is symmetric. However, since the weights w_j are not equal for most quadrature rules, the matrix $\widetilde{\mathbf{K}}$ (equation 18.1.5) is not symmetric. The matrix eigenvalue problem is much easier for symmetric matrices, and so we should restore the symmetry if possible. Provided the weights are positive (which they are for Gaussian quadrature), we can define the diagonal matrix $\mathbf{D} = \operatorname{diag}(w_j)$ and its square root, $\mathbf{D}^{1/2} = \operatorname{diag}(\sqrt{w_j})$. Then equation (18.1.7) becomes

$$\mathbf{K} \cdot \mathbf{D} \cdot \mathbf{f} = \sigma \mathbf{f}$$

Multiplying by $\mathbf{D}^{1/2}$, we get

$$\left(\mathbf{D}^{1/2} \cdot \mathbf{K} \cdot \mathbf{D}^{1/2}\right) \cdot \mathbf{h} = \sigma \mathbf{h} \tag{18.1.8}$$

where $\mathbf{h} = \mathbf{D}^{1/2} \cdot \mathbf{f}$. Equation (18.1.8) is now in the form of a symmetric eigenvalue problem.

Solution of equations (18.1.7) or (18.1.8) will in general give N eigenvalues, where N is the number of quadrature points used. For square-integrable kernels, these will provide good approximations to the lowest N eigenvalues of the integral equation. Kernels of *finite rank* (also called *degenerate* or *separable* kernels) have only a finite number of nonzero eigenvalues (possibly none). You can diagnose this situation by a cluster of eigenvalues σ that are zero to machine precision. The number of nonzero eigenvalues will stay constant as you increase N to improve their accuracy. Some care is required here: A nondegenerate kernel can have an infinite number of eigenvalues that have an accumulation point at $\sigma = 0$. You distinguish the two cases by the behavior of the solution as you increase N. If you suspect a degenerate kernel, you will usually be able to solve the problem by analytic techniques described in all the textbooks.

CITED REFERENCES AND FURTHER READING:

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [1]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.).

18.2 Volterra Equations

Let us now turn to Volterra equations, of which our prototype is the Volterra equation of the second kind,

$$f(t) = \int_a^t K(t,s) f(s)\, ds + g(t) \tag{18.2.1}$$

Most algorithms for Volterra equations march out from $t = a$, building up the solution as they go. In this sense they resemble not only forward substitution (as discussed in §18.0), but also initial-value problems for ordinary differential equations. In fact, many algorithms for ODEs have counterparts for Volterra equations.

The simplest way to proceed is to solve the equation on a mesh with uniform spacing:

$$t_i = a + ih, \quad i = 0, 1, \ldots, N, \qquad h \equiv \frac{b-a}{N} \tag{18.2.2}$$

To do so, we must choose a quadrature rule. For a uniform mesh, the simplest scheme is the trapezoidal rule, equation (4.1.11):

$$\int_a^{t_i} K(t_i, s) f(s)\, ds = h \left(\tfrac{1}{2} K_{i0} f_0 + \sum_{j=1}^{i-1} K_{ij} f_j + \tfrac{1}{2} K_{ii} f_i \right) \tag{18.2.3}$$

Thus the trapezoidal method for equation (18.2.1) is:

$$\begin{aligned} f_0 &= g_0 \\ (1 - \tfrac{1}{2} h K_{ii}) f_i &= h \left(\tfrac{1}{2} K_{i0} f_0 + \sum_{j=1}^{i-1} K_{ij} f_j \right) + g_i, \qquad i = 1, \ldots, N \end{aligned} \tag{18.2.4}$$

(For a Volterra equation of the first kind, the leading 1 on the left would be absent, and g would have opposite sign, with corresponding straightforward changes in the rest of the discussion.)

Equation (18.2.4) is an explicit prescription that gives the solution in $O(N^2)$ operations. Unlike Fredholm equations, it is not necessary to solve a system of linear equations. Volterra equations thus usually involve less work than the corresponding Fredholm equations which, as we have seen, do involve the inversion of, sometimes large, linear systems.

The efficiency of solving Volterra equations is somewhat counterbalanced by the fact that *systems* of these equations occur more frequently in practice. If we interpret equation (18.2.1) as a *vector* equation for the vector of m functions $f(t)$, then the kernel $K(t,s)$ is an $m \times m$ matrix. Equation (18.2.4) must now also be understood as a vector equation. For each i, we have to solve the $m \times m$ set of linear algebraic equations by Gaussian elimination.

The routine `voltra` below implements this algorithm. You must supply an external function that returns the kth function of the vector $g(t)$ at the point t, and another that returns the (k,l) element of the matrix $K(t,s)$ at (t,s). The routine `voltra` then returns the vector $f(t)$ at the regularly spaced points t_i.

```
#include "nrutil.h"

void voltra(int n, int m, float t0, float h, float *t, float **f,
    float (*g)(int, float), float (*ak)(int, int, float, float))
```

