Physics 115/242; Peter Young
Representation of numbers on the computer.

I. INTRODUCTION

Because only a finite number of bits are used to store numbers on the computer, there are limitations on the *range* of numbers that can be represented, and also the *precision* of numbers. If these limitations are not understood and accounted for, the results of numerical computations may be either complete garbage, or much less accurate than expected. The error in not being able to represent a number exactly on the computer is called a "*roundoff error*". This is one of the two main types of error in numerical computation (not counting mistakes by the programmer!). It is therefore important to have a basic understanding of how numbers are represented on the computer.

Numbers on the computer are stored in binary representation, i.e. a string of 1's and 0's. Eight bits make one byte, and variables are typically represented by 32-bit or 64-bit words. There are two basic types of numbers understood by compilers, integers and floating point numbers, and we will discuss each of them in turn. A discussion of the representation of numbers is given in Landau, Páez and Bordeianu, Sec. 2.1, and "Numerical Recipes". Free online versions of the older editions of numerical recipes books are available at `http://www.nr.com`.

II. INTEGERS

Integer variables, declared as `int` or `long` in C and C++, can only represent whole numbers but have the advantage that they represent them *exactly*.

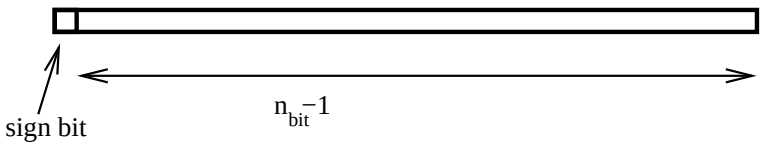
Integers are represented by a certain number of bits, $n_{\rm bit}$, usually equal to 32. The left most bit is taken to be the sign bit; if it is zero the numbers are positive and if it is one the numbers are negative. This is sketched in Fig. 1.

To illustrate how integers are represented I take, for convenience, the unrealistic case of $n_{\rm bit} = 4$, and show in Table I the representation of *all* 16 integers that can be represented with 4 bits. (C and C++ also have unsigned integers that can never be negative, for which the representation would be a bit different.)

Note that 0 and the positive integers (up to 7 here) are represented in the natural binary representation. However, the way the negative integers are represented may look a little strange. Compared with the corresponding positive number, all the bits except the first are the complement (*i.e.* 1 for 0 and vice versa) and the first bit is the same. The reason for this representation is that if one adds an integer i to its negative, $-i$, one ends up with zero (which is obviously desirable!). Consider, for example, $i = 1$. Adding 1 and -1 in the binary representation in Table I gives

```
  0001
+ 1111
 -----
 10000
```

The four least significant digits contain zero, but, in addition, there is an overflow (fifth) bit equal to 1. However, with integer arithmetic, compilers **ignore integer overflows** and discard any extra bits, so the result is zero as desired.



Representation of an integer

FIG. 1:

bits	integer
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	−8
1001	−7
1010	−6
1011	−5
1100	−4
1101	−3
1110	−2
1111	−1

TABLE I: Representation of *all* 4-bit integers.

The range of integers represented in this example is $-8 \leq i \leq 7$. It is not hard to see that with $n_{\rm bit}$ bits the range is

$$-2^{n_{\rm bit}-1} \leq i \leq 2^{n_{\rm bit}-1} - 1\,. \tag{1}$$

For the common case of $n_{\rm bit} = 32$ this gives

$$-2147483648 \leq i \leq 2147483647\,, \tag{2}$$

so the largest positive integer is about 2×10^9.

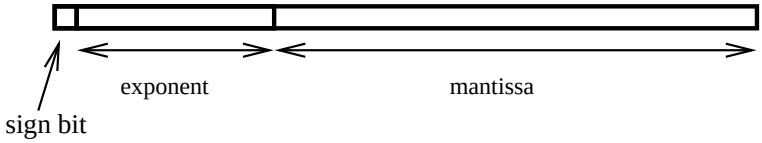
As an example, I wrote a little program that take an integer i, multiplies it by 2 and adds one. It then repeatedly does the same operations on the integer so obtained, i.e. it iterates the process $i_{n+1} = 2 * i_n + 1$ for $n = 0, 1, 2, \cdots$. Starting with $i_0 = 0$ this generates the sequence $1, 3, 7, 15, \cdots$, i.e. $i_n = 2^n - 1$, with binary representation $1, 11, 111, 1111, \cdots$. This should clearly generate a sequence of positive numbers. However, for the $n = 32$ iteration I get -1, which is wrong, but is due to integer "overflow" coming from representing integers by 32 bits. I enclose below the last few lines of output:

```
      i_n    correct value
   67108863    2^26 - 1    (correct)
  134217727    2^27 - 1    (correct)
  268435455    2^28 - 1    (correct)
  536870911    2^29 - 1    (correct)
 1073741823    2^30 - 1    (correct)
 2147483647    2^31 - 1    (correct)
         -1    2^32 - 1    (wrong!)
```

