PHAS0030: Further Practical Mathematics & Computing
Session 6: Partial Differential Equations

David Bowler

February 15, 2019

In this session, we will consider how to solve the classical wave equation, and its quantum equivalent, the Schrödinger equation (both time-dependent and time-independent). We will use finite differences for the classical case, and see how well this works in one and two dimensions. We will find that we can use methods already developed for both forms of the Schrödinger equation, after we have introduced complex numbers within Numpy.

Contents

1 Objectives

The objectives of this session are to:

- Understand how to solve the classical wave equation in one dimension
- See how to extend these solution techniques to two dimensions
- Solve the time-dependent Schrödinger equation using the Crank-Nicolson approach we used for the heat equation in Session 5
- Explore solution methods for the time-independent Schrödinger equation, matching boundary conditions and finding eigenvalues

2 Review of Session 5

In the fifth session, we saw how matrices could be used to solve the boundary value problem for the simple, steady-state heat equation introduced in Session 4. We recalled the different types of second-order partial differential equations, and the relevant boundary conditions for each of these types. We examined parabolic equations, using a simple finite difference approach to solve the full, time dependent heat equation. We also introduced the Crank-Nicolson equation, and saw how it is stable (where the simple finite difference method is unstable once a parameter determined by the time step and grid spacing is too large). We ended by looking at elliptic equations, and a mapping from a two-dimensional grid onto a matrix-vector equation. We then showed how this mapping could be bypassed and replaced with an iterative solver.

3 The classical wave equation

In one dimension, the *classical* wave equation can be written:

$$\frac{\partial^2\theta}{\partial t^2} = c^2\frac{\partial^2\theta}{\partial x^2} \tag{1}$$

where c is the wave speed, and θ is the displacement appropriate to the wave (e.g. electric or magnetic field for EM waves, pressure for sound waves, etc). This is a *hyperbolic* equation, so that the solutions propagate forwards in time from a given initial value of θ and $\partial\theta/\partial t$ for all values of x in the computational domain.

Since we have two second order differentials, we know that the centred difference formula is reasonably accurate, and we will discretise time and space using steps Δt and Δx. The resulting finite difference formulae give us:

$$\begin{aligned}
\frac{\theta_{i,n+1} - 2\theta_{i,n} + \theta_{i,n-1}}{\Delta t^2} &= c^2 \frac{\theta_{i+1,n} - 2\theta_{i,n} + \theta_{i-1,n}}{\Delta x^2} && (2)\\
r &= c\Delta t/\Delta x && (3)\\
\theta_{i,n+1} &= 2\theta_{i,n} - \theta_{i,n-1} + r^2\left[\theta_{i+1,n} - 2\theta_{i,n} + \theta_{i-1,n}\right] && (4)\\
\theta_{i,n+1} &= 2(1-r^2)\theta_{i,n} - \theta_{i,n-1} + r^2\left[\theta_{i+1,n} + \theta_{i-1,n}\right] && (5)
\end{aligned}$$

We see that there is a parameter, $r = c\Delta t/\Delta x$, which is dimensionless. We will see that if $r > 1$ then the solution is unstable; as we are using a spatial differential that only accounts for nearest neighbours, if the wave propagates further than this in one timestep, we will introduce an error. This limits the size of the timestep Δt that can be chosen given a particular spatial step, Δx.

We need to think about boundary conditions quite carefully. Many waves are periodic, and we can model them with periodic boundary conditions, where the spatial dimensions are wrapped (so that, for a grid with points running from $0 \to N-1$, N maps to 0, $N+1$ to 1 and -1 maps to $N-1$; this fits quite naturally with Python arrays). Note that the choice of grid with a periodic wave is very important: if we use `np.arange` and specify a step, then the wavelength must be an integer multiple of the step; if, however, we use `np.linspace` then we must avoid duplicating the start and end of the wave[1]. We may, however, be working in a finite system where there is a fixed point at either end, and in this case we will have to impose the boundary conditions (and be careful when using `np.roll` to ensure that we do not bring information from one end of the array into the other)[2].

