C:\WINDOWS\WORD\DOT\!5X8.DOT

Natural Systems

Numerical Methods

Note - This document was created with WinWord 1.1 and formated with Bitstream typefaces. If you are interested in acquiring the file in its native .DOC format, contact me via EMail

Michael Weber [71301,1221]

C:\WINDOWS\WORD\DOT\!5X8.DOT

TRoots

Equations posess both a right-hand and left-hand side. Subtracting the two sides leaves ¦*(x)* = 0 whose solution is desired. When there is only one independent variable, the problem is one-dimentional. We call this finding the root of a function. (i.e. the value of *x* where ¦*(x)* = 0.) In general, the number of roots an equation has is considered to be its order. This is a very common problem in engineering and research.

For complicated equations it is too time consuming, if not impossible, to solve an equation exactly for all of its roots. As an alternative, we can calculate an approximation itteratively to within a user defined error tolerance. This is true for equations with one or many dimensions. For one-dimensional equations, it is possible to bracket (read trap) a root between bracketing values, and then hunt it down like a rabit. Thus, a useful algorithm will improve the solution until some predetermined convergence criterion is satisfied. For our purposes, this criteria will be reducing the delta between X1 and X2 until it is smaller than Tolerance (more than this later). For smoothly varying functions, good algorithms will always converge, *provided* that the initial guess is good enough. It cannot be overemphasized, though, how heavily success depends on having a good first-guess for the solution.

Note: Try to get some idea of what your function looks like before trying to find its roots. If you need to mass-produce roots for many different functions, then you should at least know what some typical members of the ensemble look like. Next, you should always bracket a root that is, know that the function changes sign in an identified interval before trying to converge to the roots value.

Although there are many methods to calculate the roots of an equation, some good and some not, the three most popular are included: the bisection method, the Van Wijnaarden-Dekker-Brent method (called the Brent method for brevity), and Newton-Raphson method (called Newton's method for brevity).

- € The Bisection method is the most foolproof and easiest to understand but does not converge very fast.
- € Brent's method is the method of choice to find a bracketed root of a general one-dimensional function, when you cannot easily compute the functions derivative. The Brent method is computationally quite complex, but is considered by experts to be the fastest non-derivative based method for solving for the root of a function.
- € When you can compute the functions derivative, Newton's method is recommended. Again, you must first bracket your root.

Bracketing

figure x.1

We usually say that a root is *bracketed* in the interval (*a, b*) if ¦(*a*) and ¦(*b*) have opposite signs. If the function is continuous, then at least one root must lie in that interval (the *intermediate value theorem).* If the function is discontinuous but bounded, then instead of a root there might be a step discontinuity which crosses zero (see *fig x.1*).

Bisection Method

Once we know that an interval contains a root, several classsical procedures are available to refine it. Proceeding with varying degrees of speed and sureness toward the answer. Unfortunately, the methods that are guarenteed to converge plod slowly along, while those that rush to the solution in the best cases can also rush rapidly to infinity without warning if measures are not taken to avoid such behavior.

The *Bisection method* is one which cannot fail. It is thus not to be sneered at as a method for otherwise badly behaved problems. The idea is simple. Over some interval the function is known to pass through zero because it changes sign. Evaluate the function at the interval's midpoint and examine its sign. Use the midpoint to replace whichever limit has the same sign. After each iteration the bounds containing the root decrease by a factor of two. If after *n* iterations the root is known to be within an interval of size e_n, then after the next iteration it will be bracketed within an interval of size

Thus we know in advance the number of interations required to achieve a given tolerance in the solution

where e_0 is the size of the initially bracketed interval and *e* is the desired ending tolerance.

Bisection *must* succeed. If the interval happens to contain two or more roots, bisection will find one of them. If the interval contains no roots and merely straddles a singularity, it will converge on the singularity.

