IC-150
Computation for Engineers
Numerical Methods

by
A. Avudainayagam
Department of Mathematics
revised by
Timothy Gonsalves
Dept of Computer Science & Engg, IIT Madras

Numerical Methods

- Used because:
  - Analytical solution extremely difficult for a complex function
  - Analytical solution may require evaluation of esoteric functions
  - Mathematical functions may not be analytical
  - Function may be in the form of pairs of data
    - Eg. Given experimental data for column design:

Column radius (m)	1.2	1.5	1.8	2.0	2.95
Max Load (tons)	10.3	15.6	20.3	32.7	43.5

- What is column radius for 35 tons?

Numerical Methods

- Used for:
  - Solution of algebraic equations
  - Approximation of functions
  - Differentiation and integration of functions
  - Solution of differential equations
  - Statistical analysis of data

Numerical Errors

- Source of Errors
  - Approximate evaluation of functions
    - $\pi = 22/7 = 3.14285714\ldots$
  - Representation of numbers in a finite number of bits
  - Round-off error, eg correct to 3 decimals:
    - $2.000 + 0.77\times10^{-6} = 2.00000077 = 2.000$
- Reduction of Error
  - Iterative solutions -- repeat until error < ε
    - Efficiency of convergence
    - Stability --- may never converge

Fundamental Motifs

- Several techniques for a given problem
- Technique of choice depends on nature of the data
- One technique with modifications may be used to solve several different problems
- Error analysis essential to determine reliability of the computed results

The Bisection Method

- Find an interval $[x_0, x_1]$ such that $f(x_0)f(x_1) < 0$
  - This may not be easy. Use your knowledge of the physical phenomenon that the equation represents.
- In each iteration, cut the interval into half
  - Examine the sign of the function at the mid point

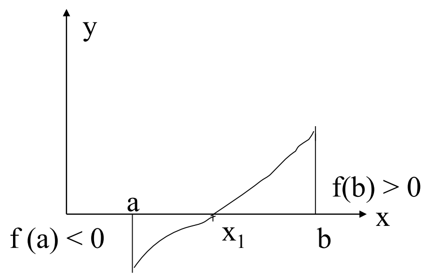
$$m = \frac{x_0 + x_1}{2}$$

  - If $f(m) = 0$, x is the root
  - If $f(m) \neq 0$ and $f(x_0)f(x) < 0$, root lies in $[x_0, m]$
  - Otherwise root lies in $[m, x_1]$
- Repeat the process until convergence (length of interval $< \varepsilon$)

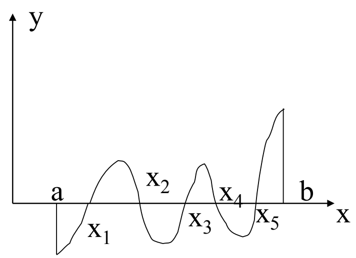
Root Finding: f(x)=0

Method 1: The Bisection method

Thm: If f(x) is continuous in [a,b] and if f(a)f(b)<0, then there is at least one root of f(x)=0 in (a,b).



Single root in (a,b)



Multiple roots in (a,b)

Number of Iterations and Error Tolerance

- Length of the interval (where the root lies) after n iterations

$$e_n = \frac{x_1 - x_0}{2^n}$$

- We can fix the number of iterations so that the root lies within an interval of chosen length $\in$ (error tolerance).

$$e_n \leq \in \Longrightarrow n \geq \frac{\ln(x_1 - x_0) - \ln \in}{\ln 2}$$

- If n satisfies this, root lies within a distance of $\varepsilon/2$ of the actual root

- Though the root lies in a small interval, $|f(x)|$ may not be small if $f(x)$ has a large slope.
- Conversely if $|f(x)|$ small, x may not be close to the root if $f(x)$ has a small slope.
- So, we use both these facts for the termination criterion. We first choose an error tolerance on $f(x)$: $|f(x)| < \in$ and K the maximum number of iterations.

