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.

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

Method	Description
`add(double a)`	Adds a scalar and returns a new polynomial.
`add(Polynomial p)`	Adds a polynomial and returns a new polynomial.
`deflate(double r)`	Reduces the degree by removing the given root `r`.
`derivative()`	Returns the derivative.
`integral(double a)`	Returns the integral with integration constant a.
`roots(double tol)`	Gets the roots using Newton's method.
`subtract(double a)`	Subtracts a scalar and returns a new polynomial.
`subtract(Polynomial p)`	Subtracts a polynomial and returns a new polynomial.

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:

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

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");
```

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