

4

NON-LINEAR EQUATIONS

4.1 INTRODUCTION

Polynomials are, in a sense, the simplest mathematical functions. They can be evaluated directly using only the + and \times operators. Indeed, polynomials are at the heart of all numerical methods because, ultimately, all calculations must be reduced to polynomials.

The IRR Problem

Consider an investment of f_0 in some project which gives returns f_1, f_2, \dots, f_n at the end of years $1, 2, \dots, n$. The *present value* of this investment is

$$P(f, r, n) = \sum_{k=0}^n \frac{f_k}{(1+r)^k},$$

where $f = \{f_0, f_1, f_2, \dots, f_n\}$ is a stream of *cash flows*, r is the interest rate, and n is the life of the project in time units. The initial investment f_0 is a negative cash flow while the remainder may be individually positive or negative.

The *internal rate of return* (IRR) of the project is defined as the interest rate r that makes $P(f, r, n) = 0$. That is, r is the solution of the equation

$$P(f, r, n) = 0 = \sum_{k=0}^n \frac{f_k}{(1+r)^k}.$$

Letting $d = 1/(1+r)$ we get an n -degree polynomial equation which we must solve for d and hence r :

$$f_0 + f_1d + f_2d^2 + \dots + f_nd^n = 0.$$

The internal rate of return is used to rank alternative investments : an investment with a higher r is preferred to one with a lower r .

If $n = 2$ the problem is easy — use the quadratic formula. There are formulas for $n = 3, 4$ but these are quite messy. When we get to $n = 5$ we hit a mathematical stone wall : Ruffini (1813), Abel (1827), and Galois (c. 1832) proved that there can be no formula using the elementary operations $\{+, -, \times, /, \sqrt{\cdot}\}$, for polynomial equations of degree $n = 5$. Instead of formulas we must rely on iterative methods that approximate the solution. (We will return to the IRR problem when we study methods for finding the zeros of polynomials.)

In this chapter we study methods for solving the general equation $f(x) = 0$, where x and $f(\cdot)$ may be scalar or vector-valued. The value of x , for which $f(x) = 0$, is called a *root of the equation* or a *zero* of $f(\cdot)$. In general $f(x)$ is a non-linear function of x and we must use iterative algorithms to generate a sequence of vectors that converges (we hope) to a solution of a problem.

4.2 ORDER OF CONVERGENCE

One of the most important features of an iterative algorithm is the speed with which it converges. Roughly speaking, this is the number of iterations necessary to obtain a fixed point with a specified accuracy. We will see in later sections that speed of convergence is not the only criterion by which an algorithm is judged : the amount of computation per iteration is also important.

We assume that any iterative algorithm for zero-finding generates a sequence $\{x_0, x_1, \dots, x_k, \dots\} \rightarrow x$, the solution to $f(x) = 0$, and that the error at stage k is defined as

$$e_k = |x_k - x|, \text{ error at stage } k.$$

Therefore, we may view an iterative algorithm as generating a sequence of errors

$$\{e_0, e_1, \dots, e_k, \dots\} \rightarrow 0.$$

In theory this sequence is infinite but in practice finite precision forces us to stop at some stage k when $e_k \leq \epsilon$, for some chosen ϵ .

Definition 4.1 (Order of Convergence). Let $\{x_0, x_1, \dots, x_k, \dots\} \rightarrow x$, and $e_k = |x_k - x|$. If there exists a number p and a constant $C \neq 0$ such that

$$\lim_{k \rightarrow \infty} \frac{e_k}{e_{k-1}^p} = C, \quad (4.1)$$

then p is called the *Order of Convergence* of $\{x_k\}$.

We assume that $e_k < 1$, for all k . A more usable definition is

$$e_k = C e_{k-1}^p,$$

where C possibly depends on k , but can be bounded above by a constant.

Sequences with $p = 1$ are said to have *Linear Convergence*, while sequences with $p > 1$ have *Super-Linear Convergence*.

4.2.1 Convergence of Successive Approximation

Consider the sequence $\{x_k = T(x_{k-1})\}$ and assume it converges to a fixed point $x = T(x)$. Assume further that at the fixed point the derivatives $T'(x), T''(x), \dots, T^{(n)}$ exist. Expanding $T(x_k)$ in a Taylor series about the fixed point x , we get

$$T(x_k) = T(x) + \frac{1}{1!}T'(x)(x_k - x) + \frac{1}{2!}T''(x)(x_k - x)^2 + \dots + \frac{1}{n!}T^{(n)}(x)(x_k - x)^n + R_{n+1},$$

or

$$T(x_k) - T(x) = \frac{1}{1!}T'(x)(x_k - x) + \frac{1}{2!}T''(x)(x_k - x)^2 + \dots + \frac{1}{n!}T^{(n)}(x)(x_k - x)^n + R_{n+1}.$$

Now $|T(x_k) - T(x)| = |x_{k+1} - x| = e_{k+1}$. Hence we get, using the triangle inequality,

$$e_{k+1} \leq |T'(x)|e_k + \frac{1}{2}|T''(x)|e_k^2 + \dots + \frac{1}{n!}|T^{(n)}(x)|e_k^n + |R_{n+1}|. \quad (4.2)$$

This expression allows us to determine the order of convergence of the successive approximation sequence $\{x_k = T(x_{k-1})\}$ if we know the derivatives of $T(\cdot)$ at the fixed point x . Thus we can deduce the following :

1. If $T'(x) \neq 0$ then $e_{k+1} \approx T'(x)e_k$, 1st order convergence.
2. If $T'(x) = 0, T''(x) \neq 0$ then $e_{k+1} \approx T''(x)e_k^2$, 2nd order convergence.
3. If $T'(x) = T''(x) = \dots = T^{(n-1)}(x) = 0, T^{(n)}(x) \neq 0$ then $e_{k+1} \approx T^{(n)}(x)e_k^n$, n th order convergence.

Example 4.1 (Square Root). We saw in Chapter 2 an algorithm for calculating \sqrt{a} :

$$x_{k+1} = \frac{1}{2} \left(x_k + \frac{a}{x_k} \right) \equiv T(x_k).$$

The mapping $T(x) = (x + a/x)/2$ has a fixed point $x = \sqrt{a}$. The first derivative is $T'(x) = (1 - a/x^2)/2$. This gives $T'(\sqrt{a}) = (1 - a/a)/2 = 0$. Hence this algorithm has 2nd order convergence, at least. This is confirmed by showing that $T''(\sqrt{a}) \neq 0$, in general. The effect of 2nd order convergence is to double the number of correct digits at each iteration or $e_k \approx e_{k-1}^2 \approx e_0^{2^k}$. If $e_0 = 2^{-1}$ then $e_k \approx e_0^{2^k} = 2^{-2^k}$. For $k = 6$ we get $e_k = 2^{-64} \approx 10^{-20}$. This means that $k = 6$ iterations are sufficient to give full IEEE double precision.

4.3 ALGORITHMS FOR NON-LINEAR EQUATIONS

Iterative algorithms for non-linear equations fall into two broad categories :

1. Globally convergent and slow,
2. Locally convergent and fast.

The simplest algorithms impose very few conditions on the function $f(\cdot)$ and work in most circumstances. These algorithms usually have linear convergence, i.e., $e_k = c e_{k-1}$ and hence are slow.

Algorithms with super-linear convergence, i.e., $e_k = c e_{k-1}^p$, $p > 1$, require a carefully-chosen starting value x_0 to ensure convergence. Even though these algorithms converge quickly, the amount of work required per iteration may offset this advantage.

4.3.1 Converting Equations to Fixed Point Form

We wish to solve $f(x) = 0$ iteratively. Hence we need to transform the equation as follows :

$$f(x) = 0 \mapsto x = T(x),$$

so that we can apply successive approximations which converge to a fixed point of T . This fixed point x must solve the original problem, i.e. x must be a zero of $f(x)$.

There are many transformations to do this conversion. Here are some of them :

1. Let $T(x) = x - f(x)$. Then $x = T(x) = x - f(x)$ is a fixed point of T and a zero of f . This simple transformation rarely works because T is rarely a contraction mapping.
2. Let $G(x)$ be any real-valued continuous function such that $0 < |G(x)| < \infty$. Then the transformation $T(x) = x - G(x)f(x)$ has fixed points that are zeros of $f(x)$. This is so because

$$T(x) = x = x - G(x)f(x) \Rightarrow G(x)f(x) = 0 \Rightarrow f(x) = 0,$$

because $0 < |G(x)| < \infty$.

3. Let $G(x)$ be a continuous function such that $G(0) = 0$ and $G(x) \neq 0$, $x \neq 0$. Then the transformation $T(x) = x - G[f(x)]$ has fixed points that are zeros of $f(x)$. This is so because

$$T(x) = x = x - G[f(x)] \Rightarrow G[f(x)] = 0 \Rightarrow f(x) = 0.$$

It would be interesting but probably a waste of time to devise our own algorithms using these or other transformations. Instead we will look at a set of algorithms that have been well tested and analysed and work on a wide range of functions. Nonetheless we must not forget that there may be problems where a purpose-built algorithm works better than the 'old reliables'.

4.3.2 The Bisection Algorithm

The simplest and most reliable algorithm for finding a zero of $f(x)$ is the Bisection or interval-halving method. It starts with two real numbers a and b such that $f(x)$ changes sign in the interval $[a, b]$, i.e., $\text{sign}[f(a)] \neq \text{sign}[f(b)]$. This implies that $f(x) = 0$ somewhere in $[a, b]$. If we evaluate $f(m)$ at the mid-point of the interval then we get three possibilities :

1. $f(m) = 0$, zero found. Stop.
2. $\text{sign}[f(m)] = \text{sign}[f(b)]$, zero in $[m, b]$.
3. $\text{sign}[f(m)] \neq \text{sign}[f(a)]$, zero in $[a, m]$.

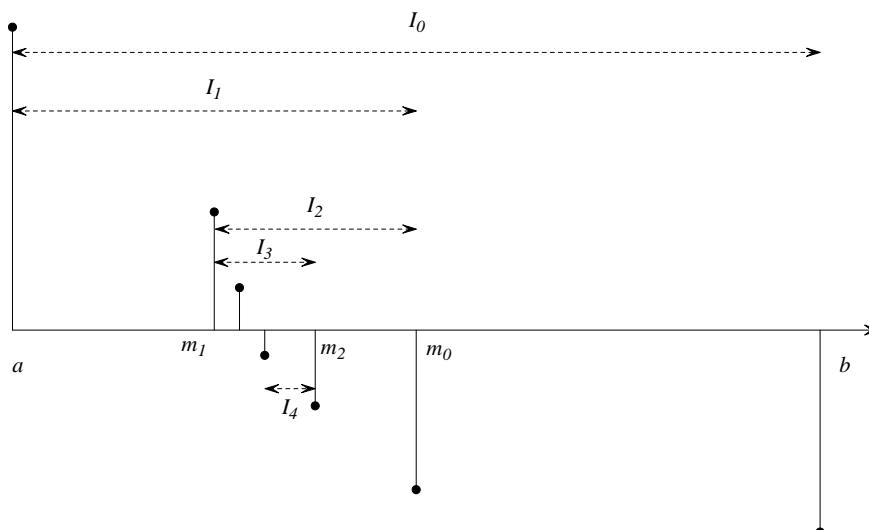


Figure 4.1 : The Bisection Method

Notice that the *value* or *shape* of the function play no part in the algorithm. Only the sign is of interest. We now formally state the algorithm.

```

algorithm Bisect( $f, a, b, \epsilon, \text{maxits}$ )

 $f_a := f(a); f_b := f(b)$ 
for  $k := 1$  to  $\text{maxits}$  do
   $m := (a + b)/2$ 
   $f_m := f(m)$ 
  if  $f_m = 0$  then return ( $m$ )
  else if  $\text{sign}(f_m) = \text{sign}(f_a)$  then
     $a := m$ 
     $f_a := f_m$ 
  else
     $b := m$ 
     $f_b := f_m$ 
  endif
  if  $|a - b| \leq \epsilon$  then return  $(a + b)/2$ 
endfor
endalg Bisect

```

Notice that the algorithm is ultimately controlled by the **for** – loop. This ensures that the algorithm terminates. We will write all subsequent algorithms in this way.

Analysis of Bisect

The algorithm performs one function evaluation per iteration. It is obvious that the interval is halved (contracted) each iteration. Hence

$$e_k = \frac{1}{2} e_{k-1}.$$

Thus the Bisection Algorithm has order 1 or linear convergence and it gains 1 bit of precision for each iteration. The error after k iterations is $e_k = 2^{-k}|b - a|$. The algorithm stops after k iterations with

$$2^{-k}|b - a| = \epsilon, \text{ or } 2^k \epsilon = |b - a|, \text{ or } 2^k = \frac{e_0}{\epsilon}.$$

Taking logs of both sides we get

$$k = \left\lceil \log_2 \frac{e_0}{\epsilon} \right\rceil.$$

If the algorithm starts with $e_0 = 2^{-1}$ then after 52 iterations the error is $e_{52} = 2^{-53} \approx 10^{-16}$, which is full IEEE double precision. Thus if the algorithm starts with an initial (relative) error of $e_0 = 2^{-1}$ then no more than 52 iterations are needed to obtain full single precision. If it starts with $e_0 > 1$ the *after it has reduced the initial interval to a relative length of 1*.

The main work of the algorithm is the evaluation of $f(\cdot)$. Although it may appear that 2 function evaluations are needed per iteration, only one, $f(m)$, is needed if $f(a)$ or $f(b)$ is saved between iterations. Thus Bisection performs about 53 function evaluations to obtain full precision.

Divide & Conquer

The Bisection algorithm is an example of a *Divide & Conquer Algorithm*. It may be used to search a sorted table. In computer science circles this method is called *Binary Search*. It can search a table of size n in $\log_2 n$ probes. Thus a telephone directory (sorry! Eircom now requires us to call it *PhoneBook*) with 10^6 entries can be searched in $\log_2 10^6 = 6 \log_2 10 \approx 20$ steps.

