```
//Question 2----------------------------------------------------------------------
----------


/*
newton-raphson method for root finding.

*/

#include <iostream>
#include <cstdlib>
#include <cmath>
#include <fstream>

using std::endl;
using std::cout;
using std::ofstream;

// define function:

double func_y(double x){
  return sqrt(pow(x,2)+2*x+1)-2*x*sin(x); /* <---- input the function that you want to
find root on here . */
}


//define the first derivative of your function:

double func_dydx(double s){
  return (s+1)/sqrt(pow(s,2)+2*s+1)-2*sin(s)-2*s*cos(s); /* <---- replace it with the d
ifferentiation of your function*/
}


int main()
{
   int N= 0;
   int N_limit = 1000;           // max number of steps
   double s = 2.0;              // initial guess for root,
   double tol = 1.0e-7;          // tolerance


   ofstream out_file;
   const char out_file_name[] = "newton.dat";  // save the data in newton.dat

   out_file.open(out_file_name);

        while ((fabs(func_y(s)) >= tol) && (N < N_limit)){

     s = s - func_y(s)/func_dydx(s);
     ++N;

   out_file << N << " " << s << endl; // write the date to output file

        }

   if(N >= N_limit){
     cout << "iteration limit exceeded" << endl;
     return 0;
   }

   if (N <= N_limit){
     cout << "estimated root = " << s << " with " << N << " iterations." << endl;
   }

   return 0;
}



/*
bisection method for root finding
of function f(x) in closed interval [a,b].
*/

#include <iostream>
#include <cstdlib>
#include <cmath>
#include <fstream>

using std::endl;
using std::cout;
using std::ofstream;

// define function:

double func_f(double x){
  return tanh(x);
}


int main()
{
   int N= 0;
   int N_limit = 1000;           // max number of steps

   double a = -10.0;   // lower bound for [a,b]
   double b = 3.0;      // upper bound for [a,b]

   double tol = 1.0e-10;          // tolerance f

   ofstream out_file;
   const char out_file_name[] = "Bisection.dat";  // save data in Bisection.dat

   out_file.open(out_file_name);

   if (func_f(a) == 0.0) {
     cout << a << " is a root." << endl;
     return 0;
   }

   if (func_f(b) == 0.0) {
     cout << b << " is a root." << endl;
     return 0;
   }

   if (func_f(a)*func_f(b) > 0.0){
     cout << "[" << a << "," << b << "] do not bracket a root." << endl;
     return 0;
   }

   double x_low = a;
   double x_high = b;
   double x_mid;

   while ((fabs(x_high - x_low) >= tol) && (N < N_limit)){
     x_mid = 0.5*(x_low + x_high);
     if ( func_f(x_mid)*func_f(x_high) < 0){
       x_low = x_mid;
     }
     else {
       x_high = x_mid;
     }
   out_file << x_mid << " " << N << endl; // write date to output file
     ++N;
   }

   if(N > N_limit){
     cout << "iteration limit exceeded" << endl;
     return 0;
   }

   if (N <= N_limit){
     cout << "estimated root = " << x_mid << " with " << N << " iterations." << endl;
   }

   return 0;
}

// plots the .dat files on gnuplot.


//Question 2--------------------------------------------------------------------------
----------

//1, Newton-Raphson


/*
newton-raphson.

*/

#include <iostream>
#include <cstdlib>
#include <cmath>
#include <fstream>

using std::endl;
using std::cout;
using std::ofstream;

// define function:

double func_y(double x){
  return sin(x); /* <----  replace it of your function */
}


//define first derivative of function:

double func_dydx(double s){
  return cos(s); /* <---- the differentiation of your function */
}


int main()
{
   int N= 0;
   int N_limit = 1000;           // max number of trys
   double s = 0.1;              // initial guess for root, /* <---- for the 2nd functio
n, replace this by 3, or -5 for the 3rd function. */
   double tol = 1.0e-4;          // tolerance for finding zero


   ofstream out_file;
   const char out_file_name[] = "newton.dat";  // save data in fern.dat

   out_file.open(out_file_name);

   while ((fabs(func_y(s)) >= tol) && (N < N_limit)){

     s = s - func_y(s)/func_dydx(s);
     ++N;

   out_file << N << " " << s << endl; // write date to output file

        }

   if(N >= N_limit){
     cout << "iteration limit exceeded" << endl;
     return 0;
   }

   if (N <= N_limit){
     cout << "estimated root = " << s << " with " << N << " iterations." << endl;
   }

   return 0;
}


//and for Bisection Algorithm, the source code is:

/*
bisection method for root finding
of function f(x) in closed interval [a,b].
*/

#include <iostream>
#include <cstdlib>
#include <cmath>
#include <fstream>

using std::endl;
using std::cout;
using std::ofstream;

// define function:

double func_f(double x){
  return sin(x); // <----replace this with x*sqrt(fabs(x)) for the 2nd function.
