1 Preliminaries

- How a computer can be used
  - to solve problems that may not be solvable by hand
  - to solve problems (that you may have solved before) in a different way
- Many of these <u>simplified</u> examples can be solved analytically (by hand)

$$x^3 - x^2 - 3x + 3 = 0, with\ solution\ \sqrt{3}$$

- But most of the examples can not be simplified and can not be solved analytically
- mathematical relationships $\Longrightarrow$ simulate some real word situations

1.1 Analysis vs Numerical Analysis

- In mathematics, solve a problem through equations; **algebra, calculus, differential equations (DE), Partial DE,** ...
- In numerical analysis; four operations (add, substract, multiply, division) and Comparision.
  - These operations are exactly those that computers can do

    $$\int_0^{\pi} \sqrt{1 + cos^2 x} dx$$

    length of one arch of the curve y-sinx; no solution with " a substitution" or "integration by parts"
    numerical analysis can compute the length of this curve by standardized methods that apply to essentially any integrand
- Another difference between a numerical results and analytical answer is that the former is always an approximation
  - this can usually be <u>as accurate as needed</u> (level of accuracy)

1.2 Computers and Numerical Analysis

- Numerical Methods require repetitive arithmetic operations $\Rightarrow$ a computer to carry out
  - also, a human would make so many mistakes
- A computer program must be written so the the computer can do numerical analysis
  - FORTRAN, Pascal, C, C++, Java, ...
  - IMSL (International Mathematical and Statistical Library)
  - LAPACK (Linear Algebra Package)
  - Distributed and Parallel Computing; any idle computers connected over network
  - CAS (Computer Algebra System)
    - Mathematica
    - MATLAB
    - Maple (For all above: if an analytical answer can not be given, answer by numerical methods)
    - This kind of programs mimics the way humans solve mathematical problems
    - ability to perform symbolically
    - ability to carry out numerical procedures with extreme precision
    - good for small problems and learning environment

1.3 An Illustrative Example

What is the longest ladder ($L_1 + L_2$)? (see the Fig. 1)

$$L_1 = \frac{w_1}{Sinb}, L_2 = \frac{w_2}{Sinc}, b = \pi - a - c, L = L_1 + L_2 = \frac{w_1}{sin(\pi - a - c)} + \frac{w_2}{sinc}$$

The maximum length of the ladder$\Rightarrow$ $\frac{dL}{dc}\rfloor_{c=C} = 0 \Rightarrow$ calculus way
MATLAB way is as the following: (see the Fig. 2)

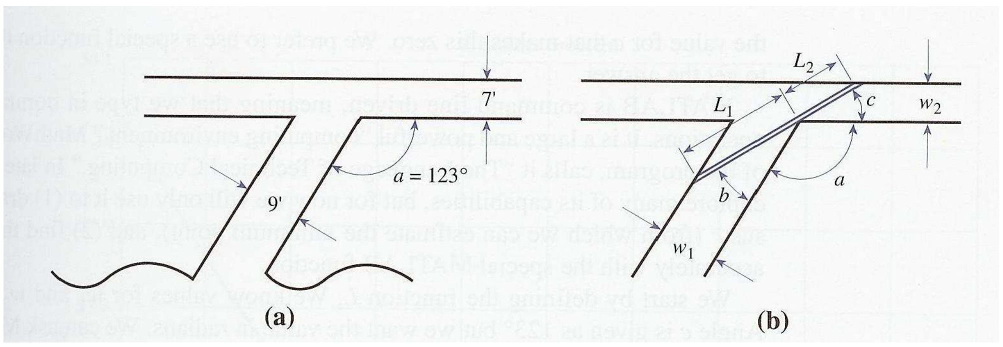
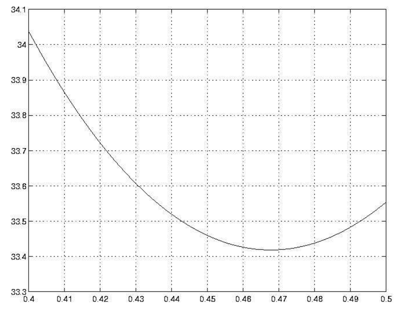


Figure 1: An illustrating example: The ladder in the mine.



```
a=123*2*pi*/360
L=inline('9/sin(pi-2.1468-c)+7/sin(c)')
fplot(L,[0.4,0.5]); grid on
fminbnd(L,0.4,0.5)
L(0.4677)
fminbnd(L,0.4,0.5,optimset('Display','iter'))
```

Figure 2: An illustrating example: The ladder in the mine. Solution with MATLAB

1.4 Some disasters attributable to bad numerical computing

Have you been paying attention in your numerical analysis or scientific computation courses? If not, it could be a costly mistake. Here are some real life examples of what can happen when numerical algorithms are not correctly applied.

