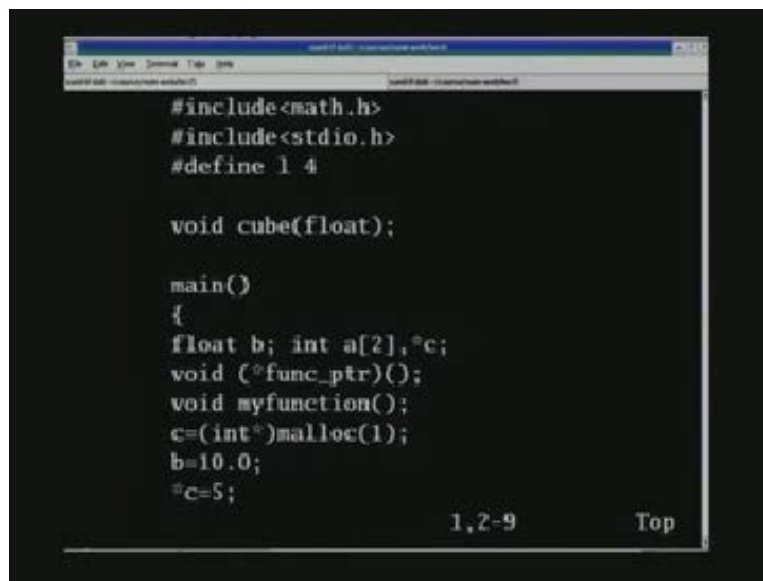


Numerical Methods and Programming
P. B. Sunil Kumar
Department of Physics
Indian Institute of Technology, Madras
Lecture - 5
Programing Representation of Numbers

In the last few lectures we saw some elements of C programming. I will just summarize in this program here, shown, what are the things which we learnt.

(Refer slide time: 01:29)



```
#include<math.h>
#include<stdio.h>
#define 1 4

void cube(float);

main()
{
float b; int a[2], *c;
void (*func_ptr)();
void myfunction();
c=(int*)malloc(1);
b=10.0;
*c=5;
}
```

1,2-9 Top

So this program, this sample program, contains almost everything which we did in the last few lectures. For example, this tells you what is the header, how do one construct a program. So there is a main program. The program has a part, some main body, and then there are some functions which is the part of the program. There, before the main program starts, you have the normal include files which has all the libraries, math library, input output libraries, etcetera. Okay and then you have some constant definitions. For example, you have defined some number here, and then you have some external functions defined. For example, here I have defined an external function called cube, which is also defined outside the main program. So, all external functions are to be defined outside the main program. So this, and then there is a main program which main program and then which calls the functions.

Inside the main program we have definitions of variables and in which we saw, there are 2 types of variables. We can define which is actually, you can have 3 types. I am showing two types here, which is floating point variables and integer variables. And among the variables itself you could have array of variables, and then we saw that we could have pointers. These are the things which we saw in the last few lectures. And then we said that the pointers could be to a variable. And we said array name itself is a pointer to the array. And we said that we could have a pointer to a function, and that is here. This is the way to define a pointer to a function, and when we said that, when we define a pointer to function, we have to say what type of argument it is

going to return, or this particular function does not return anything, So it is void. So if it is returning an integer, you have to say integer, and if it is returning a floating point you have to say float. So here is a declaration of a function. So there is a function declared here. Again, it does not return anything. Okay and here is a declaration to a function pointer.

Then we saw that we can have these pointers as to be we can have dynamic location of memory. Here is a way we allocate memory to a pointer function, pointer variable, to a pointer. So we are allocating memory using this function called MALLOC, and the assignment statements, we can, once you have allocated memory to a pointer, we can assign variables to it and they are all assignment statements, and we said that we can one is having defined a function pointer. We could actually point that function pointer to a particular function, and that is what this statement is. Remember, I have defined the function here, and there is a function pointer, and then I can point this function pointer to a particular function, and then, we saw printf statements and various formats.

We saw percentage f for floating point printing, and percentage d for integer printing, and we also saw in the previous lecture percentage u can be used; how to print addresses to which pointers are pointing, and then this is the way to call. Once you have pointed the function pointer to a function we said we can actually call that function by invoking the pointer itself, not by these kinds of statements. So that is what we have said. Now, we also saw that we have declared a function as an external object, that is as an external function, and having defined that way and you could actually pass the address of that function which you have declared externally to another function, and this is actually demonstrating that.

We have this function called cube here, which we declared as an external function, and we can pass the address of that function, that is, by saying ampersand cube to another function. In this case, we are passing it to my function through this pointer function pointer. That is what we do. And then the function pointer, my function, here gets it as some other name. It does not matter. So, and then it can execute that, so again the structure of a function. Here again, the type which is void because it is not returning anything, and the name of the function, the arguments, and the argument declarations. So this, as we saw that these declarations can also be here, we could say float x instead of x. Then we do not have to declare it here. Integer y of 2, we could say or we could declare it here and "d" is a pointer here, and power 3, it is called power 3 here, is a function.

It is a pointer to a function. So that is what has been declared here. So once, so that is the structure of this function, and the function can execute this power 3 by invoking that pointer. So these are the things we saw. We saw in general, general structure of a C program, and we saw pointers. We saw variable, both integer and floating point, and we saw character arrays, and we saw how to call a function, and how to invoke a function through a pointer, and how to pass a pointer to a function through another pointer to a function to another function. We also saw two-dimensional arrays, and how to point one-dimensional pointer arrays to two-dimensional arrays, and read out the addresses, etcetera.

So now, in this lecture, we try to go little more on to the way the variables are stored on a computer. We would need this before we actually understand what kind of errors a program can have, and how do these errors propagate, etcetera. So we will look at that now. So here is a picture of a Vehicles Autometer. You might have seen this, everybody. Now what is the speed of, this is a speedometer. Now the question is, what is the speed at which this vehicle is? Now somebody asks you: "What is the speed at which this vehicle is traveling?" Then what would you say? You would say that this is going between 50, 48 and 49. So we can see that it is between 48 and 49 km/h. So we can say with confidence that this is going between 48 and 49.

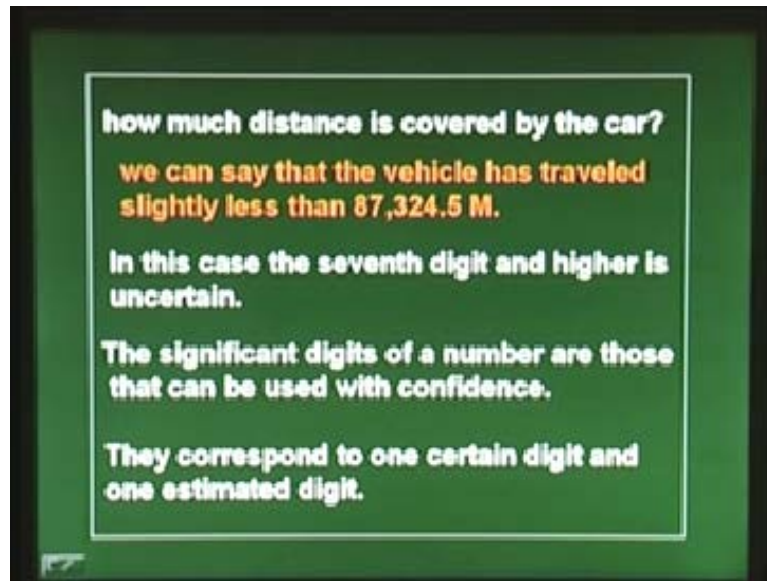
And then we ask for a little more "How much between 48 and 49?" It was, okay. It is between "48.7" and "48.8", but that is a guess. So we make a 1 decimal point guess here that's not marked on this autometer. So with confidence only we can say, we can make 48 with full confidence. But we can make a guess on this, the last digit, the 1 decimal point. So that brings in the point of, what are the numbers of significant digits on this particular instrument? Similarly, we could also ask the question: "What is the number of kilometers traveled by this particular vehicle?" Then you would say that it is 87,324. So, that is 87,324 kilometers.

