Finite Element Method and Computational Structural Dynamics
Prof. Manish Shrikhande
Department of Earthquake Engineering
Indian Institute of Technology, Roorkee

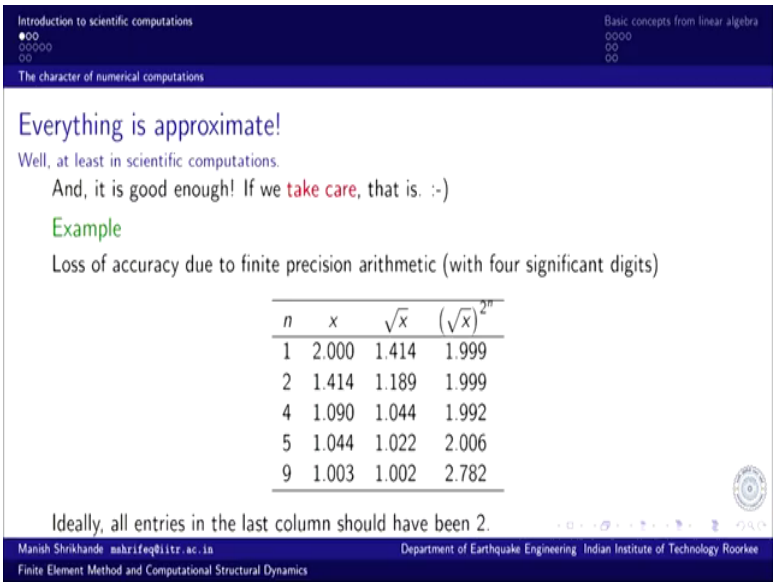
Lecture - 01
Introduction to Scientific Computations - I

Hello. So, we begin this lecture course on Finite Element Method and Computational Structural Dynamics with the basic concepts of what is meant by scientific computations. Finite element method is of course approximate method. An approximate method for solution of partial differential equations and partial differential equations are the soul of all engineering analysis all sub domains of science and engineering.

So, finite element method is an approximate technique and when we talk of approximations, it is very crucial to understand what constitutes a good approximation and in order to understand what are the measures that we use to qualify to understand the quality of approximation.

We will start this course with a discussion on basic understanding, what is involved in scientific computations, what are the sources of errors and some basic concepts from linear algebra, which is very important for constructing any approximation.

(Refer Slide Time: 01:36)



So, to begin with, as I said earlier, we are looking at approximate solution and everything is approximate. So, all solutions anything that we do on digital computers, any numerical computation, it is an approximate solution and it is good enough for all practical purposes. We can design all over complex engineering systems based on these approximate results. So, it is not as bad as it sounds approximation. We can have a really good approximation provided we take care.

So, that when we say we take care, when I say when we take care, it actually means that we should be able to control the errors from propagating and find out ways and understand what is going on in our computations in the process of computations and take appropriate measures to keep the errors from blowing up.

Now, just to put this in context, I am sure everybody must have done this sort of activity punching a very large number on your scientific calculator and then, repeatedly taking square root and ending up with 1 and after that no matter how many times you try to square it, you can never get the original result.

So, that is although we may not have paid much attention to the significance of this result, but that hits at the crux of the problem, loss of accuracy in subsequent use of results of previous analysis. So, just to drive home the point, let me take this example.

I am retaining only four significant digits in the calculations, just to keep the numbers within one slide. So, what I am doing here is take a number x that is in this case I am taking 2 and take its square root and then, square it back.

