Chapter 1

Computer Number Systems and Floating Point Arithmetic

1.1 Introduction

The main objective of this chapter is to introduce the students to modes of storage of numbers in the computer memory as well as to computer arithmetic.
In this view, we start by describing computer number representation in the binary system that uses 2 as the base. Since the usual decimal system uses base 10, we discuss therefore methods of conversion from one base to another. The octal and hexadecimal systems (respectively, base 8 and base 16 systems) are also introduced as they are often needed as intermediate stages between the binary and decimal systems. Furthermore, the subsequent hexadecimal notation is used to represent internal contents of stored numbers.
Since all machines have limited resources, not all real numbers can be represented in the computer memory; only a finite subset $\mathbb{F}$ of $\mathbb{R}$ is effectively dealt with. More precisely, $\mathbb{F}$ is a proper subset of the rationals, with $\mathbb{F} \subset Q \subset \mathbb{R}$. We shall therefore define first in general, normalized floating point systems $\mathbb{F}$ representing numbers in base $\beta \in \mathbb{N}$, $\beta \geq 2$ with a fixed precision p, and analyze particularly the standard IEEE single precision $\mathbb{F}_s$ and double precision $\mathbb{F}_d$ binary systems.
Moreover, the arithmetic performed in a computer is not exact; $\mathbb{F}$ is characterized by properties that are different from those in $\mathbb{R}$. We present therefore floating point arithmetic operations in the last sections of this chapter.

Note that IEEE stands for " Institute for Electrical and Electronics Engineers". The IEEE Standard for floating point arithmetic (IEEE 754) is the most widely used standard for floating point operation and is followed by many hardware and software implementation; most computers languages allow or require that some or all arithmetic be carried out using IEEE formats and operations.
For any base $\beta \in \mathbb{N}$, $\beta \geq 2$, we associate the set of symbols S_β, which consists of β distinct symbols. To illustrate, we have the following examples:

$$S_{10} = \{0, 1, .., 9\},$$

$$S_2 = \{0, 1\},$$

$$S_{16} = \{0, 1, .., 9, A, B, C, D, E, F\}.$$

The general representation of $x \in \mathbb{R}$ in base β is given by:

(1.1)
$$x = \pm(a_N\beta^N + ... + a_1\beta + a_0 + a'_1\beta^{-1} + ... + a'_p\beta^{-p}) = \pm(a_N a_{N-1}...a_1 a_0 \cdot a'_1...a'_p)_\beta$$

where $0 \leq N < \infty$, $1 \leq p \leq \infty$ and a_i, $a'_i \in S_\beta$, with $a_N \neq 0$ being the most significant digit in this number representation.
The number x is thus characterized by its sign $\pm$, its integral part $E(x) = \sum_{i=0}^{N} a_i\beta^i$ and its fractional part $F(x) = \sum_{i=1}^{p} a'_i\beta^{-i}$, leading to the following general expression of x:

$$x = \pm(E(x) + F(x))$$

or also equivalently: $x = \pm(E(x).F(x))$
Note that in case $p = \infty$, the fractional part of x is said to be infinite.

Example 1.1. • *The octal representation of* 0.36207 *is:*

$$(0.36207)_8 = 3 \times 8^{-1} + 6 \times 8^{-2} + 2 \times 8^{-3} + 7 \times 8^{-5}$$

• *The decimal representation of* 57.33333... *is :*

$$(57.33333...)_{10} = (57.\overline{3})_{10} = 5 \times 10 + 7 + 3 \times 10^{-1} + 3 \times 10^{-2} + ...$$

• *The hexadecimal representation of* $4.A02C$ *is :*

$$(4.A02C)_{16} = 4 + A \times 16^{-1} + 2 \times 16^{-2} + C \times 16^{-3}$$

1.2 Conversion from base 10 to base 2

Assume that a number $x \in \mathbb{R}$ is given in base 10, whereby:

$$x = \pm(d_N 10^N + ... + d_1 10 + d_0 + d'_1 10^{-1} + .. + d'_p 10^{-p}) = \pm(d_N d_{N-1} ... d_1 d_0 . d'_1 ... d'_p)_{10},$$

where d_i, $d'_i \in S_{10}$ $\forall i$, $d_N \neq 0$, and $p \leq \infty$. We seek its conversion to base 2, in a way that:

$$x = \pm(b_M 2^M + ... + b_1 2 + b_0 + b'_1 2^{-1} + ... + b'_l 2^{-l}) = \pm(b_M b_{M-1} ... b_1 b_0 . b'_1 ... b'_l)_2,$$

where b_i, $b'_i \in S_2$ $\forall i$, $b_M \neq 0$, $l \leq \infty$.
We convert successively the integral and fractional parts of x.

1.2.1 Conversion of the integral part

Starting with the integral part of x, $E(x)$ and writing:

$$E(x) = d_N 10^N + ... + d_1 10 + d_0 = b_M 2^M + ... + b_1 2 + b_0, \tag{1.2}$$

one has to find the sequence $\{b_i | i = 0, ..., M\}$ in S_2, given the sequence $\{d_i | i = 0, ..., N\}$ in S_{10}. Both sequences are obviously finite. The conversion is done using the successive division algorithm of positive integers based on the Euclidean division theorem stated as follows:

Theorem 1.1. *Let D and d be two positive integers. There exist 2 non negative integers q (the quotient) and r (the remainder), such that $r \in \{0, 1, 2, ..., d-1\}$, verifying:*

$$D = d \times q + r.$$

For notation purpose, we write $q = D \operatorname{div} d$ and $r = D \bmod d$.

Remark 1.1. *When $D < 0$ and $d > 0$, one has:*

$$D = q \times d + r, \text{ with } q = \lfloor \frac{D}{d} \rfloor < 0.$$

where $\lfloor r \rfloor : \mathbb{R} \to Z$ designates the "floor function" of the real number r.

On the base of (1.2), if $E(x) = D$, then one seeks:

$$D = E(x) = (b_M 2^{M-1} + ... + b_1) \times 2 + b_0$$

where

$$(b_M 2^{M-1} + ... + b_1) = D\, div\, 2 \quad and \quad b_0 = D\, mod\, 2.$$

Thus if D is divided once by 2, the remainder in this division is b_0. We can repeat this argument taking then $D = b_M 2^{M-1} + ... + b_1$ to find b_1, then following a similar pattern, compute successively all remainders $b_2, ..., b_M$. The process is stopped as soon as the quotient of the division is identical to zero.
The corresponding `MATLAB` function can then be easily implemented as follows:

Algorithm 1.1. Integer Conversion from Base 10 to 2

```
% Input: D an integer in decimal representation
% Output: string  s of  binary symbols (0's and 1's) representing D in base 2
% All arithmetic is based on rules of the decimal system
function s = ConvertInt10to2(D)
s=[ ];
while D>0
%Divide D by 2, calculate the quotient q and the remainder r, then add r
%in s from right to left
        q=fix(D/2);
        r= D - 2*q ;
        s=[r s];
        D=q;
end
```

As an application, consider the following example.

Example 1.2. *Convert the decimal integer $D = 78$ to base 2.*

Using the above algorithm, we have successively:
$78 = 39 \times 2 + 0$
$39 = 19 \times 2 + 1$
$19 = 9 \times 2 + 1$
$9 = 4 \times 2 + 1$
$4 = 2 \times 2 + 0$
$2 = 1 \times 2 + 0$
$1 = 0 \times 2 + 1.$
Hence, one concludes that $(78)_{10} = (1001110)_2$. ■

We can now introduce base 8 in order to shorten this procedure of conversion. The octal system is particularly useful when converting from the decimal system to the binary system, and vice versa. Indeed, if

$$E(x) = b_M 2^M + ... + b_3 2^3 + b_2 2^2 + b_1 2 + b_0\,, \text{ with } b_i \in \{0, 1\},$$

we can group the bits 3 by 3 from right to left (supplying additional zeros if necessary), then factorize successively the positive powers of 8, i.e. $8^0, 8^1, 8^2, ...$ to have:

$$E(x) = ... + (b_5 2^5 + b_4 2^4 + b_3 2^3) + (b_2 2^2 + b_1 2 + b_0)$$

then equivalently:

$$E(x) = ... + (b_8 2^2 + b_7 2 + b_6)8^2 + (b_5 2^2 + b_4 2 + b_3)8^1 + (b_2 2^2 + b_1 2 + b_0)8^0$$

$$= \sum_{i=0}^{l} (b_{3i+2} 2^2 + b_{3i+1} 2 + b_{3i}) 8^i$$

Letting $o_i = b_{3i+2} 2^2 + b_{3i+1} 2 + b_{3i}$, one writes then the integral part as follows:

$$E(x) = \sum_{i=0}^{l} o_i 8^i$$

Note that for all values of i, $0 \leq o_i \leq 7$, implying that o_i is an octal symbol. The table of conversion is set up according to the following representations:

Octal symbol	**Group of 3 binary bits** $o_i = b_{3i+2} b_{3i+1} b_{3i}$
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

table 1. Conversion of octal symbols to base 2

Thus, to convert from base 2 to base 8, groups of 3 binary digits can be translated directly to octal symbols according to the above table. Conversion of an octal number to binary can be done in a similar way but in reverse

order; i.e. just replace each octal digit with the corresponding 3 binary digits. To convert an integer from base 10 to base 2, we can therefore start by converting it to base 8:

$$(E(x))_{10} \to (E(x))_8 \to (E(x))_2$$

The algorithm implementing this conversion process is the following:

Algorithm 1.2. Integer Conversion from Base 10 to 8

```
% Input D=E(x) integer in  decimal representation
% Output : string  s of octal symbols
% All arithmetic is based on rules of the decimal system
function s=ConvertInt10to8(D)
s=[ ] ;
While D>0
   r=rem(D, 8) ;
   D=fix(D/8) ;
   s=[r s] ;
 end
```

In the preceding example, using this algorithm we have successively:
$78 = 9 \times 8 + 6$
$9 = 1 \times 8 + 1$
$1 = 0 \times 8 + 1$.
Hence, $(78)_{10} = (116)_8$ through 3 successive divisions by 8.
Referring to the above table of conversion we obviously deduce that:

$$(78)_{10} = (116)_8 = (001\,001\,110)_2 = (1001110)_2$$

1.2.2 Conversion of the fractional part

To convert the fractional part $F(x)$ of the decimal x, we introduce the **successive multiplication algorithm**. Its principle runs as follows: given the sequence $\{d'_i\} \in S_{10}$, we seek the sequence $\{b'_i\} \in S_2$ with:

$$F(x) = d'_1 10^{-1} + .. + d'_p 10^{-p} = b'_1 2^{-1} + ... + b'_l 2^{-l} \tag{1.3}$$

Let $f = F(x)$. Note then the following identity:

$$2f = b'_1 + b'_2 2^{-1} ... + b'_l 2^{1-l} = b'_1 \cdot b'_2 \cdot ... \cdot b'_{l-1}.$$

Obviously through one multiplication of f by 2, the integral and fractional parts of $2f$ are respectively:

$$E(2f) = b_1' \text{ and } F(2f) = b_2'2^{-1} ... + b_l'2^{1-l}$$

We can therefore repeat the same procedure, of multiplication by 2, to find successively b_2', then b_3', ... ,b_l'. The corresponding algorithm is the following:

Algorithm 1.3. Fraction Conversion from base 10 to base 2

```
% Input: F, fractional part of a decimal number 0<F<1
%        k, maximum number of binary bits required for binary fractional part
% Output:  string s  (up to k bits) representing F in base 2
function s=ConvertFrac10to2(F,k)
s=[  ]  ;
i=1;
while F>0 & i<=k
    G=2*F;
    b=fix(G);
    F=G-b;
    s = [ s b ] ;
    i=i+1;
end
```

Note that if f has an infinite representation in base 10, its representation in base 2 will also be infinite. However, we could have situations where f is finitely represented in base 10 and infinitely represented in base 2. To illustrate, consider the following examples.

