PHAS0030: Computational Physics
Session 4: Ordinary Differential Equations

David Bowler

In this session, we will consider methods to solve ordinary differential equations of different orders. We will see that this is not a straight-forward problem, with stability problems being very common. We will discuss methods that are stable.

1 Objectives

The objectives of this session are to:

- understand how we can create a numerical solution for a differential equation
- explore the stability of the simplest approach, Euler's method
- examine more stable methods, such as predictor-corrector and Runge-Kutta
- consider boundary conditions and how different methods are applicable depending on the boundary conditions

2 Review of Session 3

In the third session, we looked closely at the topic of discretization, covering:

- Grids that are commonly used in calculations
- The finite difference approach to differentiation on a grid
- Numerical integration in one dimension,
  including the trapezoidal rule and Simpson's rule

We will continue to use discrete space and time variables throughout the next three sessions, where we will consider the numerical solution of differential equations.

3 Introduction

Ordinary differential equations (ODEs) are equations for a function, often written as y, of one independent variable, x, and its derivatives (including polynomials in x). The most common, and certainly most easily solved, class is that of linear ODEs, where the unknown function and its derivatives only appear to first order. We can write a general linear ODE as:

$$y^{(n)}(x) = \sum_{i=0}^{n-1} a_i(x) y^{(i)}(x) + r(x)$$

where $y^{(n)}$ indicates the differential $d^n y/dx^n$, $a_i(x)$ is an arbitrary function of x and $r(x)$ is a source term (again, an arbitrary function of x). Examples of this kind of equation include: classical mechanics (particle motion), LRC circuits and harmonic oscillators. These are all second order linear equations, for instance:

$$\frac{d^2q}{dt^2} = \frac{V_0}{L}\cos(\omega t) - \frac{R}{L}\frac{dq}{dt} - \frac{q}{CL}$$

for the charge in an LRC circuit: here, $q \equiv y$ and $t \equiv x$, and the driving term $(V_0/L)\cos\omega t \equiv r(x)$, and the coefficients a_i are independent of x.

The equation of motion for the angle made by a pendulum relative to the vertical is linear for small angles, but becomes non-linear for large angles (this is defined by the point at which the approximation $\sin(x) = x$ breaks down). A simple model of radioactive decay is a first-order linear ODE: $dN/dt = -\alpha N$, where N is the number of particles in the sample at a given time and α is related to the half-life.

We will consider *partial* differential equations (PDEs) in the next two sessions; these are differential equations that feature more than one independent variable. Examples include the wave equation, Maxwell's equations for electromagnetism, heat flow and Schrödinger's equation (in both time-dependent and time-independent forms).

The most common form of ODE is one where *time* is the independent variable; while we use x commonly in these notes, it may well often be substituted by t.

3.1 Specifying conditions

In order to solve a differential equation, we must specify some conditions on the problem. The number of conditions depends on the order of the equation, so a first order equation will require one condition, and a second order equation will require two. For instance, the simple model of radioactive decay given above requires the number of atoms at $t = 0$. The LRC circuit might require an initial charge, q, and current dq/dt. Once we move beyond first order equations, there are *two* ways to specify the conditions:

- Initial value problems, where conditions are specified at one point in space, x_0, or time, t_0; for a second order ODE, we would specify $y(x_0)$ and $dy/dx(x_0)$.
- Boundary value problems, where conditions are specified for only one variable, but at extremes; for a second order ODE, we could specify the value of y at two points, x_0 and x_1.

It is of course possible to solve in more than one dimension (this is particularly common for classical mechanics). Note that in this case we still have the same *order* of differential equation (second order for problems involving non-zero acceleration, say) but the equation

is now a *vector* equation. The initial conditions that must be specified are now vector quantities.

