Computational Mathematics: Models, Methods and Analysis

Zhilin Li

Chapter 1

Introduction

- Why is this course important (motivations)? What is the role of this class in the problem solving process using mathematics and computers?
- Model problems and relations with course materials.
- Errors (definition and how to avoid them)

In Fig.1.1, we show a flow chart of a problem solving process. In this class, we will focus on numerical solutions using computers.

1.1 A computer number system

We want to use computers to solve mathematical problems and we should know the number system in a computer.

A primitive computer system is only part of a real number system. A number in a computer system is represented by

$$\begin{array}{lllll} & & \text{fraction} & \text{exponential} & \\ x_c = & \pm. & d_1 d_2 \cdots d_n & \beta^s, & 0 \le d_i \le \beta - 1, \quad -S_{max} \le s \le S_{max} \\ & \text{sign} & \text{mantissa} & \text{base} & \end{array} \tag{1.1.1}$$

Below are some examples of such numbers

$$-0.11112^5 \quad (\text{binary}, \beta = 2), \quad -0.314210^0, \quad -0.031410^0. \tag{1.1.2}$$

The choice of base number is

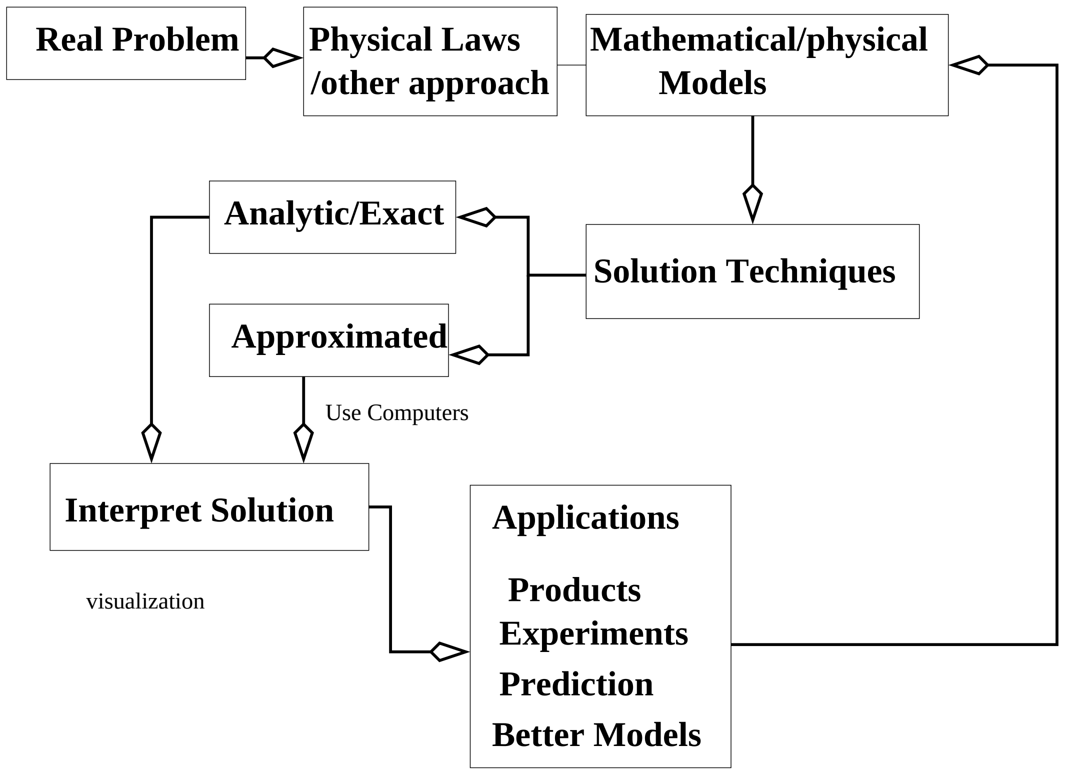

Figure 1.1: A flow chart of a problem solving process.

$\beta = 2$	binary	primitive
$\beta = 8$	octal	used for transition
$\beta = 10$	decimal	custom & convenient
$\beta = 16$	hexadecimal	save storage

The float number is denoted by $fl(x)$. We can see that the expression of a floating number is not unique. To get a unique expression, it is often designed that $d_1 \neq 0$ if x_c is a non-zero number. Such a floating number is called a *normalized* floating number. The number zero is expressed as $0.00\cdots 0\beta^0$. Note that one bite is used to represent the sign in the exponential.

