НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Дисциплина «Информатика» (углубленный уровень), 1-й курс, 1-й семестр.

Методические указания

Тематическое занятие 4

Вычисления с плавающей точкой.

Содержание

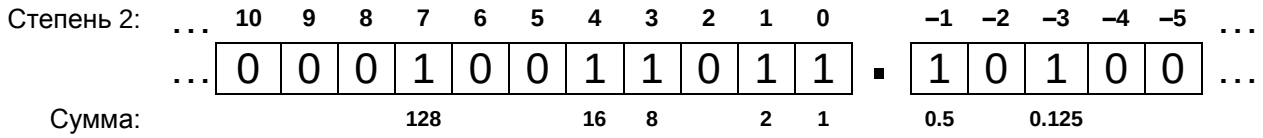
Числа с плавающей точкой

Идея представления вещественного числа

Вещественные числа хранятся в памяти компьютера в специальной форме ***с плавающей точкой (с плавающей запятой), floating point***.

Такое число представлено в виде произведения $m \cdot 2^p$. В одной ячейке памяти хранится два значения – ***мантисса m*** и ***порядок p***, причем оба эти числа – целые и могут иметь знак.

Пример: $155.625_{10} = 10011011.101_2$



Нормализация: $155.625_{10} = \mathbf{1.55625} \times \mathbf{10}^{2}$

$10011011.101_2 = \mathbf{1.0011011101} \times \mathbf{2}^{111} = m \cdot 2^{p}$

m – мантисса

p – порядок (экспонента)

Мантиссу нормализуют так, чтобы она удовлетворяла условию $\mathbf{1 \le m < 2}$, то есть ненулевая мантисса всегда должна начинаться с двоичной единицы.

В этом случае имеет смысл хранить только **дробную часть** мантиссы как целое число, а **целую часть** не хранить и считать **всегда равной единице.**

Влияние системы счисления на точность хранения

В памяти дробная часть мантиссы и порядок хранятся **в двоичной системе счисления,** что влияет на точность представлении вещественного числа уже при его хранении в памяти.

Необходимо учитывать, что дробная часть мантиссы числа с плавающей точкой хранится в двоичной системе счисления. Поэтому числа, которые точно записываются в десятичной системе счисления, в двоичной системе можно представить только в виде бесконечной дроби, которая в памяти компьютера хранится с ограниченной точностью.

Точность хранения:

$0.5_{10} = 1 \cdot 2^{-1}{}_{10} = 0.1_2 = \mathbf{1.0000000000} \times \mathbf{2}^{-1}$

$0.2_{10} = 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} + \ldots_{10} = 0.00110011\ldots_2 =$

$= \mathbf{1.1001100110...} \times \mathbf{2}^{-11}$

Здесь два числа, у которых точность представления в десятичной системе счисления – одна цифра после запятой, в двоичной системе счисления записываются с разной точностью. Число 0.3_{10} хранится в памяти компьютера с погрешностью, а число 0.5_{10} – без погрешности, поскольку является степенью двойки.

Стандарт представления

Формат представления чисел с плавающей точкой определяется международным стандартом **IEEE 754** (**IEC 60559**), который описывает числа:

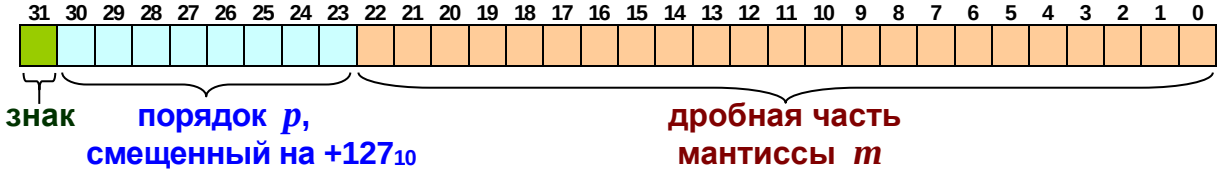
- **одинарной точности (single precision), 32-bit;**
- **двойной точности (double precision), 64-bit;**
- **двойной расширенной точности (double-extended precision), обычно 80-bit.**

Этот стандарт широко используется в программных и аппаратных реализациях арифметических действий с числами с плавающей точкой.