3.2 Discretization again

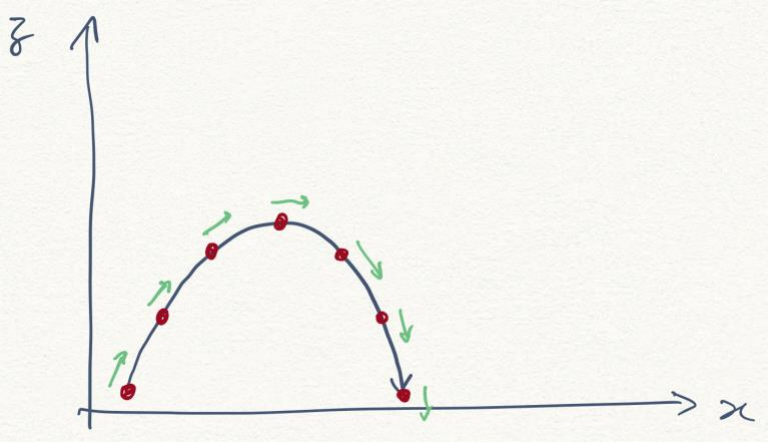
The approach that we will take to solve differential equations will be based on the finite difference work that we did in the last session. We will substitute the finite difference form of the differential, and then perform numerical integration. If we are working with $y(t)$ then you can think of the approach as stepping forward in time by a step dt; at each point in time, we calculate the appropriate differential and use that to update y.

It is useful to consider a Taylor expansion:

$$y(x + \Delta x) = y(x) + \Delta x \frac{dy}{dx} + \frac{(\Delta x)^2}{2!} \frac{d^2y}{dx^2} + \cdots$$

This will enable us to think about errors in our approach.

The overall aim of a numerical solution is to calculate $y(x)$ for a series of points $x_0, x_1, x_2, \ldots$. The way that we do this is via step-wise numerical integration: from $y(x_0)$ and we can evaluate dy/dx at x_0, and therefore find $y(x_1)$; we then evaluate dy/dx at x_1, and find $y(x_2)$; and so on. This is illustrated very roughly in the sketch below.



A sketch of a numerical integration for the trajectory of a ball, indicating the velocity at each point in the process. The gaps between points correspond to the timestep.

4 Solving first-order ODEs: Euler's method

Euler's method is a very simple approach to solving ODEs, so we will start with it. But it is often a very poor choice, prone to instabilities and should not generally be used for practical calculations; after we introduce the basic concept, we will turn to more reliable and accurate approaches. If we return to the simple definition of a (finite) differential, we find:

$$\frac{dy}{dx} \simeq \frac{y(x+\Delta x)-y(x)}{\Delta x}$$
$$\Rightarrow y(x+\Delta x) = y(x)+\Delta x\frac{dy}{dx}$$

This is equivalent to the first-order Taylor expansion, and indicates one way in which we could solve a first order equation: starting from some initial condition, $y(x_0)$, we could integrate with steps of size Δx to get values of y at successive values of x.

Why might this approximation be a poor choice? Our experience with the forward difference formula from last session gives us a hint—that formula was only accurate to *first order* in the difference. To examine integration, we need to find the order of the error. At each step, we know from the Taylor expansion that the leading term in the error in $y(x+\Delta x)$ will be:

$$\frac{(\Delta x)^2}{2!}\frac{d^2y}{dx^2}+\mathcal{O}(\Delta x^3)$$

(where $\mathcal{O}$ indicates the order of the term). However, this is just the error in a *single* step. The global (or cumulative) error when we reach x will be the sum of the errors for the number of steps required to reach x from x_0: $N=(x-x_0)/\Delta x$, so that the global error (N times the error in each step) will be of order Δx.

We could have guessed that the method would be a bad approach from our analysis of finite differences. Last week, we saw that the centred difference approach was much more accurate (second order in step size), and in Session 7 we will see that we can use the same ideas to make a stable equivalent for differential equations (these are known as Verlet approaches). In this session, we will look at alternative routes to improve the integration.

4.1 Stability analysis

In the exercises, we will see that the Euler method is unstable, particularly for step sizes that pass a certain threshold. Why might that be? Let's consider a very simple differential equation, and analyse it[1].

$$\frac{dy}{dx}=-ky$$

This clearly has the analytic solution $y=Ae^{-kx}$, with A found from the initial condition $y(x=0)$. (Notice that this is a very common equation in physics and elsewhere, for instance giving a simple model of radioactive decay.)

