MATHEMATICAL PHYSICS DEPARTMENT

MASTER OF COMPUTATIONAL SCIENCE DEGREE 2006-2007

NUMERICAL ALGORITHMS

Dr Derek O'Connor

Lab Exercise No. 3 : MATLAB'S **Floating Point Number System.**

OUT: Wed 27 Sep 2006 IN : Wed 4 Oct 2006[1]

1 PURPOSE

The purpose of this exercise is to familiarize you with MATLAB'S floating-point arithmetic system which is, by default, IEEE double precision. It is important to know the parameters of the number system you are using and to be aware of its limitations. Although MATLAB is quite good at telling you what these are, other systems or compilers are not—try doing this exercise in Microsoft's Excel.

2 EXERCISE

1. Write a MATLAB function `MachEps()` that calculates machine epsilon. Although MATLAB has the built-in function `eps` to do this, it is important to write your own because other languages do not have a built-in `MachEps` function. If written correctly, an implicit bonus of this function is that it counts the number of bits of precision in the F.P. system used. Try to find this once you have the function working properly — requires no extra code except passing back an extra parameter (argument).

2. MATLAB has various important constants and functions built in. Some of these are `eps`, `realmax`, `realmin`, `inf`, `pi`. Also available are `date`, `ver`, and `version` which are useful for annotating output.

   Determine what these are on your system.

3. Determine the response of MATLAB to the indeterminates

   $$\frac{0}{0}, \frac{\infty}{\infty}, \infty * 0, 1^{\infty}, \infty - \infty, 0^0, \infty^0.$$

   Do you agree with MATLAB's results? Explain.

4. Calculate mathematically the result `e` of the following 4 statements:

   ```
   a = 4/3;  b = a-1;  c = b+b+b;  e = 1-c;
   ```

   Now use MATLAB to do the same calculations. Explain the result.

[1] This is a slightly modified version of the original. DO'C, December 5, 2010

5. $\sin(\pi) = 0$, $\cos(\pi) = -1$, and $\sin^2(\pi) + \cos^2(\pi) = 1$. What does MATLAB give for `sin(pi), cos(pi), sin(pi)^2+cos(pi)^2` ? Explain.

You can get a deeper understanding of what MATLAB is doing by looking at the binary form of the numbers above. Use the function `num2bin(x)` to see a binary version of the decimal number x. This is available on the class website.

3 SOLUTION

3.1 Machine Epsilon

Machine epsilon (ϵ_m) is the spacing between the 1.0 and the next higher floating point number. Knowing this number allows us to calculate the spacing about any number x as $\epsilon_m|x|$. For some reason, students have difficulty understanding the meaning and significance of this (f.p) number. This is one of the reasons for getting you to calculate ϵ_m in MATLAB, even though it is a built-in function (constant?). The other reason is that many languages or systems do not provide an `eps` function and so you will have to provide your own.

The function `MachEps` uses the fact that ϵ_m *is the smallest floating point number such that* $\mathrm{fl}(1.0 + \epsilon_m) > 1.0$. It starts off with $\epsilon_m =$ `epsil` = 1.0 and repeatedly halves it (base 2) until $\mathrm{fl}(1.0 + \epsilon_m) \leq 1.0$. Thus $\mathrm{fl}(1.0 + \epsilon_m) = 1.0 + \epsilon_m$ at each iteration, except at the end of the last iteration. Thus, it generates a sequence of contiguous floating point numbers, except the last. You need to think about that.

```
%-------------------------------------------
  function [meps, prec] = MachEps()
%
% Determines Machine Epsilon and precision in bits.
% Derek O'Connor, Oct 2004.
%-------------------------------------------
    k = 1;
    epsil = 1.0;
    epone = 1.0 + epsil;
    while epone > 1.0
       epsil = epsil/2.0;
       epone = 1.0 + epsil;
       k = k + 1;
    end;
% -- gone one iteration too far, so :
    meps = 2.0*epsil;
    prec = k-1;
%--------- End [meps, prec] = MachEps()-------
```

Figure 1 shows the last 3 iterations of `MachEps`. The function has generated the sequence of floating point numbers `epone` $= \{1+1, 1+2^{-1}, \ldots, 1+2^k\epsilon_m, \ldots, 1+2^2\epsilon_m, 1+2^1\epsilon_m, 1+2^0\epsilon_m\}$. In the final iteration it generates the number `epone` $= 1+2^{-1}\epsilon_m$. This is halfway between $1+\epsilon_m$ and 1, contiguous floating point (representable) numbers. Thus it is not representable and must, therefore, be rounded.

