CS 101: Computer Programming and Utilization

Jul-Nov 2017

Umesh Bellur
(cs101@cse.iitb.ac.in)

Lecture 9: Common Mathematical Functions

Learn Methods For Common Mathematical Operations

- Evaluating common mathematical functions such as
  Sin(x)
  log(x)
- Integrating functions numerically, i.e. when you do not know the closed form
- Finding roots of functions, i.e. determining where the function becomes 0
- All the methods we study are approximate. However, we can use them to get answers that have as small error as we want
- The programs will be simple, using just a single loop

Outline

- McLaurin Series (to calculate function values)
- Numerical Integration
- Bisection Method
- Newton-Raphson Method

MacLaurin Series

When x is close to 0:

$$f(x) = f(0) + f'(0)x + f''(0)x^2 / 2! + f'''(0)x^3 / 3! + \cdots$$

E.g. if $f(x) = \sin x$

$f(x) = \sin(x)$, $f(0) = 0$

$f'(x) = \cos(x)$, $f'(0) = 1$

$f''(x) = -\sin(x)$, $f''(0) = 0$

$f'''(x) = -\cos(x)$, $f'''(0) = -1$

$f''''(x) = \sin(x)$, $f''''(0) = 0$

Now the pattern will repeat

Example

Thus $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! \ldots$

A fairly accurate value of sin(x) can be obtained by using sufficiently many terms

Error after taking i terms is at most the absolute value of the i+1th term

Program Plan-High Level

$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! \ldots$

Use the *accumulation idiom*

Use a variable called term

  This will keep taking successive values of the terms

Use a variable called sum

  Keep adding term into this variable

Program Plan: Details

$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! \ldots$

- Sum can be initialized to the value of the first term So sum = x
- Now we need to figure out initialization of term and it's update
- First figure out how to get the *k*th term from the *(k-1)* th term

Program Plan: Terms

sin(x) = x – $x^3/3!$ + $x^5/5!$ – $x^7/7!$ …

Let t_k = kth term of the series, k=1, 2, 3…

$t_k = (-1)^{k+1}x^{2k-1}/(2k-1)!$

$t_{k-1} = (-1)^k x^{2k-3}/(2k-3)!$

$t_k = (-1)^k x^{2k-3}/(2k-3)! \quad * \; (-1)\,(x^2)/((2k-2)(2k-1))$

$= \; - \; t_{k-1}\,(x)^2/((2k-2)(2k-1)$

Program Plan

- Loop control variable will be k
- In each iteration we calculate t_k from t_{k-1}
- The term t_k is added to sum
- A variable term will keep track of t_k

  At the beginning of k^{th} iteration, term will have the value t_{k-1}, and at the end of k^{th} iteration it will have the value t_k
- After k^{th} iteration, sum will have the value = sum of the first k terms of the Taylor series
- Initialize sum = x, term = x
- In the first iteration of the loop we calculate the sum of 2 terms. So initialize k = 2
- We stop the loop when term becomes small enough

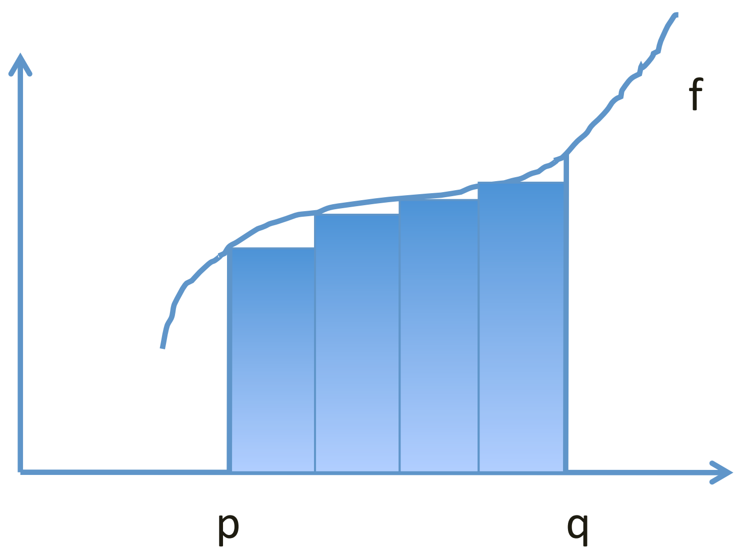
Program

```
main_program{
    double x; cin >> x;
    double epsilon = 1.0E-20; // arbitrary.
    double sum = x, term = x;
    for(int k=2; abs(term) > epsilon; k++){
        term *= -x*x / (2*k – 1) / (2*k – 2);
        sum += term;
    }
    cout << sum << endl;
}
```

Numerical Integration (General)

