5
Integration

5.1
Problem: Integrating a Spectrum

Problem: An experiment has measured $dN(t)/dt$, the number of particles per unit time entering a counter. Your **problem** is to integrate this spectrum to obtain the number of particles $N(1)$ that entered the counter in the first second:

$$N(1) = \int_0^1 \frac{dN(t)}{dt} dt \tag{5.1}$$

Although the integrand we will give you later can be integrated analytically, we wish to evaluate (5.1) for an arbitrary integrand.

5.2
Quadrature as Box Counting (Math)

The integration of a function may require some cleverness to do analytically, but it is relatively straightforward on a computer. A traditional way to do numerical integration by hand is to take a piece of graph paper and count the number of boxes or *quadrilaterals* lying below a curve of the integrand. For this reason numerical integration is also called *numerical quadrature*, even when it becomes more sophisticated than simple box counting. The Riemann definition of an integral is the limit of the sum over boxes as the width h of the box approaches zero:

$$\int_a^b f(x)dx = \lim_{h\to 0} \left[h \sum_{i=1}^{(b-a)/h} f(x_i) \right] \tag{5.2}$$

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu

ISBN: 978-3-527-40626-5

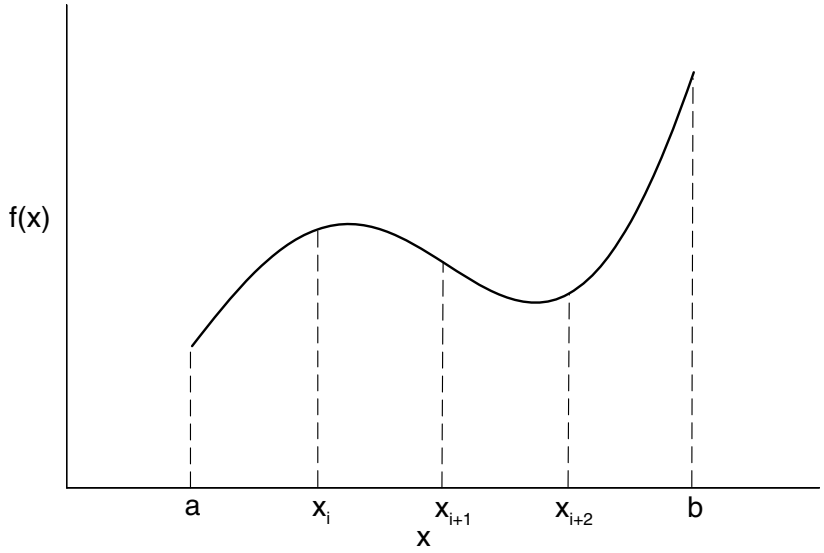

Fig. 5.1 The integral $\int_a^b f(x)\,dx$ is the area under the graph of $f(x)$ from a to b. Here we break up the area into four regions of equal widths.

The numerical integral of a function $f(x)$ is approximated as the equivalent of a finite sum over boxes of height $f(x)$ and width w_i:

$$\int_a^b f(x)dx \;\approx\; \sum_{i=1}^{N} f(x_i)w_i \tag{5.3}$$

which is similar to the Riemann definition (5.2), except that there is no limit to infinitesimal box size. Equation (5.3) is the standard form for all integration algorithms; the function $f(x)$ is evaluated at N points in the interval $[a, b]$, and the function values $f_i \equiv f(x_i)$ are summed with each term in the sum weighted by w_i. While, in general, the sum in (5.3) will give the exact integral only when $N \to \infty$, for polynomials it may be exact for finite N. The different integration algorithms amount to different ways of choosing the points and weights. Generally, the precision increases as N gets larger, with the round-off error eventually limiting the increase. Because the "best" approximation depends on the specific behavior of $f(x)$, there is no universally best approximation. In fact, some of the automated integration schemes found in subroutine libraries will switch from one method to another until they find one that works well.

In general, you should not attempt a numerical integration of an integrand that contains a singularity without first removing the singularity by hand.[1]

[1] In Chap. 30 we show how to remove such a singularity even when the integrand is unknown.

