NUMERICAL RECIPES

The Art of Scientific Computing

Third Edition

1.1 Error, Accuracy, and Stability

Computers store numbers not with infinite precision but rather in some approximation that can be packed into a fixed number of *bits* (binary digits) or *bytes* (groups of 8 bits). Almost all computers allow the programmer a choice among several different such *representations* or *data types*. Data types can differ in the number of bits utilized (the *wordlength*), but also in the more fundamental respect of whether the stored number is represented in *fixed-point* (like `int`) or *floating-point* (like `float` or `double`) format.

A number in integer representation is exact. Arithmetic between numbers in integer representation is also exact, with the provisos that (i) the answer is not outside the range of (usually, signed) integers that can be represented, and (ii) that division is interpreted as producing an integer result, throwing away any integer remainder.

1.1.1 Floating-Point Representation

In a floating-point representation, a number is represented internally by a sign bit S (interpreted as plus or minus), an exact integer exponent E, and an exactly represented binary mantissa M. Taken together these represent the number

$$S \times M \times b^{E-e} \tag{1.1.1}$$

where b is the base of the representation ($b = 2$ almost always), and e is the *bias* of the exponent, a fixed integer constant for any given machine and representation.

	S	E	F	Value
`float`	any	1–254	any	$(-1)^S \times 2^{E-127} \times 1.F$
	any	0	nonzero	$(-1)^S \times 2^{-126} \times 0.F*$
	0	0	0	$+0.0$
	1	0	0	-0.0
	0	255	0	$+\infty$
	1	255	0	$-\infty$
	any	255	nonzero	`NaN`
`double`	any	1–2046	any	$(-1)^S \times 2^{E-1023} \times 1.F$
	any	0	nonzero	$(-1)^S \times 2^{-1022} \times 0.F*$
	0	0	0	$+0.0$
	1	0	0	-0.0
	0	2047	0	$+\infty$
	1	2047	0	$-\infty$
	any	2047	nonzero	`NaN`
*unnormalized values				

Several floating-point bit patterns can in principle represent the same number. If $b = 2$, for example, a mantissa with leading (high-order) zero bits can be left-shifted, i.e., multiplied by a power of 2, if the exponent is decreased by a compensating amount. Bit patterns that are "as left-shifted as they can be" are termed *normalized*.

Virtually all modern processors share the same floating-point data representations, namely those specified in IEEE Standard 754-1985 [1]. (For some discussion of nonstandard processors, see §22.2.) For 32-bit `float` values, the exponent is represented in 8 bits (with $e = 127$), the mantissa in 23; for 64-bit `double` values, the exponent is 11 bits (with $e = 1023$), the mantissa, 52. An additional trick is used for the mantissa for most nonzero floating values: Since the high-order bit of a properly normalized mantissa is *always* one, the stored mantissa bits are viewed as being preceded by a "phantom" bit with the value 1. In other words, the mantissa M has the numerical value $1.F$, where F (called the *fraction*) consists of the bits (23 or 52 in number) that are actually stored. This trick gains a little "bit" of precision.

Here are some examples of IEEE 754 representations of `double` values:

$$\begin{aligned}
&\texttt{0 01111111111 0000} (+ \text{ 48 more zeros}) = +1 \times 2^{1023-1023} \times 1.0_2 = 1. \\
&\texttt{1 01111111111 0000} (+ \text{ 48 more zeros}) = -1 \times 2^{1023-1023} \times 1.0_2 = -1. \\
&\texttt{0 01111111111 1000} (+ \text{ 48 more zeros}) = +1 \times 2^{1023-1023} \times 1.1_2 = 1.5 \\
&\texttt{0 10000000000 0000} (+ \text{ 48 more zeros}) = +1 \times 2^{1024-1023} \times 1.0_2 = 2. \\
&\texttt{0 10000000001 1010} (+ \text{ 48 more zeros}) = +1 \times 2^{1025-1023} \times 1.1010_2 = 6.5
\end{aligned} \tag{1.1.2}$$

You can examine the representation of any value by code like this:

```
union Udoub {
    double d;
    unsigned char c[8];
};

void main() {
    Udoub u;
    u.d = 6.5;
    for (int i=7;i>=0;i--) printf("%02x",u.c[i]);
    printf("\n");
}
```

This is C, and deprecated style, but it will work. On most processors, including Intel Pentium and successors, you'll get the printed result `401a000000000000`, which (writing out each hex digit as four binary digits) is the last line in equation (1.1.2). If you get the bytes (groups of two hex digits) in reverse order, then your processor is *big-endian* instead of *little-endian*: The IEEE 754 standard does not specify (or care) in which order the bytes in a floating-point value are stored.

