Numerical solution of radial equations

Before you start coding

Before you start coding, here are some guidelines you want to follow.

- Read `c_tutorials.pdf`.
- Think modular, code modular.
  Break up your program into many small pieces. Think of each small program as a "word" and the final long program as "sentences" and "paragraphs".
- Organize your code as if you are writing a book.
  The `main` function should look like the list of Chapters. It should just say something like "Initialize", "Evolve", "WriteResults". It should not actually do anything. In the next layer, the function should be the list of Sections. For instance, "Initialize" might contain "ReadIn_Params", "AllocateMemories". In small size programs, the functions in the next layer might start to actually do something. In bigger programs, even this layer would be lists of subsections. Subsection level functions might actually start to do something.
- Give your functions descriptive names.
  A good example is Mathematica function names. Your program should read almost like a paragraph.
- Functions in the same file should be closely related and self-contained as much as possible. The names of the functions in the same file should share the same prefix.
- Put as many comments as possible.
  For instance, if you used a reference for an algorithm (no matter whether you are using actual code pieces from it or just using the algorithm idea), you *must* put that in as a comment. It's not only an ethical practice, it is also a useful reminder to yourself and others for later.
  Even a simple thing like putting a comment after a for-loop bracket, `for(i=0;...){ ... } // i-loop`, helps greatly when you are hunting for a subtle bug.

- Break up your program into clear structures and collect them in different files.
  For instance, the initialization functions and sub-routines should be separated from the implementation of physics calculations. If you are familiar with `C++`, think Classes.

- Make your functions as general and independent as possible.
  If a function is useful in one place of the code, it may be useful in many other places. So make it as general and independent as possible so that it can be used in other places in the same code and also in other projects. For instance, if you need a numerical integration program, code it once, test it thoroughly and put it in a separate file. You can then just use it with confidence whenever you need it. Good examples are the mathematical functions already defined in `math.h`. You don't need to know how `sin(x)` is calculated. You just use it since someone tested it so thoroughly already that you can just trust it to do the right thing.

- Test each of your functions as thoroughly as possible before using it.

- A function should be *short and readable.*
  A rule of thumb: If you can't see all of its parts on one screen, it's too long. A function should do one thing and one thing only and *do it well.* This is the UNIX/C philosophy and it has stood the test of time.

- Aggregate your data.
  If you need to pass many pieces of data to a function, it is a much better practice to define a C++ Class or a C `struct` to aggregate them and pass that to the function. In that way, you can avoid nasty surprises such as accidentally swapping places of two parameters.

- Avoid using common variables as much as possible.
  In general, don't use them. However, sometimes one must use it. For instance, if you want to carry out a 2-D integral with a 1-D numerical integration function, then you need to use a common variable. But even in that case, the scope of the common variable should be restricted to a single file. A possible exception for the scope restriction would be the set of static parameters that *does not and should not change* during the calculations.

- Do not put implementations in your header files.
  This is just a silly practice. It does not save time nor make the program faster. It actually makes debugging very much more complicated. Your header files should only contain function and structure (or Class) declarations.

1 Introduction

In this project, we are aiming to numerically solve the radial part of the Schrödinger equation

$$\left(-\frac{1}{2\mu}\frac{d^2}{dr^2}+\frac{\ell(\ell+1)}{2\mu r^2}+V(r)\right)u_{n_r\ell}(r)=E_{n_r\ell}u_{n_r\ell}(r) \tag{1}$$

for an arbtrary confining potential $V(r)$. In particular, we would like to obtain the bound state wavefunctions and the energy eigenvalues $E_{n_r\ell}$. Note that we have set $\hbar=1$ and $c=1$. We will supply the necessary factors of $\hbar$ and c at the end using

$$\hbar c = 197.3\,\mathrm{eV}\cdot\mathrm{nm} = 1973\,\mathrm{eV}\,\mathring{A} \tag{2}$$

Since

$$u_{n_r\ell}(r)=rR_{n_r\ell}(r) \tag{3}$$

the boundary conditions for bound state solutions are

$$\begin{aligned}\lim_{r\to 0}u_{n_r\ell}(r)&=0\\ \lim_{r\to\infty}u_{n_r\ell}(r)&=0\end{aligned} \tag{4}$$

We will assume that as $r\to 0$, the potential $V(r)$ is no more singular than $1/r$.

2 Matrix method

There are no "easy" way to solve the radial equation numerically. That is, if one wants to implement all necessary numerical methods. Modern computing environment such as Jupyter, NumPy, and GNU Scientific Library, however, can certainly make the user's job easier by providing many of the necessary numerical methods as packages. One such example is the matrix method.

Suppose we discretize r such that

$$r_n = r_0 + nh \tag{5}$$

where $r_0 = 0$ and $h = \frac{r_N - r_0}{N}$ is the size of the interval. The integer n ranges from 0 to N so that the initial radial position is r_0 and the final radial position is r_N. For sufficiently small h, we can approximate the second derivative as

$$\frac{d^2u(r_n)}{dr^2} = \frac{u_{n+1} + u_{n-1} - 2u_n}{h^2} + O(h^4) \tag{6}$$

where we introduced a short hand $u_n = u(r_n)$. The radial equation then becomes

$$-\frac{1}{2\mu}\frac{u_{n+1} + u_{n-1} - 2u_n}{h^2} + \frac{\ell(\ell+1)}{2\mu r_n^2} + V_n u_n = E u_n \tag{7}$$

which can be re-formulated as an eigenvalue problem

$$MU = EU \tag{8}$$

where

$$U = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_N \end{pmatrix} \tag{9}$$

and

$$M = \begin{pmatrix} c_0 & -d & 0 & 0 & 0 & \cdots & 0 \\ -d & c_1 & -d & 0 & 0 & \cdots & 0 \\ 0 & -d & c_2 & -d & 0 & \cdots & 0 \\ \vdots & \ddots & & & & & \vdots \\ 0 & 0 & \cdots & 0 & -d & c_{N-1} & -d \\ 0 & 0 & \cdots & & 0 & -d & c_N \end{pmatrix} \tag{10}$$

is a tridigonal matrix with

$$d = \frac{1}{2\mu h^2} \tag{11}$$

and

$$c_n = \frac{1}{\mu h^2} + \frac{\ell(\ell+1)}{2\mu r_n^2} + V(r_n) \tag{12}$$

All one has to do now is to construct this matrix, enforce suitable boundary conditions on U and M, and ask NumPy (for instance) to calculate the first few eigenvalues and eigenvectors. That's it. You've done it.

If you want to try this out, Solve only for $(u_1, \cdots, n_{N-1})$. This is because the boundary conditions set $u_0 = 0$ and $u_N \approx 0$. Also, because of the centrifugal term (and also $V(r)$ if it is singular at $r = 0$), $c_0 = \infty$.

If my goal of assigning this project to you was for you to calculate a few low-lying eigenvalues and eigenvectors (eigenfunctions) quickly and efficiently, there wouldn't be much more to the project than this. Especially since modern software and hardware can easily take on the task of diagonalizing 1000×1000 matrix.

My goal here, however, is not that. What I want you to learn is how to turn equations and ideas to algorithms on your own. As such, we are going to take a longer path and construct all we need ourselves. Also, we are not going to do this with the matrix method because there won't be any time for topics other than solving large matrix eigenvalue/vector problems. That itself is a vast and fascinating subject with countless applications. (Standard reference: `https://epubs.siam.org/doi/book/10.1137/1.9781421407944`). However, I would like you to be acquaited with broader topics.

3 The Matching Method

To put Eq.(1) on a computer, it is convenient to scale the variables so that we deal with only dimensionless variables. If we know an approximate size of the energy eigenvalues, $E_a > 0$, then the best way to scale the radial coordinate r is

$$x = kr \tag{13}$$

where $k = \sqrt{2\mu E_a}$. The resulting equation is

$$\left(-\frac{d^2}{dx^2} + \frac{\ell(\ell+1)}{x^2} + \tilde{V}(x) + \tilde{E}_{n_r\ell} \right) \tilde{u}_{n_r\ell}(x) = 0 \tag{14}$$

where $\tilde{u}_{n_r\ell}(x) = u_{n_r\ell}(x/k)$, $\tilde{V}(x) = V(x/k)/E_a$ and $\tilde{E}_{n_r\ell} = |E_{n_r\ell}|/E_a$. Here we used the fact that for a bound state, $E_{n_r\ell} = -|E_{n_r\ell}|$. The boundary conditions for bound state solutions are

$$\begin{aligned} \lim_{x\to 0} \tilde{u}_{n_r\ell}(x) &= 0 \\ \lim_{x\to\infty} \tilde{u}_{n_r\ell}(x) &= 0 \end{aligned} \tag{15}$$

The strategy to solve for a bound state is as follows. This relies on the fact that the wavefunction and its derivative must be continuous everywhere and they should satisfy the boundary condition above.

1. Choose a value for $|E_{n_r\ell}|$. Usually a value close to E_a. That means choose $\tilde{E}_{n_r\ell} \sim 1$.

2. Solve the differential equation forward from $x = 0$. Call this solution $\tilde{u}_I(x)$. The boundary condition is $\tilde{u}_I(0) = 0$.

3. Solve the differential equation backward from $x = x_{\max}$. Call this solution $\tilde{u}_{II}(x)$. The boundary condition is $\tilde{u}_{II}(x_{\max}) = \epsilon$ where $x_{\max}$ is sufficiently large and ϵ is sufficiently small.

4. At some chosen point x_b, compare $u'_I(x_b)/u_I(x_b)$ and $u'_{II}(x_b)/u_{II}(x_b)$. If they are not the same, choose a different $\tilde{E}_{n_r,\ell}$ and go back to 2 and repeat the process until convergence is achieved within a set tolerance.

The idea for this "Matching method" is simple enough. Implementation not so.

Determining E_a
First, let's see how we can determine a typical energy scale E_a. For this, of course, one needs to know how to estimate the typical energy scale of the Hamiltonian

$$H = \frac{p_r^2}{2\mu} + \frac{\ell(\ell+1)}{2\mu r^2} + V(r) \tag{16}$$

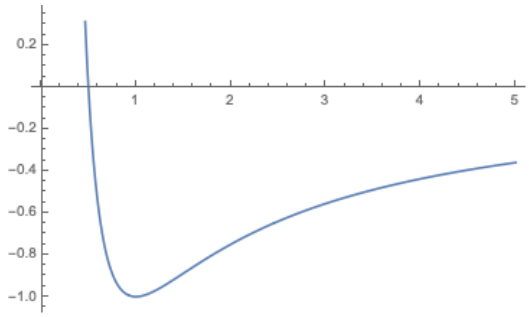
Let's consider the ground state. For this, we can set $\ell = 0$. Suppose that the typical length scale associated with the bound ground state is r_0. Using the uncertainty principle $\Delta r \Delta p_r \sim r_0 p_r \sim 1$, we can estimate the ground state energy to be

$$E(r_0) = \frac{1}{2\mu r_0^2} + V(r_0) \tag{17}$$

We can then solve for r_0 by minimizing $E(r_0)$

$$\frac{dE}{dr_0} = -\frac{1}{\mu r_0^3} + \frac{dV}{dr_0} = 0 \tag{18}$$

To be concrete, let's take a Coulomb-like potential $V(r) = -\alpha/r$. $E(r)$ roughly has the following shape (arbitrary unit)



Solving

$$E'(r_0) = -\frac{1}{\mu r_0^3} + \frac{\alpha}{r_0^2} = 0 \tag{19}$$

gives

$$r_0 = \frac{1}{\mu\alpha} \tag{20}$$

and

$$E(r_0) = \frac{1}{\mu}\frac{\mu^2\alpha^2}{2} - \alpha\mu\alpha = -\frac{\mu\alpha^2}{2} \tag{21}$$

This should be a good estimate. For example, if we are really solving the Hydrogen atom problem, the ground state energy is known to be

$$E_{\text{hydrogen}} = -\frac{\mu\alpha_{\text{EM}}^2}{2} \tag{22}$$

For other more complicated potentials, we need a method to solve Eq.(18) numerically. Let's make that utility next.

> In practice, do we *really* need to do this for every potential? No, not really. This is because physical systems have typical length scales. For instance, if you are studying atomic or molecular systems, you know that the Bohr radius (r_0 above. About 0.5 Å) is the basic length scale of the system. Finding the minimum of a given potential has other uses, but as far as the typical length scale is concerned, we already know it. If one knows the length scale, one also knows the momentum scale $p \sim 1/r_0$,

and the energy scale $\sim p^2/m$.

