10000.00

00000000000000000010011100010000

100000.00

00000000000000011000011010100000

1000000.00

00000000000011110100001001000000

10000000.00

00000000100110001001011010000000

100000000.00

00000101111101011110000100000000

1000000000.00

00111011100110101100101000000000

10000000000.00

01010100000010111111001000000000

99999997952.00

01001000011101101110000000000000

999999995904.00

11010100101001010000000000000000

9999999827968.00

01001110011100000000000000000000

100000000376832.00

00010000100000000000000000000000

999999986991104.00

10100100000000000000000000000000

10000000272564224.00

10000000000000000000000000000000

99999998430674944.00

00000000000000000000000000000000

999999984306749440.00

00000000000000000000000000000000

9999999980506447872.00

00000000000000000000000000000000

100000002004087734272.00

00000000000000000000000000000000

1000000020040877342720.00

00000000000000000000000000000000

9999999778196308361216.00

00000000000000000000000000000000

99999997781963083612160.00

00000000000000000000000000000000

1000000013848427855085568.00

00000000000000000000000000000000

10000000714945030854279168.00

00000000000000000000000000000000
100000002537764290115403776.00
00000000000000000000000000000000
1000000062271131048573140992.00
00000000000000000000000000000000
10000000622711310485731409920.00
00000000000000000000000000000000
100000001504746621987668885504.00
00000000000000000000000000000000
1000000015047466219876688855040.00
00000000000000000000000000000000
9999999848243207295109594873856.00
00000000000000000000000000000000
100000003318135351409612647563264.00
00000000000000000000000000000000
1000000071866979741764260066230272.00
00000000000000000000000000000000
10000000409184787596297531937521664.00
00000000000000000000000000000000
100000004091847875962975319375216640.00
00000000000000000000000000000000
1000000040918478759629753193752166400.00
00000000000000000000000000000000
10000000567641112624826207124609564672.00
00000000000000000000000000000000
100000006944061726476491472742798852096.00
00000000000000000000000000000000
inf
00000000000000000000000000000000
inf
000000000000000000000000000000oo

Задание 3: Создание бесконечного цикла

Основная задача: Создание бесконечного цикла на базе ошибки вычисления числа *unsigned int.*

Описание работы: в задании 2 было замечено, что при больших значениях *unsigned int* переменная, как бы, “ломается” и присваивает себе не абсолютно точные, а лишь приближенные значения. Поэтому, из-за данной особенности может возникнуть ситуация, когда цикл не прерывается, потому что переменная не достигает определенного значения. Пример подобного кода изображен на рисунке 5.

Рисунок 5: Пример бесконечного цикла, вызванного особенностями переменной *unsigned int*

```
#include <iostream>
using namespace std;
union un {
    int i;
    float f;
};

void binary(unsigned int n)
{
for (int i=1; i<=32; i++) {
     if ((n<<1)>>1==n)
{
    cout<<0;
}
else
{
    cout<<1;
}
 n=n<<1; }
}
int main()
{
    cout << fixed;
    cout.precision(2);
    un t,l;
    int k;
    k=1;
    l.f=2130000000.0;
    for(;l.f<(l.f+1);l.f++)
        {cout<<k<<")everlasting cycle"<<endl;
        k++;
        if(k==6)
            break;
        }

}
```

Рисунок 6: Появление бесконечного цикла

1)everlasting cycle
2)everlasting cycle
3)everlasting cycle
4)everlasting cycle
5)everlasting cycle
…

Как можно заметить, код, изображенный на рисунке 5, должен иметь всего один шаг в цикле, ведь счетчик увеличивается за раз на единицу, а ограничителем стоит число, большее счетчика на единицу. Но в результате данного значения не достигается, и код бы выводил *cout* бесконечно (если бы не *break).* Причина данного явления отображена в пункте “Задание 2” и заключена в особенности переменной вида *unsigned int.*

Задание 4: График числа π

Основная задача: Изучение способов нахождения числа π, попытка экспериментального подтверждения расчетов с помощью языка программирования C++, построение графика зависимости значения числа π от количества итераций.

