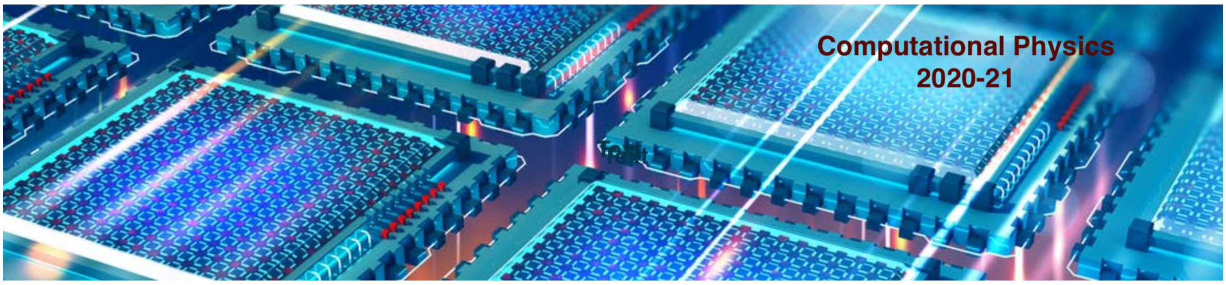
Computational Physics

numerical methods with C++ (and UNIX)
2020-21



Fernando Barao
Instituto Superior Tecnico, Dep. Fisica
email: fernando.barao@tecnico.ulisboa.pt

Computational Physics
Number representation and Machine precision

Fernando Barao, Phys Department IST (Lisbon)

Introduction

Solving physical problems with a computer

- ✔ numbers and characters representation
  - ► binary, decimal and hexadecimal systems
  - ► characters
  - ► floating point
  - ► computation errors

Numbers range...

Bohr radius calculation in SI units:

$$
\begin{aligned}
r_0 &= \frac{4\pi\varepsilon_0\hbar^2}{m_e e^2} \\
&= \frac{10^{-10} \times \left(10^{-34}\right)^2}{10^{-30} \times \left(10^{-19}\right)^2} \sim \frac{10^{-78}}{10^{-68}} \\
&\sim 5.3 \times 10^{-11}\ \mathrm{m}
\end{aligned}
$$

- ✔ What type of variables to use to store the different constants?

  single precision range: $[10^{-45}, 10^{38}]$

- ✔ How can we control the problem?

$$
\begin{aligned}
m_e &= 9.109\cdots \times 10^{-31}\ \text{Kg} \\
e &= 1.602\cdots \times 10^{-19}\ \text{C} \\
\varepsilon_0 &= 8.854\cdots \times 10^{-12}\ \text{F/m} \\
\hbar &= 1.054\cdots \times 10^{-34}\ \text{J.s}
\end{aligned}
$$

```
<type> me = 9.109E-31;
<type> e = 1.602E-19;
<type> p = 8.854E-12;
<type> h = 1.054E-34;
<type> num = p*pow(h,2.);
<type> den = me*pow(e,2.);
```

Numbers range...factorial

The calculation of factorial:

$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$

- ✔ Is it possible to calculate the factorial of any number on this way?
  it is not, because the result can assume a range very high...
- ✔ We can mitigate the problem using *log* function ...although still time expensive!
  $log(n!) = log(n) + log(n-1) + log(n-2) + \cdots + log(2)$
- ✔ Use Stirling formula for $n > 30$:
  $n! = \sqrt{2\pi n}\, n^n\, e^{-n} \left(1 + \frac{1}{12n} + \frac{1}{288n^2} \cdots\right)$
  ...and range mitigation still needed! Use log function.

Numbers 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 stored in one bit (usually the MSB)
  0 = positive
  1 = negative
- ✔ A 32 bits signed integer (4 bytes) uses 31 bits ($0 \ldots 30$) for storing the number
  max value of a signed 32bits integer:
  $2^{31} - 1 = \pm 2\ 147\ 483\ 647$

Ex: 417 conversion to binary

division	remind	coeff
$417/2 = 208$	1	1×2^0
$208/2 = 104$	0	0×2^1
$104/2 = 52$	0	0×2^2
$52/2 = 26$	0	0×2^3
$26/2 = 13$	0	0×2^4
$13/2 = 6$	1	1×2^5
$6/2 = 3$	0	0×2^6
$3/2 = 1$	1	1×2^7
$1/2 = 0$	1	1×2^8

