Numerical Methods

Floating Point Operations

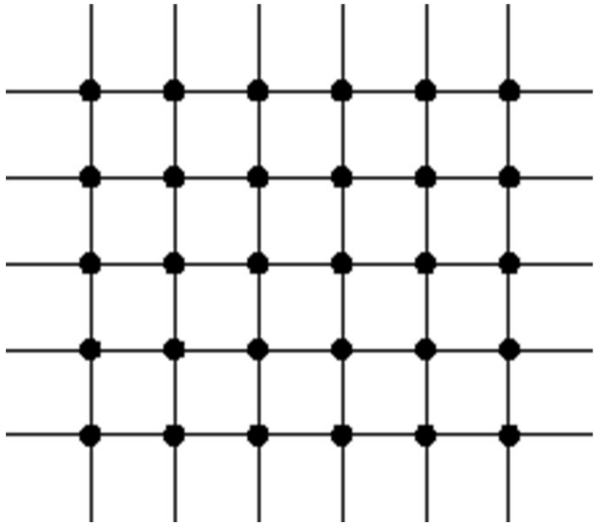
- "The Purpose of Computing is insight, not numbers"
- Richard Hamming

2023-2024

[DΣIM]

Discrete World

- We can not represent all the real numbers with a computer.
- Continuous processes in space or in time must be represented in a ***discrete world***

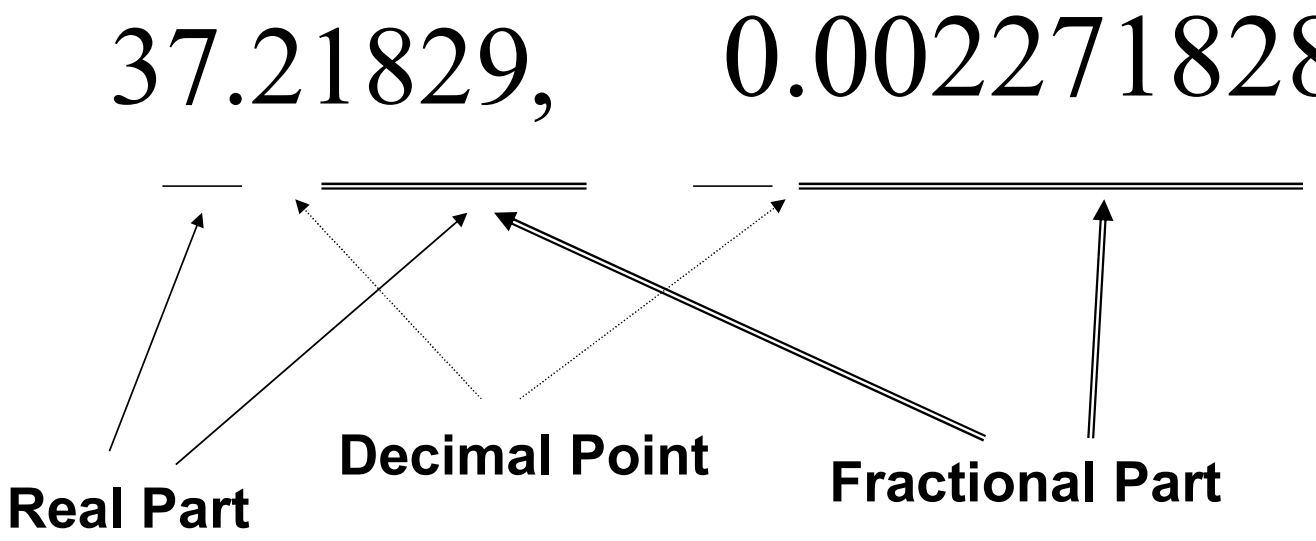


Discrete World

- The hardware of any computer only implements the ***elemental algebraic operations***: +,-,*,/
- Any other mathematical operation must be properly implemented through a ***specific algorithm***. This includes:
  - Any transcendental function: $sin(x)$, $ln(x)$
  - Any limit operation: Derivation, Integration…

Floating Point Numbers

- To describe a real number, we use a real part, a decimal point and a fractional part:



Floating Point Numbers

- But we normally use the ***normalized scientific notation (floating point numbers)***:

$$37.21829 = 0.3721829 \times 10^{2}$$

$$0.002271828 = 0.2271828 \times 10^{-2}$$

Floating Point Numbers

- Using normalized notation each number is represented by certain fraction multiplied by 10^n.

$$x = \pm 0.d_1 d_2 d_3 \cdots \times 10^n$$

- With the additional constraint that ***the first digit in the fractional part must be not zero***

$$d_1 \neq 0$$

Floating Point Numbers

- Thus, in normalized scientific form, a decimal point will be represented as:

$$x = \pm r \times 10^{n} \qquad \frac{1}{10} \le r < 1$$

- r will be called the ***normalized mantissa*** and n the ***exponent***

Floating Point Numbers

- Computers will use the normalized scientific notation in a similar way, buy using a different basis. The ***binary basis-2 system***

$$x = \pm q \times 2^m \qquad \frac{1}{2} \leq q < 1$$

- q will be then a bit sequence b_i (0 or 1) with the additional condition:

$$b_1 = 1$$

Floating Point Numbers

- Using a general basis β, we will write floating point numbers as:

$$x = \pm\left(\sum_{i=1}^{p} d_i \beta^{-i}\right)\beta^{e}$$

- Where the mantissa verifies $1 \leq d_1 \leq \beta - 1;\ 0 \leq d_i \leq \beta - 1$ for $i = 2, \ldots$ The number of digits in the mantissa p is called the ***precision***
- The integer e is the ***exponent*** with $m \leq e \leq M$ and can be also a negative number

[DΣIM]

Floating Point Numbers

- The important point is that any computer has a ***finite word length*** to complete the representation of a real number.
- We can only represent numbers with a ***finite number of digits***. No way to deal with irrational numbers
- Moreover, some real numbers can be too big or too small to be represented within a computer.

