Applied Programming

Solving Nonlinear Equations

(Finding Roots)

Open Algorithms I

Root Refinement Methods

- Bracketing methods have the advantage of guaranteed convergence
  - But their rate of convergence is relatively slow

- Open methods exhibit rapid converge ***superlinear or quadratic***
  - But sometimes they do not convergence!

Open Methods

General Approach

- Start with an ***"initial guess"***
- Iteratively refine the guess to get closer to the "true" root

Warning

- Open methods ***may diverge*** (the algorithm may proceed down the "wrong path" going away from the root.
  - Need to know when that may happen!

Bracketing vs. Open Methods

Bracketing	Open
• ***Refine the interval*** in which root is contained	• ***Refine the value*** of the initial guess of the root
• ***Guaranteed to converge*** (as long as root in bracket)	• ***May diverge*** (converge only when "close" to the root)
• ***Slow*** Converge (linear)	• ***Fast*** convergence (superlinear, quadratic)

Main Open Methods

Fixed Point Iterations:

- ***Newton's method***
  - ***Quadratic*** Convergence

Other:

- ***Secant method***
  - ***Superlinear*** Convergence

Newton’s Method

- Uses information about the ***slope of the function*** (*e.g.*, its derivative) ***at the current point*** to refine the current root estimate.

- Convergence is guaranteed only when ***“close” to the solution***; otherwise it may diverge.
  - *Hard to tell how close is close enough*

Newton's Method in Action

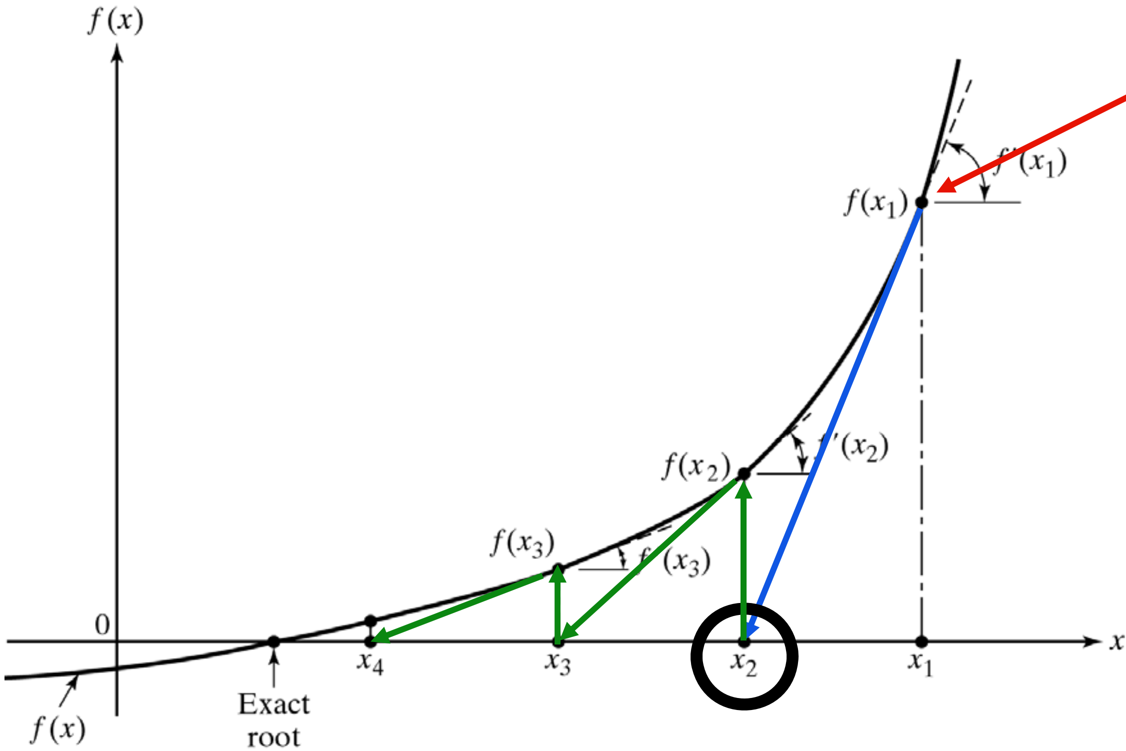


Figure 2.17
Newton's method.

