

**Machine learning for engineering and science applications**  
**Professor Balaji Srinivas**  
**Department of Computer Mechanical Engineering**  
**Indian Institute of Technology Madras**  
**Machine Representation of Numbers, Overflow, Underflow, Condition Number**

(Refer Slide Time: 0:14)


Machine Learning for Engineering and Science Applications

Machine Representation of Numbers  
Overflow  
Underflow  
Condition Number

Many of the examples in these slides are from 'Applied Numerical Methods' by Steven Chapra


In this video we will be looking at how your machine how your computer represents numbers okay and a few phenomena which can go wrong when we think that the way the computer processes numbers is the way we do things on paper intuitively okay. So this idea is something called overflow and underflow, we will also look at another idea called condition number, most of the examples not the slides but just the examples that we have taken is from a good book, an introductory book for numerical methods by Steven Chapra.

(Refer Slide Time: 0:52)



## Machine Arithmetic

- The derivations and expressions in the previous slides assume real number arithmetic.
  - That is, all calculations are assumed to take place perfectly
- However, in practice, the machine only stores numbers to a finite precision
- This arithmetic is called “Finite Precision Arithmetic”, “Machine Arithmetic”
  - Or “Floating Point arithmetic” in the special case of floating point numbers
- This can have surprisingly important consequences



So let us look at the idea of machine arithmetic, so in the previous slides when we were doing optimisation, we were doing it theoretically so if you want to find out the minimum of a function, you will simply say that the gradient at the minimum or at the optimum is 0. Now, in order to do this if you were to do it on paper or if you do it using symbols, you will assume usually real number arithmetic okay, you will assume that you can calculate digits to as much precision as you want. We will also assume you know you will say that you know if I differentiate  $X^2$  with respect to  $X$ , I am going to get  $2X$  but in practice remember we do not deal with symbols, we in fact do not even deal with images, as I have said multiple times so far, we actually deal with only Numbers okay and specifically numbers of finite precision as you will see, this will start making sense as you go a little bit further.

This kind of arithmetic is called finite precision arithmetic or machine arithmetic ok. In some cases you can call it “Floating point arithmetic” which is the most common as far as we are concerned in the special case of floating point numbers. Floating point number means **of**, the numbers where we deal with the real numbers rather than with integers. Now, the fact that we have only a finite precision can actually have surprisingly important and sometimes surprisingly catastrophic consequences okay, so let us take one such recent example ok.

(Refer Slide Time: 2:29)



### The Ariane 5 disaster


- Ariane 5 is a European launch vehicle.
- Its very first test was on June 4<sup>th</sup>, 1996.
- 37 seconds after launch the rocket turned by 90 degrees incorrectly.
- The boosters were ripped apart and the vehicle self destructed.
- The loss was about \$500 million
- The reason was found to be software failure primarily due to forgetting to account for finite precision arithmetic.





So the example is that of Ariane 5, this is a European launch vehicle, it is the very 1<sup>st</sup> test in the Ariane 5 configuration, when there were of course Ariane... 1 to 4... was on June 4<sup>th</sup> 1996 okay so the launch seemed normal until the first 37 seconds okay. After that dramatically I would recommend that you take a look at the video on YouTube or something if you simply put Ariane 5 you know launch or something, you will get this video.

So if you take a look at what happens, at approximately 37 seconds after launch the rocket suddenly turned by 90 degrees incorrectly this was not planned of course. The boosters were ripped apart and the vehicle basically it had a self-destruction instructions sitting there and it self-destructed automatically so it is a giant loss approximately you know estimates vary between 350 million to 500 million US dollars, it is perhaps one of the most expensive problems that was caused due to software failure ok so simple software failure cause this. And what really happened if you dig into it was most of it was ignorance or not ignorance really people did not really adequately take care for the fact that we are doing finite precision arithmetic ok rather than real arithmetic in some sense okay, you will see how that happened little bit later.