[1] I find it simpler to define a number of points, calculate a step size and use `np.arange` for periodic waves.

[2] In this case, `np.linspace` seems the simpler choice to me, as it makes it easy to specify the end points.

We now turn to initial conditions. In the form in Eq. (5) we see that, rather than specify the value and its time derivative, we instead specify the wave at two successive time steps; this is of course the same information in the context of a finite difference approach. Overall, this wave equation is quite simple to implement, using techniques that should be familiar from recent sessions.

3.1 Exercises

In-class

1. Create an update function based on Eq. (5). It should take as parameters two arrays, θ_n (the wave at all x points for timestep n) and θ_{n-1} (the wave at all x points for timestep $n-1$), and the value of r. It should return a new array θ_{n+1}. You may find `np.roll` useful[3].

[3] It's worth thinking briefly at this point about how `np.roll` might be used when the wave is *not* periodic: how would you adapt it?

2. We will now test this function on a simple sine wave. Set up frequency $f = 1$Hz, wavelength $\lambda = 1$m (though do NOT use a variable named `lambda`), and calculate speed, wavevector and angular frequency from these. Now define the number of points

in a wavelength to be 20, and set up an array for x that will hold *three* wavelengths.

(a) For $r = 0.1$, calculate $\Delta t = r\Delta x/c$, evaluate the wave analytically at the first two time steps (use $\sin(kx - \omega t)$ for $t = 0$ and $t = \Delta t$) and propagate the wave for 200 steps using a `for` loop and your update function. (Note that after each call to the update function you will need to update your variables; if you have `theta_0`, `theta_1` and `theta_n`, say, then you will need to set `theta_0 = theta_1` etc after each update.) Plot the wave along with the expected form: how well do they agree?

(b) Now explore values of r from $r = 0.2$ to $r = 0.9$ and note how the error increases; you could plot the difference between the numerical and analytic values to make it clearer.

3. Now for $r = 0.1$, test the effect of varying N_λ, the number of points in a wavelength. Try values between 5 and 50. Again, propagate the wave for 200 steps and compare to the expected form; note both the shape of the wave, and the location of the peaks.

Further work

1. We now want to explore the effect of discretisation on the *phase* velocity of the wave. For r=0.9, run a wave over a time interval of 9s with $N = 5$ and $N = 20$, keeping only the final waveform in both cases. (Note that you will need to use `int(9.001/dt)-1` to do this, as we start at $t = 0$.) Plot the two, along with the initial waves (which should be identical) on different subplots. You should see that the relationship between the initial and final waves is very different in the two cases.

4 *Two dimensions*

Extending this simple approach from one dimension to two dimensions (or even three dimensions) is remarkable simple. We need to replace the second derivative of position with the Laplacian:

$$\nabla^2\theta = \frac{\partial^2\theta}{\partial x^2} + \frac{\partial^2\theta}{\partial y^2} \tag{6}$$

which we then convert into a finite difference expression:

$$\begin{aligned}\nabla^2\theta &\simeq \frac{\theta_{i+1,j} - 2\theta_{i,j} + \theta_{i-1,j}}{\Delta x^2} + \frac{\theta_{i,j+1} - 2\theta_{i,j} + \theta_{i,j-1}}{\Delta y^2} & (7)\\ &= \frac{\theta_{i+1,j} + \theta_{i-1,j} + \theta_{i,j+1} + \theta_{i,j-1} - 4\theta_{i,j}}{h^2} & (8)\end{aligned}$$

where we have assumed that $\Delta x = \Delta y = h$ in the last line, and we are using $x = i\Delta x$ and $y = j\Delta y$. This should look very familiar from Session 5.

We can apply the exact same approach as we saw in Eq. (5) to propagate the solution of the wave:

$$\begin{aligned} \theta_{i,j,n+1} &= 2\theta_{i,j,n} - \theta_{i,j,n-1} + r^2\Delta^2\theta_{i,j,n} && (9)\\ \Delta^2\theta_{i,j,n} &= \theta_{i+1,j,n} + \theta_{i-1,j,n} + \theta_{i,j+1,n} + \theta_{i,j-1,n} - 4\theta_{i,j,n} && (10) \end{aligned}$$

where now $r = c\Delta t/h$.

