21

Integration of Functions

The task of finding the area enclosed by a given curve is an old problem. The ancient Egyptians, for example, tried to find the area of a circular field by relating it to the area of a square field. The search for a solution to the area problem gave rise to the field of integral calculus, where the integral of a function between two limits is equal to the area under that function. In modern science and engineering, integral equations are widely used to model physical phenomena including things such as aerodynamic analysis and radiation computations.

Integral equations can be divided into two types—proper and improper. A proper integral is one with finite integration limits and whose function can be evaluated at every point along the range of integration. An improper integral is one that has a singularity at some point in the range of integration and/or has one or both integration limits equal to positive or negative infinity. In some cases you can recast an improper integral into a proper one by using a change of variables.

In this chapter we will discuss numerical techniques to solve both proper and improper integrals. In either case the algorithms involve deriving a polynomial expression that, when evaluated, approximates the integral value. The solution process involves dividing the integration range into subelements. Generally speaking, the more subelements used the greater the precision of the final result. Many of the numerical techniques are iterative, beginning with an initial division of the integration range and then refining the solution by increasing the number of subelements until the change in the solution is less than a specified error tolerance.

At the end of the chapter we will briefly discuss the more general families of integrals known as Fredholm and Volterra integrals. We will then use one of the solution methods developed earlier in the chapter to determine the lift and moment coefficient of a NACA 2412 airfoil according to thin airfoil theory. The specific topics we will discuss in this chapter are—

- Trapezoidal algorithms
- Simpson's rule
- Solving improper integrals
- Gaussian quadrature formulas
- General integral types
- Example: thin airfoil theory

General Comments

The general form of a simple integral equation is shown in Eq. (21.1).

$$y = \int_a^b f(x)dx \tag{21.1}$$

Eq. (21.1) is an example of a larger class of integrals known as Fredholm integrals of the first kind. Fredholm integrals are discussed in more detail at the end of this chapter, but for the time being let's consider the simple integral shown in Eq. (21.1). The evaluation of the integral equation computes the area under the curve $f(x)$ from $x = a$ to $x = b$.

If the integration limits a and b are finite numbers and the function $f(x)$ is nonsingular over the integration range, the integral is termed a proper integral. Proper integrals are usually solved using *closed* techniques, those that evaluate the function at the endpoints, $f(a)$ and $f(b)$. If either integration limit is ∞ or $-\infty$, or if there is a singularity in the function $f(x)$ anywhere in the integration range, the integral is improper. An improper integral that evaluates to a finite value is convergent. Convergent improper integrals can be solved using what are known as *open* techniques that only evaluate $f(x)$ between $x = a$ and $x = b$. An improper integral that is infinite is a divergent integral. Obviously, divergent integrals cannot be solved because their values diverge.

Some integrals have precise, closed-form solutions. For those that do not, numerical methods have been developed to approximate the integral value. The numerical methods we will discuss in this chapter approximate the integral value by developing a polynomial expression based on values of the function being integrated at discrete locations along the range of integration. We will start by examining trapezoidal algorithms.

TRAPEZOIDAL ALGORITHMS

Trapezoidal algorithms are a general class of solution methods used to estimate the area under the curve *f*(*x*) from $x = a$ to $x = b$. The algorithms we will discuss in this section will be for proper integrals whose function *f*(*x*) can be evaluated at all points between *a* and *b*. A typical curve is shown in Figure 21.1.

The simplest way to approximate the area under *f*(*x*) is to draw a line between *f*(*a*) and *f*(*b*) and compute the area under the resulting trapezoidal figure.

$$y = \int_a^b f(x)dx = \Delta x\left[\frac{1}{2}f(a) + \frac{1}{2}f(b)\right] \tag{21.2}$$

The Δx term in Eq. (21.2) is the width of the trapezoid and for the two-point trapezoid method is given by the expression $\Delta x = b - a$. Of course for most curves, Eq. (21.2) is not a very good approximation. Looking at the curves in Figure 21.1, we are missing the area between the actual curve and the

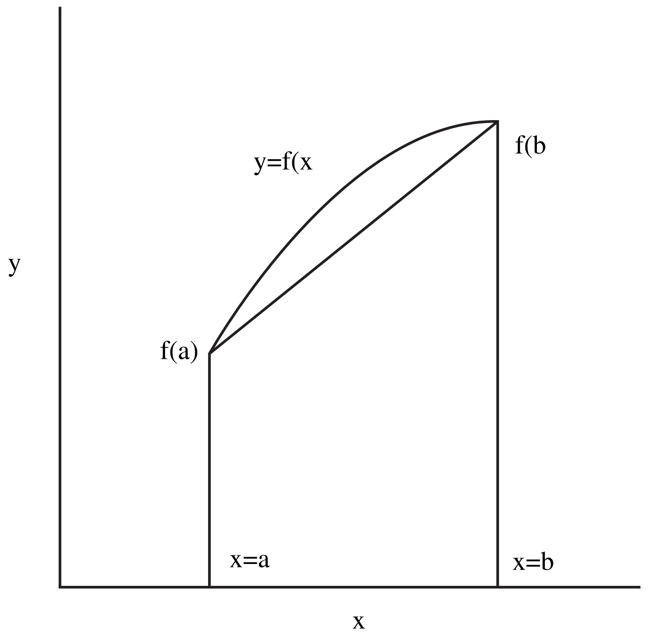

FIGURE 21.1 Two-point trapezoidal method

straight line. To improve the accuracy of the solution, we can divide the range of integration into two sections at the midpoint x location between $x = b$ and $x = a$. In this case, we have to compute the area of two trapezoids, and the approximate value of the integral becomes the following.

