Chapter IX

Roots of Equations

Root solving typically consists of finding values of x which satisfy a relationship such as:

$$3.0x^3 + 2x = 1.0 + e^{-2x}$$

The problem can be restated as: Find the root of

$$f(x) = 3.0x^3 + 2x - 1.0 - e^{-2x}$$

In this chapter we will cover three methods for root finding:

- The half-interval method
- The Newton-Raphson Method
- The Bairstow method for roots of polynomials

9.1 The Half-Interval Method

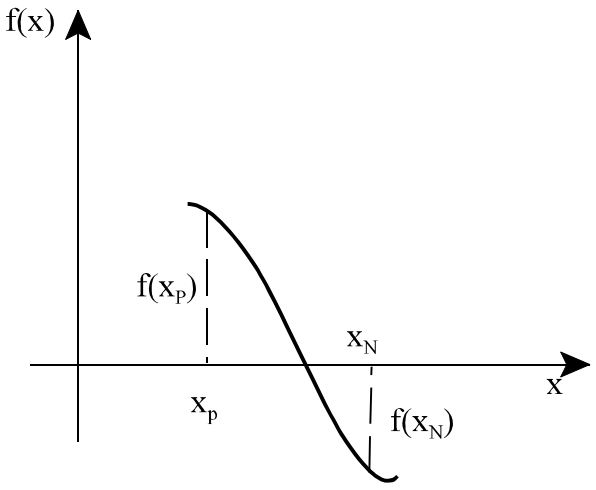

Figure 9.1 Half-Interval Method.

Find two values that bracket a root of the given function. That is, the root is bounded by (x_P, x_N) such that

$$f(x_P) > 0 \quad \text{and} \quad f(x_N) < 0$$

The half interval method is described as follows:

2. Compute $x_M = \dfrac{x_P + x_N}{2}$
3. Compute $f(x_M)$
4. If $|f(x_M)| \le \varepsilon$ $\quad Root = x_M$ exit
5. If $f(x_M) < 0$ $\quad x_N = x_M$
   else $x_P = x_M$
6. Repeat steps (1) to (4).

ε is a small number, typically 10^{-7}, that determines the accuracy of the root.

Example. Find $\sqrt{7}$ using the half-interval method.
Solution.

The equation to solve is $x^2 - 7 = 0$ or obtain the root of:

$$f(x) = x^2 - 7$$

Following the above procedure let $x_P = 3$ $\quad f(3) = 9 - 7 = 2 > 0$

and $x_N = 2$ $\quad f(2) = 4 - 7 = -3 < 0$

1. $x_M = \dfrac{3+2}{2} = \dfrac{5}{2} = 2.5$
2. $f(x_M) = (2.5)^2 - 7 = -0.75 < 0$
3. $|f(x_M)| \le 10^{-7}$? no.
4. $x_N = 2.5$ $\quad x_P = 3$
5. Repeat Steps 1 to 4.

1. $x_M = \dfrac{2.5+3}{2} = 2.75$
2. $f(x_M) = (2.75)^2 - 7 = 0.5625 > 0$
3. $|f(x_M)| < 10^{-7}$? no

4. $x_P = 2.75 \qquad x_N = 2.5$

.

.

etc.

Example. Develop a class Root based on the Half-interval method for root finding.

Solution. The following utilizes pointer to a function to allow the user of the class "Root" to supply his own function without having to make any changes in the class. Thus leaving the integrity of the class intact (the way it's supposed to be.)

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

class Root {
   private:
      double xP, xN;
      double (*f)(double x); //pointer to user supplied function
   public:
      Root(double(*F)(double x), double xdP, double xdN)
       {
       f=F;
       xP=xdP;
       xN=xdN;
       }

       double HLFINT();
       };

double Root::HLFINT()
 {
 double xM;
 double y;
 double EPS=1.e-7;
do{
 xM=(xP+xN)/2.0;
 y=f(xM);
 if(y>0.0) xP=xM;
 else xN=xM;
  } while(fabs(y) > EPS);
 return xM;
  }
```

