# Chapter 4

# Solution of ordinary Differential Equations

**Highly suggested Reading**

Press *et al.*, 1992. Numerical Recipes, 2nd Edition. Chapter 16. (Good description). See also chapter 17. And Matlab help on ODE's

## 4.1 What they are and where they come from

Ordinary differential equations (ODE's) are equations where all variables are functions of only a single independent variable such as time or position. The simplest ODE (which we will use as a model problem) is the equation for radioactive decay of some element.

$$\frac{dc}{dt} = -\lambda c \tag{4.1.1}$$

where $c$ is the mean concentration of the element in some volume and $\lambda$ is the decay constant. Equation (4.1.1) states that the rate of the decay of the element is proportional to its current concentration and has the analytic solution $c = c_0 \exp(-\lambda t)$ where $c_0$ is the initial concentration at time $t = 0$. Figure 4.1 demonstrates the basic numerical problem inherent in solving Eq. (4.1.1) (with $\lambda = 1$). The right hand side of (4.1.1) forms a *direction field* that shows how any given concentration (and time) will change. For a given initial condition, the direction field should form the tangents to a trajectory that is the true solution. I.e. the trick is to start at some specific concentration $c_0$ at $t = 0$ and pick your way carefully through the direction field to some final concentration at time $t$.

Before discussing how we actually solve this sort of problem, it is worth discussing other sources of ODE's. Single equations for higher order ODE's can also be derived such as

$$\frac{d^2 f}{dx^2} + p(x)\frac{df}{dx} + q(x) = 0 \tag{4.1.2}$$

for example from problems of coupled blocks and springs. Fortunately, one of the lovely features of higher order ODE's is that they can always be rewritten as

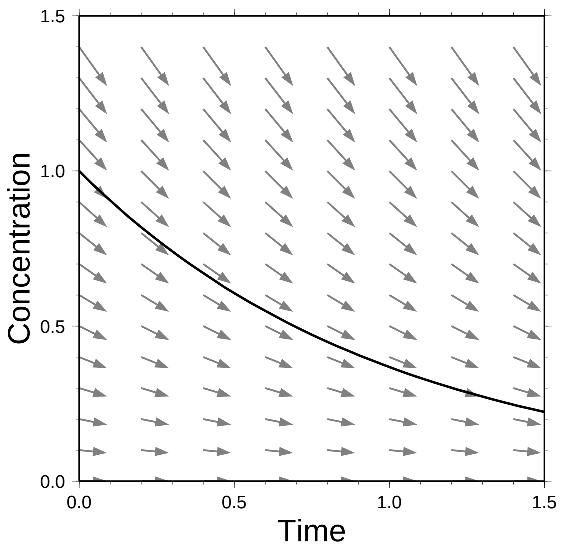


Figure 4.1: The direction field and solution trajectory for the simplest ODE $dc/dt = -c$. The basic numerical approach is to note that the derivative of the function is known for all values of concentration and time (and forms the field of arrows). The problem is to shoot your way *accurately* and efficiently from point to point. This is the entire raison d'etre of ODE solvers.

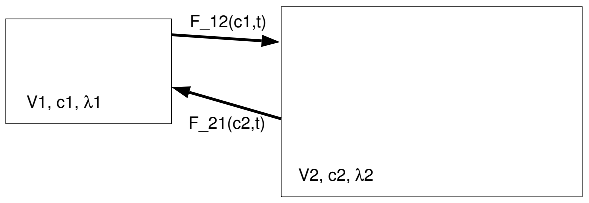
systems of first-order ODE's by defining *auxiliary variables*, e.g. Eq. (4.1.2) can also be written as

$$\frac{df}{dx} = g \tag{4.1.3}$$

$$\frac{dg}{dx} = -p(x)g - q(x) \tag{4.1.4}$$

so if Eq. (4.1.1) can be solved for one variable, the systems can be solved using the same techniques for multiple variables (if all the variables are known at $t = 0$).

