

Important Numerical Programs

Ajit Rajwade

Refer: Sections 5.7, 7.7, 8.3, 8.4 of the book by Abhiram Ranade

Numerical Programs Covered Here

1. Least squares line fitting
2. GCD computation given two positive integers
3. Bisection method for finding the roots of an equation
4. Newton-Raphson method for finding the roots of an equation

Least Squares Line Fitting

Ajit Rajwade

1. Least squares line fitting

- Suppose you are given some n points, each having an x coordinate and a y coordinate.
- Let the points be $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.
- Let us assume that these points **approximately** (but not exactly) lie on a line.
- Such situations are quite common, for example in experiments in a physics lab from your 12th grade.
- You want to find the best fit line $y = mx + c$ with slope m and x -intercept c .
- What is meant by best fit?
- The **squared vertical distance** from any point (x_i, y_i) to the line $y=mx+c$ is given by $(y_i - mx_i - c)^2$.
- We will try to find a line for which the sum total squared vertical distance for all n given points is the least, i.e. we find m, c to minimize:

$$J(m, c) = \sum_{i=1}^n (y_i - mx_i - c)^2$$

Least squares line fitting

- Taking derivatives of J w.r.t. slope m and intercept c , we get the following two linear simultaneous equations:

$$-2 \sum_{i=1}^n x_i (y_i - mx_i - c) = 0 \implies m \sum_{i=1}^n x_i^2 + c \sum_{i=1}^n x_i = \sum_{i=1}^n x_i y_i$$

$$-2 \sum_{i=1}^n (y_i - mx_i - c) = 0 \implies m \sum_{i=1}^n x_i + cn = \sum_{i=1}^n y_i$$

- Let us define a few quantities to reduce symbol clutter:

$$p \triangleq \sum_{i=1}^n x_i^2, q \triangleq \sum_{i=1}^n x_i, r \triangleq \sum_{i=1}^n x_i y_i, s \triangleq \sum_{i=1}^n y_i$$

- The solution for m and c is as follows:

$$m = \frac{nr - qs}{np - q^2}, c = \frac{ps - qr}{np - q^2}$$

Least squares line fitting: towards a program

- We are going to write a computer program for this.
- How do we get the points? It is tedious for a user to enter point coordinates manually, and definitely not a few dozen or hundred points!
- We will use the `simplecpp` package and ask the user to **click** on points on a canvas, with a mouse.
- Once the user is done clicking on the points, we will compute the **slope** and **intercept** of the best fit line.
- Finally, we will **draw** the line.
- The full program is on the next slide.
- You will learn a lot of new commands/functions in this program.

```
main_program{
int n;
double p,q,r,s;
p = q = r = s = 0.0;
```

