ECE3340
Introduction to the limit of computer numerical capability – precision and accuracy concepts

PROF. HAN Q. LE

Computing errors

…~ 99 – 99.9% of computing numerical errors are likely due to user's coding errors (syntax, algorithm)…

and the tiny remaining portion may be due to user's lack of understanding how numerical computation works…

Hope this course will help you on this problems

Homework/Classwork 1

CHECK OUT THE PERFORMANCE OF YOUR COMPUTER

CW

Click open this APP

UNIVERSITY of HOUSTON App by Han Q. Le ©

ECE 3340 – APP 1.0.0.1 – Test your computer precision

RUN STATUS/CONTROL

Click to ask your machine the smallest number it can handle (this example shows 64-bit x86 architecture)

Log base 2 of the number

Basic check of your machine floating–point precision

	Decimal	Log2	Number of exponent bits
Smallest machine–handled number	2.22507×10^{-308}	1022	10
Largest machine–handled number	1.79769×10^{308}	1024	10
Smallest relative difference between2 machine numbers	2.22045×10^{-16}	52	

Do the same for the largest number

Click to ask the computer the smallest relative difference between two number that it can tell apart. This is called **machine epsilon**

If you use an x86 64-bit CPU, the above is what you get.

HW-Problem 1

Use any software (C++, C#, MATLAB, Excel,…) but not Mathematica (*because it is too smart and can handle the test below*) to do this:

1. Let's denote xmax be the largest number your computer can handle in the APP test. Let x be a number just below xmax , such as ~0.75 xmax. (*For example, I choose x=1.5*10^308*). Double it (2*x) and print the result.
2. Let xmin be your computer smallest number. Choose x just above it. Then find 0.5*x and print output.

Here is an example what happens in Excel:

Number just slightly less than my computer xmax

Excel error: instead of giving the correct answer 3*10^308, it gives error output. This is caused by **overflow**

Enter a number	Column A*2.	Column A/2.
4	8	2
5	10	2.5
1.50E+308	**#NUM!**	7.50E+307
6.00E-308	1.20E-307	3.00E-308
5.00E-308	1.00E-307	2.50E-308
4.00E-308	8.00E-308	**0.00E+00**

Note the three key concepts highlighted in yellow

Number just slightly more than my computer xmin

Excel fails: instead of giving me the correct answer: 2*10^-308, it "**flushes to zero**" by giving zero output.

This is caused by **underflow** : a result that is too small for the computer to handle, it sets as zero.

You should get similar results in other software and language.

HW-Problem 2

Use any software (C++, C#, MATLAB, Excel,…) but not Mathematica (*because it is too smart and can handle the test below*) to do this. Let denote eps (for ε) be the smallest relative difference that your computer can handle – aka **machine epsilon** (*it is 2.22*10^-16 on my machine, for example*).

Then, generate an array of 5-20 elements (your choice), with values ranging from above your computer eps to below eps. Denote this array as x array.

Then, add 1 to x array, denote it as y: y=1+x;

Then, define x0 array: x0=y-1;

Print out all 3 arrays: x, y, and x0 and compare. Is your x0 the same as x?

Here is an example what happens in Excel:

ε array

y=1+ ε array

ε0=y-1 array

1	small number	column A+B	Column C-A
1	5.000E-15	1.000000000000010000000E+00	5.107025913275720000000E-15
1	4.000E-15	1.000000000000000000000E+00	3.996802888650560000000E-15
1	3.000E-15	1.000000000000000000000E+00	3.108624468950440000000E-15
1	2.000E-15	1.000000000000000000000E+00	1.998401444325280000000E-15
1	1.900E-15	1.000000000000000000000E+00	1.998401444325280000000E-15
1	1.800E-15	1.000000000000000000000E+00	1.776356839400250000000E-15
1	1.700E-15	1.000000000000000000000E+00	1.776356839400250000000E-15
1	1.600E-15	1.000000000000000000000E+00	0.000000000000000000000E+00
1	1.500E-15	1.000000000000000000000E+00	0.000000000000000000000E+00

Notice that ε and ε0 are not equal as expected.

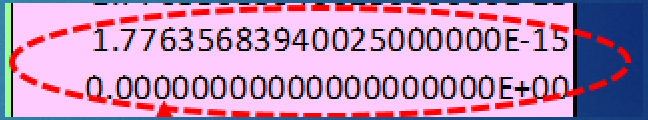
What we see here is the limit of **machine precision** which causes the **inaccuracy** observed.

what happens right here? why ε0 cannot gradually go from 1.776*10^-15 to ~ 1.6*10^-15 as expected, but jumps to zero?

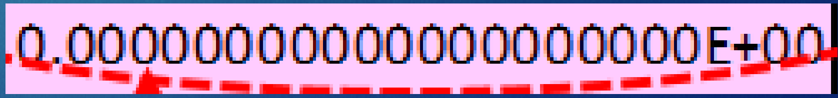
imagine this: there is an investment fund containing gazillion dollars. Asset= 1 gazillion

You add to the account your saving of $1 mil, because it promises 50% profit return in 1 year

but the investment company uses Excel on 16-bit CPU and your portion is below its precision, hence, it flushes to zero



Hence, after one year, your account is deposited with this amount $



At least, it has a lot of zeros!

All is not lost, the company kindly sent you a card…

A different kind of precision problem: Sometimes, digits are lost on the other side (overflow)

Problem 3

You don’t have to do any calculation, only discuss what you think here:

1. Describe what you observe from doing problems 1 and 2 above and what is your thought on the results? (*in other words, try your best to explain what’s wrong*).
2. What limits a computer to have a maximum and minimum magnitude for numbers? (*this is limited by machine processor and not by software like Mathematica*).
3. If a computer can handle a number as small as 3*10^-308, why can’t it handle a difference between 1 and 1+eps when eps is 1*10^-16, which is >> than the smallest number 10^-308 that it can handle?

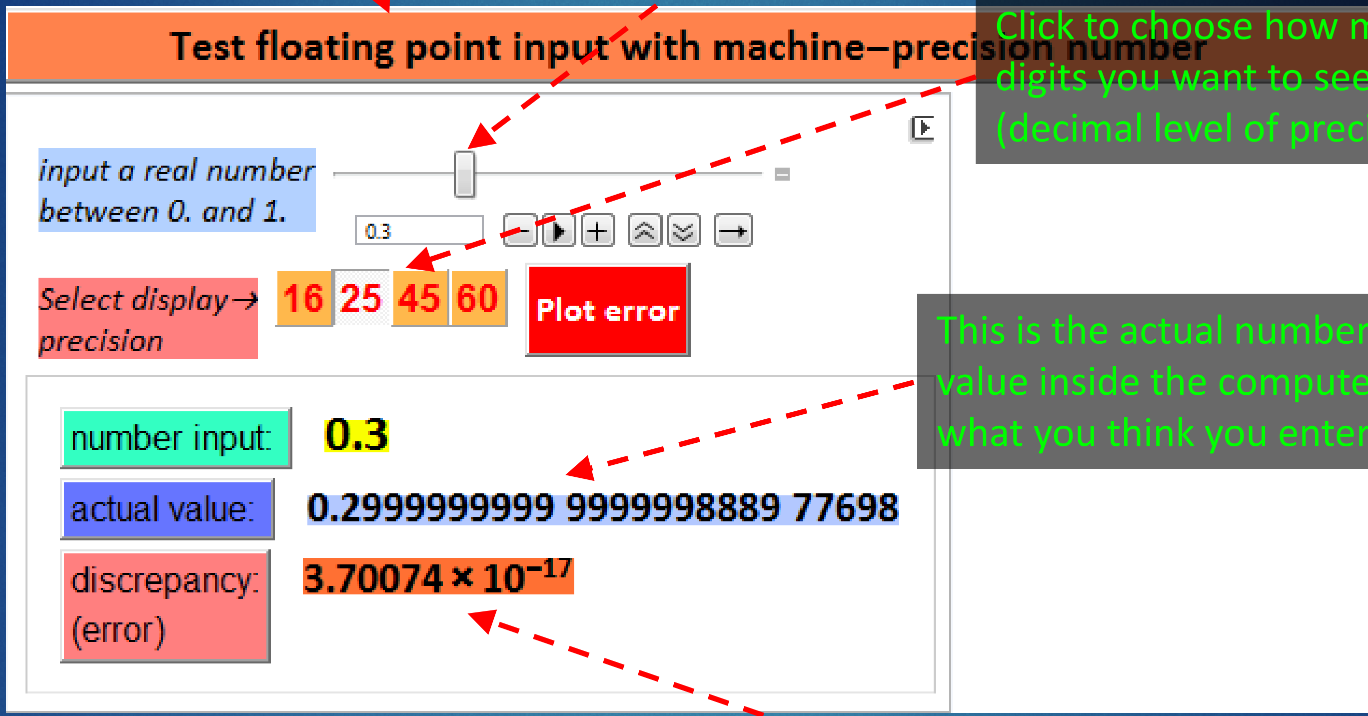
this problem is aka “how your million dollars flushed to zero”

$ 0.00000000000000000000E+00

Click open this tab of another APP.

Use slider to input a number

Click to choose how many digits you want to see (decimal level of precision)



This is the actual number value inside the computer, not what you think you enter.

