you would be constantly moving things from private to public (and back again) as you program different test cases and want to examine different internal, normally private, variables.

Because our classes are declared by `struct`, not `class`, use of the word “class” is potentially confusing, and we will usually try to avoid it. So “object” means `struct`, which is really a class!

If you are an OOP beginner, it is important to understand the distinction between defining an object and instantiating it. You define an object by writing code like this:

```cpp
struct Twovar {
    Doub a, b;
    Twovar(const Doub aa, const Doub bb) : a(aa), b(bb) {}
    Doub sum() {return a+b;}
    Doub diff() {return a-b;}
};
```

This code does not create a `Twovar` object. It only tells the compiler how to create one when, later in your program, you tell it to do so, for example by a declaration like,

```cpp
Twovar mytwovar(3.,5.);
```

which invokes the `Twovar` constructor and creates an instance of (or instantiates) a `Twovar`. In this example, the constructor also sets the internal variables `a` and `b` to 3 and 5, respectively. You can have any number of simultaneously existing, noninteracting, instances:

```cpp
Twovar anothertwovar(4.,6.);
Twovar athirdtwovar(7.,8.);
```

We have already promised you that this book is not a textbook in OOP, or the C++ language; so we will go no farther here. If you need more, good references are [1-4].

### 1.3.1 Simple Uses of Objects

We use objects in various ways, ranging from trivial to quite complex, depending on the needs of the specific numerical method that is being discussed. As mentioned in §1.0, this lack of consistency means that *Numerical Recipes* is not a useful examplar of a program library (or, in an OOP context, a *class library*). It also means that, somewhere in this book, you can probably find an example of every possible way to think about objects in numerical computing! (We hope that you will find this a plus.)

**Object for Grouping Functions.** Sometimes an object just collects together a group of closely related functions, not too differently from the way that you might use a `namespace`. For example, a simplification of Chapter 6’s object `Erf` looks like:

```cpp
struct Erf {
    Doub erf(Doub x);
    Doub erfc(Doub x);
    Doub inverf(Doub p);
    Doub inverfc(Doub p);
    Doub erfccheb(Doub z);
};
```

As will be explained in §6.2, the first four methods are the ones intended to be called by the user, giving the error function, complementary error function, and the two
3.7 Interpolation on Scattered Data in Multidimensions

We now leave behind, if with some trepidation, the orderly world of regular grids. Courage is required. We are given an arbitrarily scattered set of $N$ data points $(x_i, y_i), i = 0, \ldots, N-1$ in $n$-dimensional space. Here $x_i$ denotes an $n$-dimensional vector of independent variables, $(x_{i1}, x_{i2}, \ldots, x_{in})$, and $y_i$ is the value of the function at that point.

In this section we discuss two of the most widely used general methods for this problem, radial basis function (RBF) interpolation, and kriging. Both of these methods are expensive. By that we mean that they require $O(N^3)$ operations to initially digest a set of data points, followed by $O(N)$ operations for each interpolated value. Kriging is also able to supply an error estimate — but at the rather high cost of $O(N^2)$ per value. Shepard interpolation, discussed below, is a variant of RBF that at least avoids the $O(N^3)$ initial work; otherwise these workloads effectively limit the usefulness of these general methods to values of $N \lesssim 10^4$. It is therefore worthwhile for you to consider whether you have any other options. Two of these are

- If $n$ is not too large (meaning, usually, $n = 2$), and if the data points are fairly dense, then consider triangulation, discussed in §21.6. Triangulation is an example of a finite element method. Such methods construct some semblance of geometric regularity and then exploit that construction to advantage. Mesh generation is a closely related subject.

- If your accuracy goals will tolerate it, consider moving each data point to the nearest point on a regular Cartesian grid and then using Laplace interpolation (§3.8) to fill in the rest of the grid points. After that, you can interpolate on the grid by the methods of §3.6. You will need to compromise between making the grid very fine (to minimize the error introduced when you move the points) and the compute time workload of the Laplace method.

If neither of these options seem attractive, and you can’t think of another one that is, then try one or both of the two methods that we now discuss. RBF interpolation is probably the more widely used of the two, but kriging is our personal favorite. Which works better will depend on the details of your problem.

The related, but easier, problem of curve interpolation in multidimensions is discussed at the end of this section.

3.7.1 Radial Basis Function Interpolation

The idea behind RBF interpolation is very simple: Imagine that every known point $j$ “influences” its surroundings the same way in all directions, according to
you must also change the last formula in equation (5.8.11) to be

\[ f(x) = x[(2y - 1) d_1 - d_2 + c_0] \]  

(5.8.14)

and change the corresponding line in eval.

### 5.8.1 Chebyshev and Exponential Convergence

Since first mentioning truncation error in §1.1, we have seen many examples of algorithms with an adjustable order, say \( M \), such that the truncation error decreases as the \( M \)th power of something. Examples include most of the interpolation methods in Chapter 3 and most of the quadrature methods in Chapter 4. In these examples there is also another parameter, \( N \), which is the number of points at which a function will be evaluated.

We have many times warned that “higher order does not necessarily give higher accuracy.” That remains good advice when \( N \) is held fixed while \( M \) is increased. However, a recently emerging theme in many areas of scientific computation is the use of methods that allow, in very special cases, \( M \) and \( N \) to be increased together, with the result that errors not only do decrease with higher order, but decrease exponentially!

