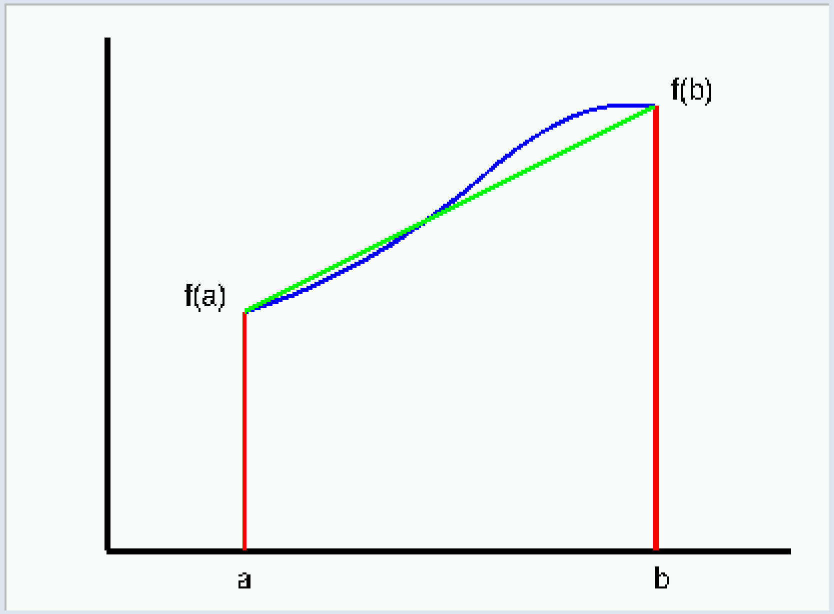
Introduction to C++

Numerical Integration Methods

The Trapezoidal Rule

If one has an arbitrary function f(x) to be integrated over the region [a,b] the simplest estimator that one can use is a linear approximation of f(x) over the integration region:



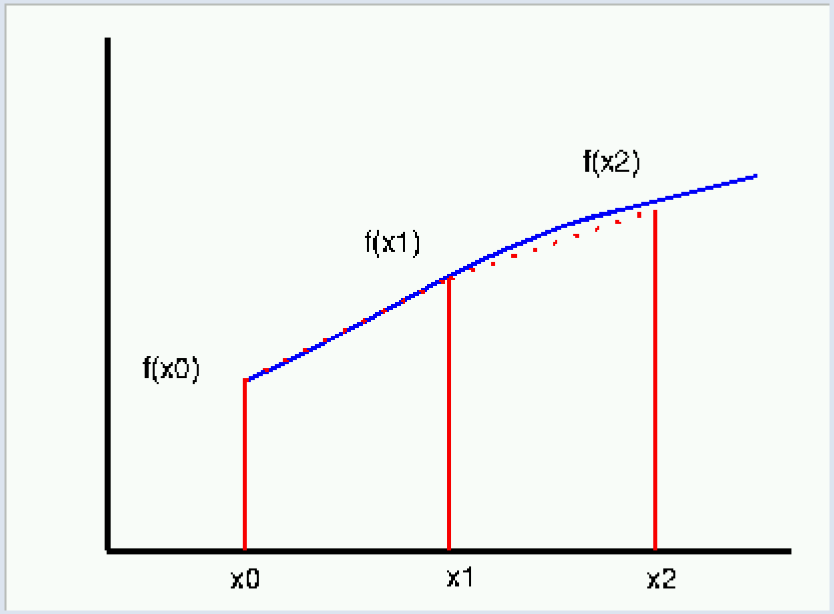
Numerical Integration Methods

Evaluating the area of the resulting trapezoidal figure, which is clearly half of that of a rectangle with sides h = b-a and f(a)+f(b) leads to the trapezoidal rule:

$$\int_a^b f(x)dx = \frac{h}{2}(f(a) + f(b))$$

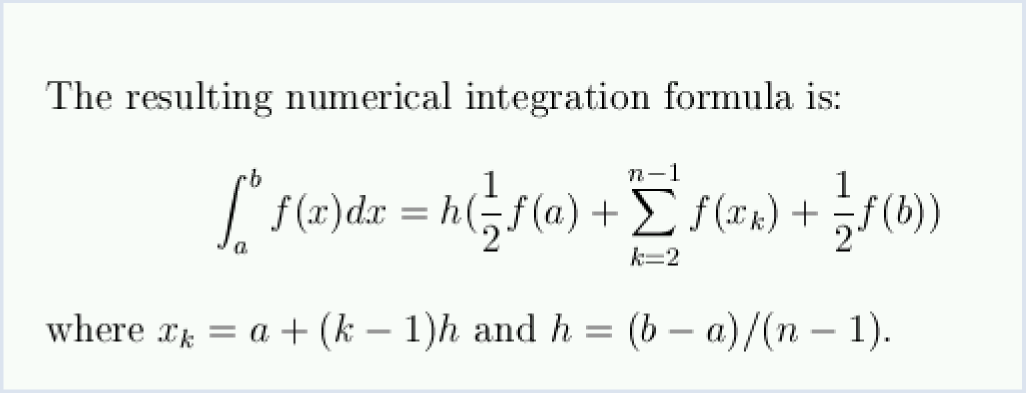
Numerical Integration Methods

The trapezoidal rule can be extended by subdividing the interval into subintervals and summing the individual contributions.



Numerical Integration Methods

The resulting extended trapezoidal rule takes the form:



The resulting numerical integration formula is:

$$\int_a^b f(x)dx = h(\frac{1}{2}f(a) + \sum_{k=2}^{n-1} f(x_k) + \frac{1}{2}f(b))$$

where $x_k = a + (k-1)h$ and $h = (b-a)/(n-1)$.

Notice that , the number of subintervals (n-1) , is arbitrary, so one can increase the precision of the approximation by simply increasing the value of n.

Numerical Integration Methods

The implementation of the extended trapezoidal rule as a C++ function is fairly straightforward:

```
double trap(double a, double b, int n)
{
    double h = (b-a)/(n-1);
//  Evaluate endpoints
    double value = 0.5*( f(a)+ f(b));
//  Now the midpoints
    for(int k=2; k < n; k++){
        value+=f( a + h*(k-1) );
    }
    value*=h;
    return value;
}
```

Here it is assumed that the (mathematical) function that we wish to integrated is implemented as a C++ function called double f(double x).

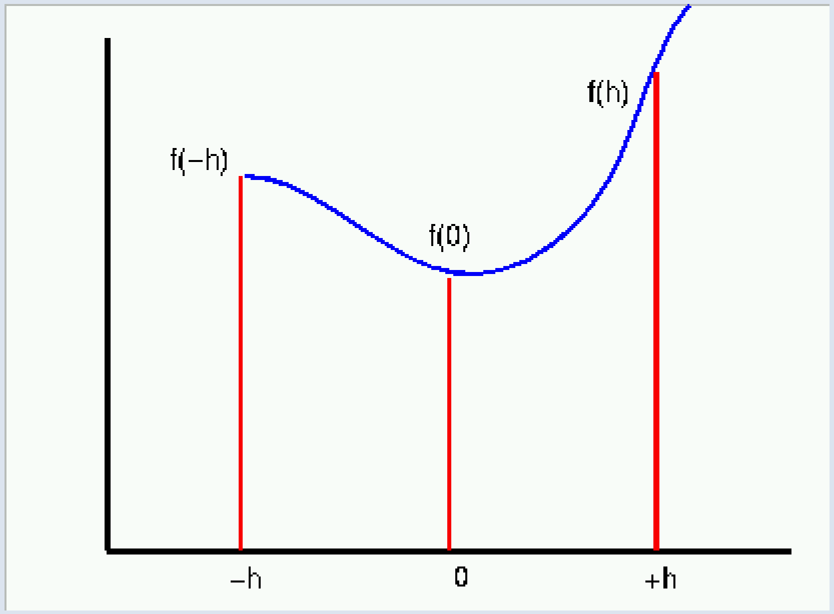
The source code for this function numint.cpp and its associated header file numint.h are available.

