CO22 Report

HENRY GRIFFITHS / TRINITY COLLEGE / trin3161

The first part of this project I worked on was the Numerov algorithm itself.

1- Numerov Solver

To find `f'(x)` and `f''(x)`, it would have been possible to make a very case-specific expression making the assumption that we know `f(x)` will be in the form `f=x^2-E`. But this didn't seem satisfactory and I could easily see it causing problems if anyone ever tried to reuse/alter my code for a different purpose. So instead I set up 3 symbolic variables to find the differentiation in the more general case and assign each to a lambda function:

```
39 -    syms x;
40 -    syms y;
41 -    syms z;
42 -    d1= diff(f(x));       %symbolic equation for f'(x)
43 -    d1f= @(y) subs(d1,y); %define f'(x) as lambda function
44 -    d2 = diff(d1f(y));    %symbolic equation for f''(x)
45 -    d2f= @(z) subs(d2,z); %define f''(x) as lambda function
```

We know the Psi(0) and dPsi(0) conditions, but we still need to find Psi(delta). This is done via Taylor Expansion around zero, assuming that delta is small and so close to 0.

Initially I wanted to use an if statement to split the program into two more simpler taylor expansions depending on if n was odd or even, as the script suggested, but confusingly the specific way the `numerov_function` is required to operate (Appendix A.1) made this impossible.

The required `numerov_function` is not allowed to take n as an input, so there was no way for me to make this refinement. Instead I just did the full Taylor Expansion, which looks messy but is simply the `n=odd` and `n=even` cases added together.

```
49 -    TaylorEven = psi0+(delta.^2)*f(0)*psi0/2+(delta.^4)*(d2f(0)*psi0+2*d1f(0)*dpsi0+((f(0)).^2)*psi0 )/24;
50 -    TaylorOdd  = (delta*dpsi0 + (delta.^3)*(f(0)*dpsi0+d1f(0)*psi0) )/6;
51
52      %Sum the two parts together for our approximation of psi(delta)
53 -    psi1 = double(TaylorEven + TaylorOdd);
```

Using a linearly spaced vector of x-values, it was then simple enough to iterate the Numerov Algorithm starting from the 2nd iteration (we used our initial conditions for the starting steps) and place the results in a vector named NumericalSol.

```
71 -    for j=2:jmax
72 -        A =  (1 - (delta.^2)*NumericalSol(j+1,2)/12);
73 -        B =  (2 + (delta.^2)*NumericalSol(j ,2)*5/6);
74 -        C = -(1 - (delta.^2)*NumericalSol(j-1,2)/12);
75          %Actual Numerov iteration step:
76 -       NumericalSol(j+1,1)= (B*NumericalSol(j,1) + C*NumericalSol(j-1,1))/A;
77 -    end
```

That was effectively all that had to be done to create our desired output vector 'psi'.

```
%-----------------------
%OUTPUT
%-----------------------
psi = NumericalSol(:,1); %output array as desired
```

Next I wrote the CO22 function:

2- CO22:

To prepare for the `numerov_solver` functioncall, Psi(o) and dPsi(o) had to be found, which was done through a simple `if/else` statement to check if `n` was even or odd for the given parameters.

```
%Set initial conditions
if mod(n,2) == 0; %i.e: if n is even
    psi0 = 1;
    dpsi0 = 0;
else
    psi0 = 0;
    dpsi0 = 1;
end
```

Due to having done the legwork earlier, the functioncall is exceptionally simple:

```
%---------------------
%NUMEROV FUNCTIONCALL
%---------------------

psi = solve_numerov(f, x, psi0, dpsi0); %Functioncall!

```

The analytic solution was then computed to allow for visual comparison. Instead of creating a 1x1 array and expanding it as needed, I created a zero-vector with values to be filled in. Due to the way MatLab handles arrays, this is much faster than the former - changing matrix sizes is a huge timesink in MatLab programmes.
After this long zero-vector is created, each value is iteratively calculated and filled in with the help of MatLab's inbuilt "`hermiteH`" function.

```
AnalyticSol=zeros(jmax+1,1); %Avoid concatenation to increase speed
    for i=0:jmax
        AnalyticSol(i+1,1) = hermiteH(n,i*delta)*exp(-((i*delta).^2)/2);
    end
```

There will be a constant scaling error involved with the comparison between the numerical and analytic solutions, which is mathematically insignificant but could be visually confusing, so I decided to automatically scale the graphs together.
My first thought was to find the mean ratio of the two graphs at each x-value, but I ran into problems. The issue is that, once you introduce non-perfect E eigenvalues, there is always a large spike in the error near the end of the x-values, which throws off any mean value you take.