Bisection Method Example

$f(x) = x^3-2 = 0$, $\varepsilon = 10^{-4}$

$x_0 = 1$, $x_1 = 2$

k	x_0	x_1	m	f(m)
1	1	2	1.5	1.375
2	1	1.5	1.25	-0.4688
3	1.25	1.5	1.375	0.5996
4	1.25	1.375	1.3125	0.2610
5	1.25	1.3125	1.2813	0.1033

- After 13 iterations, m = 1.2599 (to 4 decimal places)
- Using $n \geq \dfrac{\ln(x_1 - x_0) - \ln \in}{\ln 2}$, $n \geq 12.29$

Pseudo code (Bisection Method)

1. Input $\in > 0$, $K > 0$, $x_1 > x_0$ so that $f(x_0)\, f(x_1) < 0$.
   Compute $f_0 = f(x_0)$.
   $k = 1$ (iteration count)
2. Do
   {
   (a) Compute $m = \left(\dfrac{x_0 + x_1}{2}\right)$, and $f = f(m)$
   (b) If $f \times f_0 < 0$, set $x_1 = m$
   otherwise set $x_0 = m$
   (c) Set $k = k+1$
   } while $|f| > \in$ and $k \leq K$
3. Set root = m

Example using Scilab

Find roots of $f(x) = x^3-2 = 0$

```
→ roots(poly([-2 0 0 1],'x', "coeff"))
 ans  =

  - 0.6299605 + 1.0911236i
  - 0.6299605 - 1.0911236i
    1.259921
→
```

False Position Method (Regula Falsi)

- Root may lie near end of interval with smaller value of $|f|$
- Instead of bisecting the interval $[x_0, x_1]$, choose the point where the straight line through the end points meets the x-axis, say w
- Bracket the root with $[x_0, w]$ or $[w, x_1]$ depending on sign of $f(w)$

False Position Method (Pseudo Code)

1. Choose $\in > 0$ (tolerance on $|f(x)|$)
   $K > 0$ (maximum number of iterations)
   $k = 1$ (iteration count)
   x_0, x_1 (so that $f_0 f_1 < 0$)
2. do {
   a. Compute $w = x_1 - \left(\frac{x_1 - x_0}{f_1 - f_0}\right) f_1$ and $f = f(w)$

   b. If $f_0 \times f < 0$ set $x_1 = w$, $f_1 = f$

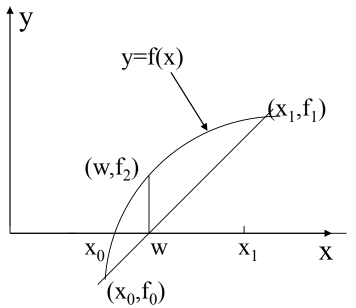
   else set $x_0 = w$, $f_0 = f$

   c. $k = k+1$

   } while ($|f| \geq \in$) and ($k \leq K$)

4. The root is $x = w$

False Position Method



Straight line through (x_0,f_0) ,(x_1,f_1) :

$$y = f_1 + \frac{f_1 - f_0}{x_1 - x_0}(x - x_1)$$

New end point w:

$$w = x_1 - \left(\frac{x_1 - x_0}{f_1 - f_0}\right) f_1$$

Regula Falsi Example

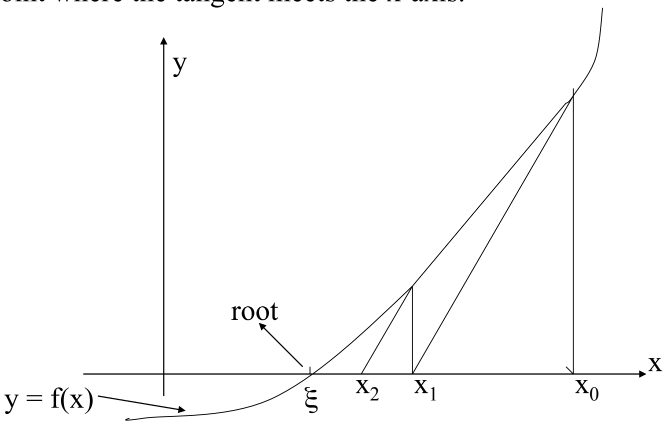
$f(x) = x^3-2 = 0$, $\varepsilon = 10^{-4}$