- Start with an initial guess (x_1) of the root and
- Use tangent at current point to approximate *f(x)* locally
- Use intersection of tangent with y=0 to obtain next point.
- Repeat until accuracy is satisfied.

Note that this method uses the tangent as local information

Used with permission from support materials for *Applied Numerical Methods for Engineers and Scientists*, S. S. Rao, Prentice Hall, 2002.

Newton update derivation

line equation: y=mx+b and m (slope) is f'(x1)

At point x1 f(x1) = f'(x1)x1+b

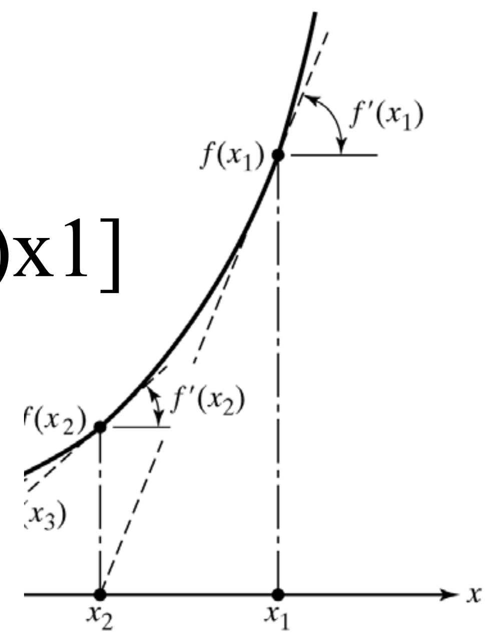
b = f(x1)-f'(x1)x1

At point x2, y is 0

0 = f'(x1)x2+[f(x1)-f'(x1)x1]

f'(x1)x2 = f'(x1)x1 -f(x1)

$$\boxed{x2 = x1 - f(x1)/ f'(x1)}$$



Newton's Method: Update Equation

- Update equation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The next guess equals the current guess less the evaluation of the ratio of the function and the derivative at the current guess.

Note: Requires that at the root *x** $f'(x^*) \neq 0$

and evaluation of ***the function and its derivative !***

Numerical Derivatives

Given: $f(v) = -1909 + 52.2\,v + 0.75\,v^2 - 0.02\,v^3$

then: $f'(v) = 52.2 + 1.5\,v - 0.06\,v^2$

Could be calculated:

```
double f[4] = { -1909.0, 52.2, 0.75, -0.02};
double df[3];
int i;
for (i = 1; i < sizeof(f)/sizeof(double); i++)
    { df[i - 1] = i*f[i]; }
printf("f(v)  = %f + %fv + %fv^2 + %fv^d3\n", f[0], f[1], f[2], f[3]);
printf("df(v) = %f + %fv + %fv^2\n", df[0], df[1], df[2]);
```

Note: Assumes low order first.

e.g. $a+bx+cx^2$

Newton's Method: Convergence

- Convergence: If the ***function is not "flat" (slope = 0) at the root*** and we start within δ (near) of the solution then the algorithm will converge.

Theorem: Let $x^* \in (a, b)$ be a root of $f(x)$, a twice continuously differentiable function on $[a, b]$. If $\boxed{f'(x^*) \neq 0}$ there exists a $\delta > 0$ such that for any $x_o \in [x^* - \delta, x^* + \delta]$ the sequence $\{x_n\}$ generated by Newton's algorithm converges to x^*.

Newton's Method: How Close is Close ?

- Newton's method converges if the initial guess is within δ of the desired root (in practice δ can be very small !)
- Atkinson showed that, for guaranteed convergence

$$\delta = \frac{1}{2}\frac{\max_{x\in\Omega}|f''(x)|}{\min_{x\in\Omega}|f'(x)|}, \qquad x^* \in \Omega$$

This means that the root must be bracketed in

$$\boldsymbol{\Omega \in [x^* - \delta, x^* + \delta]}$$

- So Newton is guaranteed to work if I'm close to the root, but I don't know the value of the root!