Of course, the actual size of the system can vary between an hydrogen atom and, say, a large protein molecule. The difference would be up to a factor of $O(1000)$. That is a big range, but not big enough for us to worry about loss of numerical accuracies. For nuclear physics, the relevant length scale is $\sim 1\,\mathrm{fm}$ which is roughly the size of a proton. Again, there is no real need to calculate the minimum of a given nuclear potential if all one needs is the typical length scale. One can, of course, always optimize using more sophisticated estimates.

So we are doing it this way not because it is strictly necessary to find the minimum of a realistic potential, but because it is a good way to introduce numerical methods for taking a derivative and solving general nonlinear equations.

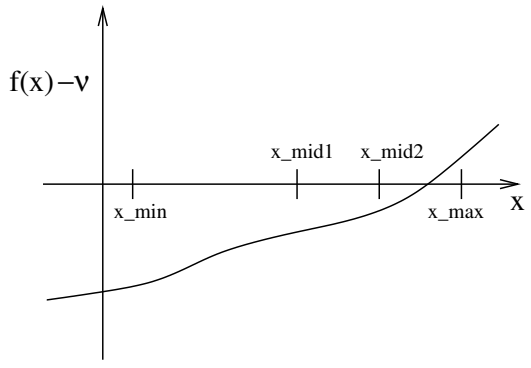
4 Solving $f(x) = \nu$

Read `c_tutorials.pdf` first. In what follows, I will assume that you have at least read `c_tutorials.pdf` and know where to look up things.

In this section, we will implement two methods for solving $f(x) = \nu$. These methods are the bisect search method and Newton's method.

Bisect Search

We would like to solve $f(x) = \nu$ within the range $x_{\min} \leq x \leq x_{\max}$.



The main idea is expresssed in the figure above.

1. Evaluate $f_{\min} = f(x_{\min}) - \nu$ and $f_{\max} = f(x_{\max}) - \nu$. If the solution x is between $x_{\min}$ and $x_{\max}$, then one of them should be positive and the other negative. That is, $f_{\min} f_{\max} < 0$.

   If $f_{\min} f_{\max} > 0$, then there may not be a solution within that range. In that case, choose a different range.

2. Evaluate $f_{\text{mid}} = f(x_{\text{mid}}) - \nu$ where $x_{\text{mid}} = (x_{\min} + x_{\max})/2$. Compare the sign of f_{mid} with that of $f_{\min}$ and $f_{\max}$.

   If $f_{\min} f_{\text{mid}} < 0$, then set new $x_{\max} = x_{\text{mid}}$.

   If $f_{\text{mid}} f_{\max} < 0$, then set new $x_{\min} = x_{\text{mid}}$.

   Keep doing so until $|f_{\text{mid}}/\nu| < \varepsilon$ (or $|f_{\text{mid}}| < \varepsilon$ if $\nu = 0$) where ε is a small enough number. When this is achieved, x_{mid} at that step is the numerical solution.

Newton's Method

Another popular method for solving $f(x) = \nu$ is Newton's method. It can be derived from the following approximation of the derivative

$$f'(x_n) \approx \frac{f(x_{n+1}) - f(x_n)}{x_{n+1} - x_n} \tag{23}$$

Rearranging, we get

$$x_{n+1} \approx x_n + \frac{f(x_{n+1}) - f(x_n)}{f'(x_n)} \tag{24}$$

What we would like to do is to use this expression as an iteration method. For that, we simply set $f(x_{n+1}) = \nu$ and get Newton's method

$$x_{n+1} = x_n + \frac{\nu - f(x_n)}{f'(x_n)} \tag{25}$$

If the function is monotonic, this method is guaranteed to get you the solution.

So here is the idea of Newton's method:

1. Start with x_0. If you know something about the function, use it to choose x_0. If you do not, try few different values.

2. Calculate $f(x_0)$ and $f'(x_0)$.

3. Get x_1:

$$x_1 = x_0 + \frac{\nu - f(x_0)}{f'(x_0)} \tag{26}$$

4. Calculate $f(x_1)$ and $f'(x_1)$ and get x_2:

$$x_2 = x_1 + \frac{\nu - f(x_1)}{f'(x_1)} \tag{27}$$

5. Keep going until $|\nu - f_{n+1}|/|\nu| < \varepsilon$ if $\nu \neq 0$ and $|f_{n+1}| < \varepsilon$ if $\nu = 0$. If this condition is met, then f_{n+1} is the numerical solution.

Note that to use the bisect method, one must specify the range $(x_{\min}, x_{\max})$ while to use Newton's method, one only need to specify the initial starting point x_0. No numerical method can claim to be the true "one method for all". There will be circumstance where one method can succeed while others fail. For instance, `Solve_Newton` won't really be able to solve $\sin(x) = 1$ as it is. The solution will actually converge to $\pi/2$ very closely but when it actually hits $\pi/2$, $f'(x) = \cos(\pi/2) = 0$ which causes the $(f(x) - c)/f'(x)$ term to diverge. It is always good to have multiple methods under one's arsenal.

Implementation

Here is a C-like code. Your task is to fill in the detail and test the function. You can choose to use any computer langauge, but I would encourage you to learn C or C++. The parts which you should fill in are marked with TD (To do) to distinguish them from just comments.

All C programing starts with a header file. In our case, the header file should look like

```
// File solve.h
#ifndef SOLVE_H // if SOLVE_H is not defined
#define SOLVE_H // define SOLVE_H
        // All header files should start like this.
        // This is to avoid including the same header file twice.
        // If you do include this header file multiple times,
```

```
        // then SOLVE_H will be defined when it was first seen by the compiler.
        // When the compiler encounters the header file the next time,
        // SOLVE_H is already defined. Hence, the content will
        // be bypassed by the ifndef/define/endif (at the end of the file)
        // statements.

// Functions in the same file should all share the same prefix.
// There are many reasons for this. The most prominent one is that
// you instantly know where the function is when the function is
// used in another file.

// Solve_Bisect solves f(x) = nu using the bisect search method
double Solve_Bisect
(double nu, double (*func)(double), double x_min, double x_max, double tol,
 int *count);

// double (*func)(double) is C's way of passing a function to another
// function. See how it is used in main_solve.c below.

// Solve_Newton solves f(x) = nu using Newton's method
double Solve_Newton
(double nu, double (*func)(double), double x_0, double tol, int *count);

#endif // end if
```

The two methods are implemented in `solve.c`:

```
// File: solve.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "solve.h"

// Numerical derivative
// This function is visible only within this file.
double Solve_Get_Df(double (*func)(double), double x);
//
// We could implement this numerical derivative as a general utility function
// and in fact we will do that shortly. We are including this here to make
```

```
// Solve function self-contained. But that is not strictly necessry.

//
// Solve_Bisect
//
// Solve f(x) = nu using bisect method
// Note that count is passed by reference (that is, the pointer is passed)
// because we want to change the value of it inside Solve_Bisect

double Solve_Bisect
(double nu, double (*func)(double), double x_min, double x_max, double tol,
 int *count)
{
 double x_mid, f_max, f_min, f_mid, err;
 int count_max;

 count_max = 1000; // Large enough.

 *count += 1; // Keep track of the number of iterations.
              // Call this function with count = 0 and
              // add 1 whenever BisectSolve is called.
              // Note that "count" is declared as the pointer
              // to a double in the function definition.
              // This practice is called "pass by reference".
              // That is, you pass to the function
              // the address of the variable, not the value.
              // To access the memory space "count" is pointing to
              // you use the "dereferncing operator" *.
              // So *count += 1 above means that
              // "go to the memory space count is pointing to
              // and add 1 to the value there".
              // In this way, this function changes the value
              // held by the variable "count" outside the function.

              // In C, when you pass a variable to a function
              // as f(..., double v, ...),
              // then the function only knows the value of v,
              // but not where it is located in the memory space.
```

```
                // So "v" inside f is not the same as "v" ouside.
                // It just has the same name for convenience.
                // Hence, if you do something to v inside f, that change
                // DOES NOT propagate to outside f.
                // This is called "pass by value"
                // If you want the variable value to be changed by f, then
                // you need to do "pass by reference".
                // This is declared as f(.., double *v, ...).
                // That is, you pass the pointer. Then the function f
                // knows where the variable a is actually located
                // in the memory space and can change its value.

                // This is an imporant concept in C:
                // If you have the pointer, you can access
                // the actual memory space where variable is stored.
                // This is useful and dangerous at the same time.
                // So be careful.
                // See tutorials.pdf for more.

// We are keeping track of count to avoid an infinite loop.
// If count is too big, then we may be in danger of
// getting into an infinite loop. In that case, warn and exit
if(*count > count_max)
 {
  fprintf
  (stderr, "Solve_Bisect: Done %d iterations without convergence.\n",
   count_max);
  fprintf(stderr, "Exiting.\n");
  exit(0);
 }

// Calculate f_max = f(x_max) - nu
// Below is how one accesss the function passed to this function
// via the function pointer (*func)
// When you call this function, you just need to pass the name
// of the function (of the right type) you want to consider.
// See main_solve.c below how this is done.
f_max = (*func)(x_max) - nu;
```

```
// TD: Calculate f_min = f(x_min) - nu

if(f_max*f_min > 0.0) // we can't find a solution within the range
 {
  // TD: Warn and exit
 }

// TD: Calculate x_mid = (x_min + x_max)/2.0
// TD: Calculate f_mid = f(x_mid) - nu

// Calculate the error
if(nu != 0.0) {err = fabs(f_mid/nu);}
else {err = fabs(f_mid);}

// If err < tol, we have a solution and the calculation ends.
if(err < tol) { return x_mid; }

if(f_mid*f_max < 0.0) // the solution is between x_mid and x_max
 {
  // Call Solve_Bisect with the range (x_mid, x_max)
  // This method of calling a function within the function itself is
  // called recursion. Most modern computer language allows it.
  // However, you need to make sure that recursion will terminate
  // at a finite step. Otherwise, it can go into an infinite loop.
  // That's why we have the if(*count > count_max) ... step above.
  //
  return Solve_Bisect(nu, func, x_mid, x_max, tol, count);
 }
else if(f_min*f_mid < 0.0) // the solution is between x_min and x_mid
 {
  // TD: Call Solve_Bisect with the range (x_min, x_mid)
  // TD: That is, return Solve_Bisect with x_max replaced by x_mid
 }
else // one of the factors is zero
 {
  // TD: if f_mid = 0.0, then return x_mid
  // TD: else if f_max = 0.0,  then return x_max
```

```
   // TD: else return x_min
  }
}// Solve_Bisect


//
// Solve_Newton
//
// solves nu = func(x) by newton's method
// using x_{n+1} = x_n + (nu -f(x_n))/f'(x_n)

// First, implement numerical derivative.
// This uses f'(x) = (f(x+h) - f(x-h))/(2h) + O(h^2).
// This is given to show you how to access the function passed
// in the argument
double Solve_Get_Df(double (*func)(double), double x_old)
{
 double h, df;

   if(x_old != 0.0) { h = x_old*1.0E-5; }
   else {h = 1.0E-5; }

   // This is how one accesss the function passed to this function
   // via the function pointer (*func)
   df = (*func)(x_old+h) - (*func)(x_old-h);
   df /= 2.0*h;

 return df;
}//Solve_Get_Df


// Newton's method
// x_0 is the starting point
double Solve_Newton
(double nu, double (*func)(double), double x_0, double tol, int *count)
{
 double x_old, x_new, err, df, h;
 int count_max;
```

```
 count_max = 1000;
 x_old = x_0; // Initial value
 do {  // Do the following while the condition in the while(...)
       // below is satisfied
   df = Solve_Get_Df(func, x_old); // Get the derivative

   if(fabs(df) < tol) // Derivative is too small
    {
     // TD: Warn and exit
    }

   // TD: Implement x_new = x_old + (nu - f(x_old))/f'(x_old);
   // TD: Calculate err = |((x_new-x_old)/x_old)|;
   // The absolute value function for a double is fabs(x).
   // The function abs(x) is for integers only.

   // TD: Set x_old = x_new
   // TD: Add 1 to *count

   // This is how you access a variable that is passed by reference.
   if(*count == count_max) // Too many iterations
    {
     // TD: Warn and exit
    }
  } while(err > tol); // while this condition is satisfied

 return x_new;
}// Solve_Newton
```

Test your implementation as follows

```
// File main_solve.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "solve.h"
```

```
double f_solve(double x);

int main(void)
{
 double x_max, x_min, x, tol;
 int count;

 fprintf(stdout, "Solve_Bisect\n");
 count = 0;
 tol = 1.0e-10;
 x = Solve_Bisect(0.0, f_solve, 0.1, 4.0, tol, &count);
 fprintf(stdout, "count = %d\n", count);
 fprintf(stdout, "x = %e\n", x);

 fprintf(stdout, "Solve_Newton\n");
 count = 0;
 tol = 1.0e-10;
 x = Solve_Newton(0.0, f_solve, 4.0, tol, &count);
 fprintf(stdout, "count = %d\n", count);
 fprintf(stdout, "x = %e\n", x);

 return 1;
}

double f_solve(double x)
{
  return sin(x);
 // return x*exp(x) - 3.0;
}
```

To compile, do `gcc -o solve_test main_solve.c solve.c -lm`
Replace `gcc` with your C compiler. It is usually `cc` or `gcc`. When you run `solve_test`, you should get 3.141593 from both methods.

