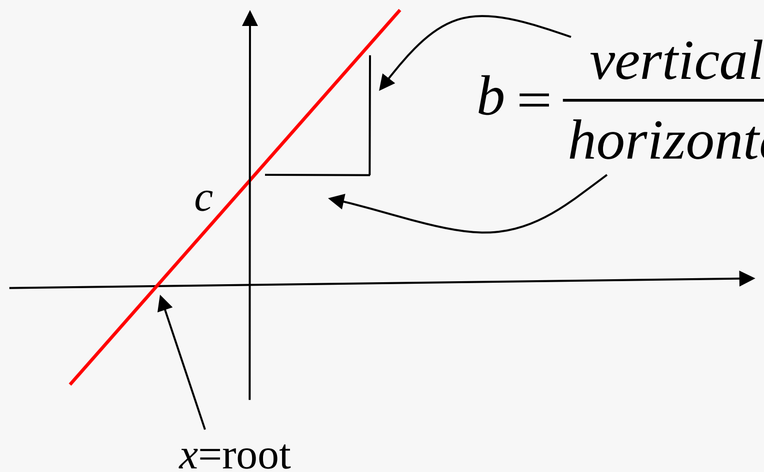
Solving non-linear equations of a single variable – the use of floating point

Linear ...

A linear equation is one where the highest power is 1

e.g. $bx + c = 0$ where ***b*** and ***c*** are numbers (-2, 12.5, etc), sometimes they are called ***coefficients***. The ***x*** is what we have to find out so it is unknown, this is a ***variable***.



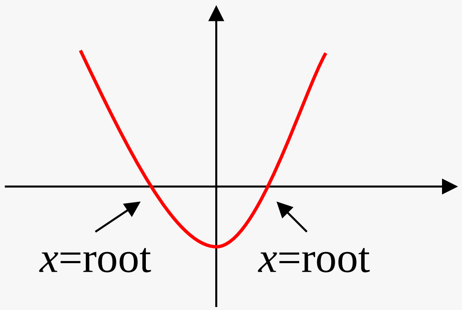
$$b = \frac{\text{vertical height}}{\text{horizontal width}}$$

In this special case c is the value where the linear equation crosses the y-axis, b is the gradient and $x = -\frac{c}{b}$

... and nonlinear

A nonlinear equation is one where there is at least one power greater than 1

e.g. $ax^2 + bx + c = 0$ where ***a***, ***b*** and ***c*** are numbers (-2, 12.5, etc) just like in the linear case. The ***x*** is what we have to find out so it is unknown, again this is the ***variable***.



$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In this special case we can find the two values using a formula

... and nonlinear

Example: Find the roots of the quadratic $x^2 + 5x + 6 = 0$

Here $a=1$, $b=5$ and $c=6$.

If we use our quadratic formula $x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ we get

$$x = \frac{-5 \pm \sqrt{5^2 - 4\text{x}1\text{x}6}}{2\text{x}1}$$

$$x = \frac{-5 \pm \sqrt{25 - 24}}{2}$$

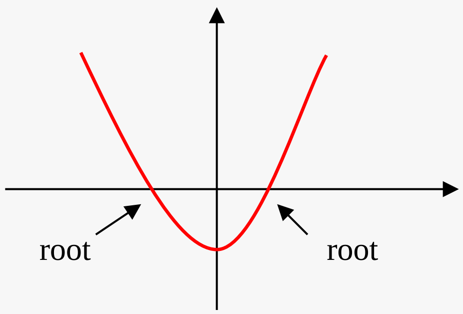
$x = \dfrac{-5 \pm 1}{2}$ this is the same as $x = \dfrac{-5+1}{2}$ and $x = \dfrac{-5-1}{2}$

$x = \dfrac{-5+1}{2}$ *and* $\dfrac{-5-1}{2}$ *so* $x = -2$ *and* $x = -3$

Example:Roots of a quadratic

problem:

Find the roots of a quadratic equation. The quadratic is of the form $ax^2 + bx + c = 0$ where a,b,c are constants



