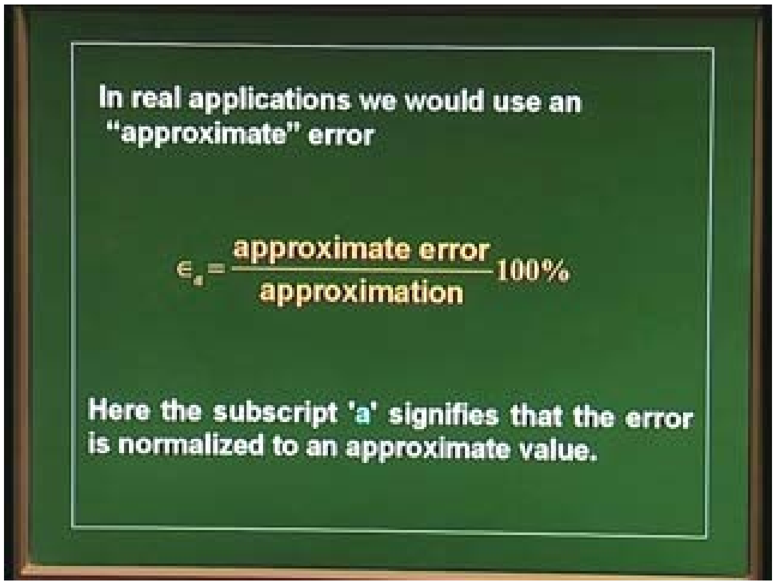
Numerical Methods and Programming

P. B. Sunil Kumar
Department of Physics
Indian Institute of Technology, Madras
Lecture - 7
Error propagation and stability

Last class we discussed about the representation of numbers and some of the errors that can come because of this finite representation of errors. We continue on that today and see some concrete examples of these errors which come and we said that the errors are two different types. One is, mainly two different types, one is the round-off error, and other is the truncation error. Now these errors we can quantify and we can write the error as the value obtained minus the true value, divided by true value, and we call that as the true error, multiplied by 100 would be the percentage error.

We saw that in real applications we would actually know what the real value is, and thus, we need another definition of the error, and that is what is shown here. So we said an approximate error. So the approximate error would be the way it is defined, is epsilon "a" to show that it is actually an approximate error, that should be the percentage error would be approximate error divided by approximation into 100. So what will be approximate error? So that would be, for example, in a series, if you make two different approximation you could take the differences between these two, and then take the, divided by the present approximation, and that would give us an approximate error.

(Refer Slide Time: 02:59)



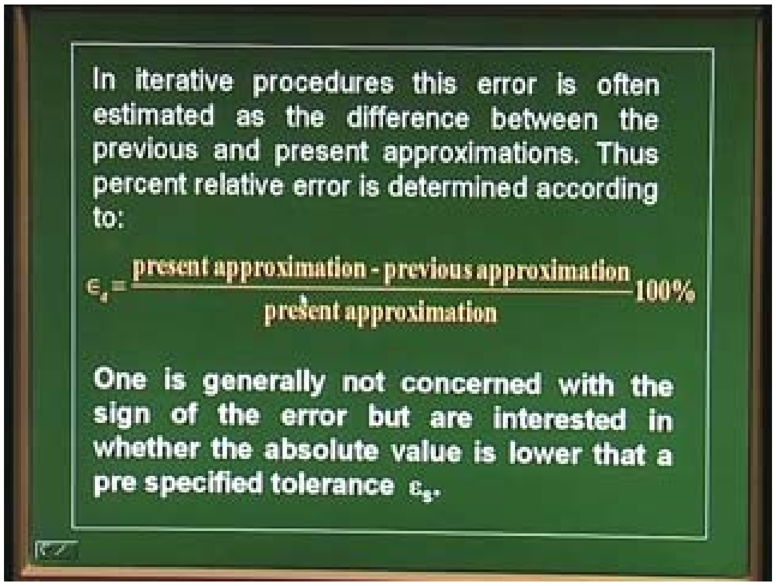
In real applications we would use an "approximate" error

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

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

So we would see this in an example here, just to remind you that "a" is used to show what that this is an approximate error. One place where we actually use this approximate error is in iterative procedures. So that is, as I said, the approximate error then would be the difference between the present and previous approximations, and then you could divide that by the present approximation to get the approximate error. So one could ask the question whether I should take the previous approximation minus present approximation, or the present approximation minus the previous approximation, or does the sign matter. We are not generally concerned about the sign. We are only concerned about the magnitude, so we could write either way and define that as approximate error.

(Refer Slide Time: 04:00)



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

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

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.

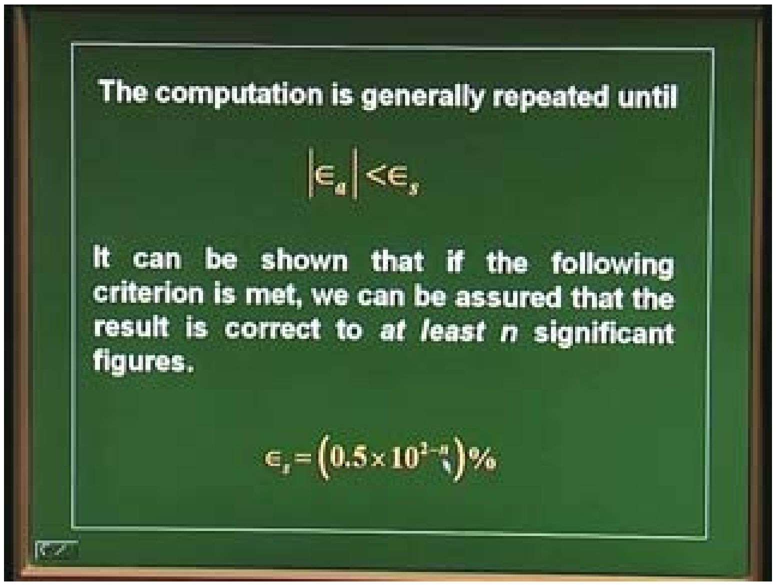
So when you are doing a series expansion, one would be interested in making this approximate error less than the machine tolerance, and we saw what this machine tolerance would be, and we said that if you want to have a precision of n digits, and the tolerance which we can calculate is “.5” into 10 to the power of 2 minus n percentage,
So, that would be the, this means we have n significant figures. So we can compute an approximate error in a series expansion, and then we could demand that the series, you keep the terms all the way up to a term such that the sum is less than, the error is approximate error, is less than the tolerance, where the tolerance we could fix to be “.5” into the power of 2 minus n, the word, n, is precision of the machine, or significance digits which we are interested in.

Due to some reason, we may not be interested in going all the way up to the machine tolerance like some other part of the computation which is, which does not have that precision. For example, when we take experimental data, and then when we want to do some analysis with this data, most of the time the measurement itself is not precise beyond certain number of digits, so there is no point in keeping very high precision which would only introduce additional errors.

So we could just cut them off, and we keep certain number of figures. So we will look at an example of this iterative method just to drive home the point that what we mean by this approximate error and. So let us look at this case of where we try to compute e to the power of “0.5”. So we will use the series expansion to actually compute e to the power of

“.05”. We know the true value in this particular case, and that happens to be “1.648721271”. I happen to know this value, so what would be the series expansion we would use. So you would expand e to the power of “0.5” e to the power of x in general, as 1 plus x plus x square by 2, etcetera, and let us say, to take an example, we want this to be kept up to 3 decimal, 3 significant digits.

(Refer Slide Time: 05:09)



The computation is generally repeated until

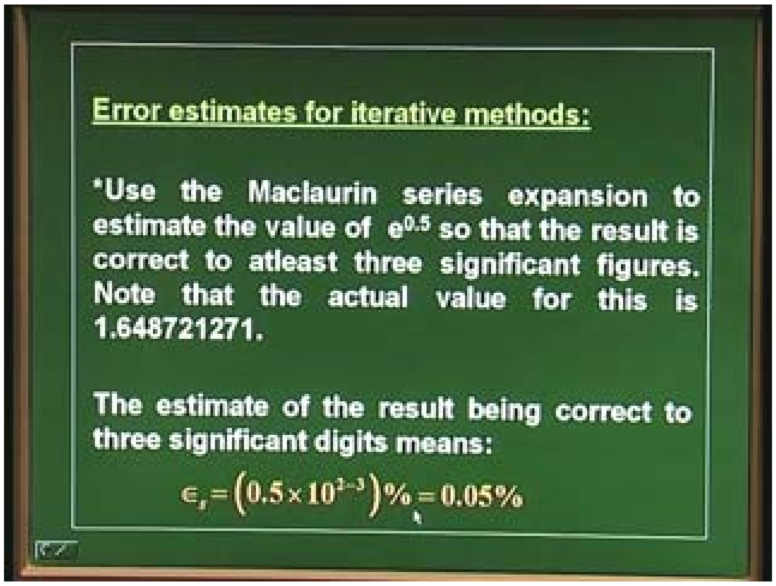
$$|\epsilon_a| < \epsilon_s$$

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.

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

That is why we want to do that. So then, what will we find? So we would want to have a tolerance of “0.5” into 10 to the power of 2 minus 3. I said 2 minus n. Here, n is 3 so 2 minus 3. That is about “0.5 percentage”.

