COMPUTATIONAL PHYSICS

Representing numbers and their precision

Numbers representation: integer, reals

Computer storage precision

Round-off and discretization errors

Introduction

- In computational physics, is the numerical precision involved in the calculation made is very important.
- Inaccurate inputs can propagate severely to our final result on some algorithms.
- Frequently, some approximations to our physical problem are made or some simplification/truncation is assumed in the algorithms → one has to be aware of the implications.
- Computers have finite memory and standardized way of representing numbers or characters → one must not assume infinite precision while representing a number !
- Understanding numerical precision is key to understand when a “hard limit” has been reached → maybe rethink the algorithm…

Numerical representation: integer

- In computers information is stored as a sequence of 0's and 1's: ***binary system***

  - **Byte**: sequence of 8 bits
  - **KByte**: 2^{10} *Bytes* = 1024 Bytes
  - **MByte**: 2^{10}*KBytes* = 1024 KBytes

- A *m* bits integer number *N* in binary representation:

$$b_{m-1}\, 2^{m-1} + b_{m-2}\, 2^{m-2} + \cdots + b_0\, 2^0$$

- The sign of the number is is stored in one bit (usually the most significant bit - *MSB*)

  0 = positive
  1 = negative

- A ***32 bits*** signed integer (4 bytes) uses 31 bits (0...30) for storing the number max value of a signed 32bits integer: $2^{31} - 1 = \pm 2\,147\,483\,647$

How to convert to binary

→Repeatedly divide by 2 (saving integer part) and stash the remainder at each division…

Example: 125

125/2	= 62	(+1)	1×2^0
62/2	= 31	(+0)	
31/2	= 15	(+1)	
15/2	= 7	(+1)	
7/2	= 3	(+1)	
3/2	= 1	(+1)	
1/2	= 0	(+0)	0×2^6

$(125)_{10} = (000..111101)_2$

Numerical representation: reals

- Consider a 32-bit real. How to use the 32 bits ?

s	exponent		mantissa	
31	30	23	22	0

- Integral part and decimal part each converted into binary !

- Stored as a sequence of of 3 bit field: $(-1)^s \times m \times 2^e$

  s = sign(0,1)   m = mantissa (drop 1.)

  p = exponent (with p=e+bias=e+127)

- For the real *7.281* on has

$$(111.010001111...)_2 = 1.11010001111... \times \overset{2^2}{100}$$

s=0 $\underbrace{\textit{p=2+127}}_{10000001}$ *m=11010001111...(23bits)*

Convert decimal part to binary

→Repeatedly ***multiply*** by 2 (saving integer part) and stash the remainder at each multiplication.

Example: 0.281

0.281*2 = 0.562 (+0) 0×2^{-1}
0.562*2 = 1.124 (+1)
0.124*2 = 0.248 (+0)
0.248*2 = 0.496 (+0)
0.496*2 = 0.992 (+0)
0.992*2 = 1.984 (+1)
0.984*2 = 1.986 (+1)
0.986*2 = 1.936 (+1)
0.936*2 = 1.872 (+1)
…..

Stop when 0 decimal part or out of bits…

$(0.281)_{10} = (.010001111\ldots)_2$

Numerical representation: reals

- Whereas a 32-bit real is represented as

s	exponent		mantissa	
31	30	23	22	0

- For a 64 bit real, substantially better…

s	exponent		mantissa	
63	62	52	51	0

- Stored again as $(-1)^s \times m \times 2^e$

  s = sign(0,1) m = mantissa (drop 1.)

  p = exponent (with p=e+bias=e+1023)

4 Bytes for a single

8 Bytes for a double

Special cases

- ***NaN***: p=255 (single) or 2047 (double) and m≠0
- **±*Inf***: p=255 (single) or 2047 (double) and m=0 (s-signal)
- ***0***: p=0 and m=0 (+0 and -0 are the same)
- p=0 and m≠0 give

  $(-1)^s \times 2^{-126} \times 0.m$ (single)

  $(-1)^s \times 2^{-1022} \times 0.m$ (double)

$$\text{Mantissa} = 1.f = 1 + m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \cdots \ldots + m_0 \times 2^{-23}$$

Computer storage precision

- Representation **$(-1)^s \times m \times 2^e$** → how does it affects reals' accuracy ?
  - ❑ For single precision : $2^{-23} \sim 10^{-7}$
  - ❑ For double precison: $2^{-52} \sim 10^{-16}$ mantissa 0.00000…1 (last bit set to 1)
- Extreme values e.g. single precision ------> Min = $2^{-23-127} \sim 10^{-45}$, Max=$2^{127} \sim 10^{38}$

Round-off errors

- We have a round-off whener the number is not represented exactly by the precision chosen e.g.