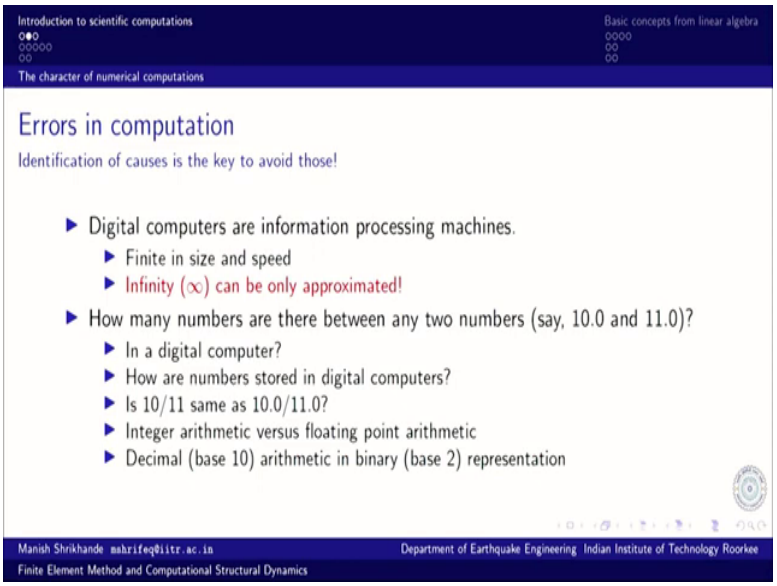
So, 1.414 is the square root and then, it is a square again is 1.999. So, we have some loss of accuracy here, and then what I do is now take this number and take it square root again and then, make it power 4 and so on. I keep on doing it up to 9 cycles and ideally, if everything was alright, if we had no loss of accuracy, if we were working with exact calculations; then, the last column should have been ideally been equal to 2.0 absolutely no difference from the original number that we started.

Now, the problem is as you can see in as early as the 9th cycle, we have a number which is obviously we cannot by any stretch of imagination, can call it a good approximation of number 2. Up to 5 iterations, it is close enough, if we can call it a good approximation of the number 2.

But 9th cycle the number is very different and by no stretch of imagination can it be called a good approximation for number 2 and that is the problem that I have been talking about the loss of accuracy during numerical computations on digital computers and then, if we use the results of previous computation in subsequent computation, then these errors are propagated and they keep on accumulating and that is a major source of problem. And we have to guard against this particular source of error because it is very surreptitious. This error creeps in surreptitiously, we have not done anything wrong as far as the operators are concerned, mathematical operations are concerned. We are using all the operations consistently and correctly. Also the error is kind of hidden in the computation, hidden in the details and if we are not careful, if one is not aware of this loss of precision during the computation, we may never be able to figure out why the results are not coming out as expected, why the results are off the mark sometimes and if we know the result.

If we do not know the result, then it is a catastrophe waiting to heaven because unwittingly, we might be using a wrong result for designing all our complex systems; complex engineering systems.

(Refer Slide Time: 06:50)



So, why do these errors occur? As we all know it is in information age and digital computers are the basic tool of this information age and they are the information processing machine. Just as in industrial age, the machines they were essentially energy transformation machines; in information age, digital computers are the machines, which transform information, one set

of information is transformed into another set of information and they are very good and useful at that.

But anything manmade is obviously finite in size and speed and computers, digital computers are also finite in size and speed. No matter what computer we are talking about, it can be as small as trivial as cell phones in our pockets or tablets or very expensive supercomputer, large supercomputer that we have. No matter what, they are all finite machines, finite in size and finite in speed.

The implications of this finite that I am harping upon is very important in the context of our concept of number line. Real number line, we have been brought up with the basic idea that number line is infinite. It extends from minus infinity to plus infinity and then, in between any two numbers, we can pack in another set of infinite set of numbers.

So, now, that is a problem with computers which are which is finite, how do we represent infinity? So, the problem is infinity can only be approximated? Infinity cannot be represented on a finite machine. Now, that is a problem, infinity can only be approximated. So, what are the implications? So, just to appreciate the implication of this a simple statement that we have to approximate the infinity. So, reiterating the earlier question how many numbers can we pack in between any two numbers, let us say 10.0 and 11.0? Immediate reply would come as infinity that is what we have been taught. We understand the number system and real numbers and why just 10 and 11, 10.0 and 10.1, we can still fit in infinite numbers in between that. But is it possible in a digital computer? As I said infinity is not possible in any form. So, it is not just the range not just the n value (minus infinity to plus infinity), it is the numbers in between any two consecutive numbers that we may choose.

