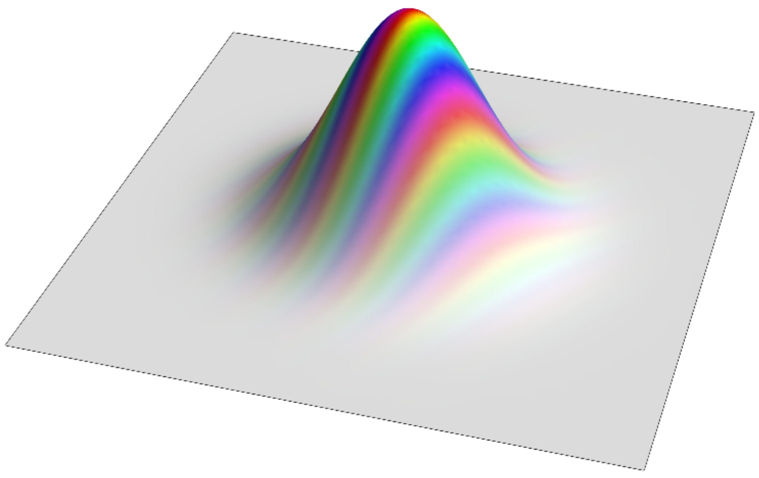
Quantum Mechanics with Mathematica

A tutorial for instructors

Daniel V. Schroeder
Weber State University

July 2019



Introduction

For more than 20 years I have used Mathematica as a computational and visualization tool when I teach quantum mechanics—both at the upper-division level and in a sophomore-level "modern physics" course. My goal in this tutorial is to help other instructors do the same.

I find Mathematica useful (in fact, almost indispensable) in these courses for several reasons:

- It allows students to quickly and easily plot any function, directly from its formula, often with only a single line of code.
- With only a little more effort students can produce multidimensional plots, animated plots, and plots of complex-valued functions that use color hue to represent phase.
- Special functions such as Hermite polynomials and spherical harmonics are built into Mathematica.
- Mathematica provides easy-to-use routines for numerical integration, solving ODEs, and diagonalizing matrices.
- Coding a PDE-solving algorithm is no harder in Mathematica than in any other language (although execution speed can sometimes be an issue).

Because Mathematica makes so many computational and visualization tasks about as easy as they could possibly be, it frees students to think more about the physics and less about the coding.

Mathematica's main disadvantage is that it is a commercial and closed-source product, sold by Wolfram Research. The license fee can be a barrier to some students, and the secrecy of its internal algorithms can be a barrier to some researchers. Nevertheless, for most of the instructional uses I'll describe here, I don't know of a suitable alternative.[1]

Mathematica is a versatile tool, and there are many ways to use it. This tutorial incorporates my personal preferences, which include the following:

- We'll use Mathematica as a tool for graphics and numerical calculations, not (to any significant degree) for algebraic manipulations. This emphasis may come as a surprise, because many people think of Mathematica primarily as algebraic manipulation software. My personal opinion, based on many years of teaching experience, is that quantum mechanics students should still do nearly all of their algebra by hand.
- I'll expect you to start with a blank window and type all of your Mathematica code from scratch, so the code becomes your own. I expect the same of my students. Please resist the urge to copy-and-paste my code examples from an

[1]Neither Wolfram Research nor anybody else is paying me to say this. My only compensation for developing this tutorial is my regular salary from Weber State University. All opinions expressed in this tutorial are my own, not those of WSU and certainly not those of Wolfram Research.

electronic version of this document. Although Mathematica can be used to produce "canned" simulations with code that is not meant to be read, I prefer to empower students to use Mathematica as a tool, not a mere demonstration.

- In the code provided herein, I've limited all variable names to use only characters that appear on the keyboard. Mathematica provides ways to enter Greek letters and other special symbols, and can even be used as a mathematical typesetting tool. But typesetting is not our purpose here, and I want to provide code examples that are easy to type without hunting for GUI buttons or special key combinations.

- Mathematica makes it possible to attach unit names to numerical quantities, but in this tutorial I will use "natural" units. Typically this means setting $\hbar$, the particle mass, and some natural length or energy scale all equal to 1. Today's students find the use of natural units disorienting, because we have taught them in introductory physics to use SI units for everything. Nevertheless, by the time they are learning quantum mechanics we need to teach them to use natural units—especially when they are doing computational work. Exercises on converting between natural units and laboratory units are an important part of a quantum mechanics course, but are not part of this tutorial.

If you would like to see how I have incorporated Mathematica calculations and visualizations into the context of a quantum mechanics course for upper-division undergraduates, feel free to download my draft textbook, temporarily titled *Notes on Quantum Mechanics*, from `http://physics.weber.edu/schroeder/quantum/`. Most of the examples in this tutorial are taken from there, although I have tried to present them here in a way that's more efficient for someone who already knows quantum mechanics. Even if that textbook isn't appropriate for the style or level of your own course, I hope you will find some useful examples in this tutorial, and that your students will gain a better understanding of quantum mechanics through the use of Mathematica.

This tutorial is organized into two major sections: Visualization and Computation.

1 Visualization

1.1 Basic plots and basic syntax

There's no better first exercise with Mathematica than making a basic plot, of a real-valued function of a single variable. Here's an example, for a harmonic oscillator eigenfunction:

```
Plot[(4x^4 - 12x^2 + 3)Exp[-x^2/2], {x, -5, 5}]
```

Type this line into Mathematica now, then hit shift-enter to execute it. The plot should appear in a moment.

This single line of code illustrates several of Mathematica's most important features:

- Mathematica code is built out of *functions*, such as Plot and Exp. Function names are case-sensitive, and all of Mathematica's built-in function names begin with capital letters.
- Function arguments are enclosed in square brackets and separated by commas. Here the Plot function has two arguments (both compound structures), while the Exp function has just one.
- The first argument of Plot is a mathematical expression, built out of the Exp function and the operators +, -, /, and ^. Multiplication requires no operator (just as in standard math notation), although you can always use * for clarity. (In a product of two variables such as a and b you need to write a*b or a b, so Mathematica won't think you're referring to a single variable named ab.)
- In compound math expressions, exponentiation takes precedence over multiplication and division, which in turn take precedence over addition and subtraction. So -x^2/2 means $-(x^2)/2$, not $-x^{2/2}$. To override the default grouping you can use parentheses. The spaces that I've inserted around the + and - are merely for readability.
- The second argument of Plot is a *list*, delimited by curly braces, with the three elements of the list separated by commas. Here the list specifies the independent variable of the function to plot, followed by the starting and ending values of this variable. I'll say more about lists below.