The initial conditions are the same as were required for one dimension: values of θ for all x and y at $t = 0$ and $t = \Delta t$. With two dimensions, we can consider mixed boundary conditions (periodic in one direction and fixed in the other) as might be found in a channel or waveguide.

4.1 *Surface plots*

We have used `plt.imshow` with colour maps to represent 2D data so far in the course. It is also relatively easy to plot surfaces using Matplotlib, and this can be helpful with this type of problem. We need a new import, and we have to use the `figure` approach to creating plots:

```
from mpl_toolkits.mplot3d import Axes3D

fig_3d = plt.figure()
ax3d = fig_3d.add_subplot(111,projection='3d')
```

Note the optional argument `projection='3d'` (which only works after importing the extra module). If we plot a surface now using the command `ax3d.plot_surface(x2d,y2d,wave)`, we find something like Fig. 1(a). This is a reasonable representation of the

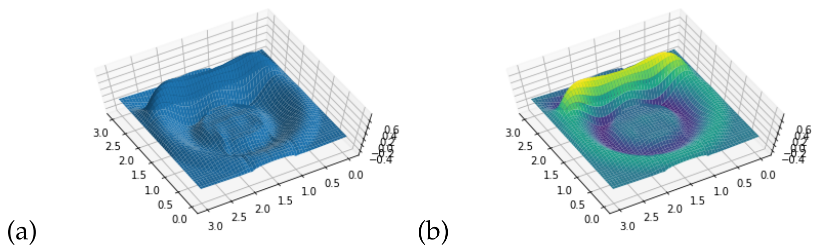

Figure 1: (a) A 3D projection of a wave in 2D with default parameters. (b) The same projection with a colormap applied.

wave, but lacks clarity. We can add a colormap; then we plot with `ax3d.plot_surface(x2d,y2d,wave,cmap='viridis')` and find the representation in Fig. 1(b). You can adjust the viewpoint using `ax3d.view_init(elevation, azimuth)` where `elevation` is the angle above the x–y plane and `azimuth` is the angle in the x–y plane.

A word of caution: be *very* careful with 3D plots: it is easy to produce something that *looks* nice but conveys *less* information than an equivalent 2D plot. It is also very easy to spend a lot of time getting something to look just right without any reward (i.e. marks...).

4.2 *Exercises*

In-class

1. Extend your solver from Sec. 3.1 to work for two dimensions (for two dimensions, use `np.roll(array,n,axis=0)` for the x derivatives (setting $n = 1$ or $n = -1$ as appropriate) and `np.roll(array,n,axis=1)` for the y derivatives). The change to the derivative is the only change that you need make.

2. Create a two-dimensional space to hold the wave, with both x and y covering three wavelengths with appropriate grid spacing (choose a number of points). Use `np.meshgrid` to create appropriate arrays for x and y. (Use a wavelength of 1m again.)

3. Create an initial wave that is made from a sine wave in x and t, `np.sin(wavevector*x2d - ang_freq*t)`, multiplied by a Gaussian function in y: `np.exp(-(y2d-midy)**2/sigma)`. Set `midy` to be the middle of the domain in y, and `sigma=1.0`. Set the function to be zero for $x > 1$m *and* at $x = 0$. Plot this with `plt.imshow` to check (and try the 3D plot as well if you like).

4. Using $r = 0.2$, calculate Δt. Create the wave at $t = \Delta t$ as well, and set both waves to be zero for $y = 0$ and $y = 3$m (N.B. these are boundary conditions, which you will have to enforce at *every* step). Using your 2D solver, propagate the wave forward in time by 100 steps and plot it using `plt.imshow` to check that it is moving. Be sure to enforce the boundary conditions.