Now comes the delicate bit. The IEEE standard allows four *rounding modes* :

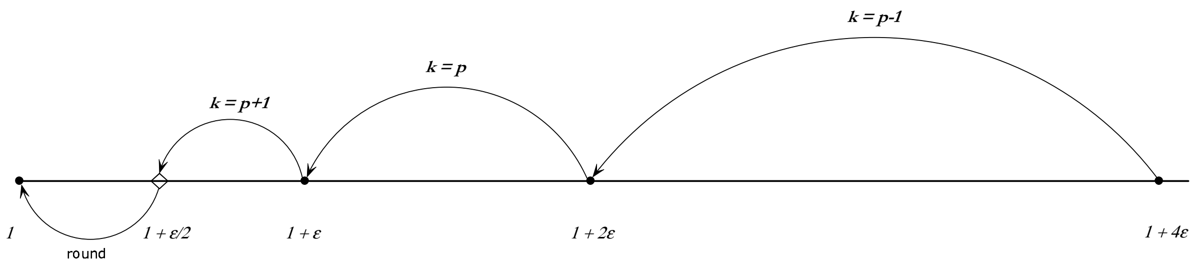


Figure 1: Calculating Machine Epsilon

1. Round Down. 2. Round Up. 3. Round towards Zero. 4. Round to Nearest.

Mode 4, the *round-to-nearest mode*, is almost always used in practice. This rule is : round x to the nearer of the two floating point numbers, x_- and x_+, adjacent to x. In the case of a tie, choose the one whose *least significant bit is zero*.

The last number generated by `MachEps` is `epone`$= 1 + 2^{-1}\epsilon_m = x$, and this is equi-distant from $x_- = 1$ and $x_+ = 1 + \epsilon_m$. Which of these two has a zero least significant bit? The floating point number $x_- = 1$ has its last bit zero, and so `epone`$= 1 + 2^{-1}\epsilon_m$ is rounded to 1. See Figure 1. Once `epone` becomes 1, the **while** –loop ends, with $k = p + 1$ and `epsil` $= \epsilon_m/2$. Finally, the adjustment `meps = 2.0*epsil; prec = k-1` is made. Table 1 shows the binary numbers generated by `MachEps`.

Table 1: Binary Output of `MachEps`

k	epsil	epone = 1 + epsil
1	$.1000000000000\cdots00 \times 2^{+1}$	$.10000000000\cdots000000 \times 2^{+2}$
2	$.1000000000000\cdots00 \times 2^{+0}$	$.11000000000\cdots000000 \times 2^{+1}$
3	$.1000000000000\cdots00 \times 2^{-1}$	$.10100000000\cdots000000 \times 2^{+1}$
4	$.1000000000000\cdots00 \times 2^{-2}$	$.10010000000\cdots000000 \times 2^{+1}$
5	$.1000000000000\cdots00 \times 2^{-3}$	$.10001000000\cdots000000 \times 2^{+1}$
6	$.1000000000000\cdots00 \times 2^{-4}$	$.10000100000\cdots000000 \times 2^{+1}$
7	$.1000000000000\cdots00 \times 2^{-5}$	$.10000010000\cdots000000 \times 2^{+1}$
8	$.1000000000000\cdots00 \times 2^{-6}$	$.10000001000\cdots000000 \times 2^{+1}$
9	$.1000000000000\cdots00 \times 2^{-7}$	$.10000000100\cdots000000 \times 2^{+1}$
10	$.1000000000000\cdots00 \times 2^{-8}$	$.10000000010\cdots000000 \times 2^{+1}$
$\vdots$	$\vdots$	$\vdots$
50	$.1000000000000\cdots00 \times 2^{-48}$	$.10000000000\cdots01000 \times 2^{+1}$
51	$.1000000000000\cdots00 \times 2^{-49}$	$.10000000000\cdots00100 \times 2^{+1}$
52	$.1000000000000\cdots00 \times 2^{-50}$	$.10000000000\cdots00010 \times 2^{+1}$
53	$.1000000000000\cdots00 \times 2^{-51}$	$.10000000000\cdots00001 \times 2^{+1}$
54	$.1000000000000\cdots00 \times 2^{-52}$	$.10000000000\cdots00000 \times 2^{+1}$

3.2 Floating Point Parameters of Matlab

MATLAB 6.5 uses IEEE double precision for all calculations. The MATLAB floating point parameters can be determined from the built-in functions `realmax`, `realmin`, `eps`, which give the values shown in Table 2.

Table 2: Machine parameters for Matlab 7.0.1.24704 (R14) Service Pack 1

`eps`	2.220446049250313 e−016	$+.1000000000000000000000000000000000000000000000000000 \times 2^{-51}$
`realmin`	2.225073858507201 e−308	$+.1000000000000000000000000000000000000000000000000000 \times 2^{-1021}$
`realmax`	1.797693134862316 e+308	$+.1111111111111111111111111111111111111111111111111111 \times 2^{+1024}$
`1/realmin`	4.494232837155790 e+307	$+.1000000000000000000000000000000000000000000000000000 \times 2^{+1023}$
`1/realmax`	5.562684646268004 e−309	$+.1000000000000000000000000000000000000000000000000000 \times 2^{-1023}$

The binary output was obtained using the `num2bin` function of Prof. Herman Gollwitzer, Mathematics and Computer Science Depts., Drexel University, Pennsylvania.

The number `1/realmin= 4.494232837155790e+307` is in the Floating Point number range but the number `1.0/realmax= 5.562684646268004e-309` is not but is displayed in MATLAB. Why? Because IEEE FP uses *gradual underflow* which allows numbers below the underflow threshold `realmin` to exist as *subnormals*.

3.3 Floating Point Exceptions

The table below gives MATLAB's response to various mathematical indeterminates. These are the correct results according to the IEEE standard.