Assigning names

For all but the simplest tasks, you'll want to break up your Mathematica code into multiple statements, to be executed in sequence. The following three lines produce the same output as the single line above, but provide more flexibility:

```
psi = (4x^4 - 12x^2 + 3)Exp[-x^2/2];
xMax = 5;
Plot[psi, {x, -xMax, xMax}]
```

Here I've assigned the names psi and xMax to the function I want to plot and the maximum x value, so I can more easily modify or reuse these quantities. The semicolons serve as statement separators, and also suppress any displayed output of the lines that precede them. Go ahead and test this code now, putting all three lines into a single Mathematica "cell" (as indicated by the bracket at the window's right margin). Again hit shift-enter to execute the code. Then try modifying xMax, and psi too if you wish, to see how the plot changes.

Because all of Mathematica's built-in names (such as Plot and Exp) begin with capital letters, it's a good habit to begin all your own names (such as psi and xMax) with lower-case letters. Then you can easily tell at a glance which names are Mathematica's and which are yours. You'll also avoid inadvertent name conflicts, without having to learn all of Mathematica's built-in names (about 5000 of them, as of this writing).

Plot options

If you set `xMax` too high in the previous example, you'll notice that Mathematica expands the vertical scale of the plot and clips off the peaks of the function's five "bumps." This is because Mathematica thinks you want to see more detail in the small function values that occur at larger values of $|x|$. To override this often-unwanted behavior, you can use the `PlotRange` option:

```
xMax = 12;
Plot[psi, {x, -xMax, xMax}, PlotRange->All]
```

Try this modification now, then try providing an explicit range for the plot, using a two-element list:

```
PlotRange->{-5, 5}
```

Here are a couple of other useful options for modifying the appearance of a `Plot`:

```
Plot[psi, {x, -xMax, xMax}, PlotStyle->{Blue, Thick},
    AspectRatio->0.4]
```

You can specify options in any order, always separated by commas. For a complete list of options to use with `Plot`, use the menus to find the Mathematica documentation ("Help → Wolfram Documentation" in my version), then use the search feature to find the `Plot` function and, on its documentation page, scroll down to the "Options" section.

Instead of the color `Blue`, you can (of course) specify other pre-defined colors, or define your own colors using either `Hue` or `RGBColor`:

```
Hue[0.75, 1, 1]
RGBColor[0.5, 0, 1]
```

Try each of these, varying the numerical values to see how the color changes. All of the arguments should be in the range from 0 to 1, although for the first argument of `Hue` only the difference from the next-lowest integer is used, so the hues cycle repeatedly as this argument increases or decreases. The second and third arguments of `Hue` are saturation and brightness, and if you omit them you'll get the default of 1 for each.

More functions to plot

Besides `Exp`, Mathematica knows every named math function you've likely to have heard of: `Sin`, `Cos`, `Tan`, `ArcSin`, `ArcCos`, `ArcTan`, `Sinh`, `Cosh`, `Tanh`, `Sqrt`, `Abs`, `Log`, and more. The trig functions take angles in radians, and the logarithm function defaults to base e, though you can specify any base you want as an additional argument (see the documentation for examples). Mathematica also knows the constants `Pi` and `E` (note the capitalization!). And Mathematica knows all sorts of special functions that come up in advanced mathematics, including Hermite polynomials, spherical harmonics, Bessel functions, and so on.

But I'm really getting ahead of myself here. There's no need to learn about all these functions now, as long as you know how to do simple stuff like plotting a basic sine function:

```
Plot[Sin[10x], {x, 0, Pi}]
```

Function definitions and variable replacements

When you assigned a mathematical expression to the name `psi` a little while ago, you were telling Mathematica to replace "`psi`" with that expression wherever this name subsequently appears. For more flexibility, though, you can instead define your own function. Try this example:

```
gaussian[x_] := Exp[-x^2]
```

Here there are two differences from simply saying `gaussian = Exp[-x^2]`. The first is the replacement of `=` with `:=`, which *delays* Mathematica's evaluation of the right-hand side until later, when the `gaussian` function is actually used. The second difference is appending `[x_]` to the name `gaussian`, indicating that you're actually defining a function that takes a single argument, and that on the right-hand side this argument has the formal name `x`. When you later use the function you can insert any expression you like for this argument, and Mathematica will insert your expression in place of `x`. Try each of these uses, one at a time:

```
gaussian[0.5]

gaussian[2]

gaussian[y]

gaussian[(a-b)/c]

Plot[gaussian[x-5] + gaussian[x+5], {x, -10, 10}]
```

Notice that Mathematica tries to provide exact answers for expressions that contain only integer numbers. If you want the *numerical* value of `gaussian[2]`, you can type either `gaussian[2.]` (with a decimal point) or `N[gaussian[2]]`. (The `N[]` function does its best to convert any expression into a number in decimal approximation.)

It can sometimes be hard to decide whether to define a function in this way, or to simply assign a name as in the definition of `psi` above. I usually stick with a simple name assignment for an expression that I won't be needing to modify in the ways illustrated here (e.g., replacing `x` with `y` or something more complicated). And if you *do* want to make such a replacement in a previously named expression, you can do it like this:

```
psi /. x -> 3z    (* replace x with 3z wherever it appears *)
```

I usually pronounce the `/.` operator as "at the point." Here I've also shown how to insert brief comments into Mathematica code (but for more lengthy explanatory comments, see below).

Working with notebooks

By now you've executed quite a few Mathematica instructions, many of which are minor modifications of earlier instructions. You may have typed some of these modifications in new cells, below your previous work, or you may have simply edited your previous code and re-executed it in the same cell.

You may be interested to know that as far as Mathematica is concerned, it makes *no difference* where you put each new instruction, or whether you delete earlier instructions. You can put new instructions into existing cells, or between them, or below them, or even in a totally separate window. All name assignments will continue to exist, in Mathematica's internal state, until you either quit Mathematica, or quit the so-called Kernel (which you can do using a menu command), or explicitly erase them using the `Clear` function. By the same token, when you save your work and reopen it during a new Mathematica session, you'll need to re-execute any cells containing name assignments that you wish to use in your subsequent work.

A Mathematica document, displayed in a single window, is called a *notebook*, and will be saved with the file extension `.nb`. Each notebook's contents are divided into cells, as indicated by the brackets at the window's right margin. Code execution takes place one cell at a time, so it's a good idea to put a group of closely related instructions into a single cell when they should always be executed together.

You can also create non-executable notebook cells that contain explanatory text and section headings. Go to Format → Style (in the menus) to denote the current cell as one of these non-executable types. Mathematica will automatically group cells together, within sections and so on, indicating the group structure with multiple levels of right-margin brackets. If you wish to delete an entire cell or group of cells, click on its bracket to select it and then choose Clear from the Edit menu. You can also "collapse" a cell group, hiding most of its content, by double-clicking on its bracket. Double-click again to un-collapse the group.

