Chapter 1

Ordinary Differential Equations (ODE's)

1.1 Ordinary Differential Equations

Differential equations are the principle mathematical tool for mathematical modelling of complex systems. An *ordinary* differential equation (ODE) is an equation involving one or more derivatives of a function of a single variable. The word "ordinary" serves to distinguish it from a *partial* differential equation (PDE) which involves one or more partial derivatives of a function of several variables. Although this course is ostensibly about solving PDEs, it makes a lot of sense to start by looking at ODEs.

Let $u : \mathbb{R} \to \mathbb{R}$ be a real–valued function of a single real variable, t, and $F : \mathbb{R}^{m+1} \to \mathbb{R}$ be a real–valued function of a vector formed from t, $u(t)$ and its first $m-1$ derivatives. The equation

$$\frac{d^m u}{dt^m}(t) = F\left(t, u(t), \frac{du}{dt}(t), \ldots, \frac{d^{m-1}u}{dt^{m-1}}(t)\right) \tag{1.1}$$

is an ordinary differential equation of order m.

The general solution of an m^{th} order ODE involves m constants of integration. Here are some simple examples which you already know:

Example 1: **Exponential growth equation**

$$\frac{du}{dt}(t) = \lambda u(t) \qquad \lambda > 0 \tag{1.2}$$

has general solution

$$u(t) = c_1 \exp(\lambda t).$$

Example 2: **Exponential decay equation**

$$\frac{du}{dt}(t) = -\lambda u(t) \qquad \lambda > 0 \tag{1.3}$$

has general solution

$$u(t) = c_1 \exp(-\lambda t).$$

Example 3: **Simple Harmonic Motion equation**

$$\frac{d^2u}{dt^2}(t) = -\omega^2 u(t) \qquad \omega > 0 \tag{1.4}$$

has general solution

$$u(t) = c_1 \sin(\omega t) + c_2 \cos(\omega t).$$

Often, physical problems correspond to a particular solution with specific values of the constants. These are determined by requiring that the solution satisfy particular initial or boundary conditions. Deterimining appropriate initial or boundary conditions is part of the modelling. There are three archetypal problems which often come up in modelling:

- **Initial Value Problems**
  The value of $u(t)$ and a sufficient number of derivatives are specified at an initial time, $t = t_0$ and we require the values of $u(t)$ for $t > t_0$. For example, taking $t_0 = 0$, find the particular solution of Example 3 which satifies: Example 1 which satifies:

$$u(0) = 0 \qquad \frac{du}{dt}(0) = 1.$$

- **Boundary Value Problems**
  Boundary value problems arise when information on the behaviour of the solution is specified at two different points, x_1 and x_2, and we require the solution, $u(x)$, on the interval $[x_1, x_2]$. Choosing to call the dependent variable x instead of t is a nod to the fact that boundary value problems often arise from problems having spatial dependence rather than temporal dependence. For example, find the solution of

$$\frac{d^2u}{dx^2}(x) = 0$$

  on the interval $[x_1, x_2]$ given that

$$u(x_1) = U_1 \qquad u(x_2) = U_2.$$

- **Eigenvalue Problems**
  Eigenvalue problems are boundary value problems involving a free parameter, λ. The objective is to find the value(s) of λ, the eigenvalue(s), for which the problem has a solution and to determine the corresponding eigenfunction(s), $u_\lambda(x)$. For example, find the values of λ for which

$$\frac{d}{dx}\left[(1-x^2)\frac{du}{dx}(x)\right] + \lambda u(x) = 0$$

  has solutions satisfying the boundary conditions

$$u(0) = 1 \qquad \frac{du}{dt}(1) = 1,$$

  and find the associated eigenfunctions, $u_\lambda(x)$.

In this course we will principally be interested in Initial Value Problems and Boundary Value Problems.

1.2 Dynamical Systems

Systems of ODEs arise very often in applications. They can arise as a direct output of the modelling. Also, more importantly for this course, when we discretise the spatial part of a PDE, we are really replacing the PDE with a (usually large) system of ODEs.

Let $\mathbf{u} : \mathbb{R} \to \mathbb{R}^n$ be a vector–valued function of a single real variable, t. Let $\mathbf{F} : \mathbb{R}^{nm+1} \to \mathbb{R}^n$ be a vector–valued function of the vector $\left(t, \mathbf{u}(t), \frac{d\mathbf{u}}{dt}(t), \ldots, \frac{d^{m-1}\mathbf{u}}{dt^{m-1}}(t)\right)$ taking values in $\mathbb{R}^n$. The equation

$$\frac{d^m\mathbf{u}}{dt^m}(t) = \mathbf{F}\left(t, \mathbf{u}(t), \frac{d\mathbf{u}}{dt}(t), \ldots, \frac{d^{m-1}\mathbf{u}}{dt^{m-1}}(t)\right) \tag{1.5}$$

is an system of m^{th} order ordinary differential equations of dimension n.

This is rather complicated and we rarely need to consider such general systems of equations. In practice, we often deal only with systems of ODEs of first order. This is a result of the fact that any ODE of order m (and by extension any system of equations of order m) can be rewritten as a system of first order ODEs of dimension m.

Reduction of Order
Consider the general m^{th} order ODE given by Eq.(1.1). Define a vector $\mathbf{u}(t)$ in $\mathbb{R}^m$ as follows

$$\mathbf{u}(t) = \left(u(t), \frac{dt}{du}(t), \ldots, \frac{d^{m-1}t}{du^{m-1}}(t)\right)$$

Define the function $\mathbf{F} : \mathbb{R}^{m+1} \to \mathbb{R}^m$ as

$$\mathbf{F}(t, \mathbf{u}(t)) = (u_2(t), \ldots, u_m(t), F(t, \mathbf{u}(t)))\,.$$

By construction, Eq.(1.1) is equivalent to the first order system of ODEs given by

$$\frac{d\mathbf{u}}{dt}(t) = \mathbf{F}\left(t, \mathbf{u}(t)\right) \tag{1.6}$$

Another distinction which we can do away with using the language of first order systems is that between autonomous and non-autonomous systems. A system of ODEs is called *autonomous* if the independent variable (t in most of our examples) does not appear explicitly on the right-hand side and is called *non-autonomous* otherwise. A non-autonomous system of dimension n can be re-written as an autonomous system of dimension $m + 1$ by augmenting it with an additional equation:

Reduction to an autonomous system
Consider the non-autonomous m-dimensional first order system of ODEs given by Eq.(1.6). Define a vector $\mathbf{v}(t)$ in $\mathbb{R}^{m+1}$ as follows $\mathbf{v}(t) = (\mathbf{u}(t), v_{m+1}(t))$. Define the function $\mathbf{G} : \mathbb{R}^{m+1} \to \mathbb{R}^{m+1}$ as

$$\mathbf{G}(\mathbf{v}(t)) = (\mathbf{F}(v_{m+1}(t), (v_1(t), \ldots, v_m(t)), 1)))\,.$$

By construction, Eq.(1.6) is equivalent to the first order autonomous system of ODEs given by

$$\frac{d\mathbf{v}}{dt}(t) = \mathbf{G}\left(\mathbf{v}(t)\right)\,. \tag{1.7}$$

While the notation might be clumsy, the principles are illustrated clearly by the following example:

Example 4: The driven damped pendulum
The angle, $\theta(t)$, made with the vertical by a damped pendulum subject to an external harmonic forcing satisfies the following ODE

$$\frac{d^2\theta}{dt^2} + \alpha\frac{d\theta}{dt} + \beta\sin(\omega_0^2\,\theta) = \delta\cos(\omega_f\,t+\phi). \tag{1.8}$$

where α, β, γ, ω_0, ω_f and ϕ are parameters. This equation is equivalent to the following first order system of dimension 3:

$$\begin{aligned}
\frac{dv_1}{dt} &= v_2 && (1.9)\\
\frac{dv_2}{dt} &= -\alpha v_2 - \beta\sin(\omega_0^2\,v_1) + \delta\cos(\omega_f\,v_3+\phi) && (1.10)\\
\frac{dv_3}{dt} &= 1. && (1.11)
\end{aligned}$$

By deriving methods of solving Eq. (1.7) we can actually solve a lot of different ODE IVP's and, as we shall see, lots of PDE IVP's too.

1.3 Approximation of Derivatives by Finite Differences

Computers are finite machines and cannot represent the real numbers. When faced with a function of a continous variable, the best we can hope to do with a computer is to approximate it by its values at a finite number of points. The process of replacing a function by an array of values at particular points is called *discretisation*. We replace $v(t)$ with $t \in [t_1, t_2]$ by a list of values, $\{v_i \;:\; i = 1\ldots N\}$ where $v_i = v(t_1 + ih)$ where $h = (t_2 - t_1)/N$. There is no reason why the discretisation should be on a set of uniformly spaced points but we will mostly deal with such uniform discretisations. Intuitively, derivatives of v a point can be approximated by differences of the v_i at adjacent points:

$$\frac{dv}{dt}(t_i) \approx \frac{v_{i+1} - v_i}{h}.$$

This intuition is made precise using Taylor's Theorem:

Reminder: Taylor's Theorem:
If $v(t)$ is a Real-valued function which is differentiable $n+1$ times on the interval $[t, t+h]$ then there exists a point, ξ, in $[t, t+h]$ such that

$$\begin{aligned}
v(t+h) &= v(t) + \frac{1}{1!}\,h\,\frac{dv}{dt}(t) + \frac{1}{2!}\,h^2\,\frac{d^2v}{dt^2}(t) + \ldots\\
&\quad + \frac{1}{n!}\,h^n\,\frac{d^nv}{dt^n}(t) + h^{n+1}R_{n+1}(\xi) && (1.12)
\end{aligned}$$

where

$$R_{n+1}(\xi) = \frac{1}{(n+1)!}\,\frac{d^{n+1}v}{dt^{n+1}}(\xi).$$

Note that the theorem does not tell us the value of ξ.

We will use Eq. (1.12) a lot. Taylor's Theorem to first order ($n = 1$) tells us that

$$v(t+h) = v(t) + h\,\frac{dv}{dt}(t) + h^2\,R_2(\xi)$$

for some $\xi \in [t, t+h]$. In the light of the above discussion of approximating derivatives using differences, we can take $t = t_i$ and re-arrange this to give

$$\frac{dv}{dt}(t_i) = \frac{v_{i+1} - v_i}{h} - R_2(\xi)\,h \tag{1.13}$$

This is called the first order forward difference approximation to the derivative of v. Taylor's theorem makes the sense of this approximation precise: we make an error proportional to h (hence the terminology "first order") when we make this approximation. Thus it is ok when h is "small". (Compared to what?)

We could equally well use the Taylor expansion of $v(t-h)$:

$$v(t-h) = v(t) - h\,\frac{dv}{dt}(t) + h^2\,R_2(\xi)$$

for some $\xi \in [t-h, t]$. Rearranging this gives what is called the first order backwards difference approximation:

$$\frac{dv}{dt}(t_i) = \frac{v_i - v_{i-1}}{h} + R_2(\xi)\,h. \tag{1.14}$$

It also obviously makes an error of order h. To get a first order approximation of the derivative requires that we know the values of v at two points. The key idea of more general finite difference approximations is that by using more points we can get better approximations. In what sense do we mean better? The error made in the approximation is proportional to a higher power of h. An important example is the (second order) centred difference formula which we can construct using three values, v_{i-1}, v_i and v_{i+1}. We have two Taylor expansions:

$$v_{i+1} = v_i + h\frac{dv}{dt}(t_i) + \frac{1}{2!}\,h^2\,\frac{d^2v}{dt^2}(t_i) + h^3 R_3^+ \tag{1.15}$$

$$v_{i-1} = v_i - h\frac{dv}{dt}(t_i) + \frac{1}{2!}\,h^2\,\frac{d^2v}{dt^2}(t_i) - h^3 R_3^- \tag{1.16}$$

Subtracting Eq. (1.16) from Eq. (1.15) and rearranging gives:

$$\frac{dv}{dt}(t_i) = \frac{v_{i+1} - v_{i-1}}{2h} + \frac{1}{2}(R_3^+ + R_3^-)h^2. \tag{1.17}$$

By using 3 points, we can get an approximation which is second order accurate. What about even higher order approximations? What would happen if the discretisation is not uniform?

