Chapter 6

Numerical Integration

In many computational economic applications, one must compute the definite integral of a real-valued function f defined on some interval I of $\Re^n$. Some applications call for finding the area under the function, as in computing the change in consumer welfare caused by a price change. In these instances, the integral takes the form

$$\int_I f(x)\ dx.$$

Other applications call for computing the expectation of a function of a random variable with a continuous probability density p. In these instances the integral takes the form

$$\int_I f(x)p(x)\ dx.$$

We will discuss only the three numerical integration techniques most commonly encountered in practice. Newton-Cotes quadrature techniques employ a strategy that is a straightforward generalization of Riemann integration principles. Newton-Cotes methods are easy to implement. They are not, however, the most computationally efficient way of computing the expectation of a smooth function. In such applications, the moment-matching strategy embodied by Gaussian quadrature is often substantially superior. Finally, Monte Carlo integration methods are discussed. Such methods are simple to implement and are particularly useful for complex high-dimensional integration, but should only be used as a last resort.

6.1 Newton-Cotes Quadrature

Consider the problem of computing the definite integral

$$\int_{[a,b]} f(x)\ dx$$

of a real-valued function f over a bounded interval $[a, b]$ on the real line.

Newton-Cotes quadrature rules approximate the integrand f with a piecewise polynomial function $\hat{f}$ and take the integral of $\hat{f}$ as an approximation of the integral of f. Two Newton-Cotes rules are widely used in practice: the trapezoidal rule and Simpson's rule. Both rules are very easy to implement and are typically adequate for computing the area under a curve.

The trapezoidal rule approximates f using a piecewise linear function. To implement the trapezoidal rule, the integration interval $[a, b]$ is partitioned into n subintervals of equal length $h = (b - a)/n$. The function is evaluated at the partition points $x_i = a + (i/n)h$, yielding the values $y_i = f(x_i)$ for $i = 0, 1, \ldots, n$. The function f is approximated over each subinterval by linearly interpolating these function values. The space under each line defines a trapezoid with an areas of the form $0.5h(y_i + y_{i+1})$. The areas of all the trapezoids are then summed to form an estimate of the integral of f:

$$\int_{[a,b]} f(x) \approx \frac{h}{2}(y_0 + 2y_1 + 2y_2 + \ldots + 2y_{n-1} + y_n).$$

Figure 6.1: Trapezoidal rule.

The trapezoidal rule is not only simple but also fairly robust. Other Newton-Cotes methods will be more accurate if the integrand f is smooth. However, the trapezoidal rule will often be more accurate if the integrand exhibits discontinuities in its first derivative, which can occur in dynamic economic applications. The trapezoidal rule is said to be first order exact because it exactly computes the integral of any first order polynomial, that is, a line. In general, if the integrand is smooth, the trapezoidal rule yields an approximation error that is $O(1/n^2)$, that is, the error shrinks quadratically with the number of partitions.

Simpson's rule is similar to the trapezoidal rule, except that it approximates the integrand f using a piecewise quadratic, rather than a piecewise linear function. As before, the integration interval is partitioned into n subintervals of equal length $h = (b-a)/n$, where, in this case, n is required to be even. The function is evaluated at the partition points $x_i = a + (i/n)h$, yielding the values $y_i = f(x_i)$ for $i = 0, 1, \ldots, n$. A series of locally quadratic approximations to the integrand are formed by interpolating the function values at successive triplets of the partition points. The area under each quadratic piece is of the form $\frac{h}{3}(y_i + 4y_{i+1} + y_{i+2})$. The areas of all these pieces are then summed to form an estimate of the integral of f:

$$\int_{[a,b]} f(x) \approx \frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 \ldots + 2y_{n-2} + 4y_{n-1} + y_n).$$

Figure 6.2: Simpson's rule.

Simpson's rule is almost as simple as the trapezoidal rule, and thus not much harder to program. Simpson's rule will yield more accurate approximation than the trapezoidal rule if the integrand is smooth. Even though Simpson's rule is based on locally quadratic approximation of the integrand, it is third order exact. That is, it exactly computes the integral of any third order (e.g., cubic) polynomial. In general, if the integrand is smooth, Simpson's rule yields an approximation error that is $O(1/n^4)$, and thus falls at twice the geometric rate as the error associated with the trapezoidal rule. Simpson's rule is the Newton-Cotes rule most often used in practice because it retains algorithmic simplicity while offering an adequate degree of approximation. Newton-Cotes rules of higher order may be defined, but are more difficult to work with and thus are rarely used.

