Secant Method

Requires two initial values (x_0, x_1)

Initial values do not have to lie on either side of root (not a bracketing method)

Iterative formula:

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)}$$

Like Newton-Raphson but with numerically estimated derivative.

$$\text{Newton} - \text{Raphson}: \quad x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - f(x_i) / f'(x_i)$$

$$\text{Secant}: \quad x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)} = x_i - f(x_i) / \left[\frac{f(x_{i-1}) - f(x_i)}{(x_{i-1} - x_i)} \right]$$

$\uparrow$

Approximate derivative

Modified Secant Method

Only one initial value required (x_0)

Iterative formula: $$x_{i+1} = x_i - \frac{\delta x_i \, f(x_i)}{f(x_i + \delta x_i) - f(x_i)}$$

where δ is a small perturbation fraction

Same basic idea but derivative estimated in a slightly different way

Approximate derivative

$$\text{Newton-Raphson:} \quad x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - f(x_i) / f'(x_i)$$

$$\text{Modified Secant:} \quad x_{i+1} = x_i - \frac{\delta x_i \, f(x_i)}{f(x_i + \delta x_i) - f(x_i)} = x_i - f(x_i) / \left[\frac{f(x_i + \delta x_i) - f(x_i)}{\delta x_i} \right]$$

Choice of δ important: too small -> numerical errors

too large -> inefficient, method may diverge

Technique	Key Formula	Notes
Bisection	$x_{ROOT} = \frac{(x_{LOW} + x_{HIGH})}{2}$	Root must be bracketed Slow but sure Error absolute
False Position	$x_{ROOT} = x_{HIGH} - \frac{f(x_{HIGH})(x_{HIGH} - x_{LOW})}{f(x_{HIGH}) - f(x_{LOW})}$	Variation on bisection Typically but not always faster Error approximate
Newton-Raphson	$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$	Fast but may go haywire Derivative required Error approximate
Secant	$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)}$	Fast but may go haywire Derivative not required Error approximate
Modified Secant	$x_{i+1} = x_i - \frac{\delta x_i f(x_i)}{f(x_i + \delta x_i) - f(x_i)}$	Fast but may go haywire Derivative not required Error approximate

Function *fzero*:

Two basic choices (as introduced earlier):

```
fzero (f, [low, high]);    % finds root between low and high
fzero (f, value);      % finds root in vicinity of value
```

Combines a number of techniques

When a single value is specified begins by using an iterative search to bracket a root (note: details not the same as the iterative search presented).

Uses faster methods (secant, inverse quadratic interpolation) when it can.

When things go wrong (e.g. guess moves outside bracket) reverts to sure and safe bisection search.

fzero **Options:**

Use *optimse*t to create an option structure:

```
>>  myOptions = optimset ('opt1', value1, 'opt2', value2, …);
```

Then add structure to *fzero* arguments:

```
>> root = fzero (f, [a b], myOptions);
```

Possible options:

‘**Display** ‘ - Level of display

‘off’   ‘iter’   ‘notify’   ‘final’

‘TolX’ - Termination tolerance on *x*

a positive scalar

N.B. Option ‘TolFun’ is NOT used by *fzero*.

```
>> f = @(x) x^2 - 4;

>> myOptions = optimset ('display', 'iter'); % display on each iteration
>> root = fzero (f, 3, myOptions);

Search for an interval around 3 containing a sign change:
 Func-count    a          f(a)             b          f(b)        Procedure
    1               3            5             3            5   initial interval
    3         2.91515      4.49808       3.08485      5.51632   search
    5            2.88       4.2944          3.12       5.7344   search
    7         2.83029      4.01057       3.16971      6.04703   search
    9            2.76       3.6176          3.24       6.4976   search
   11         2.66059      3.07873       3.33941      7.15167   search
   13            2.52       2.3504          3.48       8.1104   search
   15         2.32118      1.38786       3.67882      9.53374   search
   17            2.04       0.1616          3.96      11.6816   search
   18         1.64235     -1.30267          3.96      11.6816   search

Search for a zero in the interval [1.64235, 3.96]:
 Func-count    x          f(x)             Procedure
   18         1.64235     -1.30267        initial
   19         1.87488    -0.484837        interpolation
   20         2.00723    0.0289713        interpolation
   21         1.99977  -0.000932028       interpolation
   22               2 -1.68174e-006       interpolation
   23               2  3.53495e-013       interpolation
   24               2            0        interpolation

Zero found in the interval [1.64235, 3.96]
```

Specifying an interval eliminates the initial search phase

Specifying a relatively high x tolerance ends the search sooner

```
>> myOptions = optimset ('display', 'iter', 'TolX', 1e-3);
>> root = fzero (f, [1 4], myOptions);

 Func-count    x          f(x)             Procedure
    2               1            -3        initial
    3             1.6         -1.44        interpolation
    4         2.09451      0.386953        interpolation
    5         1.98977    -0.0408233        interpolation
    6         1.99976  -0.000946972        interpolation
    7         1.99976  -0.000946972        interpolation

Zero found in the interval [1, 4]
```

See p153-154 in text for another example (or, better yet, play around yourself)

Function *roots*:

(Previously discussed)

Finds all of the roots of a polynomial:

```
>> p = [ 2 5 -6 ];   % f(x) = 2  * x^2 + 5 * x - 6
>> roots (p)

ans =

   -3.3860
    0.8860
```

Input is a vector containing the polynomial coefficients (highest order coefficient first).

Output is a column vector.