Summary Lecture 4

- Today: function calculation: roots, derivatives (ch 5, ch 9)
- Summary Lecture 4: Chapter 5, function calculation
  - power series: range of convergence
    - Continued fractions
      - Different ranges of convergence; usually very fast. Used for instance to calculate millions of decimals of pi.
    - Acceleration via Euler transformation – forward differencing operator
      - Resumming original series. Forward differencing operator used in many algorithms
    - Clenshaw recurrence formalism
      - Use the recurrence of an existing series to recalculate the sums
      - Recurrence : stability may be problematic
      - Can be tested
    - Chebyshev: function approximation
      - Polynomials with alternating zeros and minima/maxima
      - All minima/maxima equal +/- 1
      - Very close to minimax solution (best polynomial of given order to approach an arbitrary function
      - Function approximation is exact at the roots of the highest-order polynomial
      - Methods to give derivative and integral based on same coefficients.

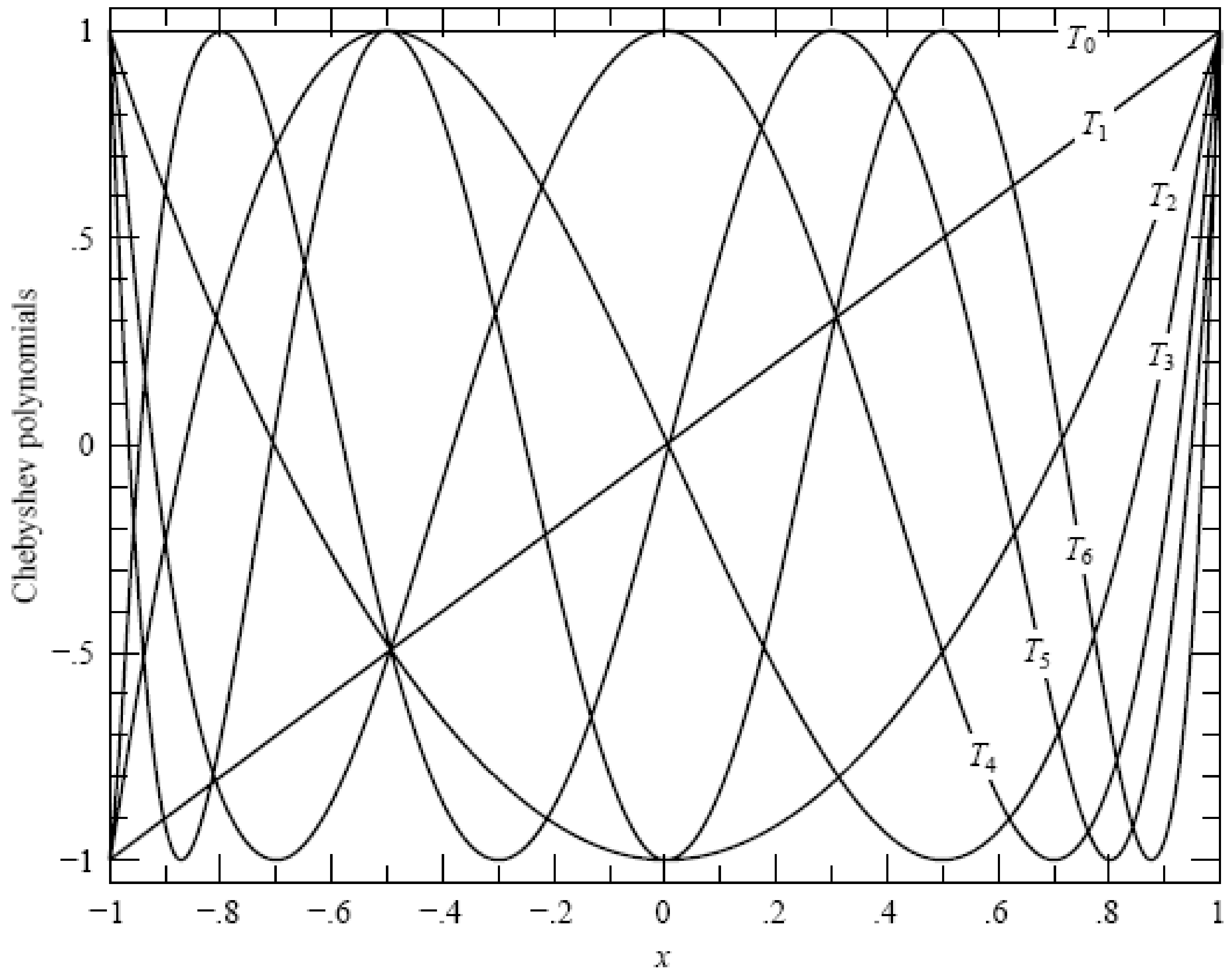


Figure 5.8.1. Chebyshev polynomials $T_0(x)$ through $T_6(x)$. Note that T_j has j roots in the interval $(-1, 1)$ and that all the polynomials are bounded between ± 1.