```
//User supplied function
double fun(double x)
 {
 return x*x-7.0;
 }

//---------------------------------------------------------------------------

int main()
{
Root sqrt7(fun,3,2);

cout << "Square root of 7 = " << sqrt7.HLFINT() << endl;

getch();
 return 1;
  }
```

9.2 Newton-Raphson's Method

The Taylor series expansion of $f(x)$ about x_0 is given by:

$$f(x) = f(x_0) + (x - x_0) f'(x_0) + \frac{(x - x_0)^2}{2!} f''(x_0) + \ldots..$$

If $f(x) = 0$ then x must be the root of $f(x)$. Neglecting terms higher or equal to second order we can write

$$0 = f(x_0) + (x - x_0) f'(x_0)$$

$$\therefore \qquad (x - x_0) = -\frac{f(x_0)}{f'(x_0)}$$

Writing the above equation in an iterative form we get:

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

A graphical interpretation of the Newton-Raphson's method is shown in Fig.9.2.

From Fig. 9.2 we can write:

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

$$\text{or} \quad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

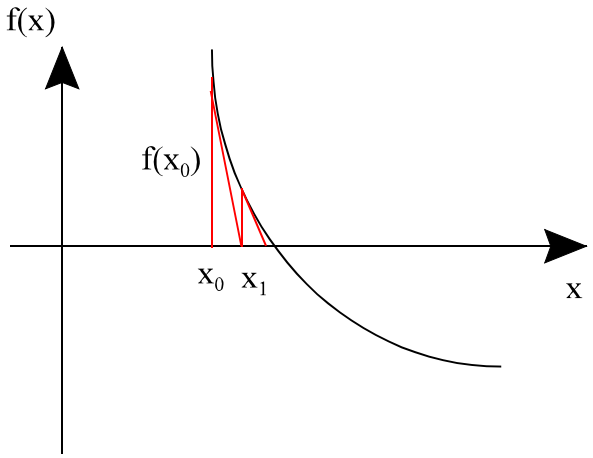

Figure 9.2 The Newton-Raphson's Method

After computing x_1 from x_0 set $x_0 = x_1$ and repeat the process. In so doing we can approach the root in a few iterations.

Procedure

1. Select an initial value x_1
2. Compute $f(x_1)$
3. Compute $f'(x_1)$
4. Compute $x_2 = x_1 - \dfrac{f(x_1)}{f'(x_1)}$
5. If $|x_2 - x_1| \leq \varepsilon$ then root x_2 and exit
6. Repeat steps (2) to (5).

In the above procedure we used a different stopping criterion from the one used with the half-interval method. You can use any one of the two criteria for terminating the iteration in either methods of

root finding, however, in some cases where the slope within the vicinity of the root is small the second criterion would be more suitable.

Example. Modify the class Root to include the Newton-Raphson's method.

Solution. To do so we will overload the constructor to add the derivative of the function and a starting point. The modified class Root is shown next.

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

class Root {
   private:
      double xP, xN;
      double xst; //start x for newton-raphson
      double (*f)(double x); //pointer to user supplied function
      double (*fd)(double x); //derivative of (*f)(x) for newton-raphson
   public:
      Root(double(*F)(double x), double xdP, double xdN)
       {
       f=F;
       xP=xdP;
       xN=xdN;
       }
      Root(double(*F)(double x), double(*FD)(double x), double xi )
       {
       f=F;
       fd=FD;
       xst=xi;
       }

       double HLFINT();  //Half interval
       double NR();  //Newton-Raphson
       };

double Root::HLFINT()
 {
 double xM;
 double y;
 double EPS=1.e-7;
do{
  xM=(xP+xN)/2.0;
  y=f(xM);
  if(y>0.0) xP=xM;
```

```
   else xN=xM;
     } while(fabs(y) > EPS);
   return xM;
    }

double Root::NR()
  {
  double EPS=1.e-7;
  double x1, x2;
  x1=xst;
  while(1)
    {
    x2=x1-(*f)(x1)/(*fd)(x1);
    if(fabs(x2-x1)<=EPS) break;
    x1=x2;
     }
   return x2;
   }

//User supplied function
double fun(double x)
  {
  return x*x-7.0;
  }

//User supplied derivative
double fund(double x)
  {
  return 2.0*x;
  }

//---------------------------------------------------------------------------

int main()
{
Root sqrt7(fun,3,2);
Root sqrt7NR(fun,fund,2);

cout << "Square root of 7 using half interval method= "
       << sqrt7.HLFINT() << endl;
cout <<  "Square root of 7 using Newton Raphson's method= "
       << sqrt7NR.NR() << endl;
```

```
getch();
 return 1;
  }
```

Example. Find $\sqrt{7}$ using the Newton-Raphson's method (ε = 0.01).

