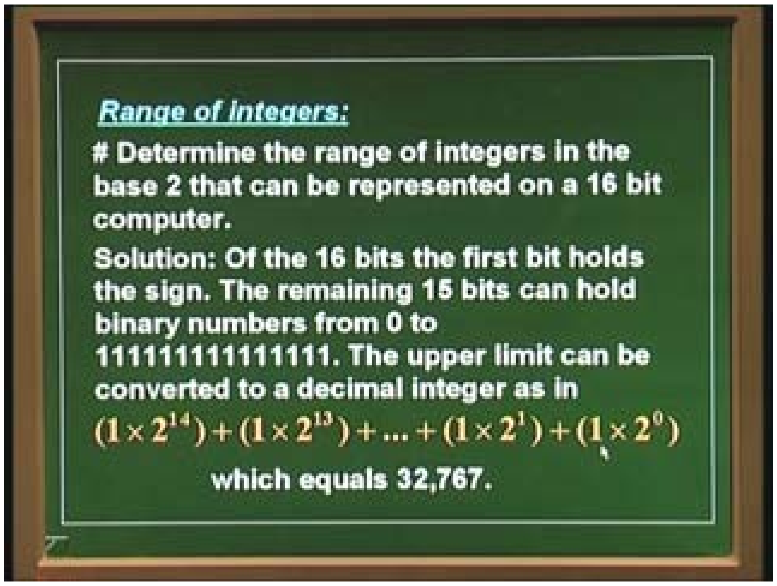
Numerical Methods and Programming

P. B. Sunil Kumar
Department of Physics
Indian Institute of Technology, Madras
Lecture - 6
Numerical Error

In the last lecture, we saw representation of numbers on a digital computer, of both the floating point and integers. Today, we continue on that topic and discuss some of the implications of that. Before we do that, we just summarize what we did yesterday, or we recap on what we did yesterday. We said, talked about the range of integers on that can be stored that can be represented on a digital computer, for example, we looked at the range of integers in the base 2 that can be represented on a 16 bit computer. We said that on 16 bits, the first bit holds the sign, and the remaining 15 bits can hold binary numbers from all 0 to all 1 right. So the upper limit is all 1.That can be converted to the integer as 1 into 2 to the power of 14.There are 0 to 14 bits here that is 15 bits. So, 1 bit is for the sign. So, that is 1 into 2 to the power of 14 on base 2, all the way to 1 into 2 to the power of 0.

(Refer Slide Time: 02:41)



Range of integers:

Determine the range of integers in the base 2 that can be represented on a 16 bit computer.

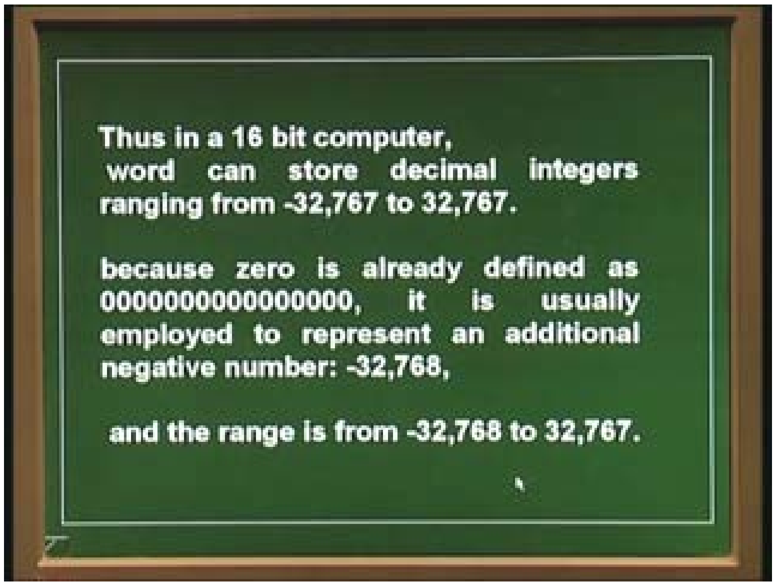
Solution: Of the 16 bits the first bit holds the sign. The remaining 15 bits can hold binary numbers from 0 to 111111111111111. The upper limit can be converted to a decimal integer as in

$$(1\times 2^{14})+(1\times 2^{13})+\ldots+(1\times 2^{1})+(1\times 2^{0})$$

which equals 32,767.

So that is equals to “32,767” and we saw that we would think that 16 bit computer, that it is minus 32,767 to 32,767 is what we can store, because 1 bit is for the sign. But, one thing is, that in all the bits, 0, it is 0, we know that okay, so there is no need to store that. So all 0s can be used to store one additional number, and the convention is that that is used to store one more negative number. So we have minus 32,768 stored in the machine. So the range goes from minus 32,768 to 32,767.Okay so that is the idea. So we always have one more negative number. That we saw, that using a program yesterday, that we have 1 additional negative number than the positive number. So coming back to the floating point representations, we said that the floating point is represented on the following form. That is, we have m, b to the power e.

(Refer Slide Time: 03:38)



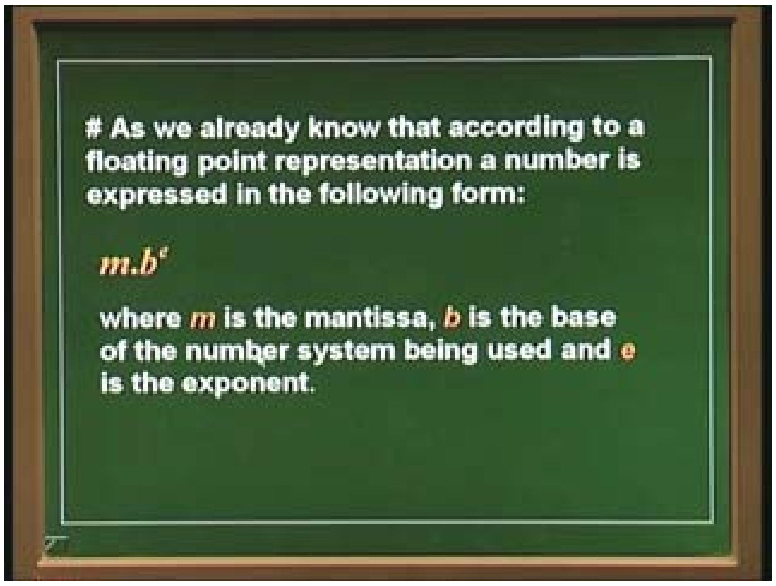
Thus in a 16 bit computer, word can store decimal integers ranging from -32,767 to 32,767.

because zero is already defined as 0000000000000000, it is usually employed to represent an additional negative number: -32,768,

and the range is from -32,768 to 32,767.

So m is the mantissa, b is the base, and e is the exponent. That is, what we saw yesterday right. So the mantissa has a finite number of bits. Exponent has a finite number of bits, and the base is fixed. So that determines the range of floating point numbers which can be fixed, which can be represented. So we will see that in a more graphical representation here. Okay so you have what is called the word.

(Refer Slide Time: 04:30)



As we already know that according to a floating point representation a number is expressed in the following form:

$m.b^e$

where m is the mantissa, b is the base of the number system being used and e is the exponent.

Okay word is the combination of the sign, the exponent, and the mantissa okay. So it is a signed exponent, and the sign, and the mantissa. In the computer sometimes, the sign is not stored. This is, exponent is stored slightly different from this, that we could have a finite number of bits for the exponent and 1 bit for the sign, and a finite number for the bits for the mantissa.

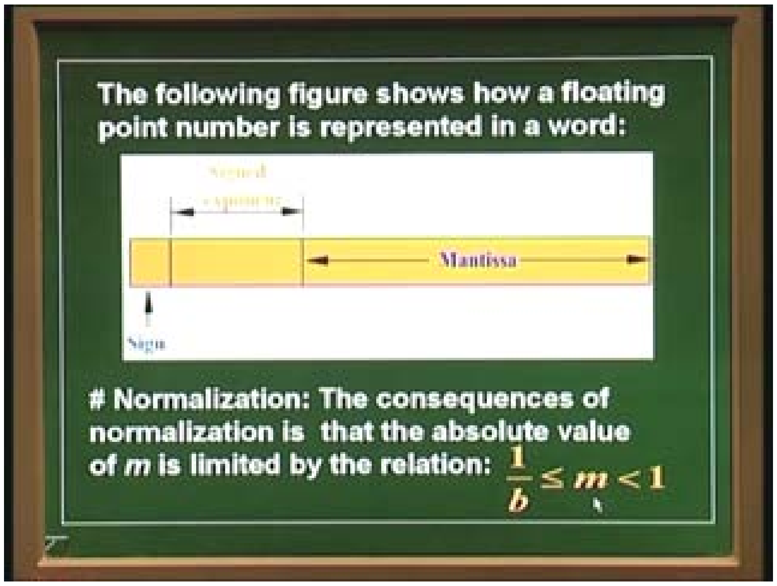
For example, you could have, let us say you could have 3 bits for the exponent, and then you would think that 1exponent, 1bit is for the sign, and 2 bits for the number. Instead of that, one could have all the 3 bits used for the number, and have a convention that it always goes from minus that maximum number to plus that maximum number, that is also possible. So that is, if you have 3 bits you have 2 to the power of 0, 2 to the power of 1, and 2 to the power of 2. That is 4 plus 2 plus 1 that is 7. So then you could say that it goes from minus 3 to plus 3.

So that is one way of choosing this exponent. So normally, for our convention, let us take this as a signed exponent, and this is signed this is the sign of the number and the mantissa, and then we talked about normalization. That is, the mantissa. If I start there, the most significant bit is always 1. So that is what we saw, it is always 1. It is normalized to 1. Okay so that bit need not be stored because it is actually 1 always okay. So thus the consequence of that is the mantissa is limited within this range. Okay that is, it has to be less than 1 of course, and it has to be greater than or equal to 1 over base, okay because 2 to the power of minus 1 is always 1.That bit which represents 2 to the power of minus 1 is always 1, okay.

So when that means the mantissa is always larger than that. So the mantissa is always larger than 1 over b. But it has to be less than 1 because whenever it reaches b, whenever it reaches 1 that can be transferred to the exponent. So, for example, in the base 10 now, you can have ".5" into 10 to the power of 3. Let us say as a number ".56" into 10 to the power of 3, and you keep increasing to ".57" ".58" and ".9", and you keep increasing this and when it reaches 1, that is, ".99", and the next bit, then what it goes it is to ".1" into 10 to the power of, you increase exponent by 1. So the mantissa is always limited between 1 by the base, and 1. This base 10, for example, the mantissa cannot be less than ".1" and it cannot be greater than 1 because it goes into the exponent.

And if it is base 2 the mantissa cannot be less than ".5" because 2 to the power of minus 1 is .5, or it cannot be less than 1. Okay so that is the, this is called the word. So, and we have the base, the mantissa, and the exponent, signed exponent okay. We will see this, the consequences of this again as we go along. So let us create a hypothetical floating point numbers, set for a machine that stores 7-bit words. So let us assume that it is a 7-bit word. So now, the first bit, as we said, goes into the sign of the number. Now the next 3 bits, let us assign the next 3 for the sign, and the magnitude of the exponent.

