Numerical Methods

Fabio Cannizzo

Math Review

- This pack of slides contains a brief summary of a number of results from calculus
- No rigorous proof or derivation is given
- Just a collection of formulas intended to be a review of already known concepts

Math Review: Some Facts

$$e = 2.7183\ldots$$

$$x^a x^b = x^{a+b}$$

$$e^{\ln a} = a$$

$$\ln e^a = a$$

$$\ln x^a = a \ln x$$

$$\ln(ab) = \ln a + \ln b$$

$$\ln(a/b) = \ln a - \ln b$$

$$\left(x^a\right)^b = x^{ab}$$

$$x^{\frac{1}{a}} = \sqrt[a]{x}$$

$$x^{-a} = \frac{1}{x^a}$$

Math Review: Derivatives

Definition:

$$\frac{df(x)}{dx} = \lim_{h \to 0^+} \frac{f(x+h) - f(x)}{h}$$

$$\frac{d}{dx} a = 0$$

$$\frac{d}{dx} ax^n = anx^{n-1} \qquad n \neq 0$$

$$\frac{d}{dx} \ln x = \frac{1}{x}$$

$$\frac{d}{dx} a^x = a^x \ln a$$

$$\frac{d}{dx} e^{ax} = ae^{ax}$$

$$\frac{d}{dx} \sin x = \cos x$$

$$\frac{d}{dx} \cos x = -\sin x$$

Math Review: Derivatives Rules

$$\frac{d}{dx}\left(f(x)+g(x)\right)=\frac{d}{dx}f(x)+\frac{d}{dx}g(x)$$

$$\frac{d}{dx}\left(f(x)g(x)\right)=g(x)\frac{d}{dx}f(x)+f(x)\frac{d}{dx}g(x)$$ product rule

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right)=\frac{g(x)\frac{d}{dx}f(x)-f(x)\frac{d}{dx}g(x)}{g(x)^2}$$ quotient rule

$$\frac{d}{dx}f\left(g(x)\right)=\frac{\partial f\left(g(x)\right)}{\partial g}\frac{d}{dx}g(x)$$ chain rule

Fundamental Theorem of Algebra

- Let $p(x)$ a polynomial of degree n, there exist uniquely n constants r_i (which do not need to be distinct or real) such that:

$$p(x) = \sum_{i=0}^{n} a_i x^i = a_n \prod_{i=1}^{n} (x - r_i)$$

- r_i are the “zeros” (or “roots”) of $p(x)$

What are Numerical Methods?

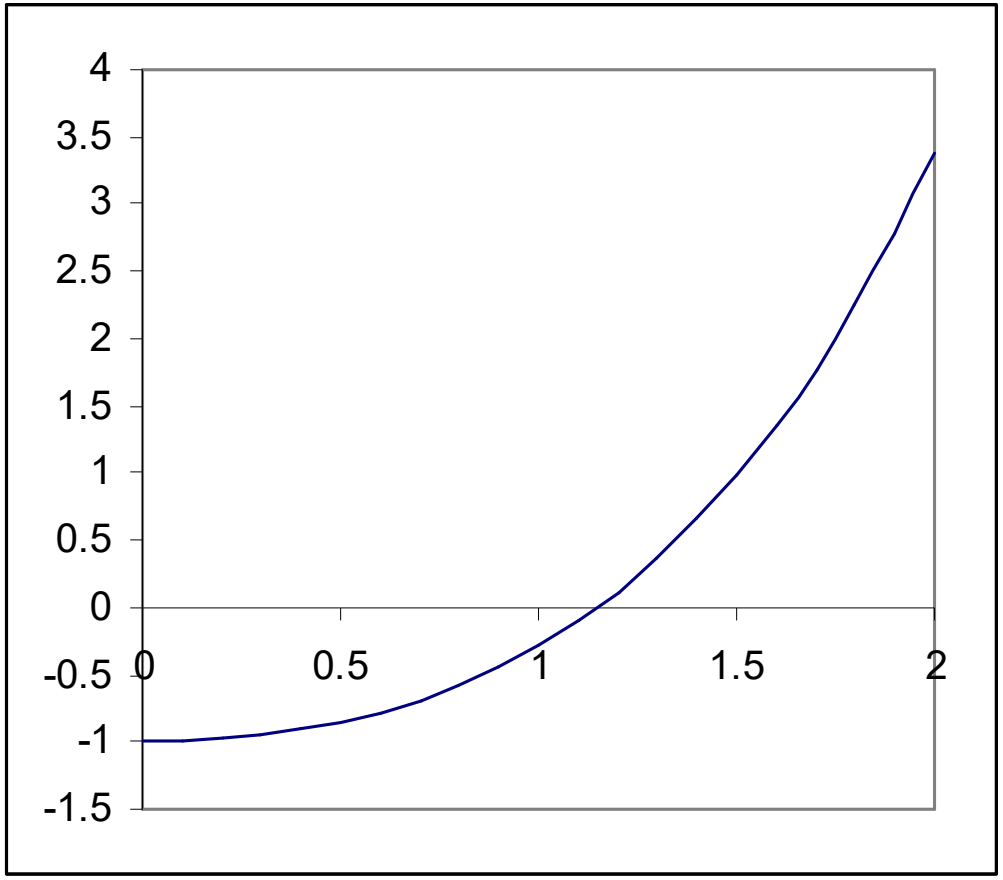
- An algorithm to solve a problem using numerical techniques, instead of symbolic analysis
- They are used because
  - a solution in closed form may not exist
  - just easier
  - suitable for computers (can be described via a mechanical sequence of elementary instructions)

Closed Form Solution Does not Exist

- Find the root of the non linear equation

  $\exp(x) - (x+2) = 0$

- We can start from an initial guess (the closer the better) and proceed by trial and error, like the game “more-less”, until we find a point which is “reasonably” close
  - Need to have a strategy
  - Need to specify what “reasonable” means



Just Easier

- Suppose we want to compute the derivative of a complicate function. We can use the definition of derivative:

$$\frac{df(x_0)}{dx} = \lim_{h \to 0^+} \frac{f(x_0+h) - f(x_0)}{h}$$

- Quick and dirty but saves us the effort to come up with the derivative in symbolic form!
- It can be easily implemented as a blind rule on a computer

Mechanical Instructions Sequence

- Computing a square root using the Babylonian/Heron method (see Wikipedia), can be implemented simply as:

```
x = 1
while (abs(x*x-y) > eps)
    x = ( y/x + x ) / 2
```

- The basic idea is that if *x* is an overestimate to *r*=sqrt(*y*) then *y/x* is an underestimate. Viceversa if *x* is an under-estimate, then *y/x* is an over-estimate. So the average of these two numbers may reasonably be expected to provide a better approximation.
- Formal proof relies on the fact that the computed average is always greater than *r*:

$$\frac{1}{2}\left(x+\frac{r^2}{x}\right)>r \quad \Leftrightarrow \quad x^2+r^2>2rx \quad \Leftrightarrow \quad \left(x-r\right)^2>0$$

always true!

Objectives of Numerical Analysis

- Purpose is to find algorithms which solve mathematical problems approximately and
  - can be implemented on a computer
  - are fast (i.e. use a minimal number of operations)
  - are accurate (i.e. give "*reasonable*" results in as many situation as possible)
  - are stable (small errors in the inputs do not cause divergence from correct solution)
  - limitations are well understood

Important Concepts

- Sources of Errors
- Minimal Number of Operations
- Stability
- Conditioning
- Convergence Speed
- Computational Cost
- Memory Requirements
- Limitations and Applicability
- Error Bounds

Sources of Errors

- Initial Data Errors
  - inaccurate measurements of data
  - miscopying of figures
  - inaccurate mathematical constants (e.g. $\pi = 3.14$)
- Truncation Errors
  - We are forced to use to use mathematical techniques which give approximate solutions, e.g. when we neglect higher order terms of a Taylor expansion
  - Usually can be improved increasing computational cost, or changing the algorithm
- Rounding Errors
  - Most numbers have an infinite number of decimal digits, but computer only represent a finite set.
  - Tend to increase with number of operations
- We are concerned in the total error, from all sources

Stability

- An algorithm is unstable if the accumulation of errors of various nature grows in explosive manner
  - The solution diverges from the correct one
  - Errors are amplified instead of being smoothed
- It is a property of the algorithm, but seen it in the context of a particular problem. For instance an algorithm could be stable for certain problems and unstable for another.