The common thread in almost all of these relatively new methods is the remarkable fact that infinitely smooth functions become exponentially well determined by \( N \) sample points as \( N \) is increased. Thus, mere power-law convergence may be just a consequence of either (i) functions that are not smooth enough, or (ii) endpoint effects.

We already saw several examples of this in Chapter 4. In §4.1 we pointed out that high-order quadrature rules can have interior weights of unity, just like the trapezoidal rule; all of the “high-orderness” is obtained by a proper treatment near the boundaries. In §4.5 we further saw that variable transformations that push the boundaries off to infinity produce rapidly converging quadrature algorithms. In §4.5.1 we in fact proved exponential convergence, as a consequence of the Euler-Maclaurin formula. Then in §4.6 we remarked on the fact that the convergence of Gaussian quadratures could be exponentially rapid (an example, in the language above, of increasing \( M \) and \( N \) simultaneously).

Chebyshev approximation can be exponentially convergent for a different (though related) reason: Smooth periodic functions avoid endpoint effects by not having endpoints at all! Chebyshev approximation can be viewed as mapping the \( x \) interval \([-1, 1]\) onto the angular interval \([0, \pi]\) (cf. equations 5.8.4 and 5.8.5) in such a way that any infinitely smooth function on the interval \([-1, 1]\) becomes an infinitely smooth, even, periodic function on \([0, 2\pi]\). Figure 5.8.2 shows the idea geometrically. By projecting the abscissas onto a semicircle, a half-period is produced. The other half-period is obtained by reflection, or could be imagined as the result of projecting the function onto an identical lower semicircle. The zeros of the Chebyshev polynomial, or nodes of a Chebyshev approximation, are equally spaced on the circle, where the Chebyshev polynomial itself is a cosine function (cf. equation 5.8.1). This illustrates the close connection between Chebyshev approximation and periodic functions on the circle; in Chapter 12, we will apply the discrete Fourier transform to such functions in an almost equivalent way (§12.4.2).

The reason that Chebyshev works so well (and also why Gaussian quadratures work so well) is thus seen to be intimately related to the special way that the the
Another cousin is the logistic equation,

\[ \frac{dy}{dt} \propto y(y_{\text{max}} - y) \]  \hspace{1cm} (6.14.17)

a differential equation describing the growth of some quantity \( y \), starting off as an exponential but reaching, asymptotically, a value \( y_{\text{max}} \). The solution of this equation is identical, up to a scaling, to the cdf of the logistic distribution.

### 6.14.5 Exponential Distribution

With the exponential distribution we now turn to common distribution functions defined on the positive real axis \( x \geq 0 \). Figure 6.14.2 shows examples of several of the distributions that we will discuss. The exponential is the simplest of them all. It has a parameter \( \beta \) that can control its width (in inverse relationship), but its mode is always at zero:

\[ x \sim \text{Exponential}(\beta), \quad \beta > 0 \]  \hspace{1cm} (6.14.18)

\[ p(x) = \beta \exp(-\beta x), \quad x > 0 \]  \hspace{1cm} (6.14.19)

\[ \text{cdf} \equiv P(< x) = \int_0^x p(x')dx' = 1 - \exp(-\beta x) \]  \hspace{1cm} (6.14.20)

\[ x(P) = \frac{1}{\beta} \log(1 - P) \]  \hspace{1cm} (6.14.21)

The mean and standard deviation of the exponential distribution are both \( 1/\beta \). The median is \( \log(2)/\beta \). Reference [1] has more to say about the exponential distribution than you would ever think possible.
The cdf is given by

\[
\text{cdf} \equiv P(< x) \equiv \int_0^x p(x')dx' = \frac{1}{2}\text{erfc}\left(-\frac{1}{\sqrt{2}} \left[\frac{\log(x) - \mu}{\sigma}\right]\right) \tag{6.14.33}
\]

The inverse to the cdf involves the inverse complementary error function,

\[
x(P) = \exp[\mu - \sqrt{2}\sigma \text{erfc}^{-1}(2P)] \tag{6.14.34}
\]

```cpp
struct Lognormaldist : Erf {
    Doub mu, sig;
    Lognormaldist(Doub mmu = 0., Doub ssig = 1.) : mu(mmu), sig(ssig) {
        if (sig <= 0.) throw("bad sig in Lognormaldist");
    }
    Doub p(Doub x) {
        Return probability density function.
        if (x < 0.) throw("bad x in Lognormaldist");
        if (x == 0.) return 0.;
        return (0.398942280401432678/(sig*x))*exp(-0.5*SQR((log(x)-mu)/sig));
    }
    Doub cdf(Doub x) {
        Return cumulative distribution function.
        if (x < 0.) throw("bad x in Lognormaldist");
        if (x == 0.) return 0.;
        return 0.5*erfc(-0.707106781186547524*(log(x)-mu)/sig);
    }
    Doub invcdf(Doub p) {
        Return inverse cumulative distribution function.
        if (p <= 0. || p >= 1.) throw("bad p in Lognormaldist");
        return exp(-1.41421356237309505*sig*inverfc(2.*p)+mu);
    }
};
```