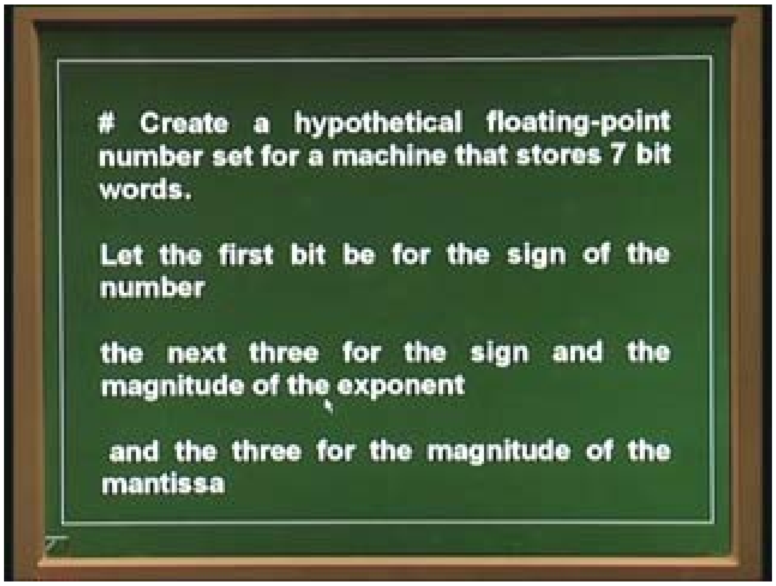
(Refer Slide Time: 08:13)


Signed exponent
Mantissa
Sign

The following figure shows how a floating point number is represented in a word:

Normalization: The consequences of normalization is that the absolute value of *m* is limited by the relation: $\frac{1}{b} \leq m < 1$

(Refer Slide Time: 09:02)



Create a hypothetical floating-point number set for a machine that stores 7 bit words.

Let the first bit be for the sign of the number

the next three for the sign and the magnitude of the exponent

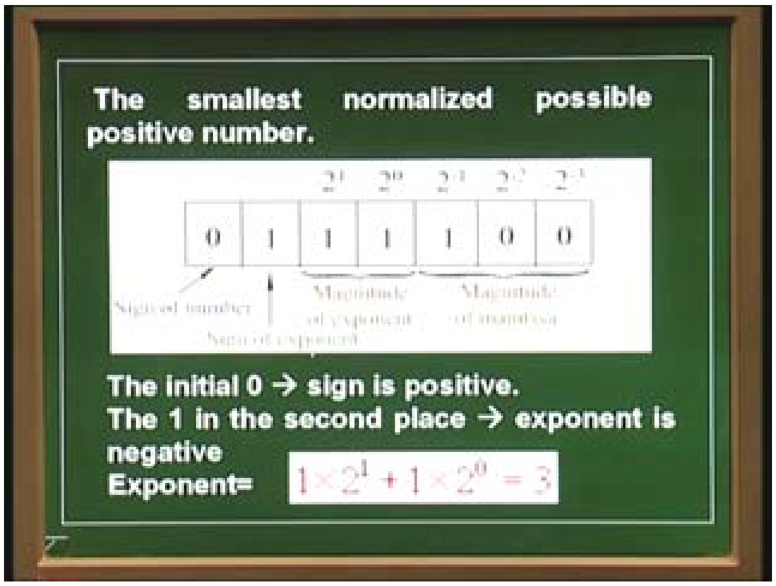
and the three for the magnitude of the mantissa

Okay let us go for the convention just to understand the word and then the next 3 goes to the magnitude of the mantissa. So remember, we have 1 bit for the sign of the number, 1 for the sign, 3 for the exponent, of which 1 is the sign of the exponent, and 2 for the exponent itself, and then you have 3 bits for the mantissa. So that is represented here. So now we can ask the question, what is the smallest number which I can represent using such a word. As we saw yesterday that you cannot, there is a limit on the range of the floating point numbers. There is also a limit on the number of sorry the limit on the number which can go inside this range right. That also, we saw yesterday. It is not a continuum.

So here is a, now we are given this word. So we have 7 bits, as I said. We have 7 bits. So, 1 bit for the sign okay, these 3 bits is for the magnitude, 3 bits for the exponent. 1 is for the sign of the exponent, and 2 for the magnitude of the exponent, and 3 for the magnitude of the mantissa. So what is the smallest number which we can represent using this. Okay the smallest positive number let us say. For example let us take, so this is fixed to 0. That means, its sign is positive, and you want the smallest number, it is obvious that sign of the exponent is negative. So we put sign of exponent as 1. Okay so one represents negative, and then you have the magnitude of the exponent. The exponent has to be maximum we know, maximum negative to get the smallest number, so we put 11 here. Okay that means, it is 2 power 0, 2 power 1.

And then you have the magnitude of the mantissa. So mantissa is 2 power minus 1 and that is the smallest possible right, because this is normalized. So this has to be 1 always. So 2 power minus and these 2 are equal to 0. So that is the smallest possible. So then, we could now ask the question: “What is the smallest possible number?” And then, we would get, the exponent would now found that it is 2 to the power 1, 2 to the power of 0, that is 3.

(Refer Slide Time: 11:36)


The smallest normalized possible positive number.

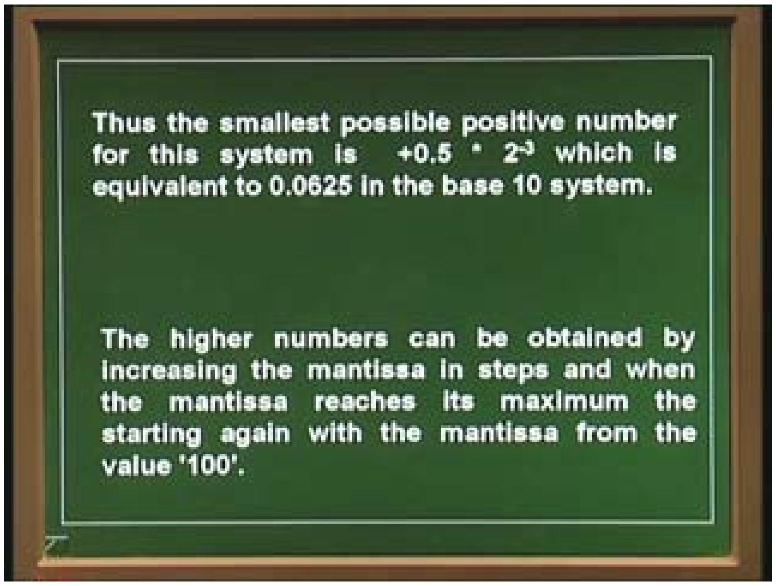
2^1 2^0 2^{-1} 2^{-2} 2^{-3}
0 1 1 1 1 0 0
Sign of number
Sign of exponent
Magnitude of exponent
Magnitude of mantissa

The initial 0 → sign is positive.
The 1 in the second place → exponent is negative
Exponent= $1\times 2^1 + 1\times 2^0 = 3$

The negative sign, okay so the smallest, the number which you would get, would be “.5” into 2 to the power of minus 3 right. That is the smallest possible, smallest possible mantissa. We said the base is 2. So the smallest possible mantissa is 1 over base that is “.5”. And exponent maximum we could maximum represent, maximum negative was minus 3. So we have “.5” into 2 to the power of minus 3 which is equal to “.0625” right in the base 10 system.

So that is how we would write it, and in the base 10 system, now normalized again, we would write it as “.625” e minus 1. We saw that yesterday. That is the representation of the number, and we can also get the highest number by similar way right. We would have all the mantissa. So we would have, if you want to take the highest number, then we would have all 111 here, and this will be 0, and that will be the highest number which is possible to get. So, which can be obtained by increasing the mantissa 1 by 1, and whenever maximum mantissa reaches 1 then you transfer that to the exponent okay, and again you continue doing that.

(Refer Slide Time: 12:49)



Thus the smallest possible positive number for this system is $+0.5 * 2^{-3}$ which is equivalent to 0.0625 in the base 10 system.

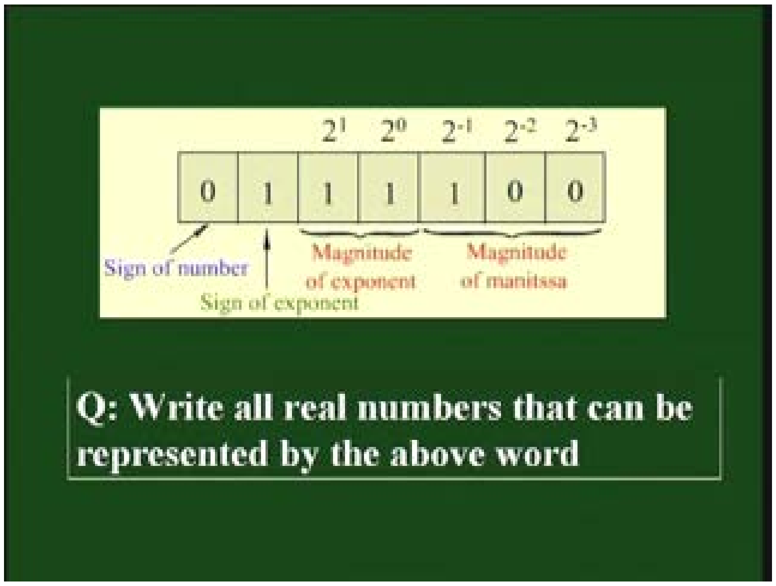
The higher numbers can be obtained by increasing the mantissa in steps and when the mantissa reaches its maximum the starting again with the mantissa from the value '100'.

That is all we would go to the, that is how you will represent all the numbers, and it is a good exercise to actually put in all these numbers actually, and see what are the gaps and you can also see that the gap increases. The gap between numbers increases as the number increases. We saw that yesterday, on graphically.

So what are the other points to remember on the floating point representation? That is, there is a limited range of quantities that can be represented, extremely important now if you go anything more than that, or anything less than that, less than the smallest number, smallest positive, or smallest negative, smallest positive number possible, if you go there you will get underflow. If you go larger than a number which is possible to store then you will get an overflow, and apart from that, even within this range, there is only a finite number of quantities that can be stored.

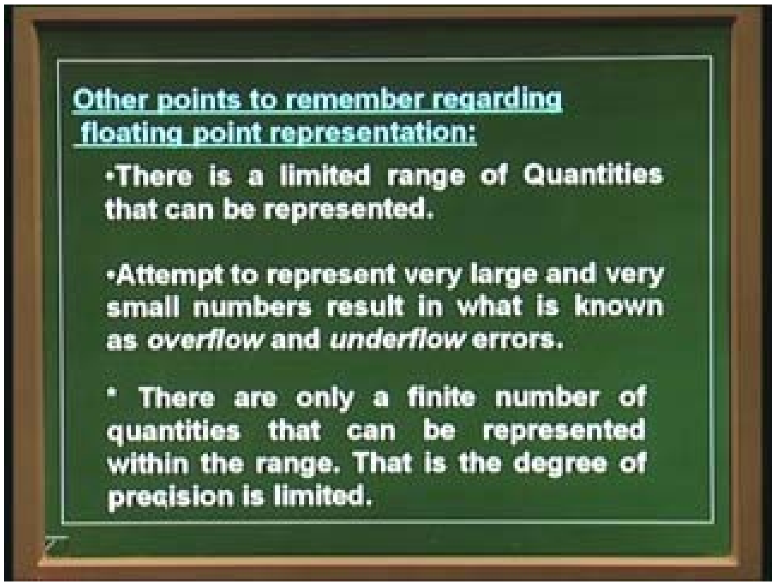
So I am repeating this. This is extremely important. This is for, we talk about the loss of significant numbers. So there is a finite number of significant digits. So there is a finite number that can be represented within a range. That is, the degree of precision is limited because we have a finite number of bits. So we have a finite number of significant digits. So there is, as I said, the precision is how close the two measurements are, but then how close it can be is limited by how close the numbers can be represented. So there is a limited precision on a digital computer.

