Chapter 11

Numerical and Monte Carlo Methods

20 June 2005

Simple classical and Monte Carlo methods including importance sampling are illustrated in the context of the numerical evaluation of definite integrals.

11.1 Numerical Integration Methods in One Dimension

In this chapter we will learn that we can use sequences of random numbers to estimate definite integrals, a problem that seemingly has nothing to do with randomness. To place the Monte Carlo numerical integration methods in perspective, we first will discuss several common classical methods for numerically evaluating definite integrals. We will find that these methods, although usually preferable in low dimensions, are impractical for multidimensional integrals and that Monte Carlo methods are essential for the evaluation of the latter if the number of dimensions is sufficiently high.

Consider the one-dimensional definite integral of the form

$$F = \int_a^b f(x)\, dx. \tag{11.1}$$

For some choices of the integrand $f(x)$, the integration in (11.1) can be done analytically, found in tables of integrals, or evaluated as a series. However, there are relatively few functions that can be evaluated analytically and most functions must be integrated numerically.

Most classical methods of numerical integration are based on the geometrical interpretation of the integral (11.1) as the area under the curve of the function $f(x)$ from $x = a$ to $x = b$ (see

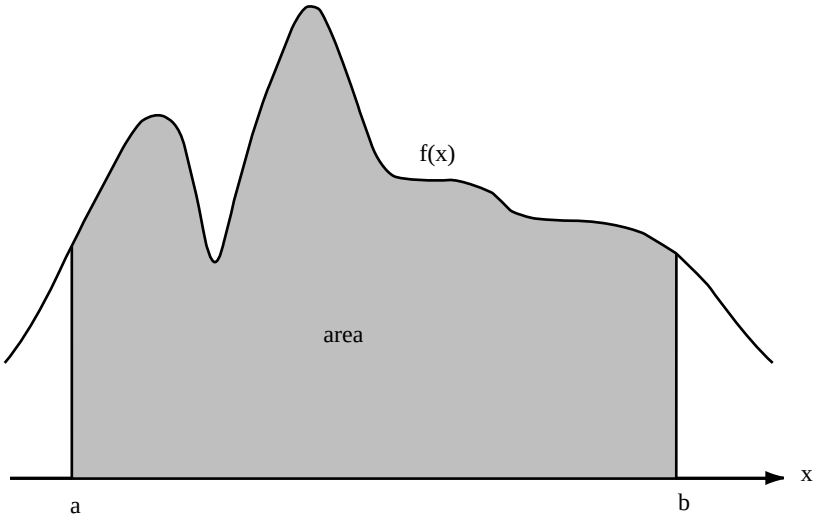

Figure 11.1: The integral F equals the area under the curve $f(x)$.

Figure 11.1). In these methods the x-axis is divided into n equal intervals of width Δx, where Δx is given by

$$\Delta x = \frac{b-a}{n}, \tag{11.2a}$$

and

$$x_n = x_0 + n\,\Delta x. \tag{11.2b}$$

In the above, $x_0 = a$ and $x_n = b$.

The simplest approximation of the area under the curve $f(x)$ is the sum of the area of the rectangles shown in Figure 11.2. In the *rectangular* approximation, $f(x)$ is evaluated at the *beginning* of the interval, and the approximate of the integral, F_n, is given by

$$F_n = \sum_{i=0}^{n-1} f(x_i)\Delta x. \qquad \text{(rectangular approximation)} \tag{11.3}$$

In the *trapezoidal* approximation the integral is approximated by a sum of trapezoids. The area is computed by choosing one side equal to $f(x)$ at the beginning of the interval and the other side equal to $f(x)$ at the end of the interval. This approximation is equivalent to replacing the function by a straight line connecting the values of $f(x)$ at the beginning and the end of each interval. Because the area of the trapezoid from x_i to x_{i+1} is given by $\frac{1}{2}[f(x_{i+1}) + f(x_i)]\Delta x$, the total area F_n of the trapezoids is given by

$$F_n = \big[\frac{1}{2}f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(x_n)\big]\Delta x. \quad \text{(trapezoidal approximation)} \tag{11.4}$$

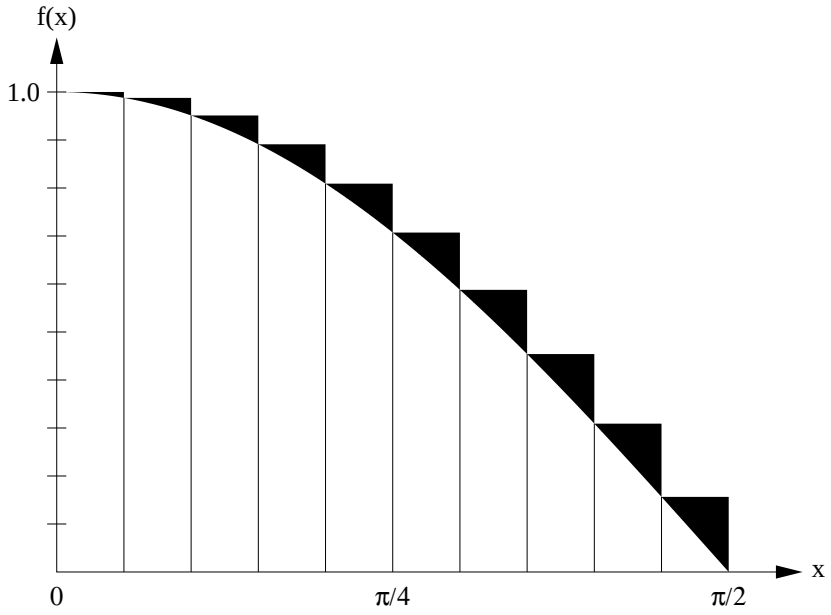

Figure 11.2: The rectangular approximation for $f(x) = \cos x$ for $0 \le x \le \pi/2$. The error is shaded. The error for various values of the number of intervals n is given in Table 11.1.

A generally more accurate method is to use a quadratic or parabolic interpolation procedure through adjacent triplets of points. (The general problem of interpolation between data points using polynomials is discussed in Appendix 11D.) For example, the equation of the second-order polynomial that passes through the points (x_0, y_0), (x_1, y_1), and (x_2, y_2) can be written as

$$\begin{aligned} y(x) \;=\; & \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 \\ & + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2. \end{aligned} \tag{11.5}$$

What is the value of $y(x)$ at $x = x_1$? The area under the parabola $y(x)$ between x_0 and x_2 can be found by integration and is given by

$$F_0 = \frac{1}{3}\big(y_0 + 4y_1 + y_2\big)\Delta x, \tag{11.6}$$

where $\Delta x = x_1 - x_0 = x_2 - x_1$. The total area under all the parabolic segments yields the parabolic approximation for the total area:

$$\begin{aligned} F_n \;=\; & \frac{1}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots \\ & + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]\Delta x. \qquad \text{(Simpson's rule)} \end{aligned} \tag{11.7}$$

This approximation is known as *Simpson's rule*, although a more descriptive name would be the *parabolic approximation*. Note that Simpson's rule requires that n be even.

To write a program that implements the rectangular approximation, we must define the function we wish to integrate. Although we could define a new integration class for each function (or change the function in the class and recompile each time), it is convenient to input the name of the function as a string and then *parse* the string so that the function can be evaluated. The `ParsedFunction` class in the numerics package is designed for this task.

```
String str = "cos(x)"; // default string; this string could be an input
Function f;
try {
   f = new ParsedFunction(str);
   } catch (ParserException ex) {
    // recover if str does not represent a valid function
   }
```

Because the `ParsedFunction` often is used with keyboard input and it is common for users to mistype the name of a function, the `ParsedFunction` constructor throws an exception that must be caught.

One way to display a function in a drawing panel is to evaluate the function $f(x)$ at various x values and plot the $(x, f(x))$ data points. Although we could do so using a loop to add a predetermined number of points to a data set, a better way is to use the `FunctionDrawer` class in the display package. The `FunctionDrawer` evaluates a given function at every pixel location within a drawing panel thereby producing a plot with optimum resolution.

```
// drawingPanel created previously
drawingPanel.addDrawable(new FunctionDrawer(f));
```

We next define the class `RectangularApproximation` which computes the area under the curve using the rectangular approximation. This class also displays the rectangles used to compute the area. Note how we have extended the `Dataset` class to produce the visualization.

Listing 11.1: The class `RectangularApproximation` illustrates the nature of the rectangular approximation.

```
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.display.Dataset;
import org.opensourcephysics.numerics.Function;

public class RectangularApproximation extends Dataset {
   double sum = 0; // will contain the integral

   public RectangularApproximation(Function f, double a, double b, int n) {
      setMarkerColor(new java.awt.Color(255, 0, 0, 128)); // transparent red
      setMarkerShape(Dataset.AREA);
      sum = 0;
      double x = a; // lower limit
      double y = f.evaluate(a);
      double dx = (b-a)/n;
      // use methods in Dataset superclass
```

```
      append(x, 0); // start on the x axis
      append(x, y); // the top left hand corner of the first rectangle
      while(x<b) {          // b is the upper limit
         x += dx;
         append(x, y);      // top right hand corner of current rectangle
         sum += y;
         y = f.evaluate(x); // calculate a new y at the new x
         append(x, y);      // the top left hand corner of the next rectangle
      }
      append(x, 0); // finish on the x axis
      sum *= dx;
   }
}
```

Listing 11.2: The target class is defined in the `NumericalIntegrationApp` class.

```
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.*;

public class NumericalIntegrationApp extends AbstractCalculation {
   PlotFrame plotFrame = new PlotFrame("x", "f(x)", "Numerical integration visualization");

   public void reset() {
      control.setValue("f(x)", "cos(x)");
      control.setValue("lower limit a", 0);
      control.setValue("upper limit b", Math.PI/2);
      control.setValue("number of intervals n", 4);
   }

   public void calculate() {
      String fstring = control.getString("f(x)");
      double a = control.getDouble("lower limit a");
      double b = control.getDouble("upper limit b");
      int n = control.getInt("number of intervals n");
      Function f;
      try {
         f = new ParsedFunction(fstring);
      } catch(ParserException ex) {
         control.println(ex.getMessage());
         plotFrame.clearDrawables();
         return;
      }
      plotFrame.clearDrawables();
      plotFrame.setPreferredMinMaxX(a, b); // sets the domain of x to the integration limits
      plotFrame.addDrawable(new FunctionDrawer(f));
      RectangularApproximation approximate = new RectangularApproximation(f, a, b, n);
      plotFrame.addDrawable(approximate);
```

```
      plotFrame.setMessage("area = "+decimalFormat.format(approximate.sum));
      control.println("approximate area under curve = "+approximate.sum);
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new NumericalIntegrationApp());
   }
}
```

We consider the accuracy of the rectangular approximation for the integral of $f(x) = \cos x$ from $x = 0$ to $x = \pi/2$ by comparing the numerical results shown in Table 11.1 with the exact answer of unity. Note that the error decreases as n^{-1}. This observed n^{-1} dependence of the error is consistent with the analytical derivation of the n dependence of the error obtained in Appendix 11A. We explore the n dependence of the error associated with other numerical integration methods in Problems 11.1 and 11.2.

n	F_n	Δ_n
2	1.34076	0.34076
4	1.18347	0.18347
8	1.09496	0.09496
16	1.04828	0.04828
32	1.02434	0.02434
64	1.01222	0.01222
128	1.00612	0.00612
256	1.00306	0.00306
512	1.00153	0.00153
1024	1.00077	0.00077

Table 11.1: Rectangular approximations of the integral of $\cos x$ from $x = 0$ to $x = \pi/2$ as a function of n, the number of intervals. The error Δ_n is the difference between the rectangular approximation and the exact result of unity. Note that the error Δ_n decreases approximately as n^{-1}, that is, if n is increased by a factor of 2, Δ_n is decreased by a factor 2.