Multiplicative random walks like (6.14.29) and lognormal distributions are key ingredients in the economic theory of efficient markets, leading to (among many other results) the celebrated Black-Scholes formula for the probability distribution of the price of an investment after some elapsed time \( \tau \). A key piece of the Black-Scholes derivation is implicit in equation (6.14.32): If an investment’s average return is zero (which may be true in the limit of zero risk), then its price cannot simply be a widening lognormal distribution with fixed \( \mu \) and increasing \( \sigma \), for its expected value would then diverge to infinity! The actual Black-Scholes formula thus defines both how \( \sigma \) increases with time (basically as \( \tau^{1/2} \)) and how \( \mu \) correspondingly decreases with time, so as to keep the overall mean under control. A simplified version of the Black-Scholes formula can be written as

\[
S(\tau) \sim S(0) \times \text{Lognormal}\left(r\tau - \frac{1}{2}\sigma^2\tau, \sigma\sqrt{\tau}\right) \tag{6.14.35}
\]

where \( S(\tau) \) is the price of a stock at time \( \tau \), \( r \) is its expected (annualized) rate of return, and \( \sigma \) is now redefined to be the stock’s (annualized) volatility. The definition of volatility is that, for small values of \( \tau \), the fractional variance of the stock’s price is \( \sigma^2\tau \). You can check that (6.14.35) has the desired expectation value \( E[S(\tau)] = S(0) \), for all \( \tau \), if \( r = 0 \). A good reference is [3].
Then equation (7.3.18) becomes
\[
p(x)dx = \int_{p' = 0}^{p' = p(x)} dp' dx = \int_{u=0}^{u=\sqrt{p(x)}} \frac{\partial(p, x)}{\partial(u, v)} du \, dv = 2 \int_{u=0}^{u=\sqrt{p(v/u)}} du \, dv
\]
because (as you can work out) the Jacobian determinant is the constant 2. Since the new integrand is constant, uniform sampling in \((u, v)\) with the limits indicated for \(u\) is equivalent to the rejection method in \((x, p)\).

The above limits on \(u\) very often define a region that is "teardrop" shaped. To see why, note that the loci of constant \(x = v/u\) are radial lines. Along each radial, the acceptance region goes from the origin to a point where \(u^2 = p(x)\). Since most probability distributions go to zero for both large and small \(x\), the acceptance region accordingly shrinks toward the origin along radials, producing a teardrop. Of course, it is the exact shape of this teardrop that matters. Figure 7.3.3 shows the shape of the acceptance region for the case of the normal distribution.

Typically this ratio-of-uniforms method is used when the desired region can be closely bounded by a rectangle, parallelogram, or some other shape that is easy to sample uniformly. Then, we go from sampling the easy shape to sampling the desired region by rejection of points outside the desired region.

An important adjunct to the ratio-of-uniforms method is the idea of a squeeze. A squeeze is any easy-to-compute shape that tightly bounds the region of acceptance of a rejection method, either from the inside or from the outside. Best of all is when you have squeezes on both sides. Then you can immediately reject points that are outside the outer squeeze and immediately accept points that are inside the inner squeeze. Only when you have the bad luck of drawing a point between the two squeezes do you actually have to do the more lengthy computation of comparing with the actual rejection boundary. Squeezes are useful both in the ordinary rejection method and in the ratio-of-uniforms method.

**7.3.9 Normal Deviates by Ratio-of-Uniforms**

Leva [2] has given an algorithm for normal deviates that uses the ratio-of-uniforms method with great success. He uses quadratic curves to provide both inner
and outer squeezes that hug the desired region in the \((u,v)\) plane (Figure 7.3.3). Only about 1% of the time is it necessary to calculate an exact boundary (requiring a logarithm).

The resulting code looks so simple and “un-transcendental” that it may be hard to believe that exact normal deviates are generated. But they are!

```c
struct Normaldev : Ran {
    Doub mu, sig;
    Normaldev(Doub mmu, Doub ssig, Ullong i) : Ran(i), mu(mmu), sig(ssig) {}
    Constructor arguments are \(\mu, \sigma\), and a random sequence seed.
    Doub dev() {
        Doub u, v, x, y, q;
        do {
            u = doub();
            v = 1.7156*(doub()-0.5);
            x = u - 0.449871;
            y = abs(v) + 0.386595;
            q = SQR(x) + y*(0.19600*y-0.25472*x);
        } while (q > 0.27597 && (q > 0.27846 || SQR(v) > -4.*log(u)*SQR(u)));
        return mu + sig*v/u;
    }
};
```

Note that the `while` clause makes use of C’s (and C++’s) guarantee that logical expressions are evaluated conditionally: If the first operand is sufficient to determine the outcome, the second is not evaluated at all. With these rules, the logarithm is evaluated only when \(q\) is between 0.27597 and 0.27846.

On average, each normal deviate uses 2.74 uniform deviates. By the way, even though the various constants are given only to six digits, the method is exact (to full double precision). Small perturbations of the bounding curves are of no consequence. The accuracy is implicit in the (rare) evaluations of the exact boundary.

### 7.3.10 Gamma Deviates

The distribution Gamma\((\alpha, \beta)\) was described in \(\S 6.14.9\). The \(\beta\) parameter enters only as a scaling,

\[
\text{Gamma}(\alpha, \beta) \equiv \frac{1}{\beta} \text{Gamma}(\alpha, 1) \tag{7.3.21}
\]

(Translation: To generate a Gamma\((\alpha, \beta)\) deviate, generate a Gamma\((\alpha, 1)\) deviate and divide it by \(\beta\).)

