Physics 24
Computational Project

Sam Greydanus

April 2015

1 Overview

The Schrodinger equation is an enormously effective model of atom-level interactions between subatomic particles. Sometimes, though, discrete solutions to the equation are not possible as in the case of the finite-depth quantum well. In this paper we present a computational model for solving the finite-depth quantum well problem and investigate its properties.

2 Methods

2.1 Overview

This section is a summary of all Matlab functions used for the project. The functions themselves are provided in the Appendix and as separate files in the same folder as this file.

2.2 Plotting ψ a single value of ϵ

To solve for a range of ψ values associated with a particular energy eigenvalue (referred to here as ϵ), we first wrote a Matlab function which takes in a dimensionless position u an returns the associated dimensionless potential V. For the finite well, the function simply returns 0 for any value between $-a/2$ and $a/2$ and 1 for all other values. The advantage of writing this as a separate function is that later we can add more complex potential equations (such as the at of the harmonic oscillator) without changing other sections of code.

Next we solved for a range of ψ values on either side of an initial position. We defined constants for the ground state solution as follows: $\psi_\circ = 1$, $\phi_\circ = 0$, and $\beta = 64$. We used a default gridsize of 2000 and solved for ψ values 5 dimensionless units in either direction from the initial position of 0. Later, we turned many of these values into parameters for the function itself to make it more general. Next, we implemented the two coupled differential equations given on page 3 of the numerical project handout in a for loop. For each iteration of the loop, we solved for the next ψ and ϕ values and saved them for use in the next iteration of the loop. In the actual iteration we used two of these for-loops: one to solve for ψ values on the right side of the initial position and one to solve for ψ values to the left. Finally, we plotted the resulting ψ function if the "flag" input parameter was set to 1 (indicating that the user wanted a plot).

The two Matlab functions described here are thoroughly commented and provided in the appendix. Their names are 'V' and 'fstep' respectively.

2.3 Optimizing ϵ

Searching for optimal ϵ values using the shooting method can be quite tedious when done manually. To avoid this, we wrote two functions to assist us in finding optimal solutions. The first function is called "epsilonCost" and it returns a "cost" value which measures how well a numerical ψ function with a particular ϵ value approximates the real ψ function. In our case, the function that diverges to $\pm\infty$ the most gradually should have the lowest cost. Following this line of reasoning, we let our cost function return the maximum numerically-derived ψ value associated with test ϵ value. Thus when our cost function is at a local minimum, we know that we have converged to a viable eigenvalue.

Next, we wrote a function to optimize our cost function which we called "minimizeEpsilon." This function initiates all the constants needed for the problem, creates a function handle for the cost function (a function handle is a technique in Matlab which allows one to pass one function as a parameter to another function), and then uses a built-in optimization function of Matlab called "fminbnd." This function simply searches a range of input values for a function and returns the input value for which the output value is minimized.

The remainder of this function we devoted to plotting the cost function, debugging, and normalization. Discussion of these features can be found in the following three subsections.

2.4 Plotting the cost function

Plotting the cost function is a simple matter of solving for cost values at many different points and plotting them. Figures 1,4, and 7 show the cost functions for the ground, first excited, and second excited states respectively. The blue circle on each of these figures is the result returned by the "fminbnd" function. Notice that these circles accurately represent solutions to local minima for the cost function. These minima correspond to optimal energy values of the finite quantum well.

2.5 Normalization

To normalize, we computed a numerical integral of ψ^2 over the domain $u = [-1, 1]$, then multiplied our vector of ψ values by the inverse of the square root of this value. This adjusted the vector of ψ values so that a numerical integral of $|\psi^*\psi|$ for the domain $u = [-1, 1]$ yielded 1. This process was coded in Matlab and can be found in the Appendix.

2.6 Debugging

The debugging functionality allows us to plot functions of ψ for a candidate ϵ value beside functions of $\epsilon \pm dx$ when dx is very small. If the ψ function generated by the candidate value of ϵ diverges to infinity more slowly than the others, we can conclude that our ϵ value is accurate to the level of $\pm dx$.

2.7 Final note

Optimizing the cost function of ϵ was far less computationally expensive than plotting said function. Having observed this, we decided to use substantially larger gridsizes when optimizing ϵ so that we could achieve more precise estimates. For this reason, we broke "minimizeEpsilon" into two parts: optimization and display. The optimization section uses very large values for the

gridsize and very small values for du compared to the display section. For example, we optimized the ground-state ϵ using a gridsize of 3000000 but plotted the cost function using a gridsize of 3000.

2.8 Other required discussion from the lab guide

We used values of 1 and 0 for ψ and ϕ respectively when solving for the ground state because, when one refers to the ground state plot in the textbook, these are the values one finds. Parity enables us to do this because the graph has even parity for the ground state and so the slope of ψ should be 0 at the y axis when ψ has has continuous derivatives.

The solution is the ground state because the shape of the ground state ψ function takes this form and because we chose the local minimum of our cost function which could be found closest to 0.

3 Results

3.1 Ground State

When solving for the local minimum which corresponds to the ground state of the system, we first need to investigate the graph of the cost function (Figure 1). We see that there is a local minimum at $\epsilon = 0.098028$. Next, we verified this value by using the debugging code in "minimizeEpsilon" to graph ψ functions for $\epsilon \pm dx$ where dx is very small. We suspect our value of ϵ to be accurate to within four decimal places because our debugging tool showed that our value produced the ψ function which diverged to infinity the most slowly for $dx = 0.0001$. The graph in Figure 3 shows the graph of ψ after optimizing ϵ for the ground state.