Example 1.3. *Convert* $(0.25)_{10}$ *to base 2.*

We apply the above algorithm to get successively:
$2 \times 0.25 = 0 + 0.5$
$2 \times 0.5 = 1 + 0.0$
Thus $(0.25)_{10} = (0.01)_2$. ■

Example 1.4. *Convert* $(0.1)_{10}$ *to base 2.*

Applying the same non-terminating procedure, we have:
$2 \times 0.1 = 0 + 0.2$
$2 \times 0.2 = 0 + 0.4$
$2 \times 0.4 = 0 + 0.8$

$2 \times 0.8 = 1 + 0.6$
$2 \times 0.6 = 1 + 0.2$
$2 \times 0.2 = 0 + 0.4$
$2 \times 0.4 = 0 + 0.8$
$2 \times 0.8 = 1 + 0.6$
$2 \times 0.6 = 1 + 0.2$
...
Thus $(0.1)_{10} = (0.0001100110011...)_2 = (0.0\overline{0011})_2$. ■
We end up with an example where both representations are infinite.

Example 1.5. *Convert* $\frac{1}{3}$ *to base 2.*

Let us apply the successive multiplication algorithm to this fraction:
$2 \times \frac{1}{3} = 0 + \frac{2}{3}$
$2 \times \frac{2}{3} = 1 + \frac{1}{3}$
..... ■
Hence: $\frac{1}{3} = (0.\overline{3})_{10} = (0.0101...)_2 = (0.\overline{01})_2$

Of course, base 8 can also be used as an intermediate stage:

$$(F(x))_{10} \to (F(x))_8 \to (F(x))_2$$

By grouping the bits 3 by 3 from left to right, supplying additional zeros if necessary, then factorizing successively negative powers of 8: 8^{-1}, 8^{-2}, ... one establishes through these steps the following identities:

$$F(x) = (b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3}) + (b_4 2^{-4} + b_5 2^{-5} + b_6 2^{-6}) + ...$$

$$= (b_1 4 + b_2 2 + b_3)8^{-1} + (b_4 4 + b_5 2 + b_6)8^{-2} + ... = o_1 8^{-1} + o_2 8^{-2} + ...$$

We can then have a new version of the successive multiplication by 8 algorithm converting a fractional decimal to octal, followed by a final conversion to a binary fractional using the table of conversion.
To illustrate, consider the following examples.

Example 1.6. *Convert* $(0.75)_{10}$ *to base 2, using base 8 as intermediate.*

A straightforward application of the procedure above yields: $8 \times 0.75 = 6 + 0.00$. Hence:

$$(0.75)_{10} = (0.6)_8 = (0.110)_2 = (0.11)_2$$

Example 1.7. *Convert $x = (0.12)_{10}$ to base 2, using base 8 as intermediate. Do not exceed 21 bits for the representation of x in base 2.*

Getting 21 bits in base 2 means reaching 7 digits in base 8. Therefore one only needs to apply 7 successive multiplications by 8. This yields:
$8 \times 0.12 = 0 + 0.96$
$8 \times 0.96 = 7 + 0.68$
$8 \times 0.68 = 5 + 0.44$
$8 \times 0.44 = 3 + 0.52$
$8 \times 0.52 = 4 + 0.16$
$8 \times 0.16 = 1 + 0.28$
$8 \times 0.28 = 2 + 0.24$
...
Hence $(0.12)_{10} = (0.0753412...)_8 = (0.000\,111\,101\,011\,100\,001\,010\,...)_2$. ■

1.3 Conversion from base 2 to base 10

We consider in this section inverse procedures that convert numbers from base 2 (or 8) to base 10. For a real number x, this is performed as precedingly on the integral part $E(x)$ first, then on the fractional part $F(x)$, Of course, the successive division and multiplication algorithms can be applied. However, this would mean dividing or multiplying successively by 10 and performing the arithmetic operations in base 2 (or 8). Instead, we follow up a straightforward **polynomial evaluation** process, with the arithmetic being performed in base 10. We start by discussing this last issue.

1.3.1 Polynomial evaluation

Consider the polynomial $p_n(y)$ of degree n, with real coefficients $\{a_i | i = 0, 1..., n\}$ and $a_n \neq 0$:

$$p_n(y) = a_0 + a_1 y + ... + a_{n-1} y^{n-1} + a_n y^n \quad ; \quad y \in \mathbb{R}$$

A first way to evaluate $p_n(y)$ is by using a straightforward sum of products, as indicated in the following algorithm:

Algorithm 1.4. Direct Polynomial Evaluation

```
function p=EvaluatePolyStraight(a,y)
```

```
% Input a=[a(1),...,a(n+1)] and y
% Output Value of p(y)=a(n+1)*y^n+a(n)*y^{n-1}+...+a(2)*y+a(1)$
n=length(a)-1;
t=y;p=a(1);
for i=2:n+1
   p=p+a(i)*t;
   t=t*y;
end
```

This algorithm requires n additions and $2n$ multiplications.
A more efficient algorithm called **Horner's algorithm**, uses **nested evaluation**. One starts by writing the given polynomial in nested form as shown below:

$$p_n(y) = a_n y^n + a_{n-1} y^{n-1} + ... + a_1 y + a_0 = (a_n y + a_{n-1}) y^{n-1} + a_{n-2} y^{n-2} + ... + a_1 y + a_0$$

$$= ((a_n y + a_{n-1}) y + a_{n-2}) y^{n-2} + ... + a_1 y + a_0 = (((a_n y + a_{n-1}) y + a_{n-2}) y + a_{n-3}) y^{n-3} ... + a_1 y + a_0$$

$$= (...(((a_n y + a_{n-1}) y + a_{n-2}) y + a_{n-3}) y + ... + a_1) y + a_0$$

This method can be implemented as follows :

Algorithm 1.5. Nested Polynomial Evaluation

```
% Input a=[a(1),...,a(n+1)] and y
% Output Value of p(y)=a(n+1)*y^n+a(n)*y^{n-1}+...+a(2)*y+a(1)$
function p=EvaluatePolyNested(a,y)
n=length(a)-1;
p=a(n+1);
for i=n:-1:1
   p=p*y+a(i);
end
```

Such procedure requires n multiplications and n additions, i.e. a total of 2n operations, that is 2/3 of the number of arithmetic operations in the previous algorithm. Thus, to minimize the number of arithmetic calculations, polynomials should always be expressed in nested form before performing an evaluation.

Example 1.8. *Write* $f(x) = 5x^3 - 6x^2 + 3x + 1$ *in nested form.*

$$f(x) = 5x^3 - 6x^2 + 3x + 1 = ((5x - 6)x + 3)x + 1$$

1.3.2 Conversion of the integral part

Rewriting identity (1.2) as:

$$E(x) = b_M 2^M + ... + b_1 2 + b_0 = d_N 10^N + ... + d_1 10 + d_0,$$

one seeks now to find the sequence $\{d_i\}$ in S_{10} given the sequence $\{b_i\}$ in S_2. Indeed, note that $E(x) = p_M(2)$, where p_M is the polynomial of degree M given by:

$$p_M(y) = b_M y^M + ... + b_1 y + b_0.$$

Hence finding $E(x)$ in base 10 reduces to the evaluation, using decimal arithmetic of the polynomial $p_M(y)$, for $y = 2$. In case one wants to use the octals as intermediates, the bits are first grouped 3 by 3 to write $E(x)$ as a polynomial in powers of 8, based on the table of conversion. That is:

$$E(x) = o_L 8^L + ... + o_1 8 + o_0 = q_L(8),$$

where q_L is a polynomial of degree L given by $q_L(y) = o_L y^L + ... + o_1 y + o_0$. Using decimal arithmetic, one computes then $q_L(y)$ for $y = 8$.

Example 1.9. *Convert the binary integer* $D = (01110101110011)_2$ *to base 10, using base 8 as intermediate.*

We first convert D to base 8 using the table of conversion:

$$D = (01110101110011)_2 = (001\,110\,101\,110\,011)_2 = (16563)_8 = 1{\times}8^4{+}6{\times}8^3{+}5{\times}8^2{+}6{\times}8{+}3.$$

Thus, using nested polynomial evaluation, one gets:

$$D = (((8 + 6)8 + 5)8 + 6)8 + 3 = (7539)_{10}.$$

1.3.3 Conversion of the fractional part

Given the sequence $\{b'_i\} \in S_2$, we seek now the sequence $\{d'_i\} \in S_{10}$, such that:

$$F(x) = f = b'_1 2^{-1} + ... + b'_l 2^{-l} = d'_1 10^{-1} + ... + d'_p 10^{-p}$$

Using decimal arithmetic, the evaluation of f is based on the following steps:

$$f = b'_1 2^{-1} + ... + b'_l 2^{-l} = 2^{-l}(b'_1 2^{l-1} + ... + b'_l)$$

that is, using nested polynomial evaluation:

$$f = 2^{-l} p_{l-1}(2),$$

where obviously:

$$p_{l-1}(y) = b'_1 y^{l-1} + b'_2 y^{l-2} ... + b'_l.$$

Clearly then, to use base 8 as an intermediate, through grouping the bits 3 by 3, then referring to the table of conversion, one gets a polynomial expression in negative powers of 8, specifically:

$$f = o'_1 8^{-1} + ... + o'_{k-1} 8^{-k+1} + o'_k 8^{-k}$$

Equivalently,

$$f = 8^{-k}(o'_1 8^{k-1} + ... + o'_{k-1} 8 + o'_k) = 8^{-k} q_{k-1}(8),$$

with $q_{k-1}(y) = o'_1 y^{k-1} + ... + o'_{k-1} y + o'_k$.
To illustrate consider the following example.