So, even in that the infinite population, infinite fill ins that we take for granted in case of a real number line is not available in a digital computer. So, this follows from basically the numbers, how the numbers are stored in digital computers and that is what we need to understand in order to appreciate why the errors occur; why the errors in computation occur because of in digital computations or computations on digital computers. And once, we understand that, then we may be able to figure out or we may be able to rearrange our computations in such a way that these errors are kept in check and we can prevent errors from blowing up. So, more on that later. So, another curious thing about digital computers is the number representation as integers and real numbers.

So, is $\frac{10}{11}\frac{10}{11}$ same as $\frac{10.0}{11.0}\frac{10.0}{11.0}$? These are two different types of number representation; one is first one is the integer representation, 10 as an integer and 11 as an integer and their ratio as an integer division and the second one is 10.0 that is a real number with whole part and fractional part and 11.0 that is a real number full whole part and fractional part and their division as a division of real number ratio. So, floating point operation as we call it. So, these two behave differently integer arithmetic works differently than the floating-point arithmetic on a digital computer, again based on the way the numbers are represented.

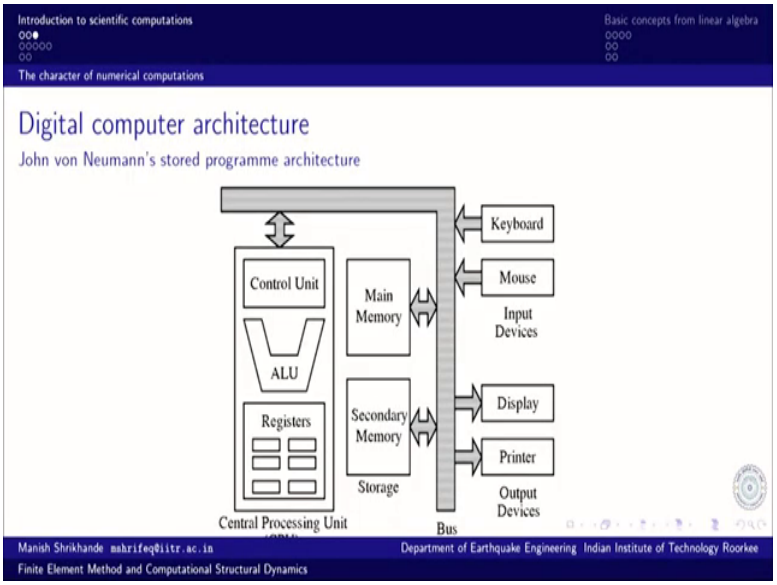
Another typical issue is we understand (the human beings understand) the decimal number system, base 10 number system. That is how we started counting because of we have 10 fingers and earliest human being started counting on fingers and that is how we have grown to understand counting in the multiples of 10. So, decimal number system is very kind of hard wired into our brains and we understand the numbers very easily, when they are expressed in decimal number system.

On the other hand, modern day digital computers, they all operate on binary representation. Binary representation because it is very easy, it is possible to design a very robust and reliable circuitry, which only has to distinguish between two voltage levels; high and low, high voltage and low voltage and that is a binary representation.

So, now, we have a dichotomy, in the sense that the input that human being has to provide to computer has to be in decimal number system, the output has to be in decimal number system and the internal calculations, internal representation in the computer has to be in a binary representation.

So, we have this a conversion from decimal to binary and then, binary to decimal during the input-output and that is how digital computers work. So, it is only the representation, when interfacing with interacting with the human beings that is the digital number decimal numbers are used; internal representation is all binary.

(Refer Slide Time: 13:55)