A more physical source for systems of ODE's are *Box Models* (see Chapter 1) where each box or reservoir contains the *average* value of some species (chemical, biological or fluid mechanical) and the boxes are connected by fluxes and may have sources or sinks. For example if we treat the upper and lower mantle as two reservoirs of volumes $V_1$ and $V_2$ that are connected by fluxes $F_{12}$ and $F_{21}$, e.g.



Then we can write a conservation equation for the average concentration of a radioactive species in both reservoirs as

$$\begin{aligned} \frac{dc_1}{dt} &= \frac{1}{V_1}\left[F_{21}(c_2,t) - F_{12}(c_1,t)\right] - \lambda c_1 && (4.1.5) \\ \frac{dc_2}{dt} &= -\frac{1}{V_2}\left[F_{21}(c_2,t) - F_{12}(c_1,t)\right] - \lambda c_2 && (4.1.6) \end{aligned}$$

As long as the right hand side of these equations can be evaluated for any given concentrations and time, then the concentrations can be updated using a classic ODE solver. It is straightforward to extend this two box model to a multi-box or to predator-prey models etc.

**Note** If your system of equations is linear, i.e. can be written as

$$\frac{d\mathbf{c}}{dt} = A\mathbf{c} \qquad (4.1.7)$$

where **c** is a vector of concentrations (or variables) and $A$ is a constant coefficient matrix, then there is an analytic solution to these equations based solely on the eigenvectors and eigenvalues of $A$. (e.g. see Strang, "An introduction to linear algebra"). If $A$ can be diagonalized then we can write the solution as

$$\mathbf{c}(t) = Se^{\Lambda t}S^{-1}\mathbf{c}_0 \qquad (4.1.8)$$

where $S$ is an *invertible* matrix of eigenvectors of $A$ with corresponding diagonal eigenvalue matrix $\Lambda$ and $\mathbf{c}_0$ is the concentration vector at $t = 0$. Note for reasonable sized $A$, matlab can be used to find both $S$ and $\Lambda$. Understanding the eigenvectors and eigenvalues of $A$ can be much more informative than the actual individual solutions in determining the behaviour of the general problem. This approach can also be useful even for non-linear systems of equations that can be linearized about certain points in phase-space.

Less obvious sources of ODE's come from *particle tracking* problems where we have some set of particles in an imposed flow field **V** and want to follow them around and record any changes of property that they might experience. In this case each particle can be described by a system of ODE's like

$$\begin{aligned} \frac{dc}{dt} &= f(\mathbf{x},t) && (4.1.9) \\ \frac{d\mathbf{x}}{dt} &= \mathbf{V} && (4.1.10) \end{aligned}$$

where **x** is the vector position of the particle and $f(\mathbf{x},t)$ is a function that says how locally $c$ changes with time. What is not obvious is that Eq. (4.1.9) is actually equivalent to the Lagrangian formulation of the simplest partial differential equation

$$\frac{\partial c}{\partial t} + \mathbf{V}\cdot\boldsymbol{\nabla} c = f \qquad (4.1.11)$$

(we will discuss the relationship between ODE's and PDE's in a later section). In addition, ODE's can also be generated by *modal expansions* or *spectral methods*.

If we can write a trial solution of our problem as a sum of known spatial functions (such as sines and cosines) with time-dependent amplitudes, i.e.

$$c(\mathbf{x}, t) = \sum_{n=1}^{N} a_n(t) f_n(\mathbf{x}) \tag{4.1.12}$$

then substitution into more general partial differential equations will yield a (generally non-linear) mess of ODE's for the $N$ amplitudes i.e.

$$\frac{d\mathbf{a}}{dt} = \mathbf{g}(\mathbf{a}, t) \tag{4.1.13}$$

This is the approach used to derive the Lorenz equations which are a simplified version of Rayleigh-Benard convection (i.e. convection in a sheet). This approach is also often used in finite-element techniques for time-dependent problems.