Описание работы: В былые времена многие математики пытались найти способ вычислить число π с наибольшей точностью. И для этого придумывались громоздкие формулы, производились сложнейшие вычисления и, стоит отметить, достигались довольно точные значения константы. Однако сейчас можно просчитать формулы всего за несколько минут с помощью того же самого языка программирования C++.

Стоит отметить, что в данном пункте все значения и все шаги итераций выполнялись на языке программирования C++. В Python были только построены графики.

4.1: Серия Мадхавы

Первая изученная мною итерационная формула - это серия Мадхавы для числа π.

$$\pi = \sqrt{12}\left(1 - \frac{1}{3\cdot 3} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right)$$

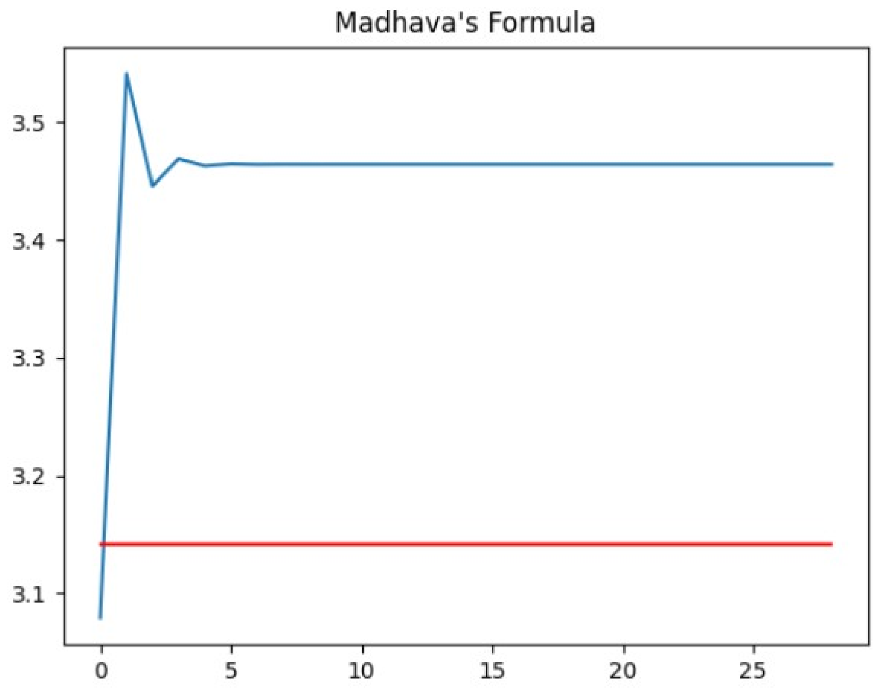
На рисунке 7 изображен код для обработки данной формулы.

График зависимости значения числа от итераций изображена на рисунке 8, где синяя ломанная - значение переменной, а красная - ожидаемая константа. Здесь сразу же можно заметить несостыковку - полученный результат очень сильно отличается от реального значения числа π. Причину этого я вижу в переменной *float*. В процессе вычисления складываются и умножаются числа с большим количеством цифр после запятой. Ранее уже было замечено такое свойство у переменных *unsigned int,* когда переменной присваивается не абсолютно точное, а приближенное значение из-за чего возникает погрешность. В данном же случае данная несостыковка оказалась крайне значимой, что можно увидеть на рисунке 8.

Рисунок 7: Код для выполнения уравнения Мантиссы

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>

using namespace std;

int main()
{
    ofstream f("methodMadhava.txt", ios::out);
    float abv;
    float abc;
    abv=1.0;
    abc=0.0;
    int above=1;
    for (float i=1;i<30;i=i+1){
        abv=1.0+pow(-1,i)/((1.0+2.0*i)*pow(3,i));
        cout<<"Number of iteration:"<<above<<endl;
        cout<<fixed<<setprecision(10)<<pow(12.0,1.0/2.0)*abv<<endl;
        f<<pow(12.0,1.0/2.0)*abv<<endl;
        above++;
    }