Floating Point Numbers

- The finite set of numbers that can be represented in a computer will be called the ***set of machine numbers***

$$x = \pm\left(0.1b_2b_3\ldots b_n\right)_2 \times 2^{\pm e}$$

- Note that $b_1 = 1$. The number is said to be ***normalized.*** The smallest number will be of the form:

$$\left(0.1\right)_2 \times 2^{-e}$$

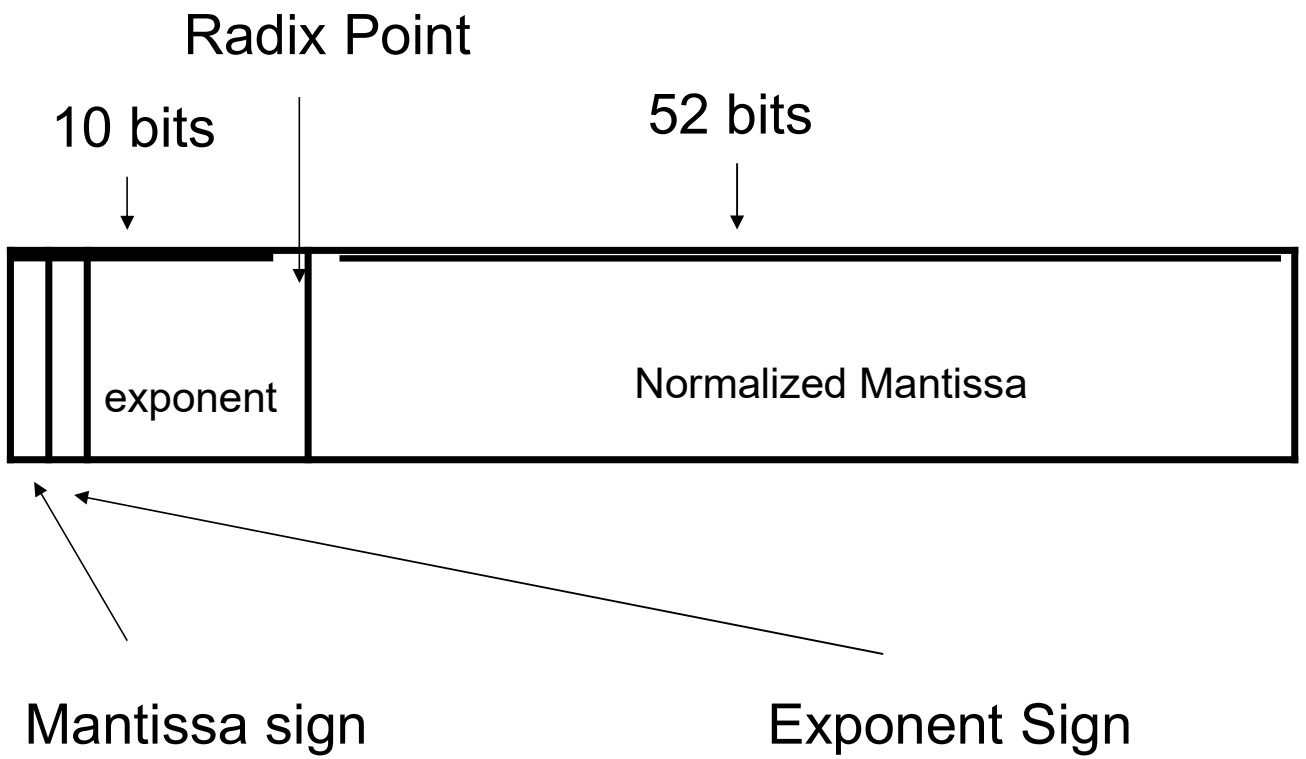
- And thus, there will be always a "hole around zero"

Floating Point Numbers

- The detailed representation of floating-point numbers will depend on the machine architecture.
- Fixing the ideas, suppose that we deal with 8 Bytes or 64 bits and:

Sign of q	1-0 bit
Sign of m	1 bit
Integer $\|m\|$	52 bits
Integer q	11-10 bits

Floating Point Numbers



Floating Point Numbers

- After 1985 all the computers use the standard IEEE for floating point arithmetic. This does not mean that all architectures are equal as there is some freedom margin in the designs.
- The mantissa must be represented in a maximum of 52 bits, i.e.:

$$0 \leq m < 2^{52}$$

- While we use a bit for the number sign and 11 bits for the exponent.

Floating Point Numbers

- With these constraints, the maximum number of ordinary exponents represented is:

$$e_{\max} = 2^{11} - 2 = 2046$$

- While the sign of the exponent is accommodated by storing the value $e + 1023$. The real value of the exponent is obtained subtracting 1023. This gives values for the exponent within the range of values:

$$-1022 \leq e < 1023$$

Overflow and Underflow

- Two additional exponents are used to signal special situations. These are denoted $e_{max} > 0$ and $e_{min} < 0$. There are a total of $e_{max} - e_{min} + 1$ exponents, including a zero exponent.
- During computing operations, we can get numbers larger than $2 \times 2^{e_{max}}$ or smaller than $1.0 \times 2^{e_{min}}$. We use the terms ***overflow*** and ***underflow*** for these numbers.
- Normally, symbols ***Inf*** and ***NaN*** (Not a Number) are used to describe these two situations.

Machine Epsilon

- On the other hand, the smallest number that can be represented by the mantissa of our machine is:

$$eps = 2^{-52} \approx 2.2204 \times 10^{-16}$$

- The finite value of m will introduce a ***limit in the precision*** of arithmetic floating-point operations, while the finite value e will introduce a ***limit in the range*** of the numbers.
- Two consecutive machine numbers, with the same exponent will have mantissas separated with this difference, known as the *machine epsilon*.

Floating Point Numbers

- Since floating point numbers are always normalized, the most significant bit of the mantissa is always 1 when using base 2, and thus this bit does not need to be stored. It is known as the ***hidden bit.***
- Using this trick, the mantissa of 1 is entirely zero such as the mantissa of 0. This requires a special convention to distinguish 0 from 1. The zero is represented by a mantissa of 0 and an exponent $e_{min} - 1$

Floating Point Numbers

- We will use a limited set of numbers, characterized by the basis β, the number of significant digits p and the range (L, U) of the exponent, with $L < 0$ and $U > 0$.
- This set is represented by

$$\mathbb{F}(\beta, p, L, U)$$

- In MATLAB we have:

$$\mathbb{F} = \mathbb{F}(2, 53, -1021, 1024)$$

- For a *format long* we get 15 digits in decimal representation.

