P 24 - Computational Project

Numerical solution of the time-independent Schrödinger equation for a *finite* square-well potential

1 Overview

In any kind of modern science, the ability to do a numerical solution of a problem is tremendously important. The numerical/computational project outlined here has several goals. In terms of physics, it wants to inspire you to explore the numerical solutions of the Schrödinger equation for a given potential. In practical terms, it wants to challenge you to use tools to be able to plan, write, and execute a computer program, and plot results and functions.

In this project, we will obtain a numerical solution of the Schrödinger equation in 1D for one particle, after defining the force acting on it by supplying a potential $V(x)$ in which the particle is supposed to move. We will not solve for the motion itself (time-dependent Schrödinger equation), but rather we will solve for energy eigenfunctions (we'll get solutions of the TISE).

We could investigate all kinds of potentials for this project, for example the harmonic oscillator ($V(x) = kx^2/2$), or a constant central force (Figure 5-21 in Eisberg & Resnick page 172), or variants of the finite square well (Figures 5-24, 5-25), or others (Figure 6-22). We will investigate, however, the finite square well potential here (Figures 6-23, 6-25 on page 212), and hence will try a hands-on, modern version of the activity of Appendix G of Eisberg & Resnick. Our potential is:

$$V(x) = \begin{cases} 0 & |x| \leq a/2 \\ V_0 & |x| > a/2 \end{cases} \tag{1}$$

with a positive potential energy constant V_0, and a positive spatial extent a of the well.

The simple method we are pursuing in this project is called "shooting method", illustrated by ER Figure 5-14: A starting point x_0 is agreed upon, and all three parameters: $\psi(x_0)$, $\frac{d\psi}{dx}(x_0)$, and E are picked (as a trial) and a preliminary solution calculated starting from x_0 ("numerical integration"). In other words, $\psi(x)$ for all $x > x_0$ is calculated. This solution satisfies the Schrödinger equation and the boundary conditions at x_0 by design. However, a guess like that will likely not do us the favor of satisfying the other boundary conditions, namely those at infinity: $\psi(x) = 0$ ($x \to \infty$) and $\frac{d\psi}{dx}(x) = 0$ ($x \to \infty$). Instead, it will most likely blow up to $+\infty$ or $-\infty$. To get closer to a real solution, one now varies the three parameters $\psi(x_0)$, $\frac{d\psi}{dx}(x_0)$, and E until a solution is obtained that has the right properties also for $x \to \infty$ ("the correct asymptotic behavior"). ER Figure 5-14 has three of these attempts, with the one labeled 3 being closest to success. An overall success will then have simultaneously determined the eigenvalue E and the energy eigenfunction $\psi(x)$ (or more appropriately put, determined one possible eigenvalue E and its associated energy eigenfunction), and it allows (see Figure 5-14) for calculating backward (solving the Schrödinger equation for $x < x_0$, using the freshly identified value of E). The numerical integration of the Schrödinger equation for one set of parameters is one "shot" of the shooting method, and the shooting method itself consists of doing this for a variety of parameters.

2 Logistics

As always, you are encouraged to work in a group and teach each other the basics of programming and plotting, and the understanding of the concepts. You may exchange or research concepts like example iteration loops, functions, routines, or sample plot commands. However, you are not to share or copy the actual commands, programs, or output related to this project. In other words, the work you submit must be genuinely your own.

You have complete freedom to choose the software tool(s) you prefer to carry out the calculations and plots. Downloading and installing Mathematica is recommended; most P24 students have acquainted themselves with Wolfram alpha already and will find Mathematica intuitive. It can be used for programming and for plotting as well. But if you have a working knowledge of another package such as Matlab, IDL, PV-Wave, Maple, or similar, you may use those instead.

