PHYS 3437: Computational Methods in Physics, Assignment 2

Set January 27th due Feb 26th

NOTE: This assignment is potentially quite lengthy if you are currently developing your programming skills. If so, I strongly advise you to make a start on the assignment as soon as possible. While you have a significant amount of time to get this assignment done, I strongly recommend you don't leave it until reading week. I have posted a code to get you started on the website (see notes below).

Write a computer program (in the language of your choice, but so long as I can run it on CRUX from the unix prompt) which will find the roots of a given function. Design the program so the user has the following options:

1. root brackets
   - user can specify specific brackets between which a single root is sought, or
   - user can ask that a "global bracket finder" (pseudo-code appended) be used.
2. root-finding algorithm
   - hybrid bisection/Newton-Raphson
   - hybrid bisection/Secant
   - Müller-Brent (*i.e.*, hybrid bisection/Müller)
3. function specification
   - The user should be able to add or replace two FORTRAN functions to your program, where the first function describes the function whose roots are to be found, and the second function describes the derivative.

To get you started, the listing of a fully debugged program called `rootnr` has been added at the end of this assignment (you may download and use the copy from the website). This program performs only the hybrid bisection/Newton-Raphson method, and requires the user to enter specific brackets surrounding each root to be found. You will have to modify this program to add the global bracket finder and the other two root-finding methods. Alternatively, you may start your own program from scratch.

To put your new root-finder and global bracket finder to work, use the Müller-Brent method to find all the energy levels of an electron, $m_e = 9.1093897 \times 10^{-31}$ kg, in a potential square well of depth $V_0 = 100$ eV (1 eV $= 1.6021774 \times 10^{-19}$ J) and half-width $a = 2$ Å (2×10^{-10} m) using the transcendental expressions given in class, namely:

$$\xi \tan \xi - \sqrt{\eta^2 - \xi^2} = 0, \qquad \text{Class I, even}$$

$$\xi \cot \xi + \sqrt{\eta^2 - \xi^2} = 0, \qquad \text{Class II, odd}$$

where

$$\xi = \sqrt{2mE}\,\frac{a}{\hbar} \quad \text{and} \quad \eta = \sqrt{2mV_0}\,\frac{a}{\hbar}.$$

where $\hbar = 1.0545727 \times 10^{-34}$ Js. Report your roots accurate to eight significant figures. Thus, you must use the constants (*e.g.*, m_e, *etc.*) with all eight significant figures given, *and* your code must be double precision (all reals declared as `real*8`). Note that obtaining numerical values for ξ will allow you to compute values for E from the above expression. Note also that $0 < E < V_0 = 100$ eV, and thus this problem places natural global limits on ξ for the purposes of your global bracket finder.

To be supplied as part of your solution:

1. Send an email with your name, class and assignment number in the subject header, *i.e.* `Smith: 3437 Assignment 2` and attach your program to this email.

2. On paper, hand in a list each of the energy levels found in the square well problem (expressed in eV) to eight significant figures. Submit also a single graph with each of the four functions

   $$f(\xi) = \xi \tan \xi, \qquad g(\xi) = \xi \cot \xi, \qquad h(\xi) = \pm\sqrt{\eta^2 - \xi^2}$$

   plotted (*e.g.*, gnuplot) and identify each of the roots found by circling the relevant points where two of these functions intersect.

3. I will insert a function into your code to test your root finder on a challenging function. In your paper submission tell me the name of the function whose roots are sought in the version of your code submitted for examination (so I can modify it to link up my function).

4. If your program is not in FORTRAN, give a complete description on how to compile your program and link FORTRAN functions to your program (so I can link my test functions to it). While I have some experience of linking FORTRAN functions to other languages I will not put significant effort into this. Pleasure ensure your instructions are clear, simple, and do not require any significant effort on my part. I simply don't have time to debug, or look up anything in manuals.

A third of your grade will be based on the solution to the square well problem.

I shall use your root-finder and global bracket finder to find the multiple roots of a rather bouncy transcendental equation, and a third of your grade will be based on how well your program performs on this function. Does it find all the roots? Does it erroneously identify odd-poles (where the function leaves the graph at $\pm\infty$ and returns from $\mp\infty$) as roots? **If so, can you think of a way to modify the algorithm to distinguish between odd-poles and roots?** Do all three root-finding methods work?

The last third of your grade will be based on how well documented and structured your code is, how readable it is, how free it is from programming sloppiness (*e.g.*, mixed type

arithmetic), and how efficient it is (thus, leave the cpu timing statements from the program I gave you in your program). The program attached would get full points in this regard.