Integral from p to q = area under curve

Approximate area by rectangles



Plan (General)

- Read in p, q
  (assume $p < q$)
- Read in n = number of rectangles
- Calculate w = width of rectangle = (q-p)/n
- ith rectangle, i=0,1,…,n-1 begins at p+iw
- Height of ith rectangle = f(p+iw)
- Given the code for f, we can calculate height and width of each rectangle and so we can add up the areas

Example: Numerical Integration To Calculate ln(x)

ln(x) = natural logarithm

= $\int$ 1/x dx from 1 to x

= area under the curve f(x)=1/x from 1 to x

```
double x; cin >> x;
double n; cin >> n;
double w = (x-1)/n;    // width of each rectangle
double area = 0;
for(int i=0; i<n; i++)
    area = area + w * 1/(1+i*w);
cout << area << endl;
```

Remarks

- By increasing n, we can get our rectangles closer to the actual function, and thus reduce the error
- However, if we use too many rectangles, then there is roundoff error in every area calculation which will get added up
- We can reduce the error also by using trapeziums instead of rectangles, or by setting rectangle height = function value at the midpoint of its width

  Instead of f(p+iw), use f(p+iw + w/2)
- For calculation of ln(x), you can check your calculation by calling built-in function log(x)

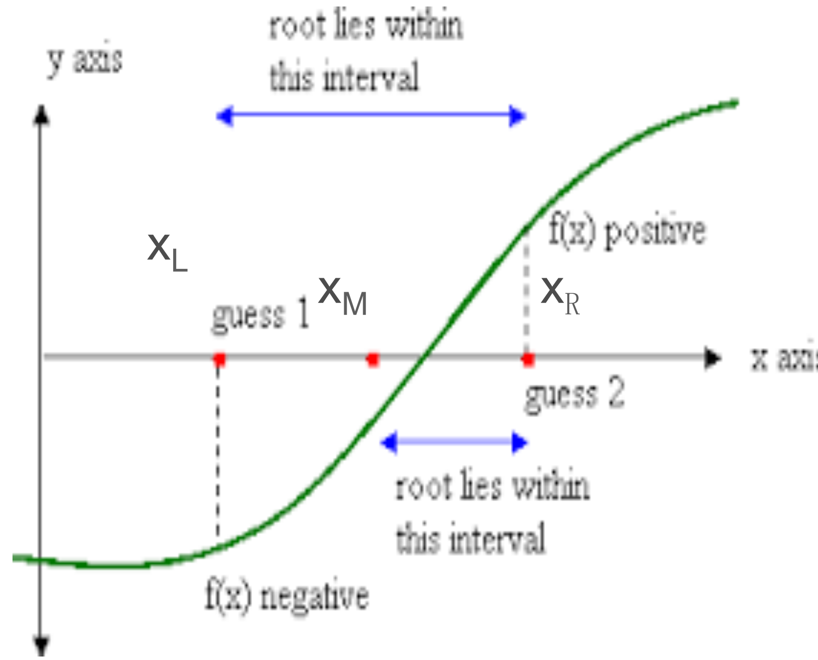
Bisection Method For Finding Roots

- Root of function f: Value x such that f(x)=0
- Many problems can be expressed as finding roots, e.g. square root of w is the same as root of $f(x) = x^2 – w$
- Requirement:
  - Need to be able to evaluate f
  - f must be continuous
  - We must be given points x_L and x_R such that $f(x_L)$ and $f(x_R)$ are not both positive or both negative