1.4 Time-stepping Algorithms 1 : Euler Methods

1.4.1 Forward Euler Method

We have seen how, if we know v_i and v_{i+1}, Eq. (1.13) allows us to approximate $\frac{dv}{dt}(t_i)$ in a controlled way. We can turn this on its head: if we know v_i and $\frac{dv}{dt}(t_i)$ then we can approximate v_{i+1}:

$$v_{i+1} = v_i + h\,\frac{dv}{dt}(t_i) + R_2 h^2. \tag{1.18}$$

For the system of ODEs given by Eq. (1.7) this gives:

$$\mathbf{v}_{i+1} = \mathbf{v}_i + h\,\mathbf{G}_i + O(h^2). \tag{1.19}$$

Here we have adopted the obvious notation $\mathbf{G}_i = \mathbf{G}(\mathbf{v}(t_i))$. Furthermore, from now on, unless we explicitly need to, we shall stop carrying around the constants preceeding the error term and simply

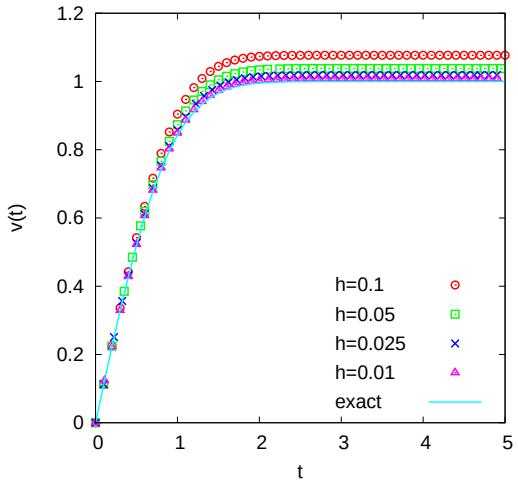

Figure 1.1: Comparison between the numerical solution of Eq. (1.20) obtained using the forward Euler Method and the exact solution, Eq. (1.21) for a range of values of the stepsize, h

denote an n^{th} order error term by $O(h^n)$. Starting from the initial condition, for Eq. (1.7) and knowledge of $\mathbf{G}(\mathbf{v})$) we obtain an approximate value for $\mathbf{v}$ a time h later. We can then iterate the process to obtain an approximation to the values of $\mathbf{v}$ at all subsequent times, t_i. Such an iteration might be possible to carry out analytically n some simple cases (for which we are likely to be able to solve the original equation anyway) but, in general, is ideal for computer implementation. This iteration process is called *time-stepping*. Eq. (1.19) provides the simplest possible time-stepping method. It is called the Forward Euler Method.

In Fig. 1.1, this method is applied to the following problem:

$$\frac{d^2v}{dt^2} + 2\,t\frac{dv}{dt} - av = 0, \tag{1.20}$$

for the particular case of $a = 1$, $v(0) = 0$, $\frac{dv}{dt}(0) = \frac{2}{\sqrt{\pi}}$. For this value of the parameter, a, and these initial conditions, Eq. (1.20) has a simple exact solution:

$$v(t) = \mathrm{Erf}(t), \tag{1.21}$$

where $\mathrm{Erf}(x)$ is defined as

$$\mathrm{Erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x \mathrm{e}^{-y^2}\,dy. \tag{1.22}$$

1.4.2 Backward Euler Method

We could equally well have used the backward difference formula, Eq. (1.14) to derive a time-stepping algorithm in which case we would have found

$$\mathbf{v}_{i+1} = \mathbf{v}_i + h\,\mathbf{G}_{i+1} + O(h^2). \tag{1.23}$$

$\mathbf{v}_{i+1}$ (the quantity we are trying to find) enters on both sides of the equation. This means we are not, in general in a position to write down an explicit formula for $\mathbf{v}_{i+1}$ in terms of $\mathbf{v}_i$ as we did for the Forward Euler Method, Eq. (1.19). Rather we are required to solve a (generally nonlinear) equation to find the value of $\mathbf{v}$ at the next timestep. For non-trivial $\mathbf{G}$ this can be quite hard.

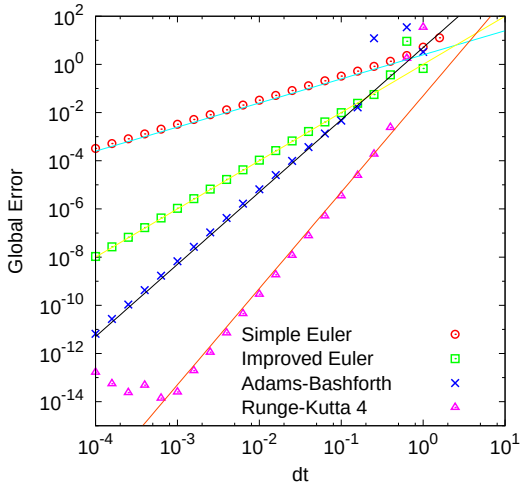

Figure 1.2: Log-Log plot showing the behaviour of the global error as a function of step size for a selection of the timestepping algorithms which we shall study in this chapter applied to Eq. (1.20). The straight lines show the theoretically expected error, h^n, where $n = 1$ for the simple Euler method, $n = 2$ for the improved Euler method, $n = 3$ for the Adams–Bashforth method and $n = 4$ for the 4th order Runge–Kutta method.

1.4.3 Stepwise vs Global Error

In Eq. (1.19), we refer to $R_2 h^2$ as the *stepwise error*. This is distinct from the *global error*, which is the total error which occurs in the approximation of $\mathbf{v}(t_2)$ if we integrate the equation from t_1 to t_2 using a particular timestepping algorithm. If we divide the time domain into N intervals of length $h = (t_2 - t_1)/N$ then $N = (t_2 - t_1)/h$. If we make a stepwise error of $O(h^n)$ in our integration routine, then the global error therefore $O((t_2 - t_1)h^{n-1})$. Hence the global error for the Forward Euler Method os $O(h)$. This is very poor accuracy. This is one reason why the Forward Euler Method is almost never used in practice.

1.4.4 Explicit vs Implicit Methods

The examples of the Forward and Backward Euler methods illustrates a very important distinction between different timestepping algorithms: *explicit* vs *implicit*. Forward Euler is an explicit method. Backward Euler is implicit. Although they both have the same degree of accuracy - a stepwise error of $O(h^2)$ - the implicit version of the algorithm typically requires a lot more work per timestep. However it has superior stability properties.