What would the Euler method give us for an update step?

$$y_{n+1}=y_n+\Delta x(-ky_n)=y_n(1-k\Delta x)$$

[1] If you find yourself using a computer to solve this equation, then you need to spend some time working on basic mathematics!

So when we apply this to n steps, we find:

$$y_n = y_0(1 - k\Delta x)^n$$

This very simple example immediately indicates some of the issues that will occur if we choose Δx unwisely. If $k\Delta x > 1$, then while at each step the solution will decay, it will *alternate* in sign (clearly wrong given the analytic solution). If $k\Delta x > 2$, then the overall magnitude will *increase* with x. Clearly the choice of step size depends on key parameters in the problem, and can have significant effects on the solution. We will return to the question of stability in later sections.

4.2 Exercises

4.2.1 In class

1. Using a simple loop, solve the differential equation $dy/dx = -ky$ using Euler's method. For flexibility, set up Δx as a variable, calculate the possible values of x and store the results in an array. You will also need to store y in an array. Use $k = 1.2$ and solve for $0 \leq x \leq 10$, with $y(0) = 1.0$. (You might find the basic set-up below useful.)

```
# Specify step size and simulation length
dx = 0.5
total_x = 20
N = int(total_x/dx)
x = np.linspace(0,total_x,N+1)
# Initial condition
y0 = 1.0
k = 1.2
y = np.zeros(N+1)
y0 = 1.0
y[0] = y0
```

2. Plot the approximate and exact solutions, and explore how the step size affects agreement. Look at the effect of increasing Δx.

4.2.2 Further work

1. Create a set of four sub-plots for values of Δx that give divergence, oscillation, stability but poor agreement, and good agreement respectively. Use the `fig = plt.figure`, `ax = fig.add_subplot`, `ax.plot` protocol that we discussed in Session 1.

2. Solve the equation $dy/dx = xy^2$ with $y(0) = -1$ using Euler's method. What values of Δx give stable solutions? How well do these match the exact solution[2]?

[2] You should be able to work this out yourself; a useful tip is to put the constant of integration with x.

4.3 Beyond first order and one dimension

There are many situations where we will have a vector ODE to solve; in terms of classical dynamics we could write:

$$\frac{d\mathbf{r}}{dt} = \mathbf{f}(\mathbf{r}, t)$$

where $\mathbf{r}$ is the position vector. This can be solved using Euler's method in exactly the same was as we did for the scalar, by resolving the components of $\mathbf{r}$ and $\mathbf{f}$. It will give *coupled* first order ODEs (one for the x-component, one for the y-component, etc). To implement the equation, we would need to work on an N-component array (for N dimensions) but otherwise there would be no change.

But we can also use this form of equation to solve second order equations, separating them into *two coupled* first order equations. Consider simple dynamics, where we have the basic equation:

$$m\frac{d^2x}{dt^2} = F.$$

If we work in terms of position x and velocity $v = dx/dt$, then we have two *coupled* first-order equations:

$$\begin{aligned} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= \frac{F}{m} \end{aligned}$$

To make this clearer, consider simple harmonic motion, where $F = -kx$. We could write this as a matrix equation (using $\dot{x} = dx/dt$):

$$\begin{aligned} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= -\frac{k}{m}x \end{aligned}$$

Notice that we will now need *two* initial conditions: one for x and one for v.

To implement this, we start by defining a numpy array which contains the two variables, say `y = np.array([x,v])` The Python function for the right-hand side of the equation, $\mathbf{f}(\mathbf{r}, t)$, would need to return a two-component array containing the derivatives of x and v: `[v,-k*x/m]` in the case of harmonic motion. This approach is actually a powerful general way to solve second order differential equations on a computer. [3]

[3] We can also extend it so that x and v are vectors: each of the arrays would become larger (for three dimensions, they would each be of length six) but no other change would occur.