(Refer Slide Time: 4:08)



### How a machine stores numbers

- Machines have a **finite** number of "bits"
  - Think of them as boxes where numbers are stored in binary (0 or 1)
- Computers usually use the binary (Base-2) system
- **Integer representation :**  
 $(10101101)_2 = 2^0 + 2^2 + 2^3 + 2^5 + 2^7 = (173)_{10}$   

- **Signed-integer representation :** Use a **sign bit** (1 for **negative**)
- **Example:** With 16-bits, **-173** will be represented as  
  
 $-173 = -(2^{15} - 1) = -32767$

This means there is a max and min number that can be represented on the computer

So let us look at this, machine has a finite number of bits okay, unlike you know how a human being writes, if you require more precision you simply keep on adding digits. A machine has finite number of predetermined number of digits ok, you can think of these digits you know whenever you store a number you can think of them as individual boxes and in each box either 0 or 1 will be stored. As you know for most part all of our arithmetic is done in binary basically to 0 or 1 system, every single thing is actually represented in terms of zeros and ones that is both the power as well as you can sometimes see there can be a problem.

So let us take a simple integer okay, so if you have an integer like 173 which is what we would call it in base 10, you can now write it in binary, you would have all done this in school, you will have this long representation because it can be written as 2 power 0, this is of course the representation for 2 power 0, 2 power 1 there is no representation + 2 power 2, 2 power 3, 2 power 4 has no representation, 2 power 5 + 2 power 7 can be written as 173 ok. So as far as the machine is concerned, it is going to look like stuff like this, so you are going to have about 8 digits here ok and you will have some 1s and some 0s and each box can either store a 0 or it can store a 1 okay.

Now suppose instead of 173 you have something like - 173, now what are you going to do? what we do in terms of representation in a machine is to use something called a sign bit ok, the sign bit will be another box upfront here, though we will see how we actually do it so this if it is 1, the machine will interpret it as negative, if it is 0 it will assume that the integer is

positive ok. Now I had 8 boxes here but let us say I have a 16-bit machine or a 16-bit representation, 16-bit simply means I have 16 boxes now ok.

So you will have something of this sort, remember the very 1<sup>st</sup> one the leftmost bit is what is called the sign bit, if it is 1 it basically means it is negative, the rest of it essentially represents the magnitude, in this case I have just copied this from there to here. So this number will be interpreted as -173, the - comes from here, 173 comes because all these are 0 and this is 173 ok. Now this has an implication, implication is that there is a maximum number that you can represent on the machine okay. That is because if you run out of digits or run out of boxes to store your number, you can no longer represent a large number, this is somewhat similar to calculators okay.

So if you have a calculator with 8 digits, you cannot store a number which is greater than 8 digits of course we will account for exponents a little bit later even in this video but the main point that I am going to make here and if you get nothing else in this particular video, please take away this one single point that there is a maximum number that the machine can store accurately and there is also a minimum number that the machine can store accurately ok. So in this case if you see the maximum minimum for 16-bit okay, remember one of the bits has been used here for sign so you have only 15 less so you can represent 2 power 15 - 1 which comes to + - 32000 something ok. So similarly if I increase from 16 bits to 32 bits, I will get to power 32 - 1 okay or 2 power 31 - 1 etc.

(Refer Slide Time: 8:35)

## Floating point representation

- Real numbers can also be represented in binary
  - e.g  $5.5 = 4 + 0 + 1 + 0.5 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) = (101.1)_2$
- The floating-point format is to use the scientific notation  $\pm S \times b^e$ 
  - S is the significand, b the base we are using, and e the exponent
  - Eg:  $0.001234 = 1.234 \times 10^{-3}$
- Binary floating point numbers are represented as  $\pm (1 + f) \times 2^e$ 
  - Ex:  $(5.5)_{10} = (101.1)_2 = 1.011 \times 2^{+2} = (1 + 0.011) \times 2^{+2}$
  - f is the mantissa and e the exponent
  - Each of f and e require separate bins to store
- For a 64-bit storage scheme

If you have understood that there is a min and max number on the computer you can skip this slide

Now, this was about integers, you have similar representation for floating point, once again you are free to skip these lines as long as you understand particularly that there is a minimum and maximum number on the computer, even a floating point number that you can use on the computer ok. So real numbers can also be represented in binary so let us say you have the number 5.5, it can be written as  $4 + 1 + 0.5$ , 0.5 of course is  $2^{-1}$ , so you will write it as so you notice this representation, 101.1 because after point what comes is the negative digits which is similar to how we deal with decimals because after the decimal whatever comes here suppose we have 0.3, this is 3 into  $10^{-1}$  in the base 10 representation, and one digit after that would be you know  $10^{-2}$ , similarly here too.