1.4.5 Improving on the Euler Method

We would like to do better than $O(h)$ for the global error. One way is to keep more terms from the Taylor expansion in the derivation of the algorithm. Using Eq. (1.12) and Eq. (1.7):

$$\begin{aligned} \mathbf{v}_{i+1} &= \mathbf{v}_i + h\frac{d\mathbf{v}}{dt}(t_i) + \frac{1}{2}h^2\frac{d^2\mathbf{v}}{dt^2}(t_i) + O(h^3) \\ &= v_i + hG_i + \frac{1}{2}h^2\frac{d\mathbf{G}}{dt}(\mathbf{v}(t_i)) + O(h^3) \end{aligned} \tag{1.24}$$

Using the Chain Rule together with Eq. (1.7), k^{th} component of $\frac{d^2\mathbf{v}}{dt^2}(t_i)$ evaluated at time t_i is

$$\frac{dG^{(k)}}{dt}(\mathbf{v}_i) = \sum_{j=1}^{m} \left.\frac{dG^{(k)}}{dv^{(j)}}\right|_{t=t_i} G_i^{(j)}.$$

This is starting to look messy and indeed, although the algorithm Eq. (1.24) now has a stepwise error of $O(h^3)$, for nontrivial $\mathbf{G}$ the pain of calculating higher order derivatives of the components of $\mathbf{G}$ with respect to the components of $\mathbf{v}$ means that this approach is not often used. Better alternatives are available to which we now turn.

1.5 Time-stepping Algorithms 2 : Predictor–Corrector Methods

1.5.1 Implicit Trapezoidal Method

So far we have thought of timestepping algorithms based on approximation of the solution by its Taylor series. A complementary way to think about it is to begin from the formal solution of Eq. (1.7):

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \int_{t_i}^{t_i+h} \mathbf{G}(\mathbf{v}(\tau))\, d\tau, \tag{1.25}$$

and think of how to approximate the integral,

$$I = \int_{t_i}^{t_i+h} \mathbf{G}(\mathbf{v}(\tau))\, d\tau. \tag{1.26}$$

The simplest possibility is to use the rectangular approximations familiar from the definition of the Riemann integral. We can use either a left or right Riemann approximation:

$$I \approx h\mathbf{G}(\mathbf{v}(t_i)) \tag{1.27}$$
$$I \approx h\mathbf{G}(\mathbf{v}(t_{i+1})). \tag{1.28}$$

These approximations obviously give us back the Forward and Backward Euler Methods which we have already seen. A better approximation would be to use the Trapezoidal Rule:

$$I \approx \frac{1}{2}h\left[\mathbf{G}(\mathbf{v}(t_i)) + \mathbf{G}(\mathbf{v}(t_{i+1}))\right]$$

which would give us the timestepping algorithm

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \frac{h}{2}(\mathbf{G}_i + \mathbf{G}_{i+1}). \tag{1.29}$$

This is known as the Implicit Trapezoidal Method. The stepwise error in this approximation is $O(h^3)$. To see this, we Taylor expand the terms sitting at later terms in Eq. (1.29) and compare the resulting expansion to the true Taylor expansion.

Example 5: **Accuracy of the Implicit Trapezoidal Method**
Let us assume that we have a scalar equation,

$$\frac{dv}{dt} = G(v).$$

(The extension of the analysis to the vector case is straight-forward but indicially messy.) The true solution at time t_{i+1} up to order h^3 is, from Taylor's Theorem and the Chain Rule:

$$\begin{aligned} v_{i+1} &= v_i + h\frac{dv}{dt}(t_i) + \frac{h^2}{2}\frac{d^2v}{dt^2}(t_i) + R_3\, h^3 \\ &= v_i + hG_i + \frac{1}{2}h^2 G_i\, G_i' + R_3\, h^3 \end{aligned}$$

Let us denote our approximate solution by $\widetilde{v}(t)$. From Eq. (1.29)

$$\widetilde{v}_{i+1} = v_i + \frac{h}{2}\left[G(v_i) + G(v(t_{i+1}))\right]$$

We can write

$$G(v(t_{i+1})) = G(v(t_i)) + h\frac{dG}{dt}(v(t_i)) + R_2 h^2 = G_i + hG_i' G_i + R_2\, h^2.$$

Substituing this back gives

$$\widetilde{v}_{i+1} = v_i + hG + \frac{1}{2}h^2 G_i\, G_i' + \frac{1}{2}h^3 R_2.$$

We now see that $\widetilde{v}_{i+1} - v_{i+1} = O(h^3)$. Hence the Implicit Trapezoidal Method has an $O(h^3)$ stepwise error.

I have written this example out in some detail since it is a standard way of deriving the stepwise error for any timestepping algorithm.

The principal drawback of the Implicit Trapezoidal Method is that it is implicit and hence computationally expensive and tricky to code. Can we get higher order accuracy with an explicit scheme?

1.5.2 Improved Euler Method

A way to get around the necessity of solving implicit equations is try to "guess" the value of $\mathbf{v}_{i+1}$ and hence estimate the value of $\mathbf{G}_{i+1}$ to go into the RHS of Eq. (1.29). How do we "guess"? One way is to use a less accurate explicit method to predict $\mathbf{v}_{i+1}$ and then use a higher order method such as Eq. (1.29) to correct this prediction. Such an approach is called a Predictor–Corrector Method. Here is the simplest example:

- Step 1
  Make a prediction, $\mathbf{v}^*_{i+1}$ for the value of $\mathbf{v}_{i+1}$ using the Forward Euler Method:

  $$\mathbf{v}^*_{i+1} = \mathbf{v}_i + h\mathbf{G}_i,$$

  and calculate

  $$\mathbf{G}^*_{i+1} = \mathbf{G}(\mathbf{v}^*_{i+1}).$$

- Step 2
  Use the Trapezoidal Method to correct this guess:

  $$\mathbf{v}_{i+1} = \mathbf{v}_i + \frac{h}{2}\left[\mathbf{G}_i + \mathbf{G}^*_{i+1}\right]. \tag{1.30}$$

Eq. (1.30) is called the Improved Euler Method. It is explicit and has a stepwise error of $O(h^3)$. The price to be paid is that we now have to evaluate $\mathbf{G}$ twice per timestep. If $\mathbf{G}$ is a very complicated function, this might be worth considering but typically this is a considerable improvement over the Forward Euler Method.

