Syllabus Content:
13.1 Data representation

13.3 Real numbers and normalised floating-point representation
- describe the format of binary floating-point real numbers
- convert binary floating-point real numbers into denary and vice versa
- normalise floating-point numbers
- Show understanding of the consequences of a binary representation only being an approximation to the real number it represents (in certain cases)
- show understanding that binary representations can give rise to rounding errors
  Notes and guidance
- Use two's complement form
- effects of changing the allocation of bits to mantissa and exponent in a floating-point representation
- Understand the reasons for normalisation
- how underflow and overflow can occur

Real numbers:

A real number is one with a fractional part. When we write down a value for a real number in the denary system we have a choice. We can use a simple representation or we can use an exponential notation (sometimes referred to as **scientific notation**). In this latter case we have options.

For example, the number **25.3** might alternatively be written as:

$$0.253 \times 10^2 \text{ or } 2.53 \times 10^1 \text{ or } 25.3 \times 10° \text{ or } 253 \times 10^{-1}$$

For this number, the simple expression is best but if a number is very large or very small the exponential notation is the only sensible choice.

Fixed-point representations:

A binary code must be used for storing a real number in a computer system. One possibility is to use a fixed-point representation. In fixed-point representation, an overall number of bits are chosen with a defined number of bits for the whole number part and the remainder for the fractional part.

One possibility for handling numbers with fractional parts is to add bits after the decimal point: The first bit after the decimal point is the halves place, the next bit the quarter's place, the next bit the eighth's place, and so on.

$$\frac{4}{4} \quad \frac{2}{2} \quad \frac{1}{1} \cdot \frac{0.5}{\frac{1}{2}} \quad \frac{0.25}{\frac{1}{4}} \quad \frac{0.125}{\frac{1}{8}}$$

Place values

Place Values

$$2^2 \quad 2^1 \quad 2^0 \cdot 2^{-1} \quad 2^{-2} \quad 2^{-3}$$

Suppose that we want to represent $1.625_{(10)}$. We would want **1** in the ones place, leaving us with **0.625**. Then we want **1** in the halves place, leaving us with **0.625 − 0.5 = 0.125**. No quarters will fit, so put a **0** there. We want a **1** in the eighths place, and we subtract **0.125** from **0.125** to get **0.**

$$\frac{0}{4} \quad \frac{0}{2} \quad \frac{1}{1} \cdot \frac{1}{\frac{1}{2}} \quad \frac{0}{\frac{1}{4}} \quad \frac{1}{\frac{1}{8}}$$

So the binary representation of **1.625** would be $1.101_{(2)}$.

So how fixed number representation stores a fractional number in binary format? See explanation below

Method 2:

A number **40.125** is to be converted into binary. First number **40** has to be converted as a normal denary to binary conversion, which gives:

$$40 = 101000$$

40 written in complete byte   **40 = 00101000**

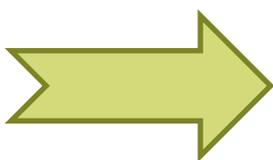Now **0.125** has to be converted to binary. We Multiply **0.125** by **2** e.g.

0.125 x 2 = 0.25 this is less than **1** so we put **0**

0.125x2 = 0.25	**0**
0.25 x 2 = 0.5	**0**
0.5 x 2 = 1	**1**

So fractional number **40.125** number becomes **00101000.001**

↓

Sign bit +ve

For the fixed-point option, a possible choice would be to use **the most significant bit as a sign bit** and **the next five bits for the whole number part**. This would leave **two bits for the fractional part**.
Some important non-zero values in this representation are shown in Table

(The bits shown with a gap indicate the implied position of the binary decimal point.)

Description	Binary code	Denary equivalent
Largest positive value	011111 11	31.75
Smallest positive value	000000 01	0.25

Place values   0.5   0.25

32	16	8	4	2	1	.	½	¼	
0	1	1	1	1	1	.	1	1	= 31.75

Place values   0.5   0.25

32	16	8	4	2	1	.	½	¼	
0	0	0	0	0	0	.	0	1	= 0.25

Negative Fixed Point Representation

Suppose **-31.75** has to converted into binary: **-31.75 = 100000 01**

Place values   0.5   0.25

-32	16	8	4	2	1	.	½	¼	
1	0	0	0	0	0	.	0	1	= -31.75

Suppose **-0.25** has to be converted into binary: **-0.25 = 111111 11**

Place values   0.5   0.25

-32	16	8	4	2	1	.	½	¼	
1	1	1	1	1	1	.	1	1	= -0.25

Consider a fractional number stored in 12 bits, 8 bits for whole number part and 4 bits for fraction -**52.625** becomes **11001100.1010**

↓

Sign bit -ve

In **Fixed-Point Representation** option, an overall number of bits are chosen with a defined number of bits for the whole number part and the remainder for the fractional part.

