Two great algorithms

Abhiram Ranade

Outline

- Newton-Raphson method for finding roots
- Euclid's algorithm for finding greatest commond divisor (GCD).

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.

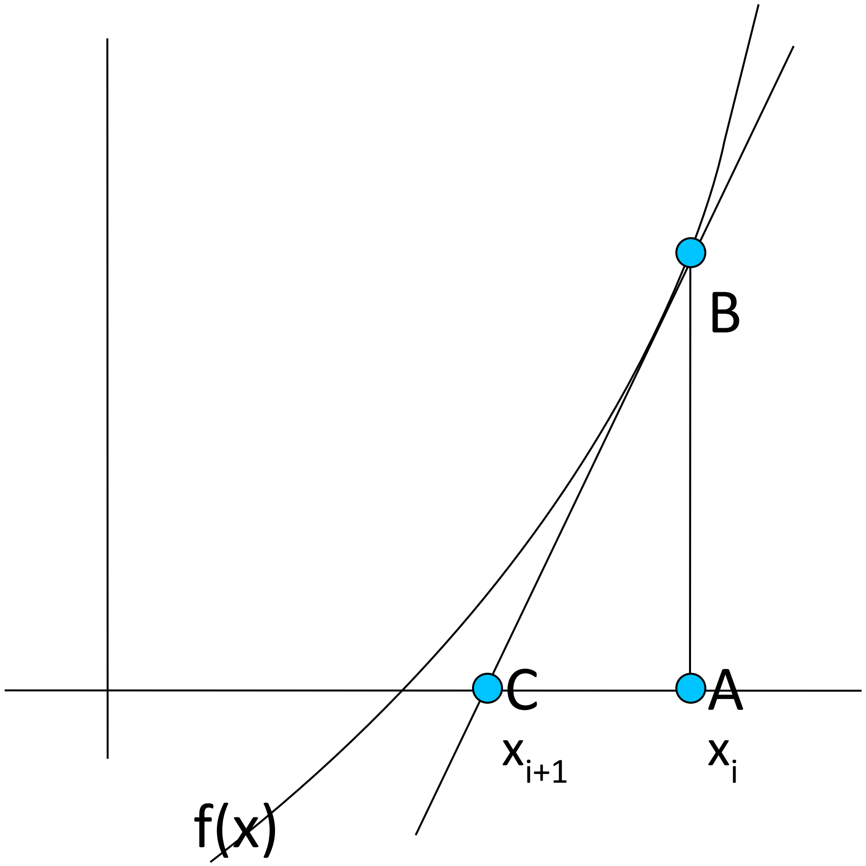
Example: To find square root of y.

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=$(x_i,f(x_i))$

Approximate f by tangent
C= intercept on x axis
C=$(x_{i+1},0)$

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

Square root of y

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

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

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

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

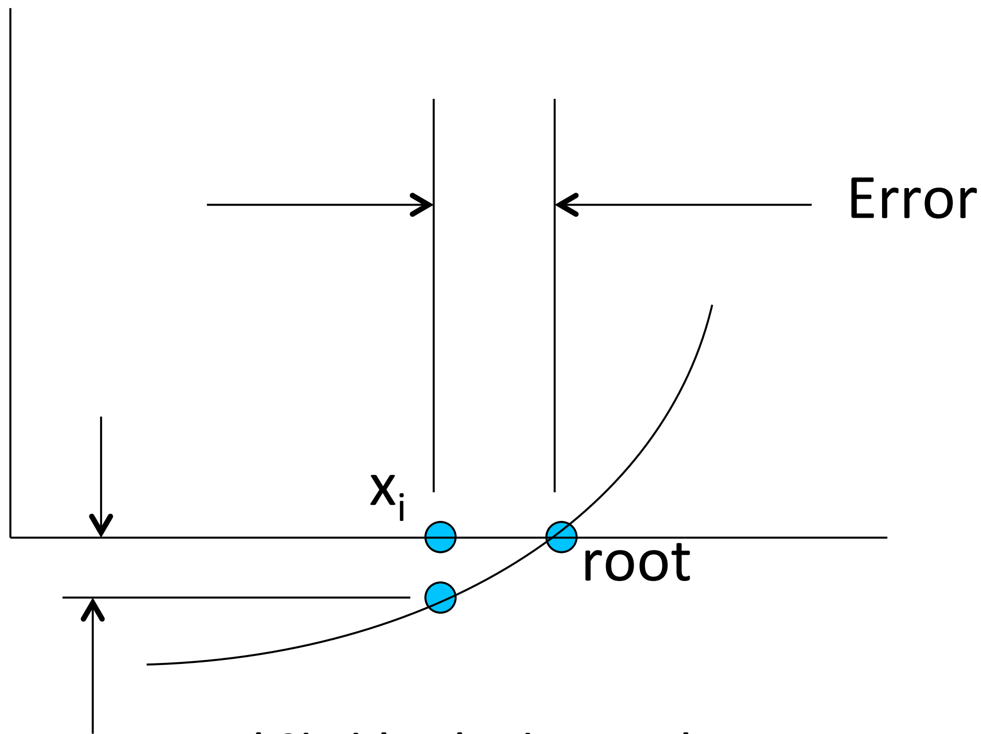
We can get as close to sqrt(y) as required.