If \(\alpha\) is a small positive integer, a fast way to generate \(x \sim \text{Gamma}(\alpha, 1)\) is to use the fact that it is distributed as the waiting time to the \(\alpha\)th event in a Poisson random process of unit mean. Since the time between two consecutive events is just the exponential distribution Exponential \((1)\), you can simply add up \(\alpha\) exponentially distributed waiting times, i.e., logarithms of uniform deviates. Even better, since the sum of logarithms is the logarithm of the product, you really only have to compute the product of \(\alpha\) uniform deviates and then take the log. Because this is such a special case, however, we don’t include it in the code below.
10.11.4 Path-Following Methods

Path-following methods don’t just aim steps in the direction of the central path; they explicitly attempt to stay close to it. These methods are currently the most successful interior-point methods. In primal-dual methods, the duality gap (10.11.4), which is equal to $x \cdot z$ for feasible points, provides a figure-of-merit for how close one is to the optimal solution. Accordingly, we set

$$
\mu = x \cdot z / n, \quad \tau = \mu \delta, \quad \delta \in [0, 1]
$$

(10.11.12)

The quantity $\delta$ is called the centering parameter, while $\mu$ is called the duality measure. If $\delta = 1$, the Newton step calculated from (10.11.11) is in a centering direction, toward a point at which each product $x_j z_j$ is equal to the average value $\mu$ defined in (10.11.12). On the other hand, the value $\delta = 0$ defines the Newton step for the original system (10.11.8). Good algorithms use intermediate values to trade off between improving centrality and reducing $\mu$.

Methods that keep $\delta$ close to 1, so that unit steps ($\alpha = 1$) stay close to the central path, are called short-step methods. Methods that allow small values of $\delta$ are called long-step methods (less conservative choices of $\delta$). There is an interesting gap between theory and practice between the methods. Short-step methods have been proved to converge in $O(\sqrt{n} \log \frac{1}{\epsilon})$ iterations, where $\epsilon$ is the desired tolerance. Long-step methods take $O(n \log \frac{1}{\epsilon})$ iterations, according to theory. Yet in practice short-step methods take hopelessly small steps, while long-step methods provide practical algorithms.

This is a somewhat academic discussion anyway. Real-life examples take many fewer than $O(\sqrt{n})$ iterations—a few dozen is typical for large problems.

10.11.5 Barrier Methods

Introducing a ‘penalty’ function is a standard technique to enforce a constraint in general optimization problems. For example, to enforce the condition $x \geq 0$, consider the logarithmic penalty function

$$
\sum_{j=1}^{n} \log x_j
$$

(10.11.13)

If any $x_j \to 0$, this function tends to $-\infty$. So instead of trying to minimize $c \cdot x$ in the standard primal problem (10.11.1), consider minimizing

$$
c \cdot x - \tau \sum_{j=1}^{n} \log x_j
$$

(10.11.14)

If one takes the limit $\tau \to 0$ after the minimization, we expect this to be equivalent to the original problem.

Equation (10.11.14) is called a logarithmic barrier function. It defines a family of nonlinear objective functions that gives the solution to the original problem as the parameter $\tau \to 0$.

The power of the barrier function idea is that it lets us handle the constraint $x \geq 0$ with calculus. To minimize (10.11.14) subject to the constraint $A \cdot x = b$, introduce a Lagrange multiplier $-y$ and extremize the Lagrangian

$$
L = c \cdot x - \tau \sum_{j=1}^{n} \log x_j - y \cdot (A \cdot x - b)
$$

(10.11.15)
The optimality conditions $\nabla_x L = 0$ and $\nabla_y L = 0$ give

$$\begin{align*}
A \cdot x &= b \\
A^T \cdot y + \tau X^{-1} \cdot e &= c
\end{align*}$$  \hfill (10.11.16)

Define the vector

$$z = \tau X^{-1} \cdot e; \quad \text{i.e.,} \quad z_j = \tau / x_j$$  \hfill (10.11.17)

Then equation (10.11.16) becomes

$$\begin{align*}
A \cdot x &= b \\
A^T \cdot y + z &= c \\
X \cdot Z \cdot e &= \tau e
\end{align*}$$  \hfill (10.11.18)

These are exactly the equations (10.11.11) defining the central path, and they reduce to the KKT conditions (10.11.7) if we set $\tau$ to zero.

Note that equation (10.11.16) can be used to define an algorithm, the primal interior-point method, that doesn’t depend on $z$. Similarly, by starting with a logarithmic barrier function for the dual objective function, one can derive a purely dual method that doesn’t involve $x$. In practice, these methods are not competitive with the primal-dual methods.

Originally the logarithmic barrier function idea played an important role in motivating interior-point methods. More recently, the viewpoint has shifted to emphasize the importance of $\tau$ as defining the centering property of the algorithm rather than being simply a parameter to enforce the nonnegativity constraint.