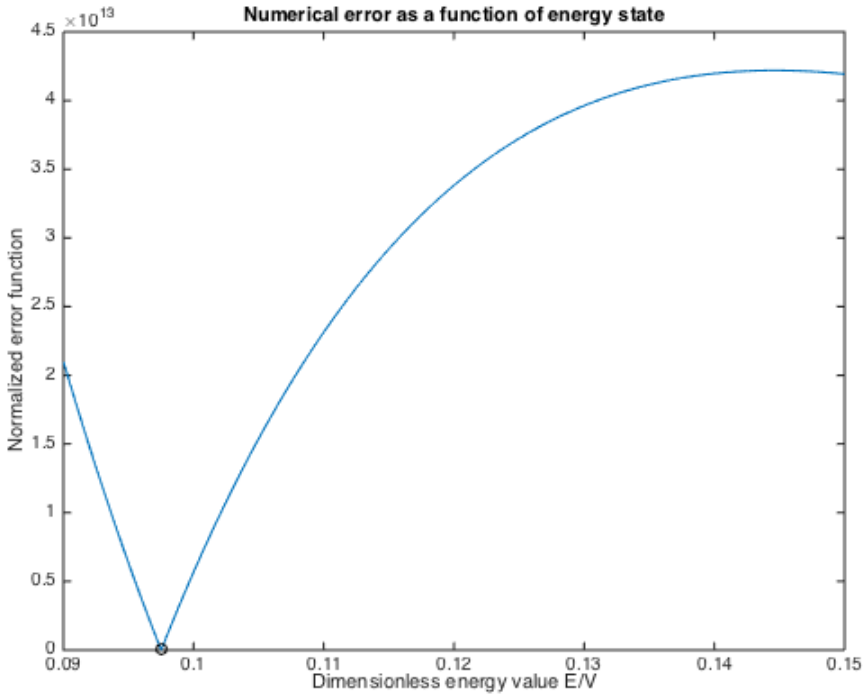
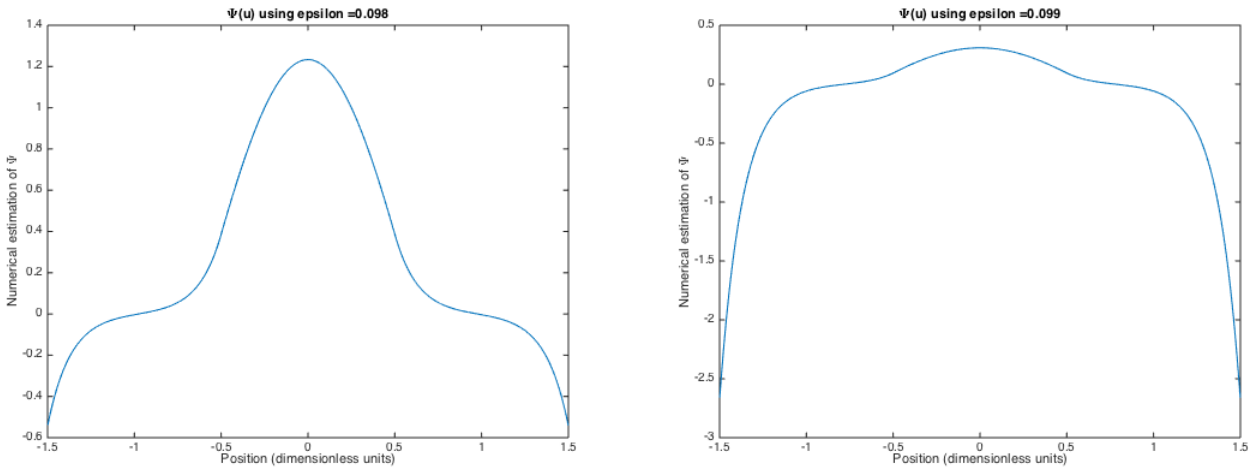


Figure 1: Cost function for first excited state



(a) Below ϵ **(b)** Above ϵ

Figure 2: Graphs of ψ for values above and below the optimal ϵ

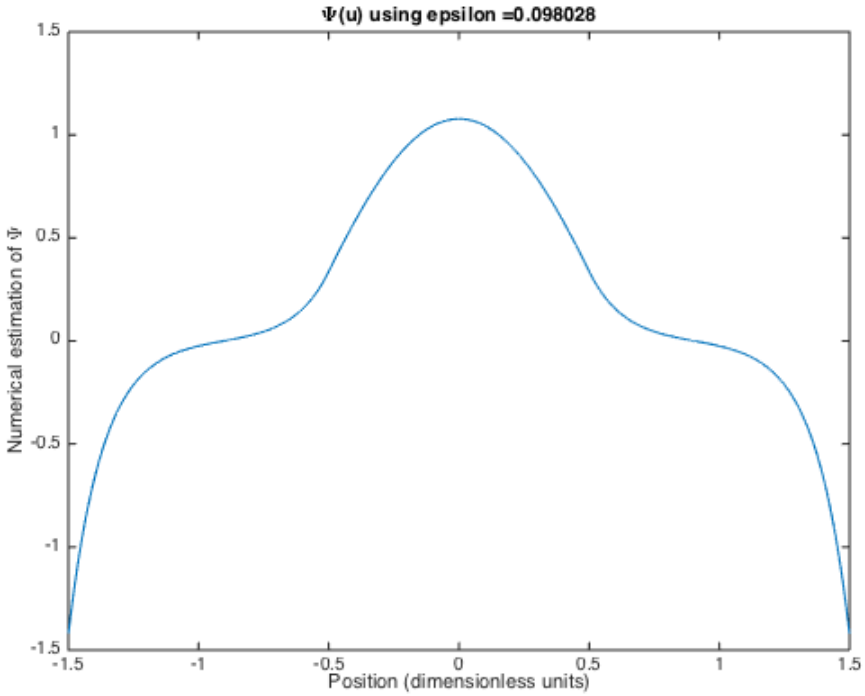


Figure 3: Numerically-derived plot of ψ for $\epsilon = 0.098028$

3.2 First excited state

We used the same process as described for the ground state to solve for the first excited state. We simply changed the initial values to $\psi_\circ = 0$ and $\phi_\circ = 1$ before optimizing epsilon.