Table 3: Floating Point Exceptions

$\frac{1}{0}$	$\frac{1}{\infty}$	$\frac{0}{0}$	$\frac{\infty}{\infty}$	$\infty \times 0$	1^∞	$\infty - \infty$	0^0	∞^0
`inf`	0	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	1	1

The expressions in the Table 3 are mathematically meaningless, but computationally they are important. This is why the IEEE standard has carefully specified the results of such floating point operations.

Example 1 (Parallel Resistors)**.** Here is a simple example that shows why operations with ∞ are necessary. The circuit below comprises a $V-$volt battery and two resistors of R_1 ohms and R_2 ohms connected in parallel across its terminals. We wish to calculate the currents i, i_1, and i_2 and hence the power dissipated by the circuit, $P = R_1 i_1^2 + R_2 i_2^2$ watts.

The state of the circuit is determined by the following equations, which are derived from *Ohm's* and *Kirchoff's* laws:

$$V = R_1 i_1 = R_2 i_2 = Ri, \text{ where } R = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}, \text{ and } \quad i = i_1 + i_2.$$

So, given V, R_1, and R_2, we wish to calculate R, i_1,i_2, and i. The calculation of R can cause floating point exceptions if one or both or the resistors is (i) 0 (short circuit) or (ii) ∞ (open

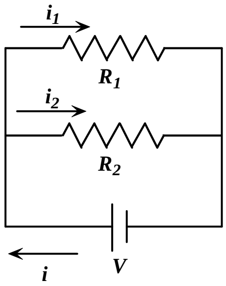


Figure 2: Parallel Resistors.

circuit). In these cases we have

$$R_{sc} = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = 0, \quad R_{oc} = \frac{1}{\frac{1}{\infty} + \frac{1}{R_2}} = \frac{1}{0 + \frac{1}{R_2}} = R_2$$

These calculations give the correct physical results but many compilers and mathematical systems will either give an error or crash.

Although the example above may be of interest to electrical engineers only, the following example shows the widespread need for proper exception handling

Example 2 (Calculating the Norm of a Vector.)**.** This is a common problem in Numerical Linear Algebra : the length or *norm* of the vector $x = (x_1, x_2, \dots, x_n)$ is

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

The MATLAB program is simple :

```
function norm = NormNS(x,n)
    sum = 0;
    for i = 1:n
        sum = sum + x(i)*x(i);
    end
    norm = sqrt(sum);
```

This piece of code is not reliable. If any $x(i)$ is of the order 10^{200} then $x(i) * x(i)$ will cause **overflow**, i.e., it will not fit in a computer word and a fatal error may occur. If the $x(i)$ values are small then there is a danger of **underflow** or negative overflow, i.e., when a result becomes too small it is set to zero. For example, if $n = 10000$ and $x_i = 10^{-200}$ then the length of this vector is 10^{-180}. All of these numbers are perfectly valid computer numbers but the code above will give 0 as the result, because $x_i^2 = 10^{-400}$ will be set to 0 and usually without warning that underflow has occurred. We can overcome these *range violation* problems if we scale the data before we calculate the norm. Here is a MATLAB function `NormS` that scales small numbers up and large numbers down.

```
function norm = NormS(x,n)

    xmax = 0;
    for i = 1:n
       if abs(x(i)) > xmax
          xmax = abs(x(i));
       end;
    end;
    sum = 0;
    if xmax > LARGE
       for i = 1:n
          xi = x[i]*LScale;
          sum = sum + xi*xi;
       end
       norm = sqrt(sum)/LScale
    elseif xmax < SMALL
       for i = 1:n
          xi = x[i]*SScale;
          sum = sum + xi*xi;
       end
       norm = sqrt(sum)/SScale;
    else
       for i = 1:n
          sum = sum + x(i)*x(i);
       end;
       norm = sqrt(sum);
    end;
```

```
function norm = NormCS(x,n)
    sum = 0;
    for i = 1:n
         sum = sum + x(i)*x(i);
    end;
    norm = sqrt(sum);
    if Norm < n*SMALL    % Underflow
        sum = 0;
        for i = 1:n
           xi = x[i]*LScale;
           sum = sum + xi*xi;
        end
        norm = sqrt(sum)/LScale;
    end;
    if Norm > n*LARGE    % Overflow
        sum = 0;
        for i = 1:n
           xi = x[i]*SScale;
           sum = sum + xi*xi;
        end;
        norm = sqrt(sum)/SScale;
    end;
```

Many linear algebra packages use scaling to avoid range violations. This code is time-consuming and very often the extra work spent in scaling is not necessary.

The MATLAB function `NormCS` above uses *conditional scaling* : it performs the ordinary unscaled sum first and then uses floating point exceptions to determine if scaling is necessary. This change can give dramatic speed-ups in large simulations that use the norm operation on many different vectors.

Question 1. In the MATLAB functions `NormS` and `NormCS` above, how should the values for `LARGE,SMALL,LScale,SScale` be chosen? What do they depend on?