The IEEE 754 standard includes representations of positive and negative infinity, positive and negative zero (treated as computationally equivalent, of course), and also `NaN` ("not a number"). The table on the previous page gives details of how these are represented.

The reason for representing some *unnormalized* values, as shown in the table, is to make "underflow to zero" more graceful. For a sequence of smaller and smaller values, after you pass the smallest normalizable value (with magnitude 2^{-127} or 2^{-1023}; see table), you start right-shifting the leading bit of the mantissa. Although

you gradually lose precision, you don't actually underflow to zero until 23 or 52 bits later.

When a routine needs to know properties of the floating-point representation, it can reference the `numeric_limits` class, which is part of the C++ Standard Library. For example, `numeric_limits<double>::min()` returns the smallest normalized `double` value, usually $2^{-1022} \approx 2.23 \times 10^{-308}$. For more on this, see §22.2.

1.1.2 Roundoff Error

Arithmetic among numbers in floating-point representation is not exact, even if the operands happen to be exactly represented (i.e., have exact values in the form of equation 1.1.1). For example, two floating numbers are added by first right-shifting (dividing by two) the mantissa of the smaller (in magnitude) one and simultaneously increasing its exponent until the two operands have the same exponent. Low-order (least significant) bits of the smaller operand are lost by this shifting. If the two operands differ too greatly in magnitude, then the smaller operand is effectively replaced by zero, since it is right-shifted to oblivion.

The smallest (in magnitude) floating-point number that, when added to the floating-point number 1.0, produces a floating-point result different from 1.0 is termed the *machine accuracy* ϵ_m. IEEE 754 standard `float` has ϵ_m about 1.19×10^{-7}, while `double` has about 2.22×10^{-16}. Values like this are accessible as, e.g., `numeric _limits <double>::epsilon()`. (A more detailed discussion of machine characteristics is in §22.2.) Roughly speaking, the machine accuracy ϵ_m is the fractional accuracy to which floating-point numbers are represented, corresponding to a change of one in the least significant bit of the mantissa. Pretty much any arithmetic operation among floating numbers should be thought of as introducing an additional fractional error of at least ϵ_m. This type of error is called *roundoff error*.

It is important to understand that ϵ_m is not the smallest floating-point number that can be represented on a machine. *That* number depends on how many bits there are in the exponent, while ϵ_m depends on how many bits there are in the mantissa.

Roundoff errors accumulate with increasing amounts of calculation. If, in the course of obtaining a calculated value, you perform N such arithmetic operations, you *might* be so lucky as to have a total roundoff error on the order of $\sqrt{N}\epsilon_m$, if the roundoff errors come in randomly up or down. (The square root comes from a random-walk.) However, this estimate can be very badly off the mark for two reasons:

(1) It very frequently happens that the regularities of your calculation, or the peculiarities of your computer, cause the roundoff errors to accumulate preferentially in one direction. In this case the total will be of order $N\epsilon_m$.

(2) Some especially unfavorable occurrences can vastly increase the roundoff error of single operations. Generally these can be traced to the subtraction of two very nearly equal numbers, giving a result whose only significant bits are those (few) low-order ones in which the operands differed. You might think that such a "coincidental" subtraction is unlikely to occur. Not always so. Some mathematical expressions magnify its probability of occurrence tremendously. For example, in the familiar formula for the solution of a quadratic equation,

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{1.1.3}$$

the addition becomes delicate and roundoff-prone whenever $b > 0$ and $|ac| \ll b^2$. (In §5.6 we will learn how to avoid the problem in this particular case.)

1.1.3 Truncation Error

Roundoff error is a characteristic of computer hardware. There is another, different, kind of error that is a characteristic of the program or algorithm used, independent of the hardware on which the program is executed. Many numerical algorithms compute "discrete" approximations to some desired "continuous" quantity. For example, an integral is evaluated numerically by computing a function at a discrete set of points, rather than at "every" point. Or, a function may be evaluated by summing a finite number of leading terms in its infinite series, rather than all infinity terms. In cases like this, there is an adjustable parameter, e.g., the number of points or of terms, such that the "true" answer is obtained only when that parameter goes to infinity. Any practical calculation is done with a finite, but sufficiently large, choice of that parameter.

The discrepancy between the true answer and the answer obtained in a practical calculation is called the *truncation error*. Truncation error would persist even on a hypothetical, "perfect" computer that had an infinitely accurate representation and no roundoff error. As a general rule there is not much that a programmer can do about roundoff error, other than to choose algorithms that do not magnify it unnecessarily (see discussion of "stability" below). Truncation error, on the other hand, is entirely under the programmer's control. In fact, it is only a slight exaggeration to say that clever minimization of truncation error is practically the entire content of the field of numerical analysis!