So instead I came up with a slightly unusual solution, where I rounded the error for each point to 2 significant figures (an arbritarily chosen, changeable specification) and took the *modal* average, which worked perfectly as the errorspike is so sharp that no two points in that error zone have the same error ratios and are passed over for the modal average.

```
59 -    error = psi./AnalyticSol; %note this is a vector of errors
60 -    errorratio = round(error,2,'significant'); %Round to 2 sig. figs (changeable
61 -    scale = mode(errorratio); %We take the mode because the error hugely spikes
```

I then plotted the two solutions on the same graph, the numerical solution with a thick blue line and the analytic with a thinner dashed red one.

```
65 -    plot(x, psi); %plot numerical solution (blue)
66 -    hold on;
67 -    plot(x,scale*AnalyticSol(:,1),':r'); %plot analytic solution (red, dashed)
68 -    hold off;
69 -    xlabel('x'),ylabel('Psi(x)'),axis square, grid on
```

All that was required to finish this function was an initial step at the beginning to find E, instead of assuming the user is capable of putting in an exact eigenvalue every time. This was done with `solve_eigenvalue`.

3- Solve Eigenvalue:

There were lots of much better ways to handle this function, for example by integrating it more fully with the main code to reduce the number of times the `numerov_solver` function had to be called. But the specific constraints on the function required it to work with ONLY the energy eigenvalue guess inputted.
In the end in order to make it work a little nicer I ended up putting in an optional input argument for "`n`" as well - this program still completely functions if no "`n`" is inputted, so I thought that would be fine and within the bounds of the brief. I detected if such an input was present or not with the following code:

```
37      %n detection
38 -    if nargin == 1 %detect if n is inputted or not
39 -        psi0 = 0; %DEFAULT to odd n
40 -        dpsi0 = 1;
41 -    else
42 -        if mod(n,2) == 0; %i.e: if n is even
43 -        psi0 = 1;
44 -        dpsi0 = 0;
45 -        else
46 -        psi0 = 0;
47 -        dpsi0 = 1;
48 -        end
49 -    end
```

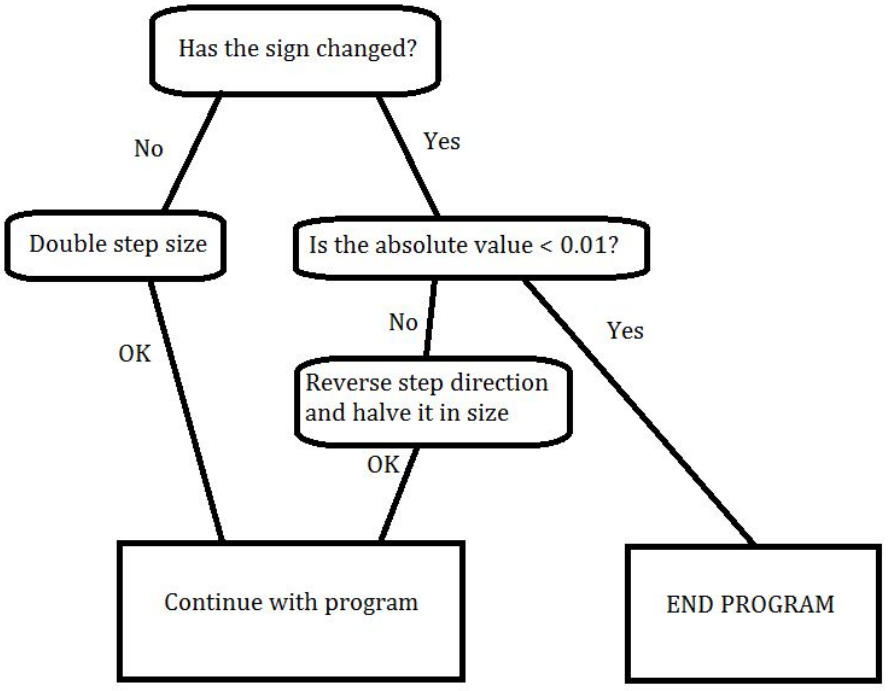
Also at the beginning of my code several arbitrary variables were chosen, in this case the ones recommended in the script (although they could be changed for increased accuracy/speed if the user wanted).

```
17      %-----------------------
18      %SET UP OF KEY VARIABLES
19      %-----------------------
20      %[ sample variables suggested in script ]
21 -    xarray=linspace(0,5,101);
22
23 -    x0= xarray(1);      %Extract the first and last values from the given xarray
24 -    xf = xarray(end);
25 -    delta = (xf-x0)/size(xarray,2); %Extract the spacing
26 -    jmax = size(xarray,2)-1;        %Set up index (note the -1 index correction)
27 -    step = delta;                   %intial step
28 -    accuracy = 0.01;                %choose accuracy desired
29      %choose direction of step (toward closest integer)
30 -    if mod(E0,1) < 0.5 && E0>0.5
31 -        step = -1*step;
32 -    else
33 -    end
```