3.4 Kahan's Machine Epsilon

These statements were first given by Prof William Kahan, Berkeley.

a = 4/3; b = a-1; c = b+b+b; e = 1-c;

Mathematically we have

$$a = \frac{4}{3}, \quad b = \frac{4}{3} - 1 = \frac{1}{3}, \quad c = \frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1, \quad e = 1 - 1 = 0.$$

Performing these statements in $F(b, p, -, -)$, where b is not a multiple (power?) of 3, we get

1. $a = \text{fl}(4/3) \ = \text{fl}(1.33\ldots33\ldots) \ = \underbrace{1.33\ldots3}_{p \text{ digits}}$
2. $b = \text{fl}(\text{fl}(a) \ - 1) \ = \text{fl}(1.33\ldots3 - 1.0) \ = 0.\underbrace{33\ldots3}_{p-1}$
3. $c = \text{fl}(b + b + b) \ = \text{fl}(0.33\ldots3 + 0.33\ldots3 + 0.33\ldots3) \ = 0.\underbrace{99\ldots9}_{p-1}$
4. $e = \text{fl}(1 - c) \ = \text{fl}(1.0 - 0.\underbrace{99\ldots9}_{p-1}) \ = 0.\underbrace{00\ldots1}_{p-1} = 1.00\ldots0 \times b^{1-p}$

Thus we get $e = b^{1-p} = \epsilon_m$. We have implicitly assumed that $b = 10$.

Notice that the only rounding error occurs in the statement `a = 4/3`. This rational number does not have a finite expansion in base 2 or 10 and so there will always be a rounding error, no matter how large p is.

Here is a small Fortran function that uses Kahan's ϵ_m calculation. This function was in the Eispack subroutine package, and is still part of Dongarra's Linpack benchmark program `1000d.for`. Note the starred sentence at the end of the comments.

```
C=======================================================================
      double precision function epslon (x)
C=======================================================================
      double precision x
c
c     estimate unit roundoff in quantities of size x.
c
      double precision a,b,c,eps
c
c     this program should function properly on all systems
c     satisfying the following two assumptions,
c        1.    the base used in representing dfloating point
c            numbers is not a power of three.
c        2.    the quantity  a  in statement 10 is represented to
c            the accuracy used in dfloating point variables
c            that are stored in memory.
c     the statement number 10 and the go to 10 are intended to
c     force optimizing compilers to generate code satisfying
c     assumption 2.
c     under these assumptions, it should be true that,
c            a  is not exactly equal to four-thirds,
c            b  has a zero for its last bit or digit,
c            c  is not exactly equal to one,
c            eps  measures the separation of 1.0 from
c                 the next larger dfloating point number.
c     the developers of eispack would appreciate being informed       *
c     about any systems where these assumptions do not hold.          *
c
c     *****************************************************************
c     this routine is one of the auxiliary routines used by eispack iii
c     to avoid machine dependencies.
c     *****************************************************************
c
c     this version dated 4/6/83.
c
      a = 4.0d0/3.0d0
   10 b = a - 1.0d0
      c = b + b + b
      eps = dabs(c-1.0d0)
      if (eps .eq.  0.0d0) go to 10
      epslon = eps*dabs(x)
      return
      end
```

3.5 Calculations with π and pi

We know that $\sin(\pi) = 0$, $\cos(\pi) = -1$, and $\sin^2(\pi) + \cos^2(\pi) = 1$. But MATLAB gives

1. `sin(pi)` $= 1.224646799147353\,e - 016$
2. `cos(pi)` $= -1$
3. `sin(pi)^2+cos(pi)^2` $= 1$

As we can see, 2 and 3 are correct but 1 is not. There are two sources of error here : (i) the error in representing π, a transcendental number, and (ii) the error in computing $\sin(x)$ or any other function.

Representing π.

Let us consider the representation of π. For reference, here are the first 100 decimal digits of π along with the binary representation :

```
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170 67
```

```
11.0010010000111111011010101000100010000101101000110000100011010011000100110001100110001010001011100000
   0011011100000111001101000100101001000000100100111000001000100010100110011111001100011101000000001000
   0010111011111010100110001110110001001110011011001000100101000101001010000010000111100110001110001101
   0000000100110111011110111110010101000110011011001111001101001110100100001100011011001100000010101100
   0010100110110111111001001011111100010100001101110011
```

MATLAB gives the value of `pi`$= 3.141592653589793\,(2384626433832795028)$ and so all 16 digits are correct. Here is a table of the floating point numbers just below and above `pi`. Notice that each adjacent pair differs by 1 bit in the last place.

Table 4: MATLAB'S `pi` $\neq \pi$

`pi`$-$2*`eps`	3.141592653589793	$+.11001001000011111101101010100010001000010110100010111 \times 2^{+2}$
`pi`	3.141592653589793	$+.11001001000011111101101010100010001000010110100011000 \times 2^{+2}$
`pi`$+$2*`eps`	3.141592653589794	$+.11001001000011111101101010100010001000010110100011001 \times 2^{+2}$

We see that the true value of π lies about $1/4 \approx (.2384626433832795028)$ the way between `pi` and the next higher floating point number `pi+2*eps`. Hence the error in `pi` is

$$\begin{aligned} e_\pi = \pi - \texttt{pi} &= 3.141592653589793\,2384626433832795028\ldots - 3.141592653589793 \\ &= 2.384626433832795028\ldots \times 10^{-16} \end{aligned}$$

