

## Topic 2 Fundamentals of Numeric Analysis

### References

- [1] R Landau, M Paez, *Computational Physics*, 1997, Wiley
- [2] W Press, B Flannery, S Teukolsky, W Vetterling, *Numerical Recipes in C/Pascal/Fortran*, Cambridge University Press, 1996
- [3] *Introduction to Scientific Programming*, Springer-verlag, 1996
- [4] *Computer Organization and Architecture*, William Stallings, Macmillan Publishing Company,

School of Computer Science & IT, RMIT

### Negative Numbers

The largest value that can be stored in a *short* is  
 $1111\ 1111\ 1111\ 1111_2 = 2^{16} - 1 = 65,535_{10}$   
 and the least is  $0000\ 0000\ 0000\ 0000_2$ .  
 For 32-bit *int*,  $2^{32} - 1 = 4,294,976,295$

Negative numbers are represented by separating all the possible numbers into those whose left-most bit is 0 and those where it is 1.

This first bit is then denoted the sign bit. Negative Numbers are represented using a *two's complement* notation.

Given x, you can find -x by inverting every bit ( $0 \leftrightarrow 1$ ) and add 1.

Thus  
 $-52 = \text{invert}(52) - 1$   
 $= \text{invert}(0000\ 0000\ 0011\ 0100_2) - 1$   
 $= 1111\ 1111\ 1100\ 1011_2 + 1$   
 $= 1111\ 1111\ 1100\ 1100_2$

Now if we add -52 to 52, we get  
 $1111\ 1111\ 1100\ 1100_2$   
 $+ 0000\ 0000\ 0011\ 0100_2$   
 $(1) 0000\ 0000\ 0000\ 0000_2$

### Binary Representation of Integers

Computers store numbers in memory in a binary form. The smallest unit of memory is the binary digit or *bit*. Other units are built up from this.

- Bit = "binary digit" = 0 or 1
- Byte = 8 bits
- Short = 16 bits (as used in Java)
- Int = 32 bits
- Long = 64 bits

Numbers are represented in a computer as a power series of bits

$$N = \sum_{i=0}^n a_i x^i$$

Consider the arabic number system you are familiar with.  
 $216 = 2 \times 10^2 + 1 \times 10^1 + 6 \times 10^0$

Similarly  
 $0000\ 0000\ 0011\ 0100_2$   
 $= 0 \times 2^{15} + \dots + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2$   
 $= 52$

### Hexadecimal Notation

If we group the bits together in lots of four, we can convert them to a single hexadecimal digit.

So  
 $0000\ 0000\ 0011\ 0100_2$   
 $= 0 \times 16^3 + 0 \times 16^2 + 3 \times 16^1 + 4 \times 16^0$   
 $= 0 + 0 + 48 + 4$   
 $= 52$

Whereas binary digits have 2 values: [0,1], hexadecimal digits have 16 values: [0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F]

Number	Binary	Hexadecimal
5	0000 0101 <sub>2</sub>	05 <sub>16</sub>
83	0101 0011 <sub>2</sub>	53 <sub>16</sub>
11	0000 1101 <sub>2</sub>	0B <sub>16</sub>
22	0001 1010 <sub>2</sub>	1A <sub>16</sub>
44	0011 0100 <sub>2</sub>	34 <sub>16</sub>
88	0110 1000 <sub>2</sub>	68 <sub>16</sub>
176	1101 0000 <sub>2</sub>	B0 <sub>16</sub>

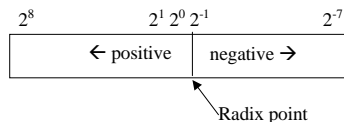
## Binary Representations of Real Numbers

### Fixed Point Representation

Binary notation can be extended to negative powers of 2:

$$\begin{aligned} 110.101 &= 1x2^2 + 1x2^1 + 0x2^0 \\ &\quad + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} \\ &= 6.625 \end{aligned}$$

Thus, in a Fixed-Point Representation of a real number we can choose a "radix point" and give bits to the left, positive powers of 2 and bits to the right, negative powers of two.



Thus instead of a range of 0...65535 for this *unsigned short* variable, we now have a range of 0...1023.984375

Thus fixed-point representation trades off range against precision.