Note the `if/else` statement in line 30 - this basically sets whether the search algorithm starts looking “up” or “down” for the eigenvalue - it would find the answer without this line, but it would be much slower.

One of the annoying parts of the specifications for the function was that I had to duplicate all the code doing, for example, the Taylor expansions and differentials of `f(x)` and so on. Lines 32-82 are copied exactly from the main `CO22.m` file.

I then introduced the iterative step - the program punts in the value of E and checks the output at x=5, and compares it to the previous output for the previous guess of E. It then makes the following decision tree:



This is represented in the code by:

```
86 -    if PrevFinalPsi/FinalPsi < 0 %CHECK FOR SIGN CHANGE!
87 -            if abs(FinalPsi) < accuracy;
88 -                break %stop when as close as required
89 -            else
90 -                step = -0.5*step; %increase finetuning granularity, REVERSE DIRECTION
91 -            end
92 -    elseif PrevFinalPsi/FinalPsi < 1.2 && abs(PrevFinalPsi/FinalPsi) > 0 %if value is barely changing!
93 -        step = 2*step;
94 -    else %else carry on as usual
95 -    end
96
97
98
99      %update values for next step
100 -    E = E + step;
```

The `break` statement triggers the end of the program and outputs the most recent guess at E as its final answer - this value is guaranteed to give an absolute value of the waveform at `x=5` of less than the "accuracy" variable (by default 0.01).

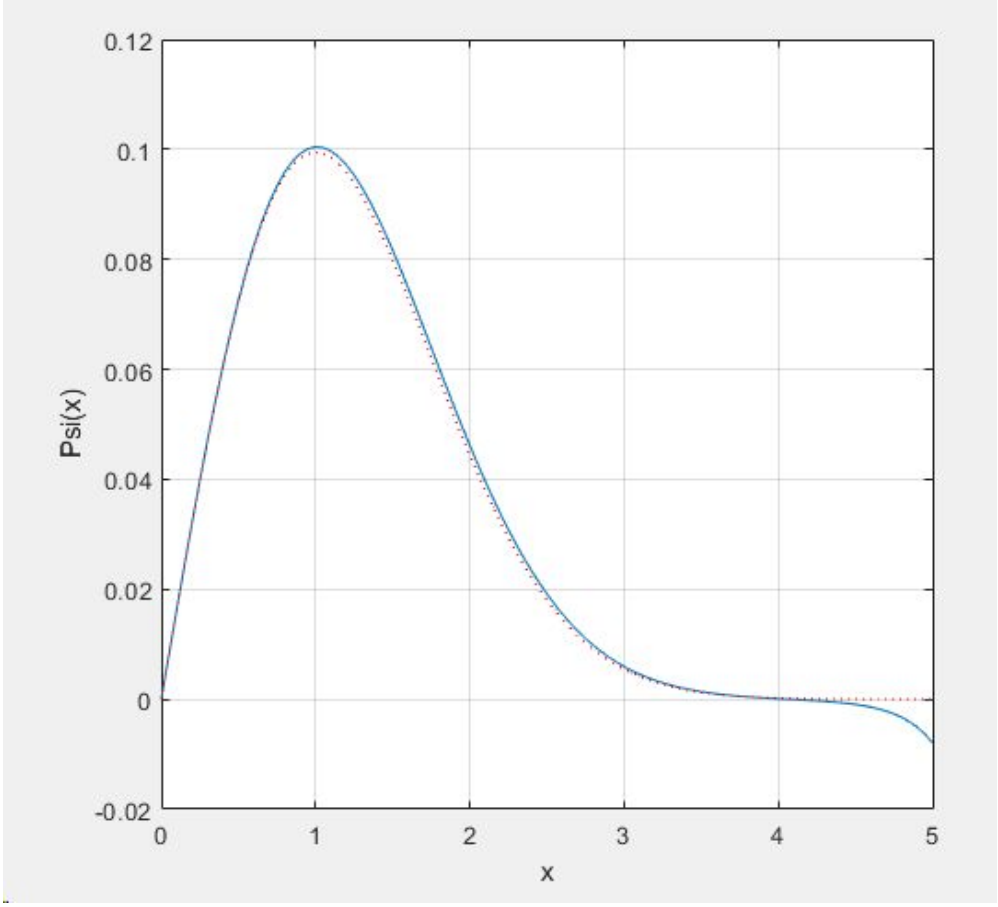
That completes the explanation of how the code was put together. My commenting was extensive, so any further details are included inside the project files themselves.