Now we have a more compact notation which we usually call the scientific notation even in calculators, so instead of simply writing it in terms of decimal places you can actually get more numbers if you present it this way,  $\pm$  some numbers times the base power and exponent okay. So S is what is called a significant which contains all the significant digits of the number, these the base we are using which is let us say if we are using base 10 it is 10, if it is 2 if it is a binary digit then it is base 2 and E is the exponent that we are using. So for example, if you have the number 0.001234, you would write it as  $1.234 \times 10^{-3}$ , where 1.234 is S, 10 is the base and exponent is - 3 okay.

Now it turns out that in binary you can get as I talked last time that if there is a maximum minimum number on the computer, you would like to increase this maximum minimum as much as possible. For binary number the first digit will always be 1 okay because if it is 0, we ignore it, we are only going to look at significant digits starting from 1 so we remove that one away and instead of S we write it as  $1 + F \times 2^E$  and this gives you a little bit of extra numbers to store okay. So for example, the same 5.5 if I write it as 101.1 in binary, this will be  $1.011 \times 2^{-2}$  and this can be written this should be  $2^2$  to the power + 2 I am sorry okay. So this is  $2^2$ , so this can be written as  $1 + F$ , this is the base and this is the exponent close 2.

F is called the mantissa and E is of course the exponent, now note that the numbers that we have given whether it is F or whether it is E now needs separate bins, so you have to store this 0.011, you also have to store these two into separate bins ok. So for a 64-bit storage scheme which is fairly standard for what is called double precision, we store digits this way. You keep one bit for the sign, you keep 11 bits for the sign exponent that is this E ok and you keep 52 bits for the mantissa which is for F okay. Now this is what is called an IEEE

standard, there is a standardised way of storing this, you know you can make other choices but this is the standard that people have agreed to on how to take 64 bits and store floating point numbers okay.

(Refer Slide Time: 12:36)

### Double precision – Range

- IEEE Double precision – 64 bits (8 bytes) are used for storing floating point numbers
  - Is the standard amount of precision used for much of scientific computation
  - MATLAB uses this by default
- Since there are only limited “boxes” for storing the exponent, there is a max/min positive number that can be represented  $2^{10} = 1024$ 
  - 11 bits for the signed exponent  $\Rightarrow$  Range of the exponent is from  $-1022$  to  $1023$
  - Largest number =  $1.111\dots111 \times 2^{+1023} \approx 1.8 \times 10^{308}$  NaN inf
  - Smallest number =  $1.000\dots000 \times 2^{-1022} \approx 2.2 \times 10^{-308}$

```

>> format long
>> realmax
ans =
1.797693134862316e+308 ✓
>> realmin
ans =
2.225073858507201e-308 ✓
            
```

MATLAB

So let us look at double precision, double precision is standard precision used for real number data, so if you use Matlab this is the default, in other cases let us say C, C++, etc. you have two options, there is something called float and there is something called double precision. Once again for most scientific computations we tend to use double precision to be as accurate as possible, you will find this within GPOs also single precision versus double precision ok. So remember that for 64-bit we had already seen the 1, 11, 52 split, this was for the sign bit, this is for exponent again signed exponent remember the exponent by itself can have signs okay and this is for the mantissa okay which was F.

Once again, since there are only limited boxes for storing the exponent, there is once again a maximum as well as a minimum positive number that can be represented ok, remember we have now 11 bits for the signed exponent so we have to remove one bit for the sign and you will get 2 power 10 is 1024, so you will have from 1023 to - 1022 that is the range within which you can represent the exponents, remember we are only talking about exponents in this particular video.

So the largest number that you can represent is let us say I have 52 digits here, so I take 1.1111 this is in binary and I can go to 2 power 1024 that is the maximum you can represent within double precision 64-bit okay. You go above this using double precision, any computer

that tries to use double precision will either give NAN which is called not a number or it will give INF which is infinity, so depending on what the compiler is like.

Similarly, you have a smallest number; this is once again in the smallest positive number, just above 0 what is the smallest number you can get? 1.000 into 2 to the power - 1022, so this is approximately 2.2 into 10 to the power -308 okay, so this seems like a very wide range but sometimes you can actually go beyond this very easily.

So I have flashed on the screen a simple example from Matlab, Matlab has a variable called Real max that tells you what the maximum number is, you can see this number here ok, approximately 1.8 into 10 to the power 308, this is the maximum number that Matlab can represent. Similarly, I have a minimum number which is approximately 2.2 into 10 to the power - 308, so this is from Matlab.