Finally, each program should be an individual effort; no "group programs" please. You may consult with each other and share ideas, but I want each program to be unique (other than `rootnr`, should you decide to start with it).

```
      program rootnr
c
c    written by: A Busy Code Developer, January 2003
c    modified 1:
c
c  PURPOSE:  This program finds the root of a given function that
c  lies between the input bounds using the hybrid Newton-Raphson/
c  bisection method.
c
c-----------------------------------------------------------------------
c
      implicit      none
c
      integer       niter
      real*8        xl      , xr      , root    , maxerr , err
      real*8        fofx    , dfdx
      real          cputot  , etime
c
      real          tdummy  (   2)
c
      external      binrph  , fofx    , dfdx
c
      data          maxerr
     1             / 5.0d-9 /
c
c-----------------------------------------------------------------------
c
c     Start up CPU time counter.
c
      cputot = etime ( tdummy )
c
c     Ask for initial limits/guesses.
c
      write(6, 2010)
      read (5,    *) xl, xr
c
c     Compute and report root, if any.  "niter" is returned as zero if
c  no root is found.  Otherwise, it is the number of iterations taken to
c  converge on the root.
c
      call binrph ( xl, xr, root, fofx, dfdx, maxerr, err, niter )
c
      if (niter .gt. 0) then
        write(6, 2020) root, err, niter
      else
        write(6, 2030) xl, xr
      endif
c
c     End CPU time counter, and report cpu time used.
```

```
c
       cputot = etime ( tdummy ) - cputot
       write(6, 2040) cputot
c
       stop
c
2010   format('ROOTNR  : Enter xL and xR such that f(xL)*f(xR) < 0.'
     1       ,'  Thus, xL < root < xR.')
2020   format('ROOTNR  : Root = ',1pg22.15,' +/- ',g9.2
     1       ,' after ',i3,' iterations.')
2030   format('ROOTNR  : No root found in domain (',1pg12.5
     1       ,', ',g12.5,').')
2040   format('ROOTNR  : Total cpu usage for this run is ',1pg12.5
     1       ,' seconds.')
c
       end
c
c=======================================================================
c
       subroutine binrph ( xl, xr, xn, f, fprime, maxerr, err, iter )
c
c    written by: A Busy Code Developer, January 2003
c    modified 1:
c
c  PURPOSE:  This routine performs the hybrid bisection/Newton-Raphson
c  method to find the root of a function.
c
c-----------------------------------------------------------------------
c
       implicit      none
c
       integer       iter    , maxiter
       real*8        xl      , xr      , xn      , xnp1    , fxl
     1             , fxr     , fxn     , dfxn    , err     , ferr
     2             , maxerr  , small
       real*8        f       , fprime
c
       data          maxiter , small
     1             / 30    , 1.0d-99 /
c
c-----------------------------------------------------------------------
c
c      Initialise variables.
c
       iter = 0
       fxl  = f(xl)
       fxr  = f(xr)
c
c      Data validation:  Make sure root lies in between the left- and
c  right-hand value.  If not, return iter=0 to the calling routine.
c
       if (fxl*fxr .gt. 0.0d0) then
         write (6, 2010) xl, xr
         return
       endif
c
c      Starting values for "xn", "fxn", and "dfxn".
c
       if (abs(fxl) .lt. abs(fxr)) then
```

```
          xn  = xl
          fxn = fxl
        else
          xn  = xr
          fxn = fxr
        endif
        dfxn = fprime(xn)
c
c-----------------------------------------------------------------------
c----- Top of hybrid loop. ---------------------------------------------
c-----------------------------------------------------------------------
c
10      continue
          iter = iter + 1
c
c       Evaluate the next Newton-Raphson guess.
c
          xnp1 = xn - fxn / ( dfxn + small )
c
c       If NR step is not justified, take a bisection step instead.
c
          if ( (xl .lt. xnp1) .and. (xnp1 .lt. xr) ) then
            write (6, 2020) iter, xnp1
          else
            xnp1 = 0.5d0 * ( xl + xr )
            write (6, 2030) iter, xnp1
          endif
c
c       Evaluate error estimate and set new guess.
c
          err = abs ( xnp1 - xn )
          xn  = xnp1
c
c       If the fractional error of "xn" is less than "maxerr", the root
c  has been found---execution is returned to calling routine.
c
          ferr = err / ( abs (xn) + small )
          if ( ferr .le. maxerr ) go to 20
c
c       If the maximum number of iterations has been reached, abort.
c
          if ( iter .ge. maxiter ) then
            write (6, 2040) maxiter, xn, err
            stop
          endif
c
c       Prepare for next iteration.
c
          fxn  = f     (xn)
          dfxn = fprime(xn)
c
c       Determine which half of the interval (xl,xr) the root is in, and
c  reset either xr or xl as appropriate.
c
          if ( fxl * fxn .le. 0.0d0 ) then
            xr  = xn
            fxr = fxn
          else
            xl  = xn
```

```
           fxl = fxn
         endif
         go to 10
20     continue
c
c-----------------------------------------------------------------------
c----- Bottom of hybrid loop. ------------------------------------------
c-----------------------------------------------------------------------
c
2010   format('BINRPH  : No unique root lies between the specified'
     1       ,' limits:',/
     2       ,'BINRPH  : xl = ', 1pg12.5,', and xr = ',g12.5)
2020   format('BINRPH  : Iteration: ',i3,' - NEWTON/RAPHSON: root ='
     1       ,1pg22.15)
2030   format('BINRPH  : Iteration: ',i3,' - BISECTION     : root ='
     1       ,1pg22.15)
2040   format('BINRPH  : Maximum number of iterations ',i9
     1       ,' exceeded.',/
     2       ,'BINRPH  : Best guess at root is ',1pg22.15
     3       ,' +/- ',g9.2,'.  Abort!')
c
       return
       end
c
c=======================================================================
c
       function fofx ( x )
c
c    written by: A Busy Code Developer, January 2003
c    modified 1:
c
c  PURPOSE:  This routine returns the value of cos(x) - x.
c
c-----------------------------------------------------------------------
c
       implicit      none
c
       real*8        x       , fofx
c
c-----------------------------------------------------------------------
c
       fofx = cos(x) - x
c
       return
       end
c
c=======================================================================
c
       function dfdx ( x )
c
c    written by: A Busy Code Developer, January 2003
c    modified 1:
c
c  PURPOSE:  This routine returns the value of the derivative of fofx
c  evaluated at x.
c
c-----------------------------------------------------------------------
c
       implicit      none
```

```
c
      real*8        x       , dfdx
c
c-----------------------------------------------------------------------
c
      dfdx =-sin(x) - 1.0d0
c
      return
      end
```