You may be able to do this very simply by breaking the interval down into several subintervals, so the singularity is at an endpoint where a Gauss point never falls, or by a change of variable:

$$\int_{-1}^{1} |x| f(x)dx = \int_{-1}^{0} f(-x)dx + \int_{0}^{1} f(x)dx \tag{5.4}$$

$$\int_{0}^{1} x^{1/3}dx = \int_{0}^{1} 3y^3 dy \qquad (y = x^{1/3}) \tag{5.5}$$

$$\int_{0}^{1} \frac{f(x)dx}{\sqrt{1-x^2}} = 2\int_{0}^{1} \frac{f(1-y^2)dy}{\sqrt{2-y^2}} \qquad (y^2 = 1-x). \tag{5.6}$$

Likewise, if your integrand has a very slow variation in some region, you can speed up the integration by changing to a variable that compresses that region and places few points there. Conversely, if your integrand has a very rapid variation in some region, you may want to change to variables that expand that region to ensure that no oscillations are missed.

Tab. 5.1 Elementary weights for uniform-step integration rules.

Name	Degree	Elementary weights
Trapezoid	1	$(1,1)\frac{h}{2}$
Simpson's	2	$(1,4,1)\frac{h}{3}$
$\frac{3}{8}$	3	$(1,3,3,1)\frac{3}{8}h$
Milne	4	$(14,64,24,64,14)\frac{h}{45}$

5.3 Algorithm: Trapezoid Rule

The trapezoid and Simpson integration rules use values of $f(x)$ at evenly spaced values of x. They use N points x_i $(i = 1, N)$, evenly spaced at a distance h apart throughout the integration region $[a, b]$ and *include the endpoints*. This means that there are $N-1$ intervals of length h:

$$h = \frac{b-a}{N-1} \qquad x_i = a + (i-1)h \quad i = 1, N \tag{5.7}$$

Notice that we start our counting at $i = 1$, and that Simpson's rule requires an *odd* number of points N.

The trapezoid rule takes the integration interval i and constructs a trapezoid of width h in it (Fig. 5.2). This approximates $f(x)$ by a straight line in that interval i, and uses the average height $(f_i + f_{i+1})/2$ as the value for f. The

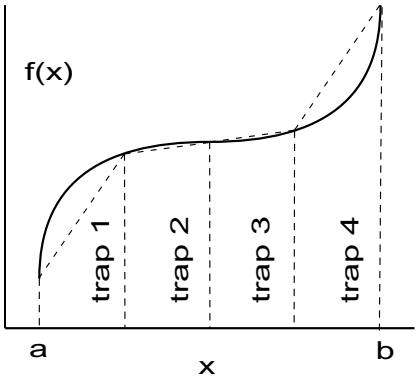

Fig. 5.2 Straight-line sections used for the trapezoid rule.

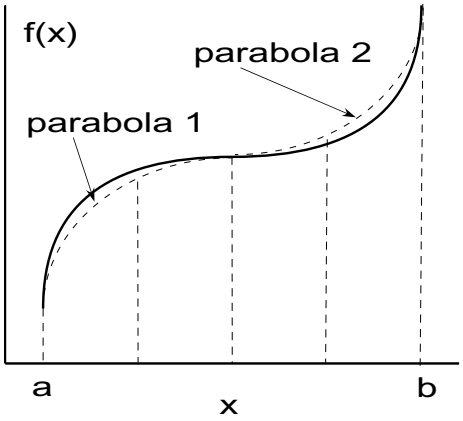

Fig. 5.3 Two parabolas used in Simpson's rule.

area of a single trapezoid is in this way

$$\int_{x_i}^{x_i+h} f(x)dx \simeq \frac{h(f_i + f_{i+1})}{2} = \tfrac{1}{2}hf_i + \tfrac{1}{2}hf_{i+1} \tag{5.8}$$

In terms of our standard integration formula (5.3), the "rule" in (5.8) is for $N = 2$ points with weight $w_i \equiv \frac{1}{2}$ (Tab. 5.1).