$\boxed{(417)_{10} = (0...0110100001)_2}$

$\boxed{(-5)_{10} = -(101)_2 = (10...0101)_2}$

Numbers representation: reals

32-bits real representation

s	exponent	mantissa
31	30 … 23	22 … 0

- ✔ real number has to be converted, **the integral part** and **the decimal part** into a binary
  $\underbrace{634}_{integral}.\underbrace{28125}_{decimal}$
- ✔ The binary representation
  $...b_3b_2b_1b_0 \cdot b_{-1}b_{-2}b_{-3}...$
  $6.28125 = (110.01001)_2 \Rightarrow 2^2 + 2^1 + 2^{-2} + 2^{-5}$
- ✔ real numbers are stored as a sequence of three bit fields: $(-1)^s \times m \times 2^e$
  s = sign (0,1) m = mantissa (hidden 1.)
  p = exponent (stored p=e+bias=e+127)

  $6.28125 = (110.01001)_2 = 1.1001001 \times \underbrace{100}_{2^2}$

  $\boxed{s = 0, \quad p = 2 + 127, \quad m = 100100100...}$

Example: 6.28125

Conversion of integral part

div	res	remain
$6/2$	3	0×2^0
$3/2$	1	1×2^1
$1/2$	0	1×2^2

stop when zero on result!

Conversion of decimal part

mult	res	int
0.28125×2	0.5625	0×2^{-1}
0.5625×2	1.1250	1×2^{-2}
0.1250×2	0.2500	0×2^{-3}
0.2500×2	0.5000	0×2^{-4}
0.5000×2	1.0000	1×2^{-5}

stop when zero on decimal part!

binary conversion: limited precision

Example: 11.20

Conversion of integral part

div	res	remain
$11/2$	5	1×2^0
$5/2$	2	1×2^1
$2/2$	1	0×2^3
$1/2$	0	1×2^4

$(11.)_{10} = (1011\cdot)_2$

Conversion of decimal part

mult	res	int
0.20×2	0.40	0×2^{-1}
0.40×2	0.80	0×2^{-2}
0.80×2	1.60	1×2^{-3}
0.60×2	1.20	1×2^{-4}
0.20×2	0.40	0×2^{-5}
…	…	…

$(0.20)_{10} = (\cdot 0011001100110011\cdots)_2$

$\boxed{+1.0110011001100110011\cdots \quad 2^{+3}}$

Example: 0.42

Conversion of decimal part

mult	res	int
0.42×2	0.84	0×2^{-1}
0.84×2	1.68	1×2^{-2}
0.68×2	1.36	1×2^{-3}
0.36×2	0.72	0×2^{-4}
0.72×2	1.44	1×2^{-5}
0.44×2	0.88	0×2^{-6}
0.88×2	1.76	1×2^{-7}
0.76×2	1.52	1×2^{-8}
0.52×2	1.04	1×2^{-9}
0.04×2	0.08	0×2^{-10}
…	…	…

$(0.42)_{10} = (\cdot 0110101110\cdots)_2$

Shift now the point to the right two times to catch the first $1 \Rightarrow \times 2^{-2}$ $\boxed{+1.101011\cdots 2^{-2}}$

$\boxed{p = e + 127 = 125 \qquad m = 101011\cdots}$

Number representation: summary

- the first digit of the mantissa is always equal to **1**
  gain one bit accuracy by avoiding the storage of the mandatory first mantissa **1** bit
  it means, the mantissa has one *hidden bit*
- the exponent (**e**) is shifted by an integer bias (127) to avoid negative numbers
  this avoid us an additional bit to store exponent signal
- particular cases

0.0

exponent bits → **all** 0**'s**	mantissa bits → **all** 0**'s**

note: no confusion with the 1.0 representation that will have mantissa **m=**$000\cdots000$ and exponent **e=**$0 \Rightarrow$ **p=**127

infinity

exponent bits → **all** 1**'s**	mantissa bits → **all** 0**'s**

overflow, underflow

when exponent takes a value higher than the maximum value or lower than the minimum value that can be described with the available number of exponent bits ($2^8 - 1 = 255$)

computer storage precision

- the number of bytes assigned to a real variable (word length) is controlled by the programmer

single precision			
4 Bytes	s(1)	p(8)	m(23)
double precision			
8 Bytes	s(1)	p(11)	m(52)

accuracy

single

$2^{-23} \sim 10^{-8}$ $\boxed{0.00\quad 00\quad 00\quad 01}$

double

$2^{-52} \sim 10^{-16}$

max/min values (range)

single

$2^{127} \simeq 1.7 \times 10^{38}$

$2^{-127}\ 2^{-23} \sim \times 10^{-45}$

double bias:1023

$2^{1023} \simeq 9 \times 10^{307}$

$2^{-1023}\ 2^{-52} \sim \times 10^{-324}$

- **round-off errors**
  - a real number with a finite number of digits in the decimal system can require an infinite number of bits in the binary system