Also test your implementation to solve

$$3 = xe^x \tag{28}$$

within 0 and 4. you should get 1.049909 from both methods.

You should really try many different functions and many different values and see if your implementation is good enough. It is important to make sure that these two methods are working because later sections will crucially depend on both `Solve_Bisect` and `Solve_Newton`. It is also something good to have in your arsenal for later use in your other projects.

5 Finding an extremum

In this section, we will solve

$$f'(x) = 0 \tag{29}$$

to get an extremum of a function. We will also calculate the curvature at the extremum given by $f''(x)$ at that point.

This section assumes that your `Solve_Newton` is working properly. If it is not, you need to get that to work first.

Numerical Derivative

In many cases, derivatives are available analytically. But in many other cases, analytic derivatives are either too complicated or not available. In those situations, it is good to have a way to calculate derivatives numerically.

The numerical methods we are going to use in this project are mostly based on the Taylor expansion of a function

$$f(x \pm h) = f(x) \pm hf'(x) + \frac{h^2}{2}f''(x) \pm \frac{h^3}{6}f'''(x) + O(h^4) \tag{30}$$

From this we get

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \tag{31}$$

and

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2) \tag{32}$$

In calculus, we take the $h \to 0$ limit to get the derivative. In numerical analysis, however, this limit is not good. This is because all numbers in a computer has a finite number of significant digits. For instance, if one adds or subtracts $h = 1.0 \times 10^{-20}$ to $x = 1.234567890123456$ which has only 16 significant digits, then it does not change x at all inside the computer. Hence, Eq.(31) will give you either 0 or something completely unrelated to $f'(x)$. If one uses $h = 10^{-15}$, then one can calculate the derivative, but the result will have just a single significant digit.

Hence, whenever one calculates derivatives numerically, accuracy is inevitably lost. The question, of course, is how good is good enough? Suppose $h_n = \eta_n x$ with $\eta_n = 10^{-n}$. Then, how many signficant digits are we

losing when calculating $(f(x+h) - f(x-h))/(2h)$ numerically? To estimate this, consider $f(x) = x^m$. Let's fix $m = 5$ and do some experiment. Letting, $x = 123.4567890123456$ (16 significant digits), we know that $f'(x) = 5x^4 = 1.161528614455904 \times 10^9$. For $n = 2$ to 10 and $n = 15$, the above formula gives

```
n = 3
f' analytic  = 1.161528614455904e+09
f' numerical = 1.161530937513428e+09

n = 4
f' analytic  = 1.161528614455904e+09
f' numerical = 1.161528637687172e+09

n = 5
f' analytic  = 1.161528614455904e+09
f' numerical = 1.161528614687156e+09

n = 6
f' analytic  = 1.161528614455904e+09
f' numerical = 1.161528614452323e+09

n = 7
f' analytic  = 1.161528614455904e+09
f' numerical = 1.161528614112434e+09

n = 8
f' analytic  = 1.161528614455904e+09
f' numerical = 1.161528614884910e+09

n = 9
f' analytic  = 1.161528614455904e+09
f' numerical = 1.161528590165672e+09

n = 10
f' analytic  = 1.161528614455904e+09
f' numerical = 1.161528744660912e+09
```

```
n = 15
f' analytic  = 1.161528614455904e+09
f' numerical = 1.205062877056503e+09
```

You can see that the maximum number of significant digits (~ 11) is obtained around $n = 6$. After that we get worse and worse results because of the loss of the significant digits. At $n = 15$, only the first digit is accurate, as expected.

For the second derivative,

```
n = 3
f'' analytic  = 3.7633527447073080e+07
f'' numerical = 3.7633546263680339e+07

n = 4
f'' analytic  = 3.7633527447073080e+07
f'' numerical = 3.7633527611142240e+07

n = 5
f'' analytic  = 3.7633527447073080e+07
f'' numerical = 3.7633526660069525e+07

n = 6
f'' analytic  = 3.7633527447073080e+07
f'' numerical = 3.7633446569736041e+07

n = 7
f'' analytic  = 3.7633527447073080e+07
f'' numerical = 3.7642456732251741e+07

n = 8
f'' analytic  = 3.7633527447073080e+07
f'' numerical = 3.5039520894383267e+07

n = 9
f'' analytic  = 3.7633527447073080e+07
f'' numerical = 0.0000000000000000e+00

n = 10
f'' analytic  = 3.7633527447073080e+07
```

```
f'' numerical = 0.0000000000000000e+00
```

The number of significant digits (~ 8) peaks at $n = 5$. For $n \geq 9$, the loss of accuracy is so severe that the 2nd derivative cannot be calculated at all.

This is just one specific example, of course. Emperically, it is found that $h = 10^{-5}x$ works for most cases when using the double precision and that is what we will use. If using the single precision, $h = 10^{-3}$ will do. Note that if you ask `C` to print a number up to 16 significant digits as I have shown above, it will happily do so. But that *does not mean* that all digits are significant.

The first task for this section is to calculate the first and the second derivatives numerically.

Here are the tasks. Fill in the details and test them.

The header file should be

```
// File derivatives.h
#ifndef DERIVATIVES_H
#define DERIVATIVES_H

double Derivative_FirstD(double x, double (*func)(double));
double Derivative_SecondD(double x, double (*func)(double));

#endif
```

Implementations go into `derivatives.c`.

```
// File derivatives.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "derivatives.h" // Contains Derivative_FirstD
                         // and Derivative_SecondD

double Derivative_FirstD(double x, double (*func)(double))
{
 double df, h;

 // TD: Set h to 1.0E-5

 // TD: If x is non-zero, set h to h*x
```

```
 // TD: Implement df = ( f(x+h) - f(x-h) )/(2h)

 return df;
}// Derivative_FirstD


double Derivative_SecondD(double x, double (*func)(double))
{
 double ddf, h;

 // TD: Set h to 1.0E-5

 // TD: If x is non-zero, set h to h*x

 // TD: Implement ddf = ( f(x+h) + f(x-h) - 2 f(x) )/(h*h)

 return ddf;
}// Derivative_SecondD
```

To test, here is the `main_deriv.c`:

```
// File main_deriv.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "derivatives.h"

double test_func(double x);

int main(void)
{
 int i, imax;
 double x, dx, x_min, x_max, f, df, ddf;
 FILE *output; // This is a pointer to a file.
               // This is used to input from or output to a file
               // See how this is used with fopen and fclose below.

 x_min = 0.0;
```

```
 x_max = 15.0;

 imax = 1000;

 dx = (x_max - x_min)/((double) imax);

 output = fopen("deriv_test.dat","w"); // Open the file named
                                       // "deriv_test.dat" to "w"rite on it.
// If you use "w", the the file is overwritten.
// That means you will lose whatever was in that file.
// If you want to write at the end of an existing file
// without touching the existing data, use "a" for "a"ppend.

 for(i=0; i<=imax; i++)
  {
   x = x_min + dx*i;

   // TD: Calculate test_func(x) and put it in f.
   // TD: Calculate the first derivative of test_func and put it in df
   // TD: Calculate the second derivative of test_func and put it in ddf

   fprintf(output, "%e  %e  %e  %e\n", x, f, df, ddf); // Write to the file
  }// i
 fclose(output); // You always need to do this after accessing the file.

 return 0;
}// main

double test_func(double x)
{
 return sin(x);
}// test_func
```

Compile with
`gcc -o deriv_test main_deriv.c derivatives.c -lm`
Running `deriv_test` should generate a file named `deriv_test.dat`.

Finding an extremum of a function

With the derivative programs working, we can now write a program that calculates an extremum (a minimum or a maximum) of a given function. Here we introduce the concept of `typedef` in C. "Type" here refers to the data type such as `int, float, double`. These are of course defined within C. But in C, you can define your own data types. In the below, we introduce
`typedef double (*FuncPT)(double);`
which means that we define `FuncPT` to be a pointer to the function that takes in a double and returns a double. Here is `extremum.c` where we calculate an extremum of a function numerically by solving $f'(x) = 0$.

We again begin with the header file:

```
// File extremum.h
#ifndef EXTREMUM_H
#define EXTREMUM_H

double Extremum_GetExtremum
(double (*func)(double), double x_init, double *curvature);

#endif
```

This function is then implemented in `extremum.c`:

```
// File extremum.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "derivatives.h"
#include "extremum.h"
#include "solve.h"

typedef double (*FuncPT)(double); // FuncPT is a pointer to a function
                                  // that takes in a double and
                                  // returns a double.
                                  // We use this to pass an arbitrary function
                                  // to another function.
FuncPT ORIG_FUNC; // A common variable. Only valid within this file
double Extremum_DF(double x); // Used only within this file
```

```
// Extremum_GetExtremum finds the miminum or maximum near x_init
// This function returns the value of x where the extremum is
// The variable curvature has the value of the second derivative
// at the extremum
//
double Extremum_GetExtremum(FuncPT func, double x_init, double *curvature)
{
 double x, tol, ddf;
 int count;

 ORIG_FUNC = func; // To communicate with Extremum_DF

// TD: Use Solve_Newton to solve 0 = Extremum_DF(x) starting with x_init

// TD: Use Derivative_SecondD to calculate the second derivative
// at the extremum

// TD: Set curvature to the second derivative.
// Note that curvature is a pointer.
// Hence, you need to do *curvature = (calculated value);
// If you do curvature = (calculated value), you'll get an error
// since the types don't match.


// TD: Return the value of x at the extremum
}// Extremum_GetExtremum


// We are using FuncPT func -> ORIG_FUNC here because
// Solve_Newton can only take in double func(double) type of function
double Extremum_DF(double x)
{
 double f;

 // TD: Calculate the first derivative of ORIG_FUNC using Derivative_FirstD

 // TD: Return the calculated value
```

```
}// Extremum_DF
```

You can then test your program:

```
// File main_extremum.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "extremum.h"

double test_func(double x);

int main(void)
{
 // TD: Declare necessary variables

 // TD: Start with test 1 below in the "test_func"
 // TD: Set x_init to an appropriate value

 x_extremum = Extremum_GetExtremum(test_func, x_init, &curvature);

 // TD: Print the x_extremum, test_func(x_extremum) and curvature
 // to the result file

 // TD: Repeat for test 2

 // TD: Repeat for test 3

 return 0;
}// main



double test_func(double x)
{
 double f;

// Uncomment for test 1
 f = x*x - 2.0*x + 1;
```

```
// This is (x-1)^2 which has the minimum at x = 1


// Uncomment for test 2
// f = 1.0/x/x - 2.0/x;
// Here, assume x > 0
// You should not set x_init = 0.0 which is a singular point.
// A good way to guess x_init is to plot the function

// Uncomment for test 3
// f = 1.0/x/x - 1.0/(1.0 + exp(x - 5.0));
// Assume x > 0
// You should not set x_init = 0.0 which is a singular point.
// A good way to guess x_init is to plot the function

 return f;
}// test_func
```

Compile with
`gcc -o extremum_test main_extremum.c extremum.c -lm`
For `test 3`, you should get `x_extremum = 2.807731e+00, f_extremum = -7.727035e-01, curvature = 1.687502e-01`.

6 Numerov Method

The radial equation we want to solve

$$\left(-\frac{d^2}{dx^2}+\frac{\ell(\ell+1)}{x^2}+\tilde{V}(x)+\tilde{E}_{n_r\ell}\right)\tilde{u}_{n_r\ell}(x)=0 \tag{33}$$

can be cast into a general form

$$\frac{d^2y}{dx^2}=F(x)y(x) \tag{34}$$

Let's see how we can solve this numerically.

