Root Finding and Special Functions

HW 3: Wednesday, Sep. 12, 2018
DUE: Wednesday, Sep. 19, 2018
READ: Numerical Recipes, Root Finding: Section 9.0-9.4 pages 442-456 (Section 9.3 is optional)
Special Functions: section 6.0, 6.5 page 255, 274-283
Horner's Rule: section 5.0,5.1, page 201-206
OPTIONAL: Landau, Paez, and Bordeianu, root finding, section 7.9, 7.10

PROBLEM (10 points):

No programming language has all of the special functions found in engineering or physics (such as the Bessel functions below). *Numerical Recipes*[2] has a listing of the code (in chap. 6) for many special functions of interest and the book *Handbook of Mathematical Functions*, edited by M. Abraomowitz and A. Stegun [1] contains many other useful numerical approximation to many special functions of interest (it is on permanent reserve in the library and several on-line versions are listed on the course web site if you need other functions). The code for many useful functions (as well as many other numerical subroutines) can also be downloaded from `www.netlib.org` in machine readable form (has mainly fortran but most C/C++ versions are there if you dig a little) or the GNU Scientific Library or GSL (`www.gnu.org/software/gsl/`, which is all in C, but a little hard to compile).

$J_n(x)$, and $Y_n(x)$ are Bessel functions of the first and second kinds (as described in section 6.5 of *Numerical Recipes*[2]). *Numerical Recipes* has software functions that calculate these special functions good to about 5 or 6 significant figures. You may use the source code for these functions in Bessjy() (and the functions called by these functions) given in *Numerical Recipes*. You can get the missing coefficients electronically from www.nr.com/webnotes in the appropriate section, and cut/paste from the pdf file or download the whole code (recommended). This edition of *Numerical Recipes* seems to be writing most code as a 'struct' instead of a 'class', but you can use it in a similar way. Declare one instance of the Bessjy() struct and then access the member functions j0() and j1() etc. There is an example of using a struct this way on pages 18-19 of *Numerical Recipes*[2]. You will need to download (from http://www.nr.com/routines/instbyfile.html using the Cornell site license) and include the nr3.h header and bessel.h files as:

```
#include "nr3.h"
#include "bessel.h"

Bessjy myBessel;  // make an instance of the Bessel function

int main()
{   :
    y = myBessel.j0(x);
    :
}
```

(If you use the Num. Rec. code verbatim, you do not need to list it in your homework, just cite

Num. Rec.) Alternately you could just ignore the struct and write them as simple functions. You can declare the Num. Rec. types Int and Doub using a typedef declaration (section 1.4.1 of NR) as:

```
typedef double Doub;
typedef int Int;
```

First test the functions by plotting $J_0(x)$, $J_1(x)$ in the range $0.0 < x < 20$ and $Y_0(x)$, $Y_1(x)$ in the range $0.75 < x < 20$ as in Figure 6.5.1 (but over twice the range) (i.e. print numbers in a file with C++ and plot with a graphing program).

Next, test two of the root finding methods discussed in class (bisection, and the false position method) to find the first five smallest positive values of x for $x > 0$, that satisfy:

$$J_0(x)J_1(x) \quad = \quad x^2 Y_0(x) Y_1^2(x)$$

good to about 5 or 6 significant figures. You should write your own root finding code as a general purpose subroutine using either a function pointer or function template (see code segment below). Compare the efficiency of bisection and the false position method (aka Reguli Falsi) by counting the number of function evaluations (a global variable with the count may be useful) to find these roots to a tolerance of about 10^{-7} (starting both methods from the same bracket).

OPTIONAL: Some compilers now include Bessel functions as a language extension. These are not yet standard and can vary in name and behavior from one compiler to another. The non-standard definitions may cause trouble (your code will not be portable) so it is still useful to have a portable version (as in Numerical Recipes). For this homework use the routine in Numerical Recipes. If you are interested you may compare them to the builtin versions (_j0() and _j1() in gcc).

References

[1] M. Abramowitz and I. A. Stegun, editors, *Handbook of Mathematical Functions*, National Bureau of Standards, 1964, and Dover 1965.

[2] W. H. Press, S. A, Teukolsky, W. T. Vetterling and B. P. Flannery *Numerical Recipes, The Art of Scientific Computing, 3rd edit.*, Camb. Univ. Press 2007.

In algorithmic form bisection is:
Assume that one root has been bracketed inside $x_1 < x_0 < x_2$

1. evaluate $f_1 = f(x_1)$, and $f_2 = f(x_2)$
2. check that f_1 and f_2 have opposite signs (exit if not)

repeat

3. Evaluate $f_3 = f(x_3)$ at $x_3 = \frac{1}{2}(x_1 + x_2)$
4. If f_3 has same sign as f_1 then $(x_1, f_1) \leftarrow (x_3, f_3)$
5. Else if f_3 has same sign as f_2 then $(x_2, f_2) \leftarrow (x_3, f_3)$

until $|f_3| < \epsilon$
x_3 is the root

In algorithmic form the secant method is:
Given two initial points x_1 and x_2 in a region near the root (bracketing is not necessary)

repeat

1. solve for x_3 to give $f(x_3) = 0$ at $x_3 = x_1 - f_1 \frac{x_2 - x_1}{f_2 - f_1}$
2. throw away the oldest $(x, f) = (x_1, f_1)$ (1st iteration is arbitrary)
   $(x_1, f_1) \leftarrow (x_2, f_2)$ and $(x_2, f_2) \leftarrow (x_3, f_3)$

until $|f_3| < \epsilon$ = specified tolerance (input)
x_3 is the root

In algorithmic form the False Position method is:
Given an initial bracket of the root x1:x2 (i.e. a root is between these two points):

1. evaluate $f_1 = f(x_1)$, and $f_2 = f(x_2)$
2. check that f_1 and f_2 have opposite signs (exit if not)

repeat

3. solve for x_3 to give $f(x_3) = 0$ at $x_3 = x_1 - f_1 \frac{x_2 - x_1}{f_2 - f_1}$
4. If f_3 has same sign as f_1 then $(x_1, f_1) \leftarrow (x_3, f_3)$
5. Else if f_3 has same sign as f_2 then $(x_2, f_2) \leftarrow (x_3, f_3)$

until $|f_3| < \epsilon$ = specified tolerance (input)
x_3 is the root

The following code is a method of passing the function name as an argument (function pointer) to bisect() so that the function can be called many times with different functions:

```
double bisect( double(*f)(double), double x1, double x2, double tol )
{  :
   f1 = f(x);    //  call the function specified in main() at run time
   :
   do{
        :
   } while( (...) && (...)... );
   :
   return(...);
}

double myfa( double x ){.....}  //  function a
double myfb( double x ){.....}  //  function b

int main()
{   :
    y = bisect( myfa, .... );    //  bisect myfa()
    y2 = bisect( myfb, .... );   //  bisect myfb()
    :
}
```

You can also use a template to do the same thing a little more efficiently because the link is resolved once at compile time instead of many times while running (type T should be float or double):

```
template<class T, T (*f)(T) >
T bisect( T xa, T xb, T tol, int maxiter )
{  :
   :
   f1 = f(x);    //  call the function specified in main() at compile time
   :
   do{
        :
   } while( (...) && (...)... );
   :
   return(...);
}  // end bisect()

double myfa( double x ){.....}  //  function a
float myfb( float x ){.....}  //  function b

int main()
{   :
    y = bisect<double, myfa>( .... );    //  bisect myfa()
    y2 = bisect<float, myfb>( .... );   //  bisect myfb()
    :
}
```