Numerical Algorithms

Numerical algorithms calculate numbers. They perform such tasks as randomizing values, breaking numbers into their prime factors, finding greatest common divisors, and computing geometric areas.

All of these algorithms are useful occasionally, but they also demonstrate useful algorithmic techniques such as adaptive algorithms, Monte Carlo simulation, and using tables to store intermediate results.

Finding Greatest Common Divisors

The *greatest common divisor* (GCD) of two integers is the largest integer that evenly divides both of the numbers. For example, GCD(60, 24) is 12 because 12 is the largest integer that evenly divides both 60 and 24. (The GCD may seem like an esoteric function, but it is actually quite useful in cryptographic routines that are widely used in business to keep such things as financial communications secure.)

NOTE **If GCD(A, B) = 1, A and B are said to be *relatively prime* or *coprime*.**

The following section explains an algorithm for finding greatest common divisors. The section after that describes an extension that lets you find an equation related to greatest common divisors.

Calculating Greatest Common Divisors

One way to find the GCD is to factor the two numbers and see which factors they have in common. However, the Greek mathematician Euclid recorded a faster method in his treatise, *Elements,* circa 300 BC.

Working with Prime Numbers

As you probably know, a *prime number* (or simply *prime*) is a counting number (an integer greater than 0) greater than 1 whose only factors are 1 and itself. A *composite number* is a counting number greater than 1 that is not prime.

Prime numbers play important roles in some applications where their special properties make certain operations easier or more difficult. For example, some kinds of cryptography use the product of two large primes to provide security. The fact that it is hard to factor a number that is the product of two large primes is what makes the algorithm secure.

The following sections discuss common algorithms that deal with prime numbers.

Finding Prime Factors

The most obvious way to find a number's prime factors is to try dividing the number by all of the numbers between 2 and 1 less than the number. When a possible factor divides the number evenly, save the factor, divide the number by it, and continue trying more possible factors. Note that you need to try the same factor again before moving on in case the number contains more than one copy of the factor.

For example, to find the prime factors of 127, you would try to divide 127 by 2, 3, 4, 5, and so on, until you reach 126.

Finding Primes

Suppose that your program needs to pick a large prime number (yet another task required by some cryptographic algorithms). One way to find prime numbers is to use the algorithm described in the preceding section to test a bunch of numbers to see whether they are prime. For reasonably small numbers, that works, but for large numbers, it can be prohibitively slow.

The *sieve of Eratosthenes* is a simple method you can use to find all of the primes up to a given limit. This method works well for reasonably small numbers, but it requires a table with entries for every number that is considered. Therefore, it uses an unreasonable amount of memory if the numbers are too large.

The basic idea is to make a table with one entry for each of the numbers between 2 and the upper limit. Cross out all of the multiples of 2 (not counting 2 itself). Then, starting at 2, look through the table to find the next number that is not crossed out (3 in this case). Cross out all multiples of that value (not counting the value itself). Note that some of the values may already be crossed out because they were also a multiple of 2. Repeat this step, finding the next value that is not crossed out and crossing out its multiples until you reach the square root of the upper limit. At that point, any numbers that are not crossed out are prime.

Performing Numerical Integration

Numerical integration, which is also sometimes called *quadrature* or *numeric quadrature*, is the process of using numerical techniques to approximate the area under a curve defined by a function. Often, the function has one variable so it looks like $y = F(x)$ and the result is a two-dimensional area, but some applications might need to calculate the three-dimensional volume under a surface defined by a function $z = F(x, y)$. You could even calculate areas defined by higher-dimensional functions.

If the function is easy to understand, you may be able to use calculus to find the exact area. Unfortunately, you may not always be able to calculate the function's antiderivative. For example, the function's equation might be very complicated, or you might have data generated by some physical process, so you don't know the function's equation. In that case, you can't use calculus, but you *can* use numerical integration.

There are several ways to perform numerical integration. The most straightforward involve Newton-Cotes formulas, which use a series of polynomials to approximate the function. The two most basic kinds of Newton-Cotes formulas are the rectangle rule and the trapezoid rule.

The Rectangle Rule

The *rectangle rule* uses a series of rectangles of uniform width to approximate the area under a curve. Figure 2.6 shows the RectangleRule sample program (which is available for download on the book's website) using the rectangle rule. The program also uses calculus to find the exact area under the curve so that you can see how far the rectangle rule is from the correct result.