(Refer Slide Time: 10:14)



And something between 4 and 5. I can say that, 87,324 kilometers it has covered, and some .4, between .4 and .5. So that is again, so we could say that it is slightly less than .5, so 87,324.5 miles or kilometers. So in this case, now we have 1, 2, 3, 4, 5 and 6 digits also we could say. But the 7th digit is uncertain, completely uncertain. We do not know what it is. So now that brings into the concept of significant digits. That is, the number of digits we can say, we can read off with confidence. So they always correspond to one digit less, one digit, and one estimated digit. That is what we said here. So this is kind of an estimate. We saw that it is between 4 and 5. We said it is closer to 5. So we are making an estimate here. That is the number of digits which we have; significant digits. Now that is important. So one other important concept which we would get from this is something called a significant digit. So when you do a computing also we need to know. What are the numbers of significant digits?

(Refer Slide Time: 11:30)

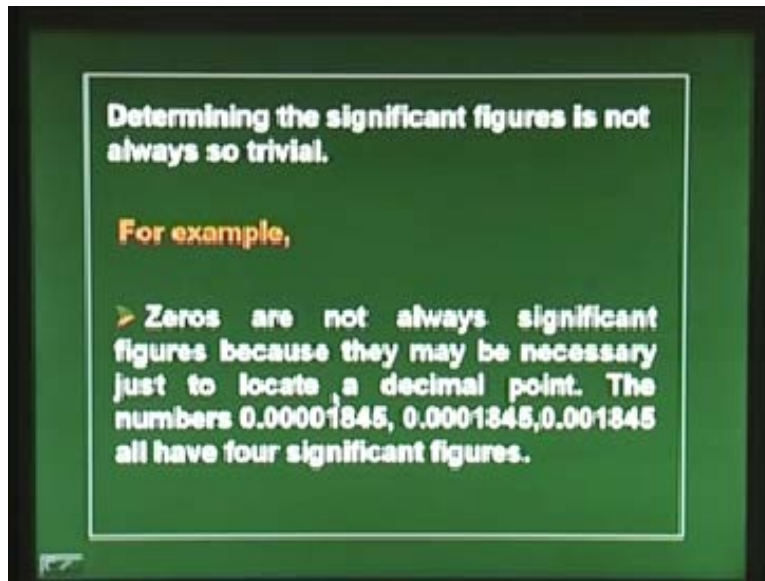


So we will see that in later in this lecture that the number of significant digits depends on what kind of arithmetic operations we are executing, with what kind of numbers. So can we just, is it easy to determine the number of significant digits, it is not always so easy. For example, here, let us look at that.

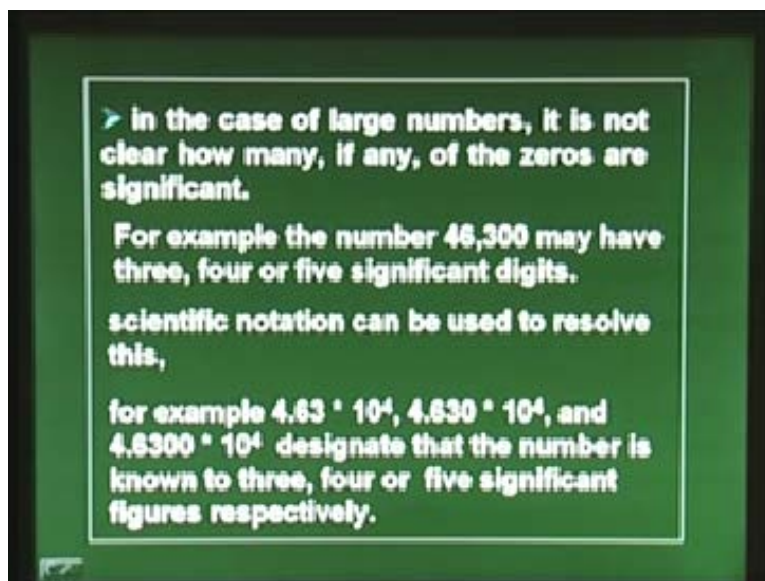
Now, if we are given these numbers: .00001845, or .0001845, and .001845, so now what is the number of significant digits in this? They all have four significant figures. That is significant digits. That is what the answer is. The 0s are basically used to locate the decimal point. So it is not so trivial. That is why I said. So we have a better notation when we use programs. We have a better notation to actually fix this. So that is what I will show in the next one. So here again, for example, you could have 463000, and it could have, you could ask, what are the number of significant digits in this? Is it 2, 3 or is it 5?

So again, in case of a large number, it is not clear how many 0s are significant. We can add any number of 0s, and the questions is, how many 0s are actually significant. That is not always quite clear. The way to overcome this is to use a more scientific notation. That is, we say that we will write 46,300 as “4.63” into 10 to the power of 4. So now, we say I have 3 significant digits in this form. If I write “4.630” into 10 to the power of 4, then I could say that I have 1, 2, 3, 4 significant digits. That is a way of telling the reader that I have 4 significant digits, and I could write “4.6300” into 10 to the power of 4, that will have 1, 2, 3, 4, 5 significant digits. Look at them. All of them represent the same number. It all is 4,6,3,0,0 saying that whether these 0s are significant or not, is represented in this form.

(Refer Slide Time: 12:35)



(Refer Slide Time: 14:14)



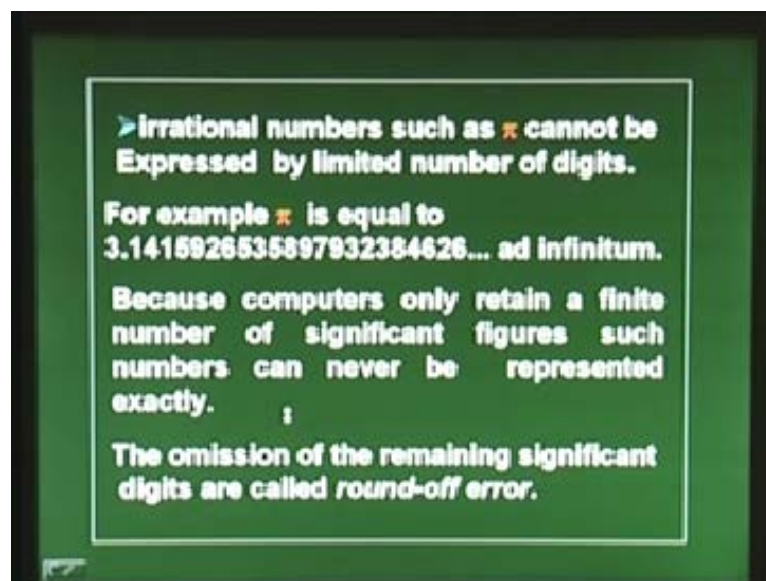
So then there are other problems in representation of numbers. So one is this number of significant digits, and then other thing is there are certain numbers like irrational numbers which cannot be represented as ratio of two integers, so in that case, you have, for example, here I have taken phi as an example. You know that phi is "3.14159, and you can go on till this and this series never ends. So now, the question is, how do I represent such a number on the computer? If it is a rational number, I could actually store it as a ratio of two integers, let us say, in a program.

So, in an irrational number like this, how do I store it? So I cannot. So I have to make approximations to this. So that brings in errors again. So how much, how many digits I can store depends on how many significant digits I would have on my storage. So what is the message here is, that the computer, since it can retain only a finite number

of significant figures, such numbers as this, that is irrational numbers, cannot be represented exactly. So there will always be some error. So those errors are called round-off errors. So we will have to round it off somewhere. So we have to say, so there is nothing called irrational number on the computer. It is always a rational number, because we are always going to chop off somewhere.

So we will round it off, that chopping, or that making this infinite series into a finite series involves, puts in some error, and it is called round-off error. That is something which we will see. So we saw two things now. We have the concept called number of significant digits, and as a consequence of number of significant digits we have what is called the round-off error.