Proof not part of the course.

Code

```
float y;  cin >> y;
float xi=1;      // Initial guess.  Known to work.
repeat(10){
   xi = (xi + y/xi)/2;
}
cout << xi;
```

Run 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;
```

Error Analysis

- Number of correct bits double with each iteration!
- Proof not in course.

“Clever” code using for

```
float xi, y;      cin >> y;
for( xi = 1 ;
      abs(xi*xi – y) > 0.001;
      xi = (xi + y/xi)/2
     ) {}
cout << xi;
// for has empty  body!
```

Remarks

- Very commonly used.
- Also useful in multiple dimensions. Given functions f, g, h, ... Find x, y, z, w, ... such that
  - f(x, y, z, w, ...) = 0
  - g(x, y, z, w, ...) = 0
  - h(x, y, z, w, ...) = 0.
  - . . .

But it is trickier too.

Greatest Common Divisor

Input: positive integers m, n.

Output: Largest integer dividing both.

Algorithm from specification: ?

Primary school algorithm: factorise numbers, multiply common factors.

Euclid's observation

If d divides m,n then d divides m-kn,n

for all integers k.

Converse implied.

Divisors of m,n = Divisors of m-kn, n

GCD(m,n) = GCD(m-kn,n)

.

Basic algorithm design principle

- Smaller problems are easier to solve than larger problems.
- Well, usually.
- Euclid's observation can be repeatedly applied to reduce numbers whose GCD we want.

Does the observation help?

GCD(3977,943)	
= GCD(3034,943)	
= GCD(2091,943)	Can we short circuit this process?
= GCD(205,943)	205 = 3977 % 943
= GCD(205, 123)	123 = 943 % 205
= GCD(82, 123)	82 = 205 % 123
= GCD(82, 41)	41 = 123 % 82
= 41	0 = 82 % 41

Algorithm idea

- Divide larger number by smaller.
- If remainder == 0, then GCD = smaller.
- Else repeat with smaller, remainder. (Note that the number that was smaller earlier is now larger, and the remainder is smaller)

Program

```
main_program{
  int Large, Small, Remainder;  cin >> Large >> Small;
  while(true){
    Remainder = Large % Small;
    if (Remainder == 0) break;
    Large = Small;
    Small = Remainder;
  }
  cout << "The GCD is: " << Small << endl;
}
```

Invariant

- Let GCD of the numbers given by the user be G. Then on any entry to the while loop, GCD (Large, Small) = G.
- Does this prove that the algorithm terminates or produces the correct answer?
- Small decreases in each iteration.
- Hence termination.

How many iterations are needed?

- $L_i >= S_i + R_i >= L_{i+1} + S_{i+1} >= 2L_{i+2}$
- Hence Larger halves every 2 iterations, or becomes even smaller
- $2\log_2$Large iterations suffice.

Twist

- Program works even if user types in smaller number first and larger second.

- Invariants also are same.
- Number of iteration proof: applies nearly.

- Proof: Exercise.