### 10.11.6 A Primal-Dual Infeasible Interior-Point Algorithm

Let’s pull all the pieces together now to define the algorithm. Write equation (10.11.11) for the new iterate:

$$\begin{align*}
A \cdot (x + \Delta x) - b &= 0 \\
A^T \cdot (y + \Delta y) + z + \Delta z - c &= 0 \\
(X + \Delta X) \cdot (Z + \Delta Z) \cdot e &= \tau e
\end{align*}$$  \hfill (10.11.19)

Drop the quadratic term $\Delta X \cdot \Delta Z \cdot e$ and get

$$\begin{bmatrix}
A & 0 & 0 \\
0 & A^T & 1 \\
Z & 0 & X
\end{bmatrix}
\begin{bmatrix}
\Delta x \\
\Delta y \\
\Delta z
\end{bmatrix}
= \begin{bmatrix}
-r_p \\
-r_d \\
\tau e - X \cdot Z \cdot e
\end{bmatrix}$$  \hfill (10.11.20)

where the primal and dual residuals are defined by

$$\begin{align*}
r_p &= A \cdot x - b \\
r_d &= A^T \cdot y + z - c
\end{align*}$$  \hfill (10.11.21)

Equation (10.11.20) is simply the Newton equation (10.11.9) for (10.11.11). Note that the only nonlinearity comes from the innocuous looking quadratic term for complementary slackness. Yet it’s exactly what leads to all the difficulty!
Figure 13.4.4. Leakage function $W(x)$ for various window functions. The top row is essentially the same as Figure 13.4.2, but squared (to get power) and plotted logarithmically. The second and third rows are examples of Slepian functions, identified by $j_{res}, k$ values. Small $k$ values have exceedingly small leakage for $|x| > j_{res}$; but as $k$ increases, so does the leakage. Shaded functions have unacceptably large leakage and are not recommended.

We discuss the body of `filltable`, where the Slepian functions are actually computed, below.
First, a few words about the use and misuse of multitaper methods. The Slepian multitaper method is fundamentally about low leakage. The fact that it can reduce the variance a bit by taking $k_T > 1$ is only a secondary consideration, because there are better ways to achieve the latter goal, for example by overlapping data segments. It follows that you should never need to take $j_{res}$ or $k_T$ very large, greater than 10, say. Your logical path for choosing parameters should be something like this: Leakage suppression of the Slepian functions is so amazingly good that you can get to any plausible desired level for the first few eigenvectors with modest $j_{res}$. Find that value, and the largest acceptable value for $k_T$. The frequency resolution is now $j_{res}$, measured in bins. You now pick $M$ to get the physical frequency resolution that you actually need,

$$f_{res} = \frac{j_{res}}{2M\Delta}$$

(compare equation 13.4.6). Don’t be too greedy, or you will produce an unacceptably large variance. Now, if you have $N_{tot}$ data points, you process $N_{tot}/(2M)$ separate data segments using adddataseg.

It would be misguided to increase $j_{res}$ to a large value just to increase $k_T$ for the purpose of variance reduction. The reason is that, for a fixed desired physical frequency resolution, you will need to increase $M$ in proportion, and thus decrease your number of separate data segments, also in proportion. You thus gain nothing in variance reduction, and (potentially) lose greatly in leakage.

If squeezing down the variance by the last little bit is important, then you might consider using only the first Slepian function for a given $j_{res}$, and then using overlapping data segments. You can code this using Spectolap and Slepian as models. As $j_{res}$ increases, the optimal spacing of overlapped segments decreases, as you can intuit from the narrowing central peaks in Figure 13.4.5. A spacing of $0.7N/\sqrt{j_{res}} + 0.3$, that is, overlap of $N - 0.7N/\sqrt{j_{res}} + 0.3$, should be about right.

### 13.4.4 Computation of the Slepian Functions

We want to find the first few eigenvectors and eigenvalues of the tridiagonal matrix, equations (13.4.18) and (13.4.19). For $N \gg 1$ (always our situation), the eigenvalues are well separated and approximately a function of $j_{res}$ only. A good starting approximation for the
Chapter 15. Modeling of Data

15.4.4 Multidimensional Fits

If you are measuring a single variable $y$ as a function of more than one variable — say, a vector of variables $x$ — then your basis functions will be functions of a vector, $X_0(x), \ldots, X_{M-1}(x)$. The $\chi^2$ merit function is now

$$\chi^2 = \sum_{i=0}^{N-1} \left[ \frac{y_i - \sum_{k=0}^{M-1} a_k X_k(x_i)}{\sigma_i} \right]^2 \tag{15.4.24}$$

All of the preceding discussion goes through unchanged, with $x$ replaced by $x$. In fact, we anticipated this in the coding of Fitsvd, above, which can do multidimensional general linear fits as easily as one-dimensional. Here is how:

A second, overloaded, constructor in Fitsvd substitutes a matrix $xx$ for what was previously a vector. The rows of the matrix are the $\text{ndat}$ data points. The number of columns is the dimensionality of the space (that is, of $x$). Similarly, the user-supplied function $\text{funks}$ now takes a vector argument, an $x$. A simple example (fitting a quadratic function to data in two dimensions) might be

```cpp
VecDoub quadratic2d(VecDoub_I &xx) {
    VecDoub ans(6);
    Doub x = xx[0], y = xx[1];
    ans[0] = 1;
    return ans;
}
```

Be sure that the argument of your user function has exactly the type “VecDoub_I &” (and not, for example, “VecDoub &&” or “VecDoub_I”), since strict C++ compilers are picky about this.

The two constructors in Fitsvd communicate to fit whether data points are one-dimensional or multidimensional by setting either $\text{xmd}$ or $x$ to NULL. This explains the oddity that $x$ was bound to the user data as a pointer, while $y$ and $\text{sig}$ were bound as references. (Yes, we know this is a bit of a hack!)