Fixed Point representation would be   **11001100   1010**

Some important non-zero values in this representation are shown in Table below.
(The bits are shown with a gap to indicate the implied position of the binary point.)

Description	Binary code	Denary equivalent
Largest positive value	011111 11	31.75
Smallest positive value	000000 01	0.25
Smallest magnitude negative value	100000 01	−0.25
Largest magnitude negative value	111111 11	−31.75

Floating-Point Number Representation

The alternative is a **floating-point representation.** The format for a floating-point number can be generalised as: $\pm M \times R^{E}$

In this option a defined number of bits are used for what is called the **significand or mantissa, ±M**. The remaining bits are used for the **exponent** or **exrad**, **E**. The **radix, R** is not stored in the representation; it has an implied value of **2**.

$$\pm M \times R^{E}$$

KEY TERMS

Floating-point representation: a representation of real numbers that stores a value for the mantissa and a value for the exponent

Conversion from +ve Real Number to Binary Number:

A number **40.125** is to be converted into binary. First number **40** has to be converted as a normal denary to binary conversion, which gives:

40 = 101000

40 written in inclusive of sign bit   **40 = 0101000**
                                                           ↓
                                                    Sign bit

Now **0.125** has to be converted to binary. We Multiply **0.125** by **2** e.g.

0.125 x 2 = 0.25 this is less than **1** so we put **0**

                        0.125x2 = 0.25   0
                        0.25 x 2 = 0.5   0
                        0.5 x 2 = 1   1

So fractional number **40.125** number becomes **0101000.001**
                                                              ↓
                                                   Sign bit +ve

But in binary numbers, decimals cannot be written. Decimal has to be converted into binary number too.

            0   1   0   1   0   0   0   .   0   0   1

Decimal moved 6 places to left so **exponent = 6**

Six in binary is represented as **6 = 110**

So the number becomes   **0 101000 001 110**   so number is **0101000001110**
                               ↓   ↓   ↓   **(normalized form)**
                          Sign bit Mantissa Exponent

Past Paper questions:

Question 1: (9608/32/O/N/16)
In a particular computer system, real numbers are stored using floating-point representation with:
- 8 bits for the mantissa
- 8 bits for the exponent
- two's complement form for both mantissa and exponent

Calculate the floating point representation of **+ 3.5** in this system. Show your working.

Solution:

3.5 has to be converted into binary. First number **3** has to be converted as a normal denary to binary conversion, which gives: **3 = 11**

3 written in inclusive of sign bit   **3 = 011**
Now **0.5** has to be converted to binary. **0.5 x 2 = 1**

3.5 = 11.1
So the number becomes **011.1**
↓
Sign bit

011.10000

0   1   1 . 1   0   0   0   0

(decimal moved 2 places right after sign bit)

So the exponent is **2**   so Mantissa and exponent would be

01110000   00000010
↓   ↓   ↓
Sign bit   Mantissa   Exponent

Conversion from -ve Real Number to Binary Number:
Question 1: (9608/32/O/N/16)

In a particular computer system, real numbers are stored using floating-point representation with:
- 8 bits for the mantissa
- 8 bits for the exponent
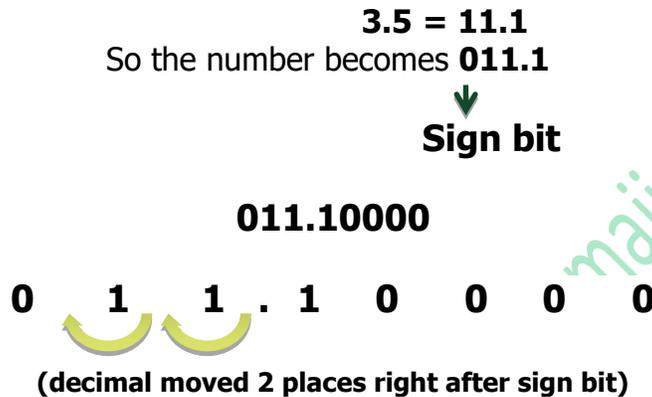- two's complement form for both mantissa and exponent

Calculate the floating – point representation of **-3.5** in this system. Show your working.