$6.28125 = 0\ 10000001\ 10010010000000000000000$ (precise)

$0.42 = 0\ 01111101\ 10101110000101000111101$ (round-off)

Types of errors

- approximation errors
  errors resulting from the problem simplification in order to be solved on the computer

  continuous functions are approximated by finite arrays of values
  - problem discretization
  - replacement of a infinite series by a sum for finite terms

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \simeq \sum_{n=0}^{N} \frac{x^n}{n!} = e^x + \Delta(x, N)$$

- round-off errors
  errors arising from using a finite number of digits to represent real numbers

Example: derivative computation

- **Function Taylor expansion:**

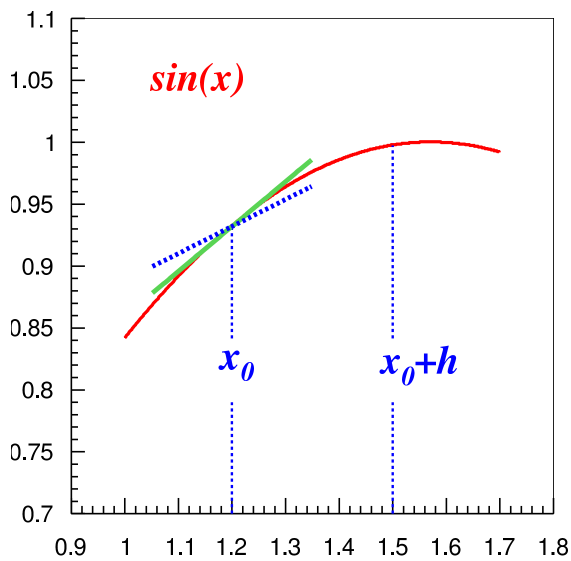
$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0) + \cdots + \frac{h^k}{k!} f^{(k)}(x_0) + \cdots$$

- **The derivative of the function can be calculated as:**

$$f'(x_0) \simeq \frac{f(x_0 + h) - f(x_0)}{h} + \frac{h}{2} f''(x_0) + \cdots$$

- **The discretization error:**

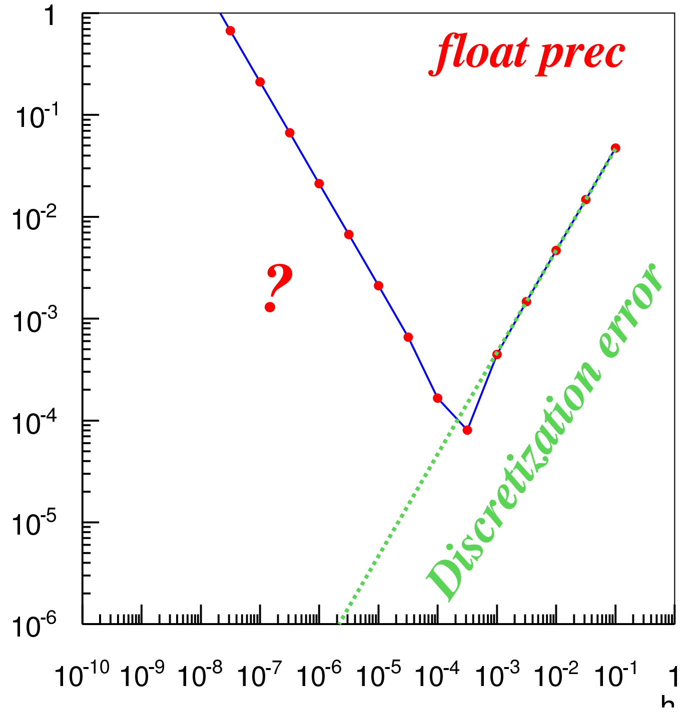
$$\Delta f'_d \simeq \frac{h}{2} f''(x_0)$$



Derivative computation errors

- Let's compute the error on the derivative $f'(x_0)$ as function of the discretization distance h: $\Delta f' = f'(x_0) - cos(x_0)$

h	$\Delta f'$
1×10^{-1}	4.7×10^{-2}
3.16×10^{-2}	1.47×10^{-2}
1×10^{-2}	4.66×10^{-3}
3.16×10^{-3}	1.46×10^{-3}
1×10^{-3}	4.44×10^{-4}
3.16×10^{-4}	8.03×10^{-5}
1×10^{-4}	1.65×10^{-4}
3.16×10^{-5}	6.55×10^{-4}