- The Patriot Missile failure, in Dharan, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths, is ultimately attributable to poor handling of rounding errors.

- The explosion of the Ariane 5 rocket just after lift-off on its maiden voyage off French Guiana, on June 4, 1996, was ultimately the consequence of a simple overflow.

- The sinking of the Sleipner A offshore platform in Gandsfjorden near Stavanger, Norway, on August 23, 1991, resulted in a loss of nearly one billion dollars. It was found to be the result of inaccurate finite element analysis.

1.5 Kinds of Errors in Numerical Procedures

Errors can occur in doing numerical procedures

```
>> format long e
>> 2.6+0.2
ans =     2.800000000000000e+00
>> ans+0.2
ans =     3.000000000000000e+00
>> ans+0.2
ans =     3.200000000000001e+00
```

Repeated addition of the perfectly reasonable fraction results in the creation of an erroneous digit.

```
>> 2.6+0.6
ans =     3.200000000000000e+00
>> ans+0.6
ans =     3.800000000000000e+00
>> ans+0.6
ans =     4.400000000000000e+00
>> ans+0.6
ans =     5
```

Not only is there no apparent loss of precision, but **MATLAB** displays the final results as 5, as integer. This behaviors are caused by the finite arithmetic used in numerical computations.

Computers use only a fixed number of digits to represent a number. As a result, the numerical values stored in a computer are said to have finite *precision.* Limiting precision has the desirable effects of increasing the speed of numerical calculations and reducing memory required to store numbers. What are the undesirable effects?

Kinds of Errors:

- **Error in original Data**
- **Blunders**: Sometimes a test run with known results is worthwhile, but is no guarantee of freedom from foolish error.
- **Truncation Error**: i.e., approximate e^x by the cubic power

  $$P_3(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}; \qquad e^x = P_3(x) + \sum_{n=4}^{\infty} \frac{x^n}{n!}$$

  approximating e^x with the cubic gives an inexact answer. The error is due to truncating the series.
  When to cut series expansion $\Longrightarrow$ be satisfied with an approximation to the exact analytical answer.

  Unlike roundoff, which is controlled by the hardware and the computer language being used, truncation error is under control of the programmer or user. Truncation error can be reduced by selecting more accurate discrete approximations. It can not be eliminated entirely.
  Example m-file: Evaluating the Series for sin(x) (sinser.m)

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

  An efficient implementation of the series uses recursion to avoid overflow in the evaluation of individual terms. If T_k is the kth term $(k = 1, 3, 5, \dots)$ then

  $$T_k = \frac{x^2}{k(k-1)} T_{k-2}$$

  ```
  >> sinser(pi/6)
  ```

  Study the effect of the parameters *tol* and *nmax* by changing their values.

- **Propagated Error**:
  - more subtle
  - by propagated we mean an error in the succeeding steps of a process due to an Occurrence of an earlier error
  - of critical importance
  - stable numerical methods; errors made at early points die out as the method continues

  - unstable numerical method; does not die out

- **Round-off Error**:

```
>> format long e  %display all available digits
>> x=(4/3)*3
x =     4
>> a=4/3          %store double precision approx of 4/3
a =     1.333333333333333e+00
>> b=a-1          %remove most significant digit
b =     3.333333333333333e-01
>> c=1-3*b        %3*b=1 in exact math
c =     2.220446049250313e-16 %should be 0!!
```

  To see the effects of roundoff in a simple calculation, one need only to force the computer to store the intermediate results.

  - All computing devices represents numbers, except for integers and some fractions, with some imprecision
  - floating-point numbers of fixed word length; the true values are usually not expressed exactly by such representations
  - if the number are rounded when stored as floating-point numbers, the round-off error is less than if the trailing digits were simply chopped off

- **Absolute vs Relative Error**:

```
>> x=tan(pi/6)
x =     5.773502691896257e-01
>> y=sin(pi/6)/cos(pi/6)
y =     5.773502691896256e-01
>> if x==y
fprintf('x and y are equal \n');
else
fprintf('x and y are not equal : x-y =%e \n',x-y);
end
x and y are not equal : x-y =1.110223e-16
```

  The test is true only if x and y are exactly equal in bit pattern. Although x and y are equal in exact arithmetic, their values differ by a small, but nonzero, amount. When working with floating-point values the question "are x and y equal?" is replaced by "are x and y close?" or, equivalently, "is $x - y$ small enough?"