Often there are two number systems in a programming language for a particular computer, single precision corresponding to 32 bits; and double precision for 64 bits.

In a 32 bites computer number system, we have

sign	exponential	fraction
1	8	23.

In a 64 bites computer number system, we have

sign	exponential	fraction
1	11	52.

1.1.1 Properties of a computer number system

- It is a subset of the real number system with finite number of floating numbers. For a 32-bit system, the total numbers is roughly $2\beta^n(S_{max} - S_{min} + 1) - 1$.
- Even if x and y are in the computer number system, their operations, for example, $fl(xy)$, can be out of the the computer number system.
- It has the maximum and minimum numbers; and maximum and non-zero minimum magnitude. For a 32-bit system, the largest and smallest numbers can be calculated from the following:

$$\text{the largest exponential:} = 2^0 + 2^1 + \cdots + 2^6 = 2^7 - 1 = 127;$$

$$\text{the largest fraction} = 0.1111\cdots 1 = 1 - 2^{-23};$$

$$\text{the largest positive number} = 2^{127}(1 - 2^{-23}) = 1.7014116 \times 10^{38}.$$

The smallest number is then $-1.7014116 \times 10^{38}$. The smallest positive number (or smallest magnitude) is

$$0.000\cdots 1 \times 2^{-127} = 2^{-127-23} = 7.0064923 \times 10^{-46}, \tag{1.1.3}$$

while the smallest normalized magnitude is

$$0.100\cdots 0 \times 2^{-127} = 2^{-128} = 2.9387359 \times 10^{-39}. \tag{1.1.4}$$

Overflow and underflow

If a computer system encounter a number whose magnitude is larger than the largest floating number of the computer system, it is called *OVERFLOW*. This often happens when a number is divided by zero, for example, we want to compute s/a but a is undefined, or evaluate a function outside of the definition, for example, $\log(-5)$. Computers often returns symbol such as NAN, inf, or simply stops the running process. This can also happen when a number is divided by a very small number. Often an overflow indicates a bug in the coding and should be avoided.

If a computer system encounter a number whose magnitude is smaller than the smallest positive floating number of the computer system, it is called *underflow*. Often, the computer system can set this number as zero and there is no harm to the running process.

- The numbers in a computer number system is not evenly spaces. It is more clustered around the origin and get sparser far away.

While a computer system is only a subset of the real number system, often it is good enough if we know how to use it. If a single precision system is not adequate, we can use the double precision system.

1.2 Round-off errors and floating point arithmetics

Since computer number system is only a subset of a real number system, errors (called round-off errors) are inevitable when we solver problems using computers. The question that we need to ask is how the errors affect the final results and how to minimize the negative impact of errors.

Input errors

When we input a number into a computer, it is likely to have some errors. For example, the number π can be represented exact in a computer number system. Thus a floating number of expression of π denoted as $fl(\pi)$ is different from π. The first question is, how a computer system approximate π. The default is the round-off approach. Let us take the cecimal system as an example. Let x be a real number that is in the range of the computer number system in terms of the magnitude, and we express it as a normalized floating number

$$x = 0.d_1d_2\cdots d_nd_{n+1}\cdots \times 10^b, \quad d_1 \neq 0. \tag{1.2.1}$$

The floating number if the computer system using the round-off approach is

$$fl(x) = \begin{cases} 0.d_1d_2\cdots d_n \times 10^b, & \text{if } d_{n+1} \leq 4, \\ 0.d_1d_2\cdots (d_n+1) \times 10^b, & \text{if } d_{n+1} \geq 5 \end{cases} \tag{1.2.2}$$

1.2.1 Definition of errors

The absolue error is defined as the difference between the true value and the approximated,

$$\text{absolute error} = \text{true - approximated}. \tag{1.2.3}$$

Thus the error for $fl(x)$ is

$$\text{absolution error of } fl(x) = x - fl(x). \tag{1.2.4}$$

The absolution error is often simply called the error.

Absolution error may not reflect the reality. One picks up 995 correct answers from 1000 problems certainly is better than the one that picks up 95 correct answers from 100 problems although both of the errors are 5. A more realistic error measurement is the relative error which is defined as

$$\text{relative error} = \frac{\text{absolution error}}{\text{true value}}, \tag{1.2.5}$$

for example, if $x \neq 0$, then

$$\text{relative error of } fl(x) = \frac{x - fl(x)}{x}. \tag{1.2.6}$$

Obviously, for different x, the error $x - fl(x)$ and the relative error are different. How do we then characterize the round-off errors? We seek the upper bounds, or the worst case, which should apply for all x's.