The entire project needs to be written up like a lab report that should flow like a continuous document, not just terse comments or answers here and there between plots and commands. Methods and results should be presented, discussed, and conclusions formulated. Within the flow of the report, also all the questions posed in this project manual should be treated. Establish your report in electronic pdf form for Blackboard submission (under "Assignments"), embedding plots and commands as objects or figures in your text. Also, for ease of looking over your lab, please submit your Mathematica notebooks (.nb files) or Matlab .m files, or similar, that contain the commands and programs you developed and used. If you decide to hand in the report as a Mathematica, Matlab, or other file with text in between the commands and plots, please make sure that nonetheless you also do a "print to file" to generate a pdf and submit the pdf as well.

The final version of your entire report and the accompanying files are **due by the end of Monday, Jan. 28, 2013.**

3 Physics

We are looking to numerically solve the time-independent Schrödinger equation (TISE)

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V(x)\,\psi(x) = E\,\psi(x)$$

obtaining solutions $\psi(x)$ which are called energy eigenfunctions, with the corresponding energy eigenvalue E, and using the above potential $V(x)$. [1] ER Figures 6-25 and 6-26 show typical bound energy levels and corresponding eigenfunctions, respectively. The entire problem of the finite square well is discussed in ER section 6-7, and in appendices G and H.

Rearranging the Schrödinger equation yields

$$\frac{d^2\psi}{dx^2} = -\frac{2m}{\hbar^2}[E - V(x)]\,\psi(x) \tag{2}$$

The second derivative of a function determines its curvature. The sign of the second derivative determines the sense of the curvature; when the second derivative is positive, the function curves in an anticlockwise sense. When the second derivative is negative, the function curves

[1] The equation is a popular example of an eigenvalue problem, with $-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)$ the operator, E the eigenvalue, and ψ the corresponding eigenfunction.

clockwise. When the second derivative changes sign, the sense of curvature changes as well. A point of sign-change in the second derivative is called inflection point. Inspecting the above equation, we see that ψ can have inflection points when the sign of ψ changes, or when the sign of $E-V(x)$ changes. (Find more information on this topic in ER section 5-7.)

In our problem, the sign of $E-V(x)$ changes only at $x=\pm\frac{a}{2}$, and additionally only when E is in the energy range that allows bound states, $0<E<V_0$. This can be seen in action in ER Figure 6-26 at both $x=-\frac{a}{2}$ and $x=+\frac{a}{2}$. The action of the zeroes of ψ also being inflection points can be seen in the same figure for ψ_1 and ψ_2. (The sequence of signs of $\frac{d^2\psi}{dx^2}$ for ψ_3 is, from left to right, "- + - + -", with a total of 4 inflection points.)

Recall some general findings of quantum mechanics:

- The Schrödinger equation is second order in space and hence requires two boundary conditions per boundary (value of wave function, and value of first derivative of wave function).
- For $E<V_0$, we have bound states and a discrete energy spectrum. For $E\geq V_0$, all such energies are allowed (continuous spectrum), and are not bound states, but rather "scattering" states.
- For $E<V_0$, we have bound states, but because of the finiteness of V_0, the region beyond $|x|=\frac{a}{2}$ (the "classically forbidden" region) is penetrated by an exponentially decaying, non-zero wave function, in contrast to the infinite square well.
- Because the potential (1) is symmetric (under $x\to -x$), the energy eigenfunctions have a definite parity, see Figure 6-26. We will make use of this in this project! We will calculate starting from $x_0=0$ for $x>0$, as further explained below. Through parity, we automatically know exactly what's going on for $x<0$ without further calculations.
- Solutions of the TISE may all contain an overall factor, of which only the magnitude is determined by the normalization condition, but not its phase. It is all right, however, to choose the phase that makes the energy eigenfunctions real functions.

