Lecture 1

[3]:
```python
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
%matplotlib inline
```

1 Foundamentals of Numerical Analysis

1.1 Binary Numbers

Numbers are stored in computers in binary form:

$$\dots b_2 b_1 b_0 . b_{-1} b_{-2} \dots$$

where each digit or **bit** is 0 or 1. Usually, binary numbers are denoted with a subscript b or 2:

$$(\dots b_2 b_1 b_0 . b_{-1} b_{-2} \dots)_b, \quad (\dots b_2 b_1 b_0 . b_{-1} b_{-2} \dots)_2$$

so it is explicitly known that the number is in binary form.

Example 1.1 $(110100)_b$ *is in binary form, where*

$$b_0 = 0, b_1 = 0, b_2 = 1, b_3 = 0, b_4 = 1, b_5 = 1$$

and all the others are zero.

Example 1.2 $(0.01)_b$ *is in binary form, where*

$$b_{-1} = 0, b_{-2} = 1$$

and all the others are zero.

1.1.1 Binary to Decimal

To convert the binary number

$$(\dots b_2 b_1 b_0 . b_{-1} b_{-2} \dots)_b$$

to decimal number, the integer part is:

$$b_0 2^0 + b_1 2^1 + b_2 2^2 + \cdots b_i 2^i + \cdots$$

and the fractional part is:

$$b_{-1} 2^{-1} + b_{-2} 2^{-2} + b_{-3} 2^{-3} + \cdots b_{-i} 2^{-i} + \cdots$$

Example 1.3 *Convert the binary number*

$$(100)_b$$

to decimal.

Solution:

$$(100)_b = 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 4$$

Example 1.4 *Convert the binary number*

$$(0.11)_b$$

to decimal.

Solution:

$$(0.11)_b = 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = \frac{3}{4}$$

Example 1.5 *Convert the binary number*

$$(10101.1011)_b$$

to decimal.

Solution:

$$(10101.1011)_b = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 21\frac{11}{16}$$

1.1.2 Decimal to Binary

To convert a decimal number to binary: 1. The integer part of the decimal number is divided by 2 successively and the remainders are recorded in the reverse order. 2. The fractional part is treated by reversing the preceding steps, that is, multiply the fractional part successively by 2 and record the integer parts

Example 1.6 *Convert* 53.25 *to binary form.*

Solution*:*

For the integer part:

$$\begin{aligned}
53 \div 2 &= 26\,R\,1 \\
26 \div 2 &= 13\,R\,0 \\
13 \div 2 &= 6\,R\,1 \\
6 \div 2 &= 3\,R\,0 \\
3 \div 2 &= 1\,R\,1 \\
1 \div 2 &= 0\,R\,1
\end{aligned}$$

So the integer part is converted to $(110101)_b$ *(note it is NOT* $(101011)_b$*).*

For the fractional part:

$$.25 \times 2 = .5 + 0$$
$$.5 \times 2 = 0 + 1$$

So the fractional part is converted to $(.01)_b$ *(note it is NOT* $(.10)_b$*). So combining the integer and fractional parts, we have*

$$53.25 = (110101.01)_b$$

1.1.3 Additional Examples

Example 1.7 *Find the binary representation of the base 10 integers*

(a) 64 *(b)* 17 *(c)* 79

Example 1.8 *Find the binary representation of the base 10 numbers*

(a) 1/8 (b) 7/8 (c) 35/16

Example 1.9 *Convert the following base* 10 *numbers to binary*

(a) 11.25 *(b)* 3.5 *(c)* 30.75

1.2 Floating Point Representation of Real Number

1.2.1 Floating point formats

There are different models to store numbers in computers and perform algebraic operations. We introduce a representative one: the IEEE 754 Floating Point Standard, which is the common standard for computer arithmetics.

Floating Point Numbers A **floating point number** consists of three parts: the **sign** (+ or -), a **mantisa**, which contains the string of significant bits, and an **exponent**.

There are two common precisions: **single** and **double**, which have the following structures

precision	sign	exponent	mantissa
single	1	8	23
double	1	11	52

Any number can be (approximately) written as an IEEE **normalized** (meaning the leading bit is 1) floating point number is in the form of

$$\pm 1.bbb \cdots \times 2^p$$

where each of the Nb's is 0 or 1, and p is an M-bit binary number representing the exponent. So the structure above can be used to represent numbers in computers.

