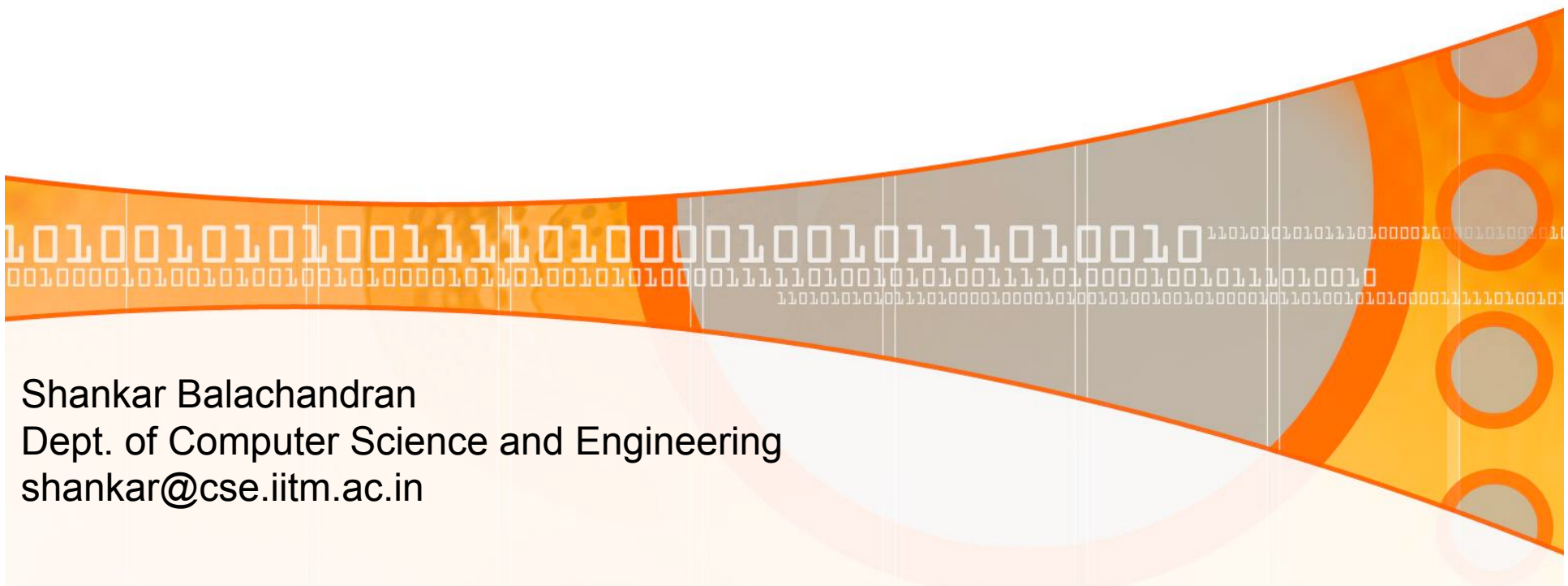


Computational Engineering

CS 110

Precision, Accuracy, Floating Points



Shankar Balachandran
Dept. of Computer Science and Engineering
shankar@cse.iitm.ac.in

Reference

- Computer Organization by Hennessy and Patterson
- IEEE 754
- <http://devshed.com>
- Wikipedia for some images
- <http://www.ima.umn.edu/~arnold/455.f96/disasters.html>



Accuracy vs Precision

- Accuracy and Precision
 - Two important engineering metrics
- Accuracy :
 - how close is the measured value to the true value?
- Precision :
 - how close do repeated experiments yield similar results.
- Four combinations of results possible
 - Accurate and precise
 - Accurate but imprecise
 - ...



Analogy : Shooting Darts at a Target

Goal : Hit the Bull's Eye



Accurate but Imprecise



Inaccurate but Precise



Floating Point Representation

- What can be represented in N bits?