}
```

Рисунок 8: График зависимости по уравнению Мантиссы в C++



4.2 Формула Валлиса

Далее я рассмотрел формулу Валлиса для нахождения значения числа π.

$$\prod_{n=1}^{\infty} \frac{(2n)(2n)}{(2n-1)(2n+1)} = \frac{\pi}{2}$$

На рисунке 9 изображен код для выполнения итераций согласно формуле Валлиса.

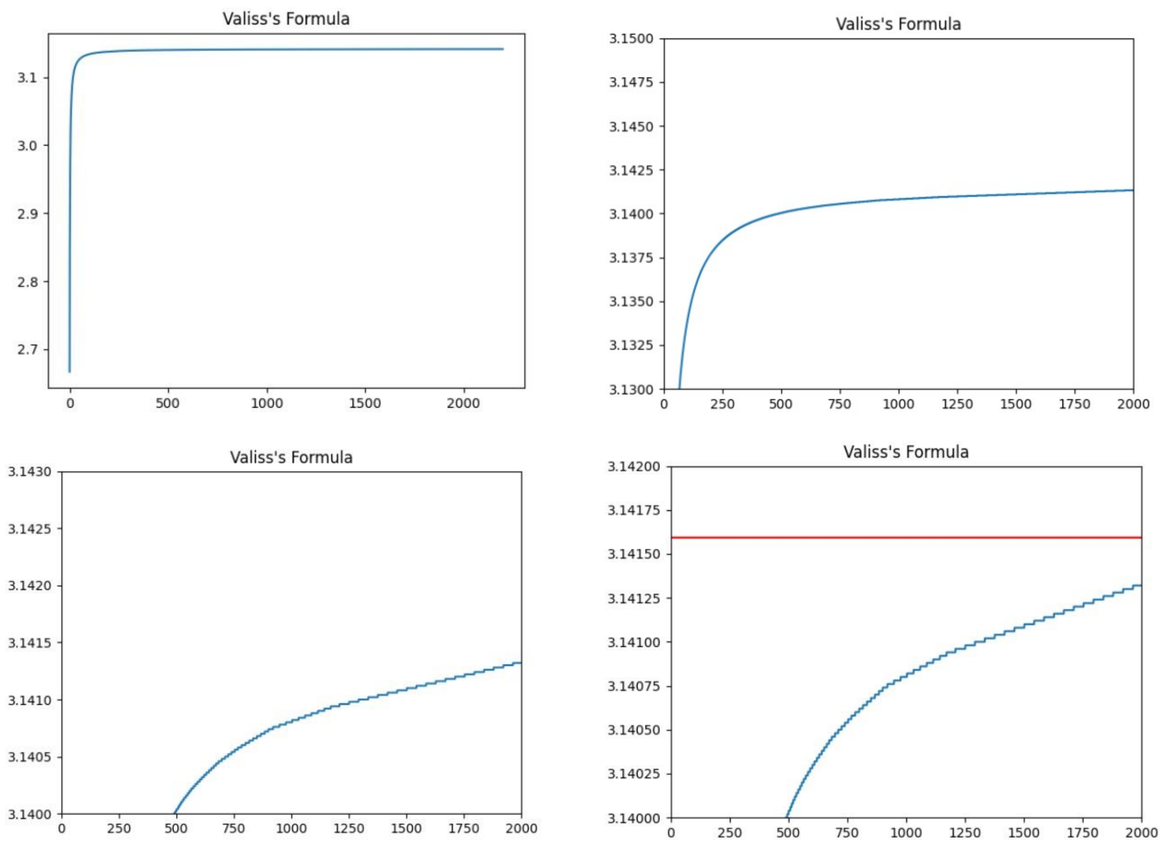
Рисунок 9: Формула Валлиса в C++

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>

using namespace std;

int main()
{
    ofstream f("0methodValiss.txt", ios::out);
    float abv;
    float abc;
    abv=1.0;
    abc=0.0;
    int above=1;
    for (float i=1;i<2200;i=i+1){
        abc=(4.0*i*i)/(4.0*i*i-1);
        abv=abv*abc;
        cout<<"Number of iteration:"<<above<<endl;
        cout<<fixed<<setprecision(10)<<abv*2<<endl;
        f<<abv<<",";
        above++;
    }
}
```

На рисунках 10-13 можно увидеть зависимость полученного значения от числа итераций

Как можно увидеть, с данной последовательностью не возникает ситуации подобной последовательности Мадхавы. Здесь значение на протяжении большого количества итераций сходится к числу π, однако при очень больших количествах шагов полученное значение калибруется крайне малым значениями, которые переменная *unsigned int* лишь приближает к точному, из-за чего последовательность “притупляется” и по итогу сводится к не совсем точному значению 3,14135

Рисунки 10-13: Графики зависимости значения числа от количества итераций для формулы Валисса в различных масштабах



В качестве эксперимента было замерено время, при котором получаются те или иные цифры числа π. Они отмечены в таблице 2, и стоит отметить, что результаты довольно любопытные.