Solution: **3** written in inclusive of sign bit   **3 = 11**
Now **0.5** has to be converted to binary. **0.5 x 2 = 1**

3.5 = 11.1

Now   **011.10000 = 3.5** written in whole byte

0   1   1 . 1   0   0   0   0

(decimal moved 2 places right after sign bit)

So exponent = 2 which will be binary in 1 byte = **00000010** (exponent)

01110000 = +3.5 (has to be converted into -3.5) See the process below:

01110000 = 3.5
10001111 = 1's Compliment

1111 ⟶ **carry bits**
10001111
  +1
10010000 = **2's** Compliment = **-3.5**

Mantissa expressed in 8 bits and **exponent expressed in 8 bits** would be

10010000   00000010

Sign bit   Mantissa   Exponent

(Another example)
Conversion from -ve Real Number to Binary Number:
Suppose **-52.625** has to be converted into binary: **52 = 110100**

52 written in complete byte **52 = 00110100**

When we multiply 0.625 by 2 e.g. 0.625 x 2 = 1.25 so we put 1
   0.625x2 = 1.25   1
   0.25 x 2 = 0.5   0
   0.5 x 2 = 1   1

      52.625 = 00110100.101

One's Compliment   = 11001011.010 (by inverting **0's** to **1** and **1's** to **0**)

Two's Compliment -52.625 = 11001011.011 (by adding 1 in One's compliment)

So fractional number **-52.125** becomes **11001011.011**

Sign bit -ve

But in binary numbers, decimals cannot be written. Decimal has to be converted into binary number too.

1   1   0   0   1   0   1   1   .   0   1   1

Decimal moved 7 places to left so **exponent = 7**
Seven in binary is represented as **7 = 111**
So the number becomes   **1 1001011011 111**   so number is **11001011011 111**

Sign bit   Mantissa   Exponent

Conversion from Binary Number to +ve Real Number:

Example 1: 9608/31/O/N/15

Q#1)   In a computer system, real numbers are stored using floating-point representation with:
  - 8 bits for the mantissa, followed by 8 bits for the exponent
  - Two's complement form is used for both mantissa and exponent.

A real number is stored as the following two bytes:

Mantissa	Exponent
0 0 1 0 1 0 0 0	**0 0 0 0 0 0 1 1**

Calculate the denary value of this number. Show your working

Solution: sign bit

As Exponent **0 0 0 0 0 0 1 1 = 3 denary**

                                  Now Mantissa = **0   0 1 0 1 0 0 0**

Decimal is actually after the sign bit
                So decimal would be **0.0101000**
As exponent is 3 so decimal will move three places to right **0 . 0 1 0 1 0 0 0**

Place Values

					0.5	0.25	0.125	0.0625
8	4	2	1	.	½	¼	1/8	1/16
0	0	1	0	.	1	0	0	0

                                2 . 5 (Calculated as per place values)

Example 2:
Q#2) Floating Point Binary representation uses 4 bits for Mantissa and 4 bits for exponent
        Convert **0110 0010**
Solution:
                As Exponent **= 0 0 1 0 = 2 denary**   sign bit

                                        Now Mantissa = **0   1 1 0**

Decimal is actually after the sign bit
                So decimal would be **0.110**

As exponent is 2 so decimal will move two places to right   **0 . 1   1   0**

011 . 0
+ 3 . 0 (Calculated as per place values)

Conversion from Binary Number to -ve Real Number:

Example 1:
Floating Point Binary representation uses 4 bits for Mantissa and 4 bits for exponent

Convert **1001 0001**
Solution: sign bit
As Exponent **= 0 0 0 1 = 1 denary**   ↓
Now Mantissa = **1 0 0 1**
Decimal is actually after the sign bit
So decimal would be **1.001**
As exponent is 1 so decimal will move one place to right **1 . 0   0 1**

. 2   1   ½   ¼ (place Values)
1   0 . 0   1
sign bit ↓   ↓
-2   + ¼   (Calculated as per place values)
= -2   + 0.25