Convergence of Newton's method

- The proof of convergence shows that the ***order of convergence*** is at least ***quadratic*** near the root with *asymptotic error constant*

$$\eta = \frac{1}{2}\left|\frac{f''(x^*)}{f'(x^*)}\right|.$$

provided that the root is not repeated

Slow Convergence of Newton's method

If the function ***f(x)*** has a ***root x^* of multiplicity $m \geq 2$*** then $f'(x^*) = 0$ and

- The order of ***convergence drops to linear*** !
- Similarly, the rate of convergence becomes:

$$\mathcal{O}\left(\left(1 - \frac{1}{m}\right)^n\right)$$

(what does it mean ?)

Rate of Convergence of Newton's method for roots of high multiplicity

Multiplicity	Bisection	Newton
2	$\mathcal{O}((\frac{1}{2})^n)$	$\mathcal{O}((\frac{1}{2})^n)$
3	$\mathcal{O}((\frac{1}{2})^n)$	$\mathcal{O}((\frac{1}{3/2})^n)$
4	$\mathcal{O}((\frac{1}{2})^n)$	$\mathcal{O}((\frac{1}{4/3})^n)$
$\vdots$	$\vdots$	$\vdots$

Slower than bisection if *m*>2 !!

Newton's Algorithm Failures

- Slope of function away from root is a bad predictor (*diverges*)
- In theory we could get trapped in a *non-convergent cycle* (oscillates)
- *f ′(x$_i$)* very small (close to zero)

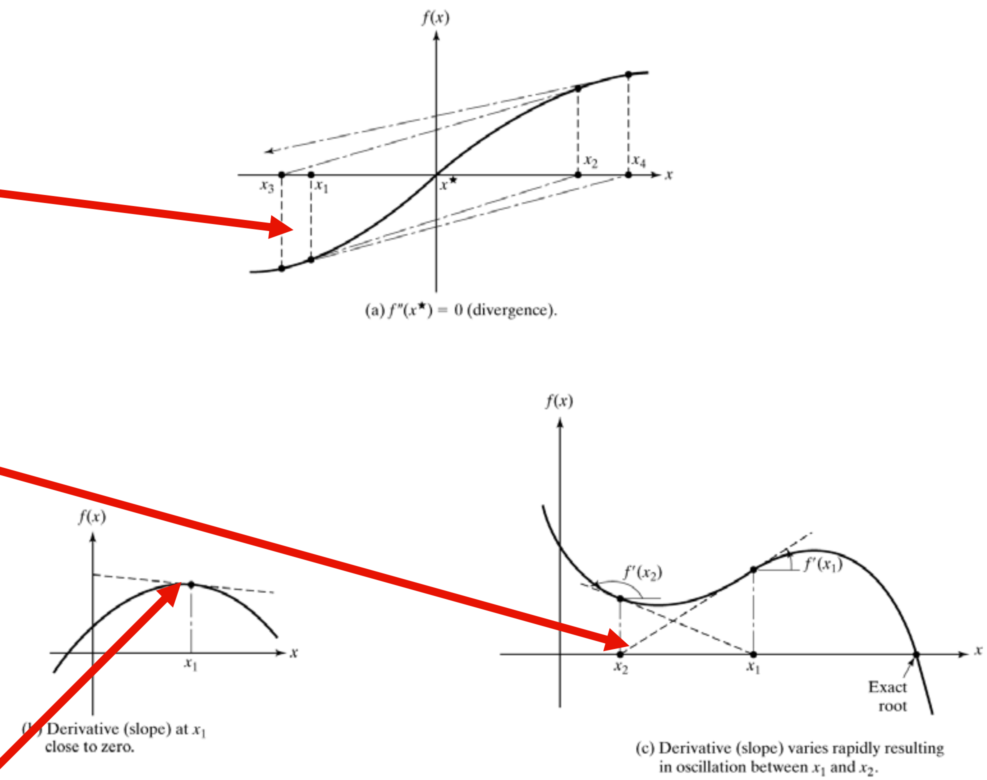


Figure 2.18
Non-convergence of Newton's method.

These could be avoided with a better initial guess !