Numerical Methods

Floating Point Numbers

T01_Ex01_FPNumberDistribution.m

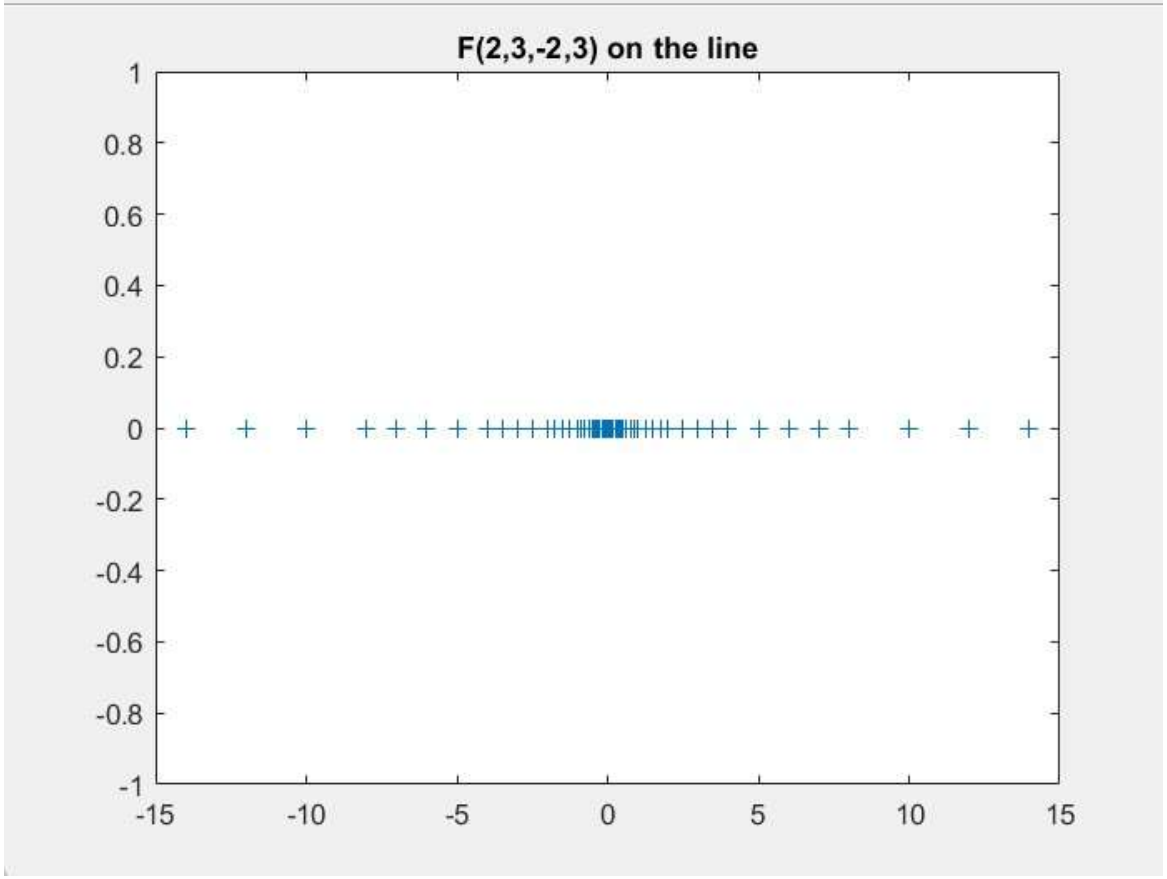
```
% Example
% Distribution of floating point numbers on the line of
% the number set F(b,p,L,U) system
% Set the example to (2,3,-2,3)

% Generate all positive numbers of the system (2,3,-2,3)
% Use a decimal mantissa.
x = [];
for i = 0.25:0.25:1.75
    for j = -2:3
        x = [x i*2^j];
    end
end
x=[x -x 0]; % Add all negative numbers and 0
x = sort(x); % Sort
y = zeros(1,length(x));

plot(x,y,'+')
title('F(2,3,-2,3) on the line')

```

2023-2024

[DΣIM]

Numerical Methods

Floating Point Numbers



2023-2024

[DΣIM]

Floating Point Numbers

- Thus, the numbers that we will use will be of the form:

$$x = (-1)^s \cdot \left(0.a_1 a_2 \ldots a_p\right) \cdot \beta^e = (-1)^s \cdot m \cdot \beta^{e-p}, \quad a_1 \neq 0$$

- They will be represented in registers of 8 Bytes. The sign s will take a bit, the exponent e in 11 bits, and the mantissa m in 52 bits. We will store only the digits a_2 ... a_{53}

Floating Point Operations

- Consider the following simple arithmetic operation:

```
>> t = 1.0

t =

     1

>> a = t - 0.2 - 0.2 - 0.2 - 0.2 - 0.2

a =

   5.5511e-17

>>
```

Floating Point Operations

- The explanation of the former result is in the fact that the computer works with basis-2 and the floating-point number $t = 0.1$ is not represented as 1/10 but as this infinite series:

$$\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \cdots$$

- And the computer will never be able to store all the mantissa digits

Computer Errors

- Suppose that we use a mantissa with p digits. Some real numbers can not be represented with this limited word structure (irrespective of the basis used):

$$x = (0.1 d_1 d_2 \ldots d_p d_{p+1} d_{p+2} \ldots)_2 \times 2^e$$

- We are forced to discard some of the digits in the mantissa, starting at the p position:

$$d_{p+1} d_{p+2} \ldots$$

Computer Errors

- We can do this in two ways. By ***cutting off or chopping*:**

$$m_c = \left(0.1 d_2 d_3 \ldots d_p\right)_2$$

- Or ***rounding off***. In this case the mantissa is obtained as follows:

$$m_r = \begin{cases} 2^{-p} \left\lfloor 2^p m + \frac{1}{2} \right\rfloor & m > 0 \\ 2^{-p} \left\lceil 2^p m - \frac{1}{2} \right\rceil & m < 0 \end{cases}$$

Computer Errors

- The function $\lfloor x \rfloor$ is the ***floor function*** and represents the largest integer less than or equal to x. The function $\lceil x \rceil$ is the ***ceiling function*** and is the smallest integer greater than or equal to x.
- For positive decimal numbers, the former relation express the familiar rule of adding 1 in the $(p+1)$ position in the case that the first discarded digit is lower or equal to 5 (in the decimal representation).