$$y = \int_a^b f(x)dx = \Delta x\left[\frac{1}{2}f(a) + f\left(\frac{a+b}{2}\right) + \frac{1}{2}f(b)\right] \quad (21.3)$$

In Eq. (21.3), the width of each of the two trapezoids is half as large as when only one trapezoid was considered so $\Delta x = (b - a)/2$. You can easily see that Eq. (21.3) can be extended to a general form with an arbitrary number of divisions of Δx.

$$y = \int_a^b f(x)dx = \Delta x\left[\frac{1}{2}f_1 + f_2 + f_3 + \ldots + f_{N-1} + \frac{1}{2}f_N\right] \quad (21.4)$$

In Eq. (21.4) $\Delta x = (b - a)/(N - 1)$ where N is the number of trapezoids considered between $x = a$ and $x = b$. Eq. (21.4) is called the *trapezoidal method*. It is a second-order method because its error is proportional to $1/N^2$. One thing to note about the trapezoidal method is that the Δx increments are uniform over the entire integration range.

But how do you know how many times to divide up the integration range for a given integral equation? You keep adding points until the computed integral value changes less than a specified convergence tolerance. If you are clever about how you add points, you can reuse the values from the previous approximation. For example, consider an integration range between $x = 0$ and $x = 1$. If three points are used the expression to evaluate is the following.

$$y_1 = \frac{1}{2}\left[\frac{1}{2}f(0) + f\left(\frac{1}{2}\right) + \frac{1}{2}f(1)\right] \quad (21.5)$$

Keep in mind when looking at Eq. (21.5) that $\Delta x = (b - a)/2 = (1 - 0)/2 = 1/2$. If you add two additional points for the next approximation, the expression becomes Eq. (21.6).

$$y_2 = \frac{1}{4}\left[\frac{1}{2}f(0) + f\left(\frac{1}{4}\right) + f\left(\frac{1}{2}\right) + f\left(\frac{3}{4}\right) + \frac{1}{2}f(1)\right] = \frac{1}{2}y_1 + \frac{1}{4}\left[f\left(\frac{1}{4} + f\left(\frac{3}{4}\right)\right]\right. \quad (21.6)$$

Adding four additional points for the next approximation yields Eq. (21.7).

$$\begin{aligned} y_3 &= \frac{1}{8}\left[\frac{1}{2}f(0) + f\left(\frac{1}{8}\right) + f\left(\frac{1}{4}\right) + f\left(\frac{3}{8}\right) + f\left(\frac{1}{2}\right) + f\left(\frac{5}{8}\right) + f\left(\frac{3}{4}\right) + f\left(\frac{7}{8}\right) + \frac{1}{2}f(1)\right] \\ &= \frac{1}{2}y_2 + \frac{1}{8}\left[f\left(\frac{1}{8}\right) + f\left(\frac{3}{8}\right) + f\left(\frac{5}{8}\right) + f\left(\frac{7}{8}\right)\right] \end{aligned} \quad (21.7)$$

You can see that a pattern is developing. If you start by adding one point to the two-point trapezoidal approximation and then double the number of additional points with each iteration, the next approximation to the integral is one half the previous approximation plus the factors due to the additional points. An equation for a general update to an integral approximation is shown in Eq. (21.8).

$$y_n = \frac{1}{2} y_{n-1} + \Delta x \sum_{j=1}^{2^{n-1}} f\left(\frac{2j - 1}{2^n}\right) \tag{21.8}$$

where

$$\Delta x = \frac{b - a}{2^n} \tag{21.9}$$

Now let's write a method to implement the trapezoidal method. As we did with our system of equation and differential equation solvers, the trapezoidal method will be defined as a `public`, `static` method and its name will be `trapezoidal()`. The method will be defined in a class named `Integrator`. This is a mathematical routine so the `Integrator` class will be placed in the `TechJava.MathLib` package.

The `trapezoidal()` method takes four arguments. The first is a `Function` object that represents the mathematical function being integrated. We'll discuss the `Function` class in more detail later in this section. The second and third arguments define the range of integration. The fourth argument defines the convergence criteria for the iteration that will be performed.

After defining some variables, the `trapezoidal()` method computes an approximation to the integral using the two-point trapezoidal algorithm shown in Eq. (21.2). Subsequent (and better) approximations are produced by successively adding points to the integration range according to Eq. (21.8). When the change in the computed value of the integral is less than the specified convergence tolerance, the method returns. If the iteration does not converge, the value `12345.6` is returned.

The `trapezoidal()` method source code is—

```
package TechJava.MathLib;

public class Integrator
{
    public static double trapezoidal(Function function,
           double start, double end, double tolerance) {
    double dx = end - start;
    double value, oldValue, sum, scale;
    int maxIter = 20;
    //  Compute an initial value for the integral using
    //  the two-point trapezoidal rule.

    value = 0.5*dx*( function.getValue(start) +
                     function.getValue(end) );

    // Subdivide the integration range by adding more
    // points, first 1 point, then two more, then four
    // more and so on. Recompute the integral value
    // each time. If the solution converges, return the
    // value.

    for(int i=0; i<maxIter; ++i) {
      sum = 0.0;
      scale = Math.pow(0.5,i+1);
      for(int j=0; j<Math.pow(2,i); ++j) {
        sum += function.getValue(
                     start + scale*(2*j + 1.0)*dx);
      }

      oldValue = value;
      value = 0.5*value + scale*dx*sum;

      if ( Math.abs(value-oldValue) <=
           tolerance*Math.abs(oldValue) ) {
         return value;
      }
    }

    //  Solution did not converge

    return 12345.6;
  }
}
```