$x_0 = 1$, $x_1 = 2$

k	x_0	x_1	$f(x_0)$	$f(x_1)$	w	f(w)
0	1.0	2.0	-1.0	6.0	1.1429	-0.5071
1	1.1429	2.0	-0.5071	6.0	1.2097	-0.2298
2	1.2097	2.0	-0.2298	6.0	1.2389	-0.0987
3	1.2389	2.0	-0.0987	6.0	1.2512	-0.0412
4	1.2512	2.0	-0.0412	6.0	1.2563	-0.0172
9	1.2598	2.0	-0.0003	6.0	1.2607	0.0039
10	1.2598	1.2607	-0.0003	0.0039	1.2599	-0.0003
11	1.2599	1.2607	-0.0003	0.0039	1.2600	0.0002

Newton-Raphson *or* Newton's Method

At an approximation x_k to the root, the curve is approximated by the tangent to the curve at x_k and the next approximation x_{k+1} is the point where the tangent meets the x-axis.



Newton's Method - Pseudo code

1. Choose $\in > 0$ (function tolerance $|f(x)| < \in$)
   $m > 0$ (Maximum number of iterations)
   x_0 - initial approximation
   k - iteration count
   Compute $f(x_0)$
2. Do { $q = f'(x_0)$ (evaluate derivative at x_0)
   $x_1 = x_0 - f_0/q$
   $x_0 = x_1$
   $f_0 = f(x_0)$
   $k = k+1$
   }
3. While ($|f_0| \geq \in$) and ($k \leq m$)
4. The root is $x = x_1$

Tangent at (x_k, f_k) :

$$y = f(x_k) + f'(x_k)(x-x_k)$$

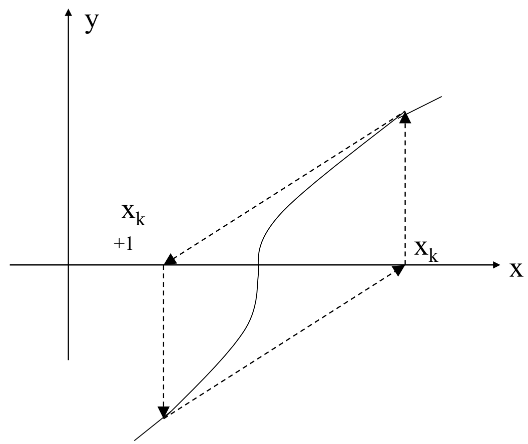
This tangent cuts the x-axis at x_{k+1}

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Warning : If $f'(x_k)$ is very small, method fails.

- Two function Evaluations per iteration

Getting caught in a cycle of Newton's Method



Alternate iterations fall at the same point .

No Convergence.

Newton's Method for finding the square root of a number $x = \sqrt{a}$

$f(x) = x^2 - a^2 = 0$

$$x_{k+1} = x_k - \frac{x_k^2 - a^2}{2x_k}$$

Example : a = 5 , initial approximation $x_0 = 2$.

$x_1 = 2.25$

$x_2 = 2.236111111$

$x_3 = 2.236067978$

$x_4 = 2.236067978$

Problems with Newton's Method

- If $|f'(x)|$ is very small, accuracy is difficult to obtain
- Depending on the initial estimate, any one of the roots may be found (answer may not have physical significance)
  - Use bisection to get close to desired root, then Newton's method for fast convergence
- May get caught in an infinite cycle