(Refer Slide Time: 16:32)

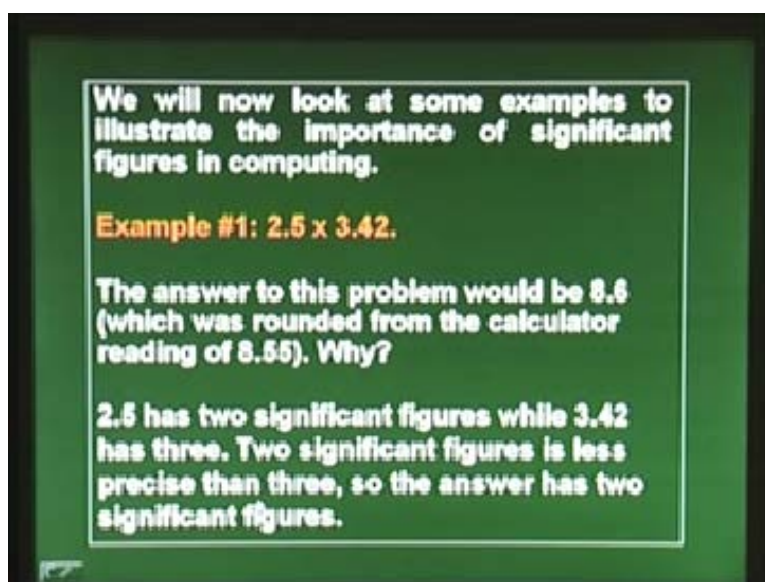


We now go into some examples of this. Let us try it. How do we determine the number of significant digits in computing? Let us say you have you represented some numbers, and you were doing some arithmetic with these numbers, and then the question is, as we go on, does the number of significant, how do we retain the number of significant digits, and how do we compute what is the number of significant digits are, because round-off error, which is creeping in some point might propagate down when you program in your calculations?

So that is what we are going to look at. We have to be careful about it. So now let us look at this case where we have “2.5” multiplied by “3.42”. So now the question is, what are the number of significant digits? So now the answer to “2.5”, “3.42” would be “8.6”. So when we actually multiply it this can be, you know we say that it is 8.6, but the question is, why are we retaining only one digit? I would say the answer is “8.6”. But why are we retaining only 1 significant digit? Why do not you retain “8.6” something, or actually, if you do this, “8.55”? The answer is 2.5 given here is not 2.50, if it is represented as “2.50”. I would have said that it has 3 significant digits. Instead of that, I have only “2.5”. I have printed it to be “2.5”. That means, it has only 2 significant digits. Now I am multiplying that by something which has 3 significant digits. So the answer will have only 2 significant digits. Third digit is not meaningful.

So hence, the answer to this is only there is no point in retaining another digit. Okay we could as well round it off to 2 digits. That is, what we are doing here. So, that means, 2 significant figures are less precise than 3. So the answer has 2 significant figures. We had 2 significant here and 3. So this is less precise than that. So we say only 2 significant figures have to be kept. So I used some new word here called precise.

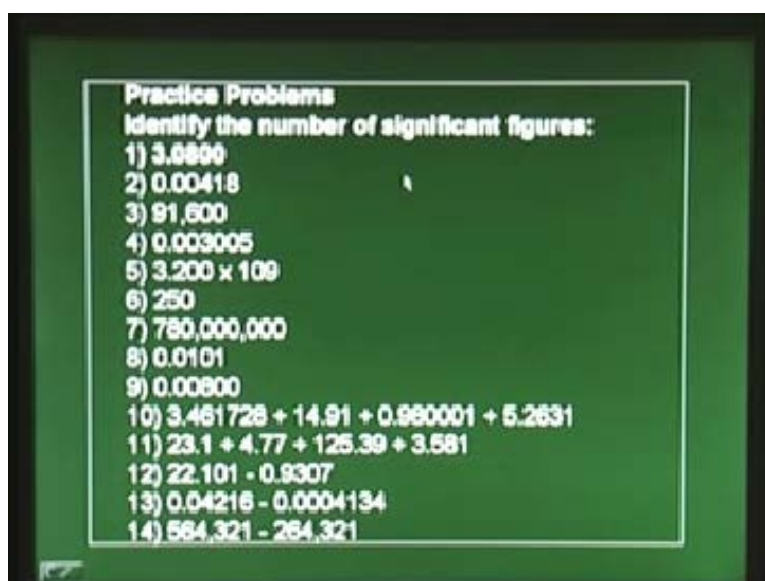
(Refer Slide Time: 19:06)



So we will see what we mean by this in a later slide. We will look at some more examples of this before that. So, here is another one. So we have “2.33” multiplied by “6.085” into “2.1” So now I guess now it must be obvious to you that how many significant figures are here. Okay we have 3 significant figures here, and 1, 2, 3, 4 here. But there are only 2 here. So the answer to this is obviously 2, which now the answer to how many significant figures this particular product will have, answer is 2. So now which number decides that the number with the least significant figure. That is “2.1”.

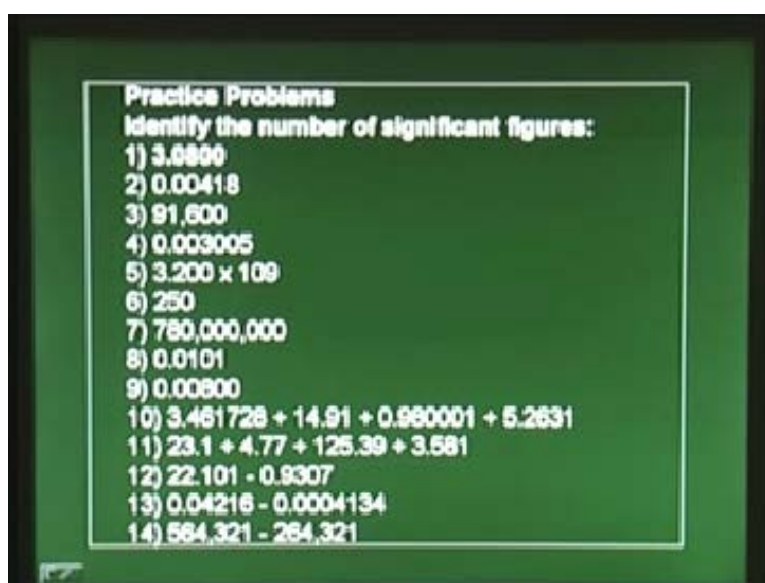
Why is that so? Because that has the least number of significant figures. So this slide makes it very clear, that if I give a computation like this, and you ask a question that how many significant figures, which you say 2. Which number decides it? “2.1”. Why is it so? Because that has the least number of significant figures, and that is the least precise measure. If I had written it as “2.10” here I would say that it has 3 significant figures. The answer to this whole question would be that this computation will give me a number which has 3 significant figures. So I hope that is clear. So here are some practice problems which you could write down and then work on. So just identify the number of significant figures. So here it is given “3.0800” and I have given “.00418”, and then “91600” “.003005”. “3.200” into “109” “250” “780 and followed by 6 0s”, and I have “.0101” “.00800”. So that is the numbers.

(Refer Slide Time: 20:40)



You could think about how many significant figures each of these numbers is trivial. And then I have some arithmetic operations here. So you have 1, 2, 3, 4, 5, 6, 7 digit number added to a 2 significant digit number, and then that is added to a 1, 2, 3, 4, 5, 6, 7 digit number again, and the 1, 2, 3, 4, 5 digit number. So you have some arithmetic operations, some addition, subtraction etcetera. You could work out how many significant figures each of them have just to give you practice.

