AMTH250

Nonlinear Equations, Optimization

Gary Bunting

October 10, 2011

Contents

1 Nonlinear Equations

The problem we will consider in this section is solving a nonlinear equation $f(x) = 0$ for x. The example we will use is

$$f(x) = x - 1 - 0.5 \sin x \tag{1}$$

which is a special case of Kepler's equation of orbital mechanics.

For our example we first define the function

```
function y = kepler(x)
  y = x - 1 - 0.5*sin(x);
endfunction
```

It is usual to define such functions in an m-file. Remember the name of the m-file has to be the same as the name of the function, so the m-file is named `kepler.m`.

The first step in solving any equation numerically is to graph the equation to obtain an idea of the number and approximate location of the roots.

```
octave:> x = linspace(-10,10,101);
octave:> plot(x, kepler(x))
octave:> hold on
octave:> plot([-10 10], [0 0])
```

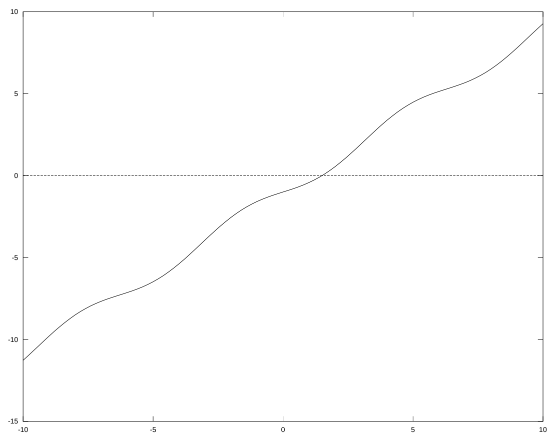


Figure 1: Kepler's equation

From the graph we see that there is one root and that it is between 0 and 3. The `axis` command can be used to zero in on the root:

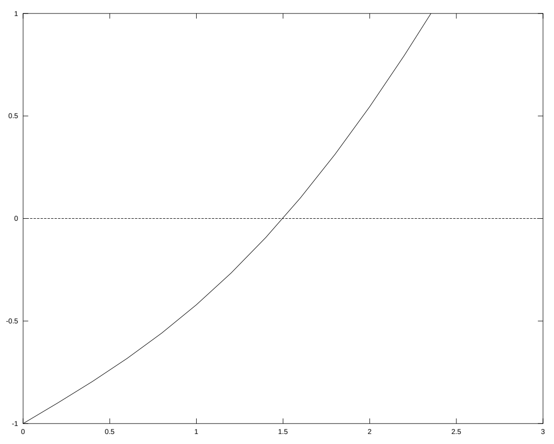


Figure 2: Kepler's equation $x \in [0, 3]$

```
octave:> axis([0 3 -1 1])
```

Now we see the root is near $x = 1.5$ and we can use `fzero` to find the root. We need to specify an interval containing the root.

```
octave:> format long
octave:> fzero(@kepler, [1 2])
ans =  1.49870113351785
```

1.1 Functions in Octave

Function Handles

The `@` in `@kepler` is what in Octave is called a *function handle*. It is used in Octave whenever we require one Octave function to be passed as an argument of another Octave function. In our example the function `kepler` is passed as an argument to the function `fzero`.

Function handles are also used with built-in Octave functions. To approximate π we could use

```
octave:> fzero(@sin, [3 3.2])
ans =  3.14159265358979
```

Anonymous Functions

The `@` notation is also used in Octave to define what it calls *anonymous functions*, for example

```
  @(x) x - 1 - 0.5*sin(x);
```

Like function handles, these can be passed as arguments to other Octave functions. We could solve Kepler's equation by

```
octave:> fzero(@(x) x - 1 - 0.5*sin(x), [0 2])
ans =  1.49870113351785
```

Anonymous functions are used to define one-line functions, or more precisely function handles. For example

```
octave:> kepler = @(x) x - 1 - 0.5*sin(x);
octave:> fzero(kepler, [0 3])
ans =  1.49870113351785
```

No `@` is now needed with `fzero` because this version of `kepler` *is* a function handle.

`ezplot`

`ezplot` provides a quick way to plot functions without having to explicitly create the vectors of x and y data needed by `plot`. For example to plot $f(x) = 1/(1 - x^2)$ for $x \in [-2, 2]$ we could use

```
octave:> ezplot(@(x) 1./(1-x.^2), [-2,2])
```

1.2 Attainable Accuracy

Irrespective of the algorithm being used, there is a limit to the accuracy that can be attained in solving a nonlinear equation due to the errors in function evaluation caused by rounding.

Example

The following is an example of the small but often unpredictable rounding errors which occur in the numerical evaluation of a function. Let

$$f(x) = (x - 1)^6 = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1.$$

We will evaluate $f(x)$ near the root $x = 1$:

```
octave:> x = 0.995 : 0.0001: 1.005;
octave:> y1 = (x - 1).^6;
octave:> y2 = x.^6 - 6*x.^5 + 15*x.^4 - 20*x.^3 + 15*x.^2 - 6*x + 1;
octave:> plot(x, y1, x, y2, [0.995 1.005], [0 0])
octave:> axis([0.995 1.005 -1e-14 2e-14])
```

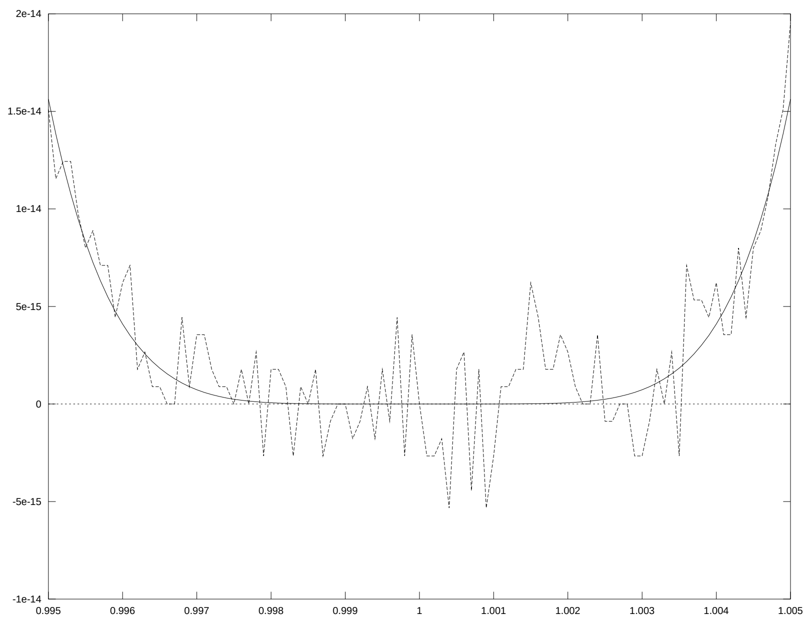


Figure 3: Rounding error in function evaluation.