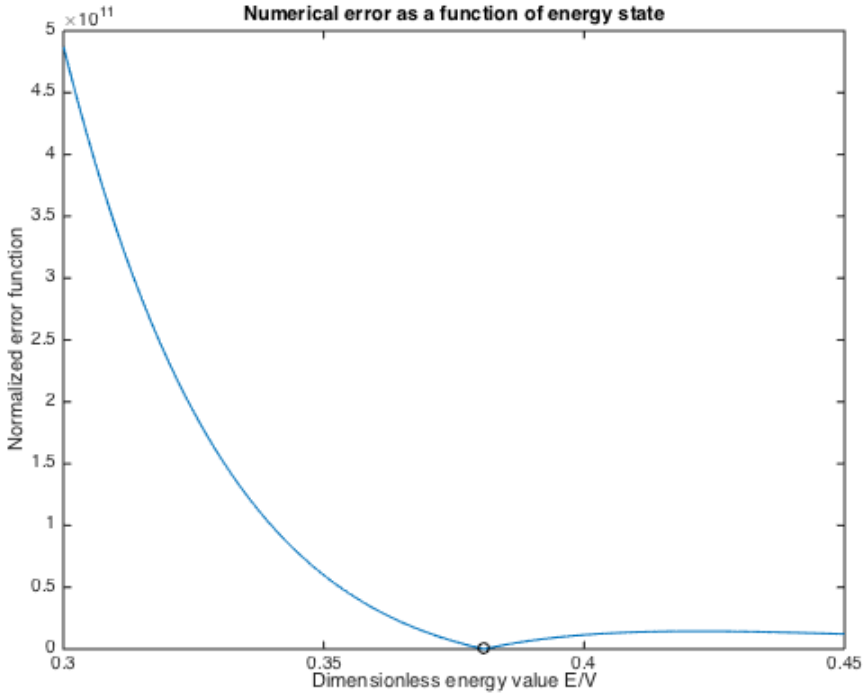
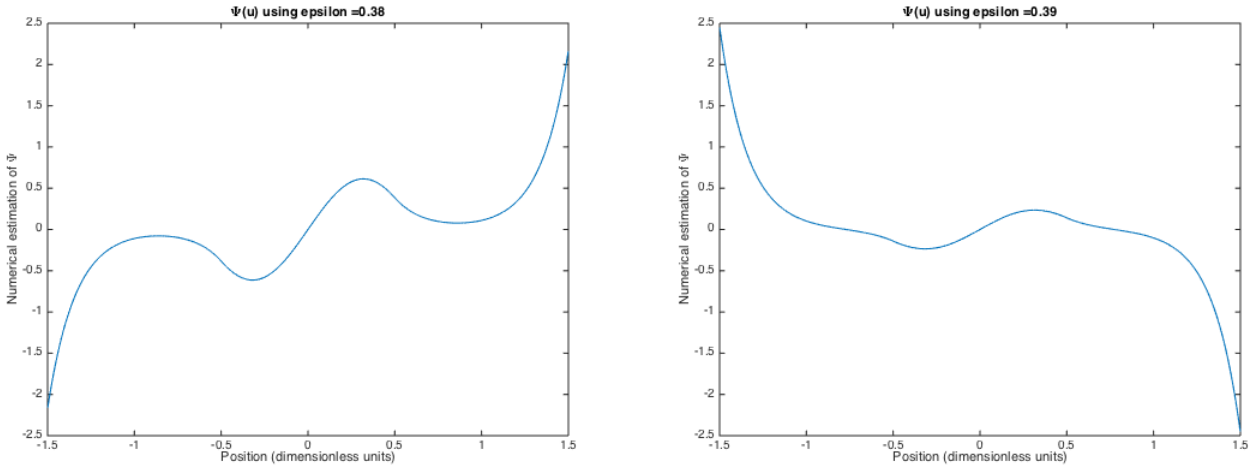


Figure 4: Cost function for first excited state



(a) Below ϵ **(b)** Above ϵ

Figure 5: Graphs of ψ for values above and below the optimal ϵ

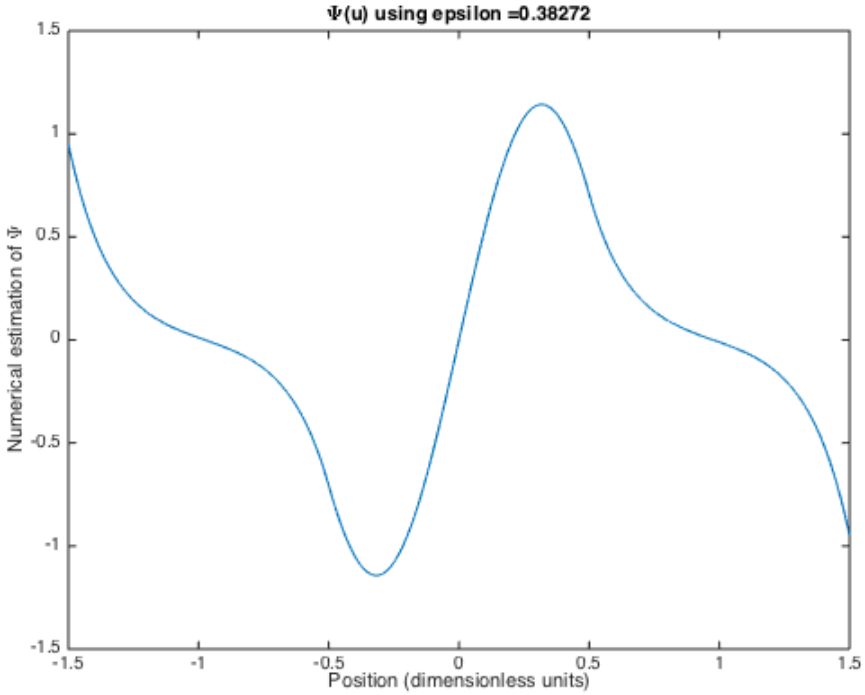


Figure 6: Numerically-derived plot of ψ for $\epsilon = 0.38272$

3.3 Second excited state

We used the same process as described for the ground state to solve for the second excited state. We simply searched a different range of ϵ values ($\epsilon = [0.5, 1.0]$). This yielded a different local minimum which we believe corresponds to the second excited state.