Suppose we discretize the x space so that the initial point is $x_{\rm init}=x_0$ and the final point is $x_{\rm fin}=x_N=x_0+Nh$. Here

$$h=\frac{x_N-x_0}{N} \tag{35}$$

is the step size. Let $x_n=x_0+nh$, $y_n=y(x_n)$ and $F_n=F(x_n)$. Using Taylor expansion, one can show

$$y_{n+1}+y_{n-1}-2y_n=h^2\left.\frac{d^2y}{dx^2}\right|_{x=x_n}+\frac{h^4}{12}\left.\frac{d^4y}{dx^4}\right|_{x=x_n}+O(h^6) \tag{36}$$

for any function $y(x)$ that has finite 4-th derivative near $x=x_n$. Using Eq.(34), we get

$$y_{n+1}+y_{n-1}-2y_n=h^2F_ny_n+\frac{h^4}{12}\left.\frac{d^2}{dx^2}F(x)y(x)\right|_{x=x_n}+O(h^6) \tag{37}$$

We can replace

$$\begin{aligned}\left.\frac{d^4y}{dx^4}\right|_{x=x_n}&=\left.\frac{d^2}{dx^2}\frac{d^2y}{dx^2}\right|_{x=x_n}=\left.\frac{d^2}{dx^2}\left(F(x)y(x)\right)\right|_{x=x_n}\\&=\frac{1}{h^2}\left((Fy)_{n+1}+(Fy)_{n-1}-2(Fy)_n\right)+O(h^2)\end{aligned} \tag{38}$$

which yields

$$y_{n+1}+y_{n-1}-2y_n=h^2F_ny_n+\frac{h^2}{12}\left(F_{n+1}y_{n+1}+F_{n-1}y_{n-1}-2F_ny_n\right)+O(h^6) \tag{39}$$

Rearranging, we get

$$y_{n+1} = \frac{1}{1-(h^2/12)F_{n+1}}\left(2(1+(5h^2/12)F_n)y_n - (1-(h^2/12)F_{n-1})y_{n-1}\right) + O(h^6) \tag{40}$$

This method is called the Numerov method. In this way, given y_0 and y_1, we can calculate all subsequent y_n for $n \geq 2$ with high accuracy.

We can now describe the algorithm to solve

$$\frac{d^2}{dx^2}y(x) = F(x)y(x) \tag{41}$$

1. First, make an array of $F[n] = F(x_n)$. In this way, they can be calculated only once and reused.

2. Starting from given $y[0]$ and $y[1]$, calculate

$$y[n+1] = \frac{1}{1-(h^2/12)F[n+1]}\Big(2(1+(5h^2/12)F[n])y[n] - (1-(h^2/12)F_[n-1])y_[n-1]\Big) \tag{42}$$

   for $y[2], \cdots, y[N]$.

Numerov Solver

To implement the above algorithm, let's first think about common data that is given externally and does not change during the calculation. These can be collected in one data structure:

```
// File numerov_params.h
#ifndef NUMEROV_PARAMS_H
#define NUMEROV_PARAMS_H
#include "params.h"

typedef double (*Func_1D)(double, DynamicVars *);
// This defines Func_1D to be a pointer to functions
// that takes a double variable and a DynamicVars type variable
// and returns a double precision number.
// DynamicVars is defined in params.h (see below).
```

```
// That is why params.h is included.

typedef struct numerov_params
{
 double x_f; // maximum x
 double x_i; // minimum x
 double y_0; // y(x_i)
 double y_1; // y(x_i + h)
 int nmax;   // number of sampling points
 double h;   // step size h = (x_i - x_f)/nmax

// The function F in y'' = Fy
 Func_1D NumerovFunc_F;

} NumerovParams; // Define a data type named NumerovParams

#endif
```

In this way, these values can set only once. The way to access data in a structure is via the . (dot) operator. For example, suppose you have a `NumerovParams` type variable `Num_Params`. Then, if you want to get the value of `nmax`, you would do `nmax = Num_Params.nmax;` If a pointer is given, you use the arrow or the `->` operator. For instance, if `Num_Params_pt = &Num_Params`, then you would do `nmax = Num_Params_pt->nmax;`

Note that this file includes `params.h` which collects the problem specific data:

```
// File params.h
#ifndef PARAMS_H
#define PARAMS_H

#define hbarc (197.3) // MeV.fm = eV.nm

// This is a collection of parameters that do not change
// during the calculation. This will be set once in the beginning
// of the calculation and be accessed via PARAM_DATA below.
typedef struct params
{
 double mass; // Mass of the particle
```

```
 double Ea;   // Energy scale
 double ka;    // Momentum scale
 double r0;   // Length scale
 double x0;   // x0 = k*r0
 int ell;

 char *mass_unit;  // The unit of mass. Either MeV or eV
 char *length_unit; // The unit of length. Either fm (goes with MeV)
                    // or nm (goes with eV)

 double nucA; // Atomic mass
 double nucZ; // Atomic number

 // To bracket the energy eigenvalue search
 double Et_min;
 double Et_max;

} Params;

// This is a collection of variables that change during the calculation
// and need to be passed frequently
// Some of these parameters will be used in later
typedef struct dynamic_vars
{
 double Eb; // Absolute value of the bound energy
 double kb; // sqrt(2*mass*Eb)
 double rc; // The turning point radius
 double Et; // Eb/Ea
 double xc; // ka*rc
 double rf; // The last point
 double xf; // xf = ka*rf.
} DynamicVars;

extern Params PARAM_DATA; // Run specific data
// The key word extern here specifies that this variable
// can be accessed by any function as long as params.h is included.
// Any variable declared as an extern in a header file must be declared
// once in a .c file. We'll do that in the main_numerov.c.
```

```
#endif
```

This contains problem-specific parameters that has nothing to do with the Numerov algorithm except that the F-function in $y'' = Fy''$ many need some of them. In this way, the files `numerov_data.h`, `numerov.h` and `numerov.c` do not need to be modified for each problem.

Here is the `main_numerov.c` to test the implementation of the numerov method. Note that you will need `vector_mtx.h` and `vector_mtx.c`. They are explained in `c_tutorials.pdf` and provided in the appendix.

```
// File main_numerov.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "vector_mtx.h"
#include "numerov.h"
#include "numerov_params.h"
#include "params.h"

double TestF(double x, DynamicVars *Dyn_Vars); // Test purporses

void ReadInNum_Params(char *input_file_name, NumerovParams *Num_Params);
void PrintParams(NumerovParams Num_Params);
void PrintY(double *y, NumerovParams Num_Params);

// The variable argc is the number of "words" this program is invoked with.
// Suppose that the executable is named numerov_test
// and you execute it with
// numerov_test input_data
// Then argc = 1 since you are using one file other than the executable.
// The variable argv contains these words.
// argv[0] is the name of the executable itself.
// In this case argv[0] is numerov_test and argv[1] is input_data
int main(int argc, char **argv)
{
 double *y; // This is going to become 1-D array of doubles
            // when a memory space is allocated

 char *input; // This is going to become 1-D array of characters
```

```
              // when a word is assigned to it

 NumerovParams Num_Params; // To hold static data needed for Numerov method

 DynamicVars Dyn_Vars; // To hold data that may change during the calculations
                        // Not used for now

 input = argv[1];

 ReadInNum_Params(input, &Num_Params); // Fill Num_Params
 // This is for this test only.
 // Later, we will use Schroedinger_InitializeCommonVar(void)

 PrintParams(Num_Params); // Check if everything is read-in correctly
                   // Record the read-in data in an output file

 y = vector_malloc(Num_Params.nmax+1); // Allocates an array
                                  // You can find vector_mtx.c
                                  // and vector_mtx.h in the appendix

 Num_Params.NumerovFunc_F = TestF; // Assign the F function in y'' = Fy
                         // to be used in the calculation
                         // The name of a function
                         // (in this case TestF defined below)
                         // is the pointer to the function

 Numerov_Advance(y, &Num_Params, &Dyn_Vars); // Solve y'' = Fy

 PrintY(y, Num_Params); // Print y to an output file

 free(y); // Always free allocated memories after use
 return 0;
}// main


double TestF(double x, DynamicVars *Dyn_Vars)
{
 return -x; // Airy
```

```
// return -1.0; // SHO
}// TestF


// This is for this test only.
// Later, we will use Schroedinger_InitializeCommonVar(void)
void ReadInNum_Params(char *input_file_name, NumerovParams *Num_Params)
{
 FILE *input_file;
 double x;
 int ix;

 input_file = fopen(input_file_name, "r"); // Open the input file
                                           // to "r"ead
// Read in the first line and put its value in x
 fscanf(input_file, "%le", &x);
 Num_Params->x_i = x;

// TD: Read in x_f, y_0 and y_1.

// TD: Read in an integer value and put its value in ix
 fscanf(input_file, "%d", &ix);
 Num_Params->nmax = ix;

// TD: Calculate h = (x_f - x_i)/nmax
// and put it in Num_Params->h

 fclose(input_file); // Always close an opened file
 return;
}// ReadInNum_Params


void PrintParams(NumerovParams Num_Params)
{
 FILE *output;

 output = fopen("params.dat","w"); // Open a file for "w"riting
```

```
 fprintf(output, "x_i = %e\n", Num_Params.x_i);

// TD: Record x_f, y_0, y_1, nmax and h
// To print an integer, use %d

 fclose(output); // Always close an opened file
 return;
}// PrintParams


void PrintY(double *y, NumerovParams Num_Params)
{
 FILE *output;
 int n;
 double xn;

 // TD: Open a file for writing and assign it to output.
 // You need to choose the name of the file.

 for(n=0; n<=Num_Params.nmax; n++)
  {
   xn = Num_Params.x_i + n*Num_Params.h;

   // TD: Print xn and y[n] as a line in the output file
   fprintf(output, "%e  %e\n", xn, y[n]);
  }

 // TD: Close output
 return;
}// PrintY
```

We need to implement `Numerov_Advance` in `numerov.c`. Before we do that, here is `numerov.h`:

```
// File numerov.h
#ifndef NUMEROV_H
#define NUMEROV_H
#include "numerov_params.h"
```

```
void Numerov_Advance
(double *y, NumerovParams *Num_Params, DynamicVars *Dyn_Vars);

#endif
```

The implementation of the Numerov Method itself is in `numerov.c`:

```
// File numerov.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "numerov.h"
#include "numerov_params.h"
#include "vector_mtx.h"

void Numerov_Make_Fn // Makes F[n]
(double *numerov_F, NumerovParams *Num_Params, DynamicVars *Dyn_Vars);

void Numerov_Advance_A_Step // Takes a step from y[n-1] to y[n]
(double *y, int n, double *numerov_F, NumerovParams *Num_Params,
 DynamicVars *Dyn_Vars);

// Just a set of instructions
void Numerov_Advance
(double *y, NumerovParams *Num_Params, DynamicVars *Dyn_Vars)
{
 double *numerov_F;
 int n;
 int nmax;

 nmax = Num_Params->nmax;

 // Allocate memory space for numerov_F[0]...numerov_F[nmax]
 numerov_F = vector_malloc(nmax+1);

 // Make numerov_F[n] to be used in the algorithm
 Numerov_Make_Fn(numerov_F, Num_Params, Dyn_Vars);
```

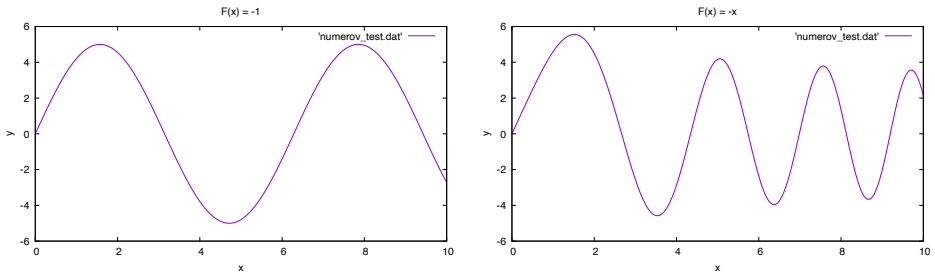
```
 // Get the initial values from Num_Params
 y[0] = Num_Params->y_0;
 y[1] = Num_Params->y_1;

 // Calculate y[n] for n >= 2
 for(n=2; n<=nmax; n++)
  {
   Numerov_Advance_A_Step(y, n, numerov_F, Num_Params, Dyn_Vars);
  }

 return;
}// Numerov_Advance


// Tabluate F in y''(t) = F(t)y(t)
void Numerov_Make_Fn
(double *numerov_F, NumerovParams *Num_Params, DynamicVars *Dyn_Vars)
{
 int n;
 double x_n;

 for(n=0; n<=Num_Params->nmax; n++)
  {
   // TD: Calculate x_n = x_i + h*n
   // TD: Fill numerov_F[n] with (Num_Params->NumerovFunc_F)(x_n, Dyn_Vars)
  }

 return;
}// Numerov_make_Fn


// Calculates
// y[n] = ( 1/(1 - (h^2/12)F[n]) )
//       *( 2(1 + (5h^2/12)F[n-1])y[n-1] - (1 - (h^2/12)F[n-2])y[n-2] )
//
void Numerov_Advance_A_Step
(double *y, int n, double *numerov_F, NumerovParams *Num_Params, DynamicVars *Dyn_
{
```

```
 double h;
 double h2_over_12;

 // TD: Get h from Num_Params
 // TD: Calculate h2_over_12 = h^2/12

 // TD: Implement the following
 // TD: Start with y[n] = 0.0;
 // Add 2(1 + (5h^2/12)F[n-1])y[n-1] only if y[n-1] is non-zero
 if(y[n-1] != 0.0)
  {
   y[n] += 2.0*(1 + 5.0*h2_over_12*numerov_F[n-1])*y[n-1];
  }
 // TD: Add -(1 - (h^2/12)F[n-2])y[n-2] only if y[n-2] is non-zero
 // TD: Divide y[n] by (1 - (h^2/12)F[n])

 return;
}// Numerov_Advance_A_Step
```