Problem 11.1. The rectangular and midpoint approximations

a. Test `NumericalIntegrationApp` by reproducing the results in Table 11.1.

b. Use the rectangular approximation to determine the approximate definite integrals of $f(x) = 2x + 3x^2 + 4x^3$ and $f(x) = e^{-x}$ for $0 \le x \le 1$ and $f(x) = 1/x$ for $1 \le x \le 2$. What is the approximate n dependence of the error in each case?

c. A straightforward modification of the rectangular approximation is to evaluate $f(x)$ at the *midpoint* of each interval. Define a `MidpointApproximation` class by making the necessary modifications and approximate the integral of $f(x) = \cos x$ in the interval $0 \le x \le \pi/2$. Remember that you need to change how the rectangles are drawn. How does the magnitude of the error compare with the results shown in Table 11.1? What is the approximate dependence of the error on n?

d. Use the midpoint approximation to determine the definite integrals considered in part (b). What is the approximate n dependence of the error in each case?

Problem 11.2. The trapezoidal approximation

a. Modify your program so that the trapezoidal approximation is computed and determine the n-dependence of the error for the same functions as in Problem (11.1). What is the approximate n dependence of the error in each case? Which approximation yields the best results for the same computation time?

b. It is possible to double the number of intervals without losing the benefit of the previous calculations. For $n = 1$, the trapezoidal approximation is proportional to the average of $f(a)$ and $f(b)$. In the next approximation the value of f at the midpoint is added to this average. The next refinement adds the values of f at the 1/4 and 3/4 points. Modify your program so that the number of intervals is doubled each time and the results of previous calculations are used. The following should be helpful:

```
if (n == 1) {
   double midpoint = a + 0.5*(b-a);
   sum = (b - a)*(0.5*f(a) + 0.5*f(b) + f(midpoint));
   n = 2;
} else {
   double delta = (b - a)/n;     // separation between new points
   double x = a + 0.5*delta;
   double intermediateSum = 0.0;
   for (int i = 0; i < n; i++) {
      intermediateSum +=  f(x);
      x += delta;
   }
   sum = 0.5*(intermediateSum*delta + sum);  // 0.5*delta = separation between points
   n *= 2;
}
```

The rectangular and trapezoidal algorithms converge relatively slowly and are therefore not recommended in general. In practice, Simpson's rule is adequate for functions that are reasonably well behaved, that is, functions that can be adequately represented by a polynomial. If $f(x)$ is such a function, we can adopt the strategy of evaluating the area for a given number of intervals n and then doubling the number of intervals and evaluating the area again. If the two evaluations are sufficiently close to one another, we stop. Otherwise, we again double n until we achieve the desired accuracy. Of course, this strategy will fail if $f(x)$ is not well behaved. An example of a poorly behaved function is $f(x) = x^{-1/3}$ at $x = 0$, where $f(x)$ diverges. Another example where this strategy might fail is when a limit of integration is equal to $\pm\infty$. In many cases we can eliminate the problem by a change of variables.

Problem 11.3. Simpson's rule

a. Write a class that implements Simpson's rule. Either adapt your program so that it uses Simpson's rule directly or convince yourself that the result of Simpson's rule can be expressed as

$$S_n = (4T_{2n} - T_n)/3, \tag{11.8}$$

where T_n is the result from the trapezoidal approximation for n intervals. It is not necessary to provide a visualization of the area. Use your program to evaluate the integral of $f(x) = (2\pi)^{-1/2}\, e^{-x^2}$ for $|x| \leq 1$. Do you obtain the same result by choosing the interval $[0, 1]$ and then multiplying by two?

b. Determine the same integrals as in Problem 11.2 and discuss the relative merits of the various approximations.

c. Evaluate the integral of the function $f(x) = 4\sqrt{1 - x^2}$ for $|x| \leq 1$. What value of n is needed for four decimal accuracy? The reason for the slow convergence can be understood by reading Appendix 11A.

c.* Our strategy for approximating the value of the definite integrals we have considered has been to compute F_n and F_{2n} for reasonable values of n. If the difference $|F_{2n} - F_n|$ is too large, then we double n until the desired accuracy is reached. The success of this strategy is based on the implicit assumption that the sequence $F_n, F_{2n}, \cdots$ converges to the true integral F. Is there a way of extrapolating this sequence to the limit? Explore this idea by using the trapezoidal approximation. Because the error for this approximation decreases approximately as n^{-2}, we can write $F = F_n + Cn^{-2}$, and plot F_n as a function of n^{-2} to obtain the extrapolated result F. Apply this procedure to the integrals considered in some of the previous problems and compare your results to those found from the trapezoidal approximation and Simpson's rule alone. A more sophisticated application of this idea is known as *Romberg* integration (see Press et al.).

Because integration is usually a routine task, we have implemented the integration methods discussed in this section in the `Integral` class in the numerics package. Listing 11.3 illustrates how the `Integral` class is used. Method `Integral.ode` computes the integrals using an `ODE` solver as discussed in the following. The solver uses an adaptive step size to achieve the desired tolerance.

Listing 11.3: The `IntegralCalcApp` program tests the `Integral` class.

```
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.numerics.*;

public class IntegralCalcApp extends AbstractCalculation {
   public void reset() {
      control.setValue("a", 0);
      control.setValue("b", 1);
      control.setValue("tolerance", 1.0e-2);
      control.setValue("f(x)", "sin(2*pi*x)^2");
   }

   public void calculate() {
```

```
      Function f;
      String fx = control.getString("f(x)");
      try { // read in function to integrate
         f = new ParsedFunction(fx);
      } catch(ParserException ex) {
         control.println(ex.getMessage());
         return;
      }
      double a = control.getDouble("a");
      double b = control.getDouble("b");
      double tolerance = control.getDouble("tolerance");
      double area = Integral.ode(f, a, b, tolerance);
      control.println("ODE area = "+area);
      area = Integral.trapezoidal(f, a, b, 2, tolerance);
      control.println("Trapezoidal area = "+area);
      area = Integral.simpson(f, a, b, 2, tolerance);
      control.println("Simpson area = "+area);
      area = Integral.romberg(f, a, b, 2, tolerance);
      control.println("Romberg area ="+area);
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new IntegralCalcApp());
   }
}
```

The algorithms in the `Integral` class are implemented using static methods so that they are easy to invoke. These methods accept a *tolerance* parameter that allows the user to specify the acceptable relative precision. Because computers store floating point numbers using a fixed number of decimal places, we use relative precision to determine the accuracy of the integration method. However, relative precision is meaningful only if the result is different from zero. If the result is zero, the only possible check is for absolute precision. Because numerical integration algorithms should check the precision of their output, the `Util` class in the numerics package defines the following general purpose method for computing the relative precision from an absolute precision and a numerical result.

```
public static double relativePrecision(final double epsilon, final double result) {
   return (result > defaultNumericalPrecision)
      ? epsilon/result   // return epsilon/result if (...) is true
      : epsilon;         // else return epsilon
}
```

The `defaultNumericalPrecision` is a named constant that is equal to `Math.sqrt(Double.MIN_VALUE)`.

Problem 11.4. The `Integral` class

Add a static counter to keep track of the number of times the test function is evaluated in the `IntegralCalcApp` class. Use this counter to compare the efficiencies of the various integration algorithms.

a. How many function calls are required for each numerical method if the tolerance is set to 10^{-3} or 10^{-12}?

b. How does the execution time depend on the numerical method? You can compute the time using the statement `System.currentTimeMillis()`, which is the time between the current time and January 1, 1970 measured in milliseconds. Note that the return value is `long` and not `int`.

c. The method `Integral.ode` determines when it has reached the input tolerance differently than the other integration algorithms in the `Integral` class. Compare the results from these other integrators. Is the accuracy of each method always within the tolerance set by the user?

Another way of evaluating one-dimensional integrals is to recast them as differential equations. Consider an indefinite integral of the form

$$F(x) = \int_a^x f(t)\,dt, \tag{11.9}$$

where $F(a) = 0$. If we differentiate $F(x)$ with respect to x, we obtain the first-order differential equation:

$$\frac{dF(x)}{dx} = f(x). \tag{11.10}$$

with the boundary condition

$$F(a) = 0. \tag{11.11}$$

Because the function $f(x)$ is known, (11.10) can be solved for $F(x)$ using the numerical algorithms that we introduced earlier for obtaining the numerical solutions of first-order differential equations.

In general, the methods for solving ordinary differential equations and evaluating numerical integrals are not equivalent. For example, we cannot use Simpson's rule to obtain the numerical solution of an differential equation of the form $dy/dx = f(x, y)$. Why?

Simpson's rule fails (or is slowly convergent) if the function has regions of rapid change. In this case, an adaptive step-size ODE algorithm is usually effective. The `RK45MultiStep` solver adapts the integration step to the behavior of the integrand and allows the user to set the tolerance. In Problem 11.5 we use it to examine an approximation to the Dirac delta function.

Problem 11.5. Delta function approximation

The Dirac delta function can be approximated by the function:

$$\delta(x) = \frac{1}{\pi} \lim_{\epsilon \to 0} \frac{\epsilon}{x^2 + \epsilon^2}. \tag{11.12}$$

Test this approximation by integrating (11.12) over a suitable range of x using various values of ϵ. Try adaptive ODE solvers with a tolerance of 10^{-8}. How small a value of ϵ produces an error of 10^{-4}? 10^{-5}?

As computers become more powerful and software to perform mathematical operations and numerical analysis becomes more prevalent, there is a strong temptation to use libraries written

by others. This use is appropriate because usually these libraries are well written, and there is no reason why a user should re-invent them. However, the downside is that users do not always know or care what the libraries are doing and sometimes can obtain puzzling or even incorrect results. In Problem 11.6 we explore this issue.

Problem 11.6. Understanding errors in integration routines

a. Use the `ode`, `simpson`, `trapezoid`, and `rhomberg` methods in the `Integral` class of the Open Source Physics library to estimate the integral of $\sin^2(2\pi x)$ between $x = 0$ and 1 with a tolerance of 0.01. The exact answer is 0.5. Do all four methods return results within the tolerance? Are some results much more accurate than the tolerance? Change the tolerance to 0.1. How do the results change? Notice that for some of the methods, the results are much better than might be expected because the positive and negative errors cancel. Why does the trapezoid method always give the exact answer?

b. Integrate the same function from $x = 0.2$ to 1.0. How accurate are the results now? Explore how the results change with the input tolerance. Does the behavior of the `ode` integrator differ from the others. Why? How do you think the tolerance parameter is used for each of the methods?

c. Integrate $f(x) = x^n$ for various values of n. How do the different integrators compare? Why does the trapezoid integrator do worse than the others for $n = 2$?

11.2 Simple Monte Carlo Evaluation of Integrals

We now explore a much different method of estimating integrals. Consider the following example. Suppose an irregularly shaped pond is in a field of known area A. The area of the pond can be estimated by throwing stones so that they land at random within the boundary of the field and counting the number of splashes that occur when a stone lands in a pond. The area of the pond is approximately the area of the field times the fraction of stones that make a splash. This simple procedure is an example of a *Monte Carlo* method.

To be more specific, imagine a rectangle of height h, width $b - a$, and area $A = h(b - a)$ such that the function $f(x)$ is within the boundaries of the rectangle (see Figure 11.3). Compute n pairs of random numbers x_i and y_i with $a \le x_i \le b$ and $0 \le y_i \le h$. The fraction of points x_i, y_i that satisfy the condition $y_i \le f(x_i)$ is an estimate of the ratio of the integral of $f(x)$ to the area of the rectangle. Hence, the estimate F_n in the *hit or miss* method is given by

$$F_n = A\frac{n_s}{n}, \qquad \text{(hit or miss method)} \tag{11.13}$$

where n_s is the number of points below the curve or "splashes," and n is the total number of points. The number of points chosen at random in (11.13) should not be confused with the number of intervals used in the numerical methods discussed in Section 11.1.