Solution.

Using Newton-Raphson's method we can derive the iterative equation:

$$x_2 = \frac{x_1^2 + a}{2x_1} \quad \text{where a = 7 for this problem.}$$

We will leave the derivation as well as the hand calculation as an exercise to the reader.

Exercise. Develop a function for the square-root of a positive number 'a'. Use as initial guess x=a/2 if a>1 and x=2a if a<1.

Exercise. The square-root of a complex number can be obtained as follows:

$$x + iy = \sqrt{a + ib}$$

$$\text{or} \quad x^2 - y^2 + i2xy = a + ib$$

Equating real to real and imaginary to imaginary we get:

$$x^2 - y^2 = a$$

$$2xy = b$$

Solving the above two equations to isolate x we get:

$$4x^4 - 4ax^2 - b^2 = 0$$

and

$$y = \frac{b}{2x}$$

Using the last two equations and Newton-Raphson's method develop an algorithm and function for obtaining the square-root of a complex number.

9.3 The Bairstow's method

Bairstow's method is an iterative method which involves finding quadratic factors, $f(x) = x^2 + ux + v$ of the polynomial:

$$a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_n = 0$$

where a_0=1.

We will first present the procedure, then the C++ code and then we will follow with the derivation of the method. This is going in reverse to the traditional approach of derivation, procedure, and then code but it helps to keep the focus on development and builds curiosity as to the source of the approach after one has experienced success with the method.

Procedure

1. Select initial values for u and v. Most frequently these are taken as u=v=0.
2. Calculate $b_1, b_2, \ldots, b_n$ from

$$b_i = a_i - b_{i-1}u - b_{i-2}v \quad (i = 2,3,\ldots,n)$$
$$\text{where } b_0 = 1 \text{ and } b_1 = a_1 - u$$

3. Calculate $c_1, c_2, c_3, \ldots, c_{n-1}$ from

$$c_i = b_i - c_{i-1}u - c_{i-2}v \quad (i = 2,3,\ldots,n-1)$$
$$\text{where } c_0 = 1 \text{ and } c_1 = b_1 - u$$

4. Calculate Δu and Δv from:

$$\Delta u = \frac{\begin{bmatrix} b_n & c_{n-2} \\ b_{n-1} & c_{n-3} \end{bmatrix}}{\begin{bmatrix} c_{n-1} & c_{n-2} \\ c_{n-2} & c_{n-3} \end{bmatrix}} \qquad \Delta v = \frac{\begin{bmatrix} c_{n-1} & b_n \\ c_{n-2} & b_{n-1} \end{bmatrix}}{\begin{bmatrix} c_{n-1} & c_{n-2} \\ c_{n-2} & c_{n-3} \end{bmatrix}}$$

5. Increment u and v by Δu and Δv

$$u = u + \Delta u$$
$$v = v + \Delta v$$

6. Return to step 2 and repeat the procedure until Δu and Δv approach zero within some preassigned value ε such that

$$|\Delta u| + |\Delta v| < \varepsilon$$

An upper limit of iterations should be specified to protect against non-convergence. If

convergence to the desired result does not occur after the specified number of iterations, then new starting values should be selected. Frequently, suitable starting values for *u* and *v* may be determined from $u = a_{n-1}/a_{n-2}$ and $v = a_n/a_{n-2}$.

