

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

OUT: Wed 27 Sep 2006

IN : Wed 4 Oct 2006

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:

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

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

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 $\text{fl}(1.0 + \epsilon_m) > 1.0$. It starts off with $\epsilon_m = \text{epsil} = 1.0$ and repeatedly halves it (base 2) until $\text{fl}(1.0 + \epsilon_m) \leq 1.0$. Thus $\text{fl}(1.0 + \epsilon_m) = 1.0 + \epsilon_m$ 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 $\text{epone} = \{1+1, 1+2^{-1}, \dots, 1+2^k \epsilon_m, \dots, 1+2^2 \epsilon_m, 1+2^1 \epsilon_m, 1+2^0 \epsilon_m\}$. In the final iteration it generates the number $\text{epone} = 1+2^{-1} \epsilon_m$. 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* :

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*.

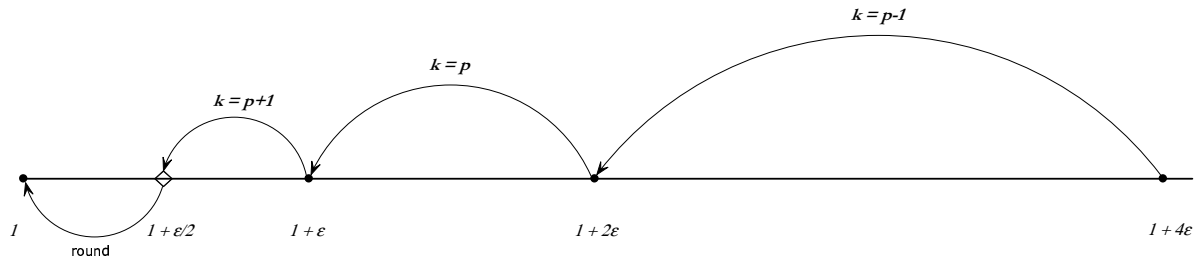


Figure 1: Calculating Machine Epsilon

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 | .100000000000...00 $\times 2^{+1}$ | .1000000000...000000 $\times 2^{+2}$ |
| 2 | .100000000000...00 $\times 2^{+0}$ | .1100000000...000000 $\times 2^{+1}$ |
| 3 | .100000000000...00 $\times 2^{-1}$ | .1010000000...000000 $\times 2^{+1}$ |
| 4 | .100000000000...00 $\times 2^{-2}$ | .1001000000...000000 $\times 2^{+1}$ |
| 5 | .100000000000...00 $\times 2^{-3}$ | .1000100000...000000 $\times 2^{+1}$ |
| 6 | .100000000000...00 $\times 2^{-4}$ | .1000010000...000000 $\times 2^{+1}$ |
| 7 | .100000000000...00 $\times 2^{-5}$ | .1000001000...000000 $\times 2^{+1}$ |
| 8 | .100000000000...00 $\times 2^{-6}$ | .1000000100...000000 $\times 2^{+1}$ |
| 9 | .100000000000...00 $\times 2^{-7}$ | .1000000010...000000 $\times 2^{+1}$ |
| 10 | .100000000000...00 $\times 2^{-8}$ | .1000000001...000000 $\times 2^{+1}$ |
| ⋮ | ⋮ | ⋮ |
| 50 | .100000000000...00 $\times 2^{-48}$ | .1000000000...01000 $\times 2^{+1}$ |
| 51 | .100000000000...00 $\times 2^{-49}$ | .1000000000...00100 $\times 2^{+1}$ |
| 52 | .100000000000...00 $\times 2^{-50}$ | .1000000000...00010 $\times 2^{+1}$ |
| 53 | .100000000000...00 $\times 2^{-51}$ | .1000000000...00001 $\times 2^{+1}$ |
| 54 | .100000000000...00 $\times 2^{-52}$ | .1000000000...00000 $\times 2^{+1}$ |

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 `tab:MachParms`.

The number $1/\text{realmin} = 4.494232837155790\text{e}+307$ is in the Floating Point number range but the number $1.0/\text{realmax} = 5.562684646268004\text{e}-309$ 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*.