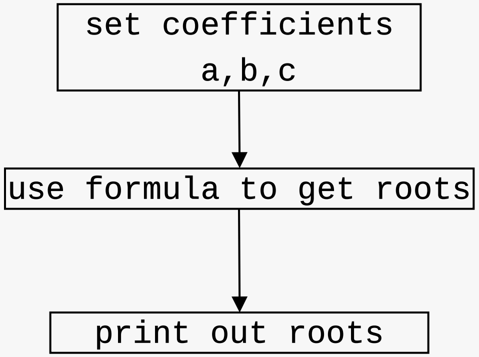
breakdown into logical steps:

1. Set up initial values for a,b,c
2. Calculate roots from formula $x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
3. Print out roots

Example:Roots of a quadratic

```
/*
 * Program to find the roots of a quadratic
 * assumes that they are real and distinct
 */
class quadraticroots
{
public static void main(String[] args)
   {
   double a,b,c,discriminant,root1,root2;
   a=1.0;
   b=-4.0;
   c=3.0;
   discriminant=b*b-4*a*c;
   /* calculate roots */
   root1=(-b+Math.sqrt(discriminant))/(2*a);
   root2=(-b-Math.sqrt(discriminant))/(2*a);
   System.out.println("roots are "+root1+" and "+root2);

   }
}
```



Example:Roots of a quadratic

STOP !!!!!!!

$\sqrt{b^2 - 4ac}$ is called the discriminant and is interesting why?

what happens if

$$b^2 - 4ac > 0$$

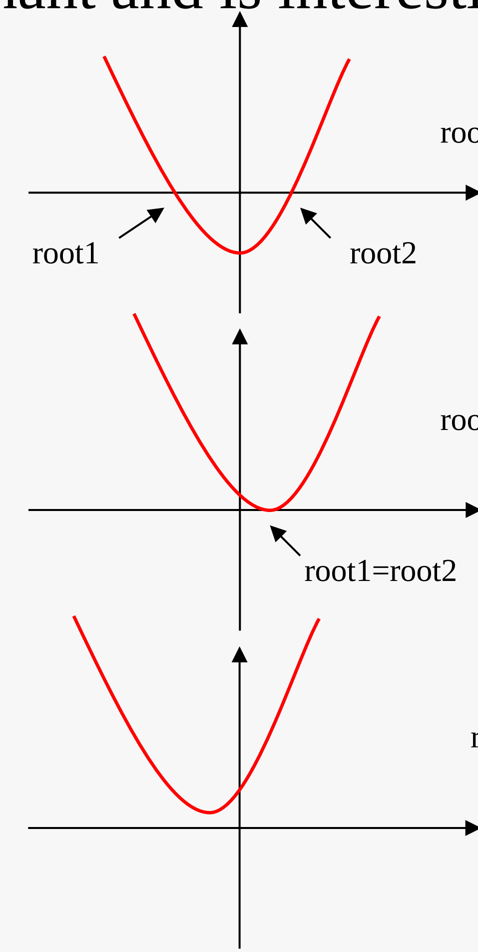
root1 and root2 are real and distinct

$$b^2 - 4ac = 0$$

root1 and root2 are real and the same

$$b^2 - 4ac < 0$$

root1 and root2 are complex



Functional iteration

- Suppose we wish to find the root of the equation

$$x - \cos x = 0$$

- If we draw a rough sketch it is easy to see that there is a root between 0 and 1.
- The equation may be written in the form: $x = \cos x$

- We can try and solve the equation by generating a sequence of approximations $x_0, x_1, x_2, x_3, \ldots$ using the following ***functional iteration***

$$x_{i+1} = \cos x_i \qquad for\ i = 0, 1, 2, 3, \ldots$$

Functional iteration

- Using a calculator we obtain the following results

$x_0 = 1.0$
$x_1 = cos(x_0)$
$x_2 = cos(x_1)$
$x_3 = cos(x_2)$
…

$x_0 = 1.0$
$x_1 = 0.5403023$
$x_2 = 0.8575532$
$x_3 = 0.6542898$
…
$x_{21} =$ **0.7**390182
$x_{22} =$ **0.739**1301
$x_{23} =$ **0.739**0547
…
$x_{38} =$ **0.7390**852
$x_{39} =$ **0.739085**0
$x_{40} =$ **0.739085**1
$x_{41} =$ **0.7390851**