There are two common approaches to making further improvements to the error:

1. Use more points to better approximate the integral, Eq. (1.26). This leads to class of algorithms known as multistep methods.

2. Use predictors of the solution at several points in the interval $[t_i, t_{i+1}]$ and combine them in clever ways to cancel errors. This approach leads to a class of algorithms known as Runge-Kutta methods.

1.5.3 Multistep Methods

Reminder: Lagrange Interpolation:
Consider the $n+1$ points, (t_0, G_0), $(t_1, G_1) \ldots (t_n, G_n)$ constructed from evaluating a function, $G(t)$ at $n+1$ points, $t_0, t_1 \ldots t_n$. The unique polynomial of degree n passing through these point is the Lagrange Interpolant:

$$L_n(t) = \sum_{i=0}^{n} G_i \, p_i(t) \tag{1.31}$$

where

$$p_i(t) = \Pi_{j=0 i \neq j}^{n} \frac{t - t_j}{t_i - t_j}. \tag{1.32}$$

Using these interpolation formulae it is possible to use previously computed values of the solution to more accurately approximate the integral in Eq. (1.26). Suppose that we have equally spaced grid-points, 0, $-h$, $-2h \ldots$. Then the first few Lagrange interpolants are

$$L_1(t) = (-\frac{t}{h})\, G_{-1} + \frac{t+h}{h}\, G_0 \tag{1.33}$$

$$L_2(t) = \frac{t(t+h)}{2h^2} G_{-2} - \frac{t(t+2h)}{h^2} G_{-1} + \frac{(t+h)(t+2h)}{2h^2} G_0 \tag{1.34}$$

$$L_3(t) = -\frac{t(t+h)(t+2h)}{6h^3} G_{-3} + \frac{t(t+h)(t+3h)}{2h^3} G_{-2} \tag{1.35}$$

$$-\frac{t(t+2h)(t-3h)}{2h^3} G_{-1} + \frac{(t+h)(t+2h)(t+3h)}{6h^3} G_0 \tag{1.36}$$

While straightforward to calculate, the formulae become increasingly horrible as n increases. We will derive a 3-step algorithm using $n = 2$ (quadratic interpolation). We shall assume a scalar equation - extension to the vector case is done component-wise. Suppose we know (from previous timesteps) the values of v_{i-2}, v_{i-1} and v_i and hence the values of G_{i-2}, G_{i-1} and G_i. The requirement to know the solution at several previous steps explains the term "multistep" used to describe these algorithms.

1.5.4 Adams–Bashforth Methods

Use the known data points $(-2h, G_{-2})$, $(-h, G_{-1})$ and $(0, G_i)$ and Eq (1.34) to calculate an approximation, $\widetilde{G}(t)$, to $G(t)$ over the interval $[-2h, 0]$. Use $\widetilde{G}(t)$ and Eq. (1.26) to calculate an extrapolated

value for v_1:

$$\begin{aligned} v_1 &= \int_0^h G(\tau)\, d\tau \\ v_1 &\approx v_0 + \int_0^h \widetilde{G}(\tau)\, d\tau \\ v_1 &= v_0 + h\left(\frac{5}{12}G_{-2} - \frac{4}{3}G_{-1} + \frac{23}{12}G_0\right) \end{aligned} \tag{1.37}$$

Eq. (1.37) is the 3-step Adams–Bashforth Method.

1.5.5 Adams–Moulton Methods:

A more sophisticated approach is to use the Adams–Bashforth Method, Eq. (1.37), as a predictor for a more refined estimate of the integral Eq. (1.26):

- Step1 - Predictor:

$$v_1^* = v_0 + h\left(\frac{5}{12}G_{-2} - \frac{4}{3}G_{-1} + \frac{23}{12}G_0\right). \tag{1.38}$$

  From v_1^* we estimate $G_1^* = G(v_1^*)$.

- Step 2 - Corrector:
  Now use the known data points $(-h, G_{-1})$ and $(0, G_0)$ with the predicted data point, (h, G_1^*) to calculate an approximation, $\widetilde{G}^*(t)$, to $G(t)$, over the interval $[-h, h]$ which we then insert into Eq. (1.26). The Lagrange polynomial passing through the points $(-h, G_{-1}), (0, G_0), (h, G_1)$ is

$$L_2'(t) = \frac{t(t-h)}{2h^2}G_{-1} - \frac{(t-h)(t+h)}{h^2}G_0 + \frac{t(t+h)}{2h^2}G_1. \tag{1.39}$$

  We use our predicted value, G_1^*, for G_1, in Eq. (1.39) and substitute the new interpolation into Eq. (1.26) and integrate to obtain the corrected value:

$$\begin{aligned} v_1 &= v_0 + \int_0^h \widetilde{G}^*(\tau)\, d\tau \\ &= v_0 + h\left(-\frac{1}{12}G_{-1} + \frac{2}{3}G_0 + \frac{5}{12}G_1^*\right) \end{aligned} \tag{1.40}$$

Be careful: the interpolating polynomial in the predictor and corrector stages are *different* so the resulting coefficients are not the same. Eqs. (1.38) and (1.40) together constitute the 3-step Adams–Moulton Method. It is explicit, multistep and has a stepwise error of $O(h^3)$. More accurate Adams–Bashforth and Adams–Moulton methods can be derived using higher order Lagrange Interpolants. For example, an Adams–Bashforth method with $O(h^4)$ stepwise error can be derived using knowledge of 3 previous steps together with Eq (1.35). The formulae for the first few are listed at [3]. The big disadvantage of multistep methods is that they are obviously not self-starting and must rely on a comparably accurate single-step method to calculate enough initial points from the initial data to get going.

1.6 Time-stepping Algorithms 3 : Runge–Kutta Methods

Runge-Kutta methods aim to retain the accuracy of the multistep predictor corrector methods described in Sec. 1.5 but without having to use more than the current value to predict the next one - ie they are *self-starting*. The idea is roughly to make several predictions of the value of the solution at several points in the interval $[t_1, t_{i+1}]$ and then weight them cleverly so as to cancel errors.