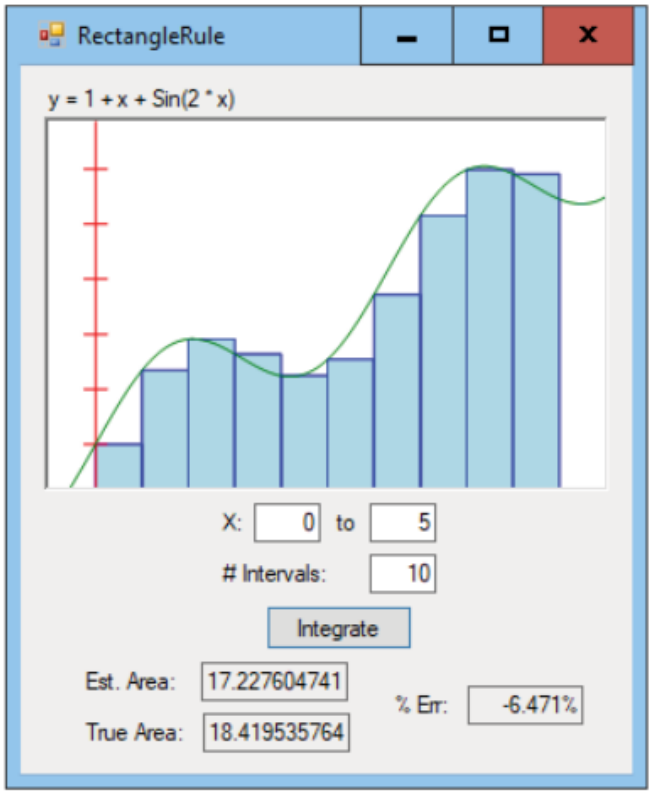

Figure 2.6: The RectangleRule sample program uses the rectangle rule to approximate the area under the curve $y = 1 + x + \mathrm{Sin}(2x)$.

The algorithm simply divides the area into rectangles of constant width and with height equal to the value of the function at the rectangle's left edge. It then loops over the rectangles and adds their areas.

The Trapezoid Rule

You can see in Figure 2.6 where the rectangles don't fit the curve exactly, producing an error in the total calculated area. You can reduce the error by using more, skinnier rectangles. In this example, increasing the number of rectangles from 10 to 20 reduces the error from roughly –6.5% to –3.1%.

An alternative strategy is to use trapezoids to approximate the curve instead of using rectangles. Figure 2.7 shows the TrapezoidRule sample program (which is available for download on the book's website) using the *trapezoid rule*.

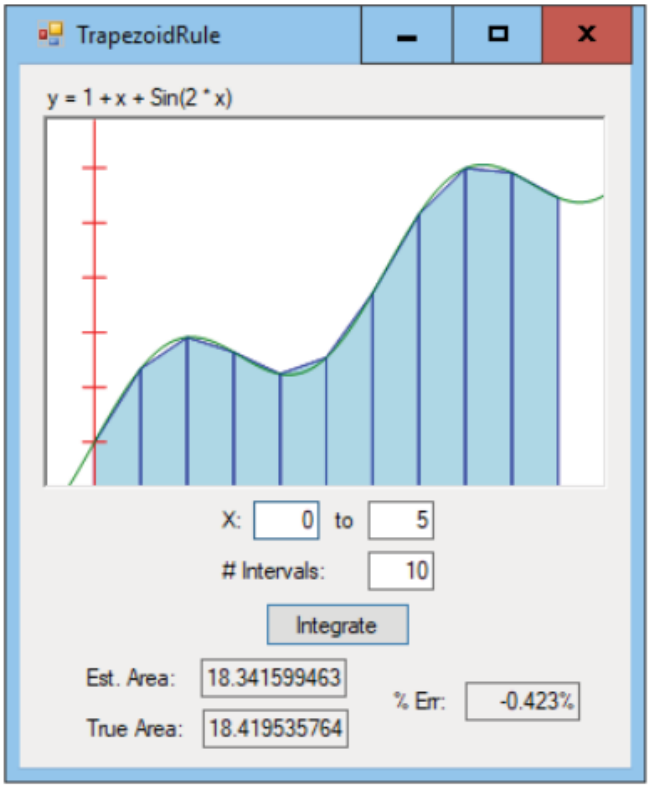

Figure 2.7: The TrapezoidRule sample program uses the trapezoid rule to make a better approximation than the RectangleRule program does.