IC150 Lecture 28: Root Finding – Secant Method

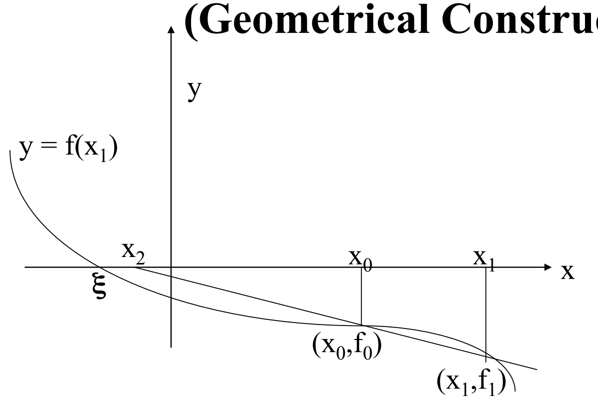
Timothy A. Gonsalves

Dept of Computer Science & Engg

The secant Method

- Newton's Method requires 2 function evaluations (f, f').
- The Secant Method requires only 1 function evaluation and converges as fast as Newton's Method at a simple root.
- Start with two points x_0, x_1 near the root (no need for bracketing the root as in Bisection Method or Regula Falsi Method) .
- x_{k-1} is dropped once x_{k+1} is obtained.

The Secant Method
(Geometrical Construction)



- Two initial points x_0, x_1 are chosen
- The next approximation x_2 is the point where the straight line joining (x_0,f_0) and (x_1,f_1) meet the x-axis
- Take (x_1,x_2) and repeat.

On Convergence

The false position method in general converges faster than the bisection method.
  # But not always, shown by counter examples

The bisection method and the false position method are guaranteed to converge

The secant method and the Newton-Raphson method are not guaranteed to converge

The secant Method (Pseudo Code)

1. Choose $\in > 0$ (function tolerance $|f(x)| \le \in$)
   $m > 0$ (Maximum number of iterations)
   x_0 , x_1 (Two initial points near the root)
   $f_0 = f(x_0)$
   $f_1 = f(x_1)$
   $k = 1$ (iteration count)
2. Do {
   $$x_2 = x_1 - \left(\frac{x_1 - x_0}{f_1 - f_0}\right) f_1$$
   $x_0 = x_1$
   $f_0 = f_1$
   $x_1 = x_2$
   $f_1 = f(x_2)$
   $k = k+1$ }
3. while $(|f_1| \ge \in)$ and $(m \le k)$

Order of Convergence

A measure of how fast an algorithm converges

Let ξ be the actual root: $f(\xi) = 0$

Let x_k be the approximate root at the kth iteration .
Error at the kth iteration, $e_k = |x_k - \xi|$

The algorithm converges with order p if there exists α such that

$$e_{k+1} = \alpha\, e_k^p$$

Order of Convergence of

- # Bisection method p = 1 (linear convergence)
- # False position - generally super linear $(1 < p < 2)$
- # Secant method $\frac{1+\sqrt{5}}{2} = 1.618$ (super linear)
- # Newton Raphson method p = 2 quadratic

Pseudo code to find 'Machine Epsilon'

1. Set $\in_M = 1$
2. Do

    {

    $\in_M = \in_M / 2$

    $x = 1 + \in_M$

    }
3. While (x > 1)
4. $\in_M = 2 \in_M$

Machine Precision

The smallest positive float $\in_M$ that can be added to one and produce a sum that is greater than one.

IC150 Lecture 29: Approximation & Interpolation

Timothy A. Gonsalves

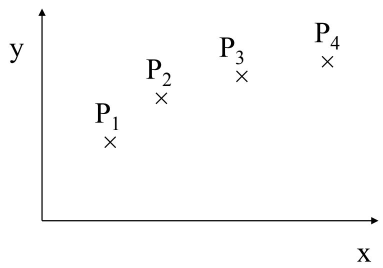
Dept of Computer Science & Engg

Approximation & Interpolation

Reasons to approximate value of a function:

- Difficult or impossible to evaluate the function analytically, eg. sine, log, etc.
- Have only a table of values and must interpolate
- Faster to compute approx function than original
- Function defined implicitly rather than by an equation

Interpolation



Given the data:

(x_k, y_k) , $k = 1, 2, 3, \ldots, n$,

find a function f which we can use to predict the value of y at points other than the samples

1. f(x) may pass through all the data points:

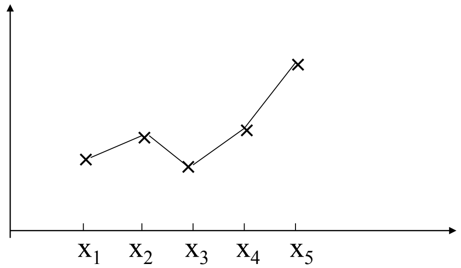
$$f(x_k) = y_k, \quad 1 \le k \le n$$

2. f(x) need not pass through any of the data points:

Need to control error $|f(x_k) - y_k|$

1. Piecewise Linear Interpolation

A straight line segment is used between each adjacent pair of data points



$$f_k(x) = y_k + \frac{x - x_k}{x_{k+1} - x_k}\left(y_{k+1} - y_k\right) \qquad 1 \le k \le n$$

Simple and computationally efficient

1. Polynomial Interpolation

For the data set (x_k, y_k), $k = 1, \ldots, n$,

we find the *one* polynomial of degree (n - 1) subject to the n interpolation constraints $f(x_k) = y_k$

$$f(x) = \sum_{k=1}^{n} a_k x^{k-1}$$

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Not feasible for large data sets, since the *condition number* increases rapidly with increasing n.

Lagrange Interpolating Polynomial

The Lagrange interpolating polynomial of degree k, $f(x)$ is constructed as follows:

1. Caculate the Lagrangian multipliers $Q_k(x)$ each of which is a polynomial of degree n-1 that is non-zero at only the one base point x_k

   Normalise by $Q_k(x_k)$

$$Q_k(x) = \prod_{i=0, i\neq k}^{n} (x - x_i) / \prod_{i=0, i\neq k}^{n} (x_k - x_i)$$

$$Q_k(x) = \frac{(x-x_1)(x-x_2)\ldots(x-x_{k-1})(x-x_{k+1})\ldots(x-x_n)}{(x_k-x_1)(x_k-x_3)\ldots(x_k-x_{k-1})(x_k-x_{k+1})(x_k-x_n)}$$

Lagrange interpolations (Pseudo Code)

```
Choose x, the point where the function value is required
y = 0
for  i = 1 to n
        p = y_i
        for j = 1 to n
            if (i ≠ j)
                p = p * (x - x_j) / (x_i - x_j)
        end for
        y = y + p
end for
```

- Each $Q_k(x)$ is a polynomial of degree (n - 1)
- $Q_k(x_j) = 1$, $j = k$

  $= 0$, $j \neq k$
- The polynomial curve that passes through the data set (x_k, y_k), k = 1, 2, ..., n is

  $f(x) = y_1 Q_1(x) + y_2 Q_2(x) + \ldots + y_n Q_n(x)$
- Polynomial is written directly without having to solve a system of equations

Lagrange Interpolation Example

- Given the following base points, estimate sin 23° to five decimal places:

i	x_i	y_i
0	20°	0.34202
1	22°	0.37641
2	24°	0.40674
3	26°	0.43837

$$Q_k(x) = \prod_{i=0, i\neq k}^{n} (x - x_i) / \prod_{i=0, i\neq k}^{n} (x_k - x_i)$$

$Q_0(x) = -0.0625$

$Q_1(x) = -0.5625$

$Q_2(x) = -0.5625$

$Q_3(x) = -0.0625$

$f(x) = y_1 Q_1(x) + y_2 Q_2(x) + \ldots + y_n Q_n(x)$

$= (0.34202)(-0.0625) +$

$(0.37461)(0.5625) +$

$(0.40674)(0.5625) +$

$(0.43837)(-0.0625)$

$= 0.39074$

True value: 0.39073, discrepancy due to accumulation of round-off error.