It is important to keep in mind that computers use a fixed number of binary digits to represent floating point numbers. While your function might analytically pass through zero, it is possible that its computed value is never zero, for any floating point argument. You must decide what accuracy on the root is atainable: convergence to within 10^{-6} in absolute value is reasonable when the root lies near 1, but certainly unachieveable if the root lies near 10^{26}. You might think to specify convergence to a relative (fractional) criteria, but this becomes unworkable for roots near zero. The Bisection method will require you to specify an absolute tolerance, such that iterations continue until the interval becomes smaller than this tolerance in absolute units. Usually, you may wish to take the tolerance to be

where *e* is the machine precision and *x1* and *x2* are the initial brackets. When the root lies near zero you ought to consider carefully what reasonable tolerance means for your function.

Van Wijnaarden-Dekker-Brent Method

While other methods (namely secant and false position) may converge faster than bisection, there are ill behaved functions for which bisection converges more rapidly. These can be chopy, discontinuous functions, or even smooth functions if the second derivative changes sharply near the root. Bisection always halves the interval, while other methods can sometimes spend many cycles slowly pulling distant bounds closer to a root. Is there anything we can do to get the best of both worlds?

Yes. We can keep track of whether a supposedy superlinear method is actually converging the way it is supposed to and, if not, we can intersperse bisection steps so as to guatrentee *at least* linear convergence. This kind of super-strategy requires attention to detail, and also careful consideration of how roundoff errors can affect the guiding strategy. Also, we must be able to determine reliably when convergence has been achieved.

An excelent algorithm that pays close attention to these matters was developed in the 1960's by van Wijngaarden, Dekker, et. al. - and later improved by Richard P. Brent. For brevity, we refer to the final form of the algorithm as *Brent*'s *method*. The method is guarenteed (by Brent) to converge, so long as the function can be evaluated within the initial interval known to contain a root.

Brent's method combines root bracketing, bisection, and inverse quadratic interpolation to converge from the neighborhood of a zero crossing. While the false position and secant methods assume approximately linear behavior between two prior root estimates, inverse quadratic interpolation uses three prior points to fit an inverse quadratic function (*x* as a quadratic function of *y*) whose value at $y = 0$ is taken as the next extimate of the root *x*. Of course we must have contingency plans should the root fall outside of the brackets. Brent's method takes care of all that.

Thus, Brent's method combines the sureness of bisection with the speed of a high-order method when appropriate. We recommend it as the method of choice for general one-dimensional root finding where a function's values only (and not its derivative or functional form) are available.

Newton-Raphson Method

Perhaps the most celibrated of all one-dimensional root-finding routines is *Newton*'s *method* also called the *Newton-Raphson method*. This method is distingu3
ished from the methods of previous sections by the fact that it requires the evaluation of both the function ¦*(x)*, and the derivative ¦¢*(x)*, at arbitrary points *x*. The Newton-Raphson formula consists geometrically of extending the tangent line at a current point x_i, until it crosses zero, then setting the next guess x_{i+1} to the abscissa of that zero-crossing (see fig x.2). Algebraicly, the method derives from the familiar Taylor series expansion of a function in the neighborhood of a point

figure x.2

The Newton-Raphson method is not restricted to one dimension. The method readily generalizes to multiple dimensions. Why do we call Newton-Raphson powerful? The answer lies in its rate of convergence; the Newton-Raphson method converges *quadratically*. Near a root, the number of significant digits approximately *doubles* with each step. This makes the Newton-Raphson method the method of choice for any function whose derivative can be evaluated efficiently, and whose derivative is continuous and nonzero in the neighborhood of a root. Even where Newton-Raphson is not feasable for the early stages of convergence, it is very common to "polish up" a root with one or two steps of Newton-Raphson, which can multiply by two or four its number of significant digits!

figure x.3

Given its strengths, the Newton-Raphson method can also give grossly inacurate results. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function (*fig x.3*). If an iteration places a trial guess near such a local extreme, so that the first derivative nearly vanishes, then Newton-Raphson sends its solution off to limbo, with an equally small hope of recovery.