Conditioning

- A problem is **ill-conditioned**, if small variations of the initial condition lead to large variations of the final solution
- This cause small errors in input data to have a large error on the final solution
- This is a property of the problem, regardless of the algorithm used to solve it

An Ill-Conditioned Problem

We want to find the root of the polynomial:

$$x^4-8x^3+24x^2-32x+16=0$$

$$(x-2)^4=0$$

If we do just a very minor change to the coefficients:

$$x^4-8x^3+24x^2-32x+15.99999999=0$$

The roots become: $2.01,\ 1.99,\ 2\pm i0.01$

Let's analyse the effect of changes in the last coefficient:

$$x^4-8x^3+24x^2-32x+\alpha=(x-2)^4+\alpha-16=0$$

$$x(\alpha)=2\pm\sqrt[4]{16-\alpha}$$

$$x'(\alpha)=\mp\frac{1}{4\sqrt[4]{(16-\alpha)^3}} \quad \text{(differentiating on } \alpha)$$

$$\lim_{\alpha\to 16} x'(\alpha)=\infty$$

What Are Rounding Errors

- On a computer integer arithmetic is “exact”, but Real numbers are represented with a finite set of digits, therefore almost anything which has to do with floating point number is approximated
  - This includes numbers (with the exception of the finite set of numbers which can be represent exactly)
  - Results of basic mathematical operations (e.g. addition)
  - Results from algorithms (e.g. exp(x))

Computer Arithmetic

- We already know that **irrational** numbers cannot be represented exactly
- Also **rationale** numbers with too many decimal digits cannot be represented exactly
- Note that numbers which have an exact representation in the decimal system, may not have it in the other systems
- This source of error is called **round off**

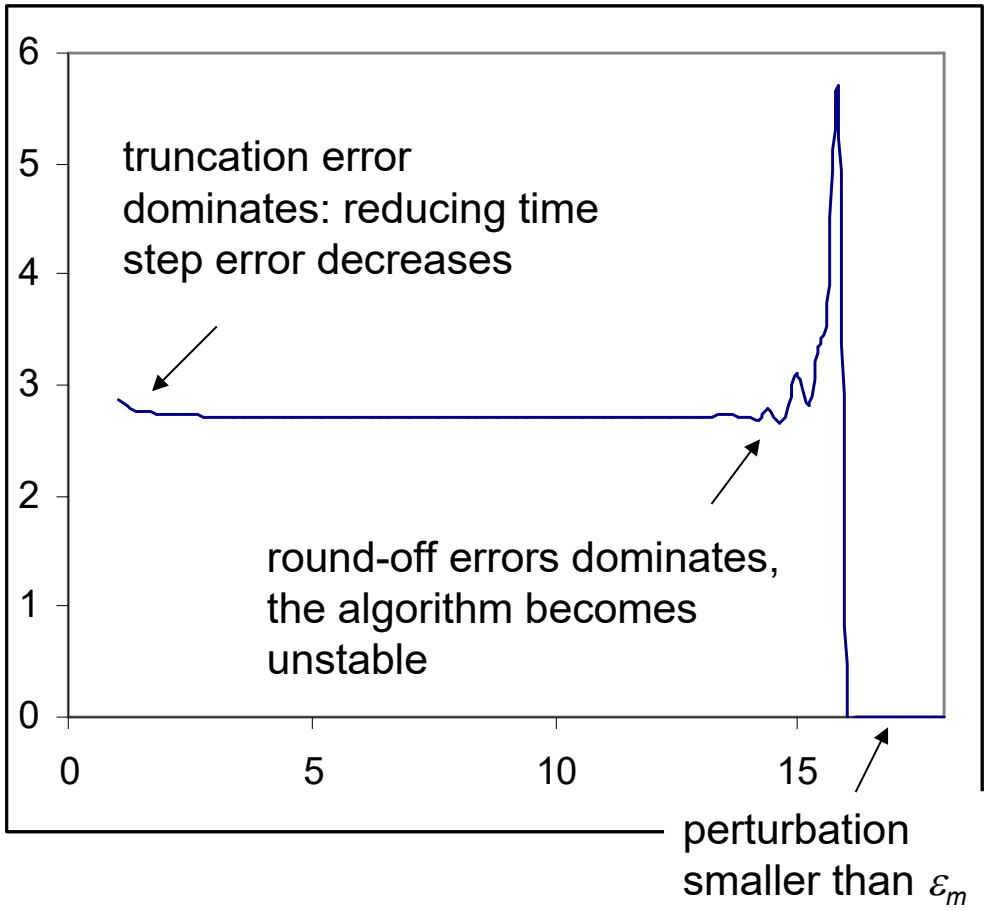
Why Does Round-Off Error Matter?

- If a numerical methods is properly designed, in absence of rounding errors the method should be able to theoretically achieve approximations of arbitrary accuracy by increasing the cost of computations. But:
  - the rounding error usually pose a limit to the achievable accuracy
  - the rounding error can invalidate results

Rounding Error

- Numerical estimation of the derivative of exp(x) in x=1
  - On the abscissa we have the exponent *b*

$$\frac{d\exp(1)}{dx} = \lim_{b\to\infty} \frac{\exp\left(1+10^{-b}\right)-\exp(1)}{10^{-b}}$$



What is going on?

- The function exp(*x*) is computed in approximate form by the computer, e.g. it contains rounding errors

$$\frac{d\exp(1)}{dx} = \lim_{b\to\infty} \frac{\left(\exp\left(1+10^{-b}\right)+\varepsilon_1\right)-\left(\exp(1)+\varepsilon_2\right)}{10^{-b}}$$

$$¿\frac{\varepsilon_1-\varepsilon_2}{10^{-b}} + \lim_{b\to\infty} \frac{\exp\left(1+10^{-b}\right)-\exp(1)}{10^{-b}}$$

When b becomes large, the sum of rounding errors becomes material

- Eventually, for very large b, the computer no longer distinguishes between original number and perturbated number, therefore the numerator drops to zero

$$1 \equiv 1+10^{-b}$$ **espilon machine**

Number Representation

- Computer represents number with a finite number of digits in base 2.
- The number of digits available depends the storage space available allocate to a certain number.
- There are some standard types, which have standard size, e.g int32 is an integer stored in a 4 byte word (32 bits).
- The **base** is the number of single-digit-numbers available in the numerical system (e.g., in the decimal system there are 10 single-digit numbers 0,1,2,3,4,5,6,7,8,9)
- Most common base representations are binary (2), decimal (10), octale (8) and hexadecimal (16)

Number Representation

- Any integer number can be represented exactly in any base finding the appropriate set of coefficients a_i such that:

$$value = \sum_{i=0}^{n} a_i \, base^i$$

$$\underbrace{2\cdot 10^1 + 4\cdot 10^0}_{decimal:24} = \underbrace{1\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 +}_{binary:\ 11000} = \underbrace{1\cdot 16^1 + 8\cdot 16^0}_{hexdecimal:18}$$

- From a computer arithmetic point of view, integer operations are exact (i.e. there is no rounding)
- There can be however overflow, if the number is too large to be represented (e.g. *n* is too big)

Decimal -> Binary Conversion

```
function m=decInt2BinInt( x, nbits )
   m=zeros(1,nbits);
   p = 1;
   for i=1:nbits
      t = p*2;
      if mod(x,t) != 0
         x=x-p;
         m(nbits+1-i)=1;
      endif
      p=t;
   endfor
endfunction

>> dec2bin(24,8)
>> ans =
>>   0   0   0   1   1   0   0   0
```

Computer Arithmetic

- Computer typically represents decimal number in terms of mantissa, base and exponent. Both mantissa and exponent have a sign. The exponent is integer. The base is fixed.

  x = (-1)^sign * mantissa * base ^ exponent

- Suppose we have three digits for the mantissa and two digits for the exponent, and the chosen base is 10, every number must be represented in the format

  x.xx * 10 ^ yy

  Example: 31.89 gets rounded to 3.19 * 10^1

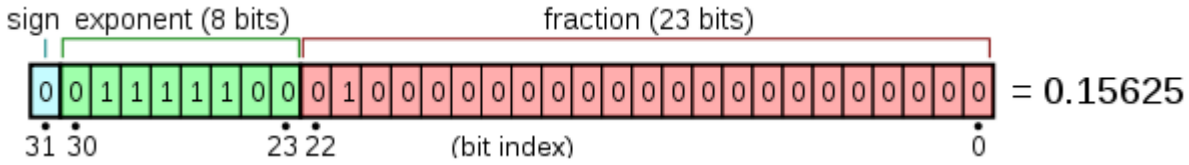
IEEE-754

- A technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).
- Many hardware floating point units use the IEEE 754 standard.
- Includes the definition of the commonly used format binary32 (float) and binary64 (double)