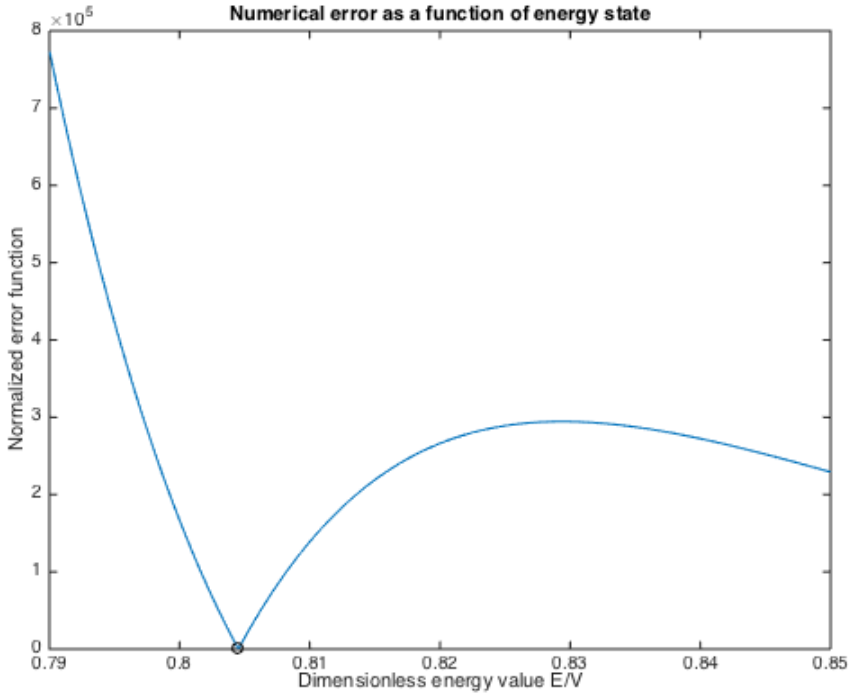
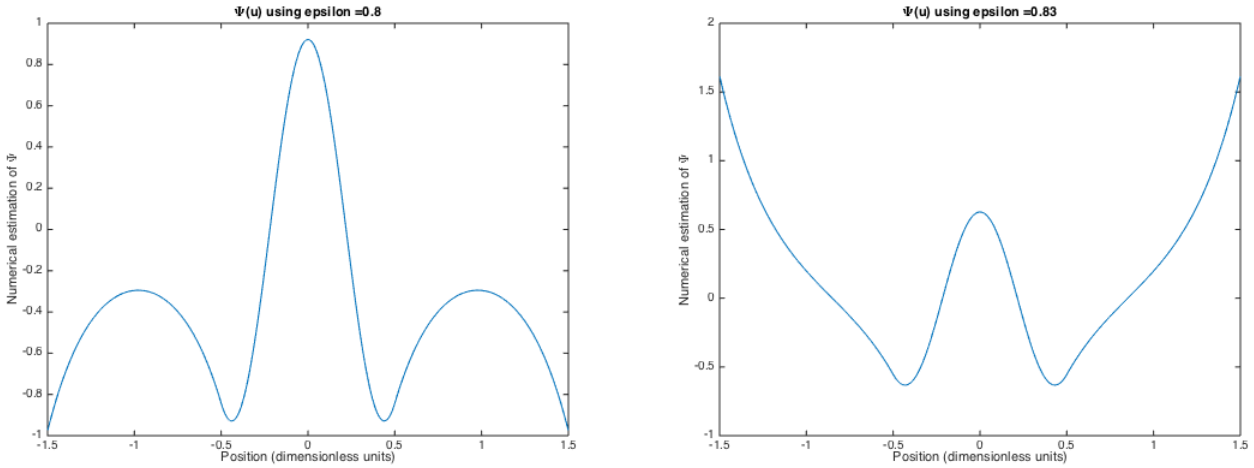


Figure 7: Cost function for first excited state



(a) Below ϵ **(b)** Above ϵ

Figure 8: Graphs of ψ for values above and below the optimal ϵ

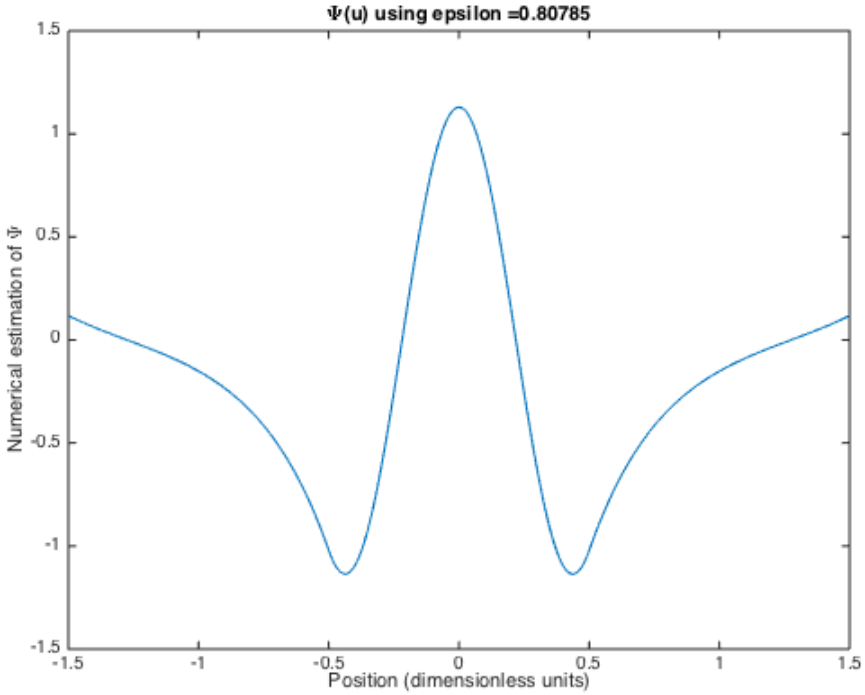


Figure 9: Numerically-derived plot of ψ for $\epsilon = 0.80785$

3.4 Solving for $\epsilon > 1$

Here we carry out problem 2 from ER appendix G. For values of $\epsilon > 1$, the particle is free of the well and does not have quantized energy levels. There are no preferred values for ϵ. The ψ function oscillates about the x-axis as shown in Figure 10. One might notice that in Figure 10, the oscillating function's amplitude increases as it moves away from the y-axis in both directions. This is due to numerical error; there should not be any change in amplitude. When we double the gridsize to 8000, the increase in amplitude is much less visible as shown in figure 11.