Using Chebyshev

#include chebyshev.h

- Chebyshev object is created with:

  `Chebyshev cheb(func,xlow,xhi,order)`

  - Calculate a function value a=func(x) using Chebyshev approximation to order order (order <= the order at creation):

    `a = cheb.eval(x,order);`

  - Make a Chebyshev object to calculate the derivative:

    `Chebyshev der = cheb.derivative();`

  - To calculate primitive:

    `Chebyshev integ = cheb.integral();`

Numerical derivatives+roots

- Quadratic roots:

  errors in roots of quadratic equation:

  $$ax^2 + bx + c = 0$$

  – Problem when a or c is small (compared to b)
- Analytically, you can get the roots in 2 ways:

Quadratic roots

- errors in roots of quadratic equation:

$$ax^2 + bx + c = 0$$

  - Problem when a or c is small (compared to b)
- Analytically, you can get the roots in 2 ways:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}}$$

- Right procedure:

Quadratic roots

- errors in roots of quadratic equation:

$$ax^2 + bx + c = 0$$

  - Problem when a or c is small (compared to b)

- Analytically, you can get the roots in 2 ways:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}}$$

$$q = -\frac{1}{2}\left(b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}\right)$$

$$x_1 = \frac{q}{a}, \quad x_2 = \frac{c}{q}$$

- Right procedure:

Minimal uncertainty: calculate roots with the term close to 2b, not close to 0

Cubic roots

Cubic roots $$x^3 + ax^2 + bx + c = 0$$

$$define: \quad Q = \frac{a^2 - 3b}{9}, \quad R = \frac{2a^3 - 9ab + 27c}{54}$$

$$if \quad R^2 < Q^3: \quad \text{3 real roots}$$

$$\theta = \arccos\left(\frac{\mathrm{R}}{\sqrt[3]{\mathrm{Q}}}\right), \quad x_1 = -2\sqrt{Q}\cos\frac{\theta}{3} - \frac{a}{3},$$

Francois Viete, 1615!

$$x_2 = -2\sqrt{Q}\cos\frac{(\theta - 2\pi)}{3} - \frac{a}{3}, \; x_3 = -2\sqrt{Q}\cos\frac{(\theta + 2\pi)}{3} - \frac{a}{3},$$

Complex roots: procedure outlined in Ch. 5

Numerical derivatives

$$f'(x) = \underset{h\to 0}{Lim}\frac{f(x+h)-f(x)}{h}, \quad f'(x) \approx \frac{f(x+h)-f(x)}{h}$$

Numerical derivatives

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}, \quad f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- correctly implemented?
- round off error; truncation error

$$f(x+h) \approx f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(x)$$

$$\frac{f(x+h) - f(x)}{h} \approx f'(x) + \frac{h}{2} f''(x) + \frac{h^2}{6} f'''(x)$$

Numerical derivatives

$$f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}, \quad f'(x) \approx \frac{f(x+h)-f(x)}{h}$$

- correctly implemented?
- round off error; truncation error

$$f(x+h) \approx f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(x)$$

$$\frac{f(x+h)-f(x)}{h} \approx f'(x) + \frac{h}{2} f''(x) + \frac{h^2}{6} f'''(x)$$

- x , h, and x+h are not exactly representable in doubles!
- better: temp=x+h; h = temp-x;
- both numbers are represented exactly! h is the bitwise exact difference between x and x+h
- Take care, that the compiler does not optimize this step away!

Numerical derivatives

- Always declare h via x_2=x+h, h=x_2-x in order to get rid of roundoff error in h and (x+h) - x
- Roundoff and truncation errors in f'(x) :

$$\varepsilon_{round-off} \approx \varepsilon_{func}\frac{f}{h}, \quad \varepsilon_{trunc} \approx hf''(x)$$

  – [illegible]$_{func}$, [illegible]$_{machine}$?)
  – Optimal choice of h?

  – $$\varepsilon_{func}\frac{f}{h} + hf'' \text{ is minimal:} \quad h \approx \sqrt{\frac{\varepsilon_{func} f}{f''}} = x_{curv}\sqrt{\varepsilon_{func}}$$

Numerical derivatives

$$\varepsilon_{round-off} \approx \varepsilon_{func} \frac{f}{h}, \quad \varepsilon_{trunc} \approx hf''(x)$$

$$\varepsilon_{func} \frac{f}{h} + hf'' \text{ is minimal: } \quad h \approx \sqrt{\frac{\varepsilon_{func} f}{f''}} = x_{curv} \sqrt{\varepsilon_{func}}$$

$$\frac{\varepsilon_{trunc} + \varepsilon_{roundoff}}{f'} \approx \sqrt{\varepsilon_{func} \frac{2ff''}{f'f'}} \approx \sqrt{\varepsilon_{func}}$$

- so relative precision is in general AT BEST the square root of the machine accuracy: ~ 1e-4 for float ~ 1e-8 for double precision.