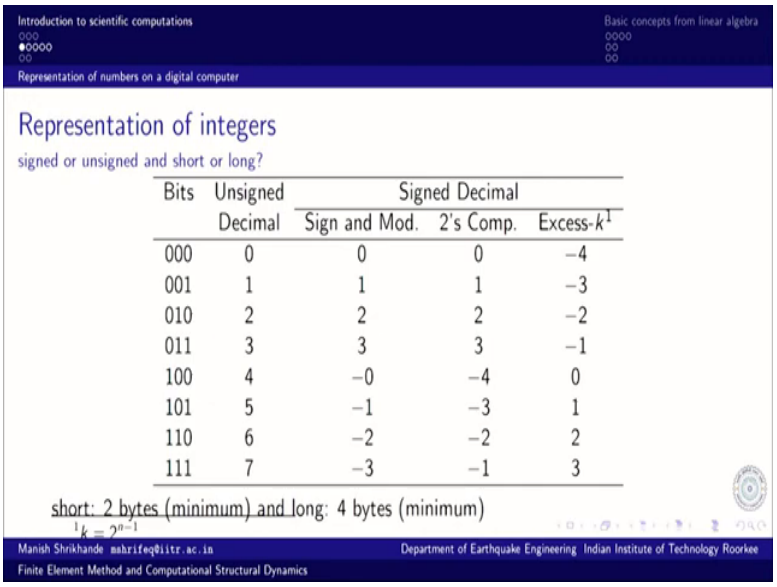


And just for the sake of completeness we will discuss a very simple schematic representation of the organization of a digital computer, which we call as a von Neumann's stored programme architecture. All present day computers are based on this architecture.

So, as you can see this comprises of a basic central processing unit and there is a control unit and arithmetic logic unit that does the basic operations and there is a small set of very small set of memory that we call as registers. So, this is a very high speed memory. In general, you can as a rule of thumb, we can say that the storage or the input output speed increases or time for input output increases, the further it is a the storage is away from the control unit or the central processing unit.

So, registers are the fastest memory, fastest storage area which is actually built into the chip basic microchip. Then, we have level to cache and RAM and then, the larger storage that is we call secondary storage. So, RAM is of course, faster to access than the large storage, and secondary storage is of course, slower to access. But then, there are peripheral devices. All of these are connected through what we call as communication bus, that allows communication between different units with the central processing unit and all peripheral devices are the lowest of all, they are even slower than the main storage. So, that is the basic computer architecture that we have and all modern day computers, they are based on this computer architecture and as I said, it is all based on binary representation; everything happens in the binary mode.

(Refer Slide Time: 16:07)



So, how are the numbers represented? So, first is as I said integer representation. So, integers if you have little bit some exposure to programming, then you might realize you might recall that there are different types of integers. We can have signed integers, we can have unsigned integers and then, we can have long integer and short integer that is depending on different lengths of the bits that are set aside for one size.

So, just an example for different types of integers, I used this 3 bit system representation. So, for different bit pattern, you can see the mapping of decimal equivalent decimal numbers. So, on in the first column, it is all binary representation, all bit patterns that we can possibly have for 3 bits. Second column is the unsigned integer in decimal number system. So, 3 bits that would be ranging from 0 to 2^3-1. So, that is 0 to 7; so, a total of 8 (2^3) representation. So, these are the total possible representation (0 to 7).

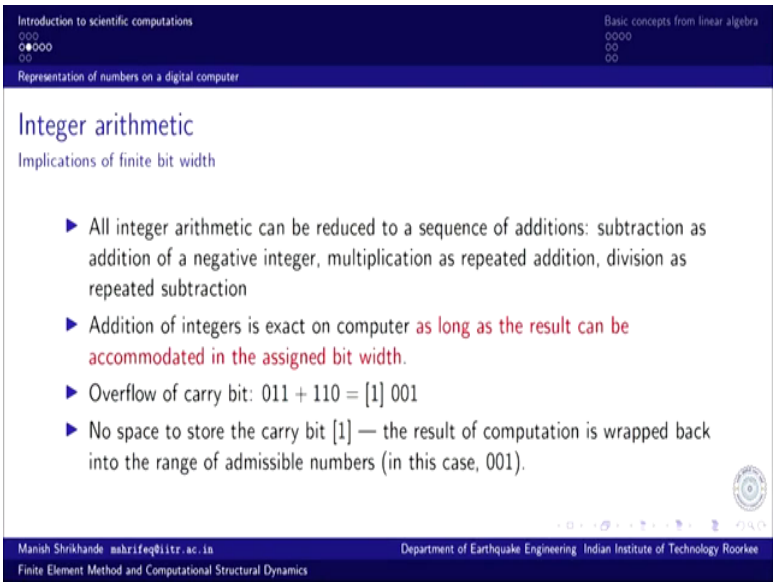
And then theoretically an integer extends from both negative range and positive up to positive range. So, we have to technically correct representation of an integer is of course, signed integer. Then, there are different ways of representing the signed integers again based on convention.