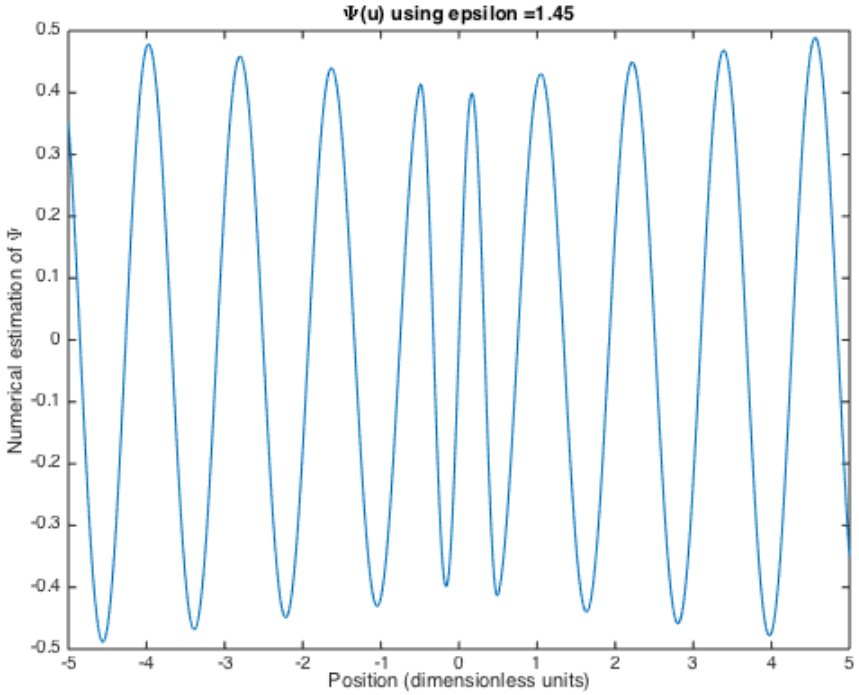


Figure 10: Plot of ψ for $\epsilon > 0$ (gridsize = 2000). Notice how it oscillates around the x-axis and increases in amplitude as it moves away from the y-axis due to numerical error

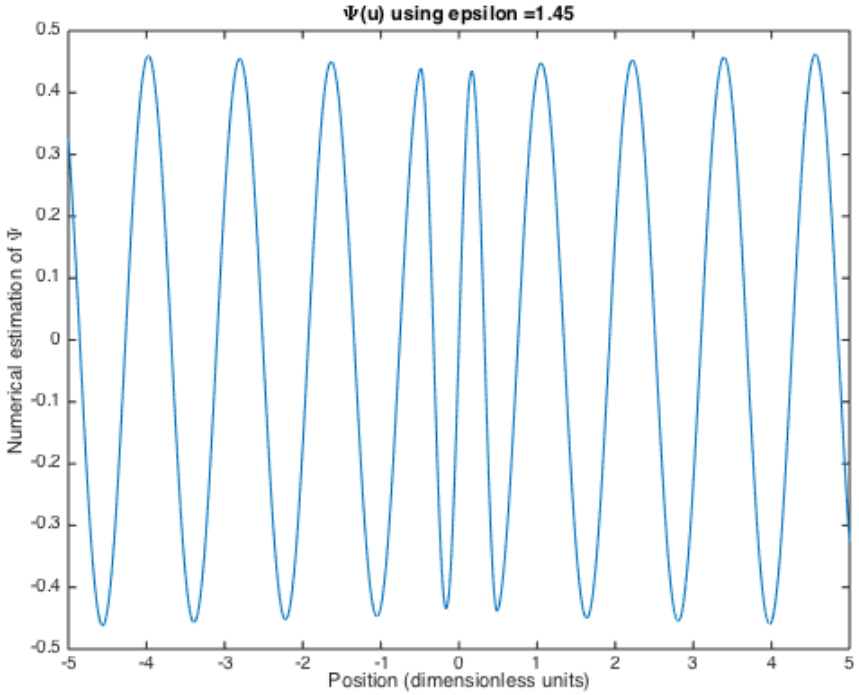


Figure 11: Plot of ψ for $\epsilon > 0$ (gridsize = 8000). Notice that the amplitude does not increase as rapidly as it moves away from the y-axis.

3.5 Converting to real world values

First, recall the equation for $V_\circ$ and let the constants $\beta = 64$, $\hbar = 1.05457 \times 10^{-34}$, $m = 9.109 \times 10^{-31}$, and $a = 100 \times 10^{-9}$. Then we have:

$$V_\circ = \frac{\beta\hbar^2}{2ma^2} = 3.91 \times 10^{-23}J = 2.44 \times 10^{-4}eV \tag{1}$$

If we use $a = 53 \times 10^{-12}$ (the radius of a hydrogen atom) instead, we find $V_\circ = 1.391 \times 10^{-16} = 868eV$.

It is also possible to solve for a by rearranging that equation and letting the constants $\beta = 64$, $\hbar = 1.05457 \times 10^{-34}$, $m = 9.109 \times 10^{-31}$, and $V = 1eV = 6.24 \times 10^{18}J$

$$a = \sqrt{\frac{\beta\hbar^2}{2mV_\circ}} = 4.938 \times 10^{-8}m \tag{2}$$

If we use $V_\circ = 13.6eV$ (the magnitude of the lowest energy state of hydrogen) instead, we find $a = 1.391 \times 10^{-16} = 1.339 \times 10^{-8}m$.

3.6 Quantum harmonic oscillator

The quantum harmonic oscillator potential is

$$V(x) = \frac{C}{2}x^2 \tag{3}$$

and the associated differential equation (taken from appendix I of Eisberg and Resnick) is:

$$\frac{d^2\psi(u)}{d^2u} = -\left(\frac{\gamma}{\alpha} - u^2\right)\psi(u) \tag{4}$$

Where $\gamma = \frac{2mE}{\hbar^2}$ and $\alpha = \frac{2\pi m\omega}{\hbar}$

Rearranging the equation in the same way as shown on the lab guide and letting $\beta = \frac{V_\circ}{\hbar\pi\omega}$, we have

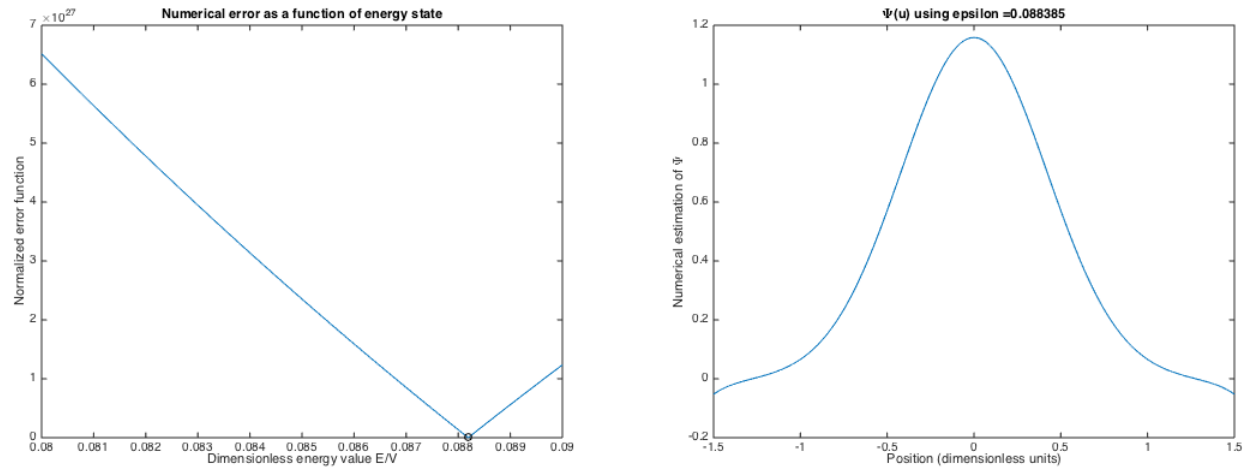
$$\frac{d^2\psi(u)}{d^2u} = -\beta[\epsilon - V(u)]\psi(u) \tag{5}$$