Таблица 2: Время, необходимое для получения той или иной цифры числа π по формуле Валисса

Необходимая точность значения	Количество итераций	Затраченное время
3.1	19	0.000 с
3.14	493	0.000 с
3.141	1315	0.000 с

Время для каждого числа - 0.000с. Опыт был проведен неоднократно, однако результат оказался тем же. Объяснить данный итог я могу следующим образом: из-за того, что для метода Валлиса выполняются простейшие операции над числами, их итог выводится крайне быстро, из-за чего компилятор просто не в силах вывести настолько малое значение времени, поэтому и выводится значение 0.

Для вывода времени использовалась команда: "Time taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC при использовании import <time.h>

4.3 Формула Виета

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{2}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \cdots$$

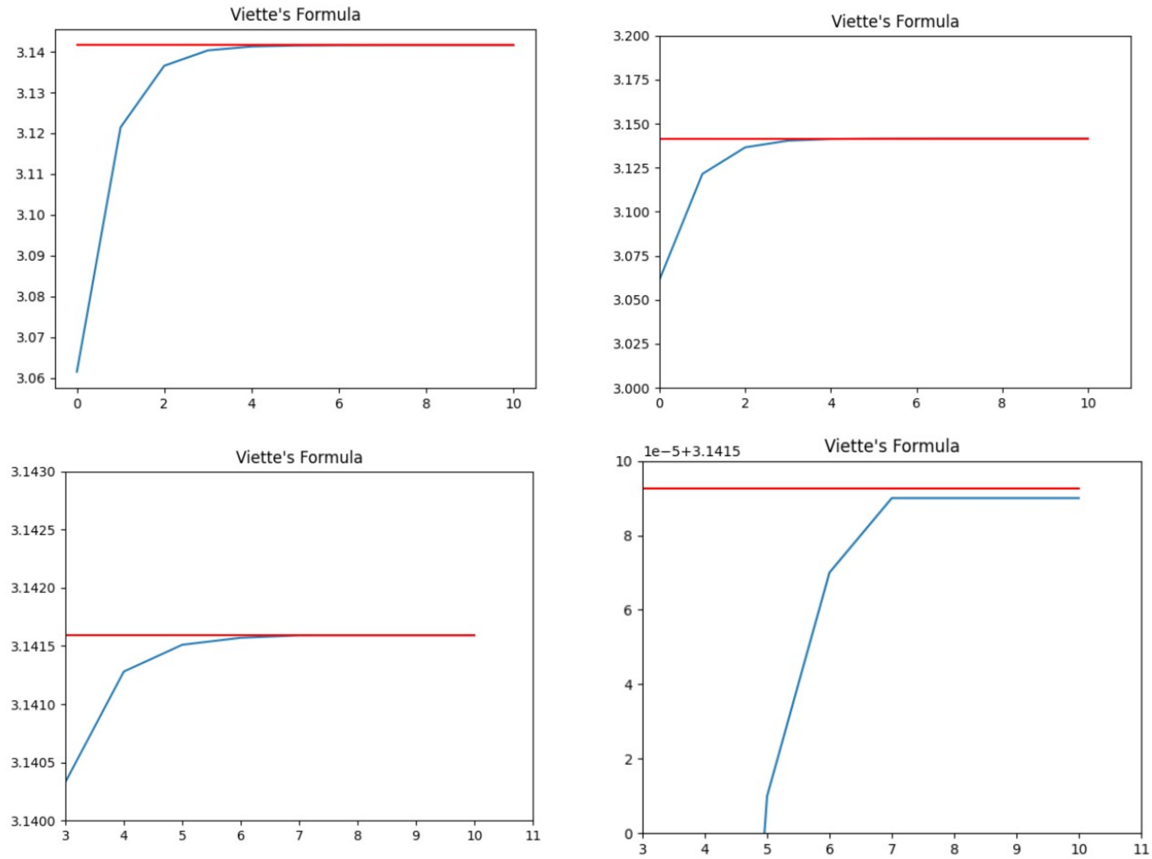
На рисунке 14 изображен код для вычисления числа π по формуле Виета.