(Refer Slide Time: 13:08)



Q: Write all real numbers that can be represented by the above word

(Refer Slide Time: 14:38)



Other points to remember regarding floating point representation:

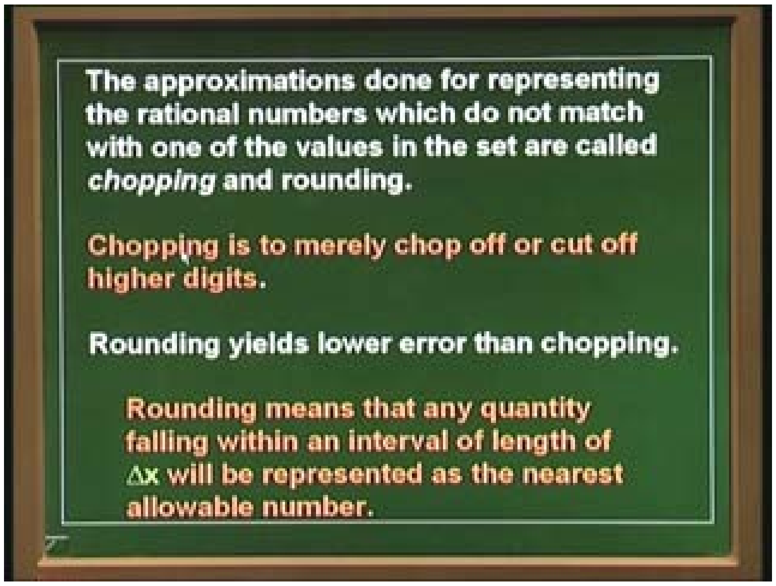
- There is a limited range of Quantities that can be represented.
- Attempt to represent very large and very small numbers result in what is known as *overflow* and *underflow* errors.
- There are only a finite number of quantities that can be represented within the range. That is the degree of precision is limited.

Okay these approximations now, apart from this, because of these limitations, when we represent numbers, we have approximation sphere to make approximations we saw that. For example, we saw when you try to represent phi that we have to actually chop after some off after some number of digits.

So there are two ways of doing that. Either we can chop it off or we can round it off. Okay they are called chopping and rounding. Chopping is merely to chop or cut off the higher digits. So that is what chopping is. In rounding, what we do is, we would look at the value. For example, if it is a base 10 system, we would say that if it is more than ".5", I will make it into the next integer. So if it's ".625", I will make it to ".63"if it's ".624" I will say it's ".62". So then, I would round it off. So that is, there is some delta x which is fixed by the system, which you used, and that allows us to represent it as, we will round it off to the next digit, the earlier digit. So that is what we are calling rounding.

So rounding yields lower error. Actually, rounding yields plus minus error, while chopping always gives you a plus error. For example, in ".625" chopping is always ".62" ".626" is made into ".62" if it is 2 digit, while ".62" will go to ".63", ".626" will go to ".63" in the case of rounding, that is the difference.

(Refer Slide Time: 16:16)



The approximations done for representing the rational numbers which do not match with one of the values in the set are called *chopping* and rounding.

Chopping is to merely chop off or cut off higher digits.

Rounding yields lower error than chopping.

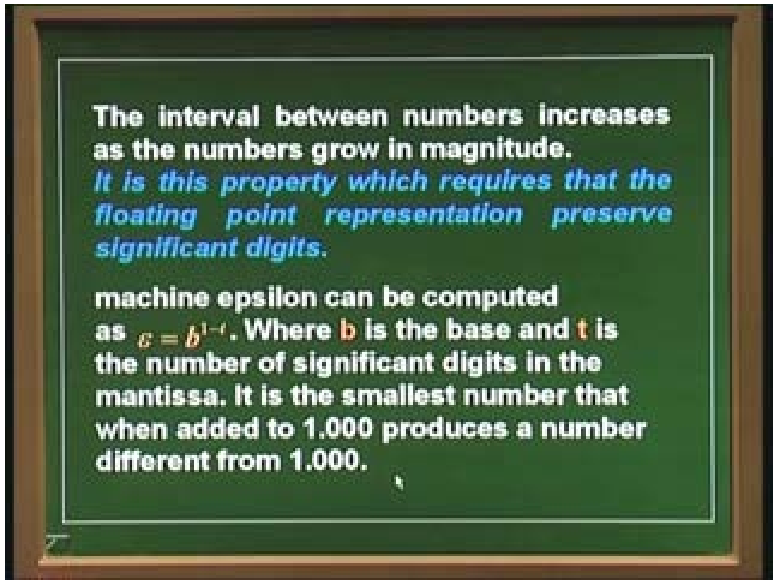
Rounding means that any quantity falling within an interval of length of Δx will be represented as the nearest allowable number.

So that is the two ways of representing an infinite series into a finite number in on a computer. Okay we saw the interval between the numbers increases as the numbers grows and we said that is what is important for preserving the significant digit, and this also we saw yesterday. And then we said another consequence of all this, that is the finite number of digits and a particular, finite number of digits, and finite number of word length, is that there is something called a machine epsilon.

That is what we saw that is the smallest number. Given the word, there is a limit on the smallest number which can be represented. We just saw that on the 7-bit machine, it was .06, and that means there is something called machine epsilon, and that is the number, which if you add to some number, let us say 1, okay I am just taking 1 here, it produces a number which is different from 1. So, you keep add some number to 1 and ask what the result is, and the result is more than 1, it looks trivial, but on a computer it is not because there is a limit on the smallest number. Anything less than that is taken as 0, and since adding a 0 is not going to make a difference, there is a limit on that okay.

So that is called machine epsilon. So that is given by b into 1 minus t, where "t" is the number of significant digits in the mantissa. So b is the base. So once you got the machine epsilon okay, if you know the machine epsilon, then you can fix the number of significant digits because you know this is a relation. B is always 2 on a machine. So epsilon is b. It is not always 2, but if you fix b to be 2 okay, if you know the b, then you can fix the number of significant digits. So remember, the machine epsilon is a number which has to be added which when added to 1 gives you a number which is different from 1. So 1, I have taken as something here. You could add to a number which is different from that. So that is the machine epsilon which basically means that that is the smallest number that the machine can store. That is fixed by b to the power of 1 minus t.

(Refer Slide Time: 18:57)


The interval between numbers increases as the numbers grow in magnitude.
It is this property which requires that the floating point representation preserve significant digits.

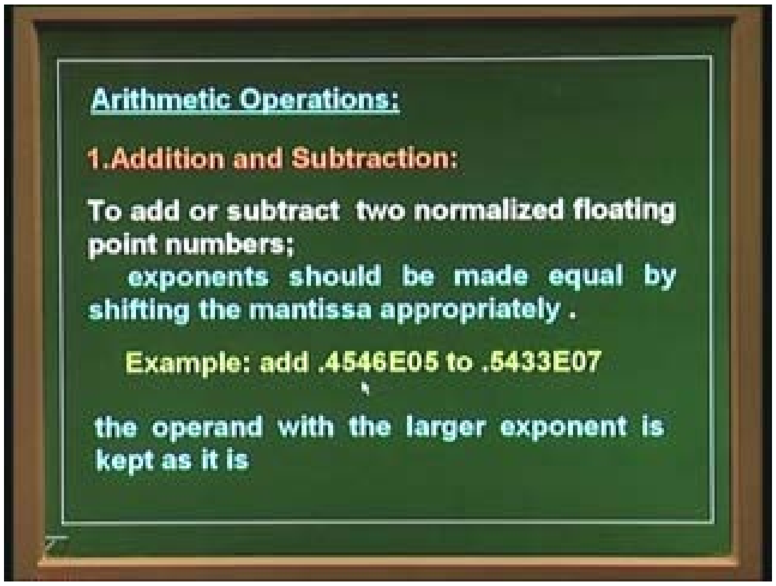
machine epsilon can be computed as $\varepsilon = b^{1-t}$. Where b is the base and t is the number of significant digits in the mantissa. It is the smallest number that when added to 1.000 produces a number different from 1.000.

So now, having gone through the representation of numbers, and we know something about the range of integers, the range of floating points, and the precision and the number of significant digits, etcetera,. Let us look at the consequences of that on arithmetic operations. So, before that we need to understand how the arithmetic operations are done on a computer. We need to know that. So we look at some of the basic arithmetic operations like, so are like addition and subtraction. So how do we add 2 floating points? To add or subtract 2 normalized floating point numbers, you remember the floating number is point something something into base to the power of something right, so that is base 10.

You will have ".625" 10 to the power of 2, and now you have to add this to something else. So when you add these two numbers, these normalized floating point numbers which have a mantissa and an exponent, we have to make the exponents the same and then we add the mantissa. For example, let us look at how do we add in the base 10 ".4546" into 10 to the power of 5 to ".5433" 10 to the power of 7. So now, how do we add this? So what we do is the one with the largest, this is the convention, because it is normalized floating point. So we have to keep this, we have to make the exponent the same, remember. So how can you do that?

If you have to do that, we keep this number with the larger exponent. We keep that as it is, and we shift this and make the exponent same. That means, you will have to make it point, you have add 0s here. You have to pad it with 0s. So it is “004546” 10 to the power 7, and then we add the thing, but we see that we have a fixed number of digits on the mantissa okay.

(Refer Slide Time: 21:28)



Arithmetic Operations:

1.Addition and Subtraction:

To add or subtract two normalized floating point numbers;
exponents should be made equal by shifting the mantissa appropriately .

Example: add .4546E05 to .5433E07

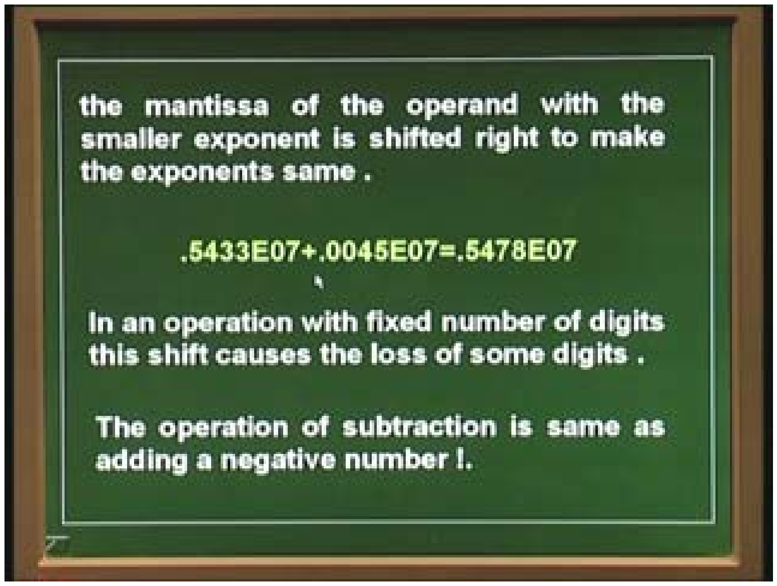
the operand with the larger exponent is kept as it is

So you have to remember that. Let us see what we get. So we shifted this. So the mantissa of the operand with smaller exponent is shifted, to make the exponents the same. So we did that. Then we got “.5433” 10 to the power of 7 added to “.0045”.The rest of that is gone okay. So we do not have that, we have only a finite number of digits. So we lost those numbers.

We added to this. So now, we see that because of the finiteness of the number of digits, we have already lost something, some we have introduced an error. Loss of some digits, so now this is the same way if you do the subtraction also. We have to equate the exponents; we have to make them the same, so the result of that would be that we would lose some significant digits. So remember as I said, actually the number was “.4546” 10 to the power of 5, and we have to make it equal to the exponent, equal to 7. So when it was “.0045”, and we simply drop this 46 again we just chopped it. In this case, chopping and rounding will not make a difference because it is 4 if it was rounding, and if it was more than 4, then I would have made it as 6. So that is what an example of that is.

So that is what see, and then now we can go to the next operations. So that is multiplication. So addition, subtraction is the same. You add with minus sign. So that is subtraction. And now we can look at how do we multiply two numbers? So when you multiply this, we first multiply the mantissa and add the exponent just like we multiply any numbers, and then we shift the mantissa such that it is normalized. So that is what is been shown here. So we multiplied these two numbers. So we got “.2278273” and we got exponent, we added the 2 exponents, so we got minus 3. But the number of digits, we have written here the full number, but the number if digits is finite we know. So we have only one 4 digits. So we keep 4 digits. We are going to chop; we are going to drop the rest of the digits.

(Refer Slide Time: 22:30)



the mantissa of the operand with the smaller exponent is shifted right to make the exponents same .

