9. QUADRATURE: INTEGRATION OF FUNCTIONS

9.1 PRELIMINARIES

The term *quadrature* is used to make a distinction between the determination of areas under curves and solving differential equations. Quadrature is the name reserved for the method by which areas are estimated by tiling them by small squares and then counting up the number of squares.

Function integration is used to (e.g., Heath, 2002):

- Construct integral transforms, e.g., Laplace, Fourier, etc.
- Create special functions – gamma, beta, Bessel, Fresnel, etc.
- Generate finite elements for solving partial differential equations.
- Form integral equations which are the basis of variational methods.
- Generate probability distributions.
- Solve problems in mechanics, both classical and quantum.

9.1.1 Riemann Integrals

A *definite* integral of a real-valued function over the interval [a, b] is represented by

$$I = \int_a^b f(x)dx$$

which can be defined by the *asymptotic* limit of a Riemann sum –

$$R_N = \sum_{i=1}^{N} (x_{i+1} - x_i) f(\xi_i),$$

$$\text{where } a < x_1, x_2 < \cdots < x_N < x_{N+1} = b,$$

$$\xi \in [x_i, x_{i+1}], i = 1{:}N,$$

$$\text{if } \lim_{N \to \infty} R_N = I.$$

If a function satisfies these requirements it is termed *Riemann integrable* on the interval [a, b]. A necessary and sufficient condition for the existence of a Riemann integral of a bounded function f is continuity *almost* everywhere (e.g. Krommer & Ueberhuber 1998 [p. 4]). *An integrable function can have singularities.* Thus integrable functions are bounded functions with a finite number of discontinuities or singularities (Heath 2002).

9.1.2 Improper Riemann Integrals (e.g. Krommer & Ueberhuber 1998 [p. 11]).

In certain situations the characterization of an integral can be extended to integrals over unbounded region. Such integrals are defined as the limits of the sequence of *proper* Riemann integrals (e.g. Krommer & Ueberhuber 1998 [p. 10]).

If a function f is Riemann integrable on all bounded subintervals [a, β], then the *improper* integral of f(x) on [a,∞) is defined to be

$$\int_a^{\infty} f(x)dx = \lim_{\beta\to\infty}\int_a^{\beta} f(x)dx$$

provided the limit actually exists! An example of a improper integral is –

$$\lim_{\beta\infty}\int_a^{\beta} e^{-x}dx = \lim_{\beta\to\infty}(1 - e^{-\beta}) = 1.$$

If a function f is Riemann integrable on all bounded subintervals $[\alpha, \beta]$, then the improper integral of f on $(-\infty, \infty)$ is defined to be

$$\int_{-\infty}^{+\infty} f(x)dx \equiv \int_{-\infty}^{\gamma} f(x)dx + \int_{\gamma}^{+\infty} f(x)dx$$

provided the two improper integrals exist. NOTE that this definition is independent of the value of γ! An example is –

$$\int_{-\infty}^{+\infty} \frac{1}{1+x^2}dx = \pi,$$

because $f(x) = (1+x^2)^{-1}$ is an even function and

$$\lim_{\beta\to\infty}\left[\int_0^{\beta} \frac{1}{1+x^2}dx\right] = \lim_{\beta\to\infty} \tan^{-1}\beta = \frac{\pi}{2}.$$

Consequently, robust algorithms for computational quadrature *should* consider

- Singularities and discontinuities in integrands.
- unbounded integrands defined over $\pm\infty$

The basic issue of quadrature is to approximate an integral by a weighted sum of integrands at a finite number of sample points. Thus an integral of a Riemann-integrable function *f*, $I_N(f)$, is approximated by an n-point *quadrature rule*,

$$I_N(f) = \sum_{i=1}^{N} w_i f(x_i),$$

where the points at which the function is evaluated, x_i, are termed *nodes* or *abscissas* and the coefficients are called *weights*. These rules are termed *open* if $a < x_1$ & $x_n < b$ and (the more familiar) *closed* if $a = x_1$ & $x_n = b$. Quadrature rules are derived from polynomial interpolation:

- the integrand is sampled at a predetermined number of points,
- a polynomial interpolant is fitted to these points,
- the integral of the interpolant approximates the integral of the original function.

Note that any N-point (interpolatory) quadrature integrates *exactly* any polynomial of N – 1.

9.2 MANUAL INTEGRATIONS

Why not integrate a problem analytically by hand? Unfortunately, even the most trivial integration creates functions that cannot be represented by *simple* combinations of elementary functions. For example, the integration of the integrand, $\exp(-x^2)$, results in the error function, which most calculus courses do not identify. Thus manual integrations

- require significant amounts of time and energy,
- but the approach is error prone! (e.g. Krommer & Ueberhuber 1998 [p. 58]).