- accuracy (how close to the true value) $\longrightarrow$ great importance
- $absolute\ error = |true\ value - approximate\ error|$
  A given size of error is usually more serious when the magnitude of the true value is small
- $relative\ error = \frac{absolute\ error}{|true\ value|}$

- **Convergence of Iterative Sequences**: Iteration is a common component of numerical algorithms. In the most abstract form, an iteration generates a sequence of scalar values $x_k, k = 1, 2, 3, \ldots$. The sequence converges to a limit ξ if

  $$|x_k - \xi| < \delta,\ for\ all\ k > N$$

  where δ is a small number called the convergence tolerance. We say that the sequence has converged to within the tolerance δ after N iterations.
  Example m-file: Iterative Computation of the Square Root (testSqrt.m, newtsqrtBlank.m)
  The goal of this example is to explore the use of different expressions to replace the "NOT_CONVERGED" string in the *while* statement. Some suggestions are given as:

  ```
  r~=rold
  (r-rold)> delta
  abs(r-rold)>delta
  abs((r-rold)/rold)>delta
  ```

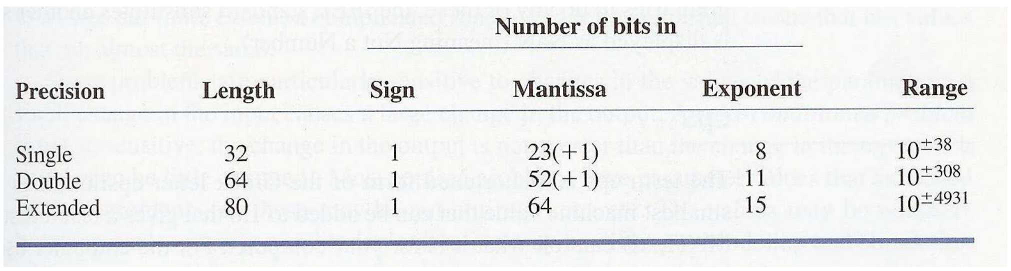
  Study each case, and which one should be used?

- **Floating-Point Arithmetic**: Performing an arithmetic operation $\Rightarrow$ no exact answers unless only integers or exact powers of 2 are involved

  - floating-point (real numbers)$\rightarrow$ not integers
  - resembles scientific notation

Table 1: Floating$\rightarrow$ Normalized.

floating	normalized (shifting the decimal point)
13.524	$.13524 * 10^2$ ($.13524E2$)
-0.0442	$-.442E-1$



		Number of bits in			
Precision	Length	Sign	Mantissa	Exponent	Range
Single	32	1	23(+1)	8	$10^{\pm 38}$
Double	64	1	52(+1)	11	$10^{\pm 308}$
Extended	80	1	64	15	$10^{\pm 4931}$

Figure 3: Level of precision.

- IEEE standard $\rightarrow$ storing floating-point numbers (see the Table 1)
  - the sign $\pm$
  - the fraction part (called the *mantissa*)
  - the exponent part
- There are three levels of precision (see the Fig. 3)
- Rather than use one of the bits for the sign of the exponent, exponents are biased. For **single** precision:
  2^8=256
  0$\longrightarrow$00000000 = 0
  255$\longrightarrow$11111111=255
  0 (255)$\Longrightarrow$ -127 (128). An exponent of -127 (128) stored as 0 (255).
  So biased$\longrightarrow$ $2^{128} = 3.40282E + 38$, mantissa gets 1 as maximum
  Largest: 3.40282E+38; **Smallest:** 2.93873E-39
  For **double** and **extended** precision the bias values are 1023 and 16383, respectively.
- Certain mathematical operations are undefined,
  $\frac{0}{0}, 0 * \infty, \sqrt{-1} \Longrightarrow NaN$

```
>> realmin
ans =  2.2251e-308
>> realmax
ans =  1.7977e+308
>> format long e
>> 10*realmax
ans =   Inf
>> realmin
```

```
ans =    2.225073858507201e-308
>> realmin/10
ans =    2.225073858507203e-309
>> realmin/1e16
ans =     0
```