Example 1.10. *Convert the fractional octal* $f = (0.00111000111)_2$ *to base 10. Use base 8 as intermediate.*

We start by converting f to base 8, yielding:

$$f = (0.1616)_8 = 1{\times}8^{-1}{+}6{\times}8^{-2}{+}1{\times}8^{-3}{+}6{\times}8^{-4} = 8^{-4}(1{\times}8^3{+}6{\times}8^2{+}1{\times}8{+}6)$$

Through nested evaluation,

$$8^3 + 6 \times 8^2 + 8 + 6 = ((8+6)8+1)8+6 = 910.$$

Thus:

$$f = 8^{-4} \times 910 = \frac{910}{4096} = 0.2221679$$

■

1.4 Normalized floating point systems

1.4.1 Introductory concepts

Recall that a standard way to represent a real number in decimal form is with a sign (+ or -), an integral part, a fractional part and a decimal point in between, for example: $+32.875$ or -0.0082.

Another standard computer notation called the **normalized floating point representation**, is obtained by shifting the decimal point and supplying appropriate powers of 10. Thus the preceding numbers have an alternate representation respectively as $+3.2875 \times 10^1$, or -8.2×10^{-3}.
In general, a non-zero real number x in the base β is written in the standard normalized floating point form:

$$x = \pm\ m\ \times \beta^e$$

where m is called the **mantissa**, with $1 \leq m < \beta$ and e the **exponent**, being a positive or negative integer. These parameters are obtained from (1.1) by writing:

$$x = \pm(a_N\beta^N + a_{N-1}\beta^{N-1} + ... + a'_p\beta^{-p}) = \pm(a_N + a_{N-1}\beta^{-1} + ... + a'_p\beta^{-(p+N)}) \times \beta^N$$

where $a_N \neq 0$, thus leading to

$$m = a_N + a_{N-1}\beta^{-1} + a_{N-2}\beta^{-2} + ... + a'_p\beta^{-(p+N)},\ \text{and}\ e = N$$

Remark 1.2. *If the number x has a non terminating fractional part, in some cases the mantissa m can reach the value β.*

For example, consider the following decimal number x:

$$x = 0.9999999... = 9 \times 10^{-1} + 9 \times 10^{-2} + ...$$

The normalized floating point representation of x is:

$$x = (9 + 9 \times 10^{-1} + 9 \times 10^{-2} + ...) \times 10^{-1} = 9.99999999... \times 10^{-1}$$

Thus, the mantissa is infinite with

$$m = 9.\overline{9} = 9(1 + \tfrac{1}{10} + \tfrac{1}{10^2} + \tfrac{1}{10^3} + ...) = 9\tfrac{1}{1-1/10} = 10 = \beta$$

Example 1.11. *Base 10, 2 and 8 representations of $\frac{1}{3}$ in normalized floating point notations.*

1. In the normalized floating point notation, $\frac{1}{3}$ in base 10 is expressed as follows:
$$\frac{1}{3} = (0.\overline{3})_{10} = 3.\overline{3} \times 10^{-1}.$$
Thus, in such system, the mantissa $m = 3.\overline{3}$ and the exponent $e = -1$.

2. However in base 2 (Example 1.5), it becomes:

$$\frac{1}{3} = (0.\overline{01})_2 = (0.0101010101...)_2 = 1.01010101... \times 2^{-2} = 1.\overline{01} \times 2^{-2},$$

i.e. the mantissa is $m = 1.\overline{01}$ and the exponent $e = -2$.

3. Finally, to convert $\frac{1}{3}$ to base 8:

$$\frac{1}{3} = (0.010101010101...)_2 = (0.2525...)_8 = 2.\overline{52} \times 8^{-1}.$$

where $m = 2.\overline{52}$ and $e = -1$.

■

Example 1.12. *Write the binary number $x = (11001.0111)_2$ in the normalized floating point notation.*

$$x = (11001.0111)_2 = 1.10010111 \times 2^4$$

■

Note that every computer system has a finite total capacity and a finite word length. Numbers used in calculations within a computer system must conform to the format imposed in that system; only real numbers with a finite number of digits can be represented, leading then to a strictly limited degree of precision. Real numbers representable in a computer are called **machine numbers**, and are written in a standard format.
A **floating point system** $\mathbb{F}$ consists of machine numbers and is defined as follows:

Definition 1.1. *A* **normalized floating-point system** $\mathbb{F} = F(\beta, p, e_{\min}, e_{\max})$ *is the set of all real numbers written in normalized floating point form $x = \pm\, m \times \beta^e$ where m is the mantissa of x and e its the exponent , such that:*

1. *If $x \neq 0$, then $m = m_0 + m_1\beta^{-1} + ... + m_{p-1}\beta^{-(p-1)}$; with $m_i \in S_\beta$, $m_0 \neq 0$, and $e_{min} \leq e \leq e_{max}$*

2. *If $x = 0$, then $m = 0$, while e could take any value or be selected according to other criteria.*

The main parameters of a floating-point system $F = F(\beta, p, e_{min}, e_{max})$, are:

1. The **base** β
2. The **number of significant digits** p, called the **precision** of the system which is a finite positive integer that could be given a specific value (IEEE systems) or be defined by the user (MATHEMATICA or MAPLE)
3. The **range of the exponent** $[e_{\min}, e_{\max}]$, with $e_{\min} < 0$ and $e_{\max} = |e_{\min}| + 1$
4. A **convention for representing zero**

Note that since there is a complete symmetry with respect to zero, between the positive and negative elements of $\mathbb{F}$, we will analyze and prove in what follows properties of the positive elements only.

Theorem 1.2. *Let $x \in \mathbb{F} = F(\beta, p, e_{\min}, e_{\max})$, with $x = +\, m \times \beta^e$ and $x \neq 0$.*

1. $1 \leq m < \beta$,
2. $x_{\min} \leq x \leq x_{\max}$, *where*
$$x_{\min} = \beta^{e_{\min}}$$
and
$$x_{\max} = (\beta - 1)(1 + \beta^{-1} + ... + \beta^{-p+1})\beta^{e_{\max}} < \beta \times \beta^{e_{\max}}.$$
3. *If $x = +m \times \beta^e \in \mathbb{F}$ with $x_{min} \leq x < x_{max}$, then the successor of x is given by*
$$succ(x) = x + \beta^{1-p}\beta^e$$
leading to:
$$\frac{succ(x) - x}{x} \leq \beta^{-p+1}.$$

Proof.

1. The first part of the theorem is obtained straightforwardly from the definition.

2. It is enough to note that the minimum value of m is reached when $a_0 = 1$ and $a_i = 0$, for $1 \leq i \leq p-1$ i.e. $m = 1$, while the maximum is obtained when $a_i = \beta - 1$ for all $0 \leq i \leq p-1$. In this case $m = (\beta-1)(1 + 1/\beta + ... + (1/\beta)^{p-1}) = \beta(1 - (1/\beta)^p) < \beta$.

3. As for the third part, if $x = (m_0 + m_1\beta^{-1} + ... + m_{p-1}\beta^{-(p-1)})\beta^e$, then the successor of x is obtained by adding 1 unit to the least significant digit of its mantissa, leading to the following identity:

$$succ(x) = x + \beta^{-p+1}\beta^e = (m + \beta^{-p+1})\beta^e \tag{1.4}$$

Thus $succ(x) - x = \beta^{-p+1}\beta^e$ and

$$\frac{succ(x) - x}{x} = \frac{\beta^{-p+1}\beta^e}{m \times \beta^e} = \frac{\beta^{-p+1}}{m} \leq \beta^{-p+1} \tag{1.5}$$

since $m \geq 1$. ■

Definition 1.2. *In a floating point system $F(\beta, p, e_{\min}, e_{\max})$, the* **system epsilon** *or* **epsilon machine** *is defined by the parameter ϵ_M:*

$$\epsilon_M = \beta^{-p+1}.$$

Clearly ϵ_M is a measure of the precision of the system, since according to (1.5) it is a maximum bound on the relative distance between two consecutive numbers in $F(\beta, p, e_{\min}, e_{\max})$. Furthermore, note that equation (1.4) can be written as:

$$succ(x) = (m + \beta^{-p+1})\beta^e$$

from which one concludes that ϵ_M also represents the difference between the mantissas of two successive positive numbers in F.

As a direct application, we consider the following example:

Example 1.13. *Display the elements of the floating point system $\mathbb{F} = F(10, 3, -2, +3)$.*

For non zero numbers, we shall display only the positive elements; the negative ones being deduced by symmetry.

Positive numbers in $F(10, 3, -2, 3)$
1.00×10^{-2}
1.01×10^{-2}
....
9.98×10^{-2}
9.99×10^{-2}
1.00×10^{-1}
1.01×10^{-1}
....
9.98×10^{1}
9.99×10^{-1}
1.00×10^{0}
1.01×10^{0}
....
9.98×10^{0}
9.99×10^{0}
1.00×10^{1}
1.01×10^{1}
....
9.98×10^{1}
9.99×10^{1}
1.00×10^{2}
1.01×10^{2}
....
9.98×10^{2}
9.99×10^{2}
1.00×10^{3}
1.01×10^{3}
....
9.98×10^{3}
9.99×10^{3}

In this decimal floating point system, the following parameters in $\mathbb{F}$ are easily computed:

- $x_{\min} = 1.00 \times 10^{-2}$
- $x_{max} = 9.99 \times 10^{3}$
- $\epsilon_M = 10^{-2} = 0.01$.

- To represent zero, one might consider ± 0. For that purpose, we adopt a convention whereby ± 0 is represented by a 0 mantissa, regardless of the exponent. Therefore zero $\in F(10, 3, -1, 2)$, and it is represented by $\pm\ 0.00 \times 10^e$ for any value of e.

- The total number of elements in F, is

$$card(\mathbb{F}) = 2 \times [(9 \times 10^2) \times 6] + 2 = 10802$$

Moreover, the absolute distances between 2 successive or neighbouring floating point numbers in $\mathbb{F}$, increase and are computed as follows:

Interval	**Neighboring numbers distance**
$[10^{-2}, 10^{-1})$	$\epsilon_M \times 10^{-2} = 10^{-4}$
$[10^{-1}, 1)$	$\epsilon_M \times 10^{-1} = 10^{-3}$
$[1, 10^1)$	$\epsilon_M \times 10^0 = 10^{-2} = \epsilon_M$
$[10^1, 10^2)$	$\epsilon_M \times 10^1 = 10^{-1}$
$[10^2, 10^3)$	$\epsilon_M \times 10^2 = 1$
$[10^3, 10^4)$	$\epsilon_M \times 10^3 = 10$

■

These results can be generalized and extended to any floating point system $\mathbb{F} = F(\beta, p, e_{min}, e_{max})$. Absolute distances decrease towards zero, on intervals that are subset of $(0, \beta)$ and in contrast these distances increase on intervals in $[\beta, x_{max}]$ towards x_{max}, with

$$\max_{x \in (-\beta, +\beta) \cap \mathbb{F}} |x - \text{succ}(x)| \leq \epsilon_M,$$

We note also that the ϵ-machine $\epsilon_M = \beta^{1-p}$ being the smallest upper bound of **relative distances** in $\mathbb{F}$ coincides with the smallest absolute distance between successive points **only** on the interval $[1, \beta)$. The following table summarizes such fact.