- We can see that the root of the equation is 0.7390851
- We have used an ***iterative process***
- ***Successive approximations*** get closer
- The process is said to ***converge***

Functional iteration

Problem

Write a program to solve for the root of the equation $x - \cos x = 0$

Breakdown into logical steps

1. Start with an initial guess `xold`
2. Calculate new approximation `xnew = cos(xold)`
3. If the two successive approximations `xold` and `xnew` are close

    enough then print solution

   else

    save `xold` as `xnew` and return to step 2

Functional iteration

```
/*
 * Program to find the root of x-cos(x)=0 accurate to 6
 * decimal places so that abs(xnew-xold) < 0.000001
 * using functional iteration
 */
class functionaliteration
{
    public static void main(String[] args)
        {
        double xold,xnew,diff;
        int iteration;
        xold=0.0;
        iteration=0;
        do
            {
            iteration = iteration + 1;
            xnew = Math.cos(xold);
            System.out.println(“Approx for iteration”+iteration+”  is "+xnew);
            diff = Math.abs(xnew-xold);
            xold = xnew;
            }
        while (diff > 0.000001);
        System.out.println("root to six decimal places is “+xnew);
         }
}
```

Bisection method

Problem

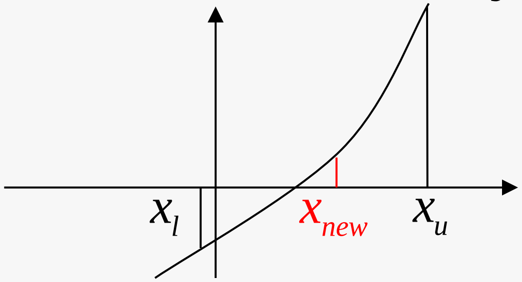
Solve for the root of the equation $x - \cos x = 0$ using the Bisection method

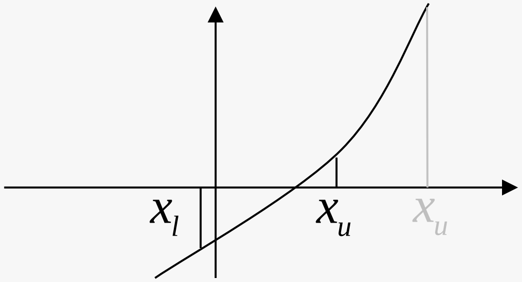
Bisection algorithm

Starting with a lower x_l and upper x_u bounds that cover the root - this means that one functional value is positive and the other is negative

The next approximation is the average of x_l and x_u

The approximation discarded is the one with the same functional sign but is further away from the root





Bisection method

Breakdown into logical steps

1. Start with initial bounds `xlower` and `xupper` then calculate `fxlower` and `fxupper`
2. Calculate new approximation `xnew=(xupper+xlower)/2` and function value `fxnew`
3. If `fxnew` and `fxlower` have the same sign then set `fxnew=fxlower`
4. If `fxnew` and `fxupper` have the same sign then set `fxnew=fxupper`
5. If the average of approximations `xlower` and `xupper` are close

   enough then print solution

   else

   return to step 2

Bisection method

```
/* Program to find the root of x-cos(x)=0 accurate to 6 decimal places so
 * that abs(xupper-xlower)/2 < 0.000001 using Bisection method. Assumes that
 * f(xlower) and f(xupper) have different signs to start with.
 */
class Bisection
{
    public static void main(String[] args)
        {
        double xlower,xupper,xnew,fxlower,fxupper,fxnew,diff;
        int iteration;
        xlower=0.0;
        fxlower = xlower - Math.cos(xlower);
        xupper=1.0;
        fxupper = xupper - Math.cos(xupper);
        iteration=0;
        do
            {
            iteration = iteration + 1;
            // determine xnew and f (xnew)
            xnew = (xlower+xupper)/2.0;
            fxnew = xnew - Math.cos(xnew);
            System.out.println("Approx for iteration"+iteration+"  is "+xnew);
            diff = Math.abs(xupper-xlower)/2;
            if (fxlower*fxnew > 0)
               {
               xlower = xnew;
               fxlower = fxnew;
               }
            else if (fxupper*fxnew > 0)
               {
               xupper = xnew;
               fxupper = fxnew;
               }
            }
        while (diff > 0.000001);
        System.out.println("root to six decimal places is "+xnew);
        }
}
```