Numerical Integration Methods

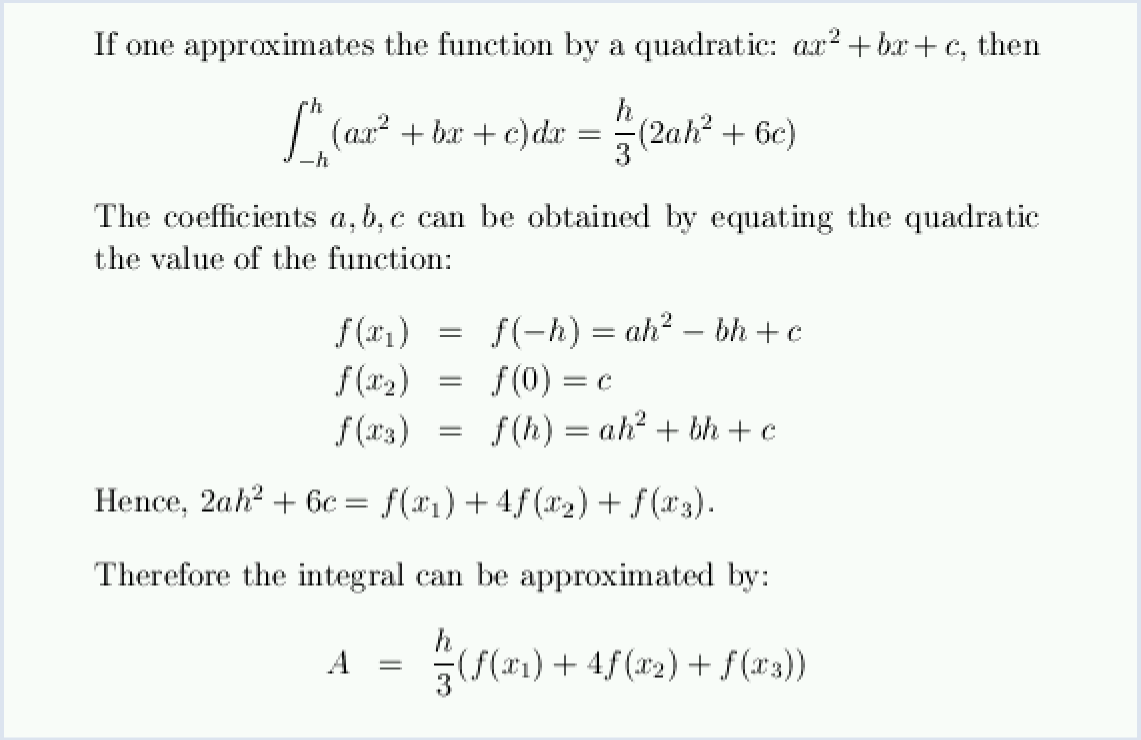
Simpson's Rule

The idea behind Simpson's rule is that we use a parabolic approximation (rather than the linear approximation used in the trapezoidal method) to the function over the integration region.

Consider a function f(x) integrated over the region [-h,h]



Numerical Integration Methods



If one approximates the function by a quadratic: $ax^2 + bx + c$, then

$$\int_{-h}^{h} (ax^2 + bx + c)dx = \frac{h}{3}(2ah^2 + 6c)$$

The coefficients a, b, c can be obtained by equating the quadratic the value of the function:

$$\begin{aligned} f(x_1) &= f(-h) = ah^2 - bh + c \\ f(x_2) &= f(0) = c \\ f(x_3) &= f(h) = ah^2 + bh + c \end{aligned}$$

Hence, $2ah^2 + 6c = f(x_1) + 4f(x_2) + f(x_3)$.