Numerical derivatives

- Two function calls:
- f'(x)={f(x+h)-f(x-h)}/2h : truncation error is h*h*f'''
- optimal h is now h~(e_{func}*f/f''')1/3
- fractional error ($e_{roundof}$f+ e_{trunc})/f' =
- = {(e_{func}f)2/3 f''' 1/3}/f' ~ e_{func} 2/3
- choose right h: correct power of $e_{machine}$*typical scale x

Numerical derivatives

- Two function calls:
- f'(x)={f(x+h)-f(x-h)}/2h : truncation error is h*h*f'''
- optimal h is now h~(efunc*f/f''')1/3
- fractional error (eroundoff+ etrunc)/f' =
- = {(e_{func}f)2/3 f''' 1/3}/f' ~ e_{func} 2/3
- choose right h: correct power of $e_{machine}$ *typical scale x

$$Example: f(x)=x^{4/3}\ at\ x=27$$

$$e_{funct}=10^{-16}, f(27)=81, f'(27)=4, f'''(27)=\frac{-8}{6661}$$

$$h=(\frac{81*6661*10^{-16}}{8})=1.9*10^{-4}$$

$$\frac{f(x-h)-f(x+h)}{2\mathrm{h}}\approx 4\pm 2.5*10^{-11}$$

$$e_{trunc}=h^2 f''=5*10^{-11}$$

Numerical derivatives

- Higher precision?: Richardsons deferred approach to the limit: (Romberg for integration, Ridders method for differentiation : NR dfridr.)
  - assumes analyticity
  - probe function around x
  - typically about 10 calculations of f; starting at large h and making h progressively smaller following Neville's algorithm to find optimal answer
- Other alternatives: fitting (discrete values of f); function approximation (Chebyshev)

roots- solving equations

$$\vec{f}(\vec{x})=0$$

- Equations can be written as f(x)=0
  - multidimensional?
  - possibly no solutions
  - possibly infinite amount of solutions
  - much harder than singular case
- Singular equation: root finding f(x)=0
  - root finding is tough. Analysis of problem is crucial in many cases, to get good starting point (proximity of the root)
  - multiple/no roots (quadratic) ?
  - poles?
  - close-by roots (sin 1/x) ?
  - incredibly small area? e.g. pi*x*x*ln(|x-pi|)<0 for |x-pi|<1e-15
- one dimension: bracket root, hunt it down.

roots

- a root is bracketed in in interval [a,b] if f(a)*f(b)<0.
- for continuous functions, at least 1 root is guaranteed then.
- try to bracket root: e.g. start with interval [x1,x2] if f(x1)*f(x2)<0, you have bracketed the interval. Else, if |f1|<|f2|, replace interval with e.g. [x1-1.6*(x2-x1),x2] or with [x1,x2+1.6*(x2-x1)] in the case that |f2|<|f1|
- If bracketed: bi-section is possible method:
  - halve the interval by calculating f(midpoint).
  - this cannot fail
  - converges on root or pole, but slowly :

$$\varepsilon_{n+1} = \varepsilon_n / 2$$

Bisection and secant methods

- Bisection is said to converge linearly. If a method converges with $\varepsilon_{n+1} = c(\varepsilon_n)^m \quad m > 1$ then it converges "super-linearly".
- How small should the interval become?
  - floating point representation: it may be that the function never evaluates to 0.
  - interval of ~ 1e-8 may be reasonable for x O(1), but not for x O(1e9).
  - relative size region 1e-8*x : fine except close to x=0.
- Secant method/ False position method
  - finds singular roots generally faster
  - linear extrapolation between [x1,x2] to find 0-crossing
  - calculate x[3] at that crossing point, discard x1 or x2.

Secant

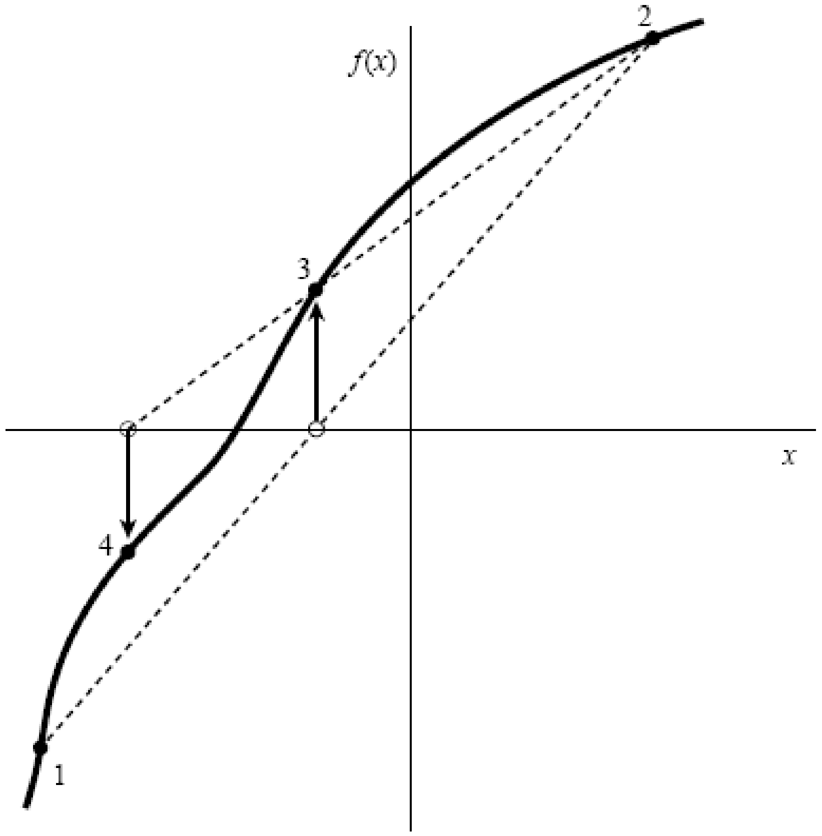


Figure 9.2.1. Secant method. Extrapolation or interpolation lines (dashed) are drawn through the two most recently evaluated points, whether or not they bracket the function. The points are numbered in the order that they are used.

