Лабораторная работа по числам с плавающей точкой

Королев Дмитрий Алексеевич

26 октября 2025 г.

1 Введение

В этой работе я разобрался в том, как устроено хранение чисел с плавающей точкой в C++ Посмотрели на переполнение мантиссы, на представление float в двоичном представлении.

2 Ход работы

2.1 Двоичное представление

Посмотрим на двоичное разложение чисел с плавающей точкой. Например 123.125

Разобъем на целую и дробную части - 123 - целая, имеет представление 1111011

дробная часть имеет представление 0.001

.125 = .125

.125*2 = .25

.25*2 = .5

.5*1 = 1

Запишем все в одну строку: 1111011.001

Сдвинем точку влево в край: 1.111011001. Тогда степень - 6 => в степень запишется 01111 + 110 = 10000101, в мантиссе будет 11101100100000000000000.

В этом и можно убедиться, запустив код.

2.2 Переполнение мантиссы

Т.к. у мантиссы ограниченное количество битов, при больших числах целая часть чисел будет обрезаться, а дробная часть будет храниться совсем неточно. Можно в этом убедиться:

Увеличивая число в 10 раз, мы придем к тому моменту, когда мантисса кончится, и целая часть не вся поместится в мантиссу.Тогда числа перестанут увеличиваться.

10000000000.00 — 0 10100000 00101010000001011111001

99999997952.00 — 0 10100011 01110100100001110110111

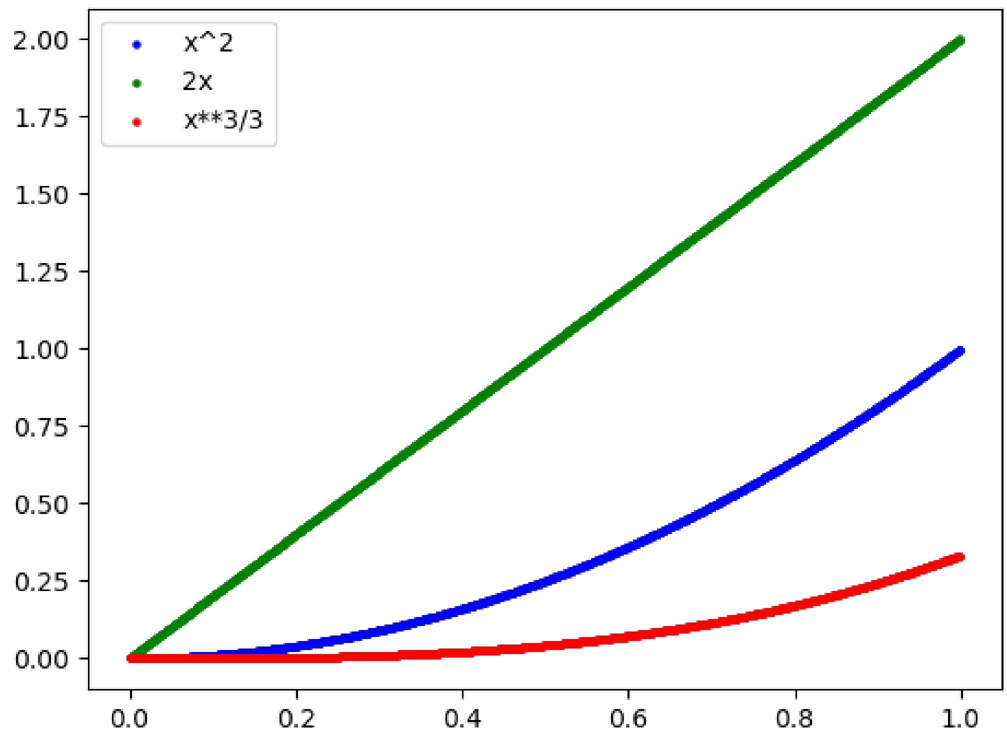
Также, можно написать простейший кусочек кода, который вроде должен закончится, но на самом деле здесь будет бесконечный цикл:

```
float a = 1e7;
while(a < 1e10):
        cout « a;
        a +=1;
        cout « fixed ;
```