(Refer Slide Time: 15:51)

**Double precision - Precision**

Consider the following example

```

>> sqrt(2)
ans =
1.414213562373095
>> err = 1e-14
ans =
1.000000000000000e-14
>> sqrt(2) + err
ans =
1.414213562373095
>> sqrt(2) + err
ans =
1.414213562373095
>> sqrt(2) + err
ans =
1.414213562373095

```

Handwritten notes and diagrams:

- $1.00$ ,  $0.00$ ,  $1.00$  with a vertical line through  $0.00$  and an arrow pointing to "out of range".
- $1.00 \dots 0$  and  $0.00 \dots 0$  with a vertical line through the  $0.00$  and an arrow pointing to "out of range".
- $1.00 \dots 0 \times 2^E$  with a circled  $1.00$  and an arrow pointing to "Happen because even mantissa is limited to 52 bits".
- $2^{-52} \approx 2.22 \times 10^{-16}$  with a circled  $10^{-16}$  and an arrow pointing to "Double Precision (called Machine epsilon) is given by".
- $\epsilon$  (epsilon) with an arrow pointing to the machine epsilon value.
- MATLAB** written in red.

Code output for machine epsilon:

```

>> eps
ans =
2.220446049250313e-16

```

So just like range, you have a slightly different idea called precision, so let us 1<sup>st</sup> start with an example ok. So let us say I have square root 2 and once again I am writing this in Matlab, you will see some set of digits okay, you will see a standard set of numbers thrown here and let me add a certain amount of error, a small number to it okay, so this number here is 10 power - 14, I am adding this here, what you will see between here which is the original and this which is the case with the error, you will see all digits are the same except for this digit which was 0 here it became 1 here because I added 10 power - 14 ok.

So now let us say if I added instead of 10 power -14, suppose I add a 10 power -16, what is it that we would expect? So since this was 10 power -14, 10 power -16 is this so suppose I add



10 to the power -16, this 5 should actually turn to a 6 right, that is what we would expect so let us see what happens. So suppose I have 10 power -16, I add it to square root 2, surprisingly enough the 16<sup>th</sup> digit stays the same okay. Why did this happen? We can now look at another example, so let us say A is 1, B is - 1, this error once again I will call it error is 10 power -16. So suppose I do A + B + error, it gives me the right thing because A + B is 0, 0 + error is 10 power -16.

But suppose I change the order that B + error instead of that I write error + B, we know that addition is you know you have distributivity, you have associativity, commutativity, all those properties are there in this case, commutation between B + error and error + B, it should give you the same result but it gives you 0 ok which also seems to be happening here instead of adding error it is actually adding 0. Now why does this happen? The reason is, notice in both these cases even the mantissa, not just the exponent, even though the exponent allows you to go till 10 power -308, the mantissa is also limited.

The mantissa is limited up to 52 bits okay so the mantissa remember is 1 point something, the number of boxes I have to store here into total power e, E we saw in the previous slide now we are looking at this portion okay, what happens here. So double precision now is now given by 2 to the power -52 okay so that is the minimum that you can represent which turns out to be approximately 2.2 into 10 to the power -16. So any number below this will simply disappear so just to give you an example, now suppose you have a calculator and it has let us say it is a very bad calculator it has 3 digit only that it can represent on the screen so 0.00, now suppose I give you the number 0.001, there is no space for it to store it.

To give you another example, suppose I have 1.00 + 0.001, what will happen is I take 1.00, it has used up my 3 digits and if I give 0.001, this is out of range what it will do is it will simply give me 1.00, this cannot come down at all ok. Now how does that affect this? Notice that when I do A + B, A + B is already 0 so this is the order in which the machine will do the algorithm, it will do the addition, A + B is 0 + error it has enough space ok it has 16 digits for you to be seeing 10 to the power - 16, however when I do A + error + B, something similar happens, I have 1.000 16 digits + 0.0001 no place to add it, so it basically sees this as A + error as simply 1 and B is - 1 which is why it gives 0 ok.

So if you go to Matlab once again, the smallest number the restriction that is given by the mantissa is called Machine Epsilon okay. So if you simply put EPS in Matlab you will get this value as you can see it in approximately 2.22 into 10 to the power -16 okay, so this is the

smallest number that the machine can represent in terms of floating point additions and subtractions okay.