Therefore the integral can be approximated by:

$$A = \frac{h}{3}(f(x_1) + 4f(x_2) + f(x_3))$$

Numerical Integration Methods

By subdividing an integral range into a series of such triplets of points and summing the approximation to the sub-integrals one obtains Simpson's Rule:

$$\int_a^b f(x)dx = \frac{h}{3}(f(x_1) + 4f(x_2) + 2f(x_3) + 4f(x_4) + 2f(x_5) + \ldots + 4f(x_{n-1}) + f(x_n))$$

where n is an odd number, $x_1 = a$, $x_n = b$ and $x_k = a + (k-1)h$,
$h = (b-a)/(n-1)$.

In The Simpson's Rule n must be odd

The Simpson's Rule is fairly readily implemented in a similar way to the extended trapezoidal rule.

Finding roots of functions

Finding Roots of Functions

Consider a function such as:

f(x) = x3 - 2 x + 1

There are several numerical methods for finding the roots, xi, where, f(x)=0.

Finding roots of functions

Bisection Method

Let y1 = f(x1) and y2 = f(x2).

If y1.y2 < 0 , then there must be (at least one) zero of the function in the region [x1,x2].

The midpoint of the interval is:

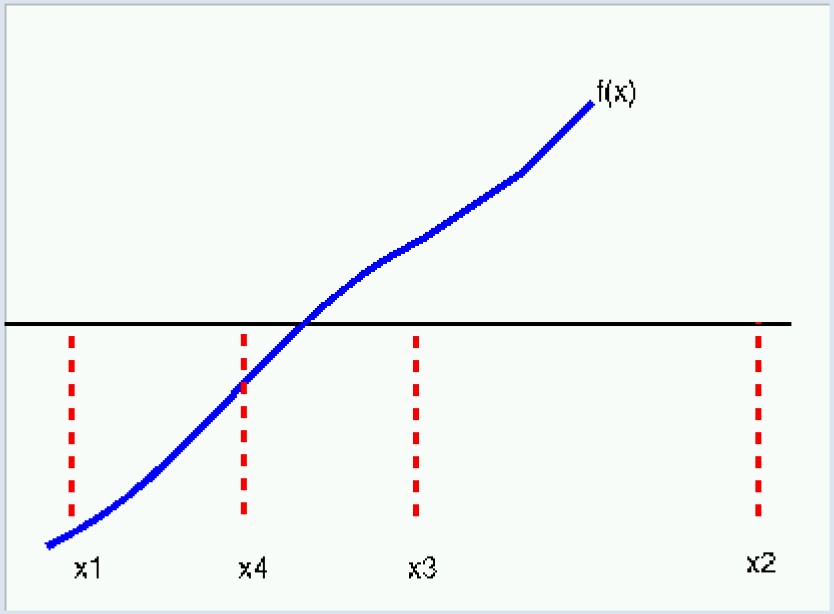
x3 = 0.5   *   (x1 + x2)

And y3 = f(x3).

Then either y1.y3 < 0 or y2.y3 < 0 . In the first case the root lies in the region [x1,x3] and in the second it lies in the region [x3,x2].

Finding roots of functions

Once we have restricted the interval we can iterate by repeating the process. The strategy is to keep bisecting the interval that contains the zero.



The difficulty with the bisection method is in finding a initial region [x1,x2] that contains one root (or an odd number of roots).

Finding roots of functions

If one writes a C++ function:

```
    double f(double x);
```

to evaluate the mathematical function of interest, then a C++ implementation of the bisection method could look like:

```
int Bisect(double x1,double x2, double precision,double& root)
{
    if( f(x1)*f(x2) > 0 ){
//      No apparent roots in this region
        root = 0;
        return 0;
    } else {
// Bisect the interval
        double x = 0.5*(x1+x2);
//      Test for convergence
        if( fabs(f(x)) < precision){
            root = x;
            return 1;
        } else {
//   Use the Bisect method on one of the subintervals
            if( f(x1)*f(x) < 0 ){
                return Bisect(x1,x,precision,root);
            } else {
                return Bisect(x,x2,precision,root);
            }
        }
    }
}
```

Finding roots of functions

Notice that in this case an initial range [x1,x2] needs to be input, and a poor choice can result in failure indicated by a return value of 0. Successive bisections are carried out by recursively calling the Bisect function itself after deciding in which of the two sub-ranges the root lies.