Одинарная точность (single precision), 32-bit, `float`

Структура формата хранения вещественных чисел с плавающей точкой **одинарной точности**, занимающих в памяти ячейку размером 4 байта = 32 бита:

4 байта = 32 бита = 1 (знак) + 8 (порядок) + 23 (мантисса)



Для хранения двоичного кода порядка p используется 8 бит. Чтобы значение p могло быть отрицательным к нему прибавляют смещение – число $127_{10}=01111111_2$. Таким образом можно хранить целые десятичные значения p от –127 до +128.

Пример 1:

$$0.375 = 0.25 + 0.125 = 0\cdot2^{-1} + 1\cdot2^{-2} + 1\cdot2^{-3}$$

с двоичным представлением мантиссы:

$$0.011\cdot2^{0},\quad \text{т.е. } m = 0.011_2;\ p = 0_{10} = 0_2.$$

После нормализации:

$$1.1\cdot2^{-2},\quad \text{т.е. } m = 1.1_2;\ p = -2_{10} = -10_2.$$

Пример 2:

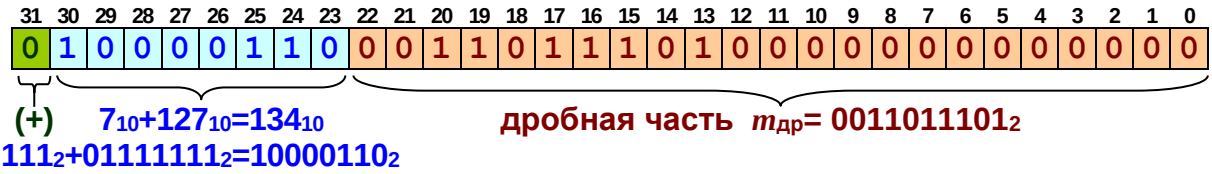
$$155.625 = 128 + 16 + 8 + 2 + 1 + 0.5 + 0.125 =$$
$$= 1\cdot2^{7} + 1\cdot2^{4} + 1\cdot2^{3} + 1\cdot2^{1} + 1\cdot2^{0} + 1\cdot2^{-1} + 1\cdot2^{-3}$$

с двоичным представлением мантиссы:

$$10011011.101\cdot2^{0},\quad \text{т.е. } m = 10011011.101_2;\ p = 0_{10} = 0_2.$$

После нормализации:

$$1.0011011101\cdot2^{7},\quad \text{т.е. } m = 1.0011011101_2;\ p = 7_{10} = 111_2.$$



Двойная точность (double precision), 64-bit, `double`

Структура чисел с плавающей точкой **двойной точности** аналогична, но занимает 8 байт = 64 бита:

8 байт = 64 бита = 1 (знак) + 11 (порядок) + 52 (мантисса)

Число **двойной расширенной точности** (double-extended precision) в соответствии со стандартом занимает $\geqslant$79 бит, но обычно ограничиваются 80 битами.

Исключения

Формат хранения чисел с плавающей точкой приводит к появлению нескольких исключительных ситуаций.

Нуль: ± 0

Нуль закодирован двумя способами: +0 и –0. Однако эти значения при вычислениях не различаются.

Денормализованные числа:

У денормализованных чисел мантисса удовлетворяет условию **0.1≤*m*<1**. Такие числа разбивают минимальный разряд нормализованного числа на некоторое подмножество.

«Бесконечность» (`infinity`): $\pm\infty$

Бесконечным считается число с максимальным порядком (все единицы) и нулевой дробной частью мантиссы (все нули). Различаются $+\infty$ и $-\infty$.

«Не число» (`Not a Number, NaN`): $\pm$`NaN`

Не считаются числами коды с максимальным порядком (все единицы) и ненулевой дробной частью мантиссы (любая ненулевая комбинация 0 и 1).

Операции, приводящие к появлению **NaN** в качестве ответа:

- все математические операции, содержащие **NaN** в качестве одного из операндов;
- деление нуля на нуль;
- деление бесконечности на бесконечность;
- умножение нуля на бесконечность;
- сложение бесконечности с бесконечностью противоположного знака;
- вычисление квадратного корня отрицательного числа;
- логарифмирование отрицательного числа;
- при вычислении `pow(±0,±0)` по стандарту IEEE 754-2008.