Solves a set of `m` linear Volterra equations of the second kind using the extended trapezoidal rule. On input, `t0` is the starting point of the integration and `n-1` is the number of steps of size `h` to be taken. `g(k,t)` is a user-supplied external function that returns $g_k(t)$, while `ak(k,l,t,s)` is another user-supplied external function that returns the (k,l) element of the matrix $K(t,s)$. The solution is returned in `f[1..m][1..n]`, with the corresponding abscissas in `t[1..n]`.

```
{
```

```
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,j,k,l,*indx;
    float d,sum,**a,*b;

    indx=ivector(1,m);
    a=matrix(1,m,1,m);
    b=vector(1,m);
    t[1]=t0;
    for (k=1;k<=m;k++) f[k][1]=(*g)(k,t[1]);        Initialize.
    for (i=2;i<=n;i++) {                             Take a step h.
        t[i]=t[i-1]+h;
        for (k=1;k<=m;k++) {
            sum=(*g)(k,t[i]);                        Accumulate right-hand side of linear
            for (l=1;l<=m;l++) {                        equations in sum.
                sum += 0.5*h*(*ak)(k,l,t[i],t[1])*f[l][1];
                for (j=2;j<i;j++)
                    sum += h*(*ak)(k,l,t[i],t[j])*f[l][j];
                a[k][l]=(k == l)-0.5*h*(*ak)(k,l,t[i],t[i]);    Left-hand side goes
            }                                                      in matrix a.
            b[k]=sum;
        }
        ludcmp(a,m,indx,&d);                         Solve linear equations.
        lubksb(a,m,indx,b);
        for (k=1;k<=m;k++) f[k][i]=b[k];
    }
    free_vector(b,1,m);
    free_matrix(a,1,m,1,m);
    free_ivector(indx,1,m);
}
```

For nonlinear Volterra equations, equation (18.2.4) holds with the product $K_{ii}f_i$ replaced by $K_{ii}(f_i)$, and similarly for the other two products of K's and f's. Thus for each i we solve a nonlinear equation for f_i with a known right-hand side. Newton's method (§9.4 or §9.6) with an initial guess of f_{i-1} usually works very well provided the stepsize is not too big.

Higher-order methods for solving Volterra equations are, in our opinion, not as important as for Fredholm equations, since Volterra equations are relatively easy to solve. However, there is an extensive literature on the subject. Several difficulties arise. First, any method that achieves higher order by operating on several quadrature points simultaneously will need a special method to get started, when values at the first few points are not yet known.

Second, stable quadrature rules can give rise to unexpected instabilities in integral equations. For example, suppose we try to replace the trapezoidal rule in the algorithm above with Simpson's rule. Simpson's rule naturally integrates over an interval $2h$, so we easily get the function values at the even mesh points. For the odd mesh points, we could try appending one panel of trapezoidal rule. But to which end of the integration should we append it? We could do one step of trapezoidal rule followed by all Simpson's rule, or Simpson's rule with one step of trapezoidal rule at the end. Surprisingly, the former scheme is unstable, while the latter is fine!

A simple approach that can be used with the trapezoidal method given above is Richardson extrapolation: Compute the solution with stepsize h and $h/2$. Then, assuming the error scales with h^2, compute

$$f_{\mathrm{E}} = \frac{4f(h/2) - f(h)}{3} \tag{18.2.5}$$

This procedure can be repeated as with Romberg integration.

The general consensus is that the best of the higher order methods is the *block-by-block method* (see [1]). Another important topic is the use of variable stepsize methods, which are much more efficient if there are sharp features in K or f. Variable stepsize methods are quite a bit more complicated than their counterparts for differential equations; we refer you to the literature [1,2] for a discussion.

You should also be on the lookout for singularities in the integrand. If you find them, then look to §18.3 for additional ideas.

CITED REFERENCES AND FURTHER READING:

Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S.I.A.M.). [1]

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [2]

18.3 Integral Equations with Singular Kernels

Many integral equations have singularities in either the kernel or the solution or both. A simple quadrature method will show poor convergence with N if such singularities are ignored. There is sometimes art in how singularities are best handled.

We start with a few straightforward suggestions:

1. Integrable singularities can often be removed by a change of variable. For example, the singular behavior $K(t,s) \sim s^{1/2}$ or $s^{-1/2}$ near $s = 0$ can be removed by the transformation $z = s^{1/2}$. Note that we are assuming that the singular behavior is confined to K, whereas the quadrature actually involves the product $K(t,s)f(s)$, and it is this product that must be "fixed." Ideally, you must deduce the singular nature of the product before you try a numerical solution, and take the appropriate action. Commonly, however, a singular kernel does *not* produce a singular solution $f(t)$. (The highly singular kernel $K(t,s) = \delta(t-s)$ is simply the identity operator, for example.)

2. If $K(t,s)$ can be factored as $w(s)\overline{K}(t,s)$, where $w(s)$ is singular and $\overline{K}(t,s)$ is smooth, then a Gaussian quadrature based on $w(s)$ as a weight function will work well. Even if the factorization is only approximate, the convergence is often improved dramatically. All you have to do is replace `gauleg` in the routine `fred2` by another quadrature routine. Section 4.5 explained how to construct such quadratures; or you can find tabulated abscissas and weights in the standard references [1,2]. You must of course supply $\overline{K}$ instead of K.

This method is a special case of the *product Nystrom method* [3,4], where one factors out a singular term $p(t,s)$ depending on both t and s from K and constructs suitable weights for its Gaussian quadrature. The calculations in the general case are quite cumbersome, because the weights depend on the chosen $\{t_i\}$ as well as the form of $p(t,s)$.

We prefer to implement the product Nystrom method on a uniform grid, with a quadrature scheme that generalizes the extended Simpson's 3/8 rule (equation 4.1.5) to arbitrary weight functions. We discuss this in the subsections below.

3. Special quadrature formulas are also useful when the kernel is not strictly singular, but is "almost" so. One example is when the kernel is concentrated near $t = s$ on a scale much smaller than the scale on which the solution $f(t)$ varies. In that case, a quadrature formula can be based on locally approximating $f(s)$ by a polynomial or spline, while calculating the first few *moments* of the kernel $K(t,s)$ at the tabulation points t_i. In such a scheme the narrow width of the kernel becomes an asset, rather than a liability: The quadrature becomes exact as the width of the kernel goes to zero.

4. An infinite range of integration is also a form of singularity. Truncating the range at a large finite value should be used only as a last resort. If the kernel goes rapidly to zero, then