When evaluated in the expanded form, it shows small (of the order of 10^{-15}) seemingly random fluctuations in the function values. In expanded form the function has numerous zeros in the range $(0.996, 1.004)$, so we certainly couldn't hope to find the true zero with accuracy greater than 0.004.

To analyse this effect quantitatively the most important thing to note is that the equation we are trying to solve will be, when viewed computationally, affected by rounding error. Let x be the exact solution of $f(x) = 0$, let $\hat{f}(x)$ the function actually computed, and let $\hat{x}$ the computed solution. We will assume that the computed solution is an exact solution of the computed function $\hat{f}(x)$, $\hat{f}(\hat{x}) = 0$, that is we assuming that the only error is that arising from function evaluation.

Let $\hat{x} = x + \Delta x$ and $\hat{f}(x) = f(x) + \Delta f(x)$. Then, using the first order Taylor series approximation, we have

$$\begin{aligned} 0 = \hat{f}(\hat{x}) &= f(\hat{x}) + \Delta f(\hat{x}) \\ &= f(x + \Delta x) + \Delta f(\hat{x}) \\ &\approx f(x) + f'(x)\Delta x + \Delta f(\hat{x}) \\ &= f'(x)\Delta x + \Delta f(\hat{x}) \end{aligned}$$

From which we get

$$\Delta x \approx -\frac{\Delta f(\hat{x})}{f'(x)} \tag{2}$$

$\Delta f(\hat{x})$, which is the difference between $f(x)$ and the function actually computed, can be interpreted as the rounding error in evaluating $f(x)$. This difference will typically be of order $\varepsilon_{\text{mach}}$.

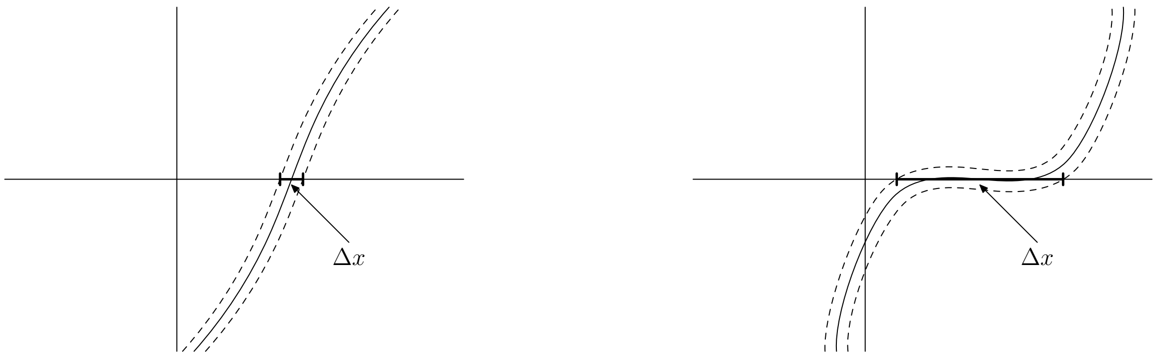


Figure 4: Errors in the Solution of Nonlinear Equations

What formula (2) shows is that the accuracy which we can determine the solution of $f(x) = 0$ depends on the slope of the function at the root. When the graph of $f(x)$ is flat, i.e the derivative $f'(x)$ is small near the root, it will be difficult to estimate the root accurately no matter what method is used.

A root of $f(x) = 0$ where $f'(x) = 0$ is called a **double root** of $f(x)$. In this case the analysis above breaks down and we have to go to the next term in Taylor series

$$f(x + \Delta x) \approx f(x) + \frac{1}{2} f''(x)(\Delta x)^2$$

resulting in

$$\Delta x \approx \sqrt{\frac{2\Delta f(\hat{x})}{f''(x)}}$$

In the example with the expanded form of $(x - 1)^6$ the first 5 derivatives vanish at the root and

$$f(x + \Delta x) \approx f(x) + \frac{1}{6!} f^{(6)}(x)(\Delta x)^6$$

Then

$$\Delta x \approx \left(\frac{6!\, \Delta f(\hat{x})}{f^{(6)}(x)} \right)^{1/6}$$

Applied to the example, from Figure 1.2, we can take $\Delta f(\hat{x}) \approx 5 \times 10^{15}$ and then

$$\begin{aligned} \Delta x &\approx \left(\frac{6!\,\Delta f(\hat{x})}{f^{(6)}(x)} \right)^{1/6} \\ &\approx \left(\frac{6!\,5 \times 10^{15})}{6!} \right)^{1/6} \\ &\approx 0.004 \end{aligned}$$

which agrees well with what is observed.

1.3 Order of Convergence

Like many numerical algorithms, algorithms for solving nonlinear equations compute a sequence of approximations $x_0, x_1, \ldots$ to the solution, the sequence terminating when we have judged that we have a sufficiently accurate approximation. Let $x_1, x_2, \ldots$ be a sequence of approximations to the some number x. In the present context we can think of a series of approximations to the solution of a nonlinear equation.

The error in the k-the approximation is

$$e_k = x_k - x.$$

If the approximations x_k converge to x then $\lim_{k\to\infty} e_k = 0$. What we want is a measure of how fast e_k converges to 0.

The sequence x_k is said to converge to x with **order of convergence** r if

$$\lim_{k\to\infty} \frac{\|e_{k+1}\|}{\|e_k\|^r} = C$$

for some constant $C > 0$. This says that for large k

$$\|e_{k+1}\| \approx C\|e_k\|^r,$$

that is, the error in the $(k+1)$-th approximation is proportional to the r-th power of the error in the k-th approximation.

Some particular cases are:

1. **Linear Convergence**, $r = 1$. Here we have

   $$\|e_{k+1}\| \approx C\|e_k\|$$

   and the error decreases by a factor of C at each iteration. Note that C must be less than one for convergence to occur. The number C is sometimes called the **rate of convergence** in this case.

2. **Superlinear Convergence**, $r > 1$.

3. **Quadratic Convergence**, $r = 2$. Here we have

$$\|e_{k+1}\| \approx C\|e_k\|^2$$

and the error at each iteration is proportional to the square of the error at the previous iteration.

The higher the order of convergence r the faster the convergence. For example, the sequence

$$e_k = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, \ldots$$

shows linear convergence (with C = 0.1), while the sequence

$$e_k = 10^{-1}, 10^{-2}, 10^{-4}, 10^{-8}, \ldots$$

shows quadratic convergence (with C = 1).