To test, compile this program. Run it and plot the data in your data file. Use

- `x_i = 0.0`
- `x_f = 10.0`
- `y_0 = 0.0`
- `y_1 = 0.1`
- `nmax = 500`

Test $F(x) = -1$ and $F(x) = -x$ (`TestF` in `main_numerov.c`). They should produce the following plots:



7 Preparation

In a previous section, you've learned to calculate the extremum point of a function. We did that because to scale the Schrödinger equation as

$$\left(-\frac{d^2}{dx^2}+\frac{\ell(\ell+1)}{x^2}+\tilde{V}(x)+\tilde{E}_{n_r\ell}\right)\tilde{u}_{n_r\ell}(x)=0 \tag{43}$$

we needed to get a typical energy scale E_a. Recall that

$$k_a = \sqrt{2\mu E_a} \tag{44}$$
$$x = k_a r \tag{45}$$
$$\tilde{V}(x) = V(x/k_a)/E_a \tag{46}$$
$$\tilde{E}_{n_r\ell} = -|E_{n_r\ell}|/E_a \tag{47}$$

We define the energy scale E_a as the minimum value of

$$E_{\text{eff}}(r) = \frac{1}{2\mu r^2}+V(r) \tag{48}$$

where we let $\ell = 0$ and used the uncertainty principle to estimate the kinetic energy to be $p_r^2/2\mu \sim 1/2\mu r^2$. We also assumed that $V(r)$ is no more singular than $1/r$ as $r \to 0$. Hence, the equation to solve to get the extremum point is

$$E'_{\text{eff}}(r) = -\frac{1}{\mu r^3}+V'(r) = 0 \tag{49}$$

We will do that using `Extermum_GetExtremum` from `extremum.h`. Some care must be taken not to approach $r = 0$ too closely.

Note that in the code, the variable `mass` *is used for the effective mass* μ.

The next task is to actually solve the differential equation Eq.(43) numerically with the boundary conditions

$$\lim_{x\to 0}\tilde{u}_{n_r\ell}(x) = 0$$
$$\lim_{x\to\infty}\tilde{u}_{n_r\ell}(x) = 0 \tag{50}$$

We will do this in 2 parts.

1. $0 < x \leq x_c$: $\tilde{u}_I(x)$
2. $x_c \leq x < x_f$: $\tilde{u}_{II}(x)$ with $x_f \gg 1$.

These two solutions, of course, need to be matched at x_c. The boundary condition at $x = 0$ is $\tilde{u}_I(0) = 0$. The boundary condition at large x_f needs more analysis. We'll do that later.

The matching method is then

1. Start with a trial $\tilde{E}_{n_r\ell}$
2. Evolve $\tilde{u}_I(x)$ forward from 0 to x_c and evolve $\tilde{u}_{II}(x)$ backward from x_f to x_c.
3. Adjust $\tilde{E}_{n_r\ell}$ until $\tilde{u}'_I(x)/\tilde{u}_I(x)$ and $\tilde{u}'_{II}(x)/\tilde{u}_{II}(x)$ match at x_c.

What should x_c be, then? We can use the classical turning point. This is defined as the point where

$$\frac{\ell(\ell+1)}{x_c^2} + \tilde{V}(x_c) + \tilde{E}_{n_r\ell} = 0 \tag{51}$$

or

$$\ell(\ell+1) + x_c^2(\tilde{V}(x_c) + \tilde{E}_{n_r\ell}) = 0 \tag{52}$$

This equation can be solved using `Solve_Newton`. The radial coordinate corresponding to x_c is $r_c = x_c/k_a$.

In this section, we will do the preparation for the shooting method. Implementation of the shooting method will be done in the next section.

Common data structure and the typical scales

The first task is to calculate the scales k_a, E_a and r_0 given μ (the effective mass of the particle), ℓ and $V(r)$. To do so, let's first think about common data structure that is given once and does not change during the calculation.

We have already seen how to do this kind of programming when we implemented `NumerovParams` and `Params` in the last section. The reason we have two separate data structures is because to separate the Numerov part from other parts. One should always strive to make each part of the program as independent as possible so that they can be written once, tested thoroughly and re-used in other programs and other parts with confidence. The data structures `NumerovParams` and `Params` should have been implemented in the previous section. We'll not repeat them here.

Here is the `main_schroedinger.c` that we are going to use for this section and the next.

```
// File main_schroedinger.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "params.h"
#include "numerov_params.h"
#include "init.h"
#include "schroedinger.h"
#include "vector_mtx.h"

// Functions needed only in this file
// Read in Params parameters
void ReadIn_Params(char *input_file);

// Read in NumerovParams parameters
// This only reads in the number of data points for u_I and u_II
// The rest of the NumerovParams are read in by
// Schroedinger_InitializeCommonVar
//
void ReadIn_Num_Params
(char *input_file_name,
 NumerovParams *Num_Params_f, NumerovParams *Num_Params_b);
```

```
// Record parameters
void Record_Params(NumerovParams Num_Params_f, NumerovParams Num_Params_b);

// Record results
void Record_Results(DynamicVars Dyn_Vars,
NumerovParams Num_Params_f, NumerovParams Num_Params_b,
double *yf, double *yb);

// This is declared as an extern in params.h
// As such, it needs to be declared outside params.h, but only once.
Params PARAM_DATA;

// This program is to be invoked as
// executable input_file1 input_file2
// the word "input_file1" and "input_file2" are then put into "argv" below
// argv[0] is the name of the executable (e.g. schroedinger)
// argv[1] is the name of the first input file (e.g. input_coulomb)
// argv[2] is the name of the second input file (e.g. input_n_params)
int main(int argc, char **argv)
{
 DynamicVars Dyn_Vars; // These parameters are calculated
 NumerovParams Num_Params_f; // For the forward evolution of u_I
 NumerovParams Num_Params_b; // For the backward evolution of u_II
 double *yf, *yb; // yf contains u_I, yb contains u_II


 ReadIn_Params(argv[1]); // Reads in the initial data
                         // from the first input file
                         // This is for PARAM_DATA

 ReadIn_Num_Params(argv[2], &Num_Params_f, &Num_Params_b);
                            // Reads in the initial data
                            // from the second input file
                            // This is for Num_Params_f and Num_Params_b

 Init_CalcScales();
                       // Get the energy and length scales
                       // to prepare for solving the differential eq
```

```
 // Record the parameters
 Record_Params(Num_Params_f, Num_Params_b);


/* Will be implemented in the next section */
/*
 // Allocate memory for the forward wavefunction yf
 yf = vector_malloc(Num_Params_f.nmax+1);

 // Allocate memory for the backward wavefunction yb
 yb = vector_malloc(Num_Params_b.nmax+1);

 Schroedinger_GetBoundState(&Dyn_Vars, &Num_Params_f, &Num_Params_b, yf, yb);

 Record_Results(Dyn_Vars, Num_Params_f, Num_Params_b, yf, yb);
*/
/* Next section */

 return 0;
}// main


// TD: Implement ReadIn_Params
// Follow the structure of ReadIn_Num_Params from the previous section
// Input_file contains the name of the input file
void ReadIn_Params(char *input_file)
{
 FILE *input;
 double x;
 int ix;
 char *mass_unit;

 input = fopen(input_file, "r"); // Open the input file to "r"ead

 // TD: Read in the mass and divide it by hbarc so that it has
 // the 1/length unit. We are going to use the length unit for everything
 // Put it in PARAM_DATA.mass.
```

```
 // Read in the mass unit
 mass_unit = (char *) malloc(sizeof(char)*10);
             // First allocate enough memory to hold it.
             // 10 should be enough for any mass units
 // From the second line, read in a line and put it in PARAM_DATA.mass_unit
 fscanf(input, "%s", mass_unit);
 PARAM_DATA.mass_unit = mass_unit;

 // According to the mass unit, determine the length unit
 // strcmp is defined in string.h. It means "string compare".
 // If the two strings are the same, then it returns 0.
 // If the two strings are not the same, it returns non-zero.
 if(strcmp(mass_unit, "eV")==0) {PARAM_DATA.length_unit = "nm";}
 else if(strcmp(mass_unit, "MeV")==0) {PARAM_DATA.length_unit = "fm";}
 else {
  fprintf(stderr, "ReadIn_Params: %s is an unknown unit.\n", mass_unit);
  fprintf(stderr, "Known units are eV and MeV.\n");
  fprintf(stderr, "Exiting.\n");
  exit(0);
 }

 // TD: From the next line, read the orbital angular momentum ell
 // and put it in PARAM_DATA.ell

 // TD: Read in the atomic mass A as a double and put it in
 // PARAM_DATA.nucA.

 // TD: Read in the atomic mass Z as a double and put it in
 // PARAM_DATA.nucZ.

 fclose(input); // Always close an opened file

 return;
}// ReadIn_Params


void Record_Params(NumerovParams Num_Params_f, NumerovParams Num_Params_b)
```

```
{
 double x;
 int i;
 FILE *output;

 // TD: Open a data file (Choose the name according to the naming scheme)
 // TD: Record mass, r0, Ea, ka, ell, x0, nucA, nucZ, and
 // nmax (both forward and backward) in the data file.
 // TD: Don't forget to close the file

 return;

}// Record_Params


// Read in NumerovParams data
void ReadIn_Num_Params
(char *input_file_name, NumerovParams *Num_Params_f, NumerovParams *Num_Params_b)
{
 FILE *input_file;
 double x;
 int ix;

 // Open the input file input_file_name to read

 // TD: Read in nmax for the forward evolution and put it in
 // Num_Params_f->nmax

 // TD: Read in nmax for the backward evolution and put it in
 // Num_Params_b->nmax

 // TD: Always close an opened file
 return;
}// ReadIn_Num_Params


void Record_Results(DynamicVars Dyn_Vars,
NumerovParams Num_Params_f, NumerovParams Num_Params_b,
```

```
double *yf, double *yb)
{
 // Next section
 return;
}// Record_Results
```

The next task is to implement `Init_CalcScales` in init.c. First the header file

```
// File init.h
#ifndef INIT_H
#define INIT_H

void Init_CalcScales(void);
                      // Get the energy and length scales
                      // to prepare for solving the radial equation

#endif
```

The implementation is in `init.c`:

```
// File init.c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "params.h"
#include "init.h"
#include "radial_eq_functions.h"
#include "solve.h"
#include "extremum.h"
#include "derivatives.h"

double Init_Rmin_Function(double r);
   // The function to minimize to get Ea and ka
   // Equals V(r) + 1/(2 m r^2)

void Init_CalcScales(void)
{
 double r_min, r_init, E_min, Eb, mass, r_c, curvature;
```

```
 double R_A;
 FILE *output;

 // TD: Get mass from PARAM_DATA

 if(PARAM_DATA.nucA == 0.0) // Not a nuclear potential
  {
   r_init = 0.01; // to start the iteration
// TD: Get the minimum radius r_min by using Extremum_GetExtremum on
// Init_Rmin_Function

// TD: Put r_min in r0 in PARAM_DATA
// TD: Calculate E_min as the value of Init_Rmin_Function at r_min
// TD: Put |E_min| in Ea in PARAM_DATA
  }
 else // Nuclear potential. We know scales in this case.
  {
  // Next section
   PARAM_DATA.r0 = 1.3*pow(PARAM_DATA.nucA, 1.0/3.0); // Nuclear radius
   PARAM_DATA.Ea = 50.0/hbarc; // about 50 MeV
  }

// TD: Calculate ka = sqrt(2*mass*Ea) and put it in ka in PARAM_DATA
// TD: Calculate x0 = ka*r0 and put it in x0 in PARAM_DATA

 return;
}// Init_CalcScales


double Init_Rmin_Function(double r) // Function to minimize = V(r) + 1/(2 m r^2)
{
 double f, mass;

 // TD: Get mass from PARAM_DATA
 // TD: Calculate f = V(r) + 1/(2 m r^2)
 // Here V(r) is RadialEqFunctions_V(r) from radial_eq_functions.c

 return f;
```

```
}// Init_Rmin_function
```

