Floating-Point Numbers

Introduction

The world of science and engineering measures quantities inexactly as fractional numbers, which means numbers that have a decimal point:

speed of light in vacuum = 2.9979×10^8 ms^{-1}

Such numbers are called **real numbers.** The overall accuracy is determined by the precision of the basic measurement and the precision of other numbers used in the derivation.

Real numbers are expressed as:

- a mantissa (2.9979)
- exponent (8)
- base (10)

Note there is no unique representation of a real number. The following are all equally valid:

- 2.9979×10^{8}
- 0.29979×10^{9}
- 29979×10^{4}
- 299790000 (to this accuracy)

Standard Form

However, to provide uniformity, a standard representation of real numbers has been established where the decimal point is positioned to the right of the first non-zero digit reading from the left and the exponent adjusted accordingly:

$$2.9979 \times 10^8$$

This is scientific/engineering **standard form**.

Computer Representation of Real Numbers

The number measured in the real world is already inexact:

- *It was measured using instruments that have limits in their accuracy*
- *by engineers that make mistakes*

In the computer it may become even more inexact since **it can only be represented by a finite number of bits that may be insufficient for its precision.**

The **floating-point representation** is how a computer stores a real number. It has features similar to that of scientific standard form. It is called floating-point because the decimal point "floats" to a normalized position.
A floating-point number has:

- a sign bit (±)
- the fractional part M called the **mantissa** or **significand** (23 bits long in a float + hidden bit) that has a precision p.
- the power e, called the **exponent** or **characteristic**
- a **base** B (usually 2, but 10 in calculators and 16 in IBM mainframes)
- a **bias** E - a machine-dependent fixed offset for dealing with signed quantities. The standard value for float is 127. *It is not the same as 2's complement*

Floating-point numbers are mostly held internally in **normalized form** (which is similar in idea but different from scientific standard form), but very small numbers near underflow are held in denormalized form – see later. The general idea is to shift the mantissa either left (if the size of the number is less than 1) or right (if the size of the number is greater than 1) until the leftmost 1 bit arrives in front of the decimal point. For example if the mantissa started off as:

.00010100000000000000000

then in normalized form it is shifted to the left until a 1 is in front of the decimal point and becomes:

1.01000000000000000000000

The Hidden Bit

In normalized form we always end with a 1 before the decimal point. Knowing this we can lose that bit from memory but remember in any calculation that it is really there. It represents the number $2^0 = 1$. Doing so adds an extra bit of precision.
The 1 before the decimal point is the "hidden bit" and is not stored in memory:

[1].01000000000000000000001

Hence in normalized form the number is represented as

$$\pm [d_1]. d_2d_3d_4\ldots\ldots d_p * B^{e-E}$$

For base 2 the value represented is $\pm [1.]M * 2^{e-E}$

The bit field is laid out in memory in a way that depends on the implementation of the language and the processor. The following is how a float number may appear:

+ or - bit	8-bit signed excess-127 representation	23-bit mantissa (plus 1 hidden bit)

Writing normalized form in powers of 2 (rather than a bit field) for the mantissa gives: $\pm ([2^0] . 2^{-1} + 2^{-2} + 2^{-3} + \ldots.2^{-p}) * B^{e-E}$

Bias (excess representation)

This is a way of representing a signed floating-point number in binary (***It is different from 2's complement)***.

Using the bias method (also known as the **excess** method), a stored exponent bit field e represents the actual number (e – E) where E is the bias.

- For a bias of 127, if the number stored in memory for the exponent is 0, then the actual exponent is –127
- For a bias of 127, if the number stored in memory for the exponent is 127, then the actual exponent is 0

  This is **excess 127 representation.**

Converting a Decimal to a Float Bit Field

The bias for the float data type = 127

The top bit of the normalized mantissa is 2^0 =1 **and is hidden**
The bit field (layout) of the number in memory is in the form:

Sign (1 bit)	Exponent (8 bits) biased by 127	Mantissa (23 bits) + 1 extra hidden bit
S 0 = +ve 1 = -ve	eeeeeeee	[1.]mmmmmmmmmmmmmmmmmmmmmmm

Example 1 12.5
This can be expressed as a simple combination of powers of 2.