IC150 Lecture 30: Curve Fitting

Timothy A. Gonsalves

Dept of Computer Science & Engg

2. Minimax

Least squares approximation gives good fit overall, but may have large deviation from one point

Minimize the maximum deviation from y_k

Also known as *Chebyshev* or *optimal polynomial* approximation

$$E = \max_k | f(x_k) - y_k |$$

2. Least Squares Fit

If the number of samples is large or the dependant variable contains measurement noise, it is often better to find a function that minimizes an error criterion such as

$$E = \sum_{k=1}^{n} [f(x_k) - y_k]^2$$

A function that minimizes E is called the Least Squares Fit

Depending on the nature of the function we have:

- linear regression
- polynomial regression (quadratic, cubic, …)
- exponential regression, etc.

Straight Line Fit *or* Linear Regression

- To fit a straight line through the n points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
- Assume $f(x) = a_1 + a_2 x$

Error

$$E = \sum_{k=1}^{n} [f(x_k) - y_k]^2$$

$$= \sum_{k=1}^{n} [a_1 + a_2 x_k - y_k]^2$$

- Find a_1, a_2 which minimizes E

$$\frac{\partial E}{\partial a_1} = 2\sum_{k=1}^{n} (a_1 + a_2 x_k - y_k) = 0$$

$$\frac{\partial E}{\partial a_2} = 2\sum_{k=1}^{n} (a_1 + a_2 x_k - y_k) x_k = 0$$

$$\frac{\partial E}{\partial a_1} = 2\sum_{k=1}^{n}(a_1 + a_2 x_k - y_k) = 0$$

$$\frac{\partial E}{\partial a_2} = 2\sum_{k=1}^{n}(a_1 + a_2 x_k - y_k)x_k = 0$$

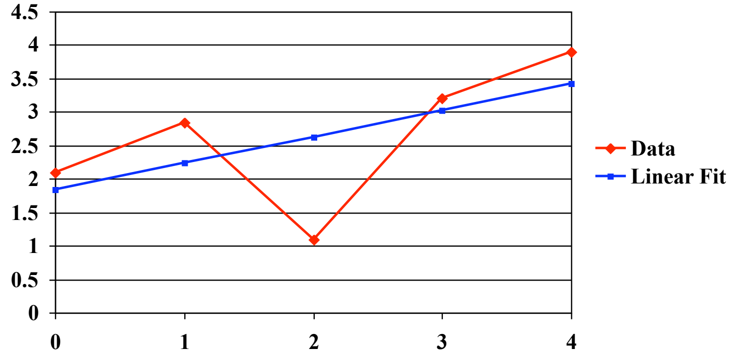
Solve:

$$\begin{bmatrix} n & \sum x_k \\ \sum x_k & \sum x_k^2 \end{bmatrix}\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum y_k \\ \sum x_k y_k \end{bmatrix}$$

Straight Line Fit (example)

Data points: (0, 2.10), (1, 2.85), (2, 1.10), (3, 3.20), (4, 3.90)

Linear fit: $f(x) = 1.84 + 0.395x$



Straight Line Fit (example)

Fit a straight line through the five points

(0, 2.10), (1, 2.85), (2, 1.10), (3, 3.20), (4, 3.90)

$a_{11} = n = 5$

$a_{12} = \sum x_k = 0 + 1 + 2 + 3 + 4 = 0$

$a_{21} = a_{12}$

$a_{22} = \sum x_k^2 = 1 + 4 + 9 + 16 = 30$

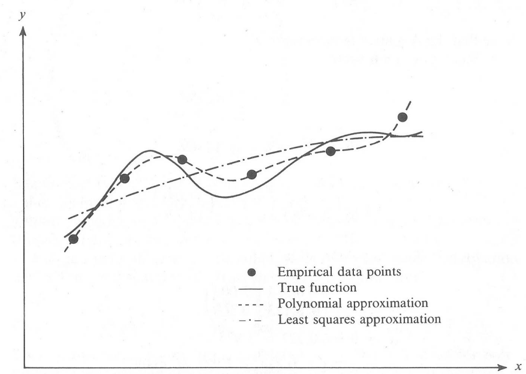
$b_1 = \sum y_k = 2.10 + 2.85 + 1.10 + 3.20 = 13.15$