4 Mathematical preparation

Two tasks need to be carried out to prepare our equation (2) for numerical treatment. Since numerical calculations can deal only with pure numbers, without units, the first task is to switch to dimensionless quantities. So we want to re-express our equation using dimensionless variables: Dimensionless position $u=\frac{x}{a}$, and dimensionless energy $\epsilon=\frac{E}{V_0}$. The derivative becomes $\frac{d}{dx}=\frac{1}{a}\frac{d}{du}$, and equation (2) becomes:

$$\frac{d^2\psi}{du^2}=-\frac{2ma^2V_0}{\hbar^2}\left[\epsilon-\frac{V(au)}{V_0}\right]\psi$$

The dimensionless form of the potential

$$\tilde{V}(u)=\begin{cases}0 & |u|\leq 1/2\\ 1 & |u|>1/2\end{cases}\tag{3}$$

together with defining a dimensionless factor β,

$$\beta = \frac{2ma^2V_0}{\hbar^2}$$

(related to the strength/depth of the potential) lets us write the Schrödinger equation in completely dimensionless form then:

$$\frac{d^2\psi(u)}{du^2} = -\beta \left[\epsilon - \tilde{V}(u)\right] \psi(u) \tag{4}$$

A benefit of this procedure is that actual values for positions, energies, β, and potentials are close to 1 within a few orders of magnitude. Not doing this conversion to dimensionless numbers would lead, for example, to typical locations in the Angstrom range for realistic quantum wells, and $\hbar^2$ would be explicitly used, with its value of 10^{-68}. This is actually a problem for most programming language implementations, which cannot distinguish between 10^{-68} and zero if no special precautions are taken (keyword: single precision vs. double precision). This entire problem is elegantly avoided by going to the dimensionless equation and variables.

The second task is to convert the differential equation of order 2 to a system of two differential equations of order 1. This task is accomplished by creating an additional function $\phi(u) = \frac{d\psi}{du}$. This is then the system of 2 simultaneous differential equations corresponding to equation (4):

$$\frac{d\phi}{du} = -\beta \left[\epsilon - \tilde{V}(u)\right] \psi(u) \tag{5}$$

$$\frac{d\psi}{du} = \phi(u) \tag{6}$$

The reason for reducing things to first order differential equations is that numerical considerations show that for lower order equations, it is easier to achieve high accuracy than for higher order equations. So this task is done for the sake of accuracy.

5 Numerical strategies

5.1 Mathematical

5.1.1 Discretization

On a computer, an arbitrary function like ψ cannot be displayed or stored with arbitrarily fine resolution. For numerical purposes, one has to go from real-valued functions ψ, ϕ, defined over a continuous spatial variable u, to a "discrete" representation that knows the values of ψ and ϕ only at certain points u_i. The set of all available u_i is called a numerical grid. Let us use here a regular grid, where the spacing (the distance from one grid point to the next) is constant, namely, Δu. If, for example, $\Delta u = 0.0025$, a grid of 2,000 points ($i = 1, \ldots, 2000$) would span a spatial range of 5 in u (recall we are talking about dimensionless things here), and as the potential fits into a range of 1 ($-0.5 < u < 0.5$), the wave function in the well would definitely fit into those 2,000 points, occupying only 400 of them, and there would be ample room for the tails beyond $|u| = 0.5$ (which is $|x| = \frac{a}{2}$) on either side, which, as we said, is the classically forbidden region that still carries a non-zero wave function. The grid needs to be at least fine enough (i.e., Δu needs to be at least so small) that the functions ψ and ϕ don't change much between neighboring grid points - then the continuous functions are well represented by the discrete representation. Going to finer resolution than that has two effects: It makes the calculations even more accurate, and it requires more calculation time and memory (hardware).

5.1.2 Numerical derivative

Another mathematical question to be addressed is how to deal with a spatial derivative in numerics. For this, we have to go way back to intro calculus, where the derivative was defined as

$$\frac{d\psi(u)}{du} = \lim_{\delta u \to 0} \frac{\psi(u+\delta u) - \psi(u)}{\delta u}$$

In discrete numerics, where we cannot let δu fall below Δu, this definition is replaced by its obvious analogue,

$$\frac{d\psi(u)}{du} \approx \frac{\psi(u+\Delta u) - \psi(u)}{\Delta u} = \frac{\psi(u_{i+1}) - \psi(u_i)}{\Delta u} \tag{7}$$

