UNIVERSITY OF PORTSMOUTH

Differential Equations

HAMIDREZA KHALEGHZADEH

Introduction

Differential equations underlie many types of scientific theory and simulation.

At various stages in past few weeks we have seen *partial differential equations*, whilst avoiding any detailed discussion.

This week we investigate numerical approaches to solving differential equations, concentrating now on *ordinary differential equations* (ODEs).

Plan:

- Reprise on ODEs
- Discretization and numerical solution
- Special methods for Newton's laws

Generic ODE

In an ordinary differential equation there is one *independent variable* often referred to as t (think, time), but we will use x and one or more *dependent variables* which we will refer to as y. Eventually y may be a vector.

A first order ODE expresses the derivative of y as a function of the current values y and t:

$$\frac{dy}{dt} = f(y, t)$$

We are given the function f, and we want to solve for y as a function of t:

$$y = y(t)$$

Initial Value Problem

To make the solution $y(t)$ unique, we usually fix an initial value of y at some initial time $t = t_0$:

$$y(t_0) = y_0$$

ODEs are thus usually presented as *Initial Value Problems* (IVPs).

Example: Consider a differential equation below with an initial condition $y(0) = 2$.

$$\frac{dy}{dt} = 4t$$

As learnt in math,

$$y(t) = 4\frac{t^2}{2} + B$$

where B is a constant. We can obtain the exact value of B using the initial condition.

Special Cases I

A simple case is when $f(y,t) = f(t)$, i.e. f only depends on the independent variable t, not on y.

For example:

$$\frac{dy}{dt} = ct$$

where c is a constant. The general solution is:

$$y = \frac{c}{2}t^2 + B$$

where B is any constant. Note this is just $y = \int f(t)\,dt$.

In the IVP, if we are given $t_0 = 0$, the solution can be written as:

$$y = \frac{c}{2}t^2 + y_0$$

Special Cases II

Another simple case is when $f(y,t) = f(y)$, i.e. f only depends on the dependent variable y, not on t.

For example:

$$\frac{dy}{dt} = cy$$

where c is a constant. The general solution is:

$$y = Be^{ct}$$

where again B is any constant.

In the IVP, if we are given $t_0 = 0$, the solution can be written as:

$$y = y_0 e^{ct}$$

General Case

For general $f(y,t)$ it may not be possible to find any closed-form analytic solution, and we may have to resort to numerical methods.

Numerical methods, including generalizations to multidimensional and higher order ODEs, are the main topic of this lecture.

Discretization of Independent Variable

In numerical methods it is always necessary to discretize the independent variable t, so we only have to consider a finite number of values.

The simplest case, which we consider in this lecture, is to discretize with a fixed step size h. So we consider values:

$$t = \{t_0, t_0 + h, t_0 + 2h, t_0 + 3h, \cdots\}$$

We label these values $t_0, t_1, t_2, t_3, \cdots$.

(Note h could take a negative value if for example we wish to integrate *backwards* from some final time).

Many more sophisticated algorithms will adjust h adaptively so it takes different values in different steps, but the basic principles we consider still apply.

Euler Method

Simplest and most naïve approach to numerical integration of ODEs.

We have $dy/dt = f(y,t)$ and if we approximate at time step n:

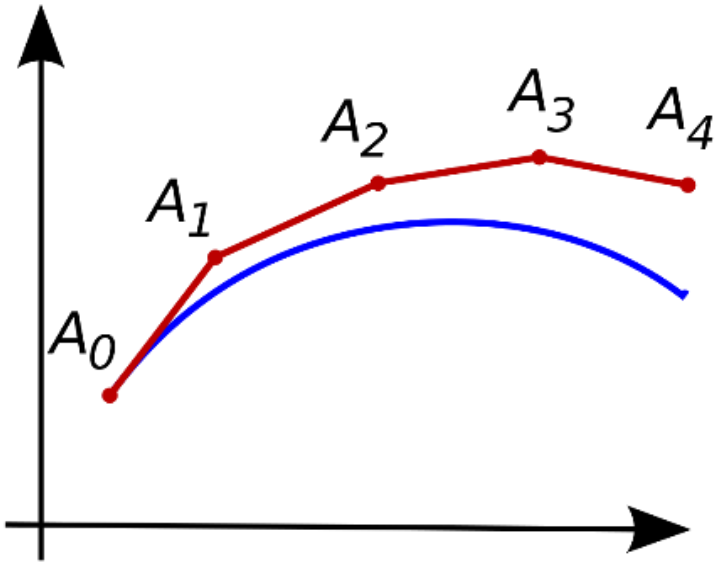
$$\frac{dy}{dt} = \frac{y_{n+1} - y_n}{h}$$

With trivial rearrangement we get the Euler method:

$$y_{n+1} = y_n + hf(y_n, t_n)$$

for any $n > 0$.

Euler Method as Polygonal Approximation†



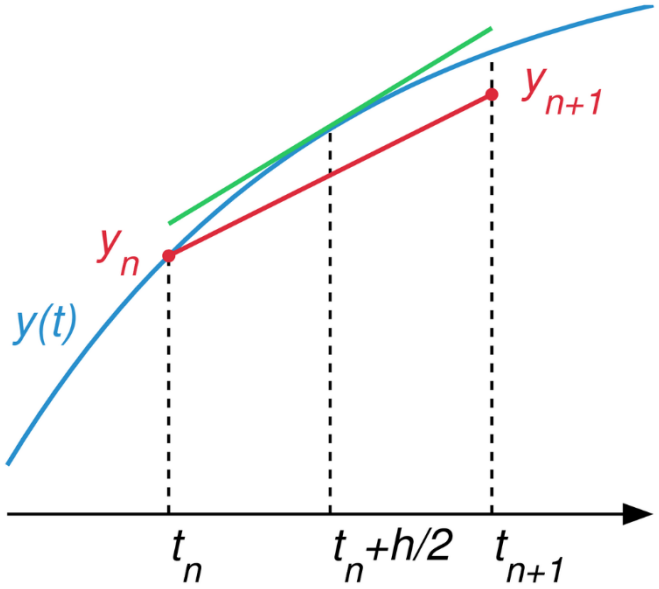
Equivalent to making polygonal approximation to true solution (blue curve).

Euler method is simple, but quite inaccurate – compared with other methods requires small step size therefore more computation for same accuracy.

†Image from https://en.wikipedia.org/wiki/Euler_method

The Midpoint Method

In computing y_{n+1}, rather than taking $f(y_n, t_n)$ as the estimate of the derivative, it would be more accurate to somehow take the value of f at some mid-point between t_n and t_{n+1} - see figure below†:



†Image from https://en.wikipedia.org/wiki/Midpoint_method

Implementing Midpoint

Unfortunately we don't know the "midpoint" value of y exactly.

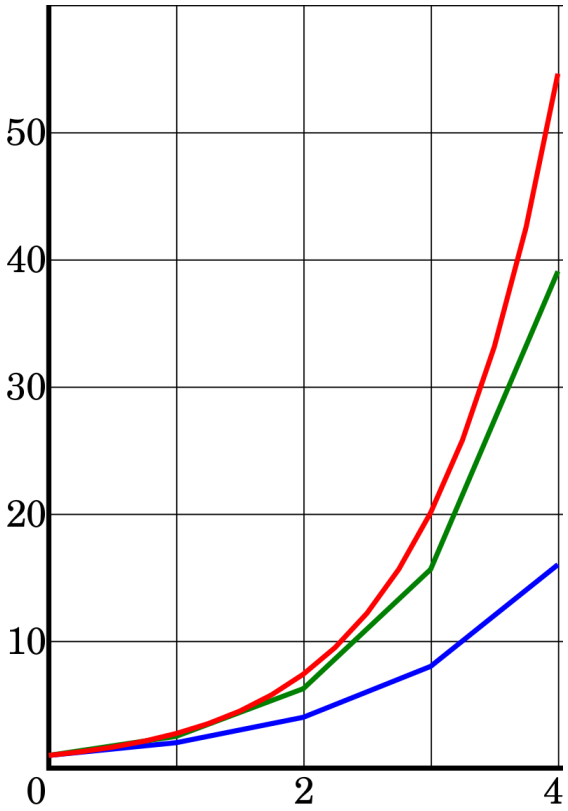
If we approximate it by $y_n + h/2 \cdot f(y_n, t_n)$, that yields the overall update:

$$y_{n+1} = y_n + hf\left(y_n + \frac{h}{2} f(y_n, t_n), t_n + \frac{h}{2}\right)$$

This involves twice as many evaluations of f in the update. But it can be shown that the error in the step is now $O(h^3)$, whereas it was $O(h^2)$ for the Euler method.

Thus method is much more accurate, and can often tolerate larger step sizes, h.

Accuracy of Midpoint vs Euler†



Equation is $dy/dy = y$. Red curve is exactly solution, blue Euler method, green Midpoint method.

†ibid.

Runge-Kutta Methods

The Midpoint Method is prototype for a whole series of related methods of increasing complexity, referred to generically as the *Runge-Kutta methods*.

These always involve multiple evaluations of the function $f(y,t)$ at points intermediate between t_n and t_{n+1}.

Generally speaking, the more intermediate evaluations the higher the accuracy of the update. In one of the best known RK methods (see next slide) the error term is of order $O(h^5)$.

(It is referred to as an $O(h^4)$ *method* because in an expansion in powers of h, it is accurate through terms up to terms proportional h^4. The Euler method by comparison is on $O(h)$ *method*).

The RK4 method

Define following different approximations to the gradient at points between t_n and t_{n+1}:

$$k_1 = f(y_n, t_n)$$

$$k_2 = f\left(y_n + h\frac{k_1}{2}, t_n + \frac{h}{2}\right)$$

$$k_3 = f\left(y_n + h\frac{k_2}{2}, t_n + \frac{h}{2}\right)$$

$$k_4 = f(y_n + hk_3, t_n + h)$$

(one approximation at t_n, two approximations at the midpoint, and one approximation at t_{n+1}). The final update is:

$$y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$

Higher Order Differential Equations

All the examples so far were for first order differential equations – i.e. they only involved the first derivative dy/dt.

Many important differential equations in practice involve higher order derivatives like d^2y/dt^2. The most famous is Newton's 2nd law, which in general looks like this:

$$m\frac{d^2x}{dt^2} = F(x, \frac{dx}{dt}, t)$$

Where now the dependent variable is the position, x, and F is the force as a function of position, velocity and time.

(Resistive forces often depend on velocity, and even for fixed position and velocity external forces may be time dependent).

Reduction to Coupled First Order Equations

By the simple trick of introducing velocity, v, as an *additional dependent variable*, Newton's law can be reduced to a pair of pair of first order equations:

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

$$m\frac{dv}{dt} = F(x, v, t)$$

Then defining vectors $y = (x, v)$ and $f(y, t) = (v, m^{-1}F(x, v, t))$ we reduce this second order system to a first order system in two dependent variables, and all the earlier methods apply.

This is a general trick – systems of higher order ordinary differential equations can always be reduced to systems of first order equations by introducing new variables to represent the lower order derivatives.

Numerical Libraries for ODEs

So the problem of solving ordinary differential equations numerical can largely be addressed by providing libraries that solve coupled systems of first order equations.

They require the user to provide:

- A vector-valued user-defined function f that depends on a vector of dependent variables and a single independent variable.
- A vector of initial values, y_0, for the dependent variables.
- An initial and final target value for the independent variable, t.