The reason why the last entry, which has all 32 bits equal to 1, gives the result -1 is because this is the value of the integer with all bits, including the sign bit, set equal to 1, see Table I. Note that the largest correct integer is indeed $2^{31} - 1$, as stated above in Eqs. (1) and (2). Note too, that no warning or error message was printed; the execution of the code continued as if nothing had happened despite the integer overflow.

Hence the following warning is in order for integer variables:

> **If you try to represent integers outside the allowed range you will get garbage but the compiler will not give you a warning**.



Representation of an real variable (float)

FIG. 2:

Some compilers, including my fortran 90 compiler, also allow 64-bit integers, for which the largest number is $2^{63} - 1 \simeq 9.2 \times 10^{18}$.

If you *divide* two integers the result will be rounded down to the nearest integer so, for example, if $i = 5, j = 3$, then i/j has value 1.

III. FLOATING POINT NUMBERS

A. Introduction

Floating point variables, declared as `float` or `double` in C and C++, can represent fractions and decimals as well as whole numbers. Although integers (which are not too large) and *some* fractions, such as 1/2, can be represented exactly by a finite number of digits, many fractions, such as 1/3 or 1/7, can not (either with numbers to base 10, everyday notation, or base 2 as used in computers). Also, irrational numbers, like π and $\sqrt{2}$, can not be represented exactly by *any* finite number of digits. Hence, in general, a floating point number is only represented *approximately*. We will be concerned about the *largest* floating point number that can be represented, as we were also for integers, but, in addition, we will need to discuss the *smallest* (in magnitude) number that can be represented which is distinct from zero, and the *precision* of the numbers.

The representation of a floating point number is

$$\text{sign\_bit} \times \text{mantissa} \times 2^{\text{exponent}-\text{bias}}, \tag{3}$$

where "sign_bit" is a single bit which specifies the sign, the mantissa is a floating point number of the form "$1.\cdots$" where "$\cdots$" is a string of bits (the initial 1 is understood and not represented explicitly; it would waste one bit of precision to do so), and the bias is to permit the power of 2 to be negative as well as positive. The representation is illustrated in the Fig. 2.

Many C compilers use 32 bits to represent a `float` and 64 bits to represent a `double`. I shall use the terminology *single precision* to denote 32-bit real numbers, and *double precision* to denote 64-bit real numbers.

B. Single Precision

The usual structure of a 32 bit (single precision) float is that 8 bits are used for the exponent, 23 bits are used for the mantissa, and one for the sign. On my fortran 90 compiler, the bias is 127; I'm not sure if this is completely standard. As an example, consider the number $5 = 2^2 + 2^0$. In floating point binary this is 1.01×2^2. Hence, as a single precision float it is represented by a sign bit which is 0 (since it is positive), an exponent of `bias` $+2 = 127 + 2 = 129 = 2^7 + 2^0$ (which is 10000001 in binary), and a mantissa of $0100\cdots$, since the leading 1 before the decimal point is omitted, *i.e.*

$$0 \quad 10000001 \quad 01000000000000000000000 \tag{4}$$

There may be some small variations in this among different compilers. I state again that the leading 1 in the binary representation of 5 (101) is omitted; it is understood to be there.

The exponent is then in the range[1]

$$-127 \leq \text{exponent} \leq 128\,, \tag{5}$$

since $2^7 = 128$. Hence the largest positive number that can be represented is around $2^{128} \simeq 10^{38}$. Trying to represent a larger integer will lead to an overflow, and no meaningful number will be represented. Execution of the code may or may not stop with an error message.

Here is some output from a code where I started with a value $x = 1$ and kept multiplying by 10:

```
  1.000000E+34
  1.000000E+35
  1.000000E+36
  1.000000E+37
  1.000000E+38
++++++++++++++
```

(previous lines omitted). In the last line, where the computer should have produced 10^{39} the number overflowed and the output is a meaningless string of characters. The way the overflowing number is printed will be different on different compilers.