The following C++ code follows from the above procedure. It does not, however, contain all the bells and whistles. For example it does not check for the number of iterations before convergence (we leave that part to the reader to modify). The class RootPoly in the program is tested on two polynomials:

$$x^5 - 3x^4 - 10x^3 + 10x^2 + 44x + 48 = 0$$

$$x^3 - 3x^2 - x + 9 = 0$$

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

class RootPoly {
   private:
      double *a;
      double *b;
      double *c;
      int n;
      double *rootr;
      double *rooti;

   public:
      RootPoly(double *coeff, int n);
      ~RootPoly();
      void Bairstow();
      double *getRootsr()
        {
        return rootr;
        }

      double *getRootsi()
        {
        return rooti;
        }
     };

RootPoly::RootPoly(double *coeff, int order)
```

```
  {
n=order;
a=new double[n+1];
b=new double[n+1];
c=new double[n];
for(int i=0; i <= n; i++)
   a[i]=coeff[i];

rootr = new double[n];
rooti = new double[n];
   }

RootPoly::~RootPoly()
 {
 delete[] a;
 delete[] b;
 delete[] c;
 delete[] rootr;
 delete[] rooti;
 }

void RootPoly::Bairstow()
  {
  double u,v;
  double EPS=1.e-7, D, Du, Dv, d;
  int i,k=0;

//selecting initial values for u and v
  if(a[n-2]!=0.0)
    {
    u=a[n-1]/a[n-2];
    v=a[n]/a[n-2];
    }
  else
     u=v=1;

//Starting the iterations
  int iter=0;
  while(n>=2)
    {
    iter++;
    do{
       b[0]=1.0; b[1]=a[1]-u;
```

```
  for(i=2; i <= n; i++)
    b[i]=a[i]-b[i-1]*u-b[i-2]*v;

  c[0]=1; c[1]=b[1]-u;
  for(i=2; i < n; i++)
    c[i]=b[i]-c[i-1]*u-c[i-2]*v;
 if(n==3)
    {
  D=c[n-1]-c[n-2]*c[n-2];
  Du=(b[n]-b[n-1]*c[n-2])/D;
     }
  else
    {
  D = c[n-1]*c[n-3]-c[n-2]*c[n-2];
  Du = (b[n]*c[n-3]-b[n-1]*c[n-2])/D;
     }
  Dv = (b[n-1]*c[n-1]-b[n]*c[n-2])/D;
  if((fabs(Du)+fabs(Dv))<EPS) break;
   u=u+Du;
   v=v+Dv;
  } while(1);
 iter=0;
 d=u*u-4.0*v;

 if(d >= 0)
   {
   rootr[k] = (-u + sqrt(d))/2.0;
   rooti[k] = 0.0;
   k++;
   rootr[k] = (-u - sqrt(d))/2.0;
   rooti[k] = 0.0;
   k++;
   }
  else
   {
   rootr[k] = -u/2.0;
   rooti[k] = sqrt(-d)/2.0;
   k++;
   rootr[k] = -u/2.0;
   rooti[k] =-sqrt(-d)/2.0;
   k++;
   }
n=n-2;
```

```
  for(i=0; i<=n; i++)
    a[i] = b[i];
   }  //closing bracket for while

  if(n == 1)
    {
    rootr[k] = -a[1];
    rooti[k] = 0.0;
    }

  if(n==2)
    {
    d=a[1]*a[1]-4.0*a[2];
    if( d < 0 )
      {
      rootr[k]=-a[1]/2.0;
      rooti[k]=sqrt(-d)/2.0;
      k++;
      rootr[k]=-a[1]/2.0;
      rooti[k]=-sqrt(-d)/2.0;
      }
     else
       {
       rootr[k]=(-a[1]+sqrt(d))/2.0;
       rooti[k]=0.0;
       k++;
       rootr[k]=(-a[1]-sqrt(d))/2.0;
       rooti[k]=0.0;
       }
     }
  }


//---------------------------------------------------------------------------

int main()
{
double coeff[]={1,-3,-10,10,44,48};
//double coeff[]={1,-3,-1,9};

int n=sizeof(coeff)/sizeof(double)-1;
cout << "n=" << n << endl;
RootPoly roots(coeff,n);
```

```
roots.Bairstow();
double *RootsReal, *RootsImag;
RootsReal=new double[n];
RootsImag=new double[n];