Another Monte Carlo integration method is based on a mean-value theorem of integral calculus, which states that the definite integral (11.1) is determined by the average value of the integrand

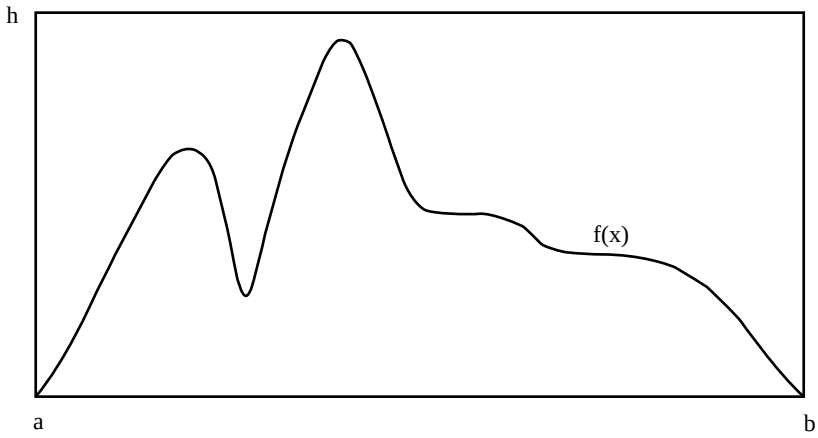

Figure 11.3: The function $f(x)$ is in the domain determined by the rectangle of height H and width $(b-a)$.

$f(x)$ in the range $a \leq x \leq b$:

$$F = \int_a^b f(x)dx = (b-a)\langle f\rangle. \tag{11.14}$$

To determine $\langle f\rangle$, we choose the x_i at random instead of at regular intervals and *sample* the values of $f(x)$. For the one-dimensional integral (11.1), the estimate F_n of the integral in the *sample mean* method is given by

$$F_n = (b-a)\frac{1}{n}\sum_{i=1}^{n} f(x_i) \approx (b-a)\langle f\rangle. \qquad \text{(sample mean method)} \tag{11.15}$$

The x_i are random numbers distributed uniformly in the interval $a \leq x_i \leq b$, and n is the number of samples. Note that the forms of (11.3) and (11.15) are formally identical except that the n points are chosen with equal spacing in (11.3) and with random spacing in (11.15). We will find that for low dimensional integrals (11.3) is more accurate, but for higher dimensional integrals (11.15) does better.

A simple program that implements the hit or miss method is given in Listing 11.4. Note the use of the `Random` class and the methods `setSeed` and `nextDouble()`. As discussed in Chapter 7, the primary reason that it is desirable to specify the seed rather than to choose it more or less at random from the time (as is done by `Math.random()`) is that it is convenient to use the same random number sequence when testing a Monte Carlo program. Suppose that your program gives a strange result for a particular run. If we find an error in the program, we can use the same random number sequence to test whether our program changes make any difference. Another reason for specifying the seed is so that other users can reproduce your results.

Listing 11.4: `MonteCarloEstimatorApp` displays the estimate of the integral for the number of samples equal to 2^p.

```
package org.opensourcephysics.sip.ch11;
```

```
import java.util.Random;
import org.opensourcephysics.controls.*;

public class MonteCarloEstimatorApp extends AbstractSimulation {
   Random rnd = new Random();
   int nSampled; // number of points already sampled
   int nTotal;   // total number of samples
   long seed;
   double a, b;  // interval limits
   double ymax;
   int hits = 0;

   public void reset() {
      control.setValue("lower limit a", 0);
      control.setValue("upper limit b", 1.0);
      control.setValue("upper limit on y", 1.0);
      control.setValue("seed", 137933);
   }

   public double evaluate(double x) {
      return Math.sqrt(1-x*x);
   }

   public void initialize() {
      a = control.getDouble("lower limit a");
      b = control.getDouble("upper limit b");
      ymax = control.getDouble("upper limit on y");
      nSampled = 0;
      nTotal = 2;
      seed = (long) control.getInt("seed");
      hits = 0;
      rnd.setSeed(seed);
   }

   public void doStep() {
      // nextDouble returns random double between 0 (inclusive) and 1 (exclusive)
      for(int i = nSampled;i<nTotal;i++) {
         double x = a+rnd.nextDouble()*(b-a);
         double y = rnd.nextDouble()*ymax;
         if(y<=evaluate(x)) {
            hits++;
         }
      }
      control.println("# of samples = "+nTotal+" estimated area = "+(hits*(b-a)*ymax)/nTotal
      nSampled = nTotal; // number of points sample so far
      nTotal *= 2;       // increase number of samples by a factor of 2 for next step
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new MonteCarloEstimatorApp());
```

```
    }
}
```

Problem 11.7. Monte Carlo integration in one dimension

a. Use `MonteCarloEstimatorApp` to estimate F_n, the integral of $f(x) = 4\sqrt{1-x^2}$ in the interval $0 \le x \le 1$, using the hit or miss method. Choose $a = 0$, $b = 1$, $h = 1$, and compute the mean value of the function $\sqrt{1-x^2}$. Multiply the estimate by 4 to determine F_n. Calculate the difference between F_n and the exact result of π. This difference is a measure of the error associated with the Monte Carlo estimate. Make a log-log plot of the error as a function of n. What is the approximate dependence of the error on n for large n, for example, $n \ge 10^6$?

b. Estimate the same integral using the sample mean method (11.15) and compute the error as a function of the number of samples n for $n \ge 10^6$. How many samples are needed to determine F_n to two decimal places? What is the approximate dependence of the error on n for large n?

c. Determine the computational time per sample using the two Monte Carlo methods. Which Monte Carlo method is preferable?

11.3 Multidimensional Integrals

Many problems in physics involve averaging over many variables. For example, suppose we know the position and velocity dependence of the total energy of ten interacting particles. In three dimensions each particle has three velocity components and three position components. Hence the total energy is a function of 60 variables, and a calculation of the average energy per particle involves computing a $d = 60$ dimensional integral. (More accurately, the total energy is a function of $60 - 6 = 54$ variables if we use center of mass and relative coordinates.) If we divide each coordinate into p intervals, there would be p^{60} points to sum. Clearly, standard numerical methods such as Simpson's rule would be impractical for this example.

A discussion of the n dependence of the error associated with the standard numerical methods for d-dimensional integrals is given in Appendix 11A. We show that if the error decreases as n^{-a} for $d = 1$, then the error decreases as $n^{-a/d}$ in d dimensions. In contrast, we find (see Section 11.4) that the error for all Monte Carlo integration methods decreases as $n^{-1/2}$ *independently* of the dimension of the integral. Because the computational time is roughly proportional to n in both the classical and Monte Carlo methods, we conclude that for low dimensions, the classical numerical methods such as Simpson's rule are preferable to Monte Carlo methods unless the domain of integration is very complicated. However, Monte Carlo methods are essential for higher dimensional integrals.

To illustrate the evaluation of multidimensional integrals, we consider the two-dimensional integral

$$F = \int_R f(x,y)\,dxdy, \tag{11.16}$$

where R denotes the region of integration. The extension to higher dimensions is straightforward, but tedious. Form a rectangle that encloses the region R, and divide this rectangle into squares of length h. Assume that the rectangle runs from x_a to x_b in the x direction and from y_a to y_b in the

y direction. The total number of squares is $n_x n_y$, where $n_x = (x_b - x_a)/h$ and $n_y = (y_b - y_a)/h$. If we use the midpoint approximation, the integral F is estimated by

$$F \approx \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} f(x_i, y_j) H(x_i, y_j)\, h^2, \tag{11.17}$$

where $x_i = x_a + (i - \frac{1}{2})h$, $y_j = y_a + (j - \frac{1}{2})h$, and the (Heaviside) function $H(x, y) = 1$ if (x, y) is in R and equals 0 otherwise.

A simple Monte Carlo method for evaluating a two-dimensional integral uses the same rectangular region as in (11.17), but the n points (x_i, y_i) are chosen at random within the rectangle. The estimate for the integral is then

$$F_n = \frac{A}{n} \sum_{i=1}^{n} f(x_i, y_i) H(x_i, y_i), \tag{11.18}$$

where A is the area of the rectangle. Note that if $f(x, y) = 1$ everywhere, then (11.18) is equivalent to the hit or miss method of calculating the area of the region R. In general, (11.18) represents the area of the region R multiplied by the average value of $f(x, y)$ in R.

Problem 11.8. Two-dimensional numerical integration

a. Write a program to implement the midpoint approximation in two dimensions and integrate the function $f(x, y) = x^2 + 6xy + y^2$ over the region defined by the condition $x^2 + y^2 \leq 1$. Use $h = 0.1$, 0.05, 0.025, and 0.0125.

b. Repeat part (a) using a Monte Carlo method and the same number of points n. For each value of n repeat the calculation several times to obtain a crude estimate of the random error.

Problem 11.9. Volume of a hypersphere

a. The interior of a d-dimensional hypersphere of unit radius is defined by the condition $x_1^2 + x_2^2 + \cdots + x_d^2 \leq 1$. Write a program that finds the volume of a hypersphere using the midpoint approximation. If you are clever, you can write a program that does any dimension using a recursive method. Test your program for $d = 2$ and $d = 3$, and then find the volume for $d = 4$ and $d = 5$. Begin with $h = 0.2$, and decrease h until your results do not change by more than 1%.

b. Repeat part (a) using a Monte Carlo method. For each value of n, repeat the calculation several times to obtain a rough estimate of the random error. Is your program applicable for any d easier to write than in part (a)?

11.4 Monte Carlo Error Analysis

Both the classical numerical integration methods and the Monte Carlo methods yield approximate answers whose accuracy depends on the number of intervals or on the number of samples, respectively. So far, we have used our knowledge of the exact value of various integrals to determine that

the error in the Monte Carlo methods approaches zero as $n^{-1/2}$ for large n, where n is the number of samples. In the following, we will learn how to estimate the error when the exact answer is unknown. Our main result is that the $n^{-1/2}$ dependence of the error is a general result, and is independent of the nature of the integrand and, most importantly, independent of the number of dimensions.

As before, we first determine the error for an explicit example. Consider the Monte Carlo evaluation of the integral of $f(x) = 4\sqrt{1-x^2}$ in the interval $[0, 1]$ (see Problem 11.7). Our result for a particular sequence of $n = 10^4$ random numbers using the sample mean method is $F_n = 3.1489$. How does this result for F_n compare with your result found in Problem 11.7 for the same value of n? By comparing F_n to the exact result of $F = \pi \approx 3.1416$, we find that the error associated with $n = 10^4$ samples is approximately 0.0073. How do we know if $n = 10^4$ samples is sufficient to achieve the desired accuracy? We cannot answer this question definitively because if the actual error were known, we could correct F_n by the required amount and obtain F. The best we can do is to calculate the *probability* that the true value F is within a certain range centered about F_n.

We know that if the integrand were a constant, then the error would be zero, that is, F_n would equal F for any n. This limiting behavior suggests that a possible measure of the error is the *sample variance* $\tilde{\sigma}^2$ defined by

$$\tilde{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}[f(x_i) - \langle f\rangle]^2, \tag{11.19}$$

where

$$\langle f\rangle = \frac{1}{n}\sum_{i=1}^{n} f(x_i). \tag{11.20}$$

The reason for the factor of $1/(n-1)$ in (11.19) rather than $1/n$ is similar to the reason for the expression $1/\sqrt{n-2}$ in the error estimates of the least squares fits (see (7.42)). To compute $\tilde{\sigma}^2$, we need to use n samples to compute the mean, $\langle f\rangle$, and, loosely speaking, we have only $n-1$ independent samples remaining to calculate $\tilde{\sigma}^2$. Because we will always be considering values of $n >> 1$, we will replace $\tilde{\sigma}^2$ by the *variance* σ^2, which is given by

$$\sigma^2 = \langle f^2\rangle - \langle f\rangle^2, \tag{11.21}$$

where

$$\langle f^2\rangle = \frac{1}{n}\sum_{i=1}^{n}[f(x_i)]^2. \tag{11.22}$$

For our example and the same sequence of random numbers that we used to obtain $F_n = 3.1489$, we obtain the standard deviation $\sigma = 0.8850$. Because this value of σ is two orders of magnitude larger than the actual error, we conclude that σ is not a direct measure of the error.