The divide and conquer principle can be stated algorithmically as follows:

<p>algorithm DaCSolve (Problem P of size n)</p> <hr/> <p>1. Divide P into two parts : P_1 of size n_1 and P_2 of size n_2.</p> <p>2(a). DaCSolve (Problem P_1 of size n_1) $\rightarrow S_1$</p> <p>2(b). DaCSolve (Problem P_2 of size n_2) $\rightarrow S_2$</p> <p>3. Combine(S_1, S_2) $\rightarrow S$, the solution of P</p> <p>endalgDaCSolve</p>
--

Notice that this is a *recursive* algorithm : it calls or invokes itself in steps 2(a) and 2(b). The divide and conquer principle is based on the fact that it is (usually) easier to solve two smaller problems than one larger problem.

Exercise 4.1. Prove or disprove : the Bisection algorithm works whether $a < b$ or $b < a$.

Exercise 4.2. How many function evaluations does Bisection require for full precision if the length of the initial interval is 1000? 10000? Demonstrate.

Exercise 4.3. What happens to Bisection if $f(x)$ has a jump discontinuity somewhere in $[a, b]$? Demonstrate.

Exercise 4.4. Demonstrate the action of Bisection on the function $f(x) = -1 + 1/x$ with $[a, b] = [0.5, 2.0]$

Exercise 4.5. Demonstrate the action of the Bisection Method on the function $f(x) = (x - 1.0)^2 - 3.0$ Start with $[a, b] = [0, 20]$ and continue until this interval has been reduced to 1/10th of its starting length. Use round-to-nearest, 4-digit decimal arithmetic.

Exercise 4.6. Regula Falsi. This is a refinement of the Bisection algorithm : it approximates the function f by passing a straight line through the points $(a, f(a))$ and $(b, f(b))$. The point c , at which this line intersects the x -axis, is the used instead of the mid-point m of the Bisection algorithm. This is the only difference.

Write an algorithm for this method, analyse it and compare it with the results of the Bisection analysis. Which do you recommend?

4.3.3 Newton's Algorithm

The Bisection algorithm uses the sign information of $f(x)$ only. This is both a strength and a weakness of the method. If the function is ill-behaved then Bisection can handle it because it needs to know very little about the function. On the other hand, if the function is smooth then it does not take advantage of the shape and smoothness of the function.

Newton's method takes the shape of the function into account by using its derivative to get a simple approximation to the function. Expanding $f(x)$ in a Taylor series about x_k we get

$$f(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2!}f''(x_k)(x - x_k)^2 + \dots + \frac{1}{(n-1)!}f^{(n-1)}(x_k)(x - x_k)^{n-1} + \dots$$

Dropping all but the first two terms we get the linear approximation

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k).$$

We use this to get an approximate solution to $f(x) = 0$. This gives

$$f(x_k) + f'(x_k)(x - x_k) = 0 \quad \text{or} \quad (x - x_k) = -f(x_k)/f'(x_k).$$

The solution of this equation is used as the next approximation to the zero of $f(\cdot)$, i.e.,

$$\boxed{x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}} \quad \text{Newton} \quad (4.3)$$

Thus the algorithm replaces $f(x)$ with the tangent at x_k and solves the linear equation to get the next approximation. The formal statement of Newton's method is as follows :

```

algorithm Newton ( $f, f', x_{old}, \epsilon, \text{maxits}$ )
 $f_{old} = f(x_{old}); f'_{old} := f'(x_{old})$ 
for  $k := 1$  to  $\text{maxits}$  do
   $x_{new} := x_{old} - f_{old}/f'_{old}$ 
   $f_{new} := f(x_{new}); f'_{new} := f'(x_{new})$ 
  if  $|x_{new} - x_{old}| \leq \epsilon$  OR  $f_{new} = 0$  then
    return ( $x_{new}$ )
  else
     $x_{old} := x_{new}$ 
     $f_{old} := f_{new}; f'_{old} := f'_{new}$ 
  endif
endfor
endalg Newton

```

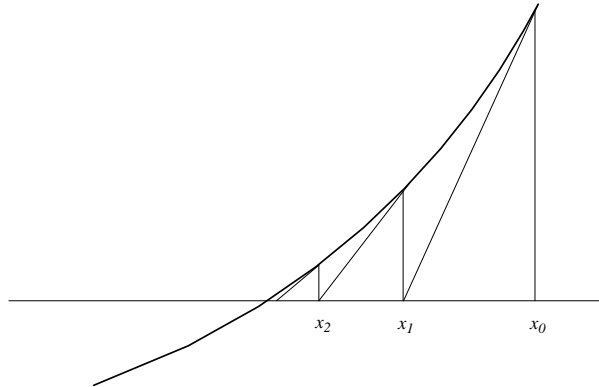


Figure 4.2 : Newton's Method

Analysis of Newton's Method

The main work is in the evaluation of f and f' , i.e., two function evaluations per iteration. Thus it requires twice as much work per iteration as Bisection. However, the order of convergence is 2, as we now show.

We get the order of convergence by examining the derivatives of $T(x)$ at the fixed point.

$$T(x) \equiv x - f(x)/f'(x).$$

At a fixed point,

$$\begin{aligned} x = T(x) &= x - f(x)/f'(x) \\ \Rightarrow f(x)/f'(x) &= 0 \\ \Rightarrow f(x) = 0 &\text{ if } f'(x) \neq 0. \end{aligned}$$

The first derivative is

$$T'(x) = 1 + \frac{f''(x)f(x)}{[f'(x)]^2} - \frac{f'(x)}{f'(x)} = f(x) \left(\frac{f''(x)}{[f'(x)]^2} \right).$$

At a fixed point $f(x) = 0$ and so $T'(x) = 0$. Hence Newton has at least 2nd order convergence. The second derivative of $T(x)$ is, using MAXIMA or MAPLE

$$T''(x) = f(x) \left(\frac{f'''(x)}{(f'(x))^2} - \frac{2f''(x)^2}{(f'(x))^3} \right) + \frac{f''(x)}{f'(x)} = \frac{f''(x)}{f'(x)} \neq 0, \quad \text{at } f(x) = 0.$$

Hence Newton's algorithm is 2nd order, i.e.,

$$e_k = ce_{k-1}^2 \quad \text{and} \quad e_k = e_0^{2^k},$$

assuming, for convenience, that $c = 1$.

Second order convergence is very rapid when we start close to the fixed point. For example, suppose we start with $e_0 = 2^{-1}$. This will give the 2nd order sequence

$$e_0 = 2^{-1}, \quad e_1 = 2^{-2}, \quad e_2 = 2^{-4}, \quad e_3 = 2^{-8}, \dots, e_k = 2^{-2^k}.$$

Thus, after 5 iterations we get $e_5 = 2^{-32} \approx 10^{-9}$.

The algorithm stops after k iterations with $e_k = e_0^{2^k} = \epsilon$. Taking logs of both sides we get

$$2^k \log_2 e_0 = \log_2 \epsilon \quad \text{or} \quad 2^k = \frac{\log_2 \epsilon}{\log_2 e_0}.$$

Again, taking logs of both sides we get

$$k = \left\lceil \log_2 \frac{\log_2 \epsilon}{\log_2 e_0} \right\rceil.$$

Hence, with $e_0 = 2^{-1}$ and $\epsilon = 2^{-53} \approx 10^{-16}$ we need

$$n = \left\lceil \log_2 \frac{\log_2 2^{-53}}{\log_2 2^{-1}} \right\rceil = \lceil \log_2 53 \rceil = 6 \quad \text{iterations.}$$

Thus we get full IEEE double precision after 6 iterations. This rapid convergence comes at a price.

Difficulties with Newton's Method

1. The function $f(x)$ and its derivative must be evaluated at each iteration. Also, we need to find and program the derivative. Finding the derivative may be difficult, if not impossible. For example, the value of $f(x_k)$ may be the result of a long numerical simulation.
2. The convergence of Newton's algorithm is local and so good starting values are needed if convergence is to occur at all.

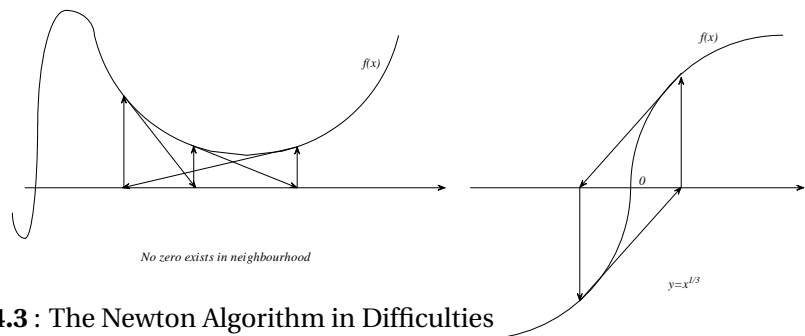


Figure 4.3 : The Newton Algorithm in Difficulties

Newton's method cycles if $x_{n+1} - a = -(x_n - a)$. This happens if $f(x)$ satisfies

$$x - a - f(x)/f'(x) = (x - a).$$

This is a separable ordinary differential equation

$$f'(x)/f(x) = 1/2(x - a).$$

A solution is $f(x) = \text{sign}(x - a)\sqrt{|x - a|}$. This has a zero at $x = a$ but its derivative $f'(x) = f(x)/2(x - a)$ is unbounded as $x \rightarrow a$.

Exercise 4.7. Newton's Algorithm for $1/a$ and \sqrt{a} . We saw in previous chapters the following two iteration formulas :

1. $x_{k+1} := x_k + x_k(1 - ax)$
2. $x_{k+1} := (x_k + a/x_k)/2$

The first converges to $1/a$ and the second to \sqrt{a} . Show how these are derived from Newton's algorithm. That is, what functions is Newton finding the zeros of?

Convergence to a Multiple Zero

If $f(x)$ has the form $f(x) = (x - a)^p g(x)$, then this function has a *zero of multiplicity p* at a . It can be seen in Figure 4.4 that these functions are very flat and close to zero about a multiple zero. This indicates that some difficulty can be expected when finding such zeros in finite precision arithmetic. Let us see how Newton's algorithm reacts to such functions. The function $f(x) = (x - a)^p g(x)$ behaves as $f(x) \approx (x - a)^p$, close to a , and so we will use this simpler function. The graph in Figure 4.4 clearly shows this behaviour.

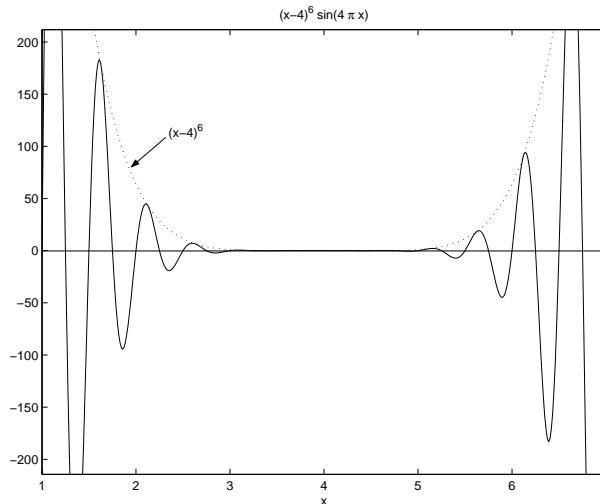


Figure 4.4 : $f(x) = (x - 4)^6 \sin 10x$

Using $f(x) \approx (x - a)^p$, we get the iteration mapping

$$\begin{aligned} T(x) &= x - \frac{f(x)}{f'(x)} \\ &= x - \frac{(x - a)^p}{p(x - a)^{p-1}} = x - \frac{(x - a)}{p}. \end{aligned}$$

Thus $T'(x) = 1 - \frac{1}{p}$.

This shows that $T'(x) \neq 0$ for $p \neq 1$ and so Newton's algorithm has 1st order convergence at a multiple zero. Using the Taylor series expansion for the error e_k (see Section 4.2) we get

$$e_{k+1} \approx T'(x_k)e_k = \left(1 - \frac{1}{p}\right)e_k.$$

Notice that the expression $(1 - 1/p) \rightarrow 1$ as $p \rightarrow \infty$. This means that the convergence becomes slower as p increases.

Multiple zeros are a cause of difficulty for all algorithms, including the Bisection method. Although the order of convergence of Bisection is not affected, finite precision means that the function $f(x)$ will be indistinguishable from zero over a wide range of x . This is clearly shown in Figure 4.5, which shows the plots for $y = (x - 1)^5$ and its expanded form $y = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$. We will discuss practical convergence tests in Section 4.4.

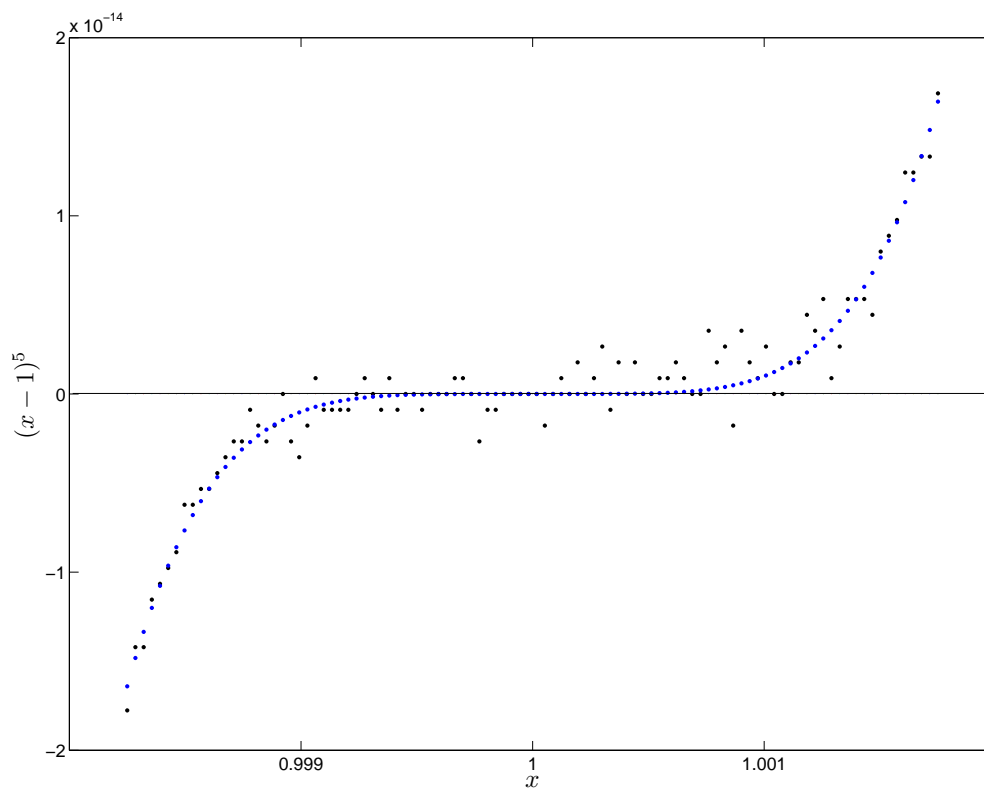


Figure 4.5 : $f(x) = (x - 1)^5 = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$

Mathematically, $f(x) = (x - 1)^5$ has 1 multiple zero at $x = 1$. Evaluating this function at 2000 evenly-spaced points $x \in [1 \pm 10^{-3}]$, MATLAB found no zeros. Using the expanded form $f(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$, it found 620 zeros.