Newton-Raphson method

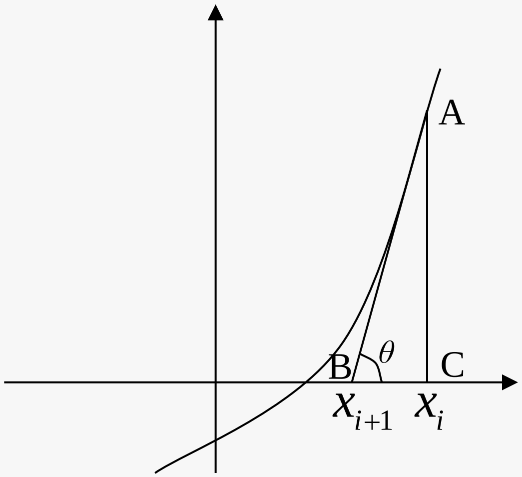
Problem

Solve for the root of the equation $x - \cos x = 0$ using Netwon-Raphson's method

Newton-Raphson algorithm

Starting with an approximation x_i to the root of a function $f(x) = 0$

We can improve the approximation by x_{i+1} (where the abscissa is crossed) by the tangent at the point x_i



$$\tan\theta = f'(x_i) = \frac{AC}{BC} = \frac{f(x_i)}{x_i - x_{i+1}}$$

and hence

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Newton-Raphson method

Breakdown into logical steps

1. Start with an initial guess `xold` and calculate `fxold`
2. Calculate new approximation `xnew=xold–fxold/fdashxold`
3. If the two successive approximations `xold` and `xnew` are close enough then print solution

   else

   save `xold` as `xnew` and return to step 2

Newton-Raphson method

```
/*
 * Program to find the root of x-cos(x)=0 accurate to 6
 * decimal places so that abs(xnew-xold) < 0.000001
 * using Newton-Raphson method
 */
class NewtonRaphson
{
    public static void main(String[] args)
        {
        double xold,xnew,fxold,fdashxold;
        int iteration;
        xold=0.0;
        iteration=0;
        do
            {
            iteration = iteration + 1;
            // determine f(xold) and f'(xold)
            fxold = xold - Math.cos(xold);
            fdashxold = 1.0 + Math.sin(xold);
            xnew = xold – (fxold/fdashxold);
            System.out.println("Approx for iteration"+iteration+"  is "+xnew);
            diff = Math.abs(xnew-xold);
            xold = xnew;
            }
        while (diff > 0.000001);
        System.out.println("root to six decimal places is "+xnew);
        }
}
```

NOTE – practise may be needed

When we differentiate *ax* we get *a*

e.g differentiate 12*x* we get 12

When we differentiate cos(*ax*) we get –*a*sin(*ax*)

e.g differentiate cos(5*x*) we get -5sin(5*x*)

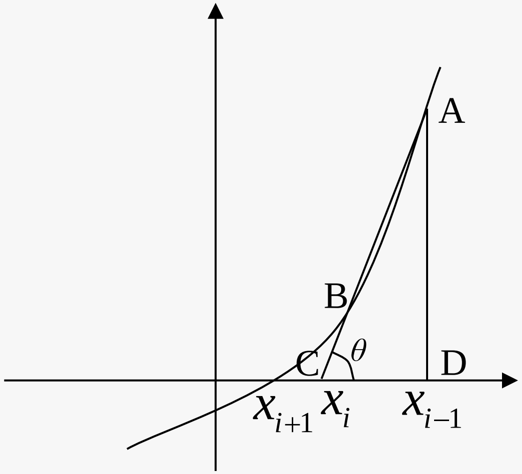
Secant method

Problem

Solve for the root of the equation $x - \cos x = 0$ using the Secant method

Secant algorithm

Starting with two approximations x_i, x_{i-1} to the root of a function $f(x) = 0$. We can improve the approximation by x_{i+1} (where the abscissa is crossed) by the extended chord or ***secant*** at the point