The functions used in `init.c` are defined in `radial_eq_functions.h` and implemented in `radial_eq_functins.c`. First, the header file:

```
// File radial_eq_functions.h
#ifndef RADIAL_EQ_FUNCTIONS_H
#define RADIAL_EQ_FUNCTIONS_H

#include "params.h" // Because we use DynamicVars below

// Potential energy in r
double RadialEqFunctions_V(double r);

// V_eff = V + ell(ell+1)/(2m r^2)
double RadialEqFunctions_Veff(double r);

// The F function in the RHS of the differential equation y'' = Fy
// This is for the evolution forward from x = x_i
double RadialEqFunctions_F_Forward(double x, DynamicVars *Dyn_Vars);
// This is for the evolution backward from x = x_f
double RadialEqFunctions_F_Backward(double x, DynamicVars *Dyn_Vars);

#endif
```

These functions need to be implemented in `radial_eq_functions.c`:

```
// File radial_eq_functions.c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "radial_eq_functions.h"
#include "numerov_params.h"
#include "params.h"

#define alpha_EM (1.0/137.0) // The fine structure constant
                             // for testing purposes
// User defined function:
// RadialEqFunctions_V(double r) is the only function one should change.
```

```
// This is in the unit of 1/fm or 1/nm
double RadialEqFunctions_V(double r)
{
 double f;
 double A, R_A, a, R0, V0;

// For testing purpose, we use the Coulomb potential for hydrogen atom
 f = -alpha_EM/r;

// Nuclear Woods-Saxon potential
// Next section
/*
 V0 = 50.0/hbarc; // MeV to 1/fm
 a = 0.7;
 R_A = PARAM_DATA.r0;
 f = -V0/(1.0 + exp((r-R_A)/a));
*/
 return f;
}// RadialEqFunctions_V


// No user serviceable parts from here on

// Veff(r) = V(r) + ell(ell+1)/(2 m r^2)
double RadialEqFunctions_Veff(double r)
{
 double f, ell, mass;

 ell = (double) PARAM_DATA.ell; // PARAM_DATA.ell is an integer
                                // ell here is a double.
                                // Hence, you need to convert the data type.

 // TD: Get mass from PARAM_DATA
 // TD: Calculate f = V(r) + ell(ell+1)/(2 m r^2)
 // V(r) here is the RadialEq_Functions_V(r) above

 return f;
}// RadialEqFunctions_Veff
```

```
// F in y'' = Fy for u_I
// This is in x = ka*r
double RadialEqFunctions_F_Forward(double x, DynamicVars *Dyn_Vars)
{
 double x0, ka, r, f, g, Ea, Et, ell, eps;

 // TD: Get ell from PARAM_DATA. Make sure to convert to convert to double
 // TD: Get x0, ka, Ea from PARAM_DATA
 // TD: Get Et from Dyn_Vars. Note that Dyn_Vars here is a pointer.
 // So you need the arrow or the -> operator.

 // Small number to prevent x = 0
 eps = 1.0e-15;
 x += eps;
 r = x/ka;

 // TD: Implement f = ell(ell+1)/x^2 + V(r)/Ea + Et

 return f;
}// RadialEqFunctions_F_Forward


// F in Y'' = FY Backward evolution
// Here we use y = x_f - x
double RadialEqFunctions_F_Backward(double y, DynamicVars *Dyn_Vars)
{
 double ka, r, f, g, ell, Ea, Et, x;

 // TD: Get ell from PARAM_DATA. Make sure to double
 // TD: Get x0, ka, Ea from PARAM_DATA
 // TD: Get Et from Dyn_Vars. Note that Dyn_Vars here is a pointer.
 // So you need the arrow or the -> operator.

 // Get x and r
 x = Dyn_Vars->xf - y;
 r = x/ka;
```

```
 // TD: Implement f = ell(ell+1)/x^2 + V(r)/Ea + Et

 return f;
}// RadialEqFunctions_F_Backward
```

To test, use $m = 0.511 \times 10^6\,\mathrm{eV}$, energy unit `eV`, $\ell = 0$, `NucA = 0`, `NucZ = 0`. Those should go into the first input file. I named mine `input_coulomb`. Also set `nmax = 500` for both the forward and the backward evolution. Those should go into the second input file. I named mine `input_n_params`.

You should get `r0` equal to the Bohr radius and `Ea` should be equal to the ground state energy of the hydrogen atom. Namely,

```
r0 = 5.289648e-02 nm
Ea = 1.361287e+01 eV
```

8 Schrödinger Equation Solver

We are almost at the end. The task now is putting them all together. First, we make a slight modification to `param.h`. We are adding two more variables in `DynamicVars`:

```
// File: params.h
#ifndef PARAMS_H
#define PARAMS_H

// The Params part did not change

typedef struct dynamic_vars
{
 double Eb; // Absolute value of the bound energy
 double kb; // New
 double rc; // The turning point radius
 double Et; // Eb/Ea
 double xc; // ka*rc
 double rf;
 double xf;

 // THESE TWO ARE ADDED
 // To bracket the energy eigenvalue search
 double Et_min;
 double Et_max;

} DynamicVars;

extern Params PARAM_DATA; // Run specific data

#endif
```

Two parameters, `Et_min` and `Et_max` are added. They are now read in by `ReadIn_Params` in `main_schroedinger.c`. Here is the modified `main_schroedinger.c`. Note that the file is slighly modified from the one in the last section.

```
// File: main_schroedinger.c
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "params.h"
#include "numerov_params.h"
#include "init.h"
#include "schroedinger.h"
#include "vector_mtx.h"

// Read in Params parameters
// THIS IS MODIFIED
// The argument now includes *Dyn_Vars.
void ReadIn_Params(char *input_file, DynamicVars *Dyn_Vars);

// Read in NumerovParams parameters
void ReadIn_Num_Params
(char *input_file_name, NumerovParams *Num_Params_f, NumerovParams *Num_Params_b);

void Record_Params(NumerovParams Num_Params_f, NumerovParams Num_Params_b);

void Record_Results(DynamicVars Dyn_Vars,
NumerovParams Num_Params_f, NumerovParams Num_Params_b,
double *yf, double *yb);

Params PARAM_DATA;

// This program is to be invoked as
// executable input_file1 input_file2
// the word "input_file1" and "input_file2" are then put into "argv" below
// argv[0] is the name of the executable
// argv[1] is the name of the first input file
// argv[2] is the name of the second input file
int main(int argc, char **argv)
{
 DynamicVars Dyn_Vars;
 NumerovParams Num_Params_f;
 NumerovParams Num_Params_b;
 double *yf, *yb;
```

```
 // THIS IS MODIFIED
 // The argument now includes *Dyn_Vars.
 ReadIn_Params(argv[1], &Dyn_Vars); // Reads in the initial data
                       // from the first input file
                       // This is for PARAM_DATA

 ReadIn_Num_Params(argv[2], &Num_Params_f, &Num_Params_b);
                          // Reads in the initial data
                          // from the second input file
                          // This is for Num_Params_f and Num_Params_b

 Init_CalcScales();
                     // Get the energy and length scales
                     // to prepare for solving the differential eq

 // Record the parameters
 Record_Params(Num_Params_f, Num_Params_b);

 // The new part starts here
 // Allocate memory for the forward wavefunction yf
 yf = vector_malloc(Num_Params_f.nmax+1);

 // Allocate memory for the backward wavefunction yb
 yb = vector_malloc(Num_Params_b.nmax+1);

 Schroedinger_GetBoundState(&Dyn_Vars, &Num_Params_f, &Num_Params_b, yf, yb);

 Record_Results(Dyn_Vars, Num_Params_f, Num_Params_b, yf, yb);

 return 0;
}// main


// THIS IS MODIFIED
// input_file contains the name of the input file
void ReadIn_Params(char *input_file, DynamicVars *Dyn_Vars)
```

```
{
 FILE *input;
 double x;
 int ix;
 char *mass_unit;

 input = fopen(input_file, "r"); // Open the input file to "r"ead

 // The first part (reading in mass, mass unit, etc) is not modified.

 // THESE TWO ARE ADDED
 // Read in the minimum value of Et for the energy eigenvalue search
 fscanf(input, "%le", &x);
 Dyn_Vars->Et_min = x;

 // TD: Read in the maximum value of Et for the energy eigenvalue search

 fclose(input); // Always close an opened file

fprintf(stderr, "Done reading in.\n");
 return;
}// ReadIn_Params

void Record_Params(NumerovParams Num_Params_f, NumerovParams Num_Params_b)
{
 double x;
 int i;
 FILE *output;


 output = fopen("schroed_params.dat","w");

 fprintf(output, "mass = %e %s\n", hbarc*PARAM_DATA.mass, PARAM_DATA.mass_unit);

 // TD: Record all parameters in PARAM_DATA, Dyn_Vars, Num_Params_f
 // TD: and Num_Params_b like the example above.
 // TD: Length quantities should have the length unit
 // TD: and the energy/momentum/mass quantities should have
```

```
 // TD: the energy unit

 fclose(output);
 return;

}// Record_Params


void ReadIn_Num_Params
(char *input_file_name, NumerovParams *Num_Params_f, NumerovParams *Num_Params_b)
{
 FILE *input_file;
 double x;
 int ix;

 // You should already have this

 return;
}// ReadIn_Num_Params


void Record_Results(DynamicVars Dyn_Vars,
NumerovParams Num_Params_f, NumerovParams Num_Params_b,
double *yf, double *yb)
{
 double Et;
 FILE *output;
 int n;
 double x;

// TD: Record the final results.
// TD: Replace the data file names appropriately.
 output = fopen("schroedinger.dat","w");
 Et = Dyn_Vars.Et;
 fprintf(output, "Et = %e\n", Et);
// TD: Record the corresponding Eb = Et*Ea (in the energy unit)
 fclose(output);
```

```
// Record the forward going solution
 output = fopen("yf.dat","w");
 for(n=0; n<=Num_Params_f.nmax; n++)
  {
   x = Num_Params_f.x_i + n*Num_Params_f.h;  // Dimensionless
   x /= PARAM_DATA.ka; // 1/energy
   x *= hbarc;         // length dimension
   fprintf(output, "%e  %e\n", x, yf[n]/yf[Num_Params_f.nmax]);
  }
 fclose(output);

// Record the backward going solution
 output = fopen("yb.dat","w");
 for(n=0; n<=Num_Params_b.nmax; n++)
  {
  // 0 < x' < x_f - x_c
  // TD: Implement x = x_f - (x_i + n*h)
  // TD: getting x_i and h from Num_Params_b
  // TD: and xf from Dyn_Vars.
  // TD: Convert x to the length unit
   fprintf(output, "%e  %e\n", x, yb[n]/yb[Num_Params_b.nmax]);
  }
 fclose(output);

}// Record_Results
```

Next, let's finish coding `init.c`:

```
// File init.c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "params.h"
#include "init.h"
#include "radial_eq_functions.h"
#include "solve.h"
#include "extremum.h"
#include "derivatives.h"
```

```
double Init_Rmin_Function(double r);

void Init_CalcScales(void)
{
 double r_min, r_init, E_min, Eb, mass, r_c, curvature;
 double R_A;
 FILE *output;

 mass = PARAM_DATA.mass;
 if(PARAM_DATA.nucA == 0.0) // Not a nuclear potential
  {
   r_init = 0.01; // to start the iteration
   r_min = Extremum_GetExtremum(Init_Rmin_Function, r_init, &curvature);

   PARAM_DATA.r0 = r_min;
   E_min = Init_Rmin_Function(r_min);
   PARAM_DATA.Ea = fabs(E_min);
  }
 else // Nuclear potential. We know scales in this case.
  {
   // TD: Implement this if you have not done so already.
   PARAM_DATA.r0 = 1.3*pow(PARAM_DATA.nucA, 1.0/3.0); // Nuclear radius
   PARAM_DATA.Ea = 50.0/hbarc; // about 50 MeV
  }

 // This should already have been implemented
 PARAM_DATA.ka = sqrt(2.0*mass*(PARAM_DATA.Ea));
 PARAM_DATA.x0 = (PARAM_DATA.ka)*(PARAM_DATA.r0);

 return;
}// Init_CalcScales

// The rest should already be there
```