The only difference between this algorithm and the rectangle rule algorithm is in the statement that adds the area of each slice. This algorithm uses the formula for the area of a trapezoid: area = width × average of the lengths of the parallel sides.

You can think of the rectangle rule as approximating the curve with a step function that jumps from one value to another at each rectangle's edge. The trapezoid rule approximates the curve with line segments.

Another example of a Newton-Cotes formula is *Simpson's rule*, which uses polynomials of degree 2 to approximate the curve. Other methods use polynomials of even higher degree to make better approximations of the curve.

Adaptive Quadrature

A variation on the numerical integration methods described so far is *adaptive quadrature*, in which the program detects areas where its approximation method may produce large errors and refines its method in those areas.

For example, look again at Figure 2.7. In areas where the curve is close to straight, the trapezoids approximate the curve very closely. In areas where the curve is bending sharply, the trapezoids don't fit as well.

A program using adaptive quadrature looks for areas where the trapezoids don't fit the curve well and uses more trapezoids in those areas.

The AdaptiveMidpointIntegration sample program, shown in Figure 2.8, uses the trapezoid rule with adaptive quadrature. When calculating the area of a slice, this program first uses a single trapezoid to approximate its area. It then breaks the slice into two pieces and uses two smaller trapezoids to calculate their areas. If the difference between the larger trapezoid's area and the sum of the areas of the smaller trapezoids is more than a certain percentage, the program divides the slice into two pieces and calculates the areas of the pieces in the same way.

If you run the AdaptiveMidpointIntegration program and start with only two initial slices, the program divides them into the 24 slices shown in Figure 2.8 and estimates the area under the curve with –0.035% error. If you use the TrapezoidRule program with 24 slices of uniform width, the program has an error of –0.072%, roughly twice as much as that produced by the adaptive program. The two programs use the same number of slices, but the adaptive program positions them more effectively.

The AdaptiveTrapezoidIntegration sample program uses a different method to decide when to break a slice into subslices. It calculates the second derivative

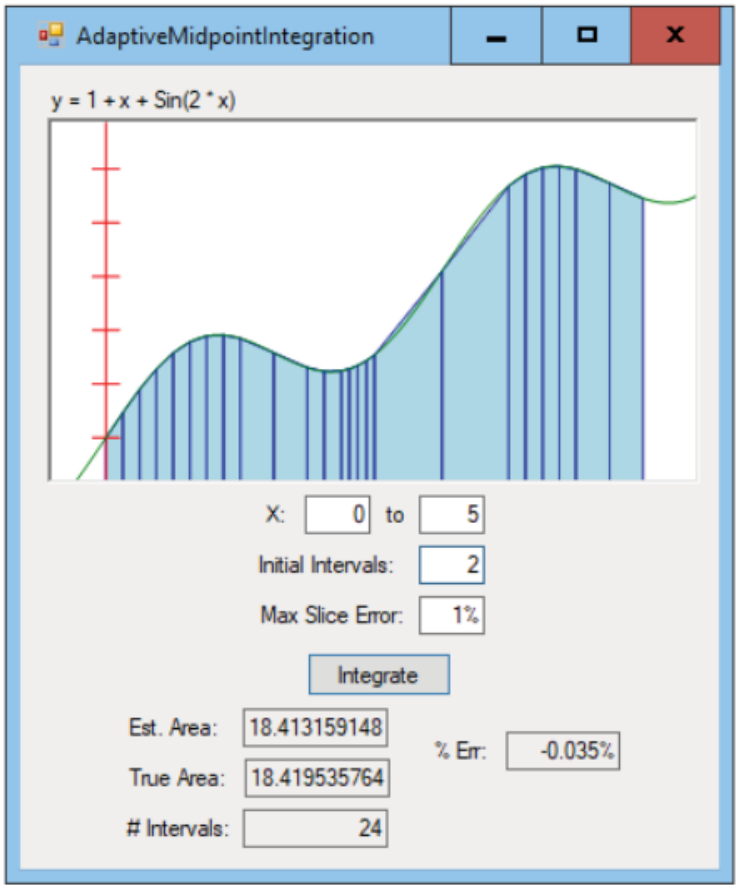

Figure 2.8: The AdaptiveMidpointIntegration program uses an adaptive trapezoid rule to make a better approximation than the TrapezoidRule program.

of the function at the slice's starting x value and divides the interval into one slice plus 1 per second derivative value. For example, if the second derivative is 2, the program divides the slice into three pieces. (The formula for the number of slices was chosen somewhat arbitrarily. You might get better results with a different formula.)