We can look at it as the left most bit that we have as a sign bit and then, rest of the two as the number representation. So, if I consider 0 as the positive sign and 1 as an indication of a negative sign. So, that would represent the first three numbers as + 0, + 1, + 2, +3… and subsequently, we would have -0, -1, -2, -3… so, very simple representation.

It is all again it based now it is based on convention. Once we agree to a common representation, then everybody understands what is the interpretation; how these bit patterns are to be interpreted and then of course, once these rules are defined, then there is no confusion and it can be done. It can be understood very well. The only problem with sign and modules form is a little inconvenient notion that we have two representations of 0; we have plus 0 and minus 0, otherwise it is all fine.

Another representation is 2's complement, again based on some representation of some standard transformation rule and third one is Excess k. Excess k is something that is similar to shift of origin. So, instead of having this unsigned decimal representation from 0 to 7, if I say instead of having 0 at 0 representation, I move my origin to somewhere in between middle of the range and that is the constant shift that I impart and that brings the entire representation. So, I extend my range almost half way from minus into some minus integer to some positive integer, almost equal value. So, that is how these are the three different representations for signed integers and all of them are used in different context. So, that is the reason, we are looking at these and we will have more on this little bit in a little while.

(Refer Slide Time: 20:27)



So, all integer arithmetic, the basic operation of integer arithmetic is it can be reduced to a sequence of additions and subtraction between two integers can be considered as addition of a negative integer to another integer. Multiplication can be represented as a repeated additions

and division as a repeated subtraction and that actually allows us to build a computer with just a addition.

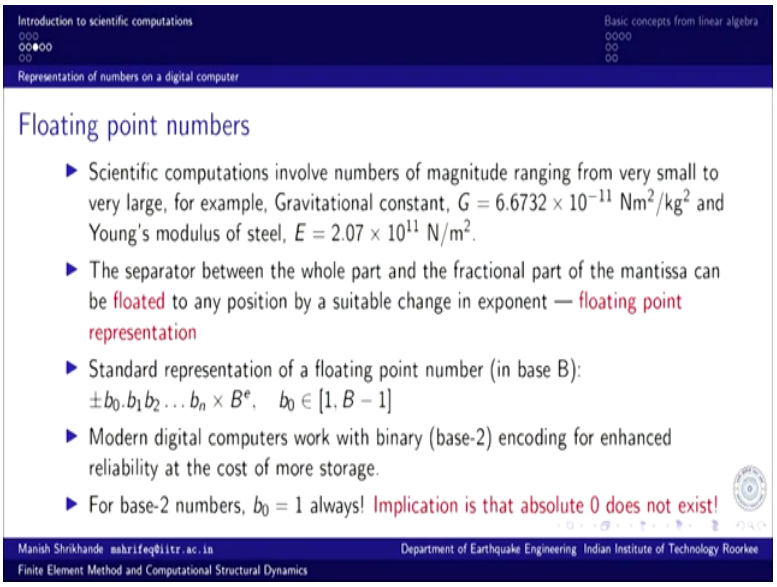
So, addition of integers is exact on computers. So, integer arithmetic is exact, there is no error whatsoever as far as addition operation is concerned. As long as the result can be accommodated in the word length whatever we have assigned, the bit width. More on this, this is an example of what I mean that by if the result can be accommodated as long as in the assigned bit width.

As you can see this overflow, whenever we have this addition, there is this carryover of fourth bit. Now, there is nothing to store, the bit that is in the square brackets, there is no space to store this particular number and this is lost; this is just lost away after the addition and the result would be erroneously reported as 001 that is just one unsigned decimal one.

Now, that is something we can look at or relate to as the milometer odometer reading after a certain range 9 9 9 9 9 and then, it again goes back to 0 0 0 0 0. It is something like this; not something, it is exactly like this because there is no other no extra space to carry that carry or to store that carryover bit and that carry over bit is lost and we start all new in the same cycle. So, the results are cycled back into the same range of numbers. So, this is the only error that can happen in a integer arithmetic and one has to be careful about this overflow error. This can happen very silently and no one would know, no warning is issued and we may we wondering what is going wrong, if we are not aware of this particular type of error.