$$\tan\theta = \frac{f(x_i)}{x_i - x_{i+1}} = \frac{f(x_{i-1})}{x_{i-1} - x_{i+1}}$$

and hence solving for x_{i+1}

$$x_{i+1} = x_i - \frac{(x_i - x_{i-1})\, f(x_i)}{f(x_i) - f(x_{i-1})}$$

Secant method

Breakdown into logical steps

1. Start with two initial guesses `xold1,xold2` and calculate `fxold1,fxold2`
2. Calculate new approximation
   `xnew=xold1–(xold1-xold2)*fxold1/(fxold1-fxold2)`
3. If the two successive approximations `xold1` and `xnew` are close
   enough then print solution

   else

   save `xold2` as `xold1` and `xold1` as `xnew` and return to step 2

Secant method

- The Secant method replaces the tangent used in the Newton-Raphson method by a secant
- It does not require the derivative to be known
- It is necessary to have two starting values x_i, x_{i-1} for the Secant method
- Only one function evaluation is needed per iteration of the Secant method
- The Secant and Newton-Raphson methods do not always converge, but when they do...they are very quick

Secant method

```
/*
 * Program to find the root of x-cos(x)=0 accurate to 6
 * decimal places so that abs(xnew-xold) < 0.000001
 * using Secant method
 */
class Secant
{
    public static void main(String[] args)
        {
        double xold1,xold2,xnew,fxold1,fxold2,diff;
        int iteration;
        xold1=0.0;
        xold2=0.5;
        iteration=0;
        do
            {
            iteration = iteration + 1;
            // determine f(xold1) and f(xold2)
            fxold1 = xold1 - Math.cos(xold1);
            fxold2 = xold2 - Math.cos(xold2);
            xnew = xold1 - (fxold1*(xold1-xold2))/(fxold1-fxold2);
            System.out.println("Approx for iteration"+iteration+"  is "+xnew);
            diff = Math.abs(xnew-xold1);
            xold2 = xold1;
            xold1 = xnew;
            }
        while (diff > 0.000001);
        System.out.println("root to six decimal places is "+xnew);
         }
}
```

Numerical (in)accuracy

- Float (or double) types are used when we are dealing with fractions of whole numbers
- They contain a whole and a fractional part and are separated by a decimal point (and more...see later slides)
- This fractional part is very helpful otherwise how can we describe 1/4, but...must exercise some caution
- Add up the numbers for 1/n where n=1,...,10 we get

1 + 0.5 + 0.3333 + 0.25 + 0.2 + 0.6666 + 0.1428 + 0.125 + 0.1111 + 0.1 = 3.4285

- Now add them up in the reverse order n=10,...,1,-1 and we get 3.4285 surprise, surprise!
- When we repeat this for n=100000 we get
- i= 1,100000 sum is 12.090146129863335
- i= 100000,1,-1 sum is 12.090146129863408 so what!

Significance to real-world

- Exploded 37 seconds after liftoff with $500 M cargo
- What happened?
  - Computed horizontal velocity as floating point number
  - Converted to 16-bit integer
  - Worked OK for Ariane 4
  - Used same software for Ariane 5 but ***Overflowed***

Ariane 5

Components for scientific/exponential notation

Sign of the mantissa

Sign of the exponent

$$-0.35790 \times 10^{-6}$$

Location of decimal point

Mantissa

Base

Exponent

Format Specification

- *Illustrate* using 8 bits (nonsensical-why?)
  - Use two bits for exponent at the expense of 5 bits for mantissa