What about integration tables? *Table of Integrals, Series, and Products*, 5th Ed. 1994, Gradshteyn and Ryhik, is 1204 pages long! One would assume that such books would contain a majority of common closed-form integrals. Such an approach

- avoids the problems of manual integrations
- however, such tables are not free of errors. In a survey of eight common tables the frequency of errors was > 5 % (e.g. Krommer & Ueberhuber 1998 [p. 58])
- hence the comment by Alan Jeffrey in *Handbook of Mathematical Formulas and Integrals (*1995) "*whenever possible, results have been checked by means of computer symbolic algebra and integration programs.*" [p. xx]

9.3 SYMBOLIC INTEGRATIONS

This is less time-consuming and less error prone than using an integral table or doing by hand. Frequently such symbolic integrations do not exist. For example[1] -

[1] Mathematica finds

```
Mathematica 8.0 for Microsoft Windows (64-bit)
Copyright 1988-2011 Wolfram Research, Inc.

In[1]:= Integrate[Sin[x]/Log[x], x]

                   Sin[x]
Out[1]= Integrate[------, x]
                   Log[x]

In[2]:= Integrate[Sin[Sin[x]], x]

Out[2]= Integrate[Sin[Sin[x]], x]

In[3]:= Integrate[x^x,x]

                    x
Out[3]= Integrate[x , x]

In[4]:= Integrate[Sin[x]*Exp[-x], x]    % Mathematica does integrate here

        -(Cos[x] + Sin[x])
Out[4]= ------------------
                    x
                2 E
```

$$\int \frac{\sin x}{\ln(x)}dx,\ \int \sin(\sin(x))dx,\ \int x^{x}dx,$$

do not possess integrals that can be expressed in terms of functions. Moreover, symbolic integrals are expressed in terms of obscure functions: hypergeometric, gamma, etc.

9.4 ELEMENTARY METHODS → NEWTON-COTES RULES

The simplest, but *not* the most efficient, quadrature rules are based on equi-spaced nodes in the region [a, b]. Equi-spaced nodes are a defining property of the Newton-Cotes rules. An N-point open Newton-Cotes rules employs points

$$x_k = a + k\frac{(b - a)}{N + 1},\ k = 1,\cdots,N,$$

but N-point closed Newton-Cotes rules uses

$$x_k = a + (k - 1)\frac{(b - a)}{N - 1},\ k = 1,\cdots,N.$$

9.4.1Midpoint Rule

Approximating an integrand at the midpoint of an interval by a constant (a polynomial of degree zero), creates the one-point *open* Newton-Cotes rule termed the *midpoint* rule. The midpoint rule can be expressed as the following-

$$I_M = \int_a^b f(x)dx = (b - a)f(\frac{a + b}{2}) + \frac{(b - a)^3}{24}f''(\xi),\quad a \le \xi \le b$$

Since the error depends on the second derivative of integrand, the midpoint rule, of order 2, integrates exactly linear polynomials.

```
   function result = midpoint (f, a, b, N)
%  function result = midpoint (f, a, b, N)
%  INPUT:
%         f      = 1D function
%         N      = number of dependent values
%         a      = minimum dependent value
%         b      = maximum dependent value
%  OUTPUT:
%         result = integral
%       .. Composite Midpoint rule ..
          result = 0.0;
          h      = (b - a)/N;
          x      = linspace(a + h/2.0, b - h/2.0, N);
          result = sum(f(x)*h);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
>> midpoint(@sin, 0,pi,5)
ans = 2.0333
>> midpoint(@sin, 0,pi,10)
ans = 2.0082
>> midpoint(@sin, 0,pi,20)
ans = 2.0021
```

9.4.2 Trapezoid Rule

In this method function values are interpolated by first-degree polynomials (straight lines) in a two-point *closed* Newton-Cotes rule called the *trapezoid rule*. The trapezoid rule can be expressed as the following,

$$I_T = \int_a^b f(x)dx = \frac{b - a}{2}\left[f(a) + f(b)\right] + \frac{(b - a)^3}{12} f''(\xi), \quad a \le \xi \le b$$