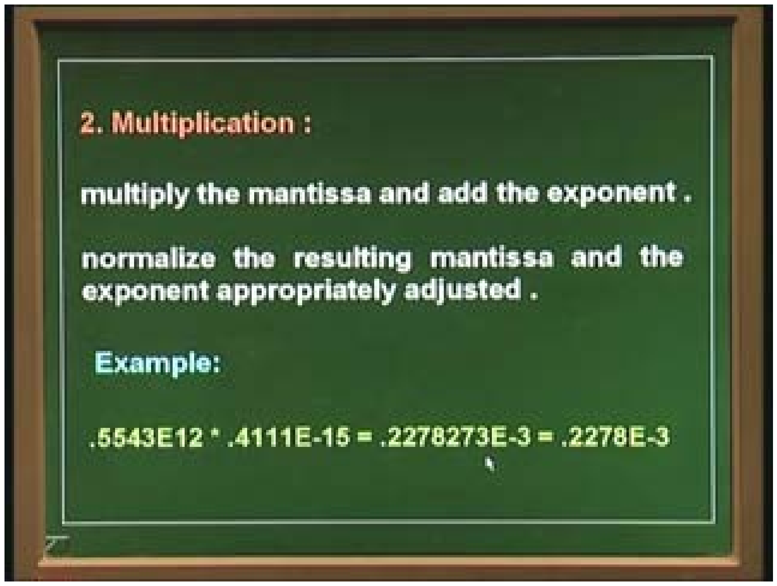
.5433E07+.0045E07=.5478E07

In an operation with fixed number of digits this shift causes the loss of some digits .

The operation of subtraction is same as adding a negative number !.

So any operations, as you can see, any operation which you do on any arithmetic operation, whether it is multiplication, whether it is addition, subtraction, division, almost always leads to loss of some digits. So this is something which we should keep in mind when you write a program.

(Refer Slide Time: 24:25)



2. Multiplication :

multiply the mantissa and add the exponent .

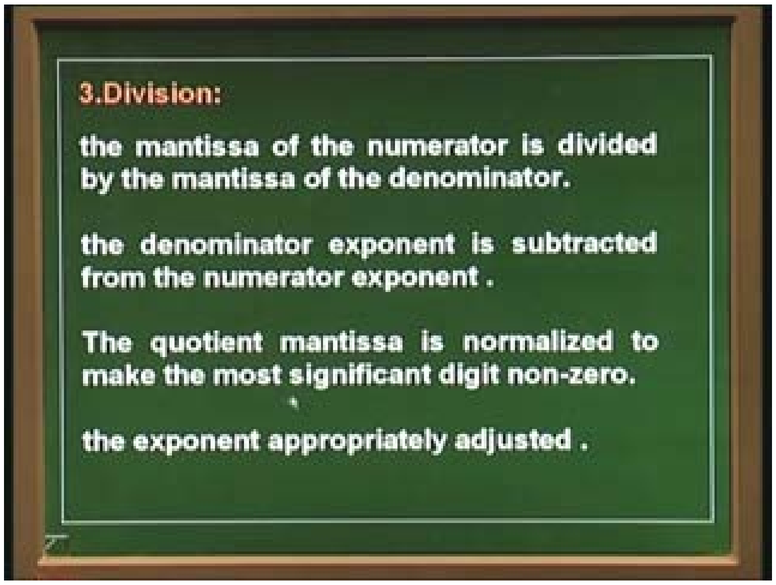
normalize the resulting mantissa and the exponent appropriately adjusted .

Example:

.5543E12 * .4111E-15 = .2278273E-3 = .2278E-3

We can quantify this, and we will come to that a little later. Again, going to division, in the division again, we would subtract the exponents, and we would divide the mantissa, and we would normalize it by shifting it again. The mantissa of the number is divided by the mantissa of the denominator. The denominator exponent is subtracted from the numerator exponent, and the quotient mantissa is normalized to make the most significant digit non-zero. This is always carried out. The most significant digit is non-zero and when you do this operation, you have to adjust the exponent, and here is an example again, particularly chosen to show that you would actually lose digits.

(Refer Slide Time: 24:58)



3.Division:

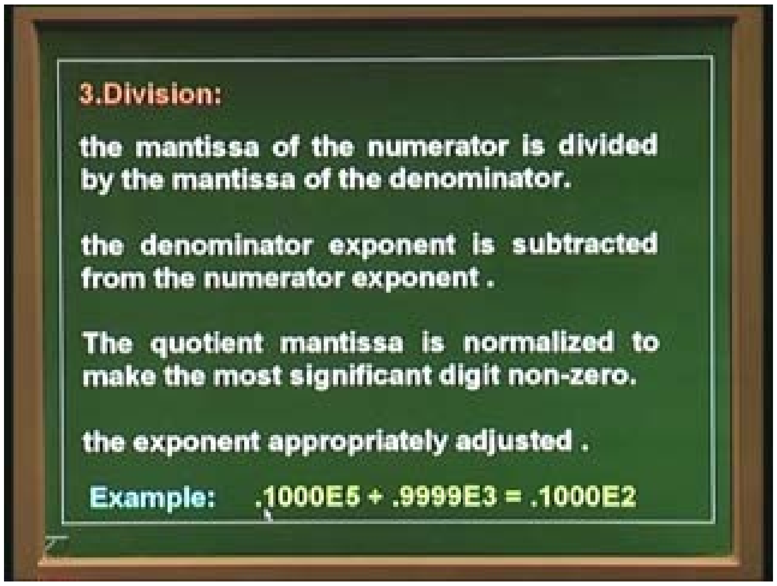
the mantissa of the numerator is divided by the mantissa of the denominator.

the denominator exponent is subtracted from the numerator exponent .

The quotient mantissa is normalized to make the most significant digit non-zero.

the exponent appropriately adjusted .

(Refer Slide Time: 25:13)



3.Division:

the mantissa of the numerator is divided by the mantissa of the denominator.

the denominator exponent is subtracted from the numerator exponent .

The quotient mantissa is normalized to make the most significant digit non-zero.

the exponent appropriately adjusted .

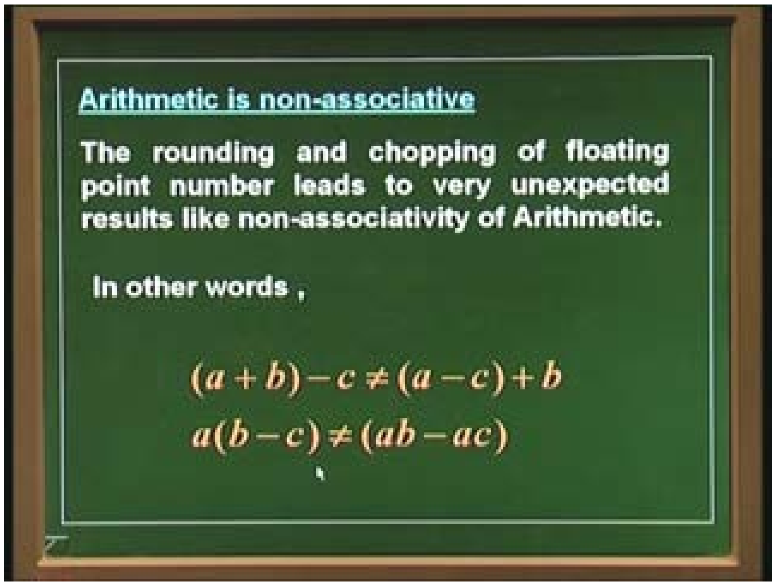
Example: .1000E5 + .9999E3 = .1000E2

So “.1 e^5” “.1” into 10 to the power of 5 divided by “.9999” 10 to the power of 3, and we get “.1” 10 to the power of 2. We know that if you do this actually on a larger number, larger precision machine, you will get a different value from this. Okay but so that is what if you have only 4 digits here then you are going to lose some digits, and we are going to introduce some errors. Of course, nothing to worry the error which we get in present-day computers are much much smaller than this. The minimum is 32 bits available today. So this is much smaller than that. This is only to demonstrate that even though small, there is a loss of significant digits.

So now, what is the consequence of this? So we have always been taught in school that addition, subtraction, multiplication are associative. So what we see is that because of this loss of significant digit, we have slightly different rules for arithmetic on a digital machine. Some arithmetic is non-associative. The reason being that we round it off, we chop it off, and we have we lose some significant digits, and hence, that introduces non-associativity of arithmetic. That is, a plus b minus c is not the same as a minus c plus b. A clever programmer will always keep this in mind when he or she is doing these calculations.

So we have to make sure that the numbers we add or subtract are of the same order of magnitude. Otherwise, we would always get errors. So that is something which we have to always be very careful about. So, “a plus b minus c” is not the same as “a minus c plus b”, and “a into b minus c” is not equal to “ab minus ac”.

(Refer Slide Time: 27:26)



Arithmetic is non-associative

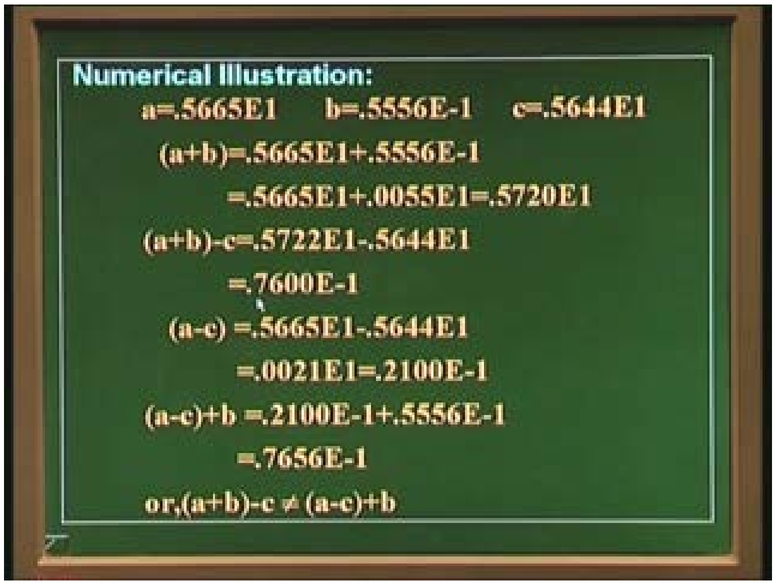
The rounding and chopping of floating point number leads to very unexpected results like non-associativity of Arithmetic.

In other words ,

$$(a+b)-c \neq (a-c)+b$$
$$a(b-c) \neq (ab-ac)$$

We will see this with some examples. So here is an example. Let us take “a” as .5665.If you have a calculator you can try it out now.5665.Your calculator will always have more precision than this. So you have to get these approximate numbers to get this demonstration working. So “a” equal to “.5665” into 10 to the power of 1,“b” is “.5556” into 10 to the power of minus 1, and c is “.5644” 10 to the power of 1.So let us add a plus b first. Keep only 4 digits. We can round it off or chop it off, do one of them always. So then we have now, we add these two. So we have exponents minus 1 here, plus 1 here. So we shifted this to make the exponents same. So it becomes ".0055 e^1".That is 10 to the power of 1, and then we add these 2 mantissas. So we get ".5720" into 10 to the power of 1. Now we have a "c" which is ".5644" 10 to the power of 1, so we subtract that from this sum. That is ".5722" 10 to the power of 1 minus "0.5644" 10 to the power of 1. So there is some, okay so this is actually 2. So this is ".5722" 10 to the power of 1 minus ".5644" 10 to the power of 1.