Most of the time, truncation error and roundoff error do not strongly interact with one another. A calculation can be imagined as having, first, the truncation error that it would have if run on an infinite-precision computer, "plus" the roundoff error associated with the number of operations performed.

1.1.4 Stability

Sometimes an otherwise attractive method can be *unstable*. This means that any roundoff error that becomes "mixed into" the calculation at an early stage is successively magnified until it comes to swamp the true answer. An unstable method would be useful on a hypothetical, perfect computer; but in this imperfect world it is necessary for us to require that algorithms be stable — or if unstable that we use them with great caution.

Here is a simple, if somewhat artificial, example of an unstable algorithm: Suppose that it is desired to calculate all integer powers of the so-called "Golden Mean," the number given by

$$\phi \equiv \frac{\sqrt{5}-1}{2} \approx 0.61803398 \tag{1.1.4}$$

It turns out (you can easily verify) that the powers ϕ^n satisfy a simple recursion relation,

$$\phi^{n+1} = \phi^{n-1} - \phi^n \tag{1.1.5}$$

Thus, knowing the first two values $\phi^0 = 1$ and $\phi^1 = 0.61803398$, we can successively apply (1.1.5) performing only a single subtraction, rather than a slower

multiplication by ϕ, at each stage.

Unfortunately, the recurrence (1.1.5) also has *another* solution, namely the value $-\frac{1}{2}(\sqrt{5}+1)$. Since the recurrence is linear, and since this undesired solution has magnitude greater than unity, any small admixture of it introduced by roundoff errors will grow exponentially. On a typical machine, using a 32-bit `float`, (1.1.5) starts to give completely wrong answers by about $n = 16$, at which point ϕ^n is down to only 10^{-4}. The recurrence (1.1.5) is *unstable* and cannot be used for the purpose stated.

We will encounter the question of stability in many more sophisticated guises later in this book.

CITED REFERENCES AND FURTHER READING:

IEEE, 1985, *ANSI/IEEE Std 754–1985: IEEE Standard for Binary Floating-Point Numbers* (New York: IEEE).[1]

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 1.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 2.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §1.3.

Wilkinson, J.H. 1964, *Rounding Errors in Algebraic Processes* (Englewood Cliffs, NJ: Prentice-Hall).

1.2 C Family Syntax

Not only C++, but also Java, C#, and (to varying degrees) other computer languages, share a lot of small-scale syntax with the older C language [1]. By small scale, we mean operations on built-in types, simple expressions, control structures, and the like. In this section, we review some of the basics, give some hints on good programming, and mention some of our conventions and habits.

1.2.1 Operators

A first piece of advice might seem superfluous if it were not so often ignored: You should learn all the C operators and their precedence and associativity rules. You might not yourself want to write

```
n << 1 | 1
```

as a synonym for `2*n+1` (for positive integer `n`), but you definitely do need to be able to see at a glance that

```
n << 1 + 1
```

is not at all the same thing! Please study the table on the next page while you brush your teeth every night. While the occasional set of unnecessary parentheses, for clarity, is hardly a sin, code that is habitually overparenthesized is annoying and hard to read.

CHAPTER 17

Integration of Ordinary Differential Equations

17.0 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

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x) \tag{17.0.1}$$

can be rewritten as two first-order equations,

$$\begin{aligned} \frac{dy}{dx} &= z(x) \\ \frac{dz}{dx} &= r(x) - q(x)z(x) \end{aligned} \tag{17.0.2}$$

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, $i = 0, 1, \ldots, N-1$, having the general form

$$\frac{dy_i(x)}{dx} = f_i(x, y_0, \ldots, y_{N-1}), \qquad i = 0, \ldots, N-1 \tag{17.0.3}$$

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 (17.0.3). 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 18.

The underlying idea of any routine for solving the initial value problem is always this: Rewrite the dy's and dx's in (17.0.3) 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* (equation 17.1.1, 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, also known as multistep 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 or *multistep 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 used to be what you used when (i) you didn't know any better, or (ii) you had an intransigent problem where Bulirsch-Stoer was failing, or (iii) you had a trivial problem where computational efficiency was of no concern. However, advances in Runge-Kutta methods, particularly the development of higher-order methods, have made Runge-Kutta competitive with the other methods in many cases. Runge-Kutta succeeds virtually always; it is usually the fastest method when evaluating f_i is cheap and the accuracy requirement is not ultra-stringent ($\lesssim 10^{-10}$), or in general when moderate accuracy ($\lesssim 10^{-5}$) is required. Predictor-corrector methods have a relatively high overhead and so come into their own only when evaluating f_i is expensive. However, 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 §17.6, so that you can use a packaged routine knowledgeably 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.

