Chapter 16. Integration of Ordinary Differential Equations

16.0 Introduction

Problems involving ordinary differential equations (ODEs) can always be reduced to the study of sets of first-order differential equations. For example the second-order equation

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x) \tag{16.0.1}$$

can be rewritten as two first-order equations

$$\begin{aligned} \frac{dy}{dx} &= z(x) \\ \frac{dz}{dx} &= r(x) - q(x)z(x) \end{aligned} \tag{16.0.2}$$

where z is a new variable. This exemplifies the procedure for an arbitrary ODE. The usual choice for the new variables is to let them be just derivatives of each other (and of the original variable). Occasionally, it is useful to incorporate into their definition some other factors in the equation, or some powers of the independent variable, for the purpose of mitigating singular behavior that could result in overflows or increased roundoff error. Let common sense be your guide: If you find that the original variables are smooth in a solution, while your auxiliary variables are doing crazy things, then figure out why and choose different auxiliary variables.

The generic problem in ordinary differential equations is thus reduced to the study of a set of N coupled *first-order* differential equations for the functions $y_i,\ i = 1, 2, \ldots, N$, having the general form

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \ldots, y_N), \qquad i = 1, \ldots, N \tag{16.0.3}$$

where the functions f_i on the right-hand side are known.

A problem involving ODEs is not completely specified by its equations. Even more crucial in determining how to attack the problem numerically is the nature of the problem's boundary conditions. Boundary conditions are algebraic conditions on the values of the functions y_i in (16.0.3). In general they can be satisfied at discrete specified points, but do not hold between those points, i.e., are not preserved automatically by the differential equations. Boundary conditions can be as simple as requiring that certain variables have certain numerical values, or as complicated as a set of nonlinear algebraic equations among the variables.

Usually, it is the nature of the boundary conditions that determines which numerical methods will be feasible. Boundary conditions divide into two broad categories.

- In *initial value problems* all the y_i are given at some starting value x_s, and it is desired to find the y_i's at some final point x_f, or at some discrete list of points (for example, at tabulated intervals).
- In *two-point boundary value problems*, on the other hand, boundary conditions are specified at more than one x. Typically, some of the conditions will be specified at x_s and the remainder at x_f.

This chapter will consider exclusively the initial value problem, deferring two-point boundary value problems, which are generally more difficult, to Chapter 17.

The underlying idea of any routine for solving the initial value problem is always this: Rewrite the dy's and dx's in (16.0.3) as finite steps Δy and Δx, and multiply the equations by Δx. This gives algebraic formulas for the change in the functions when the independent variable x is "stepped" by one "stepsize" Δx. In the limit of making the stepsize very small, a good approximation to the underlying differential equation is achieved. Literal implementation of this procedure results in *Euler's method* (16.1.1, below), which is, however, *not* recommended for any practical use. Euler's method is conceptually important, however; one way or another, practical methods all come down to this same idea: Add small increments to your functions corresponding to derivatives (right-hand sides of the equations) multiplied by stepsizes.

In this chapter we consider three major types of practical numerical methods for solving initial value problems for ODEs:

- Runge-Kutta methods
- Richardson extrapolation and its particular implementation as the Bulirsch-Stoer method
- predictor-corrector methods.

A brief description of each of these types follows.

1. *Runge-Kutta* methods propagate a solution over an interval by combining the information from several Euler-style steps (each involving one evaluation of the right-hand f's), and then using the information obtained to match a Taylor series expansion up to some higher order.

2. *Richardson extrapolation* uses the powerful idea of extrapolating a computed result to the value that *would* have been obtained if the stepsize had been very much smaller than it actually was. In particular, extrapolation to zero stepsize is the desired goal. The first practical ODE integrator that implemented this idea was developed by Bulirsch and Stoer, and so extrapolation methods are often called Bulirsch-Stoer methods.

3. *Predictor-corrector* methods store the solution along the way, and use those results to extrapolate the solution one step advanced; they then correct the extrapolation using derivative information at the new point. These are best for very smooth functions.