In order to apply the trapezoid rule to the entire region $[a, b]$, we add the contributions from each subinterval:

$$\int_a^b f(x)dx \;\approx\; \frac{h}{2}f_1 + hf_2 + hf_3 + \cdots hf_{N-1} + \frac{h}{2}f_N \tag{5.9}$$

You will notice that because each internal point gets counted twice, it has a weight of h, whereas the endpoints get counted just once and on that account have weights of only $h/2$. In terms of our standard integration rule (5.32), we

have

$$w_i = \left\{ \frac{h}{2}, h, \ldots, h, \frac{h}{2} \right\} \tag{5.10}$$

In Listing 5.1 we provide a simple implementation of the trapezoid rule.

Listing 5.1: The program `Trap.java` integrates the function $f(t) = t^2$ via the trapezoid rule. Note how the step size h depends on the interval and how the endpoint weights are set.

```
// Trap.java trapezoid-rule integration of parabola

public class Trap  {
  public static final double A = 0., B =3.;      // Constant endpoints
  public static final int N = 100;           // N points (not intervals)

  public static void main(String[] args) {      // Main does summation

    double sum, h, t, w;
    int i;

    h = (B - A)/(N - 1);                                  // Initialization
    sum = 0.;
                                                          // Trap rule
    for ( i=1 ;  i <= N;  i=i + 1)  {
      t = A + (i-1) * h;
      if ( i==1 || i==N ) w = h/2.; else w = h;          // End wt=h/2
      sum = sum + w  * t * t;
    }
    System.out.println(sum);
  }
}

//   OUTPUT                  9.000459136822773
```

5.4 Algorithm: Simpson's Rule

For each interval, Simpson's rule approximates the integrand $f(x)$ by a parabola (Fig. 5.3):

$$f(x) \approx \alpha x^2 + \beta x + \gamma \tag{5.11}$$

with the intervals still equally spaced. The area of each section is then the integral of this parabola

$$\int_{x_i}^{x_i+h} (\alpha x^2 + \beta x + \gamma) dx = \frac{\alpha x^3}{3} + \frac{\beta x^2}{2} + \gamma x \Big|_{x_i}^{x_i+h} \tag{5.12}$$

This is equivalent to integrating the Taylor series up to the quadratic term. In order to relate the parameters α, β, and γ to the function, we consider an

interval from -1 to $+1$, in which case

$$\int_{-1}^{1}(\alpha x^2+\beta x+\gamma)dx=\frac{2\alpha}{3}+2\gamma \tag{5.13}$$

But we notice that

$$\begin{array}{cccc} f(-1)=\alpha-\beta+\gamma & & f(0)=\gamma & f(1)=\alpha+\beta+\gamma \\ \Rightarrow \quad \alpha=\frac{f(1)+f(-1)}{2}-f(0) & \beta=\frac{f(1)-f(-1)}{2} & & \gamma=f(0) \end{array} \tag{5.14}$$

In this way we can express the integral as the weighted sum over the values of the function at three points:

$$\int_{-1}^{1}(\alpha x^2+\beta x+\gamma)dx=\frac{f(-1)}{3}+\frac{4f(0)}{3}+\frac{f(1)}{3} \tag{5.15}$$

Because three values of the function are needed, we generalize this result to our problem by evaluating the integral over two adjacent intervals, in which case we evaluate the function at the two endpoints and the middle (Tab. 5.1):

$$\begin{aligned}\int_{x_i-h}^{x_i+h}f(x)dx&=\int_{x_i}^{x_i+h}f(x)dx+\int_{x_i-h}^{x_i}f(x)dx\\ &\simeq\frac{h}{3}f_{i-1}+\frac{4h}{3}f_i+\frac{h}{3}f_{i+1}\end{aligned} \tag{5.16}$$