IEEE-754 Binary32 (float)

- Number are represented with 32 bits (4 bytes)
  - Sign bit: 1 bit
  - Exponent width: 8 bits (integer ranging from 0 to 255)
  - Exponent Offset: 127
  - Significand precision: 24 (23 explicitly stored)
  - Some special combinations of bits are used to represent infinity and Not-A-Number
- In most languages this format is referred to as "*float*" or "*single*"

IEEE-754 Binary32 (float)



$$value = (-1)^{a_{31}} \left(1 + \sum_{i=1}^{23} a_{23-i} 2^{-i}\right) 2^{\left(\sum_{i=0}^{7} a_{30-i} 2^{i} - 127\right)}$$

- In the example
  - sign = 0
  - exponent: $(2^2+2^3+2^4+2^5+2^6) - 127 = 124 - 127 = -3$
  - mantissa: $1 + 2^{-2} = 1.25$
  - value = $1.25\ 2^{-3} = 0.15625$

IEEE-754 Binary64 (double)

- Number are represented with 64 bits (8 bytes)
  - Sign bit: 1 bit
  - Exponent width: 11 bits (integer ranging from 0 to 2047)
  - Exponent offset: 1023
  - Significand precision: 53 (52 explicitly stored)
  - Some special combinations of bits are used to represent infinity and Not-A-Number
- In most languages this format is referred to as "*double*"

Decimal to Binary Conversion

We search for the binary representation of the decimal number y:

$y=(-1)^s q\cdot 2^n, \quad 1\le q<2,$ with n integer

It is trivial to determine s. Next we determine n:

$n=\log_2|y|-\log_2 q=\text{floor}\left(\log_2|y|\right)$ because $0\le\log_2 q<1$

We compute the fractional part:

$$r=\frac{|y|}{2^n}-1$$

now we need to find the coefficients a_i such that:

$$r=\frac{a_1}{2}+\frac{a_2}{4}+\frac{a_3}{8}+\ldots$$

$i=1$

while $(r>0)$

$r=2r$

if $(r>1)$ then $r=r-1; a_i=1;$ else $a_i=0$

$i=i+1$

This could be an infinite loop. What else do we need to check is missing here?

Decimal to Binary Conversion

$y = 0.15625$

$s = 0$

$n = \text{floor}\left(\log_2 0.15625\right) = -3$

$\text{exp} = 127 + \left(-3\right) = \left(124\right)_{10} = \left(01111100\right)_2$

$r = 0.15625 / 2^3 - 1 = 0.25$

$r = 2r = 0.5,$ since $r < 1$ we set $a\left(1\right) = 0$

$r = 2r = 1.0,$ since $r \geq 1$ we set $a\left(2\right) = 1$ and $r = r - 1 = 0$

since $r = 0$ we stop

hence: 0-01111100-01000000000000000000000

Digital Binary Conversion

```
function [binstr, s,a,n]=fpdec2bin( x, nBitMantissa, nBitExp, bitOffset )
   if x<0, s=1; x=-x; else, s=0; endif
   a=zeros(1,nBitMantissa);
   n=floor(log2(x));
   r = x/(2^n)-1;
   i = 1;
   while (r>0 && i<=nBitMantissa)
      r = 2*r;
      if r>=1
         r=r-1;
         a(i)=1;
      endif
      i=i+1;
   endwhile
   n=dec2bin(n+bitOffset,nBitExp);
   binstr = [num2str(s), "-"];
   for i=1:nBitExp binstr=strcat(binstr,num2str(n(i))); endfor
   binstr = [binstr, "-"];
   for i=1:nBitMantissa binstr=strcat(binstr,num2str(a(i))); endfor
endfunction

>> fpdec2bin(0.15625,23,8,127)
ans = 0-01111100-01000000000000000000
```

In this code, something could potential go wrong. Can you spot the issue???

Round-Off Error

- The example introduced (0.15625) has an exact representation in binary format
- However, if we try to represent in binary format the number 0.1, which is a nice power of 10, we will find that it does not have a finite binary representation
- Rounding is required

Round-Off Error Bounds

$m = 1.a_1 a_2 a_3 a_4 \ldots a_p a_{p+1}$ (unfortunately we only have p digits)

$m^- = 1.a_1 a_2 a_3 a_4 \ldots a_p$ (ignore remaining digits: round towards zero)

$m^+ = m^- + (1/2^p)$ (add the smallest digit: round away from zero)

the computer will choose the closest one, which we call $m^¿$.

What can we say about the error?

$|m^¿ - m| \le (1/2)|m^+ - m^-| = 1/2^{p+1}$ (worst case is exactly in the middle)

Absolute Error: $|m \cdot 2^e - m^¿ \cdot 2^e| = |m - m^¿| \cdot 2^e \le 2^{e-p-1}$

Relative Error: $\left|\frac{m \cdot 2^e - m^¿ \cdot 2^e}{m \cdot 2^e}\right| = \left|\frac{m - m^¿}{m}\right| \le \frac{1}{m\,2^{p+1}} \le \frac{1}{2^{p+1}}$

In essence for any Real x: $x^¿ = x(1+\delta)$, where $|\delta| \le 1/2^{p+1}$

For a machine that uses p binary digits to represent the mantissa of a machine number, the number $\varepsilon = 2^{-p-1}$ that bounds the relative error in converting real numbers to machine numbers is called the **unit roundoff error** for the computer.

Round-Off Error Propagation

- Rounding can happen (and it usually does happen) in all mathematical operations (e.g. +, -, /, *, etc).
- How does rounding error propagate across mathematical operations?

If x is a Real number, let's indicate with $c(x)$ it's machine representation and with $\circ$ a generic operation

we want $(x \circ y)$ and x and y are exact machine numbers, i.e. $x = c(x), y = c(y)$

$$c(x \circ y) = [x \circ y](1+\delta), \qquad |\delta| < 2^{-p-1}$$

we want $(x \circ y)$ and x and y are not exact machine numbers:

$$c(c(x) \circ c(y)) = [c(x) \circ c(y)](1+\delta) = [x(1+\delta_1) \circ y(1+\delta_2)](1+\delta_3), \qquad |\delta_i| < 2^{-p-1}$$

Round-Off Error Propagation

Addition of two positive reals x and y:

$$c\left(c(x)+c(y)\right)=\left(x\left(1+\delta_1\right)+y\left(1+\delta_2\right)\right)\left(1+\delta_3\right) \qquad |\delta_i|\le\varepsilon=2^{-(p+1)}$$

$$\text{¿}\; x\left(1+\delta_1+\delta_3+\delta_1\delta_3\right)+y\left(1+\delta_2+\delta_3+\delta_2\delta_3\right) \qquad |\delta_i\delta_j|\le\varepsilon^2=2^{-2(p+1)},\quad |\delta_i+\delta_j|\le 2\varepsilon$$

$$\text{¿}\; x\left(1+\delta_4\right)+y\left(1+\delta_5\right) \qquad |\delta_i|\le 2\varepsilon=2^{-p}$$

$$\text{¿}\; x+y+x\delta_4+y\delta_5$$

$$\text{¿}\left(x+y\right)\left(1+\delta_6\right) \qquad |\delta_i|\le 2^{-p}$$

The last passage is because:

$$|x\delta_4+y\delta_5|<|(x+y)\max\left(|\delta_4|,|\delta_5|\right)|=|(x+y)\delta_6|$$

- This tells us that we could loose one digit precision just by adding two numbers.
- Luckily this is a worst case scenario, often things are not that bad!