(Refer Slide Time: 21:02)

### Underflow and overflow

- Underflow happens due to numbers near zero being "rounded off" to zero
 

```

          >> err = 1.e-16
          err =
          1.0000000000000000e-16
          >> sqrt(2) + err
          ans =
          1.414213562373095
          
```

Optim. #2  
e<sup>5000</sup>
- Underflow might result in divide by zero errors even when the denominator is finite
- Overflow happens when the number being computed overshoots the max possible

Consider  $\frac{\exp(x_1)}{\exp(x_1) + \exp(x_2)}$


>> x = [5000 5000];  
 >> soft = exp(x(1)) / (exp(x(1)) + exp(x(2)))

soft = NaN

>> z = x - max(x);  
 >> soft = exp(z(1)) / (exp(z(1)) + exp(z(2)))

soft = 0.5000000000000000

Can be reformulated to account for overflow



The Ariane 5 disaster was due to an overflow problem

Now we come, because of the combination of these 2, because of the combination of the range and the precision, the main thing once again main takeaway is that there are smallest and biggest numbers that the machine can accurately add and subtract okay. Now you have 2 types of errors, so an underflow error is what happens in case numbers near 0 are rounded off to 0 ok, so this is the same kind of example that we saw last time, you had 10 to the power -16 when you add it to square root 2 nothing really happens here so effectively 10 to the power -16 is being rounded off to 0 okay so you are not getting any addition here.

So here is a simple figure to represent this so let us say this is the max positive number that you can represent and this is the mean positive number, this is the negative limit, when I call it negative max what I obviously mean is maximum in terms of absolute value. So whenever you are kind of caught between these 2 limits okay so let us say -10 to the power -16 and 10 to the power -16, it is called underflow error, in some sense you can see this is going below the least count of the machine okay so just like our scale has a least count most scales like 1 MM below that you cannot measure accurately so similarly, below 10 power -16 for double precision you will have trouble okay so if we have numbers going below that and you do not account for them in terms of the exponent and this Separately you are going to have trouble.

Another thing that can happen in terms of underflow is you might have a divide by 0 error okay, so even though your denominator is not really 0 but if it goes below your machine

Epsilon or you know even your minimum  $10^{-308}$ , you can actually have divide by 0 error, it can occur in many different ways ok. Overflow happens when you actually go above the maximum limit okay, so let us see an example, so let us say you have a simple expression, we will see that this is a special case of something called soft max as we move into the neural network portion, but let us say you have a simple function 
$$E^{x_1} \text{ by } E^{x_1 + E^{x_2}}$$
 okay.

Now let us say  $x_1 = x_2$ , in such a case it simply give you half okay. Since  $x_1 = x_2$  you simply have to  $E^{x_1} \text{ by } E^{x_1 + E^{x_1}}$ , so this is obviously half. So let us try this in Matlab, let us say I take a vector, this vector now is 5000 and 5000 these are just 2 numbers,  $x_1 = 5000$ ,  $x_2 = 5000$  and I tried this expression, I do  $E^{x_1} \text{ by } E^{x_1 + E^{x_2}}$ , I get not a number. Now why is that because  $E^{5000}$  has exceeded your maximum possible?

So even though the calculation is badly simple you can do it by hand this is what I meant in the initial side of this video which is that there are certain things that you can do by hands very easily but the machine being dumb and being doing sequential operations will simply do  $E^{500}$  first and it will say well I cannot store it so this is not a number. It turns out that there are ways of tricking the machine into doing the right thing okay so I will just show one example here, so instead of doing your calculations in terms of  $x$ , we subtract out okay so  $Z$  is  $x - \text{maximum of } x$  okay, or  $Z_i$  is  $x_i - \text{over all } i x_i$  ok. So if you subtract that thing out it turns out that this function does not change because you are simply multiplying by  $E^{-\max x}$  on the numerator and the denominator.

Now if I write it that way and I do  $E^{Z_1} \text{ by } E^{Z_1 + Z_2}$ , I get back the right result ok. So the point is, if you simply wrote this in your code in your program, if you are lucky nothing would happen, if you are unlucky you might get not a number even though you might be confused about where this not a number came from. So the fact that the machine has a maximum and minimum can cause surprising errors okay, you might not have a formula problem, you might not have a compilation problem but you could have a overflow and underflow problem because you have not accounted for the way numbers are registered. In fact, Open AI one of the companies that works on AI is now trying to exploit the fact that there is finite precision in order to come up with some machine learning algorithms so that is well beyond the scope of this course but I just wanted to point that out ok.