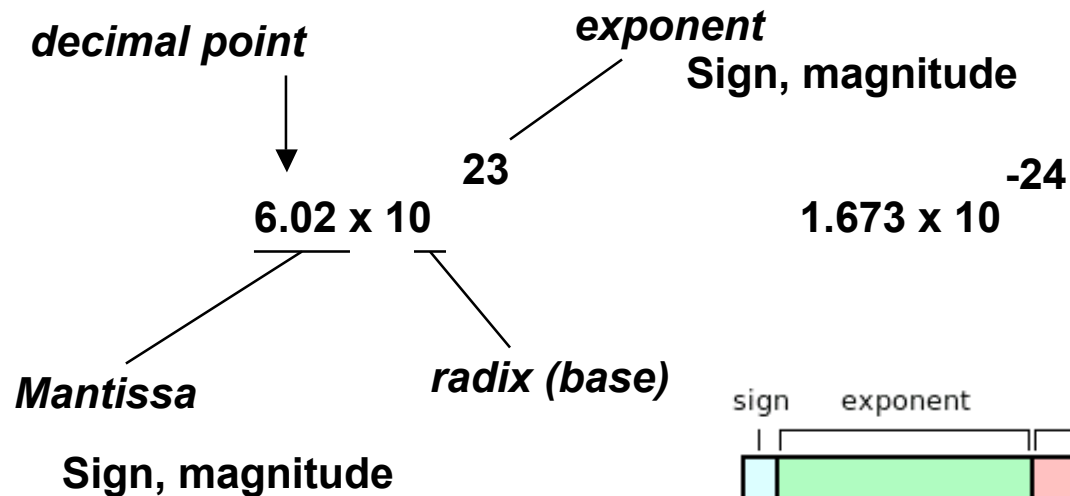
- Unsigned 0 to 2^N
- 2s Complement -2^{N-1} to $2^{N-1} - 1$
- 1s Complement $-2^{N-1} + 1$ to $2^{N-1} - 1$

- But, what about?

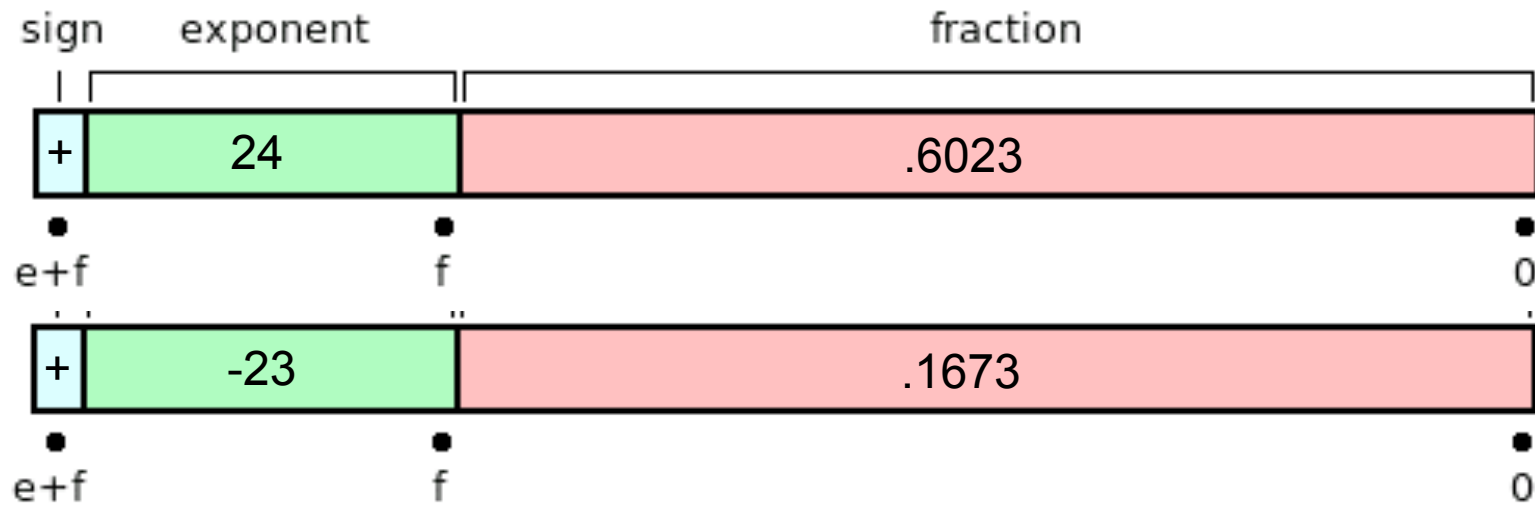
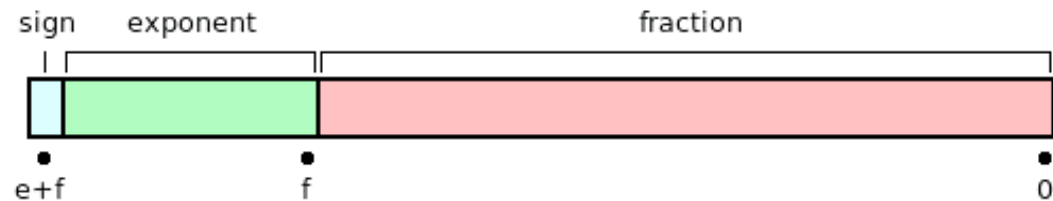
- very large numbers? 9,349,398,989,787,762,244,859,087,678
- very small number? 0.0000000000000000000000000045691
- rationals $2/3$
- Irrationals, transcendentals ...