The library should return the final values of y and possibly a sequence of intermediate t and y values.

In Python an example of such a library is *scipy.integrate*.

Further Considerations for Newtonian Systems

When simulating systems governed by the "conservative" form of Newton's 2nd law:

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

where in general x may be a (very) large vector representing positions of many objects, there may be considerations beyond having good $O(h^n)$ numerical ccuracy.

An important features of these systems is that *energy* (and possibly certain other quantities like *angular momentum*) are conserved, and this may be important in long term stability of the simulated systems.

It turns out that many standard, otherwise highly accurate, algorithms for ODEs don't do a very good job of conserving such quantities.

The Leapfrog Method

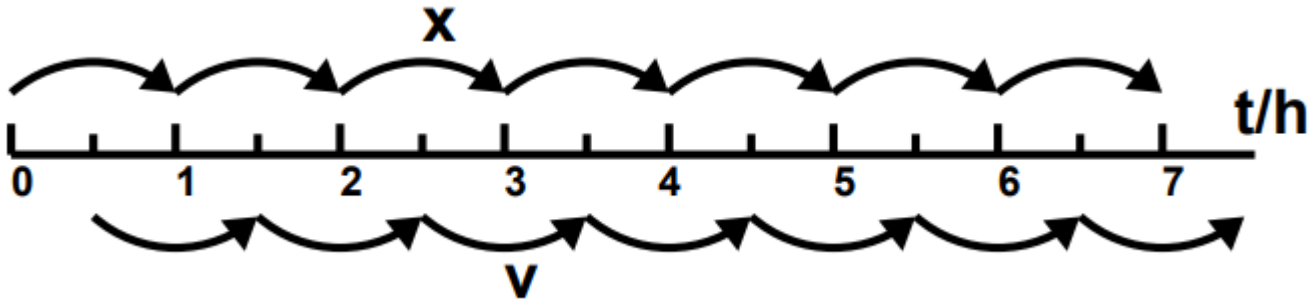
A simple algorithm that works unexpectedly well for conservative systems is the *Leapfrog method*.

It is quite similar to the Euler method, but while it defines positions only on integer steps as before, velocities are *only* defined for half-integer steps (i.e. a time values $t = t_0 + (n + ½)h$ - in other words at the midpoints).

If the differential equation is written as $\frac{d^2x}{dt^2} = A(x)$ the Leapfrog update rule is:

$$a_n = A(x_n)$$
$$v_{n+½} = v_{n-½} + ha_n$$
$$x_{n+1} = x_n + hv_{n+½}$$

Leapfrog Characteristics



The method is visualized above†.

One immediate numerical advantage over the Euler Method is that it has accuracy $O(h^2)$ rather than $O(h)$.

Physics advantages it has over most numerically more accurate systems:

- It is invariant under *time reversal* – a property of physical systems.
- It is a *symplectic* update, which has a consequence that it conserves an approximation to the *total energy* of the system.
- Reportedly can also conserve *angular momentum*.

†Image from http://physics.ucsc.edu/~peter/242/leapfrog.pdf

Starting up Leapfrog

Usually we are initially given value of velocity v_0 (and x_0) at time t_0.

We can "bootstrap" the update by initially doing a half step of Euler:

$$v_{½} = v_0 + ½hA(x_0)$$

Velocity Verlet Method

If Leapfrog is started up as on the previous slide, an entirely equivalent formulation is the *Velocity Verlet Method*:

$$v_{n+½} = v_n + ½hA(x_n)$$
$$x_{n+1} = x_n + hv_{n+½}$$
$$v_n = v_{n+½} + ½hA(x_{n+1})$$

If drop the subscripts in favour of repeated (vector) assignments of programmatic arrays x, v, and a, with a initialized to $A(x_0)$, the form of the iteration is particularly convenient:

$$v := v + ½ha$$
$$x := x + hv$$
$$a := A(x)$$
$$v := v + ½ha$$