1.4 Fixed Point Iteration

Given an equation

$$f(x) = 0$$

rewrite it in an equivalent form

$$x = g(x).$$

Start with an approximation x_0 to the solution and compute a sequence of approximations via the **iteration**

$$x_{k+1} = g(x_k).$$

That is

$$\begin{aligned} x_1 &= g(x_0) \\ x_2 &= g(x_1) \\ x_3 &= g(x_2) \\ &\vdots \end{aligned}$$

We will see that under certain conditions the sequence of approximations x_k converges to the solution x of $f(x) = 0$.

For example, Kepler's equation

$$f(x) = x - 1 - 0.5\sin(x)$$

can be written

$$x = 1 + 0.5\sin(x).$$

which gives the iteration

$$x_{k+1} = 1 + 0.5\sin(x_k).$$

Starting with approximate solution $x_0 = 1.5$, we compute

$$\begin{aligned}
x_1 &= 1 + 0.5\sin(1.5) = 1.4987475 \\
x_2 &= 1 + 0.5\sin(1.4987475) = 1.4987028 \\
x_3 &= 1 + 0.5\sin(1.4987028) = 1.4987012 \\
x_4 &= 1 + 0.5\sin(1.4987012) = 1.4987011
\end{aligned}$$

which is clearly converging.

1.4.1 Theory

We will look at the general theory of the iteration

$$x_{k+1} = g(x_k). \tag{3}$$

First, suppose the the sequence x_k converges to a number x^*, so that

$$\lim_{k\to\infty} x_k = x^*.$$

Taking limits of both sides of Equation (3) we have

$$\lim_{k\to\infty} x_{k+1} = \lim_{k\to\infty} g(x_k)$$

giving

$$x^* = g(x^*)$$

(This requires that $g(x)$ be continuous at x^*.)

Therefore we see that if an iteration

$$x_{k+1} = g(x_k)$$

converges, then it converges to a solution of

$$x = g(x).$$

Such a solution is called a **fixed point** of the function $g(x)$. This name comes from the fact that if we start an iteration $x_{k+1} = g(x_k)$ at a fixed point x_0, then it remains at that point, since, if x_0 satisfies $x_0 = g(x_0)$, then $x_1 = x_0$ and so on.

Now we turn to the question of when the iteration (3) converges. Let x^* be a fixed point of $g(x)$ so $x^* = g(x^*)$, and let $x_k = x^* + e_k$, where e_k can be thought of as the error in the approximation x_k to x^*. We will assume that x_k is close to x^* so that e_k is small and use the approximation

$$g(x + \epsilon) \approx g(x) + g'(x)\epsilon$$

valid for small ϵ. The iteration can be written

$$\begin{aligned} x^* + e_{k+1} &= g(x^* + e_k) \\ &\approx g(x^*) + g'(x^*)e_k \end{aligned}$$

Now, since $x^* = g(x^*)$,

$$e_{k+1} \approx g'(x^*)e_k.$$

The sequence x_k will converge to x^* provided e_k converges to zero. Let $g'(x^*) = K$ then

$$e_{k+1} \approx Ke_k$$

and e_k will be multiplied by a factor of K at each iteration. Thus[1] e_k will converge to zero provided $|K| = |g'(x^*)| < 1$. Conversely, if $|K| = |g'(x^*)| > 1$, then e_k will increase and the iteration $x_{k+1} = g(x_k)$ will diverge away from the fixed point x^* (but may converge to a different fixed point). Finally, if $|K| = |g'(x^*)| = 1$, then the iteration may or may not converge.

Further, from

$$e_{k+1} \approx Ke_k$$

we see that when $0 < |K| < 1$, the iteration is *linearly convergent.* When $K = g'(x^*) = 0$ convergence turns out to be at least quadratic. We will see an example of this later with Newton's method.

In our example involving Kepler's equation

$$x = 1 + 0.5\sin(x)$$

we have

$$g(x) = 1 + 0.5\sin(x)$$

and

$$g'(x) = 0.5\cos(x).$$

Since $\cos(x)$ is never greater than one in magnitude, we have

$$|g'(x)| \leq 0.5$$

and our iteration will converge.

In summary: Let x^* be a fixed point of $g(x)$,

$$x^* = g(x^*)$$

and consider the iteration

$$x_{k+1} = g(x_k).$$

Then

[1]This argument can be made rigorous using the mean value theorem.

1. If $|g'(x^*)| < 1$ the iteration will converge to x^* provided the initial approximation x_0 is sufficiently close to x^*.

2. If $|g'(x^*)| > 1$ then the iteration will diverge from x^*.

1.4.2 Application

Many algorithms for solving nonlinear equations can be viewed as fixed point methods. Here we will look at direct application of the method.

To apply the fixed point method to an equation $f(x) = 0$ we need to rewrite it in an equivalent fixed point form $x = g(x)$. This can usually be done in many ways.

Consider, for example, the equation

$$f(x) = x^2 - x - 2 = 0.$$

This has two solutions $x = -1$ and $x = 2$. The equation $f(x) = 0$ can be written in the following fixed point forms:

1. $x = x^2 - 2, \quad g(x) = x^2 - 2.$

2. $x = \sqrt{x+2}, \quad g(x) = \sqrt{x+2}.$

3. $x = 1 + 2/x, \quad g(x) = 1 + 2/x.$

To determine whether the fixed point iteration converges, we need to examine the derivative of $g(x)$ at the fixed points. In general, we will have only approximate values for the fixed points, obtained, for example, by graphing $f(x)$. This will usually allows us to determine whether $|g'(x)| < 1$ and hence whether the fixed point iteration converges. For the examples above:

1. $g'(x) = 2x$, $g'(-1) = -2$, $g'(2) = 4$, and the fixed point iteration diverges in both cases.

2. $g'(x) = 1/(2\sqrt{x+2})$, $g'(-1) = 1/2$, $g'(2) = 1/4$, and the fixed point iteration converges in both cases.

3. $g'(x) = -2/x^2$, $g'(-1) = -2$, $g'(2) = -1/2$, and the fixed point iteration converges for the second fixed point but not the first.