(-2 is **–ve** and +0.25 is **+ve**, so by adding 0.25 in -2, we get) **= - 1 . 75 (Answer)**

Binary Number to -ve Real Number with –ve Exponent:
Find the denary value for the following binary floating-point number.
(Past Paper question)

Mantissa

1	0	1	1	0	0	0	0

Exponent

1	1	1	0

Now we first solve the Exponent

Place Value   =   **-8   4   2   1**
1   1   1   0
Exponent = **-2**

Now solving Mentissa which is:
1   0   1   1   0   0   0   0

We know decimal lies after sign bit so number becomes

1 .   0   1   1   0   0   0   0

Whenever we have –ve exponent with -ve mantissa, we first solve exponent and then TAKE Two's Compliment of Mentissa (Keeping the sign of number intact)

1	.	0	1	1	0	0	0	0

Now take 1's compliment

Carry ⟶   1   1   1   1

0	.	1	0	0	1	1	1	1

Now 2's Compliment Add 1   + 1

0	.	1	0	1	0	0	0	0

Now because our exponent was -2 we have to shift decimal two places LEFT

0	0	0	.	1	0	1	0	0	0

Place values   1   0.5   0.25   0.125   0.0625   0.03125

Number becomes   - (0 . 0   0   1   0   1   0   0   0)

Keeping sign in mind, our number was –ve so we kkep the sign intact

Denary number is - (0.125+0.03125)   = - 0.15325

Precision and normalization:

In floating-point representation, decision has to be made for the total number of bits to be used and **split between** those representing the **mantissa** and the **exponent**.

In practice, a choice for the total number of bits to be used will be available as an option when the program is written.

However, the split between the two parts of the representation will have been determined by the floating-point processor.

Effects of changing the allocation of bits to mantissa and exponent in a floating-point representation:

If you have a choice you would base a decision on the fact that **increasing the number of bits for the mantissa would give better precision** for a value stored **but would leave fewer bits for the exponent so reducing the range of possible values**.

In order to achieve **maximum precision**, it is necessary to normalise a floating-point number.

Since precision increases with an increasing number of bits for the mantissa it follows that optimum precision will only be achieved if full use is made of these bits.
In practice, that means using the largest possible magnitude for the value represented by the mantissa. To illustrate this we can consider the eight-bit representation used in Tables below.

Table below shows possible representations for **denary 2** using this representation.

Denary representation	Floating-point binary representation
0.125×2^4	0 001 0100
0.25×2^3	0 010 0011
0.5×2^2	0 100 0010 ⟵ **Normalized**

Representation of denary number 2, using four bits for mantissa and four bits exponent.

For a negative number we can consider representations for -4 as shown in Table Below

Denary representation	Floating-point binary representation
-0.25×2^4	1 110 0100
-0.5×2^3	1 100 0011
-1.0×2^2	1 000 0010 ⟵ **Normalized**

Representation of denary number -4, using four bits for mantissa and four bits exponent.

It can be seen that when the number is represented with the highest magnitude for the mantissa, the two most significant bits are different. This fact can be used to recognise that a number is in a normalised representation.

The values in these tables also show how a number could be normalised.

Normalizing the Mantissa

Before a floating-point binary number can be stored correctly; its mantissa must be normalized. The process is basically the same as when normalizing a floating-point decimal number.

For example, decimal 1234.567 is normalized as 1.234567×10^3 by moving the decimal point so that only one digit appears before the decimal. The exponent expresses the number of positions the decimal point was moved left (positive exponent) or moved right (negative exponent).Similarly, the floating-point binary value 1101.101 is normalized as 1.101101×2^3 by moving the decimal point 3 positions to the left, and multiplying by 2^3. Here are some examples of normalizations:

Normalization of a +Ve binary Number:

For a positive number, the bits in the mantissa are shifted left until the most significant bits are

Floating-point binary representation
0 001 0100
0 010 0011
0 100 0010 ← **Normalized**

0 followed by **1**. For **each shift left** the value of the **exponent is reduced by 1.**

Normalization of a −Ve binary Number:

Floating-point binary representation
1 110 0100
1 100 0011
1 000 0010 ← **Normalized**

The same process of shifting is used for a **negative number** until the most significant bits are **1** followed by **0.** In this case, **no attention is paid to the fact that bits are falling off the most significant end of the mantissa.**

What are Overflow and Underflow?

Overflow occurs when calculations produce results exceeding the capacity of the result.