Sign of the mantissa

SEEMMMMM

2-digit Exponent

5-digit Mantissa

Format

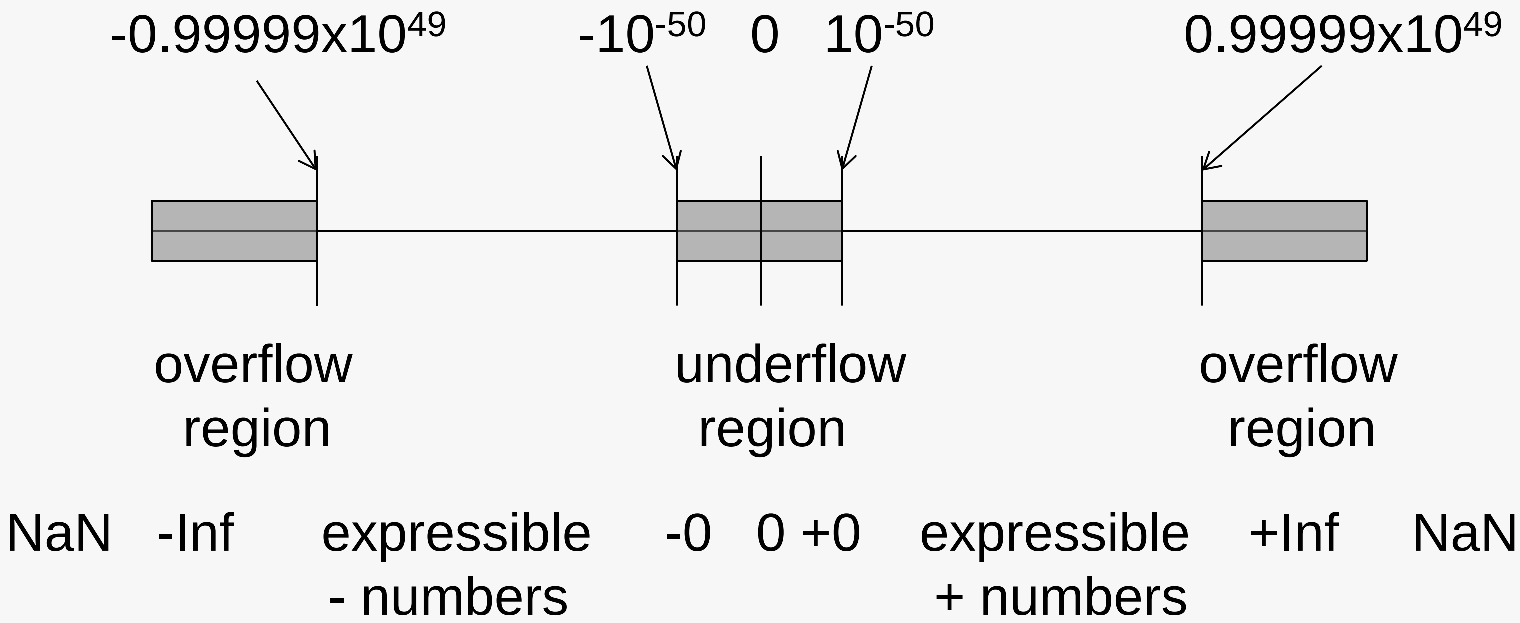
- Mantissa: sign digit is in *sign-magnitude format*
- Assume decimal point located at beginning of mantissa
- **Excess-N** notation: Complementary notation
  - Pick middle value as offset where N is the middle value
    .
  - (In another less nonsensical example assume we can have exponent in range -50 to 49.)

Representation	*0*	*49*	*50*	*99*
Exponent being represented	*-50*	*-1*	*0*	*49*

– Increasing value +

Overflow and Underflow

- Possible for the number to be too large or too small for representation



Conversion Examples

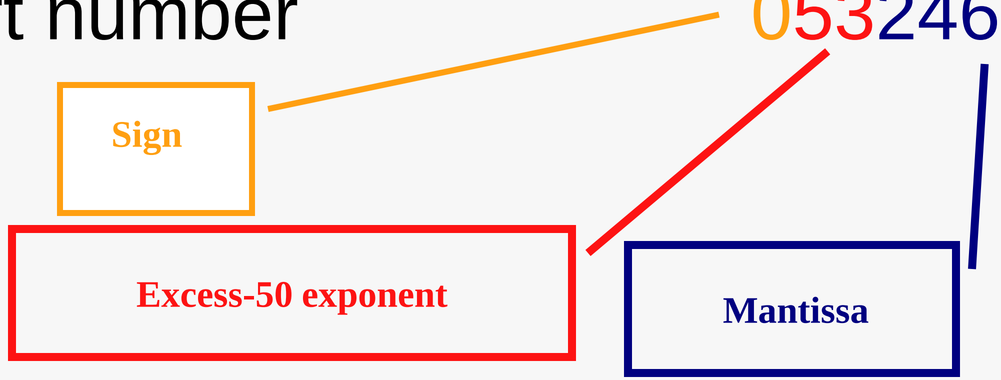
05324567	=	0.24567×10^{3}	=	245.67
14810000	=	-0.10000×10^{-2}	=	– 0.0010000
15555555	=	-0.55555×10^{5}	=	– 55555
04925000	=	0.25000×10^{-1}	=	0.025000