Example

We will see how this works in practice, using the third example above. The iteration is

$$x_{k+1} = 1 + \frac{2}{x_k}$$

We will start at $x_0 = 1$ and perform 20 iterations:

```
octave:> g = @(x) 1 +2./x;
octave:> x =zeros(1,21);
octave:> x(1) =1;
octave:> for k = 1:20
> x(k+1) = g(x(k));
> end
x =
 Columns 1 through 8:
   1.0000   3.0000   1.6667   2.2000   1.9091   2.0476   1.9767   2.0118
 Columns 9 through 16:
   1.9942   2.0029   1.9985   2.0007   1.9996   2.0002   1.9999   2.0000
 Columns 17 through 21:
   2.0000   2.0000   2.0000   2.0000   2.0000
```

We can see the iteration converging to $x = 2$ although fairly slowly. Computing the errors:

```
octave:> err = abs(x-2)
err =
 Columns 1 through 6:
   1.0000e+00   1.0000e+00   3.3333e-01   2.0000e-01   9.0909e-02   4.7619e-02
 Columns 7 through 12:
   2.3256e-02   1.1765e-02   5.8480e-03   2.9326e-03   1.4641e-03   7.3260e-04
 Columns 13 through 18:
   3.6617e-04   1.8312e-04   9.1550e-05   4.5777e-05   2.2888e-05   1.1444e-05
 Columns 19 through 21:
   5.7220e-06   2.8610e-06   1.4305e-06
```

We see that after a few steps the error approximately halves at each iteration in accord with theory. Computing the ratio of the errors at each step confirms this:

```
octave:> ratio = err(2:21)./err(1:20)
ratio =
 Columns 1 through 8:
   1.00000   0.33333   0.60000   0.45455   0.52381   0.48837   0.50588   0.49708
 Columns 9 through 16:
   0.50147   0.49927   0.50037   0.49982   0.50009   0.49995   0.50002   0.49999
 Columns 17 through 20:
   0.50001   0.50000   0.50000   0.50000
```

For this example, $K = |g'(2)| = 1/2$ and the observed errors are consistent with the relation

$$e_{k+1} \approx Ke_k$$

derived earlier.

1.5 Bisection Algorithm

This algorithm is based on a special case of the intermediate value theorem: if a continuous function $f(x)$ has opposite signs at two points a and b, then it must have a root in the interval (a, b).

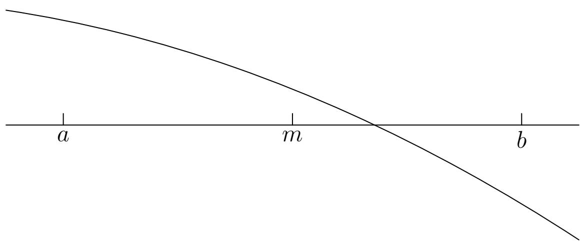


Figure 5: Bisection Method

Take the midpoint m of the interval $[a, b]$. Depending on the sign of $f(m)$ we can determine in which of the subintervals $[a, m]$ or $[m, b]$ the root lies. Repeating this process we get the **bisection algorithm**.

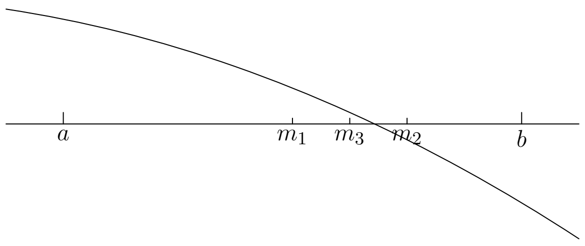


Figure 6: Bisection algorithm

The bisection algorithm is easy to implement in Octave. The following function computes the midpoint `m` of the current interval `[a,b]`, tests the sign of `f(m)`, and adjusts the endpoints until the length of the interval is less than `tol`.

```
function [a,b] = bisect (f, a0, b0, tol)
  if (sign(f(a0)) == sign(f(b0)))
    error ("function has same sign at both endpoints")
  end
  a = a0;
  b = b0;
  while (abs(a-b) > tol)
    m = a + (b-a)/2;
    if (sign(f(m)) == sign(f(a)))
      a = m;
    else
      b = m;
    end
  end
endfunction
```

Example

For Kepler's equation:

```
octave:> [a,b] = bisect(@kepler, 1, 2, 1e-14)
a =  1.49870113351785
b =  1.49870113351786
```

It is easy to see that the bisection algorithm is linearly convergent with rate of convergence 1/2, since at each iteration the size of the interval containing the root is decreased by a factor of 2. Further, it is easy to calculate the number of iterations needed to achieve a given accuracy. Starting with an interval $[a, b]$, after k iterations the length of the interval containing the root is $(b - a)/2^k$, so to obtain an interval of length Δ requires

$$\log_2 \left(\frac{b - a}{\Delta} \right)$$

iterations.

One important property of the bisection algorithm, not shared by most other algorithms, is that it always converges – provided only that the function is continuous and we start with an interval on which the function has opposite signs at the two endpoints.

1.6 Newton's Method

The idea behind Newton's method is simple. Given an approximation x_k to a solution of $f(x) = 0$, approximate $f(x)$ by its tangent at x_k and take the point at which the tangent intersects the x-axis as the next approximation x_{k+1}.

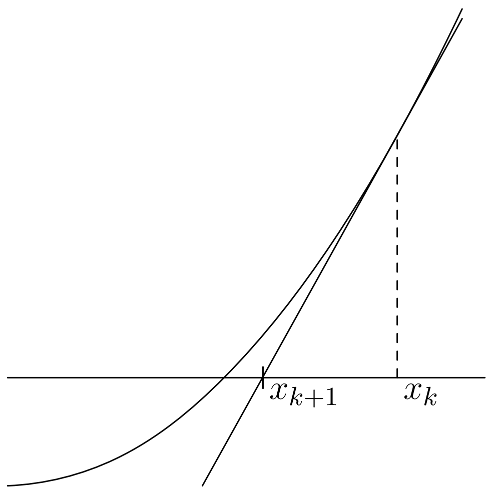


Figure 7: Newton's Method

Newton's method is easy to program in Octave. The following function performs `n` iterations of Newton's method starting with the initial approximation `x0`. Note that we need the derivative of the function $f(x)$ as well as the function itself.

```
function x = newton (x0, f, df, n)
  x = zeros(1,n+1);
  x(1) = x0;
  for k = 1:n
    x(k+1) = x(k) - f(x(k))/df(x(k));
  end
endfunction
```

Applying this to Kepler's equation gives

```
octave:> function dy = dkepler(x)
> dy = 1 - 0.5*cos(x);
> endfunction
octave:> x = newton(1.5, @kepler, @dkepler, 5)
x =
 Columns 1 through 4:
   1.50000000000000   1.49870156963680   1.49870113351790   1.49870113351785
 Columns 5 and 6:
   1.49870113351785   1.49870113351785
```

We have converged to full accuracy in 3 iterations.