Pseudo-Code for the Global Bracket Finder

All variables (`xmax, xmin, n,` *etc.*) have the same meaning as used in class.

`dx = (xmax - xmin) / n` (Set the initial value of `dx`.)
`dxmin = 0.01 * dx` (`dx` will not be allowed to fall below `dxmin`.)

(The parameter `n` is the dimension of the arrays `x` and `f` and is chosen large enough so that important features (roots and extrema) of $f(x)$ are no closer together than `dx=(xmax-xmin)/n`. However, without *a priori* knowledge of where these features are, we cannot know what optimal value of `n` to use. So we just pick a big value (*e.g.*, 10,000) and hope for the best.

With `n` chosen sufficiently large, the initial value of `dx` will be much smaller than needed, and the algorithm will increase `dx` quickly and safely to a more optimal value. Conversely, should features of the function become closer together than `dx` can resolve, the algorithm will decrease `dx` with the *proviso* that `dx` shall not be allowed to fall below some preset minimum value (*e.g.*, `dxmin` = 1% of the initial value of `dx`). Without such a safeguard, the diminution of `dx` could proceed indefinitely. Should `dx` ever be set to `dxmin`, a warning message should be issued to the user as this is indicative that the algorithm is having difficulty resolving the function, or that `n` needs to be larger. If `n` is already considered too large, further analysis of the function and/or algorithm may be needed.)

`i = 1` (start index for `x` and `f` at 1.)
`x(i) = xmin`
`f(i) = fofx(xmin)` (function call)
`xleft = x(i)`
`fleft = f(i)`

1

`i = i + 1` (increment index `i` by 1.)
escape if `i > n`
`dx = min ( dx, xmax - xleft )` (`xleft+dx` mustn't be bigger than `xmax`.)
`xright = xleft + dx`
`fright = fofx(xright)` (function call)

```
scale = max (1.0, |xleft|) / max (|fleft|, |fright|)
```

(The scaling factor `scale` ensures a unitless derivative. The numerator was derived from trial and error, and minimises unnecessary function calls near `x=0`. Dividing by `max (|fleft|, |fright|)` instead of `fleft` alone prevents the denominator from blowing up should `xleft` happen to be very close to a root.)

```
dfdx = scale * (fright - fleft) / dx                (scale renders dfdx unitless.)
dldx = sign ( 1.0, dfdx ) * sqrt ( 1.0 + dfdx**2 )
```

(The `sign` function transfers the sign of `dfdx` to `dldx`, and is necessary to ensure sensitivity of the algorithm to local extrema, where the sign of the derivative changes.)

2 (Now, halve `dx`, evaluate `dldxh`, and compare it to `dldx`.)

```
dxh = 0.5 * dx
xmid = xleft + dxh
fmid = fofx(xmid)                                                    (function call)
dfdxh = scale * (fmid - fleft) / dxh          (use the same value scale as before.)
dldxh = sign ( 1.0, dfdxh ) * sqrt ( 1.0 + dfdxh**2 )
```

Perform tests

if `dldx` and `dldxh` are within 10% of each other, then
`dx = 1.5 * dx` (increase `dx` for use next step)
`goto` **3**

if `dldx` and `dldxh` are between 10% and 50% of each other, then `dx` is fine as it is.
`goto` **3**

if `dldx` and `dldxh` differ by more than 50% and `dxh < dxmin`, `dx` is set to `dxmin` and a warning message is issued.
dx = dxmin
xright = xleft + dx
fright = fofx(xright) (function call)
issue warning message
goto **3**

if `dldx` and `dldxh` differ by more than 50% and `dxh > dxmin`, `dx` is halved and we try again.
dx = dxh
xright = xmid
fright = fmid
dldx = dldxh
goto **2**

3 For execution to have arrived here means that `dx` is a suitable step to take. Store present

values of `xright` and `fright` in `x(i)` and `f(i)` respectively, and then return to line **1** to find the next step `dx`.

```
x(i) = xright
f(i) = fright
if (xright .ge. xmax) done.
else
    xleft = xright
    fleft = fright
    goto 1
```