4.3.4 The Secant Algorithm

The main drawback with the Newton algorithm is the need to have the derivative. The Secant algorithm gets around this problem by using an approximation to the derivative :

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}.$$

Using this in Newton's algorithm we get the iteration formula :

$$x_{k+1} = x_k - f(x_k) \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \quad \text{Secant} \quad (4.4)$$

This is a *two-point* method which approximates $f(x)$ by passing a straight line through the two points $(x_k, f(x_k))$ and $(x_{k-1}, f(x_{k-1}))$. The next approximation is where this line passes through the x -axis.

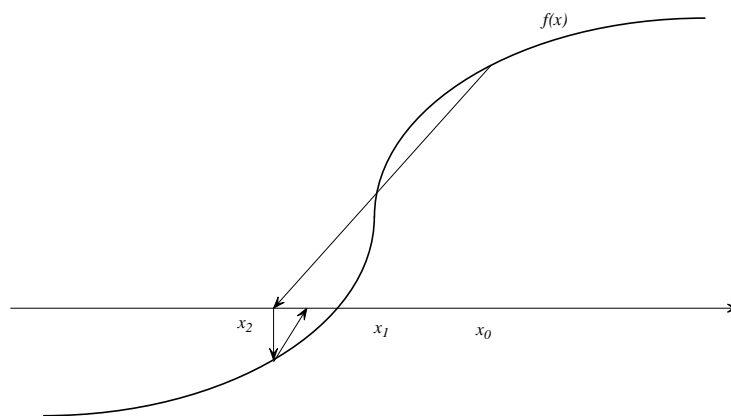


Figure 4.6 : The Secant Algorithm

The formal statement of the Secant algorithm is as follows :

```

algorithm Secant ( $f, x_0, x_1, \epsilon, \text{maxits}$ )
 $f_0 := f(x_0); f_1 := f(x_1)$ 
for  $k := 1$  to maxits do
   $x_2 := x_1 - f_1 * (x_1 - x_0) / (f_1 - f_0)$ 
   $f_2 := f(x_2)$ 
  if  $|x_2 - x_1| \leq \epsilon$  OR  $f_2 = 0$  then return ( $x_2$ )
  else
     $x_0 := x_1; f_0 := f_1$ 
     $x_1 := x_2; f_1 := f_2$ 
  endif
endfor
endalg Secant

```

Although it appears that the Secant algorithm requires 2 function evaluations, it needs only one. If we store previous f -values between iterations then we need just one f -evaluation at the new point x_2 . We will discuss this point in *Section 4.6 – Implementation Notes*.

Analysis of the Secant Algorithm

Again, the main work is in the evaluation of $f(x_2)$, i.e., one function evaluation per iteration. This is half the work per iteration of Newton. However, the order of convergence is no longer 2, as we now show. Deriving the order of convergence of the Secant algorithm is fairly tedious, so we will state the following result : (see Conte & De Boor, pg. 103)

$$e_{k+1} \approx c e_k e_{k-1} = c e_k^p,$$

where p the positive zero of $p^2 - p - 1$. Using the quadratic formula we get

$$p_1, p_2 = \frac{1 \pm \sqrt{1+4}}{2} \quad \text{or} \quad p_1 = \frac{1 + \sqrt{5}}{2} \quad \text{and} \quad p_2 = \frac{1 - \sqrt{5}}{2}.$$

The positive zero is $p_1 = 1.618\dots$ and so we have

$$e_{k+1} \approx c e_k^{1.618\dots},$$

Assuming $c = 1$, the error after k iterations is $e_k = e_0^{p^k}$. The algorithm stops after k iterations with $e_0^{p^k} = \epsilon$. Taking logs of both sides we get

$$p^k \log_2 e_0 = \log_2 \epsilon \quad \text{or} \quad p^k = \frac{\log_2 \epsilon}{\log_2 e_0}.$$

Again, taking logs of both sides we get

$$k \log_2 p = \left\lceil \log_2 \frac{\log_2 \epsilon}{\log_2 e_0} \right\rceil \quad \text{or} \quad k = \left\lceil \frac{\log_2(\log_2 \epsilon / \log_2 e_0)}{\log_2 p} \right\rceil$$

Hence, with $e_0 = 2^{-1}$ and $\epsilon = 2^{-53} \approx 10^{-16}$ we have $\log_2(\log_2 2^{-53} / \log_2 2^{-1}) = \log_2 53$ and so

$$k = \left\lceil \frac{\log_2 53}{\log_2 1.618} \right\rceil = \left\lceil \frac{5.72792045}{0.694} \right\rceil = 9 \quad \text{iterations.}$$

This is slower convergence than Newton but it is still fast, given that it requires just one function evaluation per iteration. Indeed, to get full IEEE double precision accuracy of 2^{-53} , Newton performs $6 \times 2 = 12$ function evaluations while Secant performs 9 evaluations. Clearly, under these conditions, Secant is more efficient than Newton.

4.3.5 Higher Order Algorithms

The Newton and Secant algorithms used straight lines to approximate the function at each iteration. We can get algorithms with higher orders of convergence if we use higher-order approximations to f .

Third Order Newton

In the standard Newton algorithm we used the first two terms of the Taylor series expansion of $f(x)$ about the point x_k (See Section 4.3.3). We now use the first three terms to approximate $f(x)$. That is,

$$\begin{aligned} f(x) &\approx f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2!}f''(x_k)(x - x_k)^2 \\ &= f(x_k) + f'(x_k)\delta x_k + \frac{1}{2}f''(x_k)\delta x_k^2, \quad \text{where } \delta x_k = x - x_k. \end{aligned}$$

Solving

$$f(x_k) + f'(x_k)\delta x_k + \frac{1}{2}f''(x_k)\delta x_k^2 = 0 \quad \text{for } \delta x_k, \text{ we get}$$

$$\delta x_k = \frac{-f'(x_k) \pm \sqrt{[f'(x_k)]^2 - 2f(x_k)f''(x_k)}}{f''(x_k)}.$$

Setting $x_{k+1} = x_k + \delta x_k$ gives the third-order Newton iteration formula

$$\boxed{x_{k+1} = x_k + \frac{-f'(x_k) \pm \sqrt{[f'(x_k)]^2 - 2f(x_k)f''(x_k)}}{f''(x_k)}}. \quad (4.5)$$

This is known as *Halley's Irrational Form*, after the astronomer Sir Edmond Halley (c. 1656–1743), who was a colleague of Newton.

We will not give a formal statement of this algorithm because it is very similar to the second-order algorithm, with the following differences :

1. Three function evaluations are needed per iteration : $f(x_k), f'(x_k), f''(x_k)$.
2. There are two possible δx_k 's because of the \pm and this gives rise to two possible x_{k+1} 's. The choice is a delicate one.
3. The $\sqrt{(\dots)}$ gives rise to the possibility of a complex x_{k+1} .

Generally speaking, the third-order Newton algorithm is not worth the effort. It requires too much work per iteration and is tricky to implement.

Exercise 4.8. Devise a third-order algorithm for calculating \sqrt{a} , using the third-order Newton iteration formula. Think about it.

Halley's Method

This is the rational form of third-order method just discussed. Again we start with the first three terms of the Taylor series about x_k :

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2!}f''(x_k)(x - x_k)^2.$$

If x_{k+1} is a zero of this approximate function then we have

$$0 = f(x_k) + (x_{k+1} - x_k)[f'(x_k) + \frac{1}{2}f''(x_k)(x_{k+1} - x_k)].$$

Rearranging this equation, we get

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k) + \frac{1}{2}f''(x_k)(x_{k+1} - x_k)}.$$

Substituting the Newton iteration formula $x_{k+1} - x_k = -f(x_k)/f'(x_k)$ above, we get

$$\begin{aligned} x_{k+1} &= x_k - \frac{f(x_k)}{f'(x_k) + \frac{1}{2}f''(x_k)(-f(x_k)/f'(x_k))} \\ &= x_k - \frac{f(x_k)}{f'(x_k) - f''(x_k)f(x_k)/2f'(x_k)} = T_H(x_k). \end{aligned}$$

It can be shown that $T_H'(\alpha) = T_H''(\alpha) = 0$, where α is a zero of $f(x)$, and so Halley's method has third order convergence.

4.3.6 Multi-Point Secant Algorithms

The Secant algorithm uses a straight line through two points to approximate $f(x)$. Multi-point Secant algorithms use polynomials through many points to approximate $f(x)$. This is called *interpolation*.

Müller's Quadratic Interpolation Algorithm

This algorithm uses the quadratic $a_0 + a_1x + a_2x^2$ to interpolate $f(x)$ at three points x_0, x_1, x_2 . The coefficient a_0, a_1, a_2 are found by solving

$$\begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix}. \quad (4.6)$$

The quadratic equation $a_0 + a_1x + a_2x^2 = 0$ is then solved to get a new point

$$x_{k+1} := \frac{-a_1 \pm \sqrt{a_1^2 - 4a_0a_2}}{2a_2} = T(x_k, x_{k-1}, x_{k-2}). \quad (4.7)$$

As in the third-order Newton algorithm, we have a choice of two points and the possibility that they are complex (See Figure 4.7). Notice that at each iteration of Müller's algorithm we must solve a set of 3 linear equations to get the polynomial coefficients, and then solve the quadratic equation.

Inverse Quadratic Interpolation

This is a far more elegant algorithm than the obvious quadratic interpolation. Instead of finding the coefficients of the quadratic function $p(x) = a_0 + a_1x + a_2x^2$ that passes through the points

$\{(x_0, f_0), (x_1, f_1), (x_2, f_2)\}$, we find the coefficients of the inverse quadratic function $p(f) = d_0 + d_1f + d_2f^2$ that passes through the points $\{(f_0, x_0), (f_1, x_1), (f_2, x_2)\}$. This means we must solve

$$\begin{bmatrix} 1 & f_0 & f_0^2 \\ 1 & f_1 & f_1^2 \\ 1 & f_2 & f_2^2 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \quad (4.8)$$

Now, before we rush ahead and solve for d_0, d_1, d_2 let us look at what the inverse quadratic is in graphical form (see Figure 4.7).

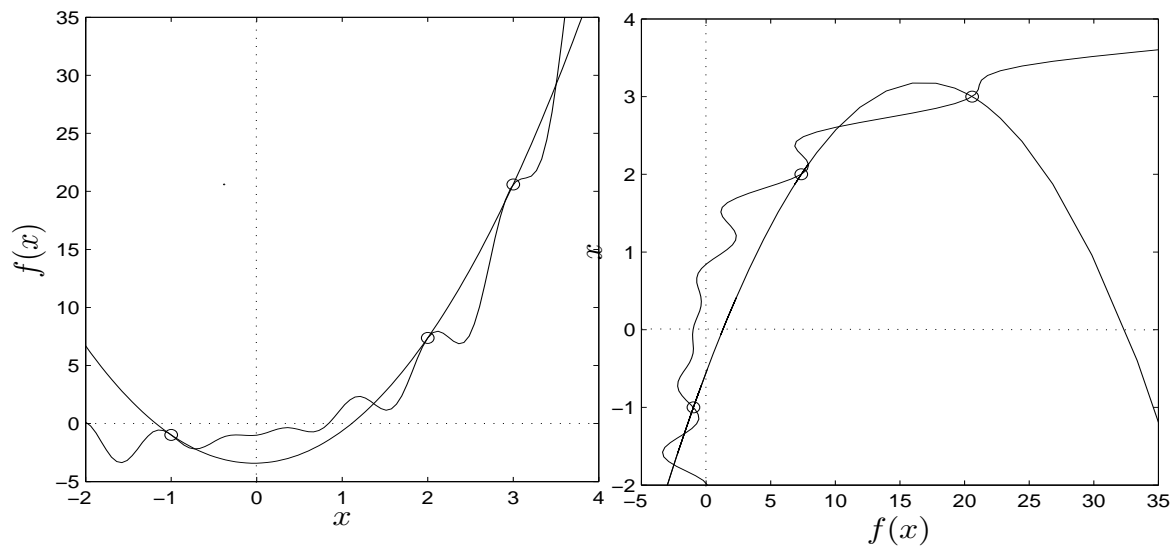


Figure 4.7 : Quadratic and Inverse Quadratic Interpolation

We want the x that makes $f(x) = 0$. In the inverse quadratic this means we want the value of $p(f)$ when $f = 0$. The value of $p(f) = d_0 + d_1f + d_2f^2$ at $f = 0$ is d_0 . Thus the solution of $a_0 + a_1x + a_2x^2 = 0$ is $x = d_0$.

We now see that using inverse quadratic interpolation, all we need to calculate is one coefficient, d_0 , and there is no need to solve the quadratic — its solution is $x = d_0$.