If we assume that $\sin(x)$ and $\cos(x)$ are calculated correctly then what effect does the error in `pi` have on the results? The Taylor series expansions of $\sin(x)$ and $\cos(x)$ are

$$\begin{aligned} \sin(x) &= x - \frac{x^3}{6} + \cdots \approx x \quad \text{for small} \quad x \\ \cos(x) &= 1 - \frac{x^2}{2} + \cdots \approx 1 \quad \text{for small} \quad x \end{aligned}$$

Using the series approximations above we get

$$\begin{aligned} \sin(\text{pi}) &= \sin(\pi - e_\pi) = +\sin(e_\pi) \approx e_\pi = 2.384626433832795028 \times 10^{-16} \\ \cos(\text{pi}) &= \cos(\pi - e_\pi) = -\cos(e_\pi) \approx -1 \end{aligned}$$

These are very close to the answers MATLAB gives. See 4.1 for more on calculations with $\sin(x)$ and $\cos(x)$.

3.5.1 Additional Questions & Exercises

Exercise 1. Work through the the function `MachEps` by hand for the floating point system $F(2,5,-6,+6)$.

Exercise 2. Modify the function `MachEps` so that it calculates `realmin`, the smallest floating point number. This f.p. number defines the *underflow threshold* of the floating point system in use.

Exercise 3. Modify the function `MachEps` so that it calculates `realmax`, the largest floating point number. This f.p. number defines the *overflow threshold* of the floating point system in use.

Exercise 4. The crucial statement in `MyEps` is `if 1.0 + epsil` $>$ `1.0`. This apparently works as intended in MATLAB, but it it is dangerous. If you use this in other programming languages then the compiler may 'optimize' the statement to `epsil` $>$ `0.0`. Mathematically this is the same as the original statement. Is it the same computationally? Check it out. Re-run your machine epsilon program with the 'optimized' statement and see what happens. General warning:

Be very wary of optimizing compilers.

Problem 1. Write the MATLAB functions `NormS` and `NormCS` in Section 3 above, and test them on the *Sea Surface Height* data whose link is on the class webpage after the Introductory Lecture slides.

3.6 Comments and Criticisms

Most of you did a good job on this exercise and your LaTeX is improving. I urge those who still balk at LaTeX to persevere.

1. The crucial statement in `MyEps` is `if 1.0 + epsil` $>$ `1.0`. This apparently works as intended in MATLAB, but it is dangerous. If you use this in other programming languages then the compiler may 'optimize' the statement to `epsil` $>$ `0.0`. Mathematically this is the same as the original statement. Is it the same computationally? Check it out. Re-run your machine epsilon program with the 'optimized' statement and see what happens. General warning : *Be very wary of optimizing compilers.*

2. You need to check that MATLAB'S results for `eps`, etc., agree with the floating point 'theory'. For example, does MATLAB'S `eps` $= b^{1-p}$?

3. Indeterminates (1^{∞} etc.). The IEEE committee thought long and hard about these and other floating point exceptions and finally agreed to what Prof W. Kahan told them to do in the first place.

4. The `a = 4/3; b = a-1; c = b+b+b; e = 1-c` Problem. Most students did not explain the result obtained by MATLAB. Those who did try, got into a muddle. The problem is that what you see (on the screen) is a decimal conversion of internal binary floating point numbers and these are not the same.

5. Programming style need to be improved. Remember that you are, in general, writing short pieces of mathematical software. Variable names need to be chosen that reflect the problem at hand. Long, tedious, 'stating-the-obvious' comments obscure the code (maybe this is your intention?)

6. Functions for calculating `eps` etc., should not have I/O statements. I/O statements should be confined to special I/O functions, not strewn willy-nilly throughout a program.

7. Functions should do one job and do it well. A function that calculates `eps`, `realmin`, `realmax`, and the the date of Easter Sunday in the year 2020, is not a good idea.

8. Those of you who are not using the MikTeX - WinEdt combination are going out of their way to make life difficult for themselves. For example, how do you typeset 3 equations whose '=' signs are aligned? WinEdt has a template for this and it is chosen from a drop-down menu.

9. When writing any report, you are trying to convince ('con' for short) the reader that you understand what you are writing. In other words, write the report (or program comments) for the reader, not yourself.

10. Test programs should state the version of MATLAB that you are using. The following is an example that writes out the version and date. Always use these where appropriate. Remember that MATLAB's function `disp([s1 s2 s3])` prints out strings. Hence all numbers must be converted to strings. MATLAB should re-design all its I/O functions to be easy and consistent, instead of a mixture of C and its own awkward I/O functions.

```
function dummy = MachParams()
format long e
disp(' ');disp(' ');
disp(['Machine parameters for Matlab ' version ;'  Date :  '  date]);
disp(['Mach Eps  :  '  num2str(eps)       '    '  num2bin(eps)]);
disp(['RealMin   :  '  num2str(realmin)   '    '  num2bin(realmin)]);
disp(['RealMax   :  '  num2str(realmax)   '    '  num2bin(realmax)]);
disp(['1/RealMin :  '  num2str(1/realmin) '    '  num2bin(1/realmin)]);
disp(['1/RealMax :  '  num2str(1/realmax) '    '  num2bin(1/realmax)]);
```

4 Real Measurement Problems

Throughout this and similar courses on numerical analysis we use relative error rather than absolute error to measure the difference between a number x and an approximation to x. Let $\hat{x} = x + E_x$ be an approximation to x. Then[2]

$$\text{Absolute Error} \quad E_x = |\hat{x} - x|, \quad \text{and} \qquad \text{Relative Error} \quad e_x = \frac{|\hat{x} - x|}{|x|} = \frac{E_x}{|x|}, \quad x \neq 0.$$

To see why we use relative error, consider an algorithm $A(x)$ that calculates $\frac{x}{3}$, using 3-digit decimal arithmetic. For $x = 1$ we get $A(x) = 0.333$, the best we can do with 3-digit arithmetic. For $x = 100$ we get $A(x) = 33.3$, the best we can do with 3-digit arithmetic. Now let us compare the errors in these two calculations.