False position

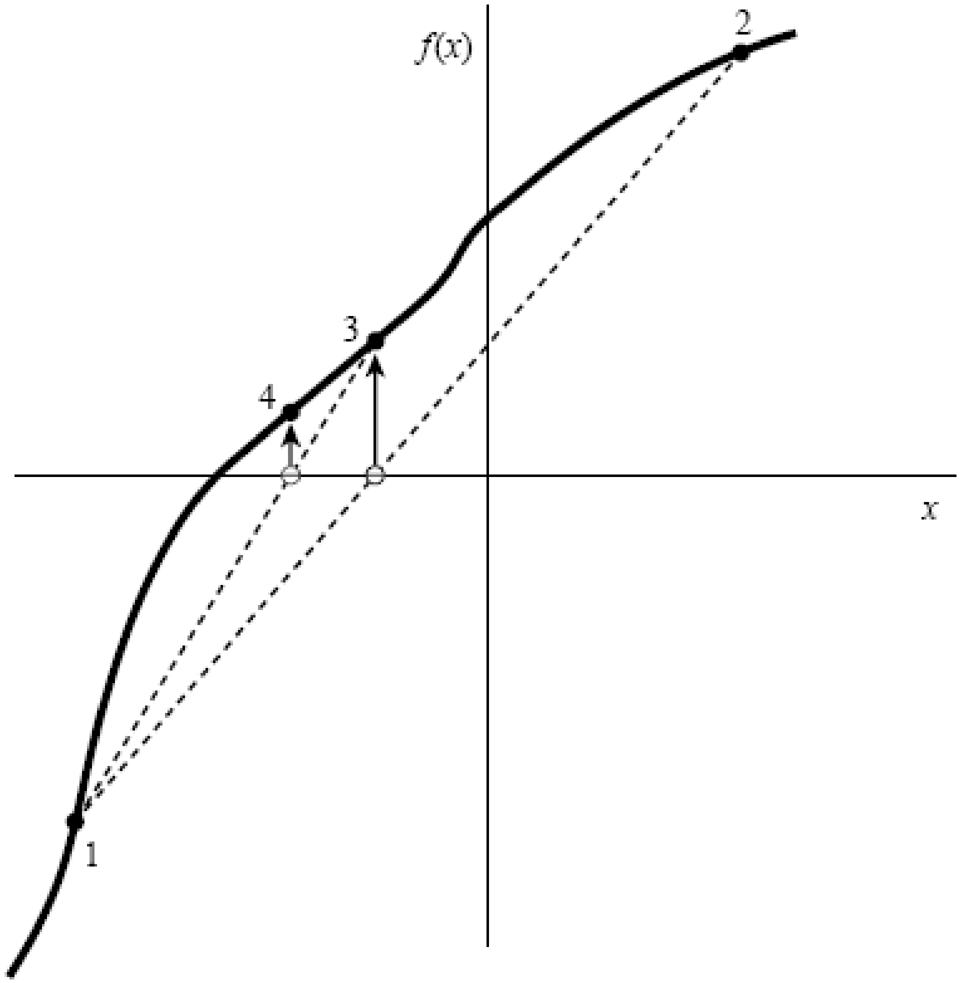


Figure 9.2.2. False position method. Interpolation lines (dashed) are drawn through the most recent points *that bracket the root*. In this example, point 1 thus remains “active” for many steps. False position converges less rapidly than the secant method, but it is more certain.

Secant and False position

- Secant method: typically it converges super-linear with m close to the golden ratio 1.618… :

$$\epsilon_{n+1} \approx \epsilon_n^{1.618\ldots.}$$

- However, root does not remain bracketed. In some cases, a step may take it towards infinity.
- False position method: generally superlinear convergence with lower power.
  - In some cases, these methods fail :