Interval	**Neighboring numbers distance**
.....	.....
$[1/\beta^3, 1/\beta^2)$	β^{-p-2}
$[1/\beta^2, 1/\beta)$	β^{-p-1}
$[1/\beta, \beta)$	β^{-p}
$[1, \beta)$	$\beta^{-p+1} = \epsilon_M$
$[\beta, \beta^2)$	β^{-p+2}
$[\beta^2, \beta^3)$	β^{-p+3}
.....	.....

Thus, when computing in $\mathbb{F}$, criteria for "numerical convergence" should be preferably established in terms of relative errors and not absolute ones.

1.4.2 IEEE floating point systems

A computer operating in binary normalized floating point mode represents numbers as described earlier except for the limitation imposed by the finite word length. In this section, we shall describe the **internal representation and storage** of numbers for IEEE floating point systems. Addressable words of 4 bytes (32 bits or digits) and 8 bytes (64 bits) are used respectively in single and double precision floating point systems referred to as $\mathbb{F}_s$ and $\mathbb{F}_d$. In what follows, we analyze some properties of these systems successively.

1. **IEEE single precision floating point system**
   By single-precision IEEE floating point numbers, we mean all acceptable numbers belonging to the normalized floating point system $\mathbb{F}_s$=$F(2, 24, -126, +127)$, where a non zero word x of 4 bytes is organized as follows:

$$x = \pm(1.f)_2 \times 2^e = (-1)^t(1.f)_2 \times 2^{c-127}$$

4 bytes, a total of 32 bits

t sign	biased exponent c	f part of mantissa m
1 bit	8 bits	23 bits

figure 2. A word of 4 bytes in IEEE single precision.

Remark 1.3.

(i) In $\mathbb{F}_s$, if $x \neq 0$, the 1^st bit in the mantissa is always 1, so that this bit does not have to be stored. The stored mantissa consists of the rightmost 23 bits and contains the fractional part f with an

understood binary point. So the mantissa actually corresponds to 24 binary digits since there is a **hidden bit**. Moreover, the mantissa of each non zero positive number is restricted by the mantissas of x_{min} and x_{max}, satisfying the following inequality:

$$1.000...000 \leq (1.f)_2 \leq 1.111....11$$

(ii) In order to store positive numbers only, the **biased exponent** c is introduced, with $e = c - 127$. The values of c in $\mathbb{F}_s$ are bounded as follows:

$$(0)_{10} = (00\ 000\ 000)_2 < c < (11\ 111\ 111)_2 = (255)_{10}$$

The values $c = 0$ and $c = 255$ are reserved for special machine numbers obtained in calculations, that are not elements of $\mathbb{F}_S$. Thus, the value $c = 0$ is reserved for ± 0 and the **subnormal or denormalized numbers** (in case of underflow in the computations), while the value $c = 255$ includes $\pm\infty$ (in case of overflow in the computations) and "undefined" **NaN** numbers as for example: $0/0, \infty/\infty, x_d/x_d, \infty - \infty, ...$ The sign of Nan has no meaning, but it may be predictable in some circumstances; most applications (as MATLAB for example) ignore its sign , and place such elements by "sort functions" at the high end of positive numbers. Note also that once generated, a NaN propagates through all subsequent computations.
The value of the biased exponent c in $\mathbb{F}_s$, $\forall\, x \neq 0$, is thus strictly restricted by the inequality:

$$(1)_{10} = (00\ 000\ 001)_2 \leq c \leq (11\ 111\ 110)_2 = (254)_{10}$$

or equivalently

$$-126 \leq e \leq +127.$$

Recalling that a "machine number" is any number representable within a system, we may extend Definition 1.1 as follows:

Definition 1.3. *Let x be a machine number in $\mathbb{F}_s(2, 24, -126, +127)$, where the biased exponent $c = e + 127$, then:*

(a) $1 \leq c \leq 254$ *i.e.* $-126 \leq e \leq 127$*: $x = (-1)^t(1.f) \times 2^{c-127}$. Moreover, if $t = 1$ then $x < 0$ and if $t = 0$ then $x > 0$.*

(b) $c = 0$*: this value is reserved for special number representations of 0 and de-normalized numbers defined as follows:*

- The case $c = f = 0$ is reserved for the zeros, where $|x| = 0$. By convention we write $x = \pm 0$
- The case $c = 0$, and $f \neq 0$, is used to fill the gap between 0 and x_{min} (or $-x_{min}$ and 0), with **de-normalized numbers**. *By convention, we write $x = x_d = \pm 0.f \times 2^{-126}$.*

(c) $c = 255$: this value of c is reserved for special number representations of $\pm\infty$ and NaN numbers defined as follows:
- The case $c = 255$ and $f = 0$ represents by convention $x = \pm\infty$.
- The case $c = 255$ and $f \neq 0$ represents by convention, "Not a Number" written as $x =$ **NaN**.

Some machine numbers of the system $\mathbb{F}_s$ are displayed in the following 2 tables.

c	**Number**	**Representation in** $F(2, 24, -126, 127)$
c=0	0	0.00...00
c=1	$x_{\min}$	$1.00..00 \times 2^{-126}$
....	....	....
c=127	1	$1.00..00 \times 2^{0}$
...	....	....
c=254	$x_{\max}$	$1.11...11 \times 2^{127}$

c	**f**	**e = c − 127**	**m**	**Number being represented**
0	0	Not Applicable	0.0	± 0
0	$\neq 0$	Not Applicable	$0.f$	$(-1)^t(0.f)2^{-126}$
$0 < c < 255$	any	$-127 < e < 128$	$1.f$	$(-1)^t(1.f)2^{c-127}$
255	0	Not Applicable	1.0	$\pm\infty$
255	$\neq 0$	Not Applicable	$1.f$	NaN (Not a Number)

Table 1. Values in **IEEE** - simple precision system.

We can next give the basic parameters of $\mathbb{F}_s$.

Parameter	**Expression**(base 2)	**Decimal value**
$x_{\min}$	2^{-126}	1.175494×10^{-38}
$x_{\max}$	$(1.1...1)_2 \times 2^{127} = 2^{128}(1 - 2^{-24})$	3.402824×10^{38}
ϵ_M	2^{-23}	1.192093×10^{-7}
p	24=23+implicit bit	≈ 7

Note that the machine epsilon $\epsilon_M = (2^{-23})_2 = (2^{1-24})_2 < (2 \times 10^{-7})_{10} < (10^{1-7})_{10}$. This implies that in a simple computation in base 10, approximately 7 significant decimal digits of accuracy may be obtained in single precision.
When more precision is needed, **double precision** can be used, in which case each double precision floating number is stored in 2 computer words in memory.

2. **IEEE double precision floating point system**
Definition 1.1 is also extended to define the IEEE double precision system $\mathbb{F}_d = F(2, 53, -1022, 1023)$, where a non zero number in standard floating point representation corresponds to:

$$x = \pm(1.f)_2 \times 2^e = (-1)^t(1.f)_2 \times 2^{c-1023}$$

where $e = c-1023$, with the biased exponent c verifying: $1 \leq c \leq 2046$.
The system $\mathbb{F}_d$ uses a word of 8 bytes organized as follows.

8 bytes, a total of 64 bits

t sign	biased exponent c	f part of mantissa m
1 bit	11 bits	52 bits

figure 3. A word of 8 bytes for IEEE double precision.

On the basis of the concepts explained above for $\mathbb{F}_s$, the number system $\mathbb{F}_d$ is displayed in the following table:

c	**f**	**e = c − 1023**	**m**	**Number being represented**
0	0	Not Applicable	0.0	± 0
0	$\neq 0$	Not Applicable	$0.f$	$(-1)^t(0.f)2^{-1022}$
$0 < c < 2047$	any	$-1023 < e < 1024$	$1.f$	$(-1)^t(1.f)2^{c-1023}$
2047	0	Not Applicable	1.0	$\pm\infty$
2047	$\neq 0$	Not Applicable	$1.f$	NaN (Not a Number)

Table 2. Values in **IEEE** - double precision system.

The basic parameters for $\mathbb{F}_d$ are displayed then, as follows:

Parameter	**Expression**(base 2)	**Decimal value**
$x_{\min}$	2^{-1022}	$2.2250738507201 \times 10^{-308}$
$x_{\max}$	$(1.1...1)_2 \times 2^{1023} = 2^{1024}(1 - 2^{-53})$	$1.79769313486231 \times 10^{308}$
ϵ_M	2^{-52}	$2.220446049250313 \times 10^{-16}$
p	53=52+implicit bit	≈ 16

The machine epsilon $\epsilon_M \approx 2^{-52} \approx 2.2 \times 10^{-16} < 10^{1-16}$. This implies that in a double computation approximately 16 significant decimal digits of precision are available.

Remark 1.4. *In the process of representing machine numbers in $\mathbb{F}_s$ or $\mathbb{F}_d$, it could be convenient to use the* **hexadecimal symbols** *(base 16) to get a more "compact" form of the storage. The symbols A, B, C, D, E, F represent 10, 11, 12, 13, 14, and 15 respectively, as displayed in the following table of equivalences:*

Hexadecimal	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Binary	*0000*	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*

Hexadecimal	*A*	*B*	*C*	*D*	*E*	*F*
Binary	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*

Representing then machine binary numbers with hexadecimal symbols is particularly easy. We need only regroup the binary digits from groups of 3 (as required in the octal system), to groups of 4. Note that the reverse procedure is also used.

Example 1.14. *Determine the hexadecimal representation of the decimal number $d = -52.234375$ in both single precision and double precision.*
We start by converting the given number to binary, then normalize it:

- *E(x)*=$(52)_{10} = (64)_8 = (110\ 100)_2$
- *F(x)*=$(0.234375)_{10} = (0.17)_8 = (0.001\ 111)_2$
- *Therefore:*
  $(52.234375)_{10} = (110\ 100.001\ 111)_2 = (1.101\ 000\ 011\ 110\ )_2 \times 2^5$

In $\mathbb{F}_s(2, 24, -126, +127)$:
- The normalized mantissa of d is m=1.101 000 011 110
- The exponent of d is $e = (5)_{10} = c - 127$ implying that the biased exponent is $c = (132)_{10} = (204)_8 = (10\ 000\ 100)_2$
The single precision machine representation of d is then:

$$[1100\ 0010\ 0101\ 0000\ 1111\ 0000\ 0000\ 0000]_2 = [C250F000]_{16}$$

In $\mathbb{F}_d(2, 53, -1022, +1023)$*:*
- The normalized mantissa of d *is* m= 1.101 000 011 110
- The exponent of d *is* $e = (5)_{10} = c - 1023$*, and the biased exponent is therefore* $c = (1028)_{10} = (2004)_8 = (10\ 000\ 000\ 100)_2$
The double precision machine representation of d *is:*

$$[1100\ 0000\ 0100\ 1010\ 0001\ \ 1110\ 0000\ ...\ 00\,00]_2 = [C04A1E0000000000]_{16}$$

Example 1.15. *Determine the binary number* x *in* $\mathbb{F}_s$ *that corresponds to* $[45DE4000]_{16}$*, then find its decimal representation.*
The 32 bits string representation of x *is:*

$$[01000101110111100100000000000000]_2$$

The biased exponent is $c = (10\ 001\ 011)_2 = (213)_8 = (139)_{10}$*, so* $e = 139 - 127 = 12$ *Therefore:*

$$(x)_2 = +(1.101\ 111\ 001)_2 \times 2^{12}$$

1.4.3 De-normalized Numbers in MATLAB

Default formats for numbers in MATLAB is IEEE double precision. One can easily check out the de-normalized numbers in the system, as indicated through the following set of commands.

```
realmin %2^(-1022)
ans =
  2.2251e-308
>> 0.5*2^(-1022)
ans =
  1.1125e-308
>> 0.25*2^(-1022)
ans =
  5.5627e-309
>> 0.125*2^(-1022)
ans =
  2.7813e-309
```

1.4.4 Rounding errors in floating point representation of numbers

Consider a general floating point system $\mathbb{F} = F(\beta, p, e_{\min}, e_{\max})$, with $\beta \geq 2$. For all $x \in \mathbb{R}$ with $x_{min} < |x| < x_{max}$, and $x \notin \mathbb{F}$, we seek for a procedure leading to the representation of x in $\mathbb{F}$. The process of replacing x by its nearest representative element in $\mathbb{F}$ is called **correctly rounding**, and the error involved in this approximation is called **roundoff error**. We want to estimate how large it can be.
For such x, there exist x_1 and $x_2 = succ(x_1)$, with $x_1, x_2 \in \mathbb{F}$, such that $x_1 < x < x_2$.