Вещественные числа в С

Вещественные типы данных

Вещественные типы хранятся как числа с плавающей точкой (*floating types*) и не являются порядковыми. Стандарт языка С устанавливает:

Название типа	Минимально допустимый диапазон возможных значений	Количество значащих цифр, не менее
`float` (одинарной точности)	1×10^{-37} ... 1×10^{37}	6
`double` (двойной точности)	1×10^{-37} ... 1×10^{37}	10
`long double` (повышенной точности)	1×10^{-37} ... 1×10^{37}	10

Любые два из этих типов могут быть синонимами, т.е. обозначать одно и то же.

Формы записи вещественных чисел

Существует два способа записи вещественных чисел в десятичной системе счисления:

● обычный с десятичной **точкой** (но не запятой):

`127.3` `25.0` `-16.005` `0.54`

● с мантиссой и порядком, которые разделяются символом `e` (или `E`):

`0.9E-3` = $0.9\cdot10^{-3}$ = 0.0009 `0.62e+4` = $0.62\cdot10^{4}$ = 6200

`20e-2` = $20\cdot10^{-2}$ = 0.2 `5.12e+02` = $5.12\cdot10^{2}$ = 512

Точность представления

Точность представления мантиссы может быть продемонстрирована следующей программой:

```
#include <stdio.h>
int main(void) {
   float f;
   double d;
   long double l;
   f = 1.234567890123456789e-30;
   d = 1.234567890123456789e-30;
   l = 1.234567890123456789e-30;
   printf(" f=%.30e\n d=%.30e\n l=%.30Le\n", f, d, l);
   return 0;
}
```

Результат зависит от вычислительной системы, на которой выполнялась программа, и может быть, например, таким:

```
f=1.23456792604715493507680000000e-30
d=1.23456789012345694936810000000e-30
l=1.23456789012345694936810000000e-30
```

Стандартные математические функции

Следующие функции объявлены в заголовочном файле `<math.h>`:

Название	Результат
`fabs(x)`	абсолютное значение
`sqrt(x)`	квадратный корень
`sin(x)`	синус
`cos(x)`	косинус
`tan(x)`	тангенс
`asin(x)`	арксинус
`acos(x)`	арккосинус
`atan(x)`	арктангенс
`exp(x)`	экспонента
`log(x)`	натуральный логарифм
`log10(x)`	десятичный логарифм
`sinh(x)`	гиперболический синус
`cosh(x)`	гиперболический косинус
`tanh(x)`	гиперболический тангенс

Имеются и другие математические функции (см. справочную литературу).

Операции с вещественными числами

Арифметические операции и сравнение

+	*сложение*
-	*вычитание*
*****	*умножение*
/	*деление*

Допустимы операции сравнения. Однако необходимо учитывать, что на вещественном типе данных ***нет отношения порядка.***

Отношение порядка существует, например, у целочисленных типов. Когда, зная некоторое значение целочисленного типа, можно однозначно определить предыдущее и последующее значения данного типа.

Вещественные типы данных не обладают этим свойством, поэтому **операции сравнения `==` и `!=` для чисел с плавающей точкой использовать *не следует*.** Результат их работы может быть неверным. Также теряет смысл использование операций с равенством: `<=` и `>=`

Сравнение чисел с плавающей точкой

Если все-таки необходимо сравнить два числа с плавающей точкой на равенство, то обычно сравнивают их разность с малой величиной ε, задающей точность сравнения. Например:

```
#include <stdio.h>
#include <math.h>
int main(void) {
   double x, y, eps=0.000001;
   ...
   if ( fabs(x-y) < eps )
      printf(" x=y с точностью %lf\n",eps);
   ...
}
```

Ошибки при вычислениях

Числа в компьютере представляются приближенно, что обусловлено конечностью разрядной сетки. Из-за этого может происходить нарушение свойств арифметических операций.