$b_2 = \sum x_k y_k = 2.85 + 2(1.10) + 3(3.20) + 4(3.90) = 30.25$

$$\begin{bmatrix} 5 & 10 \\ 10 & 30 \end{bmatrix}\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 13.15 \\ 30.25 \end{bmatrix}$$

$a_1 = 1.84$, $a_2 = 0.395$, $f(x) = 1.84 + 0.395x$

Two Kinds of Curve Fitting



Data Representation

Integers - Fixed Point Numbers

Decimal System - base 10 uses 0,1,2,…,9

$(396)_{10} = (6 \times 10^0) + (9 \times 10^1) + (3 \times 10^2) = (396)_{10}$

Binary System - base 2 uses 0,1

$(11001)_2 = (1 \times 2^0) + (0 \times 2^1) + (0 \times 2^2) + (1 \times 2^3) + (1 \times 2^4) = (25)_{10}$

Largest number that can be stored in m-digits

base - 10 : $(99999\ldots9) = 10^m - 1$

base - 2 : $(11111\ldots1) = 2^m - 1$

m = 3 $(999) = 10^3 - 1$

$(111) = 2^3 - 1$

Limitation: Memory cells consist of 8 bits (1 byte) multiples, each position containing 1 binary digit

Decimal to Binary Conversion

Convert $(39)_{10}$ to binary form

base = 2

2	39	
2	19	+ Remainder 1
2	9	+ Remainder 1
2	4	+ Remainder 1
2	2	+ Remainder 0
2	1	+ Remainder 0
	0	+ Remainder 1

Put the remainder in reverse order

$(100111)_2 = (1 \times 2^0) + (1 \times 2^1) + (1 \times 2^2) + (0 \times 2^3) + (0 \times 2^4)$

$+ (1 \times 2^5) = (39)_{10}$

- Common cell lengths for integers : k = 16 bits

  k = 32 bits

Sign - Magnitude Notation

First bit is used for a sign

0 - non negative number
1 - negative number

The remaining bits are used to store the binary magnitude of the number.

Limit of 16 bit cell : $(32767)_{10}$

Limit of 32 bit cell : $(2\ 147\ 483\ 647)_{10}$

Two's Complement notation

Definition : The two's complement of a negative integer I in a k - bit cell :

Two's Complement of I $= 2^k + I$

(Eg) : Two's Complement of $(-3)_{10}$ in a 3 - bit cell

$$= (2^3 - 3)_{10} = (5)_{10} = (101)_2$$

$(-3)_{10}$ will be stored as 101

The Two's Complement notation admits one more negative number than the sign - magnitude notation.

Storage Scheme for storing an integer I in a k - bit cell in Two's Complement notation

$$\text{Stored Value } C = \begin{cases} I \;,\quad I \geq 0 \text{ , first bit} = 0 \\ 2^k + I \;,\quad I < 0 \end{cases}$$

(Eg) Take a 2 bit cell (k = 2)

Range in Sign - magnitude notation : $2^1 - 1 = 1$

$-1 = 11$

$1 = 01$

Range in Two's Compliment notation

Two's Compliment of $-1 = 2^2 - 1 = (3)_{10} = (11)_2$

Two's Compliment of $-2 = 2^2 - 2 = (2)_{10} = (10)_2$

Two's Compliment of $-3 = 2^2 - 2 = 0$ - Not possible

Floating Point Numbers

Integer Part + Fractional Part

Decimal System - base 10

235 . 7846

Binary System - base 2

10011 . 11101

Fractional Part $(0.7846)_{10} = \frac{7}{10} + \frac{8}{10^2} + \frac{4}{10^3} + \frac{6}{10^4}$

Fractional Part $(0.11101)_2 = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{0}{2^4} + \frac{1}{2^5}$