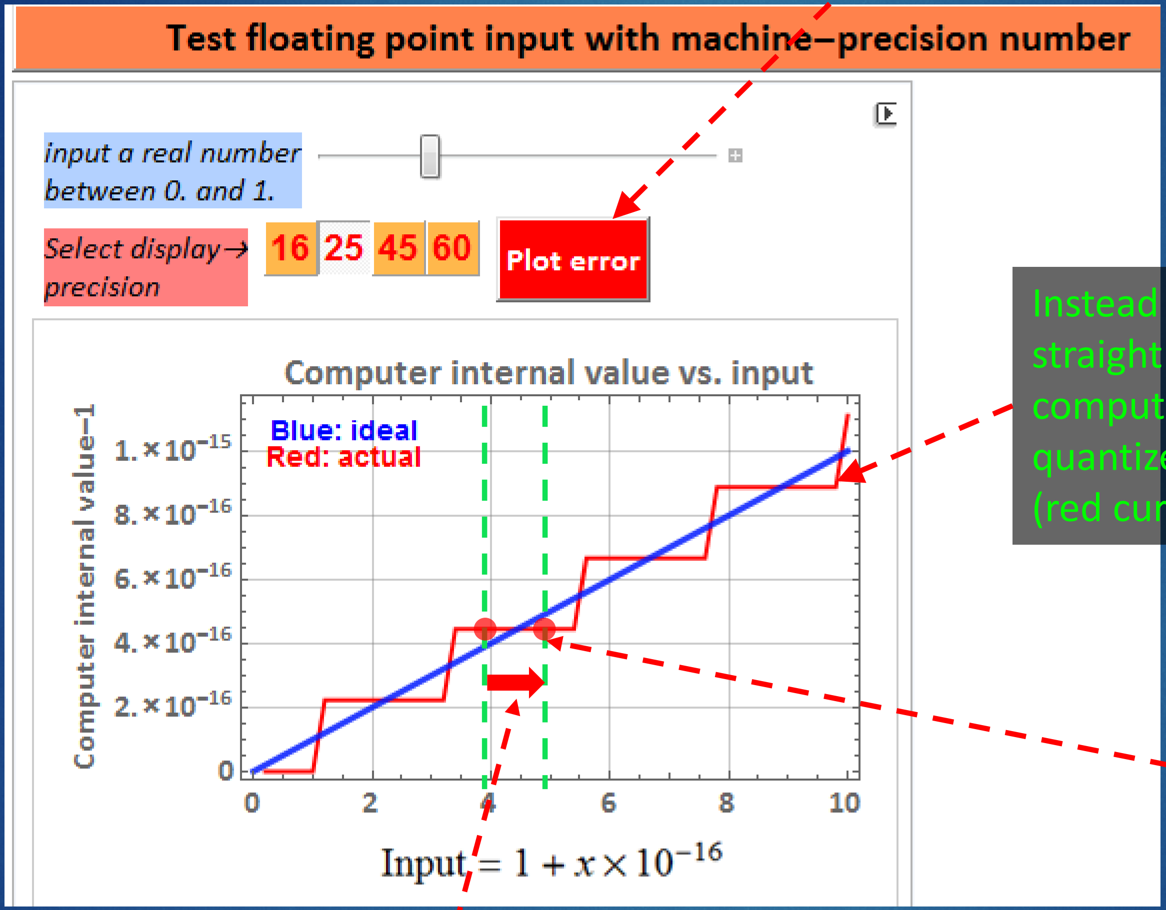
This is the error between what you want to input (for example. 0.3 here), and what it really is (0.2999 whatever…). Why can't it just record something as simple as 0.3?

Problem 4

Use the APP above to enter 3-5 non-zero numbers (*the example is 0.3, but you should enter a set of unique numbers for yourself, e. g. 0.155, 0.291, 0.43, 0.68, 0.912 ← don't copy this, please – pick your own unique values*). Record what the actual numbers are, and the errors.

1. Make a table (like in Excel) a column of numbers you enter, a column of their actual values, and a column of ratio abs(error)/number – this is called **relative error** (*for example, ratio abs(3.7*10^-17)/0.3 = 1.23*10^-16*)
2. Enter a few values that are multiple of 1/2^n, where n=1, 2, 3. For example 0.125, 0.375, etc. Do you see any errors?
3. Write why you think there are errors? Why the magnitude of relative errors is ~ the machine epsilon you find in APP 1. Why do you see errors in question 1, but the errors in question 2 are zero (which means the numbers inside the computer are exact).

Click this to see an error plot. The input varies from 1+0*10^-16 to 1+10^10-16 = 1+1*10^-15



Instead of giving us a linear straight line like the blue line, the computer actual values are quantized into discrete step values (red curve). Why?

The computer can't tell the difference and hence, it returns to you zero.

you deposited $1 mil

machine precision problem

Problem 5

Calculate or estimate the quantized step size from the APP for your computer (*not this demonstrated computer – although very likely you have exactly the same result if you have x86 64-bit CPU*).

Then, approximate your step size as $\frac{1}{2^n}$ and obtain the integer value *n*, (*you can take log2 of the step size to get -n*). This is the number of **significant bits** of your machine **floating point mantissa**.

From here on, you can use Mathematica. Example of the code is given here.

```
Log[2, "your number"]
```

```
In[10]:= Log[2, 2.22044 * 10^-16]

Out[10]= -52.
```

We have 52- bit mantissa

wait a minute, I thought the machine is 64-bit. Why the mantissa has only 52 bits? what happens to the other 12 missing bits? someone takes it?

Let's take a break here and ponder why?

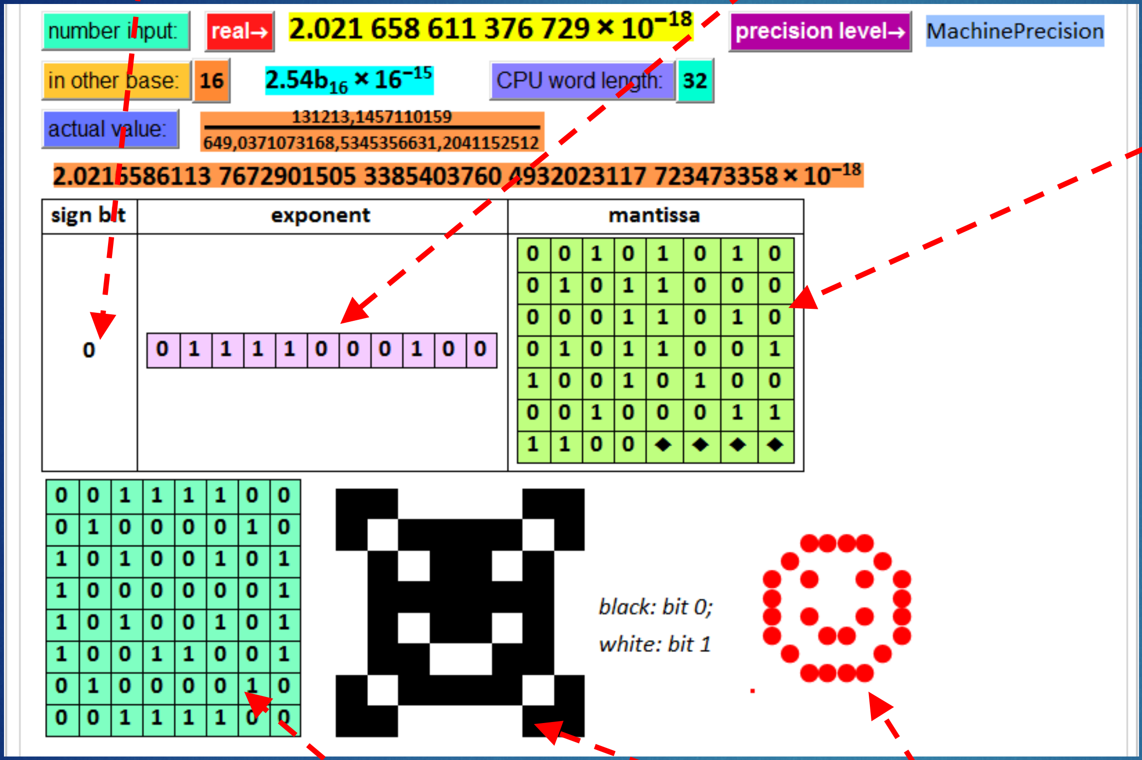
The reason will be discussed in details in the next lecture. Meanwhile, the next 2 slides give a preview of "why"

Slide from next lecture

If the input is a FP number, it will show:
- 1 sign bit: 0 for >= 0 and 1 for <=0

- 11 exponent bits

- 52 bits for the mantissa (which actually has 53 bits. The first bit is always 1 and needs not be stored here).



total is a 64-bit word – shown here as 8 bytes

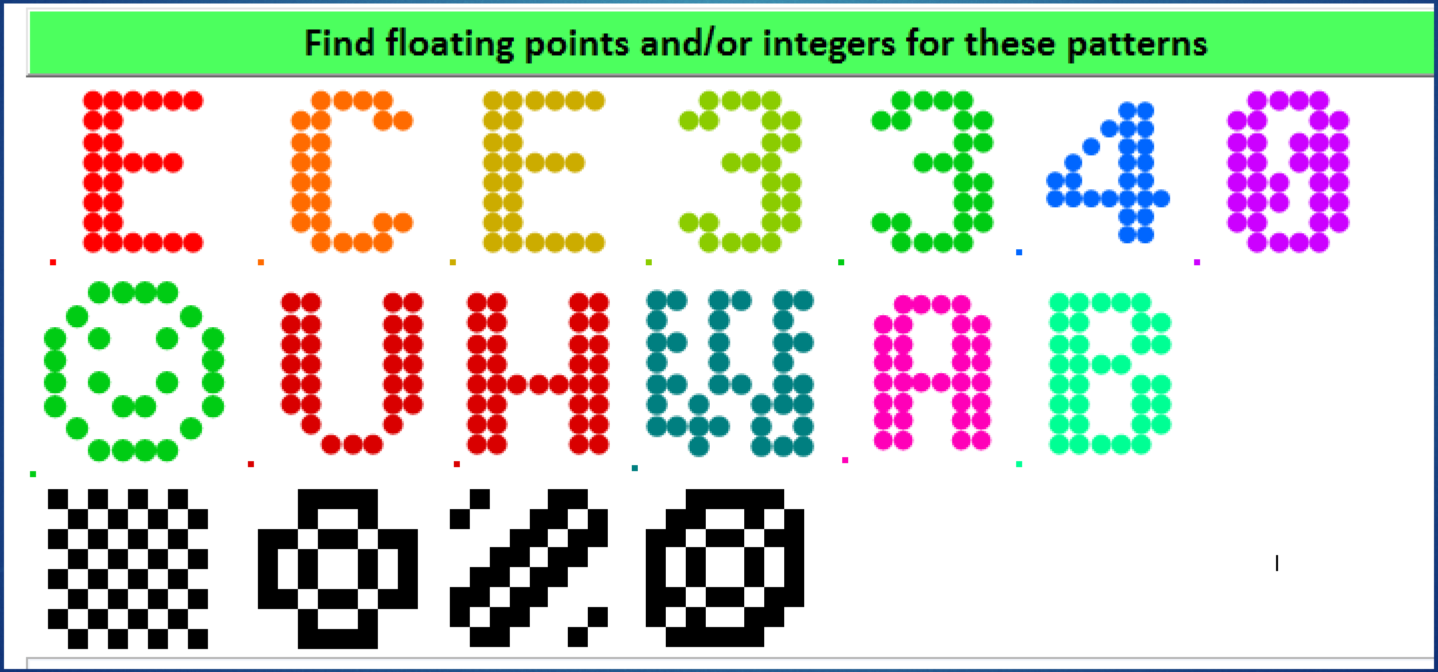
The 8 bytes can be arranged in 8x8 dot matrix display