For an efficient realization of Newton-Raphson the user provides a routine which evaluates both ¦*(x)* and its first dirivative ¦¢*(x)* at the point *x*. The Newton method does not adjust bounds, and works only on local information at the point *x*. The bounds are only used to pick the midpoint as the first guess, and to reject the solution if it wanders outside of the bounds.

Constructing ¦*(x)* and ¦¢*(x)*

In order to calculate your roots, TRoots needs to know your ¦*(x)*. If you are planing to calculate via Newton's method, it will need to know your ¦¢*(x)*as well.

The way you will provide these equations to TRoots is by extending TRoots fx and fxPrime methods. Do this by declaring in your program as an object of type TRoots. Then define the virtual method fx (and optionally fxPrime) within TRoots as shown below:

```
PMyRoots = ^TMyRoots;
TMyRoots = Object(TRoots)
   Function fx(X:TFloatingPoint)      : TFloatingPoint; Virtual;
   Function fxPrime(X:TFloatingPoint) : TFloatingPoint; Virtual;
End; {Object}
```

Next you need to express the equation you wish to solve for. For instance:

```
Function TMyRoots.fx;
   Begin
      fx := (2 * x + 3) * (x - 3);
   End;
{EndMethod}
Function TMyRoots.fxPrime;
   Begin
      fxPrime := 4 * x - 3;
   End;
{EndMethod}
```

Now, when you instantate your object (of TMyRoots) the fx / fxPrime methods you created will be called upon to evaluate your equation. For instance:

```
Procedure XYZ;
   Var
      ARoot : PMyRoots;
      ...

   Begin
      ...
      ARoot := New(PMyRoots, Init(dX1, dX2, dTolerance, dIterMax));
      dResult := ARoot^.BrentRoots(dRootValue);
      ...
```

To learn more about how this is accomplished refer to Ch. 17 *Objects* in Borland's *Windows Promgrmmers Guide*.

Note: TRoots.fx and TRoots.fxPrime are defined as *abstract* methods - thus if you call them directly (e.g. extended fx / fxPrime incorectly) your program will terminate with an error code 211.

Ancestor

TNumMethods

Fields

X1 **X1 : TFloatingPoint;**

The initial lower bracket value

X2 **X2 : TFloatingPoint;**

The initial upper bracket value

Tolerance **Tolerance : TFloatingPoint;**

Converge on root until successive values of roots differ by less than Tolerance.

IterMax **IterMax : LongInt;**

Maximum number of itterations allowed to calculate root. If IterMax is reached before a root is found the function returns with a value of 0 and places the error code in ErrorCode.

Methods

Init **constructor Init(vX1, vX2, vTolerance : TFloatingPoint; vIterMax : LongInt);**

Constructs a new TRoots object with a bounding interval of vX1 to vX2, a tolerance of vTolerance, and a maximum number of iterations of vIterMax.

BrentRoots **function : TFloatingPoint;**

Calculates the root of the equation defined in fx via Brent's method.

BisectionRoots **function : TFloatingPoint;**

Calculates the root of the equation defined in fx via the Bisection method.

NewtonRoots **function : TFloatingPoint;**

Calculates the root of the equation defined in fx via Newton's Method. Don't forget to define fxPrime.

fx **function (X : TFloatingPoint) : TFloatingPoint; virtual;**

This is an *abstract* method and must be overriden as shown above in Constructing ¦*(x)* and ¦*¢(x)*.

fxPrime **function (X : TFloatingPoint) : TFloatingPoint; virtual;**

This is an *abstract* method and must be overriden as shown above in Constructing ¦*(x)* and ¦*¢(x)*.

Done **destructor Done; virtual;**

Disposes the TRoots object by calling TNumMethods.Done.

Err Codes

0
1
2
3