For the round-off approach, we have

$$\begin{aligned} |x - fl(x)| &= \begin{cases} 0.00\cdots 0d_{n+1}\cdots \times 10^b, & \text{if } d_{n+1} \leq 4, \\ 0.00\cdots 0(10-d_{n+1})\cdots \times 10^b, & \text{if } d_{n+1} \geq 5 \end{cases} \\ &\leq 0.00\cdots 05 \times 10^b = \frac{1}{2}10^{b-n}, \end{aligned}$$

which only depends on the magnitude of x. The relative error is

$$\frac{|x - fl(x)|}{|x|} \le \frac{\frac{1}{2}10^{b-n}}{|x|} \le \frac{\frac{1}{2}10^{b-n}}{0.1 \times 10^b} = \frac{1}{2}10^{-n+1} \overset{define}{==} \epsilon = \text{machine precision} \qquad (1.2.7)$$

Note that the upper bound of the relative error for the round-off approach is independent of x, only depends on the computer number system. This upper bound is called the machine precision, or machine epsilon, which indicates the best accuracy that we can expect using the computer number system.

In general we have

$$\frac{|x - fl(x)|}{|x|} \le \frac{1}{2}\beta^{-n+1} \qquad (1.2.8)$$

for any base β. For a single precision computer number system (32 bits) we have[1]

$$\epsilon = \frac{1}{2}2^{-23+1} = 2^{-23} = 1.192093 \times 10^{-7}. \qquad (1.2.9)$$

For a 64-bits number system (double precision), we have

$$\epsilon = \frac{1}{2}2^{-52+1} = 2^{-52} = 2.220446 \times 10^{-16}. \qquad (1.2.10)$$

Relative error is closely associate with the concept of the significant digits. In general, if a relative error is of order 10^{-5}, for example, it is likely the result has 5 significant digits.

An approximate number can be regarded as a perturbation of the true vales according to the following theorem.

Theorem 1.1 *If $x \in R$, then $fl(x) = x(1+\delta)$, $|\delta| \le \epsilon$ ia the relative error.*

There are other ways to input a number into a computer system. Two other approaches are *rounding*, and *chopping*, in which we have

$$fl(x) = 0.d_1 d_2 \cdots d_n \times 10^b \qquad (1.2.11)$$

for chopping and

$$fl(x) = 0.d_1 d_2 \cdots (d_n + 1) \times 10^b, \qquad (1.2.12)$$

for the rounding approach. The errors bounds are twice as much as the round-off approach.

1.2.2 Error analysis of computer arithmetics

The primitive computer arithmetic only include addition, subtraction, multiplication, division, and logical operations. Logic operations do not generate errors. But the basic arithmetic operations will introduce errors. The error bounds are given in the following theorem.

[1]Accoding to the definition, the $\epsilon \approx 1.2 \times 10^{-7}$ which is the same as Kahaner, Moler, and Nash's book, but it twice as larger as the result given in Demmel's book, which we think it is wrong.

Theorem 1.2 *If a and b are two floating numbers in a computer number system, $fl(a \circ b)$ is in the range of the computer number system, then*

$$fl(a \circ b) = (a \circ b)\,(1+\delta)\,, \quad \circ: \quad + \quad - \quad \times \quad \div \tag{1.2.13}$$

where

$$|\delta| = |\delta(a,b)| \le \epsilon, \tag{1.2.14}$$

we also have

$$fl(\sqrt{a}) = \sqrt{a}\,(1+\delta)\,. \tag{1.2.15}$$

Note that δ is the relative error if $(a \circ b) \neq 0$ of the operations and is bounded by the machine precision. This is because

$$fl(a \circ b) - (a \circ b) = \delta(a \circ b) \quad \text{absolution error}$$

$$\frac{fl(a \circ b) - (a \circ b)}{(a \circ b)} = \delta, \quad |\delta| \le \epsilon.$$

We conclude that the arithmetic operations within a computer number system give the 'best' results that we can possible get. Does it mean that we do not need to worry about round-off errors at all? Of course not!