- Since it's a positive number the sign bit is 0.
- Expressing the number as powers of 2:
  $12.5 = 8 + 4 + 0.5 = 2^3 + 2^2 + 2^{-1} = 1100.1_2$
- Normalize the mantissa by right-shifting in this case so the top bit is hidden [1.]1001 ,
  but since this is equivalent to multiplying by 2^{-3} we have to increase the exponent by 3.
  The exponent = 127 + 3 = 130 = $1000\ 0010_2$
- Summary:
  sign bit = 0, exponent = $1000\ 0010_2$, mantissa (significand) =1001
  The complete number is

sign	exponent	mantissa
0	10000010	[1.]10010000000000000000000

The complete bit field is
01000001010010000000000000000000
Grouping into hex characters:
0100 0001 0100 1000 0000 0000 0000 0000 = **41480000_{16}**

Finally 12.5_{10} = hexadecimal in memory **41480000_{16}**

(In MS VC++ the bytes are ordered in reverse in little-endian format, so grouping into bytes and reversing the order of the bytes finally gives the actual memory layout: 0x00004841)

Example 2 15.75

This can be expressed as a simple combination of powers of 2.

- Since it's a positive number the sign bit is 0.
- Expressing the number as powers of 2:
  $15.75 = 8 + 4 + 2 + 1 + 0.5 + 0.25 = 2^3 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-2} = 1111.11_2$
- Normalize the mantissa by right-shifting in this case so the top bit is hidden [1.]11111 ,
  but since this is equivalent to multiplying by 2^{-3} we have to increase the exponent by 3.
  The exponent = 127 + 3 = 130 = $1000\ 0010_2$
- Summary:
  sign bit = 0, exponent = $1000\ 0010_2$, mantissa (significand) =111111
  The complete number is

sign	exponent	mantissa
0	10000010	[1.]11111000000000000000000

The complete bit field is
01000001011111000000000000000000
Grouping into nybbles:
0100 0001 0111 1100 0000 0000 0000 0000 = $417C0000_{16}$

Finally 15.75_{10} = hexadecimal in memory **$417C0000_{16}$**

Example 3 -1.0

This can be expressed as a simple combination of powers of 2.

- Since it's a negative number the sign bit is 1.
- Expressing the number as powers of 2:
  $1.0 = 1.0_2$
- Normalize the mantissa by left-shifting in this case so the top bit is hidden [1.]0 ,
  The exponent remains at the bias value = 127 = $0111\ 1111_2$
- Summary:
  sign bit = 1, exponent = $0111\ 1111_2$, mantissa (significand) =0
  The complete number is

sign	exponent	mantissa
1	01111111	[1.]00000000000000000000000

The complete bit field is
10111111100000000000000000000000
Grouping into nybbles:
1011 1111 1000 0000 0000 0000 0000 0000 = $BF800000_{16}$

Finally -1.0_{10} = hexadecimal in memory **$BF800000_{16}$**

Example 4 0.625

This can be expressed as a simple combination of powers of 2.

- Since it's a positive number the sign bit is 0.
- Expressing the number as powers of 2:
  $0.625 = 0.5 + 0.125 = 2^{-1} + 2^{-3} = 0.101_2$
- Normalize the mantissa by left-shifting in this case so the top bit is hidden
  [1.]01 ,
  Since this is a multiplication be 2, the exponent must be decreased by 1
  Exponent = 127 - 1 = 126 = $0111\ 1110_2$
- Summary:
  sign bit = 0, exponent = 01111110_2, mantissa (significand) =1
  The complete number is

sign	exponent	mantissa
0	01111110	[1.]01000000000000000000000

The complete bit field is
00111111001000000000000000000000
Grouping into nybbles:
0011 1111 00100 00000000000000000000 = $3F200000_{16}$

Finally 0.625_{10} = hexadecimal in memory **$3F200000_{16}$**

Example 5 1024.0

This can be expressed as a simple combination of powers of 2.