5. (Depending on how you get on, this could be treated as part of the further work.) Now use the `plt.figure` approach from Matplotlib to plot snapshots of the wave with time. If you create an initial figure (I found that passing the optional parameter `figsize=(10,6)` was helpful) then you can add subplots at every M steps using a command like `ax = fig.add_subplot(r,c,i)` where there are `r` rows, `c` columns and `i` is the index of the plot (this format means you can go beyond 9 subplots). Then you can display the wave with `ax.imshow`

Further work

1. Now we will introduce a scatterer. We do this by setting part of the waves to zero: choose a square in the middle of the domain, using array slicing. I used `theta[mid-width:mid+width, mid-width:mid+width] = 0.0` for both θ_n and θ_{n-1} to enforce this *at each step*. Using the same initial wave as above, propagate and plot over appropriate time (until the initial wave reaches the right hand side – around 1,200 steps). You should see a circular wave emerging from the scatterer as the initial pulse passes. Experiment with sizes and shapes.

5 Complex numbers in python

Before we turn to the Schrödinger equation, we have to pause to introduce the treatment of complex numbers in Python. Fortunately, this is rather simple and in most cases maps directly onto real numbers. The only care that is needed is in specifying datatypes.

A number can be made imaginary by applying `j`, so we can write `2 + 1j` for a complex number. Arrays of complex numbers can be created with the usual commands (`np.zeros()`, `np.arange()` etc) but with an extra (optional) argument: `dtype=complex`. So we might write something like `a = np.zeros(5,dtype=complex)`. It is also perfectly possible to add together two arrays, such as `b = np.ones(5) + 1j*np.zeros(5)`.

For a complex number `z`, we can access its real and imaginary parts with `z.real` and `z.imag` (this is also true for arrays); there are also Numpy functions `np.real()` and `np.imag()` which do the same thing. The Numpy functions `np.absolute` and `np.angle` allow you convert to the form $z = re^{i\theta}$. Using `np.conj(z)` or `np.conjugate(z)` gives $z^\star$. It is also possible to use `z.conjugate()`[4]. The normal arithmetic operators (`+, -, *, /`) perform complex arithmetic correctly (they have been overloaded). We will not set exercises in this section: we will use complex numbers in the following sections.

[4] Note that the conjugate here is a *function* and so requires the parentheses, where the real and imaginary parts, `z.real` and `z.imag`, described above do not.

6 Time-dependent Schrödinger equation

The time-dependent Schrödinger equation is written as:

$$i\frac{\partial\psi(x,t)}{\partial t} = -\frac{1}{2}\frac{\partial^2\psi(x,t)}{\partial x^2} + V(x)\psi(x,t) \qquad (11)$$

where we have used atomic units ($\hbar = m_e = q_e = 1/4\pi\epsilon_0 = 1$) to simplify. If we apply the standard differencing rules, we would find as usual:

$$\frac{\partial\psi(x,t)}{\partial t} \simeq \frac{\psi_{i,n+1} - \psi_{i,n}}{\Delta t} \qquad (12)$$

$$\frac{\partial^2\psi(x,t)}{\partial x^2} \simeq \frac{\psi_{i+1,n} - 2\psi_{i,n} + \psi_{i-1,n}}{\Delta x^2} \qquad (13)$$

This is very similar to the heat equation (or the diffusion equation) that we saw in Session 5; however, the presence of i makes the equation unconditionally unstable. We saw that for the heat equation, with a small enough timestep, we could use an explicit propagator and achieve stability. The time-dependent Schrödinger equation needs the implicit approach that we discussed: we will use the Crank-Nicolson approach.