1.2.3 Round-off error analysis and how to avoid round-off errors

Now we assume that x and y are two real numbers. When we input them into a computer, we will have errors. First we consider the multiplications and divisions

$$\begin{aligned} fl(x \circ y)) &= fl\left(fl(x) \circ fl(y)\right) = fl\left(x(1+\epsilon_x) \circ y(1+\epsilon_y)\right), \qquad |\epsilon_x| \le \epsilon, \quad |\epsilon_y| \le \epsilon, \\ &= \left(x(1+\epsilon_x) \circ y(1+\epsilon_y)\right)(1+\epsilon_{x \circ y}), \quad |\epsilon_{x \circ y}| \le \epsilon. \end{aligned}$$

Note that ϵ_x, ϵ_y, and $\epsilon_{x \circ y}$ are *different numbers* although they have the same upper bound! We distinguish several different cases

- Multiplication/division ($\circ = \times$ or $\div$), take the multiplication as an example, we have

  $$\begin{aligned} fl(xy)) &= x(1+\epsilon_x)y((1+\epsilon_y)(1+\epsilon_{xy}) = xy\left(1+\epsilon_x+\epsilon_y+\epsilon_{xy}+O(\epsilon^2)\right) \\ &= xy(1+\delta), \qquad |\delta| \le 3\epsilon. \end{aligned}$$

  Often we ignore the high order term (h.o.t) since they one much smaller (for single precision, we have 10^{-7} versus 10^{-14}). Thus delta is the *relative error* as we mentioned before. The error bound is understandable with the fact of 3 with two input errors and one from the multiplication. The absolute error is $-xy\delta$ which is bounded by $3|xy|\epsilon$. The same bounds hold for the division too if the divisor is not zero. Thus the errors from the multiplications/divisions are not big concerns here. But we should avoid dividing by small numbers if possible.

- Now we consideration a subtraction $\circ$ (an addition can be treated as a subtraction since $a+b = a-(-b)$ or vise versa.). Now we have

$$\begin{aligned} fl(x-y)) &= (x(1+\epsilon_x) - y((1+\epsilon_y))\,(1+\epsilon_{xy}) \\ &= x - y + x\epsilon_x - y\epsilon_y + (x-y)\epsilon_{xy} + O(\epsilon^2). \end{aligned}$$

The absolution error is

$$(x-y) - fl(x-y) = -x\epsilon_x + y\epsilon_y - (x-y)\epsilon_{xy}$$

$$|(x-y) - fl(x-y)| = |x|\epsilon + |y|\epsilon_y + |x-y|\epsilon$$

which does not seem to be too bad. But the relative error may be unbounded because

$$\begin{aligned} \frac{|(x-y) - fl(x-y)|}{|x-y|} &= \left| \frac{x\epsilon_x - y\epsilon_y - (x-y)\epsilon_{xy}}{x-y} \right| + O(\epsilon^2) \\ &\le \frac{|x\epsilon_x - y\epsilon_y|}{|x-y|} + \epsilon. \end{aligned}$$

In general, $\epsilon_x \neq \epsilon_x$ even though they are very small and have the same upper bound. Thus the relative error can be arbitrarily large if x and y are very close! That means the addition/subtraction can lead the ***loss of accuracy***, or significant digits. It is also called ***catastrophic cancellation*** as illustrate in the following example

$$0.31343639 - 0.31343637 = 0.00000002.$$

If the last two digits of the two numbers are wrong (likely in many circumstance), then there is no significant digit left in the result. In this example, the absolute error is till small, but the relative error is very large!

Round-off error analysis summary

- Use formulas $fl(x) = x(1+\epsilon_1)$, $fl(x \circ y) = (x \circ y)(1+\epsilon_2)$ etc.
- Expand and collect terms.
- Ignore high order terms.

1.2.4 An example of accuracy loss

Assume we want to solve a quadratic equation $ax^2 + bx + c = 0$ on a computer, how do we do it? First of all, we need to write down the algorithm mathematically before coding it. There are at least three methods.

- Algorithm 1: $x_1 = \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \dfrac{-b - \sqrt{b^2 - 4ac}}{2a},$

- Algorithm 2: $x_1 = \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \dfrac{c}{x_1},$
- Algorithm 3: $x_1 = \dfrac{-b - \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \dfrac{c}{x_1}.$

Mathematically, there are all equivalent (thus they are all call consistent). But occasionally, they may give very different results especially if c is very small. When we put select the algorithm to run on computer, we should choose Algorithm 2 if $b \le 0$ and Algorithm 3 if $b \ge 0$, why? This can be done using *if* $\cdots$ *then* conditional expression using any computer language.