(Refer Slide Time: 07:10)



Error estimates for iterative methods:

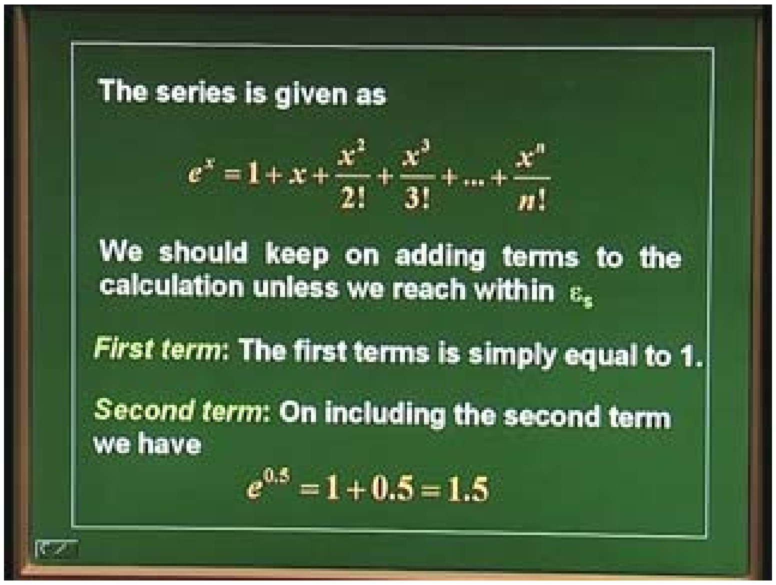
*Use the Maclaurin series expansion to estimate the value of $e^{0.5}$ so that the result is correct to atleast three significant figures. Note that the actual value for this is 1.648721271.

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

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

So let us look at that series. The series is e to the power of x, as I said, 1 plus x plus x square by 2 factorial, plus x cube by 3 factorial, up to x to the power of n by n factorial. The question is, up to what n I should go. That would depend on, so that would depend on what accuracy I wanted. I am going to do that. I am going to compute two different errors here. The true error and the approximate error, and then we could demand that it should go beyond, below the tolerance which we wanted. So let us start with that calculation. The first term is simply 1, and the second would be 1 plus 0.5 for e to the power of .5, that is “1.5”.

(Refer Slide Time: 08:01)



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.

Second term: On including the second term we have

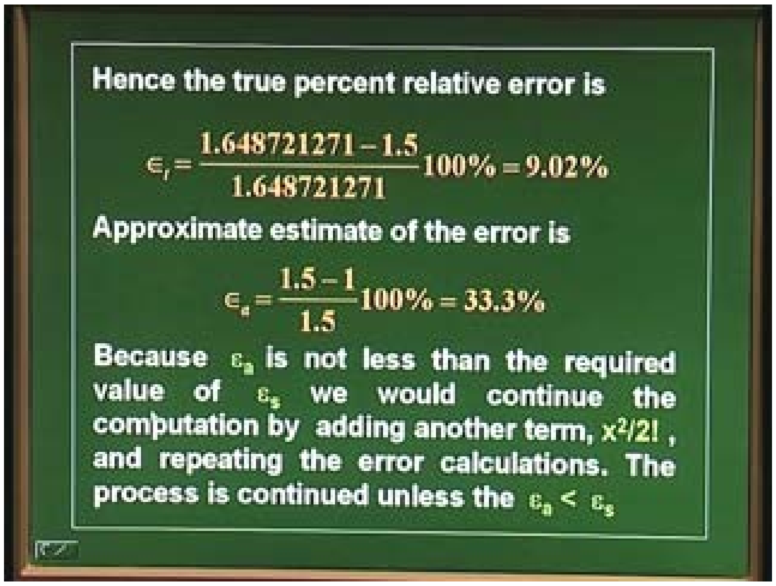
$$e^{0.5} = 1 + 0.5 = 1.5$$

So now, what will be the errors which we have? The true error would be “1.648721271”, that is the true value, minus the approximate value, divided by the true value, that is 9 percentage, 9.02 percentage error. So then, we could take the approximate error. The approximate error is “1.5”, was the present approximation. The previous approximation was 1 and we divide it by the present approximation, so that gives us “33.3 percent.

So we go to the next term. We are going to keep the x square by 2 factorial term, and then we could increase, we could decrease this error, so now the point is, we continue this series, so that is how we compute this. So we compute the next term we would keep, that is x square by 2. So that will give us “.25” divided by 2, so “.125”, and then we add that to this, and we again subtract “1.5” from this, so we get, add when we add, “.125”, we get “1.625.” We will subtract “1.5” and divide by “1.625”, that will definitely give much smaller error than this, so we could compute it in this way.

And then go up to the series where it goes below the tolerance which we have specified. We will see later that this, it is not guaranteed that when you go higher and higher in the series then you will go to get a higher precision, a better value. There will be other reasons why there will be a limiting value to a machine. We will see that later.

(Refer Slide Time: 09:02)



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

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

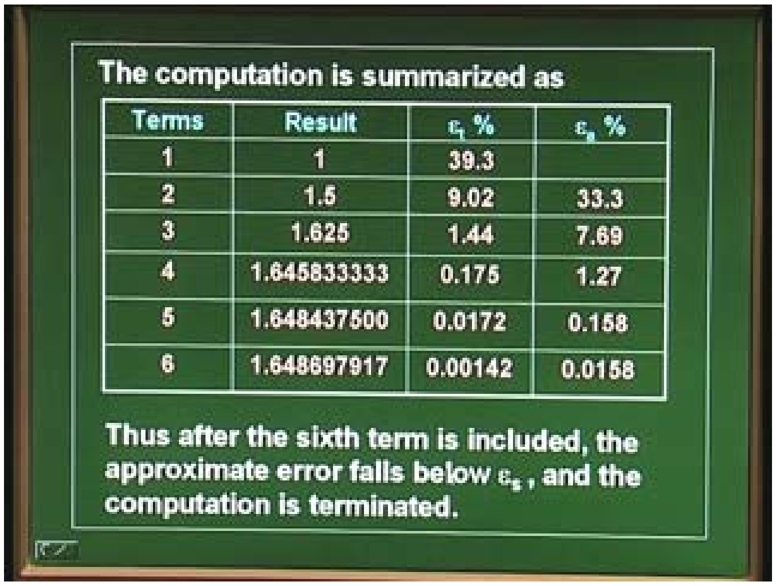
Because ε_a is not less than the required value of ε_s we would continue the computation by adding another term, $x^2/2!$, and repeating the error calculations. The process is continued unless the $\varepsilon_a < \varepsilon_s$

So we summarize this computation which we did just now, that we have said, first was 1. It got a true error of "39.3 percent" if you just take 1 sorry, that is "1.625", "1.648721271" was the correct answer. So we can compute the first approximation just as 1, and there is a true error which is very large, and we do not have an approximate error because this is the approximation. The second approximation was "1.5", and then we have a true error of 9 percent, and then we have an approximate error of "33.3 percent".

So, in this case we can compute the approximate error because we have a second approximation. We have the first approximation and the second approximation. So we could do that, and the third approximation would be adding the x square by 2 term and that will give "1.625", and then we would compute the approximate error by now taking the difference between these 2 divided by this number.

So that is how it gives us this, and when we go to the fourth term again we take the difference between these two numbers and divide it by this number. So that will give us the next error, the estimate of the next percentage, approximate error, etcetera. That is how we compute? So then, we would say that, okay, I was demanding ".05 percent" as my error. Now, by the time I reach the sixth term, I have gone below ".50 percent. So I can stop my computation here. That is an example of how we can use the error estimate to compute something. So let us just summarize this once again. There are two types of errors encountered in the numerical calculations.

(Refer Slide Time: 11:44)



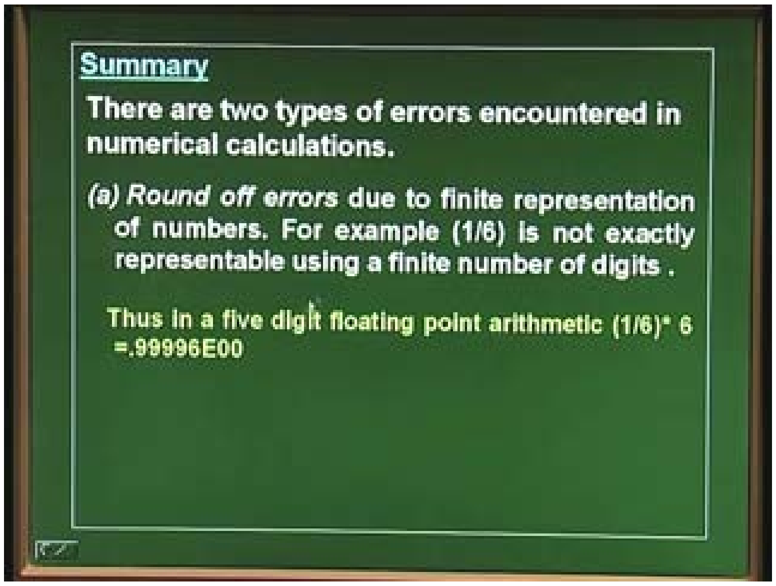
The computation is summarized as

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.

We said one is round-off error due to finite representation of numbers. So, for example, 1 by 6 cannot be represented exactly. We have to have a finite representation to this. Now, there is an error due to this which can come, because when I take 1 by 6, and if I approximate 1 by 6 by finite number of digits, and then multiply it again by 6, I would expect to get 1, but I would get something different from 1. So that is the answer. So, we will see how we can write a program to actually demonstrate these errors. So let us just look at that. It will also help us to get familiar with small programing

(Refer Slide Time: 12:53)



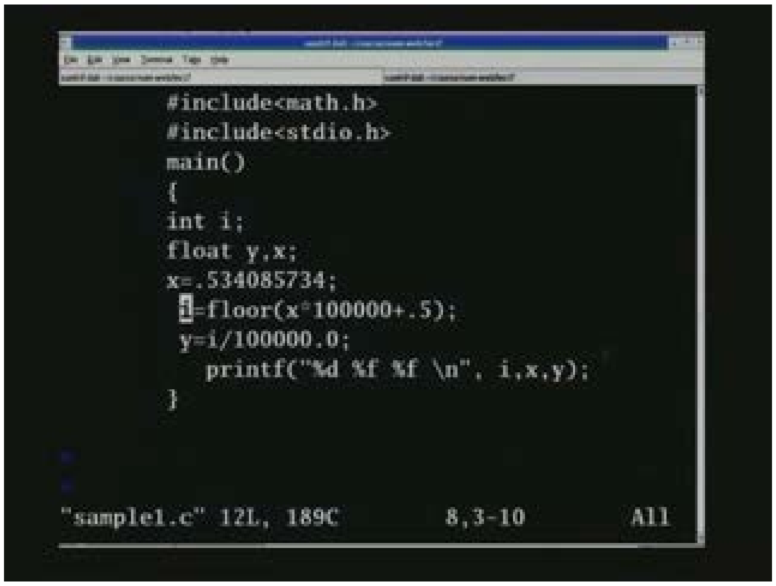
Summary

There are two types of errors encountered in numerical calculations.

