Numerical Methods

notes by G. Houseman
for EARS1160, ENVI2240, CCFD1160

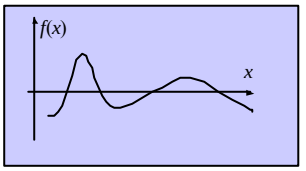
Lecture 1: Numerical Solution of Non-linear equations.

- **If $f(x) = 0$, what is x ?**
- **Well-behaved functions and Continuity**
- **Bracketing the Solution**
- **Bisection Method**
- **Iteration and Convergence**
- **The Bisection Algorithm**
- **Pitfalls and Problems**
- **Interpolation method**

Definition of the Problem

$f(x)$ is a general non-linear function of x. Solve the equation:

$f(x) = 0$



at which value(s) of x does the graph of the function cross the x-axis ? Maybe there is no value, or more than one.

Some functions can be analytically inverted. Many can not.

But there are standard methods that can be programmed into a computer to solve this problem, for just about all "well-behaved" functions. Note: what if $f(x) = g(x)$?

Well-behaved functions

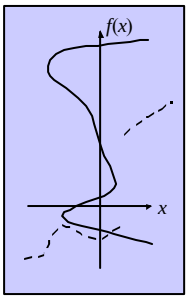
We won't be too rigorous about what this means, but ....

1. $f(x)$ should be single-valued: every x gives you one and only one value of $f(x)$. The solid line is not single-valued. So is e.g.

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

If the function is not single valued, restrict the range of f values, so that the function is single valued.

2. $f(x)$ should be continuous: the value of $f(x)$ doesn't change abruptly as x is increased or decreased. The dashed line is a single-valued function, but it is discontinuous.



Continuity

A continuous function can be defined mathematically by:

$$\lim_{e \to 0} \left[f(x+e) - f(x-e) \right] = 0$$

for all x. Example: is $f(x) = \sin(x)$ a continuous function ? What about $\tan(x)$? If not, where are the discontinuities ?

In general we might classify a well-behaved function as one that is defined everywhere, it is single-valued, continuous, and all of its derivatives are defined and continuous.

Discontinuities can cause problems for methods that use the slope (or gradient) of the function to extrapolate to the axis. Note also that the gradient or first derivative $f'(x)$ is undefined at a point of discontinuity.

The numerical methods are robust enough to work ok if some of these conditions are not met, but if they fail, look at the graph of the function that you are trying to solve. Should you expect to find a zero crossing for this function ?

Bracketing the solution

All numerical solution methods rely on the fact that $f(x)$ is defined over a domain $a < x < b$ within which the solution is sought. Moreover, we assume that the program can directly compute the value of $f(x)$ for all x in this domain.

If for two points x_1 and x_2, $f(x_1)f(x_2) < 0$, then either

$f(x_1) < 0$ and $f(x_2) > 0$, or vice versa.

Therefore computing $f(x_1)f(x_2)$ and testing its sign is a way of testing whether the points x_1 and x_2 bracket the solution we seek.

What if there are more than one zero crossings between x_1 and x_2 ? What if there are discontinuities ?

If we evaluate $f(a)f(b)$, does its value give us a reliable indication that a zero exists in $a \le x \le b$?

The Bisection Method

One way to select x_1 *and* x_2 is to plot up the graph of $f(x)$ and choose by visual inspection, two points that are on either side of the zero you want to find. No particular precision is required - so long as there is one and only one zero between x_1 *and* x_2.

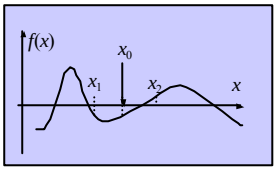
The bisection method relies on a repeated halving of the interval that brackets the zero.

1. define $x_0 = (x_1+x_2)/2$
2. if $f(x_0).f(x_1) > 0$,

   then $x_1 = x_0$

   else $x_2 = x_0$

Thus we have halved the length of the bracketing interval. We repeat these 2 steps until interval is sufficiently small.



Iteration

Iteration is a general word, that refers to a repetitive group of operations. In this instance, we repeat the interpolation process, halving the width of the bracketing range at each step, until its width is less than the required precision on the solution estimate.

In a FORTRAN program we may use a DO loop to define the operations of the interpolation algorithm:

Before the DO loop, we enter initial estimates of x_1 and x_2, and we provide a subroutine $F(x)$ which defines the function we wish to solve.

At each step of the iteration, we can examine $|x_1 - x_2|$ to see how closely we have now bracketed the solution. At any particular step, the value of x_0 is the current estimate of the solution and $|x_1 - x_2|$ is the error on that estimate.

Bisection Algorithm

Within the DO loop of length N, the algorithm will require something like the following steps (though some details are flexible):

- evaluate $f(x_1), f(x_2)$
- confirm that $f(x_1)f(x_2) < 0$; else exit with warning #1.
- bisect interval to estimate x_0, and evaluate $f(x_0)$
- if $f(x_0) = 0$, solution is found; exit loop and print solution
- if $f(x_1)f(x_0) > 0$, then $x_1 = x_0$; else $x_2 = x_0$
- evaluate whether $|x_1 - x_2| < e$
- if convergence: exit loop and print solution
- if iterations reach N, exit loop with warning #2.
- return to the top and do the next iteration

Convergence

If the error $|x_1 - x_2|$ decreases continuously and monotonically, we say that the algorithm is converging. In principle the solution is some real number that requires an infinite number of decimal places to define it precisely.