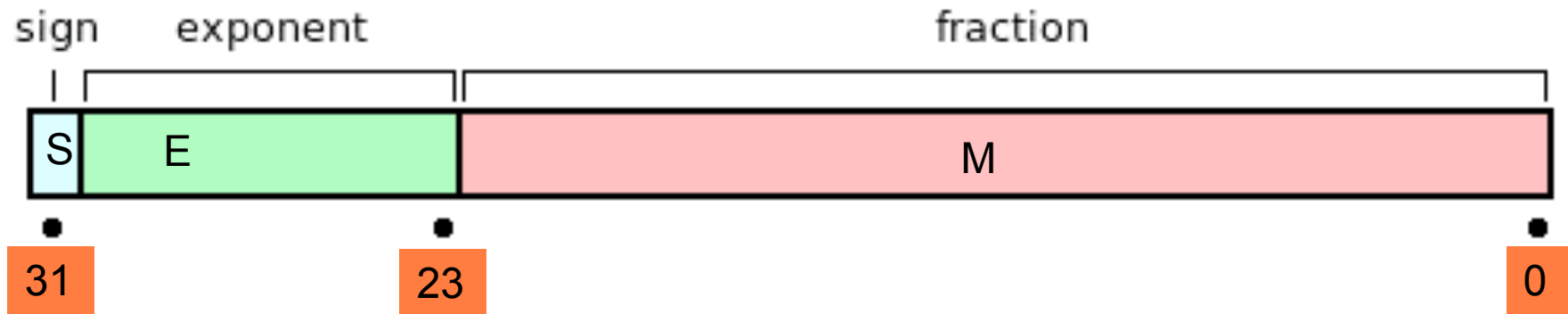
Scientific Notation



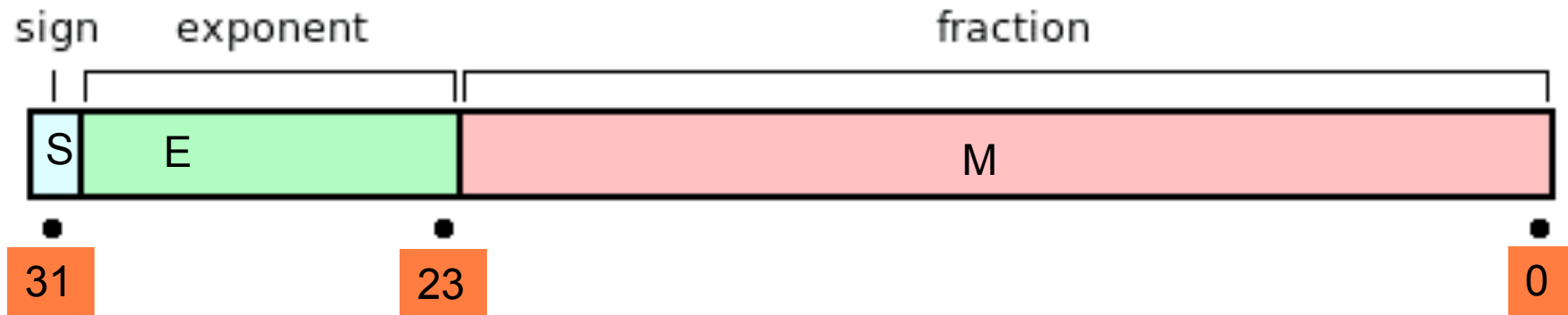
General
Layout :