NOTE **In case your calculus is a bit rusty, a function's derivative tells you its slope at any given point. Its second derivative tells you the slope's rate of change, or how fast the curve is bending. A higher second derivative means that the curve is bending relatively tightly, so the AdaptiveTrapezoidIntegration program uses more slices.**

Of course, this technique won't work if you can't calculate the curve's second derivative. The technique used by the AdaptiveMidpointIntegration program seems to work fairly well in any case, so you can fall back on that technique.

Adaptive techniques are useful in many algorithms because they can produce better results without wasting effort in areas where it isn't needed. The AdaptiveGridIntegration program shown in Figure 2.9 uses adaptive techniques to estimate the area in the shaded region. This region includes the union of vertical and horizontal ellipses, minus the areas covered by the three circles inside the ellipses.

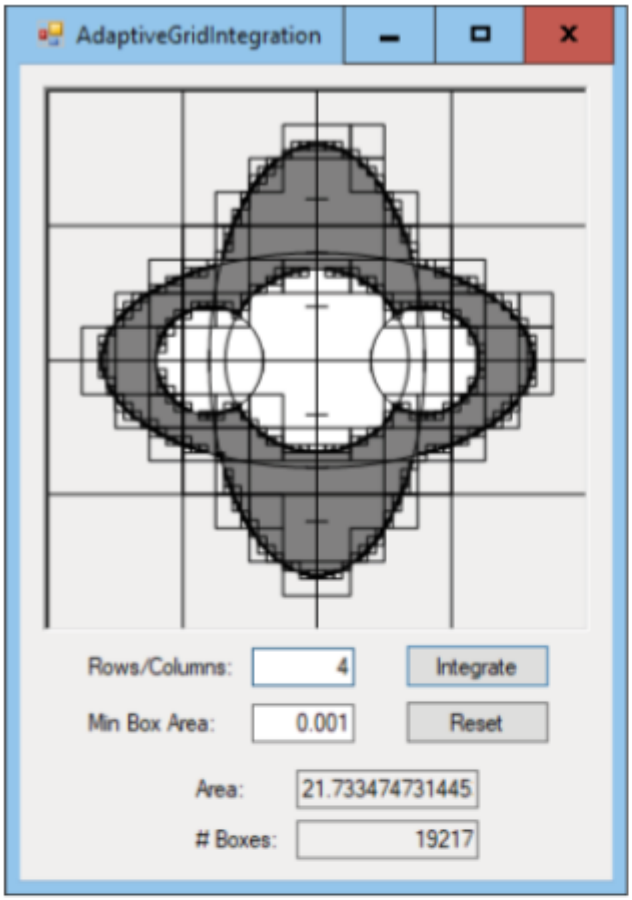

Figure 2.9: The AdaptiveGridIntegration program uses adaptive integration to estimate the area in the shaded region.

This program divides the whole image into a single box and defines a grid of points inside the box. In Figure 2.9, the program uses a grid with four rows and columns of points. For each point in the grid, the program determines whether the point lies inside or outside the shaded region.

If none of the points in the box lies within the shaded region, the program assumes that the box is not inside the region and ignores it.

If every point in the box lies inside the shaded region, the program considers the box to lie completely within the region and adds the box's area to the region's estimated area.

If some of the points in the box lie inside the shaded region and some lie outside the region, the program subdivides the box into smaller boxes and uses the same technique to calculate the smaller boxes' areas.

In Figure 2.9, the AdaptiveGridIntegration program has drawn the boxes it considered so that you can see them. You can see that the program considered many more boxes near the edges of the shaded region than far inside or outside the region. In total, this example considered 19,217 boxes, mostly focused on the edges of the area it was integrating.

Finding Zeros

Sometimes a program needs to figure out where an equation crosses the x-axis. In other words, given an equation $y = f(x)$, you may want to find x where $f(x) = 0$. Values such as this are called the equation's *roots*.

Newton's method, which is sometimes called the *Newton-Raphson method*, is a way to approximate an equation's roots successively.

The method starts with an initial guess X_0 for the root. If $f(X_0)$ is not close enough to 0, the algorithm follows a line that is tangent to the function at the point X_0 until the line hits the x-axis. It uses the x-coordinate at the intersection as a new guess X_1 for the root.