Such that $V(u) = \frac{\hbar\pi\omega^2}{V_\circ}$

$$\phi(u_{i+1}) = \phi(u_i) - \Delta u\beta[\epsilon - V(u)]\psi(u_i) \tag{6}$$

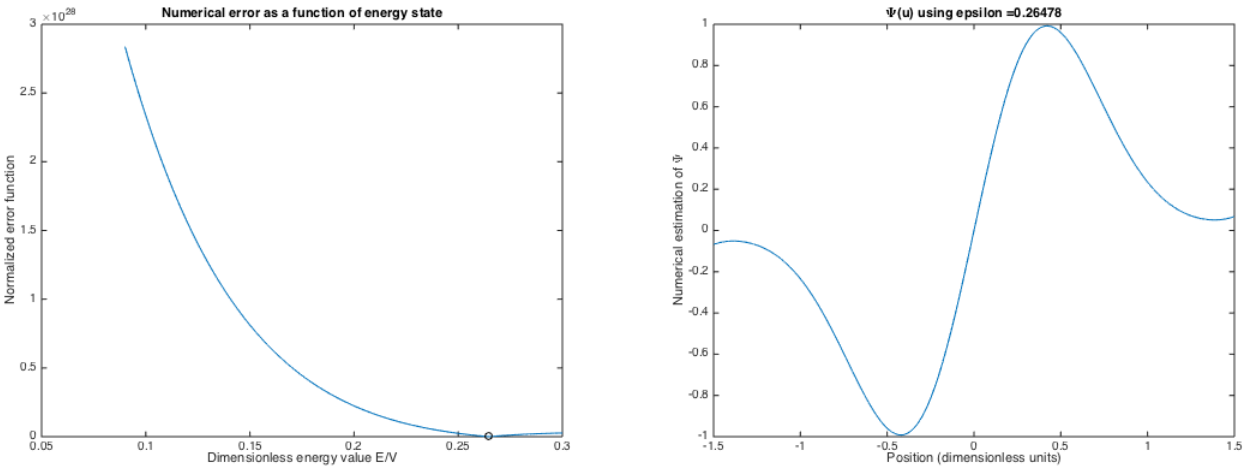
$$\psi(u_{i+1}) = \psi(u_i) - \Delta u\phi(u_i) \tag{7}$$

In light of these results, we decided not to change the coupled numerical differential equation for ψ because it works the same for any potential $V(u)$. Our value for β we chose according to the order of magnitude of its constants. We solved for the ground, first excited, and second excited states of the harmonic oscillator in the same way we did for the finite well. Our results were $\epsilon_0 = 0.088387$, $\epsilon_1 = 0.264777$, and $\epsilon_2 = 0.44145$. The graphs and their associated cost functions are are follows:



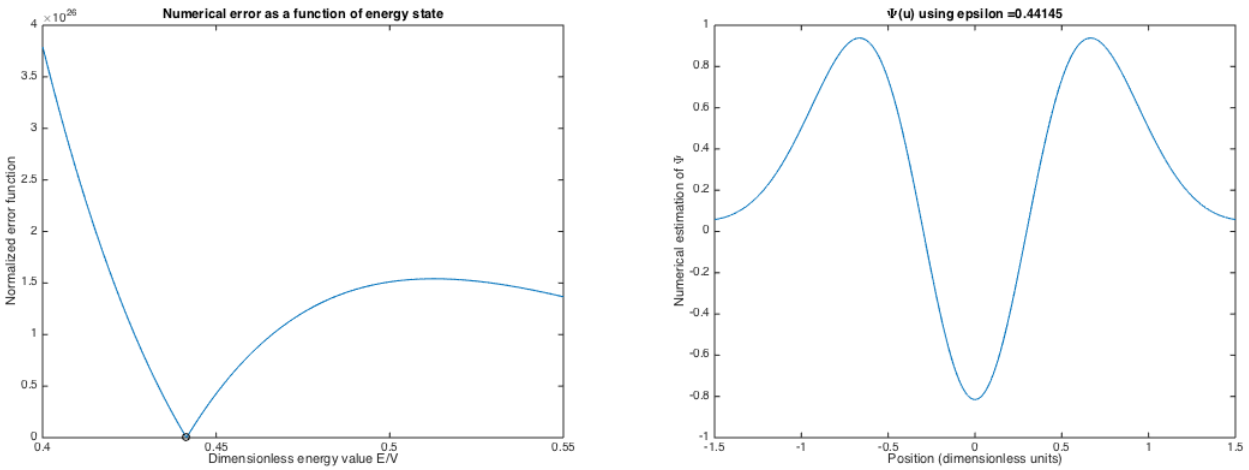
(a) Cost function **(b)** Optimized ψ

Figure 12: Solving for ground state of ϵ



(a) Cost function **(b)** Optimized ψ

Figure 13: Solving for first excited state of ϵ



(a) Cost function **(b)** Optimized ψ

Figure 14: Solving for second excited state of ϵ

NOTE The following discussion is of dubious factual integrity. We would confirm this math if we had more time, but time constraints leave us uncertain of its theoretical validity. That said, we consider it an important enough result to include in this writeup.