1.2 Lists, tables, and animation

In the examples above you saw how to create a short *list* by grouping items in curly braces, separated by commas: `{x, 0, Pi}` and `{Blue, Thick}` and so on. But you can also build a list from a formula, using the `Table` function. Here's a quick way to make a table of Hermite polynomials:

```
Table[HermiteH[n, x], {n, 0, 5}]
```

(Notice how the syntax is similar to that of `Plot`.) To format the table more nicely, try enclosing the preceding code in `TraditionalForm[TableForm[ ]]`.

As with any other expression, you can assign a name to a list:

```
shoStates =
  Table[HermiteH[n, x] Exp[-x^2/2]/Sqrt[2^n n! Sqrt[Pi]], {n, 0, 5}]
```

(Note the use of `!` for factorial.) To extract a particular element of the list, you put its index number in *double* square brackets:

```
shoStates[[3]]
```

Notice that the indexing starts with 1, not 0 (as in some computer languages), so element 3 is the third element of the list, not the fourth.

When you plot a list of functions, they all appear on the same set of axes:

```
Plot[shoStates, {x, -5, 5}]
```

Alternatively, you can use `Table` to make a list of plots:

```
Table[Plot[shoStates[[n]], {x, -5, 5}, PlotRange -> {-.8, .8}],
  {n, 1, 6}]
```

Here I've used an explicit `PlotRange` so all the plots will have the same scale. To format the table of plots more nicely, you can enclose this code in the `GraphicsColumn` function and adjust the width using `ImageSize->250` (for instance).

Alternatively, instead of showing many plots at once, you can use the `Manipulate` function to create a slider for choosing which plot to show:

```
Manipulate[
  Plot[HermiteH[n, x] Exp[-x^2/2]/Sqrt[2^n n! Sqrt[Pi]],
      {x, -12, 12}, PlotRange->{-.8, .8}], {n, 0, 50, 1}]
```

Click the little + sign next to the slider to see a numerical readout of its setting, along with animation controls for hands-free cycling through the slider settings.

1.3 Plotting complex functions

Mathematica handles complex numbers with ease; just use a capital `I` for the imaginary unit. Here, for instance, is a Gaussian wavepacket:

```
wavepacket = Exp[-x^2] Exp[I*5*x];
Plot[{Re[wavepacket], Im[wavepacket]}, {x, -3, 3}, PlotRange->All]
```

This plot shows the real and imaginary parts (`Re[]` and `Im[]`) separately. Usually, though, the real and imaginary parts of a complex function are of less concern to us than the magnitude and phase. A common way to emphasize the magnitude and phase is to plot the magnitude (or squared magnitude) of the function on the vertical axis, and fill the area beneath with color hues to indicate the phases. Here is the code to do that:

```
Plot[Abs[wavepacket], {x, -3, 3},
    PlotPoints -> 300,
    Filling -> Axis,
    ColorFunction -> Function[x, Hue[Arg[wavepacket]/(2Pi)]],
    ColorFunctionScaling -> False]
```

The absolute value function `Abs` gives the magnitude of a complex number. The critical option here is the `ColorFunction`, which uses the `Hue` function to generate a color from the phase angle (`Arg[psi1]`), scaled by 2π to give a value between 0 and 1. The `Function` function is one of Mathematica's trickiest features: it creates an "anonymous" function that associates the independent variable `x` (same that I used in defining `wavepacket`) with the desired hue. The `ColorFunctionScaling` option turns off the automatic remapping of colors to span the function's full range (see the following section for examples of when this is desirable), while the `Filling` option extends the coloring down to the x axis. The `PlotPoints` option ensures that Mathematica will evaluate the function at enough points to show the full detail of the color sequence.

Exercise: Time dependence of a superposition state

Here is a nice application of some of the tricks you've just learned. Suppose a quantum particle in a one-dimensional infinite square well is in a superposition of the ground state and the first excited state. Then in suitably chosen natural units, its time-dependent wavefunction can be written

$$\psi(x,t) = \sin(\pi x)\, e^{-it} + \sin(2\pi x)\, e^{-4it}, \tag{1}$$

where x ranges from 0 to 1. Use the `Manipulate` function to make an animated plot of this particle's probability density $|\psi|^2$, filling the area under the graph with colors to represent the wavefunction phases. Let t vary over one full period of the wavefunction's oscillation. What is the oscillation period of the probability density (ignoring the phases/colors)?

1.4 Plotting functions of two variables

Mathematica provides three different ways to plot a real-valued function of two variables. With a little work we can also plot a complex-valued function of two variables, using colors to show the phases.

Density plots

A density plot shows the two independent variables along the horizontal and vertical axes, while using a color scheme to indicate the function value at each point. The syntax is the same as for `Plot`, but with a second list specifying the second variable and its range:

```
psi320 = r^2 Exp[-r/3] (3 z^2/r^2 - 1) /. r -> Sqrt[x^2 + z^2];
rMax = 15;
DensityPlot[psi320^2, {x, -rMax, rMax}, {z, -rMax, rMax}]
```

This code plots a slice through the probability density of a hydrogen energy eigenstate. As with `Plot`, you'll often want to use some non-default options:

```
PlotRange->All,
PlotPoints->50
```

The `PlotRange` option prevents Mathematica from excluding the largest function values (showing them as ugly white patches); `PlotPoints` determines the sampling resolution of the plot in each dimension (with a default value of only about 15, which you'll usually want to increase).

The most interesting option for density plots is `ColorFunction`, which determines the color scheme. The easiest way to override the default color scheme is to use one of Mathematica's 50 or so named color schemes, such as:

```
ColorFunction -> "SunsetColors"
```

Look up "color schemes" in the documentation for a table of these named schemes.

Alternatively, you can create arbitrary color gradients using the `Blend` function. The syntax for a basic white-to-black gradient would be:

```
  ColorFunction -> Function[f, Blend[{White, Black}, f]]
```

Here `f` is merely a dummy variable, whose name is arbitrary, representing the plotted function's value.[2] The list of colors provided to `Blend` can include any of Mathematica's named colors, or arbitrary colors defined using `RGBColor` or `Hue`:

```
  ColorFunction -> Function[f,
          Blend[{Hue[.7, 1, .5], Hue[.9, .1, 1]}, f]]
```

Moreover, the list of colors can be of any length:

```
  ColorFunction -> Function[f,
          Blend[{Black, Blue, Cyan, White}, f]]
```

The possibilities are effectively endless.