Finding roots of functions

The Secant Method

Let y1 = f(x1) and y2 = f(x2) .

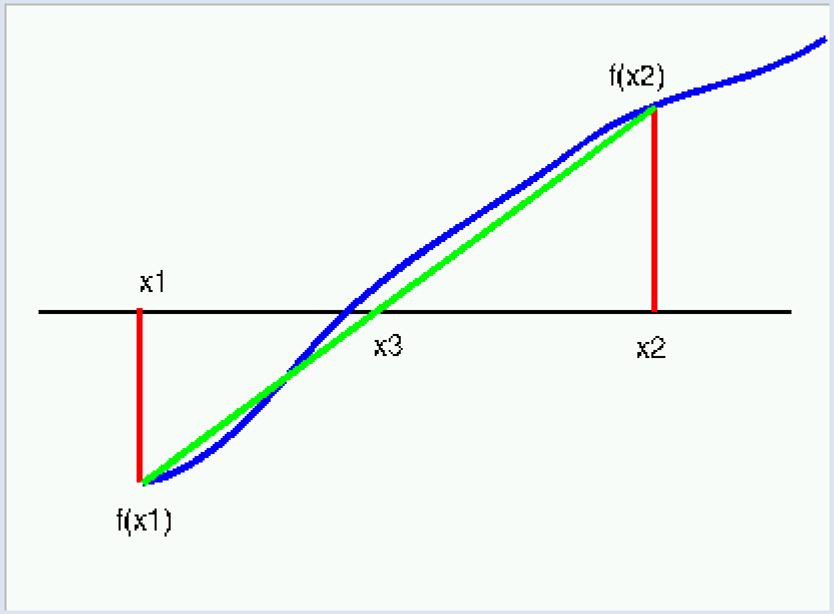
Construct a straight line through the points (x1,y1) and (x2,y2).
From similar triangles we have:

(x3 - x1) / (0 - y1) = (x2 - x1) / (y2 - y1)

Its intersection with the x-axis is therefore:

x3 = x1 - y1 (x2 - x1) / (y2 - y1)

Finding roots of functions



If one sets:

x1 = x2

x2 = x3

Then after several iterations we will, in most cases, have found the zero of the function.

Finding roots of functions

The secant method can be implemented as follows.

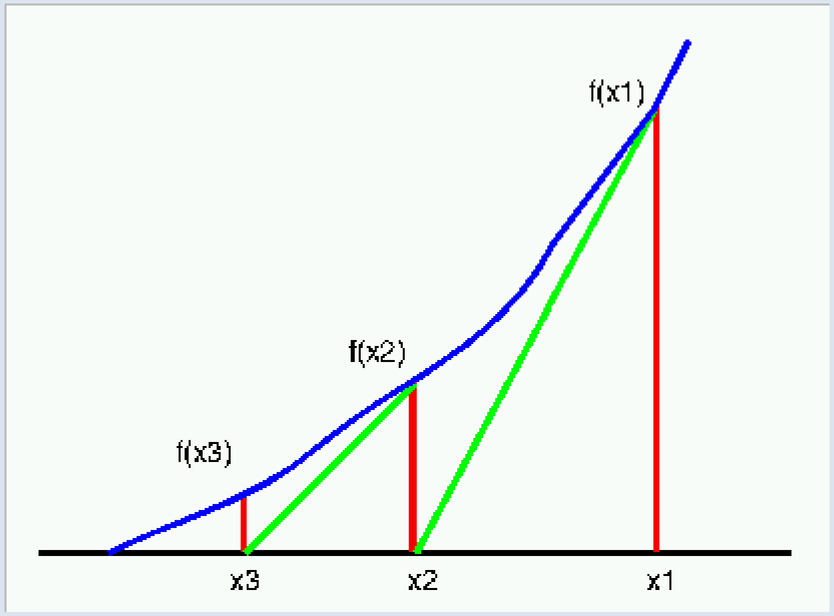
```
int Secant(double x1, double x2,double precision,double& root)
{
    if(x1==x2 || f(x2)==f(x1)){
        root=0;
        return 0;
    }
    double x = x1 - f(x1)*(x2-x1)/(f(x2)-f(x1));
    if( fabs(f(x)) < precision){
        root = x;
        return 1;
    } else {
        return Secant(x2,x,precision,root);
    }
}
```