- Since it's a positive number the sign bit is 0.
- Expressing the number as powers of 2:
  $1024 = 2^{10} = 10000000000.0_2$
- Normalize the mantissa by right-shifting in this case so the top bit is hidden [1.]0 ,
  Since this is a division be 2, the exponent must be increased by 10
  Exponent = 127 + 10 = 137 = $1000\ 1001_2$
- Summary:
  sign bit = 0, exponent = $1000\ 1001_2$, mantissa (significand) =0
  The complete number is

sign	exponent	mantissa
0	1000 1001	[1.]00000000000000000000000

The complete bit field is
01000100100000000000000000000000
Grouping into nybbles:
0100 0100 1000 0000 0000 0000 0000 0000 = 44800000_{16}

Finally 1024_{10} = hexadecimal in memory **44800000_{16}**

Special Numbers

meaning	exponent	mantissa	comments
0	All 0	All 0	zero
denormalized number	All 0	not all 0	range from 2^{-126} down to underflow – **see below**
INF	All 1	All 0	Infinity – divide by 0 (also used for 1.#IND meaning indeterminate
NaN	All 1	Not all 0	Not a Number: 0/0, √(-1)

Largest and Smallest Numbers

Overflow, Underflow and Epsilon

The number 0 is represented by all bits set to 0. It is a special number and not part of the normalized scheme. So -0 and +0 have the same value though they have different symbols.

Largest float

largest normalized mantissa bitfield = [1].11111111111111111111111 = ~ 2
largest exponent bitfield = 11111110 = 254

Sign (1 bit)	Exponent (8 bits)	Mantissa (24 bits including hidden)
0	11111110	[1.]11111111111111111111111

Which is written as:

0111 1111 0111 1111 1111 1111 1111 1111 = 0x7f7fffff

The largest number ~ $2 * 2^{254-127} = 2^{128} = 10^{38.53184} = 10^{38} * 10^{.53184}$
$= 3.403 * 10^{38}$

Overflow: attempting to store a number larger than ~10^{38} fails.
The failure is called *overflow*.

Smallest float

The smallest **normalized** float is when the mantissa has only the hidden bit of 1 and an exponent of –126. Its bit field is then (because of the 127 bias in the exponent)

Sign (1 bit)	Exponent (8 bits)	Mantissa (24 bits including hidden)
0	00000001	[1.]00000000000000000000000

Which is written as nybbles:

0000 0000 1000 0000 0000 0000 0000 0000= 0x00800000

Therefore the actual smallest normalized number → 2^{-126} * 1
= $10^{-37.92978054}$ (dividing the exponent by the magic number 3.321928)

= ~ 10^{-38}

But (IEEE Standard 754)**,** if the number is smaller than this so the exponent drops to 0, then the number becomes **denormalized**, the hidden bit is ignored and to be continuous with the normalized numbers (above) the bias is taken as 126. The number is then just what remains in the mantissa. The smallest value of the mantissa, when only the 23rd bit is set, then determines the smallest possible number ~ $2^{-126}*2^{-23} = 2^{-149}$ ~ $2*10^{-45}$. This is what is done on your PC.

***Underflow*: attempting to store a number smaller than ~10^{-45} fails.**
The failure is called underflow.

Machine Epsilon for Float

The smallest change that can be made to any floating point number number is to add (subtract) 1 to the least significant bit of the mantissa.
The value of the 23rd bit of the mantissa = $2^{-23} = 10^{-23/3.22} \sim 10^{-7}$

Example 1 What is the smallest increase than can be made to 1.0 as a consequence of epsilon?
From previous, bit field 1.0 = 00111111100000000000000000000000 = $2^0 * 1.0$
Increased by epsilon
= 00111111100000000000000000000001 = $2^0 * (1.0+10^{-7}) = 1.0 +10^{-7}$

1.0 has increased to 1.0000001. The increase is 0.0000001.
The fractional increase is $0.0000001/1.0 = 10^{-7}$

Example 2 What is the smallest increase than can be made to 1024 as a consequence of epsilon?
From previous, bit field 1024 = 01000100100000000000000000000000
$= 2^{10} * 1.0$
Increased by epsilon = 01000100100000000000000000000001
$= 2^{10} * (1.0+10^{-7})$
1024 has increased to 1024.0001024
The increase is 0.0001024 (= $1024 * 10^{-7}$)

The fractional increase = $.0001024/1024 = 10^{-7}$

Definition
Epsilon is ~ the smallest fractional change that can be made in any floating point number.
The actual value of a number = $2^{exponent} *$ mantissa
The next number up is $2^{exponent} *$ (mantissa + 2^{-23})
So the gap between the numbers =
$2^{exponent} *$ (mantissa + 2^{-23}) -$2^{exponent} *$ mantissa = $2^{exponent} * 2^{-23}$
$\sim 2^{exponent} * 10^{-7}$
The gap is large for large numbers and small for small numbers, but it's always fractionally ~ 2^{-23}.