But just because you *can* color your density plots in arbitrary ways doesn't mean you *should*. With the power to create arbitrary color schemes comes the responsibility to use a scheme that will highlight, rather than obscure, what the plot is actually trying to show. For a function whose values are always positive, it's usually best to use a scheme that changes monotonically either from dark colors to light ones, or from light to dark, but not both. By default, the full range of function values that occur within the plot will be mapped to the full range of colors. For a function whose values can be both positive and negative, it's best to map zero to either white or black (or some other reasonably neutral color), blending into different hues for positive and negative values:

```
  DensityPlot[Sin[2*Pi*x] Sin[3*Pi*y], {x, 0, 1}, {y, 0, 1},
      PlotPoints -> 50,
      ColorFunction -> Function[f, Blend[{Cyan, Black, Red}, f]]]
```

Note that a three-color blend will map zero to the middle color only when the plotted function's minimum value is minus its maximum value. When the function's minimum and maximum values are not symmetric about zero, you can shift and scale its values in the `Blend` function to map zero to 0.5 and the most extreme value to either zero or 1, then set `ColorFunctionScaling` to `False`, which maps the full color range to the interval from 0 to 1:

```
  psi3 = Sin[Pi*x] Sin[Pi*y] + Sin[2*Pi*x] Sin[2*Pi*y];
  DensityPlot[psi3, {x, 0, 1}, {y, 0, 1},
      PlotPoints -> 50,
      ColorFunction ->
          Function[f, Blend[{Cyan, Black, Red}, f*0.32 + 0.5]],
      ColorFunctionScaling -> False]
```

Alternatively, in cases like this you might find it easier to use the method described below for complex-valued functions.

[2]The `ColorFunction` for a `DensityPlot` is given the *value* of the function being plotted, unlike that for an ordinary `Plot`, which is given the independent variable. To see what variables get passed to the `ColorFunction` in different contexts, look it up in the documentation.

Contour plots

An alternative to `DensityPlot` is `ContourPlot`, which draws lines connecting points with the same function value:

```
ContourPlot[Sin[2 Pi x] Sin[3 Pi y], {x, 0, 1}, {y, 0, 1},
  ColorFunction -> Function[f, Blend[{Magenta, Black, Green}, f]]]
```

The syntax is the same as with `DensityPlot`, so you can switch between one and the other with just a few keystrokes.

One advantage of `ContourPlot` is that when you hover over a contour line with the cursor, Mathematica shows the numerical function value to which that line corresponds.

By default, a `ContourPlot` also includes the same coloring as a `DensityPlot`, except that it is discretized to show a uniform color between adjacent contour lines. Naturally there are options to omit the shading, and to change the total number of contour lines:

```
ContourShading -> False,
Contours -> 20
```

Surface plots

A surface plot of a function of two variables plots the function values on a third axis, to produce a three-dimensional surface. This surface is then drawn in perspective on a two-dimensional screen or printed page. The Mathematica function to produce such an image is `Plot3D`:

```
Plot3D[Sin[2*Pi*x] Sin[3*Pi*y], {x, 0, 1}, {y, 0, 1}]
```

Here are some options to try (preferably one at a time) that modify the plot's appearance:

```
Mesh -> None,
PlotPoints -> 100,
BoxRatios -> {1, 1, 0.6},
Axes -> False,
Boxed -> False,
ViewPoint -> {-0.5, -2, 1}
```

Notice that you can also rotate the view using the mouse (or other pointing device).

Of course you can also modify the color of the surface, but this can get complicated due to the simulated lighting and surface shininess. Some colors look ok without any further changes:

```
PlotStyle -> Hue[.33, .75, 1]
```

But most colors look odd, at least to my eyes, under the default colored light sources. The easiest fix is to switch to "neutral" lighting:

```
Lighting -> "Neutral"
```

This setting has the side effect of making the surface look too shiny (in my opinion), so I usually just turn the shininess completely off:

```
  PlotStyle -> {Hue[.75, .25, .85], Specularity[0]}
```

At this point you can experiment with a variety of different hue, saturation, and brightness values.

Complex-valued functions

The trick to making a phase-as-color density plot of a complex-valued function is not to use `DensityPlot` at all, but rather to use `RegionPlot`, which can fill any region in the plane using an arbitrary `ColorFunction`. In general you would specify the region to fill using a condition on `x` and `y`, but we can just fill the entire region using the trivial condition `True`. Here, then, is the code to plot a two-dimensional wavepacket:

```
  wp2d = Exp[-(x^2+y^2)] Exp[I(8x-5y)];
  RegionPlot[True, {x, -3, 3}, {y, -3, 3},
      PlotPoints -> 100,
      BoundaryStyle -> None,
      ColorFunction ->
          Function[{x, y}, Hue[Arg[wp2d]/(2Pi), 1, Abs[wp2d]]],
      ColorFunctionScaling -> False]
```

As before, I've set the hue to correspond to the phase of the function. Here I've also set the saturation to 1 and the brightness to the function magnitude, so we get black where the function value is zero or negligible. To map zero to white instead, set the brightness to 1 and the saturation to the function magnitude. This particular function has a maximum magnitude of 1, so the brightest parts of the image are as bright as possible. For other functions you may need to apply a scale factor to optimize the brightness.

An alternative is to make a surface plot of the function magnitude and then color the surface according to phase:

```
  Plot3D[Abs[wp2d], {x, -3, 3}, {y, -3, 3},
      PlotRange -> All,
      PlotPoints -> 100,
      ColorFunction ->
          Function[{x, y}, Hue[Arg[wp2d]/(2Pi), Abs[wp2d], 0.75]],
      ColorFunctionScaling -> False,
      Lighting -> "Neutral",
      Mesh -> None]
```

In this example I've made the areas where the function is negligible light gray, since neither black nor white looked very attractive to my eyes.

2 Computation

Enough of merely plotting formulas! Let's move on to some actual numerical calculations.

If you're continuing the same Mathematica session after working through the previous section, please erase Mathematica's knowledge of the symbol `psi`:

```
Clear[psi]
```

2.1 Integrals and sums

Integrals and sums come up all the time in quantum mechanics, when we take inner products and build superpositions. For example, suppose that a particle in a one-dimensional infinite square well (of width 1) has a rather narrow Gaussian wavefunction, centered 1/4 of the way from the left edge:

```
psi[x_] := Exp[-(x - .25)^2/.05^2];
Plot[psi[x], {x, 0, 1}, PlotRange -> All]
```

This wavefunction isn't normalized, so let's compute its normalization integral numerically using the `NIntegrate` function:

```
NIntegrate[psi[x]^2, {x, 0, 1}]
```

Now wrap a `Sqrt[]` function around this expression, take its reciprocal, and insert the result into the definition of `psi`. Then check that `psi` is indeed normalized.