Recall that to do this, we assume that the time differential is a *centred* difference, based around step $n + \frac{1}{2}$; so we approximate the position differential at that step by the average of the differentials at n and $n+1$. If we do this, we find that we can write:

$$\begin{aligned} i\frac{\psi_{i,n+1}-\psi_{i,n}}{\Delta t} &\simeq \frac{1}{2}\left[-\frac{1}{2}\left(\frac{\psi_{i+1,n}-2\psi_{i,n}+\psi_{i-1,n}}{\Delta x^2}\right)-\frac{1}{2}\left(\frac{\psi_{i+1,n+1}-2\psi_{i,n+1}+\psi_{i-1,n+1}}{\Delta x^2}\right)\right] \\ &+ \frac{1}{2}\left(V_i\psi_{i,n}+V_i\psi_{i,n+1}\right) \qquad (14) \end{aligned}$$

$$\begin{aligned} 4\psi_{i,n+1} &- i\zeta\left[\psi_{i+1,n+1}-2\psi_{i,n+1}+\psi_{i-1,n+1}\right]+2i\Delta t V_i\psi_{i,n+1} = \\ &4\psi_{i,n}+i\zeta\left[\psi_{i+1,n}-2\psi_{i,n}+\psi_{i-1,n}\right]-2i\Delta t V_i\psi_{i,n} \quad (15) \\ -i\zeta\psi_{i+1,n+1} &+ 2(2+i\zeta)\psi_{i,n+1}-i\zeta\psi_{i-1,n+1}+2i\Delta t V_i\psi_{i,n+1} = \\ &i\zeta\psi_{i+1,n}+2(2-i\zeta)\psi_{i,n}+i\zeta\psi_{i-1,n}-2i\Delta t V_i\psi_{i,n} \quad (16) \end{aligned}$$

where $\zeta = \Delta t/\Delta x^2$. The last equation is designed to make it clear how we would assemble a matrix form: $\underline{\underline{\mathbf{M}}}\,\underline{\psi_{n+1}} = \underline{\underline{\mathbf{N}}}\,\underline{\psi_n} + \underline{\mathbf{b}}$. Notice that we need a little care because of the source term (the potential) which will need to be added to the diagonal of $\underline{\underline{\mathbf{M}}}$, but is, in genearl, a position-dependent function. This means that we cannot include it in the simple stencil that we used to build the matrix.

As with the heat equation, we have boundary points that are *not* included in the solution vector. The boundary condition vector will require us to use $2i\zeta\psi_0$ and $2i\zeta\psi_{n+1}$ as the first and last entries (if we have a solution domain running from $1 \rightarrow n$), with zeros in all other places. What boundary conditions can we use? Hard walls (perfectly reflecting boundaries) are equivalent to setting the wavefunction to zero at the boundaries and are the most common, and simple; absorbing boundary conditions can be constructed, but are quite complicated and the subject of active research in some areas.

So we can propagate the time-dependent Schrödinger equation by using the formula:

$$\underline{\psi_{n+1}} = \underline{\underline{\mathbf{M}^{-1}\mathbf{N}}}\,\underline{\psi_n} + \underline{\underline{\mathbf{M}^{-1}}}\mathbf{b} \qquad (17)$$

As we saw for the Crank-Nicolson method in Session 5, we can invert $\underline{\underline{\mathbf{M}}}$ once at the start, and calculate the appropriate products at each step. This method is stable, but the accuracy depends on the choice of ζ. The basic form of the functions to calculate both matrices is the same; an outline is given in Fig. 2, with the details left to be filled in.

Note that you can also use the command `np.diag` to place a 1D array along the diagonal of a matrix; this is another way to create the potential entries.

6.1 Exercises

In-class

1. For $-100 < x < 100$ with $\Delta x = 0.5$, create an array of position and then build the initial wavefunction $\psi(x,0) = e^{ikx}e^{-(x-x_0)^2/\sigma^2}$,

with $k = 1$, $x_0 = -75$ and $\sigma = 10$. Make a plot of the real and imaginary parts to check that this looks sensible. (Make sure that both ends of the array are set to zero to establish the initial boundary conditions.)