Example 1.10 *The decimal number* 9 *can be written as IEEE normalized floating point number:*

$$+1.001 \times 2^3$$

The single precision format for the decimal number 9 *is*

$$+1.00100000000000000000000 \times 2^3$$

and the double precision format is

$$+1.0010000000000000000000000000000000000000000000000000 \times 2^3$$

Machine Epsilon The number **machine epsilon**, denoted ϵ_{m}, is the distance between 1 and the smallest floating point number greater than 1. Note the double precision number 1 is

$$+1.0000000000000000000000000000000000000000000000000000 \times 2^0$$

The next floating point number greater than 1 is

$$+1.0000000000000000000000000000000000000000000000000001 \times 2^0$$

So $\epsilon_m = 2^{-52}$. This means not all numbers can be represented by floating point numbers. Numbers between two successive floating point numbers have to be rounded.

IEEE Rounding to Nearest Rule For double precision, if the 53rd bit to the right of the binary point is 0, then round down (truncate). If the 53rd bit is 1, then round up (add 1 to the 52 bit), unless all known bits to the right of the 1 are 0's, in which case 1 is added to bit 52 if and only if bit 52 is 1.

Example 1.11 *The decimal number* 9.4 *is equivalent to* $(1001.\overline{0110})_2$*, which can be left-adjusted as*

$$+1.\boxed{0010110011001100110011001100110011001100110011001100}110\cdots \times 2^3$$

So the 53*rd bit to the right of the binary point is* 1*. To represent decimal number* 9.4*, the floating point number is*

$$+1.\boxed{0010110011001100110011001100110011001100110011001101} \times 2^3$$

Notation $\mathrm{fl}(x)$

Denote the IEEE double precision floating point number associated to x using the Rounding to Nearest Rule by $\mathrm{fl}(\boldsymbol{x})$

So for the previous example:

$$\mathrm{fl}(9.4) = +1.\boxed{0010110011001100110011001100110011001100110011001101} \times 2^3$$

So $\mathrm{fl}(9.4) \neq 9.4$, although it is very close. We call $|\mathrm{fl}(x) - x|$ the **rounding error** (or **absolute error**). The **relative error** is defined as

$$\left| \frac{\mathrm{fl}(x) - x}{x} \right|$$

Relative rounding error

In the IEEE machine arithmetic model, the relative rounding error of $\mathrm{fl}(x)$ is no more than one-half machine epsilon:

$$\left| \frac{\mathrm{fl}(x) - x}{x} \right| \leq \frac{1}{2}\epsilon_{\mathrm{m}}$$

1.2.2 Machine representation

On a computer, a double precision floating point number is assigned an 8-byte storage space, or equivalently 64 bits, where the sign is stored in the first bit, followed by 11 bits representing the exponent and the 52 bits following the binary point, representing the mantissa. The sign bit is 0 for a positive number and 1 for a negative number. A general number is represented in the form of:

$$se_1e_2\dots e_{11}b_1b_2\dots b_{52}$$

The 11 bits of the exponent can represent $2^{11} = 2028$ integers, i.e., $0, 1, 2, \dots, 2027$. To represent both positive exponents and negative exponents, the integers are transformed to the range -1023 to 1024 by subtracting 1023, which is called the **exponent bias**. The numbers -1023 and 1024 have special meanings (to represent ∞, NaN, and non-normalized numbers), so the exponent range is actually from -1022 to 1023. For single precision, the exponent bias is 127.

For example, the number 1, or

$$1 = +1.0000000000000000000000000000000000000000000000000000 \times 2^0$$

has double precision machine number form

0	01111111111	0000000000000000000000000000000000000000000000000000

Example 1.12 *Find the machine number representation form of the real number* 9.4

Solution:

0	10000000010	0010110011001100110011001100110011001100110011001101

∞ is represented as

0	11111111111	0000000000000000000000000000000000000000000000000000

and $-\infty$ is

1	11111111111	0000000000000000000000000000000000000000000000000000

NaN is similar to ∞, and the only difference is that the mantisa bits are not all 0's.

Smallest representable number

When the exponent bits are all 0's, the machine number is interpreted as the non-normalized floating point number

$$\pm 0.\boxed{b_1b_2\dots b_{52}} \times 2^{-1022}$$

This non-normalized numbers are called **subnormal** floating point numbers.