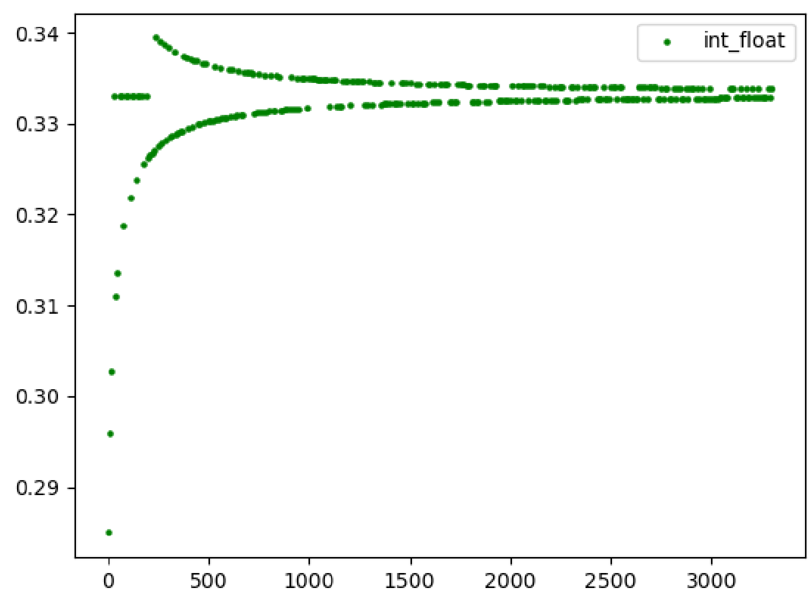
Код остановится на числе 16777215, потому что у этого числа количество знаков в двоичной записи - 24, и все 1, значит дальше, при прибавлении 1 мантисса превратится в строку из 0, дальше такой же и останется.

2.3 Излишняя точность

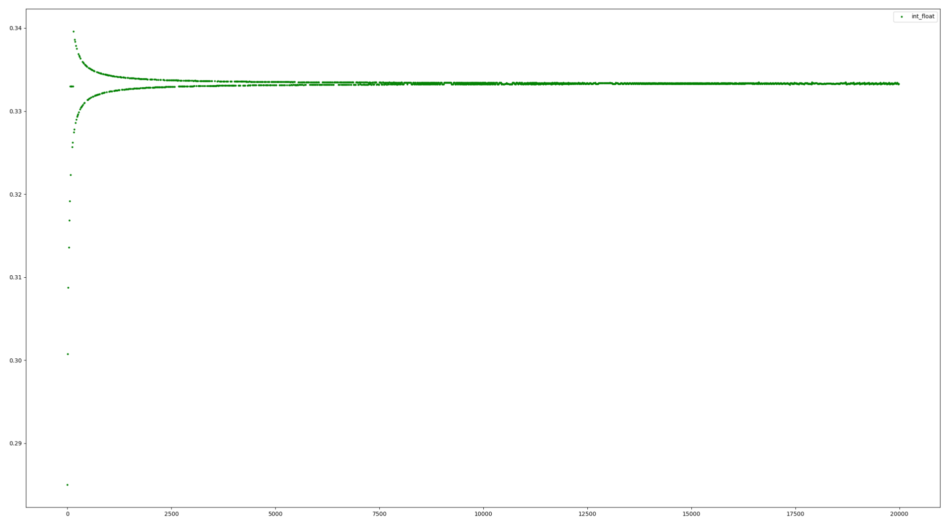
Теперь попробуем, например, проинтегрировать график x^2 от 0 до 1 Сделаем это методом прямоугольников, каждый раз выбирая ширину прямоугольников всё меньше и меньше. По формуле должна получиться 1/3.



Попробуем уменьшать ширину прямоугольника до 1/3000

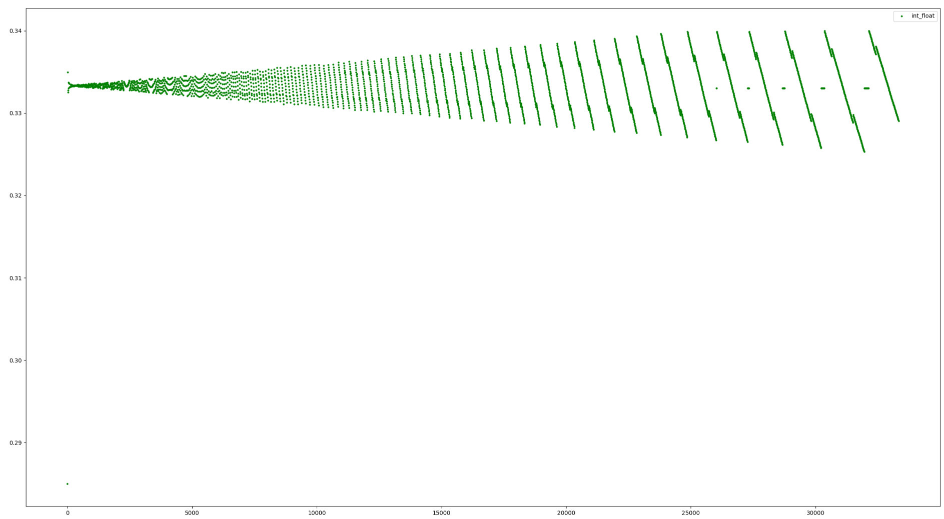


Видим, что площадь под графиком сходится к 3.333, но все-таки слегка в конце расходится. Попробуем еще уменьшить ширину, до 1/3e4