No matter where they come from, however, the general form of ODE's that we will be dealing with is

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t), \qquad \mathbf{y} = \mathbf{y}_0, t = t_0 \tag{4.1.14}$$

where **y** is a vector of state variables(i.e. a point in state space) and **f** is a function that describes how the position will change as a function of time and state. As long as the right hand side of these equations can be evaluated at a point, then the numerical problem reduces to trying to step from point to point as accurately and efficiently as possible (see Fig. 4.1).

## 4.2 Basic Stepping concepts and algorithms

More mathematically, the *initial value problem* reduces to solving the integral equation

$$\mathbf{y} = \mathbf{y}_0 + \int_{t_0}^{t} \mathbf{f}(\tau, \mathbf{y}(\tau))d\tau \tag{4.2.1}$$

Actually, the numerical problem only requires evaluating the integral in (4.2.1) for a single stepsize $h$ (i.e. for $t = t_0 + h$), because in initial value problems, if we can solve the problem for a single step, we can just restart the problem from the end-point and repeat ourselves till we get to where we want to go. Thus the problem reduces to finding a cheap but accurate approximation for the integral

$$k = \int_{0}^{h} f(\tau, y(\tau))d\tau \tag{4.2.2}$$

Now the cheapest (but least accurate approximation is an *Euler Step* where we approximate the integral just using the value of $f$ at time 0. i.e. $k \sim hf(t_0, y_0)$ so that our approximation becomes