2. Write functions to create the matrices $\underline{\underline{\mathbf{M}}}$ and $\underline{\underline{\mathbf{N}}}$ following the code in Fig. 2 and Eq. (16).
3. For a potential function $V(x) = 0$, create an array of the values of the potential. Set $\Delta t = 0.1$ and calculate ζ and both matrices $\underline{\underline{\mathbf{M}}}$ and $\underline{\underline{\mathbf{N}}}$. Invert $\underline{\underline{\mathbf{M}}}$ using `np.linalg.inv` and calculate the product $\underline{\underline{\mathbf{M}}}^{-1}\underline{\underline{\mathbf{N}}}$.
4. Now propagate the wavefunction forward in time for 1000 steps using a for loop and an update step like `psi_next = np.dot(M_inv_N,psi_now)`. Either create a set of subplots as you go along, or store the results and plot them to ensure that you see the wave propagating.

```python
def calc_M(N,zeta,V,dt):
    """Create matrix for Crank-Nicolson solution of TDSE
    Inputs:
    N    size of matrix
    zeta parameter
    V    potential (array)
    dt   time step
    Outputs:
    (NxN) matrix"""
    output = np.zeros((N,N),dtype=complex)
    stencil = np.array([ a, b, a ]) # FILL IN a, b
    for i in range(1,N-1):
        output[i,i-1:i+2] += stencil
        output[i,i] += c # FILL in c
    output[0,0:2] += stencil[1:3]
    output[N-1,N-2:N] += stencil[0:2]
    output[0,0] += c # FILL in c
    output[N-1,N-1] -= c # FILL in c
    return output
```

Figure 2: Outline of code to create Crank-Nicolson matrices

Further work

1. Extend your solver from the in-class work to include a barrier with height 0.5 at $x = 0$; experiment with the width. Note that you will have to recalculate the matrices $\underline{\underline{\mathbf{M}}}$ and $\underline{\underline{\mathbf{M}}}^{-1}\underline{\underline{\mathbf{N}}}$ for *each* different barrier that you create.

7 *Time-independent Schrödinger equation*

The time-independent Schrödinger equation in one dimension is written as:

$$-\frac{1}{2}\frac{\partial^2\psi}{\partial x^2} + V(x)\psi(x) = E\psi(x) \qquad (18)$$

which is specified by boundary conditions (at two points in space) and contains an *unknown* quantity, E. Our task is then to find values of E that satisfy the boundary equations. As you have seen in your lectures on quantum mechanics, this is an eigenvalue equation[5]. We will use the shooting method that we first saw in Session 4 to search for values of E, and we will use two different approaches to integrate for the wavefunction.

[5] You should also recognise the idea of an eigenvalue equation from the mathematical work you have done on matrices; next year, you will see that quantum mechanics can be formulated in terms of matrices.

We will need to specify an initial value of ψ and $d\psi/dx$ and integrate outwards; one of these values will be arbitrary. If we know that there a point where the wavefunction goes to zero (e.g. at a hard wall) then we can choose an arbitrary value of $d\psi/dx$ at that point; once we have a wavefunction and energy that obey the boundaries we can remove the arbitrary constant by normalising the wavefunction. Similarly, if there is a point where the gradient is zero (e.g. for the symmetric states of a quantum harmonic oscillator) then we can choose an arbitrary value for ψ at that point.

If we use a centred, second-order difference equation, then we can rewrite Eq. (18) as:

$$\begin{aligned}
-\frac{1}{2}\frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{\Delta x^2} &= E\psi_i - V_i\psi_i \qquad (19)\\
\frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{\Delta x^2} &= 2(V_i - E)\psi_i \qquad (20)\\
\psi_{i+1} &= 2\psi_i - \psi_{i-1} + 2\Delta x(V_i - E)\psi_i \qquad (21)
\end{aligned}$$

This gives us a simple approach to integrating the wavefunction which is very similar to the wave equation above (but without the derivatives on the right-hand side). In this case, we would specify the values ψ_0 and ψ_1 rather than a value and its derivative. This is a relatively simple approach which is easy to implement, but you will need to be careful to test the size of the spatial grid, Δx.