Problem 3 (bonus)

Find floating points or integers (easier) that have bits patterns representing the initials of your first and last name.

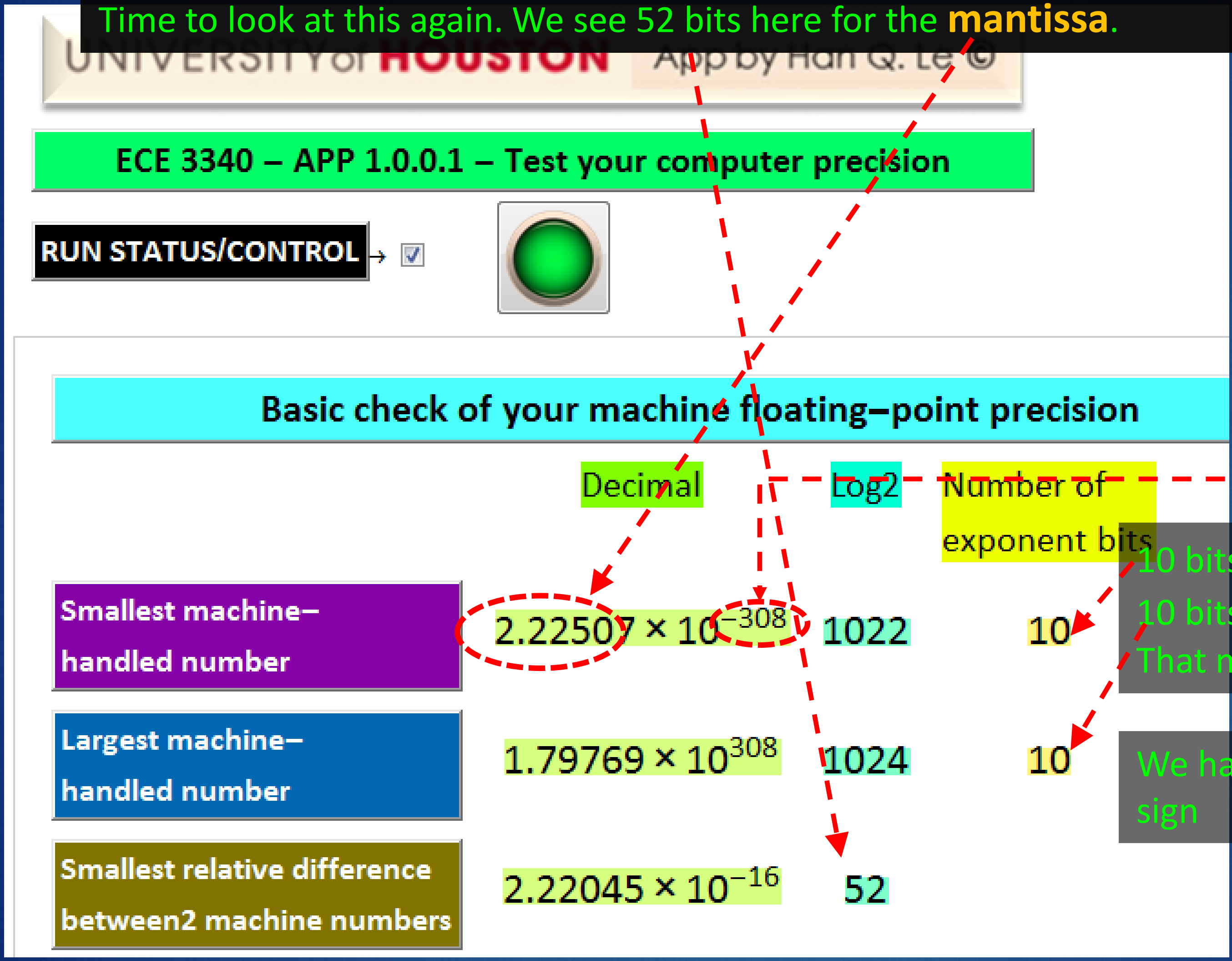
Example: 16,419,452,012,919,037,084 and 4,123,389,611,252,201,785 will give something. Check it out.

Below are illustrations of common patterns: each has a number. Find your "digital signature" with your name initials.



Back to our current lecture…

Time to look at this again. We see 52 bits here for the **mantissa**.



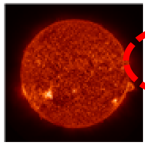
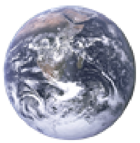
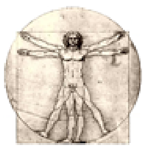
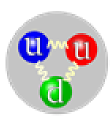
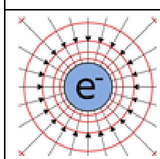
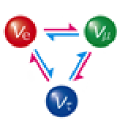
10 bits for negative **exponent**.
10 bits for positive **exponent**.
That means 11 bits for both.

We have 1 bit left. It is for the sign

Mantissa and Exponent

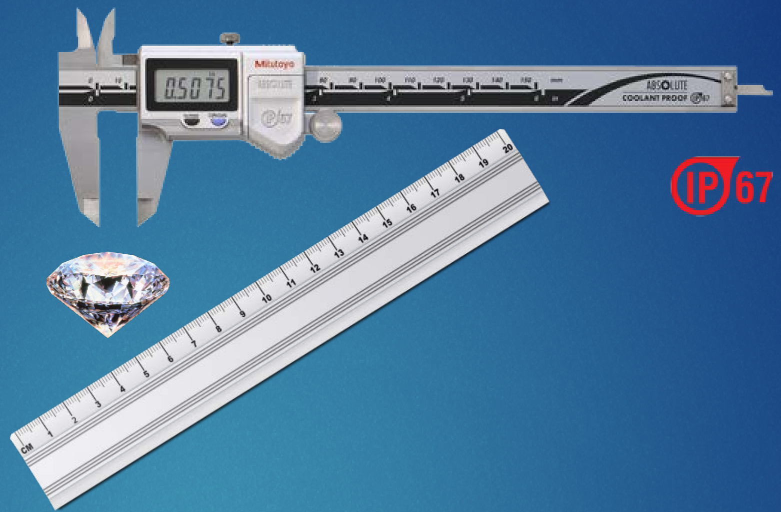
SCIENTIFIC REPRESENTATION OF NUMBERS

mantissa exponent save the hassle of all these zeros

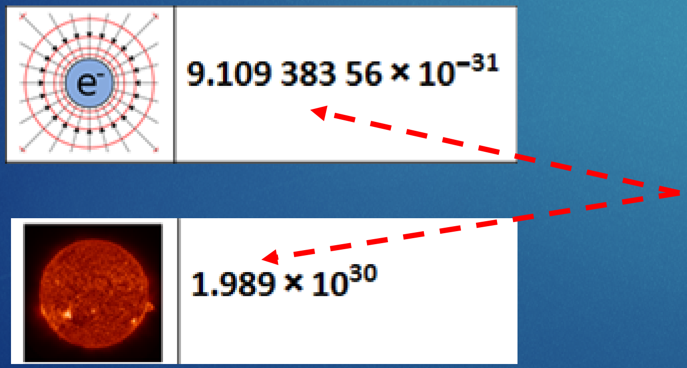
	mass (kg)	mass (kg)
	1.989×10^{30}	1,989000,000000,000000,000000,000000.
	5.972×10^{24}	5,972000,000000,000000,000000.
	65.	65.
	$1.672\,621\,9 \times 10^{-27}$	0.000000 000000 000000 000000 001672 6219
	$9.109\,383\,56 \times 10^{-31}$	0.000000 000000 000000 000000 000000 910938 356
	5.7×10^{-37}	0.000000 000000 000000 000000 000000 000000 57

electron mass and sun mass are 60 orders of magnitude different. But relatively speaking, which mass do we know “better” or more **precise**?

which tool would we want to use to measure this diamond?



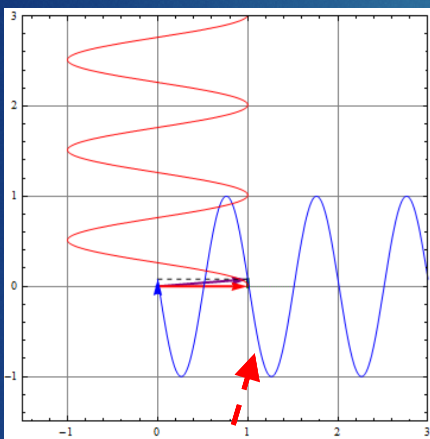
The caliper has higher precision, as it can give us a finer, or more-digit reading of the size: precision means the ability to give high resolution, more significant digit reading.