(Refer Slide Time: 26:40)

The Ariane 5 disaster

- The internal Inquiry board reported that this was due to software error in the Inertial Reference System (SRI -- Système de Référence Inertielle)
- The SRI used a floating point variable BH to determine the orientation of the rocket.
- This variable was stored as a 64-bit floating point
  - It had to be converted to a 16-bit signed integer
- This caused no problems for the previous version (Ariane 4) as the values were below the max
- After 37s of flight in Ariane 5, these values were exceeded
  - Accelerations were higher in Ariane 5 than Ariane 4
- In the words of the report –
  - The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an operand Error.

NPTEL

overflow

overflow

overflow

Now it turns out that the Ariane 5 disaster which I started this video which was also due to a overflow problem okay. So remember that it was all fine in the beginning and the internal enquiry board, the report is actually available online, I have put a reference to that somewhere later here ok, so since it is a French system this I am going to call it SRI, it is actually inertial reference system so the inertial reference system tells you **you know** which way the rocket is pointing very-very roughly, so it **it** had a variable just like we had the variable in the previous slide, it had a variable called BH which is actually used to determine the orientation of the rocket, is it pointing up or is it pointing down, etc and this orientation was represented by a floating point variable, a real number okay.

Now this variable was stored in a 64-bit floating point operation okay, but due to several internal reasons part of the reason being that the previous one Ariane 4 used some 16-bit integers so what it had to do was it had to turn this BH from a 64-bit floating point number into a 16-bit signed integer. Now this was not any problem for the previous version, I have this is a mistake, this should be Ariane 4 so the previous version of this launch the vehicle was Ariane 4, it was not a problem because all the numbers for the orientation were well within orientation speed, etc, were well within the limits of a 16-bit integer.

However, after 37 seconds okay it basically reached overflow okay so some number within the calculation actually went beyond the 16-bit limit, so 16-bit non-signed is 65,000, signed is 32,000 so in either case this number was exceeded due to the vast acceleration that was there in Ariane 5 in comparison to Ariane 4. So some numbers were exceeded and you can see now

because of that it essentially got confused so instead of going straight up, the orientation was miss read, it actually turned and then the self-destruct mechanism to cover okay. So in the words of the report, so the report says the internal inertial reference system software was cause due to the conversion from 64-bit to 16-bit signed integer value ok so as I said here this had a value greater then what would be represented so this is classic overflow.

So the overflow caused the Ariane 5 disaster, this is to tell you that though in the example that I gave it seem like extreme examples, it can actually have very-very real-life effects okay, so similar problems have happened in other cases during Gulf war, etc. So the fact that the machine is representing numbers in a finite precision have to be sometimes accounted for okay, so if you are lucky it will almost never happen but if you have a completely unexplainable phenomena happening to you where everything seems to work on paper and it seems to track maybe sometimes it could be an underflow or overflow error, thank you.

(Refer Slide Time: 30:21)

### Condition number

- Limited precision has many unexpected results
 

```

a=0;
for i = 1:10000 >> PrecisionTest
    a = a+0.0001;
end
disp(a)

```

Precision effects propagate

0.0001 = 10<sup>-4</sup>

b = Ax → A = b/x → x = A<sup>-1</sup>b

x<sub>i</sub> = A<sup>-1</sup>b<sub>i</sub>

y = A/x
- This can be lead to problems when systems of linear equations are solved
 


```

A =
1.000000000000000 2.000000000000000
2.000000000000000 4.000000000000000
x =
1
-1
>> b = A*x
b =
-1.000000000000000
-2.000000000000000
>> im(A\b)
ans =
0.999999999995000
-0.9999999999975000

```
- Some matrices are close to singular
- Small errors can be magnified by them
- This is measured by condition number
- In general  $cond(A) = \|A\| \|A^{-1}\|$
- For symmetric matrices, condition number is the ratio of the largest to smallest eigenvalue
- $cond(A) = \max_{i,j} \frac{|a_{ii}|}{|a_{jj}|}$ . Higher condition number means poorly conditioned

Small change in b<sub>i</sub> → b<sub>i</sub> = b<sub>i</sub> + Δb<sub>i</sub>

(ill conditioned) matrix



So the last topic in this video is that of condition number okay, so the fact that you have limited precision which is what we have been looking at in the previous slides can have many unexpected results, some of them you have already saw. So let us take a simple case, we are simply summing up s or this number 0.0001 and we are assuming it up 10,000 times okay, so what would you expect? 0.0001 multiplied by 10,000 should be one okay. If you actually execute this program you will find that it is not quite 1, notice that there is some error in the last 2 places.