Example:
16-bit integers can hold numbers in the range **-32768...32767**. So what happens when you add **20000 to 20000?**

$$1 \quad 111 \quad\quad 1 \quad \text{(carry digits)}$$
$$0100111000100000$$
$$+ \; 0100111000100000$$
$$\overline{}$$
$$1001110001000000$$

The sixteenth bit contains a '**1**' as a result of adding the two numbers. Yet, numbers with a '**1**' in the leading position are interpreted as *negative numbers*, so instead of '**40000**', the result is interpreted as '**-25536**'.

Overflow can also occur in the exponent of a floating point number, when the exponent has become too large to be represented using the given representation for floating-point numbers (e.g, 7 bits for 32-bit integers, or exponents larger than 63).

Underflow:

A calculation resulting in a number so small that the negative number used for the exponent is beyond the number of bits used for exponents is called *underflow* (e.g, 7 bits for 32-bit integers, or exponents smaller than -64).

The term arithmetic **underflow** (or "floating point **underflow**", or just "**underflow**") is a condition in a computer program where the result of a calculation is a **number** of smaller absolute value than the computer can actually store in memory.

Overflow:

A **CPU** with a capacity of 8 **bits** has a capacity of up to 11111111 in **binary**. **If one more bit was added there would be an overflow error**.

An explanation of binary overflow errors

Download Transcript

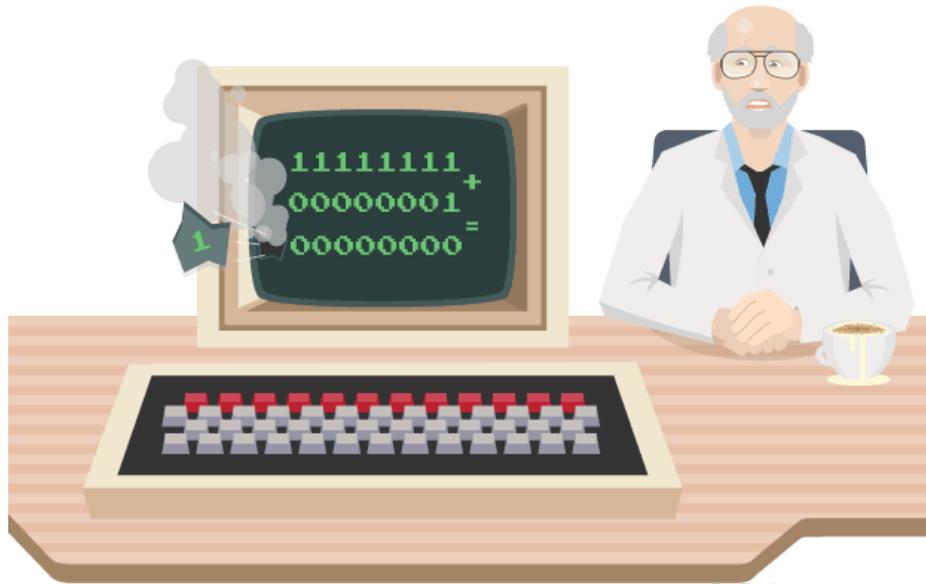## Example: 8-bit overflow

An example of an 8-bit overflow occurs in the binary sum **11111111 + 1 (denary: 255 + 1).**

```
  1 1 1 1 1 1 1 1
+ 1 0,0,0,0,0,0,0,1
  ─────────────────
  1 0 0 0 0 0 0 0 0
```

The total is a number **bigger than 8 digits**, and when this happens the **CPU drops the overflow digit** because the computer cannot store it anywhere, and **the computer thinks 255 + 1 = 0**.

Rounding errors

Because floating-point numbers have a limited number of digits, they cannot represent all real numbers accurately: when there are more digits than the format allows, the leftover ones are omitted - the number is *rounded*. There are three reasons why this can be necessary:

- **Large Denominators:** In any base, the larger the denominator of an (irreducible) fraction, the more digits it needs in positional notation. A sufficiently large denominator will require rounding, no matter what the base or number of available digits is. For example, 1/1000 cannot be accurately represented in less than 3 decimal digits, nor can any multiple of it (that does not allow simplifying the fraction).
- **Periodical digits:** Any (irreducible) fraction where the denominator has a prime factor that does not occur in the base requires an infinite number of digits that repeat periodically after a certain point.