To calculate either the probabilities of energy measurement outcomes or the time evolution of this wavefunction, we need to know its components in the energy eigenbasis, $\psi_n(x) = \sqrt{2}\sin(n\pi x)$. The $n = 1$ component is given by the integral

```
NIntegrate[psi[x] Sqrt[2] Sin[Pi x], {x, 0, 1}]
```

but let's go ahead and make a list of the first 30 components:

```
component =
    Table[NIntegrate[psi[x] Sqrt[2] Sin[n Pi x], {x, 0, 1}],
        {n, 1, 30}]
```

To eliminate the annoying warning messages from the near-zero results, you can insert `AccuracyGoal->10` as an option in `NIntegrate`. To display the results a little more nicely, try `TableForm[Chop[component]]`. Or you can plot the components using the `ListPlot` function. To display or plot the corresponding energy probabilities, just change `component` to `component^2`.

Now let's reassemble the wavefunction from its components:

```
Plot[Sum[component[[i]] Sqrt[2] Sin[i Pi x], {i, 1, 3}], {x, 0, 1},
    PlotRange -> All]
```

Notice that the `Sum` function has the same syntax as `Plot` and `Table` and `NIntegrate`. Increase the limit on the sum and watch how the series converges!

2.2 Bound states in one dimension

The paradigm one-dimensional bound state problems are the infinite square well and the harmonic oscillator. Most textbooks cover one or two other examples as well, but these quickly become much more difficult. The good news is that with the same effort or less, we can teach students to solve the time-independent Schrödinger equation (TISE) for one-dimensional potential wells of *any* shape.

The shooting method

This method is actually covered in some textbooks, so it may already be familiar to you. We simply guess an energy and then integrate the TISE across the potential well, starting with some reasonable boundary conditions. Normally we get a wavefunction that diverges as $x \to \infty$, so we then adjust our energy guess and try again until we get a normalizable wavefunction.

For definiteness let's consider a finite square well potential with width 1 and depth 50 (in natural units):

```
v[x_] := If[Abs[x] < 0.5, 0, 50];
xMax = 1.5;
Plot[v[x], {x, -xMax, xMax}, Exclusions -> None]
```

For the sake of symmetry I've centered the well at $x = 0$. Notice the use of Mathematica's `If` function, and the `Exclusions -> None` option in `Plot` to connect the graph across the discontinuities.

Before continuing, please enter `Clear[psi]` to (again) erase any earlier definition of `psi` that might be in Mathematica's memory.

To solve the TISE,

$$\frac{d^2\psi}{d\psi^2} = -2(E - V(x))\psi, \tag{2}$$

we can use the `NDSolve` function, whose syntax is a little tricky:

```
energy = 5;
solution =
    NDSolve[{psi''[x] == -2 (energy - v[x]) psi[x], psi[-xMax] == 0,
        psi'[-xMax] == 0.001}, psi, {x, -xMax, xMax}];
Plot[psi[x] /. solution, {x, -xMax, xMax}]
```

The first argument of `NDSolve` is a list of three equations: the differential equation to solve and the boundary conditions on the function and its derivative. Notice the double `==` sign to indicate that these are equations to be solved, not variable name assignments. The prime symbol `'` indicates a derivative. For boundary conditions I've set the wavefunction to zero and its derivative to a small but arbitrary constant at the left edge of the region of interest. My initial guess for the energy is close to that of an infinite square well of the same width, $\pi^2/2$. The `NDSolve` function awkwardly returns its result in the form of a so-called interpolating function embedded in a substitution rule inside a list, so to extract it for plotting I've given this structure a name (`solution`) and then used the `/.` substitution syntax.

When you execute this code you should find that the wavefunction crosses the x axis and then diverges to minus infinity. The correct ground state energy (with zero nodes) is therefore less than 5. So try a lower energy next, and then repeatedly adjust the energy until you've pinned it down to a few decimal places. Once you've got the ground state, start over with a higher energy to obtain the first excited state, and so on. Eventually you'll find that you need to increase the value of `xMax` in order to obtain a wavefunction that has the correct asymptotic behavior on both sides. Energies above 50, however, won't produce bound states at all and are therefore of little interest to us for the time being.

The shooting method is accurate, computationally efficient, and easy to code. But it gives us only one solution at a time, and guessing the energies can be awkward and hard to automate.

The matrix method

An alternative method of solving bound state problems is to introduce a set of basis functions, express the TISE in matrix form using this basis, and then diagonalize the Hamiltonian matrix to obtain the energy eigenvalues and eigenvectors. This method can be computationally expensive, but it gives us many solutions at once and it helps students see the connection between wave mechanics and matrix mechanics.[3]

For our basis functions we'll use the eigenfunctions of a fictitious infinite square that is wide enough (we hope!) to contain all the eigenfunctions that we care about:

$$\psi_n(x) = \sqrt{\frac{2}{b}}\sin\Bigl(\frac{n\pi x}{b}\Bigr) \qquad \text{for } 0 < x < b, \tag{3}$$

where b is the width of the fictitious infinite well. (When I apply this method to the finite square well example in a moment, I will take b to be several times larger than the well width, and I'll center the finite well at $b/2$.) Any unknown energy eigenfunction $\psi(x)$ of our actual potential $V(x)$ can be expressed as a linear combination of these basis functions,

$$\psi = \sum_{n=1}^{\infty} c_n\psi_n, \tag{4}$$

and our goal is to find the unknown coefficients c_n. To do so we first calculate the matrix elements of the Hamiltonian operator in our basis,

$$H_{mn} = \int_0^b \psi_m(x)\hat{H}\psi_n(x)\,dx = \langle\psi_m|\hat{H}|\psi_n\rangle. \tag{5}$$

The TISE then becomes a matrix eigenvalue equation:

$$\begin{pmatrix} H_{11} & H_{12} & \cdots \\ H_{21} & H_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}\begin{pmatrix} c_1 \\ c_2 \\ \vdots \end{pmatrix} = E\begin{pmatrix} c_1 \\ c_2 \\ \vdots \end{pmatrix}. \tag{6}$$

[3]This method has been popularized in recent years by a series papers by Frank Marsiglio and his collaborators, beginning with "The harmonic oscillator in quantum mechanics: A third way," Am. J. Phys. **77**(3), 253–258 (2009), `https://doi.org/10.1119/1.3042207`. For a more complete list of references see Kevin Randles et al., "Quantum matrix diagonalization visualized," `https://arxiv.org/abs/1905.13269`.

(For a detailed derivation of this matrix equation, starting from the TISE in its usual form, see the paper by Marsiglio cited earlier or the draft textbook cited in the introduction.)