Newton's Algorithm in a Nutshell

- Limitations:
  - Requires closed form expression for the ***function and its derivative***
  - Requires initial ***guess "close" to actual root***
    - No guaranteed convergence
  - Will fail if f'(x) is zero (or close to zero)
- Next guess: $x2 = x1 - f(x1)/ f'(x1)$
- Advantages:
  - + Convergence can be ***quadratic***
  - + Best approach when function and its derivative can be easily computed
  - + Can be ***extended to multivariable problems***

Implementation Notes

- ***For robustness***,
  - In practice, before applying any open method the root should be bracketed

  - ***To avoid divergence***, Newton's method is often ***combined with a bracketing*** method (such as Bisection). If properly designed, these hybrid algorithms have *guaranteed convergence and good convergence rate*

Example: Motor Speed

Tolerance= 0.05 RPM, Range: 0-50 Volts

Result: V=35.6856

The bisection took 10 iterations

```
» r= mynewton(@fmotor,@dfmotor,1,0.05)

Newton's Algorithm:
   k            x             f(x)                err
-------------------------------------------------------
   1        35.60235          1856.07          34.60235
   2        35.68528         2.450895        0.08293532
   3        35.68561      0.009545656      0.0003255508

r = 35.6856
```

The bisection method took 10 iterations!

Note: The roots of the cubic polynomial are:

35.685609864217469, 52.633113343145020, -50.818723207362368

Applied Programming

Solving Nonlinear Equations

(Finding Roots)

Open Algorithms II

Secant Method

- Newton's method achieves ***quadratic*** convergence at the expense of ***evaluating the derivative of the function***.

- The ***Secant method*** exhibits a ***superlinear*** convergence and ***does not require the computation of the derivative***
- Newton Update $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

is modified to $$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

- We switched to a Secant, is it legal?

Mean Value Theorem

- The Mean Value Theorem guarantees that there is at least one point on the graph of a continuous function at which the tangent is parallel to the secant

Theorem: Let $f(x)$ be continuous in $[a, b]$ and differentiable on (a, b). Then there exists a real number $\xi \in (a, b)$ such that

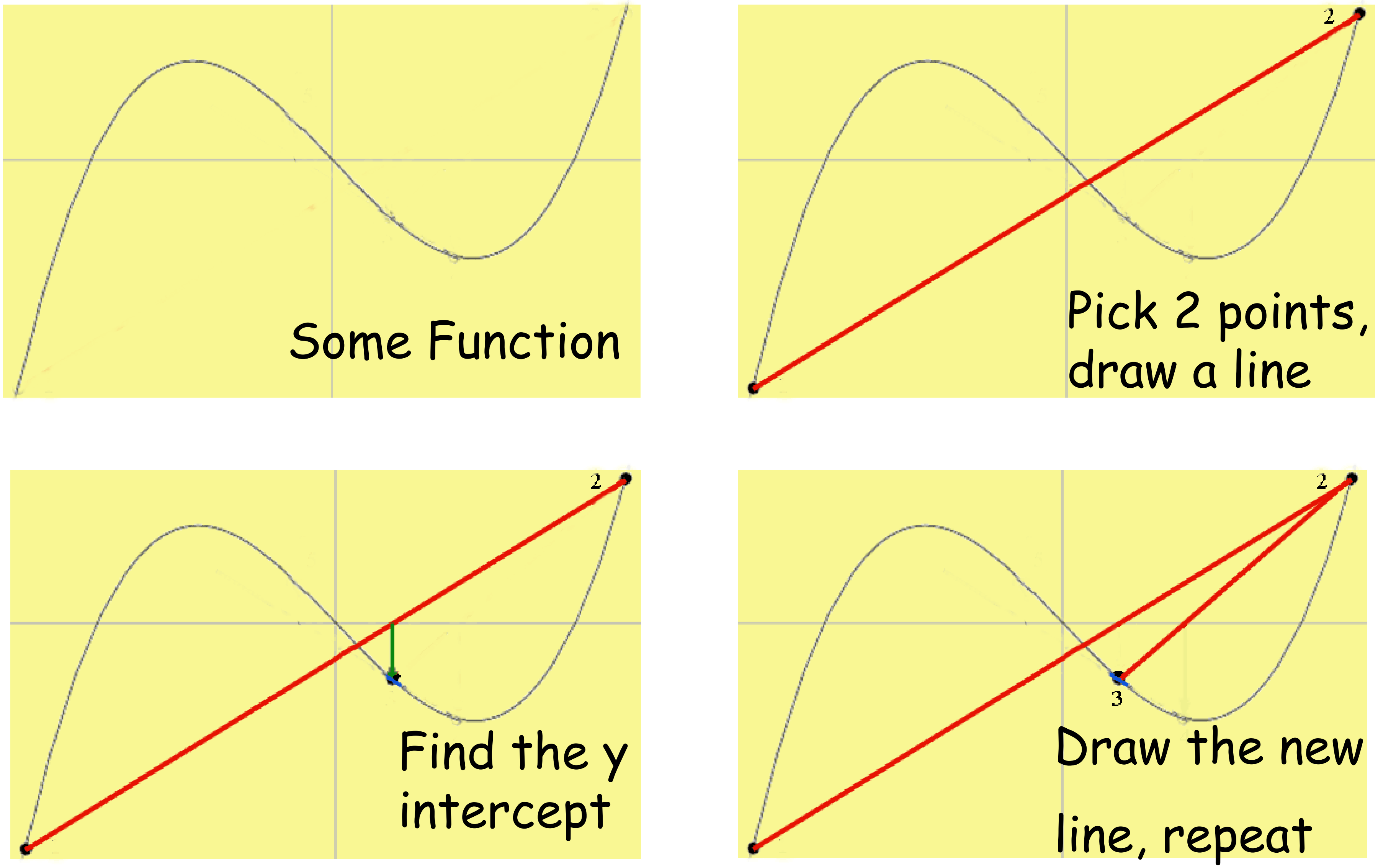
$$f'(\xi) = \underbrace{\frac{f(b) - f(a)}{b - a}}_{\text{slope of secant}}$$