If you write your Euler method carefully, it should extend to this type of problem trivially. For example, consider the following code:

```
def rhs_sho_function(x,v):
    """Implement RHS of SHO equation for x and v
    Defines k and m within function"""
    k = 1.0
    m = 1.0
    # Note that we make no assumption about what v and x
    # are: they can be scalars or arrays
    dx = v
    dv = -k*x/m
    return dx, dv

# Sample Euler update step
# Assume x and v are already defined
dx, dv = rhs_sho_function(x,v)
x_next = x + dt*dx
v_next = v + dt*dv
```

The alternative, which is more flexible and efficient computationally, is to pass a single array to the Euler update scheme, and have the update function, $f(y,t)$, split up and increment the individual components:

```
def rhs_sho_function(y,t):
    """Implement RHS of SHO equation for y (array
    containing x and v). Note that t is unused here but
    is passed for compatibility with general
    solvers.  Defines k and m within function"""
    k = 1.0
    m = 1.0
    # Separate out for clarity
    x = y[0]
    v = y[1]
    # Calculate update
    dx = v
    dv = -k*x/m
    return [dx, dv]

# Sample Euler update step
# Assume x and v are already defined
y = [x, v]
dy = rhs_sho_function(y,t)
y_next = y + dt*dy
```

In this case, if x and v are N-dimensional arrays themselves then y will be a (2,N) dimensional array, and we can again work with vector quantities.

Finally, we can consider the stability of coupled ODEs, extending our analysis in Section 4.1. We will take an equation that can be written as:

$$\frac{d\mathbf{y}}{dt} = \underline{\underline{\mathbf{A}}}\,\mathbf{y}$$

If we integrate numerically, we will be applying the matrix $\underline{\underline{\mathbf{A}}}$ repeatedly to our solution, which will accumulate powers of the matrix: $\mathbf{y}_n = (1 + \Delta t\, \underline{\underline{A}})^n \mathbf{y}_0$. If there is a large ratio between the largest and smallest eigenvalues of $\underline{\underline{A}}$, it can be shown that the ODE will be challenging to solve numerically: this ratio is known as the *stiffness*[4]. A stable solution for this type of equation can require *very* small steps, or a different kind of solution (known as implicit methods, which we will consider in the next session).

4.4 Exercises

4.4.1 In class

1. Write a function to implement the Euler method in general, that matches the following specification (you should complete the docstring!):

```
def euler_solver(fun,y0,dt,N):
    """Solve dy/dt = fun(y,t) using Euler's method.
    Inputs ...
    Returns: array of length N with values of y
    """
```

   You will need to create the return array at the start of the routine. You can allow `y` to be of arbitrary length (in which case you will need to use the python function to find the right size) or assume a simple two-component problem as seen for the SHO.

2. Apply this function to the simple harmonic oscillator above, and plot your result (you should know what it looks like!). Check the effect of step size.

4.4.2 Further work

1. Add a damping term[5] to the SHO solver you have created abovein question 4 above and explore the effect of the damping coefficient, c. Plot your solutions using sub-plots.

2. You could explore adding a driving term on the RHS of the SHO equation if you have time.

5 Beyond Euler

There are many methods that improve on the stability and accuracy of Euler. We can gain some insight into their development by considering the question: why is Euler so poor?

[4] We discussed the idea of a characteristic matrix whose eigenvalues determine the behaviour of a system already, when considering conjugate gradients.

[5] Now you have the equation $md^2x/dt^2 + cdx/dt + kx = 0$.

Part of the answer lies in the use of the gradient at the start of the interval, which means that when we step forwards, we *extrapolate*[6]. Various of the common approaches to ODEs attempt to improve on this poor gradient estimate.

The simplest thing that we have already seen is simply to reduce the timestep in the Euler method. We can see that this is equivalent to taking a larger step but using the average of the gradients at the start and middle of the step:

$$\begin{aligned} y(x+\Delta x/2) &= y(x)+\frac{\Delta x}{2}f(x,y(x)) \\ y(x+\Delta x) &= y(x+\Delta x/2)+\frac{\Delta x}{2}f(x+\Delta x/2,y(x+\Delta x/2)) \\ &= y(x)+\frac{\Delta x}{2}(f(x,y(x))+f(x+\Delta x/2,y(x+\Delta x/2))) \end{aligned}$$

The predictor-corrector method takes a step (the prediction) and calculates the gradient at the end of the step, and then averages this with the gradient at the *start* (the correction):

$$\begin{aligned} y_{Pred}(x+\Delta x) &= y(x)+\Delta x f(x,y(x)) \\ y_{Corr}(x+\Delta x) &= y(x)+\frac{\Delta x}{2}(f(x,y(x))+f(x+\Delta x,y_{Pred}(x+\Delta x))) \end{aligned}$$

There are more sophisticated versions of this approach (often associated with the names Adams, Moulton and Bashforth) which can be extremely accurate and stable.