(a) *Round off 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

(Refer Slide Time: 12:59)



```
#include<math.h>
#include<stdio.h>
main()
{
int i;
float y,x;
x=.534085734;
 i=floor(x*100000+.5);
 y=i/100000.0;
   printf("%d %f %f \n", i,x,y);
}

"sample1.c" 12L, 189C        8,3-10       All
```

We did look at some of the elements of programing before, so let us just put it into practice and look at some programs. For example, here we will see how I can approximate 1 by 6 to a certain number of digits, and look, then try to compute the errors from that. As I said, sometimes in the experimental data analysis, we need to restrict the calculation to certain number of digits.

We do not want to go beyond that. So how do we actually do this chopping and rounding, which we said before, that is 1 by 6? We can either chop off or round off up to some number of digits. So how do we do that? So we just look at this, an example. So here is a simple code to limit the number of decimal points in some variables in the calculation. So we will write a small code like this. So I have only a main program here, and I have declared an integer and two floating point numbers.

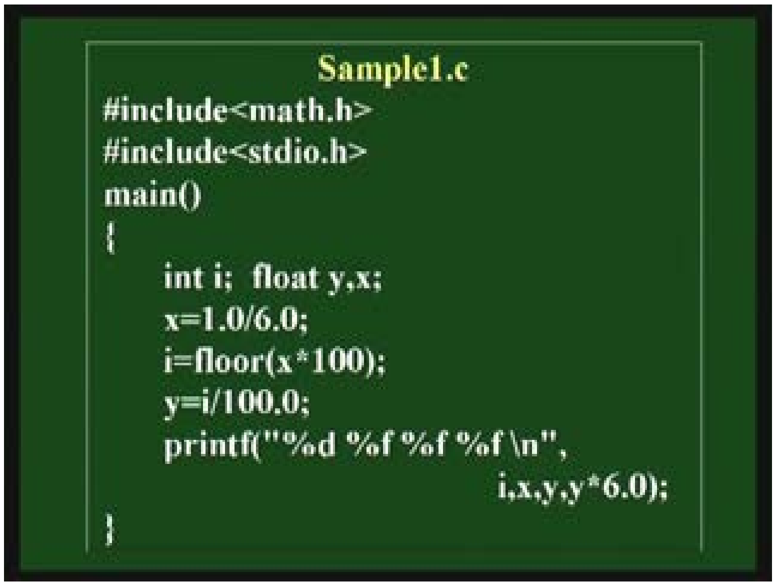
Integer as I, and floating point numbers as y and x, and I define my x up to 123456789 decimal points base 10. Now let us say I want to limit this to 2 decimal points. So how do I do that? I would do it in the following way. So I will have, I will take this number, and I will multiply, I take this number x, and I multiply it by 100,so I will get “53.4085734”, and then I will multiply it by 100. I will get the “53.40875”, and then I take the integer floor is here a function I call floor, which basically, what it does is to convert this number to its nearest integer, and gives me an integer as an output, i is an integer.

So if I take x and multiply it by 100, and take floor, I will get 53, and then I will divide that by again by 100, 100.0, and I will make another floating point number. So that will give me “.53”. So I basically, in this process, I have chopped off everything beyond two digits. So that is what we would see here if I run this program. You will see that that’s what I will get. So I started with “53.” something, and then I got I, made an integer of that, that’s 53, and I made that again into a floating point. So I got “.53”. So ignore the 0 because that does not mean anything here, because I chopped off all numbers below 3 here, below the second decimal place.

So this way, I can approximate numbers to certain decimal places. Now we can use this to actually do this 1 by 6. So I will put x as 1 by 6. Remember, when you define a floating point, and you are writing a fraction, always you have to write that as fraction of two floating points, or at least one floating point. If I write 1 by 6, this will go to 0. x is an integer, x is a floating point. It will look for an integer, if I write 1 by 6. So now, I am going to approximate that to 2 decimal places, and let's look at what we get. So we got ".16".

So the actual number should be "1.66666". You can see already here it has been rounded off, so 666 continues, and it is rounded off to 7, but I even further reduce that and I made 2 decimal places, and I chopped it off. I just simply chopped this number off after the second decimal place. So this is chopping. So now, we can also do rounding. What we have to do there is to make and add a ".5" to this, and then run that, so then we would get rounded instead of ".16" which we got here. Now we got ".17" because the next decimal place is more than .5, so that will be approximated. This decimal place has been approximated as 7. So that is an example of chopping and rounding.

(Refer Slide Time: 18:00)



Sample1.c

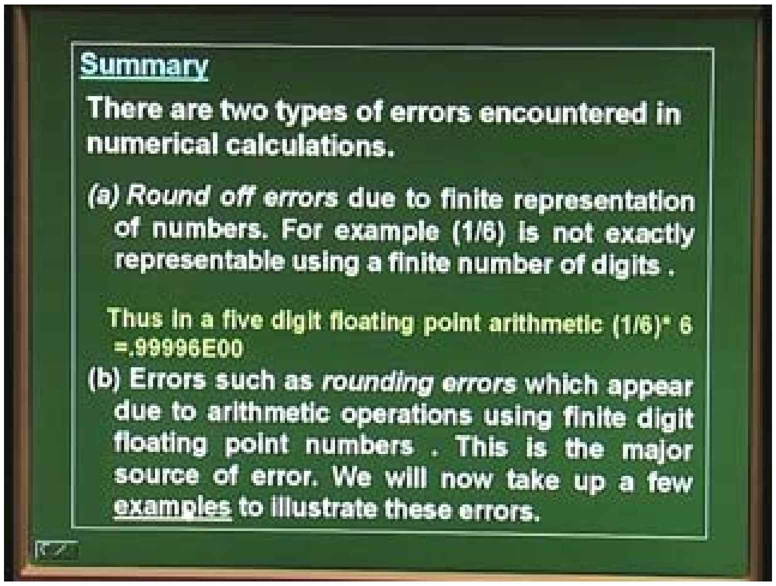
```
#include<math.h>
#include<stdio.h>
main()
{
     int i;  float y,x;
     x=1.0/6.0;
     i=floor(x*100);
     y=i/100.0;
     printf("%d %f %f %f \n",
                              i,x,y,y*6.0);
}
```

So now, if I just take this 1 by 6, let us say I had only a 2 decimal place computer, so I would have 1 by 6 rounded off as ".17". Now what will be the consequence of that?
The consequence, for example, here I had said that if I take 1 by 6 and multiply it by 6 again, it is rounded off to two digits. So what do I get? That is what we should look at.
So we should try to print here one more term, so that is 1 by 6 which is our y. 1 by 6 approximated to 2 decimal places was our y.

Now we will multiply that by 6 and see, what do we get? We have to multiply this by 6. So, y is 1 by 6, and we multiply this by 6, and that is what we want to look at. So we definitely don't get 1, we get "1.02". So that is the kind of errors which we were talking about coming from round-off errors, because we decided to actually reduce the, keep the precision, or the number of significant digits, which is also precision, only up to 2 decimal places, and as a consequence of that, if you take 1 by 6 and multiply it by 6, 6 being an integer, 1 by 6 multiplied by 6, we find that we don't get 1, and we get "1.02". so in this case, and then, we can also see how it will be if I just round it off instead of chopping it off.

So that will be interesting to see that there is a difference between these two values also. So we got ".96" instead of my ".6". So that is because I have restricted this to 2 decimal places. If you keep more decimal values, you get a slightly better answer. Here, for example, it is kept to 5 digits, and then you got "9996". It is a much better answer, because it has got 5 digits. So that is one kind of error. Then we have also rounding errors which come, appear, due to arithmetic operations using finite digit floating point numbers. Not only because of the representation of the numbers itself being rounded off, as we said, 1 by 6 being rounded off to ".17", but also because the arithmetic operations involve, also have incorporated round-off errors.

(Refer Slide Time: 21:06)



Summary

There are two types of errors encountered in numerical calculations.

(a) *Round off errors* due to finite representation of numbers. For example (1/6) is not exactly representable using a finite number of digits .

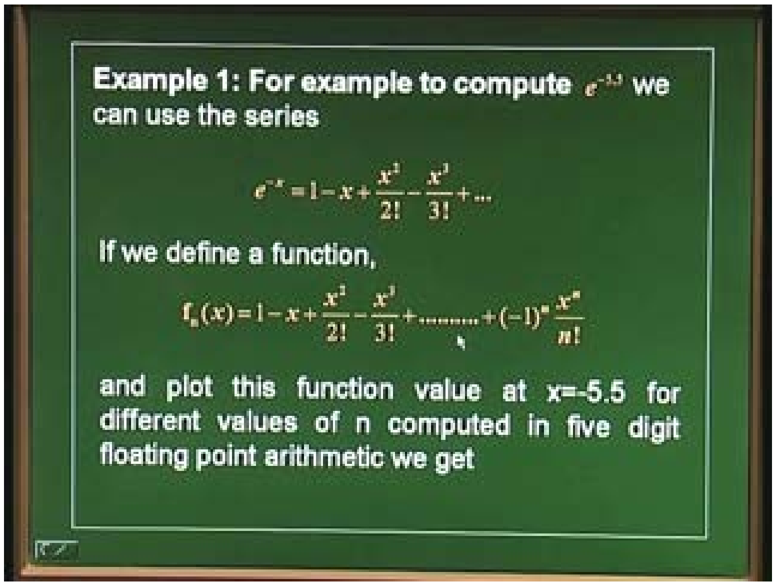
Thus in a five digit floating point arithmetic (1/6)* 6 =.99996E00

(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.