Another clue to finding an appropriate measure of the error can be found by increasing n and seeing how the actual error decreases as n increases. In Table 11.2 we see that as n is increased from 10^2 to 10^4, the actual error decreased by a factor of approximately 10, that is, as $\sim 1/n^{1/2}$. We also see that the actual error is approximately given by $\sigma/\sqrt{n}$. In Appendix 11B we show that

n	F_n	actual error	σ	$\sigma/\sqrt{n}$
10^2	3.0692	0.0724	0.8550	0.0855
10^3	3.1704	0.0288	0.8790	0.0270
10^4	3.1489	0.0073	0.8850	0.0089

Table 11.2: Examples of Monte Carlo measurements of the mean value of $f(x) = 4\sqrt{1-x^2}$ in the interval $[0, 1]$. The actual error is given by the difference $|F_n - \pi|$. The standard deviation σ is estimated using (11.21).

the *standard error of the means*, σ_m, is given by

$$\sigma_m = \frac{\sigma}{\sqrt{n-1}} \tag{11.23a}$$

$$\approx \frac{\sigma}{\sqrt{n}}. \tag{11.23b}$$

The interpretation of σ_m is that if we make many independent measurements of F_n, each with n data points, then the *probable error* associated with any single measurement is σ_m. The more precise interpretation of σ_m is that F_n, our estimate for the mean, has a 68% chance of being within σ_m of the "true" mean, and a 97% chance of being within $2\sigma_m$. This interpretation assumes a Gaussian distribution of the various measurements.

The quantity F_n is an estimate of the average value of the data points. As we increase n, the number of data points, we do not expect our estimate of the mean to change much. What changes as we increase n is our *confidence* in our estimate of the mean. Similar considerations hold for our estimate of σ, which is why σ is not a direct measure of the error.

The error estimate in (11.23) assumes that the data points are independent of each other. However, in many situations the data is correlated, and we have to be careful about how we estimate the error. For example, suppose that instead of choosing n random values for x, we instead start with a particular value x_0 and then randomly add increments such that the ith value of x is given by $x_i = x_{i-1} + (2r-1)\delta$, where r is uniformly distributed between 0 and 1 and $\delta = 0.01$. Clearly, the x_i are now correlated. We can still obtain an estimate for the integral, but we cannot use $\sigma/\sqrt{n}$ as the estimate for the error because this estimate would be smaller than the actual error. However, we expect that two data points, x_i and x_j, will become uncorrelated if $|j-i|$ is sufficiently large. How can we tell when $|j-i|$ is sufficiently large? One way is to group the data by averaging over m data points. We take $f_1^{(m)}$ to be the average of the first m values of $f(x_i)$, $f_2^{(m)}$ to be the average of the next m values, and so forth. Then we compute $\sigma_s/\sqrt{s}$, where $s = n/m$ is the number of $f_i^{(m)}$ data points, each of which is an average over m of the original data points, and σ_s is the standard deviation of the s data points, We do this grouping for different values of m (and s) and find the value of m for which $\sigma_s/\sqrt{s}$ becomes approximately independent of m. This ratio is our estimate of the error of the mean.

We see that we can make the probable error as small as we wish by either increasing n, the number of data points, or by reducing the variance σ^2. Several *reduction of variance* methods are introduced in Sections 11.6 and 11.7.

Problem 11.10. Estimate of the Monte Carlo error

a. Estimate the integral of $f(x) = e^{-x}$ in the interval $0 \leq x \leq 1$ using the sample mean Monte Carlo method with $n = 10^4$, $n = 10^6$, and $n = 10^8$. Determine the exact integral analytically and estimate the n dependence of the actual error. How does your estimated error compare with the error estimate obtained from the relation (11.23)?

b. Generate 19 additional measurements of the integral each with $n = 10^6$ samples. Compute the standard deviation of the 20 measurements. Is the magnitude of this standard deviation of the means consistent with your estimates of the error obtained in part (a)? Compute the histogram of the additional measurements and confirm that the distribution of the measurements is consistent with a Gaussian distribution.

c. Divide your first measurement of $n = 10^6$ samples into $s = 10$ subsets of 10^5 samples each. Is the value of $\sigma_s/\sqrt{s}$ consistent with your previous error estimates?

d. Estimate the integral

$$\int_0^1 e^{-x^2}\, dx \tag{11.24}$$

to two decimal places using $\sigma/\sqrt{n}$ as an estimate of the probable error.

e. Estimate the integral $\int_0^{2\pi} \cos^2\theta\, d\theta$ using $n = 10^6$, where $\theta_i = \theta_{i-1} + (2r - 1)\delta$, r is uniformly distributed between 0 and 1, and $\delta = 0.1$. Note that because $\cos\theta = \cos\theta + 2k\pi$ for any integer k, we do not have to restrict the range of θ_i. Estimate the error using (11.23). Is this error estimate accurate? Also, estimate the error by grouping the data into $m = 10$, 10^2, 10^3, 10^4, and 10^5 data points and compute $\sigma_s/\sqrt{s}$, for $s = 10^5$, 10^4, 10^3, 10^2, and 10, respectively. How large must m be so that the error estimates for different m are approximately the same? Discuss the relation between this result and the correlation of the data points.

*__Problem 11.11.__ Importance of randomness

We learned in Chapter 7 that the random number generator included with many programming languages is based on the linear congruential method. In this method each term in the sequence can be found from the preceding one by the relation

$$x_{n+1} = (ax_n + c) \bmod m, \tag{11.25}$$

where x_0 is the seed, and a, c, and m are nonnegative integers. The random numbers r in the unit interval $0 \leq r < 1$ are given by $r_n = x_n/m$. To examine the effect of a poor random number generator, we choose values of x_0, m, a, and c such that (11.25) has poor statistical properties, for example, a short period. What is the period for $x_0 = 1$, $a = 5$, $c = 0$, and $m = 32$? Estimate the integral in Problem 11.10 by making a single measurement of $n = 10^4$ samples using the linear congruential method (11.25) with these values of x_0, a, c, and m. Analyze your measurement by computing $\sigma_s/s^{1/2}$ for $s = 20$ subsets. Then divide your data into $s = 10$ subsets. Is the value of $\sigma_s/s^{1/2}$ consistent with what you obtained for $s = 20$? If not, why?

*__Problem 11.12.__ Error estimating by bootstrapping

Suppose that we have made a series of measurements, but do not know the underlying probability distribution of the data. How can we estimate the errors of the quantities of interest in an unbiased

way? One way is to use a method known as *bootstrapping*, a method that uses random sampling to estimate the errors.

Consider a set of n measurements such as n values of the pairs (x_i, y_i), and suppose we want to fit this data to the best straight line. If we label the original set of measurements, $M = \{m_1, m_2, \ldots m_n\}$, then the kth *resampled* data set M_k consists of n measurements that are randomly chosen from the original set. This procedure means that some of the m_i may not appear in M_k and some may appear more than once. We then compute the quantity G_k from the resampled data set. For example, G_k could be the slope found from a least squares calculation. If we do this resampling n_r times, a measure of the error in the quantity G is given by σ_G^2, where

$$\sigma_G = \frac{1}{n_r - 1} \sum_{k=1}^{n_r} \left[G_k - \langle G_k \rangle \right]^2. \tag{11.26}$$

with

$$\langle G_k \rangle = \frac{1}{n_r} \sum_{k=1}^{n_r} G_k. \tag{11.27}$$

a. To see how this procedure works, consider $n = 15$ pairs of points x_i randomly distributed between 0 and 1, with the corresponding values of y given by $y_i = 2x_i + 3 + s_i$, where s_i is a uniform random number between -1 and $+1$. First compute the slope, m, and the intercept, b, using the least squares method and their corresponding errors using (7.41).

b. Resample the same set of data 200 times, computing the slope and intercept each time using the least squares method. From your results estimate the probable error for the slope and intercept using (11.26). How well do the estimates from bootstrapping compare with the direct error estimates found in part a? Does the average of the bootstrap values for the slope and intercept equal m and b, respectively, from the least squares fits. If not why not? Do your conclusions change if you resample 800 times?

11.5 Nonuniform Probability Distributions

In Sections 11.2 and 11.4 we learned how uniformly distributed random numbers can be used to estimate definite integrals. We will find that it is more efficient to sample the integrand $f(x)$ more often in regions of x where the magnitude of $f(x)$ is large or rapidly varying. Because *importance sampling* methods require nonuniform probability distributions, we first consider several methods for generating random numbers that are not distributed uniformly. In the following, we will denote r as a member of a uniform random number sequence in the unit interval $0 \leq r < 1$.

Suppose that two discrete events 1 and 2 occur with probabilities p_1 and p_2 such that $p_1 + p_2 = 1$. How can we choose the two events with the correct probabilities using a uniform probability distribution? For this simple case, it is clear that we choose event 1 if $r < p_1$; otherwise, we choose event 2. If there are three events with probabilities p_1, p_2, and p_3, then if $r < p_1$ we choose event 1; else if $r < p_1 + p_2$, we choose event 2; else we choose event 3. We can visualize these choices by dividing a line segment of unit length into three pieces whose lengths are shown in Figure 11.4.

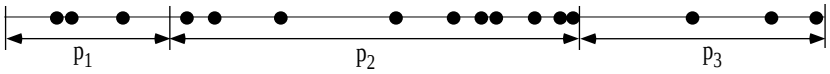

Figure 11.4: The unit interval is divided into three segments of lengths $p_1 = 0.2$, $p_2 = 0.5$, and $p_3 = 0.3$. Sixteen random numbers are represented by the filled circles uniformly distributed on the unit interval. The fraction of circles within each segment is approximately equal to the value of p_i for that segment.

Now consider n discrete events. How do we determine which event, i, to choose given the value of r? The generalization of the procedure we have followed for $n = 2$ and 3 is to find the value of i that satisfies the condition

$$\sum_{j=0}^{i-1} p_j \le r \le \sum_{j=0}^{i} p_j, \tag{11.28}$$

where we have defined $p_0 \equiv 0$. Verify that (11.28) reduces to the correct procedure for $n = 2$ and $n = 3$.

We now consider a continuous nonuniform probability distribution. One way to generate such a distribution is to take the limit of (11.28) and associate p_i with $p(x)\,\Delta x$, where the *probability density* $p(x)$ is defined such that $p(x)\,\Delta x$ is the probability that the event x is in the interval between x and $x + \Delta x$. The probability density $p(x)$ is normalized such that

$$\int_{-\infty}^{+\infty} p(x)\,dx = 1. \tag{11.29}$$

In the continuum limit the two sums in (11.28) become the same integral and the inequalities become equalities. Hence we can write

$$P(x) \equiv \int_{-\infty}^{x} p(x')\,dx' = r. \tag{11.30}$$

From (11.30) we see that the uniform random number r corresponds to the *cumulative probability distribution* function $P(x)$, which is the probability of choosing a value less than or equal to x. The function $P(x)$ should not be confused with the probability density $p(x)$ or the probability $p(x)\,\Delta x$. In many applications the meaningful range of values of x is positive. In that case, we have $p(x) = 0$ for $x < 0$.

The relation (11.30) leads to the *inverse transform* method for generating random numbers distributed according to the function $p(x)$. This method involves generating a random number r and solving (11.30) for the corresponding value of x. As an example of the method, we use (11.30) to generate a random number sequence according to the uniform probability distribution on the interval $a \le x \le b$. The desired probability density $p(x)$ is

$$p(x) = \begin{cases} (1/(b-a) & a \le x \le b \\ 0. & \text{otherwise} \end{cases} \tag{11.31}$$

The cumulative probability distribution function $P(x)$ for $a \le x \le b$ can be found by substituting (11.31) into (11.30) and performing the integral. The result is

$$P(x) = \frac{x-a}{b-a}. \tag{11.32}$$

If we substitute the form (11.32) for $P(x)$ into (11.30) and solve for x, we find the desired relation

$$x = a + (b-a)r. \tag{11.33}$$

The variable x given by (11.33) is distributed according to the probability distribution $p(x)$ given by (11.31). Of course, the relation (11.33) is obvious, and we already have used it in our programs.