(Refer Slide Time: 22:15)



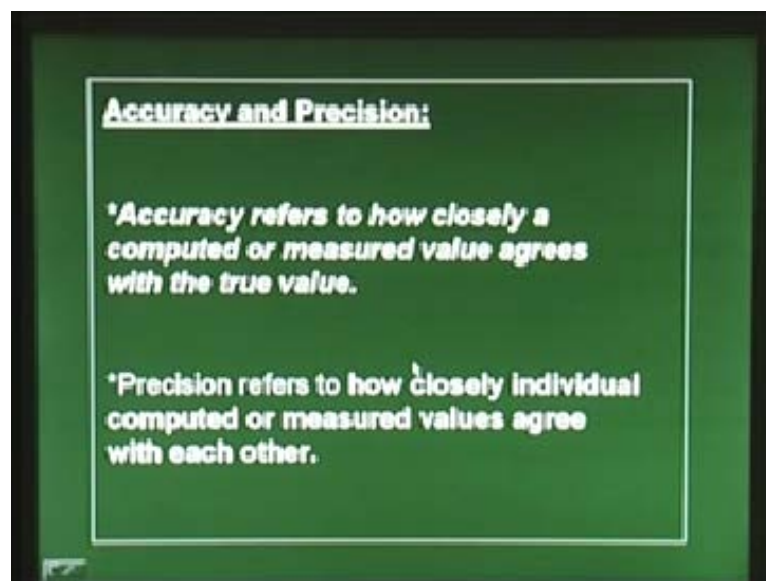
Okay so now we go, what I said in the previous slide, that something called precision and accuracy. So let us see. We often confuse between these two words. When you say accurate and precise what does it mean? What is meant by accuracy? What is meant by precision? Is there a difference? It turns out that there is actually a difference that is accuracy refers to how closely a computed or measured value agrees with the real value. So when we say something is accurate we know what the real

value is, and we say that the measured value agrees, is very close to the accurate value.

So, we have an idea about the accurate value, about the correct value, and the measured value is how close to that. That is what is referred by accuracy. Now what do we mean by precision? So precision tells that how close these measured values are. So you make many measurements of some objects of something. Let us say you measure the length of this line with the measuring scale. We measure the length of this line, and we make many measurements, and get a set a values and how close these values are. If all the values are same, we say very precise measurement, But the measuring instrument we use may not be very good, and you might be getting a wrong value. It may not be accurate, But it is very precise.

So precision just refers to the fact that how closely each of the measured values agrees with each other. So that is all the precision needs, while accuracy actually refers to how closely, how close is the measured or computed value to the real value. So that is the difference.

(Refer Slide Time: 24:14)



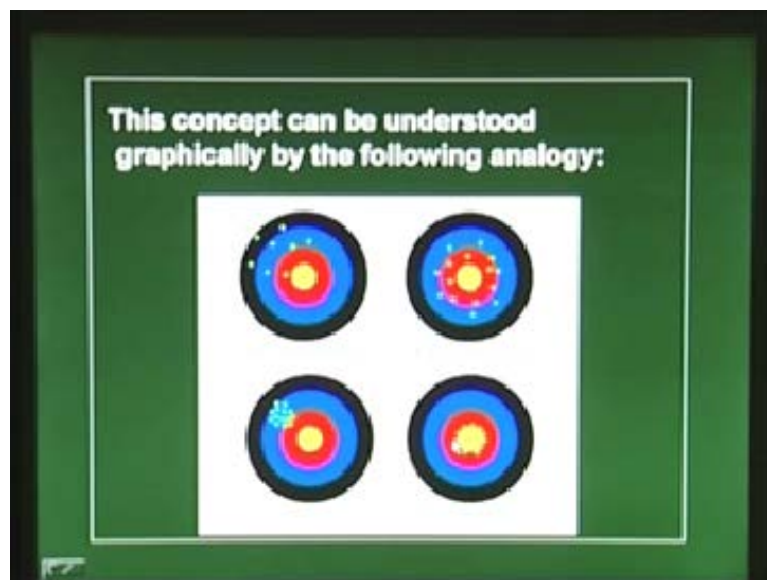
And here I have the slide which kind of demonstrates this. Before that, again the precision signifies the number of significant figures representing the quantity. That is also very important. How close to the real value you are? How many significant figures you have? That is referred by precision. So that is when you talk about that is important in precision, as I said. So here I said how closely individual computed measurements agree with each other right. So, that the number of significant digits are very very important at the spread in computation or measurements of a particular value.

So that is determined by the significant figure. So here is a graphical representation of the same. Here is a dart board or a shooting board. Okay now so look at this. There one person has hit it all over here, and the accurate point should be the center of this. The correct point should be at the center. We know that the correct point should be at

the center, and this person has hit it all over the place. Now we say that, and also this is completely spread out and we would say that this is imprecise. This person does not know how to hit it. So it is imprecise and inaccurate because there is a huge spread, and it is completely off the target, and on the other hand, here you have pretty accurate. It is all over. It is all inside this, mostly, but very imprecise.

Again the spread is large as this. There is a huge spread. So it is inaccurate sorry accurate, but imprecise. You could also look at this case, the 4th case, the 3rd case, in which it is completely off the target, but very precise. The spread is very very minimal. So you probably used a faulty equipment to shoot. So it has got off-target, but very very accurate, very very precise, or you could have accurate and precise. That is this case. Everything is in the center and very very precise. So it is at the center, on the target, and at the same time very close to each other.

(Refer Slide Time: 27:03)

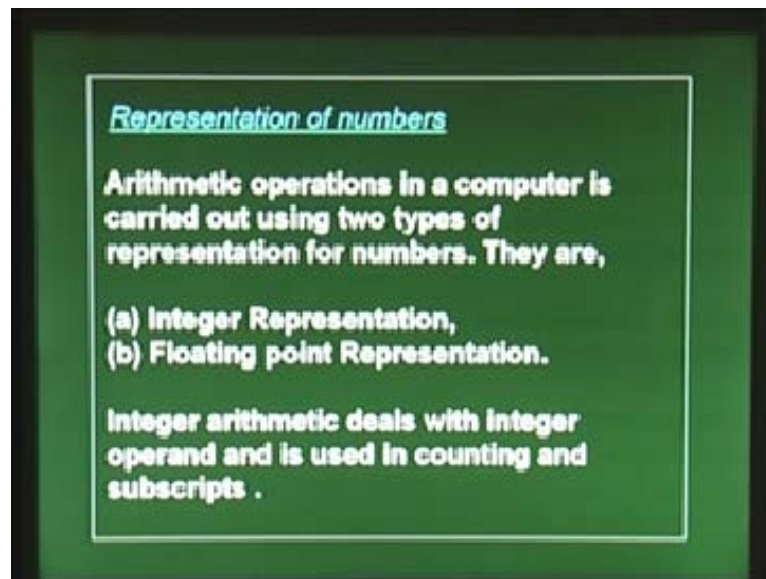


So that is accurate and precise. Having discussed this idea about accuracy and precision we would see how this would come in a program with, because of finite number of significant digits in a floating point number, etcetera. But before that, we need to understand how numbers are actually represented in a, on a machine. We said that arithmetic operations are carried out on a computer using two types of numbers. We said we have integers and floating point representation. We saw that on the program integer and floating point representations. So now, integers we use normally for counting numbers, and for counting subscripts, superscripts, etcetera. So now the question is, how do we represent these numbers, floating point and integer? First we look at the floating point numbers.