Newton's method can be viewed as a fixed point iteration

$$x_{k+1} = g(x_k)$$

with

$$g(x) = x - \frac{f(x)}{f'(x)}$$

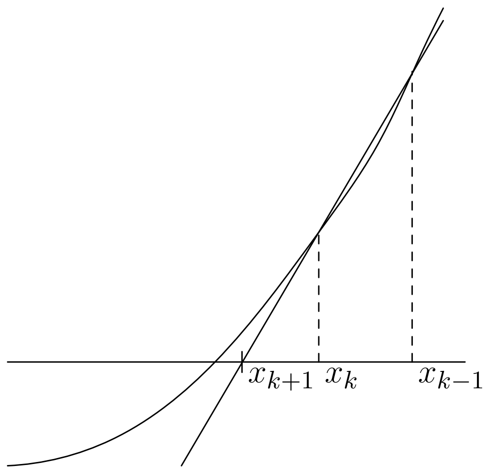
The derivative of $g(x)$ is

$$g'(x) = \frac{f(x)f''(x)}{f'(x)^2}$$

A solution of $f(x) = 0$ for which $f'(x) = 0$ is called a **multiple root** of $f(x)$. When $f(x) = 0$, and provided $f'(x) \neq 0$, we also have $g'(x) = 0$. So, by the results on fixed point iteration, Newton's method is *quadratically convergent* except at multiple roots. For a multiple root it turns out that Newton's method is only linearly convergent.

1.7 Secant Method

The secant method is similar to Newton's method except that we approximate the function by a secant rather than by the tangent. Given two approximations x_{k-1} and x_k to a solution of $f(x) = 0$, approximate $f(x)$ by its secant at x_{k-1} and x_k and take the point at which the secant intersects the x-axis as the next approximation x_{k+1}.



A simple geometric calculation gives

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

which is the iteration formula for the **secant method**. Another way to get this formula is to replace $f'(x)$ in Newton's method by the approximation

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

Below is a Octave function for the secant method [2]. It takes two initial approximations `x0` and `x1` to get started.

```
function x = secant (x0, x1, f, n)
  x = zeros(1, n+2);
  x(1) = x0;
  x(2) = x1;
  for k = 2:n+1
    if (x(k) == x(k-1))
      x(k+1) = x(k);
    else
      x(k+1) = x(k) - f(x(k))*(x(k) - x(k-1))/(f(x(k)) - f(x(k-1)));
    end
  end
endfunction
```

For Kepler's equation it gives:

```
octave:> x = secant(1.5, 2, kepler, 5)
x =
 Columns 1 through 4:
   1.50000000000000   2.00000000000000   1.49884900798874   1.49871797360755
 Columns 5 through 7:
   1.49870113416196   1.49870113351785   1.49870113351785
```

The convergence is quite rapid, but not as fast as for Newton's method (this could also have something with the starting values.) It turns out that the secant algorithm has order of convergence $r \approx 1.62$. except at multiple roots when, like Newton's method, it is only linearly convergent. Again, like Newton's method, it requires good initial approximations to ensure convergence.

1.8 Practical Algorithms

The essential properties of the algorithms we have studied can be summarized as follows

Method	Always Converges	Order of Convergence	Requires Derivatives
Bisection	Yes	Linear	No
Secant	No	1.62	No
Newton	No	Quadratic	Yes

[2]The `if` statement in the `for` loop is needed to avoid division by zero when two successive iterates are the same, a situation which will occur whenever it has converged to full accuracy.

It is clear that each of these algorithms has its advantages and disadvantages. The methods used in practice, for example Octave's `fzero`, use a combination of methods to obtain the advantages of each. These algorithms tend to be quite sophisticated; for example `fzero` uses a combination of bisection, secant and interpolation methods.

A simple way to combine the bisection method with other more rapidly convergent algorithms is to start with an interval known to contain the solution and try the faster method. Continue with the faster method while it makes good progress. When that fails, bisection can be applied to reduce the width of the interval and then the faster method tried again.

1.9 Roots of Polynomials

The **Fundamental Theorem of Algebra** states that a polynomial of degree n has exactly n roots counting multiplicities. Note that

1. The theorem is true for polynomials with both real and complex coefficients.
2. A polynomial with real coefficients may have complex roots; if so they come in complex conjugate pairs.
3. If the polynomial $p(x)$ has a root α then $(x-\alpha)$ is a factor of $p(x)$.
4. If the polynomial $p(x)$ has a root α of multiplicity m then $(x-\alpha)^m$ is a factor of $p(x)$.
5. A polynomial

   $$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

   can be factorized

   $$p(x) = a_n(x-\alpha_1)(x-\alpha_2)\ldots(x-\alpha_n)$$

   where $\alpha_1, \alpha_2, \ldots, \alpha_n$ are the (not necessarily distinct) roots of $p(x)$.

The Octave function `roots` can be used to find the roots of a polynomial. To find the roots of the polynomial

$$p(x) = x^5 - 3x + 5$$

we proceed as follows (remember that a polynomial is represented by its vector of coefficients with the highest degree term first):

```
octave:> p = [1 0 0 0 -3 5];
octave:> rp = roots(p)
rp =
  -1.57619 + 0.00000i
  -0.27365 + 1.46238i
  -0.27365 - 1.46238i
   1.06175 + 0.55302i
   1.06175 - 0.55302i
```

Polynomials and Eigenvalues

Recall that the eigenvalues of a (square) matrix $\mathbf{A}$ are the roots of its **characteristic polynomial**

$$p(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I})$$

To find the eigenvalues of $\mathbf{A}$ we could compute its characteristic polynomial and then use a nonlinear equation solver like `fzero` to find its roots. In practice however it is usual to proceed in the other direction – the roots of a polynomial are computed using eigenvalues.

The **companion matrix** of a polynomial

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

is the matrix

$$C(p) = \begin{bmatrix} -a_{n-1}/a_n & -a_{n-2}/a_n & \dots & -a_1/a_n & -a_0/a_n \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix} \tag{4}$$

The characteristic polynomial of the companion matrix $C(p)$ is $p(x)/a_n$ and so the eigenvalues of the companion matrix are exactly the roots of the polynomial $p(x)$.