So this is the major source of error again. So here are some examples for it. We can look at one of the examples here, that is, if you try to compute e to the power of "–5.5". So we can use a series, as we saw before. We could say that I will compute e to the power of minus 6 with the series as 1 minus x plus x square by 2 factorial plus x cube by 3 factorial, etcetera.

Now I will define a function which gives me this just to approximate my errors, so I can, just to quantify my errors, I have to define the function as up to n terms as this function, and you could plot this function value at x equal to “minus 5.5” for different values of n computed in 5 digit floating point arithmetic. And then you would find that this value, as you keep on increasing n, would saturate beyond some point. So that is, we would try to look at this saturation in a, with an example. So let us look at that example, that kind of a series. So we want to compute x to the power of “minus5.5”. That is what we wanted to compute.

(Refer Slide Time: 22:01)



Example 1: For example to compute $e^{-5.5}$ we can use the series

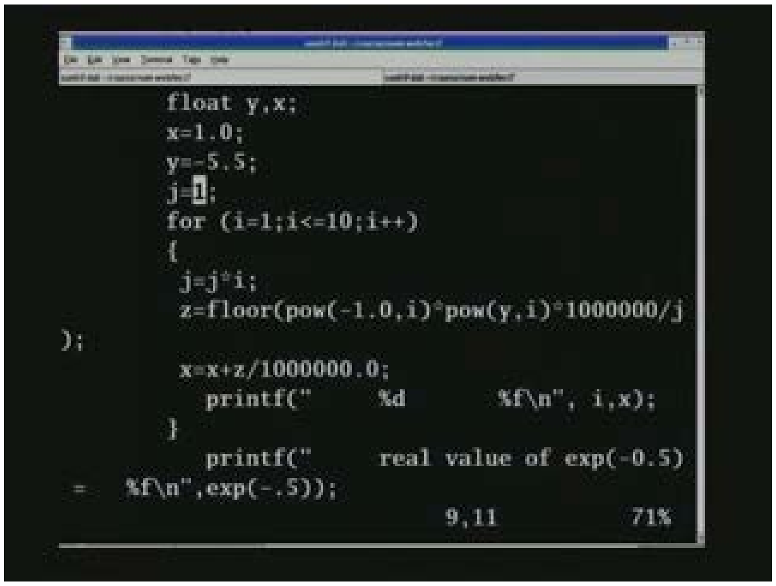
$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \dots$$

If we define a function,

$$f_n(x) = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \dots\dots\dots + (-1)^n \frac{x^n}{n!}$$

and plot this function value at x=-5.5 for different values of n computed in five digit floating point arithmetic we get

(Refer Slide Time: 22:56)



```
        float y,x;
        x=1.0;
        y=-5.5;
        j=1;
        for (i=1;i<=10;i++)
        {
         j=j*i;
         z=floor(pow(-1.0,i)*pow(y,i)*1000000/j
);
         x=x+z/1000000.0;
           printf("    %d       %f\n", i,x);
        }
           printf("    real value of exp(-0.5)
 =   %f\n",exp(-.5));
                                  9,11          71%
```

So let us compute that. So we will keep this term as “minus 5.5” and then we would just, so what I am doing here, I have taken, I have a small program here. Again, I define some integers, and I have some floating points, y and x. So I started with x equal to 1 because my first term in the series is 1. So that will be the first term in my series.

So I kept it as started into 1, and my power here, which I call y in this program, I will just change this to, so let us start with this as x and this as y. So we have the function, f, here, which I call y, which is 1 to start, and I have the power which is "minus 5.5", and then I run it through a series. I am going up to 10 places that means, 10 terms. It will go up to n equal to 10 in the series that is basically what I am going to do. I am going to, i going from 1 to 10, and then I know that the denominator of this series goes as 2 factorial, 3factorial, etcetera, up to n factorial.

So this is to compute that I have term j equal to 1, and j keeps going from j star i, so when i is equal to 2, it is 2 when i equal to 3, it's 1 into 2 into 3, etcetera. This is clear. So we have j equal to 1 to start with. So when "i" is 1, "j" is 1. When "i" is 2, the second term comes in, so the second term would be this term that is "n" equal to 2. When n equal to 2 comes it will be 2factorial which is 2. So that's what the j would take. So j has become 2, and when n becomes 3, we have 3 factorial, which is 1 into 2 into 3. So j was already 2, and then that gets multiplied by i equal to 3. So, it goes into 3 factorial, etcetera. Then I compute the "n"th term here. This is actually the nth term which I am computing, which is basically just this term. That is the "n" th term that is minus 1.

So "p o w" is a built-in C function which will compute the power of something. So I have computed the power of minus 1 to the power i here. That is minus 1 to the power i is computed using this function, p o w, which is a built in math library function, and then I multiply that by y to the power of n. That is what I do here, sorry x to the power of n, it should be. That is exactly this term. So I have minus 1 to the power of n, and I multiply that by x to the power n, and then I divide it by I, divide it by the j, which is my n factorial. So now again I want to keep this into 5 digits. So I will multiply what we saw before, the way to keep it to 5 digits would be to multiply it by 1,2,3,4,5, so, 100,000, and then I take that function and divide it by 100,000.

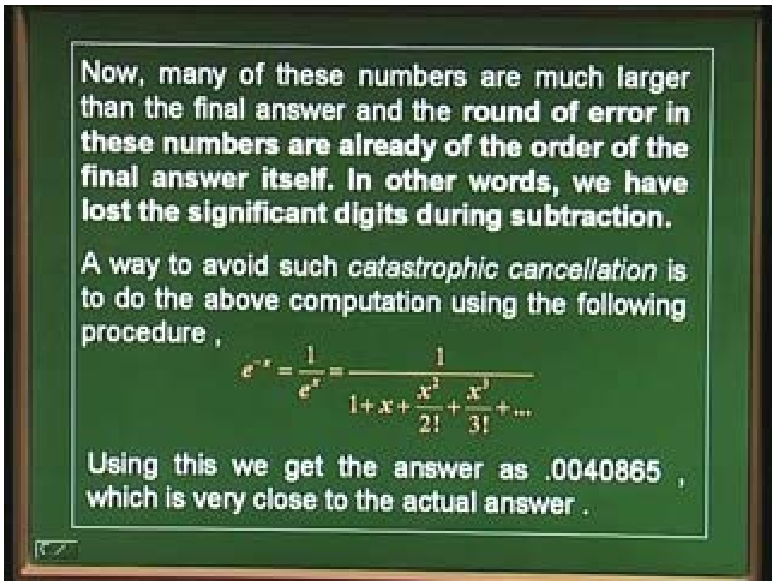
So I take this z and divide it by 100,000. Then I get up to accuracy, up to 5 digits. So that should be y. So now I am going to print this off. I am going to print the function here, the value of the function. How does it change as the series goes by? so first starting from 1, starting from i equal to 1, that means I am starting from here, that is what we are going to look at. Let us look at that. Let me just make sure that what we are doing is correct. So we took "5.5" as x, and we are multiplying x to the power of i, and dividing it by j which is the n factorial, and we keep 1,2,3,4,5 terms, and 1,2,3,4,5 terms here, and then we are adding that to the series, and we are trying to get into that.

We are trying to go up to four, n equal to 4. Let us go up to n equal to 4 and then see these terms, one by one. Let us see whether it will converge or not. So we are printing out each term, term by term, these values of x to the power of n by n factorial minus 1 to the power of n. That is, what we want to look at. Let us do that and we compile this program. We will compile this program first, and then we will run it. So the first term that we saw was, of course, 1 into 10 to the power of minus 1, 1 minus x plus x square by 2. So the first term is 1.

I have not printed that out, and the second term, that is n equal to 1 term. This is the n equal to 0 term, n equal to 1 will be simply “minus 5.5”, that is what we see here. So 1 minus that, 1 plus this term would be “minus 4.5”, and the next term is “15.125”, 5 places, 5 digits, is what I am keeping. So whole computation just done up to 5 digits. I will keep 5 digits. So it is “10.625”, and the next term is again negative, and the fourth term is positive, etcetera. So the value we are getting is going as “minus 4.5”, then it goes to 10, and then it goes to minus 17, then it goes to 21, etcetera. We say we are not reaching anywhere on this example. So the reason for this is if you look at, again summarized here, so I kept 5 decimal places, first is 1, and this is normalized floating point representation to base 10.

So the first is 1 and the next term is “minus 5.5”, or .55 into 10 to the power of 1, and the next term is, we saw was 15, that is, “1.15125” into 10 to the power of 2, and the next term is “minus .2733”, that is “27.73” into 10 to the power of 2, etcetera, that is what we saw here. The point is, the sum is not going to converge, so the reason for this is the following, is that we are actually trying to add and subtract numbers of the same magnitude. The final answer which we saw actually, is very small. The final answer is something like “.004087”. That is the final answer. We will look at the numbers which we are trying to divided and subtract, so that is of this order.

(Refer Slide Time: 29:58)



Now, many of these numbers are much larger than the final answer and the round of error in these numbers are already of the order of the final answer itself. In other words, we have lost the significant digits during subtraction.