So that is where we use fractional numbers. That is when we have numbers which are not represented as an integer. Then we need decimal points, etcetera. Okay so that is how we use floating point numbers. So, basically, numbers with decimal point numbers are called floating point numbers. Real numbers are called floating point numbers. So now these numbers on the computer, they are normally normalized. So

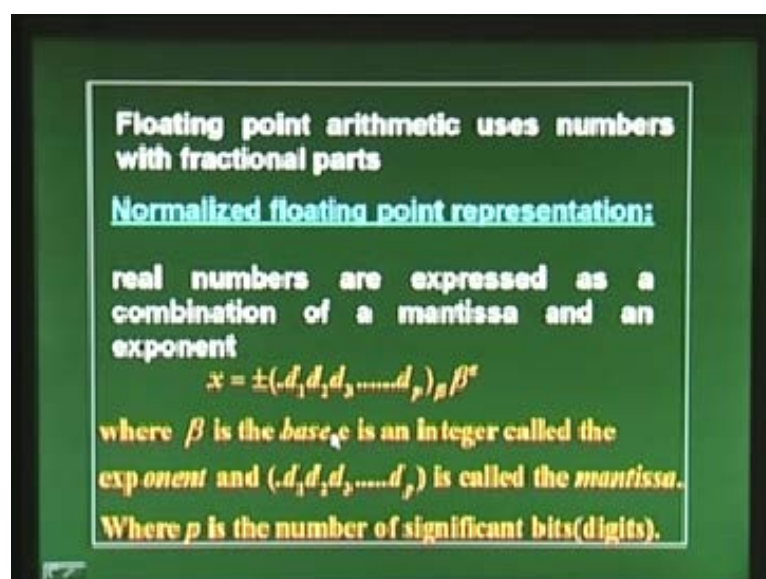
this I will explain to you. What is meant by normalized floating point representation? We will see that.

(Refer Slide Time: 28:11)



So as I said here, the floating point or the real numbers are expressed as the combination of mantissa and exponent. So that is a real number, x , is written as point a decimal point plus some numbers up to some significant digit which is given by p here. So p here is the number of significant digits, and then a base. Base, we are familiar with is 10. On a computer it is always 2. We know that, but we are familiar with 10. We try to look at examples with base 10 to understand this representation, and then there is an exponent, e . So now, β here is the base, and e is an integer. It is called the exponent. So this is an integer here, and then this is the mantissa, and the p is the number of significant digits. So that is the way a real, or floating point number, is represented.

(Refer Slide Time: 30:08)



So let us look at some examples. So here are some examples of these numbers with base 10. As I said, we would use base 10 because we are more familiar with it. So “27.39” is a real number, it is a floating point number. So how is it actually stored on the computer? It is stored, let us say, we have a base 10 computer. Let us imagine.

It will be stored in the normalized floating point representation as “.2739” into 10 to the power of 2. Base is 10, exponent is 2, the mantissa is 2739, and the 1st number after the point is always nonzero.

So that is what I meant by normalized. You do not put a .0 here. You shift all the numbers, such that the 1st number after the decimal point is non-zero. You can adjust your exponent such that it is always true, and then you have a sign. So we need to store the sign, and we need to store all these numbers, and we need to store the exponent. The base is fixed. So when you store a floating point, and then computer stores it. It needs a bit to store the sign of the mantissa, and then it needs to store the mantissa itself, and then it needs to store the exponent, and of course the sign of the exponent. So we will see that. So here the sign of both are plus.

Now to illustrate this normalization, I again show this number here, so it is minus “0.00124”. So that is stored in the normalized representation as minus “.1240”. So this is again showing that if I had to represent this on to 4 significant digits, then I will normalize. I will shift this whole thing here and make it 10 to power, and put an exponent 10 power of minus 2. Minus 2 is my exponent base 10. If I had actually 5 significant digits in the mantissa then, I would have to add 1 more 0 here to keep that same. So the fractions found here are normalized, and the first fraction digit is always nonzero. So now, you think about this. What does it mean if it is on a binary machine?

That is, if its base is 2, that would mean that this number after the decimal point is always 1 right. That is what it means.

(Refer Slide Time: 33:10)

Examples of numbers as Represented in a computer:

Some representations in base 10 are:

$$27.39 \rightarrow +.2739 * 10^2$$
$$-0.00124 \rightarrow -.1240 * 10^{-2}$$

The fractions found here are normalized- the first fraction digit is nonzero

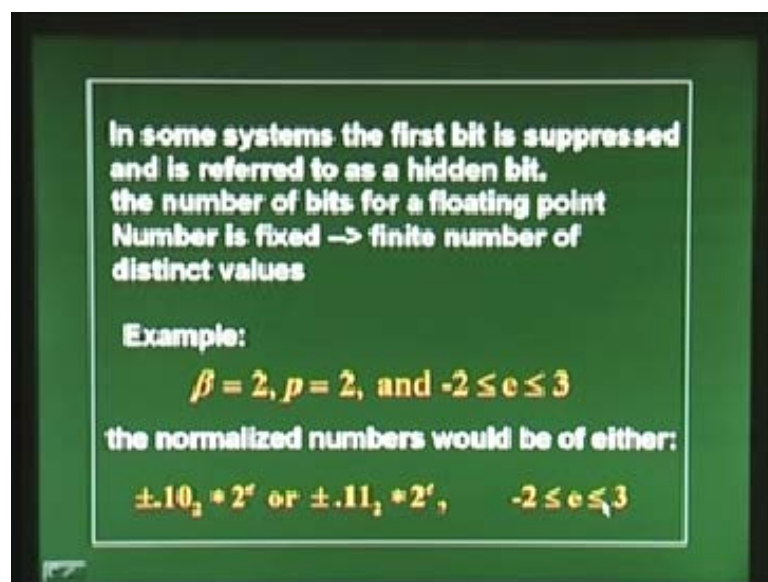
If the base is two that means that the first fraction bit is always 1.

It can be either 0 or 1, and because it is a binary and this would mean that in the normalized representation, this would mean that this bit here is always 1 because it is normalized it cannot be 0. It has to be 1. So it is always 1. In that case, we do not need to store it because we know it is, we have been told in the compiler, or the program that it is a normalized floating point representation,. So, that means the first digit after the decimal point, the first digit of a mantissa, is always 1 in a binary. So we do not need to store it. So that bit can be used for something else.

So that is what the interesting aspect is. So that is always possible, and so because the first bit of the mantissa is always 1, we do not store it. In some of the computers, in most of the machines, this bit is suppressed. As I said, we can use that for something else, that memory space. So number of bits for a floating point is always fixed, as I said, and that also implies as we have seen earlier, that this is a finite number of distinct values. Okay so now we look at this by an example.

So here is the case where the base is 2, and there are 2 significant digits and You have an exponent, which can go from minus 2 to plus 3. Let us see it. So what does it mean? What are the numbers? Suppose I have a system which is like this, which is base 2, 2 significant digits, and an exponent which goes from minus 2 to plus 3. So what are the numbers which I can store in a floating point representation in a normalized floating point representation? That is, what we are going to look at. So what can the numbers be? Then numbers can be plus or minus “.10” base 2. This is the representation, base 2. 2 power exponent e. Only 2 significant digits and this is always 1 right. So it can be 1, 0 or 1, 1, and the exponent can go between minus 2 and plus 3.

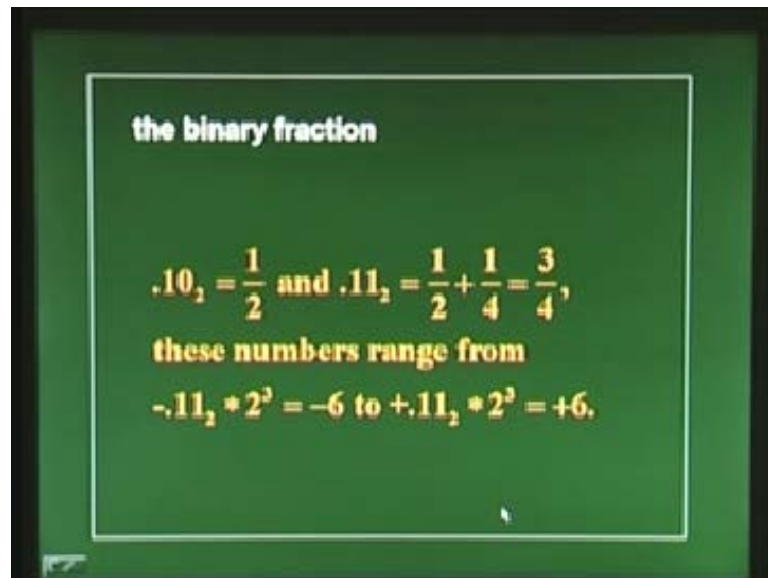
(Refer Slide Time: 35:38)



So what does it transfer to? If it is 1 0, what does it mean? 1 0 is 1 by 2. This is 2 to the power of minus 1. So that is 1 by 2. Now, if it is 1, 1 and 2, then it is 2 to the power of minus 1, 2 to the power of minus 2. That is 1 by 2 plus 1 by 4. So you have 3 by 4. So that is the way it is represented. So, now that means this numbers can now range from minus 6. Let us say 1 0 is one possibility. So what is the smallest number which it could have? You can think about is the range of, we just saw the range of the

exponent is minus 2 to plus 3. The smallest number you will be able to represent would be 1 by 2 into 10 to the power of minus 2, and the maximum number it could go the smallest positive number that is, what I mean, the maximum number it could go, would be 3 by 4 into 10 to the power of 3. That is 6. So the smallest positive number would be 1 by 2 into 2 to the power of minus 2, and the maximum would be 3 by 4 into 2 to the power of 3. That is 6. So the total number it can go the full range from minus to positive, negative to positive, would be minus 6 to plus 6.