Here's how the calculations go in Octave. We use `poly` to find the characteristic polynomial of matrix and `compan` to find the companion matrix of a polynomial. We start with a polynomial

```
octave:> p = [1 0 0 0 -3 5];
octave:> roots(p)
ans =
  -1.57619 + 0.00000i
  -0.27365 + 1.46238i
  -0.27365 - 1.46238i
   1.06175 + 0.55302i
   1.06175 - 0.55302i

octave:> cp = compan(p)
```

```
cp =
  -0  -0  -0   3  -5
   1   0   0   0   0
   0   1   0   0   0
   0   0   1   0   0
   0   0   0   1   0

octave:> eig(cp)
ans =
  -1.57619 + 0.00000i
  -0.27365 + 1.46238i
  -0.27365 - 1.46238i
   1.06175 + 0.55302i
   1.06175 - 0.55302i

octave:> poly(cp)
ans =
   1.00000   0.00000   0.00000   0.00000  -3.00000   5.00000
```

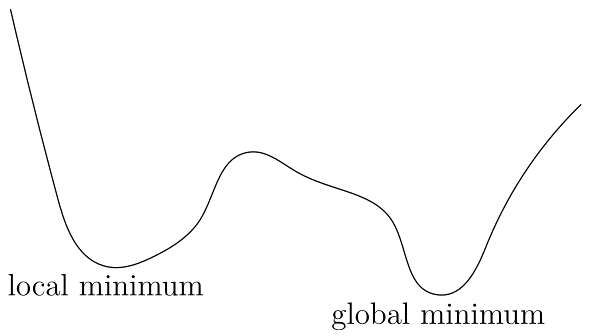
2 Optimization

Optimization problems are those concerned with finding a maximum or minimum of some function. We will only consider minimization problems with the understanding that maximizing $f(x)$ is the same as minimizing $-f(x)$.

The problem of finding a maximum or minimum of $f(x)$ is closely related, at least for differentiable functions, to solving $f'(x) = 0$. It is often the case that we do not know the derivative, and even when we do have a formula for the derivative it can be a more complicated, and therefore more expensive to evaluate, than the original function. Therefore, from a computational point of view, it is important to consider the problem of optimization separately from the problem of solving nonlinear equations.

2.1 Local and Global Minima

In optimization problems it is important to distinguish between local and global minima. A point x^* is a **global minimum** of $f(x)$ if $f(x^*) \leq f(x)$ for *all* points x. A point x^* is a **local minimum** of $f(x)$ if $f(x^*) \leq f(x)$ for all points x *near* x^*.



If you are at a local minimum x^* it is usually impossible to tell, at least by looking at points near x^*, whether or not it is a global minimum. For this reason most algorithms for minimization search for local minima only. For many numerical problems determining a global minimum requires finding all the local minima, and then taking the minimum amongst these.

2.2 Attainable Accuracy

In solving nonlinear equations we saw that the attainable accuracy was limited by rounding error in function evaluation. Analysis of rounding error showed that the error in the solution is

$$\Delta x \approx \frac{\varepsilon_{\text{mach}}}{|f'(x)|}$$

A similar analysis for optimization problems shows that the attainable accuracy is

$$\Delta x \approx \sqrt{\frac{2\varepsilon_{\text{mach}}}{|f''(x)|}}$$

There are two important points to make about this formula:

1. Because of the $\sqrt{\varepsilon_{\text{mach}}}$ term, optimization problems are harder to solve than nonlinear equations. For IEEE double precision this gives a typical accuracy of order 10^{-8} rather than 10^{-16} for nonlinear equations.
2. The $1/|f''(x)|$ term shows that the attainable accuracy is worst when the function $f(x)$ is flat around the minimum. Recall that $f'(x) = 0$ for a local minimum, so it is the second derivative which determines how flat the function is. The fact that minimum of a flat function is hard to determine should be intuitively clear.

The analysis above was based on the assumption that only function values were used in finding the minimum and that the error is due to rounding error in function evaluation. This won't apply if, instead of searching for the minimum directly, we attempt to find a local minimum by solving the nonlinear equation $f'(x) = 0$. In this case the error for solving nonlinear equations applies

$$\Delta x \approx \frac{\varepsilon_{\text{mach}}}{|f''(x)|}$$

This is usually much smaller than

$$\Delta x \approx \sqrt{\frac{2\varepsilon_{\text{mach}}}{|f''(x)|}}$$

However in the really troublesome cases where $f''(x)$ is close to zero it can turn out be larger.

2.3 Bracketing a Minimum

The key to the bisection method for solving a nonlinear equation $f(x) = 0$ was the observation that if we have two points a and b with $a < b$ and such that $f(a)$ and $f(b)$ have opposite signs, then the interval (a, b) is sure to contain a root of $f(x)$. The points a and b are then said to **bracket** the root.

The analogous idea for bracketing a minimum requires three points. Suppose we have three points a, b and c with $a < b < c$. If $f(b) < f(a)$ and $f(b) < f(c)$ the interval (a, c) is sure to contain a minimum of $f(x)$, and we say that the points (a, b, c) **bracket the minimum**.

If we now choose another point d in the interval (a, c) we can reduce the size of the interval bracketing the minimum. Suppose, for example, we

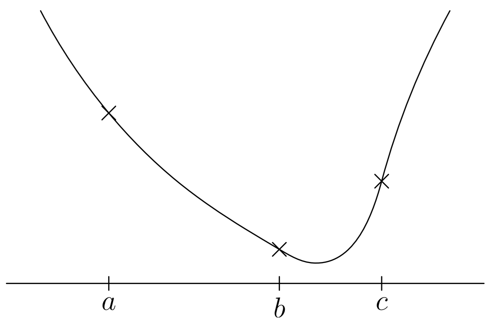
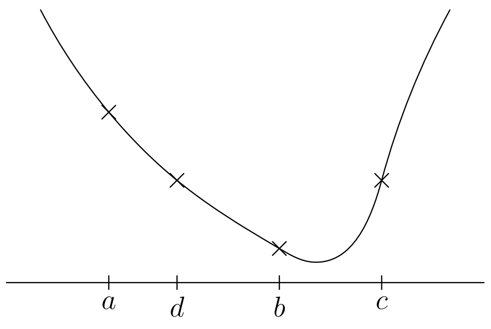


Figure 8: Bracketing a Minimum

choose d between a and b. There are two possibilities, either $f(d) < f(b)$ and (a, d, b) brackets the minimum, or $f(d) > f(b)$ and (d, b, c) bracket the minimum.

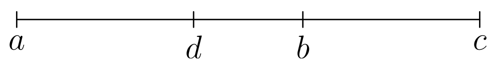


In the diagram above, the minimum is now bracketed by (d, b, c).

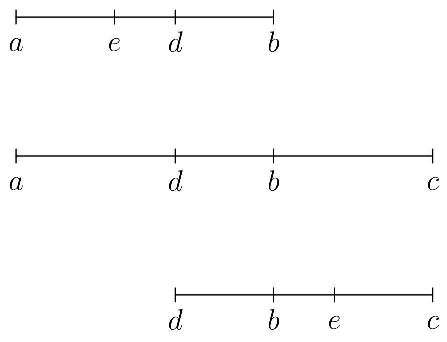
2.4 Golden Section Search

Once we have bracketed a minimum we can, by introducing new points and comparing function values, systematically reduce the size of the interval bracketing the minimum. We want to choose the new points in such a way that the relative positions of the points are the same at each iteration, thus allowing the length of interval bracketing the minimum to be reduced by a constant factor at each iteration.