(Refer Slide Time: 29:02)



Numerical Illustration:

a=.5665E1 b=.5556E-1 c=.5644E1

(a+b)=.5665E1+.5556E-1

=.5665E1+.0055E1=.5720E1

(a+b)-c=.5722E1-.5644E1

=.7600E-1

(a-c) =.5665E1-.5644E1

=.0021E1=.2100E-1

(a-c)+b =.2100E-1+.5556E-1

=.7656E-1

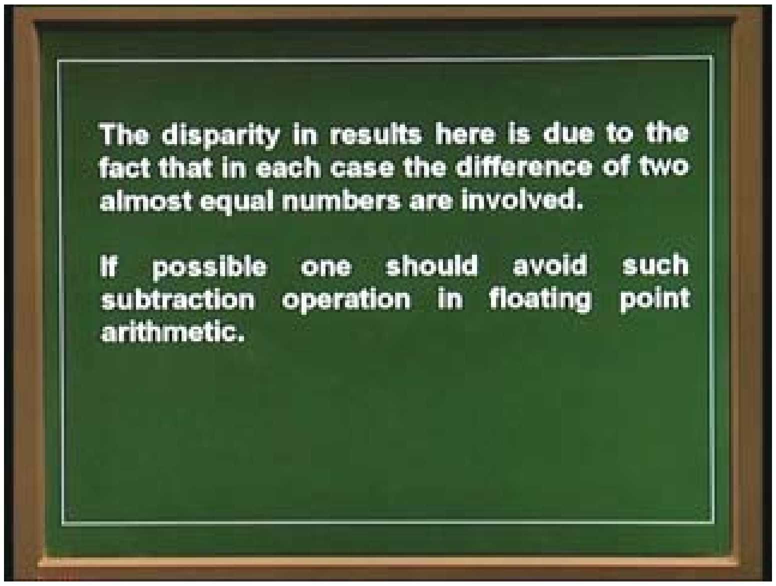
or,(a+b)-c $\neq$ (a-c)+b

So we subtract this from this, we get ".7600" 10 to the power of 1, and then now we do a different operation. We do a minus c first. So when you do a minus c you get ".5665" 10 to the power 1 minus ".5644" 10 to the power 1and we subtract that first. So now we are going to, instead of a plus b minus c, we are going to do a minus c first, and that leads to ".21"10 to the power of minus 1, and then we add b to it, and we get ".7656" 10 to the power of minus 1. So there is difference in the last two digits. That is what, I am trying to say.

So we could say that is the kind of error which we would have. We would compute the error actually by doing these two different operations and see what the error which you would get, and you know that if it is full representation of the floating point, or if it is actually a real number, if they are treated as proper real number, then this error should not come. So we find a new rule that a plus b minus c is not equal to a minus c plus b merely because of the finiteness of the word, that is the floating point representation.

Okay so now, we saw that this disparity is due to difference coming because of the finiteness of the number. So we should be careful when you do subtraction, or addition, or multiplication, or division of numbers which are of completely different magnitude. So this has an interesting consequence, that as I said, we have something called a machine epsilon, that is the smallest number which can be represented on a machine, or the smallest number which when added to 1 would give you a number which is different from 1. Similarly, we also have a concept of 0, what is 0 on a digital machine. So, when you are trying to do that, for example, let us take this quadratic equation: x square plus 2 x minus 2 is equal to 0, and answers are minus 1 plus minus root 3 right. We know that. We can solve this. So the question is, if I put this minus 1 plus minus root 3 back into this equation, what do I get? So if I get minus 1 plus root 3, so what do I get? So minus plus root 3, according to me is, or according to the computer is, ".7320" right and minus root 3 is ".2732" 10 to power of 1.

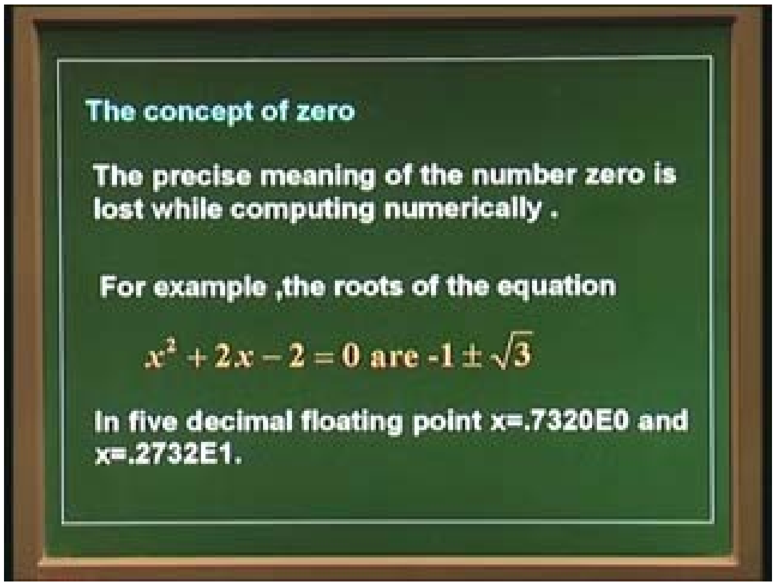
(Refer Slide Time: 30:56)



The disparity in results here is due to the fact that in each case the difference of two almost equal numbers are involved.

If possible one should avoid such subtraction operation in floating point arithmetic.

(Refer Slide Time: 32:45)



The concept of zero

The precise meaning of the number zero is lost while computing numerically .

For example ,the roots of the equation

$$x^2 + 2x - 2 = 0 \text{ are } -1 \pm \sqrt{3}$$

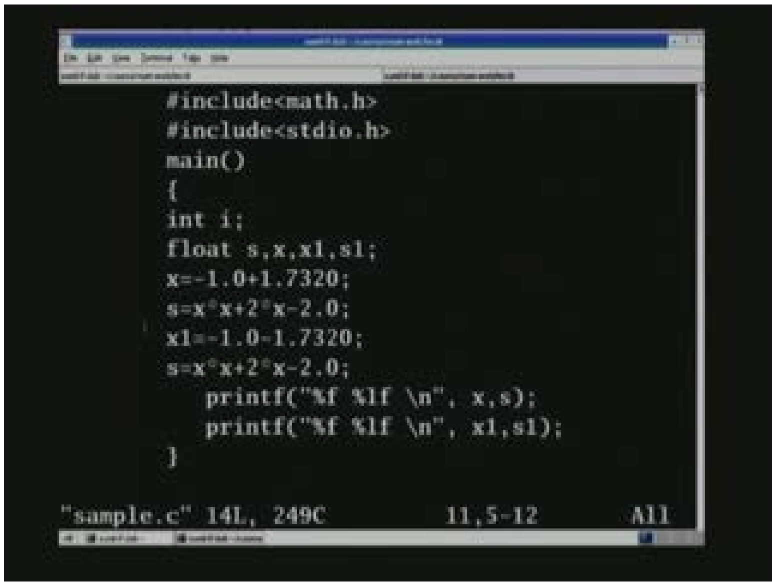
In five decimal floating point x=.7320E0 and x=.2732E1.

The question is, suppose I put this back here, this number back into the equation, what will I get? Whatever I get is my 0. So that is something which we can look at. So now, we will see that, we would see whether actually we get that 0 or not. We just look at a program which would demonstrate this. So I have a small program here which would try to look at this. So, I have represented these two solutions. One solution is minus 1 plus root 3, and I have taken the root 3 as “1.7320”.

I just took it like that because I think that is the only precision that my machine has. I am assuming that is the only precision it has, my machine has, and I put that, and then I substitute that here, and compute the value s. If this is actual solution, I should get my s equal to 0, and then I take the other root which is minus 1 minus root 3, which is “1.7320” and then I compute the value “s” again.

So I call this root with minus 1 plus root 3 as x, and the corresponding value of my quadratic equation as s, and the root with minus 1 minus root 3 as x and the corresponding value of the quadratic equation as s_1 and then I try to now compute this value, and then I print out both minus 1 plus root 3, and the quadratic equation, the value of the equation, the value of the expression, and the second root, which is minus 1 minus root 3,and the value of the expression, and then we just see what we get. Okay that is what we get.

(Refer Slide Time: 33:11)



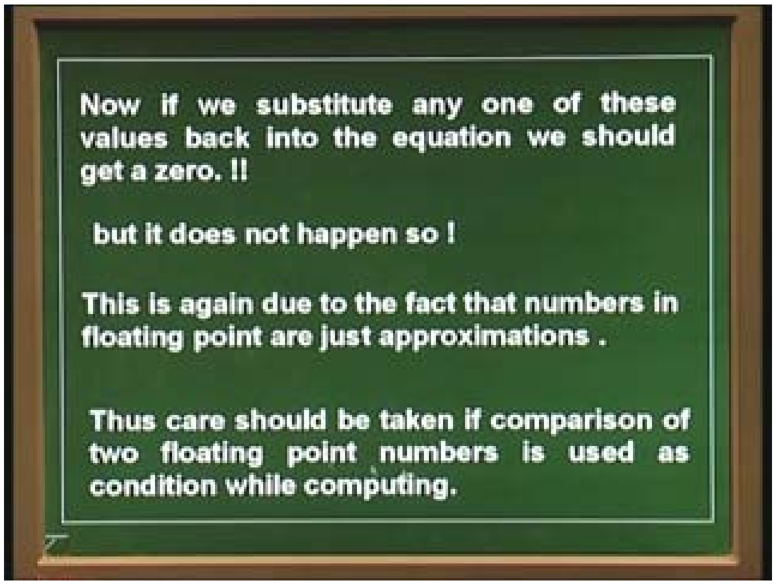
```
#include<math.h>
#include<stdio.h>
main()
{
int i;
float s,x,x1,s1;
x=-1.0+1.7320;
s=x*x+2*x-2.0;
x1=-1.0-1.7320;
s=x*x+2*x-2.0;
   printf("%f %lf \n", x,s);
   printf("%f %lf \n", x1,s1);
}

"sample.c" 14L, 249C          11,5-12        All
```

So we know what we are supposed to get is 0, because we said that it is a 0 of the, it is actually solution of this equation, or the 0 of this function. It is a solution of the equation which we wrote as minus 1 plus or minus root 3 which we know right. So now I am approximating root 3 as “.1732” 10 to the power 1. That is what, I would have, 4 digits for the mantissa correct. So I put that back here and I am looking at what is the value I get, and I see I do not get 0. So if I have 4 values, if I have 4 digits for the number, 4 digits for the mantissa, and I am going to get the 0 as “.001” right, 4 digit, if I get “.001” or if I chop it or “.002” in the case of rounding. That is, what I put the value as. So that introduces the idea of 0 on a machine.

So you have to be very careful when you are actually programing that you should not be saying, you should not be equating or a number to 0 as a condition. For example, if you want to put an if statement, and if you say the solution of the value of some expression, x minus something, x minus 3 is equal to 0, x minus “3.0” equal to 0 as a condition, and we assume that when x is a solution of, for example, x was the solution of this equation, and we said that x plus x minus of 1 plus root 3 is equal to 0 as a condition of something, and then you will get into trouble. So because 0 on the computer is not the 0, what do you think it is?

It is always actually a finite number. Okay that is what, we should be careful about. So always avoid equating to a 0. So we have to say that instead of saying a minus b equal to 0, we have to say a minus b less than some epsilon. That epsilon is the smallest number which the machine can represent. That is what you have to be careful about, Otherwise you will never get that condition correct. So thus care should be taken when you compare two floating point numbers in the condition for computing.

(Refer Slide Time: 37:38)



Now if we substitute any one of these values back into the equation we should get a zero. !!

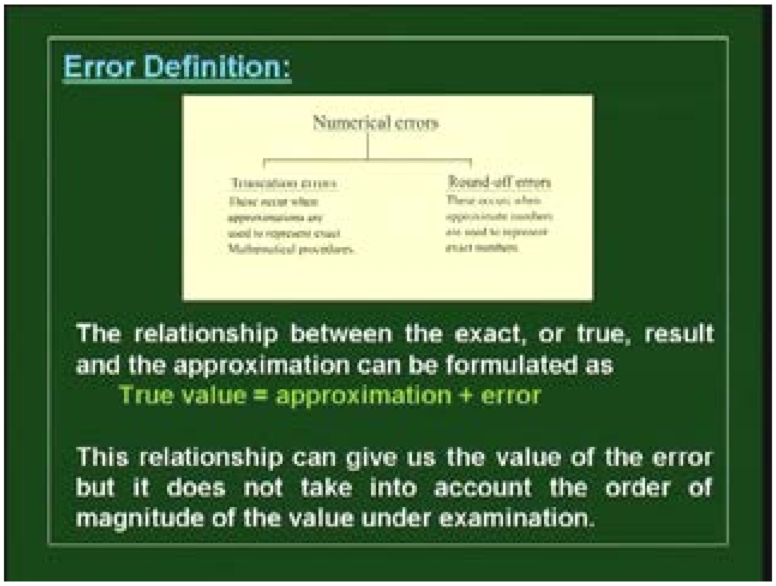
but it does not happen so !