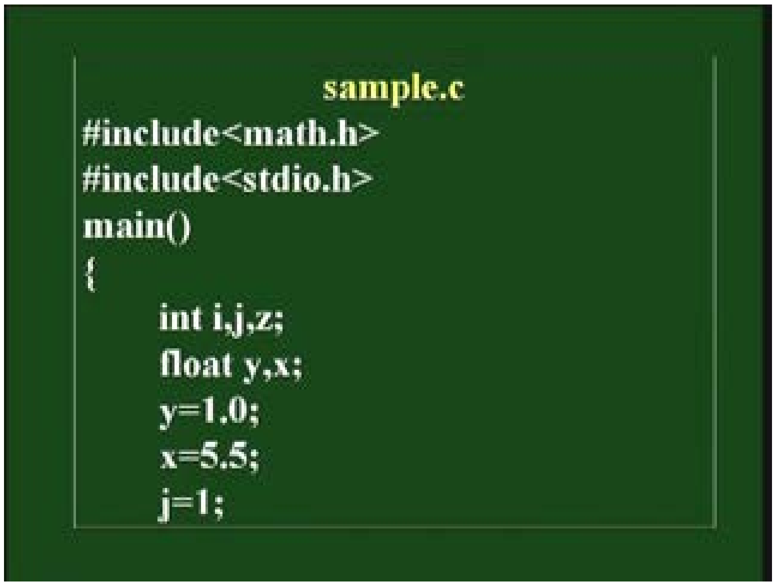
A way to avoid such *catastrophic cancellation* is to do the above computation using the following procedure ,

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

Using this we get the answer as .0040865 , which is very close to the actual answer .

So we are trying to do “5.5” 15, 27, 38, etcetera. They are all much larger than the final answer which you want to get, so basically, we are losing, we are trying to look for something which is much smaller than the numbers which are involved in addition and subtraction. So we are actually losing significant figures. We are not going to get a very good answer by doing this kind of a series expansion in this particular case. It is a very typical example of a bad series approximation.

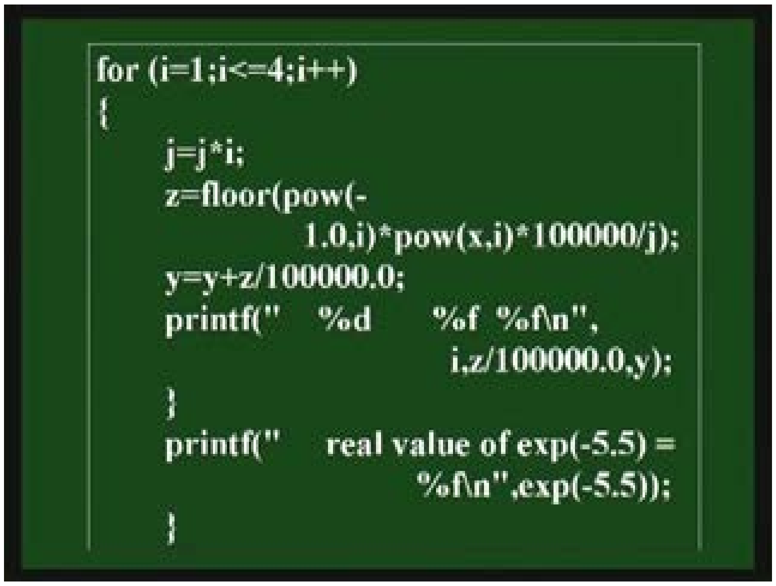
(Refer Slide Time: 30:38)



sample.c

```
#include<math.h>
#include<stdio.h>
main()
{
      int i,j,z;
      float y,x;
      y=1.0;
      x=5.5;
      j=1;
```

(Refer Slide Time: 30:56)



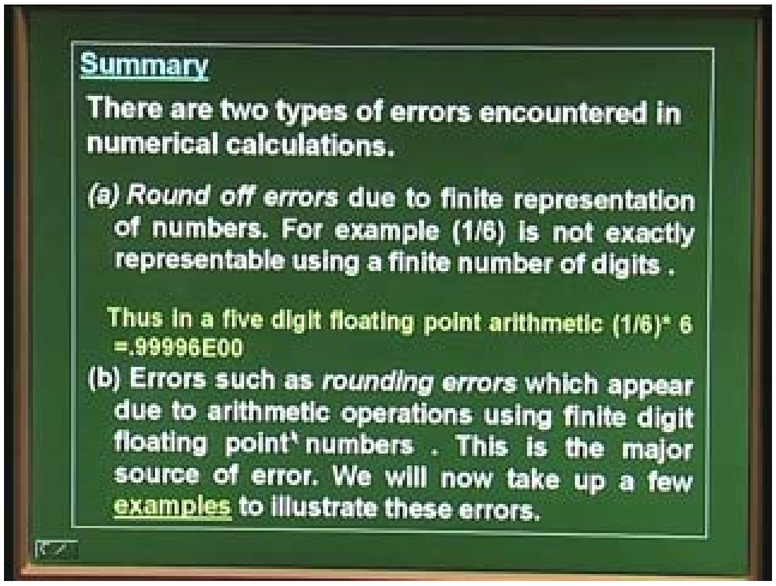
```
for (i=1;i<=4;i++)
{
      j=j*i;
      z=floor(pow(-
                  1.0,i)*pow(x,i)*100000/j);
      y=y+z/100000.0;
      printf("    %d      %f  %f\n",
                              i,z/100000.0,y);
      }
      printf("    real value of exp(-5.5) =
                              %f\n",exp(-5.5));
      }
```

So one way to get rid of this is to, this catastrophic cancellation as it is called, of digits of precision, or significant digits, is to use a different type of series. Instead of using e to the power of x, you put 1 by e to the power of x. So what is the advantage? The advantage is that we are simply summing these terms. So we do e to the power of x instead of e to the power of minus x. We put 1 by e to the power of x, and then we just, we are not doing plus minus here. Everything is plus, and we take 1 by that term. That gives us much faster, much more reliable answer as “.0040865”. We can actually try and do this. Instead of computing e to the power of minus that, we would just simply compute plus.

We just do e to the power of x, and the final answer will then turn out to be 1 by that answer. So instead of plotting 1 by, instead of printing out y, we should have 1 by y there. So, that is the way of doing this. Now I am going to do e to the power of x instead of e to the power o minus x. I am going to use a series for e to the power of x. That will be this. So that is the series we are going to do here. We are going to do the power of x to the power of 5, or x is positive, and every odd term will be positive because now I am taking power of 1 to the power of i which is 1. So I just take the power series and then I again approximate to5 decimal places, and then I compute the value 1 by y, that is what we wanted.

Now let us see what we get with this example. So, if you do this same thing here again, so I am getting a value which is much closer, much closer, I am supposed to plot 1 by y. I am keeping the same terms here. I am just taking the same terms here and then computing it. So, I am keeping the same terms, that is, x value “5.5”, now it’s plus, and then x square by 2 x cube, 3 factorial, that is 6 and x to the power of 4 by 4 factorial, etcetera. Now I summed up this thing, these terms, so I am taking the 1 by that. That is the inverse of that. So I can see that this time it is converging into the value which I want, unlike the previous case, where it was not converging at all.

(Refer Slide Time: 35:43)



Summary

There are two types of errors encountered in numerical calculations.

(a) *Round off 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

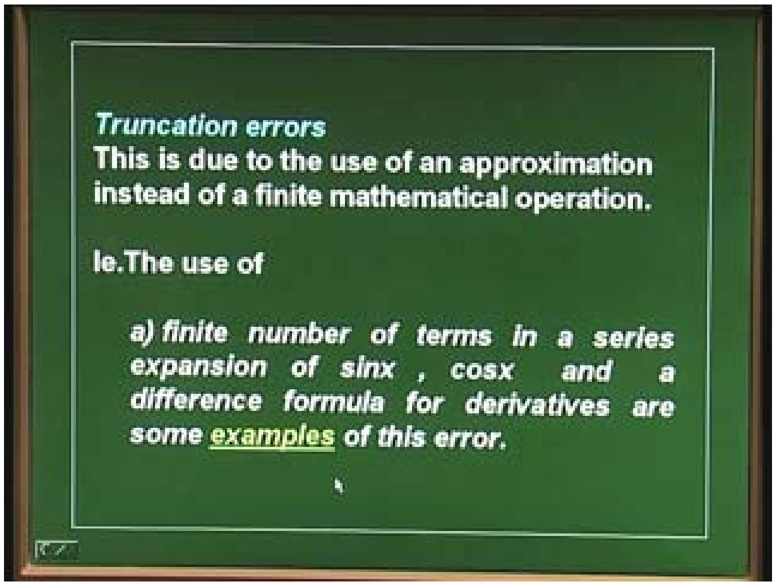
(b) Errors such as *rounding errors* which appear due to arithmetic operations using finite digit floating point numbers . This is the major source of error. We will now take up a few examples to illustrate these errors.

So here we are getting a much better convergence, and if you go into a series with some more terms here, we would find that we have come very close to the value which we wanted. So we are going very close to the value which we want, which we were not getting, by doing e to the power of minus x with 5 decimal places. So that is the clear demonstration of cancellation errors, or the catastrophic cancellation of errors. So we can compute, i repeat e to the power minus x in two different ways, we can compute them computed using a power series of 1 minus x plus x square by 2 minus x cube by 3, and, or, we could compute it using 1 by "e" to the power of x, which is 1 by x plus x square by 2 factorial plus x cube by 3 factorial, etcetera.

And what we find is that these two give us very different results, and this is because of the loss of significant digits due to the fact that we are trying to get a number which is much smaller than the whole numbers involved in the arithmetic operation itself. So, now that is, one thing which we can see. To summarize, there are two types of errors. We have round off errors, and errors which are appearing due to arithmetic operations. That is, what we just now saw.

And then we have other types of errors which are truncation errors, which we said. So that is due to the use of approximate instead of finite mathematical operations, approximation, instead of finite mathematical operation. What I mean by this, for example, if you want to do sine of x, instead of using sine of x as finite mathematical operation we could use a series for sine of x, and that could give us truncation errors.