For the bisection algorithm, this was easily achieved by taking the new point as the midpoint of the interval bracketing the root, thus reducing the size of the interval bracketing the root by a factor of 2 at each iteration. For minimization things a bit more complicated. Suppose we start with an interval (a, b, c) bracketing the minimum and we introduce a new point d:



Now depending on the value of $f(d)$ one of the intervals (a, d, b) or (d, b, c) will now bracket the minimum. In the first case we will introduce a new point e between a and d, and in the second case the new point e will be between b and c as below:



Here is the algebra of the situation:

First we require the size of the new interval is the same in either case. Thus

$$b - a = c - d$$

Now let

$$b - a = \tau(c - a) \quad \text{and} \quad c - d = \tau(c - a).$$

When we move to a new subinterval we require the same ratios so that

$$d - a = \tau(b - a) \quad \text{and} \quad c - b = \tau(c - d)$$

which gives us

$$d - a = \tau^2(c - a) \quad \text{and} \quad c - b = \tau^2(c - a).$$

The first pair of equations can be written

$$\begin{aligned} b &= (1 - \tau)a + \tau c \\ d &= (1 - \tau)c + \tau a \end{aligned}$$

while the second pair can be written

$$\begin{aligned} b &= \tau^2 a + (1 - \tau^2)c \\ d &= \tau^2 c + (1 - \tau^2)a \end{aligned}$$

Consistency then requires that

$$\tau^2 = 1 - \tau$$

so that

$$\tau = \frac{\sqrt{5} - 1}{2} \approx 0.618$$

Thus, if the points b and d are chosen at relative positions τ and $1 - \tau \approx 0.382$ in the interval (a, c), then no matter which subinterval is chosen its length will be τ times the length of the original interval. Further the interior point which is retained will be at a relative position of either τ or $1 - \tau$ in the new interval and the process can repeated. This choice of points is called **golden section search** after the well known 'golden ration' $(1 + \sqrt{5})/2 \approx 1.618$.

2.4.1 Unimodal Functions

We can relax the bracketing requirements on a minimum of a function $f(x)$ if we know that that $f(x)$ is unimodal. A function $f(x)$ is **unimodal** on an interval $[a, b]$ if it has exactly one local minimum x^* on the interval and is strictly decreasing for $x < x^*$ and strictly increasing for $x > x^*$.

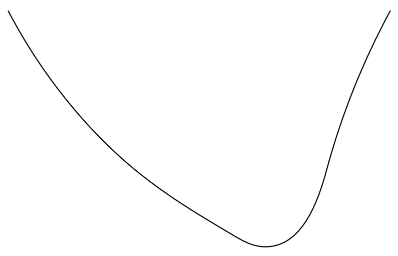
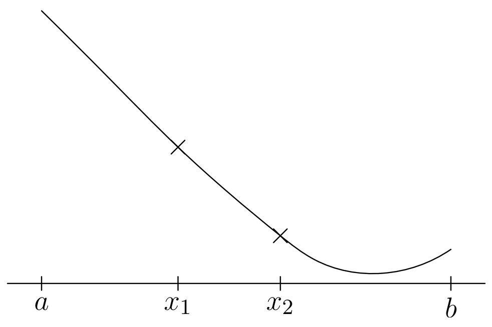


Figure 9: A unimodal function

Let $f(x)$ be unimodal on the interval $[a, b]$ and choose two points x_1 and x_2 in the interval (a, b) with $x_1 < x_2$ If $f(x_1) < f(f_2)$ then the minimum must lie in the interval $[a, x_2]$, While if $f(x_1) > f(x_2)$ then the minimum must lie in the interval $[x_1, b]$.



Note that in the diagram above we know that there is a minimum in the interval $[x_1, b]$ although the points (x_1, x_2, b) would not bracket the minimum without the assumption that $f(x)$ is unimodal on the interval $[a, b]$.

2.4.2 Implementing Golden Section Search

The assumption of unimodality makes golden section search a little easier to implement, since we do not have to worry about choosing points that bracket the minimum in the stricter sense. We start with an interval $[a_0, b_0]$ and choose points x_1 and x_2 in the interval at the relative positions τ and $1-\tau$. We then compare the values of $f(x_1)$ and $f(x_2)$, move to the appropriate subinterval and repeat the process. Because of the way golden section section search works, we need only compute one new point at each iteration. The algorithm is a bit more complicated than bisection since we need to keep track of function values rather than just their signs.

```
function [a,b] = goldsec(f, a0, b0, tol)
  tau = (sqrt(5) - 1)/2;
  a = a0;
  b = b0;
  x1 = a + (1 - tau)*(b-a);
  x2 = a + tau*(b-a);
  f1 = f(x1);
  f2 = f(x2);
  while (abs(b-a) > tol)
    if (f1 > f2)
       a = x1;
       x1 = x2;
       x2 = a + tau*(b-a);
       f1 = f2;
       f2 = f(x2);
    else
       b = x2;
       x2 = x1;
       x1 = a + (1 - tau)*(b-a);
       f2 = f1;
       f1 = f(x1);
    end
  end
endfunction
```

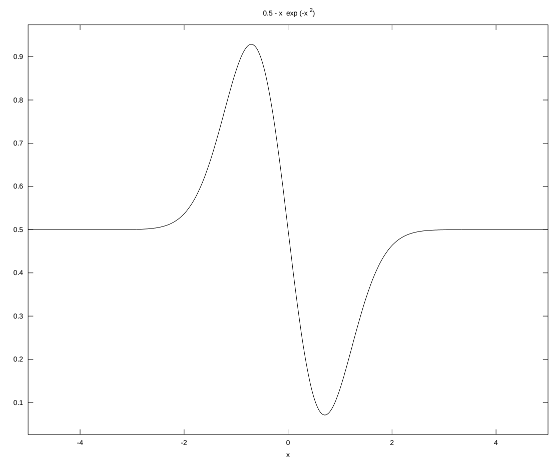
Example

For our example we will use the function

$$f(x) = 0.5 - xe^{-x^2}$$

As with solving nonlinear equations, the first thing to do is graph the function to determine the number of local minima and their approximate location.

```
octave:> f = @(x) 0.5 - x.*exp(-x.^2);
octave:> ezplot(f, [-5,5])
```



The graph shows that the function has one local minimum at $x \approx 0.7$ and that the function is unimodal over the interval $[0, 2]$.

Now we can determine the minimum using golden section search:

```
octave:> [a,b] = goldsec(f, 0, 2, 1e-8)
a =  0.707106776307258
b =  0.707106785047518
```

The remarks we made about the bisection algorithm for solving nonlinear equations also apply to golden section search:

1. Since the interval containing the minimum is reduces by a factor of $\tau \approx 0.618$ at each iteration, the algorithm is linearly convergent with constant τ (as opposed to 0.5 for bisection).

