ROOTS of EQUATIONS

Introduction
The root, $x_o$, of a function $f(x)$ is such that $f(x_o) = 0$. For example if $f(x) = x^3 - 28$, then

$$x_o = 28^{1/3} = 3.0365889718756625194208095785 \ldots$$

Numerical methods for finding roots of functions are used when analytical solutions are difficult (or impossible), or when a calculation is part of a larger numerical algorithm. Our strategy will be to define how accurate we want the solution to be and then compute the result approximately to this accuracy. This is called a tolerance, for example: Tolerance = 0.0001, this means that the root is required to be correct within plus or minus 0.0001 (four decimal place accuracy). For this to work we also need to be able to determine an error estimate with which the tolerance is compared and the algorithm terminates when “error estimate” $< \text{Tolerance}$.

In this chapter, we will study a number of basic numerical methods starting from very simple (and inefficient) sequential searches to very powerful Newton’s methods. The algorithms are divided into two groups: closed methods (where the solution is initially bracketed), and open method (where the solution is not bracketed).
Sequential Search (closed method)
In the sequential search first the position of the root is estimated such that a bracket a,b can be formed placing a lower- and upper-bound on the root. For this some initial analysis of the function is required. Note that if a single root (or odd number of roots) is bracketed by a and b then there will be a sign change between f(a) and f(b). During this search (scan) of the function we can identify a root as follows:

Search the function in the range a < x < b in steps of dx until we see a sign change. An estimate of the root can then be given as the center of the last inspected step with a maximum error of dx/2 (and mean error of about dx/4).

Computer Program:
```cpp
#include <iostream>
using namespace std;

double f(double x){
    return x*x*x-28;
}

int main(){
    double a, b, tol, x, dx;
    int i=1;
    cin >> a >> b >> tol;
    dx = 2*tol;
    while(1){
        x = a + dx*i++;
        if(f(x)*f(x+dx)<0) break; // sign changes
    }
    cout << "root = " << x+dx/2. << " + - " << fabs(dx/2.) << endl;
}
```

Output (for a=3, b=3.1 and tol = 1.0e-5)
root = 3.03659 + - 0.10e-06

Random Search (closed method)
In the random search first the position of the root is estimated such that a bracket a, b can be formed placing a lower- and upper-bound on the root. The search is done by randomly as given below program. Note that the error of this method is half of the tolerance.

Computer Program:
```cpp
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;

double f(double x){
    return x*x*x-28;
}

int main(){
    double a, b, tol, x;
    cin >> a >> b >> tol;
    while(1){
        x = a + (b-a) * rand()/(RAND_MAX+1.0); // random value between [a,b]
        if( fabs(f(x))<tol ) break;
    }
    cout << "root = " << x << " + - " << tol/2. << endl;
}
```

Output (for a=3, b=3.1 and tol = 1.0e-5)
root = 3.03659 + 5.0e-06
Bisection Method (closed method)
In the Bisection method first the position of the root is estimated such that a bracket can be formed placing a lower- and upper-bound on the root. For this some initial analysis of the function is required. A first estimate of the root is then computed as the mid-point between the two bounds.

Algorithm:

```
input lb, ub, tolerance
do{
    hb = (ub-lb)/2 // the error estimate
    mp = (ub+lb)/2 // the new root estimate
    output mp, hb
    if ( hb < tolerance ) break // terminate if tolerance is satisfied
    if ( f(lb)*f(mp) < 0 ) ub=mp else lb=mp
}while(1);
```

Fixed Point Iteration (open method)
To find the root of a function \( f(x)=0 \), we can derive one or more iteration of the form \( x = g(x) \) by solving for \( x \). We plot \( g(x) \) and \( y = x \) line.

- Iterative formula is \( x_{i+1} = g(x_i) \)
- Error estimate \( |x_{i+1}-x_i|<tol \)
- Around the initial guess \( (x_0) \) the rule \( |g'(x_0)|<1 \) must be satisfied.

Algorithm:

```
Define g(x) = ...
input x0, tol
do{
    y = g(x0) // the error estimate
    x1 = y // the new root estimate
    error = x1 - x0
    output x1, err
    if ( fabs(err) < tol ) break // terminate if tolerance is satisfied
    x0 = x1
}while(1);
```

Newton-Raphson (open method)
For a function \( f(x) \), we can find the root satisfying \( f(x)=0 \) iteratively:

\[
x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}
\]

where

- \( x_i \) is the current estimate,
- \( x_{i+1} \) is the next, improved, estimate.
- \( E = f(x)/f'(x) \) is the error estimate
- Iterations are terminated when \( E < \text{tolerance} \)
Algorithm:
```plaintext
input x // input the initial root estimate
input tol // input the tolerance (required accuracy)
do{
    Error = f(x) / f'(x) // the error estimate
    output x, Error // output current estimates
    if ( |Error| < Tolerance ) break // terminate if tolerance is satisfied
    x = x - Error // subtract the error estimate
}while(1);
```

Modified Newton-Raphson (open method)
Alternatively, one can replace the derivative with CDA to obtain:

\[
x_{i+1} = x_i - \frac{f(x_i)}{[f(x_i + h) - f(x_i - h)]/2h}
\]

where \( h \) is also input.