  For example, in decimal 1/4, 3/5 and 8/20 are finite, because 2 and 5 are the prime factors of 10. But 1/3 is not finite, nor is 2/3 or 1/7 or 5/6, because 3 and 7 are not factors of 10. Fractions with a prime factor of 5 in the denominator can be finite in base 10, but not in base 2 - the biggest source of confusion for most novice users of floating-point numbers.

- **Non-rational numbers** Non-rational numbers cannot be represented as a regular fraction at all, and in positional notation (no matter what base) they require an infinite number of non-recurring digits.

Many new programmers become aware of binary floating-point after seeing their programs give odd results: "Why does my program print 0.10000000000000001 when I enter 0.1?"; "Why does 0.3 + 0.6 = 0.89999999999999991?"; "Why does 6 * 0.1 not equal 0.6?" Questions like these are asked every day, on online forums like stackoverflow.com.

The answer is that most decimals have infinite representations in binary. Take 0.1 for example. It's one of the simplest decimals you can think of, and yet it looks so complicated in binary:

```
0.0001100110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
00110011001100110011001100110011001100110011001100110011001100110011001100110011
0011...
```

Decimal 0.1 in Binary (To 1369 Places)
The bits go on forever; no matter how many of those bits you store in a computer, you will never end up with the binary equivalent of decimal 0.1.

0.1 In Binary
0.1 is one-tenth, or 1/10. To show it in binary — that is, as a bicimal — divide binary 1 by binary 1010, using binary long division:

Computing One-Tenth In Binary
The division process would repeat forever — and so too the digits in the quotient — because 100 ("one-zero-zero") reappears as the working portion of the dividend. Recognizing this, we can abort the division and write the answer in repeating bicimal notation, as 0.00011.

Summary
In pure math, every decimal has an equivalent bicimal. In floating-point math, this is just not true.

```
              ___
         0.00011
1010 )1.000000
       0 |
       10
        0 |
      (100)
         0 |
       1000
          0 |
      10000
       1010 |
        1100
        1010 |
         100
```

Even with 10, 20, or 100 digits, you would need to do some rounding to represent an infinite number in a finite space. If you have a lot of digits, your rounding error might seem insignificant. But consider what happens if you add up these rounded numbers repeatedly for a long period of time. If you round **1/7 to 1.42 x 10^{-1} (0.142)** and add up this representation 700 times, you would expect to get 100. (1/7 x 700 = 100) but instead you get 99.4 (0.142 x 700).

Relatively small rounding errors like the example above can have huge impacts. Knowing how these rounding errors can occur and being conscious of them will help you become a better and more precise programmer.

Errors due to rounding have long been the bane of analysts trying to solve equations and systems. Such errors may be introduced in many ways, for instance:

- inexact representation of a constant
- integer overflow resulting from a calculation with a result too large for the word size
- integer overflow resulting from a calculation with a result too large for the number of bits used to represent the mantissa of a floating-point number
- accumulated error resulting from repeated use of numbers stored inexactly

Summary

Rounding error is a natural consequence of the representation scheme used for integers and floating-point numbers in digital computers. Rounding can produce highly inaccurate results as errors get propagated through repeated operations using inaccurate numbers. Proper handling of rounding error may involve a combination of approaches such as use of high-precision data types and revised calculations and algorithms. Mathematical analysis can be used to estimate the actual error in calculations.

Past Paper question: 9608/33/O/N/15

(c) A student writes a program to output numbers using the following code:

```
X ← 0.0
FOR i ← 0 TO 1000
    X ← X + 0.1
    OUTPUT X
ENDFOR
```

The student is surprised to see that the program outputs the following sequence:

0.0   0.1   0.2   0.2999999   0.3999999 ......

Explain why this output has occurred.

Solution:
Any one of the below mentioned answers:
- o   cannot be represented exactly in binary
- o   0.1 represented here by a value just less than 0.1
- o   the loop keeps adding this approximate value to counter
- o   until all accumulated small differences become significant enough to be seen

Refrences
- Book: AS and A-level Computer Science by
- http://www.bbc.co.uk/education/guides/zjfgjxs/revision/1
- https://www.tutorialspoint.com/vb.net/vb.net_constants.htm
- https://www.cs.drexel.edu/~introcs/F2K/lectures/5_Scientific/overflow.html
- https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/underflow.html