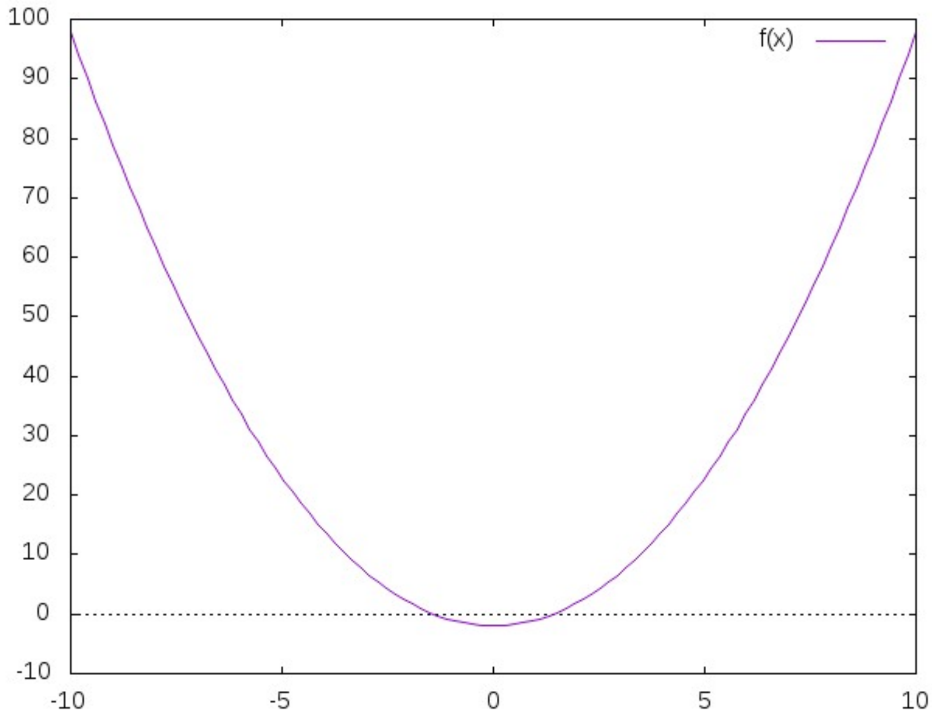
Bisection Method For Finding Roots

- Because of continuity, there must be a root between x_L and x_R (both inclusive)
- Let $x_M = (x_L + x_R)/2$ = midpoint of interval (x_L, x_R)
- If $f(x_M)$ has same sign as $f(x_L)$, then $f(x_M)$, $f(x_R)$ have different signs

  So we can set $x_L = x_M$ and repeat
- Similarly if $f(x_M)$ has same sign as $f(x_R)$
- In each iteration, x_L, x_R are coming closer.
- When they come closer than certain epsilon, we can declare x_L as the root



Bisection Method For Finding Square Root of 2



- Same as finding the root of $x^2 - 2 = 0$
- Need to support both scenarios:
  - xL is negative, xR is positive
  - xL is positive, xR is negative
- We have to check if xM has the same sign as xL or xR

Bisection Method for Finding √2

```
double xL=0, xR=2, xM, epsilon=1.0E-20;

// Invariant: xL < xR
while(xR – xL >= epsilon){      // Interval is still large
    xM = (xL+xR)/2;             // Find the middle point
    bool xMisNeg = (xM*xM - 2) < 0;
    if(xMisNeg)                 // xM is on the side of xL
        xL = xM;
    else xR = xM;               // xM is on the side of xR
    // Invariants continues to remain true
}
cout << xL << endl;
```

Newton Raphson method

- Method to find the root of f(x), i.e. x s.t. f(x)=0
- Method works if:

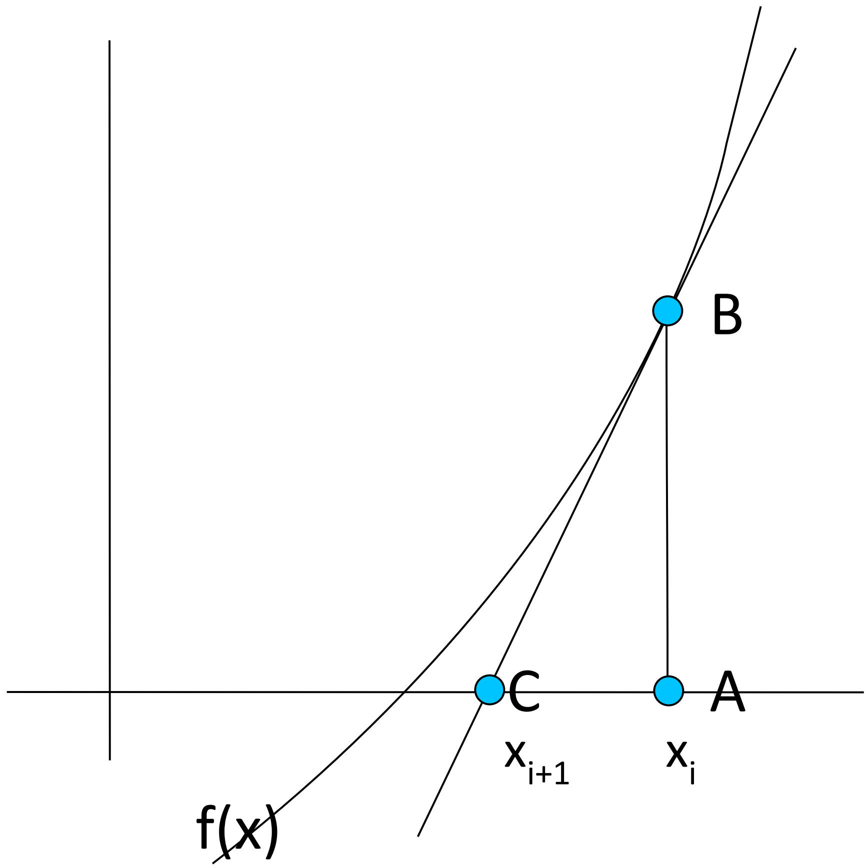
  f(x) and derivative f'(x) can be easily calculated

  A good initial guess x_0 for the root is available
- Example: To find square root of y

  use $f(x) = x^2 - y$. $f'(x) = 2x$

  f(x), f'(x) can be calculated easily. 2,3 arithmetic ops
- Initial guess $x_0 = 1$ is good enough!