A different approach, known as the mid-point method, uses the gradient half-way along the step:

$$\begin{aligned} k_1 &= \Delta x f(x,y(x)) \\ y(x+\Delta x) &= y(x)+\Delta x f\left(x+\frac{\Delta x}{2},y(x)+\frac{\Delta x}{2}k_1\right) \end{aligned}$$

Notice that this differs from the updated Euler method above, in that we calculate and use the gradient at the mid-point rather averaging the initial and mid-point gradients. It can be shown that this formula is accurate to second-order in Δx.

The mid-point method is an example of a class of methods called Runge-Kutta methods. These use intermediate gradients to improve the accuracy in sub-divisions, and are named by the order of accuracy that they achieve. The most commonly used is RK4: it is fourth order in accuracy (Δx^4) *but* it requires *four* function evaluations. The update scheme can be written as:

6 Extrapolation is almost always dangerous; interpolation is generally much safer

$$\begin{aligned} y(x+\Delta x) &= y(x) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ k_1 &= \Delta x f(x, y(x)) \\ k_2 &= \Delta x f(x + \Delta x/2, y(x) + k_1/2) \\ k_3 &= \Delta x f(x + \Delta x/2, y(x) + k_2/2) \\ k_4 &= \Delta x f(x + \Delta x, y(x) + k_3) \end{aligned}$$

Note that the first quantity, k_1, is the same as the Euler update, while the next two are mid-point evaluations, and the final is an end-point. By combining these, we can eliminate errors below fourth order.

Note that all these methods allow us to solve most ODEs, with a reasonably large step size (though this must always be tested). There are certain classes of equation which require alternative methods, but we will not consider them here.

5.1 Exercises

5.1.1 In class

1. Write a function to implement a fourth-order RK solver, using Eq. ([eq:12]) above, using your Euler function as a basis.

2. Apply it to a pendulum, for the moment working simply with the linear solution:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\theta$$

3. where $g = 9.8m/s^2$, the acceleration due to gravity, and $L = 1m$, the length of the pendulum. You will need to:

    a. break this into two coupled equations;

    b. write an appropriate function for the right-hand side;

    c. write a loop calling your RK4 solver to propagate the motion of the pendulum

    How large can the step size be while still giving a stable solution? (If you have time, try the same solution with an Euler solver, and compare the necessary step sizes). Be sure to make the initial angle (**in radians!**) small enough that the approximation is good.

5.1.2 Further work

1. Write either a predictor-corrector function or a mid-point function

2. Now model the non-linear pendulum, comparing the RK4 and the function that you coded in the first part:

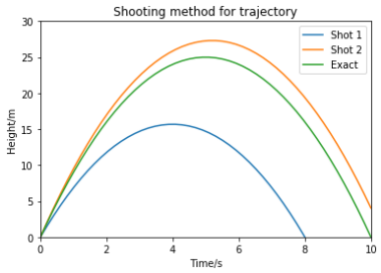
$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin(\theta)$$

3. You will need to write a new function for the right-hand side. Start with an angle of $\pi/2$, and compare the two for the same step size.

4. You should now extend the model further: test an initial angle of just less than π (to observe strongly non-linear effects); add a damping term ($-cd\theta/dt$) and test the effect of c.

6 Boundary Value Problems

Instead of specifying all conditions at the *start* of the simulation, as is done for initial value problems, we can also specify conditions at the *boundaries* of the problem. A simple example is the motion of a projectile, which as a second order ODE requires two pieces of information: we can either specify the position and the velocity at $t = 0$; or we can specify the position at two different times[7]. This gives us a condition to meet *at the end of the simulation*, but does not give enough information to start the solver; we have to guess an initial condition on the velocity, solve to find the position at the end of the simulation, and then update the initial velocity until the boundary condition is met. On a computer, we would use one of the root finding approaches (bisection, secant) we discussed in Session 2.