Exercise 4.9 (Inverse Quadratic Interpolation). Derive an expression for d_0 from (4.8) and use this to write a complete zero-finding algorithm. How many function evaluations does your algorithm use?

Analysis of Three-Point Secant Algorithms

Although these algorithms do more work than the Two-Point Secant algorithm, the main work is still in the evaluation of the function at each iteration. We have not given a formal statement of these algorithms but it should be obvious that they require one function evaluation per iteration if we save previous values between iterations.

A tedious analysis shows that the order of convergence of the Three-Point Secant algorithm is

$$e_{k+1} = ce_k e_{k-1} e_{k-2} \approx ce^{1.84k}.$$

This order of convergence is close to that of Newton's algorithm yet requires just one function evaluation per iteration. Assuming $c = 1$, the error after k iterations is $e_k = e_0^{1.84^k}$. The algorithm stops after k iterations with $e_0^{1.84^k} = \epsilon$.

Hence, with $e_0 = 2^{-1}$ and $\epsilon = 2^{-53} \approx 10^{-16}$ we need

$$k = \left\lceil \log_2 \frac{53}{\log_2 1.84} \right\rceil = \lceil \log_2 (53/0.880) \rceil = \lceil \log_2 60 \rceil = 6 \text{ iterations.}$$

This is the same number of iterations as required by Newton's algorithm for the same accuracy, but with only 1 function evaluation per iteration.

4.3.7 Linear vs Super-Linear Algorithms

We have seen that we can get higher orders of convergence by using higher order approximations to f . These higher order algorithms are more complicated and require more function evaluations per iteration. This suggests the question : "Is it worth the extra effort to attain a higher order of convergence?". Consider the following paragraph from Lloyd N. Trefethen's essay *Numerical Analysis*, Oxford University, March 2006 :¹

Students often think it might be a good idea to develop formulas to enhance the exponent [p , the order of convergence] in this estimate to 3 or 4. However, this is an illusion. Taking two steps at a time of a quadratically convergent algorithm yields a quartically convergent one – so the difference in efficiency between quadratic and quartic is at best a constant factor. The same goes if the exponent 2, 3, or 4 is replaced by any other number greater than 1. The true distinction is between all of these algorithms that converge super-linearly, of which Newton's method is the prototype, and those that converge linearly or geometrically, where the exponent is just 1.

Trefethen's idea, like most good ideas, is obvious once it has been pointed out. Let us derive a general expression for the error based on Trefethen's idea. Consider an algorithm with an order of convergence p , that is

$$|x_{k+1} - x_k| = e_{k+1} = ce_k^p, \quad (4.9)$$

for one iteration of the algorithm. The error for two successive iterations is

$$e_{k+2} = ce_{k+1}^p = c(ce_k^p)^p = c^{p+1} e_k^{p^2}.$$

If we construct a new algorithm that takes two steps of the old p -convergent algorithm at each iteration then we have a new algorithm whose order of convergence is defined by

$$\tilde{e}_{k+1} = \tilde{c} \tilde{e}_k^{p^2}. \quad (4.10)$$

We can see that the new order of convergence is p^2 and this is higher than the old algorithm if, and only if, $p > 1$. If the old algorithm has linear convergence then the new algorithm is linear. This shows that

¹Trefethen's essay is here <http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen/NAessay.pdf>

*The true distinction between algorithms is
whether they have linear or super-linear convergence*

Once we have a super-linear algorithm we may use it to construct a new algorithm with any order of convergence, by taking the appropriate number of steps of the old algorithm, at each iteration or the new algorithm.

4.3.8 Comparison of Algorithms

The discussion of linear and super-linear algorithms ignored the amount of work required at each iteration. We now take into account all the factors that affect the running time of an algorithm. This gives a complete answer to the question posed in Section 4.3.7.

The parameters that affect the running time of an algorithm, assuming they attain the same precision ϵ and start with the same initial error e_0 , are (i) p – order of convergence, and (ii) w – work per iteration. We could include the number of lines of code needed to implement each algorithm but each of the algorithms above is small—less than one page—and so we omit this parameter.

The efficiency of an algorithm is a function of p and w , $E(p, w)$. The form of the function is a matter of debate. We will use the function (see Ralston & Rabinowitz)

$$E(p, w) = p^{1/w} = \sqrt[w]{p}, \quad (4.11)$$

which seems reasonable.

The table below compares the five algorithms we have discussed. We assume that the work per iteration w is the number of function evaluations.

Table 4.1: Efficiency of Zero-Finding Algorithms

Algorithm	Order p	Work w	Effic. $p^{1/w}$
Bisect	1	1	1.00
Newton 2nd	2	2	1.41
2-Secant	1.62	1	1.62
Newton 3rd	3	3	1.44
3-Secant	1.84	1	1.84

We can see that the 3-point Secant is the most efficient although the 2-point Secant is not far behind. Neither Newton algorithm is efficient because each requires too much work. In fact an n -order Newton algorithm that requires n function evaluations per iteration has an efficiency $E(p, n) = n^{1/n}$. This has a maximum at $n = e = 2.7183$ and $E(n, n) = 1.44467$. If n is integer then $E(n, n)$ has a maximum at $n = 3$. That is, $E(3, 3) = \sqrt[3]{3} \approx 1.44$.

An n -point Secant algorithm has an error which is $e_{k+1} = C e_k e_{k-1} \cdots e_{k-n+1}$. This gives an order of convergence p which is the largest zero of the polynomial

$$p^n - p^{n-1} - \cdots - p - 1.$$

Here are some of the values of p as n increases

$$n = 2, 3, 4, 5, 6 \quad \text{and} \quad p = 1.61, 1.84, 1.93, 1.96, 1.98$$

This sequence converges to 2 and so there is no real benefit in going beyond $n = 3$.

q -step Efficiency

We have shown that any order of convergence is attainable by using successive repetitions of a super-linear algorithm with order of convergence $p > 1$. Assume this 1-step algorithm does w units of work per iteration. If we construct a new algorithm that makes q steps per iteration to we get p^q – order of convergence for qw units of work per iteration, then we have

$$E_1(p, w) = p^{\frac{1}{w}} \quad \text{and} \quad E_q(p, w) = (p^q)^{\frac{1}{qw}} = p^{\frac{1}{w}}$$

Thus there is no gain in efficiency despite having raised the order of convergence. If we can reduce the amount of work per q -step iteration to αqw then the efficiency is improved to $(p^q)^{\frac{1}{\alpha qw}} = p^{\frac{1}{\alpha w}}$. This leads us to the conclusion that there is no point in constructing higher-order algorithms unless we can reduce the amount of work per iteration.

The 2-Secant and 3-Secant algorithms discussed above are not pure q -step algorithms and do not attain 2nd or 3rd order convergence. But they gain in efficiency because they use just one unit of work (a function evaluation) per iteration

One final point about higher-order algorithms : they are much more sensitive to starting and intermediate values than lower order methods. Hence they go wrong more often than simpler methods. It is for this reason that algorithms which use a combination of simple algorithms were invented. The following section discusses these *poly-algorithms*.

Starting Accuracy

The number of iterations needed to attain full precision accuracy depends on, (i) the order of convergence of the algorithm, (ii) the number of digits of precision in the floating point system, and (iii) the accuracy of the starting point x_0 .

Newton's method has 2nd-order convergence. This means that the number of accurate digits doubles at each iteration, *if the initial relative error* $e_0 < 1$. If we are using IEEE double-precision floating point arithmetic then we have $p = 53$ bits precision. If x_0 has 1 bit accuracy, i.e., $e_0 = 2^{-1}$, then we have 2 bit accuracy after the first iteration, 4 bits, 8 bits, 16 bits, 32 bits, and 64 bits accuracy after iterations 2,3,4,5, and 6. Obviously the higher the starting accuracy, the lower the number of iterations needed to reach full-precision accuracy.

In general, with p -digit base b arithmetic and order of convergence m , full-precision accuracy is attained after n iterations when $e_0^{m^n} = b^{-p}$. Taking $e_0 = b^{-s}$ we get $m^n = p/s$ and so we need, for $m \geq 2$,

$$n = \frac{\log_b p - \log_b s}{\log_b m} \quad \text{iterations for full-precision accuracy.} \quad (4.12)$$

Although this formula should not be taken too literally, it does show how precision p , starting accuracy s , and order of convergence m , interact.

4.3.9 Poly-Algorithms

These are algorithms that, usually, combine Bisection, 2-Secant, and 3-Secant. The algorithm switches between the three, depending on how well-behaved the function is at the current point. Needless to

say, these algorithms require great care in their implementation. The original algorithm was due to Dekker, 1969. Brent's 1972 implementation of it seems to be the most famous and widely-used.

TO BE COMPLETED

4.4 STOPPING RULES

All algorithms, to be algorithms, must stop after a finite number of steps, i.e., in a finite time. Any algorithm must have, built into it, some rule(s) to bring it to a halt. For example an algorithm to calculate $S = \sum_{i=1}^n |x_i|$ could have a **for** - loop that would stop after the n values have been summed. In fact, built into the **for** - loop is the test : IF $i > n$ THEN STOP.

The iterative algorithms in this chapter generate sequences of the form $\{x_k, f(x_k)\}$. Mathematically, $\lim_{k \rightarrow \infty} \{x_k, f(x_k)\} \rightarrow \{x, f(x) = 0\}$. Obviously no finite algorithm can attain this mathematical ideal and so we devise rules that stop the algorithm when it gets within some tolerance ϵ of the limit.

In the case of zero-finding algorithms, we know that $\lim_{k \rightarrow \infty} \{f(x_k)\} = 0$, so we can test explicitly for the closeness of $f(x_k)$ to 0. However, we do not know $\lim_{k \rightarrow \infty} \{x_k\}$ (this is what the algorithm is trying to find) and so we must test this sequence in the *Cauchy sequence* sense (see Chap 3). That is, stop when $\|x_{k+1} - x_k\| \leq \epsilon_x$.

These mathematical stopping rules must be modified to take into account the effects of finite-precision floating point arithmetic.

4.4.1 General Rules

There are many possible stopping rules because there are many different algorithms working on many different functions. We examine the the major rules that apply to all zero-finding algorithms that generate the sequence $\{x_k, f(x_k)\}$. These are :

1. f -sequence convergence.
2. x -sequence convergence.
3. Cycling, no f or x sequence convergence.
4. f or x sequence divergence.
5. Cannot proceed, e.g., about to divide by zero or some other exception.

1. f - Convergence

is often tested with

$$|f(x_k) - 0| \leq \epsilon_f \quad \text{that is} \quad |f(x_k)| \leq \epsilon_f.$$

If ϵ_f is below the underflow threshold ($x_{\min} = 10^{-38}$ IEEE single precision) we might as well test for

$$|f(x_k)| = 0,$$

because the only way the test can be passed is if $f(x_k)$ has underflowed to 0. If $\epsilon_f > x_{\min}$ then what should the value be? We cannot answer this without looking at the x -convergence ϵ_x because f and x -convergence are related.

2. x -Convergence

. We show the mathematical relationship between the two types of convergence before we decide on the criteria for either.

We assume throughout the following discussion that the zero-finding algorithm is close to x , where $f(x) = 0$. That is $e_k = |x_k - x|$ is small. Expanding $f(x_k)$ in a Taylor series about the zero (and a fixed point) x we get

$$f(x_k) = f(x) + \frac{1}{1!}f'(x)(x_k - x) + \frac{1}{2!}f''(x)(x_k - x)^2 + \cdots + \frac{1}{n!}f^{(n)}(x)(x_k - x)^n + R_{n+1}.$$

If $|x_k - x|$ is small then $|x_k - x|^2$, $|x_k - x|^3$, etc. are negligible and we have

$$f(x_k) \approx f(x) + f'(x)(x_k - x) \quad \text{or} \quad f(x_k) - f(x) = f'(x)(x_k - x).$$

This shows the relationship between changes in f and x . That is

$$\Delta f_k = f'(x)\Delta x_k.$$

This means we must choose ϵ_f and ϵ_x according to the following rule :

1. If $f'(x) \approx 1$ then choose $\epsilon_f \approx \epsilon_x$.
2. If $f'(x) \ll 1$ then choose $\epsilon_f \ll \epsilon_x$.
3. If $f'(x) \gg 1$ then choose $\epsilon_f \gg \epsilon_x$.

The problem with this rule is that, in general, we do not know $f'(x)$. This forces us to choose ϵ_f and ϵ_x independently so that one or other of the two tests is reasonable under all reasonable circumstances.

The f -convergence test is chosen to be $|f(x_k)| = 0.0$ because zero-finding algorithms, by design, spend most of their time hovering about the machine number 0.0 and so the probability of hitting 0.0 is higher than normal. This behavior can be seen in Figure 4.8

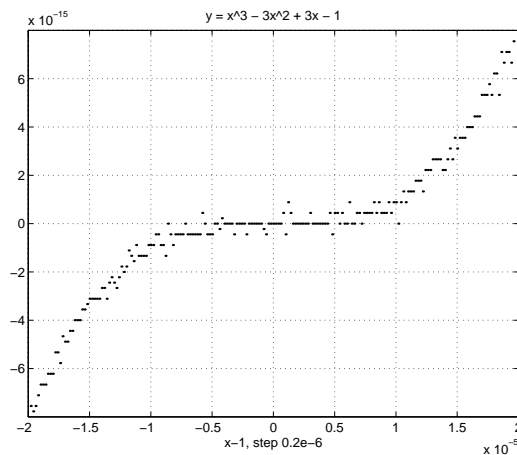


Figure 4.8 : The Function $x^3 - 3x^2 + 3x - 1$ in Machine Numbers

The x - sequence convergence test must be a Cauchy-type test because we do not know the limit of the sequence (that is what the algorithm is trying to find). Let us examine a test that is too often seen in textbooks and amateur programs :

$$\text{IF } |x_k - x_{k-1}| \leq 10^{-6} \quad \text{THEN STOP}$$

If $|x_k| = 10^{12}$ then the floating point number spacing around x_k is $\epsilon_m \times |x_k| = 10^{-16} \times 10^{12} = 10^{-4}$. This means that the two floating point numbers x_k, x_{k-1} can never get closer than 10^{-4} and so passing this test is impossible. If this was the only test used in the algorithm then it would never stop.