Now we need to check to see if our results correspond to a real-world situation. Consider the HCl molecule, where there is oscillation between the H and Cl atoms. Using $\omega = 2\pi 8.66 \times 10^{13}$ and $\hbar = 1.05457 \times 10^{-34}$ we can solve for the ground energy state as follows:

$$E_1 = \frac{1}{2}\hbar\omega = 2.87 \times 10^{-20} J \tag{8}$$

Now we can use the value of the second excited state ϵ we solved for numerically by working backwards from dimensionless parameters and using the constants $\beta = 64$, $\hbar = 1.05457 \times 10^{-34}$, $m = 1.67 \times 10^{-27}$, and $a = .13 \times 10^{-}9$. The values for m and a are the mass of a proton and the bond length of HCl respectively.

$$E_1 = \frac{\beta\hbar^2}{2\epsilon m a^2} = 2.87 \times 10^{-20} J \tag{9}$$

We get the same energy value when we use the ϵ value we solved for numerically, meaning that our answer is correct. One could do the same process for the first and second excited states of the harmonic oscillator and find similar results.

4 Discussion

4.1 Alternate numerical solution

An alternate numerical solution to the finite quantum well would be to split the function ψ into three parts: 1) an exponential function to the left of the well, 2) a trigonometric function inside the well and 3) an exponential function to the right of the well. The two exponential functions would be of the form

$$Ce^{\alpha x} \tag{10}$$

The trigonometric would be of the form (for the ground state):

$$Dcos(kx) \tag{11}$$

Next we could impose the boundary condition that the derivative of the left exponent be equal to the derivative of the trigonometric function at the left wall of the well. Similarly, the derivative of the right exponent should be equal to the derivative of the trigonometric function at the right wall of the well. With these conditions and the original two equations we could relate k to α and then use numerical methods to optimize three equations. The numerical optimization would be feasible because we would be optimizing only one constant, having written the other constants in terms of that constant.

5 Appendix

5.1 Code for shooting method

```
function [ V ] = V( x )
%V A function which defines the potential energy of a 1D system
%   V(x) where x is position and V is dimensionless energy units

% V = 0; % V is zero in the well
% if abs(x) > 1/2 % if x is outside the well, it goes to 1
%     V = 1;
% end

V = 1/2*x^2;


end


```

Figure 15: Cost function (set up for harmonic oscillator)

```
1   function [ psi, u ] = fstep( psi_init, phi_init, epsilon, du, gridsize, flag,
2   %FSTEP Solves numerically for a set of psi values
3   %   Numerical derivative formula taken from the pdf handout implemented on
4   %   on lines 29 and 30
5
6   % define default parameters for depugging
7   if nargin == 0,
8       psi_init = 0;
9       phi_init = 1;
10      epsilon = 1.5;
11      gridsize = 2000;
12      du = 1.5/gridsize;
13      flag = 1;
14      fnorm = normalizeEig(epsilon, psi_init, phi_init, du, gridsize);
15  elseif nargin == 5,
16      flag = 0;
17      fnorm = 1;
18  end
19
20  % initialize constants
21  u_init = 0;
22  beta = 64;
23
24  % create a vector of possible position values (where u is the dimensionless
25  % variable which represents x/x0
26  u = [u_init - du * (gridsize - 1) : ...
27      du : u_init + du * (gridsize - 1)];
28
29  %set up a vector which we will fill with phi values (using phi0 as our
30  %middle value
```

Figure 16: Numerical function for ψ part 1

```
phi = zeros(1, 2*gridsize - 1);
phi(gridsize) = phi_init;

%set up a vector which we will fill with psi values (using psi0 as our
%middle value)
psi = zeros(1, 2*gridsize - 1);
psi(gridsize) = psi_init;

%apply the numerical derivative formula given in the handout and solve for
%all the values of psi and phi to the *right* of our initial values

for i = gridsize:2*gridsize - 2,
    phi(i + 1) = phi(i) - du * beta * (epsilon - V(u(i))) * psi(i);
    psi(i + 1) = psi(i) + du * phi(i);
%     phi(i + 1) = phi(i) - du * (beta / alpha - V(u(i))) * psi(i);
%     psi(i + 1) = psi(i) + du * phi(i);
end

%apply the numerical derivative formula given in the handout and solve for
%all the values of psi and phi to the *left* of our initial values

for i = gridsize:-1:2,
    phi(i - 1) = phi(i) + du * beta * (epsilon - V(u(i))) * psi(i);
    psi(i - 1) = psi(i) - du * phi(i);
end


psi = psi*fnorm;

%if the flag (t/f) tells us to plot our result, we plot our result
if flag == 1,
    plot(u, psi);
    title(strcat('\Psi(u) using epsilon = ', num2str(epsilon)));
    xlabel('Position (dimensionless units)');
    ylabel('Numerical estimation of \Psi');
end
```

Figure 17: Numerical function for ψ part 2

```
phi = zeros(1, 2*gridsize - 1);
phi(gridsize) = phi_init;

%set up a vector which we will fill with psi values (using psi0 as our
%middle value)
psi = zeros(1, 2*gridsize - 1);
psi(gridsize) = psi_init;

%apply the numerical derivative formula given in the handout and solve for
%all the values of psi and phi to the *right* of our initial values

for i = gridsize:2*gridsize - 2,
    phi(i + 1) = phi(i) - du * beta * (epsilon - V(u(i))) * psi(i);
    psi(i + 1) = psi(i) + du * phi(i);
%     phi(i + 1) = phi(i) - du * (beta / alpha - V(u(i))) * psi(i);
%     psi(i + 1) = psi(i) + du * phi(i);
end

%apply the numerical derivative formula given in the handout and solve for
%all the values of psi and phi to the *left* of our initial values

for i = gridsize:-1:2,
    phi(i - 1) = phi(i) + du * beta * (epsilon - V(u(i))) * psi(i);
    psi(i - 1) = psi(i) - du * phi(i);
end


psi = psi*fnorm;

%if the flag (t/f) tells us to plot our result, we plot our result
if flag == 1,
    plot(u, psi);
    title(strcat('\Psi(u) using epsilon = ', num2str(epsilon)));
    xlabel('Position (dimensionless units)');
    ylabel('Numerical estimation of \Psi');
end
```

Figure 18: Numerical function for ψ part 2

5.2 Code for optimization of shooting method

```
function [ cost ] = epsilonCost( e, gridsize, du, psi_init, phi_init, opt )
%EPSILONCOST Returns a number indicating how good our choice of epsilon
%was. Lower is better
%   This function returns the maximum value over the absolute value of
%   a range of numerically-derived \Psi values

% %get a set of psi values for a given epsilon and gridsize
psi = fstep(psi_init, phi_init, e, du, gridsize);

%get "cost" value which depends on how far the psi function deviates from
%zero outside of the well 1) at its maximum and 2) on average. These
%measurements are both useful, but in different circumstances
if (opt == 0)
    cost = max(abs(psi));
else
    cost= numIntegral(du, abs(psi).^2);
end


%%%%%%%%%%%%%%%%%%%%%%%% HELPER FUNCTIONS:

%this function performs a numerical integral for some set of y values
%separated by a constant dx
    function [area] = numIntegral(dx, y)
        area = 0;
        for index = 1:length(y)
            area = area + y(index) * dx;
        end
    end

end


