# *PHAS0030: Further Practical Mathematics & Computing*
# *Session 4: Ordinary Differential Equations*

*David Bowler*

*January 25, 2019*

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.

*Contents*

## 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:

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

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. A pendulum 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.

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 substitute by $t$.

### 3.1 Specifying conditions

In order to solve a differential equation, we must specify some conditions on the problem. There are two ways to specify the necessary conditions for solving an ODE; notice that we need M pieces of information for an Mth order problem (e.g. for particle motion, we need two conditions). It is possible to solve N dimensional problems[1], of course, in which case the conditions would have to be specified in N dimensions.

[1] Be careful to distinguish the *dimension* of the problem—often 2D or 3D—and the *order* which specifies the highest differential.

The two kinds of problems that are found are:

- Initial value problems, where conditions are specified at one point or time ($x_0$); for a second order ODE, we would specify $y(x_0)$ and $y^{(1)}(x_0)$.
- Boundary value problems, where conditions are specified for only one variable, but at extremes; for a second order ODE, we would specify $y$ at two points, $x_0$ and $x_1$.

### 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. Here, we'll consider only a first order equation for simplicity (approaches for higher orders will follow). Using a Taylor expansion, we can write:

$$y(x+h) = y(x) + h\frac{dy}{dx} + \frac{h^2}{2!}\frac{d^2y}{dx^2} + \dots \tag{2}$$

where we will consider $h = \Delta x$ and discard derivatives beyond first order because we have a first-order ODE. The right-hand side of the equation, which sets $f(x, y)$, will generally be specified analytically by the physics of the situation.

# 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:

$$\frac{dy}{dx} \simeq \frac{y(x+\Delta x)-y(x)}{\Delta x} \tag{3}$$

$$\Rightarrow y(x+\Delta x) = y(x)+\Delta x\frac{dy}{dx} \tag{4}$$

As we saw above, 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 $\Delta x$.

Why might this 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, Eq. (2)) that the leading term in the error will be:

$$\frac{1}{2}\Delta x^2\frac{d^2y}{dx^2}+\mathcal{O}(\Delta x^3) \tag{5}$$

(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 $\Delta x$.

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[2].

$$\frac{dy}{dx} = -ky \tag{6}$$

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

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

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?

$$y_{n+1} = y_n + \Delta x(-k y_n) = y_n\,(1 - k\Delta x) \tag{7}$$

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

$$y_n = y_0\,(1 - k\Delta x)^n \tag{8}$$

This very simple example immediately indicates some of the issues that will occur if we choose $\Delta x$ 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 Beyond first order and one dimension*

In general, we can imagine having a vector ODE to solve:

$$\frac{d\mathbf{r}}{dt} = \mathbf{f}(\mathbf{r}, t) \tag{9}$$

where the vectors can be any quantities that obey equations like this (you might think about standard kinematics with zero acceleration, for example). 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.

But we can also use this form of equation to solve second order equations, through a useful trick. Consider simple dynamics, where we have the basic equation:

$$m\frac{d^2x}{dt^2} = F. \tag{10}$$

If we work in terms of position and velocity, then we have two *coupled* first-order equations:

$$\frac{dx}{dt} = v \tag{11}$$

$$\frac{dv}{dt} = \frac{F}{m} \tag{12}$$

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

$$\begin{pmatrix} \dot{v} \\ \dot{x} \end{pmatrix} = \begin{pmatrix} 0 & -k/m \\ 1 & 0 \end{pmatrix} \begin{pmatrix} v \\ x \end{pmatrix} \tag{13}$$

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

To implement this, we could start by defining a numpy array `y = [x,v]` and the function for $f(y,t)$ would need to return the array

`[v,-k*x/m]`. This is a general approach to solving second order differential equations on a computer. We can of course 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 an Euler function carefully, it should extend to this type of problem trivially. For example:

```
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, for example:

```
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:

$$\frac{d\mathbf{y}}{dt} = \underline{\underline{\mathbf{A}}}\mathbf{y} \tag{14}$$

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. If it has a large ratio between its largest and smallest eigenvalues, it can be shown that the ODE will be challenging to solve numerically: this ratio is known as the *stiffness*[3]. 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).

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

## 4.3 *Exercises*

*In class*

1. Using a simple **for** loop, solve the differential equation $dy/dx = -ky$ using Euler's method. For flexibility, set up $\Delta x$ 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 $\Delta x$.

3. 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 `len()` to find the right size) or assume a simple two-component problem as seen for the SHO.

4. Apply this function to the simple harmonic oscillator in Eq. (11) above, and plot your result (you should know what it looks like!). Check the effect of step size.