$$y(t_0 + h) = y_0 + hf(t_0, y_0) \tag{4.2.3}$$

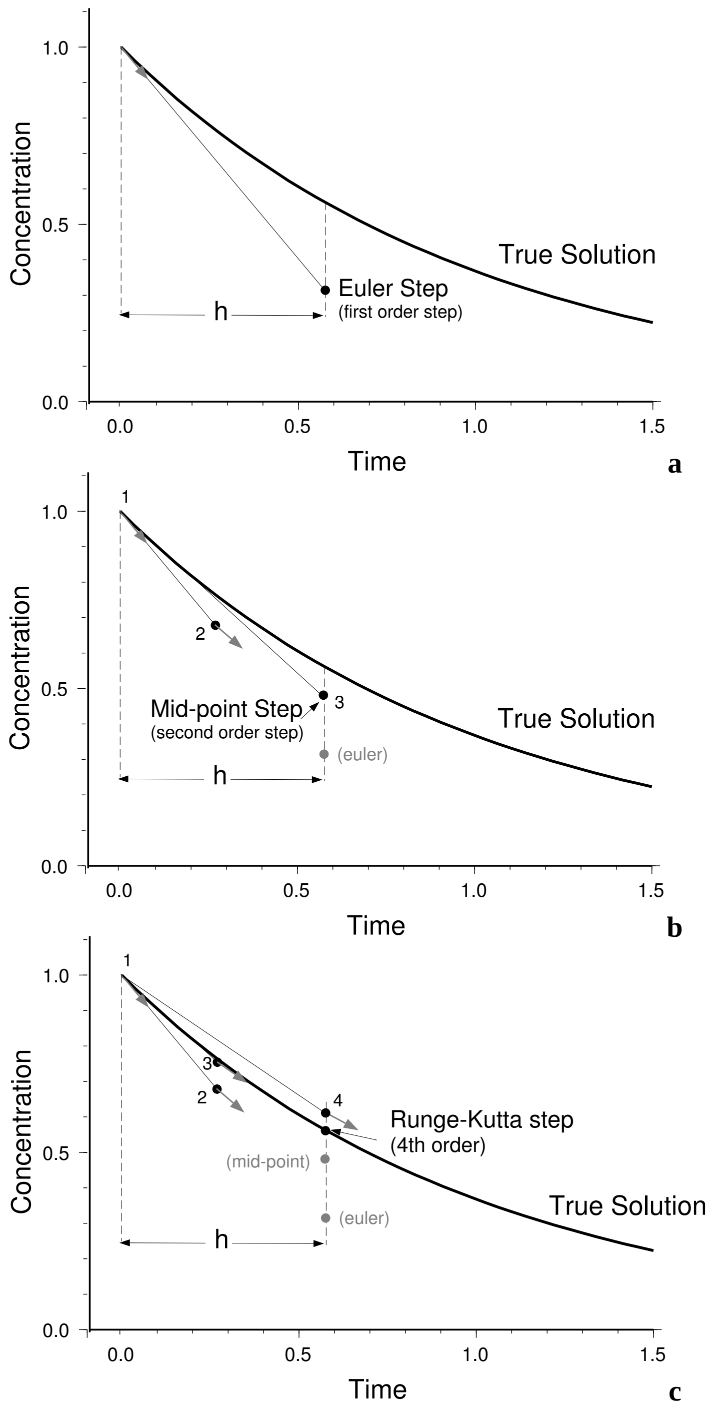


Figure 4.2: Simple stepping schemes of increasing order. (*a*) Euler Step: just uses derivative information at $t = 0$ and extrapolates linearly. Highly inaccurate for functions with curvature. (*b*) Mid-point scheme: 2nd order accurate stepper, uses two derivative evaluations to gain information on curvature. (*c*) Standard 4th order Runge-Kutta scheme: uses 4 function evaluations for high order accuracy.

Unfortunately, if we compare that to the Taylor series expansion of $y$

$$y(t_0 + h) = y(t_0) + hf(t_0, y_0) + \frac{h^2}{2}\frac{d^2y}{dt^2} + \ldots \tag{4.2.4}$$

We see that if the function has any curvature, even a moderate step-size $h$ will cause an error that is only of order $h$ smaller than our approximation (which is why the $O(h^2)$ term is known as a *first order error*). More physically, inspection of Fig. 4.2a shows that an Euler step is simply a linear extrapolation of our derivative at $t_0$. Clearly, if our true solution is anything but a straight line, we need to take a very small step to stay on course.

So what to do? Qualitatively, it would seem that the more information we have about our function the better we can approximate the integral Eq. (4.2.2). For example, we ought to be able to get some curvature information with two function evaluations and the *mid-point method* does just that. Fig. 4.2b shows that the mid-point first takes an Euler step with step-size $h/2$ then uses the derivative information at the midpoint. This algorithm can be written

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h/2, y_n + k_1/2) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{4.2.5}$$

which is a second order method.

It follows that if we gain more information about our function we can get even higher order schemes[1]. A classic is the *4th-order Runge Kutta* scheme which gets us 4th order accuracy with only 4 function evaluations (which turns out to be the turning point for RK schemes). This stepping scheme is illustrated in Fig. 4.2c and can be written

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h/2, y_n + k_1/2) \\ k_3 &= hf(t_n + h/2, y_n + k_2/2) \\ k_4 &= hf(t_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5) \end{aligned} \tag{4.2.6}$$

(If you're really wondering, this scheme is related to Simpson's rule for integration of quadratic functions).

## 4.3 Getting clever: adaptive stepping

Equation (4.2.6) is a workable scheme that will get us from point $A$ to $B$ in a single step. The question is still "how big a step can we take?". Any good ODE integrator worth its salt will sort this out for you by using *adaptive stepping*. The basic idea