Let's also make sure that `radial_eq_functions.c` is complete:

```
// radial_eq_functions.c
#include <math.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include "radial_eq_functions.h"
#include "numerov_params.h"
#include "params.h"

#define alpha_EM (1.0/137.0)

// User defined functions
// RadialEqFunctions_V(double r) is the only function one should change.
// This is in the unit of 1/fm
double RadialEqFunctions_V(double r)
{
 double f;
 double A, R_A, a, R0, V0;

// TD: Implement the following
// if PARAM_DATA.nucA is 0.0, then use the Coulomb potential
// else, use the Nuclear Woods-Saxon potential

// For testing purpose, we use the Coulomb potential for hydrogen atom
// f = -alpha_EM/r;
// Nuclear Woods-Saxon potential
// V0 = 50.0/hbarc; // MeV to 1/fm
// a = 0.7;
// R_A = PARAM_DATA.r0;
// f = -V0/(1.0 + exp((r-R_A)/a));

 return f;
}// RadialEqFunctions_V

// The rest should already be there
```

The above are all minor modifications of what we already have. The main part of this section is in `schroedinger.c`. First the header:

```
// File schroedinger.h
#ifndef SCHROEDINGER_H
#define SCHROEDINGER_H
#include "params.h"
```

```
#include "numerov_params.h"

void Schroedinger_GetBoundState
(DynamicVars *Dyn_Vars, NumerovParams *Num_Params_f,
 NumerovParams *Num_Params_b, double *yf, double *yb);

#endif
```

It has only one function that is accessible from outside `schroedinger.c`. `schroedinger.c` is as follows:

```
// File schroedinger.c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "numerov.h"
#include "numerov_params.h"
#include "radial_eq_functions.h"
#include "params.h"
#include "schroedinger.h"
#include "solve.h"
#include "init.h"

// Common variables to be used only within this file
// These variables are needed because our equation solvers
// Solve_Bisect and Solve_Newton needs a function that is
// a function only of a single double argument.
//
NumerovParams *COM_NUM_PARAMS_F;
NumerovParams *COM_NUM_PARAMS_B;
DynamicVars *COM_DYN_VARS;
double *COM_Y_F; // Forward wavefunction
double *COM_Y_B; // Backward wavefunction

// These functions are needed only within this file
double Schroedinger_GetDf_nmax(double *y, NumerovParams *Num_Params);
void Schroedinger_InitializeCommonVar(void);
double Schroedinger_GetError(void);
void Schroedinger_EvolveForward(void);
```

```
void Schroedinger_EvolveBackward(void);
void Schroedinger_CalcRunScales(double Et);
double Schroedinger_CalcRc(double Eb, double r_init);
double Schroedinger_GetBoundStateError(double Et);
void Schroedinger_PlotData(double Et_min, double Et_max);


void Schroedinger_GetBoundState
(DynamicVars *Dyn_Vars, NumerovParams *Num_Params_f,
 NumerovParams *Num_Params_b, double *yf, double *yb)
{
 double Et_min, Et_max, tol, err;
 int count;
 double x, y;

 COM_NUM_PARAMS_F = Num_Params_f;
 COM_NUM_PARAMS_B = Num_Params_b;
 COM_DYN_VARS = Dyn_Vars;
 COM_Y_F = yf;
 COM_Y_B = yb;

 Et_min = Dyn_Vars->Et_min;
 Et_max = Dyn_Vars->Et_max;

// To plot the error err = u'_I/u_I - u'_II/u_II at x = xc
//
 Schroedinger_PlotData(Et_min, Et_max);

// Schroedinger_GetBoundStateError returns
// err = u'_I/u_I - u'_II/u_II at x = xc
 count = 0;
 tol = 1.0e-6;
 Solve_Bisect(0.0, Schroedinger_GetBoundStateError, Et_min, Et_max, tol, &count);
 fprintf(stderr, "count = %d\n", count);

 return;
}// Schroedinger_GetBoundState
```

```
// To plot
// err = u'_I/u_I - u'_II/u_II at x = xc
// between Et_min and Et_max
void Schroedinger_PlotData(double Et_min, double Et_max)
{
  FILE *output;
  int n, nmax;
  double dEt, Et, err;

  // TD: Open a file to record this data and put it to output
  // Choose the file name according to the naming convention

  // TD: Set nmax = 1000;
  // TD: Set dEt = (Et_max - Et_min)/(nmax)

  for(n=0; n<=nmax; n++)
  {
   // TD: Set Et = Et_min + n*dEt
   err = Schroedinger_GetBoundStateError(Et);

   // TD: Print Et and err to output
   fprintf(output, "%e  %e\n", Et, err);
  }
  // TD: Don't forget to close the file

  return;
}// Schroedinger_Plot_Data


// This function returns
// err = u'_I/u_I - u'_II/u_II at x = xc
double Schroedinger_GetBoundStateError(double Et)
{
 double err;

 Schroedinger_CalcRunScales(Et);
 Schroedinger_InitializeCommonVar();
```

```
 Schroedinger_EvolveForward();
 Schroedinger_EvolveBackward();

 err = Schroedinger_GetError();

 return err;
}// Schroedinger_GetBoundStateError


// TD: Get the classical turning point r_c
// and use it to set xc and xf
//
void Schroedinger_CalcRunScales(double Et)
{
 double r_init, r_min, r_c, Ea, Eb;

 Ea = PARAM_DATA.Ea; // Energy scale
 Eb = Et*Ea;         // Energy corresponding to Et

 // TD: Get COM_DYN_VARS->kb = sqrt(2.0*Eb*mass)
 // getting mass from PARAM_DATA

 // TD: Set Et in COM_Dyn_Vars to be Et
 // TD: Set Eb in COM_Dyn_Vars to be Eb

 // TD: Set r_min to be r0 from PARAM_DATA
 // TD: Set r_init to be 1.1*r_min

 // This calculates the classical turning point
 // by solving -Eb = V(r) + ell(ell+1)/(2 mass r^2)
 r_c = Schroedinger_CalcRc(Eb, r_init);

 // TD: Set rc in COM_DYN_VARS to be r_c
 // TD: Set xc in COM_DYN_VARS to be r_c*ka
 // getting ka from PARAM_DATA

// The wavefunction behaves like exp(-(kb/ka)*x)
// where kb = sqrt(2*mass*Eb)
```

```
// We take x_f = 20*(ka/kb)

 // TD: Set COM_DYN_VARS->xf to be 20 times ka/kb.
 // getting ka and kb from appropriate parameter structures
 // TD: Set COM_DYN_VARS->rf to be xf/ka
 // getting ka and xf from appropriate parameter structures

 return;
}// Schroedinger_CalcRunScales


// Solve V(r) + ell(ell+1)/(2mu r^2) = -Eb
// This is given
double Schroedinger_CalcRc(double Eb, double r_init)
{
 double f, E_min, r_c, tol;
 int count;
 // solve -Eb = V_eff(r)

 tol = 1.0e-8;
 count = 0;
 r_c = Solve_Newton(-Eb, RadialEqFunctions_Veff, r_init, tol, &count);

 return r_c;
}// Get the turning point larger than Rmin

// Initialize all other necessary variables
void Schroedinger_InitializeCommonVar(void)
{
 double kb, rf, h;

// For the forward evolution, x_i = 0 and x_f = xc
// y_0 = y[0] = 0 and y_1 = y[1] can be any small number
// We set it to 0.1

// TD: Set COM_NUM_PARAMS_F->x_i to be 0.0
// TD: Set COM_NUM_PARAMS_F->x_f to be COM_DYN_VARS->xc
// TD: Set COM_NUM_PARAMS_F->y_0 to be 0.0
```

```
// TD: Set COM_NUM_PARAMS_F->y_1 to be 0.1
// TD: Set COM_NUM_PARAMS_F->h to be (x_f - x_i)/(COM_NUM_PARAMS_F->nmax);

// Backward evolution params
// The wavefunction behaves like exp(-(kb/ka)*x)
// where kb = sqrt(2*mass*Eb)
// We take x_f = 20*(ka/kb)
// so that y[0] = exp(-20) = 2E-9
// and y[1] = exp(-(kb/ka)*(x_f-h))
// y_0 = y[0] should be a small number and
// y_1 = y[1] should be a small number > y[0]
// The x range is xc < x < xf
// or 0 < x' < xf-xc

// TD: Set COM_NUM_PARAMS_B->x_i to be 0.0;
// TD: Set COM_NUM_PARAMS_B->x_f to be COM_DYN_VARS->xf - COM_DYN_VARS->xc;
// TD: Set COM_NUM_PARAMS_B->h to be (x_f - x_i)/(COM_NUM_PARAMS_B->nmax);

 kb = COM_DYN_VARS->kb;
 rf = COM_DYN_VARS->rf;
 h  = COM_NUM_PARAMS_B->h;

// TD: Set COM_NUM_PARAMS_B->y_0  to be exp(-kb*rf);
// TD: SEt COM_NUM_PARAMS_B->y_1  to be exp(-kb*(rf-h));

 return;
}// Schroedinger_Initialize


// This is given
void Schroedinger_EvolveForward(void)
{
 double f, df;
 int nmax;
 double *yf;
 NumerovParams *Num_Params_f;
 DynamicVars *Dyn_Vars_f;
```

```
 yf = COM_Y_F;
 Num_Params_f = COM_NUM_PARAMS_F;
 Dyn_Vars_f = COM_DYN_VARS;

 nmax = Num_Params_f->nmax;

 Num_Params_f->NumerovFunc_F = RadialEqFunctions_F_Forward;

 Numerov_Advance(yf, Num_Params_f, Dyn_Vars_f);

 return;
}// Schroedinger_EvolveForward


// This is given
void Schroedinger_EvolveBackward(void)
{
 double f, df;
 int nmax;
 double *yb;
 NumerovParams *Num_Params_b;
 DynamicVars *Dyn_Vars_b;

 yb = COM_Y_B;
 Num_Params_b = COM_NUM_PARAMS_B;
 Dyn_Vars_b = COM_DYN_VARS;

 nmax = Num_Params_b->nmax;

 Num_Params_b->NumerovFunc_F = RadialEqFunctions_F_Backward;

 Numerov_Advance(yb, Num_Params_b, Dyn_Vars_b);

 return;
}// Schroedinger_EvolveBackward
```

```
// This implements the numerical derivative at the end of the list.
// That is, given y[n-2], y[n-1], y[n], get the slope at y[n].
// Given these 3 points, the slope at x_n is given by
// y'[n] = (3y[n] - 4y[n-1] + y[n-2])/(2h)
//
double Schroedinger_GetDf_nmax(double *y, NumerovParams *Num_Params)
{
 double df, h;
 int nmax;

 nmax = Num_Params->nmax;
 h = Num_Params->h;

// TD: Implement
// df = (3y[n] - 4y[n-1] + y[n-2])/(2h)
// with n = nmax

 return df;
}// Schroedinger_GetDf_nmax


double Schroedinger_GetError(void)
{
 double *yf, *yb;
 NumerovParams *Num_Params_f, *Num_Params_b;
 double df, df_f, df_b;

 Num_Params_f = COM_NUM_PARAMS_F;
 Num_Params_b = COM_NUM_PARAMS_B;
 yf = COM_Y_F;
 yb = COM_Y_B;

// TD: Get the derivative at nmax using yf and Num_Params_f
// TD: Devide the derivative by yf[nmax] and put it in df_f

// TD: Get the derivative at nmax using yb and Num_Params_b
// TD: Devide the derivative by yb[nmax] and put it in df_b
// TD: Remember that for the backward evolving solution,
```

```
// TD: r decreases as n increases.

 df = df_f - df_b;

 return df;
}// Schroedinger_GetError
```

That's it. We are done. So the `.c` files you need to compile the final program are

```
SRCS= \
derivatives.c extremum.c init.c  \
main_schroedinger.c numerov.c radial_eq_functions.c  \
schroedinger.c solve.c vector_mtx.c
```

Testing your Schroedinger Equation Solver

Let's first see if we can reproduce Coulomb potential case. First, the numerov part of the input

```
500
500
```

The first `500` is the `nmax` for the forward evolving wavefunction (`u_I`) and the second `500` is the `nmax` for the backward evolving wavefunction (`u_II`). I've named it `input_n_params`.