Secant Method

- The secant method approximates the derivative by a secant line through the previous two iterates
  – Assumes the function is "sort of linear" in the area of interest

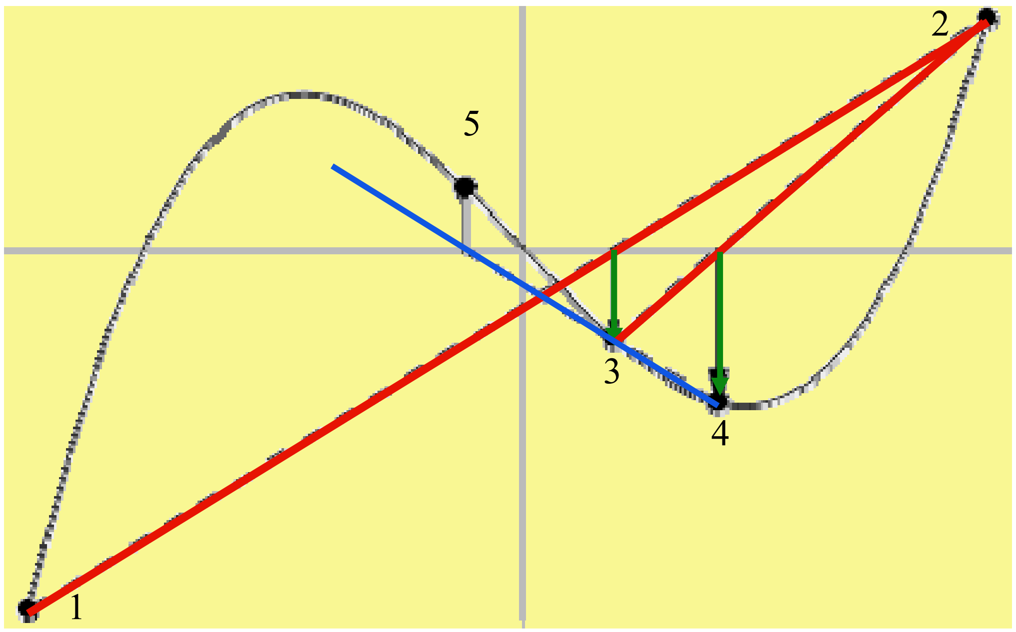
- Pick any 2 points on the curve
  – Initial guess does not need to bracket the root
  – draw a line
  – Find the zero intersect
  – Replace the first point with the new point
  – repeat