This is again due to the fact that numbers in floating point are just approximations .

Thus care should be taken if comparison of two floating point numbers is used as condition while computing.

So now we saw that there are many errors. So we have errors coming because of the finiteness, because of rounding, chopping, and because there is only a finite number of floating points which I can store given the range, etcetera. So now, how do we quantify this? So we need some way of quantifying these errors. So that is what we would discuss next. So the errors can be classified into two. We can have errors due to truncation. As we said, we have 1 by 3, or some large number series like that, and then you need to chop it off somewhere. Or phi, for example, we have to chop off somewhere.

(Refer Slide Time: 38:27)



Error Definition:

Numerical errors

Truncation errors
These occur when approximations are used to represent exact Mathematical procedures.

Round-off errors
These occurs when approximate numbers are used to represent exact numbers.

The relationship between the exact, or true, result and the approximation can be formulated as

True value = approximation + error

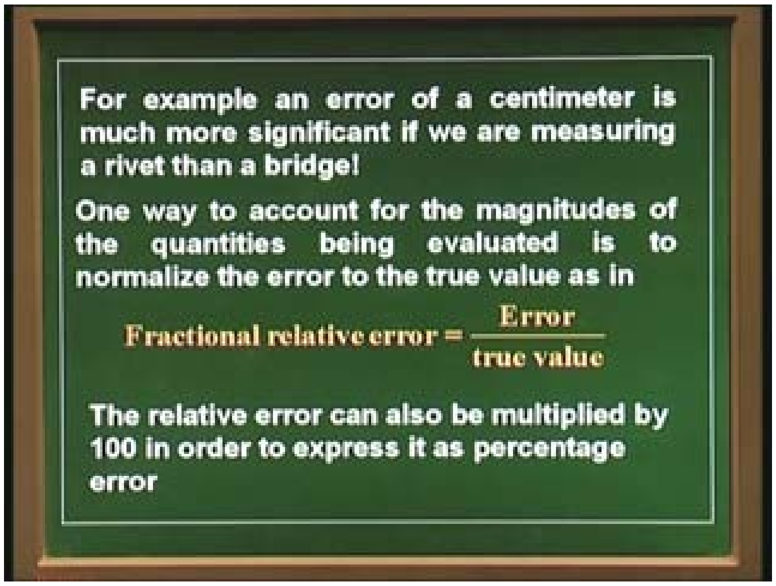
This relationship can give us the value of the error but it does not take into account the order of magnitude of the value under examination.

We need to truncate this number. Some truncation errors which come from that. And then we also have errors coming from, for example, when you add, subtract, multiply, we have some round-off errors coming from that. So here, we have these two errors. So the question is, how do we compute the errors which are and make sure that our calculations are within the error limit? Or, at least, we should be able to say I have done this calculation, and my error, possible error in this is so, an upper limit on that. How do I say that?

What is the quantities right. So that is what you should try to see. So, if you know the exact value, then we know that the true value is the approximation plus the error okay, and then this relationship we can use to actually quantify the error. But the problem with this is it does not represent, it does not give us a feeling for the magnitude of the quantity which we are measuring. For example, you could measure a distance of 1 km or 1 m, and make an error of a cm. So, according to our previous definition, that is the true value, is the approximate plus error, or if you take true value minus approximate value as error, both will have the same amount of error. That does not give us the feeling for that, so we need to have a fractional error. For example, if you measure a rivet or a bridge, and if you make an error of .1 cm, that does not have the same significance.

So we need something which represents, or which reflects that significance of the error, and that is why we use the fractional relative error. That is error divided by the true value. Okay so this can be, we will always express this in terms of percentage. We say error by the true value multiplied by 100. So that is the percentage error. So now the question is, now let us look at some examples of this. So that is ".9" of, 9999 cm, where the true value was 10,000 cm, or then, we have 9 cm where the true value was 10 cm. Let us say the measuring instrument we were using was had an error of 1 cm. So what does it mean? so error in the first case would be 1 cm. The error in the second case would be 1 cm. again that does not signify much.

(Refer Slide Time: 40:35)



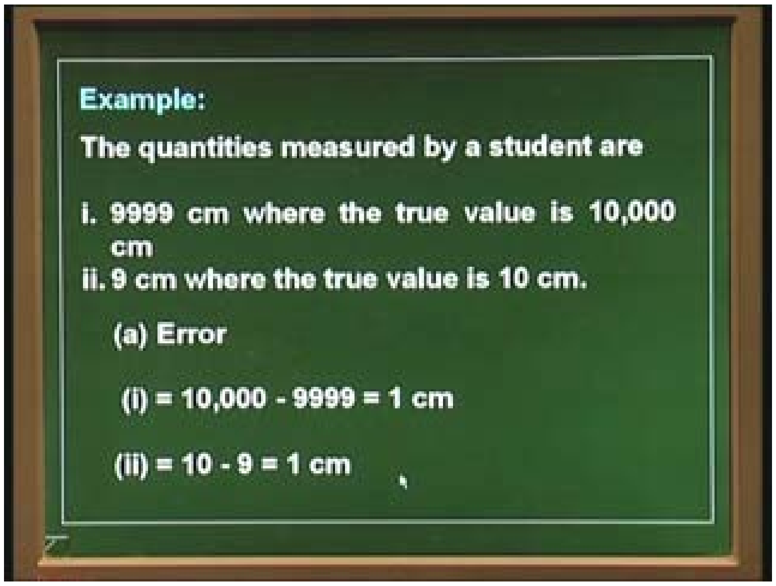
For example an error of a centimeter is much more significant if we are measuring a rivet than a bridge!

One way to account for the magnitudes of the quantities being evaluated is to normalize the error to the true value as in

$$\text{Fractional relative error} = \frac{\text{Error}}{\text{true value}}$$

The relative error can also be multiplied by 100 in order to express it as percentage error

(Refer Slide Time: 41:06)



Example:

The quantities measured by a student are

i. 9999 cm where the true value is 10,000 cm

ii. 9 cm where the true value is 10 cm.

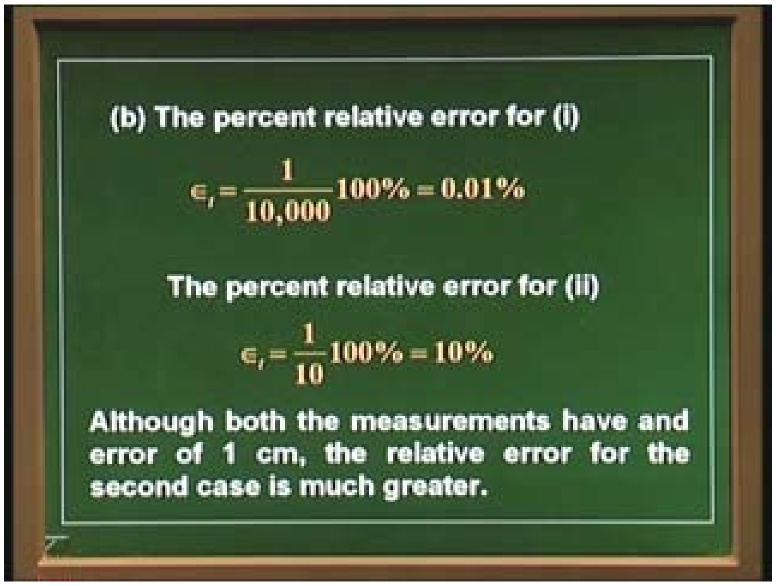
(a) Error

(i) = 10,000 - 9999 = 1 cm

(ii) = 10 - 9 = 1 cm

So, what we have to do is, we have to have the percentage error. The percentage error in the first case would be 1 by 10,000 multiplied by 100, that will be ".01" percent, while in the other case it will be 1 by 10 multiplied by 100, that is 10%. So that uses much more field. We have 10% error if we use a scale with 1 cm as the precision, and if you measure something like a rivet, or something like a 10 cm object, we have 10% error, and while if you measure the bridge with it we have an error of ".01%". That is what it means. So both the errors are 1, but the significance is different. That is what we said. Another case of importance is that we do not know the true value, and we still wanted to say what the error is. So the question is, how do we quantify that? Suppose you do not know the true value, and you still want to quantify it, so how do we do this? In that case, we can actually use two measurements, and take the relative error between that.

(Refer Slide Time: 41:57)



(b) The percent relative error for (i)

$$\epsilon_t = \frac{1}{10,000} 100\% = 0.01\%$$

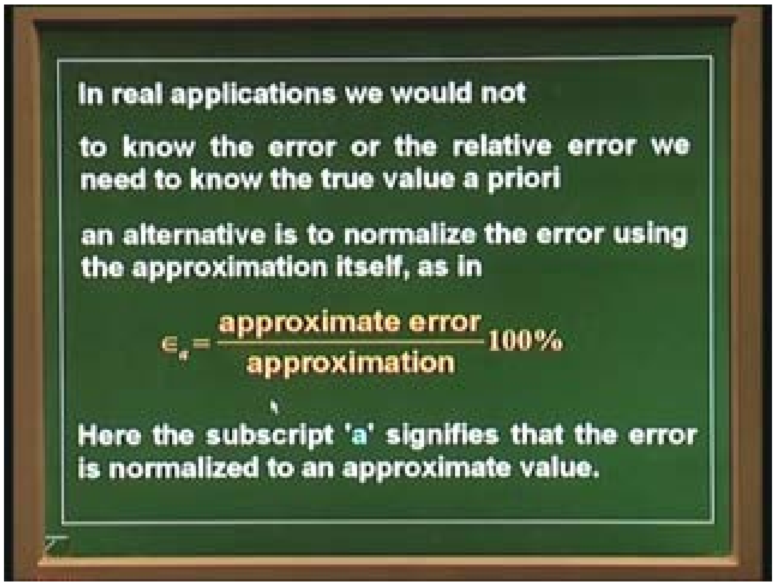
The percent relative error for (ii)

$$\epsilon_t = \frac{1}{10} 100\% = 10\%$$

Although both the measurements have and error of 1 cm, the relative error for the second case is much greater.

So that is, we have an approximate error. So that is, we will do that with again with an example here. So we have approximate error divided by approximation into 100.So that is kind of a percentage error again, but not with the true value. So normally, it is represented with a subscript, a.

(Refer Slide Time: 43:11)



In real applications we would not

to know the error or the relative error we need to know the true value a priori

an alternative is to normalize the error using the approximation itself, as in

$$\epsilon_a = \frac{\text{approximate error}}{\text{approximation}} 100\%$$

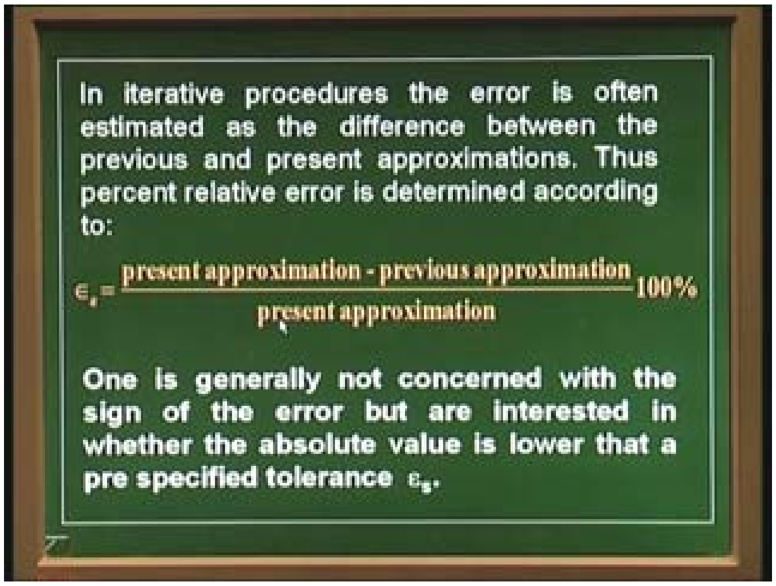
Here the subscript 'a' signifies that the error is normalized to an approximate value.