Below, this equation will be used in a bootstrap-type of way: If you know both $\psi(u_i)$ and $\frac{d\psi(u_i)}{du}$, then you can calculate ψ at the next grid point:

$$\psi(u_{i+1}) = \psi(u_i) + \Delta u \frac{d\psi(u_i)}{du}$$

Throwing our Schrödinger equation into the mix, as well as the function ϕ, our system of 2 differential equation hence yields the following prescription:

$$\phi(u_{i+1}) = \phi(u_i) - \Delta u \; \beta \left[\epsilon - \tilde{V}(u_i)\right] \psi(u_i) \tag{8}$$

$$\psi(u_{i+1}) = \psi(u_i) + \Delta u \; \phi(u_i) \tag{9}$$

which indeed is a prescription for calculating the next grid point $i+1$ if you happen to have the values of ψ, ϕ, and $\tilde{V}$ at the current grid point i (all the right-hand-sides). [2]

This procedure of starting with known values at one point and then calculating all values for x locations to the right of the point is called "numerical integration".

5.2 Technical

To make the calculation equations (8) and (9) happen, you need to write a program. We are all hoping that it will look a lot more legible and understandable than Tables G-1 and G-2 of Eisberg and Resnick. So we will stay away from the BASIC programming language, and we also seek to not reproduce the structure of their program. And we are more ambitious (the ER program does not do that): We want to store all the thousands of values $\psi(u_i)$ of our single calculation of the solution, and when done with the last i, we want to plot the stored values, much like ER Figure G-2.

[2] Here is an important note: What is written here is quite crude, qualifying for the label "quick and dirty". It turns out that the best way to develop an accurate framework for numerical derivatives is to apply (your best friend) the Taylor approximation to the problem. So beware that in your research career, you will encounter much refined variants of equation (7), in particular the fact that given the right-hand side of it, the left-hand side strictly speaking is the derivative at location $u_{i+1/2}$, the midpoint between u_i and u_{i+1}. Do not worry about this in this project, though. In equation (8) and (9), we have behaved as if the result defines the derivative at u_i. We just will set Δu extra small to recover the accuracy that is compromised by this negligence.

It is assumed that everyone knows what a variable is in the context of programming. For storing thousands of values $\psi(u_i) \equiv \psi_i$, we need thousands of variables. The trick to do that in one step is to declare an array instead of a simple variable. Then we will have a variable, call it psi for example, and to get to a specific element ψ_i, we have to use the array variable with an integer index, let's call it i. This concept is realized differently in different languages, so here is an example in Mathematica:

```
example = Table[0, {10}];
Do[  example[[i]] = Sin[i*0.5], {i, 1, 10}  ];
```

The first line sets up an array named example, and sets it up so that it is an array of 10 variables. In each of the ten variables, it put a 0 (the first argument of the Table function). The ten variables are: example[[1]], example[[2]], example[[3]], ..., example[[10]]. The second line is a computational loop ("do loop"), which carries out the expression inside (the example[[i]] = Sin[i*0.5] part before the comma, i.e., the first argument of Do) several times, first with i=1, next with i=2, and so on, until last with i=10. For example, after the Do command is finished, the variable example[[3]] will no longer have the value 0 that it had before, but the value sin(1.5), where Mathematica interprets the 1.5 as an angle in radians (hence, pretty close to 86 degrees). Note that the variable i here is a prime example for a *local* variable; it does not exist (is not defined) outside the command, in this case the Do command. Note further that the spaces in the example have been only added for clarification, they are not required but make things more readable. [3]

The Mathematica command `Print[example]` gives all ten values of the array, as a list. In the above case, the resulting output is: { 0.479426, 0.841471, 0.997495, 0.909297, 0.598472, 0.14112, -0.350783, -0.756802, -0.97753, -0.958924 }. In principle, this is a discrete representation of three quarters of a sine wave with a grid of only 10 points.