Definition 1.4. *The floating point representation of x in $\mathbb{F}$ is an application fl: $\mathbb{R} \to \mathbb{F}$, such that $fl(x) = x_1$ or $fl(x) = x_2$ following one of the rounding procedures defined below.*

1. **Rounding by Chopping**:
   $fl_0(x) = x_1$, if $x > 0$, (and $fl_0(x) = x_2$, if $x < 0$)
   (i.e. $fl_0(x)$ is obtained by simply dropping the excess of digits in x)
2. **Rounding to the closest**:
   (a) $fl_p(x) = x_1$ if $|x - x_1| < |x - x_2|$
   (b) $fl_p(x) = x_2$ if $|x - x_2| \leq |x - x_1|$

Remark 1.5. *Let $x = (1.b_1..b_{23}b_{24}b_{25}...)_2$ .Rounding x in $\mathbb{F}_s$ to the closest stands as follows:*

- *If $b_{24} = 0$, then $fl_p(x) = x_1$.*
- *If $b_{24} = 1$ then $fl_p(x) = x_2$.*

Proof. To obtain this result, based on the definition above, simply note that if

$$x_1 = (1 \cdot b_1 b_2 ... b_{23}) 2^e, \text{and } x_2 = succ(x_1) = x_1 + (2^{-23}) 2^e$$

then the midpoint of the line segment $[x_1, x_2]$ is

$$x_M = \frac{x_1 + x_2}{2} = x_1 + (2^{-24}) 2^e = 1 \cdot b_1 ... b_{23} 1 \,;\, (x_M \notin \mathbb{F})$$

■

Consequently, since in the general case $x_M = (x + \frac{\beta^{-p+1}}{2} \times \beta^e)$ is the midpoint of the line segment $[x, succ(x)]$, one easily verifies the following result

graphically:

Theorem 1.3. *Let $x \in \mathbb{R}$ and $x \notin \mathbb{F} = F(\beta, p, e_{min}, e_{max})$, with $x_{min} < |x| < x_{max}$. Then:*

$$fl_p(x) = fl_0(x + \frac{\beta^{-p+1}}{2} \times \beta^e)$$

Example 1.16. *Let $x = (13.14)_{10}$. Find the internal representation of x using* **IEEE** *single precision notation, (rounding to the closest if needed). Find then the hexadecimal representation of x.*

As a first step we convert x to a binary number:

$$x = (1101.00100011110101110000101000111 1...)_2$$

We next normalize the number obtained:

$$x = (1.10100100011110101110000101000111 1...)_2 \times 2^3$$

Hence, the 2 successive numbers x_1 and x_2 of $\mathbb{F}_s$ are:

$$x_1 = (1.10100100011110101110000)_2 \times 2^3$$

$$x_2 = (1.10100100011110101110001)_2 \times 2^3$$

Obviously, rounding x to the closest gives $fl_p(x) = x_2$.
Note also that $e = 3$ and $c = (130)_{10} = (10000010)_2$.
Hence, the machine number in $\mathbb{F}_s$ is as follows:

4 bytes = 32 bits

t	c	f
0	10000010	10100100011110101110001

or also equivalently:

01000001010100100011110101110001

with hexadecimal representation:

$$[4\ 1\ 5\ 2\ 3\ D\ 7\ 1]_{16}$$

Remark 1.6.

Note that to round $x < 0$, we could apply the above procedures to $|x|$ first, then multiply the result obtained by -1. ■

We turn now to the error that can occur when we attempt to represent a given real number x in $\mathbb{F}$. As for relative error estimates we have the following.

Proposition 1.1. *Let $x \in \mathbb{R}$ with $x \notin \mathbb{F} = F(\beta, p, e_{\min}, e_{\max})$ and $x_{\min} < |x| < x_{\max}$. Then, the representations of x in $\mathbb{F}$ verify the following relative error estimates:*

1. $\frac{|x - fl_0(x)|}{|x|} < \epsilon_M$,
2. $\frac{|x - fl_p(x)|}{|x|} \leq \frac{1}{2}\epsilon_M$,

where $\epsilon_M = \beta^{-p+1}$ is the epsilon machine of the system.

Proof. Without loss of generality, we shall prove the above properties for positive numbers. Let x_1 and x_2 be in $\mathbb{F}(\ \beta, p, e_{\min}, e_{\max})$, such that

$$x_1 < x < x_2 = succ(x_1).$$

Then,

$$|x - fl_0(x)| < (x_2 - x_1) \text{ and } |x - fl_p(x)| \leq \frac{(x_2 - x_1)}{2}.$$

Furthermore, given that $x_1 < x$, the estimates of the proposition are obviously verified since in both cases $\frac{x_2 - x_1}{x_1} \leq \epsilon_M$. ■

Remark 1.7. *Note that Proposition 1.1 can be summarized by the following estimate:*

$$\frac{|x - fl(x)|}{|x|} \leq u \text{ where } u = \begin{cases} \epsilon_M, \text{ if } fl = fl_0 \\ \epsilon_M/2, \text{ if } fl = fl_P \end{cases}$$

This inequality can also be expressed in the more useful form:

$$fl(x) = x(1 + \delta) \text{ where } |\delta| \leq u \tag{1.6}$$

To see that, simply let $\delta = \frac{fl(x) - x}{x}$. Obviously $|\delta| \leq u$, with $fl(x)$ yielding the required result.

■

Remark 1.8. *When computing a mathematical entity $E \in \mathbb{R}$ (for example, E=π, $\sqrt{2}$, $\ln 2$,..)* **up to r decimal figures**, *one seeks an approximation $\hat{E}$ to E such that $\hat{E} \in \mathbb{F}(\ 10, r, e_{\min}, e_{\max})$, a user floating-point system with a base of 10 and r significant digits. A rounding procedure to the closest would yield $\hat{E}$ satisfying the following error estimate:*

$$\frac{|E - \hat{E}|}{|E|} \leq \frac{1}{2} 10^{1-r}.$$

To illustrate, we give some examples.

Example 1.17. *1. Consider $E = \pi = 3.14159265358979... \in \mathbb{R}$. In seeking for the representative $\hat{E}$ of $\pi \in \mathbb{F} = F(10, 6, e_{\min}, e_{\max})$, we first look for 2 successive numbers x_1 and x_2 in $\mathbb{F}$ such that*

$$x_1 \leq E \leq x_2.$$

Obviously $x_1 = 3.14159$ and $x_2 = 3.14160$. Rounding to the closest would select $\hat{E} = 3.14159$, with

$$\frac{|E - \hat{E}|}{|E|} \leq \frac{1}{2} \frac{|x_2 - x_1|}{x_1} = 1.59155077526 \times 10^{-6} \leq \frac{1}{2} 10^{1-6} = 5 \times 10^{-6} = \frac{\epsilon_M}{2}$$

2. Similarly, $\hat{E} = 1.4142136$ approximates $E = \sqrt{2}$ up to 8 significant figures. Since

$$x_1 = 1.4142135 < \sqrt{2} = 1.414213562373095... < x_2 = 1.4142136$$

and

$$\frac{|x_2 - x_1|}{2x_1} = \frac{7.071067628}{2} \times 10^{-8} = 0.35 \times 10^{-7} < 0.5 \times 10^{1-8} = \frac{\epsilon_M}{2}.$$

1.5 Floating Point Operations

For a given arithmetic operation $\cdot = \{+, -, \times, \div\}$ in $\mathbb{R}$, we define respectively in $\mathbb{F}$ **the floating point operations**: $\odot = \{\oplus, \ominus, \otimes, \oslash\}$, i.e.

$$\odot : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$$

Each of these operations is called a **flop** and according to IEEE standards, is designed as follows.

Definition 1.5. *In the standards of floating point operations in IEEE convention:*

$$\forall x \text{ and } y \in \mathbb{F},\ x \odot y = fl(x \cdot y).$$

This definition together with (1.6) leads to the following estimate:

$$x \odot y = (x \cdot y)(1+\delta), \text{ with } |\delta| \leq \mathbf{u},$$

where $u = \epsilon_M$ or $u = \frac{\epsilon_M}{2}$, depending on the chosen rounding procedure. Practically, Definition 1.5 means that $x \odot y$ is computed according to the following steps:

- 1^{st}: correctly in $\mathbb{R}$ as $x \cdot y$
- 2^{nd}: normalizing in $\mathbb{F}$
- 3^{rd}: rounding in $\mathbb{F}$

Under this procedure, the relative error will not exceed u.

Remark 1.9. *: Let $x, y \in \mathbb{F} = F(\beta, p, e_{min}, e_{max})$.*

$$x \oplus y = fl(x+y) = (x+y)(1+\delta) = x(1+\delta) + y(1+\delta)$$

meaning that $x \oplus y$ is not precisely $(x+y)$, but is the sum of $x(1+\delta)$ and $y(1+\delta)$, or also that it is the exact sum of a slightly perturbed x and a slightly perturbed y.

Example 1.18. *If x, y, and z are numbers in $\mathbb{F}_s$, what upper bound can be given for the relative roundoff error in computing $z \otimes (x \oplus y)$, with $(fl = fl_p)$. In the computer, the innermost calculation of $(x+y)$ will be done first:*

$$fl(x+y) = (x+y)(1+\delta_1)\ , |\delta_1| \leq 2^{-24}$$

Therefore:

$$fl[z\ fl(x+y)] = z\ fl(x+y)(1+\delta_2)\ , |\delta_2| \leq 2^{-24}$$

Putting both equations together, we have:

$$fl[z\ fl(x+y)] = z(x+y)(1+\delta_1)(1+\delta_2) = z(x+y)(1+\delta_1+\delta_2+\delta_1\delta_2) = z(x+y)(1+\delta_1+\delta_2) = z(x+y)(1+\delta),$$

where $\delta = \delta_1 + \delta_2$.