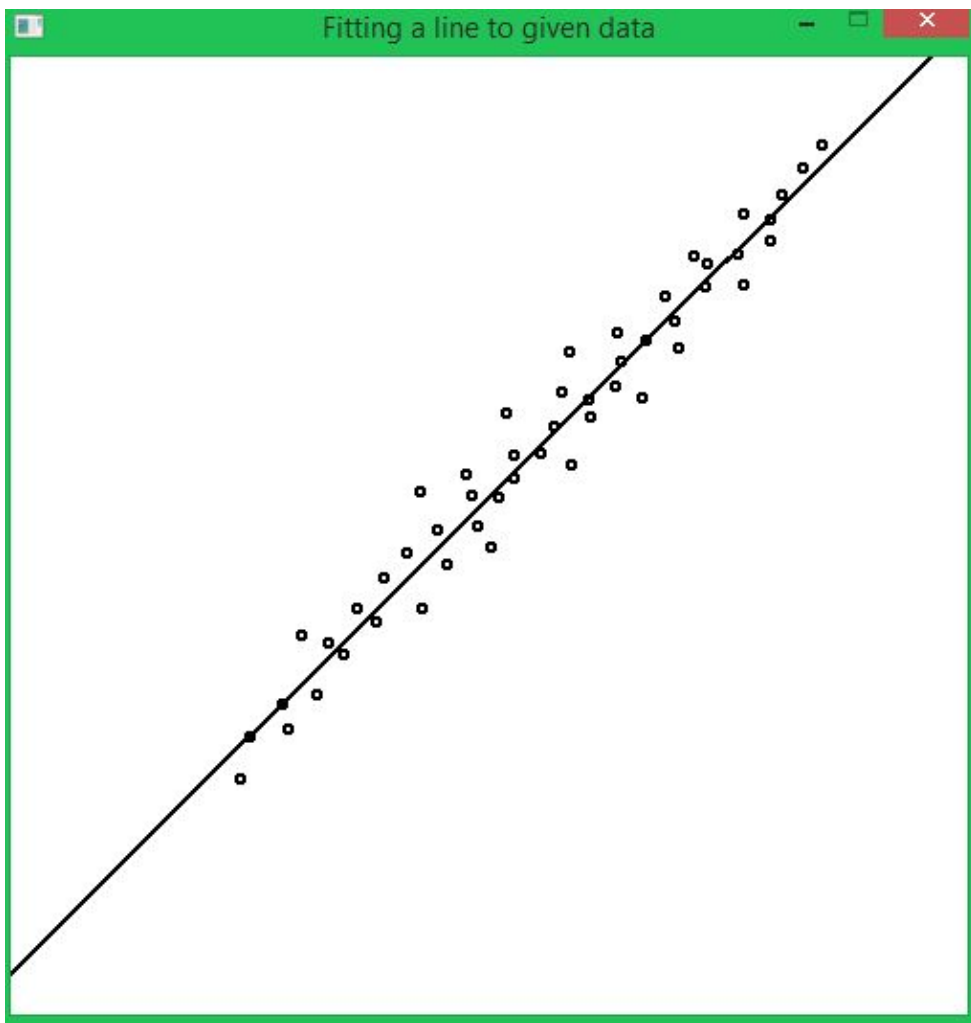
The position returned by `getClick()`; is given by an integer value $v = 65536x + y$. The x and y coordinates of the point where the mouse was clicked are given by $x = \text{floor}(v/65536)$ and $y = v \% 65536$.

```
cout << "Enter the number of points: "; cin >> n;
initCanvas ("Fitting a line to given data",500,500); // inbuilt function to create a new
// window

Circle pt(0,0,0); // this will be used to draw a small circle around each clicked point
for (int i = 0; i < n; i++)
{
    int cPos = getClick(); // ask the user to click on a point
    double x = cPos/65536; // x coordinate of the point
    double y = cPos%65536; // the point's y coordinate
    pt.reset(x,y,3); // draw a small circle around that point
    pt.imprint(); //to retain that circle even when we move pt for getting other points

    p += x*x; q += x; r += x*y; s += y;
}

double m = (n*r-q*s)/(n*p-q*q); // calculate slope
double c = (p*s - q*r)/(n*p - q*q); // calculate intercept
Line l (0,c,500,500*m+c); // draw the line
wait (15);
}
```



Sample Output

Least squares line fitting

- What we just did is called **least squares fitting**.
- In this case we fit a **line**. You could also fit a circle, parabola, etc.
- If instead of the squared vertical distance, we had minimized the **squared perpendicular distance**, it would be called **total least squares fitting**.
- Both least squares and total least squares are very important techniques in computer science and data science.

Greatest Common Divisors

Ajit Rajwade

Determining the greatest common divisor (GCD) of two numbers

- The GCD is the largest integer that divides both positive integers m , n .
- One strategy is to examine all numbers from 2 to $\min(m, n)$ and find the largest number that divides both.
- This method is correct, but slower than the method we will see later.
- Here is its code

```
main_program
{
    int m,n,i,min_mn;
    cout << "Enter the numbers "; cin >> m >> n;
    min_mn = min(m,n);
    for (i=min_mn; i >= 2; i--){
        if (m%i == 0 && n%i == 0) break;
    }
    cout << "Their GCD is: " << i;
}
```

Euclid's method for GCD

- Based on the following fact: If d is a common divisor of m and n , then d is also a divisor of $m-n$.
- Proof (assuming $m \geq n$):
 - Since d divides m , then $m = ad$. Since d divides n , then $n = bd$. Here a and b are two integers.
 - Hence $m-n = (a-b)d$.
 - As a and b are integers, $a-b$ is also an integer.
 - Thus we have proved that d divides $m-n$.
- Likewise, if d is a common divisor of $m-n$ and n , then it is also a common divisor of m and n . The proof is similar.
- Thus $\text{GCD}(m, n) = \text{GCD}(m-n, n)$.