The Summation Problem

- One problem with summation is that as we add new terms, the accumulation variable becomes large, and the digit of the new terms which get added become less and less significant with respect to the accumulation variable
- Simple strategies are to chunk the sum in sub-sums, and then sum the results of each sub-sum
- Another is to order terms first and add from the smaller to the larger, which is useful if the terms added are very different in magnitude
  - Imagine on a 2-digits decimal computer we add:
    10 + 0.2 + 0.2 + 0.2 + 0.2 + 0.2 = 11
  - If we add from left to right, the result is 10
  - If we add from add from right to left the result is 11
  - If we split in 2 sub-sums the result is 11
- There are also more sophisticated techniques, e.g. Kahan

Rounding Error of a Summation

Theorem

- Let $x_1, x_2, \ldots, x_n$ be positive machine numbers in a computer whose unit round-off number is ε, then the relative round-off number of their sum is:

$$c\left(\sum_{i=1}^{n} x_i\right)=\left(\sum_{i=1}^{n} x_i\right)(1+\delta)$$

where

$$|\delta| \leq (1+\varepsilon)^n - 1 \approx n\varepsilon$$

Kahan Summation Algorithm

- When accuracy of a sum is critical, there are techniques which allow to increase accuracy. One is Kahan summation algorithm:

```
function sum=kahan(data)
    sum = 0.0;
    c = 0.0;  # Keep track of lost low-order bits.
    for i = 1:length(data)
        y = data(i) - c; # add cumulated error
        t = sum + y;     # here there is round-off error
        c = (t-sum)-y;   # track new error
        sum = t;
    endfor
endfunction
```

- Note this reduce the error, but it does not eliminate it

SumDemo.m

N	Sum(1/n)	1-Sum(1/n)	KahanSum(1/n)	1-KahanSum(1/n)
1.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
2.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
3.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
4.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
5.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
6.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
7.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
8.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
9.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
10.0	1.0000001192092	-0.0000001192092	1.0000000000000	0.0000000000000
11.0	1.0000001192092	-0.0000001192092	1.0000000000000	0.0000000000000
12.0	0.9999998807907	0.0000001192092	1.0000000000000	0.0000000000000
13.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
14.0	1.0000001192092	-0.0000001192092	1.0000000000000	0.0000000000000
15.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
16.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
17.0	1.0000000000000	0.0000000000000	1.0000000000000	0.0000000000000
18.0	1.0000002384185	-0.0000002384185	1.0000000000000	0.0000000000000
19.0	0.9999998211860	0.0000001788139	1.0000000000000	0.0000000000000
20.0	1.0000001192092	-0.0000001192092	1.0000000000000	0.0000000000000

X+Y=X

- Let's go back to decimal system (*base*=10) and assume we are working with 3 digit precision, i.e., ignoring the *sign*, every number is represented as: *d.dd**10^m
- Let's consider the sum:

$$4.23\cdot10^{2}+1.20\cdot10^{-2}=4.23\cdot10^{2}+0.000120\cdot10^{2}=4.230120\cdot10^{2}=4.23\cdot10^{2}$$

- First we equalize the exponent, so that the two mantissas are comparable and can be added, then we do the addition, last we round the result
- We obtain the first operand unchanged!
- **In mathematics *x*+*y*==*x* implies that *y*=0, but in computer arithmetic this law does not hold.**

Epsilon Machine

- We define **epsilon machine** (ε_m) as the smallest number which added to one, yields a result different from one:
- Epsilon machine depends on the chosen floating point representation (e.g. *single* or *double*). In double precision, Octave returns:

```
>> eps(1)
ans = 2.2204604925031e-016
```

- We already encountered it with the name **round-off unit**
- It should be no surprise to find in numerical algorithms the expression

```
if x+y==x ...
```

- Zero machine is the smallest number the computer can represent. In double precision;

```
>> eps(0)
ans = 4.9406564584124e-324
```

Some Consequences of Round-Off

Possible bizarre behaviours which may occurr:

$x+y=x \quad \text{with} \quad y\neq 0$

$x+(y+z)\neq(x+y)+z \quad (\text{associative property does not hold})$

$x*y=0 \quad \text{is possible with} \quad x\neq 0 \quad \text{and} \quad y\neq 0$

$x*(1/x)\neq 1$

$$\sum_{i=1}^{n}\frac{1}{n}\neq 1$$

Veryfing equality on floating point numbers is a tricky thing:

```
>> 0.1+0.2==0.3
ans = 0
>> 0.1+0.3==0.4
ans =  1
```

Subtractive Cancellation

- Describes what can happen when subtracting two numbers very close to each others
- All the significant digits cancel out and only numbers which are purely due to round-off errors (i.e. computation noise) remain
- However these numbers are treated in successive computations as if they were reliable ones

Subtractive Cancellation

- Consider the subtraction:

```
>> a=0.3721478693;
>> b=0.3720230572;
>> a-b
ans = 1.24812099999982e-004
```

- If we do the same operation using a decimal representation with 5 digits mantissa:

$$c(a)=0.37215$$
$$c(b)=0.37202$$
$$c(a)-c(b)=0.00013$$
$$\left|\frac{[a-b]-[c(a)-c(b)]}{a-b}\right|=\left|\frac{0.000124812-0.00013}{0.000124812}\right|\approx 4\%$$

Subtractive Cancellation

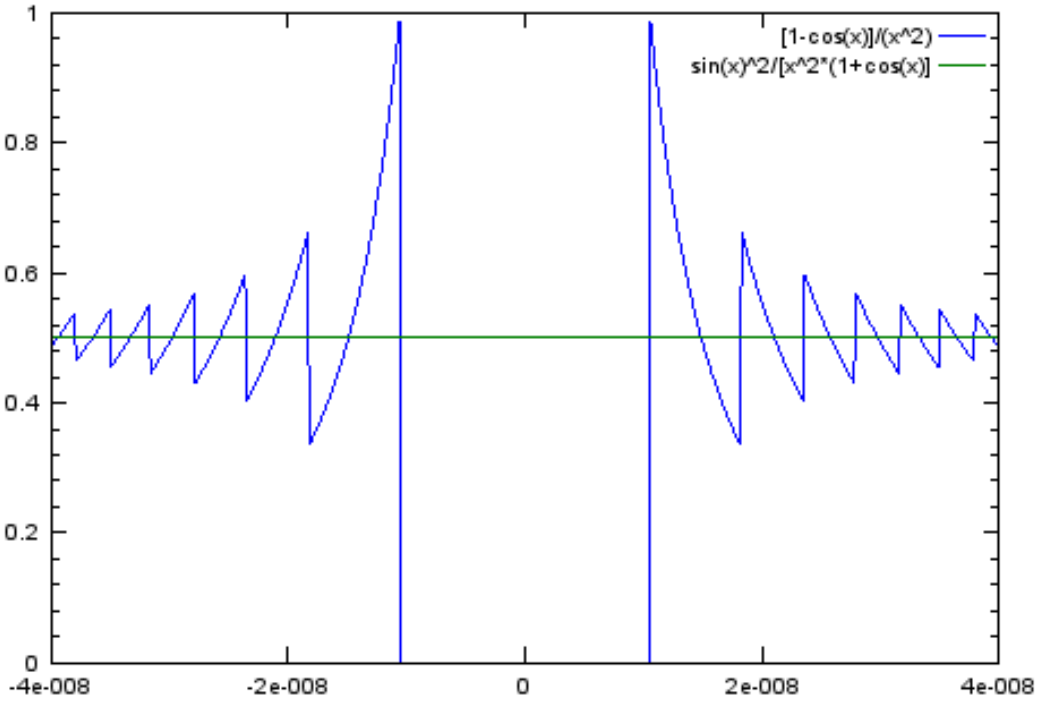
- Subtractive cancellation can be:
  - **benign**: if x and y are both perfect machine numbers
  - **catastrophic**: if x or y are machine rounded numbers
- When such type of error occurs, we may try to identify the “sensitive” subtractions and eliminate them, if possible, by analytic manipulation

Catastrophic Cancellation

- $b^2-a^2=(b+a)(b-a)$
- The first formulation is affected by catastrophic cancellation, because b^2 and a^2 are most likely affected by rounding error
- The second formulation is much more robust and it is not more expensive: 2mul,1sum vs 2sum,1mul

Catastrophic Cancellation

- Consider the function:
  $[1-\cos(x)]/x^2$
- Close to x=0 its value is roughly 0.5
- Even using IEEE-754 in double precision, the error is massive (100%)
- We can fix it by rewriting it to avoid the subtraction:
  $[\sin(x)/x]^2/[1+\cos(x)]$




```
>> myfun1=@(x) (1-cos(x))./(x.^2);
>> myfun2=@(x) (sin(x)./x).^2./(1+cos(x));
>> x=[-4e-8:1e-10:4e-8];plot(x,myfun1(x),x,myfun2(x))
```

Double Precision

- Computation in double precision cost about double computation in single precision and require double memory
- However with modern fast computer, with plenty of ram, this extra costs are usually negligible
- Double are much more accurate than single precision numbers and make round-off errors less of a problem. I.e. significant round-off errors appears less frequently.
- Beware, this does not mean we should lower the guard!