Computer Errors

T01_Ex02_RoundingErrors.m

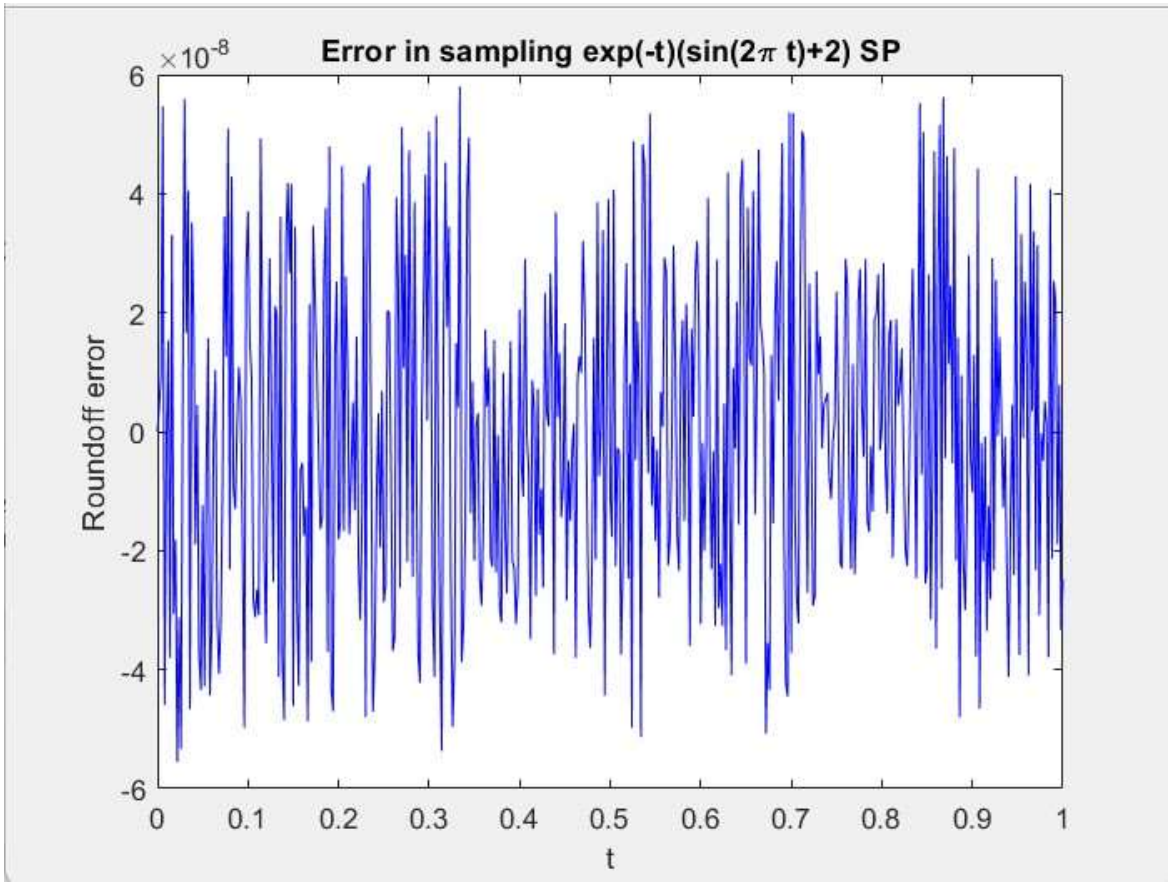
```
% Comparison of Rounding Error using single and double precision
% evaluations

% Evaluate g(t) = exp(-t) * (sin(2pit)+2)

t = 0:.002:1;
tt = exp(-t) .* (sin(2*pi*t)+2);
rt = single(tt);
round_err = (tt - rt) ./tt ;

% Plot the result

plot (t,round_err,'b-');
title ('Error in sampling exp(-t)(sin(2\pi t)+2) SP')
xlabel('t')
ylabel('Roundoff error')

% Relative error should be about eta = eps(single)/2
% The Relative error is about the rounding level.

d = eps('single') / 2;
rel_round_err =  max(abs(round_err))/ d;

fprintf(' Rounding Level      : %12.10f \n', d)
fprintf(' Max. Relative Error : %12.10f \n', max(abs(round_err)))
fprintf(' Ratio               : %12.10f \n', rel_round_err)

```

```
>> T01_Ex02_RoundingErrors
 Rounding Level      : 0.0000000596
 Max. Relative Error : 0.0000000580
 Ratio               : 0.9737441540
>>
```


[DΣIM]

Numerical Methods

Computer Errors



2023-2024

[DΣIM]

Absolute Error

- The ***absolute error*** will be defined as:

$$\xi_a = \left| x - x^* \right|$$

- If we use the chopping or truncation of numbers, we have:

$$\xi_a = \left| x - x^* \right| \leq \left| m - m_c \right| 2^e \leq 2^{e-p-1}$$

- The absolute error will depend on the magnitude of the number (through e)

Absolute Error

- In the case of rounding, we have a similar relation:

$$\xi_a = |x - x'| \le |m - m_r| 2^e \le \frac{1}{2} 2^{-p-1} \cdot 2^e = \frac{1}{2} 2^{e-p-1}$$

- In this case the absolute error still depends on the magnitude of the number, through the exponent e, but the bound is slightly lower.

Relative Error

- The ***relative error*** is defined as:

$$\xi_r = \left| \frac{x - x^*}{x} \right|$$

- In the case of truncation, the relative error is:

$$\left| \frac{x - x^*}{x} \right| \leq \frac{2^{-e-p-1}}{m \times 2^e} \leq \frac{2^{-p-1}}{1/2} = 2^{-p}$$

- And the error does ***not depend anymore on the magnitude*** of this number.

Relative Error

- The absolute error ξ_a and the relative error ξ_r are related. If we write

$$x^* = x + \xi_a = x + \varepsilon$$

- The relative error can be written also as:

$$x^* = x(1 + \xi_r) = x(1 + \delta)$$

- Then

$$\varepsilon = x\delta \quad \text{or, if } x \neq 0, \quad \delta = \frac{\varepsilon}{x}.$$

Floating Point Operations

- When representing floating point numbers, the computer will make a relative representation error which will have as upper limit the value:

$$\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2} eps$$

- Where *fl*(x) is the floating-point representation. The constant *eps* is called the ***machine epsilon***.