Since the error depends on the second derivative of integrand, the trapezoid rule of order 2 integrates exactly linear polynomials, as does the midpoint rule. However, the trapezoid rule employs two function calls compared to the single function call of the midpoint rule, yet the error estimate of the trapezoid rule is twice that of the midpoint rule. This behavior might seem surprising. However, Newton-Cotes rules with *odd* values of N fit exactly polynomials of degree one greater than the degree of the interpolants upon which they are based. This results because positive and negative errors cancel for these odd-order quadrature rules.

```
   function result = trapezoid(f, a, b, N)
%  function result = trapezoid(f, a, b, N)
%  INPUT:
%         f    = 1D function
%         a    = minimum dependent value
%         b    = maximum dependent value
%         N    = number of dependent values
%  OUTPUT:
%         result = integral
%       .. Composite trapezoid rule ..
 result     = 0.0;
 h          = (b - a)/(N - 1);
 x          = a:h:b;
 integrand  = f(x);
 result     = h* ( 0.5*(f(a) + f(b)) + sum( f(x(2:end - 1)))  );
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

>> trapezoid(@sin, 0,pi,5)
ans = 1.8961
>> trapezoid(@sin, 0,pi,10)
ans = 1.9797
>> trapezoid(@sin, 0,pi,20)
ans = 1.9954
```

9.4.3 Simpson Rule

In this method function values are interpolated by a second-degree polynomial in a three-point (end points and midpoint) *closed* Newton-Cotes rule called the *Simpson's rule*. The Simpson's rule can be expressed as the following,

$$I_S = \int_a^b f(x)dx = \frac{b-a}{6}\left[f(a) + 4f(\frac{a+b}{2}) + f(b) \right] - \frac{(b-a)^5}{16} f^4(\xi), \quad a \le \xi \le b$$

Since the error depends on the fourth derivative of integrand, the Simpson's rule of fourth order integrates *exactly* cubic[2] equations.

```
  function result = simpson(f, a, b, N)
% function result = simpson(f, a, b, N)
%  INPUT:
%         f    = 1D function
%         a    = minimum dependent value
%         b    = maximum dependent value
%         N    = number of dependent values
%  OUTPUT:
%         result = integral
%       .. Composite Simpson's rule ..

          result    = 0.0;
          h         = (b - a)/N;
          x         = linspace(a, b, N + 1);
          integrand = f(x);

%         .. Loop for odd points ..
          result = 4.0 * sum( integrand(2:2:end - 1) );

%          .. Loop for even points  ..
          result = result + 2.0 * sum( integrand(3:2:end - 1) );

%          .. Add the endpoints ..
           result = result + (f(a) + f(b);
           result = result*h/3.0;

>>  simpson(@sin, 0,pi,5)
ans = 1.9338
>> simpson(@sin, 0,pi,10)
ans = 2.0001
>> simpson(@sin, 0,pi,20)
ans = 2.000006784441801
```

Please note that the trapezoid and Simpson's rule are *closed* quadrature rules (evaluated at both boundaries) and consequently are unable to handle non-removable singularities at integral boundaries, unlike open quadrature rules.

---

[2]
```
>> simp = @(a, b, g) (b - a)*(g(a) + 4.0*g((a+b)/2.0) + g(b))/6.0;
>> a = 0.0; b = 1.0; f = @(x) x.^3;
>> simp(a, b, f)
ans = 0.250000000000000   % yes, simpson's rule integrates exactly a cubic!
>> q = @(x) x.^4;
>> simp(a, b, q)
ans = 0.208333333333333
```

9.4.4 Higher-Order Newton-Cotes Rules

Examples of higher-order Newton-Cotes closed rules can be examine, e.g. in van Loan (2000). However, higher-order methods are used less frequently than the trapezoid and the Simpson rules. Why? First, fitting these high-order interpolants to equispaced data points creates unphysical "wiggles" or oscillations between the data points, particularly near the boundaries. Second, far more efficient rules, developed by Gauss exist.

```
>> newton_cotes(@sin, 0, pi, 6)
ans = 1.99920309391571
>> newton_cotes(@sin, 0, pi, 7)
ans = 2.00001781363666
>> newton_cotes(@sin, 0, pi, 8)
ans = 2.00001086554154
>> newton_cotes(@sin, 0, pi, 8)
ans = 2.00001086554154
>> newton_cotes(@sin, 0, pi, 10)
ans =1.99999989482634
>> newton_cotes(@sin, 0, pi, 11)
ans = 2.00000000114677
```

9.5 GAUSSIAN QUADRATURE

A collection of Gaussian algorithms, *QUADPACK* (Piessens et al. 1983) has become a "*de facto standard for one-dimensional integrations*" (Krommer & Ueberhuber 1998. [p. 69]). The comprehensive procedure libraries of IMSL (International Mathematical & Statistical Libraries) and Nag (Numerical Algorithms Group) use Gaussian quadrature rules and NOT Newton-Cotes routines for 1D integrations.