Section 17.5 of this chapter treats the subject of *stiff equations*, relevant both to ordinary differential equations and also to partial differential equations (Chapter 20).

17.0.1 Organization of the Routines in This Chapter

We have organized the routines in this chapter into three nested levels, enabling modularity and sharing common code wherever possible.

The highest level is the *driver* object, which starts and stops the integration, stores intermediate results, and generally acts as an interface with the user. There is nothing canonical about our driver object, `Odeint`. You should consider it to be an example, and you can customize it for your particular application.

The next level down is a *stepper* object. The stepper oversees the actual incrementing of the independent variable x. It knows how to call the underlying *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.

All our steppers are derived from a base object called `StepperBase`: `StepperDopr5` and `StepperDopr853` (two Runge-Kutta routines), `StepperBS` and `StepperStoerm` (two Bulirsch-Stoer routines), and `StepperRoss` and `StepperSIE`

(for so-called stiff equations).

Standing apart from the stepper, but interacting with it at the same level, is an `Output` object. This is basically a container into which the stepper writes the output of the integration, but it has some intelligence of its own: It can save, or not save, intermediate results according to several different prescriptions that are specified by its constructor. In particular, it has the option to provide so-called dense output, that is, output at user-specified intermediate points without loss of efficiency.

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 some information about the quality of the solution after the step. The routine is dumb, however, in that it is unable to make any adaptive decision about whether the solution is of acceptable quality. Each algorithm routine is implemented as a member function `dy()` in its corresponding stepper object.

17.0.2 The Odeint Object

It is a real time saver to have a single high-level interface to what are otherwise quite diverse methods. We use the `Odeint` driver for a variety of problems, notably including garden-variety ODEs or sets of ODEs, and definite integrals (augmenting the methods of Chapter 4). The `Odeint` driver is templated on the stepper. This means that you can usually change from one ODE method to another in just a few keystrokes. For example, changing from the Dormand-Prince fifth-order Runge-Kutta method to Bulirsch-Stoer is as simple as changing the template parameter from `StepperDopr5` to `StepperBS`.

The `Odeint` constructor simply initializes a bunch of things, including a call to the stepper constructor. The meat is in the `integrate` routine, which repeatedly invokes the `step` routine of the stepper to advance the solution from x_1 to x_2. It also calls the functions of the `Output` object to save the results at appropriate points.

odeint.h

```
template<class Stepper>
struct Odeint {
```

Driver for ODE solvers with adaptive stepsize control. The template parameter should be one of the derived classes of `StepperBase` defining a particular integration algorithm.

```
    static const Int MAXSTP=50000;
    Doub EPS;
    Int nok;
    Int nbad;
    Int nvar;
    Doub x1,x2,hmin;
    bool dense;
    VecDoub y,dydx;
    VecDoub &ystart;
    Output &out;
    typename Stepper::Dtype &derivs;
    Stepper s;
    Int nstp;
    Doub x,h;
    Odeint(VecDoub_IO &ystartt,const Doub xx1,const Doub xx2,
        const Doub atol,const Doub rtol,const Doub h1,
        const Doub hminn,Output &outt,typename Stepper::Dtype &derivss);
```

Take at most `MAXSTP` steps.

`true` if dense output requested by `out`.

Get the type of `derivs` from the stepper.

Constructor sets everything up. The routine integrates starting values `ystart[0..nvar-1]` from `xx1` to `xx2` with absolute tolerance `atol` and relative tolerance `rtol`. The quantity `h1` should be set as a guessed first stepsize, `hmin` as the minimum allowed stepsize (can be zero). An `Output` object `out` should be input to control the saving of intermediate values.