Secant and False position

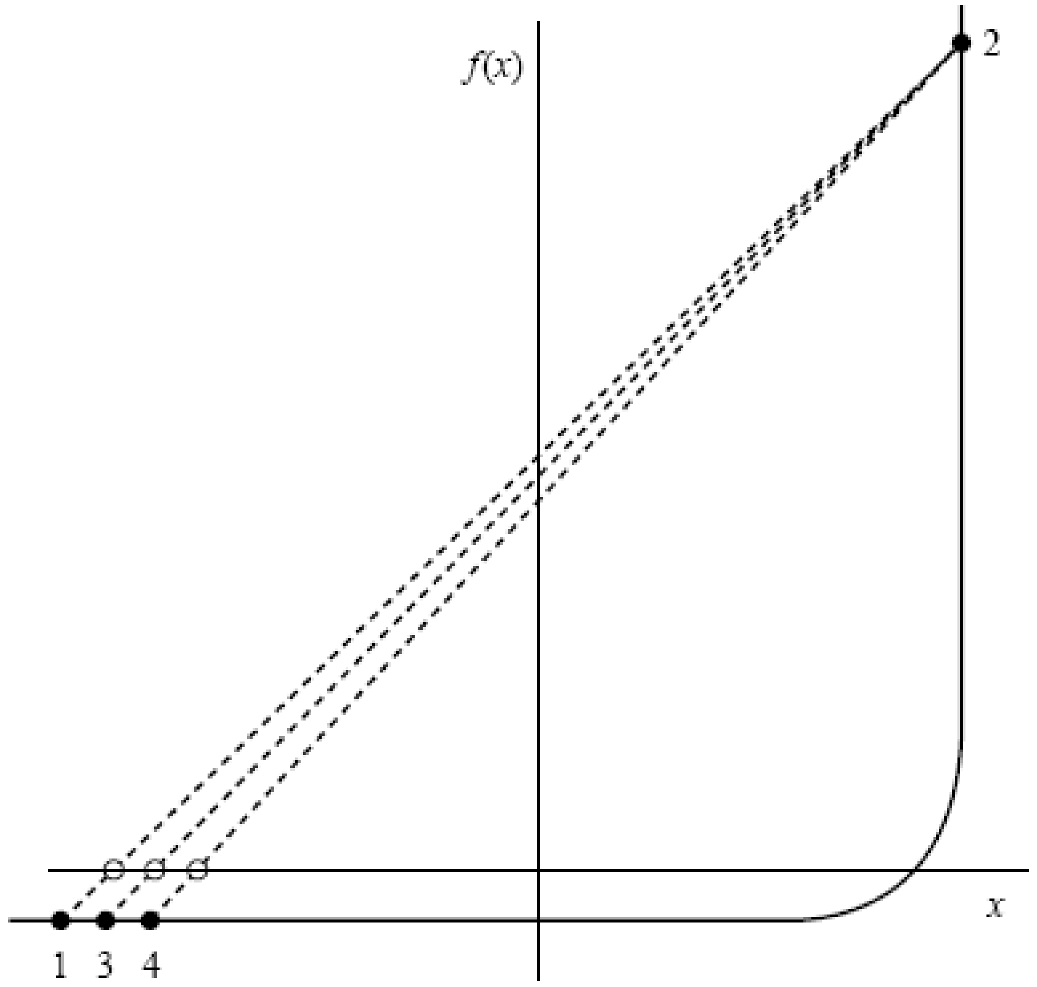


Figure 9.2.3. Example where both the secant and false position methods will take many iterations to arrive at the true root. This function would be difficult for many other root-finding methods.

van Wijngaarden, Dekker, Brent method

- Inverse quadratic interpolation used:
  - take 3 values x1,x2,x3
  - calculate y1,y2,y3.
  - make a quadratic polynomial of x as a function of y:

$$x = \frac{(y-y1)(y-y2)x3}{(y3-y1)(y3-y2)} + \frac{(y-y1)(y-y3)x2}{(y2-y1)(y2-y3)} + \frac{(y-y2)(y-y3)x1}{(y1-y2)(y1-y3)}$$

$$y = 0 \rightarrow x = x2 + P/Q$$

  - P and Q are given in ratio's of y1,y2,y3,x1,x2,x3.
  - correction should be small, x2 should be close to root.
  - if Q is small or the correction large, the routine takes a bi-section step instead!

Newton-Rhapson

- when derivative is known, usually Newton-Rhapson method is superior.
  - f(x+eps)~f(x)+eps f'(x) + O(eps^2)
  - f(x+delta)=0 -> step delta = - f(x)/f'(x).
- For well-behaved roots: convergence is quadratically. Near root, the number of significant digits DOUBLES per step

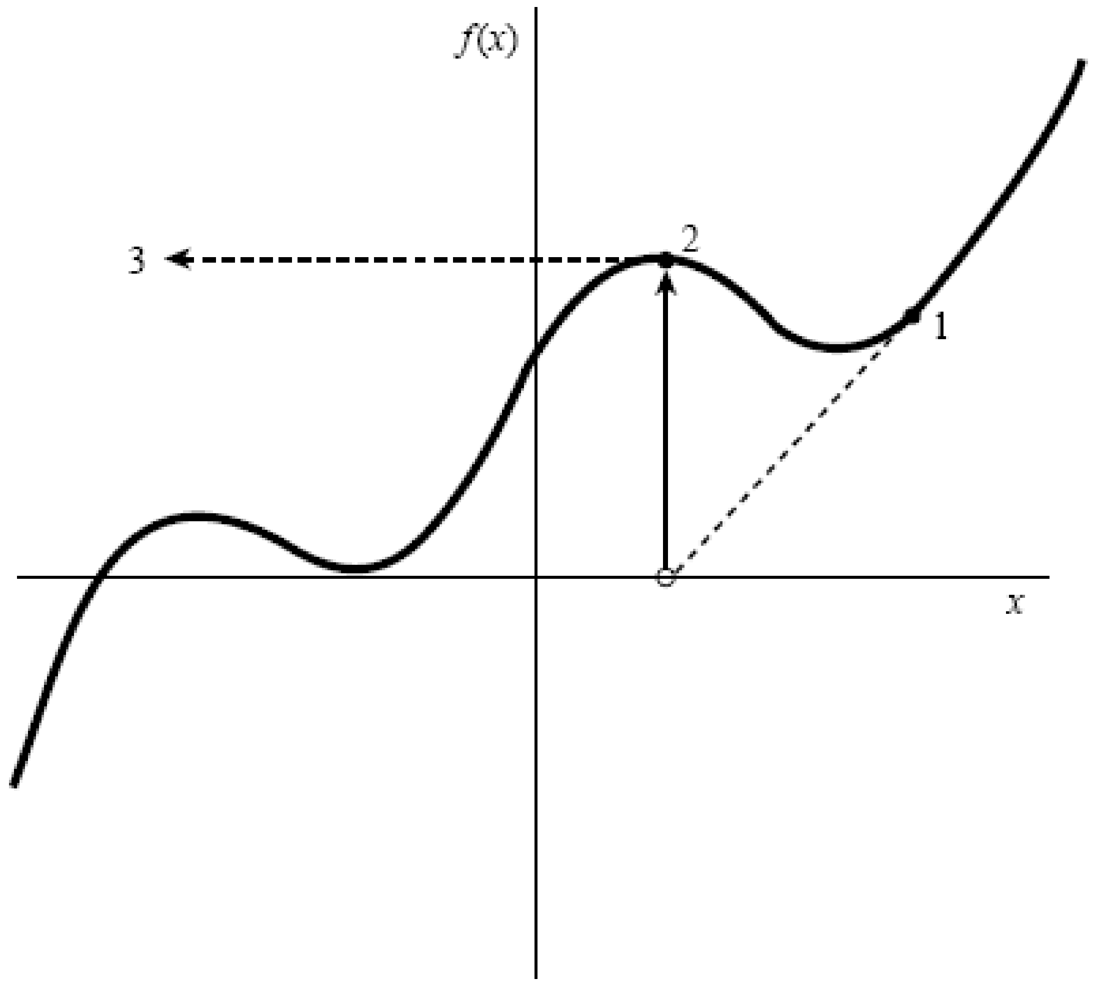
$$f(x+\varepsilon)=f(x)+\varepsilon f'(x)+\frac{1}{2}\varepsilon f''(x)+\dots$$