(Refer Slide Time: 37:18)



But note, that this also does not mean that I can go from minus 6 to plus 6 in all values, because there are only a finite number of significant digits here. So that could mean that I have not only a limit on the values which I can store, I also have a limit on the number of values I can store in that range. I not only have the limit on the range, but I also have a limit on the number of values I can store. So there is a finite number of values in a given range which I can store. So this limits the accuracy of computation often.

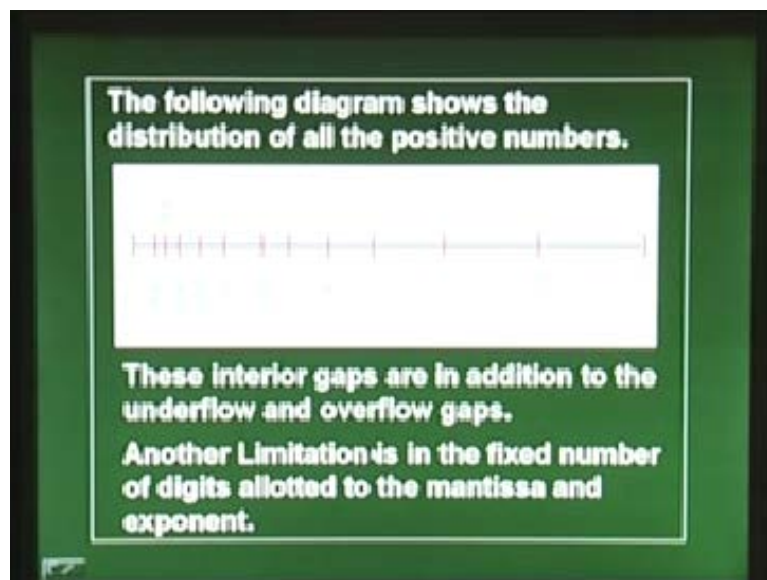
Even though present day computers store a large, mantissa is very large, so this could be very close, but still it is not continuous. It is still a discrete set of points. That is something which we should remember. Now this diagram is basically showing that, illustrating this aspect of it. That is, it is going from, I have only shown positive numbers here. So going from some value here that is close, that is 1 by 2 into 10 to the power of minus 2, we said. So 1 by 8 is the smallest it can do, and then it can go all the way to plus 6, but in a finite spacing.

So now we can also see, we should note that the gap between numbers increases as the number increases. That is also an important thing to remember because that, actually keeps the number of significant digits same. Okay so now, when the number goes, if this is your representation, and if you try to put in a number in this system, if you try to compute something which is more than 6, let us say you have a 2 significant digit, floating pint normalized system, and you try to do something more than 6 into it, so the maximum number that it can represent is 6, as I said. So if you

represent something more than 6 it will give an error saying there is an overflow. Now if you try to put something less than 1 by 8, then it will say there is an underflow. Apart from these overflow and underflow errors which we know, there are also errors, there are also possibility of errors coming because of finite number of representations.

Okay this is coming because there is finite number of digits bytes allotted for the mantissa and exponent for any system. We will see how much that is later. Depends on the machine you need to allot certain number of bits for the mantissa and certain number of bits for the exponent, and then you have to allot one bit for the sign of the mantissa, and in some cases, one bit for the sign of the exponent, but normally that is not given. We will discuss that in a later lecture, the actual implementation. But at present, we understand that there is a finite number. That is what more important.

(Refer Slide Time: 40:53)

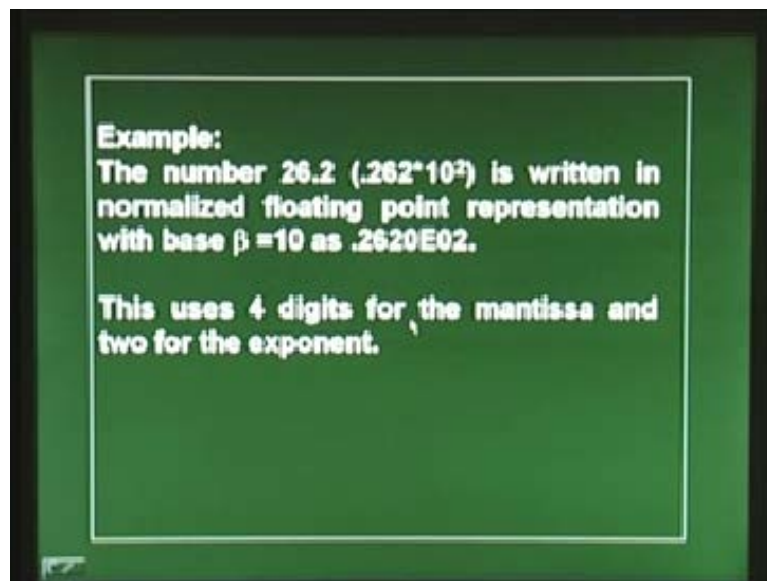


So some examples again, two point, “26.2” is in base 10 would be “.262” into 10 power 2, and so you would write this, you would represent this on the computer as “.2620” into e 0 2. If, there are 4 digits for the mantissa then, we would write it like this, and 2 digits for the exponent. Let us say we have a system in which we have 4 digits for the mantissa and 2 digits for the exponent, and this number will then be represented like this. Exponent is 02 and 2620 is the mantissa. In this case, if you do some arithmetic with this number, and if this number goes, exponent goes above 99, then you will not be able to store this because it has only 2 digits. Then it would give an error, and it would say there is an overflow.

And similarly, if this exponent goes below minus 99, then again it would give an error like that. That is underflow. There are two ways of, now let us look at the way of translating a given real number into a n digit floating point number. We will represent this as f_1 of x. You have given a number, and you have given a fixed number of digits, and you have said that you convert this fixed number of digits floating point operation like we saw phi. You have been given phi, and you have said that you have a 6 digit floating point number. You convert that How do you do this? So It is obvious

that you cannot represent phi accurately or completely in a finite number of digits, so you have to do something to this, and that is as I said, you either round it off or you chop it. There is a slight distinction between rounding and chopping. Okay that is what we would see now. For example, let us take 2 by 3. So that is a rational number. 2 by 3 is given to you and you are asked to do it in a 6 digit floating point arithmetic with base 10. How do you represent that? So you have 6 digits total, and you said I took 4 digits for the mantissa and 2 digits for the exponent. Okay so I have 6 digits in total. And then how do I represent it? I could write it as 6667 okay when rounded. Because I know it is 6666 it keeps going like that.