Simpson's rule requires the elementary integration to be over *pairs* of intervals, which in turn requires that the total number of intervals be even or the number of points N be odd. In order to apply Simpson's rule to the entire interval, we add up the contributions from each pair of subintervals, counting all but the first and last endpoints twice:

$$\int_a^b f(x)dx\;\approx\;\frac{h}{3}f_1+\frac{4h}{3}f_2+\frac{2h}{3}f_3+\frac{4h}{3}f_4+\cdots+\frac{4h}{3}f_{N-1}+\frac{h}{3}f_N \tag{5.17}$$

In terms of our standard integration rule (5.3), we have

$$w_i=\left\{\frac{h}{3},\frac{4h}{3},\frac{2h}{3},\frac{4h}{3},\ldots,\frac{4h}{3}\,\frac{h}{3}\right\} \tag{5.18}$$

The sum of these weights provides a useful check on your integration:

$$\sum_{i=1}^{N}w_i=(N-1)h \tag{5.19}$$

Remember that the number of points N must be odd for Simpson's rule.

5.5 Integration Error (Analytic Assessment)

In general, you want to choose an integration rule that gives an accurate answer using the least number of integration points. We obtain a feel for the absolute *approximation* or *algorithmic error* E and the relative error ϵ, by expanding $f(x)$ in a Taylor series around the midpoint of the integration interval. We then multiply that error by the number of intervals N to estimate the error for the entire region $[a, b]$. For the trapezoid and Simpson's rules this yields

$$E_t = O\left(\frac{[b-a]^3}{N^2}\right) f^{(2)} \qquad E_s = O\left(\frac{[b-a]^5}{N^4}\right) f^{(4)} \qquad \epsilon_{t,s} = \frac{E_{t,s}}{f} \tag{5.20}$$

We see that the third derivative term in Simpson's rule cancels (much like the central difference method in differentiation). Equations (5.20) are illuminating by showing how increasing the sophistication of an integration rule leads to an error that falls off with a higher inverse power of N, yet that is also proportional to higher derivatives of f. Consequently, for small intervals and $f(x)$ functions with well-behaved high derivatives, Simpson's rule should converge more rapidly than the trapezoid rule.

To be more specific, we assume that after N steps the *relative* roundoff error is random and of the form

$$\epsilon_{\text{ro}} \approx \sqrt{N}\epsilon_m \tag{5.21}$$

where ϵ_m is the machine precision, $\epsilon \sim 10^{-7}$ for single precision and $\epsilon \sim 10^{-15}$ for double precision. Because most scientific computations are done with doubles, we will assume double precision. We want to determine an N that minimizes the total error, that is, the sum of the approximation and roundoff errors,

$$\epsilon_{\text{tot}} = \epsilon_{\text{ro}} + \epsilon_{\text{approx}} \tag{5.22}$$

This occurs, approximately, when the two errors are of equal magnitude, which we approximate even further by assuming that the two errors are equal:

$$\epsilon_{\text{ro}} = \epsilon_{\text{approx}} = \frac{E_{\text{trap,simp}}}{f} \tag{5.23}$$

To continue the search for optimum N for a general function f, we set the scale of function size and the lengths by assuming

$$\frac{f^{(n)}}{f} \approx 1 \qquad b - a = 1 \qquad \Rightarrow \qquad h = \frac{1}{N} \tag{5.24}$$

The estimate (5.23), when applied to the **trapezoid rule**, yields

$$\sqrt{N}\epsilon_m \approx \frac{f^{(2)}(b-a)^3}{fN^2} = \frac{1}{N^2} \tag{5.25}$$

$$\Rightarrow \quad N \approx \frac{1}{(\epsilon_m)^{2/5}} = (1/10^{-15})^{2/5} = 10^6, \tag{5.26}$$

$$\Rightarrow \epsilon_{\text{ro}} \approx \sqrt{N}\epsilon_m = 10^{-12} \tag{5.27}$$

The estimate (5.23), when applied to **Simpson's rule** yields

$$\sqrt{N}\epsilon_m = \frac{f^{(4)}(b-a)^5}{fN^4} = \frac{1}{N^4}, \tag{5.28}$$

$$\Rightarrow N = \frac{1}{(\epsilon_m)^{2/9}} = (1/10^{-15})^{2/9} = 2154, \tag{5.29}$$

$$\Rightarrow \epsilon_{\text{ro}} \approx \sqrt{N}\epsilon_m = 5 \times 10^{-14} \tag{5.30}$$

These results are illuminating in that they show how

- Simpson's rule is an improvement over the trapezoid rule;
- it is possible to obtain an error close to machine precision with Simpson's rule (and with other higher order integration algorithms);
- obtaining the best numerical approximation to an integral is not obtained by letting $N \to \infty$, but with a relatively small $N \leq 1000$.