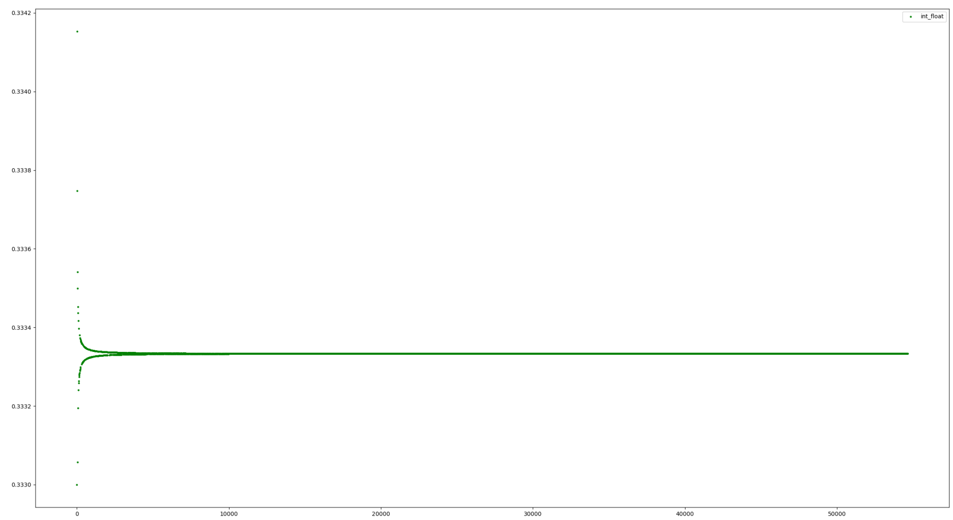
Теперь точность очень хорошая, видно, что все точки сошлись в одном значении - почти 1/3

Однако что будет, если продолжить уменьшать ширину прямоугольников? уменьшим до 1.3e6



Наблюдаем очень серьезное отклонение от 1/3. Что происходит? Числа, которые мы получаем при делении 1 на 3e6 получаются слишком маленькими, и теряется точность. Таким образом обрезаются единички, которые влияют на общее значение выражения.

Если мы попробуем провернуть то же самое, но заменив float на double, то мы не сможем получить переполнение мантиссы(точнее сможем, но для этого потребуются совсем маленькие значения ширин).Даже на 3e6 double отлично справляется:



2.4 Антипереполнение

Подробнее разобрав устройство чисел с плавающей точкой, можно заметить, что минимальное положительное число - это

$$0\ 00000000\ \ 0000...0001 = 2^{-127} * (1 + 2^{-23}) 2^{-127}$$

следующее за ним

$$0\ 00000000\ \ 0000...0010 = 2^{-127} * (1 + 2^{-22}) 2^{-127}$$

Разница между ними:

$$2^{-127} * (1 + 2^{-22}) - 2^{-127} * (1 + 2^{-23}) 2^{-150}$$

Это на 23 порядка меньше, чем самое маленькое число. Т.е. разница между двумя разными float-ами может быть не 0, но их разница в двоичном представлении обратится в ноль. Это чревато ошибками, неточностями в расчетах и другими неприятностями. Поэтому, на современных процессорах встрена функция денормализации очень маленьких чисел:

Когда процессор видит, что все биты в степени 0, то он полагает, что это очень маленькое число, и не учитывает дополнительную единицу в мантиссе.

$$0\ 00000000\ \ 0000...0001 = 2^{-127} * (2^{-23}) 2^{-150}$$

Тогда наименьшее число станет меньше, равное как раз минимальной разнице между двумя float-ами.

Посмотрим, какой предел у денормализованных чисел.

Воспользуемся тем же способом, что и в просматривании чисел с плавающей точкой. Запишем в 32 бита всего 1 бит в конце единичкой, и просмотрим это число как float. Получим в точности 2^{-150}, потому что до равенства с 1 надо умножить на 2 ровно 150 раз. Теперь отключим денормализацию чисел. Включим DAZ(denormalised are zero) и FTZ(flush to zero). Получим, что то же самое число(с одним битом в конце) - это теперь около 2^{-127}