$$f'(x+\varepsilon)=f'(x)+\varepsilon f''(x)+\dots$$

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

$$\varepsilon_{i+1}=\varepsilon_i-f/f'=-\varepsilon_i^2\frac{f''}{2f'}$$

Newton-Rhapson



Newton Rhapson

- If derivative is not known, calculate it numerically ?
  - 2 function calculations, so convergence is superlinear by square root of 2 at most.
  - small steps: roundoff error dominates
  - Secant method will be better in cases where Newton-Rhapson would behave well, Brents method is superior also in cases where f'(x) close to zero
- Typically, Newton-Rhapson is used whenever possible, at least to polish up roots (as a last step).
  - **Provided that you can calculate the derivative as accurately as the function itself!**

Roots, polynomial

- Polynomial roots can be quite hard to find.
- Roots may be real or complex.
- Polynomial may be deflated:
  - P(x)=(x-r)Q(x)
  - Q(x) lower order, easier to find next root
  - algorithm will not zoom in on same root twice
  - complex roots: either go to complex numbers or deflate quadratically:
  - (x-(a+ib))(x-(a-ib)) = x*x-2ax+(a*a+b*b), deflate with x*x+cx+d
  - routine poldiv, chapter 5.3 recalculates polynomial coefficients for quadratic deflation

Forward and backward deflation

- Forward deflation : start with highest coefficient (Chapter 5.3)

```
P(x)/(x-root)=Q(x) {+ remainder/(x-root)} :
{
  rem=c[n];
  c[n]=0;
  for (i=n-1;i>0; i--) {
     temp=c[i];
     c[i]=rem;
     rem=temp+rem*root;
  }
}
```

- Forward deflation is stable when starting with smallest root (absolute value).

backward deflation

- Alternatively, you can start from the lowest coefficients and work back towards the highest coefficients. This (backward deflation) is stable when you start with the roots with the highest absolute value.
- Deflation will generally cost precision, because the position of the roots is not absolutely known. Best approach is to find the roots with the deflated polynomial, and polish them by relocating them in the original, non-deflated, polynomial
- double root: take 1 root out and find next root on the once-deflated polynomial. Double roots are hard to find, the function doesn’t change sign

Laguerre's method

- general method for finding real and complex roots
- extremely stable.
- Motivation:

$$P_n(x) = (x - x_0)\ldots(x - x_{n-1})$$

$$\ln|P_n(x)| = \ln|x - x_0| + \ldots. + \ln|x - x_{n-1}|$$

$$\frac{d \ln|P_n(x)|}{dx} = \frac{1}{|x - x_0|} + \ldots + \frac{1}{|x - x_{n-1}|} = \frac{P_n'}{P_n} \equiv G$$

$$-\frac{d^2 \ln|P_n(x)|}{dx^2} = \frac{1}{(x - x_0)^2} + \ldots + \frac{1}{(x - x_{n-1})^2} = \left[\frac{P_n'}{P_n}\right]^2 - \frac{P_n''}{P_n} \equiv H$$

Laguerre's method

- drastic assumption: trial root is located at distance a = x-x_0, all other roots are located at an equal larger distance b= x-x_i

$$\frac{1}{|x-x_0|}+\ldots+\frac{1}{|x-x_{n-1}|}=\frac{P_n'}{P_n}\equiv\frac{1}{a}+\frac{n-1}{b}$$

$$\frac{1}{(x-x_0)^2}+\ldots+\frac{1}{(x-x_{n-1})^2}=\left[\frac{P_n'}{P_n}\right]^2-\frac{P_n''}{P_n}\equiv\frac{1}{a^2}+\frac{n-1}{b^2}$$

$$a=\frac{n}{G\pm\sqrt{(n-1)(nH-G^2)}}$$

- a may be complex
- next trial value: x-a.
- routine laguer in roots_poly.h

Example root finding

The polynomial $\sum_{i=0}^{12} c_i x^i$ with coefficients

c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12
-360360000	-29664000	127595600	-6162360	-11818240	892882	389596	-31705	-2208	401	-93	-2	1

has roots in the interval [-10,10]. Find all roots with 7 digits accuracy (error < 10-6).