Normalization

- Shift numbers left by increasing the exponent until leading zeros eliminated
- Converting decimal number into standard format
  1. Provide number with exponent (0 if not yet specified)
  2. Increase/decrease exponent to shift decimal point to proper position
  3. Decrease exponent to eliminate leading zeros on mantissa
  4. Correct precision by adding 0’s or discarding/rounding least significant digits

Example 1: 246.8035

1. Add exponent — 246.8035×10^0
2. Position decimal point — $.2468035 \times 10^3$
3. Already normalized
4. Cut to 5 digits — $.24680 \times 10^3$
5. Convert number — 05324680



Example 2: 1255×10^{-3}

1. Already in exponential form	1255×10^{-3}
2. Position decimal point	$0.1255 \times 10^{+1}$
3. Already normalized	
4. Add 0 for 5 digits	$0.12550 \times 10^{+1}$
5. Convert number	05112550

Example 3: - 0.00000075

1. Exponential notation $- 0.00000075 \times 10^{0}$
2. Decimal point in position
3. Normalizing $- 0.75 \times 10^{-6}$
4. Add 0 for 5 digits $- 0.75000 \times 10^{-6}$
5. Convert number 14475000

Floating Point Calculations

- Addition and subtraction – three rules
  1. Exponent and mantissa treated separately
  2. Exponents of numbers must agree
     - Align decimal points
     - Least significant digits may be lost
  3. Mantissa overflow requires exponent again shifted right

Addition and Subtraction

- Do the following computation 9.952+0.06785

Add the floating point numbers	05199520
	+ 04967850
Align exponents	05199520
	0510067850
Add mantissas; (1) indicates a carry	(1)0019850
Carry requires right shift	05210019(850)
Round	05210020 -> **0.10020×10^2**
Check results	

$05199520 = 0.99520 \times 10^{1} = 9.9520$

$04967850 = 0.67850 \times 10^{-1} = \underline{0.06785}$

$= 10.01985$

In exponential form $=$ **0.1001985×10^2**

Multiplication and Division

- Mantissas: multiplied or divided
- Exponents: added or subtracted
  - Normalization necessary to
    - Restore location of decimal point
    - Maintain precision of the result
  - Adjust excess value since added twice
    - For example, two numbers with exponent of 3 represented in excess-50 notation
    - 53 + 53 =106
    - Since 50 added twice, subtract: 106 – 50 =56