Calculating the matrix elements is somewhat simplified if we break the Hamiltonian into two pieces:

$$\hat{H} = \hat{H}_0 + \Delta V(x). \tag{7}$$

Here $\hat{H}_0$ is the Hamiltonian whose eigenfunctions are $\psi_n(x)$, and $\Delta V(x)$ is whatever is left of our actual Hamiltonian. For our infinite-square-well basis, $\hat{H}_0$ is just the kinetic energy operator and $\Delta V(x)$ is simply the $V(x)$ for our actual potential energy function. Thus, the matrix elements are

$$\begin{aligned} H_{mn} &= \int_0^b \psi_m(x)\hat{H}_0\psi_n(x)\,dx + \int_0^b \psi_m(x)V(x)\psi_n(x)\,dx \\ &= E_n\delta_{mn} + \int_0^b \psi_m(x)V(x)\psi_n(x)\,dx, \end{aligned} \tag{8}$$

where in the first term I've used the fact that ψ_n is an eigenfunction of H_0 with eigenvalue E_n, and the fact that the ψ_n functions are orthonormal. Written out as a matrix, the first term would be simply a diagonal matrix whose entries are the eigenvalues of the infinite square well, $\pi^2 n^2/(2b^2)$ in natural units.

The Mathematica code to solve a system by the matrix method is shorter than the theoretical summary I've just given. First we define our potential function:

```
b = 4;
v[x_] := If[Abs[x - b/2] < 0.5, 0, 50];
Plot[v[x], {x, 0, b}, Exclusions -> None]
```

Here I've taken the potential to be the same finite square well as above, but shifted it to the right so it's centered inside the fictitious infinite square well of our basis functions. Next we decide how many basis functions to use, define these basis functions, and calculate all the matrix elements:

```
nMax = 10;
basis[n_, x_] := Sqrt[2/b] Sin[n Pi x/b];
hMatrix =
    Table[(n^2 Pi^2/(2 b^2)) KroneckerDelta[m, n] +
      NIntegrate[basis[m, x] v[x] basis[n, x], {x, 0, b},
          AccuracyGoal->10], {m, 1, nMax}, {n, 1, nMax}];
```

(The `AccuracyGoal` option suppresses warnings when matrix elements are zero due to symmetry.) At this point you may optionally wish to view the matrix by typing `MatrixForm[Chop[hMatrix]]`, or display it graphically with the following code:

```
contrast = 0.01;
MatrixPlot[hMatrix*contrast + 0.5,
  ColorFunction -> Function[f, Blend[{Green, White, Purple}, f]],
  ColorFunctionScaling -> False]
```

(The nonstandard color function is pretty necessary here, but feel free to choose different colors and to adjust the `contrast` parameter as needed.)

To diagonalize the matrix we simply use the `Eigensystem` function,[4] which returns a list consisting of the eigenvalues followed by the eigenvectors:

```
{eValues, eVectors} = Eigensystem[hMatrix];
eValues
```

The eigenvalues are inconveniently sorted in descending order. We're interested only in the lowest ones, below 50, which represent bounds states of the finite well. Unfortunately, keeping only 10 basis functions doesn't give very accurate eigenvalues. So once you have the code working, please increase `nMax` until the bound state eigenvalues have approximately converged. Do they agree with the values you found using the shooting method?

You can easily plot an energy level diagram by passing the list of eigenvalues to `Plot`:

```
Plot[eValues, {x, 0, 1}, PlotRange -> {0, 50},
    Ticks -> {None, Automatic}]
```

Another way to check convergence is to look at the components of the eigenvectors:

```
ListPlot[eVectors[[-1]], PlotRange -> All, Filling -> Axis]
```

The list index -1 gives the last element of the list, in this case the eigenvector with the smallest eigenvalue. Change it to -2, -3, etc., to examine the rest of the bound state eigenvectors.

To see the actual eigenfunctions, we need to build them as superpositions of the basis functions according to equation 4:

```
Plot[Sum[eVectors[[-1, n]] basis[n, x], {n, 1, nMax}], {x, 0, b}]
```

Do all the bound state eigenfunctions look reasonably similar to those you found by the shooting method? Can you explain any differences that you notice?

As you increased the value of `nMax`, you probably noticed that computing all the matrix elements by numerical integration becomes a bit time-consuming. The cited paper by Marsiglio shows how to speed up these calculations enormously, at the cost of a few more lines of code, using a product-to-sum trig identity. For the finite square well you can also just compute the integrals analytically, but that doesn't work for arbitrary potential shapes.

Exercise: The quantum bouncer

I've illustrated the shooting method and the matrix method for a finite square well potential, but both methods work for one-dimensional potential wells of any shape. One interesting example is the so-called quantum bouncer potential,

$$V(x) = \begin{cases} \alpha x & \text{for } x > 0 \\ \infty & \text{for } x < 0, \end{cases} \tag{9}$$

[4]Admittedly, `Eigensystem` is a black box whose inner workings will be a total mystery to most students. For an attempt to make the diagonalization process less mysterious, see the paper by Randles et al. cited previously.

where α is a constant. Find the low-lying energy eigenvalues and eigenfunctions for this potential, using your choice (or both!) of the methods of this section. Use natural units in which $\alpha = 1$. Be sure to consider a wide enough range of x values that the low-lying eigenfunctions have room to go asymptotically to zero at large x. Note, however, that because all eigenfunctions must be exactly zero for $x < 0$, there is no need to look at negative x values at all.

2.3 Scattering in one dimension

Numerical methods are just as useful for one-dimensional scattering states as they are for bound states. Here I'll describe two very different approaches to scattering, one based on stationary states and the other based on wavepackets.

Energy eigenstates by the shoot-once method

Suppose we have a localized potential barrier or well in one dimension, and we send in a right-moving particle with well-defined momentum k. We expect the potential to reflect the particle at momentum $-k$ with some probability R, or to transmit the particle with momentum k with probability $T = 1 - R$. (For simplicity I'll assume that the potential does not involve any "step," so the transmitted momentum is the same as the incident momentum. It's easy, though, to relax this assumption.) In summary, the wavefunction away from the nontrivial part of the potential has the form

$$\psi(x) = \begin{cases} Ae^{ikx} + Be^{-ikx} & \text{as } x \to -\infty, \\ Ce^{ikx} & \text{as } x \to +\infty. \end{cases} \tag{10}$$

The reflection probability is then $R = |B|^2/|A|^2$.

This wavefunction must obey the TISE everywhere, so we can find the unknown amplitudes by numerically integrating the TISE for any desired k. The trick is to start on the right, where ψ has a simple known form, and "shoot" just once, integrating leftward across the potential until we're in the asymptotic region where we can essentially read off the coefficients A and B. Suitable boundary conditions on the right-hand side are $\psi = 1$ and $\psi' = ik$, both at the same (sufficiently large) x, arbitrarily choosing the magnitude of C to be 1.