RootsReal=roots.getRootsr();
RootsImag=roots.getRootsi();
cout << "ROOTS : " << endl;
for(int i=0; i<n; i++)
  {
  if( RootsImag[i]>0)
  cout << RootsReal[i] << " +i" << RootsImag[i] << endl;
  else if(RootsImag[i]<0)
  cout << RootsReal[i] << " -i" << -RootsImag[i] << endl;
  else
  cout << RootsReal[i] << endl;
   }

getch();
 return 1;
  }
```

Those who have utilized procedural languages such FORTRAN, BASIC or C can realize the great advantage of working with OOP C++. Classes offer the programmer the ability of developing efficient code. Each member function specializes in one specific task. For example memory allocation is carried-out in the constructor and deallocation in the destructor. Other member functions each is specialized to carry-out a specific task without the need to include data initialization. This is carried out by the constructor as in the above program along with memory allocation of private pointer variables. This a far superior approach to just structure programming used in procedural languages.

Running the above program for each polynomial we get the following answers:

```
n=5
ROOTS :
3
-2
-1 +i1
-1 -i1
4

n=3
```

```
ROOTS :
2.26255 +i0.884368
2.26255 -i0.884368
-1.5251
```

You can check if the answers are correct by substituting each coefficient in the polynomial. The program does not carry-out this task but that is something that can be included in the version that the reader will develop. Anyway, the answers are correct and now we are ready to move on to the derivation of the Bairstows method.

Dividing the polynomial $x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_n$ by a quadratic equation of the form $x^2 + ux + v$ yields a quotient which is polynomial of order *n*-2 and a remainder which is a polynomial of order one. That is:

$$x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_n$$
$$= (x^2 + ux + v)(x^{n-2} + b_1x^{n-1} + b_2x^{n-2} + \ldots + b_{n-3}x + b_{n-2}) + b_{n-1}(x + u) + b_n$$

where $b_{n-1}(x + u) + b_n$ is the remainder after division of the polynomial by the quadratic equation. The reason the remainder was placed in this form is to provide simplicity in the solution as we will see later. Equating coefficients of like powers on both sides we get:

$$\begin{aligned}
a_1 &= b_1 + u \\
a_2 &= b_2 + ub_1 + v \\
a_3 &= b_3 + ub_2 + vb_1 \\
&\vdots \\
a_i &= b_i + ub_{i-1} + vb_{i-2} \\
&\vdots \\
a_{n-1} &= b_{n-1} + ub_{n-2} + vb_{n-3} \\
a_n &= b_n + ub_{n-1} + vb_{n-2}
\end{aligned}$$

or we can write

$$
\begin{aligned}
b_1 &= a_1 - u \\
b_2 &= a_2 - ub_1 - vb_0 \\
b_3 &= a_3 - ub_2 - vb_1 \\
&\cdot \\
&\cdot \\
&\cdot \\
b_i &= a_i - ub_{i-1} - vb_{i-2} \\
&\cdot \\
&\cdot \\
&\cdot \\
b_{n-1} &= a_{n-1} - ub_{n-2} - vb_{n-3} \\
b_n &= a_n - ub_{n-1} - vb_{n-2}
\end{aligned}
\tag{9.1}
$$

We would like both b_{n-1} and b_n to be zero to ensure that the quadratic $x^2 + ux + v$ is a factor of the polynomial. Since the b's are functions of u and v and if we can adjust u by Δu and v by Δv such that both b_{n-1} and b_n approach zero. Of course we will have to do this iteratively as we have observed in the procedure we used in developing the C++ code for Bairstow's method . Expressing b_{n-1} and b_n as functions of two variables $u+\Delta u$ and $v+\Delta v$ and expanding in a Taylor series expansion for two variables. We assume that Δu and Δv are small enough that we can neglect higher-order terms.

$$
\begin{aligned}