If $|x_k| = 10^{-3}$ then the floating point number spacing around x_k is $10^{-16} \times 10^{-3} = 10^{-19}$ and so the test is valid but crude by a factor of 10^{13} .

This example shows that we must account for floating point number spacing when choosing ϵ_x . We do this by making sure that the test never requires two numbers to be closer than the number spacing. The number spacing around x_k is $\epsilon_m x_k$ and the test

$$\text{IF } |x_k - x_{k-1}| \leq 4.0 \times \epsilon_m \times |x_k| \quad \text{THEN STOP}$$

is valid for all magnitudes of x_k .² The floating point number 4.0 in the test is a safety factor. This test ensures that we get almost full precision, i.e., the last two terms of the sequence are as close as they can possibly be in the floating point number system being used.

Sometimes we may not need convergence to full precision because a crude approximation to the limit is all that is required. This is elegantly handled by this modified test :

$$\text{IF } |x_k - x_{k-1}| \leq \epsilon_{\text{tol}} + 4.0 \times \epsilon_m \times |x_k| \quad \text{THEN STOP} \quad (4.13)$$

To see how this test works, suppose that $|x_k| = 0.1$ and we want to stop when successive terms of the sequence get within 10^{-3} of each other. Set $\epsilon_{\text{tol}} = 10^{-3}$ and the test becomes

$$\text{IF } |x_k - x_{k-1}| \leq 10^{-3} + 4.0 \times 10^{-16} \times 10^{-1} \approx 10^{-3} \quad \text{THEN STOP}$$

Also, the test correctly deals with the case if ϵ_{tol} is set too small. Suppose, in the example above, that $\epsilon_{\text{tol}} = 10^{-20}$. This is an impossible requirement because the machine spacing is $10^{-16} \times 10^{-1} = 10^{-17}$. The test is now

$$\text{IF } |x_k - x_{k-1}| \leq 10^{-20} + 4.0 \times 10^{-16} \times 10^{-1} \approx 4.0 \times 10^{-17} \quad \text{THEN STOP}$$

and 4.0×10^{-17} is greater than the spacing about $x_k = 10^{-1}$.

3. Cycling

. If neither f nor x sequence convergence occurs then the algorithm can be stopped after a set number of iterations, *maxits* is reached. Choosing *maxits* depends on the algorithm and the problem. There is no general rule for choosing *maxits*.

4. Divergence

. This occurs when either x_k or $f(x_k) \rightarrow \infty$. We could check the values of x_k or $f(x_k)$ at each iteration or, assuming that the compiler correctly handles IEEE exceptions, check each value for ∞ once at the end of the algorithm.

²Is it valid for $x_k = 0$?

5. Cannot Proceed

. Check for divide by zero or some other exception.

Exercise 4.10. Does the test

$$\text{if } |x_k - x_{k-1}| \leq 4.0 \times \epsilon_m \times |x_k| \text{ then stop}$$

always work. What happens if the algorithm is converging to $x = 0$? Consider finding a zero of the function $f(x) = \tan^{-1} x$.

4.5 IMPLEMENTATION NOTES

The zero-finding algorithms of this chapter are very simple and easily stated, except that great care is needed in designing the stopping rules.

Implementing these algorithms in a programming language so that they run correctly is a more difficult task. Here are some general rules-of-thumb that may help in the translation of an algorithm to a program :

4.5.1 General Rules

1. Direct translation of an algorithm into a program is rarely successful.
2. Remember that numbers have finite precision and that the ideal rules of arithmetic do not always work.
3. Understand the machine number concepts, such as (i) machine epsilon and precision, (ii) number spacing, (iii) overflow and underflow, (iv) largest and smallest f.p. numbers.
4. Always write iteration formulas in *correction form*. For example, the iteration formula for Newton's Algorithm is

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = x_k + \Delta x_k,$$

where Δx_k is the correction to the previous approximation x_k . If x_k is a good approximation and, for some reason, Δx_k is bad, then little harm is done because Δx_k is usually small. However, if we re-write the iteration formula as

$$x_{k+1} = \frac{x_k f'(x_k) - f(x_k)}{f'(x_k)}$$

then the good value x_k has been multiplied by a possibly bad value for $f'(x_k)$ and the entire calculation is contaminated, whereas in the correction form only the correction is contaminated.

Applying this rule to the Bisection algorithm we get

$$m := a + \frac{(b-a)}{2} \text{ rather than } m := \frac{(a+b)}{2}$$

5. Remember that the subtraction of nearly-equal numbers may cause difficulties.
6. Always check for division by zero.

4.5.2 Specific Rules

Newton's Algorithm

This is the simplest to implement. Apply Rule 4 and under Rule 6 check for $f'(x_k) = 0$. Remember that this algorithm has only local convergence and so a good starting value is required.

Secant Algorithm

The advice for Newton applies here except we must check for $f(x_k) = f(x_{k-1})$. instead of $f'(x_k) = 0$.

Bisection Algorithm

Surprisingly, the conceptually simplest algorithm is the hardest to implement correctly. A recent browse through the textbook section of a Dublin university bookstore revealed 4 numerical computation books with the following 'howlers':

1. **Sign Test** : if $f(a)f(b) < 0$ then. This is mathematically correct but what happens if, in single precision, $f(a) = 10^{-20}$ and $f(b) = -10^{-20}$? Yes. The product underflows and is set to 0 and the test is incorrectly failed. This failure is very likely to happen because zero-finding algorithms spend much of their time hovering about $f(x) = 0$. This is a very common error in textbooks, especially if written by numerically naive mathematicians. This is a truly nasty error : if the algorithm is implemented in double precision then the value above will not cause any difficulty. However, such a test is still waiting in ambush until the partners $f(a) \approx f(b) = 10^{-200}$ come along.
2. **Calculating the Mid-Point** : $m := (a + b)/2.0$. This violates Rule 4. It should be written in correction form :

$$m := a + (b - a)/2.0$$

This may seem to be a quibble but consider this example, simplified to make the point obvious : assume floating point decimal arithmetic with 3 digit precision, with round-to-nearest. If $a = 0.982$ and $b = 0.984$ then the correct value is $m = 0.983$. With the first formula we get

$$m := \text{fl}(\text{fl}((a + b))/2.0) = \text{fl}(\text{fl}(1.966)/2.0) = \text{fl}(1.97/2.0) = 0.985.$$

Thus m is *outside* the original interval. Using the correction form we get

$$m := \text{fl}(a + \text{fl}(\text{fl}(b - a)/2.0)) = \text{fl}(a + \text{fl}(0.002/2.0)) = \text{fl}(0.982 + 0.001) = 0.983.$$

Here is some advice from Peter Montgomery :

With binary arithmetic, if there is no exponent overflow or underflow, then this problem will not occur. If a and b are representable, then so are $2*a$ and $2*b$, by changing the exponent. If $a \leq b$ on input, then $2*a \leq a + b \leq 2*b$

The computed $a+b$ will be in the interval $[2*a, 2*b]$ after rounding. Division by 2 adjusts the exponent, so the computed $(a + b)/2$ will be in $[a, b]$. *Peter L. Montgomery, Microsoft Research.*

Although Montgomery's advice is sound, overflow can cause a disaster. If a and b are close to the overflow threshold then $m := a + b$ will overflow and $(a + b)/2$ will be wrong. Using $m := a + (b - a)/2$ will not cause overflow. Here is what happens in MATLAB:

```
>> a=2^1023; b=2^1023;
>> m1 = (a+b)/2
m1 =
    Inf
>> m2 = a + (b-a)/2
m2 =
    8.988465674311580e+307
```

See 4.8.1 in the Notes section for a 'real-life' example.

3. **Convergence Test** : if $|a - b| \leq 10^{-6}$ then. This is not specific to the Bisection algorithm but was used in the the 4 textbooks as the convergence test in their implementation of the Bisection algorithm. As we saw in Section ??, this is a meaningless test which can cause an infinite loop or a wrong answer.

Exercise 4.11. Show that the problem in calculating the mid-point as $m := (a + b)/2$ cannot occur in binary arithmetic, provided $(a + b)$ does not overflow.

Exercise 4.12. The function

$$f(x) = 3x^2 + \frac{1}{\pi^4} \ln(\pi - x)^2 + 1$$

has a pole at $x = \pi$, i.e., $f(x) \rightarrow -\infty$ as $x \rightarrow \pi$. Mathematically, this function has two zeros in the interval $[3.14, 3.15]$ because MATLAB gives the results $f(3.14) = 29.44652582187$ and $f(3.15) = 29.6693849404716$ and the function goes negative in this interval. Try to find these zeros using any method or software.

Can they be found using IEEE D.P. = $\mathbb{F}(2, 53, -1021, 1024)$?

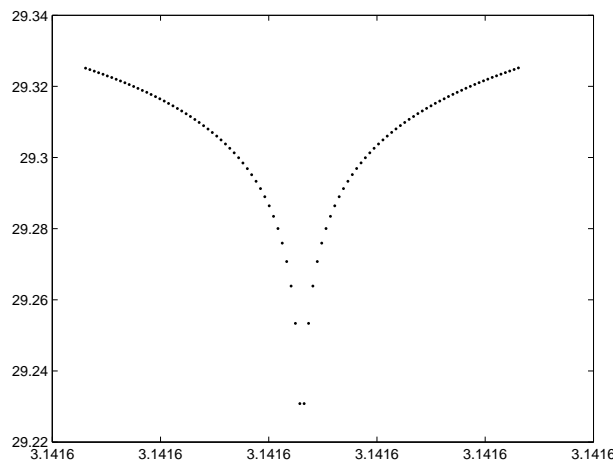


Figure 4.9 : $f(x) = 3x^2 + \frac{1}{\pi^4} \ln(\pi - x)^2 + 1$

4.5.3 Some Examples

Example 4.2. Find the zero of the function $f(x) = e^x + x - 2$ in the range $[-20, 11]$ to full precision.

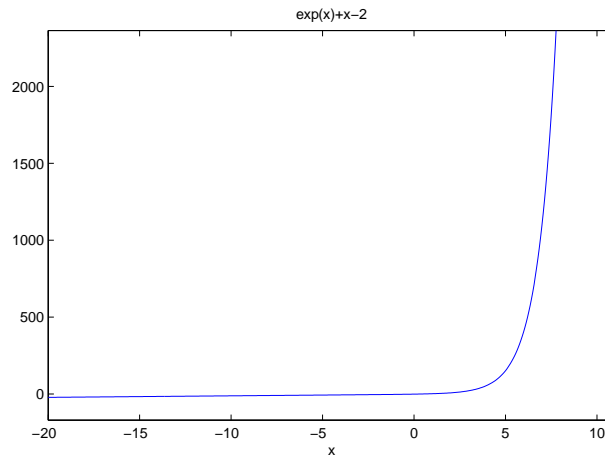


Figure 4.10 : $f(x) = e^x + x - 2$

Table 4.2: Bisection : $f(x) = e^x + x - 2$

k	a	b	E_{rel}	$f(a)$	$f(b)$
1	-20.000000	11.000000	6.888889	-22.000000	59883.141715
2	-4.500000	11.000000	4.769231	-6.488891	59883.141715
3	-4.500000	3.250000	12.400000	-6.488891	27.040340
4	-0.625000	3.250000	2.952381	-2.089739	27.040340
5	-0.625000	1.312500	5.636364	-2.089739	3.027951
6	0.343750	1.312500	1.169811	-0.246024	3.027951
7	0.343750	0.828125	0.826667	-0.246024	1.117148
⋮					
56	0.442854	0.442854	0.000000	-0.000000	0.000000
57	0.442854	0.442854	0.000000	-0.000000	0.000000
58	0.442854	0.442854	0.000000	-0.000000	0.000000

We can see that the Bisection algorithm takes quite a few iterations to reach full precision. If we had started closer to the zero so that $E_{\text{rel}} = 2^{-1}$ then we would expect at most 52 or 53 iterations. Notice that there is a transient of 6 or 7 iterations until the method settles down to a monotonic decrease in E_{rel} .

Table 4.3: Newton : $f(x) = e^x + x - 2$

k	x_{old}	x_{new}	E_{rel}	$f(x_{\text{old}})$	$f'(x_{\text{old}})$
1	-20.000000	2.000000	11.000000	-22.000000	1.000000
2	2.000000	1.119203	0.786986	7.389056	8.389056
3	1.119203	0.582178	0.922440	2.181615	4.062412
4	0.582178	0.448802	0.297184	0.372112	2.789933
5	0.448802	0.442865	0.013405	0.015236	2.566434
6	0.442865	0.442854	0.000024	0.000028	2.557162
7	0.442854	0.442854	0.000000	0.000000	2.557146

Newton's method converges very quickly for this function and a starting value of $x_{\text{old}} = -20$. Again there is a transient of 3 iterations before monotone convergence sets in, after which the number of accurate digits doubles at each iteration.

Table 4.4: Secant : $f(x) = e^x + x - 2$

k	x_1	x_2	E_{rel}	$f(x_1)$	$f(x_2)$
1	11.000000	-19.988615	1.550313	59883.141	-21.988615
2	-19.988615	-19.977241	0.000569	-21.988615	-21.977241
3	-19.977241	2.000000	10.988621	-21.977241	7.389056
4	2.000000	-3.529845	1.566597	7.389056	-5.500535
5	-3.529845	-1.170025	2.016896	-5.500535	-2.859666
6	-1.170025	1.385306	1.844597	-2.859666	3.381356
7	1.385306	0.000840	1648.438056	3.381356	-0.998320
8	0.000840	0.316420	0.997346	-0.998320	-0.311373
9	0.316420	0.459464	0.311326	-0.311373	0.042688
10	0.459464	0.442217	0.039000	0.042688	-0.001629
11	0.442217	0.442851	0.001431	-0.001629	-0.000008
12	0.442851	0.442854	0.000007	-0.000008	0.000000
13	0.442854	0.442854	0.000000	0.000000	-0.000000

The Secant method converges relatively quickly but has a long transient of 7 or 8 iterations. Note the huge E_{rel} at iteration 7. After that, super-linear convergence sets in.