Minimal Number of Operations

- Consider evaluating a polynomial

  $a_0 + a_1 x + a_2 x^2 + a_3 x^3 = a_0 + a_1 x + a_2 x x + a_3 x x x$

  (3 sum, 6 mul) n sum, n(n+1)/2 mul
- We can do better storing cumulated x^i

  $a_0 + a_1 x + a_2 x^2 + a_3 x^3 = a_0 + a_1 x + a_2 x x + a_3 x^2 x$

  (3 sum, 5 mul) n sum, n+(n-1) mul
- We can do better using **Horner rule**

  $a_0 + a_1 x + a_2 x^2 + a_3 x^3 = a_0 + x (a_1 + x (a_2 + a_3 x))$

  (3 sum, 3 mul) n sum, n mul

Designing an Algorithm

- A number of metrics can be used to describe the properties of numerical methods and to compare the relative “goodness” of different numerical methods which solve the same problem:
  - execution speed
  - resources requirements (memory)
  - accuracy achievable (bias)
  - convergence rate
  - stability (effects of rounding errors)
- Usually there is trade-off between accuracy and speed/resources

Designing an Algorithm

- Accuracy is defined as the “closeness” of the result obtained to the theoretically correct value
- Sometime it is difficult to measure it, because the theoretical value is unknown
- We can categorize algorithms as **fixed** cost and **parametric** cost
- A fixed cost algorithm
  - is executed on average in constant time
  - use on average the same amount of resources
  - its accuracy cannot be controlled
- A variable cost algorithm
  - Based on a parameter, which allow to improve accuracy at expense of speed and resource requirements

Math Review: Taylor Expansion

Let $f(x)$ be a function continuous with its first (n+1) derivatives in $[a,b]$, let x and $x+h$ be two points in $[a,b]$, then there exist a number ξ between x and $x+h$ such that:

$$f(x+h)=f(x)+\sum_{n=1}^{\infty}\frac{1}{n!}\frac{d^n f(x)}{dx^n}h^n$$

$$f(x+h)=f(x)+\sum_{n=1}^{N}\frac{1}{n!}\frac{d^n f(x)}{dx^n}h^n+\frac{1}{(N+1)!}\frac{d^{N+1} f(\xi)}{dx^{N+1}}h^{N+1} \quad \xi\in[x,x+h]$$

$$=f(x)+\sum_{n=1}^{N}\frac{1}{n!}\frac{d^n f(x)}{dx^n}h^n+O(h^{n+1})$$

Designing an Algorithm

- We illustrate some concepts with a simple example, the computation of sin(x)
- A possible algorithm is based on Mc Laurin expansion (this is for illustration, there are better algorithms)

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

- The error we make by taking only a few terms of the infinite series is called **truncation error**

Designing an Algorithm

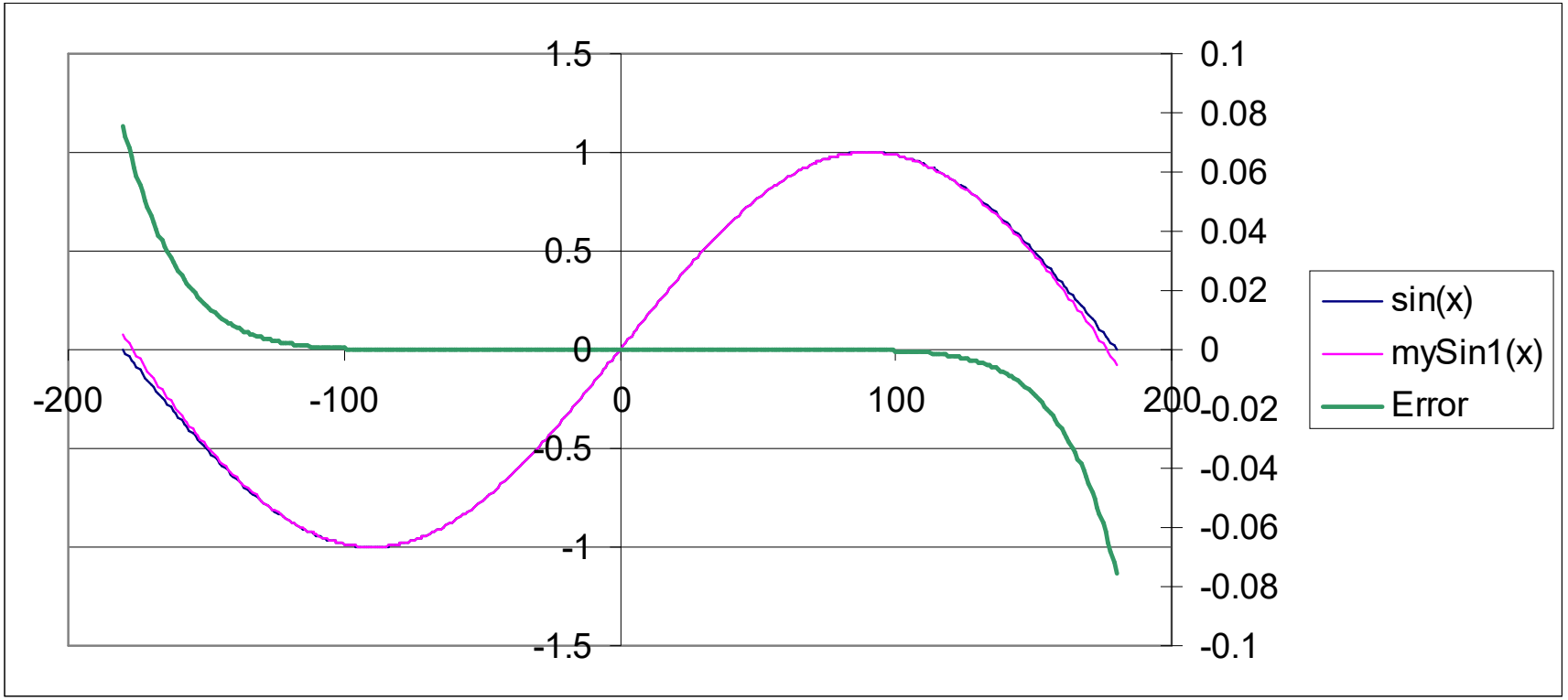
- sin(x): algorithm 1
  - It uses just the first 4 terms
  - Execution time is constant
  - Precision cannot be controlled and the error depends on the value of x

```
##      mySin1( x )
## Approximation of Sin( x ) by taylor expansion
## of order 7
function sinx = mySin1( x )

    c3 = -0.1666666666666667; ## 1/3!
    c5 = 8.333333333333333e-3; ## 1/5!
    c7 = -1.984126984126984e-4; ## 1/7!
    x2 = x .* x;

    sinx = x .* ( 1.0 + x2 .* ( c3 + x2 .* ( c5 + c7 .* x2 ) ) ); ## Horner rule
```

hardcoding constants: a good idea?

note the variation of Horner, based on x^2

Designing an Algorithm



Designing an Algorithm

- We have Taylor terms up x^7, and we know the eight term is null. Hence we can estimate bounds for the error

$$|e(h)| = \frac{1}{9!}\left|\frac{d^9 f(\xi)}{dx^9} h^9\right| \leq \frac{1}{9!}|h^9| = E(h)$$

h ranges in [-π,π], hence |*h*| can range in [0,π]

- If *h* reduces by a factor of 10, we expect the error to reduce by a factor of 10^9.

$$\frac{|E(h)|}{\left|E\left(\frac{h}{10}\right)\right|} = \frac{|h^9|}{\left|\left(\frac{h}{10}\right)^9\right|} = 10^9 \quad \Rightarrow \quad E\left(\frac{h}{10}\right) = 10^{-9}|E(h)|$$