The smallest (positive) number that can be represented is

0	00000000000	0000000000000000000000000000000000000000000000000001

which is

$$+0.0000000000000000000000000000000000000000000000000001 \times 2^{-1022} = 2^{-52} \times 2^{-1022} = 2^{-1074}$$

Note the smallest normalized number is

$$\boxed{0}\boxed{00000000001}\boxed{0000000000000000000000000000000000000000000000000000}$$

which is

$$+1.0000000000000000000000000000000000000000000000000000 \times 2^{-1022} = 2^{-1022}$$

Note the largest subnormal floating point number is slightly less than 2^{-1022}, so the subnormal floating point numbers has the ability to represent fractions smaller than the smallest normalized number, which can reduce the loss of precision when an *underflow* occurs.

1.2.3 Addition of floating point numbers

Machine addition consists of lining up the decimal points of the two numbers to be added, adding them, and then storing the result again as a floating point number.

Example 1.13 *Adding* 1 *to* 2^{-53} *would appear as follows:*

$$1.\boxed{00\ldots0} \times 2^0 + 1.\boxed{00\ldots0} \times 2^{-53} \tag{1}$$

$$= 1.\boxed{0000000000000000000000000000000000000000000000000000} \times 2^0 \tag{2}$$

$$+ 0.\boxed{0000000000000000000000000000000000000000000000000000}1 \times 2^0 \tag{3}$$

$$= 1.\boxed{0000000000000000000000000000000000000000000000000000}1 \times 2^0 \tag{4}$$

which is saved as $1.\boxed{0000000000000000000000000000000000000000000000000000} \times 2^0$ *due to the rounding rule. So* $1 + 2^{-53} = 1$ *in double precision IEEE arithmetic.*

Computer arithmetic can give surprising results sometimes, as shown in the following example:

Example 1.14 *Compute* $9.4 - 9 - 0.4$ *with floating point arithmetic.*

Solution: *First compute* $9.4 - 9$*:*

$$+ 1.\boxed{0010110011001100110011001100110011001100110011001101} \times 2^3 \tag{5}$$

$$- 1.\boxed{0010000000000000000000000000000000000000000000000000} \times 2^3 \tag{6}$$

$$= 0.\boxed{0000110011001100110011001100110011001100110011001101} \times 2^3 \tag{7}$$

$$= 1.\boxed{1001100110011001100110011001100110011001100110100000} \times 2^{-2} \tag{8}$$

Then compute $(9.4 - 9) - 0.4$*:*

$$+ 1.\boxed{1001100110011001100110011001100110011001100110100000} \times 2^{-2} \tag{9}$$

$$- 1.\boxed{1001100110011001100110011001100110011001100110011010} \times 2^{-2} \tag{10}$$

$$= 0.\boxed{0000000000000000000000000000000000000000000000000110} \times 2^{-2} \tag{11}$$

$$= 1.\boxed{1000000000000000000000000000000000000000000000000000} \times 2^{-52} \tag{12}$$

[20]:
```
# Check the 9.4-9-0.4=1.5*2^{-52}
print(9.4-9-0.4)
print(1.5*2**(-52))
```

```
3.3306690738754696e-16
3.3306690738754696e-16
```

1.2.4 Additional Problems

Example 1.15 *Convert the following base* 10 *numbers to binary and express each as a floating point number $fl(x)$ by using the Rounding to Nearest Rule:*

(a) $1/4$ *(b)* $1/3$ *(c)* $2/3$

Example 1.16 *Do the following sum by hand in IEEE double precision computer arithmetic, using the Rounding to Nearest Rule. Check your answer using Python*

(a) $(1 + (2^{-51} + 2^{-53})) - 1$

[1]:
```
print(1+(2**(-51)+2**(-53))-1)
print(2**(-51))
```

```
4.440892098500626e-16
4.440892098500626e-16
```

1.3 Loss Of Significance

Example 1.17 $123.4567 - 123.4566 = 000.0001$ *The subtraction problem began with two input numbers that we knew to seven-digit accuracy, and ended with a result that has onely one-digit accuracy. This is known as loss of significant numbers. This example is straightforward, but other examples can be more subtle.*

Example 1.18 *Calculate* $\sqrt{9.01} - 3$ *with three-digit arithmetic.*

Solution: *We use this example for illustrative purposes. Instead of using* 52*-bit mantissa, we use only three decimal digits.*

Since $\sqrt{9.01} \approx 3.0016662 \approx 3.00$, $\sqrt{9.01} - 3 = 0$ *with three-digit arithmetic.*

To solve the problem, we can use a different way to compute the result:

$$\sqrt{9.01} - 3 = \frac{(\sqrt{9.01} - 3)(\sqrt{9.01} + 3)}{\sqrt{9.01} + 3} = \frac{9.01 - 9}{\sqrt{9.01} + 3} = \frac{0.01}{3.00 + 3} = 0.00167 \approx 1.67 \times 10^{-3}$$

[15]:
```
# check the value of sqrt(9.01)-3
np.sqrt(9.01)-3
```

[15]: 0.0016662039607266976

The lesson is that it is important to find ways to avoid subtracting nearly equal numbers in calculations, if possible.