The algorithm then repeats the process starting from the new guess X_1. The algorithm continues the process of following tangents to the function to find new guesses until it finds a value X_k where $f(X_k)$ is sufficiently close to 0.

The only tricky part is figuring out how to follow tangent lines. If you use a little calculus to find the derivative of the function $f(x)$, which is also written $df/dx(x)$, then the following equation shows how the algorithm can update its guess by following a tangent line:

$$X_{i+1} = X_i = -\frac{f(X_i)}{f'(X_i)}$$

NOTE **Unfortunately, explaining how to find a function's derivative is outside the scope of this book. For more information, search online or consult a calculus book.**

Figure 2.11 shows the process graphically. The point corresponding to the initial guess is labeled 1. That point's y value is far from 0, so the algorithm follows the tangent line until it hits the x-axis. It then calculates the function at the new guess to get the point labeled 2 in Figure 2.11. This point's y-coordinate is also far from 0, so the algorithm repeats the process to find the next guess, labeled 3. The algorithm repeats the process one more time to find the point labeled 4. Point 4's y-coordinate is close enough to 0, so the algorithm stops.

The following pseudocode shows the algorithm:

```
// Use Newton's method to find a root of the function f(x).
Float: NewtonsMethod(Float: f(), Float: dfdx(), Float: initial_guess,
  Float: maxError)

    float x = initial_guess
    For i = 1 To 100  // Stop at 100 in case something goes wrong.
        // Calculate this point.
        float y = f(x)

        // If we have a small enough error, stop.
        if (Math.Abs(y) < maxError) break

        // Update x.
        x = x - y / dfdx(x)
    Next i

    Return x
End NewtonsMethod
```

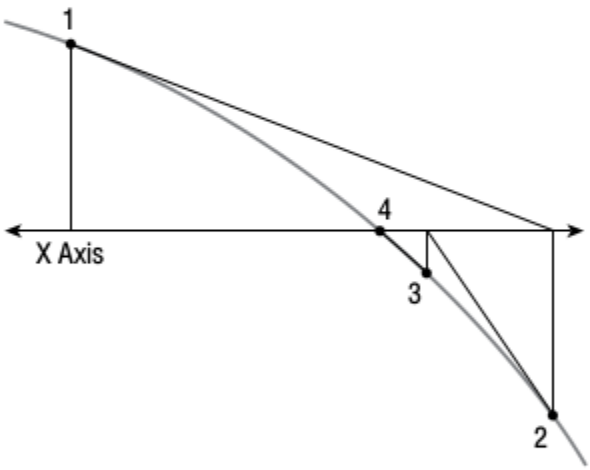

Figure 2.11: Newton's method follows a function's tangent lines to zero in on the function's roots.

The algorithm takes as parameters a function `y = f(x)`, the function's derivative `dfdx`, an initial guess for the root's value, and a maximum acceptable error.

The code sets the variable `x` equal to the initial guess and then enters a `For` loop that repeats, at most, 100 times. Normally, the algorithm quickly finds a solution. But sometimes, if the function has the right curvature, the algorithm can diverge and not zero in on a solution. Or it can get stuck jumping back and forth between two different guesses. The maximum of 100 iterations means the program cannot get stuck forever.

Within the `For` loop, the algorithm calculates `f(x)`. If the result isn't close enough to 0, the algorithm updates `x` and tries again.

Note that some functions have more than one root. In that case, you need to use the FindZero algorithm repeatedly with different initial guesses to find each root.

Figure 2.12 shows the NewtonsMethod sample program, which is available for download on this book's website. This program uses Newton's method three times to find the three roots of the function $y = x^3 / 5 - x^2 + x$. Circles show the program's guesses as it searches for each root.