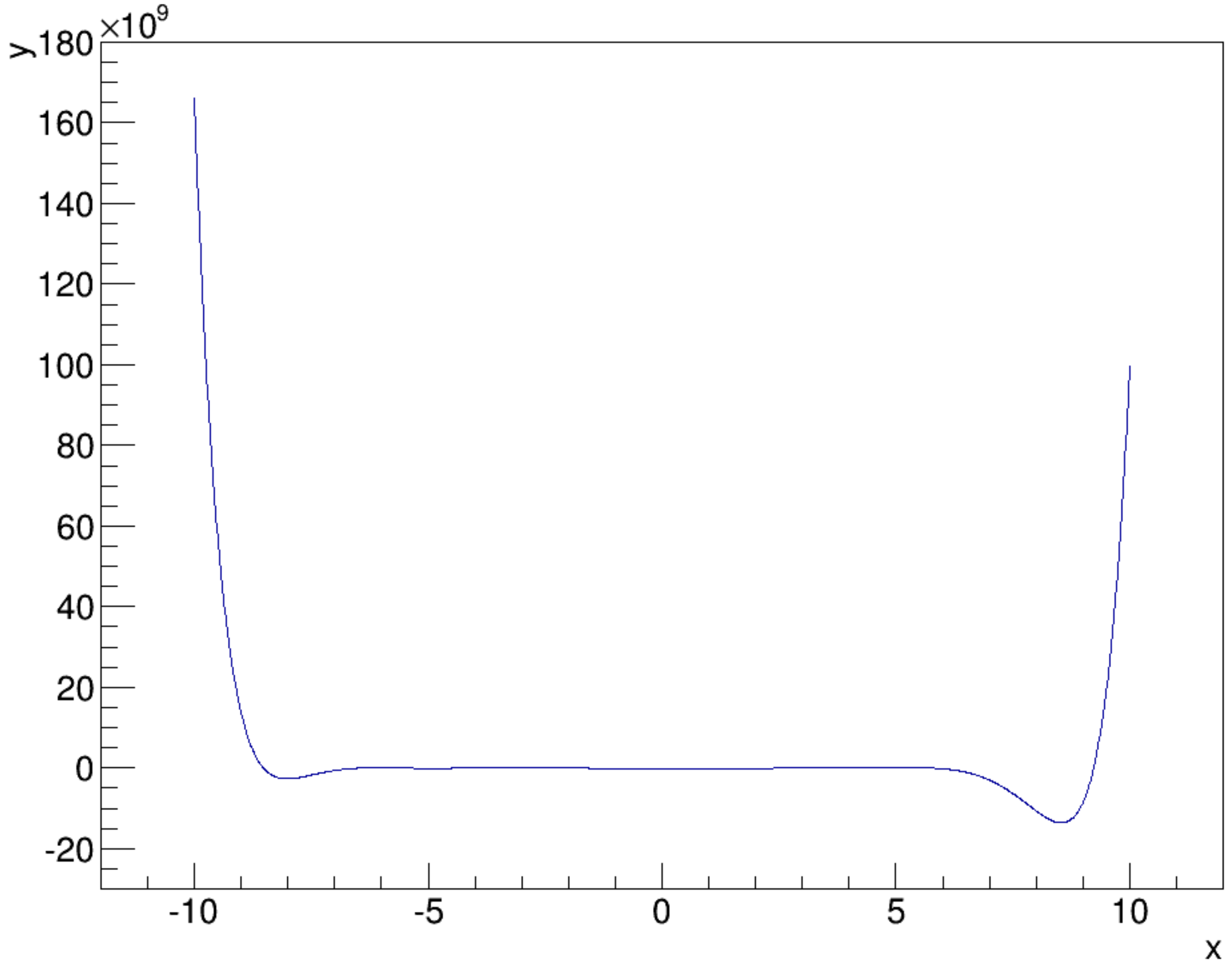
Note: at x=10, the terms in the series reach 10^{12}. A delicate cancellation occurs.
Note: it is not obvious from the coefficients how many roots exist.
Note: roots may be close (double roots). Numerically tough.

One could try such a challenge with e.g. the Laguerre algorithm, with bisection, or with Newton-Rhapson.

In case of Newton-Rhapson one needs a procedure to find the next root: with too large steps one skips roots and one can shoot off to infinity.

Example, root finding

- The polynomial under consideration.