CITED REFERENCES AND FURTHER READING:


Now, to generate a step starting at $x_1$, first generate a candidate point $x_{2c}$ by drawing from the proposal distribution. Second, calculate an acceptance probability $\alpha(x_1, x_{2c})$ by the formula

$$\alpha(x_1, x_{2c}) = \min\left(1, \frac{\pi(x_{2c}) q(x_1|x_{2c})}{\pi(x_1) q(x_{2c}|x_1)}\right)$$  \hspace{1cm} (15.8.5)

Finally, with probability $\alpha(x_1, x_{2c})$, accept the candidate point and set $x_2 = x_{2c}$; otherwise reject it and leave the point unchanged (that is, $x_2 = x_1$). The net result of this process is a transition probability,

$$p(x_2|x_1) = q(x_2|x_1) \alpha(x_1, x_2), \quad (x_2 \neq x_1)$$  \hspace{1cm} (15.8.6)

To see how this satisfies detailed balance, first multiply equation (15.8.5) by the denominator in the second argument of the min function. Then write down the identical equation, but exchange $x_1$ and $x_2$. From these pieces, one writes,

$$\pi(x_1)q(x_2|x_1)\alpha(x_1, x_2) = \min[\pi(x_1)q(x_2|x_1), \pi(x_2)q(x_1|x_2)]$$

$$= \min[\pi(x_2)q(x_1|x_2), \pi(x_1)q(x_2|x_1)]$$

$$= \pi(x_2)q(x_1|x_2)\alpha(x_2, x_1)$$  \hspace{1cm} (15.8.7)

which, using equation (15.8.6), can be seen to be exactly the detailed balance equation (15.8.1).

It is often possible to choose the proposal distribution $q(x_2|x_1)$ in such a way as to simplify equation (15.8.5). For example, if $q(x_2|x_1)$ depends only on the absolute difference $|x_1 - x_2|$, as in the case of a normal distribution with fixed covariance, then the ratio $q(x_1|x_{2c})/q(x_{2c}|x_1)$ is just 1. Another case that occurs frequently is when, for some component $x$ of $x$, $q(x_{2c}|x_1)$ is lognormally distributed with a mode at $x_1$. In that case the ratio for this component is $x_{2c}/x_1$ (cf. equation 6.14.31).

### 15.8.2 Gibbs Sampler

An important special case of the Metropolis-Hastings algorithm is the Gibbs sampler. (Historically, the Gibbs sampler was developed independently of Metropolis-Hastings, see [2,5], but we discuss it here in a unified framework.) The Gibbs sampler is based on the fact that a multivariate distribution is uniquely determined by the set of all of its full conditional distributions; but if you don’t know what this means, just read on anyway.

A full conditional distribution of $\pi(x)$ is obtained by holding all of the components of $x$ constant except one (call it $x$), and then sampling as a function of $x$ alone. In other words, it is the distribution that you see when you “drill through” $\pi(x)$ along a coordinate direction, and with fixed values of all the other coordinates. We’ll denote a full conditional distribution by the notation $\pi(x | x^-)$, where $x^-$ means “values of all the coordinates except one.” (To keep the notation readable, we are suppressing an index $i$ that would tell which component of $x$ is $x_i$.)

Suppose that we construct a Metropolis-Hastings chain that allows only the one coordinate $x$ to vary. Then equation (15.8.5) would look like this:

$$\alpha(x_1, x_{2c}|x^-) = \min\left(1, \frac{\pi(x_{2c}|x^-)q(x_1|x_{2c}, x^-)}{\pi(x_1|x^-)q(x_{2c}|x_1, x^-)}\right)$$  \hspace{1cm} (15.8.8)
close to one or zero; but if the voltage is only \( t = 0.5 \) (say) standard deviations away from zero, we may want to assign a probability of 0.6915 to one more favored outcome, and 0.3085 to the less favored, since

\[
\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{0.5} e^{-z^2/2} dz \approx 0.6915 \tag{16.2.3}
\]

(By the end of this section we will be more sophisticated about the notion of assigning probabilities to transitions.)

The second great idea is that the problem of finding the maximum likelihood path through a trellis — that is, the path with the maximum product of the probabilities at each stage — is just a dynamic programming problem, where the cost of traversing an edge whose probability is \( p \) is taken as \(-\log(p)\), a positive number, since \( 0 \leq p \leq 1 \). The minimum cost path, with this metric, is the maximum likelihood path. In each stage, all the 0 edges (dotted lines in the figure) get one probability, and all the 1 edges (solid lines) get its complement. The edge probabilities can, and in general will, vary with each bit received (that is, from stage to stage), since the noise and path loss can vary with time.

Take these two ideas together and you have soft-decision decoding using the Viterbi algorithm.

The following tableau decodes one codeword for the shortened Hamming (6, 3) code given above. In this example, five of the six bits are received fairly unambiguously, while one bit (the second) is seen to be rather ambiguous. Nevertheless, the algorithm treats all bits equally “softly.” Reviewing §10.13 as necessary, you should be able to see where all the numbers in the tableau come from, as well as how the darker path (which is the final “hard” decision for the codeword 011011) is obtained by backtracking. Given the appropriate cost function, the routine \texttt{dynpro} in §10.13 does exactly this calculation.

You might have the “bright idea” of converting the final minimum path length, 1.15 in the above example, into a probability by taking its negative exponential. The result is 0.3166. Does this mean that you’ll get the right codeword only 31% of the time? No! Go stand in the corner! You are confusing likelihood with probability. The likelihoods of all eight codewords (not found by the DP algorithm, but computed exhaustively) are, for this example,
Figure 16.3.1. Example of a Markov model. Transitions occur between states along the directed edges shown. Each outgoing edge is labeled by its probability. The sum of the probabilities on the outgoing edges from each state is 1. This model is called “Teen Life.”