To demonstrate the method I'll use a simple rectangular barrier, with "height" 50 in arbitrary energy units:

```
v[x_] := If[x > 0 && x < 1, 50, 0];
xMax = 3;
Plot[v[x], {x, -xMax, xMax}, Exclusions -> None,
    AxesOrigin -> {-xMax, 0}]
```

To solve the TISE we simply use `NDSolve` with the appropriate boundary conditions. The complex wavefunction poses no difficulty at all for `NDSolve`, though we have to work a little harder to plot it:

```
energy = 40;
k = Sqrt[2 energy];
solution =
  NDSolve[{psi''[x] == -2 (energy - v[x]) psi[x], psi[xMax] == 1,
      psi'[xMax] == I*k}, psi, {x, -xMax, xMax}];
Plot[Abs[psi[x]]^2 /. solution, {x, -xMax, xMax},
    PlotRange -> {0, All}, Filling -> Axis,
    ColorFunction -> Function[x, Hue[Arg[psi[x]/.solution]/(2Pi)]],
    ColorFunctionScaling -> False, PlotPoints -> 500]
```

For the numbers shown you should obtain a large-amplitude interference pattern to the left of the barrier, with an exponential attenuation in the classically forbidden barrier and a negligibly small amplitude (by comparison) on the right. Now try it again with a barrier width of 0.1 instead of 1, and see how there's now a significant tunneling probability.

The larger the interference fringes in comparison to the average wave amplitude on the left, the greater the reflection probability. It's a reasonably easy exercise[5] to make this idea precise and derive the formula

$$R = \frac{|B|^2}{|A|^2} = \left(\frac{|\psi|_{\max} - |\psi|_{\min}}{|\psi|_{\max} + |\psi|_{\min}}\right)^2, \tag{11}$$

where "max" and "min" refer to the peaks and troughs of $|\psi|$ in the left-hand region. In Mathematica code this becomes:

```
max = FindMaximum[Abs[psi[x]] /. solution,
        {x, -.8 xMax, -.2 xMax}][[1]];
min = FindMinimum[Abs[psi[x]] /. solution,
        {x, -.8 xMax, -.2 xMax}][[1]];
reflection = ((max - min)/(max + min))^2
```

(I've specified a somewhat arbitrary search interval to Mathematica's numerical optimization functions. Each of these functions returns a list whose first element is the location of the extremum, so I've extracted this element with `[[1]]`.)

Time evolution of wavepackets

Using stationary states in scattering calculations can be confusing for students, because there doesn't appear to be anything actually going in or coming out. A more intuitive alternative is to solve the time-*dependent* Schrödinger equation (TDSE), with a localized right-moving wavepacket as the initial state. Let me now explain how to do this.

In natural units the TDSE is

$$i\frac{\partial\psi}{\partial t} = -\frac{1}{2}\frac{\partial^2\psi}{\partial x^2} + V(x)\psi. \tag{12}$$

[5]See the draft textbook cited in the introduction, or R. C. Greenhow and J. A. D. Matthew, "Continuum computer solutions to the Schrödinger equation," Am. J. Phys. **60**(7), 655–663 (1992).

We discretize x into a list of evenly spaced values, using the "second centered difference approximation" for the second spatial derivative:

$$i\frac{\partial\psi}{\partial t} = -\frac{1}{2}\frac{\psi(x{+}dx) + \psi(x{-}dx) - 2\psi(x)}{(dx)^2} + V(x)\psi(x), \tag{13}$$

where dx is now the (non-infinitesimal) spacing between adjacent x values. To simplify this equation I will use units in which $dx = 1$. Then with a bit of rearrangement, the TDSE becomes

$$i\frac{\partial\psi}{\partial t} = -\tfrac{1}{2}\bigl[\psi(x{+}1) + \psi(x{-}1)\bigr] + \bigl[1 + V(x)\bigr]\psi(x). \tag{14}$$

We still need to discretize time, replacing $\partial\psi/\partial t$ with a discrete approximation. The easiest way to do this would be to write

$$\frac{\partial\psi}{\partial t} \approx \frac{\psi(x, t{+}dt) - \psi(x, t)}{dt}, \tag{15}$$

with the understanding that each ψ on the right-hand side of equation 14 is evaluated at t (rather than at $t + dt$ or some other time). But this approximation is inaccurate because it has a forward bias, taking the future time $t + dt$ into account but not the past time $t{-}dt$. It is much more accurate to use the centered-difference approximation

$$\frac{\partial\psi}{\partial t} \approx \frac{\psi(x, t{+}dt) - \psi(x, t{-}dt)}{2\,dt}, \tag{16}$$

which treats the near future and recent past symmetrically. Inserting this approximation into equation 14 and solving for $\psi(x, t{+}dt)$, we obtain our fully discretized version of the TDSE:

$$\psi(x, t{+}dt) = \psi(x, t{-}dt) + i\,dt\bigl[\psi(x{+}1, t) + \psi(x{-}1, t) - 2\bigl(1 + V(x)\bigr)\psi(x, t)\bigr]. \tag{17}$$

Equation 17 is the kernel of what I call the *centered-difference method* of solving the TDSE. The procedure is to loop over all x points on the discretized lattice, using this equation to calculate $\psi(x, t{+}dt)$, and then step forward in time and repeat the process to calculate ψ at the next time increment, and the next, and so on. There are just a few more details to fill in:

- **Choosing the time step.** The calculation will go faster if we use a larger value of dt, but it turns out that if we make dt too large, the process becomes unstable: small spurious inaccuracies grow exponentially with time until ψ becomes huge and nonsensical. The largest dt you can get away with depends on $V(x)$.[6] In our units this largest value is always less than 0.5, and in practice $dt = 0.45$ works well most of the time.

[6]See P. B. Visscher, "A fast explicit algorithm for the time-dependent Schrödinger equation," Computers in Physics **5** (6), 596–598 (1991); or H. Gould, J. Tobochnik, and W. Christian, *An Introduction to Computer Simulations Methods,* third edition (Pearson, San Francisco, 2007, electronic version available at `http://www.opensourcephysics.org/items/detail.cfm?ID=7375`), equation 16.34.

- **Boundary conditions.** Equation 17 won't work at either end of the spatial interval of our calculation, because it would then refer to x values that are beyond the ends by dx. The usual approach is therefore to set $\psi = 0$ at each end at all times, effectively embedding our system inside an infinite square well. We then use equation 17 only at the interior points.