On output, nok and nbad are the number of good and bad (but retried and fixed) steps taken, and ystart is replaced by values at the end of the integration interval. derivs is the user-supplied routine (function or functor) for calculating the right-hand side derivative.

```
    void integrate();                                   Does the actual integration.
};

template<class Stepper>
Odeint<Stepper>::Odeint(VecDoub_IO &ystartt, const Doub xx1, const Doub xx2,
    const Doub atol, const Doub rtol, const Doub h1, const Doub hminn,
    Output &outt,typename Stepper::Dtype &derivss) : nvar(ystartt.size()),
    y(nvar),dydx(nvar),ystart(ystartt),x(xx1),nok(0),nbad(0),
    x1(xx1),x2(xx2),hmin(hminn),dense(outt.dense),out(outt),derivs(derivss),
    s(y,dydx,x,atol,rtol,dense) {
    EPS=numeric_limits<Doub>::epsilon();
    h=SIGN(h1,x2-x1);
    for (Int i=0;i<nvar;i++) y[i]=ystart[i];
    out.init(s.neqn,x1,x2);
}

template<class Stepper>
void Odeint<Stepper>::integrate() {
    derivs(x,y,dydx);
    if (dense)                                          Store initial values.
        out.out(-1,x,y,s,h);
    else
        out.save(x,y);
    for (nstp=0;nstp<MAXSTP;nstp++) {
        if ((x+h*1.0001-x2)*(x2-x1) > 0.0)
            h=x2-x;                                     If stepsize can overshoot, decrease.
        s.step(h,derivs);                               Take a step.
        if (s.hdid == h) ++nok; else ++nbad;
        if (dense)
            out.out(nstp,x,y,s,s.hdid);
        else
            out.save(x,y);
        if ((x-x2)*(x2-x1) >= 0.0) {                    Are we done?
            for (Int i=0;i<nvar;i++) ystart[i]=y[i];        Update ystart.
            if (out.kmax > 0 && abs(out.xsave[out.count-1]-x2) > 100.0*abs(x2)*EPS)
                out.save(x,y);                          Make sure last step gets saved.
            return;                                     Normal exit.
        }
        if (abs(s.hnext) <= hmin) throw("Step size too small in Odeint");
        h=s.hnext;
    }
    throw("Too many steps in routine Odeint");
}
```

The Odeint object doesn't know in advance which specific stepper object it will be instantiated with. It does, however, rely on the fact that the stepper object will be derived from, and thus have the methods in, this StepperBase object, which serves as the base class for all subsequent ODE algorithms in this chapter:

```
struct StepperBase {                                                    stepper.h
Base class for all ODE algorithms.
    Doub &x;
    Doub xold;                          Used for dense output.
    VecDoub &y,&dydx;
    Doub atol,rtol;
    bool dense;
    Doub hdid;                          Actual stepsize accomplished by the step routine.
    Doub hnext;                         Stepsize predicted by the controller for the next step.
```

```
    Doub EPS;
    Int n,neqn;                          neqn = n except for StepperStoerm.
    VecDoub yout,yerr;                   New value of y and error estimate.
    StepperBase(VecDoub_IO &yy, VecDoub_IO &dydxx, Doub &xx, const Doub atoll,
        const Doub rtoll, bool dens) : x(xx),y(yy),dydx(dydxx),atol(atoll),
        rtol(rtoll),dense(dens),n(y.size()),neqn(n),yout(n),yerr(n) {}
        Input to the constructor are the dependent variable vector y[0..n-1] and its derivative
        dydx[0..n-1] at the starting value of the independent variable x. Also input are the
        absolute and relative tolerances, atol and rtol, and the boolean dense, which is true
        if dense output is required.
};
```

17.0.3 The Output Object

Output is controlled by the various constructors in the `Output` structure. The default constructor, with no arguments, suppresses all output. The constructor with argument `nsave` provides *dense output* provided `nsave` > 0. This means output at values of x of your choosing, not necessarily the natural places that the stepper method would land. The output points are `nsave` $+ 1$ uniformly spaced points including `x1` and `x2`. If `nsave` ≤ 0, output is saved at every integration step, that is, only at the points where the stepper happens to land. While most of your needs should be met by these options, you should find it easy to modify `Output` for your particular application.

odeint.h

```
struct Output {
```
Structure for output from ODE solver such as odeint.
```
    Int kmax;                            Current capacity of storage arrays.
    Int nvar;
    Int nsave;                           Number of intervals to save at for dense output.
    bool dense;                          true if dense output requested.
    Int count;                           Number of values actually saved.
    Doub x1,x2,xout,dxout;
    VecDoub xsave;                       Results stored in the vector xsave[0..count-1] and the
    MatDoub ysave;                           matrix ysave[0..nvar-1][0..count-1].
    Output() : kmax(-1),dense(false),count(0) {}
```
Default constructor gives no output.
```
    Output(const Int nsavee) : kmax(500),nsave(nsavee),count(0),xsave(kmax) {
```
Constructor provides dense output at nsave equally spaced intervals. If nsave ≤ 0, output is saved only at the actual integration steps.
```
        dense = nsave > 0 ? true : false;
    }
    void init(const Int neqn, const Doub xlo, const Doub xhi) {
```
Called by Odeint constructor, which passes neqn, the number of equations, xlo, the starting point of the integration, and xhi, the ending point.
```
        nvar=neqn;
        if (kmax == -1) return;
        ysave.resize(nvar,kmax);
        if (dense) {
            x1=xlo;
            x2=xhi;
            xout=x1;
            dxout=(x2-x1)/nsave;
        }
    }
    void resize() {
```
Resize storage arrays by a factor of two, keeping saved data.
```
        Int kold=kmax;
        kmax *= 2;
        VecDoub tempvec(xsave);
```

```
        xsave.resize(kmax);
        for (Int k=0; k<kold; k++)
            xsave[k]=tempvec[k];
        MatDoub tempmat(ysave);
        ysave.resize(nvar,kmax);
        for (Int i=0; i<nvar; i++)
            for (Int k=0; k<kold; k++)
                ysave[i][k]=tempmat[i][k];
    }
    template <class Stepper>
    void save_dense(Stepper &s, const Doub xout, const Doub h) {
```