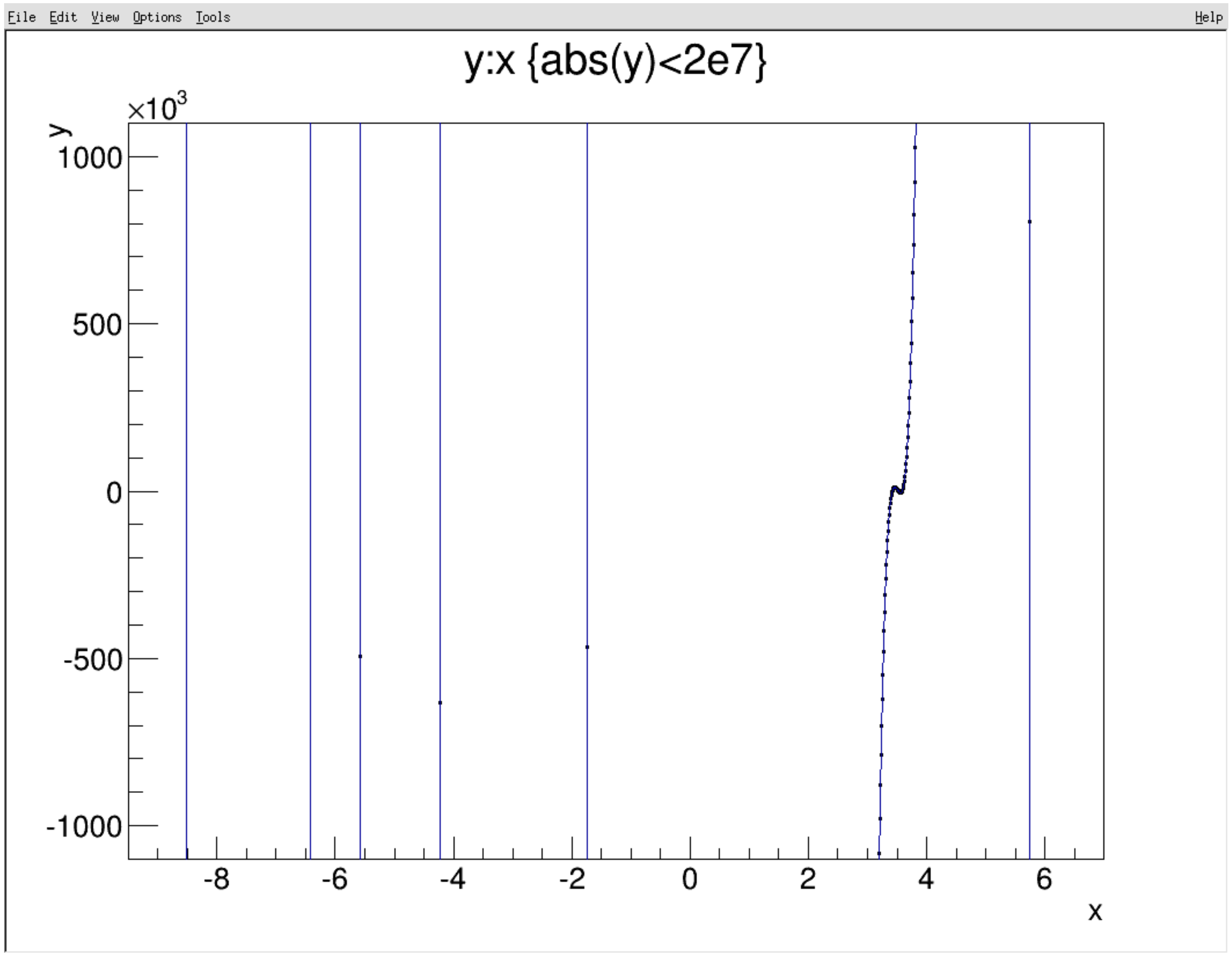


Example, root finding

The polynomial under consideration.

Calculated every x step 0.01; typically the derivatives around the poles exceed 10^8, but for the roots around 3.5 the derivatives are much smaller.

Hunting routines need to take care of that.



Example : Laguer routine

- Gives all roots correct to <1e-12 in this example

```
#include "nr3.h"

#include "roots_poly.h"

int main(){

    VecComplex c(13),x(12);

    c[0]=-360360000; c[1]=-29664000; c[2]=127595600; c[3]=-6162360;

    c[4]=-11818240; c[5]=892882; c[6]=389596; c[7]=-31705; c[8]=-2208;

    c[9]=401; c[10]=-93; c[11]=-2; c[12]=1;

    zroots(c,x,true);

    for(int i=0;i<12;i++){

        if (fabs (x[i].imag()) < 1e-14) cout<<setprecision(12) << x[i].real()<<endl;

        else cout << " complex root found: " << setprecision(12) << x[i] << endl;

    }

    return 0;

}
```

Example: with Newton-Rhapson

- When close to the root, Newton-Rhapson zooms in fast.
- To find the next root: hunting needed
  - Make small steps in x.
  - Check whether the function is bracketed (changes sign between steps)
  - Start with a new Newton-Rhapson seed x around the next root.
  - Code : newtonrhapson.cpp

Example root finding: accurate to 1e-13

- Output: laguer

```
-8.5207972894
-6.4244289009
-5.58257569496
-4.22681202354
-1.74165738677
 complex root found: (1.5,8.23103881658)
 complex root found: (1.5,-8.23103881658)
3.4244289009
3.5207972894
3.58257569495
5.74165738677
9.22681202354
```

Newton:

```
-8.5207972894
-6.4244289009
-5.58257569496
-4.22681202354
-1.74165738677
3.4244289009
3.5207972894
3.58257569496
5.74165738677
9.22681202354
```