We next apply the inverse transform method to the probability density function

$$p(x) = \begin{cases} (1/\lambda)\, e^{-x/\lambda} & 0 \le x \le \infty \\ 0. & x < 0. \end{cases} \tag{11.34}$$

If we substitute (11.34) into (11.30) and integrate, we find the relation

$$r = P(x) = 1 - e^{-x/\lambda}. \tag{11.35}$$

The solution of (11.35) for x yields $x = -\lambda \ln(1-r)$. Because $1-r$ is distributed in the same way as r, we can write

$$x = -\lambda \ln r. \tag{11.36}$$

The variable x found from (11.36) is distributed according to the probability density $p(x)$ given by (11.34). Because the computation of the natural logarithm in (11.36) is relatively slow, the inverse transform method might not be the most efficient method to use in this case.

From the above examples, we see that two conditions must be satisfied in order to apply the inverse transform method: the form of $p(x)$ must allow us to perform the integral in (11.30) analytically, and it must be practical to invert the relation $P(x) = r$ for x.

The Gaussian probability density,

$$p(x) = \frac{1}{(2\pi\sigma^2)^{1/2}} e^{-x^2/2\sigma^2}, \tag{11.37}$$

is an example of a probability density for which the cumulative distribution $P(x)$ cannot be obtained analytically. However, we can generate the two-dimensional probability $p(x,y)\,dx\,dy$ given by

$$p(x,y)\,dx\,dy = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}\,dx\,dy. \tag{11.38}$$

First, we make a change of variables to polar coordinates:

$$r = (x^2+y^2)^{1/2}, \qquad \theta = \tan^{-1}\frac{y}{x}. \tag{11.39}$$

We let $\rho = r^2/2$ and write the two-dimensional probability as

$$p(\rho,\theta)\,d\rho\,d\theta = \frac{1}{2\pi} e^{-\rho} d\rho\,d\theta, \tag{11.40}$$

where we have set $\sigma = 1$. If we generate ρ according to the exponential distribution (11.34) and generate θ uniformly in the interval $0 \le \theta < 2\pi$, then the quantities

$$x = (2\rho)^{1/2}\cos\theta \quad \text{and} \quad y = (2\rho)^{1/2}\sin\theta \qquad \text{(Box-Muller method)} \tag{11.41}$$

will each be generated according to (11.37) with zero mean and $\sigma = 1$. (Note that the two-dimensional density (11.38) is the product of two independent one-dimensional Gaussian distributions.) This way of generating a Gaussian distribution is known as the *Box-Muller* method. We discuss other methods for generating the Gaussian distribution in Problems 11.14 and 11.17 and in Appendix 11C.

Problem 11.13. Nonuniform probability densities

a. Write a program to simulate the simultaneous rolling of two dice. In this case the events are discrete and occur with nonuniform probability.

b. Write a program to verify that the sequence of random numbers $\{x_i\}$ generated by (11.36) is distributed according to the exponential distribution (11.34).

c. Generate random variables according to the probability density function

$$p(x) = \begin{cases} 2(1-x) & 0 \le x \le 1 \\ 0. & \text{otherwise} \end{cases} \tag{11.42}$$

d. Verify that the variables x and y in (11.41) are distributed according to the Gaussian distribution. What is the mean value and the standard deviation of x and of y?

e. How can you use the relations (11.41) to generate a Gaussian distribution with arbitrary mean and standard deviation?

Problem 11.14. Generating normal distributions

Fernández and Criado have suggested another method of generating normal distributions that is much faster than the Box-Muller method. We will just summarize the algorithm; the proof that the algorithm leads to a normal distribution is given in their paper.

i. Begin with N numbers, v_i, in an array. Set all the $v_i = \sigma$, where σ is the desired standard deviation for the normal distribution.

ii. Update the array by randomly choosing two different entries, v_i and v_j from the array. Then let $v_i = (v_i + v_j)/\sqrt{2}$ and use the new v_i to set $v_j = -v_i + v_j\sqrt{2}$.

iii. Repeat step (ii) many times to bring the array of numbers to "equilibrium."

iv. After equilibration, the entries v_i will have a normal distribution with the desired standard deviation and zero mean.

Write a program to produce a series of random numbers according to this algorithm. Your program should allow the user to enter N and σ and a button should be implemented to allow for equilibration before various averages are computed. The desired output is the probability distribution of the random numbers that are produced as well as their mean and standard deviation. First make sure that the standard deviation of the probability distribution approaches the desired input σ for sufficiently long times. What is the order of magnitude of the equilibration time? Does it depend on N? Plot the natural log of the probability distribution versus v^2 and check that you obtain a straight line with the appropriate slope.

11.6 Importance Sampling

In Section 11.4 we found that the error associated with a Monte Carlo estimate is proportional to the standard deviation σ of the integrand and inversely proportional to the square root of the number of samples. Hence, there are only two ways of reducing the error in a Monte Carlo estimate – either increase the number of samples or reduce the variance. Clearly the latter choice is desirable because it does not require much more computer time. In this section we introduce *importance sampling* techniques that reduce σ and improve the efficiency of each sample.

To do importance sampling in the context of numerical integration, we introduce a positive function $p(x)$ such that

$$\int_a^b p(x)\,dx = 1, \tag{11.43}$$

and rewrite the integral (11.1) as

$$F = \int_a^b \Big[\frac{f(x)}{p(x)}\Big] p(x)\,dx. \tag{11.44}$$

We can evaluate the integral (11.44) by sampling according to the probability distribution $p(x)$ and constructing the sum

$$F_n = \frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)}{p(x_i)}. \tag{11.45}$$

The sum (11.45) reduces to (11.15) for the uniform case $p(x) = 1/(b-a)$.

The idea is to choose a form for $p(x)$ that minimizes the variance of the ratio $f(x)/p(x)$. To do so we choose a form of $p(x)$ that mimics $f(x)$ as much as possible, particularly where $f(x)$ is large. A suitable choice of $p(x)$ would make the integrand $f(x)/p(x)$ slowly varying, and hence reduce the variance. Because we cannot evaluate the variance analytically in general, we determine σ *a posteriori*.

As an example, we again consider the integral (see Problem 11.10d)

$$F = \int_0^1 e^{-x^2}\,dx. \tag{11.46}$$

The estimate of F with $p(x) = 1$ for $0 \le x \le 1$ is shown in the second column of Table 11.3. A simple choice for the weight function is $p(x) = Ae^{-x}$, where A is chosen such that $p(x)$ is normalized

	$p(x) = 1$	$p(x) = Ae^{-x}$
n (samples)	5×10^6	4×10^5
F_n	0.74684	0.74689
σ	0.2010	0.0550
$\sigma/\sqrt{n}$	0.00009	0.00009
Total CPU time (s)	20	2.5
CPU time per sample (s)	4×10^{-6}	6×10^{-6}

Table 11.3: Comparison of the Monte Carlo estimates of the integral (11.46) using the uniform probability density $p(x) = 1$ and the nonuniform probability density $p(x) = Ae^{-x}$. The normalization constant A is chosen such that $p(x)$ is normalized on the unit interval. The value of the integral to five decimal places is 0.74682. The estimate F_n, variance σ of f/p, and the probable error $\sigma/n^{1/2}$ are shown. The CPU time (in seconds) is shown for comparison only. (The number of samples was chosen so that the error estimates are comparable.)

on the unit interval. Note that this choice of $p(x)$ is positive definite and is qualitatively similar to $f(x)$. The results are shown in the third column of Table 11.3. We see that although the computation time per sample for the nonuniform case is larger, the smaller value of σ makes the use of the nonuniform probability distribution more efficient.

Problem 11.15. Importance sampling

a. Choose $f(x) = \sqrt{1 - x^2}$ and consider $p(x) = A(1 - x)$ for $x \geq 0$. What is the value of A that normalizes $p(x)$ in the unit interval $[0, 1]$? What is the relation for the random variable x in terms of r for this form of $p(x)$? What is the variance of $f(x)/p(x)$ in the unit interval? Evaluate the integral $\int_0^1 f(x)dx$ using $n = 10^6$ and estimate the probable error of your result.

b. Choose $p(x) = Ae^{-\lambda x}$ and evaluate the integral

$$\int_0^\pi \frac{1}{x^2 + \cos^2 x}\, dx. \tag{11.47}$$

Determine the value of λ that minimizes the variance of the integrand.

Problem 11.16. An adaptive approach to importance sampling

An alternative approach is to use the known values of $f(x)$ at regular intervals to sample more often where $f(x)$ is relatively large. Because the idea is to use $f(x)$ itself to determine the probability of sampling, we only consider integrands that are non-negative. To compute a rough estimate of the relative values of $f(x)$, we first compute its average value by taking k equally spaced points s_i and computing the sum

$$S = \sum_{i=1}^{k} f(s_i). \tag{11.48}$$

This sum divided by k gives an estimate of the average value of f in the interval. The approximate value of the integral is given by $F \approx Sh$, where $h = (b - a)/k$. This approximation of the integral is equivalent to the rectangular or mid-point approximation depending on where we compute the

values of $f(x)$. We then generate n random samples as follows. The probability of choosing subinterval (bin) i is given by the probability

$$p_i = \frac{f(s_i)}{S}. \tag{11.49}$$

Note that the sum of p_i over all subintervals is normalized to unity.

To choose a subinterval with the desired probability, we generate a random number uniformly in the interval $[a, b]$ and determine the subinterval i that satisfies the inequality (11.28). Now that the subinterval has been chosen with the desired probability, we generate a random number x_i in the subinterval $[s_i, s_i + h]$ and compute the ratio $f(x_i)/p(x_i)$. The estimate of the integral is given by the following considerations. The probability p_i in (11.49) is the probability of choosing the subinterval i, not the probability $p(x)\Delta x$ of choosing a value of x between x and $x + \Delta x$. The latter is p_i times the probability of picking the particular value of x in subinterval i:

$$p(x_i)\Delta x = p_i \frac{\Delta x}{h}. \tag{11.50}$$

Hence, we have that

$$F_n = \frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)}{p(x_i)} = \frac{h}{n}\sum_{i=1}^{n} \frac{f(x_i)}{p_i}. \tag{11.51}$$

Apply this method to estimate the integral of $f(x) = \sqrt{1 - x^2}$ in the unit interval. Under what circumstances would this approach be most useful?

11.7 Metropolis Algorithm

Another way of generating an arbitrary nonuniform probability distribution was introduced by Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller in 1953. The *Metropolis* algorithm is a special case of an importance sampling procedure in which certain possible sampling attempts are rejected (see Appendix 11C). The Metropolis method is useful for computing averages of the form

$$\langle f \rangle = \frac{\int f(x)p(x)\,dx}{\int p(x)\,dx}, \tag{11.52}$$

where $p(x)$ is an arbitrary function that need not be normalized. In Chapter 16 we will discuss the application of the Metropolis algorithm to problems in statistical mechanics.

For simplicity, we introduce the Metropolis algorithm in the context of estimating one-dimensional definite integrals. Suppose that we wish to use importance sampling to generate random variables according to $p(x)$. The Metropolis algorithm produces a random walk of points $\{x_i\}$ whose asymptotic probability distribution approaches $p(x)$ after a large number of steps. The random walk is defined by specifying a *transition probability* $T(x_i \to x_j)$ from one value x_i to another value x_j such that the distribution of points x_0, x_1, x_2, ... converges to $p(x)$. It can be shown that it is sufficient (but not necessary) to satisfy the *detailed balance* condition

$$p(x_i)T(x_i \to x_j) = p(x_j)T(x_j \to x_i). \tag{11.53}$$

The relation (11.53) does not specify $T(x_i \to x_j)$ uniquely. A simple choice of $T(x_i \to x_j)$ that is consistent with (11.53) is

$$T(x_i \to x_j) = \min\left[1, \frac{p(x_j)}{p(x_i)}\right]. \tag{11.54}$$

If the "walker" is at position x_i and we wish to generate x_{i+1}, we can implement this choice of $T(x_i \to x_{i+1})$ by the following steps:

1. Choose a trial position $x_{\rm trial} = x_i + \delta_i$, where δ_i is a uniform random number in the interval $[-\delta, \delta]$.
2. Calculate $w = p(x_{\rm trial})/p(x_i)$.
3. If $w \geq 1$, accept the change and let $x_{i+1} = x_{\rm trial}$.
4. If $w < 1$, generate a random number r.
5. If $r \leq w$, accept the change and let $x_{i+1} = x_{\rm trial}$.
6. If the trial change is not accepted, then let $x_{i+1} = x_i$.