The `trapezoidal()` method makes use of a reference to a `Function` object. The `Function` class is defined to be the abstract parent class of classes that represent mathematical functions. The class declares one method named `getValue()` that simply returns the value of the function evaluated at a given independent variable value. The `Function` class listing is—

```
package TechJava.MathLib;

public abstract class Function
{
   abstract double getValue(double u);
}
```

Let's test the `trapezoidal()` method by having it integrate two mathematical functions. Both functions have an exact, closed-form solution, so an exact value of the integral can be computed. The two functions, their integral solutions, and the numerical values of their solutions are shown in Eq. (21.10) and Eq. (2.11). In Eq. (21.11), if $a = 2$, $y = 7.797853348$.

$$y = \int_{x=2}^{4} x\ln(x)dx = \frac{x^2}{2}\left(\ln(x) - \frac{1}{2}\right)\Bigg|_2^4 = 6.704060528 \tag{21.10}$$

$$y = \int_{x=0}^{3} \sqrt{x^2 + a^2}\,dx = x\frac{\sqrt{x^2 + a^2}}{2} + \frac{a^2}{2}\ln(x + \sqrt{x^2 + a^2})\Bigg|_0^3 \tag{21.11}$$

To use the `trapezoidal()` method on these functions, we will have to create two `Function` subclasses to represent the functions. The `LogFunction` class encapsulates Eq. (21.10). Its code listing is shown here.

```
package TechJava.MathLib;

//  This class represents the function
//  y = u*ln(u)

public class LogFunction extends Function
{
   public double getValue(double u) {
      return u*Math.log(u);
   }
}
```

The `SqrtFunction` class represents the expression shown in Eq. (21.11) and here is its source code.

```
package TechJava.MathLib;

//  This class represents the function
//  y = sqrt(u^2 + a^2)

public class SqrtFunction extends Function
{
   private double a;

   public SqrtFunction(double a) {
      this.a = a;
   }

   public double getValue(double u) {
      return Math.sqrt(u*u + a*a);
   }
}
```

All that remains is to write a driver program that will send the appropriate input to the `trapezoidal()` method and display the results. The `TestTrap` class performs this function. It simply creates a `Function` subclass object and sends a reference to the object to the `trapezoidal()` method. The computed integral value is then printed to standard output. The `TestTrap` class source code is shown next.

```
import TechJava.MathLib.*;

public class TestTrap
{
  public static void main(String args[]) {

    double value;

    //  Integrate the function y = x*ln(x) from
    //  x=2 to x=4 with an error tolerance of 1.0e-4

    LogFunction function = new LogFunction();
    value = Integrator.trapezoidal(
                         function,2.0,4.0,1.0e-4);
    System.out.println("value = "+value);

    //  Integrate the function y = sqrt(x^2 + 2^2) from
    //  x=0 to x=3 with an error tolerance of 1.0e-4

    SqrtFunction function2 = new SqrtFunction(2.0);
    value = Integrator.trapezoidal(
                         function2,0.0,3.0,1.0e-4);
    System.out.println("value = "+value);
  }
}
```

Output—

```
value = 6.704116936052852
value = 7.798005701125523
```

The results from the `trapezoidal()` method are the same as the exact results to within four decimal places, which is the value to which the convergence tolerance was set.

SIMPSON'S RULE

The trapezoidal algorithm is a second-order function integration technique. There are many other integration schemes with the same general form as Eq. (21.4). A commonly used example is the fourth-order technique known as Simpson's rule.

$$y = \int_a^b f(x)dx = \Delta x\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \ldots + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N\right] \qquad (21.12)$$

In Eq. (21.12), the 4/3 and 2/3 factors alternate for the interior points and $\Delta x = (b - a)/(N - 1)$. The number of divisions, N, must be odd since Simpson's rule uses parabolas rather than lines to approximate the intervals.

Simpson's rule shares a common characteristic with the trapezoidal algorithm in that one approximation can be used as a building block toward a more accurate approximation. In the case of Simpson's rule, this feature is accomplished by adding three times the points that were added in the previous approximation. Another interesting feature of Simpson's rule is that the expression for any given level of approximation is related to the corresponding trapezoidal rule expressions. For example, consider an integration range between $x = 0$ and $x = 1$. If you multiply 4/3 times Eq. (21.6) and subtract 1/3 times Eq. (21.5) you obtain the following expression.

$$\frac{4}{3}y_2 - \frac{1}{3}y_1 = \frac{1}{4}\left[\frac{1}{3}f(0) + \frac{4}{3}f\left(\frac{1}{4}\right) + \frac{2}{3}f\left(\frac{1}{2}\right) + \frac{4}{3}f\left(\frac{3}{4}\right) + \frac{1}{3}f(1)\right] \quad (21.13)$$

Eq. (21.13) is the five-point version of Simpson's rule. Indeed, the Simpson's rule value at any given level of refinement can always be expressed in terms of the corresponding trapezoidal expressions.

$$S_n = \frac{4}{3}y_n - \frac{1}{3}y_{n-1} \quad (21.14)$$

The S term in Eq. (21.14) is Simpson's rule's evaluation of the integral and the y terms are the trapezoidal approximations.

Using Eq. (21.14) it is very easy to implement Simpson's rule with only slight modifications to the `trapezoidal()` method. We will call our Simpson's rule method `simpsonsRule()` and place it inside the `Integrator` class. The method computes successive approximations to the integral function with the trapezoidal rule and uses the trapezoidal results to compute Simpson's rule approximations according to Eq. (21.14).