We know the electron mass with more **precision** than the Sun mass: The number of significant digits (with respect of measurement uncertainty) of the **mantissa** is the determinant of **precision**. The exponent is not relevant.

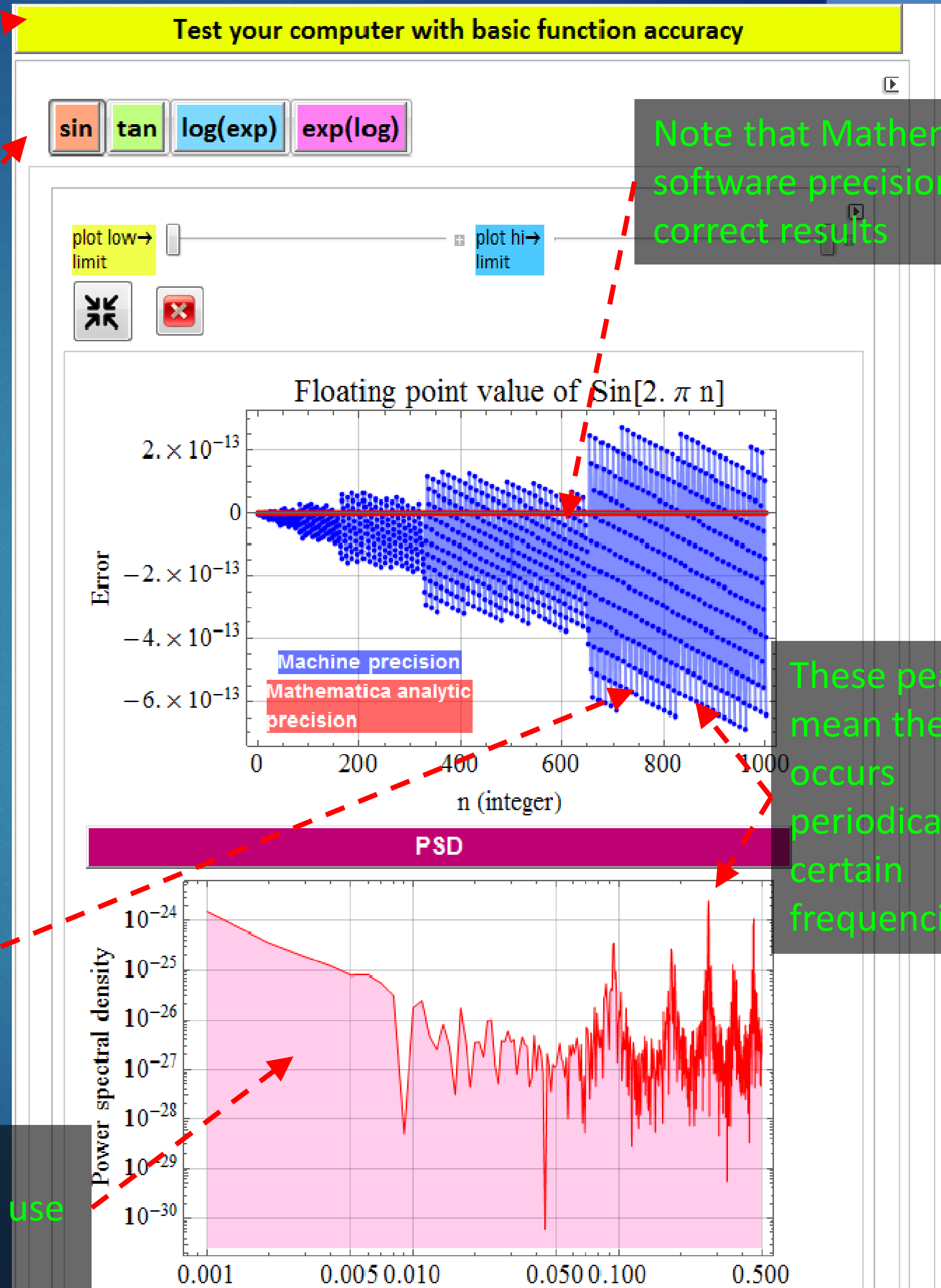
Click open this tab of another APP.

Here, we test how the computer computes these four functions. The 1st one we test is sin(2 π n), where n is an integer from 0 – 1000.



We expect sin(2 π *n*)=0 for all *n*. But we see here that is not. The error gets worse with larger *n*.

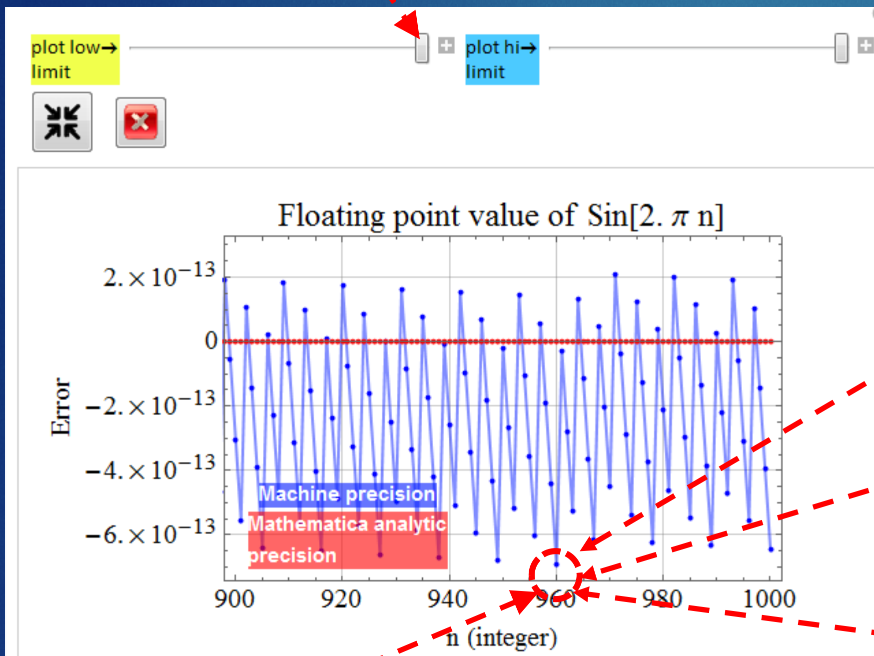
This chart is known as power spectral density (PSD) plot. We'll use it a lot later in the course.



Note that Mathematica software precision gives correct results

These peaks mean the error occurs periodically with certain frequencies.

grab this slider, zoom in for a close look of the range n from 900-1000



2*960	PI	SIN(B*C)
1920	3.1415927	-6.89975854228919OE-13

```
>> format long
>> sin(2*960*pi)

ans =

  -6.898795363227559e-13
```

MATLAB

In[84]:= Sin[960 * 2. * π]

Out[84]= -6.8988×10^{-13}

Wolfram Mathematica 9

In[85]:= NumberForm[%, 16]

Out[85]//NumberForm= $-6.898795363227560\times10^{-13}$

In[86]:= Sin[960 * 2 * π]

Out[86]= 0

Let's look at this point n=960.

All three software yield comparable results -6.8987953.. x10^{-13}, with plenty of digits for **precision**, but **not accurate**!

Note: Mathematica analytic calc does yield correct result: 0.

This error is huge compared with machine epsilon and machine smallest number.

To really see this problem, do HW problem 6

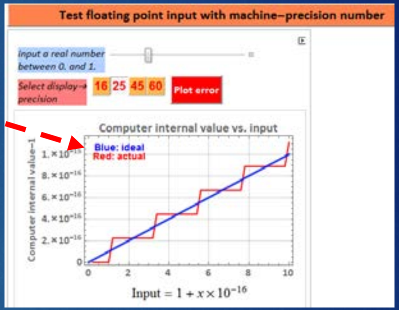
Problem 6

Calculate and plot y=sin(2 π n+x) for:

- n=0, 2000, 4000, 8000, 16000;
- x from -10^-11 to 10^-11

using any software you like, including Mathematica, but not the analytic (arbitrary precision) option. If you use Matlab, you can generate a C++ code and it is the same as writing in C++.

Discuss you results in terms of what you learn in this APP.