How To Get Better x_{i+1} Given x_i



Point A =$(x_i,0)$ known

Calculate $f(x_i)$

Point B=$(x_i,f(x_i))$

Draw the tangent to f(x)

C= intercept on x axis

C=$(x_{i+1},0)$

$f'(x_i)$ = derivative

$= (d\, f(x))/dx$ at x_i

$\approx AB/AC$

$$x_{i+1} = x_i - AC = x_i - AB/(AB/AC) = x_i - f(x_i) / f'(x_i)$$

Square root of y

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

$f(x) = x^2 - y$, $\quad f'(x) = 2x$

$x_{i+1} = x_i - (x_i^2 - y)/(2x_i) = (x_i + y/x_i)/2$

Starting with $x_0=1$, we compute x_1, then x_2, …

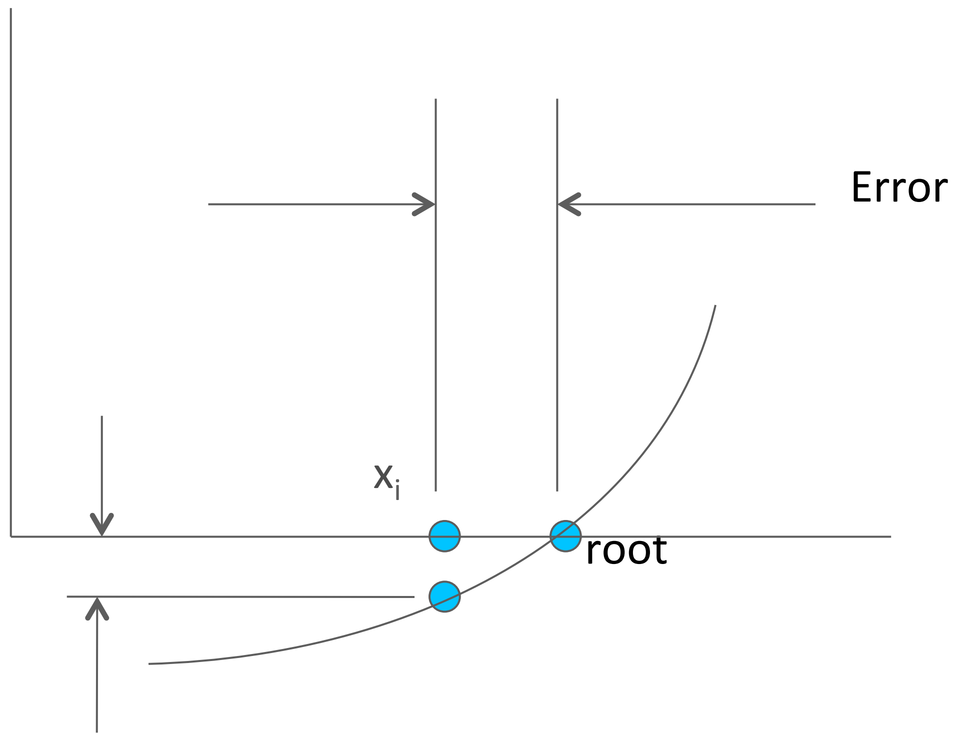
We can get as close to sqrt(y) as required

Proof not part of the course.

Computing $\sqrt{y}$ Using the Newton Raphson Method

```
float y;  cin >> y;

float xi=1;    // Initial guess. Known to work

repeat(10){  // Repeating a fixed number of times

    xi = (xi + y/xi)/2;

}

cout << xi;
```

How To Iterate Until Error Is Small



Error Estimate = $|f(x_i)| = |x_i * x_i - y|$

Make $|x_i*x_i - y|$ Small

```
float y; cin >> y;

float xi=1;

while(abs(xi*xi – y) > 0.001){

   xi = (xi + y/xi)/2 ;

}

cout << xi;
```

Concluding Remarks

If you want to find f(x), then

- use MacLaurin series for f, if f and its derivatives can be evaluated at 0
- Express f as an integral of some easily evaluable function g, and use numerical integration
- Express f as the root of some easily evaluable function g, and use bisection or Newton-Raphson

All the methods are iterative, i.e. the accuracy of the answer improves with each iteration