The mass of the electron is $m_e = 0.511 \times 10^5$ eV. Hence, we use the energy scale `eV` and the length scale `nm`. The input file for the Coulomb part is

```
0.511e+6
eV
0
0
0
0.01
1.1
```

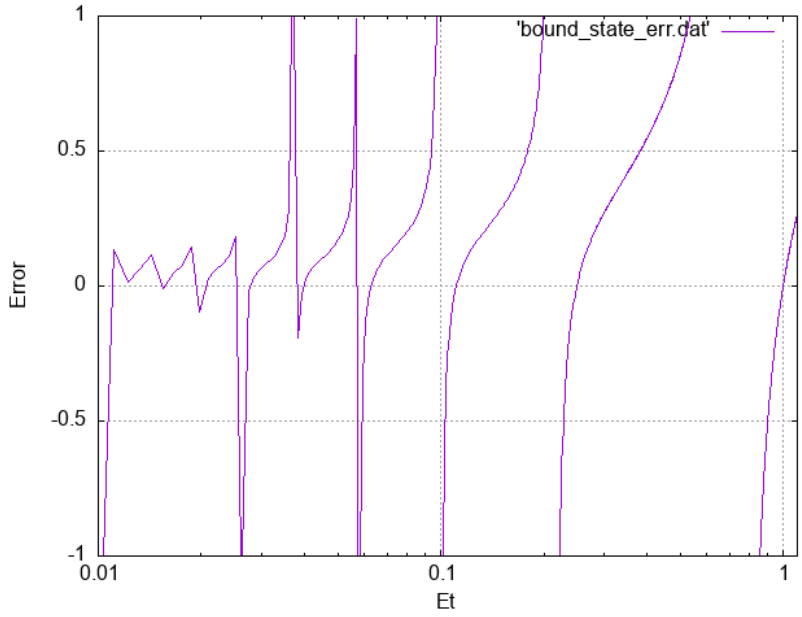
The first number is the mass, `eV` is the unit we wish to use, the three `0`'s are `ell`, `nucA` and `nucZ` respectively. Note that we put both `nucA` and `nucZ` to be zero. This means that we are choosing the coulomb potential parts

in `init.c` and `radial_eq_functions.c`, etc. The last two numbers, `0.01` and `1.1` indicates the `Et` range we wish to consider. I am naming this file `input_coulomb`.

The `Et` range above is chosen to contain most of the possible solutions. This is We first plot the quantity `error = u'_I/u_I - u'_II/u_II` as a function of `Et`. With the above parameters, when I run

```
schroedinger input_coulomb input_n_params
```

I get (`schroedinger` is my executable name)



It also gives me

```
Record_Results: Et = 2.772197e-02
Record_Results: Eb = 3.773756e-01
```

as a solution. Note that

$$\frac{1}{0.02772197} = 36.07 \approx 6^2 \tag{53}$$

We know that the hydrogen atom spectrum is

$$E_n = -\frac{13.6\,\mathrm{eV}}{n^2} \tag{54}$$

Since our $E_a = 13.6\,\mathrm{eV}$, this is at least one of the right answers.

There are few things one should notice in the figure above. One is that there are certainly many zeros. Starting from one close to $E_t = 1$, there are zeros close to, $0.25, 0.111, \cdots$. Another thing to notice is that there are points where the error is diverging. This is when the value of E_t is making $u_I(x_c) \to 0$. The sign of $u_I(x_c)$ changes as E_t crosses the value that makes $u_I(x_c) = 0$ but the value and the sign of $u_I'(x_c)$ will not change much. That's what is making the divergence near $E_t = 0.75, 0.214, \cdots$. Across these points, even the error changes sign, there is no zero.

Once we have this plot, we can find each eigenvalue by bracketing the range of E_t. For instance, since we know there is a zero near $E_t = 1$, we can try `Et_min = 0.9`, `Et_max = 1.1`. This gives

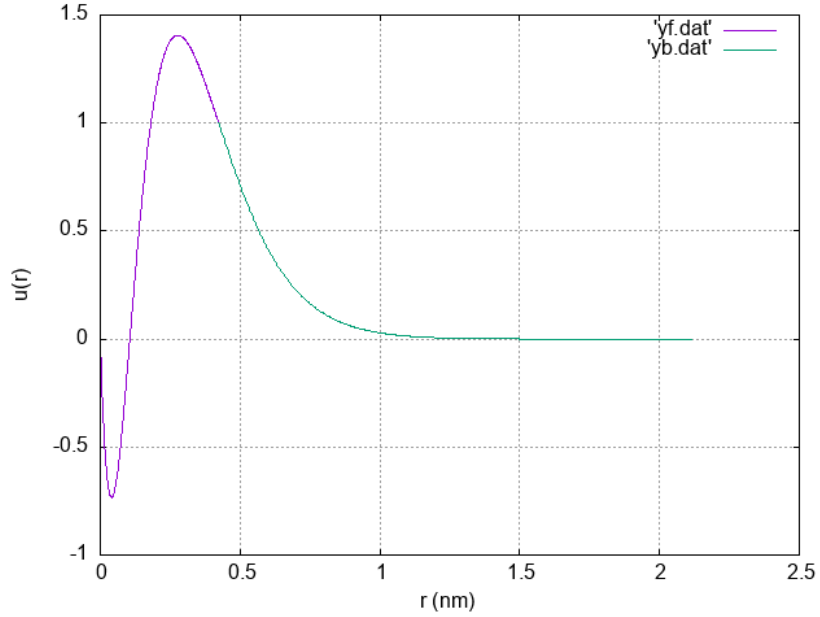
```
Record_Results: Et = 9.999302e-01
Record_Results: Eb = 1.361192e+01 eV
```

which is very close to the groud state energy $E_t = 1$.

The table also indicates that there is a zero between $E_t = 0.25089$ and $E_t = 0.2498$. Using `Et_min = 0.24`, `Et_max = 0.26`, I get

```
Record_Results: Et = 2.499753e-01
Record_Results: Eb = 3.402882e+00 eV
```

which is very close to the first excited state energy of $E_t = 1/4$. Ploting `yf.dat` and `yb.dat` for `Et = 2.499753e-01` gives me the wavefunction (not normalized)



To test:

1. Reproduce the above results. Submit the error plot with `Et_min = 0.01` and `Et_max = 0.9999`. Submit the result files for the first 4 energy levels. Name your files appropriately. Submit the plot of the wavefunctions associated with `Et = 1/4`.

2. Change `ell` to 1 and repeat the above analysis for the first 3 energy levels. Submit the plot of the wavefunctions for the lowest `Et` you found.

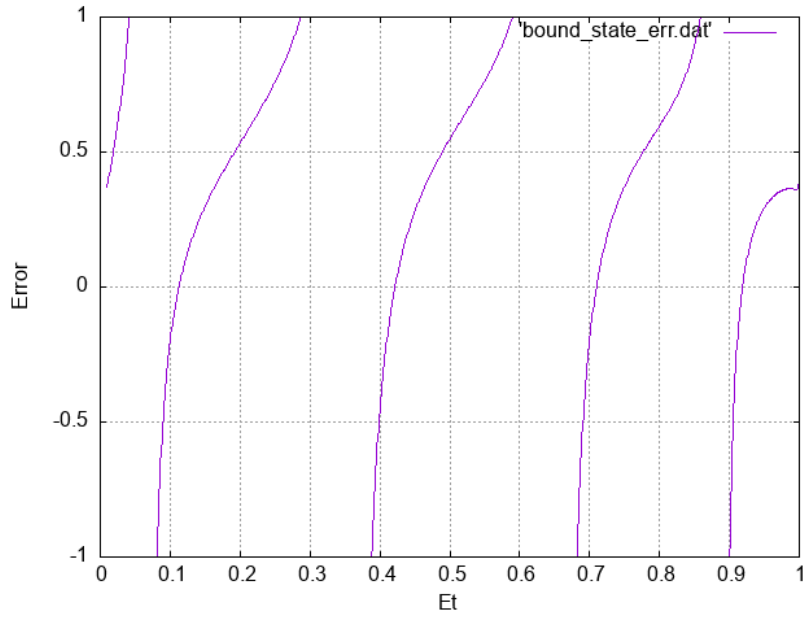
Energy levels of the Woods-Saxon potential

Now let's change gear. The input file `input_nuc` reads

```
940.0
MeV
0
208
```

```
82
0.1
0.9999
```

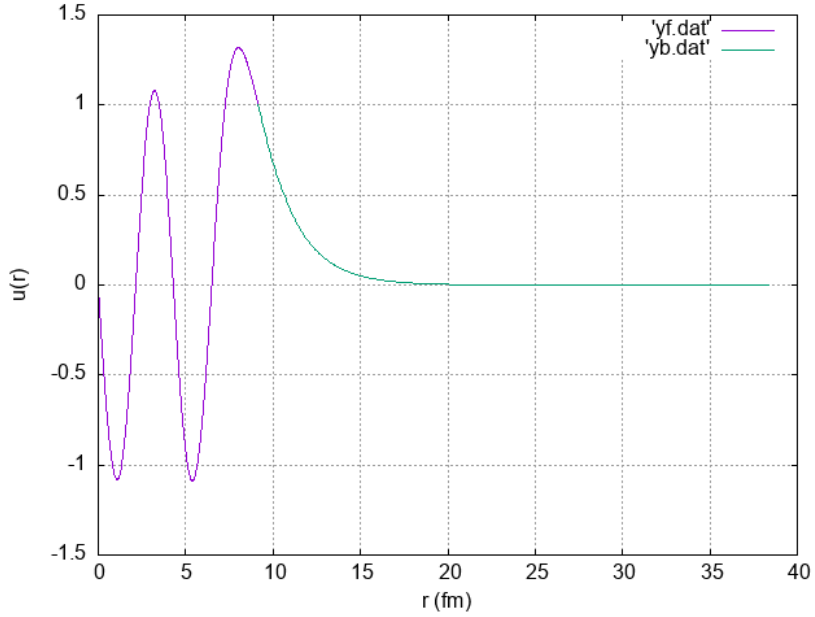
which sets `ell = 0`. This will not give you any energy eigenvalues but it does produce the following plot



Looking up the data file, I find that there is a zero between `Et = 0.918` and `0.919` and other zeros within `(0.709, 0.710)`, `(0.421, 0.422)`, and `(0.112, 0.113)`. Using these values as `Et_min` and `Et_max`, I get

```
Record_Results: Et = 9.186995e-01
Record_Results: Et = 7.098093e-01
Record_Results: Et = 4.215432e-01
Record_Results: Et = 1.126115e-01
```

The wavefunction for the last one is



To test:

1. Reproduce the above results.
2. Change `ell = 1` and repeat the above analysis for the first 3 energy levels.

A Vector and Matrix functions

Here are the vector and matrix dynamic memory allocation functions.

```
// File vector_mtx.h
#ifndef VECTOR_MTX_H
#define VECTOR_MTX_H

double *vector_malloc(int nmax); // allocates memory space for 1D arrays
double **mtx_malloc(int mmax, int nmax); // allocates memory space for 2D arrays
void mtx_free(double **mtx, int mmax);   // frees the memory space allocated by
                                         // mtx_malloc
```

```
#endif

// File vector_mtx.c
#include <stdio.h>
#include <stdlib.h>
#include "vector_mtx.h"

// allocate memory space for a 1D array
double *vector_malloc(int nmax)
{
 double *pt;
 int n;

 pt = (double *)malloc(sizeof(double)*nmax);

 // initialize all entries to 0
 for(n=0; n<nmax; n++) pt[n] = 0.0;

 return pt;
}// vector_malloc


// allocate memory space for a 2D array
double **mtx_malloc(int mmax, int nmax)
{
 double **pt;
 int m, n;

 pt = (double **)malloc(sizeof(double *)*mmax);

 for(m=0; m<mmax; m++)
  {
   pt[m] = (double *)malloc(sizeof(double)*nmax);
  }// m-loop

 // initialize all entries to 0
 for(m=0; m<mmax; m++)
```

```
  {
   for(n=0; n<nmax; n++) {pt[m][n] = 0.0;}
  }// n-loop

 return pt;
}// mtx_malloc

// free the memory space allocated by mtx_malloc
void mtx_free(double **mtx, int mmax)
{
 int m;

 for(m=0; m<mmax; m++) { free(mtx[m]); }
 free(mtx);

 return;
}// mtx_free
```