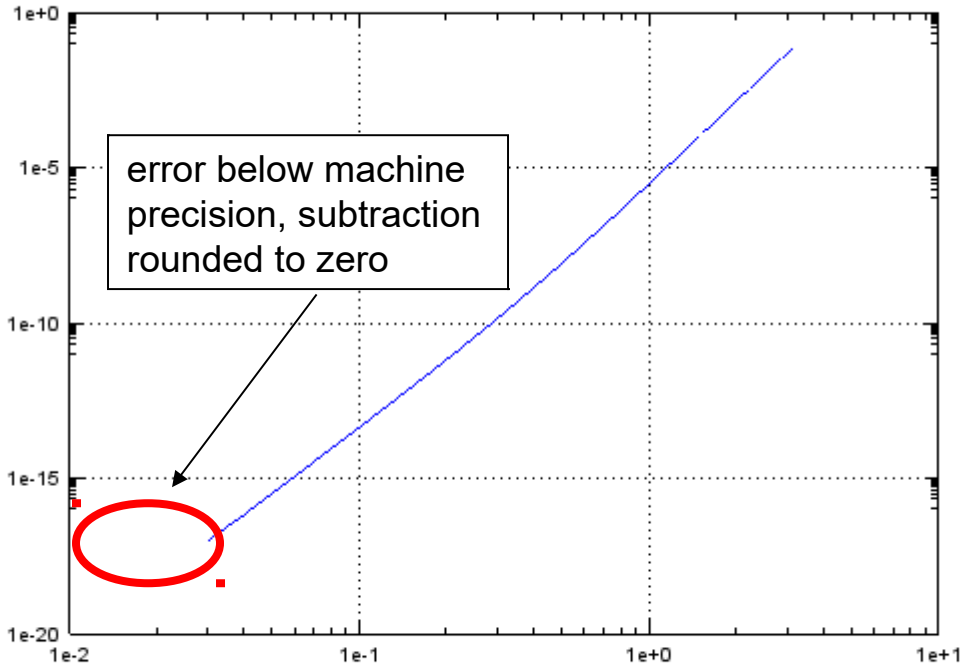
Plotting Polynomial Convergence

- We can show graphically the rate of reduction of the error by plotting in logarithmic scales

$$|e(h)| \propto |h^9| \quad \Rightarrow \quad \log|e(h)| \propto \log|h^9| \quad \Rightarrow \quad \log|e(h)| \propto 9\log|h|$$

- If we plot the logarithm of the error in absolute value against the logarithm of *h*, we expect to see roughly a straight line with slope 9
- A caveat: we will use the Octave sin(x) function as a reference of the true value of sin(x), but in reality we do not know exactly the error

Error in [0,π] in log-log Scale



The function loglog in Octave plots x and y, but transforms both of them in log

The scale of the chart is now in log, but the labels are the original numbers

See also semilogy and semilogx

```
>> x=[0.01:0.01:pi];  # it would be better equally spaced in log!
>> y=abs(mySin1(x)-sin(x));
>> loglog(x,y)
```

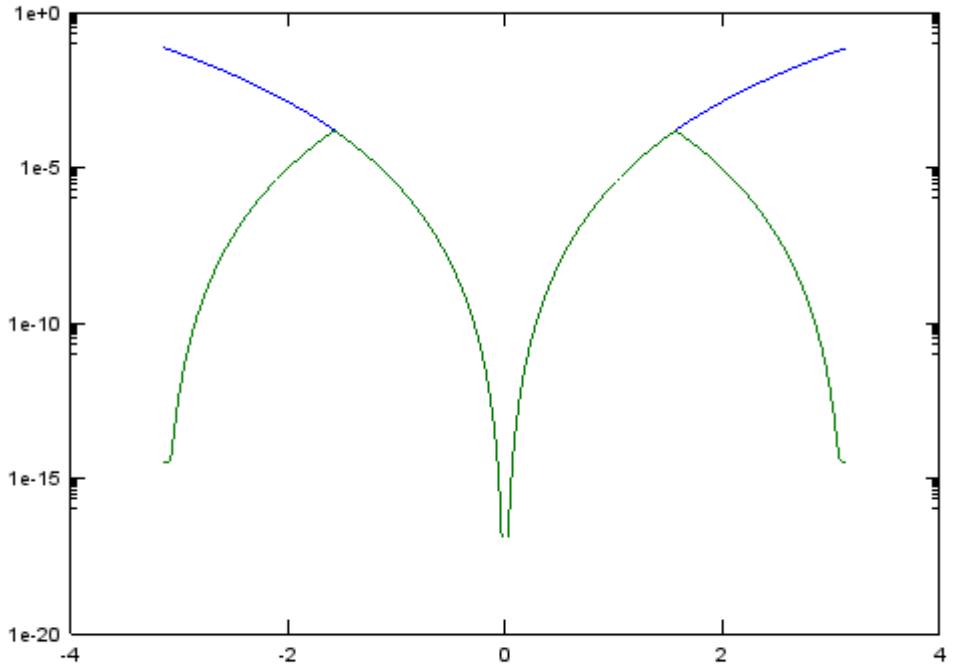
Exploit Properties of the Problem

- Can we do better, without adding extra Taylor terms?
  - Recognizing symmetries in the function ($\sin(x)=\sin(\pi-x)$) we could restrict the range to $[-\pi/2,\pi/2]$, and compute the values in $[-\pi, -\pi/2]$ and $[\pi/2, \pi]$ by symmetry. For a small extra cost of a few checks at the begin of the function, we would reduce the maximum error by $(1/2)^9$

```
##     mySin5( x )
## Approximation of Sin( x ) on [-pi,pi] by Taylor expansion using symmetry
function sinx = mySin5( x )
   pi     = 3.14159265358979;
   halfPi = 1.5707963267949;
   if ( x > halfPi )
          sinx = mySin1( pi - x );
   elseif ( x < -halfPi )
          sinx = -mySin1( pi + x );
   else
          sinx = mySin1( x );
   end
```

Warning! This is not implemented in vectorial form!

Error of mySin1 vs mySin5



Error in log scale in $[-\pi,+\pi]$ of *mySin1* (blue) vs *mySin5* (green)

```
>> x=[-pi():pi()/400:pi()]; y=zeros(size(x));
>> for i=1:length(x), y(i)=mySin5(x(i));endfor   # we do a loop because
mySin5 is not vectorial
>> semilogy(x, abs(mySin1(x)-sin(x)), x, abs(y-sin(x)));
```

Designing an Algorithm

- sin(x): algorithm 2
  - The user specifies how many terms *n*
  - Execution time is constant for same *n*, but increasing in *n*
  - Accuracy improves with *n* and can be controlled. Once *n* is fixed the error depends on the value of *x*
  - Slower than algorithm 1 even for n=4: **flexibility comes at a cost!**

```
function sinx = mySin2( x, nTerms )
   assert ( nTerms >= 1, "nTerms needs to be greater than 1" );

   x2 = -x * x;
   aux = x;        # we use aux in the loop to reduce num ops
   sinx = x;

   for i = 2:nTerms
        n = 2 * i - 1;
        aux *=  x2 / (( n - 1 ) * n); # term of order 2i-1
        sinx += aux;  # addition of the term to the series
   end
```

note the trick with the accumulation variable aux:

$$\frac{x^5}{5!} = \frac{x^3}{3!} \cdot \frac{-\left(x^2\right)}{4 \cdot 5}$$

Adaptive Algorithms

- Here for example we can control precision

```
function sinx = mySin4( x, eps )

    x2 = x * x;
    aux = x;
    sinx = x;
    n = 1;

    do
        n += 2;
        aux *=  (-x2) / ( n - 1 ) / n; ## term of order n
        sinx += aux; ## addition of the term to the series
    until abs( aux ) < eps ## stop when conctribution less than eps
```

Adaptive Algorithms

- Or we go for maximum possible precision

```
function sinx = mySin3( x )

    x2 = x * x;
    aux = x;
    sinx = x;
    n = 1;

    do
        newsinx = sinx;
        n += 2;
        aux *=  (-x2) / ( n - 1 ) / n; ## term of order n
        sinx += aux; ## addition of the term to the series
    until newsinx == sinx ## stop when term computed is less than machine epsilon
```

- Here we could use Kahan summation

Adaptive Algorithms

- Adaptive algorithms self-adapt the input size *n* to achieve a desired level of accuracy
- Let's consider a root-search algorithm, which finds a root of f(*x*) in the range [a,b], based on progressive restrictions of the input range based on some clever rule (e.g. bisection)
- The parameter *n* is the number of iterations done by the algorithm, i.e. the number of evaluations of the function f(*x*)
- Normally accuracy is specified as the size of the final range around the root, therefore *n* is self determined accordingly by the algorithm

Exit Criteria

- Fixing maximum number of iterations (or recursion)
- Fixing the accuracy in relative terms
- Fixing the accuracy in absolute terms
- Combination of above

- Need also to detect unfeasibility of a problem or failures of the algorithm

Computation Cost

- Computation cost is usually measured in term of number of floating point operations (*flop*) required to produce the result
  - In *mySin1*: 8 flop
  - In *mySin2*: 1+3(n-1) flop
    - I am not counting integer instructions, as usually they get executed in parallel with floating point instructions
    - I am not counting *if-then-else*, and *for-next*, although they do have a cost
    - In mySin2, increasing *n* the computation cost increases