- **Getting started.** Of course we must provide an initial wavefunction, $\psi(x, 0)$. But in order to use equation 17 for the first time, we also need to know $\psi(x, -dt)$, one step *before* time zero. A simple solution is to calculate it using equation 15 with a negative value of dt. Even though this approximation isn't very accurate, the harm is minimal because we're using it only once.

Now let me show an implementation of the centered-difference method using Mathematica. I'll start by defining the size of the spatial lattice, creating a table of potential energy values, and plotting the potential function:

```
xMax = 500;
v = Table[If[x > 250 && x < 255, 0.05, 0], {x, 1, xMax}];
ListPlot[v, Joined -> True]
```

Because Mathematica list indices start at 1 rather than 0, it's easiest to let `x` range from 1 to `xMax`. My arbitrary potential energy function is again a rectangular barrier, with a width and height that work well when the grid spacing is one distance unit.

Next I'll initialize the time variable, the time step, and the wavefunction itself:

```
t = 0;
dt = 0.45;
x0 = 100;
p0 = 0.25;
a = 30;
psi = Table[Exp[-(x-x0)^2/a^2] Exp[I p0 x], {x, 1, xMax}];
psiLast = psiNext = Table[0, {x, 1, xMax}];
For[x = 2, x < xMax, x++,
  psiLast[[x]] = psi[[x]]
    - (I*dt/2)*(psi[[x+1]] + psi[[x-1]] - 2(1+v[[x]])psi[[x]])];
```

The initial wavefunction is a Gaussian wavepacket, centered in the region where the potential energy is zero, with a positive momentum so it will move to the right. I chose the value of p_0 so the nominal wavelength, $2\pi/p_0$, will be significantly larger than the lattice spacing, $dx = 1$. After initializing the "current" wavefunction `psi`, I've initialized the wavefunctions at the previous and next time steps (`psiLast` and `psiNext`) to zero—although here it would actually suffice just to initialize the end-points. The `For` loop at the end of this code block uses equation 15 to properly initialize `psiLast`.

The initializations are now complete, but before we start calculating forward in time we need a plot to show what this wavefunction looks like. This part of the code will be pretty arcane, but I hope you'll like the result:

```
Dynamic[(psiInterp = Interpolation[psi];
  Plot[Abs[psiInterp[x]]^2, {x, 1, xMax},
    PlotRange -> {0, 1},
    Filling -> Axis,
    ColorFunction -> Function[x, Hue[Arg[psiInterp[x]]/(2Pi)]],
    ColorFunctionScaling -> False,
    PlotPoints -> 500,
    Epilog -> Inset[Style[StringJoin["t = ",
        ToString[Round[t]]], 20], {10, 0.9}, {Left, Top}]] )]
```

The outer `Dynamic` function causes the plot to update itself whenever `psi` changes (as it soon will). Before creating the plot we create what Mathematica calls an interpolating function, so it can evaluate the function at any `x` value it wants (not just at integer values). The function we're plotting is $|\psi|^2$, with the area beneath it colored according to phase as usual. The `Epilog` adds text to show the current value of the time variable, rounded to the nearest integer, in 20-point type, near the upper-left corner of the plot.

After all these preliminaries, the simulation code itself is remarkably concise:

```
While[t < 1200,
  For[x = 2, x < xMax, x++,
    psiNext[[x]] = psiLast[[x]] +
        I*dt*(psi[[x+1]] + psi[[x-1]] - 2*(1+v[[x]])psi[[x]])];
  psiLast = psi;
  psi = psiNext;
  t += dt];
```

The `For` loop uses equation 17 to calculate the new wavefunction at each (interior) point. The rest of the code gets us ready for the next time step, by copying the current wavefunction into the space for the previous one, copying the next wavefunction into the space for the current one, and incrementing the time variable. All of this code repeats until the time variable reaches 1200, long enough for the wavepacket to move a few hundred distance units at its nominal initial velocity of 0.25.

Once you have this code working, you can modify it to handle virtually any initial wavefunction and any potential—localized or not. The main restriction is the discrete lattice of x values, which limits the wavefunction momentum, the abruptness of changes in the potential, and the total space in which everything needs to happen. I suggest starting with small changes to the (nominal) wavefunction momentum and to the barrier width and height. Then try a barrier with a gradual slope, or try a step potential, or replace the barrier with a negative-energy potential well.

2.4 Beyond one dimension

What about problems in more than one spatial dimension? I'll postpone those for another day, and merely conclude with a few general remarks:

- Multidimensional problems are important! Actual space has three dimensions, and we also need additional dimensions to handle systems of more than one particle. Generic multi-particle states are *entangled*, that is, they cannot be separated into products of single-particle states.[7]

- Multidimensional problems are hard! When we discretize space, the number of grid locations—and therefore the computational cost—grows exponentially with the number of dimensions. (This is why the idea of quantum computers is so attractive, as Feynman famously pointed out long ago.[8])

- If you can separate a multidimensional problem into two or more one-dimensional problems, you should! At a resolution of 100 grid locations in each dimension, a two-dimensional problem requires 10,000 grid locations, but two one-dimensional problems require only 200 grid locations. (Separable systems, however, are by definition never entangled.)

- The centered-difference algorithm for solving the TDSE generalizes easily to two (or more) dimensions, but is too computationally expensive to try in Mathematica. In a lower-overhead computing environment you can run 2D simulations on a 400×400 lattice at nice frame rates on today's personal computers.[9]

- The shooting method doesn't generalize at all to higher dimensions, because there are now boundary conditions on all sides; there's no good place to shoot from, or to shoot to.

- The matrix method does generalize to higher dimensions, though it also becomes computationally expensive due to the large number of required basis states. Still, in two dimensions it's quite feasible in Mathematica.

- I've published a paper[10] on solving multidimensional bound state problems using a finite-difference relaxation method, similar to the standard relaxation approach to Laplace's equation.

- In a quantum mechanics course, my preferred method for solving two-dimensional bound state problems is the somewhat slower "fake time" method, in which we evolve an initial trial function according to the TDSE with an imaginary time variable. This method is explained, though not yet coded, in the draft textbook cited in the introduction.

[7]See D. V. Schroeder, "Entanglement isn't just for spin," Am. J. Phys. **85**(11), 812–820 (2017), `https://doi.org/10.1119/1.5003808`.

[8]R. P. Feynman, "Simulating Physics with Computers," Intl. J. Theor. Phys. **21**, 467–488 (1982), `https://doi.org/10.1007/BF02650179`.

[9]For an implementation in JavaScript, see "Quantum Scattering in Two Dimensions," `http://physics.weber.edu/schroeder/software/QuantumScattering2D.html`.

[10]D. V. Schroeder, "The variational-relaxation method for finding quantum bound states," Am. J. Phys. **85**(9), 698–704 (2017), `https://doi.org/10.1119/1.4997165`