Sample Usage:

INPUT:

```
CO22 (0.05 , 5 , 1 , 3.1)
```

OUTPUT:



Note the deviation from the analytic solution (red dots) near the end - this deviation can be reduced by setting the accuracy to finer degrees.

4- Questions

3. Show that this leads to...
All that is required is to note that `Ψ''(x)=f(x)Ψ(x)` and substituting that into the general expression of the Taylor Expansion:

$$\Psi(x) = \Psi(0) + x\Psi'(0) + \frac{x^2\Psi''(0)}{2} + \frac{x^3\Psi'''(0)}{6} + \frac{x^4\Psi''''(0)}{24}\cdots$$

$$\Psi(x) = \Psi(0)+x\Psi'(0)+\frac{x^2\Psi(0)f(0)}{2}+\frac{x^3(f'(0)\Psi(0)+f(0)\Psi')}{6}+\frac{x^4(f''(0)\Psi(0)+f'(0)\Psi'(0)+f'(0)\Psi'+f(0)\Psi'')}{24}$$

Then note that in the odd/even cases one half of the equation disappears:
Odd:

$$\Psi(x) = x\Psi'(0) + \frac{x^3(f'(0)\Psi(0)+f(0)\Psi')}{6}$$

Even:

$$\Psi(x) = \Psi(0) + \frac{x^2\Psi(0)f(0)}{2} + \frac{x^4(f''(0)\Psi(0)+f'(0)\Psi'(0)+f'(0)\Psi'+f(0)\Psi'')}{24}$$

4. Why does your numerical solution differ from the analytic solution for large x?
Our numerical solution is not using the exact eigenvalue the same way the analytic one does - the eigenvalue solver function merely gets within a specified range of the answer and then stops to save computation time. This imperfect eigenvalue causes the solution to differ from the analytic one at larger x values.

5. Consider the relative advantages and disadvantages of Numerov's method:
Can be used for situations where no analytic solution exists, but its reliance on eigenvalues makes it difficult to use in more complex situations where eigenvalues may not be so easy to find.
That said, it is often possible to find upperbounds for systems via the "variational method" (to wit: guessing), so this method has more applicability than may at first appear.

5.2. What other numerical methods could be used to solve equation 2?
Probably the most interesting/useful method aside from the Numerov Algorithm would have been a variational one, which is of extra interest due to how easily it can be generalised to 1 or 2 dimensional problems (unlike Numerov's method).

All that is required is a knowledge of the eigenfunctions (i.e: eigenstates) of the system, which can then be used as a basis set which can be used to expand any arbitrary wavefunction at all. This gives an integral which can be evaluated numerically, for example by a Romberg algorithm, to give a good approximate form of the wavefunction.

EG: for a square potential well with eigenstates of `cos(kx)` and `sin(kx)`, the integration is given by:

$$H_{pk} = \langle p|(-\frac{1}{2}\frac{d^2}{dx^2} + V)|k\rangle = K_k + V_c \int_{-a}^{a} dx\phi_p(x)\phi_k(x).$$

Which can be used to find the matrix elements of the Hamiltonian and, using the TISE, the wavefunction. (Source for this integral is “Numerical Solutions of the Schrodinger Equation” by Anders W Sandvik of Boston University)