Floating Point Operations

- This constant eps is the minimum distance between two floating point numbers. It is also the smallest relative error. This value is predefined in MATLAB in the variable ***eps*** and takes the value $eps = 2^{-e} \approx 2^{-54}$
- The function $eps(x)$ gives the distance between the floating-point value x and the next number

Floating Point Operations

- This constant value can be calculated in any computer using the following algorithm:

T01_Ex03_Machine_eps.m

```
1   % Exemple
2   % Compute the Machine Epsilon in MATLAB
3
4   n = 1;
5   m_eps = 1;
6
7   while 1.0 + m_eps / 2 > 1.0
8       n = n + 1;
9       m_eps = m_eps / 2;
10  end
11
12  fprintf('Number of iterations n = %d\n', n)
13  fprintf('Machine Epsilon = %16.14e \n', m_eps)
14  fprintf('MATLAB eps      = %16.14e \n', eps)
15
```

```
>> T01_Ex03_Machine_eps
Number of iterations n = 53
Machine Epsilon = 2.22044604925031e-16
MATLAB eps      = 2.22044604925031e-16
```

Floating Point Operations

- There are many ways to compute the machine epsilon. Here we have another example:

```
>> format long
>> a = 4/3

a =

   1.333333333333333

>> b = a - 1

b =

   0.333333333333333

>> c = 3*b

c =

   1.000000000000000

>> e = 1 - c

e =

     2.220446049250313e-16

>>
```


[DΣIM]

Arithmetic Operations

- Any elemental arithmetic operation will not improve the precision of your numbers:

$$x \pm y \leq x' \pm y' + \xi_a(x) + \xi_a(y)$$

- Your ***errors will always pile up***. Never disappearing

$$\boxed{\xi_a(x \pm y) = \xi_a(x) \pm \xi_a(y)}$$

Arithmetic Operations

- In the case of the product, we have:

$$\begin{aligned} xy &\le \left(x' + \xi_a(x)\right)\left(y' + \xi_a(y)\right) \\ &= x'y' + x'\xi_a(y) + y'\xi_a(x) + \xi_a(x)\xi_a(y) \\ &\approx x'y' + x'\xi_a(y) + y'\xi_a(x) \end{aligned}$$

- This is:

$$\boxed{\xi_a(xy) = x'\xi_a(y) + y'\xi_a(x)}$$

General Error Formula

- Consider a function of one variable, $y(x)$ and suppose we want to estimate a bound for the magnitude of the error using the mean value theorem:

$$\xi_a(y(x)) = y(x) - y(x') = y(\zeta)(x - x')$$

- Where x' is the approximate value of x and $x < \zeta < x'$. A natural way to approximate this error is to use the differential of $y(x)$

$$\xi_a(y(x)) \approx |y'(x)| \xi_a(x)$$

General Error Formula

- The quantity $|y'(x)|$ is a measure of the sensitivity of $y(x)$ for disturbances in the argument x.
- In the more general case where we want to estimate the value of $y(x_1, x_2, \ldots, x_n)$ and we have the approximate values of the variables, $x' = (x'_1, x'_2, \ldots, x'_n)$, we have a similar general error propagation formula:

$$\xi_a(x_1, \ldots, x_n) \leq \sum_{i=1}^{n} \left| \frac{\partial y(x')}{\partial x_i} \right| |\xi_a(x_i)|$$

Loss of Significance

- The limited precision implies that we must pay special attention if we want to make good programs.
- A very dangerous error appears when subtracting similar numbers. The relative error can be huge. This kind of error is called ***loss of significance or catastrophic cancellation***

Loss of Significance

- In the case of subtraction, the relative error can be computed as:

$$\xi_r = \frac{\xi_a(x) + \xi_a(y)}{|x - y|}$$

- Note that if x and y are very similar, we can obtain a huge relative error

Loss of Significance

- A simple calculation will show the effect of this error. Take the following subtraction:

$$\begin{array}{r} 0.12345678 \\ -0.12344444 \\ \hline 0.00001234 \end{array}$$

- Then, you get the floating-point number 0.1234×10^{-4}.

Loss of Significance

- If we use the floating-point representation and the two values are rounded. The relative error in the representation of the two first numbers will then be $\varepsilon = \frac{1}{2} \times 10^{-8}$, while the relative error in the result will be $\varepsilon = \frac{1}{2} \times 10^{-4}$, i.e., **10.000 times worst. !**

Loss of Significance

- Consider for instance the expression

$$y = x - \sin(x)$$

- Where:

$$x = 0.6666666667 \times 10^{-1}$$
$$\sin(x) = 0.6661729492 \times 10^{-1}$$
$$x - \sin(x) = 0.0004937175 \times 10^{-1}$$
$$= 0.4937175000 \times 10^{-4}$$

- We have lost three significant digits, i.e., a relative error of 10^3

Loss of Significance

- To evaluate the expression

$$y(x) = x - \sin(x)$$

- Is a clever solution to use the Taylor series for *sin*(*x*). Thus

$$f(x) \approx x - \left(x - \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \cdots \right) \approx \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \cdots$$

Loss of Significance

- If we want to avoid this problem, we are be forced to make changes in the mathematical expression used. For instance, consider:

$$f(x) = \sqrt{x^2 + 1} - 1$$

- If x is a near zero, then is much better to use instead:

$$f(x) = \left(\sqrt{x^2 + 1} - 1\right) \cdot \left(\frac{\sqrt{x^2 + 1} + 1}{\sqrt{x^2 + 1} + 1}\right) = \frac{x^2}{\sqrt{x^2 + 1} + 1}$$

Loss of Significance

- With this program we can explore the effect of loss of significance:

T01_Ex04_LossSignificance.m

```
% Loss of Significance

clear
f1=@(x)sqrt(x)*(sqrt(x+1)-sqrt(x));
f2=@(x)sqrt(x)./(sqrt(x+1)+sqrt(x));

x=1;
format long e
for k=1:17
    fprintf('At x=%17.0f, f1(x)=%20.18f, f2(x)=%20.18f\n', x,f1(x),f2(x));
    x = 10*x;
end

% Last Value
sx1=sqrt(x+1); sx=sqrt(x);
d=sx1-sx; s=sx1+sx;
fprintf('sqrt(x+1)=%25.13f, sqrt(x)=%25.13f \n',sx1,sx);
fprintf(' diff=%25.23f, sum=%25.23f \n\n',d,s);
```

Loss of Significance

- This program will compute the two equivalent functions:

$$f_1(x) = \sqrt{x}\left(\sqrt{x+1} - \sqrt{x}\right) \quad \text{i} \quad f_2(x) = \frac{\sqrt{x}}{\sqrt{x+1} + \sqrt{x}}$$