Lets check with the simple case $a = 1$a $b = 2$, $x^2 + 2x + e = 0$. When e is very small, we have

$$x_1 = \frac{-2 - \sqrt{4 - 4e}}{2} = -1 - \sqrt{1 - e} \approx -2,$$
$$x_2 = \frac{-2 + \sqrt{4 - 4e}}{2} = -1 + \sqrt{1 - e} = \frac{e}{-1 - \sqrt{1 - e}} \approx -0.5e.$$

The last equality was obtained by rationalize to the denomenator.

Below is a Matlab code to illustrate four different algorithms:

```
function [y1,y2,y3,y4,y5,y6,y7,y8] = quad_err(e)

y1 = (-2 + sqrt(4 - 4*e))/2; y2 = (-2 -sqrt(4 - 4*e))/2;
y3 = (-2 + sqrt(4 - 4*e))/2; y4 = e/y3;
y5 = (-2 -sqrt(4 - 4*e))/2; y6 = e/y5;

y7 = -4*e/(-2 -sqrt(4 - 4*e))/2; y8 = e/y7;
```

From input various e, we can see how the accuracy gets lost. In general, when we have $e = 2 \times 10^{-k}$, we will lose about k significant digits.

1.2.5 How to avoid loss of accuracy?

- Use different formula, for example,

$$-b + \sqrt{b^2 - 4ac} = -\frac{4ac}{b + \sqrt{b^2 - 4ac}} \quad \text{if } b > 0.$$

A good complete root finding algorithm for quadratic equations would be

```
if b>0
  x1 = 2*c/(b + sqrt(b*b - 4*a*c) );
  x2 = (-b-sqrt(b*b - 4*a*c))/(2*a);
else
  x1 = (-b+sqrt(b*b - 4*a*c))/(2*a);
  x2 = -2*c/(b + sqrt(b*b - 4*a*c) );
end
```

- Use Taylor expansion, for examples,

$$1-\cos x = 1-\left(1-\frac{x^2}{2}+\frac{x^4}{4!}-\cdots\right) \approx \frac{x^2}{2}.$$

$$f(x+h)-f(x) = hf'(x)+h^2\,\frac{f''(x)}{2}+h^3\,\frac{f'''(x)}{3!}+\cdots$$

- Another rule of thumb for summations $fl(\sum_{i=1}^{n})x_i$. We should add those numbers with small magnitude first to avoid "large numbers eat small numbers".

- Add a small quantity to prevent break downs, for example, to compute $x/(x^2+y^2)$, it is better to use $x/(x^2+y^2+\epsilon)$, where ϵ is the machine precision.

1.3 Some basic algorithms and Matlab codes

- Sum $\sum_{i=1}^{n} x_i$:

```
s=0;      % initialize
for i=1:n
  s = s + a(i);     % A common mistake is forget the s here!
end
```

- Product $\prod_{i=1}^{n} x_i$:

```
s=1;      % initialize
for i=1:n
  s = s * a(i);     % A common mistake is forget the s here!
end
```

Example: Matrix-vector multiplication $y = Ax$

In Matlab, we can simply use $y = A * x$. Or we can use the component form so that we can easily convert the code to other computer languages. We can put the following into a Matlab .m file, say, test_Ax.m with the following contents:

```
n=100; A=rand(n,n); x=rand(n,1);   % Generate a set of data
for i=1:n;
  y(i) = 0;      % initialize
  for j=1:n
     y(i) = y(i) + A(i,j)*x(j);    % Use ';' to compress the outputs.
  end
end
```

We wish to develop efficient algorithms (fast, less storage, accurate, and easy to program). Note that Matlab is case sensitive and the index of arrays should be positive integers (can not be zero).

1.3.1 Hornet's algorithm for computing a polynomial

Consider a polynomial $p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$. How do we evaluate its value at a point x. We can store the coefficients first in an array, say, $a(1), a(2), \cdots a(n+1)$ since we can not use $a(0)$ in Matlab, then we can evaluate $p_(x)$ using

```
p = a(1);
for i=1:n
   p = p + a(i+1)*x^(i+1);
end
```

The total number of operations are about $O(n^2/2)$ multiplications, and n additions. However, from the following observations

$$\begin{aligned} p_3(x) &= a_3x^3 + a_2x^2 + a_1x + a_0 \\ &= x(a_3x^2 + a_2x + a_1) + a_0 \\ &= x(x(a_3x + a_2) + a_1) + a_0, \end{aligned}$$

we can form the Hornet's algorithm (pseudo-code)

```
  p = a(n)
  for i=n-1,-1,0
     p = x*p + a(i)
  endfor
```

which only requires n multiplications and additions!