So there is a definition if you would see everywhere when you look at error analysis for numerical computing that you would see this kind of numbers represented. So, epsilon is the true error, and epsilon a is the approximate error which is probably the difference between the two measurements divided by the mean of the measurements multiplied by 100.

So, for example, iterative procedures, this is very meaningful. So you could take, let us say, if you want to measure, if you want to compute e to the power of minus ".2", something like that, and then you would say that okay I would represent that as a series, and then I chop off the series somewhere, the question is, how many terms in the series should I keep? So then for that kind of measurement, it makes sense to actually have a definition of error in this form. So as I said, you could take, for example, 2 by 3. So 2 by 3, you start doing that, you get ".6666", and then you want to know where you should stop.

How many digits you should stop, and then you say that I could use a kind of error definition, and I say my percentage, my error, in this calculation is this much, I do not care if it goes. If the percentage error is something in the order of 10 to the power of minus 3, and then I could keep different each term in the series, each term in the number of digits, and then I could compute the difference between ".66" and ".666", and then divide it by the present approximation which is ".666", and then see what the error is. The sign is not important here. What is only important here is the magnitude. So that is the kind of, that is the example which I said. Another example would be computing the exponential series, the log 1 plus x series, and similar things. So we will see that in an example, when we actually quantify this in some program.

(Refer Slide Time: 45:03)



In iterative procedures the error is often estimated as the difference between the previous and present approximations. Thus percent relative error is determined according to:

$$\epsilon_a = \frac{\text{present approximation} - \text{previous approximation}}{\text{present approximation}} 100\%$$

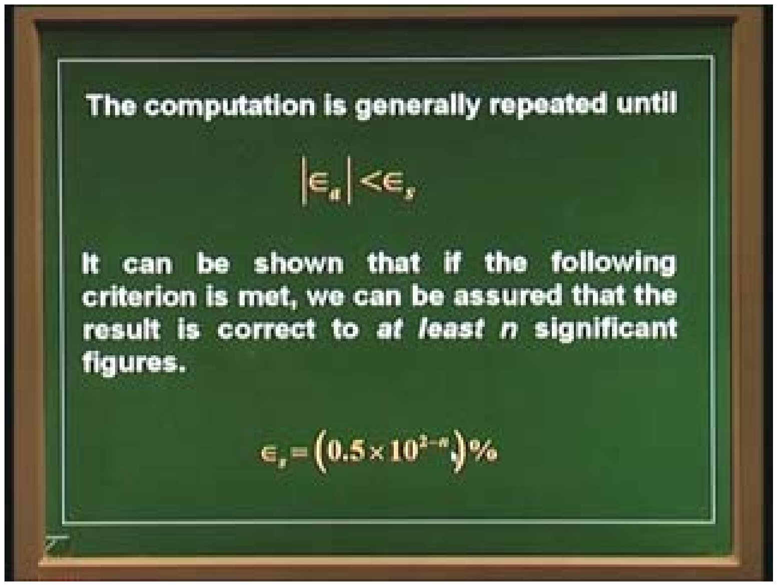
One is generally not concerned with the sign of the error but are interested in whether the absolute value is lower that a pre specified tolerance ϵ_s.

The sign is, I said the sign is not very important. What is important is only the magnitude, and so what we would do is we would compute this kind of an approximation. The present approximation, minus the previous approximation, and then divided by the present approximation, compute the percentage error. And then we compare it with some predetermined error. Like, as I said, 10 to the power of minus 3 is my predetermined error, it is my tolerance. So that is fixed, and then I compare it with that If it is below the tolerance, I say this is acceptable. This number is acceptable, and that series can be terminated then. So that is what the computation is generally repeated till we go below this limit. So now, there is an important thing to remember, that the details of which I will not go into.

It can be shown that if a criterion of this order is met, that is, if you say that epsilon, if I have a machine, if I want a precision up to a certain number of digits, that is n is my number of significant figures, then 2 on a machine with base 2, then ".5" into 10 to the power of 2 minus 1 percentage would be the would tell me that my answer is as the number of significant digits as n. That is, I have to fix my epsilon s as ".5" into 10 to the power of 2 minus n, and then, if I do a computation and make sure that my approximate errors are less than that epsilon s, then I will have n significant digits in my answer. So that is the point.

So if I want to keep n significant digits, I should have. On the other hand, if I want to keep n significant digits, then I should keep my tolerance as ".5" 10 to the power of 2 minus n percentage, or in other words, if you have a machine with n significant digits, there is no point in keeping a tolerance which is less than this. That is the maximum tolerance which you, that is the minimum tolerance which you can get in your error. If you go to next term in the series, you are not going to get a better answer, because it is not meaningful to go beyond this. So remember that ".5" into 10 to the power of 2 minus n percentage.

(Refer Slide Time: 48:02)



The computation is generally repeated until

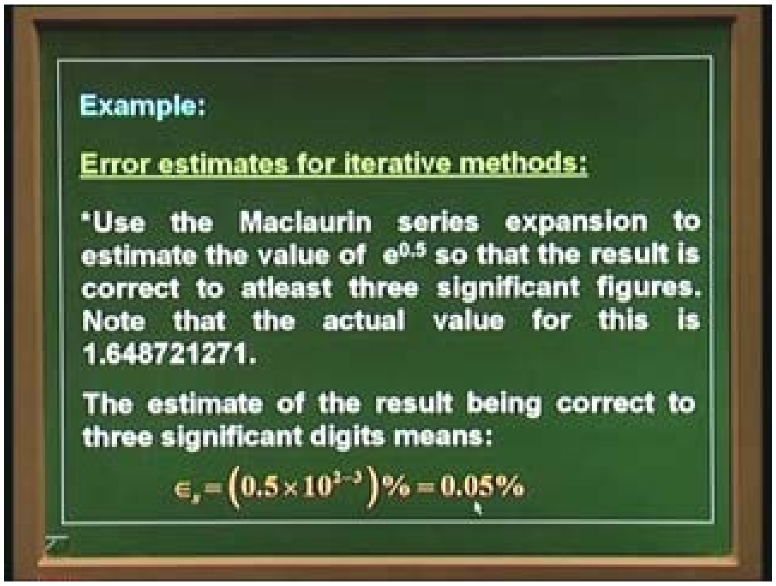
$$|\epsilon_a| < \epsilon_s$$

It can be shown that if the following criterion is met, we can be assured that the result is correct to *at least n* significant figures.

$$\epsilon_s = (0.5 \times 10^{2-n})\%$$

Okay that is, if you have 3 significant digits, and then 3 significant figures, if you want to compute e to the power of “0.5” let us say. And then the correct, you can expand this in a series okay and make sure that the difference between the two approximations in the series divided by the present approximation is “.05%”. It is below “.05%”. We should not try to go below this. It does not make sense.

(Refer Slide Time: 48:41)



Example:

Error estimates for iterative methods:

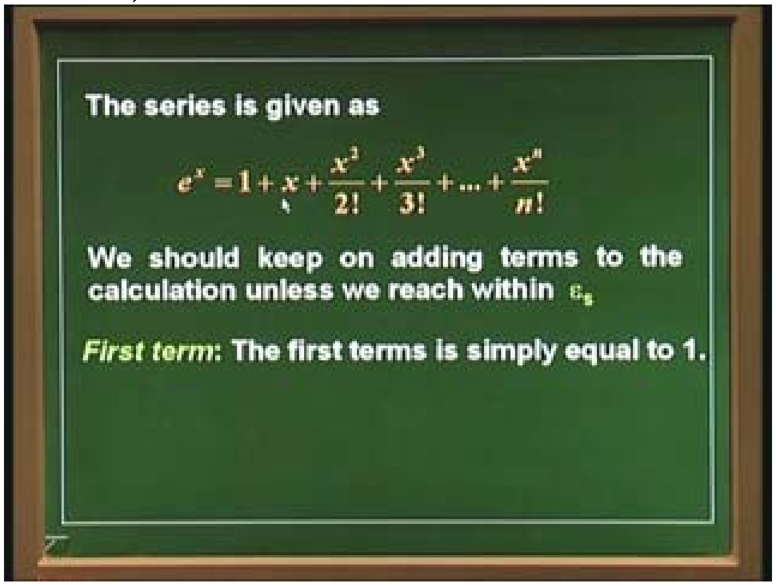
*Use the Maclaurin series expansion to estimate the value of $e^{0.5}$ so that the result is correct to atleast three significant figures. Note that the actual value for this is 1.648721271.

The estimate of the result being correct to three significant digits means:

$$\epsilon_s = \left(0.5 \times 10^{2-3}\right)\% = 0.05\%$$

So that is what we would see. The series, for example, here e to the power of x, I can write it as 1 plus x plus x square by 2 factorial x cube by 3 factorial, etcetera. We know that, and then we should keep different terms. The first term is just 1 and we just saw that if I do x to the power of .5, the correct answer is 1 point something “1.6”.

(Refer Slide Time: 49:07)



The series is given as

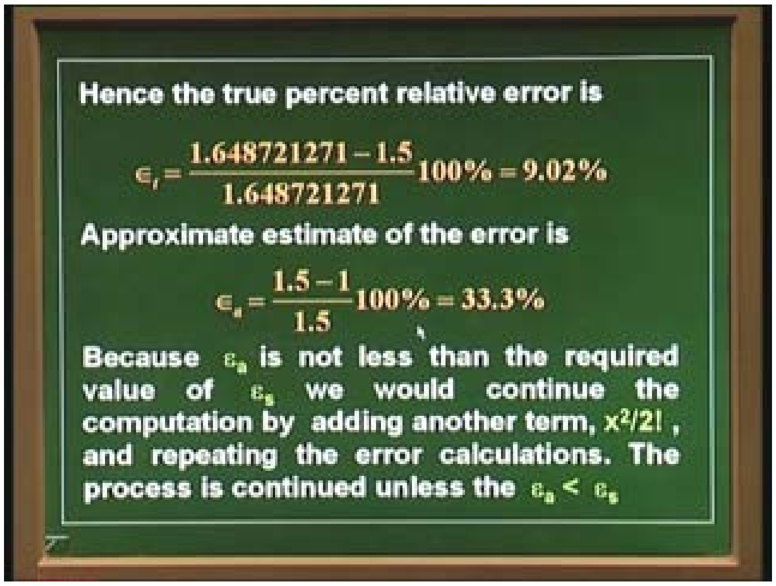
$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

We should keep on adding terms to the calculation unless we reach within ϵ_s

First term: The first terms is simply equal to 1.

So we have "1.64872", let us say, as the correct thing. So, if you do the, just the first term, then we get 1, so the and if you give the second term, you are getting "1.5".So, we take "1.64" minus "1.5" divided by "1.5", that is one possibility. That is a true error, or you could say that my relative error here is 1 minus "1.5" "1.5" minus 1, since we are not bothered about sign "1.5" minus 1 divided by "1.5" into 100. So that should be less than ".05", that is what we wanted to do okay. So here the first term, the true error is "1.648" minus "1.5" divided by "1.648" into 100 that is 9 percent, while my approximate error was "1.5" minus 1 divided by "1.5", and that is 33 percent. So we go to the next term in the series, etcetera, till this error goes to ".05 percent". That is what we saw, is the error which we can tolerate.