So, as I said, the result of the computation is wrapped back into the range of admissible numbers. In this case 001 and this error is very silent, I mean it is like a silent killer. Here, we may compute happily and use the results happily without even knowing that this result is actually the wrong result.

(Refer Slide Time: 23:30)



Now, coming back to the real numbers, the floating-point numbers, so scientific computations involve numbers of magnitudes ranging from very small to very large. Just for an example gravitational constant is a very small number about 6.6 x 10^{-11}Nm^2 /kg^2 and compared that with Young's modulus of steel, that would be 2.07 x 10^{11} N/m^2.

Now, these are two, I chose these two numbers for the similar range of exponent of 10. So, one is 10^{-11}, the other one is 10^{11}. Now, this is a wide range and of course, there are mass of electrons very small number, this speed of light and energy released during a atomic reaction or very large numbers.

So, we have to deal with very small numbers and very large numbers and obviously, no person no computer can accommodate, can have the storage to represent these numbers as integers. I mean the word length that would be required would be humongous and that would be simply beyond the capacity of even the largest computer to deal with such kind of numbers if we provide this kind of range.

So, we have to deal with, we have to find a mechanism of doing computations which can deal with very small numbers as well as very large numbers and that is what we use floating point numbers, we call them. This is the floating point number as you can see, this decimal point that can be shifted by suitable change in the exponent field.

So, 6.6732×10^{-11} can be written as 0.66732×10^{-10}. So, the decimal point that is the separator between the whole part and the fractional part, that can be floated around by change in exponent and that is why it is called a floating point representation. So, the standard format for a floating point number in any base, it can be what we show what we are seeing these numbers, I have written it in decimal number system base 10. But it can be in any base. So, let us say if it is our base B, then it is a written as $\pm b_0.b_1.b_2.b_3\ldots\ldots\ldots b_n \times B^e$. where e is the exponent whatever is the exponent number and B is the base of the number system.

So, that is how you have that 10 there in the decimal number system, 10 is the base of the number system. Now, the important thing to note here in this floating-point representation is the whole part whole part b_0 has to be a number between 1 and B-1. So, in case of decimal number system, it has to be standard representation would involve a whole part to be a number between 1 and 9 right.

So, 0.66732×10^{-10} is not a standard representation. Standard representation is 6.6732×10^{-11}. So, that is a standard representation. So, this is the floating-point representation in base B. Now, what happens in case of binary representation? Digital computers work with binary base-2 encoding for enhanced reliability and of course, if it is enhanced binary coding requires more space to store the information.

So, cost of storage is of course coming down, I mean it really does not cost much to store now. So, we can actually have large storage and reliability of computers is much more. So, if the base is 2, if the base of the number system is 2, then very peculiar thing happens here. The whole part b_0 has to be only 1 always. It can always be 1; b_0 has to be between 1 and 1. So, that there is no choice. The number is always one point something multiplied by 2 raised to the power something and that is how we deal with it and once, it is done that, it is it has to be 1; then, we can just assume keep in mind that storage was not so cheap always. In 1970s, 1960s, it was very precious commodity.

So, it did matter a lot if we could save even a single bit in 1 a number representation. So, now, when look at that look at this from that perspective. If I can say 1 bit for by not storing something that is that I can assume to be there always, that is the whole part of this mantissa b_0 is equal to 1 always, then I do not need to store it. I can only always assume that it is always there; 1 is always there.

So, the number becomes 1 point something b_1, b_2, b_3 multiplied by 2 raised to the power something. Now, the implication of this is the number 0 does not exist. It is always something; 1 point something, something multiplied by 2 raised to the power something. So, 2 raised to the power may be something negative, some large exponent; but that still is a very small number, but still not equal to 0.

And that is a very crucial thing that we need to understand that absolute 0 does not exist on floating point numbers that are available on digital computers. In other words, I would like to emphasize, it is impossible to arrive at a result that is equal to 0 by way of computation on a digital computer. Think about the enormity of this statement and we will continue from this point in our next lecture.

Thank you.