Also, plot this:

$$yb = \sin\left(n2\pi + x - 2\pi\, floor\left(\frac{n2\pi+x}{2\pi}\right)\right)$$

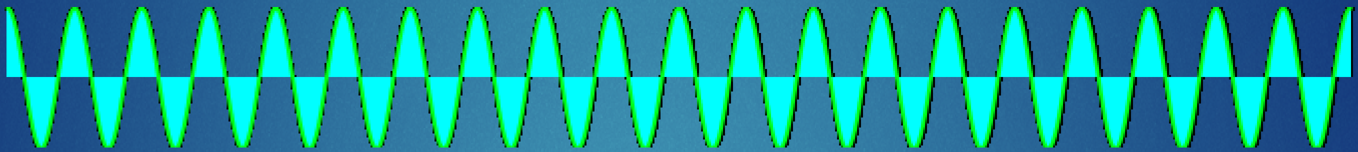
and compare with the above result, discuss.

floor() is a function that take the lower integral value of an integer. For example, floor(3.2)=3, which the integer immediately below 3.2. floor(4.8)=4. etc. One can also use the mod-function instead:

$$yb = \sin(mod(n2\pi + x, 2\pi)\,)$$

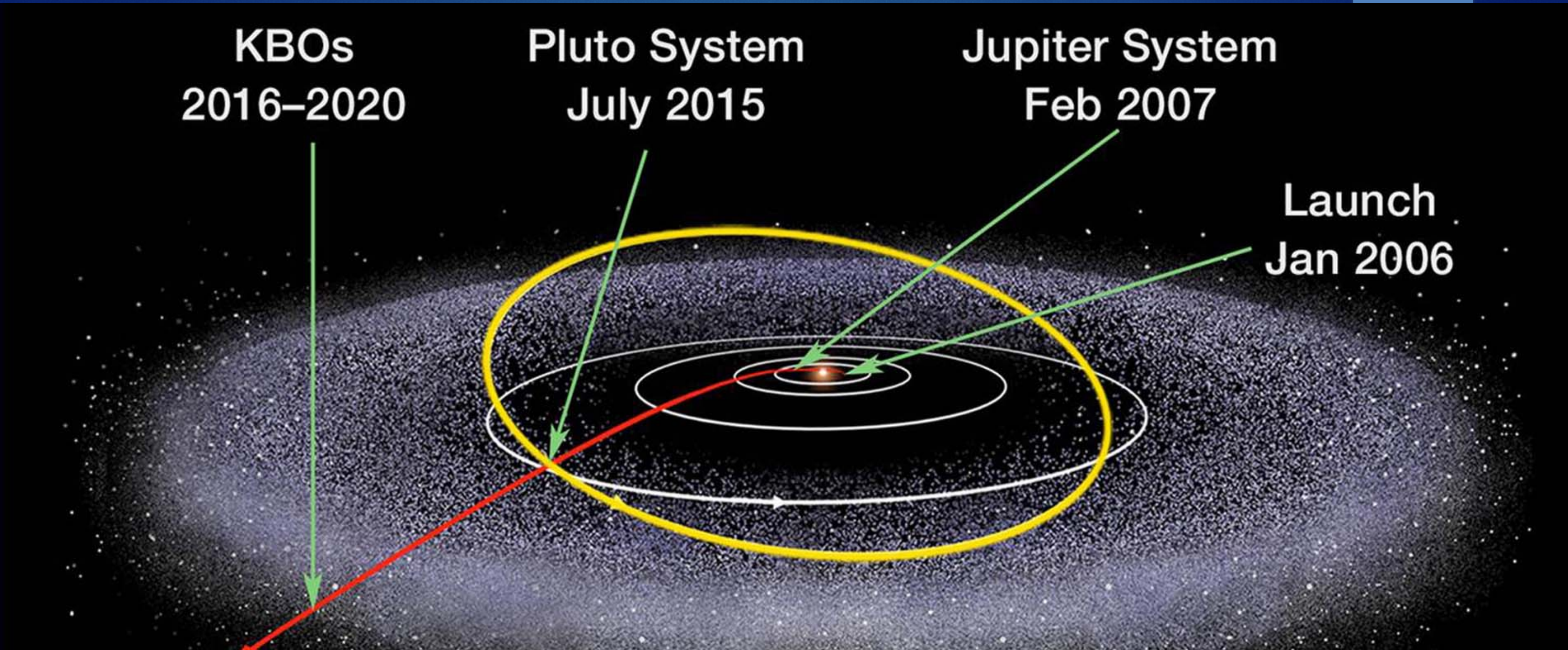
if the software has this function that works reliably (some may not).

Before we get into this, you may wonder, why do we care about $y = \sin(n2\pi + x)$?
What does that have to do with electrical engineering?



What does this remind you? An electromagnetic wave!

$E = \mathrm{A}\sin\left(2\pi\left(\frac{x}{\lambda} - ft\right)\right)$ How large is $\frac{x}{\lambda}$?



New Horizon uses X-band frequency, wavelength λ ~ 3.75 cm, to send images from Pluto which is at a distance x~ 7.5 billions km = 7.5*10[??] cm? (you figure the **??** out).

What is the ratio $\frac{x}{\lambda}$ of its EM wave when it reaches Earth?

$$E = A\sin\left(2\pi\left(\frac{x}{\lambda} - ft\right)\right)$$

The Globe is crisscrossed with million miles of optical fibers that conduct lightwave signals for the Internet and virtually all our communication needs.

Consider a lightwave with $\lambda \sim 1\ \mu\text{m}$ in a typical optical fiber short span of 100 km, what is the ratio $\frac{x}{\lambda}$ at the end of a span?

$$E = \mathrm{A}\sin\left(2\pi\ \left(\frac{x}{\lambda} - ft\right)\right)$$

Problem 6

Calculate and plot y=sin(2 π n+x) for:

- n=0, 2000, 4000, 8000, 16000;
- x from -10^-11 to 10^-11

Back to Prob. 6: it is not uncommon for us to write code:

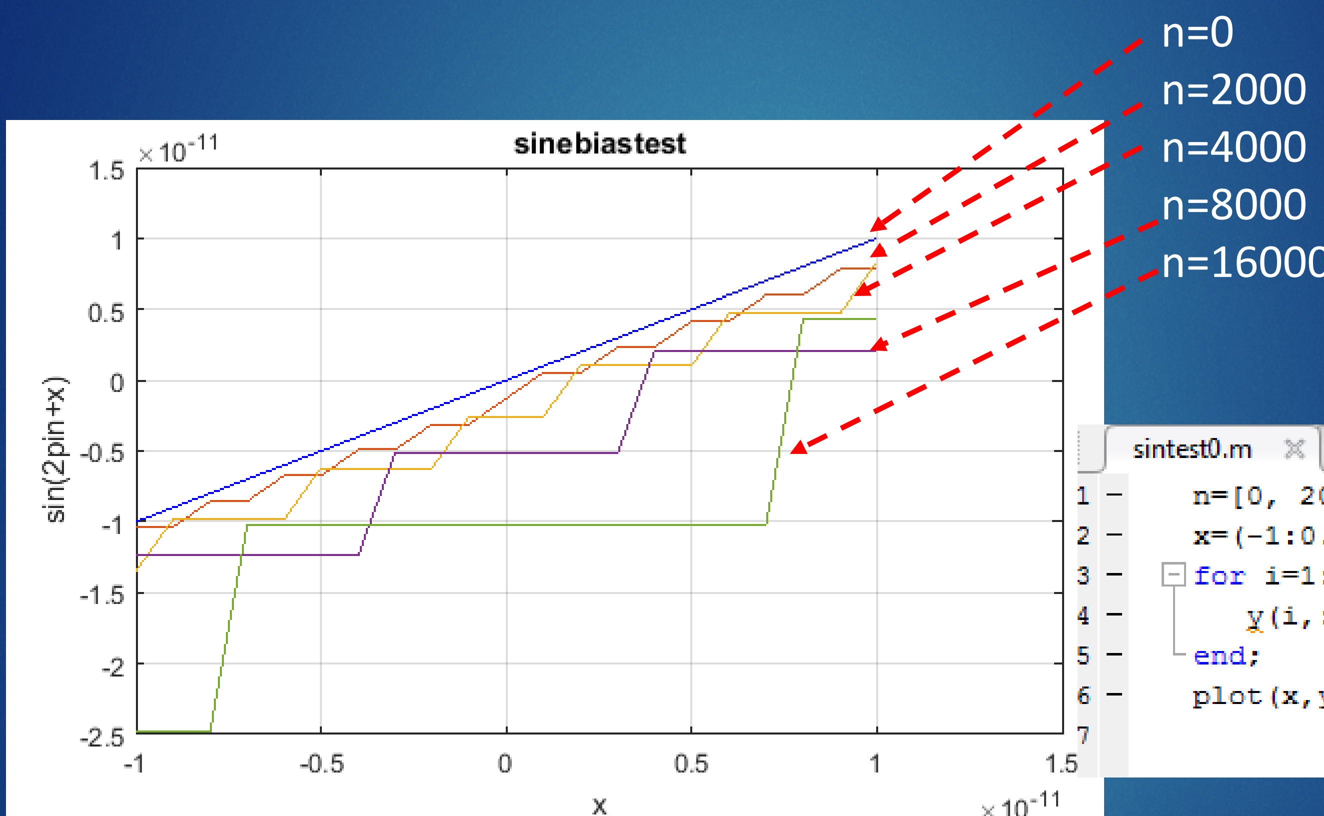
$$y = \sin\left(2\pi\frac{x}{\lambda}\right) \text{ (or } \sin(kx) \text{)}$$

without thinking how large $2\pi\frac{x}{\lambda}$ or kx can get.

The art of scientific/engineering coding is not to let the argument get out of hand for being too large (overflow) or too small (underflow). Hence, we use $\mathrm{mod}(kx, 2\pi)$ to ensure a small argument – or know where to re-choose the axis origin for the relevant problem.

But let's say we are careless, do the HW and see what you get.