An n^{th} order Runge–Kutta scheme for Eq. (1.7) looks as follows. We first calculate n estimated values of $\mathbf{G}$ which are somewhat like the predictors used in Sec. 1.5:

$$\begin{aligned}
\mathbf{g}_1 &= \mathbf{G}(\mathbf{v}_i) \\
\mathbf{g}_2 &= \mathbf{G}(\mathbf{v}_i + a_{2\,1}\, h\, \mathbf{g}_1) \\
\mathbf{g}_3 &= \mathbf{G}(\mathbf{v}_i + a_{3\,1}\, h\, \mathbf{g}_1 + a_{3\,2}\, h, \mathbf{g}_2) \\
&\vdots \\
\mathbf{g}_n &= \mathbf{G}(\mathbf{v}_i + a_{n\,1}\, h\, \mathbf{g}_1 + \ldots + a_{n\,n-1}\, h, \mathbf{g}_{n-1})
\end{aligned}$$

We then calcalate v_{i+1} as

$$\mathbf{v}_{i+1} = \mathbf{v}_i + h\,(b_1\mathbf{g}_1 + b_2\mathbf{g}_2 + \ldots + b_n\mathbf{g}_n). \tag{1.41}$$

The art lies in choosing the $a_{i\,j}$ and b_i such that Eq. (1.41) has a stepwise error of $O(h^{n+1})$. The way to do this is by comparing Taylor expansions as we did to determine the accuracy of the Improved Euler Method in Ex. 5 and choose the values of the constants such that the requisite error terms vanish. It turns out that this choice is not unique so that there are actually parametric families of Runge-Kutta methods of a given order.

We shall derive a second order method explicitly for a scalar equation. Derivations of higher order schemes provide nothing new conceptually but require a lot more algebra. Extension to the vector case, as usual, is straightforward but requires care with indices. Let us denote our general 2-stage Runge-Kutta algorithm by

$$\begin{aligned}
g_1 &= G(v_i) \\
g_2 &= G(v_i + a\, h\, g_1) \\
v_{i+1} &= v_i + h\,(b_1 g_1 + b_2 g_2).
\end{aligned}$$

Taylor expand g_2 up to second order in h:

$$\begin{aligned}
g_2 &= G(v_i) + a\, h\, G_i\, \frac{dG}{dv}(v_i) + R_2\, h^2 \\
&= G_i + a\, h\, G_i\, G_i' + R_2\, h^2
\end{aligned}$$

Our approximate value of v_{i+1} is then

$$\widetilde{v}_{i+1} = v_i + (b_1 + b_2)hG_i + ab_2h^2 G_i\, G_i' + R_2\, h^3.$$

If we compare this to the usual Taylor expansion of v_{i+1} we see that we can make the two expansions identical up to $O(h^3)$ if we choose

$$\begin{aligned}
b_1 + b_1 &= 1 \\
ab_2 &= \frac{1}{2}.
\end{aligned}$$

The method then has a step-wise error of $O(h^3)$. Note that we have 2 equations for 3 unknowns so there is a one–parameter family of Runge–Kutta algorithms of this order. A popular choice is $b_1 = b_2 = \frac{1}{2}$ and $a = 1$. This gives, what is often considered the "standard" second order Runge-Kutta scheme:

$$\begin{aligned}
g_1 &= G(v_i) \\
g_2 &= G(v_i + h\, g_1) \\
v_{i+1} &= v_i + \frac{h}{2}\,(g_1 + g_2).
\end{aligned} \tag{1.42}$$

Perhaps it looks familiar. The standard fourth order Runge–Kutta scheme, with a stepwise error of $O(h^5)$, is really the workhorse of numerical integration since it has a very favourable balance of

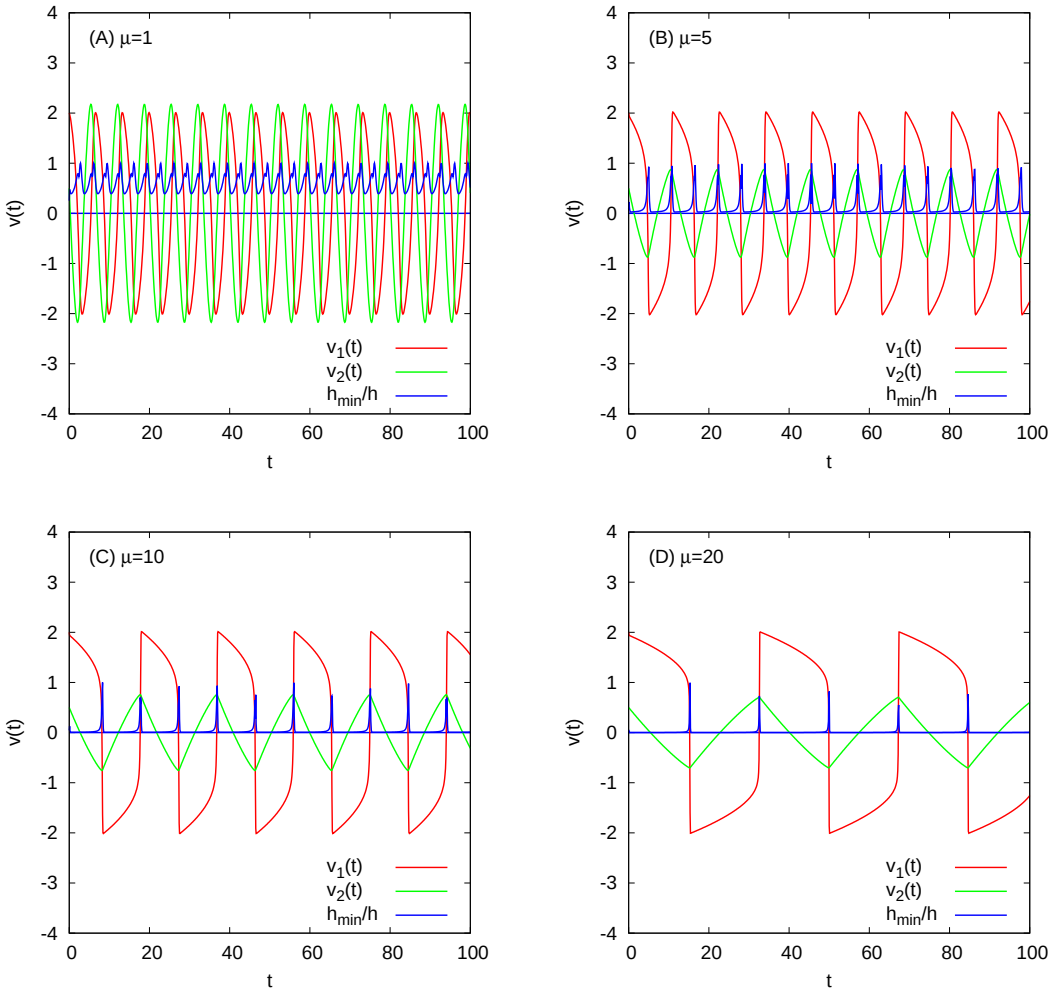

Figure 1.3: Solutions of Eq. (1.44) for several different values of μ obtained using the forward Euler Method with adaptive timestepping. Also shown is how the timestep varies relative to its global minimum value as the solution evolves.