Tab. 5.2 Types of Gaussian integration rules.

Integral	Name	Integral	Name
$\int_{-1}^{1} f(y)dy$	Gauss	$\int_{-1}^{1} \frac{F(y)}{\sqrt{1-y^2}}dy$	Gauss–Chebyshev
$\int_{-\infty}^{\infty} e^{-y^2}F(y)dy$	Gauss–Hermite	$\int_{0}^{\infty} e^{-y}F(y)dy$	Gauss–Laguerre
$\int_{0}^{\infty} \frac{e^{-y}}{\sqrt{y}}F(y)dy$	Associated Gauss–Laguerre		

5.6 Algorithm: Gaussian Quadrature

It is often useful to rewrite the basic integration formula (5.3) such that we separate a weighting function $W(x)$ from the integrand:

$$\int_a^b f(x)dx \equiv \int_a^b W(x)g(x)dx \approx \sum_{i=1}^{N} w_i g(x_i) \tag{5.31}$$

In the Gaussian quadrature approach to integration, the N points and weights are chosen to make the approximation error actually vanish for $g(x)$, a $2N-1$ degree polynomial. To obtain this incredible optimization, the points x_i end up having a very specific distribution over $[a, b]$.

In general, if $g(x)$ is smooth, or can be made smooth by factoring out some $W(x)$, Gaussian algorithms produce higher accuracy than lower order ones, or conversely, the same accuracy with a fewer number of points. If the function being integrated is not smooth (for instance, if it contains noise), then using a higher order method such as Gaussian quadrature may well lead to lower accuracy. Sometimes the function may not be smooth because it has different behaviors in different regions. In these cases it makes sense to integrate each region separately and then add the answers together. In fact, some of the "smart" integration subroutines will decide for themselves how many intervals to use and what rule to use in each interval.

All the rules indicated in Tab. 5.2 are Gaussian with the general form (5.31). We can see that in one case the weighting function is an exponential, in another a Gaussian, and in several an integrable singularity. In contrast to the equally spaced rules, there is never an integration point at the extremes of the intervals, yet all of the points and weights change as the number of points N changes.

Although we will leave it to the references on numerical methods for the derivation of the Gauss points and weights, we note here that for ordinary Gaussian (Gauss–Legendre) integration, the points y_i turn out to be the N zeros of the Legendre polynomials, with the weights related to the derivatives, $P_N(y_i) = 0$, and $w_i = 2/([(1-y_i^2)[P_N'(y_i)]^2]$. Subroutines to generate these points and weights are standard in mathematical function libraries, are found in tables suc h as those in [7], or can be computed. The *gauss* subroutines we provide on the CD also scale the points to a specified region. As a check that your points are correct, you may want to compare them to the four-point set in Tab. 5.3.

Tab. 5.3 Points and weights for four-point Gaussian quadrature

$\pm y_i$	w_i
0.33998 10435 84856	0.65214 51548 62546
0.86113 63115 94053	0.34785 48451 37454

5.6.1 Mapping Integration Points

Our standard convention (5.3) for the general interval $[a, b]$ is

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} f(x_i)w_i \tag{5.32}$$

With Gaussian points and weights, the y interval $-1 < y_i \leq 1$ must be *mapped* onto the x interval $a \leq x \leq b$. Here are some mappings we have found useful in our work. In all cases (y_i, w_i') are the elementary Gaussian points and weights for the interval $[-1, 1]$, and we want to scale to x with various ranges:

1. $[-1, 1] \to [a, b]$ uniformly, $\frac{a+b}{2}$ = **midpoint**:

$$x_i = \frac{b+a}{2} + \frac{b-a}{2} y_i \qquad w_i = \frac{b-a}{2} w_i' \tag{5.33}$$

$$\Rightarrow \qquad \int_a^b f(x)dx = \frac{b-a}{2} \int_{-1}^{1} f[x(y)]dy \tag{5.34}$$

2. $[0 \to \infty]$, a = **midpoint**:

$$x_i = a\frac{1+y_i}{1-y_i} \qquad w_i = \frac{2a}{(1-y_i)^2} w_i' \tag{5.35}$$

3. $[-\infty \to \infty]$**, scale set by** a**:**

$$x_i = a\frac{y_i}{1-y_i^2} \qquad w_i = \frac{a(1+y_i^2)}{(1-y_i^2)^2} w_i' \tag{5.36}$$

4. $[b \to \infty]$ $a + 2b =$ **midpoint:**

$$x_i = \frac{a+2b+ay_i}{1-y_i} \qquad w_i = \frac{2(b+a)}{(1-y_i)^2} w_i' \tag{5.37}$$

5. $[0 \to b]$**,** $ab/(b+a) =$ **midpoint:**

$$x_i = \frac{ab(1+y_i)}{b+a-(b-a)y_i} \qquad w_i = \frac{2ab^2}{(b+a-(b-a)y_i)^2} w_i' \tag{5.38}$$

As you can see, even if your integration range extends out to infinity, there will be points at large but not infinite x. As you keep increasing the number of grid points N, the last x_i get larger but always remains finite.

5.6.2 Gauss Implementation

Listing 5.2: `IntegGauss.java` integrates the function $f(x)$ via Gaussian quadrature with points and weights generated by the method `gauss`, which is the same for all applications. The parameter `eps` controls the level of precision desired and should be set by the user, as should the value for `job`, which controls the mapping of the Gauss points to (a,b).

```
// IntegGauss.java: Gauss quadrature, need include gauss method

import java.io.*;                              // Location of PrintWriter

public class IntegGauss  {
  static final double max_in = 1001;                  // Numb intervals
  static final double vmin = 0., vmax = 1.;               // Int ranges
  static final double ME = 2.7182818284590452354E0 ; // Euler's const

 public static void main(String[] argv)
                     throws IOException, FileNotFoundException {
   int i;
   double result;

   PrintWriter t = new PrintWriter(
                   new FileOutputStream("IntegGauss.dat"), true);
   for ( i=3;  i <= max_in;  i  += 2) {
     result = gaussint(i, vmin, vmax);
     t.println("" + i + " " + Math.abs(result-1 + 1/ME));
   }
   System.out.println("Output in IntegGauss.dat");
  }
                                                              // f(x)
  public static double f (double x)       {return (Math.exp(-x));}

  public static double gaussint (int no, double min, double max)  {
    int n;
    double quadra = 0.;
    double w[] = new double[2001], x[] = new double[2001];
    Gauss.gauss (no, 0, min, max, x, w);          // Returns pts & wts
    for ( n=0;  n <  no;  n++ )  {
      quadra += f(x[n])*w[n]; }                  // Calculate integral
    return (quadra);
  }
}
```

Write a double-precision program to integrate an arbitrary function numerically using the trapezoid rule, the Simpson rule, and Gaussian quadrature. For our **problem** we assume exponential decay so that there actually is an analytic answer:

$$\frac{dN(t)}{dt} = e^{-t} \qquad \Rightarrow \qquad N(1) = \int_0^1 e^{-t}dt = 1 - e^{-1} \tag{5.39}$$

In Listing 5.2 we give an example of a program that calls the `gauss` method. Note that the `gauss` method is contained in the `Gauss` class file, which is

a different class file than the one containing `IntGauss`. Consequently, as is Java's conventions, the `gauss` method is called as `Gauss.guass`, where `Gauss` is the name of the class containing `gauss`. In Listing 5.3 we show the class file `Gauss.java` containing a method `gauss.java` that computes the Gaussian points and weights.