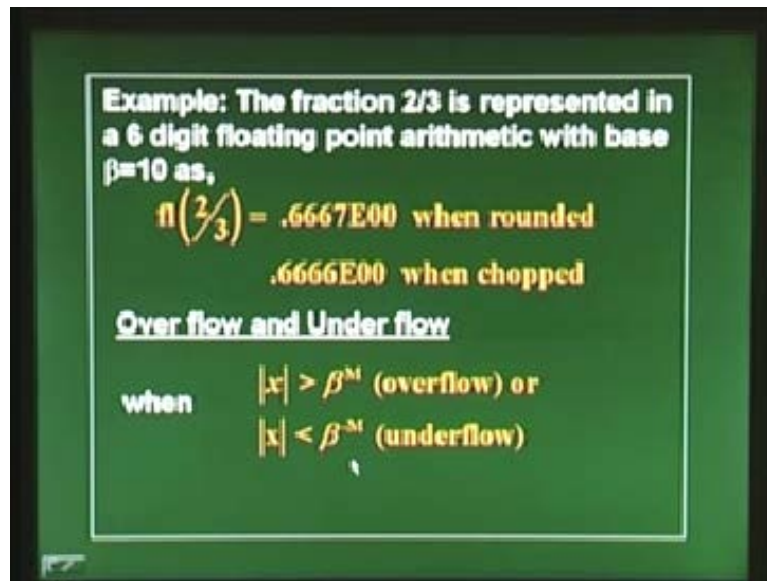
(Refer Slide Time: 42:04)



Okay so I could say that I have round it off after the 4th digit. 5th digit was more than 5. So I round it off to the next integer here and make it as 6667. That is rounding, or I could just chop it off. So you could see that they give slightly different errors depending upon whether you round it or chopped it.

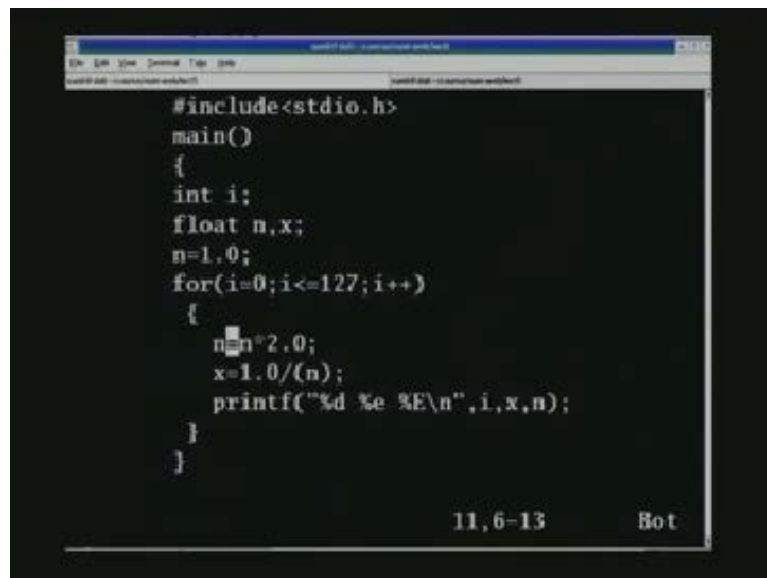
So as I said, to summarize this, as I said there is finiteness of the representation, and because there are again, there are gaps. In this case, there is a gap. So there is a finiteness of the representation and finite number of floating points allowed within a given gap, and then you have overflow and underflow, because the exponent is limited. We could see this in a program here.

(Refer Slide Time: 44:37)



So I have this program here, which is, you have a floating point, n, which is given, as it just starts as n equal to 1. I run it through a loop and then I multiply it by 2. Everytime, every loop, and then I am just printing out here using this percentage e, this kind of format okay.

(Refer Slide Time: 45:06)



I am printing out both the x values. That is, 1 by of n multiplied by this 2 every time. It keeps increasing by 2. It just doubles every time, every loop, and it just prints this out. Let us see, what we see, what we get. So you can see that this is the 1 by n which is doubling. So it is half. It goes to half every time-step, in every run in the loop, and this is the one which is doubling every time-step, for every value of i. As I increases here, this doubles, and this goes half. I am just trying to represent that using this floating point format.

In this format, **this is the way it is printed out**. This is not the way it is stored. This is the way it is printed out here and this is in base 10. This is printed out in base 10 and then you can see that it was going half every time-step. Every value of I, it was going down by half, and then it reaches “5.8” into 10 power minus 39 here, and it is “1.701412” into 10 power 38, and the next answer we says it is infinite here, and it says it is 0 here. Okay that is that means, it cannot represent a number which is below this value here and a number which is above this value here. So for example, it cannot do “3.1402824” 10 to the power of 38. It cannot do that. So that is that shows the finiteness of the numbers it can represent.

Okay so this m and minus m, at which it will show here, for example, if I go by this example, m and minus are the lower and upper bounds on the exponent. I have taken this the same here. It may not be the same because the way it is represented on a machine it may not be the same. So here it is minus 39 and plus 38. After that it is not able to do it. So that is the example of, there is one more example here. It is a 6 digit floating point okay operation. Here it is 1111E51 multiplied by 444E50. Now, if I multiply this again base 10, I am representing here, and multiply this, I get something “.04”, etcetera. So even if I normalize this, I get “.4937284” into 10 to the power of 100. So E100, and then it would represent, it would give that as an overflow. Similarly, this will be an underflow.

So here are some problems. You could find, try to find, the largest and smallest numbers that can be represented. If you have base 10 and 4 significant digits on the mantissa, and you have an exponent which could go from 4 to 5, this is minus 4 to 5, minus 4 to 5, minus 3 to 4, minus 2 to 3, and this is 16 base 16 and 4 again, 4 significant digits on the mantissa and exponent minus 3 to 4. Base 2, 8 significant digits. Note the mantissa and exponent minus 2 to 3. So if I try to find out what is the smallest and largest number which you can actually represent on this?

(Refer Slide Time: 49:03)

Example: For 6 digit floating point
 $.1111E51 \times .4444E50 = .04937284E101$
 Answer Overflow
 $.1234E-49 \times .1111E-54 = .1370E104$
 Answer Underflow

Problem:
 1. Find the largest and smallest numbers that can be represented by using:

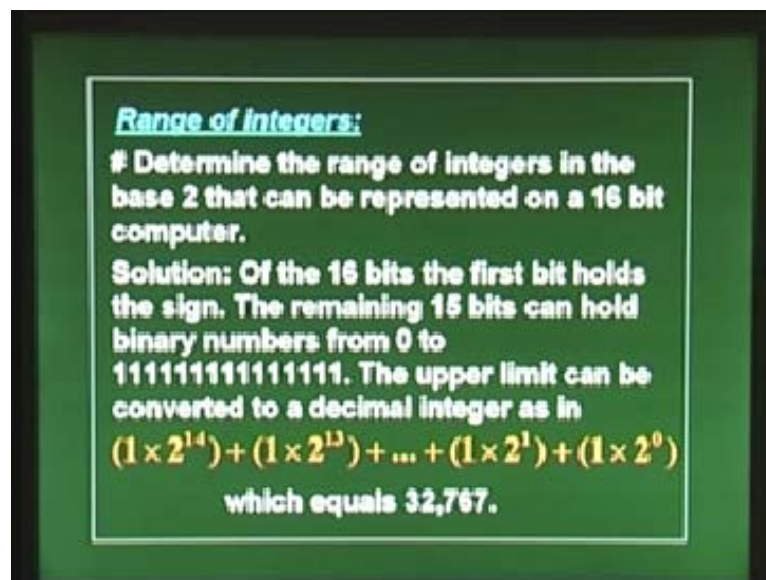
- (a) $\beta = 10, P = 4 \quad 4 \leq e \leq 5$
- (b) $\beta = 16, P = 4 \quad 3 \leq e \leq 4$
- (c) $\beta = 2, P = 8 \quad 2 \leq e \leq 3$

So, before we actually close today’s lecture, we just look at integers. We just looked at floating points so far. Let us look at integers. Okay so now even in there is range at

which for integers too. We just saw range for the floating point. Similarly, there is a range for integers too. Again there is a finite number which you can store. So, that here is a kind of a problem. That is, we try to determine the range of integers in the base 10 again. That can be represented on a 16 bit computer. Okay so what is that? a 16 bit computer. Of the 16bits, the first bit has to hold the sign.