Example: this is what one gets with MATLAB

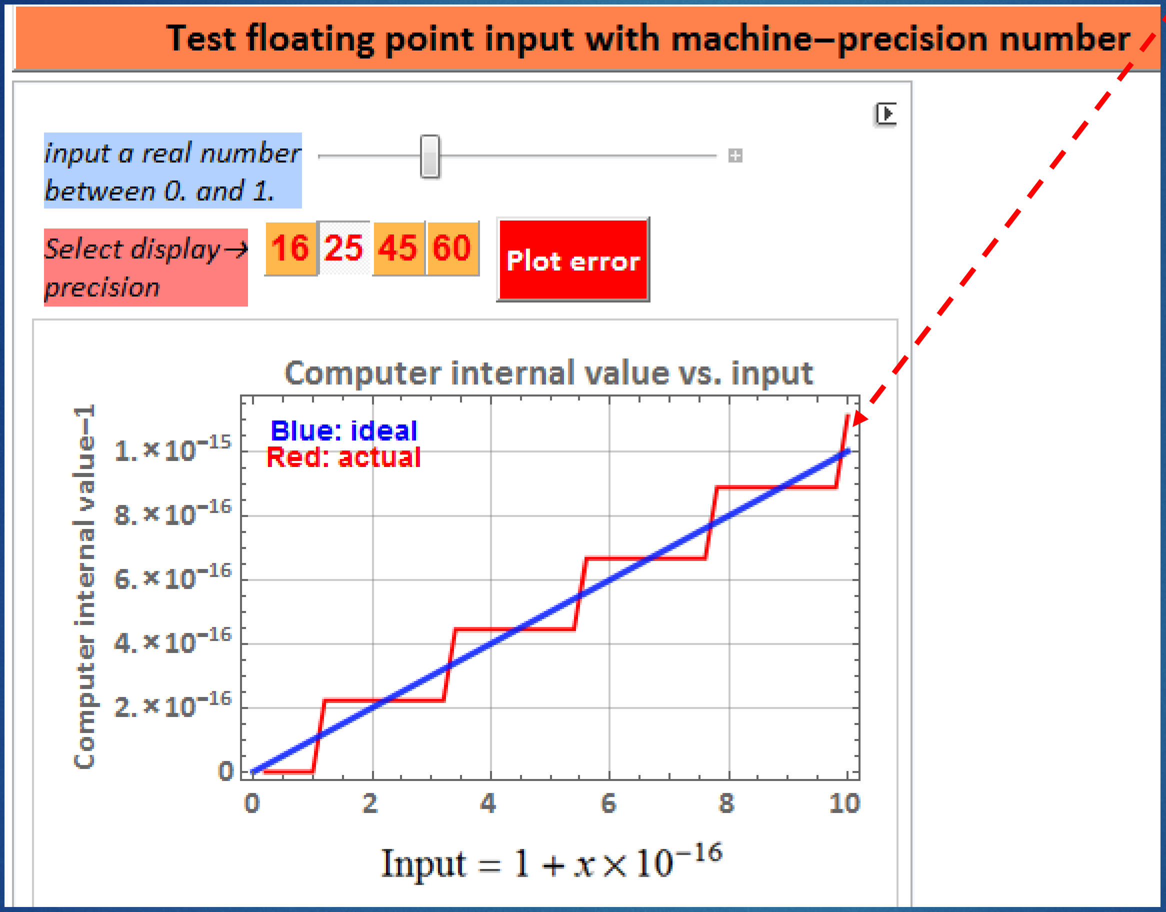


sintest0.m

```
n=[0, 2000, 4000, 8000, 16000] ;
x=(-1:0.1:1)*10^-11;
for i=1:5
    y(i,:)=sin(2*pi*n(i)+x);
end;
plot(x,y)
```

What are the problems here?

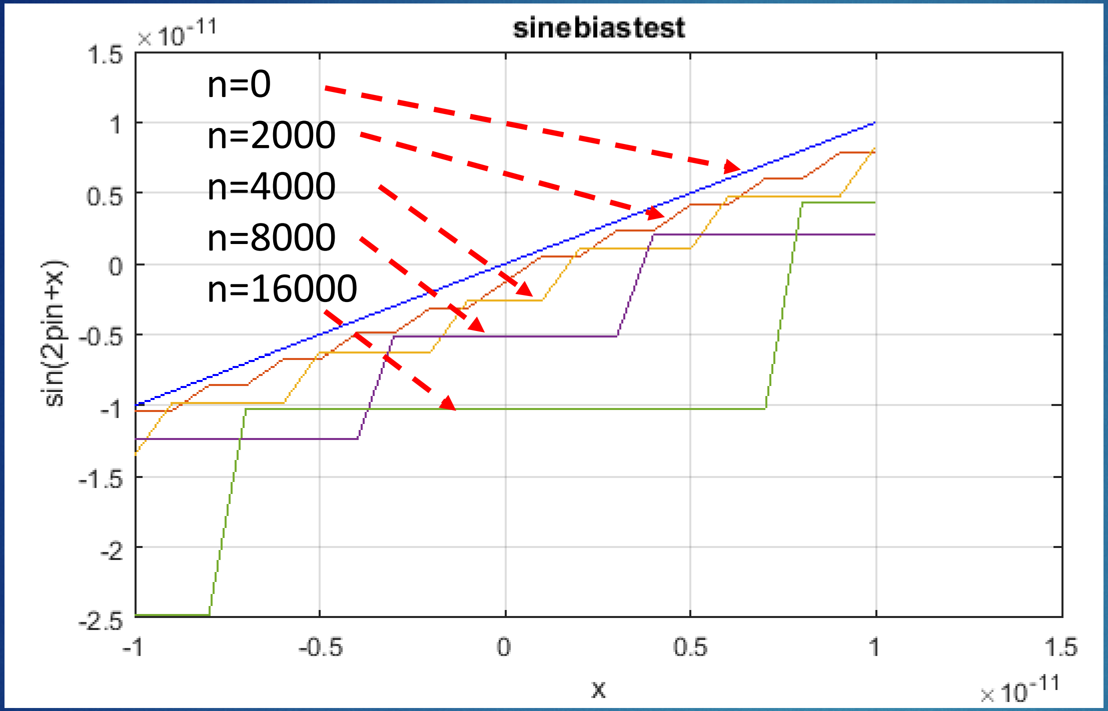
The two problems are different: precision and accuracy



machine precision problem

It can't have enough precision, but at least, it tries to be **accurate**: close to the correct value in blue

Example: this is what one gets with MATLAB



The step quantization is due to the input limited precision

The precision error is magnified <- this is analogous to what is known as "feedback error"
We can also think of it as propagated & magnified errors: GIGO

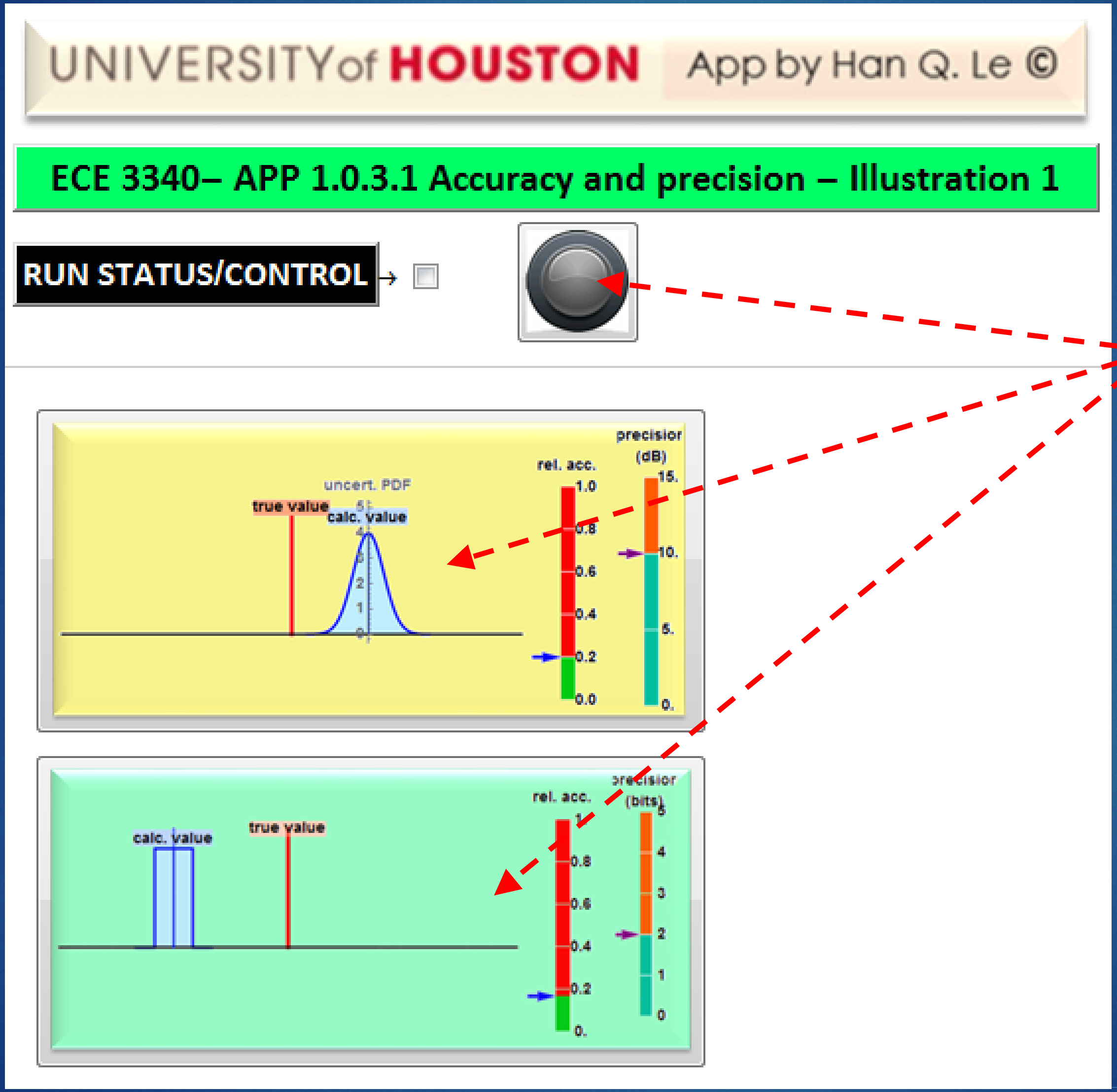
But we see another troublesome type of error: systemic bias that makes the calculation increasingly **inaccurate**: even if we try to fix the quantization error by using a fitting line, we still have **accuracy** problem with a bias.
In the second part of problem 6, you will see that the bias can be removed.

Imagine you go to a showroom and step on these scales for sale. All are **precise** down to 0.1 lb. But they give readings differing by 10's lbs as above! What can you say?

At least 3, if not all are wrong. This means they are **precise**, but terribly **inaccurate**. (Well, may be the first one is the correct one?)