An illustration of how the shooting method works. The initial guess (blue line) falls short, while the second guess (orange line) over-shoots. The root finder is able to find the correct initial condition to match the specified boundary conditions with the exact trajectory (green line).

1. The shooting method guesses initial values for the gradient as well as the position at t_0 (or x_0), and then uses a root finder to adjust the initial gradient until they match the specified boundary condition at t_1 or x_1 The method is sketched in the figure above.

Let's use the example of the projectile which is specified to have height 0 after time t_1. To solve this using a simple root-finding approach we would need to find solutions for the height of the projectile based on two different initial velocities, such that the final heights at t_1 bracketed 0. We would then use (say) the secant method to search for the value of the initial velocity which gave the final height at t_1 equal to zero. In terms of the secant method, which usually seeks roots of a function $f(x)$, the independent variable x would be the initial velocity, while the function whose roots we are seeking would be the final height.

[7] We might specify that the height is zero at $t = 0$ and some later time, t_1

There is another class of solutions for boundary value problems that we will meet in the next session, called relaxation methods (which include the Gauss-Seidel method that we mentioned in Session 2).

6.1 Exercises

6.1.1 In class

1. Write a simple piece of code (either as a function or just a loop) to find the initial vertical velocity at which a ball moving vertically under the influence of gravity needs to be launched from the ground at $t = 0s$ so that it reaches the ground again at $t = 10s$. You should use the RK4 solver that you have written, and implement a basic bisection or secant solver, as described in the text above. Plot the trajectory versus time.

2. If you have time, try storing and plotting all the trajectories that the solver works through.

6.1.2 Further work

1. Try extending the solver so that you are now working in two spatial dimensions (x and z, say). You will need to be careful about how you specify your equations—you will need position and velocity for both dimensions (four variables in total). You will need to specify both x and z at t_0 and t_1.

2. If you want to extend this model, you could add an air-resistance term (proportial to the velocity) in one or two dimensions.

7 SciPy Functions

There are of course implementations of ODE solvers in SciPy, which generally assume that we have $y(t)$ rather than $y(x)$ as written above[8]. These are all part of the `integrate` module in SciPy, so you will need to import it as usual: .

For initial value problems, there is a general solver which implements various methods:

```
integrate.solve_ivp(fun, t_span, y0, t_eval = array)
```

where `fun` is an implementation of $f(t, y)$, `t_span` is a tuple[9] of two times, the start and end, and `y0` gives the initial value. The optional argument `t_eval` allows the user to provide an array of times at which to return the solution (otherwise the solver chooses the times). The solver returns a single object; if you call with routine with `a = integrate.solve_ivp`), you can then extract two arrays, `a.t` and `a.y`, which give values $y(t)$ at times t. Remember

[8] These are of course completely equivalent.

[9] In Python, a tuple is a set of numbers in round brackets, e.g. `(0,10)`.

that if you pass a two-dimensional array for `y`, then it will return a two-dimensional array. If you want to specify the points at which the result should be returned, you can pass an optional parameter which is an array of times, which should of course lie between the start and end times, `t_eval=your_array`. You can select the method used to solve the problem with the optional parameter `method`; for a fourth-order Runge-Kutta solver, you would pass the parameter `method='RK45'`.

An older function which is still useful, and provides a link to a FORTRAN library, is:

```
integrate.odeint(func, y0, t, args=())
```

where `func` is an implementation of $f(y,t)$ which can take extra arguments (optionally specified by the parameter `args=()` as we have seen before), `y0` is the initial value, and `t` is an array of times at which to solve for y; the first value should correspond to `y0`. The function returns an array `y` with the same number of entries as `t`. **Be careful:** the order of y and t differs between these two solvers!

There is also a boundary-value problem solver, `solve_bvp`, but its use is somewhat complex (you have to create a function to evaluate the error in the boundary conditions, amongst other things) so we will not consider it here.

7.1 Exercises

7.1.1 In class

1. Use `solve_ivp` and `odeint` to solve for the linear pendulum model from section 5.

7.1.2 Further work

1. Write a non-linear pendulum function that takes parameters (g, `L` and a damping term) and pass it to `odeint` to make sure that you understand how optional arguments are passed.