(Refer Slide Time: 36:10)



Truncation errors

This is due to the use of an approximation instead of a finite mathematical operation.

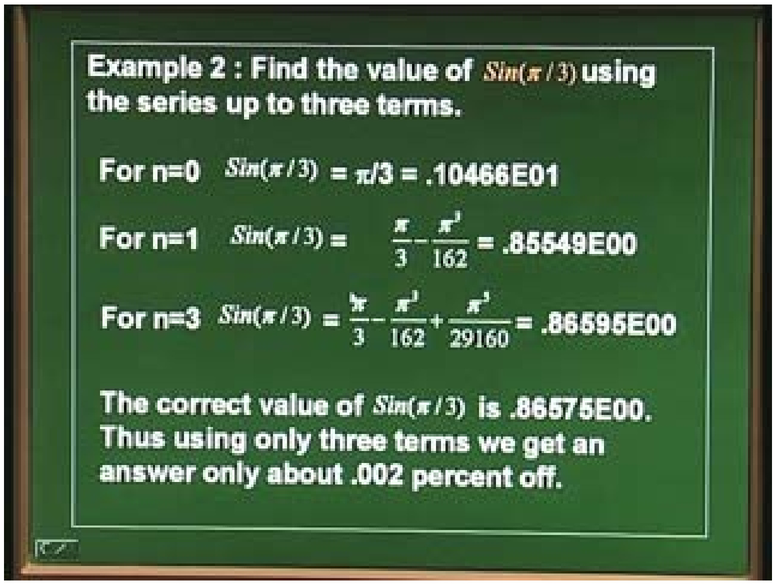
Ie.The use of

a) finite number of terms in a series expansion of sinx , cosx and a difference formula for derivatives are some examples of this error.

So that is, for example, we have sine phi by 3 is this value. If I just use the series terms, and stop it at n equal to 0, that is, number of terms in the series is n equal to 1, n equal to 2, n equal to 3, etcetera, and the correct value would be ".86575". So now, if I use only one term in the series, I am getting a value which is completely off. Or if I use two terms in the series, I get a slightly better, and if I use three terms in the series, I get little better, etcetera. What I want to show you here is that if you use a series for this particular function you will introduce some errors.

So that is another type of error which you would have, called the truncation error. That is the summary of errors which we have. So now, the point is this. Two errors, that is the truncation errors and round-off errors, can often compete with each other, as we saw that in the sine series, I can use one term in the series, or I can use two terms in the series, or I can use three terms in the series, etcetera, up to n terms, and I increase the number of terms in the series. I get better and better answers, as we saw, but it is not always true.

(Refer Slide Time: 37:00)



Example 2 : Find the value of $Sin(\pi/3)$ using the series up to three terms.

For n=0 $Sin(\pi/3) = \pi/3 = .10466E01$

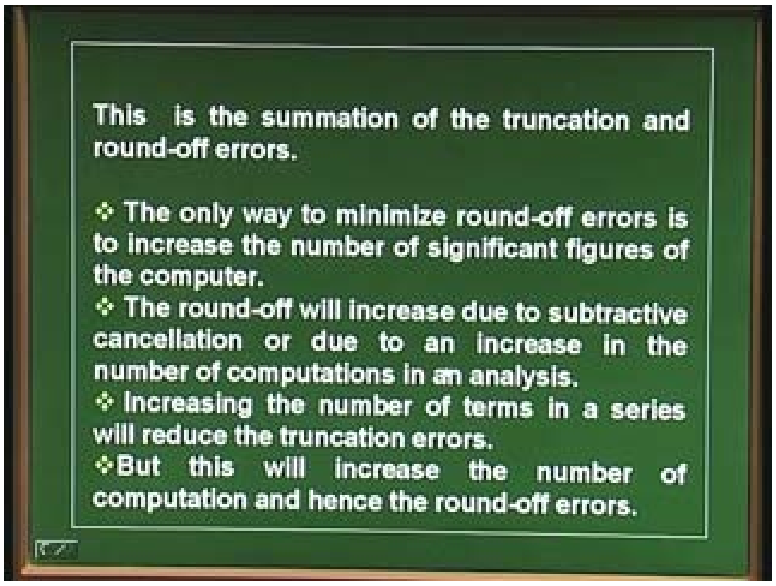
For n=1 $Sin(\pi/3) = \frac{\pi}{3} - \frac{\pi^3}{162} = .85549E00$

For n=3 $Sin(\pi/3) = \frac{\pi}{3} - \frac{\pi^3}{162} + \frac{\pi^5}{29160} = .86595E00$

The correct value of $Sin(\pi/3)$ is .86575E00. Thus using only three terms we get an answer only about .002 percent off.

The reason being the previous one in which we saw that as we increase the number of mathematical operations, you have round-off errors introduced. So we are kind of competing between the round-off errors and truncation errors.

(Refer Slide Time: 37:59)

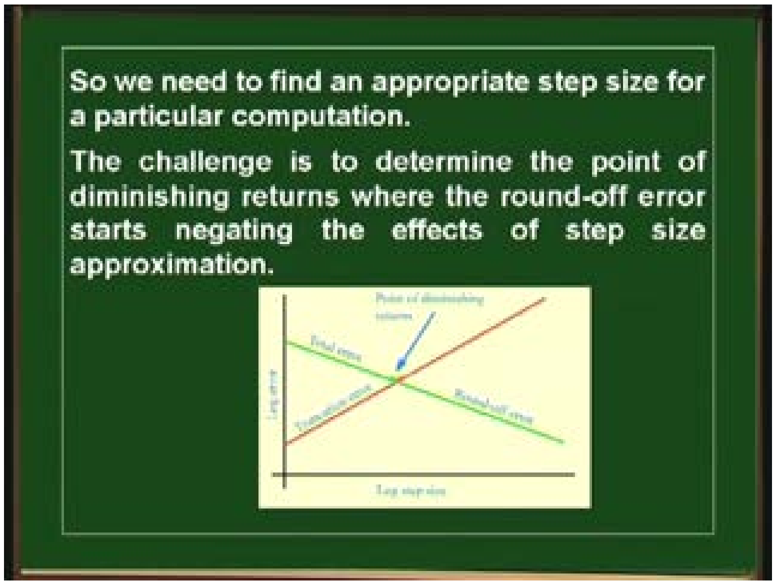


This is the summation of the truncation and round-off errors.

- 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.
- Increasing the number of terms in a series will reduce the truncation errors.
- But this will increase the number of computation and hence the round-off errors.

So the increasing number of terms in the series will reduce the truncation errors, but this will increase the number of computations, number of operations, arithmetic operations, and that would introduce round-off errors.

(Refer Slide Time: 38:08)



So we need to find an appropriate step size for a particular computation.

The challenge is to determine the point of diminishing returns where the round-off error starts negating the effects of step size approximation.

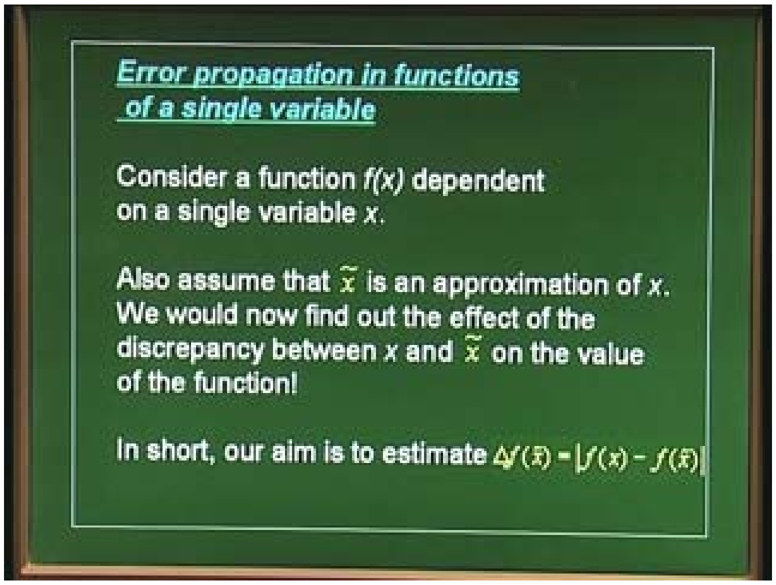
Point of diminishing returns
Total error
Truncation error
Round-off error
Log error
Log step size

So there is actually a competition. That is what we are trying to show here. So if you use long step size, for example, or if you use less number of terms in the series then we have larger truncation errors, but then you have round-off errors going down so the total error which is dominated in the beginning by, intervals of step size, which is the inverse of the number of terms in the series. So if I, step size, if I use, I have large truncation large error due to round-off errors in the beginning, while less truncation errors, because I used large number of steps to get into one particular point. For Example if I want to actually compute some series up to, for an interval of pi by 3, sine pi by 3, I could use small values of sine.
I could compute small values of sine and then go step by step, and then go into sine pi by 3.

So that means that the truncation errors are very low when I use a small step size, while the round-off errors are larger, but as I increase the step size, I will have less round-off errors, because to get to that point I have to take less number of mathematical operations.
So round-off errors come down but the truncation error goes down. So there is a point at which this kind of balance is what is called the point of diminishing returns. That is the point. It is not always true that if you use large number of terms in the series, you get a better answer. That's what the point. I want to emphasize here. So then now we could see that we have introduced, we are introducing errors when we are doing a mathematical operation. So now, that is one particular operation.

Now the question: "Can we say something about how does the error propagate down the program?" That means, you have a series of computations in a program. So now, if I make an error, a round-off error or a truncation error at one step, how does it propagate? That is something which we should be looking at. That is this part of the, this is what this part we will look at. So we look at error propagation in functions of a single variable. To start with, we just try to quantify this, and then we would like to look at multiple variables, and then we would like to see some examples of this using program.

(Refer Slide Time: 39:59)



Error propagation in functions of a single variable

Consider a function $f(x)$ dependent on a single variable x.