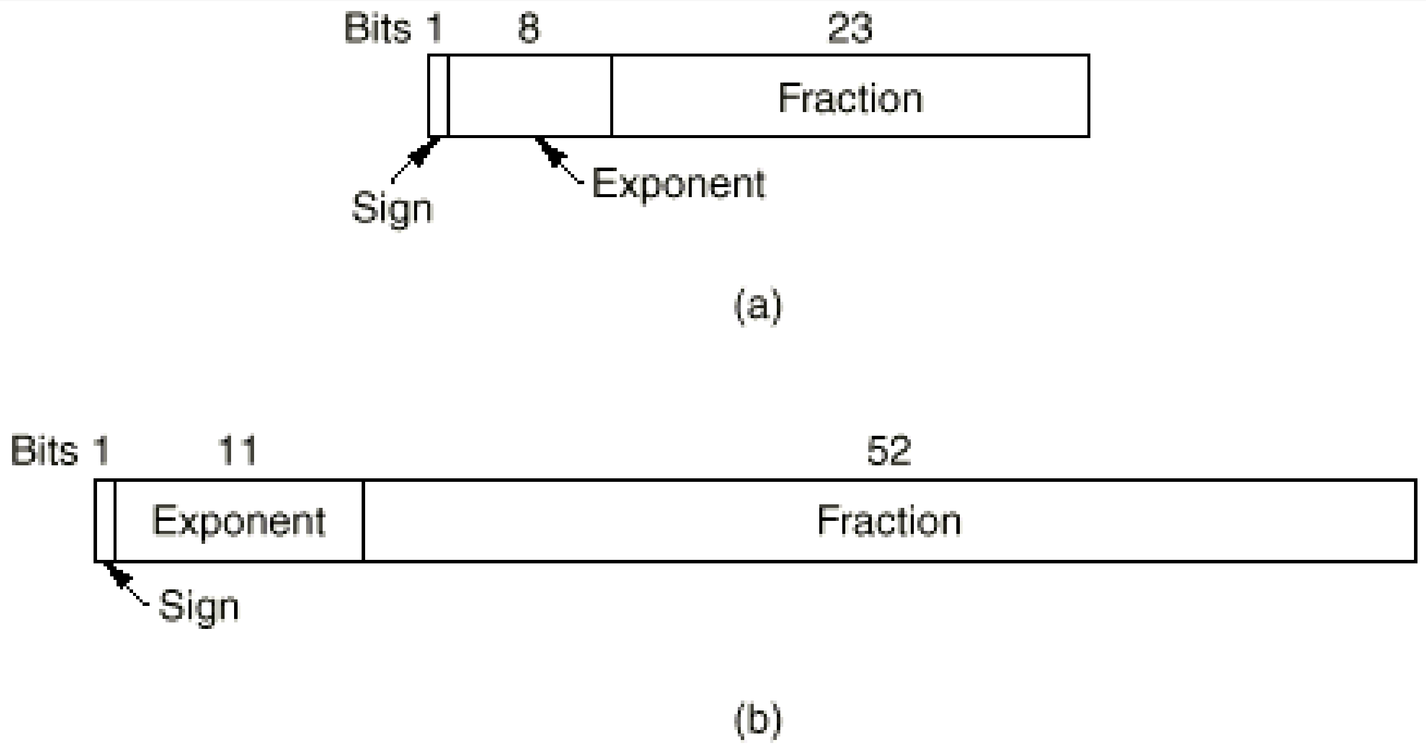
Multiplication and Division

- Do the following computation 20.000 x 0.000125
- Maintain precision by normalizing and rounding

Multiply 2 numbers	05220000 x 04712500
Add exponents, subtract offset	52 + 47 – 50 = 49
Multiply mantissas	0.20000 x 0.12500 = 0.025000000
Normalize the results	04825000
Round	04825000 -> **0.25000 x 10^{-2}**
Check results	
05220000 =	0.20000 x 10^{2}
04712500 =	0.125 x 10^{-3}
=	0.0250000000 x 10^{-1}
Normalizing and rounding =	**0.25000 x 10^{-2}**

Floating Point in the Computer

- Floating point representation using (a) 32 and (b) 64 bit precision



IEEE 754 Standard

Precision	Single (32 bit)	Double (64 bit)
Sign	1 bit	1 bit
Exponent	8 bits	11 bits
Notation	Excess-127	Excess-1023
Implied base	2	2
Range	2^{-126} to 2^{127}	2^{-1022} to 2^{1023}
Mantissa	23	52
Decimal digits	≈ 7	≈ 15
Value range	$\approx 10^{-45}$ to 10^{38}	$\approx 10^{-300}$ to 10^{300}

Subtleties of floating point calculations

- Use double instead of float for accuracy.
- Use float only if you really need to conserve memory, and are aware of the associated risks with accuracy.
- Usually it doesn't make things faster, and occasionally makes things slower.
- Be careful of calculating the difference of two very similar values and using the result in a subsequent calculation.
- Be careful about adding two quantities of very different magnitudes.

Subtleties of Rounding

- Suppose we insist on floating-point operations being properly rounded.
- What does *properly rounded* mean for 0.5 ?
- Typical rule, round up always if half way
- Introduces Bias - Some computations sensitive to this bias
- Computation of orbit of pluto was significantly off because of this problem "long-term behaviour of the motion of Pluto over 5.5 billion years" , Kinoshita and Nakai (2004)

Acknowledgements

A number of sources, but most prominently

"The Architecture of Computer Hardware and Systems Software: An Information Technology Approach" L.Senne and W.Wong , Bentley College, 2003.