Loss of precision: sources

- Every real number $\mathbf{x}$ has a machine representation $\mathbf{x_c}$ that is an approximation to the true value, x
- The absolute error of x is,
  $\Delta x = x$ (true value) $- x_c$ (approx. value from computing represent.)
- The relative error is,
  $\varepsilon_x = \frac{\Delta x}{x} = 1 - \frac{x_c}{x}$
- That means every **represented number** can be expressed in terms of the true one, as

  $\boxed{x_c = x \; (1 - \varepsilon_x)}$

  where $|\varepsilon_x| \leq \varepsilon_M$ is the relative error associated to the machine precision
  $\sim 10^{-7}$ for single precision representation
  $\sim 10^{-16}$ for double precision representation

Loss of precision: math operations

two numbers subtraction

$$a_c = b_c - c_c = b\,(1-\varepsilon_b) - c\,(1-\varepsilon_c)$$
$$= \underbrace{(b-c)}_{a\ \text{(true nb)}} - b\,\varepsilon_b + c\,\varepsilon_c$$
$$\frac{a_c}{a} = 1 - \left(\frac{b}{a}\right)\varepsilon_b + \left(\frac{c}{a}\right)\varepsilon_c = 1 + \varepsilon_a$$

suppose that,

$b \simeq c \;\Rightarrow\; a = b - c << 1,$

we get,

$$\varepsilon_a \simeq \frac{b}{a}(\varepsilon_b - \varepsilon_c) \simeq \frac{b}{a}\,\varepsilon_M$$

Very large error on result! That's called catastrophic subtraction! We have a loss of significance!

two numbers multiplication

$$a_c = b_c \times c_c$$
$$= b(1-\varepsilon_b) \times c(1-\varepsilon_c)$$
$$\simeq \underbrace{bc}_{a} - bc\,\varepsilon_b - bc\,\varepsilon_c$$
$$\frac{a_c}{a} = 1 - \varepsilon_b - \varepsilon_c$$
$$\varepsilon_a \simeq \varepsilon_b + \varepsilon_c$$

suppose that,

$b >> c \;\Rightarrow\; \left(\varepsilon_c \sim \frac{\varepsilon_M}{c}\right) >> \left(\varepsilon_b \sim \frac{\varepsilon_M}{c}\right)$

we get,

$\varepsilon_a \simeq \varepsilon_c$

Very large error on result! That's called error magnification!

Derivative computation errors (cont.)

operation

$$f' = \frac{f(x_0+h) - f(x_0)}{h} = \frac{\delta f}{h}$$

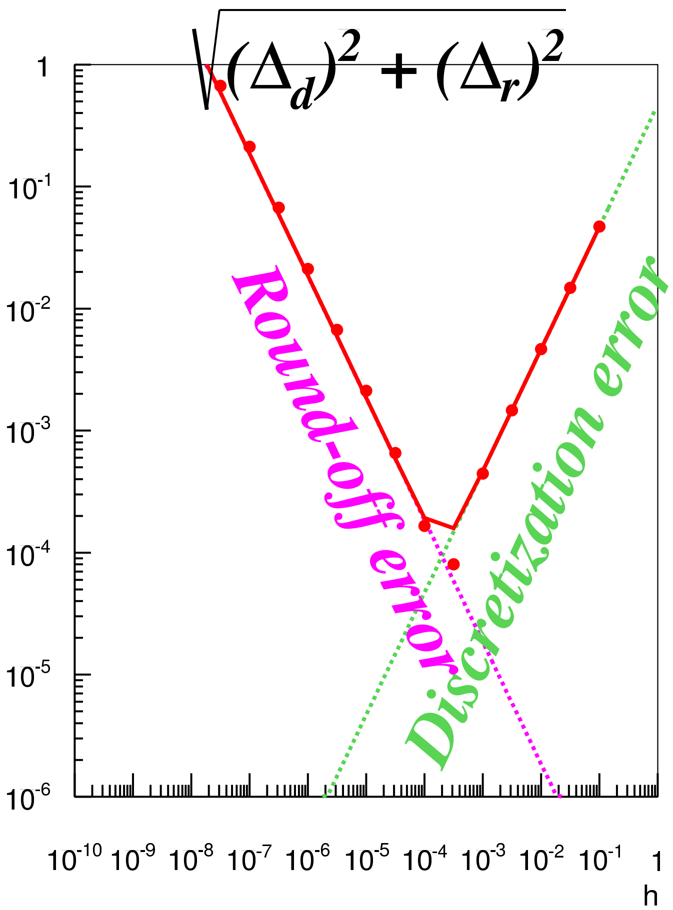
errors

- ✔ Discretization:

$$\left(\Delta f'\right)_d = \frac{h}{2} f''(x_0)$$

- ✔ Round-off:

$$\left(\Delta f'\right)_r = \frac{\Delta(\delta f)}{h} = \frac{f(x_0)}{h}\varepsilon_M \sim \frac{10^{-7}}{h}$$



Error sum

- The total error if a sequence of N arithmetic operations are made, can be estimated assuming **uncorrelated errors**

$$F = \sum_{i=1}^{N} x$$

$$(\Delta_F)^2 = \sum_{N} (\Delta_x)^2 = N\,(\Delta_x)^2$$

$$\Delta_F = \sqrt{N}\Delta_x$$

round-off error example

- Subtract two numbers, a and b steadily growing
  $a = pow(10., i) \times 1.$
  $b = pow(10., i) \times 1. + x$
- the accuracy of our result drops when the two numbers are very large (how large?)

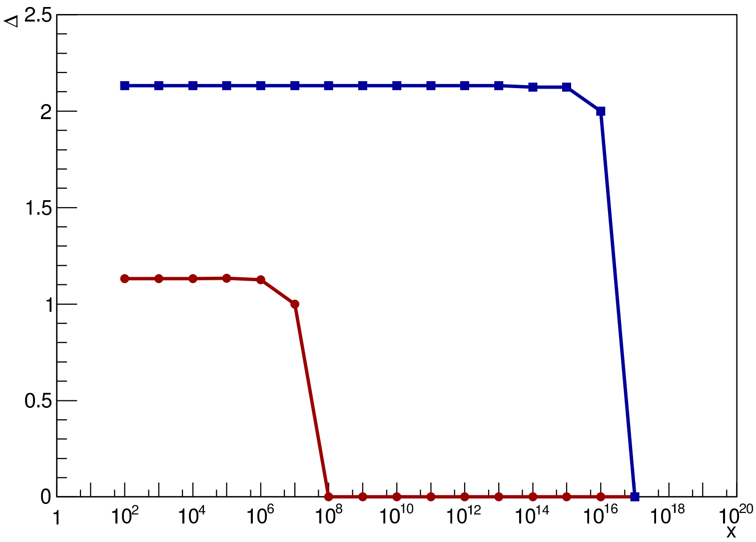
```
i=0 a=1.000000E+00 b-a=5.678900 fb-fa=5.678900
i=1 a=1.000000E+01 b-a=5.678900 fb-fa=5.678900
i=2 a=1.000000E+02 b-a=5.678900 fb-fa=5.678902
i=3 a=1.000000E+03 b-a=5.678900 fb-fa=5.678894
i=4 a=1.000000E+04 b-a=5.678900 fb-fa=5.678711
i=5 a=1.000000E+05 b-a=5.678900 fb-fa=5.679688
i=6 a=1.000000E+06 b-a=5.678900 fb-fa=5.687500
i=7 a=1.000000E+07 b-a=5.678900 fb-fa=6.000000
i=8 a=1.000000E+08 b-a=5.678900 fb-fa=8.000000
i=9 a=1.000000E+09 b-a=5.678900 fb-fa=0.000000
i=10 a=1.000000E+10 b-a=5.678900 fb-fa=0.000000
i=11 a=1.000000E+11 b-a=5.678894 fb-fa=0.000000
i=12 a=1.000000E+12 b-a=5.678955 fb-fa=0.000000
i=13 a=1.000000E+13 b-a=5.679688 fb-fa=0.000000
i=14 a=1.000000E+14 b-a=5.671875 fb-fa=0.000000
i=15 a=1.000000E+15 b-a=5.625000 fb-fa=0.000000
i=16 a=1.000000E+16 b-a=6.000000 fb-fa=0.000000
```

```
int main() {
  for (int i=0; i<20; i++) {
    double a = pow(10,i);
    double b =  a + 5.6789;
    float fa = (float)pow(10,i);
    float fb =  fa + 5.6789;
    printf("i=%d a=%f b-a=%E
    fb-fa=%f\n",i, a, b-a, fb-fa);
  }
}
```

operation: $a - b$

relative error:

$$\frac{\delta(a-b)}{|a-b|} \sim \frac{\varepsilon_M}{|a-b|} \sim \frac{10^{-7}}{|a-b|}$$



Avoid large errors: tips

1. When making multplications / divisions, make your intermediate results as close as possible to 1

How to make the operation $\frac{ab}{c}$?