IEEE 754 Floating Point Format



IEEE 754



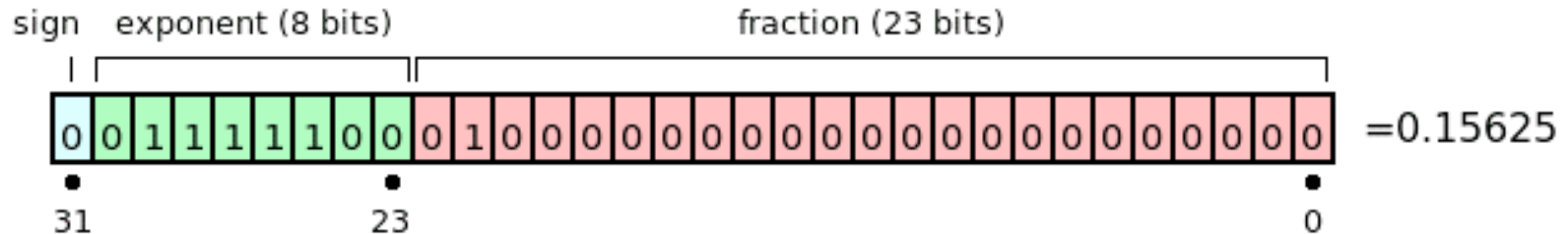
- S = 0 if +ve 1 if -ve
- E : Excess 127 Exponent
 - Actual exponent $e = E - 127$
- M : Mantissa
 - Magnitude, Normalized
 - Hidden integer bit 1
 - Actual fraction $1.M$ in base 2

Number

$$N = (-1)^S * 2^{(E-127)} * 1.M$$



Example 1

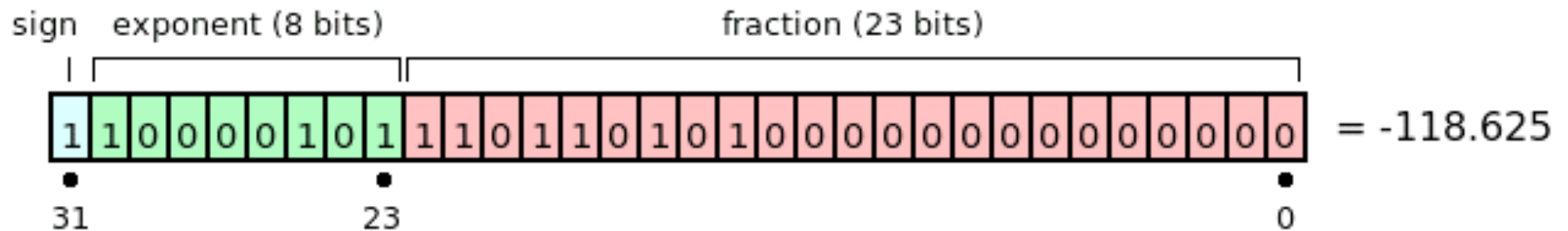


$$S = 0 \quad E = 124 \quad M = (1.010000000000\dots0)_2$$

$$\begin{aligned} N &= (-1)^S * 2^{(E-127)} * (1.M)_2 \\ &= (-1)^0 * 2^{(124-127)} * (1.01)_2 \\ &= 2^{-3} * 1.25 \\ &= 1.25 / 8 = 0.15625 \end{aligned}$$