### The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127.

Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73.

Exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.

For double precision, the exponent field is 11 bits, and has a bias of 1023.

### The Mantissa

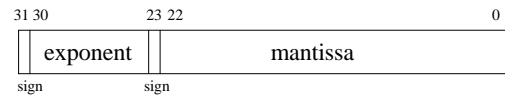
The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

## Floating-Point Representation

A Floating-Point Representation of a real number allows for larger range without sacrificing precision.

It does this by storing the exponent of 2 itself, rather than implicitly by position.



$$\text{Number} = \text{sign} \times \text{mantissa} \times 2^{\text{exponent}}$$

Different fields of the binary word can be reserved for the sign, mantissa, and exponent (which is an integer with a sign).

IEEE Standard 754:

For 32-bit *int*:

- 1-bit sign, 8-bit exponent, 23-bit mantissa
- Largest is  $2^{126} \approx 10^{38}$ , smallest is  $2^{-150} \approx 10^{-47}$ .

For 64-bit *longs*:

- 1-bit sign, 11-bit exponent, 52-bit mantissa
- Largest is  $2^{1022} \approx 10^{308}$ , smallest is  $2^{-1074} \approx 10^{-323.3}$

$$\begin{aligned} &5.00 \times 10^0, \\ &0.05 \times 10^2, \\ &5000 \times 10^{-3} \end{aligned}$$

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in **normalised** form. This puts the radix point after the first non-zero digit. In normalised form, five is represented as  $5.0 \times 10^0$ .

Since the only possible non-zero digit is 1, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

The following program will display the bit patterns of float provided on the command line:

```

class floattest {
    public static void main(String args[] ) {
        float n = Float.parseFloat(args[0]);
        int ni = Float.floatToRawIntBits(n);
        System.out.println(
            "Float bits of " + ni + " are: "
            + Integer.toHexString(ni));
    }
}

```

Program output:

Float bits of 7.0 are: 40e00000  
(= 0 sign, 1000 0001 exp, 1100 0000 0000 000 mant)

Float bits of 14.0 are: 41600000  
(= 0 sign, 1000 0010 exp, 1100 0000 0000 000 mant)

Float bits of 28.0 are: 41e00000  
(= 0 sign, 1000 0011 exp, 1100 0000 0000 000 mant)

Float bits of 0.7 are: 3f333333

Float bits of 70.0 are: 428c0000

Float bits of 700.0 are: 442f0000

## Sources of Error

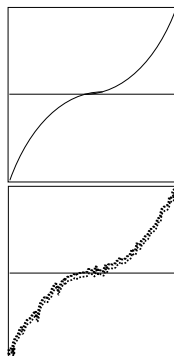
There are a number of principal sources of numerical errors that you should be aware of:

**Analytical Error:** An error that is built into the maths (for example, a truncated infinite series:  $1+x^{-1}+x^{-2}$ )

**Programmer Mistakes and Coding Errors:** To be able to test for these errors, it is always wise to have some test data whose answers you have worked out by hand or by some other means.

**Observational and Roundoff Error:**  
Many experiments obtain values with an error tolerance  
(eg height of desk =  $72.0 \pm 0.3\text{cm}$ )

**Random Errors (noise):**  
Compare the 2 functions to right. The one above is smooth, and so the zero crossing is easy to find. Not so the other.



### Machine Errors:

- These will be covered further
- Overflow and Underflow
- Loss of Significant Errors:

## Floating Point Special Conditions

**Infinity:** The exponent is all 1's and the mantissa is all 0's. The sign bit determines the sign of the infinity.

**Denormalisation** occurs when an operation is performed on numbers of very different magnitude.

**NaN** occurs whenever an operation is invalid or has an invalid result.

**Roundoff Error:** There are approximately  $10^{38}$  integers in the range between the largest positive and negative numbers that can be represented using the 32-bit IEEE standard, but there are only  $2^{32} = 10^9$  different bit patterns. Thus, many numbers do not have an exact representation, leading to roundoff error..

**Underflow:** There is a finite gap between 0 and the smallest positive number. The roundoff of such numbers to 0 produces an underflow.