Рисунок 14: Код для нахождения числа π по формуле Виета

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>

using namespace std;

int main()
{
    ofstream f("methodViette.txt", ios::out);
    float abv;
    float abc;
    abv=pow(2,1.0/2);
    abc=0.0;
    float itog;
    itog=2.0/pow(2,1.0/2);
    float mi;
    mi=-1.0;
    float c;
    c=0.0;
    int above=1;
    for (float i=1;i<12;i=i+1){
        abc=pow(2+abv,1.0/2);
        abv=abc;
        itog=itog*(2/abv);
        cout<<"Number of iteration:"<<above<<endl;
        cout<<fixed<<setprecision(10)<<itog*2<<endl;
        f<<itog*2<<endl;
        mi=-1.0;
        c=c+1.0;
        above++;
    }

}
```

На рисунках 15-18 можно увидеть графики зависимости значения числа от количества итераций

Рисунки 15-18: График зависимостей значения числа от количества итераций по формуле Виета в различных масштабах



Можно сразу же отметить, что формула Виета получает число π при крайне малом количестве итераций, это можно увидеть в таблице 3:

Таблица 3: Время, необходимое для получения той или иной цифры числа π по формуле Виета

Необходимая точность значений	Количество итераций	Затраченное время
3.1	2	0.000 с
3.14	4	0.000 с
3.141	5	0.000 с
3.1415	6	0.000 с
3.14159	8	0.000 с

Здесь повторилась та же ситуация, что наблюдалась для формулы Валлиса. Однако точное значение получается при очень малом количество итераций, после чего она сходится к 3.14159 и далее перестает давать точное значение. А так как малое количество итераций занимает крайне малое время, компилятор выводит значение 0.000 сек.

4.4 Формула Лейбница

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

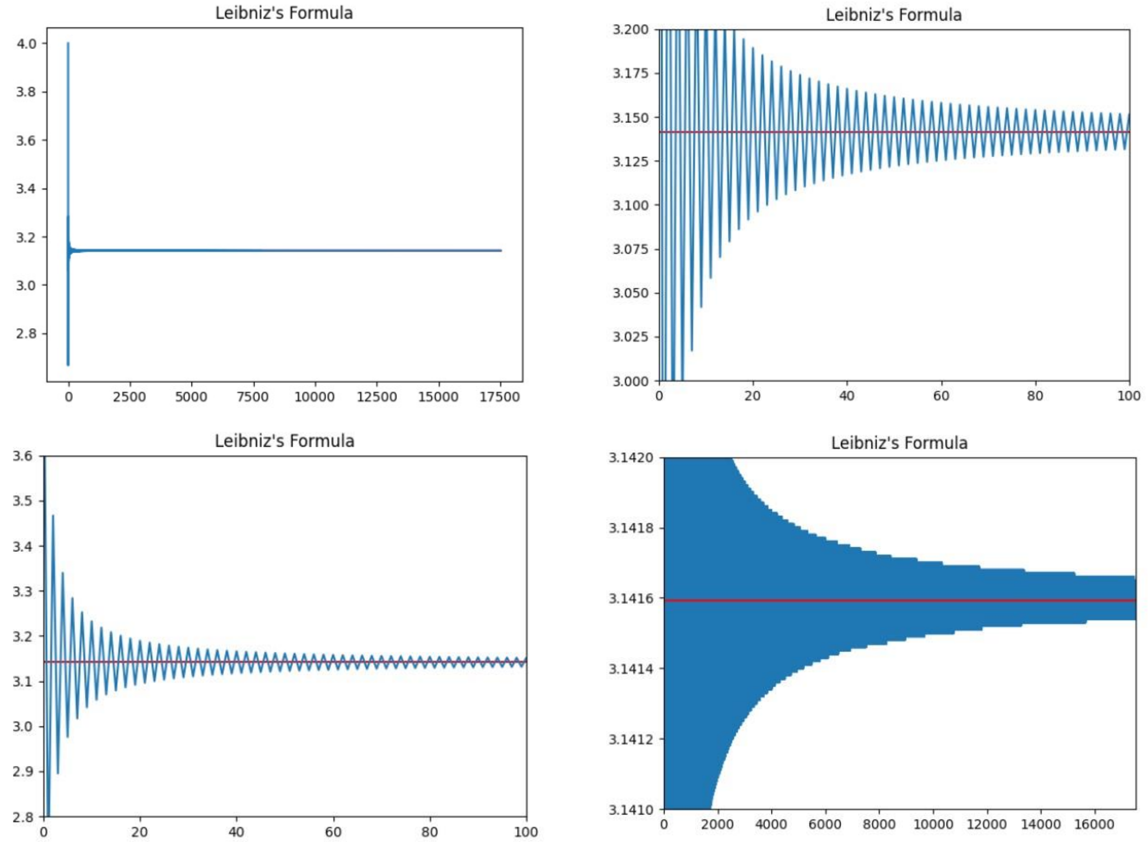
На рисунке 19 изображен код для нахождения числа π по методу Лейбница.

Рисунок 19: Код для нахождения числа π по формуле Лейбница

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>

using namespace std;

int main()
{
    ofstream f("methodLeibniz.csv", ios::out);
    float abv;
    float abc;
    abv=0.0;
    abc=0.0;
    float mi;
    mi=-1.0;
    float c;
    c=0.0;
    int above=1;
    for (float i=0;i<100000;i=i+2){
        for(float k=0;k<=c;k=k+1.0){
            mi=mi*(-1.0);
        }
        abc=4/(i+1)*mi;
        abv=abv+abc;
        cout<<"Number of iteration:"<<above<<endl;
        cout<<fixed<<setprecision(10)<<abv<<endl;
        f<<abv<<endl;
        mi=-1.0;
        c=c+1.0;
        above++;
    }
}
```