Also assume that $\tilde{x}$ is an approximation of x. We would now find out the effect of the discrepancy between x and $\tilde{x}$ on the value of the function!

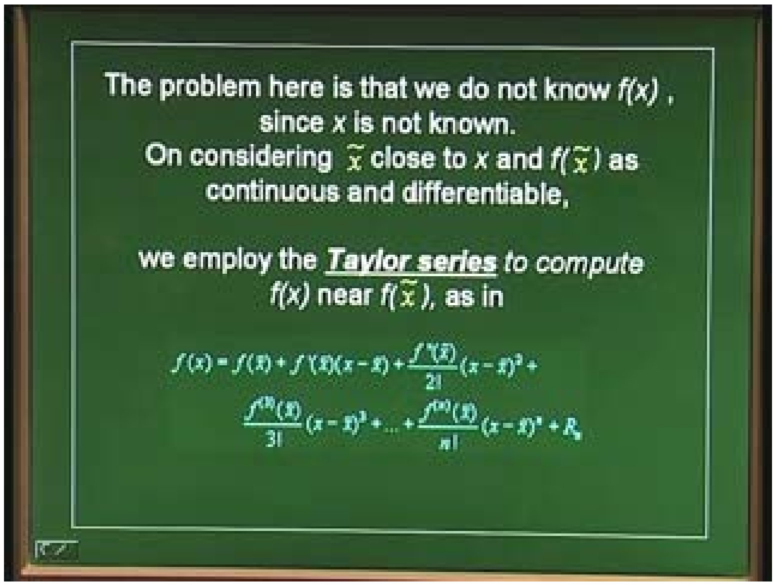
In short, our aim is to estimate $\Delta f(\tilde{x}) = |f(x) - f(\tilde{x})|$

So let us first define this correctly. So we have a function f of x, which is dependent on a variable x. Now, let us say we introduce an error in x itself that is representation. Because of the representation finite number of digits there is an error in x. So how does that reflect on the function? The error, how does this reflect on a function? Or, what I mean by this is, how much of this error in x will propagate to the evaluation of the function f of x?

So that is what is called, that will be called propagation of error. So we assume that x tilde here is an approximation of x. Let us say x was 1 by 6, and then we approximate into 2 decimal places. That is one possibility. So that is one example. So, or 3 decimal places we approximated x. Now we would find out what, find out the effect of the discrepancy between x and x tilde, on the value of the function, on the value of f of x. That is what we want to look at, or in short, our aim would be to approximate, to find out what f of x minus f of x tilde is.

I said estimate, because we really do not know what f of x would be, because we may not know what x, actual value of x is. So we approximate what the value of x is or in other words, how much, delta x here, that is, x minus x tilde, and delta f, what is the relation between delta f, which is the error in the function f of x, and delta x, which would be x minus x tilde, which is the error in the function, error in the variable x. So how much, what is the relation between these two? So that is something which we should try to see. Again, we are not interested in the sign, only interested in the mod. So as I said, the problem here is we do not know the actual value of f of x, so what we know is f of x tilde. So we can compute only f of x tilde.

(Refer Slide Time: 43:01)



The problem here is that we do not know *f(x)*, since *x* is not known.
On considering $\tilde{x}$ close to *x* and $f(\tilde{x})$ as continuous and differentiable,

we employ the ***Taylor series*** *to compute* *f(x)* near $f(\tilde{x})$, as in

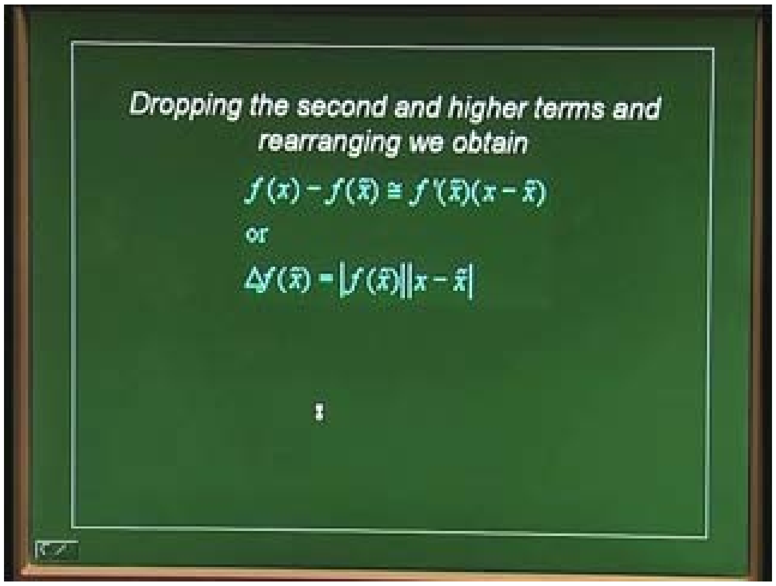
$$f(x) = f(\tilde{x}) + f'(\tilde{x})(x-\tilde{x}) + \frac{f''(\tilde{x})}{2!}(x-\tilde{x})^2 + \frac{f^{(3)}(\tilde{x})}{3!}(x-\tilde{x})^3 + \ldots + \frac{f^{(n)}(\tilde{x})}{n!}(x-\tilde{x})^n + R_n$$

So let us assume that x tilde is close to x. We assume that, and then we will also assume that f of x tilde is a continuous and differentiable function. So we make two assumptions here, that x tilde is close to x, and f of x tilde is a continuous and differentiable function, and then we can employ a Taylor's series to compute f of x near f of x tilde. So that is the series which I have given here. That is the Taylor series of f of x. x is the real value. So f of x. So we expand f of x in around f of x tilde. So f of x tilde is an approximation to x.
We expand this. To expand this in the series called Taylor series, we need to find the derivative of the function f, and hence, is the approximation that it is differential at all points.

What we mean by this is that f is a smooth function. It does not have sharp kicks. So we will expand the series So the first term is f of x tilde, and then f prime, that is f prime means derivative of "f" with respect to x at x equal to x tilde. Remember, f prime of x is the derivative of the function which we know with respect to x at x equal to x tilde into delta x, x minus x tilde, and the next term is f double prime which means the second derivative of "f" del square f by del x square at x equal to x tilde divided by 2 factorial into delta x whole square.

And the third term is the third derivative here. The notation is f 3 is the third derivative, that is, del cube f divided by del x cube at x equal to x tilde divided by 3 factorial x minus x tilde q, etcetera, up to the nth term. That is how we could expand this in this series. So that gives us a relation between f of x and f of x tilde x minus x tilde. So I could take the difference, this, that tells me that the delta f which is the relation, which is difference between f of x and f of x tilde, is related to delta x tilde, delta x which is the difference between x and x tilde through the derivative of "f" to the first approximation. That is what we would write here. To first approximation, I could just write f of x minus f of x tilde as f prime of x into x minus x tilde.

(Refer Slide Time: 46:04)



Dropping the second and higher terms and rearranging we obtain

$$f(x) - f(\tilde{x}) \cong f'(\tilde{x})(x - \tilde{x})$$

or

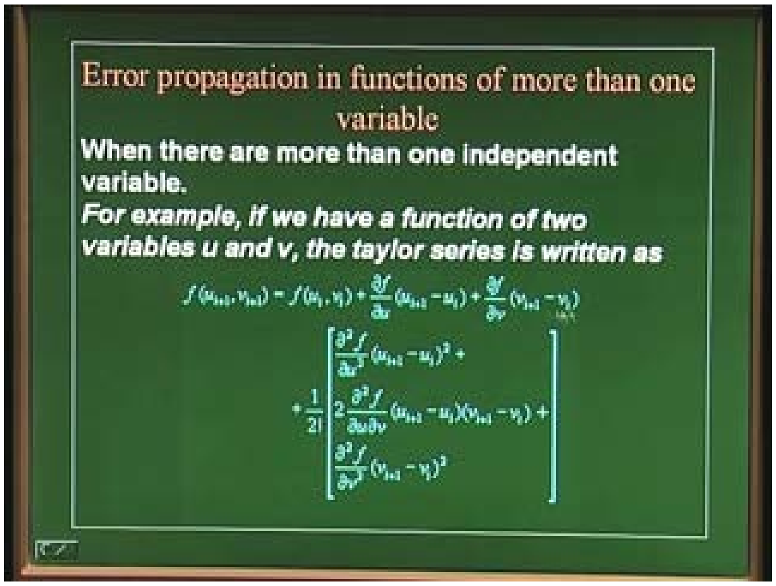
$$\Delta f(\tilde{x}) = |f(\tilde{x})||x - \tilde{x}|$$

So what I said here is I am assuming that this delta x which is x minus x tilde which is pretty small. So as I go to higher terms in the series, I go to 0 because it decreases pretty fast. Delta x square by 2 and delta x cube by 6, etcetera,. So if delta x is smaller than 1, much smaller than 1, then it would just reduce very fast, so I can, to first approximation, assume that f of x is simply f of x tilde plus f prime of x into x minus x tilde. That is what I have done here. So that gives us the propagation of error. So how does the, now ask me the question, that, that's f prime here, that is a mistake, so if I, you ask me the question that if I make any errors in x, the representation of x, so what is the error in delta f, and that's given by this. So, f prime, that derivative of f at x equal to x tilde into x minus x tilde.

So if it is a sharply varying function, if f tilde, if f prime, is very large, then that will introduce large errors. So we can actually generalize this idea into functions of more than one variable, and then we would again expand that in a Taylor series. So the Taylor series, only up to the second term here, we are only interested in the first approximation. So I have given in the series only up to second term, that is, say 2 variables, say u and v. So then, I would say f of u, v that is the real values. Next approximation would be the f of u_i, v_i here. I mean the next approximation to this.