In this calculation, we neglect $|\delta_1\delta_2| \leq 2^{-48}$. Moreover, $|\delta| = |\delta_1 + \delta_2| \leq |\delta_1| + |\delta_2| \leq 2^{-24} + 2^{-24} = 2^{-23}$ ■

Although rounding errors are usually small, their accumulation in long and complex computations may give rise to unexpected wrong results, as shown in the following example:

Example 1.19. [24], p.7 *Consider the following sequence of numbers:*

$$z_2 = 2,\ z_{n+1} = 2^{n-1/2}\sqrt{1 - \sqrt{1 - 4^{1-n}z_n^2}},\ n = 2, 3, ...$$

It can be proved that this sequence theoretically converges to $\pi = 3.141592653589793$. *But when* `MATLAB` *is used to compute* z_n, *the relative error between* π *and* z_n *decreases till nearly the* 16^{th} *iteration, then grows vastly because of roundoff errors". These results are partly summarized in the table below.*

i	2	.....	6	.....	16
z_i	2.000000000000000	.....	3.136548490545931	.....	3.141592654807589
$\lvert\frac{z_i-\pi}{\pi}\rvert$	$O(10^{-1})$	.....	$O(10^{-3})$	.....	$O(10^{-10})$

i	.....	28	.....	30
z_i	.....	3.162277660168379	.....	4.000000000000000
$\lvert\frac{z_i-\pi}{\pi}\rvert$	.....	$O(10^{-3})$	.....	$O(10^{-1})$

■

We look now for specific problems caused by rounding errors propagation.

1.5.1 Algebraic properties in floating point operations

Since $\mathbb{F}$ is a proper subset of $\mathbb{R}$, elementary algebraic operations on floating point numbers do not satisfy all the properties of analogous operations in $\mathbb{R}$.
To illustrate, let $x,\ y,\ z \in \mathbb{F}$. The floating point arithmetic operations verify the following properties. :

1. Floating point addition is commutative in $\mathbb{F}$

$$x \oplus y = fl(x + y) = fl(y + x) = y \oplus x,$$

2. Floating point multiplication is commutative in $\mathbb{F}$

$$x \otimes y = y \otimes x$$

3. Floating point addition is not associative in $\mathbb{F}$

$$(x \oplus y) \oplus z \not\equiv x \oplus (y \oplus z)$$

4. Floating point multiplication is not associative in $\mathbb{F}$

$$(x \otimes y) \otimes z \not\equiv x \otimes (y \otimes z)$$

5. Floating point multiplication is not distributive with respect floating point addition in $\mathbb{F}$

$$x \otimes (y \oplus z) \neq (x \otimes y) \oplus (x \otimes z),$$

Example 1.20. *Let* $x = 3.417 \times 10^0$, $y = 8.513 \times 10^0$, $z = 4.181 \times 10^0 \in \mathbb{F}(10, 4, -2, 2)$. *Verify that addition is not associative in* $\mathbb{F}$.

$x \oplus y = 1.193 \times 10^1$ and $(x \oplus y) \oplus z = 1.611 \times 10^1$,
while: $y \oplus z = 1.269 \times 10^1$ and $x \oplus (y \oplus z) = 1.610 \times 10^1$. ■
Particularly, associativity is violated whenever a situation of overflow occurs as in the following example.

Example 1.21. *Let* $a = 1 * 10^{308}$, $b = 1.01 * 10^{308}$ *and* $c = -1.001 * 10^{308}$ *be 3 floating point numbers in* F_D *expressed in their decimal form.*

$$a \oplus (b \oplus c) = 1 * 10^{308} \oplus 0.009 * 10^{308} = 1.009 * 10^{308}$$

while

$$(a \oplus b) \oplus c = \infty$$

since $(a \oplus b) = 2.01 * 10^{308} \equiv \infty > x_{max} \approx 1.798 * 10^{308}$ in F_D ■

1.5.2 The problem of absorption

Let x, y be two non-zero positive numbers $\in \mathbb{F}_s$, with

$$x = m_x \times 2^{e_x},\ y = m_y \times 2^{e_y},$$

Assume $y < x$, so that:

$$x + y = (m_x + m_y \times 2^{e_y - e_x}) \times 2^{e_x}.$$

Clearly, since $m_y < 2$, if also $e_y - e_x \leq -25$, then

$$x + y < (m_x + 2^{-24}) \times 2^{e_x} = (x + succ(x))/2.$$

Hence using $fl = fl_p$, one gets:

$$x \oplus y = fl_p(x + y) = x,$$

although $y \neq 0$. In such situation, we say that y is **absorbed** by x.

Definition 1.6. *(Absorption Phenomena) Let $x, y \in \mathbb{F}(\beta, p, e_{\min}, e_{\max})$, y is said to be absorbed by x, if $x \oplus y = x$.*

Example 1.22. *Consider the sum of n decreasing positive numbers $\{x_i | i = 1, ..n\}$, with $x_1 > x_2 > ... > x_i > x_{i+1} > ... > x_n$, and let $S_n = \sum_{i=1}^{n} x_i$. There are two obvious ways to program this finite series; by increasing or decreasing index. The corresponding algorithms are as follows:*

Algorithm 1.6. Harmonic series evaluation by increasing indices

```
% Input : x=[x(1),...,x(n)]
% Output : sum of all components of x by Increasing index
function S=sum1(x)
S=0 ;
n=length(x) ;
for i=1:n
 S=S+x(i)
end
```

which leads then for example for $n = 4$ to the floating point number

$$S_1 = (((x_1 \oplus x_2) \oplus x_3) \oplus x_4).$$

Algorithm 1.7. Harmonic series evaluation by decreasing indices

```
function S=sum2(x)
```

```
% Input x=[x(1),...,x(n)]
% Output : sum of all components of x by Decreasing index
S=0 ;
n=length(x) ;
for i=n:-1:1
 S=S+x(i)
end
```

which gives for $n = 4$, the floating point number

$$S_2 = (((x_4 \oplus x_3) \oplus x_2) \oplus x_1)$$

Obviously, $S_1 \neq S_2$ and S_2 is more accurate than S_1 that favors the absorption phenomena .

Example 1.23. *Consider the following sequence of numbers in* $\mathbb{F}(10, 4, -3, 3)$, $x_1 = 9.999 \times 10^0$, $x_2 = 9.999 \times 10^{-1}$, $x_3 = 9.999 \times 10^{-2}$ *and* $x_4 = 9.999 \times 10^{-3}$.

The exact value of $\sum_{i=1}^{4} x_i$ is $11.108899 = 1.1108899 \times 10^1$. Using rounding by chopping for example, the first algorithm would give 1.108×10^1 while the second provides 1.110×10^1! ■

1.5.3 The problem of Cancellation or Loss of precision

The problem of cancellation occurs when subtracting two positive floating-point numbers of almost equal amplitude. To start, consider the following example.

Example 1.24. *Let* x_1, $x_2 \in \mathbb{F}(10, 5, -3, 3)$. *To subtract* $x_2 = 8.5478 \times 10^3$ *from* $x_1 = 8.5489 \times 10^3$, *the operation is done in two steps:*

x_1	8.5489×10^3
x_2	8.5478×10^3
$x_1 - x_2 =$	0.0011×10^3
Normalized result	1.1000×10^0

Hence the result appears to belong to a new floating-point system $\mathbb{F}(10, 2, -3, 3)$ that is less precise ($p = 2$) than the original one ($p = 5$). We are experiencing the phenomenon of **Cancellation** that causes **loss of significant figures** in floating-point computation. This can be summarized by the following proposition.

Proposition 1.2. *Let $x, y \in \mathbb{F} = F(\beta, p, e_{\min}, e_{\max})$. Assume x and y are two numbers of the same sign and the same order, ($|x|, |y| = O(\beta^e)$). Then there exists $k > 0$, such that $x - y$ is represented in a less precise floating point system $F(\beta, p - k, e_{\min}, e_{\max})$.*

Proof. Assume the two numbers x and y are expressed as follows.

$$x = (a_0 + a_1\beta^{-1} + ... + a_k\beta^{-k} + ... + a_{p-1}\beta^{-p+1}) \times \beta^e$$

and

$$y = (a'_0 + a'_1\beta^{-1} + ... + a'_k\beta^{-k} + ... + a'_{p-1}\beta^{-p+1}) \times \beta^e$$

with $a_i = a'_i$ for $i \leq k - 1 < p - 1$. It is obvious that:
$x - y = ((a_k - a'_k)\beta^{-k} + ... + (a_{p-1} - a'_{p-1})\beta^{-p+1}) \times \beta^e = (c_k\beta^{-k} + ... + c_{p-1}\beta^{-p+1}) \times \beta^e$
Hence: $x - y = (c_k + ... + c_{p-1}\beta^{-(p-k-1)}) \times \beta^{e-k}$, with $c_k \neq 0$
Consequently, $x - y$ is represented in a system which precision is $p - k$. ■

Example 1.25. Alternate series and the phenomenon of cancellation.

Consider the example of computing $\exp(-a)$, $a > 0$. For that purpose, we choose one of the following alternatives:

1. A straightforward application of the Taylor's series representation of $\exp(x)$, giving for $x = -a$, an alternating series:

$$\exp(-a) = 1 - a + \frac{a^2}{2!} - \frac{a^3}{3!} + \frac{a^4}{4!} + ... + (-1)^n\frac{a^n}{n!} + ..., \tag{1.7}$$

2. On the other hand, computing first $\exp(a)$ for $a > 0$, using the same series representation, which however has all its terms positive,

$$\exp(a) = 1 + a + \frac{a^2}{2!} + \frac{a^3}{3!} + \frac{a^4}{4!} + ... + \frac{a^n}{n!} + ..., \tag{1.8}$$

followed up by an inverse operation:

$$\exp(-a) = 1/\exp(a). \tag{1.9}$$

would yield more accurate results.

Computing with the first power series for large negative values of a, leads to drastic cancellation phenomena, while the second alternative provides accurate results as the following example indicates.

Example 1.26. *Consider the computation of* $\exp(-20)$ **which exact value is** $2.061153622438558 \times 10^{-9}$.

If the implementation is done in `MATLAB` which uses double precision IEEE formats, using successively the following algorithms:

Algorithm 1.8. Implementing e^x: alternative 1

```
function y=myexp(x)
tol=0.5*10^(-16);
y=1;
k=1;
T=x;
while abs(T)/y>tol;
    y=y+T;k=k+1;T=T*x/k;
end
```

A second alternative would be to compute $e^{|x|}$, then use for $x < 0$: $e^x = 1/e^{|x|}$.

Algorithm 1.9. Implementing e^x: alternative 2

```
function y=myexp(x)
tol=0.5*10^(-16);
y=1;
k=1;
v=abs(x);
T=v;
while abs(T)/y>tol;
    y=y+T;k=k+1;T=T*v/k;
end
if x<0
     y=1/y;
end
```

The results came as follows.