Euclid's method for GCD

- Using the earlier fact, we have:
- $\text{GCD}(3977, 943) = \text{GCD}(3977 - 943, 943) = \text{GCD}(3034, 943) = \text{GCD}(2091, 943) = \text{GCD}(1148, 943) = \text{GCD}(205, 943)$
- You see that instead of running so many steps, we can also subtract all multiples in one shot leaving behind the remainder of dividing 3977 by 943.
- This can be formalized into the following fact: Let $m = nq + r$ where m, n, q, r are integers and q, r are the quotient and remainder obtained when dividing m by n . Then $\text{GCD}(m, n) = \text{GCD}(n, r)$.
- **Proof:** If d divides m and n , then it divides $nq + r$ and n . As d divides n , then it must divide r (otherwise it would not divide $m = nq + r$). Thus $\text{GCD}(m, n) = \text{GCD}(n, r)$. In other words $\text{GCD}(m, n) = \text{GCD}(n, m \% n)$.
- **Note:** if $r = 0$, we can directly state $\text{GCD}(m, n) = n$.
- We will use these facts to design the algorithm for GCD computation.

Euclid's method for GCD

- Using the aforementioned fact, we have $\text{GCD}(3977, 943) = \text{GCD}(3977 \% 943, 943) = \text{GCD}(205, 943)$.
- Continuing further, $\text{GCD}(205, 943) = \text{GCD}(205, 943 \% 205) = \text{GCD}(205, 123)$
- This is further equal to $\text{GCD}(123, 205 \% 123) = \text{GCD}(123, 82) = \text{GCD}(82, 123 \% 82) = \text{GCD}(82, 41) = 41$.
- We now write a piece of code for this.

Euclid's method for GCD

```
main_program
{
    int m,n,i;
    cout << "Enter the positive numbers (largest first): "; cin >> m >> n;
    while (m%n !=0)
    {
        int remainder = m%n;
        m = n;
        n = remainder; //if n does not divide m, GCD(m,n) = GCD(n,m%n)
    }
    cout << "Their GCD is: " << n; // if n divides m, GCD(m,n) = n
}
```

Let the original numbers be called m_0 and n_0 . Inside the while loop, after any iteration the GCD of m and n is equal to the GCD of m_0 and n_0 (as per our mathematical arguments in the previous slides). Also, at the end of each iteration, we always have $m > n > 0$.

The while loop is guaranteed to terminate in a finite number of iterations. Why? Because the value of n reduces to $m \% n$ which is clearly smaller than n . The value of m also reduces to n . Both m and n are guaranteed to remain positive (otherwise the while loop terminates)! The number of iterations cannot exceed n_0 ever (in fact it will terminate in **much fewer** iterations).

Bisection Method for Finding Roots of an Equation

Ajit Rajwade

Bisection Method for Root-finding

- The roots of a function $f(x)$ are defined as those values for which $f(x) = 0$.
- It is easy to find roots of certain specific functions, like quadratic functions, because of readily available formulae.
- However for most functions, computational methods are required to determine (approximate) roots, i.e. values of x for which $f(x)$ is close to zero (but maybe not exactly zero).
- The bisection method is one method to find the approximate root(s) of a continuous function $f(x)$.
- The method requires the user to specify an interval $[x_L, x_R]$ such that $x_L < x_R$ and $f(x_L)$ and $f(x_R)$ have opposite signs.
- By the intermediate value theorem, $f(x)$ must then have a root inside $[x_L, x_R]$.