Invokes dense_out function of stepper routine to produce output at xout. Normally called by out rather than directly. Assumes that xout is between xold and xold+h, where the stepper must keep track of xold, the location of the previous step, and x=xold+h, the current step.

```
        if (count == kmax) resize();
        for (Int i=0;i<nvar;i++)
            ysave[i][count]=s.dense_out(i,xout,h);
        xsave[count++]=xout;
    }
    void save(const Doub x, VecDoub_I &y) {
```

Saves values of current x and y.

```
        if (kmax <= 0) return;
        if (count == kmax) resize();
        for (Int i=0;i<nvar;i++)
            ysave[i][count]=y[i];
        xsave[count++]=x;
    }
    template <class Stepper>
    void out(const Int nstp,const Doub x,VecDoub_I &y,Stepper &s,const Doub h) {
```

Typically called by Odeint to produce dense output. Input variables are nstp, the current step number, the current values of x and y, the stepper s, and the stepsize h. A call with nstp=-1 saves the initial values. The routine checks whether x is greater than the desired output point xout. If so, it calls save_dense.

```
        if (!dense)
            throw("dense output not set in Output!");
        if (nstp == -1) {
            save(x,y);
            xout += dxout;
        } else {
            while ((x-xout)*(x2-x1) > 0.0) {
                save_dense(s,xout,h);
                xout += dxout;
            }
        }
    }
};
```

17.0.4 A Quick-Start Example

Before we dive deep into the pros and cons of the different stepper types (the meat of this chapter), let's see how to code the solution of an actual problem. Suppose we want to solve Van der Pol's equation, which when written in first-order form is

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= [(1 - y_0^2)y_1 - y_0]/\epsilon \end{aligned} \tag{17.0.4}$$

First encapsulate (17.0.4) in a functor (see §1.3.3). Using a functor instead of a bare function gives you the opportunity to pass other information to the function,

such as the values of fixed parameters. Every stepper class in this chapter is accordingly templated on the type of the functor defining the right-hand side derivatives. For our example, the right-hand side functor looks like:

```
struct rhs_van {
    Doub eps;
    rhs_van(Doub epss) : eps(epss) {}
    void operator() (const Doub x, VecDoub_I &y, VecDoub_O &dydx) {
        dydx[0]= y[1];
        dydx[1]=((1.0-y[0]*y[0])*y[1]-y[0])/eps;
    }
};
```

The key thing is the line beginning `void operator()`: It *always* should have this form, with the definition of `dydx` following. Here we have chosen to specify ϵ as a parameter in the constructor so that the main program can easily pass a specific value to the right-hand side. Alternatively, you could have omitted the constructor, relying on the compiler-supplied default constructor, and hard-coded a value of ϵ in the routine. Note, of course, that there is nothing special about the name `rhs_van`.

We will integrate from 0 to 2 with initial conditions $y_0 = 2$, $y_1 = 0$ and with $\epsilon = 10^{-3}$. Then your main program will have declarations like the following:

```
const Int nvar=2;
const Doub atol=1.0e-3, rtol=atol, h1=0.01, hmin=0.0, x1=0.0, x2=2.0;
VecDoub ystart(nvar);
ystart[0]=2.0;
ystart[1]=0.0;
Output out(20);                          Dense output at 20 points plus x1.
rhs_van d(1.0e-3);                       Declare d as a rhs_van object.
Odeint<StepperDopr5<rhs_van> > ode(ystart,x1,x2,atol,rtol,h1,hmin,out,d);
ode.integrate();
```

Note how the `Odeint` object is templated on the stepper, which in turn is templated on the derivative object, `rhs_van` in this case. The space between the two closing angle brackets is necessary; otherwise the compiler parses >> as the right-shift operator!