Decimal Fraction → Binary Fraction

Convert $(0.9)_{10}$ to binary fraction

$$\begin{array}{r l}
0.9 & \\
\times 2 & \\
\hline
0.8 & + \text{ integer part } 1 \\
\times 2 & \\
\hline
0.6 & + \text{ integer part } 1 \\
\times 2 & \\
\hline
0.2 & + \text{ integer part } 1 \\
\times 2 & \\
\hline
0.4 & + \text{ integer part } 0 \\
\times 2 & \\
\hline
0.8 & + \text{ integer part } 0
\end{array}$$

Repetition

$(0.9)_{10} = (0.1\overline{1100})_2$

Binary Fraction → Decimal Fraction

(10.11)

Integer Part $(10)_2 = 0.2^0 + 1.2^1 = 2$

Fractional Part $(11)_2 = \frac{1}{2} + \frac{1}{2^2} = 0.5 + 0.25 = 0.75$

Decimal Fraction = $(2.75)_{10}$

Scientific Notation (Decimal)

$0.0000747 = 7.47 \times 10^{-5}$

$31.4159265 = 3.14159265 \times 10$

$9{,}700{,}000{,}000 = 9.7 \times 10^9$

Binary

$(10.01)_2 = (1.001)_2 \times 2^1$

$(0.110)_2 = (1.10)_2 \times 2^{-1}$

Computer stores a binary approximation to x

$$x \approx \pm q \times 2^n$$

q is mantissa , n is exponent

$$(-39.9)_{10} = (-100111.1\ 1100)_2$$

$$= (-1.0001111\ 1100)_2 \times (2^5)_{10}$$

… Floating Point

- Normalisation
  - Decimal point and base of mantissa, exponent not represented
  - high-order bit is always 1
  - it is implicit (saves 1 bit storage per number)
- Exponent
  - Excess-128 notation
- Suppose use base-16:
  - Eg: $-a.8 \times 16^{-100} = -1010.1000 \times 2^{-400}$
  - High-order digit must be represented (4 bits)
  - Exponent range is 16^8 rather than 2^8
- Most commonly used is IEEE format

Decimal Value of stored number $(-39.9)_{10}$

$$= -1.\underbrace{001111\ 1100\ 1100\ 1100\ 11001}_{\text{23 bit}}$$

32 bits : First bit for sign

Next 8 bits for exponent

23 bits for mantissa

$$= -39.\ 900001525\ \ 878\ \ 906\ \ 25$$

Round off Errors can be reduced by Efficient Programming Practice

The number of operations (multiplications and additions) must be kept minimum. (Complexity theory)

An Example of Efficient Programming

Problem : Evaluate the value of the Polynomial.

$P(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$

at a given x.

Requires 13 mulitiplications and 4 additions.

Bad Programme!

An Efficient method (Horner's method)

$P(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$

$= a_0 + x(a_1 + x(a_2 + x(a_3 + xa_4)))$

Requires 4 multiplications and 4 additions.

Pseudo-code for an n^{th} degree polynomial

Input a's, n, x

```
p = 0
for i = n, n-1, … , 0
  {
        p = x × p + a[i]
  }
```

Summary

- Finding the root(s) of an equation
  - Bisection, regula falsi, Newton's, secant
- Fitting curves to data:
  - Exact: Lagrange interpolation
  - Least squares fit: linear regression (also polynomial, exponential, etc)
    - OpenOffice functions: LINEST, LOGEST, TREND
- Several techniques for a given problem
- Technique of choice depends on nature of the data
- Error analysis essential to determine reliability of the computed results

References

M.C. Kohn, *Practical Numerical Methods: Algorithms and Programs*, Macmillan, 1987

R. Bhat & S. Chakraverty, *Numerical Analysis in Engineering*, Narosa, 2004

M.K. Jain, S.R.K. Iyengar & R.K. Jain, *Numerical Methods for Scientific and Engineering Computation*, 3rd ed., Wiley Eastern, 1994