Bisection Method for Root-finding

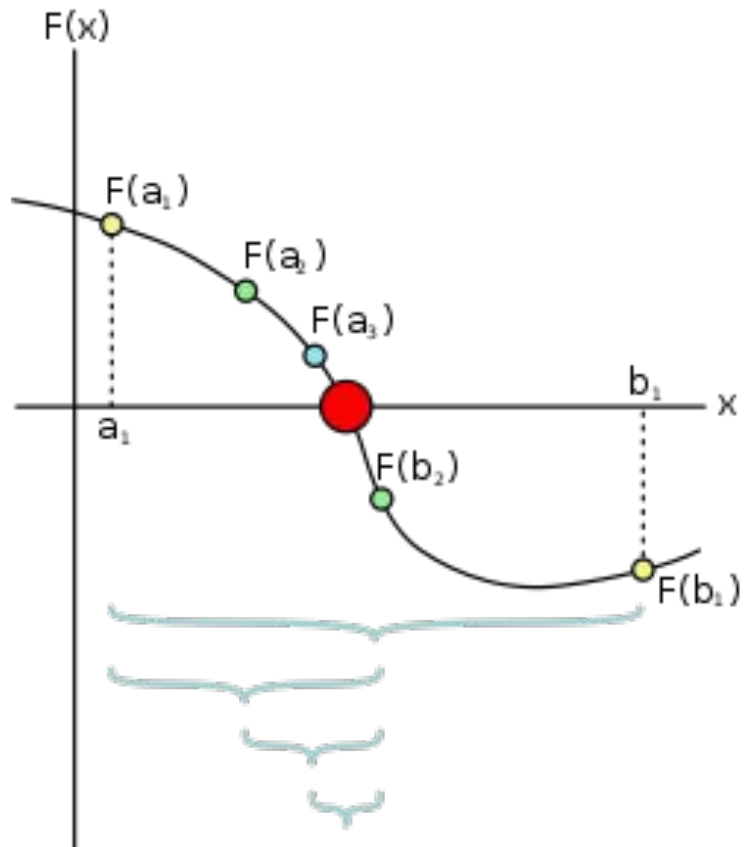
- By the intermediate value theorem, $f(x)$ must then have a root inside $[x_L, x_R]$.
- **Intermediate value theorem:** If $f(x)$ is a continuous function whose domain contains the interval $[a,b]$, then $f(x)$ must take on every value between $f(a)$ and $f(b)$ for some x in $[a,b]$.
- Corollary: If a continuous function has values of opposite sign inside an interval, then it has a root in that interval (**Bolzano's theorem**).

Bisection Method for Root-finding

- We can think of $x_R - x_L$ as the maximum possible error in determining the root.
- A better approximation basically means obtaining a smaller interval.
- If the interval length is really small, either endpoint can be treated as an approximate root.
- How can we shrink the interval given initial values of x_L , x_R ?
- Compute the midpoint $x_M = (x_L + x_R) / 2$.
- If $f(x_M)$ and $f(x_L)$ have different signs, then the root must lie in $[x_L, x_M]$, so you can set x_R to x_M .
- Instead if $f(x_M)$ and $f(x_R)$ have different signs, then the root must lie in $[x_M, x_R]$, so you can set x_L to x_M .
- Note that in either case, the new x_L , x_R satisfy the requirement that $f(x_L)$ and $f(x_R)$ have different signs.
- In both cases, the new interval has smaller width than the earlier one.
- This process needs to be continued in a loop until $x_R - x_L \geq$ some tolerance value 'epsilon', maybe something like 0.001 if you desire precision till the third decimal place.

Bisection Method for Root-finding

https://en.wikipedia.org/wiki/Bisection_method



Bisection method for roots of $f(x) = x^3 - 5$

```
main_program{
double xL = 1, xR = 2; // initial interval
double epsilon = 0.001; // precision of the root's value
bool flagL, flagR;
double xM;
bool flagM;
flagL = xL*xL*xL-5 > 0; flagR = xR*xR*xR-5 > 0; // signs of f(xL) and f(xR)
if (flagL != flagR)
{
    while (xR - xL > epsilon) // until the required precision is reached
    {
        xM = (xR+xL)/2; // interval midpoint
        flagM = xM*xM*xM-5 > 0;
        if (flagM != flagL) xR = xM; // signs of f(xM) and f(xL)
        else if (flagM != flagR) xL = xM;
    }
    cout << "the root is " << xL;
}
}
```