Runge-Kutta is what you use when (i) you don't know any better, or (ii) you have an intransigent problem where Bulirsch-Stoer is failing, or (iii) you have a trivial problem where computational efficiency is of no concern. Runge-Kutta succeeds virtually always; but it is not usually fastest, except when evaluating f_i is cheap and moderate accuracy ($\lesssim 10^{-5}$) is required. Predictor-corrector methods, since they use past information, are somewhat more difficult to start up, but, for many smooth problems, they are computationally more efficient than Runge-Kutta. In recent years Bulirsch-Stoer has been replacing predictor-corrector in many applications, but it is too soon to say that predictor-corrector is dominated in all cases. However, it appears that only rather sophisticated predictor-corrector routines are competitive. Accordingly, we have chosen *not* to give an implementation of predictor-corrector in this book. We discuss predictor-corrector further in §16.7, so that you can use a canned routine should you encounter a suitable problem. In our experience, the relatively simple Runge-Kutta and Bulirsch-Stoer routines we give are adequate for most problems.

Each of the three types of methods can be organized to monitor internal consistency. This allows numerical errors which are inevitably introduced into the solution to be controlled by automatic, (*adaptive*) changing of the fundamental stepsize. We always recommend that adaptive stepsize control be implemented, and we will do so below.

In general, all three types of methods can be applied to any initial value problem. Each comes with its own set of debits and credits that must be understood before it is used.

We have organized the routines in this chapter into three nested levels. The lowest or "nitty-gritty" level is the piece we call the *algorithm* routine. This implements the basic formulas of the method, starts with dependent variables y_i at x, and calculates new values of the dependent variables at the value $x + h$. The algorithm routine also yields up some information about the quality of the solution after the step. The routine is dumb, however, and it is unable to make any adaptive decision about whether the solution is of acceptable quality or not.

That quality-control decision we encode in a *stepper* routine. The stepper routine calls the algorithm routine. It may reject the result, set a smaller stepsize, and call the algorithm routine again, until compatibility with a predetermined accuracy criterion has been achieved. The stepper's fundamental task is to take the largest stepsize consistent with specified performance. Only when this is accomplished does the true power of an algorithm come to light.

Above the stepper is the *driver* routine, which starts and stops the integration, stores intermediate results, and generally acts as an interface with the user. There is nothing at all canonical about our driver routines. You should consider them to be examples, and you can customize them for your particular application.

Of the routines that follow, `rk4`, `rkck`, `mmid`, `stoerm`, and `simpr` are algorithm routines; `rkqs`, `bsstep`, `stiff`, and `stifbs` are steppers; `rkdumb` and `odeint` are drivers.

Section 16.6 of this chapter treats the subject of *stiff equations*, relevant both to ordinary differential equations and also to partial differential equations (Chapter 19).

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall).

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 5.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 7.

Lambert, J. 1973, *Computational Methods in Ordinary Differential Equations* (New York: Wiley).

Lapidus, L., and Seinfeld, J. 1971, *Numerical Solution of Ordinary Differential Equations* (New York: Academic Press).

16.1 Runge-Kutta Method

The formula for the Euler method is

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{16.1.1}$$

which advances a solution from x_n to $x_{n+1} \equiv x_n + h$. The formula is unsymmetrical: It advances the solution through an interval h, but uses derivative information only at the beginning of that interval (see Figure 16.1.1). That means (and you can verify by expansion in power series) that the step's error is only one power of h smaller than the correction, i.e $O(h^2)$ added to (16.1.1).

There are several reasons that Euler's method is not recommended for practical use, among them, (i) the method is not very accurate when compared to other, fancier, methods run at the equivalent stepsize, and (ii) neither is it very stable (see §16.6 below).

Consider, however, the use of a step like (16.1.1) to take a "trial" step to the midpoint of the interval. Then use the value of both x and y at that midpoint to compute the "real" step across the whole interval. Figure 16.1.2 illustrates the idea. In equations,

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{16.1.2}$$

As indicated in the error term, this symmetrization cancels out the first-order error term, making the method *second order*. [A method is conventionally called nth order if its error term is $O(h^{n+1})$.] In fact, (16.1.2) is called the *second-order Runge-Kutta* or *midpoint* method.

We needn't stop there. There are many ways to evaluate the right-hand side $f(x, y)$ that all agree to first order, but that have different coefficients of higher-order error terms. Adding up the right combination of these, we can eliminate the error terms order by order. That is the basic idea of the Runge-Kutta method. Abramowitz and Stegun [1], and Gear [2], give various specific formulas that derive from this basic