Similarly, in single precision, the smallest non-zero number that can be represented is about $2^{-127} \simeq 10^{-38}$. Trying to represent a smaller number leads to "floating point underflow" in which the number is represented by exactly zero. On most systems, execution continues with no warning message.

In addition to knowing what are the largest and smallest (but different from zero) floating point numbers that can be represented, we also need to know with what *precision* floating point numbers can be represented? A convenient measure of precision is the difference between 1 and the closest number to 1 that is distinct from 1. With a 23-bit mantissa the precision is $2^{-23} \simeq 1.2 \times 10^{-7}$. The precision is also a typical value of the *relative error* of a number that is not represented exactly by 23 bits of mantissa. The relative error is defined by

$$\text{relative error} = \frac{\text{estimated value} - \text{exact value}}{\text{exact value}}. \tag{6}$$

The precision of a 32-bit float is illustrated by the output below, in which, at each stage of iteration, we set $\epsilon \to \epsilon/10, x = 1 - \epsilon$, and $y = 1 - x$ (starting with $\epsilon = 1$). We see that for $\epsilon < 10^{-7}$, y is represented as zero rather than the correct result of ϵ:

```
   epsilon      1 - (1 - epsilon)
 1.000000E-01   1.000000E-01     (correct)
 1.000000E-02   9.999990E-03     (accurate)
 1.000000E-03   9.999871E-04     (accurate)
 1.000000E-04   1.000166E-04     (accurate)
 1.000000E-05   1.001358E-05     (somewhat accurate)
 1.000000E-06   1.013279E-06     (roughly correct)
 1.000000E-07   1.192093E-07     (very roughly correct)
 1.000000E-08   0.000000E+00     (wrong; should be epsilon not 0)
```

These numbers also illustrate an important point, which we shall elaborate on in the handout on numerical differentiation, that relative errors *increase* when subtracting numbers of nearly equal magnitude.

I emphasize that, in many cases, a precision of 10^{-7} is **not enough**. For example, suppose that we want to average a dataset of 10^7 numbers (on a modern computer this would be very quick; it would probably take less than a second). Towards the end of the list, the numbers would be added very imprecisely because the computer can only very imprecisely represent the difference between the running total before the number was added and after it was added. I therefore **recommend double precision for most numerical work**.

Note the distinction between the largest and smallest numbers, which are related to the number of digits in the *exponent*, and the precision, which is related to the number of digits in the *mantissa*.

---

[1] Actually, on my compiler, if the exponent has the largest possible value, 255 (so `exponent - bias` in Eq. (3) is 128), the value of the real number is "infinity", and if the exponent has the smallest possible value, 0 (so `exponent - bias` $= -127$), the real number is exactly zero.

	single precision	double precision
largest number	$\simeq 10^{38}$	$\simeq 10^{308}$
smallest number	$\simeq 10^{-38}$	$\simeq 10^{-308}$
precision	$\simeq 10^{-7}$	$\simeq 10^{-16}$

TABLE II:

C. Double Precision

For 64-bit (double precision) floating point numbers, there are usually 52 bits in the mantissa, and 11 in the exponent and, on my compiler, the bias is 1023. Hence the exponent is in the range

$$-1023 \leq \text{exponent} \leq 1024, \tag{7}$$

and consequently the largest number is about $2^{1024} \simeq 10^{308}$, and the smallest number is about 10^{-308}. Similarly, the precision is $2^{-52} \simeq 2.2 \times 10^{-16}$.

D. Summary

We summarize the results of this section in Table II

The following warnings are in order for floating point variables:

- Floating point numbers larger than the largest allowed number shown in Table II will cause floating point overflow and the **resulting number will be garbage**.
- Numbers smaller (in magnitude) than the smallest allowed number shown in Table II will cause floating point "underflow" and will be **represented by exactly zero**.
- In general, floating point numbers are only represented **approximately**. Errors in representing floating point numbers lead to **roundoff errors** in computations, as we shall discuss in the course.
- It is generally advisable to use double precision. On most compilers the cost in speed compared with single precision is not very great.

IV. OTHER COMPUTER PROGRAMS

Powerful programs like Mathematica (used in the second half of the course) and Maple, which do symbolic manipulation as well as numerical calculations, can treat numbers of arbitrary size and precision, limited only by the memory and CPU power of the computer. However, this arbitrary precision arithmetic is much slower than standard double precision arithmetic using a C or fortran compiler.