0.281 = $(.010001111\ldots)_2$ = 0 01111101 00001111110111110011101... (round-off)

5.75 = *0 10000001 0111* (precise) → 4 x (1 + 0.25 + 0.125+0.0625)

Overflow and Underflow

- The exponent is too large/small to be represented in the exponent field (*e*)

Round-off and discretization errors

- Programmer's worst nightmare with errors :

> Am i discretizing/oversimplifying
> or
> Limited by rounding-off errors

- Approximation/discretizing errors, occur when we simplify our problem to be solved by the computer code e.g.
  - truncating a Taylor Series (taking first N terms only)
  - using analytical approximations to complex functions
  - Discretizing a continuous function

- Round-off errors, arise when the limited number of digits (*thus precison*) that are used to represent a number prevent us from accurately representing them (no exact cancellation in some cases)

  - Subtractive cancellation …..let's ckeck it out !

Round-off – subtractive cancellation

- We know a real number **X** might not have an exact representation($\mathbf{X_c}$):

$$x_c = x(1+\varepsilon_x) \qquad |\varepsilon_x| \le \varepsilon_{precision} \longleftarrow 10^{-7} \text{ or } 10^{-16}$$

- Suppose we wish to calculate the subtraction a = b – c where b and c are large and possibly close to each other.

$$a_c = b_c - c_c = (b-c) + b\varepsilon_b - c\varepsilon_c$$

$$\frac{a_c - a}{a} \cong \frac{b}{a}(\varepsilon_b - \varepsilon_c)$$

Since the “errors” have unknown sign, caution dictates we take the worst case scenario i.e.

$$\varepsilon_b - \varepsilon_c \approx |\varepsilon_{precision}|$$

$$\boxed{\varepsilon_a \cong \frac{b}{a}\varepsilon_{precision}}$$

- Since b and c are close….a is as small as we wish → its' relative imprecision is as high as we may think of !!!
- Classical example: calculating the derivative of a function....let's analyse it...

Round-off *vs* discretization

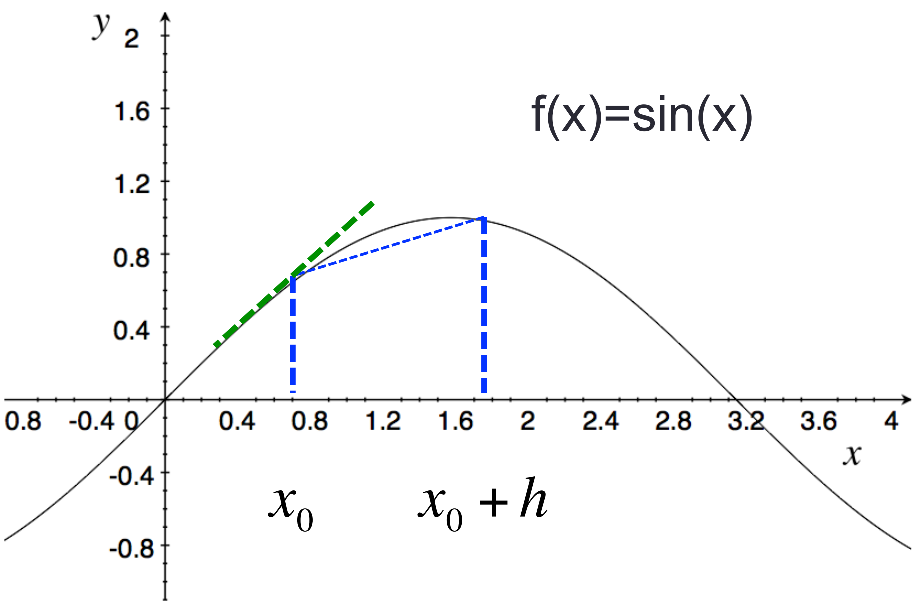
- Consider the Taylor series expansion of a function at x_0:

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0) + \dots$$

- If we approximate the derivative by $f'(x_0) \cong \dfrac{f(x_0+h) - f(x_0)}{h}$