- For different values of x. When $x=10^{15}$ we have that:

$$\sqrt{x+1} \approx 3.162277660168381 \times 10^7 = 31622776.60168381$$

$$\sqrt{x} \approx 3.162277660168379 \times 10^7 = 31622776.60168379$$

Loss of Significance

- And we obtain the following results:

```
>> T01_Ex04_LossSignificance
At x=                1, f1(x)=0.414213562373095145, f2(x)=0.414213562373095090
At x=               10, f1(x)=0.488088481701514754, f2(x)=0.488088481701515475
At x=              100, f1(x)=0.498756211208899458, f2(x)=0.498756211208902733
At x=             1000, f1(x)=0.499875062461021868, f2(x)=0.499875062460964859
At x=            10000, f1(x)=0.499987500624854420, f2(x)=0.499987500624960890
At x=           100000, f1(x)=0.499998750005928860, f2(x)=0.499998750006249937
At x=          1000000, f1(x)=0.499999875046341913, f2(x)=0.499999875000062488
At x=         10000000, f1(x)=0.499999987401150925, f2(x)=0.499999987500000576
At x=        100000000, f1(x)=0.500000005558831617, f2(x)=0.499999998749999952
At x=       1000000000, f1(x)=0.500000077997506343, f2(x)=0.499999999874999990
At x=      10000000000, f1(x)=0.499999441672116518, f2(x)=0.499999999987500054
At x=     100000000000, f1(x)=0.500004449631168080, f2(x)=0.499999999998750000
At x=    1000000000000, f1(x)=0.500003807246685028, f2(x)=0.499999999999874989
At x=   10000000000000, f1(x)=0.499194546973835973, f2(x)=0.499999999999987510
At x=  100000000000000, f1(x)=0.502914190292358398, f2(x)=0.499999999999998723
At x= 1000000000000000, f1(x)=0.589020114423405183, f2(x)=0.499999999999999833
At x=10000000000000000, f1(x)=0.000000000000000000, f2(x)=0.500000000000000000
sqrt(x+1)=  316227766.0168379545212, sqrt(x)=  316227766.0168379545212
 diff=0.000000000000000000000000, sum=632455532.03367590904235839843750
```

Loss of Significance

- Another example can be found when evaluating polynomials like

$$p(x) = (x-1)^7$$

- Taking the x values within [0.988, 1.012] and developing the polynomial expression to obtain:

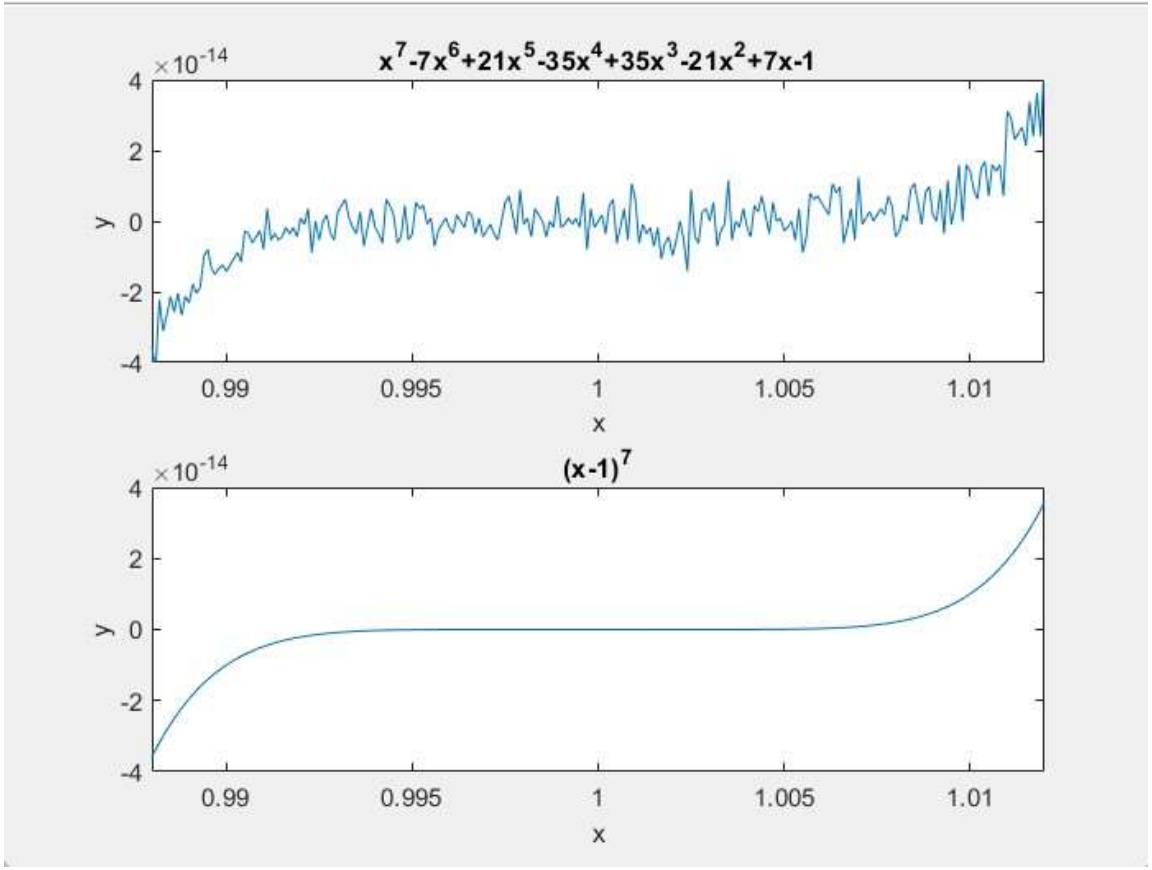
$$p(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$$

Loss of Significance

T01_Ex05_LossSignificanceII_Pol.m

```
% Loss of Significance in the computation of a polynomial
clear
f1 = @(x) x.^7 - 7*x.^6 + 21*x.^5 - 35*x.^4 + 35*x.^3 - 21*x.^2 + 7*x - 1;
f2 = @(x) (x - 1).^7;

% Set the range for x
xp=0.988:0.0001:1.012;

y1 = f1(xp);
y2 = f2(xp);

% Plot the results
subplot(2,1,1)
plot(xp,y1)
xlabel('x');
ylabel('y');
title('x^7-7x^6+21x^5-35x^4+35x^3-21x^2+7x-1')
axis([0.988 1.012 -4e-14 4e-14])

subplot(2,1,2)
plot(xp,y2)
xlabel('x');
ylabel('y');
title('(x-1)^7')
axis([0.988 1.012 -4e-14 4e-14])
```

Loss of Significance



[DΣIM]

Transcendental Functions

- The last example introduces a new problem. A computer will not be able to evaluate directly expressions like:

$$\ln(2)$$

- Transcendental functions are those functions that can not be evaluated using elementary arithmetic operations.