Inaccuracy in instruments – especially high precision, are usually caused by erroneous calibration or some incorrectly adjusted bias.

In empirical science, **accuracy** is defined as the degree that a measurement is close to the true value relative to uncertainty. Similarly in computing, **accuracy** is determined by the magnitude of error: the discrepancy between a calculation and the <u>known correct value</u> (via analytical knowledge).



Click open this APP for problem 7 (any button)

Problem 7

Use the APP 1.0.3.1 on accuracy and precision to obtain the follow cases for each category: empirical measurements and numerical calculations. Show a case for:

- low accuracy, <= 0.5, low precision, <= 2.5 dB or LSB=bit 0
- high accuracy, =1, low precision,<= 1 dB or bit LSB=bit 0
- low accuracy<= 0.1, high precision>= 12 dB or LSB=bit 4
- high accuracy=1, high precision >= 12 dB or LSB=bit 4

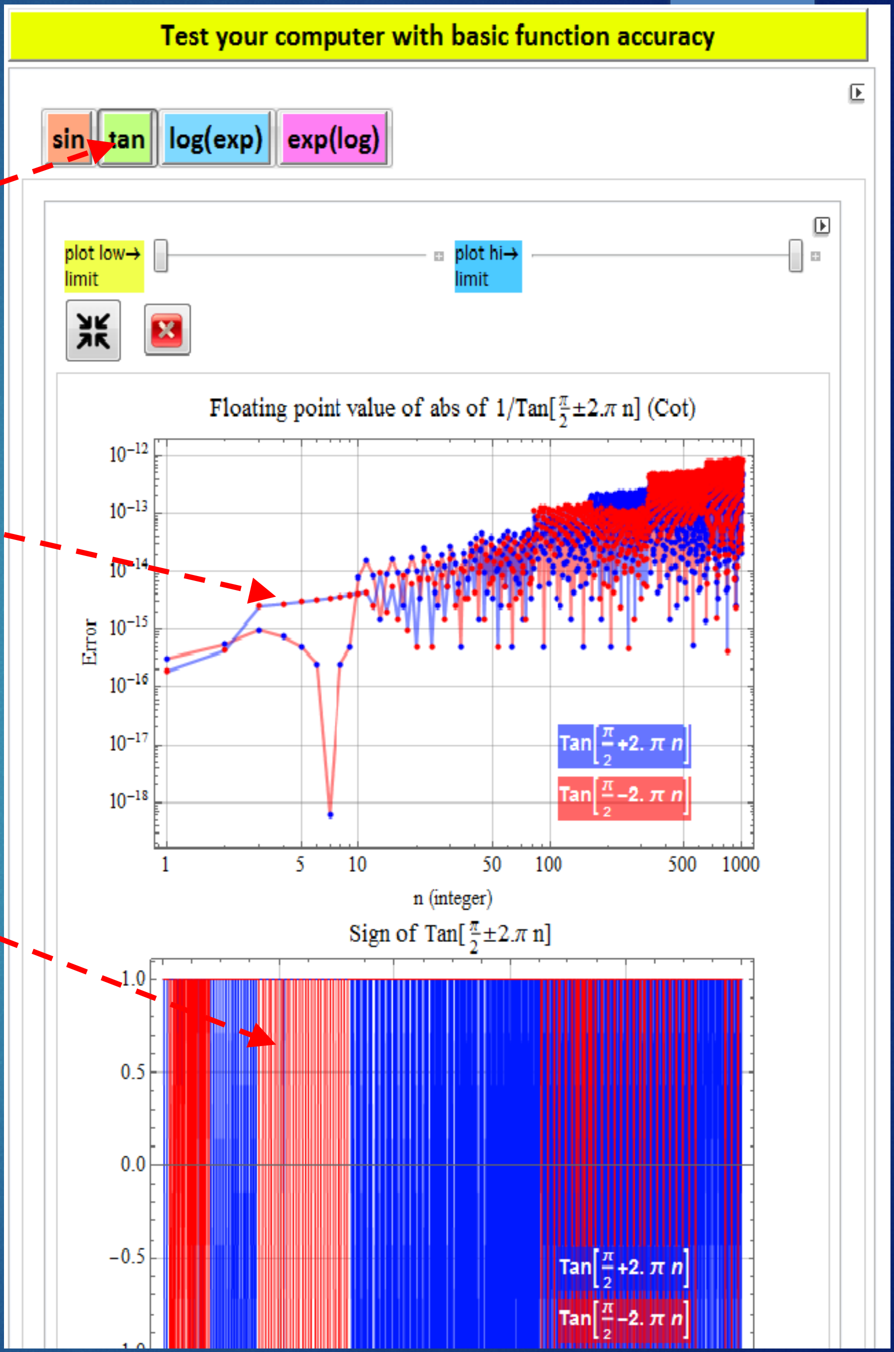
Copy and paste for each case, labeled it properly with accuracy and precision. Do not mix cases of the two categories. Each category should have its own section.

Let's get back to this

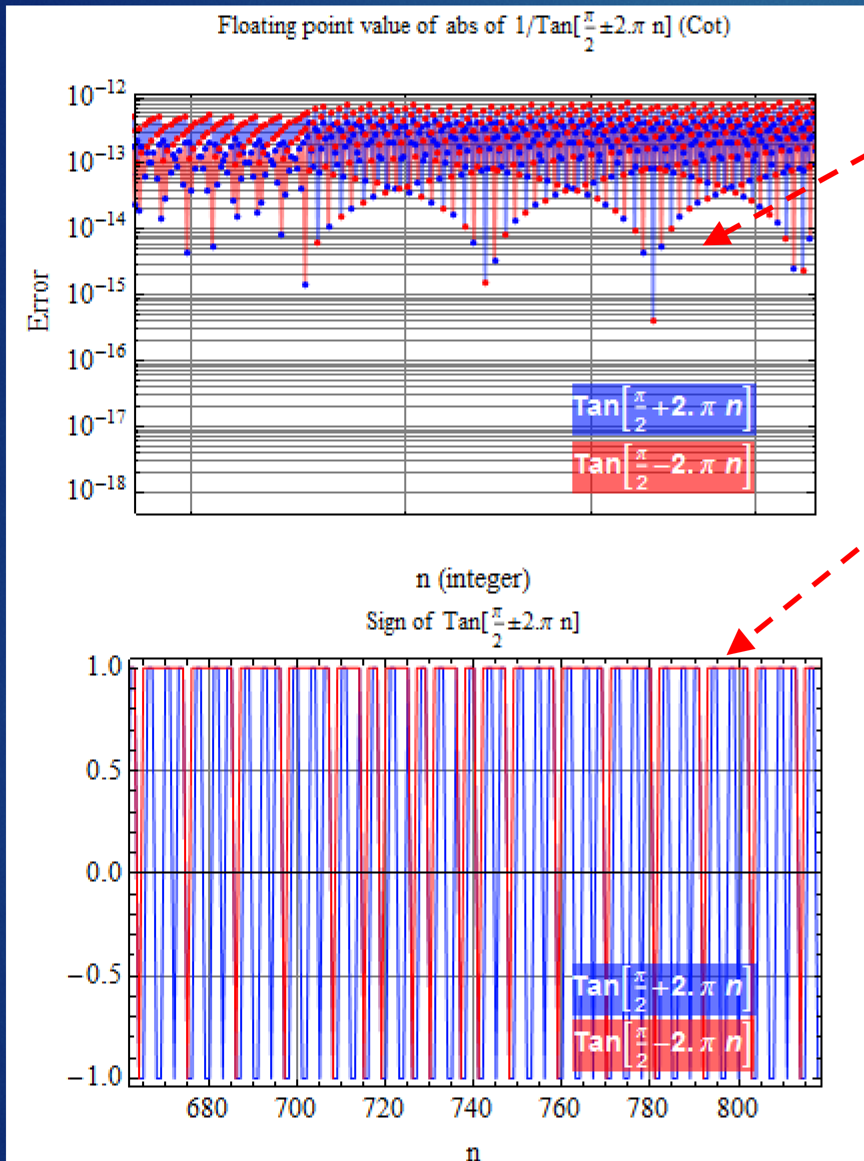
Here, click on this and we test 1/tan function (cot)

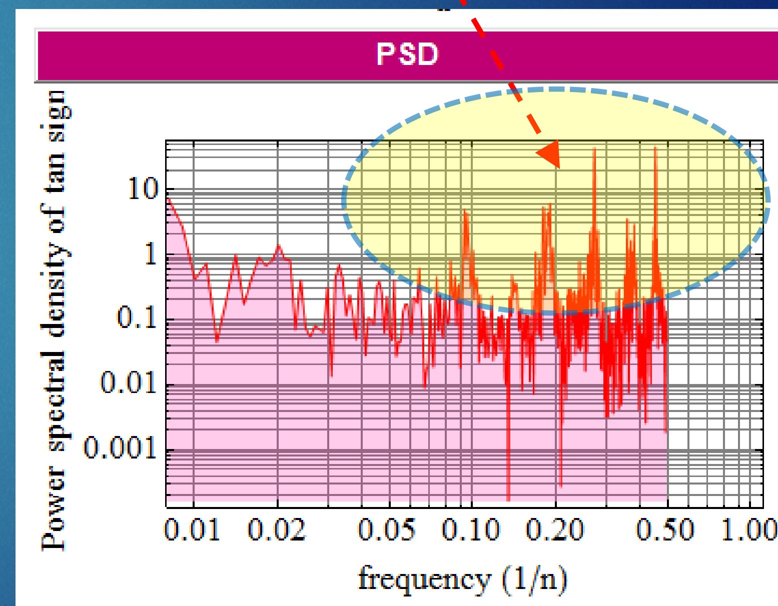
It should be infinite, but none is! The error appears similar to sin function, which is to be expected since this is cos/sin.