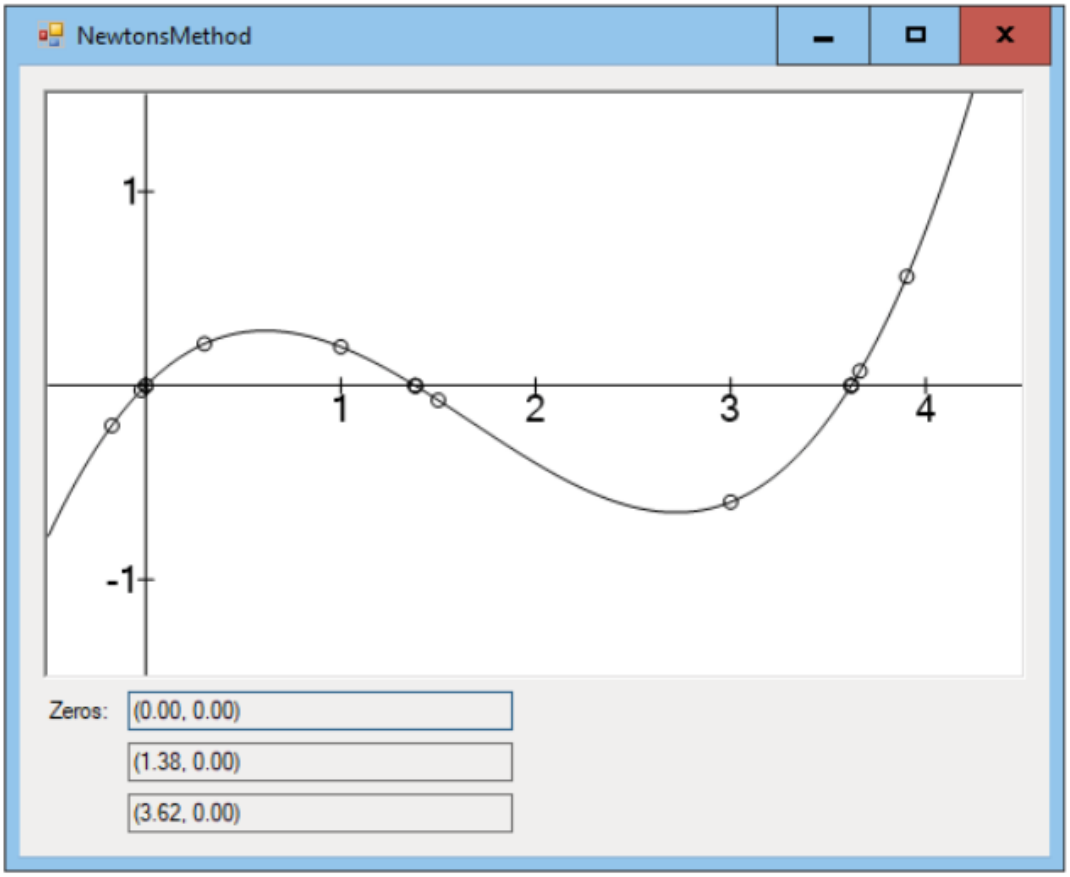

Figure 2.12: The NewtonsMethod sample program demonstrates Newton's method to find the three roots of the function $y = x^3 / 5 - x^2 + x$.

Gaussian Elimination

Root finding algorithms let you find values for X that make the equation $y = f(x)$ equal to zero. *Gaussian elimination* is a technique that does something similar for a system of linear equations. It attempts to find values for the x's in the following equations to make all of the equations true simultaneously:

$$A_{11} \cdot x_1 + A_{12} \cdot x_2 + \ldots + A_{1n} \cdot x_n = C_1$$

$$A_{21} \cdot x_1 + A_{22} \cdot x_2 + \ldots + A_{2n} \cdot x_n = C_2$$

$$\ldots$$

$$A_{n1} \cdot x_1 + A_{n2} \cdot x_2 + \ldots + A_{nn} \cdot x_n = C_n$$

Here all of the A and C values are numbers given by the problem. For a concrete example, consider the following system of equations:

$$2x_1 + 4x_2 + 6x_3 = -2$$

$$3x_1 + 6x_2 + 7x_3 = 2$$

$$6x_1 + 10x_2 + 4x_3 = 1$$

The goal is to find numbers x_1, x_2, and x_3 that simultaneously satisfy all three equations.

It's easier to work with the equations if you represent them as an augmented matrix. The first entries in each row hold the equations' coefficients (the A values). An extra final column holds the C values. The following shows the augmented matrix for the preceding equations:

$$\left|\begin{array}{cccc} 2 & 4 & 6 & -2 \\ 3 & 6 & 7 & 2 \\ 6 & 10 & 4 & 1 \end{array}\right|$$

NOTE **Often, people draw a vertical line separating the final column containing the C values from the other columns that hold the A values.**

Gaussian elimination works in two stages that are sometimes called *forward elimination* and *back substitution*.

Least Squares Fits

A *least squares fit* attempts to find a function $y = f(x)$ to fit a collection of data values. The result is called a least squares fit because it finds the least possible sum of the squares of the distances between the data points and the corresponding points on the function.