The remaining 15 bits can hold binary numbers from 0 to all 0s to, you can put all 1s. So you can find out the minimum number is, everything is 0. Minimum means positive positive number. Everything is 0. That is one possibility, or everything is 1. So if everything is 1, you could have 1 into 2 to the power of 14 to 1 into 2 to the power of 0 summed over. So that is the number which you can store. That is 32,767, and then you have 1 bit for the sign. That could be plus or minus. We say it could go from minus 32,767 to plus 32,767.

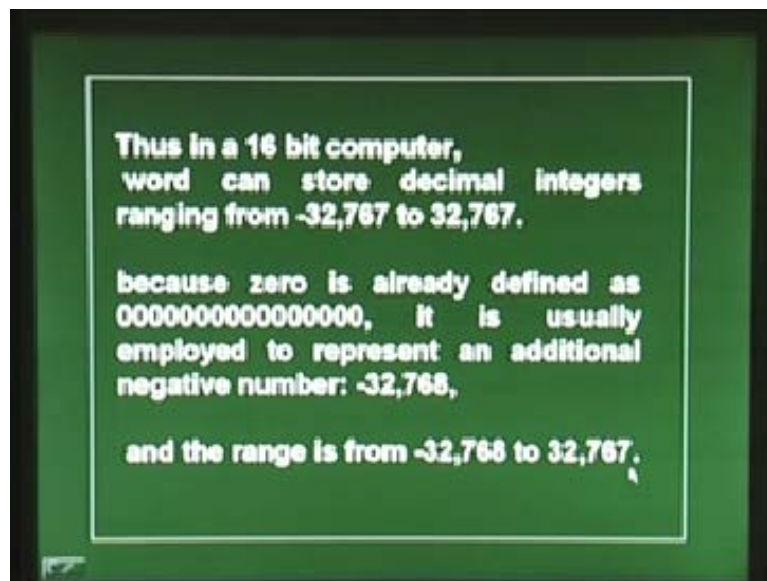
(Refer Slide Time: 50:54)



So that is what we would think. But if you ask someone, that is what they can do, but there is also everything is 0 and that is true. You do not need to store that. Everything is 0. You know is 0. You do not need to store that. So I say that I can use that to store one more number. So I store minus 32,768 on that number, so my range now has become minus 32,768 to plus 32,767. That is the range in which I can store these numbers.

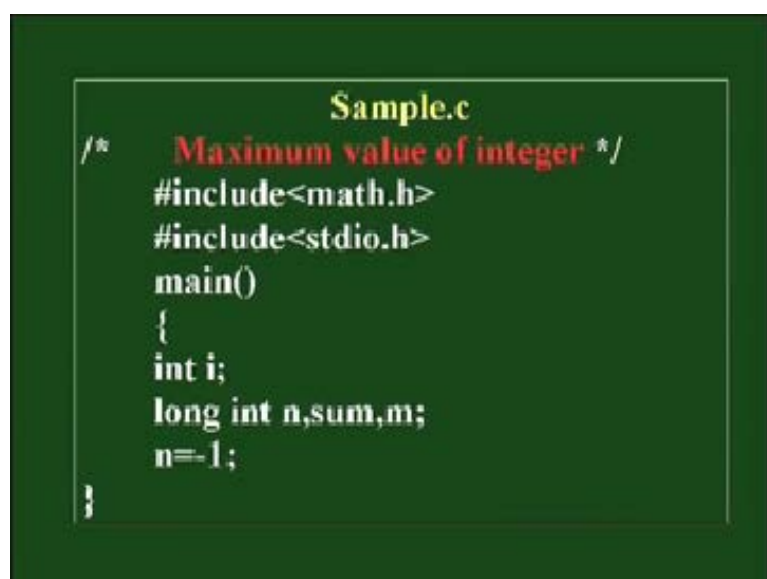
We can see this in a small program here. Here, I am trying to print out the maximum value of the integer which this machine, a 32 bit machine can store. So that is I will do that here. I will just again double the number. You can see how much it can store. I will double the number and I print out both n and minus n values. I start with n equal to minus 1 and I double it, and I print out both the positive and negative numbers. I will also show you that I can print out. I will also, can do another thing. I will just say, start from m equal to 1 and I run this from 1 to 30, and then I will sum up all of these numbers. So, that is actually doing that sum which I showed you here right. I am just going to do the sum and then print out the number, and then you see what we can see in this case. So what do we see? That we can actually print out these numbers.

(Refer Slide Time: 51:34)



So we see that the number keeps increasing and the number keeps increasing after it reached some number here, that is “2147483648”, and then the next number it shows is 0. It cannot represent anything more than that. Then it just puts in a 0, and when I actually sum it up, I can see that this is the sum which I showed you there, and the sum, the positive value, can go up to 7 and the negative value can go up to 8. So that is “2147483648” is possible on the negative number, but I cannot do that for the positive number. That is, what you see here also. This number, if it was double of this one more than “2147483647”, we went to 48 then it lost its sign. It becomes minus. It cannot represent that number, but while it has no problem with the negative number that is because I use this 000.

(Refer Slide Time: 54:06)



(Refer Slide Time: 54:31)

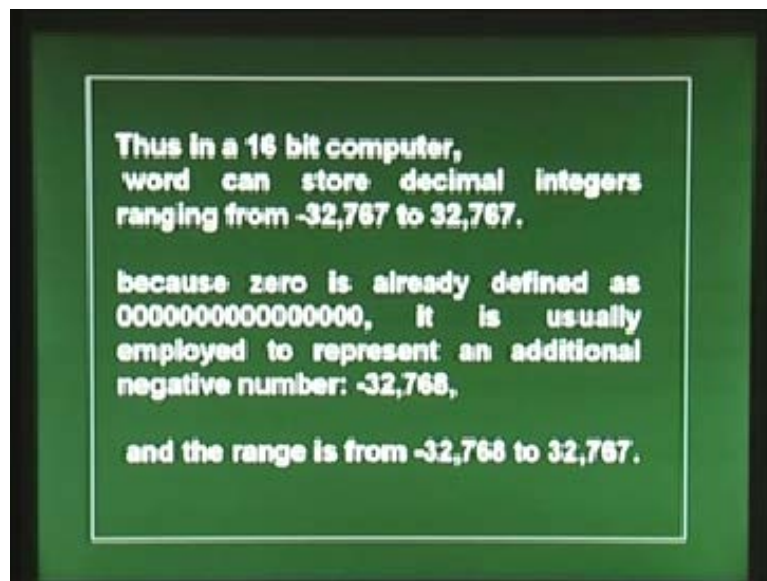
```
for(i=1;i<=33;i++)
{
    n=n*2;
    printf("%d %d %d\n",i,n,-n);
}
m=1; sum=1;
for(i=1;i<=30;i++)
{
    m=m*2;
    sum=sum+m;
}
```

(Refer Slide Time: 54:52)

```
printf("%d %d \n", sum,sum+1);
m=-1;
sum=-1;
for(i=1;i<=30;i++)
{
    m=m*2;
    sum=sum+m;
}
printf("%d %d \n", sum,sum-1);
/* ALL ZERO IS USED TO STORE THE
EXTRA NEGATIVE NUMBER */
}
```

So that is a 16 bit number, 16 bit computer here I showed here and I showed you 32 bit computer here. This is a 32 bit computer, can store from “minus 2147483648” to “plus 2147483647”. That is what it can store. While a 16 bit computer would be able to store from minus 32,768 to minus 32,767, this is the range of integers.

(Refer Slide Time: 55:49)



So we discussed range of integers, and the range of floating point numbers, and saying that the floating point numbers are not represented as a continuum. There are gaps within the numbers, and there are errors which can appear because of this. So we stop here now, and will go into the concept of what is called a word hour of floating point representation in the next class.