Example 4.3 (Multiple Zeros). A function of the form $f(x) = (x - a)^p g(x)$ has a zero of multiplicity p at $x = a$. Such functions can cause difficulties for all zero-finding methods, as we now demonstrate.

Table 4.5: Bisection on $(x - 1)^5$

k	a	b	E_{rel}	$f(a)$	$f(b)$
1	$-2.000000e + 001$	$1.100000e + 001$	$6.888889e + 000$	$-4.084101e + 006$	$1.000000e + 005$
2	$-4.500000e + 000$	$1.100000e + 001$	$4.769231e + 000$	$-5.032844e + 003$	$1.000000e + 005$
3	$-4.500000e + 000$	$3.250000e + 000$	$1.240000e + 001$	$-5.032844e + 003$	$5.766504e + 001$
4	$-6.250000e - 001$	$3.250000e + 000$	$2.952381e + 000$	$-1.133096e + 001$	$5.766504e + 001$
5	$-6.250000e - 001$	$1.312500e + 000$	$5.636364e + 000$	$-1.133096e + 001$	$2.980232e - 003$
6	$3.437500e - 001$	$1.312500e + 000$	$1.169811e + 000$	$-1.217157e - 001$	$2.980232e - 003$
7	$8.281250e - 001$	$1.312500e + 000$	$4.525547e - 001$	$-1.499904e - 004$	$2.980232e - 003$
8	$8.281250e - 001$	$1.070313e + 000$	$2.551440e - 001$	$-1.499904e - 004$	$1.718552e - 006$
\vdots					
54	$1.000000e + 000$	$1.000000e + 000$	$3.441691e - 015$	$-6.262791e - 075$	$3.187232e - 074$
55	$1.000000e + 000$	$1.000000e + 000$	$1.665335e - 015$	$-6.262791e - 075$	$5.397605e - 079$
56	$1.000000e + 000$	$1.000000e + 000$	$8.881784e - 016$	$-1.311618e - 076$	$5.397605e - 079$

The Bisection method has no difficulty with this function and converges to the multiple zero at $x = 1$ with full precision. However, notice that the function values are about 10^{-80} which is very much larger than the smallest floating point number $\text{realmin} = 2.225073858507201 \times 10^{-308}$.

Table 4.6 shows that Newton's method takes many iterations to converge, thus demonstrating that quadratic convergence has been destroyed by the multiple zero. Again, the final function values are high relative to realmin .

Table 4.6: Newton on $(x - 1)^5$

k	x_{old}	x_{new}	E_{rel}	$f(x_{\text{old}})$	$f'(x_{\text{old}})$
1	$4.000000e + 000$	$3.400000e + 000$	$1.764706e - 001$	$2.430000e + 002$	$4.050000e + 002$
2	$3.400000e + 000$	$2.920000e + 000$	$1.643836e - 001$	$7.962624e + 001$	$1.658880e + 002$
3	$2.920000e + 000$	$2.536000e + 000$	$1.514196e - 001$	$2.609193e + 001$	$6.794772e + 001$
4	$2.536000e + 000$	$2.228800e + 000$	$1.378320e - 001$	$8.549802e + 000$	$2.783139e + 001$
5	$2.228800e + 000$	$1.983040e + 000$	$1.239309e - 001$	$2.801599e + 000$	$1.139974e + 001$
6	$1.983040e + 000$	$1.786432e + 000$	$1.100562e - 001$	$9.180280e - 001$	$4.669332e + 000$
7	$1.786432e + 000$	$1.629146e + 000$	$9.654533e - 002$	$3.008194e - 001$	$1.912558e + 000$
8	$1.629146e + 000$	$1.503316e + 000$	$8.370102e - 002$	$9.857251e - 002$	$7.833839e - 001$
9	$1.503316e + 000$	$1.402653e + 000$	$7.176635e - 002$	$3.230024e - 002$	$3.208741e - 001$
\vdots					
157	$1.000000e + 000$	$1.000000e + 000$	$4.440892e - 016$	$8.692897e - 074$	$1.779515e - 058$
158	$1.000000e + 000$	$1.000000e + 000$	$4.440892e - 016$	$3.187232e - 074$	$7.974454e - 059$
159	$1.000000e + 000$	$1.000000e + 000$	$2.220446e - 016$	$9.071755e - 075$	$2.918254e - 059$

The Secant method takes even more iterations to converge than Newton's method, as we might expect. However, remember that the Secant method requires just 1 function evaluation per iteration,

Table 4.7: Secant on $(x - 1)^5$

k	x_1	x_2	E_{rel}	$f(x_1)$	$f(x_2)$
1	1.100000e + 001	1.025910e + 001	7.221880e - 002	1.000000e + 005	1.000000e + 005
2	1.025910e + 001	8.680885e + 000	1.818035e - 001	6.805247e + 004	6.805247e + 004
3	8.680885e + 000	7.659773e + 000	1.333084e - 001	2.673353e + 004	2.673353e + 004
4	7.659773e + 000	6.678507e + 000	1.469291e - 001	1.310078e + 004	1.310078e + 004
5	6.678507e + 000	5.873426e + 000	1.370717e - 001	5.904330e + 003	5.904330e + 003
6	5.873426e + 000	5.172029e + 000	1.356135e - 001	2.748982e + 003	2.748982e + 003
7	5.172029e + 000	4.575034e + 000	1.304897e - 001	1.263970e + 003	1.263970e + 003
8	4.575034e + 000	4.062324e + 000	1.262111e - 001	5.839842e + 002	5.839842e + 002
9	4.062324e + 000	3.623521e + 000	1.210985e - 001	2.693121e + 002	2.693121e + 002
⋮					
233	1.000000e + 000	1.000000e + 000	4.440892e - 016	2.004093e - 073	2.004093e - 073
234	1.000000e + 000	1.000000e + 000	4.440892e - 016	8.692897e - 074	8.692897e - 074

whereas Newton's method requires 2 function evaluations.

This example shows that the Bisection method is very robust compared to the higher-order algorithms. However, for well-behaved functions it can be very slow.

Example 4.4 (Failure of the Standard Convergence Test). The convergence test we have used in all the algorithms discussed so far is

$$\text{if } |x_k - x_{k-1}| \leq \epsilon_{\text{tol}} + 4.0 \times \epsilon_m \times |x_k| \text{ then stop} \quad (4.14)$$

This test has a 'flaw'. It will fail if the sequence $\{x_k\} \rightarrow 0$. Here is an example, with $f(x) = \tan^{-1} x$, which has a single real zero at $x = 0$, using the Bisection Method.

We can see in Table 4.8 that the Bisection method failed to converge in 100 iterations. Notice that E_{rel} is large and does not decrease. However, a and b seem to be converging to 0, along with the function values. If we allow the Bisection method to run 'forever' then the convergence test (4.14) stops the method after 1078 iterations, as can be seen in Table 4.9.

Table 4.9 shows that the Bisection method is converging to 0, although $E_{\text{rel}} = |a - b|/|a|$ never becomes less than 1.

Why does Bisect take 1078 iterations to converge? We can see that the bracketing interval $[a, b]$ is being reduced by 1/2 at each iteration and that $f(a)$ and $f(b)$ have opposite signs. Thus, Bisection is working properly and will continue iterating because $E_{\text{rel}} > 1$. It stops only when a , b , and the interval $[a, b]$ go to zero. That is, when $|a - b| \leq 4 \times \epsilon_m \times |a|$ goes to $0 \leq 4 \times \epsilon_m \times 0$. This happens when a and b underflow (to zero). Note that the inequality ' \leq ' in the convergence test is vital here.

The smallest *normalized* floating point number, `realmin`, is

$$0.\overbrace{1000 \dots 00}^{53} \times 2^{-1021} = 2^{-1022} = 2.225073858507201 \times 10^{-308}.$$

Underflow does not occur just below this number because the IEEE standard specifies that there should be *gradual underflow*. Rather than jumping from $0.1000 \dots 00 \times 2^{-1021}$ to 0, the standard allows *de-normalized* numbers such as $0.0100 \dots 00 \times 2^{-1021}$, $0.0010 \dots 00 \times 2^{-1021}$, etc. The smallest

Table 4.8: Apparent Failure of Convergence Test on $\tan^{-1} x$

k	a	b	E_{rel}	$f(a)$	$f(b)$
1	$-2.000000e+001$	$1.100000e+001$	$6.888889e+000$	$-1.520838e+000$	$1.480136e+000$
2	$-4.500000e+000$	$1.100000e+001$	$4.769231e+000$	$-1.352127e+000$	$1.480136e+000$
3	$-4.500000e+000$	$3.250000e+000$	$1.240000e+001$	$-1.352127e+000$	$1.272297e+000$
4	$-6.250000e-001$	$3.250000e+000$	$2.952381e+000$	$-5.585993e-001$	$1.272297e+000$
5	$-6.250000e-001$	$1.312500e+000$	$5.636364e+000$	$-5.585993e-001$	$9.197196e-001$
6	$-6.250000e-001$	$3.437500e-001$	$6.888889e+000$	$-5.585993e-001$	$3.310961e-001$
7	$-1.406250e-001$	$3.437500e-001$	$4.769231e+000$	$-1.397089e-001$	$3.310961e-001$
8	$-1.406250e-001$	$1.015625e-001$	$1.240000e+001$	$-1.397089e-001$	$1.012154e-001$
9	$-1.953125e-002$	$1.015625e-001$	$2.952381e+000$	$-1.952877e-002$	$1.012154e-001$
\vdots					
97	$-1.135960e-028$	$2.776790e-028$	$4.769231e+000$	$-1.135960e-028$	$2.776790e-028$
98	$-1.135960e-028$	$8.204153e-029$	$1.240000e+001$	$-1.135960e-028$	$8.204153e-029$
99	$-1.577722e-029$	$8.204153e-029$	$2.952381e+000$	$-1.577722e-029$	$8.204153e-029$
100	$-1.577722e-029$	$3.313216e-029$	$5.636364e+000$	$-1.577722e-029$	$3.313216e-029$

Table 4.9: Convergence of Bisection on $\tan^{-1} x$

k	a	b	E_{rel}	$f(a)$	$f(b)$
1	$-2.000000e+001$	$1.100000e+001$	$6.888889e+000$	$-1.520838e+000$	$1.480136e+000$
2	$-4.500000e+000$	$1.100000e+001$	$4.769231e+000$	$-1.352127e+000$	$1.480136e+000$
3	$-4.500000e+000$	$3.250000e+000$	$1.240000e+001$	$-1.352127e+000$	$1.272297e+000$
4	$-6.250000e-001$	$3.250000e+000$	$2.952381e+000$	$-5.585993e-001$	$1.272297e+000$
5	$-6.250000e-001$	$1.312500e+000$	$5.636364e+000$	$-5.585993e-001$	$9.197196e-001$
6	$-6.250000e-001$	$3.437500e-001$	$6.888889e+000$	$-5.585993e-001$	$3.310961e-001$
7	$-1.406250e-001$	$3.437500e-001$	$4.769231e+000$	$-1.397089e-001$	$3.310961e-001$
8	$-1.406250e-001$	$1.015625e-001$	$1.240000e+001$	$-1.397089e-001$	$1.012154e-001$
9	$-1.953125e-002$	$1.015625e-001$	$2.952381e+000$	$-1.952877e-002$	$1.012154e-001$
\vdots					
1021	$-1.780059e-306$	$9.790325e-307$	$6.888889e+000$	$-1.780059e-306$	$9.790325e-307$
1022	$-4.005133e-307$	$9.790325e-307$	$4.769231e+000$	$-4.005133e-307$	$9.790325e-307$
1023	$-4.005133e-307$	$2.892596e-307$	$1.240000e+001$	$-4.005133e-307$	$2.892596e-307$
\vdots					
1075	$-4.940656e-323$	$1.037538e-322$	$5.166667e+000$	$-4.940656e-323$	$1.037538e-322$
1076	$-4.940656e-323$	$2.964394e-323$	$8.000000e+000$	$-4.940656e-323$	$2.964394e-323$
1077	$-9.881313e-324$	$2.964394e-323$	$4.000000e+000$	$-9.881313e-324$	$2.964394e-323$
1078	$-9.881313e-324$	$9.881313e-324$	Inf	$-9.881313e-324$	$9.881313e-324$

de-normalized floating point number is

$$\overbrace{0.0000\dots 01}^{53} \times 2^{-1021} = 2^{-53} \times 2^{-1021} = \frac{1}{2} \epsilon_m \times 2^{-1021} = 2^{-1074} = 4.940656458412465 \times 10^{-324},$$

and it is at this point that underflow to zero occurs. Thus the Bisection method continues until underflow occurs. It takes 1078 iterations because of the initial transient of 4 or 5 iterations.

Table 4.10: Convergence of Secant on $\tan^{-1} x$

k	x_1	x_2	E_{rel}	$f(x_1)$	$f(x_2)$
1	$1.100000e+001$	$-4.289777e+000$	$3.564236e+000$	$1.480136e+000$	$1.480136e+000$
2	$-4.289777e+000$	$2.980271e+000$	$2.439392e+000$	$-1.341774e+000$	$-1.341774e+000$
3	$2.980271e+000$	$-5.217652e-001$	$6.711901e+000$	$1.247061e+000$	$1.247061e+000$
4	$-5.217652e-001$	$4.528800e-001$	$2.152105e+000$	$-4.809078e-001$	$-4.809078e-001$
5	$4.528800e-001$	$-4.508330e-003$	$1.014540e+002$	$4.252463e-001$	$4.252463e-001$
6	$-4.508330e-003$	$2.898578e-004$	$1.655359e+001$	$-4.508300e-003$	$-4.508300e-003$
7	$2.898578e-004$	$-1.837521e-009$	$1.577450e+005$	$2.898578e-004$	$2.898578e-004$
8	$-1.837521e-009$	$5.146101e-017$	$3.570705e+007$	$-1.837521e-009$	$-1.837521e-009$
9	$5.146101e-017$	0	Inf	$5.146101e-017$	0

We can see from Table 4.10 that the Secant method was lucky : it hit the zero exactly with $x_2 = 0$. This explains why $E_{\text{rel}} = \text{inf}$ — a division by $x_2 = 0$.