First alternative (1.7)	Value
	-19

Second alternative (1.9)	Value
	$2.061153622438558 \times 10^{-9}$

Another example deals with the computation of the roots of a quadratic equation.

Example 1.27. *Consider the computation of the roots of* $x^2 + 2bx + c = 0$, *where* c *is a positive number "much smaller" than* b^2.

There are 2 ways for handling the numerical computation of the solutions to this obvious problem.

1. A straightforward application of the well-known formulae:

   $$x_1 = -b - \sqrt{b^2 - c} \approx -2b; \quad x_2 = -b + \sqrt{b^2 - c} \approx 0. \tag{1.10}$$

   There is obviously in this way, loss of significant figures when computing x_2

2. However, computing first x_1 then using

   $$x_2 = \frac{c}{x_1} \tag{1.11}$$

   would not result in loss of digits.

1.6 Computing in a floating point system

Clearly in normalized floating point systems $\mathbb{F} = F(\beta, p, e_{min}, e_{max})$, no irrational nor rational numbers that do not fit the finite format imposed by the computer can be represented, neither too large nor too small real numbers are. Thus the effective number system for a computer is not a continuum, but rather a non uniformly distributed finite subset of the rational numbers, i.e a "strange" set of rational numbers with irregular gaps. The total number of elements in $\mathbb{F}$ is easily computed and is given by:

$$card(\mathbb{F}) = 2(\beta - 1)(\beta)^{p-1}(e_{max} - e_{min} + 1) + 2 \tag{1.12}$$

Note that this count excludes the de-normalized numbers, but includes ± 0. In what follows, we analyze particularly some cardinality and distribution properties of floating point systems $\mathbb{F}$, where the exponents are such that $e_{\max} = |e_{\min}| + 1$, as for example the cases of the IEEE single and double precision systems F_s and F_d.

1.6.1 Cardinality and distribution of special Floating- point System

Let $\mathbb{F} = \mathbb{F}(2, p, E_{\min}, E_{\max})$, with $E_{\max} = |E_{\min}| + 1$, and $E_{\min} < 0$. Note that

$$card(\mathbb{F}) = 2 * card(\mathbb{F}_+) + 2,$$

where $\mathbb{F}_+$ is the set of all non zero positive elements of $\mathbb{F}$. Based on (1.12), it can be easily shown that:

$$card(\mathbb{F}_+) = 2^{p-1}(E_{\max} + |E_{\min}| + 1).$$

Hence:

$$N_F = card(\mathbb{F}) = 2^p(E_{\max} + |E_{\min}| + 1) + 2$$

Since also $E_{\max} = |E_{\min}| + 1$, then:

$$N_F = 2^p(2E_{\max}) + 2 = 2^{p+1}(E_{\max}) + 2.$$

On the other hand, if we consider now $\mathbb{F}_0$, the subset of non zero elements of $\mathbb{F}$ defined as follows:

$$\mathbb{F}_0 = \{x \in \mathbb{F} | x = \pm 1.f \times 2^e,\ E_{\min} \leq e \leq 0\}$$

one finds that:

$$N_{F_0} = card(\mathbb{F}_0) = 2^p(E_{\max})$$

since in that case the number of different values taken by the exponent in $\mathbb{F}$ is

$$|E_{min}| + 1 = E_{max}$$

Note now that N_{F_0} represents half of the total of the non zero elements of $\mathbb{F}$, since:

$$\frac{N_{F_0}}{N_F - 2} = \frac{2^p(E_{\max})}{2^{p+1}(E_{\max})} = \frac{1}{2}. \tag{1.13}$$

This leads to the following proposition:

Proposition 1.3. *In a Floating point system* $\mathbb{F}(2, p, E_{\min}, E_{\max})$*, with* $E_{\max} = |E_{\min}| + 1$*, half of the non zero floating-point numbers are located in the interval* $(-2, 2)$ *with the other half located in* $[-x_{\max}, -2] \cup [2, x_{\max}]$.
Proof. This follows from formula (1.13). ■

It is also worth noting that all floating point numbers $\pm 1.f \times 2^e$ become

integers for $e \geq p-1$. These facts are visualized in the simulation that follows in next section.

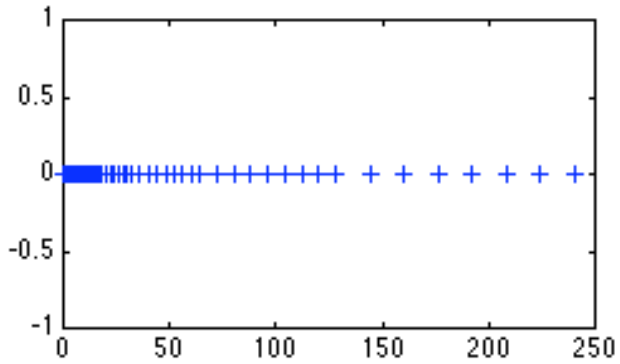
1.6.2 A MATLAB simulation of a floating-point system

The following function generates the non-negative numbers of a floating-point system $\mathbb{F}(b, p, emin, emax)$.

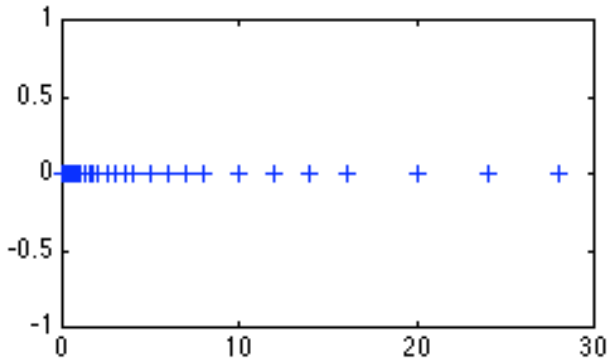
Algorithm 1.10. Simulation of a floating-point system

```
function x=float_v(b,p,emin,emax)
x=[];
epsm=b^(-p+1);
M=1:epsm:b-epsm;
E=1;
for e=0:emax
x=[x M*E];
    E=b*E;
end
E=1/b;
for e=-1:-1:emin
x=[M*E x];
    E=E/b;
end
x=[0 x];
```

As a result, we plot the distribution of non-negative numbers of $\mathbb{F}(2, 4, -6, 7)$,



and those of $\mathbb{F}\ (2, 3, -3, 4)$



1.6.3 Tips for floating point computation

To conclude, we may set an ensemble of rules that could avoid situations where accuracy can be jeopardized by the propagation of rounding errors through all type of floating-point operations and more particularly through absorption and cancellation. Finite precision arithmetic requests when programming some safeguarding habits. For example:

1. Scale the problem if possible to have its parameters on intervals with high "density" of floating-point numbers.
2. Seek always algorithms that would solve numerically a problem with the least number of flops.
3. Use Taylor's series expansions.
4. Avoid using alternating series.
5. Sum up positive elements of a series by adding from the smallest to the largest.
6. Rationalize expressions.
7. Use trigonometric identities
   .....

1.7 Exercises

1. Find the binary representation of the following numbers. Check the results by reconverting them to decimals.

   (a) $e \approx (2.718)_{10}$

   (b) $\frac{7}{8}$

   (c) $(792)_{10}$

2. Convert the following decimal numbers to octal numbers.

   (a) 37.1

   (b) 12.34

   (c) 3.14

   (d) 23.38

   (e) 75.231

   (f) 57.231

3. Convert the following binaries to octal and then to decimal numbers.

   (a) $(110\ 111\ 001.101\ 011\ 101)_2$

   (b) $(1\ 001\ 100\ 101.011\ 01)_2$

4. Convert the following numbers as required.

   (a) $(100\ 101\ 101)_2 = (\quad)_8 = (\quad)_{10}$

   (b) $(0.782)_{10} = (\quad)_8 = (\quad)_2$

   (c) $(47)_{10} = (\quad)_8 = (\quad)_2$

   (d) $(0.47)_{10} = (\quad)_8 = (\quad)_2$

   (e) $(51)_{10} = (\quad)_8 = (\quad)_2$

   (f) $(0.694)_{10} = (\quad)_8 = (\quad)_2$

   (g) $(110\ 011.111\ 010\ 110\ 110\ 1)_2 = (\quad)_8 = (\quad)_{10}$

   (h) $(351.4)_8 = (\quad)_2 = (\quad)_{10}$

   (i) $(45753.127664)_8 = (\quad)_2 = (\quad)_{10}$

5. Convert $x = (0.6)_{10}$ first to octal and then to binary. Check your result by converting directly to binary.

6. Prove that the decimal number $\frac{1}{5}$ cannot be represented by a finite expansion in the binary system.

7. Prove that a real number has a finite representation in the binary number system if and only if it is of the form $\pm m/2^n$, where n and m are positive integers.

8. Prove that any number that has a finite representation in the binary system must have a finite representation in the decimal system.

9. Display the positive elements of the floating point system $\mathbb{F} = F(2, 3, -2, +3)$. Determine the cardinality of $\mathbb{F}$.

10. Determine the IEEE single precision representation of the decimal number 64.015625.

11. Determine the IEEE single and double precision representations of the following decimal numbers:

    (a) 0.5, -0.5
    (b) 0.125, -0.125
    (c) 0.0625, -0.0625
    (d) 0.03125, -0.03125
    (e) 1.0, -1.0
    (f) +0.0, -0.0
    (g) -9876.54321
    (h) 0.236375
    (i) 294.78125
    (j) 54.37109375
    (k) -285.75
    (l) 10^{-2}

12. Which of these are machine numbers on the IEEE single precision system?

    (a) 10^{403}
    (b) $1 + 2^{-32}$
    (c) 1/5
    (d) 1/10

(e) $1/256$

(f) $2^4 + 2^{27}$

13. Identify the floating-point numbers corresponding to the following bit strings in the IEEE single precision system:

(a) 0 00000000 00000000000000000000000

(b) 1 00000000 00000000000000000000000

(c) 0 11111111 11111111111111111111111

(d) 1 11111111 11111111111111111111111

(e) 0 00000001 00000000000000000000000

(f) 0 10000001 01100000000000000000000

(g) 0 01111111 00000000000000000000000

(h) 0 01111011 10011001100110011001101

14. In the IEEE single precision system, what are the bit-string representation for the following sub-normal numbers?

(a) $2^{-127} + 2^{-129}$

(b) $2^{-127} + 2^{-145}$

(c) $2^{-127} + 2^{-130}$

(d) $\sum_{k=127}^{149} 2^{-k}$

15. Determine the decimal numbers that have the following IEEE single precision system representations:

(a) $[3F27E520]_{16}$

(b) $[CA3F2900]_{16}$

(c) $[C705A700]_{16}$

(d) $[494F96A0]_{16}$

(e) $[4B187ABC]_{16}$

(f) $[45223000]_{16}$

(g) $[45607000]_{16}$

(h) $[C553E100]_{16}$

(i) $[437F0001]_{16}$

(j) $[1A1A1A1A]_{16}$

16. Convert the greatest positive element in Single precision to an octal number "o" and write it in normalized floating point notation. Convert then the resulting "o" to a decimal number "d" and write it in normalized floating point notation.