When a calculation results in a value smaller than **realmin**, there are two types of outcomes. If the result is slightly smaller than **realmin**, the number is stored as a denormal (they have fewer significant digits than normal floating point numbers). Otherwise, It is stored as 0.
Example m-file: Interval Halving to Oblivion (halfDiff.m)

```
$x_1 = \ldots$, $x_2 = \ldots$
for k=1,2,...
$\delta = (x_1 - x_2)/2$
if $\delta = 0$, stop
$x_2 = x_1 + \delta$
end
```

As the floating-point numbers become closer in value, the computation of their difference relies on digits with decreasing significance. When the difference is smaller than the least significant digit in their mantissa, the value of δ becomes zero.

- **EPS:** short for epsilon$\longrightarrow$used for represent the smallest machine value that can be added to 1.0 that gives a result distinguishable from 1.0 In MATLAB:

  ```
  >> eps
  ans=2.2204E-016
  ```

  $eps \longrightarrow \varepsilon \longrightarrow (1+\varepsilon)+\varepsilon = 1 \; but \; 1+(\varepsilon+\varepsilon) > 1$
  Two numbers that are very close together on the *real* number line can not be distinguished on the *floating-point* number line if their difference is less than the least significant bit of their mantissas.

- **Round-off Error vs Truncation Error:** Round-off occurs, even when the procedure is exact, due to the imperfect precision of the computer
  Analytically $\frac{df}{dx} \Rightarrow f'(x) = lim_{h\to 0}\frac{f(x+h)-f(x)}{(x+h)-x}$: procedure
  approximate value for $f'(x)$ with a small value for $\boldsymbol{h}$.
  $h \longrightarrow smaller$, the result is closer to the true value$\longrightarrow$truncation error is reduced

but at some point (depending of the precision of the computer) round-off errors will dominate$\longrightarrow$less exact$\Longrightarrow$***There is a point where the computational error is least.***
Example m-file: Roundoff and Truncation Errors in the Series for e^x (expSeriesPlot.m)
Let T_k be the kth term in the series and S_k be the value of the sum after k terms:

$$T_k = \frac{x^k}{k!}, S_k = 1 + \sum_{j=1}^{k} T_k$$

If the sum on the right-hand side is truncated after k terms, the absolute error in the series approximation is

$$E_{abs,k} = |S_k - e^x|$$

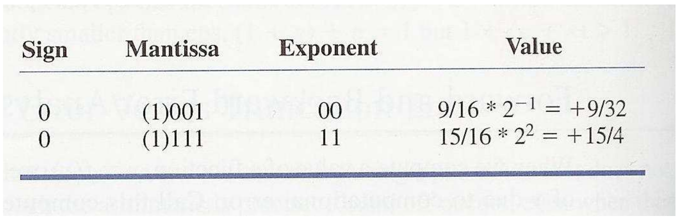
```
>> expSeriesPlot(-10,5e-12,60)
```

As k increases, $E_{abs,k}$ decreases, due to a decrease in the truncation error. Eventually, roundoff prevents any change in S_k. As $T_{k+1} \to 0$, the statement

$$ssum = ssum + term$$

produces no change in $ssum$. For $x = -10$ this occurs at $k \sim 48$. At this point, the truncation error, $|S_k - e^x|$ is not zero. Rather, $|T_{k+1}/S_k| < \varepsilon_m$. This is an example of the independence of truncation error and roundoff error. For $k < 48$, the error in evaluating the series is controlled by truncation error. For $k > 48$, roundoff error prevents any reduction in truncation error.

- **Well-posed and well-conditioned problems**: The accuracy depends not only on the computer's accuracy
  - A problem is well-posed if a solution; exists, unique, depends on varying parameters
    - A nonlinear problem$\longrightarrow$linear problem
    - infinite$\longrightarrow$large but finite
    - complicated$\longrightarrow$simplified
  - A well-conditioned problem is not sensitive to changes in the values of the parameters (small changes to input do not cause to large changes in the output)
  - Modelling and simulation; the model may be not a really good one



Sign	Mantissa	Exponent	Value
0	(1)001	00	$9/16 * 2^{-1} = +9/32$
0	(1)111	11	$15/16 * 2^{2} = +15/4$

Figure 4: Computer numbers with six bit representation.