4.6 THE ZEROS OF POLYNOMIALS

We now consider functions of the form

$$a_0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n = \sum_{i=1}^n a_i x^i,$$

which is called an n -degree polynomial in x . We have seen these polynomials in Chapter 3 and know that they form a linear vector space \mathbb{P}^n of dimension $n + 1$. One basis for this space is $\{1, x, x^2, \dots, x^n\}$. A vector in this space is $(a_0, a_1, a_2, \dots, a_n)$.

Theorem 4.1 (Fundamental Theorem of Algebra). *The polynomial $x_0 + x_1a^1 + x_2a^2 + \cdots + x_na^n$ with $n \geq 1$ and a_0, a_1, \dots, a_n real or complex and $a_n \neq 0$ has at least one zero, i.e., there exists a number α , real or complex, such that $p_n(\alpha) = 0$.*

Corollary 4.1 (FTA). *The polynomial $p_n(x)$ has n zeros, counting multiplicities.*

Proof. The fundamental theorem says that $p_n(x)$ has at least one zero. Call it α_1 . Then we have

$$p_n(x) = (x - \alpha_1)p_{n-1}(x),$$

where $p_{n-1}(x)$ is a polynomial of degree $n - 1$.

By induction on this process we get

$$p_n(x) = (x - \alpha_1)p_{n-1}(x) = (x - \alpha_1)(x - \alpha_2)p_{n-2}(x) = (x - \alpha_1)(x - \alpha_2) \cdots (x - \alpha_n) \quad \square$$

Evaluation of Polynomials by Nested Multiplication

If we evaluate $a_0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n$ term-by-term then this will require $O(n^2)$ multiplications. We can re-arrange the polynomial as follows :

$$\begin{aligned} p_n(x) &= a_0 + x(a_1 + a_2x + a_3x^2 + \cdots + a_nx^{n-1}) \\ &= a_0 + x(a_1 + x(a_2 + a_3x^2 + \cdots + a_nx^{n-2})) \\ &\quad \vdots \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x(a_n)) \cdots)). \end{aligned}$$

This nested multiplication expression requires only n multiplications. We implement this idea in the algorithm below, where a and b are arrays of coefficients, n is the degree, and α is the point at which the polynomial is to be evaluated.

algorithm EvalPoly (a, α, n, b)

```

    b[n] := a[n]
    for k := n - 1 downto 0 do
        b[k] := a[k] + α × b[k + 1]
    endfor
endalg EvalPoly

```

Claim The value b_0 calculated by the algorithm above is the value $p_n(\alpha)$ and

$$p_n(x) = b_0 + (x - \alpha)p_{n-1}(x) \text{ where } p_{n-1}(x) = b_1 + b_2x + \cdots + b_nx^{n-1}.$$

Proof

$$\begin{aligned} p_n(x) &= b_0 + (x - \alpha)(b_1 + b_2x + \cdots + b_nx^{n-1}) \\ &= (b_0 - \alpha b_1) + (b_1 - \alpha b_2)x + (b_2 - \alpha b_3)x^2 + \cdots + (b_{n-1} - \alpha b_n)x^{n-1} + b_nx^n \end{aligned}$$

From the algorithm we have $b_k = a_k + \alpha b_{k+1}$ or $a_k = b_k - \alpha b_{k+1}$. Also, the algorithm sets $b_n = a_n$. \square

Nested multiplication is sometimes called *synthetic division* or *Horner's Method*.

Example 4.5. Synthetic Division. Re-write a polynomial as follows :

$$p_n(x) = 4 + 2x + 9x^2 + 5x^3 = b_0 + (x - 2)p_{n-1}(x).$$

This is the same as doing the long division $p_n(x)/(x - 2)$:

$$\frac{4 + 2x + 9x^2 + 5x^3}{x - 2} = 40 + 19x + 5x^2 + \frac{84}{x - 2}.$$

The value of $p_n(2)$ is 84 and we have $b_0 = 84$, $b_1 = 40$, $b_2 = 19$, and $b_3 = 5$. From the algorithm, with $\alpha = 2$, we have

$$\begin{aligned} b_3 &= a_3 = 5 \\ b_2 &= a_2 + 2b_3 = 9 + 10 = 19 \\ b_1 &= a_1 + 2b_2 = 2 + 38 = 40 \\ b_0 &= a_0 + 2b_1 = 4 + 80 = 84 \end{aligned}$$

Thus, the algorithm for evaluating a polynomial $p_n(x)$ at $x = \alpha$ is in fact doing synthetic division and it gives

$$\begin{aligned} p_n(x) &= 4 + 2x + 9x^2 + 5x^3 \\ &= b_0 + (x - 2)(b_1 + b_2x + b_3x^2) \\ &= 84 + (x - 2)(40 + 19x + 5x^2). \end{aligned}$$

Evaluating the Derivative of a Polynomial

The derivative of $p_n(x)$ can be evaluated algebraically as follows :

$$\begin{aligned} p_n(x) &= b_0 + p_{n-1}(x)(x - \alpha) \\ p'_n(x) &= p'_{n-1}(x)(x - \alpha) + p_{n-1}(x) \end{aligned}$$

Replacing x with α in the latter equation we get

$$p'_n(\alpha) = p_{n-1}(\alpha).$$

Thus we can evaluate the derivative of a polynomial by a purely algebraic calculation. To evaluate $p_{n-1}(x)$ we need to get its coefficients. These are the b_k 's calculated by EvalPoly. Instead of doing the two jobs separately we roll the two into one algorithm, as shown below.

```

algorithm EvalPolyD ( $a, \alpha, n, b, c$ )


---


 $b[n] := a[n]; c[n] := b[n]$ 
for  $k := n - 1$  downto 1 do
     $b[k] := a[k] + \alpha \times b[k + 1]$ 
     $c[k] := b[k] + \alpha \times c[k + 1]$ 
endfor
 $b[0] := a[0] + \alpha \times b[1]$ 
endalg EvalPolyD  $b_0 = p_n(\alpha), c_1 = p'_n(\alpha)$ 

```

4.6.1 Algorithms for the Zeros of Polynomials

Once we have the algorithm EvalPolyD we may use it in any of the previous algorithms. For example, Newton's Algorithm for polynomials is as follows :

```

algorithm NewtonPoly( $a, n, x_{old}, \epsilon, \text{maxits}$ )


---


EvalPolyD( $a, x_{old}, n, b, c$ )
 $f_{old} := b[0]; f'_{old} := c[1]$ 
for  $k := 1$  to maxits do
     $x_{new} := x_{old} - f_{old}/f'_{old}$ 
    EvalPolyD ( $a, x_{new}, n, b, c$ )
     $f_{new} := b[0]; f'_{new} := c[1]$ 
    if  $|x_{new} - x_{old}| \leq \epsilon$  OR  $f_{new} = 0$  then
        return ( $x_{new}$ )
    else
         $x_{old} := x_{new}$ 
         $f_{old} := f_{new}; f'_{old} := f'_{new}$ 
    endif
endfor
endalg NewtonPoly

```

Deflation of the Polynomial

If we wish to find all the zeros of a polynomial then we need to eliminate those zeros that have been found so far or else use a new starting value close to the next zero to be found.

There are two methods of eliminating the zeros found so far. The first method is called *explicit deflation* or *zero suppression*. This works as follows : suppose we have found a zero α_1 . The polynomial $p_{n-1}(x) = p_n(x)/(x - \alpha_1)$ will not have α_1 as a zero unless α_1 has multiplicity greater than 1. We say that the polynomial has been *deflated by* α_1 . We can extend this to any set of zeros using

$$p_{n-k}(x) = \frac{p_n(x)}{(x - \alpha_1)(x - \alpha_2) \cdots (x - \alpha_k)}$$

Evaluating this deflated polynomial requires the evaluation of $p_n(x)$ and $\prod_{i=1}^k (x - \alpha_i)$. If, in addition, we need to evaluate the deflated derivative then this can be tedious.

The alternative is to use *implicit deflation* of $p_n(x)$. We have

$$p_n(x) = b_0 + p_{n-1}(x)(x - \alpha_1).$$

If we have found a zero at $x = \alpha_1$ then $b_0 = 0$ and

$$p_n(x) = p_{n-1}(x)(x - \alpha_1).$$

The new polynomial deflated by α_1 is

$$p_{n-1}(x) = b_1 + b_2x + \dots + b_nx^{n-1},$$

where the b_k 's were calculated when evaluating $p_n(\alpha_1)$ by *EvalPolyD* at $x = \alpha_1$. We now apply Newton or any other algorithm to $p_{n-1}(x)$ and repeat the process until we get to $p_1(x) = a'_0 + a'_1x$ and we get the final zero $\alpha_n = -a'_0/a'_1$.

Implicit deflation has a major drawback: there will be rounding errors in the calculated coefficients of the deflated polynomial $p_{n-k}(x)$ and so the calculated zero, α_k , for this polynomial will have error. This in turn causes propagated error in the coefficients of the next deflated polynomial $p_{n-k-1}(x)$.

To overcome this roundoff problem the algorithm should switch to using the full polynomial $p_n(x)$ once an approximate zero has been found for the deflated polynomial. This approximate zero can be used as the starting point for the iterations using the full polynomial. This should give a more accurate zero and thus, a more accurate set of b_k 's.

The Condition of Polynomials

In Chapter 2 we defined the *condition* of a problem p as a measure of the change in the solution s due to a change in the problem p . That is,

$$\text{Cond}(p) = \max_{\Delta p} \frac{\|\Delta s\|/\|s\|}{\|\Delta p\|/\|p\|}.$$

Polynomials, even of low degree, can be ill-conditioned, as we will see in the examples that follows.

Example 4.6 (Ill-Condition). The polynomial

$$p_4(x) = (x - 1)^4 = 1 - 4x + 6x^2 - 4x^3 + x^4$$

has 4 zeros at $x = \alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 1$. Let us perturb the first coefficient by 10^{-8} . This gives $p_4(x) = (x - 1)^4 - 10^{-8}$ or $(x - 1)^4 = 10^{-8}$. Hence

$$(x - 1)^2 = \pm 10^{-4} \quad \text{or} \quad x - 1 = \pm \sqrt{\pm 10^{-4}} \quad \text{or} \quad x = 1 \pm \sqrt{\pm 10^{-4}}$$

Hence the zeros of the perturbed polynomial are

$$\alpha_1 = 1 + 10^{-2}, \quad \alpha_2 = 1 - 10^{-2}, \quad \alpha_3 = 1 + i10^{-2}, \quad \alpha_4 = 1 - i10^{-2},$$

We can see that a change of 10^{-8} in the problem has caused a change of 10^{-2} in the solution. Therefore $\text{Cond}(p) \approx 10^{-2}/10^{-8} = 10^6$, which shows that this problem, finding the zeros of $1 - 4x + 6x^2 - 4x^3 + x^4$, is very ill-conditioned.

Figure 4.11 shows the effect of changing $p_4(x) = (x - 1)^4$ to $p_4(x + \epsilon) = (x - 1)^4 - 10^{-12}$. Note the scale in this figure: the perturbed function shifts down by 10^{-12} but its zeros shift from 1.0 to 1.0 ± 10^{-3} . In other words, the change in $p_4(x)$ has been magnified by a factor of $10^{-3}/10^{-12} = 10^9$ in the zeros of $p_4(x)$.

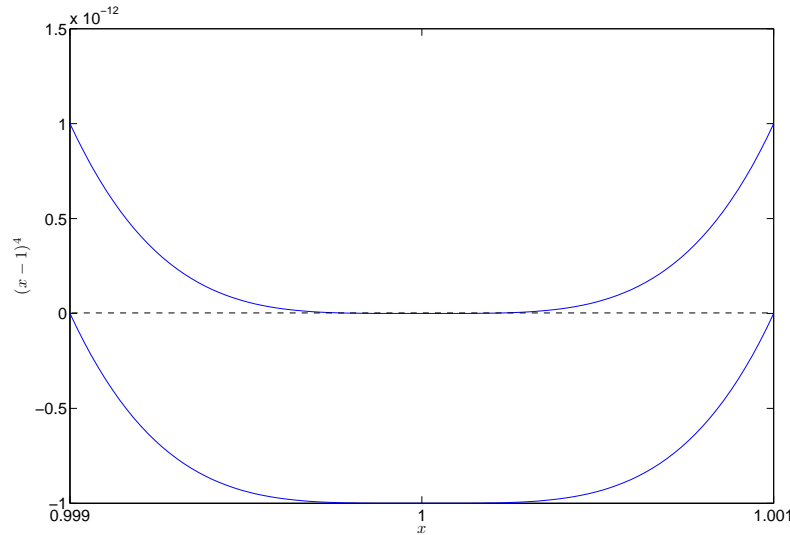


Figure 4.11 : $(x - 1)^4$ and $(x - 1)^4 - 10^{-12}$

The example above shows how multiple zeros are sensitive to small changes and so we may expect trouble with any function that has multiple zeros.

The next example shows that polynomials may be very ill-conditioned even if the zeros are distinct and well-spaced apart.

Example 4.7. Wilkinson's Polynomial The polynomial

$$p_{20}(x) = \prod_{i=1}^{20} (x - i) = 20! + \dots + x^{20},$$

has 20 simple zeros at $x = 1, 2, \dots, 20$. Despite this the polynomial is very sensitive to changes in some of its coefficients.