Next question: How to plot such a thing? The answer is, with ListPlot. Just try this example:

```
example = Table[0, {1000}];
Do[  example[[i]] = Sin[i*0.01], {i, 1, 1000}  ];
ListPlot[example]
```

Here, we took a larger array (1000) and a much finer resolution (0.01) to make the future plot quite smooth. Note that effectively, we are probing the sine function from 0.01 to 10 (a bit more than one and a half cycles), but ListPlot does not know this, and hence the x axis is labeled with the integers, ranging from 1 to 1000. In your project hopefully you can refine this approach so that the correct labels (values) appear on the x axis - the help function helps ("Documentation Center").

The final piece you have to know about programming is the existence of functions, something akin to outsourcing a task to a different country. Imagine you know about 5 lines of code that converts a number x into the value of the error function erf(x). If you now imagine that you need the erf function[4] at several points within your program, your choice is: Either

[3]Mathematica Tidbits: (1) The semicolon at the end of a line in Mathematica suppresses output. (2) Be aware that an element [[0]] of an array is not defined in Mathematica - it is not a valid variable, and use of it will lead to errors. Of course, if you define an array with Table[0,n] then also any element [[n+1]], [[n+2]], ... is undefined, as are any elements with negative index such as [[-1]]. (3) The executions inside a do loop can be several commands, not just one, separated by enter and/or by semicolon.

[4]If you are not familiar with it, you can look it up: this function exists and is very relevant to physics and statistics.

write those 5 lines every time you need it in the program, or: write 5-line function erf(x) and call it when you need it with a simple "erf(x)"

Here is a useful example of a function "pot" that accepts as argument a value of u, and returns the value $\tilde{V}(u)$:

```
pot[u_]:= If[ Abs[u]<0.5, 0, 1 ];
```

Note that Mathematica requires that in such a definition, the argument variables in the argument list have an extra underscore at the end of their name (note: only in the argument list, not in the body of the function). You can use this function "pot" for example through executing "pot[0.3]" or "pot[3.14]" or "pot[3a]". Note in passing the form of the ever-important "if" function in Mathematica; it has three arguments: If[<test that yields 'true' or 'false'>, <action to be taken in case of 'true'>, <action to be taken in case of 'otherwise'>]. Here is an equivalent example in the language FORTRAN which is old (older than C) but still popular in science:

```
if ( abs(u) .lt. 0.5 ) then
   pot = 0.0
else
   pot = 1.0
endif
```

Java, C, Matlab, and IDL have similar constructs, using different actual commands.

Here is a silly example for a longer function that warrants the use of the command 'Module' just because it is longer than a 'one-liner': A function that accepts as argument an integer number, say n, and returns an array of length n that has sine values of one period equally spaced:

```
sinarr[n_]:=Module[ {examp},
  examp = Table[0,{n}];
  Do[  examp[[i]] = Sin[i*2*Pi/n], {i, 1, n}  ];
  examp
]
```

You can use it[5]. Try the following: sinarr[10], N[sinarr[10]], N[sinarr[50]], or ListPlot[sinarr[50]], or ListPlot[sinarr[1000]]. Notes: the list of variables preceding the program inside the Module (i.e., the first argument of the Module command) defines which variables are to be considered local to the Module, meaning the variable examp does not exist outside sinarr, even after you have used that function. Further note: we had to end the code with a line saying 'examp' so that the array would be output, which in this case means that the array examp is considered the legitimate output of the function sinarr. Otherwise, sinarr would be confused about what to return, because as a rule it seeks to return the output of the

[5]Mathematica Tidbit: Commands can go on over several lines in Mathematica, thus vastly increasing readability. Note the indentation of the example, and the closing parenthesis, as an aide for humans to understand the hierarchy, completely superfluous from the computer's point of view. When using several lines, created with the Enter key rather than Shift-Enter, just make sure that the entire thing really does reside in one single bracket as seen on the right hand side of your MMA notebook. (Recall how Shift-Enter is used to carry out the content of such a cell; it also needs to be pressed to make active a function definition like the one above.)

last command executed, which would be the Do loop, which unfortunately has no standard output. In many other cases, this extra line would not be necessary. CompSci purists would have written the following instead of the second-to-last line: `Return[examp]`

6 Numerical Project: Assignment

6.1 Development

For the particle-in-a-finite-well problem, write a program that implements the shooting method. Use the dimensionless β as a global variable, and set it rather arbitrarily to $\beta = 64$, as in ER Appendix G.