```

Figure 19: Cost function for finding ψ of best fit

```
1   function [ ] = minimizeEpsilon()
2   %MINIMIZEEPSILON Solve for the lowest value of epsilon within a range of
3   %possible epsilon values
4   %   Use a built-in Matlab function to solve for a local minimum in the cost
5   %   function for epsilon
6
7   %initialize constants and settings
8   debugGridsize = 200000; %use for less computationally expensive routines
9   debugGraphRange = 5; %use for less computationally expensive routines
10  gridsize = 3000; %use for more computationally expensive routines
11  costFunctionRange = 5; %use for more computationally expensive routines
12  energyGraphRange = 1.5; %use for routines which display final psi
13
14  du = costFunctionRange/debugGridsize;
15  psi_init = 1;
16  phi_init = 0;
17  dx = 0.0001; %this is used later for debugging/fine tuning with graphs
18
19  e1 = .3;
20  e2 = .5;
21  displayMode = 1;
22  %ground: 0.098028
23  %first:  0.382721
24  %second: 0.807892
25  %third:  1.451748
26
27  %create function handle for cost function
28  fun = @(x)epsilonCost(x, debugGridsize, du, psi_init, phi_init, 1);
29
30  %use efficient built-in Matlab solver to find a minimum value
```

Figure 20: Code for minimizing cost function and controlling all the other functions

```
% options = optimoptions('fminbnd','TolX','8');
minx = fminbnd(fun,e1,e2);

disp('Optimal epsilon value * 100:');
minx*100 %we do this because it shows us more decimal places

if (displayMode), %if we are displaying our findings

    % decrease the gridsize for computational efficiency
    du = costFunctionRange/gridsize;
    fun = @(x)epsilonCost(x, gridsize, du, psi_init, phi_init, 0);

    %now we need to present our findings with a graph, so we get data
    epsilon = zeros(1,gridsize);
    eCost = zeros(1,gridsize);
    for i = 1:gridsize,
        epsilon(i) = e1 + (e2-e1)*i/gridsize;
        eCost(i) = fun(epsilon(i));
    end

    %plot cost function with the minumim we found
    plot(epsilon, eCost);
    hold on;
    plot(minx, fun(minx), 'ko');
    title('Numerical error as a function of energy state');
    xlabel('Dimensionless energy value E/V');
    ylabel('Normalized error function');

    %plot the actual energy level we've solved for:
    figure(2)
```

Figure 21: Code for minimizing cost function and controlling all the other functions

```
61          %change the range of the function so that our graph is more pretty
62 -        du = energyGraphRange/gridsize;
63          %get an array pf psi values
64 -        psi = fstep( psi_init, phi_init, minx, du, gridsize);
65          %normalize by a constant factor, fnorm
66 -        fnorm = normalizeEig(minx, psi_init, phi_init, du, gridsize);
67 -        psi = psi*fnorm;
68
69          % check normalization
70 -        disp('Making sure normalization was successful:');
71 -        numIntegral(du, abs(psi).^2)
72
73          %a plot comes from this function because we pass in a flag=1 parameter
74          %which tells the subroutine to plot its result
75 -        fstep( psi_init, phi_init, minx, du, gridsize, 1, fnorm);
76 -    else
77          %this snippet of code lets me make sure that there are not better
78          %eigenvalues in the neighborhood of the one I've solved. If the
79          %black lines are on either side of the red line diverge to infinity
80          %more rapidly than the red line, then I know that I have converged to
81          %a local minimum. Think of it as a debugging tool
82 -        du = debugGraphRange/debugGridsize;
83 -        figure(2);
84 -        psi = fstep( psi_init, phi_init, minx, du, debugGridsize)*...
85              normalizeEig(minx, psi_init, phi_init, du, gridsize);
86 -        plot(psi,'k-');
87
88 -        hold on;
89
90 -        for i=1:2,
```

Figure 22: Code for minimizing cost function and controlling all the other functions

```
91 -            pause;
92 -            psi = fstep( psi_init, phi_init, minx + i*dx, du, debugGridsize)*...
93              normalizeEig(minx, psi_init, phi_init, du, gridsize);
94 -            plot(psi,'r-');
95 -        end
96 -        for i=1:2,
97 -            pause;
98 -            psi = fstep( psi_init, phi_init, minx - i*dx, du, debugGridsize)*...
99                  normalizeEig(minx, psi_init, phi_init, du, gridsize);
100 -           plot(psi,'r-');
101 -       end
102 -   end
103
104     %%%%%%%%%%%%%%%%%%%%%%%% HELPER FUNCTIONS:
105
106     %this function performs a numerical integral for some set of y values
107     %separated by a constant dx
108         function [area] = numIntegral(dx, y)
109 -           area = 0;
110 -           for index = 1:length(y)
111 -               area = area + y(index) * dx;
112 -           end
113 -       end
114 -   end
115
```

Figure 23: Code for minimizing cost function and controlling all the other functions

5.3 Code for normalization

```
function [ A ] = normalizeEig( eigV, psi_init, phi_init, du, gridsize )
%NORMALIZEEIG This function estimates the normalization constant for a
%numericall-computed psi function using numerical integration

if nargin == 0,
    psi_init = 1;
    phi_init = 0;
    eigV = 0.1;
    gridsize = 30000;
    du = 1/30000;
end

%get a set of psi values for a given epsilon and gridsize
psi = fstep(psi_init, phi_init, eigV, du, gridsize);

%estimate the area under the probability function curve using numerical
%integration
cost = numIntegral(du, abs(psi).^2);

%To make this area 1, we set the constant to the inverse of that area
A = 1/sqrt(cost);

%%%%%%%%%%%%%%%%%%%%%%%% HELPER FUNCTIONS:

%this function performs a numerical integral for some set of y values
%separated by a constant dx
    function [area] = numIntegral(dx, y)
        area = 0;
        for index = 1:length(y)
            area = area + y(index) * dx;
        end
    end

end
```

Figure 24: Code for finding a normalization constant