- if the problem is well-conditioned, the model still gives useful results in spite of small inaccuracies in the parameters

- **Forward and Backward Error Analysis**:
  $y = f(x)$
  $y_{calc} = f(x_{calc})$: *computed value*
  $E_{fwd} = y_{calc} - y_{exact}$ where y_{exact} is the value we would get if the computational error were absent
  $E_{backwd} = x_{calc} - x, \; y_{calc} = f(x_{calc})$
  Example:
  $y = x^2, \; x = 2.37$ used only two digits
  $y_{calc} = 5.6$ *while* $y_{exact} = 5.6169$
  $E_{fwd} = -0.0169$, relative error $\rightarrow$0.3%
  $\sqrt{5.6} = 2.3664 \Rightarrow E_{backw} = -0.0036$, relative error $\rightarrow$0.15%

- **Examples of Computer Numbers**:
  Say we have six bit representation (not single, double) (see the Fig. 4)

  - 1 *bit* $\rightarrow$ *sign*
  - 3(+1) *bits* $\rightarrow$ *mantissa*
  - 2 *bits* $\rightarrow$ *exponent*

  For positive range $\frac{9}{32} \longleftrightarrow \frac{15}{4}$
  For negative range $\frac{-15}{4} \longleftrightarrow \frac{-9}{32}$; even discontinuity at point zero since it is not in the ranges.
  Very simple computer arithmetic system $\Rightarrow$ the gaps between stored values are very apparent. Many values can not be stored exactly. i.e., 0.601, it will be stored as if it were 0.6250 because it is closer to $\frac{10}{16}$, an error of 4% In IEEE system, gaps are much smaller but they are still present. (see the Fig. 5)

- **Anomalies with Floating-Point Arithmetic**:
  For some combinations of values, these statements are not true

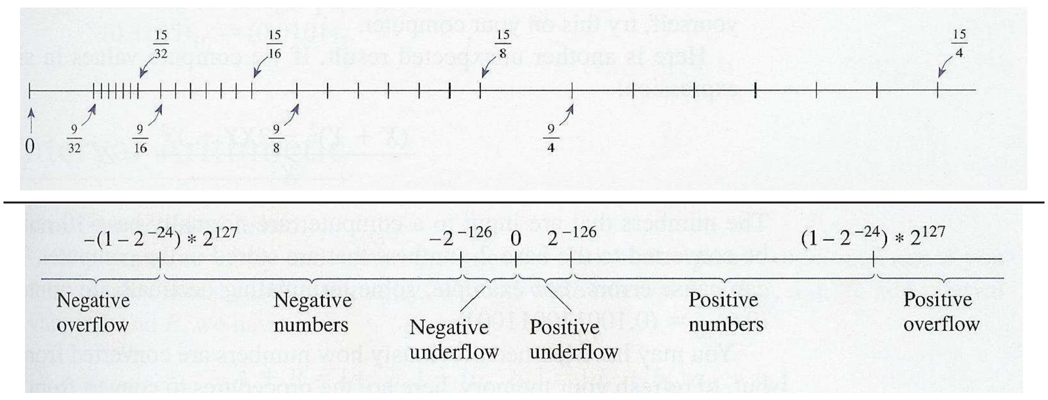


Figure 5: Upper: number line in the hypothetical system, Lower: IEEE standard.

- $(x + y) + z = x + (y + z)$
  $(x * y) * z = x * (y * z)$
  $x * (y + z) = (x * y) + (x * z)$

- adding 0.0001 one thousand times should equal 1.0 exactly but this is not true with single precision
- $z = \frac{(x+y)^2 - 2xy - y^2}{x^2}$, problem with single precision

1.6 Parallel and Distributed Computing

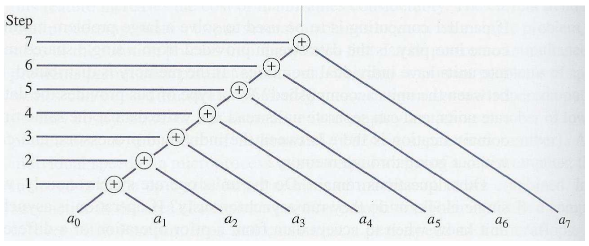
- Tremendously large problems and the solution may be needed almost instantaneously. (real time)

- Computer systems (mostly) run their instructions in sequence

- Super computers $\longrightarrow$ billions of operations per second $\longrightarrow$ not fast enough

- First technique; **pipelining**: performing a second instruction within the CPU before the previous instruction is completed. Pipelining permits a speed up by a factor of two or more.