The number of good steps taken is available in `ode.nok` and the number of bad steps in `ode.nbad`. The output, which is equally spaced, can be printed by statements like

```
for (Int i=0;i<out.count;i++)
    cout << out.xsave[i] << " " << out.ysave[0][i] << " " <<
        out.ysave[1][i] << endl;
```

You can alternatively save output at the actual integration steps by the declaration

```
Output out(-1);
```

or suppress all saving of output with

```
Output out;
```

In this case, the solution values at the endpoint are available in `ystart[0]` and `ystart[1]`, overwriting the starting values.

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, DC: Mathematical Association of America), Chapter 5.

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 7.

Hairer, E., Nørsett, S.P., and Wanner, G. 1993, *Solving Ordinary Differential Equations I. Nonstiff Problems*, 2nd ed. (New York: Springer)

Hairer, E., Nørsett, S.P., and Wanner, G. 1996, *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*, 2nd ed. (New York: Springer)

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).

17.1 Runge-Kutta Method

The formula for the Euler method is

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{17.1.1}$$

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 17.1.1). 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 (17.1.1).

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 §17.5 below).

Consider, however, the use of a step like (17.1.1) to take a "trial" step to the midpoint of the interval. Then use the values of both x and y at that midpoint to compute the "real" step across the whole interval. Figure 17.1.2 illustrates the idea. In equations,

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{17.1.2}$$

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, (17.1.2) 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 idea. By far the most often used is the classical *fourth-order Runge-Kutta formula*,

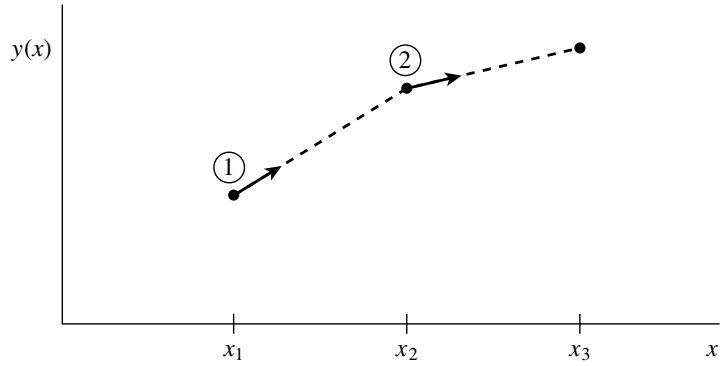

Figure 17.1.1. Euler's method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

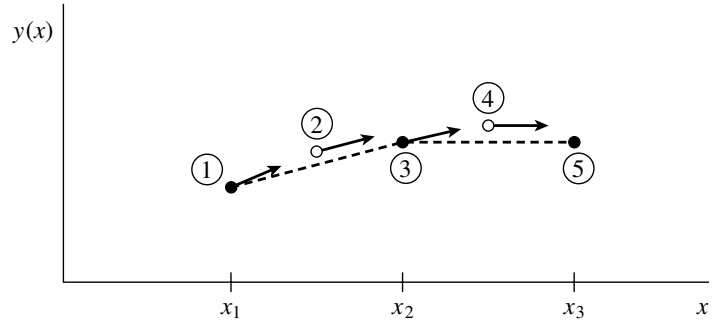

Figure 17.1.2. Midpoint method. Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open dots represent function values that are discarded once their derivatives have been calculated and used.

which has a certain sleekness of organization about it:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1) \\ k_3 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}k_1 + \tfrac{1}{3}k_2 + \tfrac{1}{3}k_3 + \tfrac{1}{6}k_4 + O(h^5) \end{aligned} \tag{17.1.3}$$

The fourth-order Runge-Kutta method requires four evaluations of the right-hand side per step h (see Figure 17.1.3). This will be superior to the midpoint method (17.1.2) *if* at least twice as large a step is possible with (17.1.3) for the same accuracy. Is that so? The answer is: often, perhaps even usually, but surely not always! This takes us back to a central theme, namely that *high order* does not always mean *high accuracy*. The statement "fourth-order Runge-Kutta is generally superior to second-order" is a true one, but as much a statement about the kind of problems that people solve as a statement about strict mathematics.