Computation cost

- Computation cost can also be measured as number of CPU clock cycles
- This is easy to measure, as there are programs who can do this for us (**profilers**), thus helping us to identify bottlenecks and improve the algorithm
- To give a rough idea of relative cost of various operations:
  - mul,div,add,sub: 1 cycle
  - exp: 30 cycles
  - pow: 150 cycles
  - heap memory allocation: 1000 cycles

  Note these things tend to change with hardware and algorithms improvements, in a couple of years this info may be totally obsolete

Computation Cost

- In general, let $T(n)$ be the running time required by an algorithm measured somehow (flops, cpu cycles, …), where n is some measure of the input size of the algorithm
- The order of growth of $T(n)$ tells us how fast $T(n)$ grows in the limit of large n. You could have an order of growth of n, n^2, n^3, $\log(n)$, $n*\log(n)$, …

Computation Cost

- The **order of growth** is labeled as O(g(*n*)) if exists constant *c* and *d* such that

$$c < \lim_{n \to \infty} \frac{T(n)}{G(n)} < d$$

- For instance if T(*n*) = $3n^3 + 2n^2$, we can say the order of growth is O($3n^3$), or more simply O(n^3)
- For mySin2 order of growth was O(3*n*)
- A good link:

  http://en.wikipedia.org/wiki/Time_complexity

Empirical Performance Measurement

- To measure the performance of a function, we can simply measure how long it takes to run
- A simple measurement:

```
tic; # returns elapsed time since epoch in sec
x=fun(…);  # call function to measure
elapsedTime = toc; # elapsed time in seconds
```

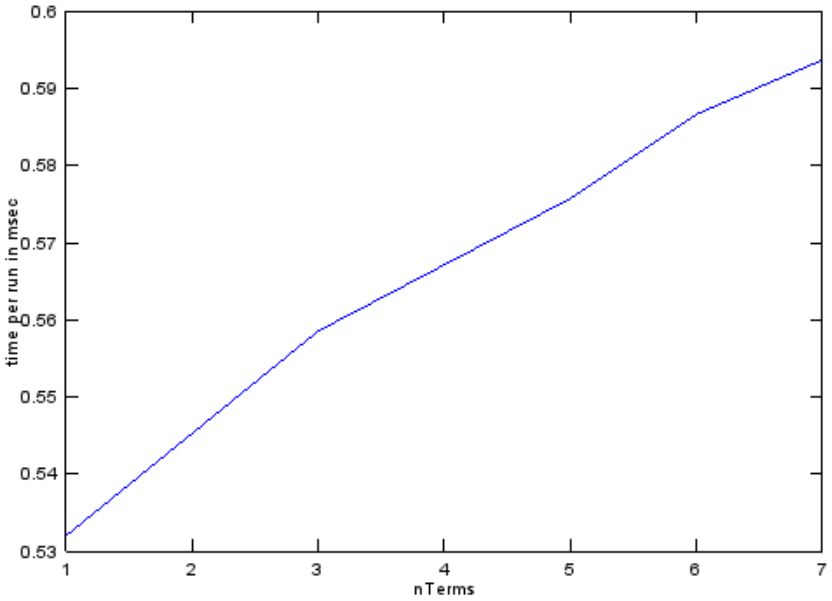
Empirical Performance Measurement

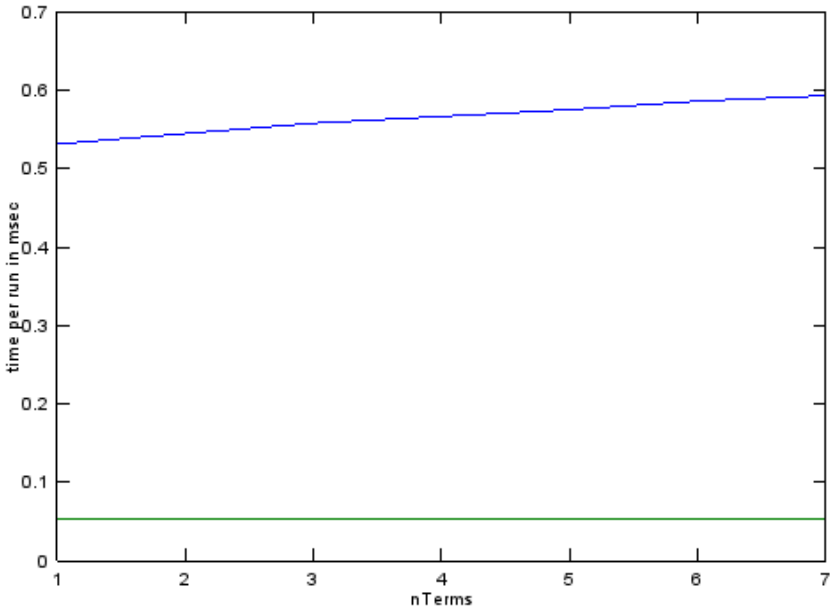
- What are the issues with this approach?
  - The function *time*() has some execution time itself, thus it affects measurements: t_2-t_1=t_{fun}+t_{time}
  - The function *time*() has a certain maximum resolution. If *fun* is faster than that, then it will be t_2==t_1
  - *fun*() may have some one time only initialization costs (e.g. loading a DLL). We would like to exclude this from the measurement
  - Execution time is not completely deterministic (may depend on what else the machine is doing). We need a statistical experiment.

Empirical Performance Measurement

```
function dt=measureTime(fun, nRepeat, dryRun, args)
    # dry run: one-off init costs (e.g. loading DLL)
    if (dryRun), fun(args); endif

    tic
    # total cost of the loop is nRepeat*costOf(fun)
    # if nRepeat is large enough, this will dominate
    # on the cost of time() and will be material with
    # respect to the accuracy ot time()
    for i=1:nRepeat,
        fun(args);
    end
    dt=toc/nRepeat;
```

Comparing mySin1 and mySin2






```
>> function fun1(n), mySin1(2); endfunction
>> function fun2(n), mySin2(2,n); endfunction
>> t1=measureTime(@fun1,20000,1,[]) * 1000;  # multiply by 1000 (milliseconds)
>> t2=[];for n=1:7, t2(n)=measureTime(@fun2,20000,1,n) * 1000; endfor # multiply by 1000
```

- Chart1: *mySin2* increase linearly with nTerms
- Chart2: *mySin2* is much slower than *mySin1*, more than we expected

Different Computation Cost Measurements

- Other considerations:
  - Wall Clock: the time elapsed for a human users
  - CPU Time: the actual CPU time used (in case of multiple threads, include the sum of time used by all threads)

Order of Convergence

- Suppose x_n is the output of an algorithm when the input size is n. Suppose the sequence of x_n has limit X for $n \to \infty$

$$\lim_{n \to \infty} x_n = X$$

- The **order of convergence** is said to be O(g(n)) if there exist a number n_0 and a positive constant c such that:

$$|x_n - X| < c\, g(n), \quad \text{for all } n > n_0$$

- For example, you can have an order of convergence of n^{-1}, n^{-2}, $1/\log n$
- If X is not the theoretically correct value, then the algorithm is affected by **systematic error (bias)**

Heron Source Code

```
function [r,h]=heron(x)
    assert( x>=0, 'x must be positive' );
    rold = 0;
    r = 1;                          # arbitrary initialization
    h = [0,r];                      # just for display
    i = 1;
    while ( r != rold )             # maximum precision
  achieved?
        rold=r;
        r = 0.5 * ( r + x/r );      # heron step
        h=[h; [ i, r] ];            # just for display
        i=i+1;
    end
```