It is necessary to sample many points of the random walk before the asymptotic probability distribution $p(x)$ is attained. How do we choose the maximum step size δ? If δ is too large, only a small percentage of trial steps will be accepted and the sampling of $p(x)$ will be inefficient. On the other hand, if δ is too small, a large percentage of trial steps will be accepted, but again the sampling of $p(x)$ will be inefficient. A rough criterion for the magnitude of δ is that approximately one third to one half of the trial steps should be accepted. We also wish to choose the value of x_0 such that the distribution $\{x_i\}$ will approach the asymptotic distribution as quickly as possible. An obvious choice is to begin the random walk at a value of x at which $p(x)$ is a maximum. A code fragment that implements the Metropolis algorithm is given below.

```
double xtrial = x + (2*rnd.nextDouble() - 1.0)*delta;
double w = p(xtrial)/p(x);
if (w > 1 || w > rnd.nextDouble()) {
   x = xtrial;
   naccept++;        // number of acceptances
}
```

Problem 11.17. Generating the Gaussian distribution

a. Use the Metropolis algorithm to generate the Gaussian distribution, $p(x) = Ae^{-x^2/2}$. Is the value of the normalization constant A relevant? Determine the qualitative dependence of the acceptance ratio and the equilibration time on the maximum step size δ. One possible criterion for equilibrium is that $\langle x^2 \rangle \approx 1$. What is a reasonable choice for δ? How many trials are needed to reach equilibrium for your choice of δ?

b. Modify your program so that it plots the asymptotic probability distribution generated by the Metropolis algorithm.

c.* Calculate the autocorrelation function $C(j)$ defined by (see Problem 7.31)

$$C(j) = \frac{\langle x_{i+j} x_i \rangle - \langle x_i \rangle^2}{\langle x_i^2 \rangle - \langle x_i \rangle^2}, \tag{11.55}$$

where $\langle \ldots \rangle$ indicates an average over the random walk. What is the value of $C(j = 0)$? What would be the value of $C(j \neq 0)$ if x_i were completely random? Calculate $C(j)$ for different values of j and determine the value of j for which $C(j)$ is close to zero.

Problem 11.18. Application of the Metropolis algorithm

a. Although the Metropolis algorithm is not the most efficient method in this case, write a program to estimate the average

$$\langle x \rangle = \frac{\int_0^\infty x e^{-x}\, dx}{\int_0^\infty e^{-x}\, dx}, \tag{11.56}$$

with $p(x) = Ae^{-x}$ for $x \geq 0$ and $p(x) = 0$ for $x < 0$. Compute the histogram $H(x)$ showing the number of points in the random walk in the region x to $x + \Delta x$, with $\Delta x = 0.2$. Begin with $n \geq 1000$ and maximum step size $\delta = 1$. Allow the system to equilibrate for at least 200 steps before computing averages. Is the integrand sampled uniformly? If not, what is the approximate region of x where the integrand is sampled more often?

b. Calculate analytically the exact value of $\langle x \rangle$ in (11.56). How do your Monte Carlo results compare with the exact value for $n = 100$ and $n = 1000$ with $\delta = 0.1$, 1, and 10? Estimate the standard error of the mean. Does this error give a reasonable estimate of the error? If not, why?

c. In part (b) you should have found that the estimated error is much smaller than the actual error. The reason is that the $\{x_i\}$ are not statistically independent. The Metropolis algorithm produces a random walk whose points are correlated with each other over short times (measured by the number of steps of the random walker). The correlation of the points decays exponentially with time. If τ is the characteristic time for this decay, then only points separated by approximately 2 to 3τ can be considered statistically independent. Compute $C(j)$ as defined in (11.55) and make a rough estimate of τ. Rerun your program with the data grouped into 20 sets of 50 points each and 10 sets of 100 points each. If the sets of 50 points each are statistically independent (that is, if τ is significantly smaller than 50), then your estimate of the error for the two groupings should be approximately the same. The importance of correlations between sampled points is discussed further in Section 16.7.

11.8 *Neutron Transport

We consider the application of a nonuniform probability distribution to the simulation of the transmission of neutrons through bulk matter, one of the original applications of a Monte Carlo method. Suppose that a neutron is incident on a plate of thickness t. We assume that the plate is infinite in the x and y directions and the z axis is normal to the plate. At any point within

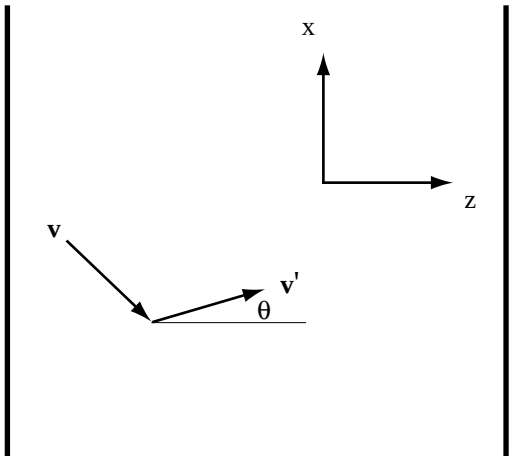

Figure 11.5: The definition of the scattering angle θ. The velocity before scattering is $\mathbf{v}$ and the velocity after scattering is $\mathbf{v}'$. The scattering angle θ is independent of $\mathbf{v}$ and is defined relative to the z axis.

the plate, the neutron can either be captured with probability p_c or scattered with probability p_s. These probabilities are proportional to the capture cross section and scattering cross section, respectively. If the neutron is scattered, we need to find its new direction as specified by the polar angle θ (see Figure 11.5). Because we are not interested in how far the neutron moves in the x or y direction, the value of the azimuthal angle ϕ is irrelevant.

If the neutrons are scattered equally in all directions, then the probability $p(\theta, \phi)\, d\theta d\phi$ equals $d\Omega/4\pi$, where $d\Omega$ is an infinitesimal solid angle and 4π is the total solid angle. Because $d\Omega = \sin\theta\, d\theta d\phi$, we have

$$p(\theta, \phi) = \frac{\sin\theta}{4\pi}. \tag{11.57}$$

We can find the probability density for θ and ϕ separately by integrating over the other angle. For example,

$$p(\theta) = \int_0^{2\pi} p(\theta, \phi)\, d\phi = \frac{1}{2}\sin\theta, \tag{11.58}$$

and

$$p(\phi) = \int_0^{\pi} p(\theta, \phi)\, d\theta = \frac{1}{2\pi}. \tag{11.59}$$

Because the point probability $p(\theta, \phi)$ is the product of the probabilities $p(\theta)$ and $p(\phi)$, θ and ϕ are independent variables. Although we do not need to generate a random angle ϕ, we note that because $p(\phi)$ is a constant, ϕ can be found from the relation

$$\phi = 2\pi r. \tag{11.60}$$

To find θ according to the distribution (11.58), we substitute (11.58) in (11.30) and obtain

$$r = \frac{1}{2}\int_0^\theta \sin x\, dx \tag{11.61}$$

The integration in (11.61) gives

$$\cos\theta = 1 - 2r. \tag{11.62}$$

Note that (11.60) implies that ϕ is uniformly distributed between 0 and 2π and (11.62) implies that $\cos\theta$ is uniformly distributed between -1 and $+1$. We could invert the cosine in (11.62) to solve for θ. However, to find the z component of the path of the neutron through the plate, we need to multiply the path length ℓ by $\cos\theta$, and hence we need $\cos\theta$ rather than θ.

The path length, which is the distance traveled between subsequent scattering events, is obtained from the exponential probability density, $p(\ell) \propto e^{-\ell/\lambda}$ (see (11.34)). From (11.36), we have

$$\ell = -\lambda \ln r, \tag{11.63}$$

where λ is the mean free path.

Now we have all the necessary ingredients for calculating the probabilities for a neutron to pass through the plate, to be reflected off the plate, or be captured and absorbed in the plate. The input parameters are the thickness of the plate t, the capture probability p_c and the mean free path λ. The scattering probability is $p_s = 1 - p_c$. We begin with $z = 0$, and implement the following steps:

1. Determine if the neutron is captured or scattered. If it is captured, then add one to the number of captured neutrons, and go to step 5.
2. If the neutron is scattered, compute $\cos\theta$ from (11.62) and ℓ from (11.63). Change the z coordinate of the neutron by $\ell\cos\theta$.
3. If $z < 0$, add one to the number of reflected neutrons. If $z > t$, add one to the number of transmitted neutrons. In either case, skip to step 5 below.
4. Repeat steps 1–3 until the fate of the neutron has been determined.
5. Repeat steps 1–4 with additional incident neutrons until sufficient data has been obtained.

Problem 11.19. Elastic neutron scattering

a. Write a program to implement the above algorithm for neutron scattering through a plate. Assume $t = 1$ and $p_c = p_s/2$. Find the transmission, reflection, and absorption probabilities for $\lambda = 0.01$, 0.05, 0.1, 1, and 10. Begin with 1000 incident neutrons, and increase this number until satisfactory statistics are obtained. Give a qualitative explanation of your results.

b. Choose $t = 1$, $p_c = p_s$, and $\lambda = 0.05$, and compare your results with the analogous case considered in part (a).

c. Repeat part (b) with $t = 2$ and $\lambda = 0.1$. Do the various probabilities depend on λ and t separately or only on their ratio? Answer this question before doing the simulation.

d. Draw some typical paths of the neutrons. From the nature of these paths, explain the results in parts (a)–(c). For example, how does the number of scattering events change as the absorption probability changes?

Problem 11.20. Inelastic neutron scattering

a. In Problem 11.19 we assumed elastic scattering, that is, no energy is lost during scattering. Here we assume that some of the neutron energy E is lost and that the mean free path is proportional to the speed and hence to $\sqrt{E}$. Modify your program so that a neutron loses a fraction f of its energy at each scattering event, and assume that $\lambda = \sqrt{E}$. Consider $f = 0.05, 0.1$, and 0.5, and compare your results with those found in Problem 11.19a, using the values for λ in Problem 11.19a to determine the initial values for E.

b. Make a histogram for the path lengths between scattering events and plot the path length distribution function for $f = 0.1$, 0.5, and 0 (elastic scattering).

This procedure for simulating neutron scattering and absorption is more computer intensive than necessary. Instead of considering a single neutron at a time, we can consider a collection of M neutrons. Then instead of determining whether one neutron is captured or scattered, we determine the number that is captured and the number that is scattered. For example, at the first scattering site, $p_c M$ of the neutrons are captured and $p_s M$ are scattered. We also assume that all the scattered neutrons move in the same direction with the same path length, both of which are generated at random as before. At the next scattering site there are $p_s^2 M$ scattered neutrons and $p_s p_c M$ captured neutrons. At the end of m steps, the number of neutrons remaining is $w = p_s^m M$ and the number of captured neutrons is $(p_c + p_c p_s + p_c p_s^2 + \ldots + p_c p_s^{m-1})M$. If the new position at the mth step is at $z < 0$, we add w to the sum for the reflected neutrons; if $z > t$, we add w to the neutrons transmitted. When the neutrons are reflected or transmitted, we start over again at $z = 0$ with another collection of neutrons.

Problem 11.21. More efficient neutron scattering method

Apply the improved Monte Carlo method to neutron transmission through a plate. Repeat the simulations suggested in Problem 11.19 and compare your new and previous results. Also compare the computational times for the two approaches to obtain comparable statistics.

The power of the Monte Carlo method becomes apparent for more complicated geometries or when the material is spatially nonuniform so that the cross sections vary from point to point. A problem of current interest is the absorption of various forms of radiation in the human body.