For $x = 1$ we have

$$E_1 = \left|0.333 - \frac{1}{3}\right| = \left|\frac{333}{1000} - \frac{1}{3}\right| = \frac{999 - 1000}{3000} = \frac{1}{3000}, \quad \text{and} \quad e_1 = \frac{E_1}{\frac{1}{3}} = \frac{1}{1000}.$$

For $x = 100$ we have

$$E_{100} = \left|33.3 - \frac{100}{3}\right| = \left|\frac{333}{10} - \frac{100}{3}\right| = \frac{999 - 1000}{30} = \frac{1}{30}, \quad \text{and} \quad e_{100} = \frac{E_{100}}{\frac{100}{3}} = \frac{1}{1000}.$$

We can see that $E_{100} >> E_1$ which might lead the naïve to conclude that the algorithm $A(x)$ performs badly for larger values of x. This is obviously a false conclusion because $A(x)$ is doing the best it can with 3-digit arithmetic. The fact that the relative errors are equal shows that $A(x)$ performs properly in both cases.

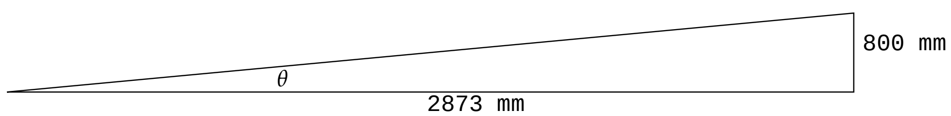
None-the-less, relative error can be misleading when comparing real measurements, and dangerously so when safety is involved. We now demonstrate this point with two examples.

4.1 A Windows 2006 Problem.

In May 2006 I was asked to solve the following problem by Eamon Heffernan, Brusselstown Window Company, Co. Wicklow.

Problem

Given a right-angled triangle with base 2873 mm and height 200 mm, find the angles and the hypotenuse. The angle θ is between the base and the hypotenuse. Measurements are within $\pm$5mm.



[2]Generally the sign of the error in numerical analysis is not important, but in real applications it maybe very important.

Solution.

1. I calculated the angle θ on a 10-digit decimal calculator as $\theta = \tan^{-1}(200/2873) = 3.982143737^\circ$.
2. It seemed reasonable to round θ to 4.0°, a relative change of $2/398 = 0.005 = 1/2\%$
3. Calculated the hypotenuse as $200/\sin(4^\circ) = 2867$. Less than the base and obviously wrong.
4. Calculated the hypotenuse as $2873/\cos(4^\circ) = 2880$. Not obviously wrong, but is it accurate?
5. Check with $\sqrt{200^2 + 2873^2} = 2880$. Seems ok.

What went wrong?

Expanding $\sin\theta$ and $\cos\theta$ in Taylor series about 0 gives

$$\begin{aligned}\sin\theta &= \theta - \frac{\theta^3}{6} + \frac{\theta^5}{120} - \frac{\theta^7}{5040} + \cdots \approx \theta \quad \text{for small } \theta \\ \cos\theta &= 1 - \frac{\theta^2}{2} + \frac{\theta^4}{24} - \frac{\theta^6}{720} + \cdots \approx 1 \quad \text{for small } \theta\end{aligned}$$

This shows that the calculation of $\cos\theta$ is impervious to small changes about $\theta = 0$, while $\sin\theta$ is not. The calculation $2873/\cos(4^\circ) = 2880$ has virtually no error due to the rounded θ, but $200/\sin(4^\circ) = 2867$ has a significant error (-13 mm) which is outside the ± 5 mm measurement tolerance.

Questions.

Because of the simplicity of this problem it was immediately obvious that the $200/\sin(4^\circ) = 2867$ calculation was wrong.

- Would the error have been obvious to a computer program or, God forbid, a spreadsheet?
- Imagine if this had been a bridge design problem where the 'mm's become 'm's.
- Imagine if this had been an aircraft navigation problem where the 'mm's become 'miles'. An error of 13 miles is still 13 miles, even if the plane has flown 10,000 miles. [3]

Moral.

$\boxed{\text{Check your answers. Nature will anyway.}}$

[3] Fly into Hong Kong airport some time. No, don't!

A High-Precision Solution.