In most computational economic applications, it is not possible to determine a priori how many partition points are needed to compute an indefinite integral to a desired level of accuracy. One solution to this problem is to use an *adaptive quadrature* strategy whereby one increases the number of points at which the integrand is evaluated until the sequence of estimates of the definite integral converge. Efficient adaptive Newton-Cotes quadrature schemes are specially easy to implement. One simple, but powerful, scheme calls for the number of intervals to be doubled with each iteration. Because the new

partition points include the partition points used in the previous iteration, the computational effort required to form the new integral estimate is cut in half. More sophisticated adaptive Newton-Cotes quadrature techniques relax the requirement that the intervals be equally spaced and concentrate new evaluation points in those areas where the integrand appears to be most irregular.

Matlab offers two Newton-Cotes quadrature routines, `quad` and `quad8` both of which are based on an adaptive Simpson's rule.

6.2 Gaussian Quadrature

Consider the problem of computing the expectation of a real-valued function f of a random variable $\tilde{X}$ with probability density function $p(\cdot)$:

$$Ef(\tilde{X}) = \int f(x)p(x)\ dx.$$

Gaussian quadrature is a numerical integration technique that calls for $\tilde{X}$ to be replaced with a discrete random variable whose distribution matches that of $\tilde{X}$ as closely as possible. Specifically, in a Gaussian quadrature scheme, the mass points $x_1, x_2, \ldots, x_n$ and probabilities $p_1, p_2, \ldots, p_n$ of the discrete approximant are chosen in such a way that the approximant possesses the same moments of order $2n-1$ and below as $\tilde{X}$:

$$\sum_{i=1}^{n} p_i x_i^k = E(\tilde{X}^k) \qquad k = 0, \ldots, 2n-1.$$

Given the mass points and probabilities of the discrete approximant, the expectation of $f(\tilde{X})$ is approximated as follows:

$$Ef(\tilde{X}) = \int f(x)p(x)\ dx \approx \sum_{i=1}^{n} f(x_i)p_i.$$

Computing the n-degree Gaussian mass points x_i and probabilities p_i for a random variable involves solving $2n$ nonlinear equations in as many unknowns. Efficient, specialized numerical routines for computing Gaussian mass points and probabilities are available for virtually every major distribution, including the normal, uniform, gamma, exponential, Chi-square, and beta distributions. In our applications, we will be exclusively concerned with computing expectations of functions of normal random variates and related

random variates. For this purpose, we supply a Matlab routine `gaussnrm` that may be used to compute, in vector format, the n-degree Gaussian mass points `x` and probabilities `p` for normal random variates. The routine is called as follows:

```
[x,p] = gaussnrm(n,mu,sigma);
```

Here, `n` is the degree of approximation and `mu` and `sigma` are the mean and standard deviation of the distribution, respectively. If `mu` and `sigma` are not specified, it is assumed that the mean is zero and the standard deviation is one.

By design, an n-point Gaussian quadrature rule will compute the expectation of $f(\tilde{X})$ exactly if f is a polynomial of order $2n-1$ or less. Thus, if f can be closely approximated by a polynomial, the Gaussian quadrature rule should provide an accurate estimate of the expectations. Gaussian quadrature rules are consistent for Riemann integrable functions. That is, if f is Riemann integrable, then the approximation afforded by Gaussian quadrature can be made arbitrarily precise simply by increasing the number of mass points in the discretizing distribution.

Gaussian quadrature is the numerical integration method of choice when the integrand is bounded and possesses continuous derivatives, but should be applied with great caution otherwise. If the integrand is unbounded, it is often possible to transform the integration problem into an equivalent one with bounded integrand. If the function possesses known kink points, it is often possible to break the integral into the sum of two integrals of smooth functions. If these or similar steps do not produce smooth, bounded integrands, then Newton-Cotes quadrature methods may be more accurate than Gaussian quadrature because they contain the error caused by the kinks and singularities to the interval in they occur.

The Gaussian quadrature scheme for normal variates may be used to develop a good scheme for lognormal random variates. By definition, $\tilde{Y}$ is lognormally distributed with parameters μ and σ if, and only if, it is distributed as $\exp(\tilde{X})$ were $\tilde{X}$ is normally distributed with mean μ and standard deviation σ. It follows that if $\{x_i, p_i\}$ are Gaussian mass points and probabilities for a normal(μ, σ) distribution, then $\{y_i, p_i\}$, where $y_i = \exp(x_i)$, provides a reasonable a discrete approximant for a lognormal(μ, σ) distribution. Given this discrete approximant for the lognormal distribution, one can estimate

the expectation of a function of $\tilde{Y}$ follows:

$$Ef(\tilde{Y}) = \int f(y)p(y)\ dy \approx \sum_{i=1}^{n} f(y_i)p_i.$$

This integration rule for lognormal distributions will be exact if f is a polynomial of degree $2n-1$ and less in $\log(y)$.