accuracy, stability and efficiency properties. It is often the standard choice. We shall not derive it but you would be well advised to use it in your day-to-day life. It takes the following form:

$$\begin{aligned} \mathbf{g}_1 &= \mathbf{G}(\mathbf{v}_i) \\ \mathbf{g}_2 &= \mathbf{G}(\mathbf{v}_i + \frac{h}{2}\mathbf{g}_1) \\ \mathbf{g}_3 &= \mathbf{G}(\mathbf{v}_i + \frac{h}{2}\mathbf{g}_2) \\ \mathbf{g}_4 &= \mathbf{G}(\mathbf{v}_i + h\,\mathbf{g}_3) \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + \frac{h}{6}(\mathbf{g}_1 + 2\mathbf{g}_2 + 2\mathbf{g}_3 + \mathbf{g}_4). \end{aligned} \tag{1.43}$$

1.7 Adaptive Timestepping

Up until now we have talked a lot about the behaviour of numerical algorithms as $h \to 0$. In practice we need to operate at a finite value of h. How do we choose it? Ideally we would like to choose

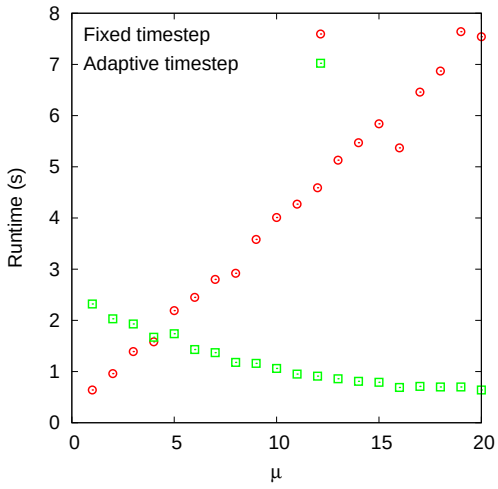

Figure 1.4: Comparison of the performance of the simple Euler method with adaptive and fixed timestepping strategies applied to Eq. (1.44) for several different values of μ.

the timestep such that the error per timestep is less than some threshold, ϵ. We measure the error by comparing the numerical solution at a grid point, $\widetilde{\mathbf{v}}_i$, to the exact solution, $\mathbf{v}(t_i)$, assuming it is known. Two criteria are commonly used:

$$\begin{aligned} E_a(h) = |\widetilde{\mathbf{v}}_i - \mathbf{v}_i| \leq \epsilon \quad & \text{Absolute error threshold,} \\ E_r(h) = \frac{|\widetilde{\mathbf{v}}_i - \mathbf{v}_i|}{|\mathbf{v}_i|} \leq \epsilon \quad & \text{Relative error threshold.} \end{aligned}$$

Intuitively, and mathematically from Taylor's Theorem, the error is largest when the solution is rapidly varying. Often the solution does not vary rapidly everywhere. By setting the timestep in this situation so that the error threshold is satisfied during the intervals of rapid variation, we end up working very inefficiently during the intervals of slower variation where a much larger timestep would have easily satisfied the error threshold. Here is a two–dimensional example which you already know: a relaxation oscillator when $\mu \gg 1$:

$$\begin{aligned} \frac{dx}{dt} &= \mu(y - (\frac{1}{3}x^3 - x)) \\ \frac{dt}{d}y &= -\frac{1}{\mu}x. \end{aligned} \tag{1.44}$$

The solution of this system has two widely separated timescales - jumps which proceed on a time of $O(\frac{1}{\mu})$ (very fast when $\mu \gg 1$) and crawls which proceed on a timescale of μ (very fast when $\mu \gg 1$). To integrate Eqs. (1.44) efficiently for a given error threshold, we need to take small steps during the jumps and large ones during the crawls. This process is called *adaptive timestepping*.

The problem, of course, is that we don't know the local error for a given timestep since we do not know the exact solution. So how do we adjust the timstep if we do not know the error? A common approach is to use trial steps: at time i we calculate one trial step starting from the current solution, $\mathbf{v}_i$ using the current value of h and obtain an estimate of the solution at the next time which we shall call $\mathbf{v}_{i+1}^{\mathrm{B}}$. We then calculate a second trial step starting from $\mathbf{v}_i$ and taking *two* steps, each of length $h/2$, to obtain a second estimate of the solution at the next time which we shall call $\mathbf{v}_{i+1}^{\mathrm{S}}$. We can then estimate the local error as

$$\Delta = \left|\mathbf{v}_{i+1}^{\mathrm{B}} - \mathbf{v}_{i+1}^{\mathrm{S}}\right|.$$

We can monitor the value of Δ to ensure that it stays below the error threshold.

If we are using an n^th order method, we know that $\Delta = ch^n$ for some c. The most efficient choice of step is that for which $\Delta = \epsilon$ (remember ϵ is our error threshold. Thus we would like to choose the

new timestep, $\tilde{h}$ such that $c\tilde{h}^n = \epsilon$. But we know from the trial step that $\frac{1}{2}\Delta/h^n$. From this we can obtain the following rule to get the new timestep from the current step, the required error threshold and the local error estimated from the trial steps:

$$\tilde{h} = \left(\frac{\epsilon}{\Delta}\right)^{\frac{1}{n}} h. \tag{1.45}$$

It is common to include a "safety factor", $\sigma_1 < 1$ to ensure that we stay a little below the error threshold:

$$\tilde{h}_1 = \sigma_1 \left(\frac{\epsilon}{\Delta}\right)^{\frac{1}{n}} h. \tag{1.46}$$

Equally, since the solutions of ODE's are usually smooth, it is often sensible to ensure that the timestep does not increase or decrease by more than a factor of σ_2 (2 say) in a single step. To do this we impose the second constraint:

$$\begin{aligned} \tilde{h} &= \max\left\{\tilde{h}_1, \frac{h}{\sigma_2}\right\} \\ \tilde{h} &= \min\left\{\tilde{h}_1, \sigma_2 h\right\}. \end{aligned}$$

1.8 Snakes in the Grass

Equiped with an accurate, stable integration algorithm and an adaptive timestepping strategy you might now feel confident in your ability to obtain the solution to any problem which applications might throw at you. Of course problems arise. We will mention two common ones which you should be aware of - singular problems and stiff problems.