На рисунках 20-23 можно увидеть графики зависимости значения числа от количества итераций

Рисунки 20-23: График зависимостей значения числа от количества итераций по формуле Лейбница в различных масштабах



Используя формулу Лейбница, компилятор выводит каждый раз значения, которые колеблются около числа π. Однако для этого необходимо большое число итераций и в конечном счете выводится чередование 3.14162, 3.14158. На таблице 4 представлены время и количество и итераций для этого:

Таблица 4: Время, необходимое для получения той или иной цифры числа π по формуле Лейбница

Необходимая точность значений	Количество итераций	Затраченное время
3.1	19	0.013 с
3.14	119	0.034 с
3.141	1696	0.516 с
3.1415	10200	3.215 с

Можно заметить, что в отличие от предыдущих формул, здесь компилятору требуется какое-то время для получения результата.

Таким образом можно сделать вывод, что наиболее точный и быстрый способ найти число π - это формула Виета, она получает значение с точностью до 5 цифры после запятой все за 8 итераций. Если же необходимо получить красивый график, демонстрирующий то, как последовательность стремится к значению числа π, то лучше всего использовать формулу Лейбница. Однако следует помнить, что переменные вида *unsigned int* и *float* при больших или наоборот малых значениях выдают не абсолютно точный, а приближенный результат, из-за чего выдается не совсем точное, а лишь приближенное значение, поэтому на определенном шаге значение числа престает меняется и принимает лишь приближенное к числу π значение.

3. Подведение итогов

Таким образом, изучив особенности переменных вида *unsigned int* и *float* можно констатировать следующее:

1. Данные переменные удобны для работы и выполнения математических операций, если их значения не является очень большим или очень малыми
2. При очень малых или очень больших значениях переменных она принимают не абсолютно точные, а лишь приближенные значения. Это не вызывает проблем для обычного пользователя, но если необходимо с данными числами провести какие-либо операции, требующие абсолютной точности, то с этим могут возникнуть определенные проблемы из-за появления такой погрешности у переменных

Язык программирования С++ позволяет проводить очень громоздкие вычисления за малый промежуток времени, в этом он удобен для создания, например, каких-либо игр. Если же говорить о математических расчетах, опять же, требующих абсолютной точности, то по моему мнению, к плюсам в этом деле нужно относиться с крайней осторожностью.