17. Find a method for computing $f(x) = \sqrt{x+4} - 2$ accurately when x is small?

18. What is a good way to compute values of the function $f(x) = e^x - e$ if full machine precision is needed? Note: There is difficulty when $x = 1$.

19. What problem could the following assignment statement cause?

$$y \leftarrow 1 - \sin x$$

Circumvent it without resorting to a Taylor series if possible.

20. Find a method for computing

$$y \leftarrow \frac{1}{x}(\sinh x - \tanh x)$$

that avoids loss of significance when x is small. Find appropriate identities to solve this problem without using Taylor series.

21. For some values of x, the assignment statement $y \leftarrow 1 - \cos x$ involves a difficulty. What is it, what values of x are involved, and what remedy do you propose?

22. For some values of x, the function $f(x) = \sqrt{x^2+1} - x$ cannot be accurately computed by using this formula. Explain and find a way around the difficulty.

23. The inverse hyperbolic sine is given by $f(x) = \ln(x + \sqrt{x^2+1})$. Show how to avoid loss of significance in computing $f(x)$ when x is negative. Hint: Find and exploit the relationship between $f(x)$ and $f(-x)$.

24. Criticize and recode the assignment statement $z \leftarrow \sqrt{x^4+4} - 2$ assuming that z will sometimes be needed for an x close to zero.

25. How can values of the function $f(x) = \sqrt{x+2} - \sqrt{x}$ be computed accurately when x is large?

26. Find a way to calculate $f(x) = (\cos x - e^{-x})/\sin x$ correctly. Determine $f(0.008)$ correctly to ten decimal places (rounded).

27. Let
$$f(x) = \frac{e^x - e^{-x}}{x}$$
    (a) Find $\lim_{x\to 0} f(x)$
    (b) Use 3-digit rounding to the closest arithmetic to evaluate $f(0.1)$.
    (c) Replace each exponential function with its 3rd MacLaurin polynomial and repeat part (b).
    (d) The actual value is $f(0.1) = 2.003335000$. Find the relative error for the values obtained in parts (b) and (c).

28. Write a function procedure that returns accurate values of the hyperbolic tangent function
$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
    for all values of x. Notice the difficulty when $|x| < \frac{1}{2}$.

29. Find ways to compute these functions without serious loss of significant figures:
    (a) $e^x - \sin x - \cos x$
    (b) $\ln(x) - 1$
    (c) $\log x - \log(1/x)$
    (d) $x^{-2}(\sin x - e^x + 1)$
    (e) $arctanhx - x$

30. Suppose that two points (x_0, y_0) and (x_1, y_1) are on a straight line (L), with $y_1 \neq y_0$. To find the x-intercept of (L), two formulas are available:
$$x = \frac{x_0 y_1 - x_1 y_0}{y_1 - y_0} \text{ and } x = x_0 - \frac{(x_1 - x_0)y_0}{y_1 - y_0}$$
    Let $(x_0, y_0) = (1.31, 3.24)$ and $(x_1, y_1) = (1.93, 4.76)$. Use 4-digit rounding to the closest arithmetic to compute the x-intercept both ways. Which method gives more accuracy ? Justify.

31. The Taylor polynomial of degree n for $f(x) = e^x$ is $\sum_{i=0}^{n} \frac{x^i}{i!}$. Use the Taylor polynomial of degree 4 and three-digit chopping arithmetic to find an approximation to e^{-5} by each of the following methods:

(a) $e^{-5} = \sum_{i=0}^{8} \frac{(-1)^i 5^i}{i!}$

(b) $e^{-5} = \frac{1}{e^5} = \frac{1}{\sum_{i=0}^{8} \frac{5^i}{i!}}$

(c) An approximate value of e^{-5} correct to three digits is 6.74×10^{-3}. Which formula (a) or (b) gives the most accuracy ? Justify your answer.

32. Let

$$f(x) = 1.01e^{4x} - 4.62e^{3x} - 3.11e^{2x} + 12.2e^{x} - 1.99$$

(a) Use rounding to the closest with a precision $p = 3$ to evaluate $f(1.53)$ considering that $e^{1.53} = 4.62$

(b) Redo the calculations in part (a) using the Polynomial Nesting technique described in section 1.3.1.

(c) Compute the absolute and relative errors in parts (b) and (c) if the true 3-digit result $f(1.53) = -7.61$.

1.8 Computer Projects

Exercise 1 : Conversion: Decimal - Binary

1. Write a MATLAB function:

   function [Ibase2, Fbase2, b] = Convert10to2(d, k)

   which takes as input a non zero decimal number d and a positive integer k and converts d to a binary number b up to k fractional digits. Your function should output the 2 vectors Ibase2 and Fbase2 that represent respectively the integral and fractional parts of b, and the binary number b displayed with its sign and its integral and fractional parts.

2. Write a MATLAB function:

   function [Ibase10, Fbase10, d] = Convert2to10(Ibase2, Fbase2)

   which takes as input two vectors Ibase2 and Fbase2 that represent respectively the integral and fractional parts of a binary number, converts them to base 10 and outputs the results as 2 numbers Ibase10 and Fbase10 that are respectively the integral and fractional parts of the corresponding decimal number d and the decimal number d displayed with its sign and its integral and fractional parts.
   Hint: Use Nested Polynomial Evaluation.

3. Write a MATLAB function:

   function [B, I] = ConvertFraction10to2Pattern(D,m)

   which takes as input a decimal integer D consisting of k digits and the integer $m = 10^k$.
   This function converts the decimal fractional $f = \frac{D}{m}$ into a binary fractional represented by the vector B, and identifies the repeating pattern in B (if there is any), starting at component I and ending at n=length(B). In case the converted fractional part is finite, so no repeating pattern occurs, the value of I should be zero.
   For example:

   (a) To convert $f = 0.1$: input $D = 1$ and $m = 10$. This function outputs $B = [00011]$ and $I = 2$, since $(0.1)_{10} = (0.0\ 0011\ 0011\ 0011\ .....)_2$

(b) To convert $f = 0.25$: input $D = 25$ and $m = 100$. This function outputs $B = 01$ and $I = 0$, since $(0.25)_{10} = (0.01)_2$.

REMARK To minimize rounding errors in case I is a "large" number, it is more efficient to express fractional numbers as a ratio of 2 integers (for example f=D/m ...).

4. Test each one of the 3 functions above for 3 different test cases and save the results in a word document.

Exercise 2 : Conversion from Double to Single precision

1. Write a MATLAB function:

function [t e f] = GetVectorD(v)

which takes as input a binary vector v of 64 bits or components representing a machine number in IEEE - double precision, and extracts the values of the sign (t), the exponent (e) and the fractional part of the mantissa (f).

2. Write a MATLAB function:

function x = ConvertDoubletoSingle(v)

which takes as input a binary vector v of 64 bits representing a machine number in the IEEE double precision system. Your function should convert v to a single precision machine number and should output the result as a vector x of 32 bits, unless x represents a "denormalised number" or "Not a Number". In these 2 cases, your function should only display a message: ' x represents NaN ' or ' x represents a denormalised number ' .
At the end, if x represents an element of $F_S(2, 24, -126, +127)$, your function should also display the corresponding number in normalised floating point form, i.e. $xs = \pm 1.f \times 2^e$ or $xs = \pm 0$. Note the following remarks:

(a) Use rounding by chopping when needed.(fl_0).

(b) The smallest single precision denormalised number is: 2^{-149}

(c) For any exponent $e < -149$, the corresponding number in single precision is rounded to zero .

3. In Exercise 2, test function 1 for 3 different test cases , then function 2 for 5 different test cases including: "NaN', denormalised numbers, ± 0 and $\pm\infty$. Save the results in a word document.

Call for previous functions when needed.

Exercise 3 : Conversion: Decimal - Octal - Binary

1. Write a MATLAB function:

function [E8 , F8] = Convert2to8(E2, F2)

which takes as input two binary vectors E2 and F2 that are respectively the integral and fractional parts of a positive binary number b, converts them to octals and outputs the results as 2 vectors E8 and F8 that are respectively the integral and fractional parts of a positive octal number o.

2. Write a MATLAB function:

function [E10, F10, d] = Convert8to10(E8, F8)

which takes as input two octal vectors E8 and F8 that represent respectively the integral and fractional parts of a positive octal number o, converts them to base 10 and outputs the results as 2 decimal numbers, E10 and F10 that represent respectively the integral and fractional parts of the positive decimal number d using Nested Polynomial Evaluation. At the end, this function should also display d as a decimal number.

3. Test each one of the 2 functions above for 3 different test cases and save the results in a word document.(consider different lengths for all input vectors).

Exercise 4 : Successors and Rounding Procedures

Let $x = +mx \times 10^{ex}$ be a positive decimal number in $F(10, p, -20, +20)$, written in normalized floating point form, with $-20 \leq ex < +20$, and $p < 15$.

1. Write a MATLAB function :

function [my, ey] = GetSuccessor(mx, ex, p)

which takes as inputs:

- mx : the mantissa of x in standard normalised floating point notation
- ex : the exponent of x
- p : the precision of the floating point system to which x belongs

Let y be the successor of x in $F(10, p, -20, +20)$. This function should output:

- my : the mantissa of y displayed with a precision p (the non significant digits of the fractional part need not be displayed)
  HINT : first compute my, then use **num2str(my,p)** to output my in the required format
- ey : the exponent of y

2. Let $m = +m_1.m_2m_3...m_p$ be a positive decimal number whose integral part is m_1, and whose fractional part is $0.m_2m_3...m_p$.

   Write a MATLAB function :

   function [m] = ConvertVectortoDecimal(M)

   which takes as input a vector M of length p whose i^{th} component is the decimal digit m_i, for $i = 1, ..., p$, and whose output is the decimal number m represented by M.
   Use **" format long g"** to display m in double precision, discarding the non significant zeros of the fractional part .

3. Write a MATLAB function :

   function [mz, ez] = Round(Mx, ex, n, t)

   which takes as inputs:

   - Mx : a vector of length p whose components represent the mantissa mx of the decimal number $x \in F(10, p, -20, +20)$
   - ex : the exponent of x
   - n : a positive integer less then or equal to p $(n \leq p)$, representing the precision required to reach
   - t : a parameter taking the values 1 or 2

   This function should compute z: the representative of x in $F(10, n, -20, +20)$ by rounding x to the closest if $t = 1$ or by chopping if $t = 2$, and output

- mz : the mantissa of z displayed with a precision n.
  HINT : first compute mz, then use **num2str(my,n)** to output mz in the required format (the non significant zeros of the fractional part will be discarded)
- ez: the exponent of z

At the end your function should also **display** z in normalized floating point representation in $F(10, n, -20, +20)$.

4. Test each one of the 2 functions above for 3 different test cases and save the results in a word document.

Remark: Call for previous functions when needed.