2. Golden section search always converges to a local minimum provided the function is unimodal, or more generally if we can bracket the minimum.

2.5 Newton's Method

As mentioned earlier, we can find local minima of a function $f(x)$ by solving the nonlinear equation $f'(x) = 0$. This requires that we know the derivative of the function. If we use Newton's to solve $f'(x) = 0$ then we need the second derivative as well and Newton's method becomes

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

The remarks we made Newton's method for solving nonlinear equations also apply to optimization:

1. It is usually quadratically convergent, except when $f''(x) = 0$ at the minimum, in which case it is linearly convergent.
2. We need an initial approximation close to desired minimum or the method may diverge or converge to a local maximum or point of inflection.

Example

The derivatives of the function

$$f(x) = 0.5 - xe^{-x^2}$$

are

$$f'(x) = (2x^2 - 1)e^{-x^2}$$

and

$$f''(x) = 2x(3 - 2x^2)e^{-x^2}$$

We use the function `newton` for nonlinear equations:

```
octave:> df = @(x) (2*x.^2-1).*exp(-x.^2);
octave:> d2f = @(x) 2*x.*(3 - 2*x.^2).*exp(-x.^2);
octave:> xn = newton(1.0, df, d2f, 7)
xn =
 Columns 1 through 4:
   1.000000000000000   0.500000000000000   0.700000000000000   0.707072135785007
 Columns 5 through 8:
   0.707106780337929   0.707106781186548   0.707106781186547   0.707106781186548
```

The minimum occurs a $x^* = 1/\sqrt{2}$ so the error is

```
octave:> err = xn - 1/sqrt(2)
err =
 Columns 1 through 4:
   0.292893218813453  -0.207106781186547  -0.007106781186548  -0.000034645401540
 Columns 5 through 8:
  -0.000000000848618   0.000000000000000   0.000000000000000   0.000000000000000
```

2.6 Successive Parabolic Interpolation

Suppose we have three points (a, b, c) which bracket a minimum of $f(x)$. Interpolate the function values at these points by a quadratic polynomial and let

$$x = b = \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)] - (b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)] - (b-c)[f(b)-f(a)]}$$

be the minimum of this quadratic polynomial. Depending on the position of x and the value of $f(x)$ we get a new bracket for the minimum:

1. If $x > b$ then
   (a) If $f(x) > f(b)$ then (a, b, x) brackets the minimum.
   (b) If $f(x) < f(b)$ then (b, x, c) brackets the minimum.
2. If $x < b$ then
   (a) If $f(x) > f(b)$ then (x, b, c) brackets the minimum.
   (b) If $f(x) < f(b)$ then (a, x, b) brackets the minimum.

Iterating this procedure is called the method of **successive parabolic interpolation**.

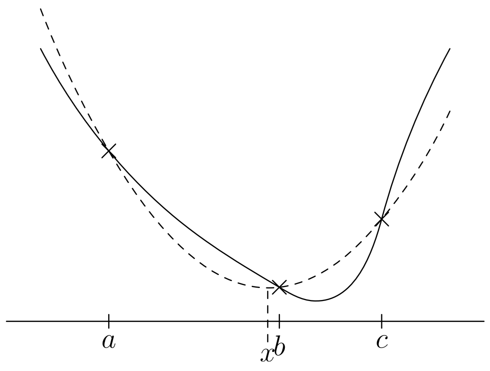


Figure 10: Parabolic Interpolation

In the diagram above $x < b$ and $f(x) > f(b)$ so (x, b, c) is the new bracket for the minimum.

The algorithm for successive parabolic interpolation has been formulated in terms of brackets for the minimum and in this form it always converges. More generally, the algorithm can be applied to any three points, but in this case it is not guaranteed to converge unless started close enough to the minimum. One possible cause of failure is when the interpolating quadratic has a maximum rather than a minimum. When successive parabolic interpolation converges it has order of convergence ≈ 1.324.

Implementation

Here is the Octave implementation of successive parabolic interpolation. The function `parab` takes the function f and a bracket `a0`, `b0`, `c0` for the minimum and performs n iterations, returning the successive minimum points of the interpolating parabolas.

```
function xx = parab(f, a0, b0, c0, n)
  xx = zeros(1,n);
  a  = a0;
  b  = b0;
  c  = c0;
  fa = f(a);
  fb = f(b);
  fc = f(c);
  for i = 1:n
    x = b - ((fb-fc)*(b-a)^2 - (fb-fa)*(b-c)^2) / ...
            (2*((fb-fc)*(b-a) - (fb-fa)*(b-c)));
    fx = f(x);
    xx(i) = x;
    if (x > b)
      if (fx > fb)
        c  = x;
        fc = fx;
      else
        a  = b;
        fa = fb;
        b  = x;
        fb = fx;
      end
    else
      if (fx > fb)
        a  = x;
        fa = fx;
      else
        c  = b;
        fc = fb;
        b  = x;
        fb = fx;
      end
    end
  end
endfunction
```

For our example problem we will use $(0, 1, 2)$ as the initial bracket for the minimum.

```
octave:> xp = parab(f,0,1,2,10)
xp =
 Columns 1 through 4:
   1.026197848245628   0.753375012260478   0.727937917951594   0.707906570615742
 Columns 5 through 8:
   0.707212253880617   0.707106954387123   0.707106783812903   0.707106780889157
 Columns 9 and 10:
   0.707106792021628   0.707106785041948
```

The error is

```
octave:> err = xp - 1/sqrt(2)
err =
 Columns 1 through 6:
   3.1909e-01   4.6268e-02   2.0831e-02   7.9979e-04   1.0547e-04   1.7320e-07
 Columns 7 through 10:
   2.6264e-09  -2.9739e-10   1.0835e-08   3.8554e-09
```

There are a couple of things to note:

1. The convergence is clearly faster than linear.
2. The limits on the attainable accuracy for optimization problems, typically of the order of $\sqrt{\varepsilon_{\text{mach}}}$, but a bit better than that here, are apparent.

2.7 Minimization in Octave

The Octave function for finding the minimum of a function is `fminbnd`. It employs a combination of successive parabolic interpolation and golden section search. The strategy is the same as that for `fzero`; when parabolic interpolation fails to make progress golden section search is used to narrow the interval bracketing the minimum.

Using `fminbnd` is straightforward.

```
octave:> xm = fminbnd(f,0,2)
xm =  0.707106784481841
octave:> xm - 1./sqrt(2)
ans =  3.29529348253033e-09
```