Figure 2.13 shows a function approximating a set of data values. The vertical lines show the distances between the data points and the corresponding points on the function. A least squares fit considers variations of the function and finds the one that minimizes the sum of the squares of those vertical distances.

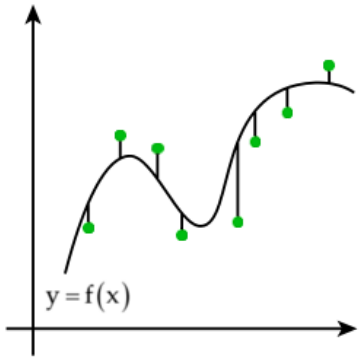

Figure 2.13: A least squares fit minimizes the sum of the squares of the distances between the data points and the function.

Calculating a least squares fit can be intimidating, mostly because it can involve a lot of terms. Fortunately, those terms are often relatively simple. They can look scary when they're all arranged in one grand equation, but individually the terms are easy to manage. I admit that you need to use a little calculus to find a least squares fit. Fortunately, it's pretty easy calculus, so you should be able to understand the following discussion even if you haven't taken a derivative in a while.

The following sections describe two kinds of least squares fits. In the first, the function that approximates the data is a line. In the second, the function is a polynomial of any degree.

Linear Least Squares

In a *linear least squares fit*, the goal is to find a line that minimizes the sum of the squares of the vertical distances between the data points and the line. It's the same situation shown in Figure 2.13, except the curve is a line.

You can represent a line with the equation $y = m\,x + b$ where m is the line's slope and b is its Y-intercept. (The point where the line crosses the y-axis.)

Suppose that you have a set of n data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ Then the vertical distance between one of the points (x_i, y_i) and the line is simply $y_i - (m\,x_i + b)$. The square of that distance is $\left(y_i - (m\,x_i + b)\right)^2$. If you add up all of the squared terms for all of the points, you get the following equation:

$$E = \left(y_0 - (m\,x_0 + b)\right)^2 + \left(y_1 - (m\,x_1 + b)\right)^2 + \ldots + \left(y_n - (m\,x_n + b)\right)^2$$

You can write this more concisely by using the mathematical summation symbol Σ:

$$E = \Sigma\left(y_i - (m x_i + b)\right)^2$$

Here the symbol Σ simply means that you should add up all of the values for $i = 0, 1, 2, \ldots, n$.

This equation looks pretty intimidating in both forms. After all, they include two variables, m and b, plus a bunch of x_i and y_i values. Things are simpler if you remember that the x_i and y_i values are part of the data—they're just numbers like 6 and –13.

Polynomial Least Squares

A linear least squares fit uses a line to fit a set of data points. A *polynomial least squares fit* uses a polynomial of the form $A_0 \cdot x^0 + A_1 \cdot x^1 + A_2 \cdot x^2 + \ldots + A_d \cdot x^d$ to fit the data points.

The *degree* of the polynomial is the largest power of x used by the equation. The preceding equation has degree d. You can pick the degree to fit the data. In general, higher degrees will fit the data points more closely, although they may imply an artificially-high accuracy.

For example, a degree $d - 1$ polynomial can fit d data points exactly, but it may need to wiggle all over the place to do so. Figure 2.14 shows a degree 5 polynomial that exactly fits six data points.