8 Assignment

This week, you will investigate the energy and eigenstates for the particle in a box, both by solving the Schrödinger equation as a boundary value problem (BVP) and by implementing the known solution on a grid (using finite differences and integration). For the BVP, you will adjust the energy of the solution until the wavefunction found goes to zero at the edge of the box (the necessary boundary value).

We will work in atomic units, so that $m_e = \hbar = 1$, distances are measured in Bohr radii (1 Bohr radius $\simeq 0.529$Å) and energies are in Hartrees. Consider an electron in a box with length 1 Bohr radius. The Schrödinger equation for this problem is:

$$-\frac{1}{2}\frac{d^2}{dx^2}\psi(x) = E\psi(x)$$

We can solve for $\psi(x)$ starting at $x = 0$ and integrating to $x = 1$ Bohr radius for *any* value of E. However, only values of E which correspond to the eigenstates of the system will match the necessary boundary condition of $\psi(x = 1) = 0$.

1. By writing $\phi = d\psi/dx$, separate the equation above into two linked first order equations, as we did in class (in practice you just need to work out $d\phi/dx$). Now implement a function to calculate the right-hand sides of these equations, taking a two-component array and the energy as input, and returning the first order differentials as a two-component array (follow the template).

2. Now write a RK4 routine (you *may* copy the routine from the in-class exercises if you want, but be sure that you understand it), adapting it so that it accepts an extra argument (the energy) which it passes to the function it calls. (Again, see the template.)

3. The initial conditions are $\psi = 0$ and $\phi = a$ where a is actually *any* arbitrary value that is not zero (I chose $a = 1$). It is arbitrary because we require the *final* wavefunction to be normalised (so we work out the correct prefactor for ψ *after* solving the equation; until that time, the initial value of ϕ fixes the prefactor for ψ). Set the spacing in x, $dx = 0.01$, the energy to 1.0, and create an initial input `psi0 = np.array((0.0, 1.0))`, and solve for the resulting wavefunction. Make a plot (it should *not* go to zero at $x = 1$ Bohr).

4. Implement a simple bisection procedure (you do not need to use a function) to solve for the correct value of E for the ground state. Set the initial brackets to be $E_{low} = 1$, $E_{high} = 11$, and run your procedure to adjust the value of E until it finds $\psi(1) = 0$. (Using the terminology of Session 2, a is E_{low}, b is E_{high} and $f(x)$ is $\psi(1)$ with $x \equiv E$.) Choose an appropriate tolerance for your search, and count the number of iterations.

5. Finally, normalise the output wavefunction. We require:

$$\int_0^1 |\psi(x)|^2 dx = 1$$

6. Using `integrate.simps` (remember `from scipy import integrate`) find the integral of ψ^2, A, and normalise ψ by $1/\sqrt{A}$. Make a plot of the final wavefunction.

7. Now you will compare the wavefunction found to the known analytic solution. Using the same spacing in x, create an array for x running from 0 to 1 *inclusive*. Create an array for the analytic solution $\psi_A(x) = B\sin(kx)$, where $k = \pi$ and you should calculate B to normalise ψ_A (set $B = 1$ and use the same procedure as before). Plot the difference between the two solutions, and comment *briefly* on the result and its relation to your tolerance.

If you want to practise finite differences a little more (a good idea, but *not needed* for the assignment) you could also evaluate the energy for the analytic solution, using:

$$E_A = \int_0^1 -\frac{1}{2}\psi(x)\frac{d^2}{dx^2}\psi(x)dx$$

where you will need to form a second derivative using finite differences and then calculate the energy using Simpson's rule. You could compare this to both the value found from bisection and the analytic result ($\pi^2/2$ for the ground state). You might also like to explore other states beyond the ground state by testing different starting energies.

9 Progress Review

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

- write a simple Euler solver for first-order differential equations
- understand how mid-point and Runge-Kutta solvers work, and write a function to implement them given the formulae
- use SciPy solvers for ODEs.

You should also check that you understand all the concepts and skills practised in Sessions 1–3.