The method takes the same input arguments as the `trapezoidal()` method. After declaring some variables, the method computes initial values of the trapezoidal and Simpson's rule approximations. The `simpsonsRule()` method then goes through the same iteration sequence as the `trapezoidal()` method. Points are added to the integration domain, each iteration adding twice as many points as the one before it. The Simpson's rule approximation is computed from the current and previous trapezoidal values. When the Simpson's rule values converge to within a certain tolerance, the method returns.

The `simpsonsRule()` method source code is—

```
public static double simpsonsRule(Function function,
          double start, double end, double tolerance) {
  double dx = end - start;
  double value, oldValue, sum, scale;
  double valueSimpson, oldValueSimpson;
  int maxIter = 20;

  //  Set initial values for the trapezoidal
  //  method (value) and Simpson's rule (valueSimpson)

  value = 0.5*dx*( function.getValue(start) +
                   function.getValue(end) );
  valueSimpson = 1.0e+6;

  //  Subdivide the integration range by adding more
  //  points, first one point, then two more, then
  //  four more, and so on. Recompute the extended
  //  trapezoidal value and the Simpson's rule value.
  //  If the Simpson's rule value converges, return
  //  the value.

  for(int i=0; i<maxIter; ++i) {
    sum = 0.0;
    scale = Math.pow(0.5,i+1);
    for(int j=0; j<Math.pow(2,i); ++j) {
      sum += function.getValue(
                start + scale*(2*j + 1.0)*dx);
    }

    oldValue = value;
    value = 0.5*value + scale*dx*sum;
    oldValueSimpson = valueSimpson;
    valueSimpson = (4.0*value - oldValue)/3.0;

    if ( Math.abs(valueSimpson-oldValueSimpson) <=
         tolerance*Math.abs(oldValueSimpson) ) {
      return valueSimpson;
    }
  }

  //  Solution did not converge

  return 12345.6;
}
```

Let's apply the `simpsonRule()` method to the same two problems as were solved by the `trapezoidal()` method. The driver to do this computation is called `TestSimpson.java` and its source code is shown here.

```
import TechJava.MathLib.*;

public class TestSimpson
{
  public static void main(String args[]) {

    double value;

    //  Integrate the function y = x*ln(x) from
    //  x=2 to x=4 with an error tolerance of 1.0e-4

    LogFunction function = new LogFunction();
    value = Integrator.simpsonsRule(
                           function,2.0,4.0,1.0e-4);
    System.out.println("value = " + value);

    //  Integrate the function y = sqrt(x^2 + 2^2) from
    //  x=0 to x=3 with an error tolerance of 1.0e-4

    SqrtFunction function2 = new SqrtFunction(2.0);
    value = Integrator.simpsonsRule(
                          function2,0.0,3.0,1.0e-4);
    System.out.println("value = " + value);

  }
}
```

Output—

```
value = 6.704064541919408
value = 7.7978469366358025
```

The `simpsonsRule()` method provides better accuracy than the `trapezoidal()` method with fewer iterations and function evaluations required to achieve a converged solution. The `simpsonsRule()` method results match the exact results to eight and five decimal places respectively. The `simpsonsRule()` method converged in three iterations requiring nine function evaluations. The `trapezoidal()` method, on the other hand, required five iterations and 65 function evaluations to achieve convergence.

Solving Improper Integrals

So far in this chapter, we have been exploring methods to solve proper integrals, expressions where the function can be evaluated across the entire integration range and where the limits of integration are finite numbers. Now we turn to the issue of solving improper integrals. If you recall, an improper integral is one that either has an infinite integration limit or one or more singularities in the range of integration.

In some cases, you can turn an improper integral into a proper one and then apply the trapezoidal or Simpson's rule methods. If one or both of the integration limits is positive or negative infinity you can sometimes perform a change of variables to eliminate the infinite limit. For example, the transformation in Eq. (21.15) could be performed on an integral with an upper integration limit of ∞ to convert the integral from improper to proper.

$$y = \int_a^{\infty} f(x)dx = \int_0^{\frac{1}{a}} \frac{1}{t^2} f\left(\frac{1}{t}\right)dt \tag{21.15}$$

Eq. (21.15) would require that the function $f(x)$ decrease faster than $1/x^2$ as $x \rightarrow \infty$. A similar transformation could be achieved for an integration limit of $-\infty$. If both limits are infinity or if one is infinity and the other 0, the best thing to do is to split the integration into pieces and apply the change of variables to one or both.

Now let's look at the case where the integration limits are finite but there is a singularity at one of the limits. We can't use the trapezoidal or Simpson's rule methods because they evaluate the function at the integration limits. Fortunately, there is a family of what are called *open* formulas that approximate an integral using only points interior to the integration limits. The algorithm we will implement in this chapter is called the *midpoint rule* and is given by the expression in Eq. (21.16).

$$y = \int_a^b f(x)dx = \Delta x\left[f_{\frac{3}{2}} + f_{\frac{5}{2}} + f_{\frac{7}{2}} + \ldots + f_{N-\frac{1}{2}}\right] \tag{21.16}$$

The midpoint method is second-order accurate similar to the trapezoidal algorithm. It is called a midpoint method because the function evaluations are performed midway between the Δx intervals. For example, consider an integration domain between $x = 0$ and $x = 1$. A three-point trapezoidal algorithm would evaluate the function at $x = 0$, $x = 1/2$, and $x = 1$. To model that same situation with the midpoint rule, you would evaluate the equation at $x = 1/4$ and $x = 3/4$.

$$y_1 = \int_a^b f(x)dx = \frac{1}{2}\left[f\left(\frac{1}{4}\right) + f\left(\frac{3}{4}\right)\right] \tag{21.17}$$