[1]Beware: higher order does not always mean higher accuracy unless your function is well approximated by high order polynomials

is that if we know the order of error as a function of step-size for any stepping scheme (and can monitor that error), we can adjust the step-size to keep the error within some tolerance. One simple approach is to use *step-doubling* which first takes a big step of size $h$ and then starts from the same point but takes two half-steps of size $h/2$. In a perfect world, the two trial solutions should be the same but the actual truncation error on the big step is of order $h^5$ while the error on the two half-steps is of order $2(h/2)^5$. Comparing the difference between the two results gives a measure of the *relative truncation error* $\Delta$ which should still be

$$\Delta \propto h^5 \tag{4.3.1}$$

Thus, if our measured truncation error for a trial step $h_0$ is $\Delta_0$, and we would really like our error to be some other $\Delta_1$ then we should adjust our step-size to a $h_1$ such that

$$h_1 = h_0 \left|\frac{\Delta_1}{\Delta_0}\right|^{1/5} \tag{4.3.2}$$

Actually as Numerical Recipes discusses in detail, there are some slightly more practical ways to choose the new step-size but the idea is the same. All that's left now is to somehow specify the desired accuracy $\Delta_1$. In the Numerical recipes routines, this is done by specifying a tolerance `eps` and some scale `yscale(i)` for each variable `y(i)` such that

$$\Delta_1 = \texttt{eps} \times \texttt{yscale(i)} \tag{4.3.3}$$

Moreover, they suggest that a handy trick for getting constant fractional error is to set `yscale(i)=abs(y(i))+abs(h*dydt(i))`, which protects you from zero-crossings. The result of this approach, is to make the scheme track the fastest changing variable (which is as it should be).

**Beyond Step-doubling: embedded Runge-Kutta** Step-doubling is rough and ready but costs eleven function evaluations for every good step $h$[2]. A more efficient but inscrutable technique uses *embedded Runge-Kutta* schemes such as the *5th-order Runge-Kutta Cash-Karp* scheme implemented in Numerical recipes. This scheme is 5th-order and uses six function evaluations per step. The peculiar part though is that there exists another combination of these 6 evaluations that is 4th order. Thus with 6 evaluations you get your cake and eat it too, i.e. you get your 5th order accuracy along with a measure of the truncation error. For rough and ready jobs, these schemes are quite robust and quick.

**A note on the NumRec routines** Numerical Recipes (Press *et al.*, 1992) provides a useful set of ODE integrators which are easily adapted to most problems. In particular, they have taken a very modular approach to coding and have separated the routines into 3 levels. At the lowest level is the user supplied routine `derivs(t,y,dydt)` such as

[2] that's 3 steps with 4 evaluations per step minus 1 because they start in the same place

```
c***********************************************************************
c decay1:  subroutine that returns the rate of decay of a radioactive
c   substance as a function of  concentration (and time)
c   c is concentration
c   dcdt is the change in c with time
c   t is time
c   here, dcdt= -c
c***********************************************************************

      subroutine decay1(t,c,dcdt)
      implicit none
      real t, c, dcdt

      dcdt= -c

      return
      end
```

which returns an array of values `dydt(i)` given a time `t` and a state-vector `y(i)`. Unfortunately, if your function requires additional parameters to evaluate the derivatives, these need to passed into the routine using *common blocks* (in Fortran or global variables in C). Ugly but necessary. Some examples are given in the problem set. Still, for the extra coding headache, these routines provide quite good flexibility.

Above the `derivs` level is the *algorithm* which takes one step without regard to accuracy. In Numerical Recipes, the standard RK algorithm Eq. (4.2.6) is implemented in `rk4` while the Runge-Kutta Cash-Karp scheme is in `rkck`. Above the algorithmic level comes quality control which adjusts the step size until the desired accuracy is reached. This *stepper* routine (e.g. `rkqs`) job is to try to take the largest step possible while remaining within the described tolerance. This is where the adaptive stepping is implemented and the routine will keep adjusting the step size to get to its goal. Finally, there are the *driver* routines (e.g. `odeint` or my version of it `odeintnc`). which keep track of the progress, say when to stop and possibly store intermediate information. In general, most user modifications go in the driver routines and they should be tailored to your problem.