## Overflow and underflow

Observe the output of following java program. You will see the effects of overflow and underflow.

```

import java.util.*;

class overunder
{
    public static void main(String args[])
    {
        double under = 1.0, over = 1.0;

        int k = Integer.parseInt(args[0],10);
        int n = Integer.parseInt(args[1],10);

        for (int i = 0; i < n; i++) {
            under /= 2;
            over *= 2;
            if (i > k)
                System.out.println("N = " + i +
                    ", over = " + over + " under = " + under);
        }
    }
}

```

its initial output is:

```

> java overunder 0 5
N = 0, over = 2.0 under = 0.5
N = 1, over = 4.0 under = 0.25
N = 2, over = 8.0 under = 0.125
N = 3, over = 16.0 under = 0.0625
N = 4, over = 32.0 under = 0.03125

```

If we continue this program we will get the following behaviour:

```
>java overunder 1020 1025
N = 1020, over = 2.247116418577895E307 under = 4.450147717014403E-308
N = 1021, over = 4.49423283715579E307 under = 2.2250738585072014E-308
N = 1022, over = 8.98846567431158E307 under = 1.1125369292536007E-308
N = 1023, over = Infinity under = 5.562684646268003E-309
N = 1024, over = Infinity under = 2.781342323134002E-309
```

And

```
>java overunder 1065 1078
N = 1065, over = Infinity under = 1.265E-321
N = 1066, over = Infinity under = 6.32E-322
N = 1067, over = Infinity under = 3.16E-322
N = 1068, over = Infinity under = 1.58E-322
N = 1069, over = Infinity under = 7.9E-323
N = 1070, over = Infinity under = 4.0E-323
N = 1071, over = Infinity under = 2.0E-323
N = 1072, over = Infinity under = 1.0E-323
N = 1073, over = Infinity under = 4.9E-324
N = 1074, over = Infinity under = 0.0
N = 1075, over = Infinity under = 0.0
N = 1076, over = Infinity under = 0.0
N = 1077, over = Infinity under = 0.0
```

Note the reduction in precision prior to underflow and the sudden onset of the overflow state. The published range for *doubles* is  $1.7e-308$  to  $1.7e+308$ . Why the difference?

## Loss of Significance

Because floating point numbers have a fixed-size mantissa, we have fixed precision in the numbers we use.

This can cause problems when we mix numbers of very different magnitude. When two floating-point numbers are added, they are first adjusted so that their exponents match. The mantissas can then be added directly.

```
class epstest
{
    public static void main(String args[])
    {
        float eps = 1.0f;
        int k = Integer.parseInt(args[0],10);
        int n = Integer.parseInt(args[1],10);

        for (int i = 0; i < n; i++) {
            eps /= 2;
            float one = 1.0f + eps;
            if (i > k)
                System.out.println("N = " + i +
                    ", one = " + one + " eps = " + eps);
        }
    }
}
```

Program output:

```
N = 2, one = 1.125 eps = 0.125
N = 3, one = 1.0625 eps = 0.0625
...
N = 21, one = 1.0000002 eps = 2.3841858E-7
N = 22, one = 1.0000001 eps = 1.1920929E-7
N = 23, one = 1.0 eps = 5.9604645E-8
N = 24, one = 1.0 eps = 2.9802322E-8
```

## LOS: Subtractive Cancellation

An equation

$$a = b - c$$

might be represented in the computer as

$$a_c = b_c - c_c \\ = b(1 + \epsilon_b) - c(1 - \epsilon_c)$$

$$a_c/a = 1 + \epsilon_b(b_c/a) - \epsilon_c(c_c/a)$$

We see that the average error in  $a$  is a weighted average of the errors in  $b$  and  $c$ . The error in  $a$  increases though, when  $b \approx c$ , because we subtract off, and thereby lose the most significant parts of both numbers.

### This is a general rule

**If you subtract two large numbers and end up with a small one, there is less significance in the small one.**

For example, given that  $ax^2 + bx + c = 0$ , has solution

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

It can be seen that subtractive cancellation arises when  $4ac \ll b^2$ , the  $b$  and the square root will nearly cancel., whereas if  $4ac \approx b^2$ , the subtractive cancellation happens inside the square root.