Roundoff Error
Calculations in the computer are done in double precision by default. If the result is stored as a float, it will be reduced to the lower precision of the float – this is called **roundoff**. The loss in precision is called the **roundoff error**. Any floating point calculation done in a computer has the roundoff error set by the precision of the data type ~ 10^{-7} for float, ~ 10^{-16} for double.

Code doesn't always work like you hope

```
void main(void){
        float q;
       q = 3.0/7.0;
        if(q==3.0/7.0)
            cout<<"q equals 3.0/7.0"<<endl;
        else
            cout<<"q does not equal 3.0/7.0"<<endl;
}
```

Output after compiling in MS Visual C++: `q does not equal 3.0/7.0`
Can you explain this output?

Observations

- You can store very small numbers on a computer ~ as small as ~10^{-45} for a float

  float x = 1.0e-44 // OK
- But you cannot add a small number to a large number if the fractional increase in the large number is ~less than epsilon

```
float s = 2.0e-8;             // below epsilon
printf("s = %.8e\n", s);      // s = 1.99999999e-008
float t = 5.0e-7;             // above epsilon
float v = 1.0e-4;
float a = 1.0;
float b = 100.0;

printf("a + s = %.8e\n", a + s);    //    a + s = 1.00000000e+000
printf("a + t = %.8e\n", a + t);    //    a + t = 1.00000048e+000
printf("b + t = %.8e\n", b + t);    //    b + t = 1.00000000e+002
printf("b + v = %.8e\n", b + v);    //    b + v = 1.00000099e+002

printf("regulated oxygen flow (mm/hour) = %lf\n", 1/(1 - a + s));
// regulated oxygen flow(mm/hour) = 50000000.000000

printf("tragic accident(mm/hour) = %lf\n", 1/(1 - (a - s)));
// tragic accident(mm/hour) = 1.#INF00
```

All of which leads to the following picture:

Floating Limits (normalized numbers):
Microsoft Specific (Information in float.h in MS Visual C++)

Constant	Meaning	Value
FLT_DIG DBL_DIG LDBL_DIG	Number of digits, q, such that a floating-point number with q decimal digits can be rounded into a floating-point representation and back without loss of precision.	6 15 15
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	Smallest positive number x, such that x + 1.0 is not equal to 1.0.	1.192092896e–07F 2.2204460492503131e–016 2.2204460492503131e–016
FLT_GUARD		0
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	Number of digits in the radix specified by FLT_RADIX in the floating-point significand. The radix is 2; hence these values specify bits.	24 53 53
FLT_MAX DBL_MAX LDBL_MAX	Maximum representable floating-point number.	3.402823466e+38F 1.7976931348623158e+308 1.7976931348623158e+308
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	Maximum integer such that 10 raised to that number is a representable floating-point number.	38 308 308
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	Maximum integer such that FLT_RADIX raised to that number is a representable floating- point number.	128 1024 1024
FLT_MIN DBL_MIN LDBL_MIN	Minimum positive value.	1.175494351e–38F 2.2250738585072014e–308 2.2250738585072014e–308
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	Minimum negative integer such that 10 raised to that number is a representable floating- point number.	–37 –307 –307
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	Minimum negative integer such that FLT_RADIX raised to that number is a representable floating-point number.	–125 –1021 –1021
FLT_NORMALIZE		0
FLT_RADIX _DBL_RADIX _LDBL_RADIX	Radix of exponent representation.	2 2 2
FLT_ROUNDS _DBL_ROUNDS _LDBL_ROUNDS	Rounding mode for floating-point addition.	1 (near) 1 (near) 1 (near)