However, what matters here is not just about the magnitude, but the sign: it alternates between +1 and -1: imagine if your calculation critically depends on the sign: a big error!

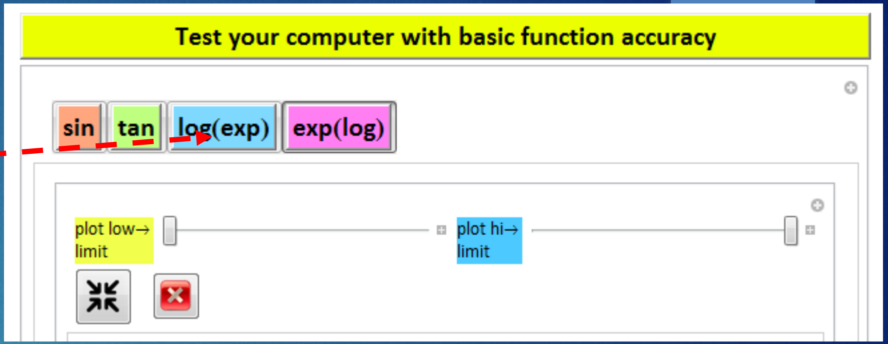


Here, we zoom in a segment of the error, we see that it is not as random as it looks. In fact, the power spectral density of the sign (+,-) error has some special frequencies as shown here

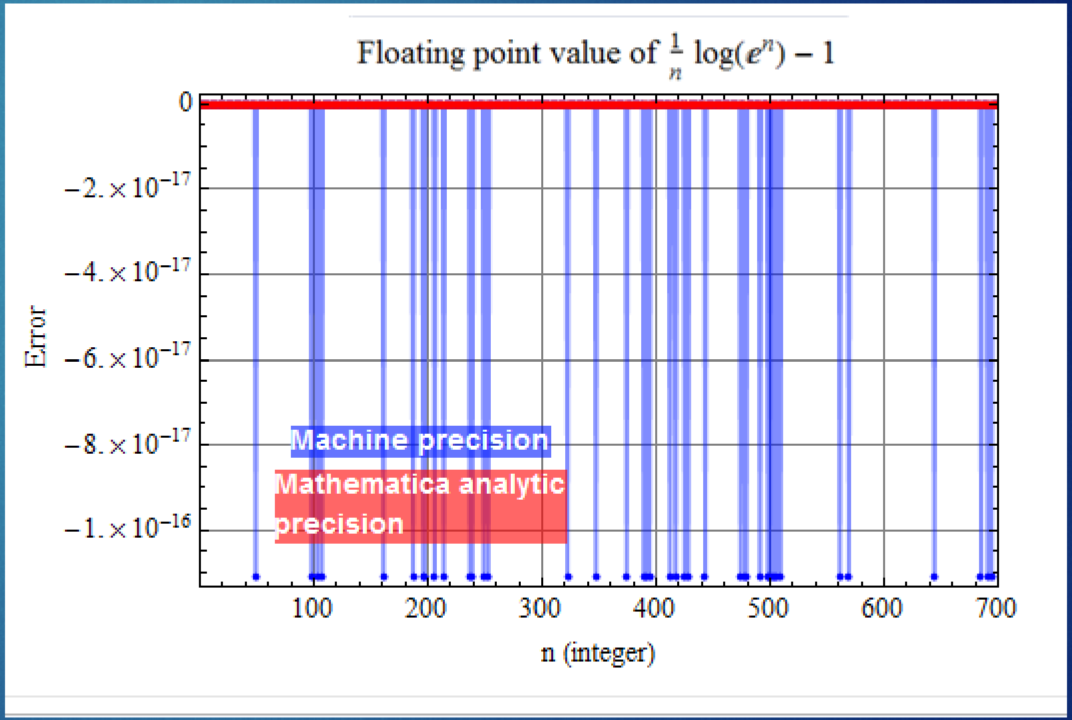




Here, we test taking log of a large number.



We see only the error due to machine epsilon.
However, note that Mathematica analytic result is again, correct.



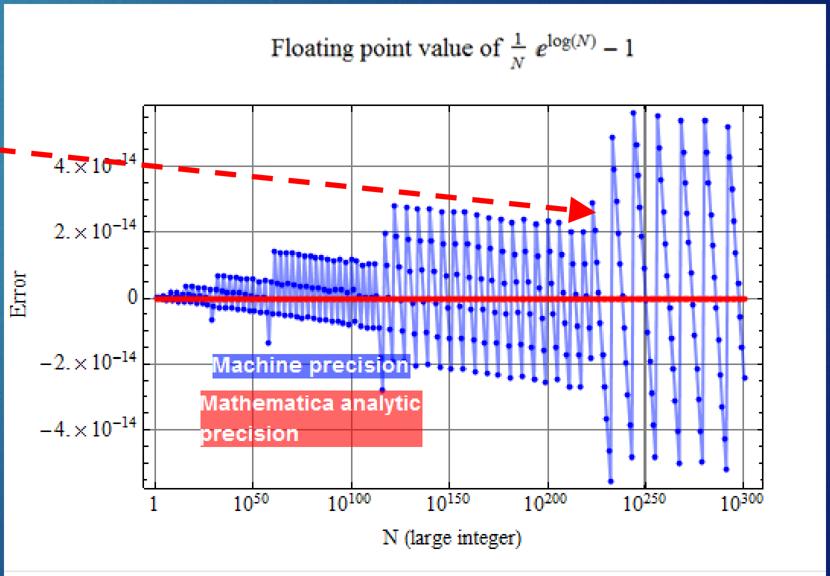
Problem 7

Calculate and plot $y = \frac{1}{N} e^{\log(N)} - 1$ for $N = 10^n$ where n is from 0 to 300. (*why do we stop at 300? and not go to 400? at what value of n do you think we will be in trouble*?)

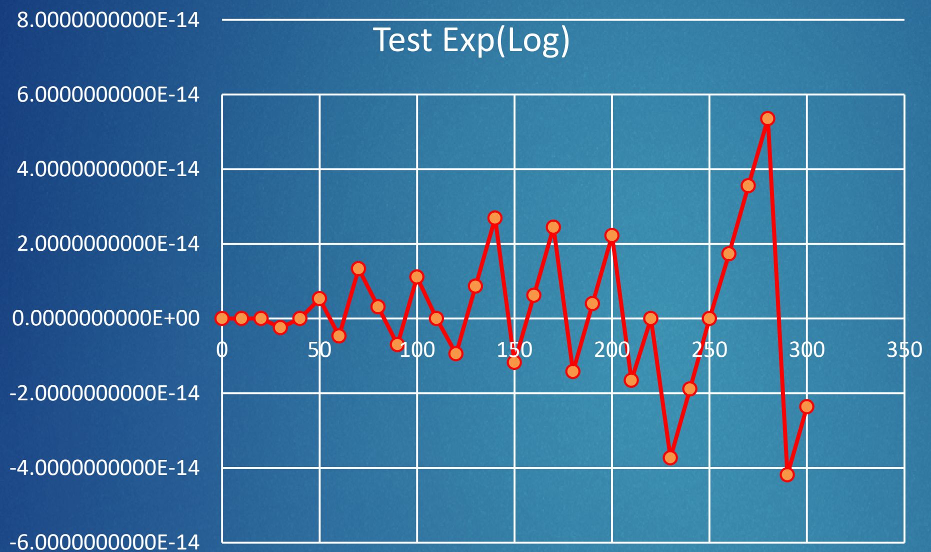
Calculate and plot $y = \frac{1}{N} e^{\log(N)} - 1$ for $N = 10^n$ where n is from 0 to 300. (*why do we stop at 300? and not go to 400? at what value of n do you think we will be in trouble*?)

This test is to check the exp of the log of a large number

Note how the relative error is magnified: the larger N is, the larger is the error.



Example: this is what one gets with Excel



Although there are plenty of precision digits –the inaccuracy is due to the precision loss when we take the log and then, magnified with exponent.

A summary of what we learn

- Computing errors are inevitable and not as small or trivial as we might assume, especially when we neglect handling quantities at special values with significant consequence (underflow, overflow, non-zero when should be 0., unpredictable + or - sign , or finite when should be infinite).
  - It's better for a program to crash to let us know what's wrong rather than give us a huge error without warning.
- The objective of the numerical methods is to learn how to obtain accurate and precise calculation results within certain acceptable limits: this is the tolerance of the calculation.
  - When doing computation on a scientific/engineering problem, you must know or set specifications on tolerance of the results.
  - Sanity check: test the computation on analytically known results to verify (sanity check) for algorithm possible errors.

wait, why we don't see errors in those tests with Mathematica analytic calculation?

- Mathematica is a high-level software designed to overcome those common numerical computation errors to give us exact results (or as accurately and precisely possible). In fact, it can give us arbitrary precision as long as given sufficient processing power and memory.
- It does this with software that has "built-in" analytic rules like our knowledge of mathematics. Its origin is from language for symbolic manipulation such as Lisp. Macsyma/Maxima is also similar.

Examples

If so, why don't we just use Mathematica and skip learning about these computation errors?

- There are a lot more about numerical methods and scientific/engineering computing than just numerical errors.
- Mathematica, as smart as it is, still can't fix seriously wrong algorithms. In this course, we learn about efficient, reliable algorithms for accurate and precise computation.
- For serious number crunching, Mathematica, like virtually all other software, still relies on machine dedicated floating point unit (FPU) operations for speed. It is generally slower to run software-based exact, analytic-computing in Mathematica.
- In fact, MATLAB can be used for high-speed large array processing and Mathematica is used for other capabilities.

The goal is NOT to force a machine to increase its precision and accuracy to satisfy the way we code. It is just a tool.

The goal is to know how to use it, and write codes so that its precision and accuracy are well within our tolerance. This is done by writing smart, robust, error-proof algorithms.