Transcendental Functions

- We can use a polynomial to approximate this kind of functions, thanks to:

- ***Weierstrass Approximation Theorem:*** *If f is defined and it is continuous within the interval* $[a,b]$, *given any* $\varepsilon > 0$, *there is some polynomial P, defined within* $[a,b]$, *for which*

$$\left| f(x) - P(x) \right| < \varepsilon$$

for any

$$x \in [a,b]$$

Taylor's Polynomial

- ***Taylor's Theorem:*** *If* $f \in \mathbb{C}^n[a,b]$ *and if* $f^{(n+1)}$ *exists within the interval* (a,b)*, then, for any point* c *and* x *in* $[a,b]$ *there is some point* ξ *between* c *and* x *for which*

$$f(x) = \sum_{k=0}^{n} \frac{1}{k!} f^{(k)}(c)(x-c)^k + E_n(x),$$

where:

$$E_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x-c)^{n+1}.$$

Taylor's Polynomial

- In this way we can write

$$\ln(x) = (x-1) - \frac{1}{2}(x-1)^2 + \frac{1}{3}(x-1)^3 - \cdots$$
$$+ (-1)^{n-1}\frac{1}{n}(x-1)^n + E_n(x)$$

- This expression can be used in any computer as we are using only arithmetic operations.

Taylor's Polynomial

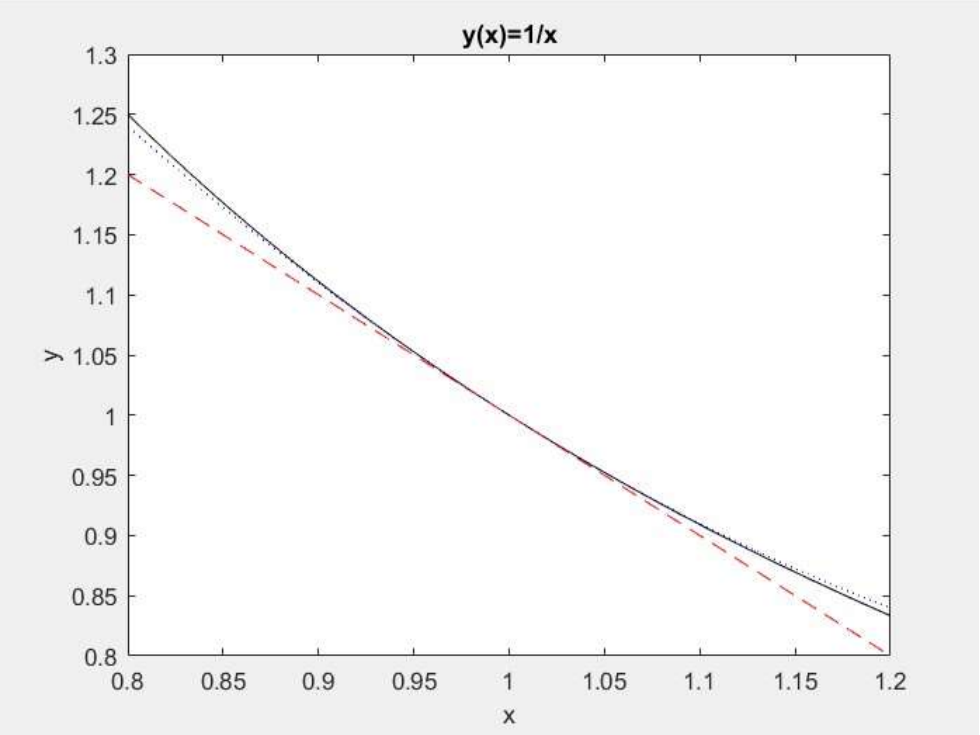
- Suppose that we want to calculate $\ln(2)$ with a precision of 8 digits. Then, it must be true that:

$$\left|E_n(2)\right| = \frac{1}{n+1}\xi^{-(n+1)}(2-1)^{n+1} \leq \frac{1}{n+1} < 10^{-8}$$

- Thus $n+1 \geq 108$, and we will need more than a hundred million terms

Taylor's Polynomial

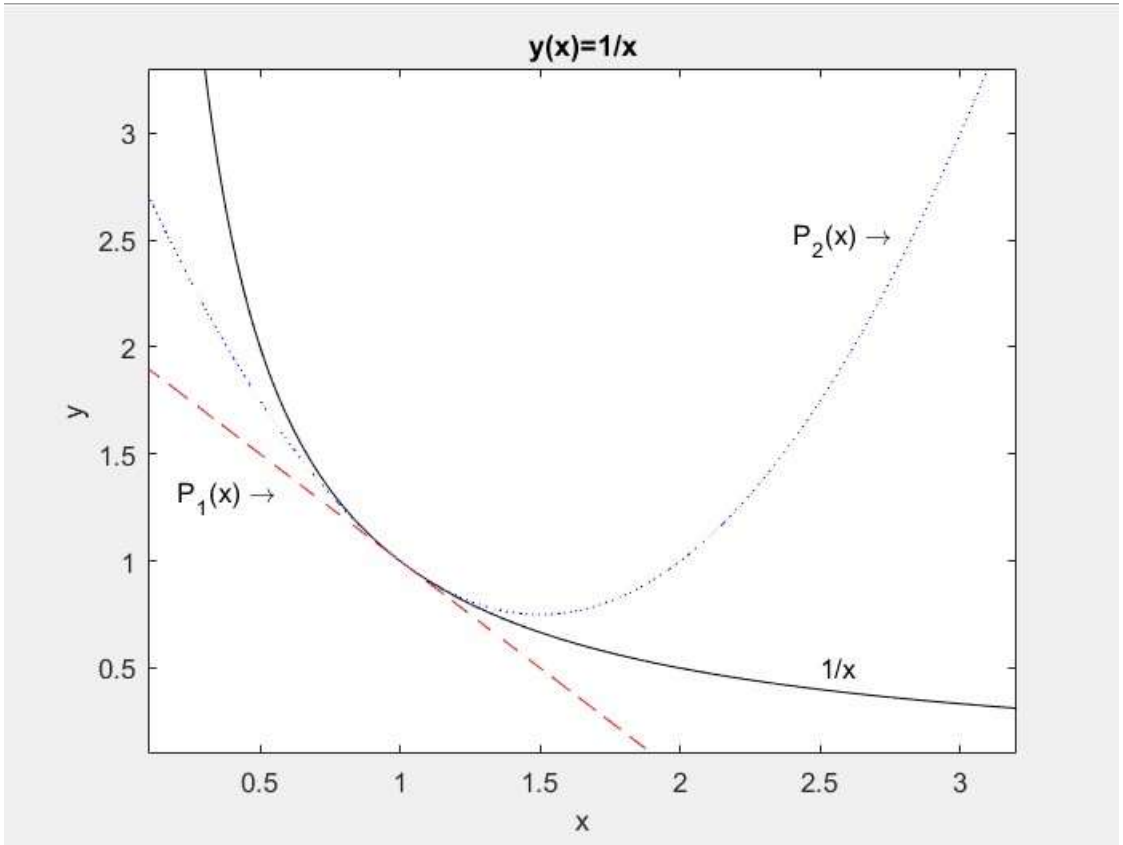
- The Taylor Polynomial is just a ***local*** approximation to a function