As with the trapezoid and Simpson's rule methods, a more accurate midpoint solution can be obtained by adding more evaluation points. Once again, if you increase the number of points wisely you can reuse previous evaluations of $f(x)$. For example, let's say you start with the two-point midpoint formula shown in Eq. (21.17). If you want to reuse the two function evaluations, the next level in the approximation uses six points (adds four to the two existing points). The six-point midpoint method is defined by the expression in Eq. (21.18).

$$\begin{aligned} y_2 &= \frac{1}{6}\left[f\left(\frac{1}{12}\right) + f\left(\frac{1}{4}\right) + f\left(\frac{5}{12}\right) + f\left(\frac{7}{12}\right) + f\left(\frac{3}{4}\right) + f\left(\frac{11}{12}\right) \right] \\ &= \frac{1}{3} y_1 + \frac{1}{6}\left[f\left(\frac{1}{12}\right) + f\left(\frac{5}{12}\right) + f\left(\frac{7}{12}\right) + f\left(\frac{11}{12}\right) \right] \end{aligned} \tag{21.18}$$

To reuse the function evaluations from the six-point level, the next level in the approximation would add 12 points to the existing six for an 18-point midpoint algorithm.

$$\begin{aligned} y_3 &= \frac{1}{18}\left[f\left(\frac{1}{36}\right) + f\left(\frac{3}{36}\right) + f\left(\frac{5}{36}\right) + \ldots + f\left(\frac{33}{36}\right) + f\left(\frac{35}{36}\right) \right] \\ y_3 &= \frac{1}{3} y_2 + \frac{1}{18}\left[f\left(\frac{1}{36}\right) + f\left(\frac{5}{36}\right) + f\left(\frac{7}{36}\right) + f\left(\frac{11}{36}\right) + f\left(\frac{13}{36}\right) \right. \\ &\quad \left. + \ldots + f\left(\frac{29}{36}\right) + f\left(\frac{31}{36}\right) + f\left(\frac{35}{36}\right) \right] \end{aligned} \tag{21.19}$$

Once again, we can see a pattern developing. Each level in the midpoint approximation adds three times the number of points that were added at the previous level. The value at each new level of approximation is equal to one-third the value at the previous approximation added to the contributions of the additional points added at the current level.

The midpoint method can evaluate an integral with a singularity at one or both of the integration limits, but what happens if a singularity occurs in the middle of the range of integration. The answer is to split the range of integration into two pieces at the singular point. The singularity now occurs at one of the integration limits and the midpoint methodology can be applied.

We will implement the midpoint technique in a method named `midpoint()` that will be defined in the `Integrator` class. The method will take the same four input arguments as the `trapezoidal()` and `simpsonsRule()` methods. The first argument is a reference to a `Function` object that is used to evaluate the function being integrated. The second and third arguments are the integration limits. The fourth argument is the convergence tolerance.

After declaring some variables, the method computes a two-point midpoint approximation to the integral equation. It then refines the integral value with successive approximations adding points to the integration domain with each approximation. The first iteration adds four points, the second adds 12 points, the third adds 36 points, and so on. If the integration value converges, the method returns the value.

Here is the `midpoint()` method source code.

```
public static double midpoint(Function function,
         double start, double end, double tolerance) {
  double dx = end - start;
  double value, oldValue, sum, temp;
  double scale = 1.0/3.0;
  int numPoints, numPairs;
  int maxIter = 20;

  //  Start with a two-point midpoint
  //  evaluation.

  value = 0.5*dx*(
            function.getValue(start + 0.25*dx) +
            function.getValue(end - 0.25*dx) );

  //  Refine the solution by adding points to the
  //  integration domain. Each iteration adds three
  //  times as many points as the previous iteration.
  //  When the solution converges, return the result.

  for(int i=0; i<maxIter; ++i) {

    numPoints = 4*(int)Math.pow(3,i);
    numPairs = (numPoints - 2)/2;
    temp = 1.0/(3.0*numPoints);

    //  Add the two endpoint values to the sum

    sum = function.getValue(start + temp*dx) +
             function.getValue(end - temp*dx);

    //  Add in each pair to the sum

    for(int j=0; j<numPairs; ++j) {
      sum = sum +
        function.getValue(start + (6*j + 5)*temp*dx) +
        function.getValue(start + (6*j + 7)*temp*dx);
    }

    oldValue = value;
    value = value/3.0 +
            0.5*Math.pow(scale,i+1)*dx*sum;

    if ( Math.abs(value-oldValue) <=
         tolerance*Math.abs(oldValue) ) {
       return value;
    }
  }
```

```
    //  Solution did not converge

    return 12345.6;
  }
```

Let's test the `midpoint()` method by applying it to solve two integral equations. Midpoint methods can also be applied to proper integrals, so the first equation to be solved will be the logarithmic integral in Eq. (21.10). The `midpoint()` method will then be applied to the following improper integral.

$$y = \int_2^3 \frac{dx}{\sqrt{x-2}} \tag{21.20}$$

The integral shown in Eq. (21.20) has a singularity at $x = 2$, but is convergent and has a value of $y = 2$. To represent this function, we will define another `Function` subclass called the `SqrtFunction2` class. Its code listing is—

```
package TechJava.MathLib;

//  This class represents the function
//  y = 1/sqrt(x-2)

public class SqrtFunction2 extends Function
{
   public double getValue(double u) {
       return 1.0/Math.sqrt(u-2.0);
   }
}
```