Пример нарушения коммутативности сложения. Если порядок мантиссы не позволяет различить числа порядка 10^{30}, то выполняя суммирования чисел в различной последовательности, получаем различный результат:

```
#include <stdio.h>
int main(void) {
   double x, y;
   x =(1.0 + 1.0e30)+ -1.0e30;
   y = 1.0 +(1.0e30 + -1.0e30);
   printf(" x=%e\n y=%e\n", x, y);
   return 0;
}
```

Результат:

```
x=0.000000e+00
y=1.000000e+00
```

Вычисление суммы ряда

Пусть дан следующий числовой ряд:

$$\frac{1}{3}+\frac{1}{3^2}+\frac{1}{3^3}+\frac{1}{3^4}+\ldots+\frac{1}{3^n}+\ldots$$

и необходимо вычислить сумму ряда напрямую («в лоб»).

Однако не трудно заметить, что данный ряд представляет собой бесконечно убывающую геометрическую прогрессию $b_1, b_2, b_3 \ldots$, у которой:

первый член $b_1 = \frac{1}{3},$ знаменатель $q = \frac{1}{3}.$

По формуле для суммы бесконечно убывающей геометрической прогрессии:

$$S = \frac{b_1}{1-q} = \frac{\frac{1}{3}}{1-\frac{1}{3}} = \frac{1}{2}.$$

Теперь составим программу, которая вычисляет сумму ряда напрямую, каждый раз прибавляя очередное слагаемое b_i. Суммирование будем проводить до тех пор, пока значение членов ряда не станет меньше заданного $\varepsilon = 10^{-6}$:

```
#include <stdio.h>
int main(void) {
   double b, q, S, eps=1e-6;
   b = q = (double)1/3;  /* b - значение первого члена ряда */
   S = 0.0;              /* начальное значение суммы S */
   while (b > eps) {  /* условие остановки суммирования */
      S += b;         /* накапливание суммы */
      b = b*q;        /* вычисление следующего члена ряда */
   }
   printf("S=%.20lf\n", S);
   return 0;
}
```

Результат может быть таким:

```
S=0.49999905916178840926
```

что согласуется с аналитическим расчетом по формуле для суммы прогрессии в пределах определенной погрешности.

В данном примере при суммировании к переменной `S` прибавляется очередной член ряда – переменная `b`, значение которой уменьшается на каждой итерации цикла.

Заметим, что на последних итерациях значения переменных `S` и `b` различаются на 5–6 порядков. Это не очень много, чтобы оказать влияние на точность вычислений. Но бывает, что значения переменных различаются существенно.

Суммирование чисел

При решении задачи нахождения суммы большого множества чисел возникает ошибка округления, связанная со следующим обстоятельством. Переменная, в которой накапливается сумма, после некоторого количества итераций принимает достаточно большое значение. На каждом следующем шаге суммирования к ней прибавляется очередное число, существенно меньшее по значению. При этом младшие разряды меньшего числе теряются, увеличивая ошибку округления.

Чтобы уменьшить влияние данного эффекта, можно использовать ***метод суммирования Кохена*** *(Kahan Summation)*. В этом методе проводится коррекция промежуточной суммы на протяжении всей работы алгоритма.

Метод Кохена

В общем виде идея метода Кохена реализуется следующим образом. Имеется цикл для вычисления суммы, очередное слагаемое присваивается переменной `f` на каждой итерации цикла, сумма накапливается в переменной `S`. Тогда введем значение коррекции – переменную `cor`, а также промежуточные переменные `fcor` – скорректированное слагаемое и `Scor` – скорректированная сумма.

```
S = 0.0; cor = 0.0; Scor = 0.0;
for (i=1; i<=MAX; i++) {
     f = <очередное i-е слагаемое>;
     fcor = f - cor;
     Scor = S + fcor;
     cor = (Scor - S) - fcor;
     S = Scor;
}
```

Здесь `S` – большое число, а `fcor` – маленькое, и при суммировании младшие биты `fcor` теряются.

Следует обратить внимание, что алгебраически данная процедура ни делает ничего особенного. Подставляя выражение

```
Scor = S + fcor
```

в строку

```
cor = (Scor - S) - fcor
```

для точных значений получается

```
cor = ((S + fcor  ) - S) - fcor = 0
```

Но для чисел с плавающей точкой поправочный член `cor` чаще всего ненулевой, и точность вычисления повышается.