1.3.1 Additional Examples

Example 1.19 ***Example 19*** *Identify for which values of x there is subtraction of nearly equal numbers, and find an alternate form that avoids the problem.*

a. $\frac{1-\sec x}{\tan^2 x}$ *b.* $\frac{1-(1-x)^3}{x}$ *c.* $\frac{1}{1+x} - \frac{1}{1-x}$

1.4 Evaluating a Polynomial

To evaluate the polynomial

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1$$

how to minimize the number of additions and multiplications required to get $P(\frac{1}{2})$?

Method 1

The most straight forward approach is

$$P\left(\frac{1}{2}\right) = 2 * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} + 3 * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} - 3 * \frac{1}{2} * \frac{1}{2} + 5 * \frac{1}{2} - 1 = \frac{5}{4}$$

There are 10 multiplications and 4 additions.

Method 2

Find the poswers of the input number $x = 1/2$ first, and store thme for future use:

$$\frac{1}{2} * \frac{1}{2} = \left(\frac{1}{2}\right)^2 \tag{13}$$

$$\left(\frac{1}{2}\right)^2 * \frac{1}{2} = \left(\frac{1}{2}\right)^3 \tag{14}$$

$$\left(\frac{1}{2}\right)^3 * \frac{1}{2} = \left(\frac{1}{2}\right)^4 \tag{15}$$

Then adding up the terms:

$$P\left(\frac{1}{2}\right) = 2 * \left(\frac{1}{2}\right)^4 + 3 * \left(\frac{1}{2}\right)^3 - 3 * \left(\frac{1}{2}\right)^2 + 5 * \frac{1}{2} - 1 = \frac{5}{4}$$

There are 7 multiplications and 4 additions.

Method 3 (Nested Multiplication)

Rewrite the polynomial so that it can be evaluated from the inside out:

$$\begin{aligned} P(x) &= -1 + x(5 - 3x + 3x^2 + 2x^3) && (16)\\ &= -1 + x(5 + x(-3 + 3x + 2x^2)) && (17)\\ &= -1 + x(5 + x(-3 + x(3 + 2x))) && (18) \end{aligned}$$

This needs only 4 multiplications and 4 additions. The method is called **nested multiplication** or **Horner's method**. A general degree d polynomials can be evaluated in d multiplications and d additions.

The lesson learned from the example is it is important to design efficient algorithms to solve problems as fast as possible. Usually an efficient algorithm is not obvious.

Example 1.20 *Find an efficient method for evaluating the polynomial* $P(x) = 4x^5 + 7x^8 - 3x^{11} + 2x^{14}$

Solution:

$$P(x) = x^5(4 + 7x^3 - 3x^6 + 2x^9) = x^5(4 + x^3(7 + x^3(-3 + 2x^3)))$$

The following code evaluate a polynomial using nested multiplication

[17]:
```
# Program 1.1 Nested multiplication

def nestmul(d,c,x):
    """
    Evaluates polynomial from nested form using Horner's Method
    You need to add "import numpy as np" if you have not
    Input:
    d: degree of polynomial
    c: array of d+1 coefficients (constant term first)
    x: x-coordinate at which to evaluate
    """

    y = c[d]
    for i in range(d-1,-1,-1):
        y = y*x+c[i]

    return y
```

[18]:
```
d = 4
c = np.array([-1,5,-3,3,2])
x = 0.5
print(nestmul(d,c,x))
```

```
1.25
```

1.4.1 Additional Examples

Example 1.21 *Rewrite the following polynomials in nested form. Evaluate with and without nested form at* $x = 1/3$.

(a). $P(x) = 6x^4 + x^3 + 5x^2 + x + 1$

(b). $P(x) = -3x^4 + 4x^3 + 5x^2 - 5x + 1$

(c). $P(x) = 2x^4 + x^3 - x^2 + x + 1$