Secant Method (open method)
The main disadvantage of the Newton-Raphson method is that it requires a knowledge of the first derivative. If the first derivative is not known, or is inconvenient to implement, then it can be approximated numerically by the iterative form of the Forward Difference Approximation, this leads to the Secant Method:

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

where
- \( x_{i-1} \) is the previous estimate,
- \( x_i \) is the current estimate,
- \( x_{i+1} \) is the next, improved, estimate.

Conclusion
Below is a comparison of the Bisection, Newton-Raphson, and Secant methods for finding the root of \( f(x) = x^3 - 28 \) with a tolerance of \( 10^{-12} \).

<table>
<thead>
<tr>
<th>Method</th>
<th>Root Estimate</th>
<th>Error Estimate</th>
<th>True Error</th>
<th>Number of Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>3.036 588 971 875 663</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bisection</td>
<td>3.036 588 971 876 336</td>
<td>728E-15</td>
<td>673E-15</td>
<td>36</td>
</tr>
<tr>
<td>Secant</td>
<td>3.036 588 971 875 642</td>
<td>20E-15</td>
<td>20E-15</td>
<td>4</td>
</tr>
<tr>
<td>Newton-Raphson</td>
<td>3.036 588 971 875 664</td>
<td>1E-15</td>
<td>1E-15</td>
<td>3</td>
</tr>
</tbody>
</table>

The Newton-Raphson method has the fastest convergence with the Secant method a close second. An additional advantage of the Newton-Raphson method over the Bisection and Secant methods is that it does not require upper and lower bounds as inputs. However, a disadvantage is that it requires a knowledge of the first derivative (which is not always available). Also, the
Newton-Raphson method can fail at or close to turning points in the function (why?). The Bisection method guarantees convergence whereas both the Newton-Raphson and Secant methods can fail to converge on a root.

Fig 3.1 shows the effect of the number of iterations on the error estimate for the methods of Bisection (the initial bracket $a=3$, $b=4$) and Newton-Raphson (initial estimate $x=3$) for the function $f(x) = x^3 - 28$.

Fig 3.1 Comparison of the error estimates of the Bisection and Newton-Raphson methods with respect to iterations for the function $f(x) = x^3 - 28$. (a) log(error) vs integrations and (b) error vs iterations. Logarithmic graph is more clear for the comparison.
An Engineering Application

The velocity $v$ of a falling parachutist is given by

$$v = \frac{gm}{b} \left(1 - e^{-bt/m}\right)$$

Where $g = 9.8 \text{ m/s}^2$. For a parachutist with a drag coefficient $b = 14 \text{ kg/s}$, compute the mass $m$ so that the velocity is $v = 35 \text{ m/s}$ at $t = 7 \text{ s}$.

Solution

For the given values, the velocity as a function of $m$ can be written as follows

$$35 = 0.7m(1 - e^{-98/t/m})$$

The purpose is to find the mass $m$ from this equation. Note that there is no analytical solution.

The results of numerical solutions (Secant and Bisection method) for the function are as follows.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Root</th>
<th>Error Est.</th>
<th>Iter</th>
<th>Root</th>
<th>Error Est.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>70.0000000</td>
<td>6.1465650</td>
<td>0</td>
<td>65.0000000</td>
<td>5.0000000</td>
</tr>
<tr>
<td>1</td>
<td>63.8534350</td>
<td>0.2151667</td>
<td>1</td>
<td>62.5000000</td>
<td>2.5000000</td>
</tr>
<tr>
<td>2</td>
<td>63.6382683</td>
<td>-0.0114443</td>
<td>2</td>
<td>63.7500000</td>
<td>1.2500000</td>
</tr>
<tr>
<td>3</td>
<td>63.6497126</td>
<td>0.0000204</td>
<td>3</td>
<td>63.1250000</td>
<td>0.6250000</td>
</tr>
<tr>
<td>4</td>
<td>63.6496922</td>
<td>0.0000000</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The results of numerical solutions (Secant and Bisection method) for the function are as follows.
Computer implementations of the methods are given below:

// The secant method
#include <iostream>
#include <cmath>
using namespace std;

double f(double m){
    double y = 0.7*m*( 1.0-exp(-98/m) ) - 35.0;
    return y;
}

int main()
{
    const double TOL = 1.0e-6;
    int iter = 0;
    double x0 = 60.0, x1 = 70.0, error;

    while(1){
        error = f(x1)*(x1-x0)/(f(x1)-f(x0));
        cout << iter << '\t' << x1 << '\t' << error << endl;

        if(fabs(error)<TOL) break;
        x0 = x1;
        x1 = x1 - error;
        iter = iter + 1;
    }
}

! Bisection Method
PROGRAM Bisection
IMPLICIT NONE
DOUBLE PRECISION, PARAMETER :: TOL = 1.0e-6
INTEGER :: iter = 0
DOUBLE PRECISION :: lb = 60.0, ub =70.0, mp, hb

do
    hb = (ub-lb)/2.0
    mp = (ub+lb)/2.0
    PRINT *, iter, mp, hb

    if(ABS(hb)<TOL) exit
    if(f(lb)*f(mp)<0) THEN
        ub = mp
    else
        lb = mp
    end if
    iter = iter + 1
end do

CONTAINS

DOUBLE PRECISION FUNCTION f(m)
DOUBLE PRECISION :: m
    f = 0.7*m*( 1.0-exp(-98/m) ) - 35.0
END FUNCTION f

END PROGRAM Bisection