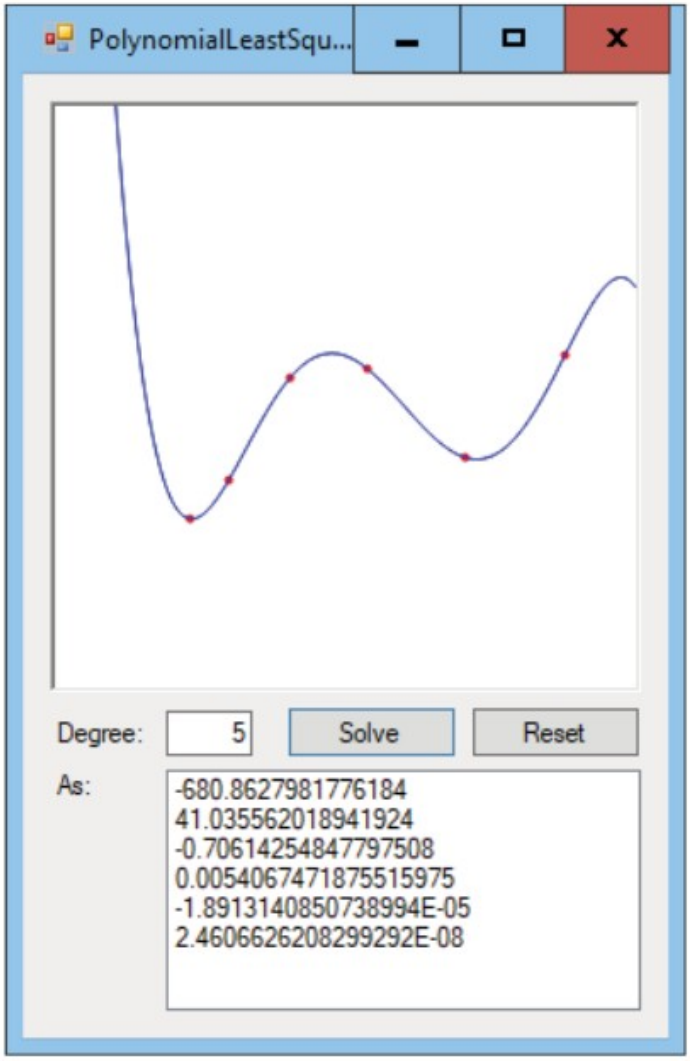

Figure 2.14: A high-degree polynomial may match a set of data values very closely but misleadingly.

It's usually better to pick the smallest degree that fits the data reasonably well. Figure 2.15 shows the same data points as shown in Figure 2.14, but this time fit by a degree 3 polynomial that probably does a better job of representing the data.

You find a polynomial fit in the same way that you find a linear fit: you take the partial derivatives of the error function with respect to the A values, set the derivatives equal to zero, and solve for the `A` values.

The following equation shows the error function:

$$E = \sum\left(y_i - \left(A_0 * x_i^0 + A_1 * x_i^1 + A_2 * x_i^2 + \ldots + A_n * x_i^d\right)\right)^2$$
$$= \sum\left(y_i - A_0 * x_i^0 - A_1 * x_i^1 - A_2 * x_i^2 - \ldots - A_d * x_i^d\right)^2$$

Here the sum is taken over all of the data points (x_i, y_i).

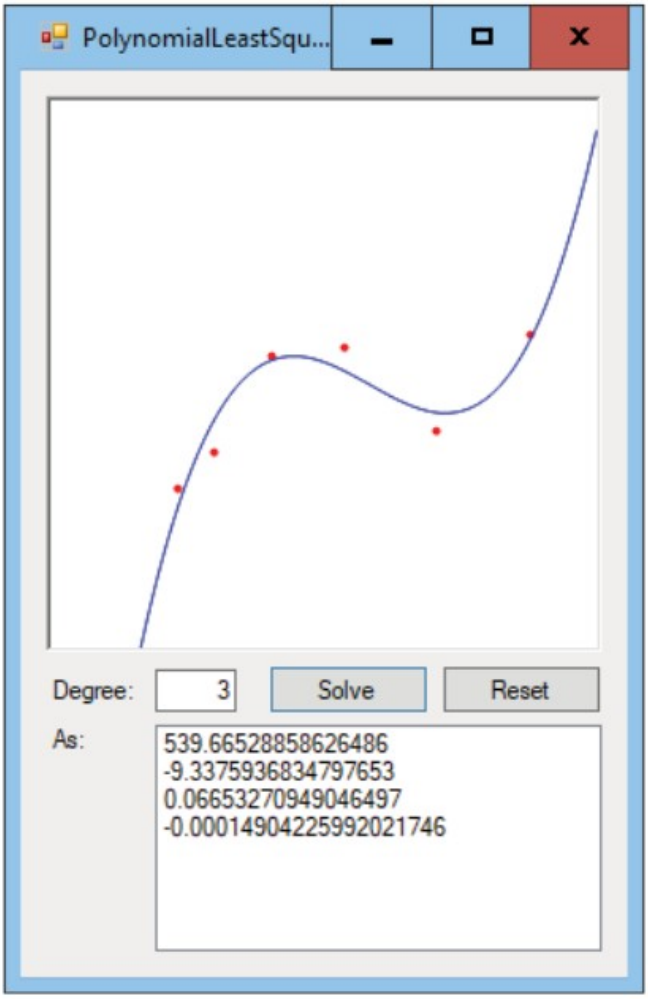

Figure 2.15: You should use lowest-degree polynomial that fits the data reasonably well.