(Refer Slide Time: 50:16)



Hence the true percent relative error is

$$\epsilon_t = \frac{1.648721271 - 1.5}{1.648721271} 100\% = 9.02\%$$

Approximate estimate of the error is

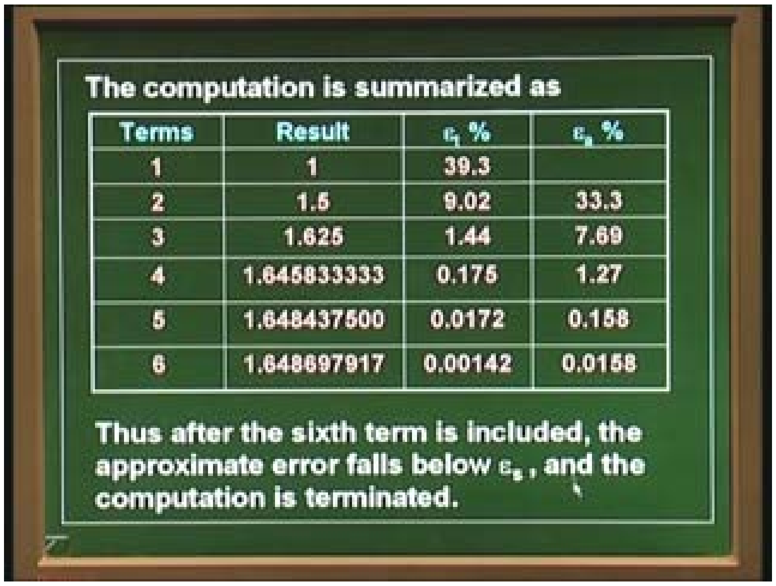
$$\epsilon_a = \frac{1.5 - 1}{1.5} 100\% = 33.3\%$$

Because ε_a is not less than the required value of ε_s we would continue the computation by adding another term, $x^2/2!$, and repeating the error calculations. The process is continued unless the $\varepsilon_a < \varepsilon_s$

Okay so we go to the next term and continue this series. So the we can summarize this computation by taking various terms in the series. We will keep going down on the percentage error, and the true error, sorry, approximate error, and the true error. We can actually see that the true error converts itself faster than the approximate error okay. So now, we should look at some problems which is just listed here. So you can try to find the error in the value of x cube, and x given by 4.0 by 3.0, and then you could approximate it with different representations, this floating point here 4 by 3. So 4.0 by 3.0. You could approximate it with different floating point representations and compute the errors.

That is one example which you should try. For example, you could try it with base 10 and keep the 4 significant figures on the mantissa, and take an exponent which goes from minus 2 to plus 3, or you do with base 2 and keep 4 significant digits, and then go from minus 2 to plus 3 with the exponent from minus 2 to plus 3 and compute what the error is. So we would now look at how does this error propagate? As this we saw that every arithmetic operation would introduce some error, and we know how to compute this error.

(Refer Slide Time: 50:49)

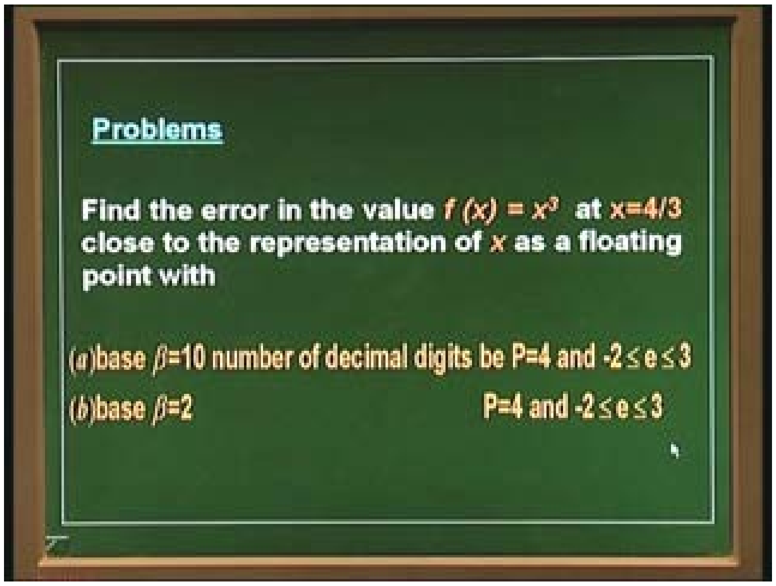


The computation is summarized as

Terms	Result	ε_t %	ε_a %
1	1	39.3	
2	1.5	9.02	33.3
3	1.625	1.44	7.69
4	1.645833333	0.175	1.27
5	1.648437500	0.0172	0.158
6	1.648697917	0.00142	0.0158

Thus after the sixth term is included, the approximate error falls below ε_s, and the computation is terminated.

(Refer Slide Time: 51:51)



Problems

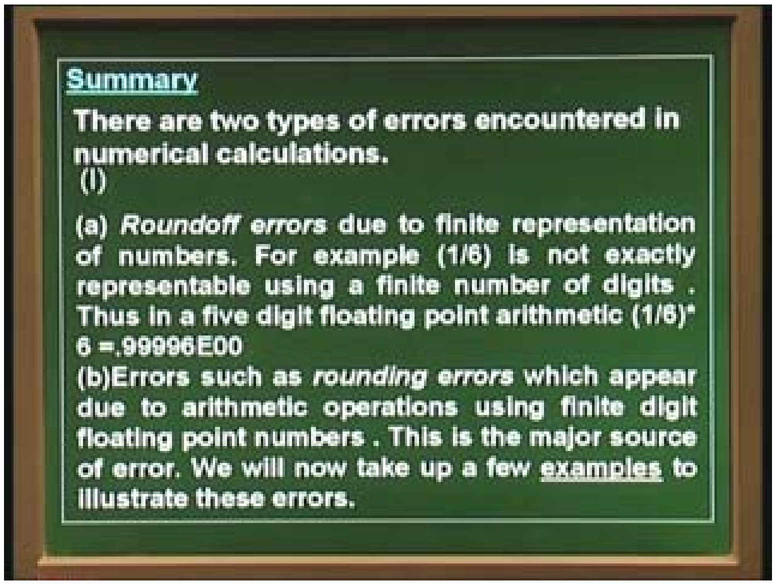
Find the error in the value $f(x) = x^3$ at $x=4/3$ close to the representation of x as a floating point with

(a) base $\beta=10$ number of decimal digits be P=4 and $-2 \le e \le 3$

(b) base $\beta=2$ P=4 and $-2 \le e \le 3$

Another important thing to actually figure out is how do these errors propagate in a computation, and that is something which we would see later. So, in summary, we saw that there are two types of errors encountered in the numerical computation. We have round-off error due to finite representation of numbers, that is, we said, for example, 1 by 6 cannot be represented completely, so we have to represent it as some approximation to this, because this is a finite number of digits. And then we have rounding errors which appear due to arithmetic operations using finite digits which also we saw in, using few examples, so then we have truncation errors. For example, if you have a series, then you would truncate that.

(Refer Slide Time: 52:52)



Summary

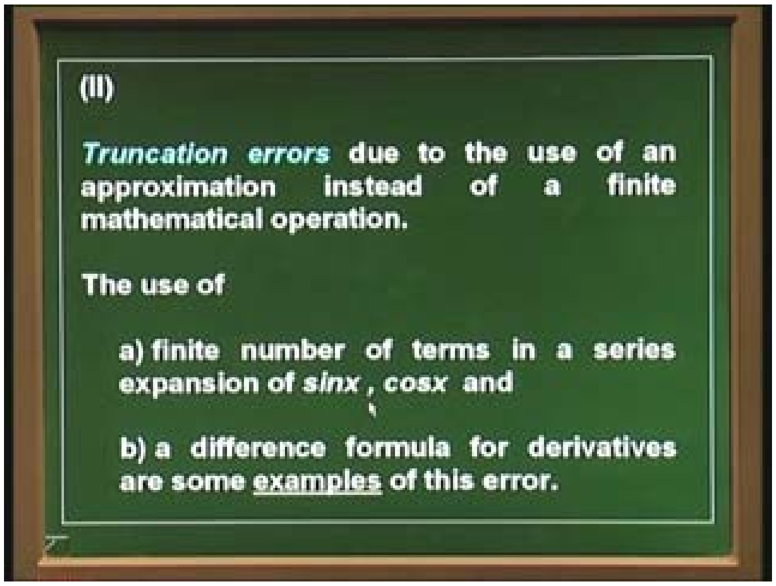
There are two types of errors encountered in numerical calculations.

(I)

(a) *Roundoff errors* due to finite representation of numbers. For example (1/6) is not exactly representable using a finite number of digits . Thus in a five digit floating point arithmetic (1/6)* 6 =.99996E00

(b)Errors such as *rounding errors* which appear due to arithmetic operations using finite digit floating point numbers . This is the major source of error. We will now take up a few examples to illustrate these errors.

(Refer Slide Time: 53:36)



(II)

Truncation errors due to the use of an approximation instead of a finite mathematical operation.

The use of

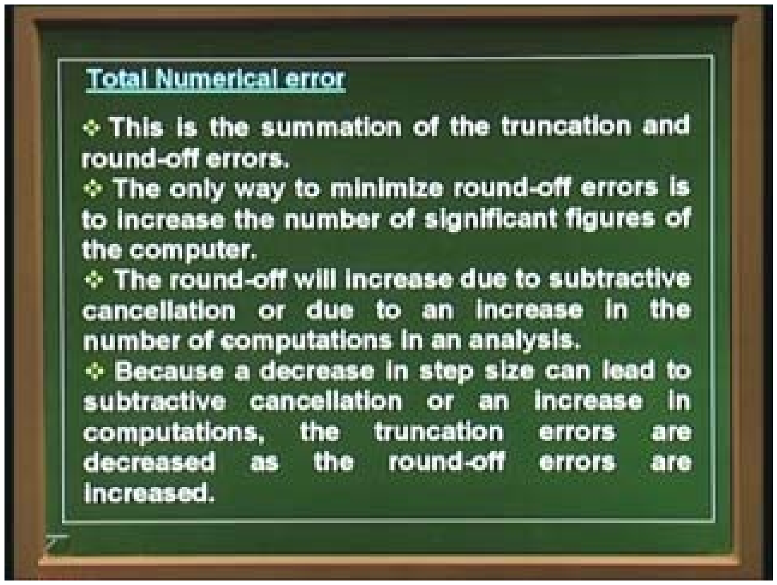
a) finite number of terms in a series expansion of *sinx* , *cosx* and

b) a difference formula for derivatives are some examples of this error.

So we have finite representation of numbers, and then we have, when you do arithmetic addition, subtraction, we have round-off errors, and then we do series expansion, for example, we have truncation errors all coming from the finiteness of the number. For example, instead of using a full expansion for a sin x, or a cos x, or log x, we would use some kind of a series, and we truncate that. That is, it gives you some kind of truncation error.

Okay so we would look at examples of this and actually how they would come in the next class, so the total numerical errors in that way, so now we can summarize it here. The total numerical error is the summation of truncation and round-off error. So, if you want to minimize this round-off errors, we know round-off because of the number of significant digits are finite,

(Refer Slide Time: 54:06)



Total Numerical error

- This is the summation of the truncation and round-off errors.
- The only way to minimize round-off errors is to increase the number of significant figures of the computer.
- The round-off will increase due to subtractive cancellation or due to an increase in the number of computations in an analysis.
- Because a decrease in step size can lead to subtractive cancellation or an increase in computations, the truncation errors are decreased as the round-off errors are increased.

So the only way we can actually minimize the round-off errors is to increase the significant figures in the computer, but the round-off errors will increase due to subtractive cancellation, or due to the increase in the number of computations. That is also true.

So now, if you try to-, for example, in a series, if you try to, in a computation, if you try to increase the number of steps, and then you have subtractive cancellations coming in which also increase the errors, so this part, there is actually a tradeoff in increasing the round-off errors, and also the truncation errors. So that is something which we should look at in detail, and that is something which we would do in the next class. So I will stop with here. I will continue from this propagation of errors, or actually quantifying the different types of numerical errors which is coming in, and what is the optimum method to compute various quantities, that is what we should be looking at in the next class.