So if I check the i plus 1th digit approximation to "i" th digit approximation, so then I would say that f of u_i v_i, and the derivative of that, evaluated at that values of u_i and v_i, the derivative of "f" with respect to u evaluated at u_i and v_i multiplied by delta u, and the derivative of "f" with respect to v evaluated again at the u_i and v_i multiplied by the error in v between the two approximations ,i plus 1 level approximation to "i" th approximation. So the first term involves the first ordered terms are 2. That is, one derivative with respect to u, and another derivative with respect to v, and the second order terms now have three terms, one is the second derivative of the function with respect to u again evaluated at u_i and v_i multiplied by delta u square and plus 2 into the second derivative, or the cross derivative, the cross second derivative, that is del square f by del u del v multiplied by delta u, and delta v plus second derivative of "f" with respect to v multiplied by delta v whole square. So again, the first approximation, the error in f due to error in u and v would be given by the derivatives of these functions at that value multiplied by the error in the delta u and delta v. Again, some of that we do not know what the actual values of these functions are? So if we have, the only thing which we are trying to address here is if we have an idea about how much is the error here then how much of that is propagated into the function evaluation? That is the only thing which we are trying to understand here. So that is summarized here.

(Refer Side Time: 48:28)



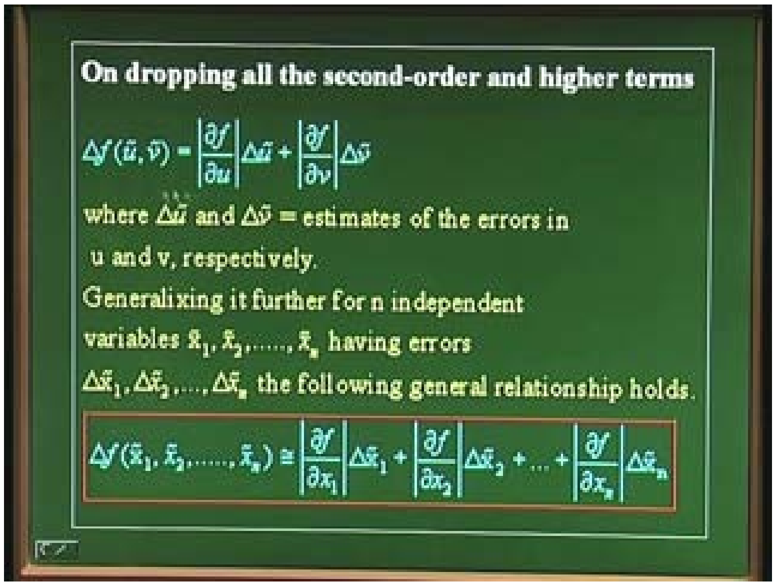
Error propagation in functions of more than one variable

When there are more than one independent variable.

For example, if we have a function of two variables u and v, the taylor series is written as

$$f(u_{i+1}, v_{i+1}) = f(u_i, v_i) + \frac{\partial f}{\partial u}(u_{i+1} - u_i) + \frac{\partial f}{\partial v}(v_{i+1} - v_i) + \frac{1}{2!}\left[\frac{\partial^2 f}{\partial u^2}(u_{i+1} - u_i)^2 + 2\frac{\partial^2 f}{\partial u \partial v}(u_{i+1} - u_i)(v_{i+1} - v_i) + \frac{\partial^2 f}{\partial v^2}(v_{i+1} - v_i)^2\right]$$

(Refer Slide Time: 50:02)



On dropping all the second-order and higher terms

$$\Delta f(\tilde{u}, \tilde{v}) = \left|\frac{\partial f}{\partial u}\right|\Delta\tilde{u} + \left|\frac{\partial f}{\partial v}\right|\Delta\tilde{v}$$

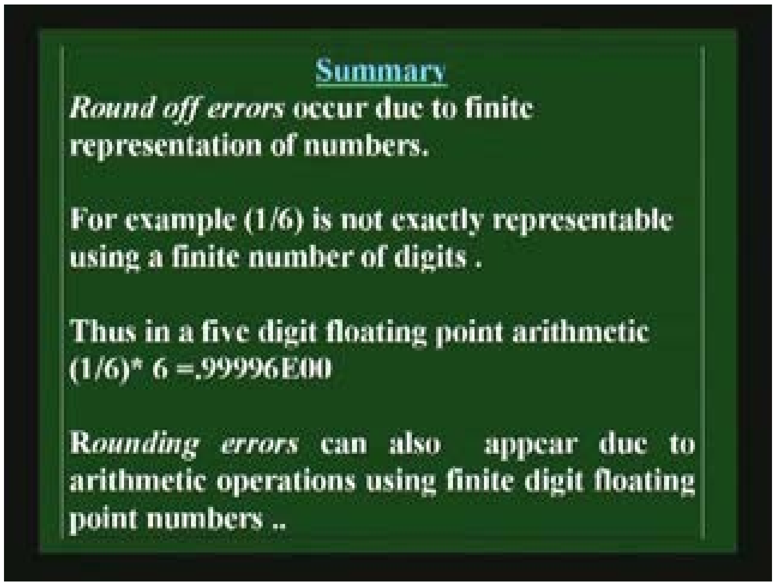
where $\Delta\tilde{u}$ and $\Delta\tilde{v}$ = estimates of the errors in u and v, respectively.

Generalixing it further for n independent variables $\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n$ having errors $\Delta\tilde{x}_1, \Delta\tilde{x}_2, \ldots, \Delta\tilde{x}_n$ the following general relationship holds.

$$\boxed{\Delta f(\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n) \cong \left|\frac{\partial f}{\partial x_1}\right|\Delta\tilde{x}_1 + \left|\frac{\partial f}{\partial x_2}\right|\Delta\tilde{x}_2 + \ldots + \left|\frac{\partial f}{\partial x_n}\right|\Delta\tilde{x}_n}$$

So I can keep writing like this. So basically, the delta f here would be the derivative of "f" with respect to, for any number of terms I generalize this to many number of terms now. I have many variables, x_1, x_2, x_3, etcetera, up to n terms. And then I have approximations of that as x1 tilde, x2 tilde, up to x n tilde, having errors. Let us say delta x_1, delta x_2, delta x_n, etcetera. Now if I generalize this as our idea of delta f with 2 variable things into n variables, then I would get an approximation like that. So, any of these derivatives being large would mean that this function will have a large error. So that is where we stop. We try to see some of the implementation of these ideas into evaluating errors and propagation of errors in some real problems as we go along in this course.

(Refer Slide Time: 51:02)



Summary

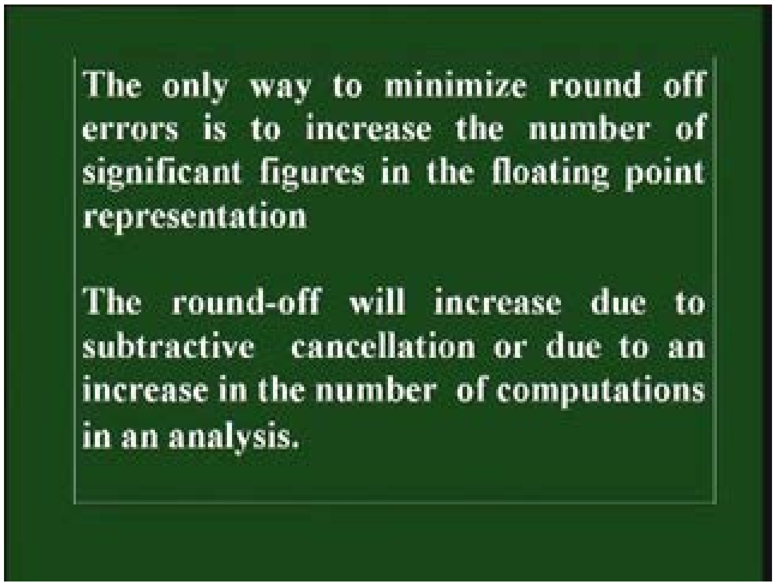
Round off errors **occur due to finite representation of numbers.**

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

R*ounding errors* can also appear due to arithmetic operations using finite digit floating point numbers ..

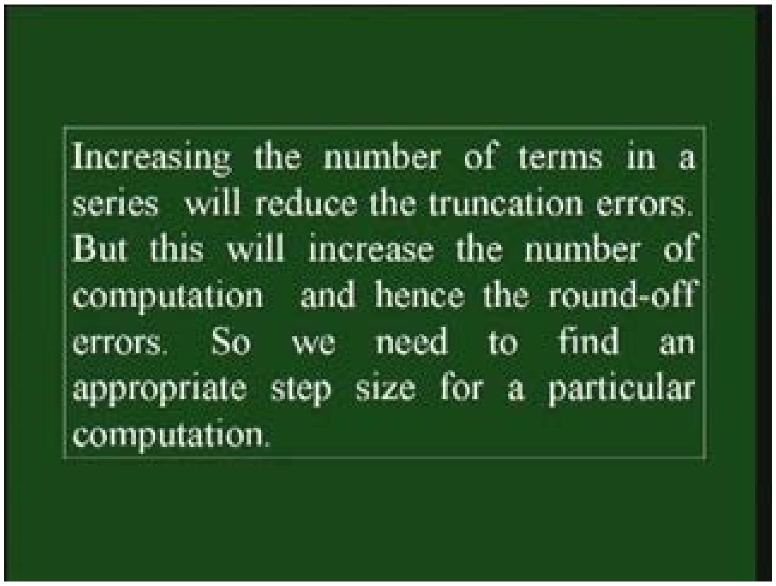
(Refer Slide Time: 51:22)



The only way to minimize round off errors is to increase the number of significant figures in the floating point representation

The round-off will increase due to subtractive cancellation or due to an increase in the number of computations in an analysis.

(Refer Slide Time: 51:42)



Increasing the number of terms in a series will reduce the truncation errors. But this will increase the number of computation and hence the round-off errors. So we need to find an appropriate step size for a particular computation.