A more accurate and robust approach is to follow the work we did in Session 4. We saw there that we can rewrite the second order equation as two coupled first order equations:

$$\begin{aligned}
\frac{\partial\psi}{\partial x} &= \phi \qquad (22)\\
\frac{\partial\phi}{\partial x} &= 2(V(x) - E)\psi(x) \qquad (23)
\end{aligned}$$

This very general approach is an important one to master. Now that we have done this, we can use very accurate methods such as RK4 (see Session 4) to solve the equation. To do this, we will need a function to return the right hand sides of the equation. We will use something like that shown in Fig. 3.

```
def TISE_coupled_first_order(y,x,E,V):
    """TISE split into two first-order ODEs

    Inputs:
    y  Two entry array containing psi and phi (dpsi/dx)
    x  Value of x
    E  Energy
    V  Potential function (given x returns V(x))

    Output:
    Two entry array of dpsi and dphi"""
    psi = y[0]
    phi = y[1]
    dpsi = phi
    dphi = 2.0*(V(x) - E)*psi
    return np.array([dpsi,dphi])
```

Figure 3: Function to evaluate the coupled first-order version of Schrödinger equation.

Then this can be passed to the RK4 solver and the wavefunction can be integrated. Note that as I have specified this function we pass in the energy and the potential; this would mean that we could not use `integrate.solve_ivp` from SciPy. We can of course write the function with `E` and `V(x)` specified explicitly, but that would mean that we couldn't use the shooting method.

What should we do about this? There are three options. First, we could use our own RK4 solver. Second, we could use the `integrate.odeint` function from SciPy which allows us to pass arguments to the function. Or, third, we could specify `E` and `V(x)` in the calling routine or code. Each of these has its own merits; as writing a simple RK4 solver is easy (and we already did this in Session 4) we will stick with this.

It is perfectly possible to go beyond one dimensional problems for the time-independent Schrödinger equation, but not using this approach. Typically either matrix methods or a variational minimisation are used: both of these require a basis set (which you will learn more about in the third year quantum mechanics course). The solution of the Schrödinger equation for atoms, molecules and materials forms a large research area spanning condensed matter physics, quantum chemistry and related disciplines.

7.1 Exercises

In-class

1. Write a simple integration function based on the second-order finite difference update in Eq. (21). Your function should take as parameters: ψ_0 and ψ_1; an array of the potential V_i; the energy, E; and the grid spacing and length Δx and N_x. It should return an array containing the wavefunction for all grid points.

2. We are going to solve for the eigenvalues and wavefunctions for an infinite square well. Create an x array from -5 to 5 using an appropriate number of points (you may need to test this later), along with an array of the potential, which you should set to zero. Create the starting values of the wavefunction, $\psi_0 = 0.0$ and $\psi_1 = a$, where a is an arbitrary number.

3. For $E = 0$ and $E = 0.1$ call your solver, store the output wavefunctions and plot against x to check that it works. (You should be able to deduce the form with $x = 0$ just by looking at Eq. (21).)

4. Now write a simple secant solver. You will need to use the ψ_{i-1} as $f(x)$ and E as x. This can be written quite simply by adapting your code from Session 2. Use the energy window specified above, and be sure to write out the number of iterations required. Plot the final wavefunction (you may have to calculate it again after the secant has finished).

Further work

1. Write a function to integrate the wavefunction using the coupled first-order differential equation approach in Fig. 3 and the Runge-Kutte fourth-order solver that you wrote in Session 4. Check that it gives you the same answer as the simple second-order finite difference approach. Then adapt the secant code you wrote to use this solver.

2. If you have time, consider ways to scan roughly over energy and identify different eigenstates. (You should think about nodes in the wavefunction.) Once a pair of energies bracketing an eigenstate has been identified, the secant method can be used to find it. You might like to plot the first few eigenstates for the square well, or add another potential into the well.

8 *Progress Review*

Once you have finished *all the material* associated with this session (both in-class and extra material), you should be able to:

- Solve the classical wave equation in one and two dimensions, understanding the relationship between grid spacing, time step and wave speed;
- Display the results of your solution in different ways;
- Solve the time-dependent Schrödinger equation using implicit methods;
- Find eigenvalues and wavefunctions for the time-independent Schrödinger equation using an appropriate integration scheme and a boundary value solver.