b_n(u + \Delta u, v + \Delta v) = 0 &\cong b_n + \frac{\partial b_n}{\partial u}\Delta u + \frac{\partial b_n}{\partial v}\Delta v \\
b_{n-1}(u + \Delta u, v + \Delta v) = 0 &\cong b_{n-1} + \frac{\partial b_{n-1}}{\partial u}\Delta u + \frac{\partial b_{n-1}}{\partial v}\Delta v
\end{aligned}
\tag{9.2}
$$

where the terms on the right are evaluated for values of u and v.

Taking partial derivatives of equations (9.1) with respect to u and defining these in terms of c's, we obtain the following:

$$
\begin{aligned}
\frac{\partial b_1}{\partial u} &= -1 = -c_0 \\
\frac{\partial b_2}{\partial u} &= u - b_1 = -c_1 \\
\frac{\partial b_3}{\partial u} &= -b_2 + c_1 u + v = -c_2 \\
\frac{\partial b_4}{\partial u} &= -b_3 + c_2 u + c_1 v = -c_3 \\
&\vdots \\
\frac{\partial b_i}{\partial u} &= -b_{i-1} + c_{i-2} u + c_{i-3} v = -c_{i-1} \\
&\vdots \\
\frac{\partial b_{n-1}}{\partial u} &= -b_{n-2} + c_{n-3} u + c_{n-4} v = -c_{n-2} \\
\frac{\partial b_n}{\partial u} &= -b_{n-1} + c_{n-2} u + c_{n-3} v = -c_{n-1}
\end{aligned}
\tag{9.3}
$$

Similarly, taking the partial derivatives of equations (9.1) with respect to *v* and relating these with respect to the c's, we obtain the following:

$$
\begin{aligned}
\frac{\partial b_1}{\partial v} &= 0 \\
\frac{\partial b_2}{\partial v} &= -1 = c_0 \\
\frac{\partial b_3}{\partial v} &= u - b_1 = -c_1 \\
\frac{\partial b_4}{\partial u} &= -b_2 + c_1 u + v = -c_2 \\
&\vdots \\
\frac{\partial b_i}{\partial v} &= -b_{i-2} + c_{i-3} u + c_{i-4} v = -c_{i-2} \\
&\vdots \\
\frac{\partial b_{n-1}}{\partial v} &= -b_{n-3} + c_{n-4} u + c_{n-5} v = -c_{n-3} \\
\frac{\partial b_n}{\partial v} &= -b_{n-2} + c_{n-3} u + c_{n-4} v = -c_{n-2}
\end{aligned}
\tag{9.4}
$$

From either Eq.9.3 and 9.4, we see that

$$c_i = b_i - c_{i-1} - c_{i-2}v \qquad (i = 2,3,\ldots,n-1) \tag{9.5}$$

where $c_0 = 1$ and $c_1 = b_1 - u$.

From Eqs. (9.3) and (9.4), Eq.(9.2) can be written in the form:

$$b_n = c_{n-1}\Delta u + c_{n-2}\Delta v$$
$$b_{n-1} = c_{n-2}\Delta u + c_{n-3}\Delta v$$

Solving these two simultaneous equations we get:

$$\Delta u = \frac{\begin{bmatrix} b_n & c_{n-2} \\ b_{n-1} & c_{n-3} \end{bmatrix}}{\begin{bmatrix} c_{n-1} & c_{n-2} \\ c_{n-2} & c_{n-3} \end{bmatrix}} \qquad \Delta v = \frac{\begin{bmatrix} c_{n-1} & b_n \\ c_{n-2} & b_{n-1} \end{bmatrix}}{\begin{bmatrix} c_{n-1} & c_{n-2} \\ c_{n-2} & c_{n-3} \end{bmatrix}}$$

which leads us to the procedure described previously for the Bairstow's method.

Problems.

1. Find a root of $F(x) = x - \tan x$ in the interval [4.0,4.5] using the Half-interval method.
2. Repeat problem 1 using the Newton-Raphson method with a starting value of 4.5.
3. Find the root of $F(x) = x\cos x - \sin x$ in the interval [4.0,4.5] using the Newton-Raphson method with a starting value of 4.
4. Obtain the square root of the complex number 7+i6 using the Newton-Raphson Method.
5. When the Earth is assumed to be a spheroid, the gravitational attraction at a point on its surface at the latitude angle ϕ is

$$g(\phi) = 9.78049(1 + 0.0052884\sin^2(\phi) - 0.0000059\sin^2(2\phi)) \ \text{m/sec}^2$$

   At what latitude(s), in degrees, is g(ϕ)=9.8?
6. Obtain the roots of the following polynomial using the Bairstow method:

$$4x^4 + 2x^3 + 3x^2 + x + 1 = 0$$

(use the program given in sec.9.3).

7. Improve the program given in sec. 9.3 by including convergence checking and calculation of residual value for each root, i.e. the value obtained when substituting each root in the given polynomial.