- Second technique; build **vector processing** operations in the CPU. To solve sets of equations involve may multiplications of a vector by another vector. Improvement by a factor of 5 or 10, not by the factor of 10,000. (Eventhough the cost increases considerably)



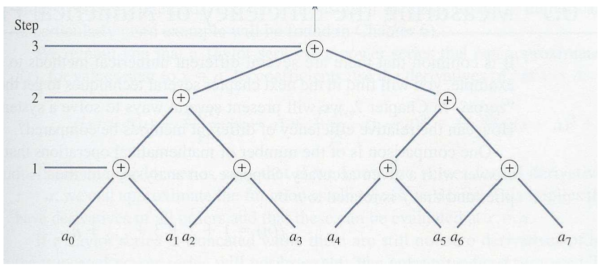


Figure 6: Left: Adding eight numbers sequentially. Right: Adding eight numbers with parallel processors.

- Third technique (current trend); **parallel processing**: Put several machines to work on a single problem, dividing the steps of the solution process into many steps that can be performed simultaneously
  - Massively parallel computers; employ a massive number of low cost processors (1000 Pentium Pros$\longrightarrow$1.3 Teraflops)
  - Beowulf class; PCs joined (cluster) to work together to create a modestly priced supercomputer
- Fourth technique; **distributed computing**: to connect many different computers, which can work separately on their own tasks as well as in conjunction with each other. Asynchronous operations (interrupts constantly occur through the system to coordinate the actions)
  - data can flow from one computer to another
  - OS, software, and connecting the computers are major challenges

1.6.1 Speed -up and Efficiency

See the Fig. 6.

- $S_p(n) = \frac{T_1(n)}{T_p(n)}, \frac{7}{3}$; speed-up
- $E_p(n) = \frac{S_p(n)}{p}, \frac{7/3}{4}$; efficiency

1.7 Measuring the Efficiency of Numerical Procedures

- One comparison is of the number of mathematical operations that are needed to get the answer with a given accuracy. An equation $f(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$ As n gets large, the first term dominates and

we say that $f(n)$ if "of order n^2"; $f(n) = O(n^2)$.
$n \longrightarrow double$
$number\ of\ multiplications \longrightarrow four\ times$

- say x values differ by h (commonly used variable for such spacing). The error in the answer is proportional to the third power of h;
$Error = \frac{M}{6}h^3$; the error is of the order h^3;$Error = O(h^3)$

1.7.1 Taylor Series

The expression for the order of error given above is found by comparison of the procedure with a Taylor series.

- **A Taylor series is a power series that can approximate a function**, f(x), for values near to x=a.
- Its coefficients use the derivatives of f at $x = a$
- $f(x) = f(a) + \frac{f^{'}(a)}{1!}(x-a) + \frac{f^{''}(a)}{2!}(x-a)^2 + \frac{f^{'''}(a)}{3!}(x-a)^3 + \dots$
- The Taylor series says that if we know the values for all derivatives of $f(x)$ at $x = a$, we can approximate the function as closely as we desire.
- $Error\ of\ TS = \frac{f^{n+1}(\xi)}{(n+1)!}$: The error term for a truncated Taylor series after the n^{th} term
- where ξ is a value between x and $(x+a)$. Since the value of ξ is not known, there is still uncertainty in the exact value of the error.

Example m-file: Taylor Series Approximations to $1/(1-x)$ (demoTaylor.m)
Consider the function

$$f(x) = \frac{1}{1-x}$$

Make the Taylor series expansion of this function up to third order.

```
demoTaylor(1.6,0.8)
```

All of the Taylor polynomials agree with $f(x)$ near $x_0 = 1.6$. The higher order polynomials agree over a larger range of x.

1.7.2 Polynomials

- A truncated Taylor series is just a polynomial, and a computer can handle
- only maths needed are addition and multiplication
- Any continuous function can be approximated uniformly over a finite interval by a polynomial
- *Chebyshev* polynomial; better then Taylor series in approximating functions
- *Legendre* polynomial; good way to integrate a function numerically
- use nested form $P(x) = ((a_3x + a_2)x^2 + a_1x) + a_0$ instead of $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3$
- since the former has four multiplications and three additions instead of six multiplications and three additions