Next we'll create a driver program similar to the ones developed in the previous sections. The code will simply create the proper `Function` subclass object, call the `midpoint()` method, and display the result. The class is called `TestMidpoint` and its source code is—

```
import TechJava.MathLib.*;

public class TestMidpoint
{
  public static void main(String args[]) {

    double value;

    //  Integrate the function y = x*ln(x) from
    //  x=2 to x=4 with an error tolerance of 1.0e-4

    LogFunction function = new LogFunction();
    value = Integrator.midpoint(
                           function,2.0,4.0,1.0e-4);
    System.out.println("value = " + value);
    //  Integrate the function y = 1/sqrt(x-2) from
    //  x=2 to x=3 with an error tolerance of 1.0e-4

    SqrtFunction2 function2 = new SqrtFunction2();
    value = Integrator.midpoint(
                           function2,2.0,3.0,1.0e-4);
    System.out.println("value = " + value);
  }
}
```

Output—

```
value = 6.7040209108022335
value = 1.9998044225262488
```

The `midpoint()` method successfully evaluated the integrals to four-decimal place precision. Improper integrals are tougher to compute and generally take significantly more iterations to achieve convergence. In this example, three iterations were required for the proper logarithmic integral but 14 were necessary to converge the improper integral computation.

Gaussian Quadrature Methods

The three techniques to solve integral equations that we have implemented so far in this chapter divide the integration domain into constant-sized increments of Δx. This is acceptable for smoothly varying functions but may be inefficient or even inaccurate for functions with widely varying slopes. In this case, you would want to use a variable step size with small Δx increments in regions of high gradients and larger Δx increments in smoothly varying regions.

The trapezoidal, Simpson's rule, and midpoint methods are really specialized cases of a more general class of integration techniques called *Gaussian quadrature formulas.* Gaussian quadrature formulas assume that an integral equation can be approximated by a summation of weights multiplied by function evaluations at various x locations along the integration domain, as shown in Eq. (21.21).

$$y = \int_a^b W(x)f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{21.21}$$

The x locations at which the function is evaluated are called the *abscissas*. The coefficients in front of the function evaluations are called the *weights*. The determination of the weights depends on the choice for the $W(x)$ function. There are a number of classical forms for $W(x)$, each designed to be used for a certain type of integral equation. In this section, we will work with a commonly used form called the *Gauss-Legendre formula,* which assumes that $W(x) = 1$.

The implementation of the Gauss-Legendre formula requires two methods, one to compute the weights and abscissas and a second to integrate a specified function. You could alternately use a hard-coded array of weight and abscissa data, but evaluating this data directly gives you more flexibility in choosing the number of points to be considered in the integration domain.

Without going into all of the details, the `computeGaussWeights()` method uses Newton's method to determine the roots of the orthogonal polynomial equation associated with the Gauss-Legendre formula. The weights and abscissas are computed from the polynomial roots and are normalized for an integration ranging from $x = 0$ to $x = 1$. The weights and abscissas can be applied to any integration domain by multiplying the resulting integral value by $b - a$, where a and b are the limits of integration.

The `computeGaussWeights()` method source code follows.

```
public static void computeGaussWeights(
             double x[], double w[], int numPoints) {
  int i,j,m;
  double z, zOld, root, p3, p2, p1;
  double EPS = 1.0e-10;

  //  The weights and abscissa values are normalized
  //  for an integration range between 0 and 1. The
  //  values are symmetric about x=0.5 so you only
  //  have to compute half of them.

  m = (numPoints + 1)/2;

  //  Compute the Legendre polynomial, p1, at point
  //  z using Newton's method.

  for(i=0; i<m; ++i) {
    z = Math.cos(Math.PI*(i+0.75)/(numPoints+0.5));
    do {
      p1 = 1.0;
      p2 = 0.0;
      for(j=0; j<numPoints; ++j) {
        p3 = p2;
        p2 = p1;
        p1 = ((2.0*(j+1) - 1.0)*z*p2 - j*p3)/
             (j+1.0);
      }
      root = numPoints*(z*p1 - p2)/(z*z - 1.0);
      zOld = z;
      z = zOld - p1/root;
    } while ( Math.abs(z-zOld) > EPS);
```

```
      //  Compute the weights and abscissas. The values
      //  are symmetric about the point z=0.5

      x[i] = 0.5*(1.0 - z);
      x[numPoints-1-i] = 0.5*(1.0 + z);
      w[i] = 1.0/((1.0- z*z)*root*root);
      w[numPoints-1-i] = w[i];
    }
  }
```

Once we have the weight and abscissa data, the rest of the implementation is simple. The `gaussLegendre()` method integrates a function using the Gauss-Legendre weighting coefficients and abscissa data. The method takes the same four input arguments as the `trapezoidal()`, `simpsonsRule()`, and `midpoint()` methods.

After declaring some variables, the method enters an iteration loop. Starting with 10 points in the integration domain, the `computeGaussWeights()` method is called to obtain the 10-point Gauss-Legendre weights and abscissas. The integral value is then obtained by summing up the weights and function evaluations at the abscissas. The number of abscissa points is then doubled, the process is repeated, and the current solution is compared against the previous solution. If the two results are within the convergence tolerance, the value is returned. Otherwise, the number of points is again doubled and the process repeats itself.