16.3 Markov Models and Hidden Markov Modeling

Trellises, like those in §16.2, are directed graphs without any loops, so a path that begins at the leftmost node inevitably ends, after a finite number of stages, at the rightmost node. The more general Markov model lives on a graph that can have loops (as in Figure 16.2.1), so some paths can continue indefinitely. Indeed, as a convention, one can add a self-loop (a directed edge connecting a state to itself) to any state that would otherwise have “no way out.” Then, all paths can be continued indefinitely, even those whose fate is to remain stuck in a single state.

To turn such a directed graph into a Markov model (also known as a Markov chain or first-order Markov process), we simply label all of its edges with transition probabilities, such that the sum of probabilities over the outgoing edges from each node is 1. Figure 16.3.1 shows an example, a Markov model with five states, that we call “Teen Life.”

A single realization of a Markov model is a random path that moves from state to state according to the model’s probabilities. These are conveniently organized into a transition matrix $A$ whose element $A_{ij}$ is the probability associated with an $i \rightarrow j$ transition, that is, the probability of moving to state $j$, given state $i$ as the starting point. A valid transition matrix satisfies

$$ 0 \leq A_{ij} \leq 1 \quad \text{and} \quad \sum_j A_{ij} = 1 \quad (16.3.1) $$

The transition matrix for Teen Life (Figure 16.3.1) is

$$ A = \begin{pmatrix}
0 & 0.7 & 0.1 & 0 & 0.2 \\
0.2 & 0.4 & 0 & 0.2 & 0.2 \\
0 & 1.0 & 0 & 0 & 0 \\
0 & 0.3 & 0 & 0.7 & 0 \\
0.1 & 0.1 & 0 & 0 & 0.8
\end{pmatrix} \quad (16.3.2) $$

where the states are numbered in the order (Eat, Hang, Study, Talk, and Sleep).

A routine for generating a realization of a Markov model from its $M \times M$ transition matrix, using the Ran structure of §7.1 to get random numbers, is straightforward.
17.0 Introduction

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 ($\leq 10^{-10}$), or in general when moderate accuracy ($\leq 10^{-5}$) 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
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.

odint.h

```cpp
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;  // Take at most MAXSTP steps.
  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 hmin,Output &outt
          typename Stepper::Dtype &derivss);
};
```
CITED REFERENCES AND FURTHER READING:


Hackbusch, W. 1985, *Multi-Grid Methods and Applications* (New York: Springer).[3]


### 20.7 Spectral Methods

Spectral methods are a very powerful tool for solving PDEs. When they can be used, they are the method of choice if you need high spatial resolution in multidimensions. For a second-order accurate finite difference code in three dimensions, increasing the resolution by a factor of 2 in each dimension requires eight times as many grid points, and improves the error typically by a factor of 4. In a spectral code, a similar increase in resolution often gives an improvement of a factor of $10^6$. Even for one-dimensional problems, spectral methods will amaze you with their power and efficiency.

Spectral methods work well for smooth solutions. Discontinuities like shocks are bad — don’t even try spectral methods. Even mild nonsmoothness (like a discontinuity in some high-order derivative of the solution) can spoil the convergence of spectral methods. (Actually, getting spectral methods to work with discontinuities and shocks is an active research area; see [1] for an introduction.)

The key difference between finite difference methods and spectral methods is that in finite difference methods you approximate the *equation* you are trying to solve, whereas in spectral methods you approximate the *solution* you are trying to find. While finite differencing replaces the continuum equation by an equation on grid points, a spectral method expresses the solution as a truncated expansion in a set
of basis functions:
\[ f(x) \simeq f_N(x) = \sum_{n=0}^{N} a_n \phi_n(x) \quad (20.7.1) \]

Different choices of basis functions and methods of computing \( a_n \) give different flavors of spectral methods.

### 20.7.1 Example

We illustrate the idea of spectral methods with an example. Consider the one-sided wave equation (advective equation) in one dimension:
\[ \frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} \quad (20.7.2) \]
with periodic boundary conditions on \([0, 2\pi]\) and initial condition
\[ u(t = 0, x) = f(x) \quad (20.7.3) \]
You get the analytic spectral solution by expanding \( u \) in a Fourier series,
\[ u(t, x) = \sum_{n=-\infty}^{\infty} a_n(t) e^{inx} \quad (20.7.4) \]
Substituting this expansion into equation (20.7.2) gives
\[ \frac{da_n}{dt} = i n a_n \quad (20.7.5) \]
with solution
\[ a_n(t) = a_n(0)e^{int} \quad (20.7.6) \]
You get \( a_n(0) \) from the initial condition: Expand
\[ f(x) = \sum_{n=-\infty}^{\infty} f_n e^{inx} \quad (20.7.7) \]
from which you see that
\[ a_n(0) = f_n \quad (20.7.8) \]
For example, suppose
\[ f(x) = \sin(\pi \cos x) \quad (20.7.9) \]
which gives the analytic solution
\[ u(t, x) = \sin[\pi \cos(x + t)] \quad (20.7.10) \]
The spectral coefficients in the solution (20.7.4) are
\[
   a_n(0) = \frac{1}{2\pi} \int_{0}^{2\pi} \sin(\pi \cos x)e^{-inx} \, dx \\
   = (-1)^{(n-1)/2} J_n(\pi), \quad n \text{ odd} \quad (20.7.11)
\]
21.4 Lines, Line Segments, and Polygons

Figure 21.4.6. Is a point inside a polygon? (a) For a simple polygon, either the winding number, or the Jordan curve theorem (even or odd number of crossings of a ray) can be used. (b) For complex polygons, there is no simple test.

check whether the counter is even or odd. But if \texttt{polywind} as written returns 0, it \textit{must} have encountered the same number of increments as decrements, hence an even number of crossings. And if it returns ±1 (the only other possible value for a simple polygon), it must similarly have encountered an odd number. So the two methods are really the same.