Problem 11.22. Transmission through layered materials

Consider two plates with the same thickness $t = 1$ that are stacked on top of one another with no space between them. For one plate, $p_c = p_s$, and for the other, $p_c = 2p_s$, that is, the top plate is a better absorber. Assume that $\lambda = 1$ in both plates. Find the transmission, reflection, and absorption probabilities for elastic scattering. Does it matter which plate receives the incident neutrons?

Appendix 11A: Error Estimates for Numerical Integration

We derive the dependence of the truncation error on the number of intervals for the numerical integration methods considered in Sections 11.1 and 11.3. These estimates are based on the assumed adequacy of the Taylor series expansion of the integrand $f(x)$:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2}f''(x_i)(x - x_i)^2 + \ldots, \tag{11.64}$$

and the integration of (11.1) in the interval $x_i \le x \le x_{i+1}$:

$$\int_{x_i}^{x_{i+1}} f(x)\,dx = f(x_i)\Delta x + \frac{1}{2}f'(x_i)(\Delta x)^2 + \frac{1}{6}f''(x_i)(\Delta x)^3 + \ldots \tag{11.65}$$

We first estimate the error associated with the rectangular approximation with $f(x)$ evaluated at the left side of each interval. The error Δ_i in the interval $[x_i, x_{i+1}]$ is the difference between (11.65) and the estimate $f(x_i)\Delta x$:

$$\Delta_i = \Big[\int_{x_i}^{x_{i+1}} f(x)\,dx\Big] - f(x_i)\Delta x \approx \frac{1}{2}f'(x_i)(\Delta x)^2. \tag{11.66}$$

We see that to leading order in Δx, the error in each interval is order $(\Delta x)^2$. Because there are a total of n intervals and $\Delta x = (b-a)/n$, the total error associated with the rectangular approximation is $n\Delta_i \sim n(\Delta x)^2 \sim n^{-1}$.

The estimated error associated with the trapezoidal approximation can be found in the same way. The error in the interval $[x_i, x_{i+1}]$ is the difference between the exact integral and the estimate, $\frac{1}{2}[f(x_i) + f(x_{i+1})]\Delta x$:

$$\Delta_i = \Big[\int_{x_i}^{x_{i+1}} f(x)\,dx\Big] - \frac{1}{2}[f(x_i) + f(x_{i+1})]\Delta x. \tag{11.67}$$

If we use (11.65) to estimate the integral and (11.64) to estimate $f(x_{i+1})$ in (11.67), we find that the term proportional to f' cancels and that the error associated with one interval is order $(\Delta x)^3$. Hence, the total error in the interval $[a, b]$ associated with the trapezoidal approximation is order n^{-2}.

Because Simpson's rule is based on fitting $f(x)$ in the interval $[x_{i-1}, x_{i+1}]$ to a parabola, error terms proportional to f'' cancel. We might expect that error terms of order $f'''(x_i)(\Delta x)^4$ contribute, but these terms cancel by virtue of their symmetry. Hence the $(\Delta x)^4$ term of the Taylor expansion of $f(x)$ is adequately represented by Simpson's rule. If we retain the $(\Delta x)^4$ term in the Taylor series of $f(x)$, we find that the error in the interval $[x_i, x_{i+1}]$ is of order $f''''(x_i)(\Delta x)^5$ and that the total error in the interval $[a, b]$ associated with Simpson's rule is $O(n^{-4})$.

The error estimates can be extended to two dimensions in a similar manner. The two-dimensional integral of $f(x, y)$ is the volume under the surface determined by $f(x, y)$. In the "rectangular" approximation, the integral is written as a sum of the volumes of parallelograms with cross sectional area $\Delta x \Delta y$ and a height determined by $f(x, y)$ at one corner. To determine the error, we expand $f(x, y)$ in a Taylor series

$$f(x, y) = f(x_i, y_i) + \frac{\partial f(x_i, y_i)}{\partial x}(x - x_i) + \frac{\partial f(x_i, y_i)}{\partial y}(y - y_i) + \ldots, \tag{11.68}$$

and write the error as

$$\Delta_i = \Big[\int\int f(x,y)\,dxdy\Big] - f(x_i,y_i)\Delta x\Delta y. \tag{11.69}$$

If we substitute (11.68) into (11.69) and integrate each term, we find that the term proportional to f cancels and the integral of $(x-x_i)\,dx$ yields $\frac{1}{2}(\Delta x)^2$. The integral of this term with respect to dy gives another factor of Δy. The integral of the term proportional to $(y-y_i)$ yields a similar contribution. Because Δy also is order Δx, the error associated with the intervals $[x_i, x_{i+1}]$ and $[y_i, y_{i+1}]$ is to leading order in Δx:

$$\Delta_i \approx \frac{1}{2}[f'_x(x_i,y_i) + f'_y(x_i,y_i)](\Delta x)^3. \tag{11.70}$$

We see that the error associated with one parallelogram is order $(\Delta x)^3$. Because there are n parallelograms, the total error is order $n(\Delta x)^3$. However in two dimensions, $n = A/(\Delta x)^2$, and hence the total error is order $n^{-1/2}$. In contrast, the total error in one dimension is order n^{-1}, as we saw earlier.

The corresponding error estimates for the two-dimensional generalizations of the trapezoidal approximation and Simpson's rule are order n^{-1} and n^{-2} respectively. In general, if the error goes as order n^{-a} in one dimension, then the error in d dimensions goes as $n^{-a/d}$. In contrast, Monte Carlo errors vary as $n^{-1/2}$ independent of d. Hence for large enough d, Monte Carlo integration methods will lead to smaller errors for the same choice of n.

Appendix 11B: The Standard Deviation of the Mean

In Section 11.4 we found empirically that the probable error associated with a single measurement consisting of n samples is $\sigma/\sqrt{n}$, where σ is the standard deviation associated with n data points. We derive this relation in the following. The quantity of experimental interest is denoted as x. Consider m sets of measurements each with n samples for a total of mn samples. For simplicity, we will assume that $n >> 1$. We use the index α to denote a particular measurement and the index i to designate the ith sample within a measurement. We denote $x_{\alpha,i}$ as sample i in the measurement α. The value of a measurement is given by

$$M_\alpha = \frac{1}{n}\sum_{i=1}^{n} x_{\alpha,i}. \tag{11.71}$$

The mean $\overline{M}$ of the *total* mn individual samples is given by

$$\overline{M} = \frac{1}{m}\sum_{\alpha=1}^{m} M_\alpha = \frac{1}{nm}\sum_{\alpha=1}^{m}\sum_{i=1}^{n} x_{\alpha,i}. \tag{11.72}$$

The difference between measurement α and the mean of all the measurements is given by

$$e_\alpha = M_\alpha - \overline{M}. \tag{11.73}$$

We can write the variance of the means as

$$\sigma_m^2 = \frac{1}{m}\sum_{\alpha=1}^{m} {e_\alpha}^2. \tag{11.74}$$

We now wish to relate σ_m to the variance of the individual measurements. The discrepancy $d_{\alpha,i}$ between an individual sample $x_{\alpha,i}$ and the mean is given by

$$d_{\alpha,i} = x_{\alpha,i} - \overline{M}. \tag{11.75}$$

Hence, the variance σ^2 of the nm individual samples is

$$\sigma^2 = \frac{1}{mn} \sum_{\alpha=1}^{m} \sum_{i=1}^{n} {d_{\alpha,i}}^2. \tag{11.76}$$

We write

$$e_\alpha = M_\alpha - \overline{M} = \frac{1}{n} \sum_{i=1}^{n} \left(x_{\alpha,i} - \overline{M} \right) \tag{11.77a}$$

$$= \frac{1}{n} \sum_{i=1}^{n} d_{\alpha,i}. \tag{11.77b}$$

If we substitute (11.77b) into (11.74), we find

$$\sigma_m^2 = \frac{1}{m} \sum_{\alpha=1}^{n} \left(\frac{1}{n} \sum_{i=1}^{n} d_{\alpha,i} \right) \left(\frac{1}{n} \sum_{j=1}^{n} d_{\alpha,j} \right). \tag{11.78}$$

The sum in (11.78) over samples i and j in set α contains two kinds of terms – those with $i = j$ and those with $i \neq j$. We expect that $d_{\alpha,i}$ and $d_{\alpha,j}$ are independent and equally positive or negative on the average. Hence in the limit of a large number of measurements, we expect that only the terms with $i = j$ in (11.78) survive, and we write

$$\sigma_m^2 = \frac{1}{mn^2} \sum_{\alpha=1}^{n} \sum_{i=1}^{n} {d_{\alpha,i}}^2. \tag{11.79}$$

If we combine (11.79) with (11.76), we arrive at the result

$$\sigma_m^2 = \frac{\sigma^2}{n}. \tag{11.80}$$

We intepret σ_m as the error in the original n measurments because σ_m provides an estimate of how much an average over n measurments will deviate from the exact mean.

Appendix 11C: The Acceptance-Rejection Method

Although the inverse transform method discussed in Section 11.5 can be used in principle to generate any desired probability distribution, in practice the method is limited to functions for which the equation, $r = P(x)$, can be solved analytically for x or by simple numerical approximation. Another method for generating nonuniform probability distributions is the *acceptance-rejection* method due to von Neumann. Suppose that $p(x)$ is the (normalized) probability density function

that we wish to generate. For simplicity, we assume $p(x)$ is nonzero in the unit interval. Consider a positive definite *comparison function* $w(x)$ such that $w(x) > p(x)$ in the entire range of interest. A simple although not generally optimum choice of w is a constant greater than the maximum value of $p(x)$. Because the area under the curve $p(x)$ in the range x to $x + \Delta x$ is the probability of generating x in that range, we can follow a procedure similar to that used in the hit or miss method. Generate two numbers at random to define the location of a point in two dimensions which is distributed uniformly in the area under the comparison function $w(x)$. If this point is outside the area under $p(x)$, the point is rejected; if it lies inside the area, we accept it. This procedure implies that the accepted points are uniform in the area under the curve $p(x)$ and that their x values are distributed according to $p(x)$. One procedure for generating a uniform random point (x, y) under the comparison function $w(x)$ is as follows.

1. Choose a form of $w(x)$. One convenient choice would be to choose $w(x)$ such that the values of x distributed according to $w(x)$ can be generated by the inverse transform method. Let the total area under the curve $w(x)$ be equal to A.
2. Generate a uniform random number in the interval $[0, A]$ and use it to obtain a corresponding value of x distributed according to $w(x)$.
3. For the value of x generated in step 2, generate a uniform random number y in the interval $[0, w(x)]$. The point (x, y) is uniformly distributed in the area under the comparison function $w(x)$. If $y \leq p(x)$, then accept x as a random number distributed according to $p(x)$.

Repeat steps 2 and 3 many times. Note that the acceptance-rejection method is efficient only if the comparison function $w(x)$ is close to $p(x)$ over the entire range of interest.

Appendix 11D: Polynomials and Interpolation

Interpolation is a technique that allows us to estimate a function within the range of a tabulated set of *sample points*. For example, Fourier analysis (see Chapter 9) generates a trigonometric series that can be evaluated between the points that are used to calculate the coefficients. We now describe how polynomials are implemented and used to interpolate between sample points.

A polynomial is a function that is expressed as

$$p(x) = \sum_{i=0}^{n} a_i x^i, \tag{11.81}$$

where n is the *degree* of the polynomial and the n constants a_i are the *coefficients*. The evaluation of (11.81) as written is very inefficient because x is repeatedly multiplied by itself and the entire sum requires $\mathcal{O}(N^2)$ multiplications. A more efficient algorithm was published in 1819 by W. G. Horner.[1] It uses a factored polynomial and requires only n multiplications and n additions and is known as *Horner's rule*. It is written as follows:

$$p(x) = a_0 + x\Big[a_1 + x\big[a_2 + x[a_3 + \cdots]\big]\Big]. \tag{11.82}$$

[1] This method of evaluating polynomials by factoring was already known to Newton.