Now why does this happen? This is because precision has effects that propagate okay, what do I mean by that? Remember we are adding  $0.0001$ , this is  $10^{-4}$ , this does not have an exact representation in binary, it has a repeating decimal representation in binary so that the last digit when it gets chopped off has an actual effect okay. So in that case as you add this problem in  $10^{-4}$  in the 16 digit many-many times the effect actually starts from your end starts leaking upwards towards the left okay. So this effect of additive effect of precision can be particularly bad okay, if you have multiple calculations so I will show you one example.

Let us say you are solving a system of linear equations okay, so let us give you an example, so this is the system  $Ax = b$ , let us say this is the matrix  $A$ ,  $\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$  there is a  $1$  okay sitting somewhere in  $A$  ok. So let us say my  $X$  is this;  $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$  fairly simple example, suppose I define  $d = K \text{ times } X$ , this of course means that  $X = A^{-1}b$  so if I do,  $X = A^{-1}B$ , I should recover  $X$ , I do not quite recover  $X$  you can see that instead of  $1$  the affect propagated a little about by 5 digits. Similarly, instead of getting  $-1$  I got some error propagation ok however, something even more serious can happen okay now because typically you solve  $X = A^{-1}B$ , let us say I introduce an error in  $B$  ok.

Now instead of  $B$  being this suppose I added  $0.01$  okay or subtracted  $0.01$ , so you can see that now between this and this I have made a small difference, small change in  $B$  so  $B$  goes to  $B + \Delta B$  or  $B - 1$  is  $B + \Delta B$ . Now the question is if I change  $B$  to  $B + \Delta B$ ,  $A$  remains the same, what is the change in  $X$ ? That is if my number remembers since I am doing is finite precision arithmetic you saw earlier that some numbers might not be exactly represented okay. So if I make a small change in a number instead of storing  $1$ , I store  $1.00001$  how much of a change will it make while solving linear systems of equations? Okay, so when you do this if I do  $X = A^{-1}B$ , what I would expect is only a small change in  $X$  since I have made only a small change in  $B$  but you can see this is a huge change okay.

Now from being  $1 - 1$  it has actually turned into  $10^8$ , the sign has changed  $-10^8$  and  $1$  into  $10^8$ . So just a small change of  $0.01$  in  $B$  has caused the change of  $10^8$  in  $X$  so this is quite worrying, so this is why we look at what is the nature of this matrix  $A$  ok. So just like when you have division by numbers, so suppose I have  $Y = A/X$  and if  $X$  is very-very small then small changes in  $A$  can cause large changes in  $Y$  similarly, if  $A$  is close to singular, a small change in  $B$  can actually be magnified by  $A$  okay.

So this is measured by something called the condition number, condition number is defined as norm of  $A$  remember you can go back to our norm videos, norm of  $A$  is some measure of  $A$  multiplied by norm of  $A$  inverse is given as condition number. For symmetric matrices there is an easy way of measuring this condition number okay whiches you find out the ratio the maximum ratio of eigenvalues, which is find out the maximum eigenvalue in magnitude and divide by the minimum eigenvalue magnitude and that tells you roughly how much your answer is going to be banking side.

So for example, we can see that 2 decimal places were increased okay, so this **this** was increased to  $10^8$  which means there is an increase in 10 decimal places so basically are magnifying an error by a factor of  $10^{10}$  and if you look at the condition number of this matrix just to clarify this. So if you see condition number this is  $10^{11}$  and it tells you very roughly this is not very precise, it tells you very roughly that your answers are going to be magnified by a factor of  $10^{11}$  that is the worst-case scenario and we are getting close to the worst-case scenario here ok.

So in general if you have a high condition number, this means you have a poorly conditioned matrix and in certain software for example, Matlab will tell you, you will have an ill condition matrix. Ill conditioned means the condition number is really high and any small errors may be magnified very-very largely okay.

So in this video, just a recapitulation we looked at a few ideas, a few implications of the fact that your numbers are not exactly represented as you might think that there is a finite amount of precision and that finite amount of precision has a minimum limit and maximum limit and sometimes this imprecision can multiply upon themselves and lead to large really poor effects, thank you.