What if your polygon is not simple? As Figure 21.4.6(b) illustrates, you are in deep waters. Both the winding number method and the Jordan curve theorem method will say that the upper point in the figure is inside the complex polygon shown, and this seems intuitively correct. However, both methods will say that the lower point is \textit{outside} the polygon. Indeed, there are some self-consistent ways of defining “insideness” for complex polygons that make this the case. The result is so counterintuitive, however, as to be useless in most practical applications. It is generally better just to avoid using the idea of “insideness” with complex polygons.

\textbf{Classification of Polygons.} We are now in a position to combine several of the ideas already introduced into a function that classifies any polygon as either simple or complex, and (if it is simple) whether it is convex or concave, and whether it is \texttt{CCW} (total winding number 1) or \texttt{CW} (total winding number -1).

```c
int ispolysimple(const vector<Point<2> > &vt) {
    polygon.h
    Classifies a polygon specified by a vector of vertex points \texttt{vt}. Returns 0 if the polygon is complex (has intersecting edges). Returns 1 if it is simple and convex. Returns 2 if it is simple and concave. The sign of the returned value indicates whether the polygon is \texttt{CCW} (+) or \texttt{CW} (-).
    int i,ii,j,jj,np,schg=0,wind=0;
    Initialize sign change and winding number.
    Doub p0,p1,d0,d1,pp0,pp1,dd0,dd1,t,tp,t1,t2,crs,crsp=0.0;
    np = vt.size();
    p0 = vt[0].x[0]-vt[np-1].x[0];
    p1 = vt[0].x[1]-vt[np-1].x[1];
    for (i=0,ii=1; i<np; i++,ii++) {
        Loop over edges.
        if (ii == np) ii = 0;
        d0 = vt[ii].x[0]-vt[i].x[0];
        d1 = vt[ii].x[1]-vt[i].x[1];
        crs = p0*d1-p1*d0;
        Cross product at this vertex.
        if (crs*crsp < 0) schg = 1;
        Sign change (i.e., concavity) found.
        if (p1 <= 0.0) {
            Wind number logic as in \texttt{polywind}.
            if (d1 > 0.0 && crs > 0.0) wind++;
        } else {
            if (d1 <= 0.0 && crs < 0.0) wind--;
        }
        p0=d0;
        p1=d1;
        Save previous cross product only if it has a sign!
        if (crs != 0.0) crsp = crs;
    }
    return wind;
}
```
The volume $V_n$ of the $n$-dimensional ball is equal to $r/n$ times the area of the enclosing sphere in $n$ dimensions, and also has a simple recurrence,

\begin{align*}
V_1 &= 2r \quad \text{(length of line)} \\
V_2 &= \pi r^2 \quad \text{(area of circle)} \\
\vdots \\
V_n &= \frac{r}{n} S_n = \frac{2\pi r^2}{n} V_{n-2}
\end{align*}

(21.5.6)

Closed-form formulas require a gamma function,

\begin{align*}
S_n &= \frac{2\pi^{n/2} r^{n-1}}{\Gamma(\frac{1}{2}n)} \\
V_n &= \frac{2\pi^{n/2}}{n \Gamma(\frac{1}{2}n)} r^n
\end{align*}

(21.5.7)

As $n$ becomes large, the ratio of the volume of a ball to the volume of the circumscribed (hyper-)cube rapidly becomes small,

\[ \frac{V_n}{2^n} \to 0, \quad n \to \infty \]

(21.5.8)

### 21.5.1 Picking a Random Point on the Sphere

You don’t get a random point on the sphere in $n$ dimensions by picking uniformly random values for the $n - 1$ angles in equation (21.5.2), just as you don’t get a random point on the Earth’s surface by throwing darts at a Mercator map (or any other non-equal-area projection).

An elegant general method is to generate $n$ independent, identically distributed, normal (Gaussian) deviates of zero mean, say $y_0, \ldots, y_{n-1}$ (see §7.3), and then calculate a point $x$ on the unit sphere in $n$ dimensions by

\[ x = \frac{y}{|y|} \]

(21.5.9)

or, in other words,

\[ x_i = y_i \sqrt{\sum_{j=0}^{n-1} y_j^2} \]

(21.5.10)

This works because the spherically symmetric Gaussian distribution in $n$ dimensions trivially factorizes into a product of independent one-dimensional Gaussians. If you want a random point inside the enclosed $n$-volume, generate an additional uniform random deviate $u$ in $[0, 1]$ and calculate the point’s coordinates as

\[ x_i = u^{1/n} y_i \sqrt{\sum_{j=0}^{n-1} y_j^2} \]

(21.5.11)

You can of course scale to any other radius of sphere.