Within your program, write a function for the dimensionless potential $\tilde{V}$. Writing it as a separate function enables you to reuse the program for other potentials later, without too much effort.

Also within your program, write a "shooting" function that has four input arguments: the value of ψ at $x_0 = 0$, the value of $\frac{d\psi(u)}{du}$ at $x_0 = 0$, the value of ϵ, and that of Δu. The function should carry out the shooting method to solve the Schrödinger equation with the given parameters, using a grid defined by the given Δu. The function should return an array that represents the values of ψ on the grid. Document the lines of your codes with ample annotations/comments; in Mathematica, comments enclosed in a pair of (* text *) do not disturb the computation or syntax of the code. The code is part of what you write in your report, and it also will be part of the program file that you submit.

Then develop a variant of ListPlot to be able to plot one such result, in the spirit of ER Figure G-2. Note that in Mathematica, axes are not always plotted through zero unless a plot range is specified explicitly - do not be fooled by this issue.

6.2 Applications

1. Vary the parameter ϵ and continue plotting each shooting method result. Through trial and error, find the actual solution that goes to zero for large values of u; one energy eigenvalue is apparently encountered around $\epsilon = 0.1$. Because of parity, your trials can safely assume (and you can plug in as arguments when using your function) that $\psi = 1.0$ at $x_0 = 0$, $\frac{d\psi(u)}{du} = 0$ at $x_0 = 0$; please discuss in your report why that is. The obtained solution is the ground state, please discuss why. On your path to the eigenvalue, please investigate it down to the fourth digit, as in Figure G-2.

   Write these results up in your report, including plot(s) that show at least several trials and their parameters (see Figure G-2), and the final eigenstate/eigenvalue identified.

2. Carry out the work of problem 1, ER appendix G; in other words, find the first and second excited energy eigenstate and their corresponding energy eigenvalue, with your code. Comment on whether you can imagine another way of a numerical solution of the finite-well problem that does not involve the shooting method.

3. Carry out problem 2, ER appendix G. Does the result conform to your expectations in all respects? Discuss any discrepancies. Are discrepancies affected when a smaller grid step Δu is chosen?

4. Similar to problem 3, apply your code to the infinite-quantum-well problem. For this, the quantity V_0 needs re-interpretation: It is best to set it to an arbitrary energy scale, for example, V_0=1 eV. With such a definition, the dimensionless Schrödinger equation is still valid as written above. The potential function $\tilde{V}$ should be changed so that the value outside is not 1, but a much larger number, to mimick the infinite height. The exact value of this number does not influence the result.

   The trial-and-error of the shooting method now aims at solutions that arrive at $\psi(x = a/2) = 0$ exactly. Please get ground state and first excited state, and compare: (1) the numerically determined eigenvalues to those theoretically predicted, and (2) the obtained eigenfunctions to the analytical ones on the same plot. Discuss matches or discrepancies.

5. Lastly, do one or the other of the following two problems:

   (a) Rewrite your code for the simple quantum harmonic oscillator potential (write out a dimensionless form of it, using Appendix G problem 5 or also x_0, ϵ encountered in problem sets 1 and 2 of P24), adjust the Schrödinger equation in your code to reflect the dimensionless quantum harmonic oscillator case, and do the trial and error shooting method to obtain the first two eigenvalues and eigenstates. Compare to the known analytical results.

   (b) Rewrite your finite-quantum-well code for the "constant central force" potential of ER Figure 5-21, page 176-177. Write out the potential, then convert it and the Schrödinger equation into a suitable dimensionless form, then code up this result. As an application, do the trial and error shooting method to obtain the first two eigenvalues and eigenstates, and plot and discuss them.