Heron Convergence Rate

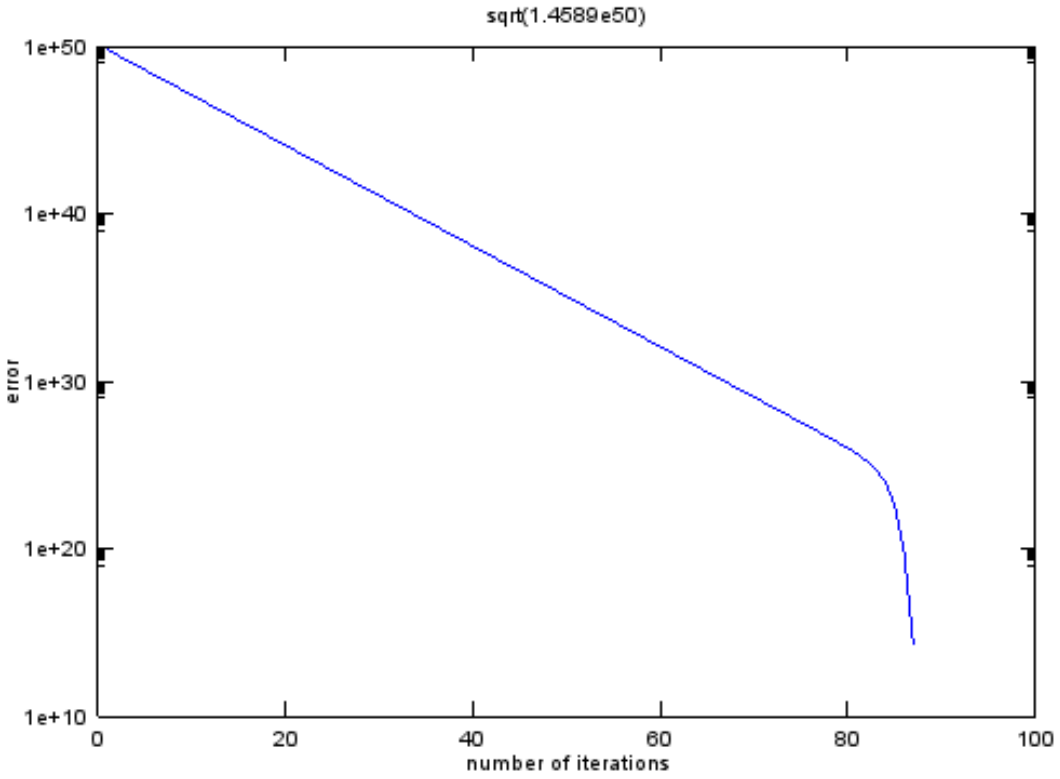
```
octave:9> [root,trace]=heron(2.0)
root =  1.41421356237309
trace =
   iter   approximation
   0      1.000000000000000  (initial guess)
   1      1.500000000000000
   2      1.416666666666667  (gain 2 digits)
   3      1.414215686274510  (gain 3 digits)
   4      1.414213562374690  (gain 5 digits)
   5      1.414213562373095  (achieved maximum accuracy)
   6      1.414213562373095
```

- The digit in red is the first incorrect digit, in green the new digits accuracy gained
- The number of accurate digits we gain doubles at every iteration

Heron Convergence Rate

On a large number (1.4589e+50), it takes more than 80 iterations

Note that, when we get close to the solution, convergence rate increases dramatically




```
>> [root,trace]=heron(1.4589e+50);
>> trace(:,3)=trace(:,2)-trace(end,2);
>> semilogy(trace(:,1),trace(:,3)), title('sqrt(1.4589e50)')
>> ylabel('error'), xlabel('number of iterations')
```

Improving Heron Method

- We could show that if y is in [1/2, 2], then, in the worst case, 5 iterations are sufficient to find sqrt(y) with the maximum possible accuracy in *double* precision
- This suggests the transformation:

  $y = 2^{2k} z$ ➔ $\text{sqrt}(y) = 2^k \text{sqrt}(z)$
- This requires knowledge of k, but remember that k is known straight away (at no extra cost) from the binary representation of a *double*: $y = 1.xxx\ 2^p$
  - if p is even, $k=p/2$ and z=1.xxx
  - otherwise, $k=(p+1)/2$ and z=1.xxx / 2
- So we can compute sqrt(z) instead, then add k to the exponent of the result, which is just an integer operation (virtually no extra cost)

Improved Heron Algorithm

```
function [r,h]=heron2(x)
    assert( x>=0, 'x must be positive' );
    [z,p] = log2( x );     # mantissa in [0.5,1]
    if mod(p,2)  # is the exponent odd?
       z=z*2;
       p=p-1;
    end
    h = [];              # h is just for display
    r = 1;               # arbitrary initialization
    for i=1:5            # exactly five iterations
       r = 0.5 * ( r + z/r );  # heron step
       h=[h; [ i, r] ];        # h is just for display
    end
    r = pow2( r, p/2 );             # add back exponent
    h(:,2) = pow2( h(:,2), p/2 ); # h is just for display
```

Improved Heron Algorithm

```
>> [root,trace]=heron2(y)
root = 1.20784932835184e+025
trace =
  1  1.23780395012936e+025
  2  1.20821177644620e+025
  3  1.20784938271674e+025
  4  1.20784932835185e+025
  5  1.20784932835184e+025
```

- sqrt is now available as part of SIMD instruction. Does Intel use this in the CPU?
- note that operations of multiplying/dividing by powers of two are virtually cost-free if done via direct manipulations of the binary representation

Comparing Algorithms

- Different algorithms which perform the same task can be compared in terms of order of convergence and computation cost
- Need to choose the input size *n* and the typology of the algorithm so that the desired accuracy is achieved with the minimum possible computation cost

Robustness and Limitations

- Robustness is the ability of an algorithm to return reasonable results for any value of the input arguments. I.e. an algorithm is robust if it does not blow up in correspondence of some particular inputs
- An example is the finite difference calculation of derivatives we did before, which returns non-sense result if the increment is too small
- Usually all algorithms have limitations, which needs to be documented and understood

Numerical Disasters

- Here are some real life examples of what can happen when numerical algorithms are not correctly applied.
  - Ariane 5 rocket. [June 4, 1996]
    - 10 year, $7 billion ESA project exploded after launch.
    - 64-bit float converted to 16 bit signed int.
    - Unanticipated overflow.
  - Vancouver stock exchange. [November, 1983]
    - Index undervalued by 44%.
    - Recalculated index after each trade by adding change in price.
    - 22 months of accumulated truncation error.
  - Patriot missile accident. [February 25, 1991]
    - Failed to track scud; hit Army barracks, killed 28.
    - Inaccuracy in measuring time in 1/20 of a second
    - since using 24 bit binary floating point.
  - The sinking of the Sleipner offshore platform [August 23, 1991]
    - loss of nearly one billion dollars.
    - Inaccurate finite element analysis.

Further Readings

- Computer Arithmetic
  - http://en.wikipedia.org/wiki/Single_precision_floating-point_format
  - http://en.wikipedia.org/wiki/IEEE_754-1985
  - http://www.math.okstate.edu/~binegar/4513-F98/4513-l03.pdf
- Unavoidable Errors in Computing
  - http://web.cecs.pdx.edu/~gerry/nmm/course/slides/ch05Slides.pdf
- What Every Computer Scientist Should Know About Floating Point Arithmetic
  - http://cr.yp.to/2005-590/goldberg.pdf
- Kahan Summation Algorithm
  - http://en.wikipedia.org/wiki/Compensated_summation
- Babylonian-Heron method (chapter 1)
  - http://people.cs.uchicago.edu/~ridg/newna/nalrs.pdf

Examples of Numerical Methods

- Linear algebra (factorization, linear systems, eigensystems)
- Interpolation
- Quadrature (integration)
- Minimization (linear, quadratic, least squares, non linear)
- Root search
- ODEs
- PDEs

Examples of Numerical Methods

- But, most algorithms solve well scoped problems of general interest, and are highly researched
- Theory is available in books and papers
- Source code is available in the web or in vendor libraries
- Do not reinvent the wheel, always do some research

Some Public Libraries

- BLAS: collection of elementary **dense** linear algebra subroutines (e.g. matrix multiplication) specialized for matrices stored in various formats
- LAPACK: collection of advanced **dense** linear algebra subroutines
- SUPERLU, ARPACK, PARDISO, TAUCS: collections of advanced **sparse** linear algebra subroutines
- www.netlib.org
- GNU Scientific Library
- Boost
- Scilab
- Octave

Some Vendor Libraries

- NAG
- IMSL
- Matlab
- Numerical Recipes in C++

Numerical Analysis Books

- Press, Taukolsky, Flattery, Vetterling, *Numerical Recipes in C++*
- Johnson, Riess, *Numerical Analysis*
- Golub, Van Loan, *Matrix Computations*
- Higham, *Accuracy and Stability of Numerical Algorithms*
- *Lapack User Guide, http://www.netlib.org/lapack/lug/*

Some Implementation Tips

- Try to get it right first, optimize later
- Understand the convergence properties of your algorithm. Even experimentally is good enough.
- Test your algorithm on simple cases for which you know the theoretical results. If it does not work on these, it won't work on others either
- Stress test the input parameters and understand the limitations of the algorithm
- Try and identify properties the results must satisfy (e.g. probability distribution)