1.8.1 Singularities

A singularity occurs at some time, $t = t^*$ if the solution of an ODE tends to infinity as $t \to t^*$. Naturally this will cause problems for any numerical integration routine. A properly functioning adaptive stepping strategy should reduce the timestep to zero as a singularity is approached. The following is a simple example which you might like to test some of your integration routines on.

Example 6: **A simple singularity**
Consider the equation,

$$\frac{dv}{dt} = \lambda v^2.$$

with initial data $v(0) = v_0$. This equation is separable and has solution

$$v(t) = \frac{v_0}{1 - \lambda\, v_0\, t}.$$

This clearly has a singularity at $t^* = (\lambda\, v_0)^{-1}$.

1.8.2 Stiffness

Stiffness is an unpleasant property of some problems which causes adaptive algorithms to require very small timesteps even though the solution is not changing very quickly. There is no uniformly accepted definition of stiffness. Whether a problem is stiff or not depends on the equation, the initial condition, the numerical method being used and the interval of integration. The common feature of stiff problems elucidated in a nice article by Moler [1], is that the required solution is slowly varying but "nearby" solutions vary rapidly. The archetypal stiff problem is the decay equation, Eq. (1.3), with

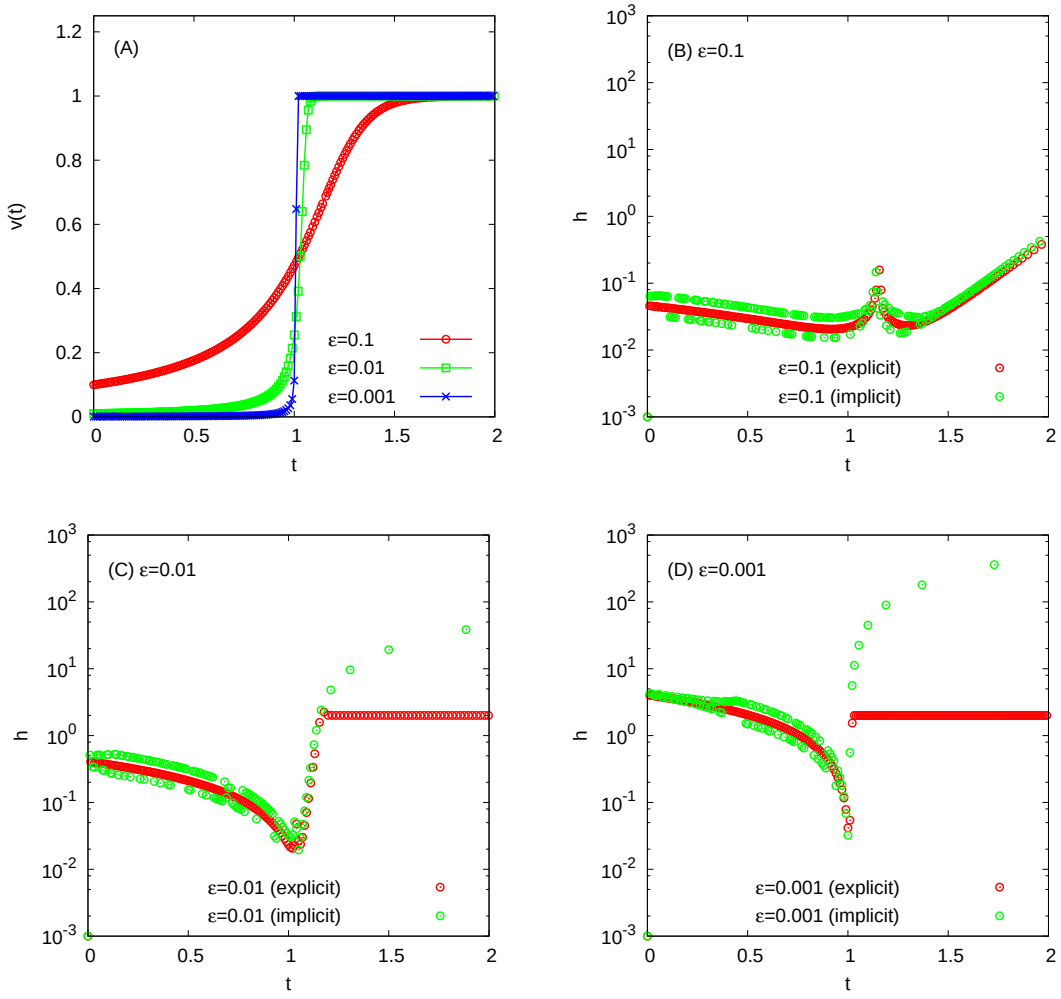

Figure 1.5: Comparison between behaviour of the forward and backward Euler schemes with adaptive timestepping applied to the problem of Ex. 7 for a range of values of ϵ.

$\lambda >> 1$. Efficient solution of stiff problems typically require implicit algorithms. Explicit algorithms with proper adaptive stepping will work but usually take an unfeasibly long time.

Here is a more interesting nonlinear example taken from [1]:

Example 7: A Stiff Problem
Consider the equation,

$$\frac{dv}{dt} = v^2 - v^3.$$

with initial data $v(0) = \epsilon$. Find the solution on the time interval $[0, 2/\epsilon]$.

Its solution is shown in Fig. 1.5(A). The problem becomes stiff as ϵ is decreased. We see, for a given error tolerance (in this case, a relative error threshold of $10\times^{-5}$), that if $\epsilon \ll 1$ the implicit backward Euler scheme can compute the latter half of the solution (the stiff part) with enormously larger timesteps the corresponding explicit scheme. An illuminating animation and further discussion of stiffness with references is available on Scholarpedia [2].