*Further work*

1. Create a set of four sub-plots for values of $\Delta x$ 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 $\Delta x$ give stable solutions? How well do these match the exact solution[4]?
3. 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.
4. You could explore adding a driving term on the RHS of the SHO equation if you have time.

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

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

## 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? 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.

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

The simplest thing that we have already seen is simply to reduce the timestep in the Euler method. But this still uses the gradient at the start of the interval; we can improve a little by using the *average* gradient from the start and middle of an interval:

$$y(x + \Delta x/2) = y(x) + \frac{\Delta x}{2} f(x, y(x)) \quad (15)$$

$$y(x + \Delta x) = y(x + \Delta x/2) + \frac{\Delta x}{2} f(x + \Delta x/2, y(x + \Delta x/2)) \quad (16)$$

$$= y(x) + \frac{\Delta x}{2} \left( f(x, y(x)) + f(x + \Delta x/2, y(x + \Delta x/2)) \right)$$

This method updates $y$ at $x + \Delta x$ by using an intermediate calculation.

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):

$$\begin{aligned} y_{Pred}(x+\Delta x) &= y(x) + \Delta x f(x, y(x)) && (17)\\ y_{Corr}(x+\Delta x) &= y(x) + \frac{\Delta x}{2}\left(f(x, y(x)) + f(x+\Delta x, y_{Pred}(x+\Delta x))\right) \end{aligned}$$

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:

$$\begin{aligned} k_1 &= \Delta x f(x, y(x)) && (18)\\ y(x+\Delta x) &= y(x) + \Delta x f\left(x + \frac{\Delta x}{2}, y(x) + \frac{\Delta x}{2} k_1\right) && (19) \end{aligned}$$

Notice that this differs from the updated Euler method in Eq. (15) 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 $\Delta x$.

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 ($\Delta x^4$) *but* it requires *four* function evaluations. The update scheme can be written as:

$$\begin{aligned} y(x+\Delta x) &= y(x) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) && (20)\\ k_1 &= \Delta x f(x, y(x)) && (21)\\ k_2 &= \Delta x f(x + \Delta x/2, y(x) + k_1/2) && (22)\\ k_3 &= \Delta x f(x + \Delta x/2, y(x) + k_2/2) && (23)\\ k_4 &= \Delta x f(x + \Delta x, y(x) + k_3) && (24)\\ & && (25) \end{aligned}$$

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*

*In class*

1. Write a function to implement a fourth-order RK solver, using Eq. (20) above, using your Euler function as a basis.

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

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\theta \tag{26}$$

where $g = 9.8m/s^2$, the acceleration due to gravity, and $L = 1m$, the length of the pendulum. You will need to break this into two coupled equations, and write an appropriate function for the right-hand side. 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 step sizes). Be sure to make the initial angle (**in radians!**) small enough that the approximation is good.

*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:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin(\theta) \tag{27}$$

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.

3. You should now extend the model further: test an initial angle of just less than $\pi$ (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 find some way to guess an initial condition, and then update this until the boundary conditions are met.

The shooting method guesses initial values for the gradients as well as the positions 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 Fig. 1.

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 two initial velocities, such that the

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

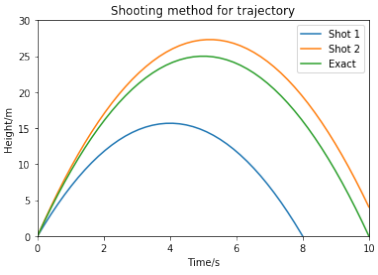


Figure 1: An illustration of how the shooting method works. The initial guess (blue line) falls short, while the second guess (orange line) over-shoots. The root finder is able to find the correct initial condition to match the specified boundary conditions with the exact trajectory (green line).

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.

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

### 6.1 *Exercises*

*In class*

1. Write a simple piece of code (either as a function or just a `for` loop) to find the initial velocity at which a ball 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 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.

*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: `from scipy import integrate`.

[8] These are of course completely equivalent.

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

```
integrate.solve_ivp(fun, t_span, y0)
```

where `fun` is an implementation of $f(t, y)$, `t_span` is a tuple of two times (start and end) and `y0` gives the initial value. The solver returns a single object (say `a = integrate.solve_ivp`) from which we can extract two arrays, `a.t` and `a.y`, which give values $y(t)$ at

times selected by the solver itself. Remember 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`. (It is possible to call a fourth-order Runge-Kutta solver explicitly, but it is better to specify it by passing the parameter `method='RK45'` to `solve_ivp`.)

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*

*In class*

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

*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 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.