- ✔ $\frac{(ab)}{c}$, if **a** and **b** have very differente values
- ✔ $\left(\frac{a}{c}\right)b$, if **a** and **c** are close in magnitude

2. Prevent loss of significance avoiding bad-subtraction operations

math.h constants

- ✔ Solving problems with a computer and requiring a good precision implies the use of double-precision in numbers representation
  - unless you are short in computer memory!
- ✔ a set of mathematical constants are already defined in the unix operating system
  - file: **/usr/include/math.h** (double-precision!)

```
/* Some useful constants.  */
#if defined __USE_BSD || defined __USE_XOPEN
# define M_E        2.7182818284590452354   /* e */
# define M_LOG2E    1.4426950408889634074   /* log_2 e */
# define M_LOG10E   0.43429448190325182765  /* log_10 e */
# define M_LN2      0.69314718055994530942  /* log_e 2 */
# define M_LN10     2.30258509299404568402  /* log_e 10 */
# define M_PI       3.14159265358979323846  /* pi */
# define M_PI_2     1.57079632679489661923  /* pi/2 */
# define M_PI_4     0.78539816339744830962  /* pi/4 */
# define M_1_PI     0.31830988618379067154  /* 1/pi */
# define M_2_PI     0.63661977236758134308  /* 2/pi */
# define M_2_SQRTPI 1.12837916709551257390  /* 2/sqrt(pi) */
# define M_SQRT2    1.41421356237309504880  /* sqrt(2) */
# define M_SQRT1_2  0.70710678118654752440  /* 1/sqrt(2) */
#endif
```

speed up you program...

- ✔ It is important to realise that multiplication (*) and division operations (/)consume considerably more CPU time than addition (+), subtraction (-), comparison or assignment operations

  ➜ **try to avoid redundant multiplication and division operations**

- ✔ For instance try to define a 3rd-order polynomial written like this:

$$f(x) = P_0 + P_1\ x + P_2\ x^2 + P_3\ x^3$$

  In total, we have 6 multiplication operations. We can optimize the polynomial expression to reduce the number of multiplications:

$$f(x) = P_0 + x\ \ (P_1 + x\ \ (P_2 + P_3\ x))$$

  The number of multiplications is now 3.

- ✔ math library tend to be extremely expensive in terms of CPU time

  ➜ **only use when absolutely necessary**
  For instance, instead of $pow(x, 2)$ use $x * x$

The hexadecimal system

- ✔ Binary numbers can be arranged in groups of 4-bits
  $(b_3\ b_2\ b_1\ b_0)_2 = b_0\ 2^0 + b_1\ 2^1 + b_2\ 2^2 + b_3\ 2^3$
  min: $(0000)_2 = 0$
  max: $(1111)_2 = 15$
  base-16 system: $0, 2, 3, 4, 5, ..., 9, A, B, C, D, E, F$
- ✔ one byte (8-bits) is represented by two hexadecimal numbers
- ✔ Examples:
  $(10\ 1111)_2 = (2F)_{16}$
  The corresponding decimal value:
  $2 \times 16^1 + 15 \times 16^0 = 47$
  $1 \times 2^5 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 47$

Characters representation

- ✔ Characters are 8-bit (byte) numbers
- ✔ ASCII (American Standard Code for Information Interchange) convention
  128 characters are represented by numerical values in the range 0-127
  7 bits needed
- ✔ The extended ASCII character set (ECS) includes 128 additional characters encoded by integers in the range 128-254
  8 bits required

Characters representation (cont.)

The ASCII Table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
00	00	NUL	32	20	SP	64	40	@	96	60	‘
01	01	SOH	33	21	!	65	41	A	97	61	a
02	02	STX	34	22	"	66	42	B	98	62	b
03	03	ETX	35	23	#	67	43	C	99	63	c
04	04	EOT	36	24	$	68	44	D	100	64	d
05	05	ENQ	37	25	%	69	45	E	101	65	e
06	06	ACK	38	26	&	70	46	F	102	66	f
07	07	BEL	39	27	’	71	47	G	103	67	g
08	08	BS	40	28	(	72	48	H	104	68	h
09	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m

• • •

24-1

24-2