Listing 5.3: The program `IntegGauss.java` integrates the function $f(x)$ via Gaussian quadrature. The points and weights are generated in the method `gauss`, which is the same for all applications. However, the parameter `eps`, which controls the level of precision desired, should be set by the user, as should the value for `job`, which controls the mapping of the Gauss points onto arbitrary intervals (they are generated for $-1 \leq x \leq 1$).

```
// gauss.java Gaussian quadrature points and weights
public class Gauss{

  public static void gauss(int npts, int job, double a, double b,
                                                  double x[], double w[]) {
     int m = 0, i = 0, j = 0;
     double t = 0., t1 = 0., pp = 0., p1 = 0., p2 = 0., p3 = 0., xi;
     double eps = 3.E-14;                          // Accuracy: ADJUST!
     m = (npts + 1)/2;
     for ( i=1;  i <= m;  i++ )  {
       t = Math.cos(Math.PI*((double)i-0.25)/((double)npts + 0.5));
       t1 = 1;
       while((Math.abs(t-t1)) >= eps)  {
         p1 = 1. ;  p2 = 0. ;
         for ( j=1;  j <= npts;  j++ )  {
           p3 = p2;   p2 = p1;
           p1=((2.*(double)j-1)*t*p2-((double)j-1.)*p3)/((double)j);
         }
         pp = npts*(t*p1-p2)/(t*t-1.);
         t1 = t;          t = t1 - p1/pp;
       }
       x[i-1] = -t;    x[npts-i] = t;
       w[i-1]    = 2./((1.-t*t)*pp*pp);
       w[npts-i] = w[i-1];
       System.out.println(" x[i-1]"+ x[i-1] +" w " + w[npts-i]);
     }
     if (job==0)  {
       for ( i=0;  i < npts ;  i++ )  {
         x[i] = x[i]*(b-a)/2. + (b + a)/2.;
         w[i] = w[i]*(b-a)/2.;
       }
     }
     if (job==1)  {
       for ( i=0;  i < npts;  i++ )  {
         xi=x[i];
         x[i] = a*b*(1. + xi) / (b + a-(b-a)*xi);
         w[i] = w[i]*2.*a*b*b/((b + a-(b-a)*xi)*(b+a-(b-a)*xi));
       }
     }
     if (job==2) {
       for ( i=0;  i < npts;  i++ )  {
          xi=x[i];
```

```
            x[i] = (b*xi+ b + a + a) / (1.-xi);
            w[i] = w[i]*2.*(a + b)/((1.-xi)*(1.-xi));
          }
        }
        return;
    }
}
```

5.7 Empirical Error Estimate (Assessment)

Compute the relative error $\epsilon = |(\text{numeric-exact})/\text{exact}|$ for the trapezoid rule, Simpson's rule, and Gaussian quadrature. You should observe that

$$\epsilon \simeq CN^{\alpha} \quad \Rightarrow \quad \log \epsilon = \alpha \log N + \text{constant} \tag{5.40}$$

This means that a power-law dependence appears as a straight line on a log–log plot, and that if you use $\log_{10}$, then the ordinate on your log–log plot will be the negative of the number decimal places of precision in your calculation.

1. Present your data in the tabular form,

N	ϵ_T	ϵ_S	ϵ_G
10	$\cdots$	$\cdots$	$\cdots$

   with spaces or tabs separating the fields. Try N values of 2, 10, 20, 40, 80, 160, ... (*Hint*: These are even numbers, which may not be the assumption of every rule.)

2. Make a plot of $\log_{10} \epsilon$ versus $\log_{10} N$ that gives the number of decimal places of precision obtained for each value of N. To illustrate, if $\epsilon \simeq 10^{-7}$, then $\log_{10} \epsilon \simeq -7$.

3. Use your plot or table to estimate the power-law dependence of the error ϵ on the number of points N and to determine the number of decimal places of precision in your calculation. Do this for both the trapezoid and Simpson rules, and for both the algorithmic and the roundoff error regimes. (Note that it may be hard to reach the roundoff error regime for the trapezoid rule because the approximation error is so large.)