The `gaussLegendre()` method source code is—

```
  public static double gaussLegendre(Function function,
       double start, double end, double tolerance) {
    double dx = end - start;
    double value = 1.0e+10;
    double oldValue;
    int numPoints = 10;
    int maxPoints = 100000;
    double x[] = new double[maxPoints];
    double w[] = new double[maxPoints];

    //  Compute the integral value adding points
    //  until the solution converges or maxPoints
    //  is reached.

    while(true) {

      //  Calculate the Gauss-Legendre weights and
      //  abscissas

      computeGaussWeights(x, w, numPoints);

      //  Compute the integral value by summing up the
      //  weights times the function value at each
      //  abscissa
      oldValue = value;

      value = 0.0;
      for(int i=0; i<numPoints; ++i) {
       value += w[i]*function.getValue(start + x[i]*dx);
      }
      value *= dx;

      //  If the solution is converged, return the
      //  value. Otherwise, double the points in the
      //  integration domain and try again.

      if ( Math.abs(value-oldValue) <=
           tolerance*Math.abs(oldValue) ) {
        return value;
      }

      numPoints *= 2;
      if (numPoints >= maxPoints) break;
    }

    //  Solution did not converge

    return 12345.6;
  }
```

The Gaussian quadrature technique can be applied to both proper and improper integrals so we use the `gaussLegendre()` method to solve Eq. (21.10) and Eq. (21.20). The driver class is called `TestGauss` and its source code is:

```
import TechJava.MathLib.*;

public class TestGauss
{
  public static void main(String args[]) {

    double value;

    //  Integrate the function y = x*ln(x) from
    //  x=2 to x=4 using 10 points in the integration

    LogFunction function = new LogFunction();
    value =
        Integrator.gaussLegendre(function,2.0,4.0,1.0e-4);
    System.out.println("value = " + value);

    //  Integrate the function y = 1/sqrt(x-2) from
    //  x=2 to x=3 using 10 points in the integration

    SqrtFunction2 function2 = new SqrtFunction2();
    value =
       Integrator.gaussLegendre(function2,2.0,3.0,1.0e-4);
    System.out.println("value = " + value);
  }
}
```

Output—

```
value = 6.70406052759485
value = 1.9998289705958587
```

The `gaussLegendre()` method successfully reproduced the exact solutions to within four decimal places for both the proper and improper integrals. The logarithmic integral converged on the first iteration. The improper integral took longer—10 iterations—but this was somewhat more efficient than the midpoint method that required 14 iterations to converge.

General Integral Types

The integral equations that we have been working with in this chapter are part of a larger class of equations known as Fredholm integrals. There are Fredholm integrals of the first and second kind that take the general form of Eq. (21.22) and Eq. (21.23).

$$g(t) = \int_a^b K(t,s)f(s)ds \tag{21.22}$$

$$f(t) = \lambda \int_a^b K(t,s)f(s)ds + g(t) \tag{21.23}$$

The quantity $K(t,s)$ is known as the kernel. The equations used in the examples in this chapter have been Fredholm integrals of the first kind with $K(t,s) = 1$. Another important type of integral is a Volterra integral that is similar to a Fredholm integral of the first kind except the upper integration limit is t. We won't implement methods to solve general Fredholm or Volterra integrals, but they are typically solved using Gaussian quadrature methods described in the previous section.

Example: Thin Airfoil Theory

As a practical application of the tools we developed in this chapter, let's use the `simpsonsRule()` method to solve for the lift and moment coefficients of an airfoil using thin airfoil theory. Once again, when reading through this

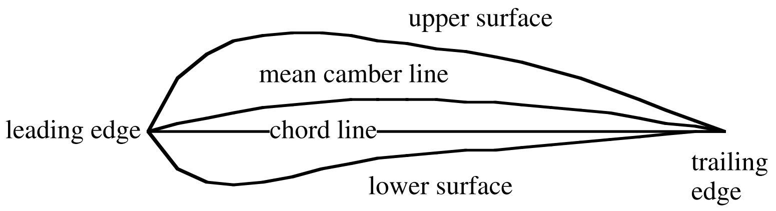

Figure 21.2 Generic airfoil configuration

section concentrate on the process. Even if you never need to compute the aerodynamic characteristics of a thin airfoil, you can apply the same solution process to your own integral problems.

Before we discuss what thin airfoil theory is, let's define some terminology about an airfoil. Consider the airfoil shown in Figure 21.2. A straight line drawn from the wing leading edge to trailing edge is called the *chord line*. The length of the chord line is the *chord* of the wing. The *mean camber line* is the line from the leading to trailing edge such that there is an equal airfoil thickness above and below the line. The difference between the mean camber and chord lines is the *camber* of the wing. A symmetrical airfoil would be one that has zero camber. The angle that an airfoil's chord line makes with the airfoil's direction of travel is called the *angle of attack*.

Now let's move on to thin airfoil theory. The German aerodynamicist Ludwig Prandtl developed this theory in the early 1900s. It starts by modeling an airfoil as a thin plate that follows the mean camber line. Because you can't have air flow through the solid surface of the airfoil, the normal velocity at the airfoil surface is balanced by the induced normal velocity of a sheet of vortices distributed along the airfoil. This assumption results in the following integral equation.

$$\frac{1}{2\pi U_\infty}\int_0^c \frac{\gamma(x')}{x - x'}\,dx' = \alpha - \frac{\partial z}{\partial x} \qquad (21.24)$$

In Eq. (21.24), U_∞ is the freestream velocity, c is the chord length, α is the angle of attack, and $z = z(x)$ is the equation that describes the mean camber line. The quantity γ is the unknown vorticity distribution function that we need to compute. It turns out that the vorticity distribution can be approximated using a Fourier series.

$$\gamma(\theta) = 2U_\infty\left[A_0\cot\left(\frac{\theta}{2}\right) + \sum_{n=1}^{\infty} A_n\sin(n\theta)\right] \qquad (21.25)$$

Eq. (21.25) uses a change of variables from x to θ with the relation in Eq. (21.26).

$$x = \frac{c}{2}(1 - \cos(\theta)) \tag{21.26}$$

The θ quantity varies from 0 to π. After performing some trigonometric manipulations, we arrive at the expressions in Eq. (21.27) for A_0 and A_n.