Secant: Method in Action



Secant: Method in Action

- The secant method (as its name says) approximates the derivative by a secant line through the previous two iterates



Picture from http://mathworld.wolfram.com/SecantMethod.html

Secant: Order of Convergence

- Recall that

$$|e_{n+1}| \approx \eta |e_n|^{\alpha}$$

- The ***order of convergence*** for the Secant method is

$$\boxed{\alpha = \frac{1+\sqrt{5}}{2} = 1.618\ldots = \varphi}$$

The Golden Ratio !

with asymptotic error constant

$$\eta \approx \left(\frac{1}{2} \left| \frac{f''(x^*)}{f'(x^*)} \right| \right)^{\alpha - 1}$$

Secant: Convergence

- Secant's method ***converges if*** the initial guess is ***sufficiently close to the desired root***

- Atkinson showed a ***sufficient condition*** for convergence:

$$\max\left\{\delta\left|x^{*}-x_{o}\right|,\delta\left|x^{*}-x_{1}\right|\right\}<1$$

where

$$\delta=\frac{1}{2}\frac{\max_{x\in\Omega}\left|f''(x)\right|}{\min_{x\in\Omega}\left|f'(x)\right|},\qquad x^{*}\in\Omega$$

δ is the same as in Newton's method

Exit Criteria

- ***How do we know when we are done?***
  - –If the difference between 2 successive "guesses" are less than the tolerance
  - –We have run "too many" iterations
  - –For Newton and Secant

Example: Motor Speed

Tolerance= 0.05 RPM, Range: 0-50 Volts

Result: V=35.6856

```
» r = mysecant(@fmotor,1,2,0.05)

Secant Algorithm:
   k            x               err
----------------------------------------------------
   1        35.17547          33.17547
   2        35.45992          0.284445
   3        35.68039         0.2204668
   4        35.68555       0.005168759

r = 35.6856
```

The bisection took 10 iterations

Newton took 3 iterations

Note: The roots of the cubic polynomial are:

35.685609864217469, 52.633113343145020, -50.818723207362368

Summary: Open Methods

- For convergence, algorithms require that the ***initial guess*** be ***close to the root***.
- ***Newton***'s method converges ***quadratically to simple roots*** and linearly to roots of higher multiplicity (>1)
- Newton's method requires evaluation of the ***function and its first derivative***
- ***Secant***'s method converges ***superlinearly to simple roots*** and linearly to roots of higher multiplicity (>1)
- Secant method requires evaluation ***only of the function***

Exercise 1

- What are the key differences between Newton's method and the Secant method?
  - Newton requires the derivative, secant does not

- How are they similar?
  - Both are using the slope to compute the next point
  - Both terminate when the Δ of two answers is small OR after some iteration limit.
  - Both may never converge

Exercise 2

- What is Newton method in a nutshell?
  - Start with an guess
  - Compute the tangent to find the y=0 intersect.
  - Use that new point as the next guess
  - Repeat until accuracy is satisfied.

Exercise 3

- What is Secant method in a nutshell?
  - Start with an guess and a 2nd point for Secant
    - Don't worry about function signs
  - Compute the secant to find the y=0 intersect.
  - Use that new point as the next guess
  - Repeat until accuracy is satisfied.

- Practical Programming Problems
  - Storing polynomials
  - Evaluating polynomials
  - Unfortunate Integers

Storing Polynomials Numerically

e.g. $f(v) = 0.02v^3 - 0.75v^2 - 52.2v + 1909$

- Polynomials are stored as array coefficients
- Can be stored **high** power first
  - **int x[4] = { 0.02, -0.75, -52.2, 1909};**
- Or stored **low** power first
  - **int x[4] = {1909, -52.2, -0.75, 0.02};**
  - Low is nice because index values and coefficient weights match.
    - E.g. x[3] is the coefficient for v^3
- Either will work but you must keep track!