[DΣIM]

Taylor's Polynomial

- The Taylor polynomial will not be useful outside the local range



[DIMI]

Unstable Problems (Ill-conditioning)

- Consider the system of two linear equation:

$$x + 2y = 3$$
$$0.499x + 1.001y = 1.5$$

- Whose solution is $x = y = 1.0$. If the second equation is replaced by:

$$0.5x + 1.001y = 1.5$$

- Then the solution becomes $x = 3, y = 0$.

Unstable Problems (Ill-conditioning)

- A mathematical problem in which the solution is very sensitive to changes in the data is said to be ***unstable*** or ***ill-conditioned problem***.
- Conversely, if the problem is such that small changes in the data cause equally small changes in the solution, then the problem is called ***stable*** or ***well-conditioned***.

Condition Numbers

- We need a measure of how a perturbation in the input data are magnified by some numerical algorithm. Namely, the sensitivity of the solution to perturbations in the data, which we call the ***condition of the problem***. This relationship will be problem dependent
- The condition of the problem is measured by the ***condition number***, which can be defined in relative or absolute terms and can be evaluated norm-wise or component-wise

Condition Number

- The norm-wise relative condition number κ_{rel} is the maximum of the ratio of the relative change in the solution to the relative change in the input and can be expressed as:

$$\kappa_{rel} = \sup_{x} \frac{\|\delta \mathbf{y}\|}{\|\delta \mathbf{x}\|} = \sup_{x} \frac{\|\mathbf{x}\|}{\|\mathbf{y}\|} \frac{\|\Delta \mathbf{y}\|}{\|\Delta \mathbf{x}\|} \Leftrightarrow \|\delta \mathbf{y}\| \leq \kappa_{rel} \|\delta \mathbf{x}\|$$

- The absolute condition number κ_{abs} can be defined similarly:

$$\kappa_{abs} = \sup_{x} \frac{\|\Delta \mathbf{y}\|}{\|\Delta \mathbf{x}\|} \Leftrightarrow \|\delta \mathbf{y}\| \leq \kappa_{abs} \|\delta \mathbf{x}\|$$

Condition Number

- If κ has moderate size, we say that the problem is well-conditioned. Otherwise, we say that the problem is ill-conditioned. Consequently, even for a very good algorithm, the approximate solution to an ill-conditioned problem may give a result with a large error. This is ***totally independent of the algorithm*** selected to solve the problem.

Condition Number

- For the case of a differentiable scalar function $y = f(x)$, the relative condition number can be expressed using the derivative. As the perturbation can be as small as possible, we have:

$$\lim_{\Delta x \to 0} \frac{\delta x}{\delta y} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \frac{x}{y} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \frac{x}{y}$$

- And so

$$\kappa_{rel} \approx |x| \frac{|f'(x)|}{|f(x)|}$$

- Will provide a sufficiently good measure of the conditioning of a problem for small Δx.

Unstable Problems

- We can compute recursively the value of the Bessel functions to see how the propagation of errors can be catastrophic.
- The value of the Bessel functions $J_n(x)$ when $x = 1$. Can be obtained with the recursive relation:

$$J_{n+1}(x) = 2nx^{-1}J_n(x) - J_{n-1}(x)$$

- Where:

$$J_0(1) \approx 0.7651976865$$
$$J_1(1) \approx 0.4400505857$$

Unstable Problems

- The Bessel functions are defined by the relation

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x\sin\theta - n\theta)d\theta$$

- And it is true that

$$\left|J_n(x)\right| \le \left|\frac{1}{\pi}\int_0^{\pi} 1 d\theta\right| = 1$$

Unstable Problems

- If we use the recurrence:

T01_Ex08_BesselRecurrence.m

```
% Bessel Functions computed through a recurrence

clear
format long
j = zeros(1,21);
j(1) = 0.7651976865;
j(2) = 0.4400505857;

fprintf('j( 1) = %16.14f \n', j(1))
fprintf('j( 2) = %16.14f \n', j(2))

% Compute the Bessel functions using the recurrence

for i = 2:20
    j(i+1) = 2.0*i*j(i) - j(i-1);
    fprintf('j( %d) = %16.14f \n', i, j(i))
end
```

Unstable Problems

- You will get!?!

```
>> T01_Ex08_BesselRecurrence
j( 1) = 0.76519768650000
j( 2) = 0.44005058570000
j( 2) = 0.44005058570000
j( 3) = 0.99500465630000
j( 4) = 5.52997735210000
j( 5) = 43.24481416050001
j( 6) = 426.91816425290006
j( 7) = 5079.77315687430018
j( 8) = 70689.90603198729514
j( 9) = 1125958.72335492237471
j( 10) = 20196567.11435661464930
j( 11) = 402805383.56377732753754
j( 12) = 8841521871.28874397277832
j( 13) = 211793719527.36608886718750
j( 14) = 5497795185840.22949218750000
j( 15) = 153726471483999.06250000000000
j( 16) = 4606296349334132.00000000000000
j( 17) = 147247756707208224.00000000000000
j( 18) = 5001817431695745024.00000000000000
j( 19) = 179918179784339619840.00000000000000
j( 20) = 6831889014373209866240.00000000000000
fx >>
```