Convergence may require an infinite number of iterations but, in practice, we stop the algorithm after a finite number of iterations, because:

(a) we only need the solution to some specified accuracy,

(b) a digital computer permits only a finite number of significant figures to be stored. Once you reach this limit this algorithm is unable to provide a better estimate.

For single precision estimate (32 bits) the solution has about 7 significant figures. If $|x_1 - x_2| < 10^{-7}|x_1 + x_2|$, nothing more can be gained by further iteration.

Pitfalls

Things that can go wrong:

If Warning #1 is given (the current estimates of $f(x_1)$ and $f(x_2)$ are not of opposite sign):

Why ? no solution in bracketed range;
even number of solutions in bracketed range;

If Warning #2 is detected (the upper limit of iterations is reached):

Why ? convergence too slow;
convergence criterion too small;
too few iterations permitted.

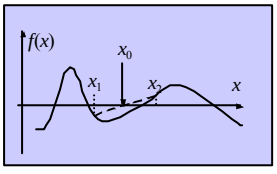
To diagnose: examine the values at each iteration of
x_1, x_2, $f(x_1)$, $f(x_2)$, $|x_1 - x_2|$, x_0, $f(x_0)$

The Interpolation Method

The bisection method is very robust if you provide two initial values that bracket the solution - but it does not necessarily converge quickly (how many steps required to reduce interval range by 100, or 1000 ?).

Faster convergence may be obtained if we draw a straight line between the two points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ then use the zero of that straight line to provide an approximate estimate of the zero of $f(x)$.

If the zero is within the range $[x_1,x_2]$ it is interpolated. If it is outside that range, it is extrapolated. It can work in either case, though interpolation is probably more accurate.



Interpolation

The interpolation is made by looking at the similar triangles formed by the dashed lines in the preceding sketch:

$$\frac{f(x_2)}{(x_2 - x_0)} = \frac{-f(x_1)}{(x_0 - x_1)}$$

solving for x_0:

$$x_0 = \frac{x_1 f(x_2) - x_2 f(x_1)}{[f(x_2) - f(x_1)]}$$

The interpolated point x_0 is closer to the actual solution (generally), so for the next estimate, we can set either x_1 or x_2 to x_0

if $|f(x_1)| > |f(x_2)|$ then $x_1 = x_0$, else $x_2 = x_0$.

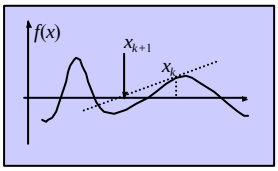
Newton's Method

Newton's method is based on a local linearisation of the function. It assumes that we can calculate the function $f(x_k)$ and its gradient $f'(x_k)$ at any x_k. The line that is tangent to the curve is projected back to the point where it intersects the x-axis: x_{k+1}. The diagram shows the relation between current estimate, tangent line, and new estimate. The whole process is iterated, as before.

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

or

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



Gradient Methods - possible problems

Newton's method is very powerful, and is the basis of a class of inversion methods applied to multi-dimensional problems.

It may also fail however: The function is linearized at the current estimate in order to project back along the gradient to get the "improved estimate". Depending on the function and the initial guess, the new estimate may not be an improvement, and the method may not converge, or may converge to a different, unexpected solution.

In general the method will work well if the function is well-behaved, the initial guess is "close enough" to the solution, and away from regions where the gradient is close to zero.

If convergence is obtained, the method is usually very efficient and quickly obtains an accurate answer.

Evaluation of Derivatives

Newton's method requires knowledge of the first derivative, and is most easily implemented if we have an analytic expression for the gradient $f'(x)$.

If this is unavailable, or difficult to evaluate, we can still proceed by means of a numerical evaluation of the derivative

$$f'(x_k) \approx \frac{f(x_k + h) - f(x_k)}{h}$$

where we increment the x_k value by the small h. The expression that we then obtain for x_{k+1} is exactly the same as that given by the interpolation method, but obviously relies on extrapolation.

Behaviour of the iterative algorithm is somewhat different because at each stage the extrapolation is based on the interval of length h.

Other Possible Problems

"Segmentation Fault": This is the helpful label that the computer gives you if you ask it to do something that is inconsistent with what the compiler expects of logically constructed code. E.g., the number and type of parameters in a CALL statement don't match those in the corresponding SUBROUTINE statement.

The number you print shows "Inf" or just "*********": This is caused by the calculation overflowing the capacity of the binary representation of a real number (around $\pm 10^{38}$ for 32-bit variables). E.g., if $|f(x_0) - f(x_1)|$ is too small, the interpolation step may overflow.

The program starts OK, but doesn't finish; nothing appears to happen, or if there is a print statement in the loop, the screen is filled and refilled until you interrupt it (Ctrl-C): The algorithm may have gone into an "infinite loop" because it wasn't terminated properly.