Through the use of tensor product principles, the Gaussian quadrature scheme for the univariate normal distribution may be used to construct a Gaussian quadrature scheme for the multivariate normal distribution. Suppose, for example, that $\tilde{X} = (\tilde{X}_1, \tilde{X}_2)$ is a bivariate normal random variable in row form with (row) mean vector μ and variance-covariance matrix Σ. Then $\tilde{X}$ is distributed as $\mu + \tilde{Z} * R$ where R is the Cholesky square root of Σ (e.g., $\Sigma = R'R$) and $\tilde{Z}$ is a row 2-vector of independent standard normal variates. If $\{z_i, p_i\}$ are the degree n Gaussian mass points and weights for a standard normal variate, then an n^2 degree discrete approximant to $\tilde{X}$ may be constructed by forming the mass points

$$x_{ij} = (\mu_1 + R_{11}z_i + R_{21}z_j, \mu_2 + R_{12}z_i + R_{22}z_j)$$

and associating to them the probabilities

$$p_{ij} = p_i p_j.$$

A similar procedure may be used for multivariate normal random variates of higher dimension.

6.3 Monte Carlo Integration

Monte Carlo integration methods are motivated by the Strong Law of Large Numbers. One version of the Law states that if $x_1, x_2, \ldots$ are independent realizations of a random variable $\tilde{X}$ and f is a continuous function, then

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(x_i) = Ef(\tilde{X})$$

with probability one.

The Monte Carlo integration scheme is thus a simple one. To compute an approximation to the integral of $f(\tilde{X})$, one draws a random sample $x_1, x_2, \ldots, x_n$ from the distribution of $\tilde{X}$ and sets

$$Ef(\tilde{X}) \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i).$$

A fundamental problem that arises with Monte Carlo integration is that it is almost impossible to generate a truly random sample of variates for any distribution. Most compilers and vector processing packages provide intrinsic routines for computing so-called random numbers. These routines, however, employ iteration rules that generate a purely deterministic, not random, sequence of numbers. In particular, if the generator is repeatedly initiated at the same point, it will return the same sequence of 'random' variates each time. About all that can be said of numerical random number generators is that good ones will generate sequences that appear to be random, in that they pass certain statistical tests for randomness. For this reason, numerical random number generators are often more accurately said to generate sequences of 'pseudo-random' rather than random numbers.

Matlab offers two intrinsic random number generators. The routine `rand` generates a random sample from the uniform(0,1) distribution stored in either vector or matrix format. Similarly, the routine `randn` generates a random sample from the standard normal distribution stored in either vector or matrix format. In particular, a call of the form `x=rand(n)` or `x=randn(n)` generates a random sample of n realizations and stores it in a row vector.

The uniform random number generator is useful for generating random samples from other distributions. Suppose $\tilde{X}$ has a cumulative distribution function

$$F(x) = \Pr(\tilde{X} \leq x)$$

whose inverse has a well-defined closed form. If $\tilde{U}$ is uniformly distributed on $(0, 1)$, then

$$F^{-1}(\tilde{U})$$

is distributed as $\tilde{X}$. Thus, to generate a random sample $x_1, x_2, \ldots, x_n$ from the $\tilde{X}$ distribution, one generates a random sample $u_1, u_2, \ldots, u_n$ from the uniform distribution and sets $x_i = F^{-1}(u_i)$.

The standard normal random number generator is useful for generating random samples from related distributions. For example, to generate a random sample of n lognormal variates, one may use the script

```
x = exp(mu+sigma*randn(n));
```

where `mu` and `sigma` are the parameters of the distribution. To generate a random sample of n d-dimensional normal variates one may use the script

```
x = mu+randn(d,n)*chol(S);
```

where `S` is the d by d variance-covariance matrix and `mu` is the mean vector in row form.

Monte Carlo integration is easy to implement and may be preferred over Gaussian quadrature if the a routine for computing the Gaussian mass points and probabilities is not readily available or if the integration is over many dimensions. Monte Carlo integration, however, is subject to a sampling error that cannot be bounded with certainty. The approximation can be made more accurate, in a statistical sense, by increasing the size of the random sample, but this can be expensive if evaluating f or generating the pseudo-random variate is costly. Approximations generated by Monte Carlo integration will vary from one integration to the next, unless initiated at the same point, making the use of Monte Carlo integration in conjunction within other iterative schemes, such as dynamic programming or maximum likelihood estimation, problematic. So-called quasi Monte-Carlo methods can circumvent some of the problems associated with Monte-Carlo integration.