}


int main()
{
   int N= 0;
   int N_limit = 1000;           // max number of trys

   double a = -2.9;   // lower bound for [a,b]
   double b = 3.0;      // upper bound for [a,b]
   double tol = 1.0e-4;          // tolerance for finding zero

   ofstream out_file;
   const char out_file_name[] = "Bisection.dat";  // save data in fern.dat

   out_file.open(out_file_name);

   if (func_f(a) == 0.0) {
     cout << a << " is a root." << endl;
     return 0;
   }

   if (func_f(b) == 0.0) {
     cout << b << " is a root." << endl;
     return 0;
   }

   if (func_f(a)*func_f(b) > 0.0){
     cout << "[" << a << "," << b << "] do not bracket a root." << endl;
     return 0;
   }

   double x_low = a;
   double x_high = b;
   double x_mid;

   while ((fabs(x_high - x_low) >= tol) && (N < N_limit)){
     x_mid = 0.5*(x_low + x_high);
     if ( func_f(x_mid)*func_f(x_high) < 0){
       x_low = x_mid;
     }
     else {
       x_high = x_mid;
     }
   out_file << x_mid << " " << N << endl; // write date to output file
     ++N;
   }

   if(N > N_limit){
     cout << "iteration limit exceeded" << endl;
     return 0;
   }

   if (N <= N_limit){
     cout << "estimated root = " << x_mid << " with " << N << " iterations." << endl;
   }

   return 0;
}

/*
iterations table:


                                        newton          Bisection

sin(x)                          2                       16

x*sqrt(fabs(x))         1                       16


*/
```

```
//Question 3--------------------------------------------------------------------
----------

/*

 mumerov method to find eigenvalues and eigenfunctions of
            a particle in a potential well.

*/
#include <iostream>
#include <cmath>
#include <fstream>

using namespace std;


const double V0 = -83.0;             // depth of potential well in MeV
const double E_min = -83.0;          // energy band to find
const double E_max = -65.0;          // energy eigenvalue
const double eps = 1E-6;             // matching tolerance
double h = 0.004;                    // step size
                                     // well is assumed 4 fm wide
                                     // universe is 8 fm wide
                                     // 2000 total steps


double k2(int i, double E);          /* returns potential at x */
double diff(double E);               /* difference of derivatives */
void write_data(double E);           /* saved data for final plot */

int main()
{
  double E_mid, E_lo, E_hi;
  int i=0;                           // counter for iterations

  E_lo = E_min;
  E_hi = E_max;

  do
    {
      ++i;
      E_mid =(E_lo + E_hi)/2.0;              // first guess for energy eigenvale
      if (diff(E_mid)*diff(E_hi) < 0.0) E_lo = E_mid;   // the bisection algorithm
      else E_hi = E_mid;
    }while(fabs(diff(E_mid))>eps);

  cout << "Eigenvalue E = " << E_mid << endl;
  cout << "after " << i << " iterations." << endl;
  write_data(E_mid);
  return 0;
}


/*----------------------- end of main program -----------------------*/

/* function returns difference between left and right wavefunction */
double diff(double E)
{
  double psi_now, psi_last, psi_next, psi_left, psi_right;
  int i;

  psi_last = 0.0;
  psi_now = 0.00001;
  for (i=1; i<=1500; ++i)               /* left side first */
    {
      psi_next = (2*psi_now*(1 - ((5./12.)*h*h*k2(i,E)))-
                  (1. + ((1./12.)*h*h)*k2(i-1,E))*psi_last)/(1.+ ((h*h)/12.)*k2(i+1,E))
;
      psi_last = psi_now;
      psi_now = psi_next;
    }
  psi_left = psi_now;                               /* value at matching point */

  psi_last = 0.0;                                   /* reset starting conditions */
  psi_now = 0.00001;
  for (i=1; i<500; ++i)         /* now the right side */
    {
      psi_next = (2*psi_now*(1 - ((5./12.)*h*h*k2(i,E)))-
                  (1. + ((1./12.)*h*h)*k2(i-1,E))*psi_last)/(1.+ ((h*h)/12.)*k2(i+1,E))
;

      psi_last = psi_now;
      psi_now = psi_next;
    }
  psi_right = psi_now;                              /* value at matching point */
  return((psi_left - psi_right));
}

/*---------------------------------------------------------------------*/

// function returns k^2 depending on position i
double k2(int i, double E)
{
  const double units_convert = 0.4829;          // MeV^1 fm^-2
  if (i<500) return(units_convert*E);           // outside the well
  if (i>=500) return (units_convert*(E-(V0)));  // inside the well
}
/*---------------------------------------------------------------------*/

/* write data for eigenfuntion into files left.dat, right.dat */
void write_data(double E)
{
  double psi_last, psi_now, psi_next;
  int i;

  ofstream out_file_left;
  ofstream out_file_right;

  out_file_left.open("left.dat");    // save data in files
  out_file_right.open("right.dat");

  psi_last = 0.0;
  psi_now = 0.00001;
  for (i=1; i<=1500; i++)            // left side first
    {
      psi_next = (2*psi_now*(1- ((5./12.)*h*h*k2(i,E)))-
                  (1 + ((1./12.)*h*h)*k2(i-1,E))*psi_last)/(1 + ((h*h)/12.)*k2(i+1,E));

      out_file_left << (i-1000)*h << "\t" <<  psi_next/3.0 << endl;

     psi_last = psi_now;
     psi_now = psi_next;
   }

  psi_last = 0.0;                                   /* reset starting conditions */
  psi_now = 0.00001;
  for (i=1; i<500; i++)         /* now the right side */
   {
     psi_next=(2*psi_now*(1- ((5./12.)*h*h*k2(i,E)))-
               (1 + ((1./12.)*h*h)*k2(i-1,E))*psi_last)/(1 + ((h*h)/12.)*k2(i+1,E));

      out_file_right << (1000 -i)*h << "\t" <<  psi_next/3.0 << endl;

     psi_last = psi_now;
     psi_now = psi_next;
   }
  out_file_left.close();
  out_file_right.close();
  cout << "data saved in left.dat and right.dat" << endl;
}
```