Bisection Method for Root-finding

- We could have simply divided the interval $[x_L, x_R]$ into a grid with $1/0.001 = 1000$ points
- And evaluated $f(x)$ at all those points, and chosen the value of x which produced the **smallest** value of $|f(x)|$ as the approximate root.
- But this would require 1000 function evaluations.
- In contrast, the bisection method is much **faster**.
- How many evaluations of $f(x)$ does it require? Equivalently, how many iterations does it require?
- To answer this, notice that the length of the interval is shrinking by a factor of 2 in each iteration.
- And we desire a precision of just about epsilon. In how many iterations (N) will the interval width fall below epsilon?
- $2^N = (x_R - x_L) / \text{epsilon}$, i.e. $N = \log_2((x_R - x_L) / \text{epsilon})$.
- Note: we have assumed that the interval $[x_L, x_R]$ contained only one root! If there are multiple roots in this interval, the method will output just one of the root and you will need to select a fresh interval for the others (why?)

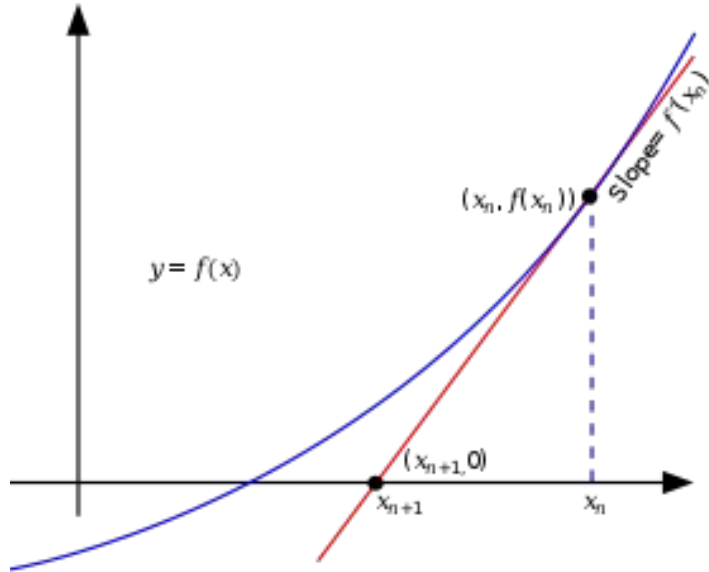
Newton-Raphson Method for Finding Roots of an Equation

Ajit Rajwade

Newton-Raphson Method for Root-finding

- This method starts from an initial guess for the root, call it x_0 , and keeps **evolving** the guess to make it better and better.
- It then **approximates** the function $f(x)$ at x_0 by means of a tangent, i.e. $f(x) \approx f(x_0) + (x-x_0)f'(x_0)$.
- So what should the next guess x_1 be?
- Since we took the **tangent** approximation of $f(x)$, let x_1 be the root of that approximate function.
- This gives $f(x_0) + (x-x_0)f'(x_0) = 0$, and $x_1 = x = x_0 - f(x_0)/f'(x_0)$.
- Of course the roots of $f(x)$ and its tangent approximation are **different**, so x_1 only acts as a guess for the root.
- This process is iterated **several** times producing guesses $x_0, x_1, \dots, x_i, \dots$ until when $f(x_i)$ is close to 0, i.e. less than or equal to $\epsilon = 0.001$ (say).

Newton-Raphson Method for Root-finding



https://en.wikipedia.org/wiki/Newton%27s_method

Newton-Raphson Method for finding roots of roots of

$$f(x) = x^3 - 5$$

```
main_program{  
  
double x, epsilon = 0.001;  
  
double dfx,fx;  
  
cout << "enter initial guess of root: "; cin >> x;  
  
while (fabs(x*x*x-5) > epsilon)  
{  
  
    fx = x*x*x-5;  
  
    dfx = 3*x*x;  
  
    x = x - fx/dfx;  
  
}  
  
cout << "The root is: " << x << "and the value of the function there is: " << fx;  
  
}
```

Newton-Raphson Method

- Unlike the bisection method, the Newton-Raphson method makes use of the first derivative of the function.
- It often works faster than the bisection method.
- But it is not always guaranteed to converge, i.e. end in a finite number of iterations.
- Also there is the problem of the derivative becoming zero.
- A full analysis of this method is beyond the scope of this course.