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 of the resistors is (i) 0 (short circuit) or (ii) ∞ (open 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;
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;
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.

3.4 Kahan's Machine Epsilon

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

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

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

$$1. \quad a = \text{fl}(4/3) = \text{fl}(1.33\dots33\dots) = \underbrace{1.33\dots3}_{p \text{ digits}}$$

$$2. \quad b = \text{fl}(\text{fl}(a) - 1) = \text{fl}(1.33\dots3 - 1.0) = \underbrace{0.33\dots3}_{p-1}$$

$$3. c = \text{fl}(b + b + b) = \text{fl}(0.33\dots 3 + 0.33\dots 3 + 0.33\dots 3) = 0.\underbrace{99\dots 9}_{p-1}$$

$$4. e = \text{fl}(1 - c) = \text{fl}(1.0 - \underbrace{0.99\dots 9}_{p-1}) = \underbrace{0.00\dots 1}_{p-1} = 1.00\dots 0 \times b^{1-p}$$

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.

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. $\sin(\text{pi}) = 1.224646799147353 e - 016$
2. $\cos(\text{pi}) = -1$
3. $\sin(\text{pi})^2 + \cos(\text{pi})^2 = 1$

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 :

```
3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117067
11.0010010000111111011010101000100010000101101000110000100011010011000100110001100110001010001011100000
0011011100000111001101000100101001000000100100111000001000100010100110011111001100011101000000001000
0010111011111010100110001110110001001110011011001000100101000101001010000010000111100110001110001101
00000001001101110111101101000110011011001111001101001110100111010010000110001101100110000010101100
00101001101101111100100101111100010100001101110011
```

MATLAB gives the value of $\text{pi} = 3.141592653589793(2384626433832795028)$ 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 $\text{pi} \neq \pi$

| | | |
|---------------------------|-------------------|---|
| $\text{pi}-2^*\text{eps}$ | 3.141592653589793 | $+.11001001000011111101101010100010001000010110100010111 \times 2^{+2}$ |
| pi | 3.141592653589793 | $+.11001001000011111101101010100010001000010110100011000 \times 2^{+2}$ |
| $\text{pi}+2^*\text{eps}$ | 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 $\text{pi}+2^*\text{eps}$. Hence the error in pi is

$$e_\pi = \pi - \text{pi} = 3.141592653589793 2384626433832795028\dots - 3.141592653589793 = 2.384626433832795028\dots \times 10^{-16}$$

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

$$\sin(x) = x - \frac{x^3}{6} + \dots \approx x \quad \text{for small } x$$

$$\cos(x) = 1 - \frac{x^2}{2} + \dots \approx 1 \quad \text{for small } x$$

Using the series approximations above we get

$$\sin(\pi) = \sin(\pi - e_\pi) = +\sin(e_\pi) \approx e_\pi = 2.384626433832795028 \times 10^{-16}$$

$$\cos(\pi) = \cos(\pi - e_\pi) = -\cos(e_\pi) \approx -1$$

These are very close to the answers MATLAB gives. See 4 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 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.*

Question 1. In the MATLAB functions `NormS` and `NormCS` in Section 3 above, how should the values for `LARGE`, `SMALL`, `LScale`, `SScale` be chosen? What do they depend on?

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 = b1-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 Calculations with $\sin(x)$ and $\cos(x)$ **A Windows 2006 Problem.**

From E. Heffernan, Brusselstown Windows, Co. Wicklow, May 2006.

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 ± 5 mm.

Solution.

1. I calculated the angle θ on a 10-digit calculator as $\theta = \tan^{-1}(200/2873) = 3.982143737^\circ$.
2. Given that the measurements are accurate to within ± 5 mm 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

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

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

Moral.

Check your answers. Nature will anyway.

Unintended Learning Outcome.

You may wind up installing double glazing for a living

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 = 2879.952951004581949679215264
>> 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 |

Note that the rounded angle is $\theta \approx 4.0^\circ = 0.06981317007977318307694763074$ rads

The unrounded angle is $\theta = 0.06950151949217482078904047120$ rads

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

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

Important Tip for Aspiring Masters of Computational Science.

Check whether your calculator is in the *degree* or *radian* mode