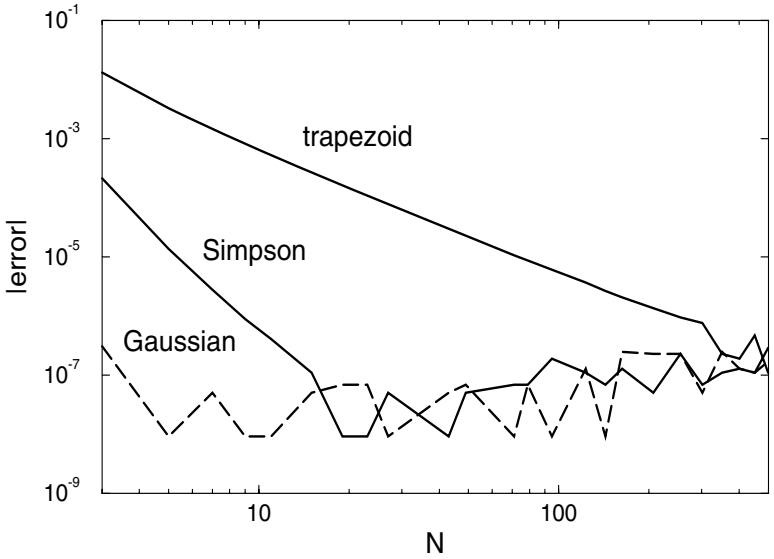

Fig. 5.4 Log–log plot of the error in integration of exponential decay using the trapezoid rule, Simpson's rule, and Gaussian quadrature, versus the number of integration points N. Approximately seven decimal places of precision are attainable with single precision (shown here) and 15 places with double precision.

5.8 Experimentation

Try two integrals for which the answers are less obvious:

$$F_1 = \int_0^{2\pi} \sin(100x)dx, \quad F_2 = \int_0^{2\pi} \sin^x(100x)dx \tag{5.41}$$

Explain why the computer may have trouble with these integrals.

5.9 Higher Order Rules (Algorithm)

As done with numerical differentiation, we can use the known functional dependence of the error on the interval size h to reduce the integration error. For simple rules such as trapezoid and Simpson's, we have the analytic estimates (5.23), while for others you may have to experiment to determine the h dependence. To illustrate, if $A(h)$ and $A(h/2)$ are the values of the integral determined for the intervals h and $h/2$, respectively, we know that the integrals

have expansions with a leading error term proportional to h^2:

$$A(h) \approx \int_a^b f(x)dx + \alpha h^2 + \beta h^4 + \ldots, \tag{5.42}$$

$$A(\tfrac{h}{2}) \approx \int_a^b f(x)dx + \frac{\alpha h^2}{4} + \frac{\beta h^4}{16} + \cdots. \tag{5.43}$$

Consequently, we make the h^2 term vanish by computing the combination

$$\tfrac{4}{3}A(\tfrac{h}{2}) - \tfrac{1}{3}A(h) \approx \int_a^b f(x)dx - \frac{\beta h^4}{4} + \cdots \tag{5.44}$$

Clearly this particular trick (Romberg's extrapolation) works only if the h^2 term dominates the error, and then only if the derivatives of the function are well behaved. An analogous extrapolation can also be made for other algorithms.

In Tab. 5.1 we gave the weights for several equal-interval rules. Whereas the Simpson's rule used two intervals, the 3/8 rule uses three, and the Milne[2] rule four. (These are single-interval rules and must be strung together to obtain a rule *extended* over the entire integration range. This means that the points that end one interval and begin the next get weighted twice.) You can easily determine the number of elementary intervals integrated over, and check whether you and we have written the weights right, by summing the weights for any rule. The sum is the integral of $f(x) = 1$ and must equal h times the number of intervals (which, in turn, equals $b - a$):

$$\sum_{i=1}^{N} w_i = h \times N_{\text{intervals}} = b - a \tag{5.45}$$

[2] There is, not coincidentally, a Milne Computer Center at Oregon State University, although there is no longer a central computer in it.