Evaluating Polynomials Numerically

- A polynomial of degree n is a function of the form

$$P_n(x) = \sum_{k=0}^{n} a_k x^k$$

e.g. **$f(v) = 0.02v^3 - 0.75v^2 - 52.2v + 1909$**

- An simple evaluation will compute each term and accumulate the sum. e.g.

```
sum=0
for k=0:n
 sum+=a[k]*b^k
end
```

Note: Assumes Low order first

Simple Evaluation Issues

```
sum=0
for k=0:n
 sum+=a[k]*b^k
end
```

- Notice that we raise the independent variable "b" to the power "k" in the loop.
  - This introduces an excessive number of floating point evaluations
- e.g. b^5 = is really b*b*b*b*b
  - FIVE floating point operations
  - Sloooow…..

Horner's Factorization

- ***Rewrite*** the equation using a minimum of multiplications by factoring out the independent variable.

  $$f(v) = 0.02v^3 - 0.75v^2 - 52.2v + 1909$$

  $$f(v) = (0.02v^2 - 0.75v - 52.2)v + 1909$$

  $$f(v) = ((0.02v - 0.75)v - 52.2)v + 1909$$

- "Rewrite" in **Horner's form**

  $$f(v) = ((0.02v - 0.75)v - 52.2)v + 1909$$

- All calculations going forward will use Horner's Factorization

Horner's Factorization

- ***Rewrite*** the equation using a minimum of multiplications.

$$P_3(x) = \sum_{k=0}^{3} a_k x^k = x(x(a_3 x + a_2) + a_1) + a_0)$$

- All calculations going forward will use Horner's Factorization

```
sum=a[n];
for k=n-1:-1:0
  sum = sum*b + a[k];
end
```

Note that we count down from n-1

Example

- Simple cubic polynomial (motor example)

  f(v) = 0.02v³ - 0.75v² - 52.2v + 1909

- "Rewrite" in **Horner's form**

  f(v) =((0.02v - 0.75)v - 52.2)v + 1909

- Evaluate it ***from the inner most pair of parenthesis*** "outwards"

  Let **p[0]=1909,p[1]=-52.2,p[2]=-0.75,p[3]=0.02**

```
f_at_v = p[n];
for k=n-1:-1:0
  f_at_v = f_at_v * v + p[k];
end
```

Note: Low order first

Other “C” math bugs

- C loves integers and tries to use them when it can

Consider:

```
float x = 3.0;
float f1, f2;
f1 = sin((30/53)*x);
f2 = sin((30.0/53.0)*x);
printf("%f %f\n", f1, f2);
```

- **f1 != f2**

“C” calculates **INTEGER** 30/54 and generates 0
“C” then converts to 0.0*x

“C” calculate **FLOAT** 30.0/54.0 and generates 0.56*x

Linked List HW 4

- Container class structure for the linked list
  - keeps a counter of the size of the linked list
  - Points to the start and end of the ACTUAL linked list

```
typedef struct LinkedLists  {
  /* Number of elements in the list */
  int NumElements;
  /* Pointer to the front of the list of elements, possibly NULL */
  struct LinkedListNodes *FrontPtr;
  /* Pointer to the end of the list of elements, possibly NULL */
  struct LinkedListNodes *BackPtr;
 } LinkedLists;
```

Linked List HW 4

- The linked list structure for individual nodes
  - Actual data is NOT in the node, only a pointer

```
typedef struct LinkedListNodes   {
  /* The user information field, the pointer to the actual data */
  ElementStructs *ElementPtr;
  /* Link pointers to OTHER notes*/
  struct LinkedListNodes *Next;
  struct LinkedListNodes *Previous;
 } LinkedListNodes;
```

Linked List HW 4

- LinkedLists starts with nothing in the list
  - NumElements is zero
  - FrontPtr, BackPtr point to nothing (null)

Add the FIRST element

- Allocate space for a "LinkedListNodes"
- LinkedLists "FrontPtr" and "BackPtr" point to this new node
- NumElements is set to 1
- Allocate space for the associated ElementPtr
- Next and Previous point to nothing.
- Copy the data

Add the NEXT

- Allocate space for a “LinkedListNodes”
- NumElements is incremented
- Allocate space for the associated ElementPtr

- Adding to the “front” or “back” of the list
  - adding to the “back” or “front” of a linked list is defined by the structure pointer relationship.
  - E.g . If I have the data “A B C”, adding each line to the “back” of the link linked list will result in a linked list (starting from the front) of “A B C”

- Copy the data
- Fix the previous/next, last/first pointers based on your implementation