Example 2



$$S = 1 \quad E = 133 \quad M = (1.1101101010\dots0)_2$$

$$N = (-1)^S * 2^{(E-127)} * 1.M$$

$$= (-1)^1 * 2^{(133-127)} * (1.110110101000000)_2$$

$$= -2^6 * (1.110110101)_2$$

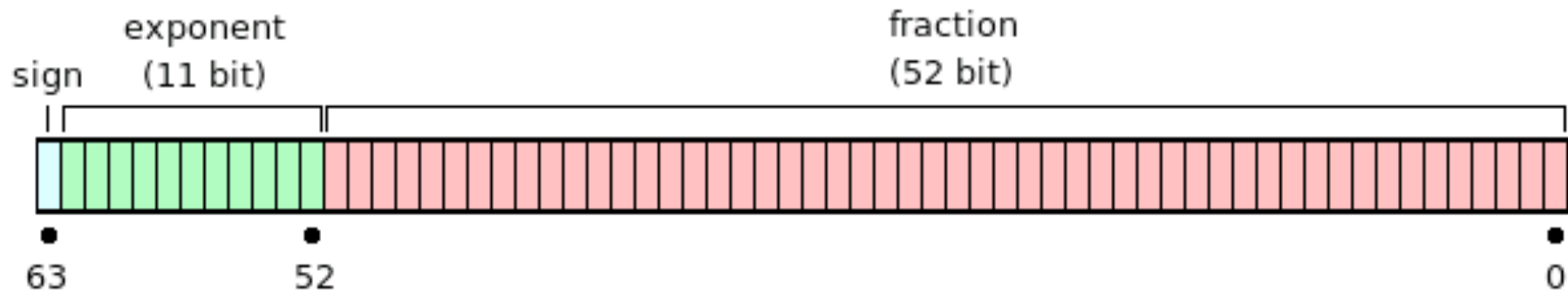
$$= -(1110110.101)_2$$

$$= -118.625$$



Float and Double

- *float* : Single precision
 - $\pm 1.175494351 \times 10^{-38}$ to $\pm 3.4028235 \times 10^{38}$
- *double* : Double precision



— $\pm 2.2250738585072020 \times 10^{-308}$ to $\pm 1.7976931348623157 \times 10^{308}$



Some Special Values

Type	Exponent	Significand	Value
Zero	0000 0000	000 0000 0000 0000 0000 0000	0.0
One	0111 1111	000 0000 0000 0000 0000 0000	1.0
Small denormalized number	0000 0000	000 0000 0000 0000 0000 0001	1.4×10^{-45}
Large denormalized number	0000 0000	111 1111 1111 1111 1111 1111	1.18×10^{-38}
Large normalized number	1111 1110	111 1111 1111 1111 1111 1111	3.4×10^{38}
Small normalized number	0000 0001	000 0000 0000 0000 0000 0000	1.18×10^{-38}
Infinity	1111 1111	000 0000 0000 0000 0000 0000	Infinity
NaN	1111 1111	non zero	NaN



Floating Point Calculations

- Needs a lot of care
- Picking single precision vs. double
- Scanning and printing floating point values
- Ranges
- Accuracy and precision
- We will see some gotchas 😊



How many numbers can we represent?

- 32 bits
 - 2^{32} values
- How many floating point numbers are really there?
 - Uncountable (Cantor's argument)
- There are uncountable floating point values
 - Between 0 and 1
 - Between 0 and 0.1
 - Between 0 and 0.01
 - Or in general, between any number and any other number
- Clearly, our representation is finite in size
 - Hence, not all numbers can be represented faithfully



Exercise 1

```
int main( )
{
    float a=1.0;
    long i;

    for(i=0; i<100; i++){
        a = a - 0.01;
    }
    printf("%f\n",a);
}
```

- What must a be at the end of the loop?
- Expected output
 - 0
- Printed output
 - 0.000001
- Actual value
 - 6.591528745047981e-7



What really happened?

- Actual value : $6.591528745047981e-7$
 - Error creeps in during subtraction
 - Unable to represent the intermediate values faithfully
 - Error propagates
- But why print as 0.000001?
 - Printing rounds off
 - $0.00000065915\dots = 0.000001$



Exercise 2

```
int main( )
{
    float a=1.0;
    long i;
    for(i=0; i<100; i++){
        a = a - 0.01;
    }
    for(i=0; i<100; i++){
        a = a + 0.01;
    }
    printf("%f\n",a);
}
```

- What must a be at the end of the loop?
- Expected output
 - Some $1+\Delta$
- Printed output
 - 1
- Actual value
 - 1



What really happened?