Polynomial methods	
`add(double a)`	Adds a scalar to this polynomial and returns a new polynomial.
`add(Polynomial p)`	Adds a polynomial to this polynomial and returns a new polynomial.
`deflate(double r)`	Reduces the degree of this polynomial by removing the given root `r`.
`derivative()`	Returns the derivative of this polynomial.
`integral(double a)`	Returns the integral of this polynomial having the value a at $x = 0$.
`roots(double tol)`	Gets the roots of this polynomial using Newton's method.
`subtract(double a)`	Subtracts a scalar from this polynomial and returns a new polynomial.
`subtract(Polynomial p)`	Subtracts a polynomial from this polynomial and returns a new polynomial.

Table 11.4: Some of the methods for manipulating polynomials in the Open Source Physics library.

This algorithm can dramatically reduce processor time if large polynomials are repeatedly evaluated.

Polynomials are important computationally because most analytic functions can be approximated as a polynomial using a Taylor series expansion. Polynomials can be added, multiplied, integrated, and differentiated analytically and the result is still a polynomial. The `Polynomial` class in the numerics package implements many of these algebraic operations (see Table 11.4). Listing 11.5 shows how this class is used to calculate and display a polynomial's roots.

Listing 11.5: The `PolynomialApp` class tests the `Polynomial` class.

```
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.*;

public class PolynomialApp extends AbstractCalculation {
   PlotFrame plotFrame = new PlotFrame("x", "f(x)", "Polynomial visualization");
   double xmin, xmax;
   Polynomial p;

   public void reset() {
      control.setValue("coefficients", "-2,0,1");
      control.setValue("minimum x", -10);
      control.setValue("maximum x", 10);
   }

   public void calculate() {
      xmin = control.getDouble("minimum x");
      xmax = control.getDouble("maximum x");
      String[] coefficients = control.getString("coefficients").split(",");
```

```
      p = new Polynomial(coefficients);
      plotAndCalculateRoots();
   }

   void plotAndCalculateRoots() {
      plotFrame.clearDrawables();
      plotFrame.addDrawable(new FunctionDrawer(p));
      double[] range = Util.getRange(p, xmin, xmax, 100); // finds ymin and ymax within (xmi
      plotFrame.setPreferredMinMax(xmin, xmax, range[0], range[1]);
      plotFrame.repaint();
      double[] roots = p.roots(0.001); //desired precision of roots is 0.001
      control.clearMessages();
      control.println("polynomial = "+p);
      for(int i = 0, n = roots.length;i<n;i++) {
         control.println("root = "+roots[i]); // print each root
      }
   }

   public void derivative() {
      p = p.derivative();
      plotAndCalculateRoots();
   }

   public static void main(String[] args) {
      CalculationControl control = CalculationControl.createApp(new PolynomialApp());
      control.addButton("derivative", "Derivative", "The derivative of the polynomial.");
   }
}
```

Exercise 11.23. Taylor series

Use the `PolynomialApp` class to plot the first three nonzero terms of the Taylor series expansion of the sine function. How accurate is this expansion in the interval $|x| < \pi/2$?

Exercise 11.24. Polynomials

Write a test program to do the following:

a. Create the polynomial $x^4 - 5x^3 + 5x^2 + 5x - 6$ and divide this polynomial by $x - 2$. Is 2 a root of the original polynomial?

b. Find the roots of $x^5 - 6x^4 + x^3 - 7x^2 - 7x + 12$.

Problem 11.25. Chebyshev polynomials

Orthogonal polynomials often can be written in terms of simple recurrence relations. For example, the Chebyshev polynomials of the first kind, $T_n(x)$, can be written as

$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x), \tag{11.83}$$

where $T_0(x) = 1$ and $T_1(x) = x$. Write and test a class using a static method that creates the Chebyshev polynomials. To improve efficiency, your class should store the polynomials as they are created during recursion.

It always is possible to construct a polynomial that passes through a set of n data points (x_i, y_i) by creating a *Lagrange interpolating polynomial* as follows:

$$p(x) = \sum_{i=0}^{n} \frac{\prod_{i \neq j}(x - x_j)}{\prod_{i \neq j}(x_- x_j)} y_i. \tag{11.84}$$

For example, three data points generate the second-degree polynomial (see (11.5)):

$$p(x) = \frac{(x - x_1)(x_0 - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_3. \tag{11.85}$$

Note that terms multiplying the y values are zero at the sample data points except for the term multiplying the sample data point's abscissa y_i. Various computational tricks can be used to speed the evaluation of (11.84), but these will not be discussed here (see Besset or Press et al.). We have implemented Lagrange's polynomial interpolation formula using a generalized Horner expansion in the `LagrangeInterpolator` class in the numerics package. Listing 11.6 tests this class.

Listing 11.6: The `LagrangeInterpolatorApp` class tests the `LagrangeInterpolator` class by sampling an arbitrary function and fitting the samples by a polynomial.

```
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
import org.opensourcephysics.numerics.*;

public class LagrangeInterpolatorApp extends AbstractCalculation {
   PlotFrame plotFrame = new PlotFrame("x", "f(x)", "Lagrange interpolation");

   public void reset() {
      control.setValue("f(x)", "sin(x)");
      control.setValue("minimum x", -2);
      control.setValue("maximum x", 2);
      control.setValue("n", 5);
      control.setValue("random y-error", 0);
      calculate();
   }

   public void calculate() {
      String fstring = control.getString("f(x)");
      double a = control.getDouble("minimum x");
      double b = control.getDouble("maximum x");
      double err = control.getDouble("random y-error");
      int n = control.getInt("n"); // number of intervals
      double[] xData = new double[n];
      double[] yData = new double[n];
```

```
      double dx = (n>1) ? (b-a)/(n-1) : 0;
      Function f;
      try {
         f = new ParsedFunction(fstring);
      } catch(ParserException ex) {
         control.println(ex.getMessage());
         return;
      }
      plotFrame.clearData();
      double[] range = Util.getRange(f, a, b, 100);
      plotFrame.setPreferredMinMax(a-(b-a)/4, b+(b-a)/4, range[0], range[1]);
      FunctionDrawer func = new FunctionDrawer(f);
      func.color = java.awt.Color.RED;
      plotFrame.addDrawable(func);
      double x = a;
      for(int i = 0;i<n;i++) {
         xData[i] = x;
         yData[i] = f.evaluate(x)*(1+err*(-0.5+Math.random()));
         plotFrame.append(0, xData[i], yData[i]);
         x += dx;
      }
      LagrangeInterpolator interpolator = new LagrangeInterpolator(xData, yData);
      plotFrame.addDrawable(new FunctionDrawer(interpolator));
      double[] coef = interpolator.getCoefficients();
      for(int i = 0;i<coef.length;i++) {
         control.println("c["+i+"]="+coef[i]);
      }
   }

   public static void main(String[] args) {
      CalculationControl.createApp(new LagrangeInterpolatorApp());
   }
}
```

Problem 11.26. Lagrange interpolation

Use the `LagrangeInterpolatorApp` class to answer the following questions.

a. How do the interpolating polynomial's coefficients compare to the series expansion coefficients of the sine and exponential functions?

b. How well does an interpolating polynomial match a unit step function? You will need to write a step function class that implements the `Function` interface and thus contains an `evaluate` method.

c. Do your answers depend on the number of sample points?

d. Add random error to the sample data points for each function. How sensitive is the fit to random errors?

Lagrange polynomials should be used cautiously. If the degree of the polynomial is high, the distance between points is large, or if the points are subject to experimental error, the resulting polynomial can oscillate wildly. Press et al. recommend that interpolating polynomials be small. If the data is accurate but the amount of data is large, we often use a polynomial constructed from a small number of nearest neighbors. *Cubic spline* interpolation uses polynomials in this way.

A cubic spline is a third-order polynomial that is required to have a continuous second derivative with neighboring splines at its end points. Because it would be inefficient to store a large number of `Polynomial` objects, the `CubicSpline` class in the numerics package stores the coefficients for the multiple polynomials needed to fit a data set in a single array. The `CubicSplineApp` program tests this class, but it is not shown here because it is similar to `LagrangeInterpolatorApp`.

Exercise 11.27. Cubic splines

Compare the cubic spline interpolating function to the Lagrange polynomial interpolating function using the same samples as were used in Problem 11.26.

If the sample data is inaccurate, we often compute the coefficients for a polynomial of lower degree that passes as close as possible to the sample points. This fitting procedure often is used to construct an *ad hoc* function that describes the experimental data. The `PolynomialLeastSquareFit` class in the numerics package implements such a fitting algorithm (see Besset) and the `PolynomialFitApp` program tests this class. It is not shown because it is similar to `LagrangeInterpolatorApp`.

Exercise 11.28. Polynomial fitting

The `PolynomialFitApp` simulates experimental data from a particle trajectory near the Earth. How large a relative error in the y-values can be tolerated if we wish to determine the acceleration of gravity to within ten percent? How does this answer change if the number of samples is increased by a factor of two? Four? Discuss the effects of changing the the degree of the fitting polynomial.

Suppose you are given a table of $y_i = f(x_i)$ and are asked to determine the value of x that corresponds to a given y. In other words, how do we find the inverse function $x = f^{-1}(x)$? An interpolation routine that does not require evenly spaced ordinates, such as the `CubicSpline` class, provides an easy and effective solution. The following code uses this technique to define the arcsin function.

Listing 11.7: The `Arcsin` class demonstrates how to use interpolation to define an inverse function.

```
package org.opensourcephysics.sip.ch11;
import org.opensourcephysics.numerics.*;

public class Arcsin {
   static Function arcsin;

   private Arcsin() {} // probibit instantiation because all methods are static

   static public double evaluate(double x) {
      if((x<-1)||(x>1)) {
```

```
            return Double.NaN;
        } else {
            return arcsin.evaluate(x);
        }
    }

    static { // creates the static function arcsin when class is loaded
        int n = 10;
        double[] xValues = new double[n];
        double[] yValues = new double[n];
        double
            x = -Math.PI/2, dx = Math.PI/(n-1);
        for(int i = 0;i<n;i++) {
            xValues[i] = x;
            yValues[i] = Math.sin(x);
            x += dx;
        }
        arcsin = new CubicSpline(yValues, xValues);
    }
}
```

Problem 11.29. Inverse functions

a. How accurate is the $\arcsin x$ function shown in Listing 11.7 in the interval $|x| < 0.5$?

b. Compare the number of tabulated points need to produce relative accuracies of $1 : 10^2$, $1 : 10^3$, and $1 : 10^4$ in the interval $-0.5 < x < 0.5$.

c. Is polynomial interpolation more or less efficient than spline interpolation for evaluating inverse functions?

d. Discuss the accuracy of the inverse interpolation of $\sin x$ if the interval is extended to $|x| \leq 1$.

References and Suggestions for Further Reading

Forman S. Acton, *Numerical Methods That Work*, Harper & Row (1970); corrected edition, Mathematical Association of America (1990). A delightful book on numerical methods.

Didier H. Besset, *Object-Oriented Implementation of Numerical Methods*, Morgan Kaufmann (2001).

Isabel Beichl and Francis Sullivan, "The importance of importance sampling," Computing in Science and Engineering **1** (2), 71–73 (1999).

P. H. Borcherds, "Importance sampling: An illustrative introduction," Eur. J. Phys. **21**, 405–411 (2000).

Bradley Efron and Robert J. Tibshirani, *An Introduction to the Bootstrap*, Chapman and Hall (1993).

Julio Fernández and Carlos Criado, "Algorithm for normal random numbers," Phys. Rev. E **60**, 3361–3365 (1999).

Steven E. Koonin and Dawn C. Meredith, *Computational Physics*, Addison-Wesley (1990). Chapter 8 covers much of the same material on Monte Carlo methods as discussed in this chapter.

Malvin H. Kalos and Paula A. Whitlock, *Monte Carlo Methods, Vol. 1: Basics*, John Wiley & Sons (1986). The authors are well known experts on Monte Carlo methods.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, second edition, Cambridge University Press (1992).

Reuven Y. Rubinstein, *Simulation and the Monte Carlo Method*, John Wiley & Sons (1981). An advanced, but clearly written treatment of Monte Carlo methods.

I. M. Sobol, *The Monte Carlo Method*, Mir Publishing (1975). A very readable short text with excellent sections on nonuniform probability densities and the neutron transport problem.