Using PariGP 2.2.13 the following calculations were performed in 28-digit precision to show how the the $\sin\theta$ and $\cos\theta$ solutions vary.

```
» atan(200/2873)
%1 = 0.06950151949217482078904047120
» 200/sin(%1)
%2 = 2879.952951004581949679215264x
» 2873/cos(%1)
%3 = 2879.952951004581949679215264
» sqrt(200^2+2873^2)
%4 = 2879.952951004581949679215264
» for(k = 1,10,print(2873/cos(0.0690+k/10000)));
» for(k = 1,10,print(200/sin(0.0690+k/10000)));
```

θ rads	Hypot. $= 200/\sin\theta$	Hypot. $= 2873/\cos\theta$
0.0691	2896.660622865911756378360527	2879.872687151729620366215998
0.0692	2892.481366398525702188297372	2879.892633342864914104913784
0.0693	2888.314180929662978422210474	2879.912608609624961404480758
0.0694	2884.159014279308061993612363	2879.932612953014291599917688
0.0695	2880.015814567762201235078096	2879.952646374038912993428880
0.0696	2875.884530213485967830661337	2879.972708873706312978191603
0.0697	2871.765109930960380754635414	2879.992800453025458162310485
0.0698	2867.657502728566415964346892	2880.012921113006794492956926
0.0699	2863.561657906482717726611174	2880.033070854662247380693534
0.0700	2859.477525054601329561314703	2880.053249679005221823983630

$$\text{Note that the rounded angle is} \quad \theta \approx 4.0^\circ \quad = \quad 0.06981317007977318307694763074 \text{ rads}$$
$$\text{The unrounded angle is} \quad \theta \quad = \quad 0.06950151949217482078904047120 \text{ rads}$$

The difference in the lengths of the hypothenuse calculated by these angles is 13mm, which is very significant: if the width of a window and the concrete wall opening differ by 13mm then either the wall or the window or both have to be altered(cut). It is no use saying that 13mm is just 100*13/2890 = 1/2% off. In physical measurements the absolute error is important.

For any θ in the range $[0.05 - 0.09]$ radians, the value of the hypotenuse calculated by

- $2873/\cos\theta$ is within $\pm$5mm of 2880.
- $200/\sin\theta$ ranges from 4001mm to 2225mm.

4.2 A DW&W Railway Co. Problem

On the morning of February 14th 1900, a cattle train of the Dublin Wicklow & Wexford Railway Co. departed from Enniscorty at 10.a.m. bound for Harcourt Street Dublin, a distance of about 100 miles. Having loaded cattle at various stations along the way, it 'arrived' in Harcourt St. station. This arrival is shown in Figure 3.

Figure 3: A Small Relative Error at Harcourt St Station, Dublin, February 14, 1900.

We can guess from the picture that the train overshot the platform by about 100 ft. Hence the relative error was $100/(100 \times 5280) \approx 2 \times 10^{-4}$. This is quite a small relative error but, as the picture shows, it has little meaning when measurements involve safety.

Harcourt Street Station Crash

This is an interesting history of this crash, reproduced from http://www.harcourtstreettraincrash.com/

In 1900 the City of Dublin was served by 6 railway termini, which circled the City. Moving clockwise

1. The Great Northern Railway Co., operated from Amiens Street Station (Connolly Station).
2. The London and North Western Railway Co., from the North Wall Station (the Point Depot), closed April 1923.
3. The Dublin Wicklow and Wexford Railway Co., from Westland Row (Pearse Station) which it leased from the Dublin and Kingston Railway Co., and from Harcourt Street which closed in December 31st 1958.
4. The Great Southern Railway Co., operated from Kingsbridge Station (Hueston Station).
5. The Midland Great Western Railway Co. operated from Broadstone, which closed 16th January 1937 and is currently used as a bus depot.
6. The Northside of the Liffey was connected to the Southside via the City of Dublin Junction Railway "The Loop Line" which had been opened in 1891 connecting Amiens Street with Westland Row.

It was not until the turn of the century that lines had been laid down linking all Dublin termini to facilitate the transfer of rolling stock from one terminus to another.

Railways first came to Dublin in 1834 with the opening of the Dublin and Kingstown Railway which ran from the present day Pearse Station to Salthill near the West Pier of Dun Laoghaire, then called Kingstown. In 1837 the line was extended to the modern day Dun Laoghaire Station, to that part of the Station where today Dun Laoghaire only DART trains and depart from.

With all this rail traffic, there were no major mishaps at any of the Dublin termini until the evening of February 14th 1900 when a steam locomotive went out of control and broke through the end wall of Harcourt Street Station, finishing up hanging up over Hatch Street where it remained in this precarious position until it was taken down. It is one if not the main event associated with the Station.

Uniquely Harcourt Street Station which was opened in 1859 was at the foot of a gradient that sloped downwards from Ranelagh and this meant that locomotive drivers had to exercise care when approaching the station. On the morning of February 14th 1900, Bray engine driver William Hyland and his fireman Peter Jackson, also from Bray, departed from Enniscorty with 0-6-0 locomotive No 17, *Wicklow* built in 1899 by Dublin Wicklow and Wexford Railway Co., at 10.a.m. bound for Harcourt Street Dublin, collecting wagons of cattle en route for the Dublin City Market with the majority of them being collected at Arklow, Co.Wicklow, where a fair had taken place.