- Actual value : 1
 - Error creeps in during addition now as it was in subtraction
 - Error propagates as before
 - One error cancels the other
- But why print as 1?
 - Printed faithfully, d'oh



Exercise 3

- Let's repeat the previous experiment with subtractions of 10^{-3} , 10^{-4} , 10^{-5} etc. and correspondingly 10^3 , 10^4 , 10^5 loops

	Actual	Printed	Actual	Printed
10^{-3}	9.325e-6	0.0000009	1	1
10^{-4}	-5.358e-5	-0.0000054	1	1
10^{-5}	-9.888e-4	-0.000989	1	1
10^{-6}	-9.477e-3	-0.009478	1	1
10^{-7}	-5.453e-2	-0.054531	1	1



Observations and Inferences

- Smaller the value that is subtracted, more the error.
- Error grows exponentially
 - Watch out
- Values are faithfully restored in every single case
 - Systematic error
 - Does it mean predictable??



Exercise 4 : With 10^{-8}

- Printed Value
 - 1 after subtraction
 - 1 after addition also
- The very first subtraction and addition has no effect on a
 - Repeatedly no effect at all
- Exercise 5 : Try the experiments with $a = 0$, be ready for a surprise
 - Especially for 10^{-8}



Exercise 6 : Redo Ex 1 to Ex4 with *double*

```
int main( )
{
    double a=1.0;
    long i;

    for(i=0; i<100; i++){
        a = a - 0.01;
    }
    printf("%g\n",a);
}
```

- What must a be at the end of the loop?
- Expected output
 - 0
- Printed output
 - 7.5287e-16
- Actual value
 - 7.5286998857393e-16



What happened now?

- Actual value
 - Still erroneous, but far lesser
 - Can still be problematic
 - What if the error is accumulated 10^{16} times
- Printed value
 - %g for double
 - Automatically prints a wider range
 - Some digits rounded off in printing
 - Exponent gives us a ballpark estimate



Exercise 7

- Let's repeat the previous experiment with subtractions of 10^{-3} , 10^{-4} , 10^{-5} etc. and correspondingly 10^3 , 10^4 , 10^5 loops

	Actual	Printed
10^{-3}	Same as printed	-8.8124e-16
10^{-4}	”	9.38176e-14
10^{-5}	”	1.91625e-12
10^{-6}	”	-7.9183e-12
10^{-7}	”	2.4983e-10
10^{-8}	”	-2.28987e-09



Observations and Inferences

- Smaller the value that is subtracted, more the error
 - Even now
- Error grows exponentially
 - You can run, but you can't hide
 - Probably insignificant ?
 - Depends on applications
- Values are still faithfully restored in every single case



A new set of experiments; Exercise 8

```
int main( )
{
    float b=1.0;
    long i=0;
    while (b > 0.0){
        b -= 0.01;
        i++;
    }
    printf("%ld\n",i);
}
```

- What must i be at the end of the loop?
- Expected output
 - 100
- Printed output
 - 101



Exercise 9

- Let's repeat the Exercise 8 with subtractions of 10^{-3} , 10^{-4} , 10^{-5} etc.

	i
10^{-3}	1001
10^{-4}	10000
10^{-5}	99902
10^{-6}	990522
10^{-7}	9456691
10^{-8}	Guess



Exercise 10 : Using *double*

10^{-2}	100
10^{-3}	1000
10^{-4}	10001
10^{-5}	100001
10^{-6}	1000000
10^{-7}	10000001
10^{-8}	100000000



Some Disasters

- Patriot Missile Failure : 1991
 - Time measured in units of 10^{th} of a second
 - Chopped off at 24 bits
 - For hours, the error was 0.34 seconds
 - Scud travels at 1.6 km/s
 - Traveled more than half a kilometer away
- Ariane 5 rocket launched by ESA in 1996
 - Exploded 40 seconds after lift-off
 - \$500 million in damage
 - 64 bit float assigned to integer and the integer saturated



Summary

- Floating Point calculations can be tricky
- Extensive checks required
 - Can baffle even the most experienced programmers
- IEEE 754 has its inherent problems
- You have to write your own libraries for handling more precision
 - And should carefully debug them



Summary of Class

- You guys and girls have been a joy to teach
- Lots of talent, lots of ideas
- Keep up your good work and spirits
- I enjoyed teaching your class a lot
 - And learnt a lot as well
- Thank You