## 4.4 Beyond Runge-Kutta: Bulirsch-Stoer methods

Embedded Runge-Kutta with adaptive step-size control is a pretty solid ODE integrator that will get you through most problems without any trouble. In particular, if you have coarsely gridded data or the function is cheap to evaluate or you have internal singularities that need to be stepped carefully around then these schemes are just peachy. But of course you can always do better. One useful approach for problems that require high accuracy is the *Bulirsch-Stoer* method which actually approximates an infinite order scheme yet allows you to take very large steps. Sound too good to be true? Then check out Fig. 4.3.

The basic idea is that if we have a scheme that allows us to take a large step $H$ with a variable number of smaller substeps, then as we increase the number of steps we form increasingly better approximations to the end solutions. Moreover, if the progression of approximate solutions is smooth and well behaved, we can actually *extrapolate* our series to the limit of taking an infinite number of sub-steps (i.e. if

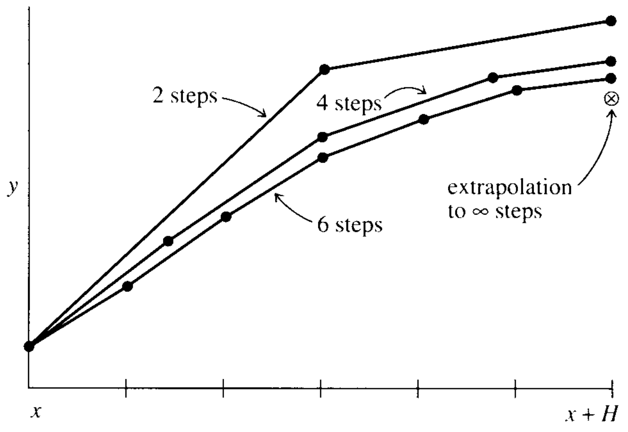


Figure 4.3: Richardson extrapolation as used in the Bulirsch-Stoer method. A large interval $H$ is spanned by different sequences of finer and finer substeps. Their results are extrapolated to an answer that is supposed to correspond to infinitely fine substeps. In the Bulirsch-Stoer method, the integrations are done by the modified midpoint method, and the extrapolation technique is rational function or polynomial extrapolation. (figure and caption from Numerical Recipes, 2nd ed., p. 719)

$h = 0$). The trick in the Bulirsch-Stoer method is to use a *modified mid-point* scheme to take the sub-steps. This scheme is an example of a *leapfrog or centered difference scheme* where we take an Euler step for the first and last points but use a mid-point method to take us from point $y_{n-1}$ to $y_{n+1}$ using the derivative information at $y_n$ (easier to show than to say). As discussed in Numerical Recipes, the important feature of the modified mid-point method is that the relative truncation error between successive trials with increasing numbers of substeps have only even powers of step-size so you tend to get two-orders of magnitude error reduction for only a few extra function calls. With the modular form of the Numerical recipes routines, it is just as easy to use BS methods as RK methods, you simply replace the algorithm routine with `mmid` and the stepper routine with `bsstep`. These routine implement a more complicated form of adaptive step-size control that is still based on the relationships between step size and truncation error inherent in the method. You don't really have to worry about how it works in detail. What you have to worry about is when not to use it. Basically, these schemes assume that function behaves very smoothly and has no internal singularities. All of the grace of this method will be lost if your function is rough or has a spike somewhere in the middle of it. For these problems go back to the Runge Kutta schemes.

## 4.5 Even more ODE's: stiff equations

The routines so far have never failed me but that may have more to do with the type of problems I solve where either the variables all change at similar rates or I have lots of time and computer resources. Sometimes, however, you may run into the problem of *stiff equations* where one or more of your variables change at a rate that is much faster than the thing you are interested in. An adaptive stepper routine could catch this but it forces your problem to evolve on the time scale of your most annoying variable rather than your most interesting one. Numerical recipes supplies a specific example (and we will see this phenomena in other situations as well). One standard approach is to use *implicit differencing schemes* which are more stable but more difficult to solve. Numerical Recipes discusses them in detail and provides routines for stiff equations.