When the last wagon was coupled to the train, it numbered 29 wagons, all of which were loose coupled with braking being provided by the locomotive and the guard's van. This was the era before air breaks were in use and locomotive drivers usually gave a series of steam whistle blasts if they wanted the guard to apply his brake in a hurry. All went well until the train passed Ranelagh Station and began its approach to Harcourt Street Station when Driver William Hyland applied the locomotive brakes to slow the train down for a gentle entry into the station after which the wagons would be shunted to the cattle bank for unloading. To the horror and amazement of Driver Hyland, the locomotive began to slide on the tracks, propelled in part by the weight of the unbraked wagons behind the locomotive.

Passengers waiting in the station suddenly saw the cattle train enter the station sliding along the track, at a walking pace, unstoppable, and travel the full length of the platform before striking the stationery buffers at the end of the platform, dislodging them. The locomotive then punched a hole in the station wall before coming to a stop in a raised position. A drayman passing down Hatch Street had a narrow escape from the falling masonry which fell into the street below with his dray sustaining minor damage following the impact of the locomotive with the rear wall of the station. Fireman Peter Jackson, realizing that the train was not going to stop as it entered the station and was going to collide with the buffers and end wall, jumped off the locomotive footplate before the locomotive struck the buffer bank and escaped injury. The Driver William Hyland was not so fortunate as he remained in the locomotive cab right up to the moment of impact when he was thrown clear. But his right arm became trapped between some metal bars at the side of the locomotive coal tender and a wall and was unable to move. At first those in the station though that the locomotive crew were trapped beneath the locomotive or the debris surrounding it but this was found to be untrue.

All the cattle wagons remained upright and on the rails. Immediately the stationmaster sent for medical assistance and also notified various officials of the company of what had happened. Medical assistance in the form of Surgeon Ormbsy, Chief Consultant Surgeon to the Dublin Wicklow and Wexford Railway Company arrived on the scene and was joined by a Dr. McCausland. Company officials who arrived on the scene included Mr. J.W. Pim. Chairman, Mr. Scallon, Deputy Chairman, Mr. Coghlan, Traffic Manager, Mr. Shannon, Chief Engineer, Mr. Cronin, Locomotive Superintendent, and Mr. Hickey, Traffic Superintendent.

Initially it was feared that the locomotive might explode due to the build up of steam pressure, but the safety valve activated and allowed the excess steam to blow off and once a breakdown crew arrived, they drew the fire, opened valves and released all the remaining steam. The locomotive tender, full of coal, acted as a break keeping the locomotive in place. The cattle wagons were uncoupled from the tender and drawn away from it by another locomotive leaving just the tender and raised locomotive in place.

Word of the accident spread all over Dublin and soon a very large crowd gathered in Hatch Street but were kept away from the accident site by members of the Dublin Metropolitan Police who cordoned off the street to spectators. In the station itself only bona fide passengers were allowed to remain in case the locomotive exploded or further quantities of masonry fell into the street. Later, the first task of the break down crew was to prop up the locomotive in a fixed position using timber baulks.

While all activity was in progress, attempts were made to extricate Driver Hyland from his position by the medical and engineering personnel there, and once the metal bars, which had trapped his arm, were broken, he was freed. It was then possible to carry out a medical examination on site, which revealed that his forearm was nearly severed and that he was suffering from severe exhaustion and shock, and required prompt hospitalization if his life was to be saved.

The Dublin fire brigade and one of there ambulances arrived at the scene and Driver Hyland was immediately rushed to the Meath Hospital for treatment and surgery. There a through and detailed examination of his injuries revealed most of his right arm was severely damaged with much of the forearm being crushed. Amputation was seen as the only option available to save his life, but Driver Hyland refused to allow the operation to proceed. A priest summed to administer the Last Rights managed to persuade him that the operation was necessary to save his life. He urged him to follow the advice of Surgeon Ormsby, who advocated the operation as the only option available, and in the end Driver Hyland consented and the operation then proceeded. During the operation, Surgeon Ormsbsy removed Driver Hyland's right arm just below the shoulder and also treated his right foot, which had several broken bones in it. Following the operation Driver Hyland rallied and made good recovery.

Back to the station, the wall had been shored up and made safe. The coal in the tender was unloaded and when this was done a crane was user to lift up the tender and place it back on the rails and it was then moved of the station. In the case of the locomotive, a special frame was built with which it was lowered to the ground and it was then placed on a set of temporary rails laid in Hatch Street and brought along these to the station yard for examination and repair. A large crowd of spectators who were kept behind a barrier witnessed this part of the operation.

A railway accident of this scale was subject to an independent investigation by the Board of Trade with the inquiry being carried out by Colonel von Donop. He formed the opinion that Driver Hyland had miscalculated his speed while approaching the station (steam locomotives at that time were not fitted with speedometers) but was also critical of the station layout which prevented any train directly entering the good's yard. He suggested that pending this being remedied, all trains should stop at the home signal or previous station. While the directors made all trains approaching Harcourt Street station stop at Ranelagh station, a procedure observed right up to the day the station closed, the issue of altering the station layout so as to make direct access from the main line to good's yard possible, was never remedied.

When he recovered from his ordeal Driver William Hyland returned to work with the company and served until the 1930s as a goods checker in Bray. The locomotive *Wicklow* was repaired and returned to traffic, continuing in service until 1925 when it was assigned the No, 440 by the Great Southern Railways and was withdrawn 1929.