$$A_0 = \alpha - \frac{1}{\pi}\int_{\theta=0}^{\pi} \frac{\partial z}{\partial x}\, d\theta \qquad A_n = \frac{2}{\pi}\int_{\theta=0}^{\pi} \frac{\partial z}{\partial x}\cos(n\theta) d\theta \tag{21.27}$$

Once the A coefficients are known, the lift and leading edge moment coefficients can be computed from the relations shown in Eq. (21.28) and Eq. (21.29). The terms ρ_∞ and U_∞ are the freestream density and velocity, L is the lifting force on the airfoil, and M is the moment about the leading edge.

$$C_l = \frac{L}{\frac{1}{2}\rho_\infty U_\infty^2} = \pi(2A_0 + A_1) \tag{21.28}$$

$$C_{M_{LE}} = \frac{M}{\frac{1}{2}\rho_\infty U_\infty^2\, l} = -\frac{\pi}{2}\left(A_0 + A_1 - \frac{A_2}{2}\right) \tag{21.29}$$

We will now write a program to compute the A coefficients for an arbitrarily cambered wing using Simpson's rule integration method we developed earlier in this chapter. Once the A coefficients have been found, the program will calculate the lift and leading edge moment coefficients for the airfoil.

We will apply our program to compute the aerodynamic coefficients for a NACA 2412 airfoil traveling at an angle of attack of 5 degrees. The NACA 2412 is an airfoil with a maximum thickness of 0.12*chord and has a 2 percent maximum camber at 40 percent chord. The equation for its camber line is defined in two pieces.

$$\begin{aligned} z(x) &= 0.125(0.8x - x^2) & \frac{x}{c} &\le 0.4 \\ z(x) &= 0.0555(0.2c + 0.8x - x^2) & \frac{x}{c} &> 0.4 \end{aligned} \tag{21.30}$$

Taking the derivatives of Eq. (21.30) and converting x to θ, we obtain expressions (Eq. 21.31) for $\partial z/\partial x$ that we need to compute the A coefficients.

$$\begin{aligned} \frac{\partial z}{\partial x} &= 0125\cos(\theta) - 0.025 \qquad \frac{x}{c} \le 0.4 \\ \frac{\partial z}{\partial x} &= 0.0555\cos(\theta) - 0.0111 \qquad \frac{x}{c} > 0.4 \end{aligned} \tag{21.31}$$

The first code to write is a `Function` subclass that will return the function $\partial z/\partial x$ cos(nθ). This function is needed to evaluate the A integrals. The class is named `CamberFunction` and its source code is shown next. The power to be applied to the cosine term is passed to the `CamberFunction` constructor.

```
package TechJava.MathLib;

//  This class represents the camber
//  function for a NACA 2412 airfoil

public class CamberFunction extends Function
{
  private int n;

  public CamberFunction(int n) {
    this.n = n;
  }

  //  getValue() returns the height of the camber
  //  line for a given theta where
  //  cos(theta) = 1.0 - 2x/c

  public double getValue(double theta) {
    if (theta < 1.36944) {
      return (0.125*Math.cos(theta) - 0.025)*
              Math.cos (theta*n);
    } else {
      return (0.0555*Math.cos(theta) - 0.0111)*
              Math.cos (theta*n);
    }
  }
}
```

Now we can write our thin airfoil analysis code, which we will call `ThinAirfoil.java`. In this case, the program computes the aerodynamic coefficients for a 5 degree angle of attack. With a little extra effort, you could modify the code to take the angle of attack as an input argument. After declaring some variables, the code calculates the A_0, A_1, and A_2 coefficients. It does this by creating a `CamberFunction` object and then integrating the associated function using the `simpsonsRule()` method. Once the A coefficients are in hand, the lift and moment coefficients are computed and the results are printed to the console.

Here is the `ThinAirFoil` class source code—

```
import TechJava.MathLib.*;

public class ThinAirfoil
{
  public static void main(String args[]) {

    double alpha, Cl, Cm;
    double A[] = new double[3];
    CamberFunction naca2412;

    //  Compute the conditions for a 5 degree
    //  angle of attack.

    alpha = 5.0*Math.PI/180.0;

    //  Calculate the A0, A1, and A2 coefficients

    double temp = 1.0/Math.PI;

    naca2412 = new CamberFunction(0);
    A[0] = alpha - temp*Integrator.simpsonsRule(
                      naca2412,0.0,Math.PI,1.0e-4);

    for(int i=1; i<3; ++i) {
      naca2412 = new CamberFunction(i);
      A[i] = 2.0*temp*Integrator.simpsonsRule(
                   naca2412,0.0,Math.PI,1.0e-4);
    }

    //  Compute Cl and Cm(leading edge)

    Cl = Math.PI*(2.0*A[0] + A[1]);
    Cm = -0.5*Math.PI*(A[0] + A[1] - 0.5*A[2]);

    //  Print out the results

    for(int i=0; i<3; ++i) {
      System.out.println("A["+i+"] = "+A[i]);
    }
    System.out.println("\nCl = "+Cl);
    System.out.println("Cm = "+Cm);
  }
}
```

Output—

```
A[0] = 0.08274980955387196
A[1] = 0.08146092605250739
A[2] = 0.01387204625538765

Cl = 0.7758494344019757
Cm = -0.02470465406592427
```

The compute lift coefficient value is within 3.5 percent of the experimentally obtained value[1] of 0.75.

References

1. University of Tennessee Aerodynamic Database, *www.engr.ut.edu/~rbond/airfoil.html.*