Again recursion is used to produce compact code

Finding roots of functions

Newton-Raphson Method

This method is applicable whenever the derivative of the function is known. In this case the trial solution consists of a single value x1



Finding roots of functions

since   f '(x1) = Delta_y/Delta_x = (f(x1) - 0) / (x1 - x2)

where f '(x) indicates the first derivative.

The intercept of the tangent to the curve at (x1,y1) with the x axis is at:

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

This will usually be a better approximation to the root than x1. We continue the iteration

x3 = x2 - f(x2)   / f '(x2)

x4 = x3 - f(x3)   / f '(x3)

One can test for convergence by testing the value of the function divided by the derivation at the current value of x,   f(x_n)   / f '(x_n)

Finding roots of functions

Finding roots of functions

If one writes a C++ function:

```
    double f(double x);
```

to evaluate the mathematical function of interest, then a C++ implementation of the bisection method could look like:

```
int Bisect(double x1,double x2, double precision,double& root)
{
    if( f(x1)*f(x2) > 0 ){
//      No apparent roots in this region
        root = 0;
        return 0;
    } else {
// Bisect the interval
        double x = 0.5*(x1+x2);
//      Test for convergence
        if( fabs(f(x)) < precision){
            root = x;
            return 1;
        } else {
//   Use the Bisect method on one of the subintervals
            if( f(x1)*f(x) < 0 ){
                return Bisect(x1,x,precision,root);
            } else {
                return Bisect(x,x2,precision,root);
            }
        }
    }
}
```

Finding roots of functions

Notice that in this case an initial range [x1,x2] needs to be input, and a poor choice can result in failure indicated by a return value of 0. Successive bisections are carried out by recursively calling the Bisect function itself after deciding in which of the two sub-ranges the root lies.

Finding roots of functions

The Secant Method

Let $y1 = f(x1)$ and $y2 = f(x2)$.

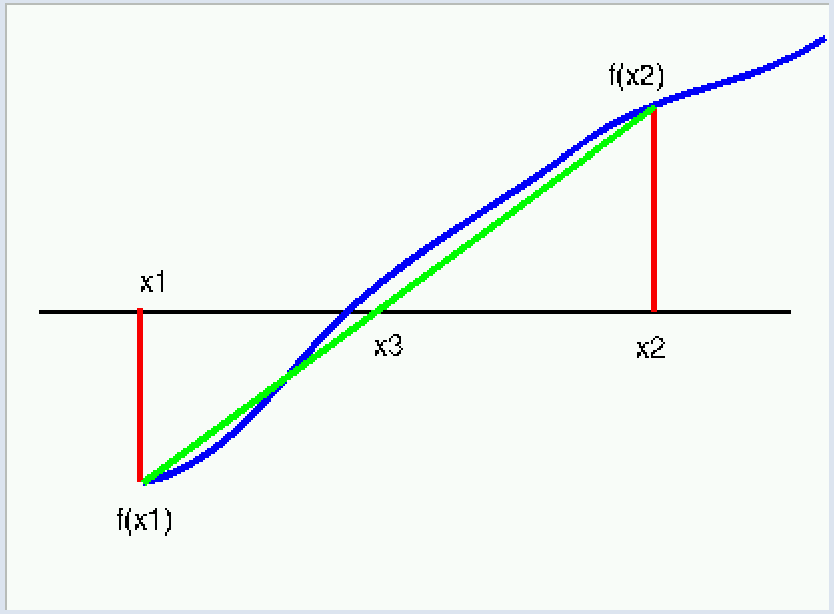
Construct a straight line through the points (x1,y1) and (x2,y2).
From similar triangles we have:

$$(x3 - x1) / (0 - y1) = (x2 - x1) / (y2 - y1)$$

Its intersection with the x-axis is therefore:

$$x3 = x1 - y1 (x2 - x1) / (y2 - y1)$$

Finding roots of functions



If one sets:

x1 = x2

x2 = x3

Then after several iterations we will, in most cases, have found the zero of the function.

Finding roots of functions

The secant method can be implemented as follows.

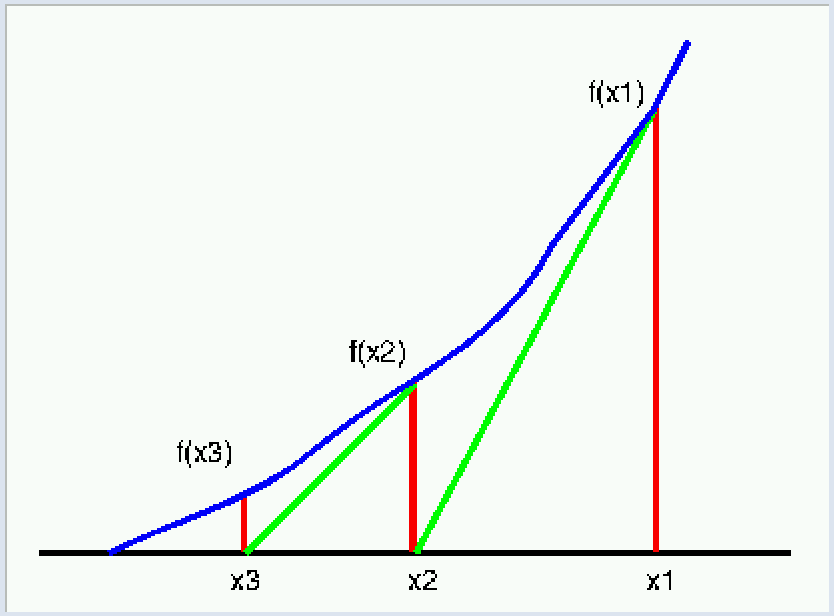
```
int Secant(double x1, double x2,double precision,double& root)
{
    if(x1==x2 || f(x2)==f(x1)){
        root=0;
        return 0;
    }
    double x = x1 - f(x1)*(x2-x1)/(f(x2)-f(x1));
    if( fabs(f(x)) < precision){
        root = x;
        return 1;
    } else {
        return Secant(x2,x,precision,root);
    }
}
```

Again recursion is used to produce compact code

Finding roots of functions

Newton-Raphson Method

This method is applicable whenever the derivative of the function is known. In this case the trial solution consists of a single value x1



Finding roots of functions

since   f '(x1) = Delta_y/Delta_x = (f(x1) - 0) / (x1 - x2)

where f '(x) indicates the first derivative.

The intercept of the tangent to the curve at (x1,y1) with the x axis is at:

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

This will usually be a better approximation to the root than x1. We continue the iteration

x3 = x2 - f(x2)   / f '(x2)

x4 = x3 - f(x3)   / f '(x3)

One can test for convergence by testing the value of the function divided by the derivation at the current value of x,   f(x_n)   / f '(x_n)

Finding roots of functions

A C++ implementation of the Newton-Raphson Method to find a root to a required precision could be:

```
int Newton(double x, double precision, double& root)
{
    double delta = f(x)/dfdx(x);
    while ( fabs(delta) > precision ){
        x-= delta;
        delta = f(x)/dfdx(x);
    }
    root=x;
    return 1;
}
```

Here it is assumed that both the function and its derivative

```
    double f(double x);
    double dfdx(double x);
```

Have been defined to calculate the value of the required function and its derivative respectively. N.B. The integer return value is there simply to indicate success.