$$\boxed{\varepsilon_D = \frac{h}{2} f''(x_0)}$$

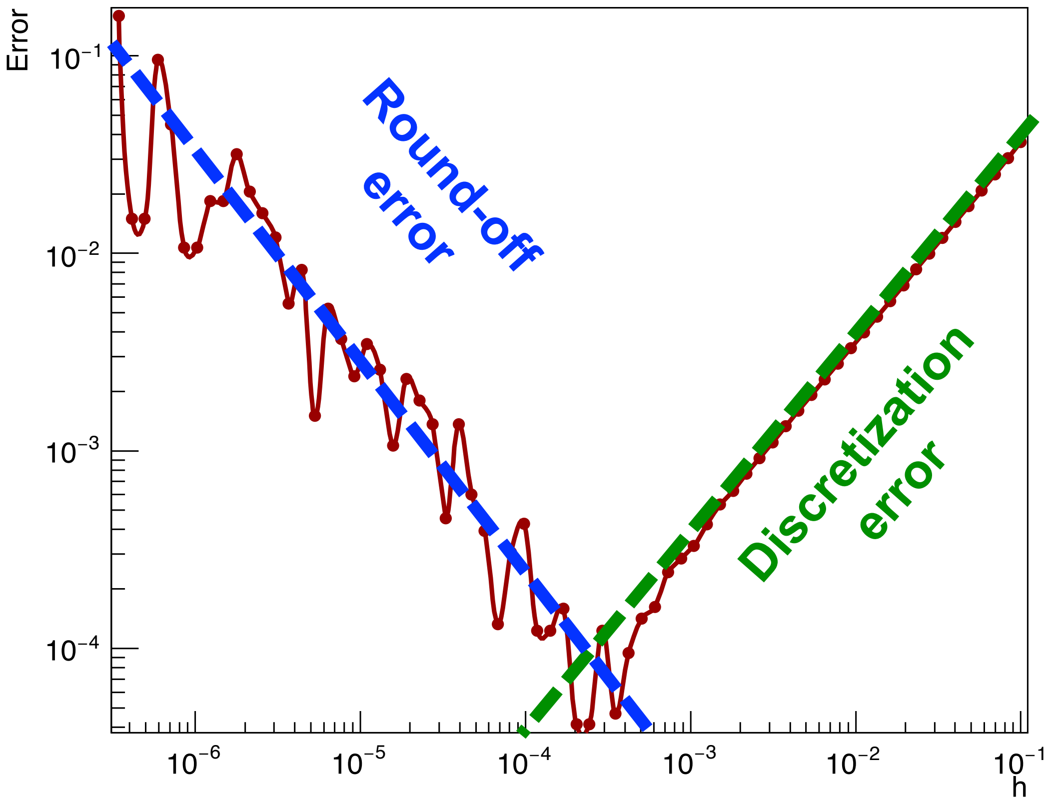
Discretization error



- ➢ For high h, the approximation clearly deviates from the true value…
- ➢ Also, as h→0 the finite difference Δf decreases, possibly reaching machine precision…

Round-off *vs* discretization

- Not surprisingly, the error first decreases with h and then increases back again !



$$\varepsilon_D = \frac{h}{2} f''(x_0)$$

Discretization error

$$\varepsilon_R = \frac{Error\left[f(x_0+h) - f(x_0)\right]}{h} = \frac{f(x_0)\varepsilon_{precision}}{h} \sim \frac{10^{-7}}{h}$$

Round-off error

Recall

$$a_c = b_c - c_c$$

$$a_c - a \cong b\varepsilon_M$$

- Best compromise : $\varepsilon_D \sim \varepsilon_R$ → $\boxed{h \sim \sqrt{\varepsilon_{precision}} \sim 3\times10^{-4}}$

Round-off *revisited*

- Round-off plagues also calculations with big numbers…

$$a_c = b_c - c_c = (b - c) + b\varepsilon_b - c\varepsilon_c$$
$$\frac{a_c - a}{a} \cong \frac{b}{a}(\varepsilon_b - \varepsilon_c)$$

$$\varepsilon_a \cong \frac{b}{a}\varepsilon_{precision}$$

- What happens if ***b*** and ***c*** become increasingly big (*assume constant difference between them or order unity…*) ?
- As soon as b ~ $1/\varepsilon_{precision}$ the two numbers ***b*** and ***c*** become so large that we loose precision is assessing the difference $\longrightarrow$ b – c yields 0 (*check !*)
- Upgrading the calculations to double precision is always helpful (*RAM memory is cheap*). Consider that C++ has already many double precision defined macros e.g. M_PI (π), M_E (Neper number), M_LN2 (ln(2)),… all courtesy of ***<math.h>***

Representation of Chars

- Characters are represented using 8 bits (*1 byte*)
- Depending on the convention, either 7 bits (ASCII format) or 8 bits (Extended ASCII) are used.
- In both cases numerical values i.e. 0..127 or 0..255 are used to code characters.

ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)

ASCII printable characters					
32	**space**	64	**@**	96	**`**
33	**!**	65	**A**	97	**a**
34	**"**	66	**B**	98	**b**
35	**#**	67	**C**	99	**c**
36	**$**	68	**D**	100	**d**
37	**%**	69	**E**	101	**e**
38	**&**	70	**F**	102	**f**
39	**'**	71	**G**	103	**g**
40	**(**	72	**H**	104	**h**
41	**)**	73	**I**	105	**i**
42	*****	74	**J**	106	**j**
43	**+**	75	**K**	107	**k**
44	**,**	76	**L**	108	**l**
45	**-**	77	**M**	109	**m**
46	**.**	78	**N**	110	**n**
47	**/**	79	**O**	111	**o**

…………

Curiosity:

→ Though obvious to detect, a simple "encryption" can be made

→ Simply shift all your password characters by adding and integer "key" !