For many scientific users, fourth-order Runge-Kutta is not just the first word

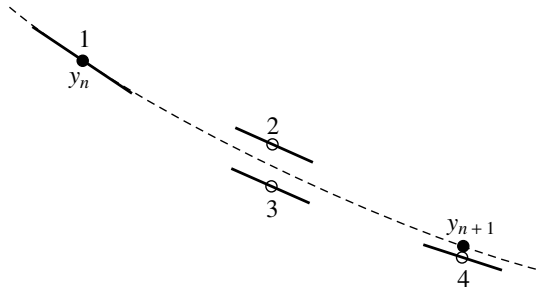

Figure 17.1.3. Fourth-order Runge-Kutta method. In each step the derivative is evaluated four times: once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the final function value (shown as a filled dot) is calculated. (See text for details.)

on ODE integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm. Keep in mind, however, that the old workhorse's last trip may well be to take you to the poorhouse: Newer Runge-Kutta methods are *much* more efficient, and Bulirsch-Stoer or predictor-corrector methods can be even more efficient for problems where very high accuracy is a requirement. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields. However, even the old workhorse is more nimble with new horseshoes. In §17.2 we will give a modern implementation of a Runge-Kutta method that is quite competitive as long as very high accuracy is not required. An excellent discussion of the pitfalls in constructing a good Runge-Kutta code is given in [3].

Here is the routine `rk4` for carrying out one classical Runge-Kutta step on a set of `n` differential equations. This routine is completely separate from the various stepper routines introduced in the previous section and given in the rest of the chapter. It is meant for only the most trivial applications. You input the values of the independent variables, and you get out new values that are stepped by a stepsize `h` (which can be positive or negative). You will notice that the routine requires you to supply not only function `derivs` for calculating the right-hand side, but also values of the derivatives at the starting point. Why not let the routine call `derivs` for this first value? The answer will become clear only in the next section, but in brief is this: This call may not be your only one with these starting conditions. You may have taken a previous step with too large a stepsize, and this is your replacement. In that case, you do not want to call `derivs` unnecessarily at the start. Note that the routine that follows has, therefore, only three calls to `derivs`.

```
void rk4(VecDoub_I &y, VecDoub_I &dydx, const Doub x, const Doub h,                    rk4.h
    VecDoub_O &yout, void derivs(const Doub, VecDoub_I &, VecDoub_O &))
```
Given values for the variables `y[0..n-1]` and their derivatives `dydx[0..n-1]` known at `x`, use the fourth-order Runge-Kutta method to advance the solution over an interval `h` and return the incremented variables as `yout[0..n-1]`. The user supplies the routine `derivs(x,y,dydx)`, which returns derivatives `dydx` at `x`.

```
{
    Int n=y.size();
    VecDoub dym(n),dyt(n),yt(n);
    Doub hh=h*0.5;
    Doub h6=h/6.0;
    Doub xh=x+hh;
```

```
    for (Int i=0;i<n;i++) yt[i]=y[i]+hh*dydx[i];
    derivs(xh,yt,dyt);
    for (Int i=0;i<n;i++) yt[i]=y[i]+hh*dyt[i];
    derivs(xh,yt,dym);
    for (Int i=0;i<n;i++) {
        yt[i]=y[i]+h*dym[i];
        dym[i] += dyt[i];
    }
    derivs(x+h,yt,dyt);
    for (Int i=0;i<n;i++)
        yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
}
```

First step.
Second step.

Third step.

Fourth step.
Accumulate increments with proper weights.

The Runge-Kutta method treats every step in a sequence of steps in an identical manner. Prior behavior of a solution is not used in its propagation. This is mathematically proper, since any point along the trajectory of an ordinary differential equation can serve as an initial point. The fact that all steps are treated identically also makes it easy to incorporate Runge-Kutta into relatively simple "driver" schemes.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at `http://www.nr.com/aands`, §25.5.[1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 2.[2]

Shampine, L.F., and Watts, H.A. 1977, "The Art of Writing a Runge-Kutta Code, Part I," in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, "The Art of Writing a Runge-Kutta Code. II," *Applied Mathematics and Computation*, vol. 5, pp. 93–121.[3]

17.2 Adaptive Stepsize Control for Runge-Kutta

A good ODE integrator should exert some adaptive control over its own progress, making frequent changes in its stepsize. Usually the purpose of this adaptive stepsize control is to achieve some predetermined accuracy in the solution with minimum computational effort. Many small steps should tiptoe through treacherous terrain, while a few great strides should speed through smooth uninteresting countryside. The resulting gains in efficiency are not mere tens of percents or factors of two; they can sometimes be factors of ten, a hundred, or more. Sometimes accuracy may be demanded not directly in the solution itself, but in some related conserved quantity that can be monitored.

Implementation of adaptive stepsize control requires that the stepping algorithm signal information about its performance, most important, an estimate of its truncation error. In this section we will learn how such information can be obtained. Obviously, the calculation of this information will add to the computational overhead, but the investment will generally be repaid handsomely.

With fourth-order Runge-Kutta, the most straightforward technique by far is *step doubling* (see, e.g., [1]). We take each step twice, once as a full step, then,