The condition of the j th zero x_j with respect to changes in the i th coefficient a_i of any polynomial is [see Trefethen & Bau]

$$\text{cond}(p, a_i, x_j) = \frac{|\delta x_j|}{|x_j|} \bigg/ \frac{|\delta a_i|}{|a_i|} = \frac{|a_i x_j^{i-1}|}{|p'(x_j)|} \quad (4.15)$$

For the Wilkinson polynomial the zero $x = 15$ is the most sensitive to changes in the coefficient $a_{15} \approx 1.67 \times 10^9$. The condition number is $\approx 5.1 \times 10^{13}$. The graph below shows a plot of the zeros of 500 randomly perturbed Wilkinson polynomials, where each coefficient a_i is replaced by $a_i(1 + u_i[0, 10^{-7}])$. The Matlab program for this plot was written by Prof J. Demmel, U.C. Berkeley.

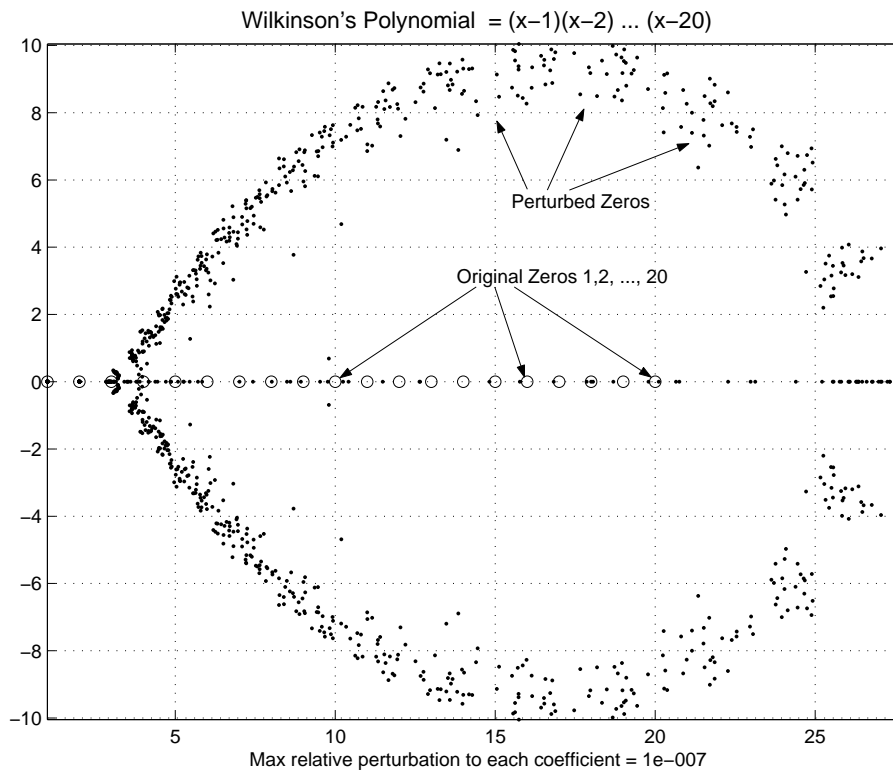


Figure 4.12 : Zeros of Perturbed Wilkinson Polynomial

4.7 ALGORITHMS FOR SYSTEMS OF EQUATIONS

We now wish to solve a system of non-linear equations, i.e., find the n -vector x that satisfies

$$f(x) = 0,$$

where $x \in \mathbb{R}^n$, $0 \in \mathbb{R}^m$ and $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$.

In component form the equation above is

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= 0 \\ f_2(x_1, x_2, \dots, x_n) &= 0 \\ &\vdots \\ f_m(x_1, x_2, \dots, x_n) &= 0 \end{aligned}$$

i.e., m non-linear equations in n unknowns.

TO BE COMPLETED

4.8 NOTES

4.8.1 The Golden Mean, Fibonacci Numbers, and Aspect Ratios

The *Golden Mean or Ratio*

$$\phi = 1.6180339887498948482045868343656\dots,$$

has the property that $1/\phi = 0.618\dots = \phi - 1$. and arises in many different applications. For example, the Greeks considered the most aesthetically pleasing aspect-ratio for buildings to be $\phi : 1$, as shown in Figure 4.5. (Sam Stevenson, the Central Bank architect never heard of this, obviously.)

It also arises in the *Fibonacci Sequence* defined by

$$F_{k+2} = F_{k+1} + F_k, \quad F_0 = F_1 = 1.$$

This gives the sequence

$$\{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots\}$$

where the ratio of successive terms $F_{k+1}/F_k \rightarrow 1.618\dots = \phi$, the Golden Ratio. The solution to the difference equation above is

$$F_k = \frac{1}{\sqrt{5}} \left(\phi^k - \frac{1}{\phi^k} \right) = \frac{\phi^k}{\sqrt{5}}$$

rounded to the nearest integer.

The *continued fraction expansion* of ϕ is interesting :

$$\phi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \dots}}}$$

Aspect Ratios

This is the ratio of the width or horizontal side of a rectangle w to its height or vertical side h , that is, $r = w/h$. Rotating the rectangle so that the long side is vertical is called the *Portrait* orientation; the reverse is called *Landscape*. Thus a rectangle has two aspect ratios. This ambiguity can cost money if you (mis-) order Venetian blinds but is OK for plain sheets of glass. WHY?

The Golden Mean aspect ratio is, of course, $\phi : 1 = 1.618\dots$ in Portrait orientation and $1 : \phi = 0.618\dots$ in Landscape. ϕ is irrational.

The aspect ratio for standard TV and computer monitors is $4/3 = 1.333\dots$, while the modern large TVs are $(4/3)^2 = 16/9 = 1.777\dots$.

European and American Paper Sizes

Printed paper, such as books, reports, newspapers etc. come in all sorts of sizes and aspect ratios. However there are very few standard paper *stock* sizes and these are trimmed to get non-standard sizes.

European or ISO standard paper sizes are A4: 297mm \times 210mm, A3: 420mm \times 297mm, etc. ISO paper has a curious feature : if you fold a sheet of A_x paper in half (long dimension) you get 2 sheets of A_{x+1} paper. For example, two A4 sheets fit on one A3 sheet.

American *Letter* size is 11" \times 8.5", which is a mess. The European or ISO paper standard is one of the few European rules that sensible Americans advocate.

Exercise 4.13. (ISO Paper Aspect Ratio). Given the rule that two A_{x+1} sheets fit on one A_x sheet of ISO paper, what are the aspect ratios of the paper sizes A0, A1, ..., A6?

Exercise 4.14. Printed paper is usually viewed in Portrait orientation (art books and pretentious manuals excepted) while all screens (computer, TV, cinema) are viewed in Landscape. WHY?

Calculating the Mid-Point

"Extra, Extra - Read All About It: Nearly All Binary Searches and Merge-Sorts are Broken" 6/02/2006 08:34:00 AM

Posted by Joshua Bloch, Software Engineer

I remember vividly Jon Bentley's first Algorithms lecture at CMU, where he asked all of us incoming Ph.D. students to write a binary search, and then dissected one of our implementations in front of the class. Of course it was broken, as were most of our implementations. This made a real impression on me, as did the treatment of this material in his wonderful *Programming Pearls* (Addison-Wesley, 1986; Second Edition, 2000). The key lesson was to carefully consider the invariants in your programs.

Fast forward to 2006. I was shocked to learn that the binary search program that Bentley proved correct and subsequently tested in Chapter 5 of *Programming Pearls* contains a bug. Once I tell you what it is, you will understand why it escaped detection for two decades. Lest you think I'm picking on Bentley, let me tell you how I discovered the bug: The version of binary search that I wrote for the JDK contained the same bug. It was reported to Sun recently when it broke someone's program, after lying in wait for nine years or so.

So what's the bug?

Here's a standard binary search, in Java. (It's one that I wrote for the `java.util.Arrays`):

```

1: public static int binarySearch
      (int[] a, int key) {
2:     int low = 0;
3:     int high = a.length - 1;
4:
5:     while(low <= high) {
6:         int mid = (low + high) / 2;
7:         int midVal = a[mid];
8:
9:         if (midVal < key)
10:            low = mid + 1;
11:        else if (midVal > key)
12:            high = mid - 1;
13:        else
14:            return mid; // key found
15:    }
16:    return -(low + 1); // key not found.
17: }
```

The bug is in this line:

```
6: int mid =(low + high) / 2;
```

In *Programming Pearls* Bentley says that the analogous line "sets m to the average of l and u , truncated down to the nearest integer." On the face of it, this assertion might appear correct, but it fails for large values of the int variables `low` and `high`. Specifically, it fails if the sum of `low` and `high` is greater than the maximum positive int value ($2^{31} - 1$). The sum overflows to a negative value, and the value stays negative when divided by two. In C this causes an array index out of bounds with unpredictable results. In Java, it throws `ArrayIndexOutOfBoundsException`.

This bug can manifest itself for arrays whose length (in elements) is 2^{30} or greater (roughly a billion elements). This was inconceivable back in the '80s, when *Programming Pearls* was written, but it is common these days at Google and other places. In *Programming Pearls*, Bentley says "While the first binary search was published in 1946, the first binary search that works correctly for all values of n did not appear until 1962." The truth is, very few correct versions have ever been published, at least in mainstream programming languages.

So what's the best way to fix the bug? Here's one way:

```
6: int mid = low + ((high - low) / 2);
```

Probably faster, and arguably as clear is:

```
6: int mid = (low + high) >>> 1;
```

In C and C++ (where you don't have the `>>>` operator), you can do this:

```
6: mid = ((unsigned) (low + high)) >> 1;
```

And now we know the binary search is bug-free, right? Well, we strongly suspect so, but we don't know. It is not sufficient merely to prove a program correct; you have to test it too. Moreover, to be really certain that a program is correct, you have to test it for all possible input values, but this is seldom feasible. With concurrent programs, it's even worse: You have to test for all internal states, which is, for all practical purposes, impossible.

The binary-search bug applies equally to MergeSort, and to other divide-and-conquer algorithms. If you have any code that implements one of these algorithms, fix it now before it blows up. The general lesson that I take away from this bug is humility: It is hard to write even the smallest piece of code correctly, and our whole world runs on big, complex pieces of code.

We programmers need all the help we can get, and we should never assume otherwise. Careful design is great. Testing is great. Formal methods are great. Code reviews are great. Static analysis is great. But

none of these things alone are sufficient to eliminate bugs: They will always be with us. A bug can exist for half a century despite our best efforts to exterminate it. We must program carefully, defensively, and remain ever vigilant.

Resources³

J. Bentley, *Programming Pearls*- Highly recommended. Get a copy today!

The Sun bug report describing this bug in the JDK

A 2003 paper by Salvatore Ruggieri discussing a related problem - The problem is a bit more general but perhaps less interesting: the average of two numbers of arbitrary sign. The paper does not discuss performance, and its solution is not fast enough for use in the inner loop of a MergeSort.

Joshua Bloch is a Senior Staff Engineer at Sun Microsystems, Inc., where he is an architect in the Core

Java Platform Group. He designed, implemented, and maintained many parts of the Java platform, including the award-winning Java Collections Framework, the `assert` construct, and the `java.math` package. He led the JCP expert groups for Assertions (JSR-14), Preferences (JSR-10), and Metadata (JSR-175). Previously he was a Senior Systems Designer at Transarc Corporation, where he designed and implemented many parts of the Encina distributed transaction processing system.

In addition to *Effective Java Programming Language Guide*, Bloch wrote chapters in *The Java Tutorial Continued* and *Camelot and Avalon - A Distributed Transaction Facility*. He has also written a handful of technical papers.

He holds a B.S. from Columbia University and a Ph.D. from Carnegie-Mellon University. His Ph.D. thesis on the replication of abstract data objects was nominated for the ACM Distinguished Dissertation Award.

You can read the original at

<http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html>

4.8.2 Overflow and Underflow

The following is from Michael Overton's book ??.

Overflow. Traditionally, overflow is said to occur when the exact result of a floating point operation is finite but with an absolute value that is larger than the largest floating point number. As with division by zero, in the days before IEEE arithmetic was available the usual treatment of overflow was to set the result to (plus or minus) the largest floating point number or to interrupt or terminate the program. In IEEE arithmetic, the standard response to overflow is to deliver the correctly rounded result, either $\pm N_{\max}$ or $\pm\infty$. The range of numbers that round to $\pm\infty$ depends on the rounding mode; see Chapter 5.

To be precise, overflow is said to occur in IEEE arithmetic when the exact result of an operation is finite but so big that its correctly rounded value is different from what it would be if the exponent upper limit E_{\max} were sufficiently large. In the case of *round to nearest*, this is the same as saying that overflow occurs when an exact finite result is rounded to $\pm\infty$, but it is not the same for the other rounding modes. For example, in the case of *round down* or *round towards zero*, if an exact finite result x is more than N_{\max} , it is rounded down to N_{\max} no matter how large x is, but overflow is

said to occur only if $x \geq N_{\max} + \text{ulp}(N_{\max})$, since otherwise the rounded value would be the same even if the exponent range were increased.

Gradual Underflow. Traditionally, underflow is said to occur when the exact result of an operation is nonzero but with an absolute value that is smaller than the smallest normalized floating point number. In the days before IEEE arithmetic, the response to underflow was typically, though not always, flush to zero: return the result 0. In IEEE arithmetic, the standard response to underflow is to return the correctly rounded value, which may be a subnormal number, ± 0 or $\pm N_{\min}$. This is known as *gradual underflow*. Gradual underflow was and still is the most controversial part of the IEEE standard. Its proponents argued (and still do) that its use provides many valuable arithmetic rounding properties and significantly adds to the reliability of floating point software. Its opponents argued (and still do) that arithmetic with subnormal numbers is too complicated to justify inclusion as a hardware operation which will be needed only occasionally. The ensuing debate accounted for much of the delay in the adoption of the IEEE standard. Even today, some IEEE compliant microprocessors support gradual underflow only in software. The standard gives several options for defining exactly when the underflow exception is said to occur; see

³These used to be called 'References' — doc

[CKVV02] for details. The motivation for gradual underflow can be summarized very simply: compare Figure 3.1 with Figure 4.1 to see how the use of subnormal numbers fills in the relatively large gap between $\pm N_{\min}$ and zero. The immediate consequence is that

the worst case absolute rounding error for numbers that underflow to subnormal numbers is the same as the worst case absolute rounding error for numbers that round to N_{\min} . This is an obviously appealing property.