All three of these functions are demonstrated in the example program roots.cpp

Function minimization

Function Minimization

Function Minimization Using the Newton-Raphson Method

Many physics problems involve the minimization of a function with respect to one (or more) variables. In the simple case of one variable the minimum of a function can usually be found by finding where the first derivative is zero. This is equivalent to finding roots of an equation, except now we are working with the first derivation function.

The Newton-Raphson iteration formula now becomes:

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

where f ' and f '' are the first and second derivatives respectively.

An Example of Function Minimization: Fitting data

As an example of function minimization, consider the problem of fitting a straight line that goes through the origin

y = a x , where a is an unknown parameter

through a set of data points. For example:

x	y	sigma
0.1	0.2	0.1
0.2	0.7	0.1
0.3	1.0	0.1
0.5	1.4	0.1

For each x value we have a measurement of y together with its associated error sigma.

If we want to fit data we normally want to minimize the Chi-squared function given by:

Function minimization

If we want to fit data we normally want to minimize the Chi-squared function given by:

$$\chi^2 = \sum_i \frac{\Delta y_i^2}{\sigma_i^2}$$

where Δy is the difference between the actual and fitted value of y

with respect to the fit parameters (in this case the gradient a). We are therefore trying to find:

$$\frac{d\chi^2}{da} = 0$$

Function minimization

This type of problem can be fairly readily solved using the Newton-Raphson iteration. To do this we need to find the first and second derivatives of the Chi-squared function in terms of the measurements are their errors:

$$\frac{d\chi^2}{da} = \sum_i \frac{2\Delta y}{\sigma_i^2}\frac{dy}{da} = \sum_i \frac{2x_i \Delta y}{\sigma_i^2}$$

$$\frac{d^2\chi^2}{da^2} = \sum_i \frac{2x_i}{\sigma_i^2}\frac{dy}{da} = \sum_i \frac{2x_i^2}{\sigma_i^2}$$

Using this technique a function which fits such a straight line to n data points and errors stored in arrays x, y and sigma can be written.

Function minimization

```
void nrfit(double* x, double* y, double* sig, int n,
           double& a, double& sig_a, double&  chi2)
{
        const double precision = 1e-6;
        double g1,G11;
        double yy,dy,s2,chi2old,dchi2;
        chi2 =DBL_MAX;
        dchi2=DBL_MAX;
        while( fabs(dchi2) > precision ){
            chi2old=chi2;
            chi2=0;
            g1=0;
            G11=0;
            for (int i=0; i < n; i++ ){
// calculate delta_y and chi2 for this point
            yy= a * x[i];
            dy= yy - y[i];
            chi2+=pow( dy/sig[i], 2);
            s2=1.0/(sig[i]*sig[i]);
// Contribution to the first and second derivatives
            g1 += 2*x[i]*dy*s2;
            G11+= 2*x[i]*x[i]*s2;
        }
// Calculate updated value for fit parameter a
        a-= ( g1/G11);
// Calculate change of chi2
        dchi2=chi2old-chi2;
    }
// Calculate error on parameter a
    sig_a=sqrt(2.0/G11);
}
```

Function minimization

Here the variables g1 and G11 represent the first and second derivatives wrt the fit parameter a. After each iteration the change in Chi-squared dchi2 is calculated and the fit is terminated if this is sufficiently small.

Once the fit is complete, the error (uncertainty) on the fitted parameter is calculated: sig_a=sqrt(2.0/G11).

The complete program can be found in the example linefit.cpp

Finding roots of functions