Gaussian quadrature has the *optimum* accuracy for the number of nodes employed. Note that this *optimum* accuracy results *"only when the integrand is very smooth in the sense of being well approximated by a polynomial"* ([p. 140], Press et al. 1992). Moreover, to employ Gaussian quadrature even for smooth integrands it is *essential* that the integrand function be accessible (not in tabulated data format) since the Gaussian nodes are irrational ([p. 193] Neumaier 2001).

9.5.1 Finite Domain [a, b]

Gaussian quadrature rules, like the Newton-Cotes rules, are based on polynomial interpolants. However, locations of nodes are not pre-assigned, but are parameters to be optimized by the fitting procedure. Thus for N points a Gaussian quadrature rule can be determined to fit *exactly* a polynomial up to and including polynomials of 2N –1 order. Gaussian quadrature rules are expressed in the following format -

$$\int_{-1}^{+1} f(x)dx = \sum_{n=1}^{N} \omega_n f(x_n) + R_n,$$

$$R_n = \frac{2^{2n+1}(n!)^4}{(2n+1)[(2n)!]^3} f^{(2n)}(\xi), \quad -1 \le \xi \le +1$$

where x_n and ω_n are termed, respectively, nodal abscissas and weights ([p. 887] Abramowitz & Stegun 1972). The *nominal* Gaussian integration range is [-1, +1], but a following simple linear substitution allows the range to be extended to [a, b]

$$\text{Tabulated rule: } \int_{-1}^{+1} g(x)dx = \sum_{k=1}^{N} \omega_k g(x_k) \Rightarrow \int_{a}^{b} g(t)dt$$

$$t = 0.5\left[(b - a)x + (b + a)\right], \qquad a \le t \le b \;\Rightarrow\; -1 \le x \le +1$$

$$I(g) = 0.5(b-a)\sum_{k=1}^{N} \omega_k g(\{0.5(b - a)x_k + (a+b)\})$$

Gaussian quadrature rules can be derived by the method of undetermined coefficients (easier to understand) and from orthogonal polynomials (more accurate). For example, as noted above, a two-point Gaussian quadrature rule must integrate *exactly* cubic equations. Here the 2-point Gaussian quadrature rule

$$I_2(f) = w_1 f(x_1) + w_2 f(x_2),$$

is derived by the method of undetermined coefficients. Such a quadrature rule must integrate exactly cubic equations. The four variables, the two weights and two nodes, are determined from a system of *nonlinear* equations fitting the first four monomial bases, 1, x, x^2, & x^3.

$$f(x) = 1: \quad w_1 + w_2 = \int_{-1}^{+1} dx = 2,$$

$$f(x) = x: \quad w_1 x_1 + w_2 x_2 = \int_{-1}^{+1} x dx = [x^2/2]_{-1}^{+1} = 0,$$

$$f(x) = x^2: \quad w_1 x_1^2 + w_2 x_2^2 = \int_{-1}^{+1} x^2 dx = [x^3/3]_{-1}^{+1} = \frac{2}{3},$$

$$f(x) = x^3: \quad w_1 x_1^3 + w_2 x_2^3 = \int_{-1}^{+1} x^3 dx = [x^4/4]_{-1}^{+1} = 0.$$

Solving this system of four non-linear equations with **fsolve.m** -

Derivation of Gauss-Lengendre Quadrature Weights

```
function F = nonlinear_gauss2(x)
    x1 = x(1);
    x2 = x(2);
    A1 = x(3);
    A2 = x(4);
    F = [(A1      + A2 -2.0);
         (A1*x1   + A2*x2);
         (A1*x1^2 + A2*x2^2 -2/3);
         (A1*x1^3 + A2*x2^3)];
```

```
>> options = optimset('Display', 'off', 'TolX', 1.0e-13, 'TolFun', 1.0e-13);
>> x0                          = [1 -1 1 1]';
>> [r, fval, flag, output] = fsolve(@nonlinear_gauss2, x0, options);
>> flag
flag   = 1
x0     = [1 -1 1 1]';
r      =    +0.577350269189626   % +1/√3
            -0.577350269189626   % -1/√3
            +1.000000000000000
            +1.000000000000000
>> x = [+0.577350269189626  -0.577350269189626]';
>> w = [+1.000000000000000  +1.000000000000000]';
>> a = 0.0; b = 1.0; f = @(t) t.^3;
>> t = ((b - a)*x + (b + a))/2.0
>> t = ((b - a)*x + (b + a))/2.0
t = 0.788675134594813
    0.211324865405187
>> sum(w.*f(t))*0.5*(b - a)     % 2-pt Gaussian solves exactly a cubic
ans = 0.250000000000000

>> h = @(t) t.^4;
>> sum(w.*h(t))*0.5*(b - a)
ans = 0.194444444444445
```

```
function F = nonlinear_gauss3(x)
    x1 = x(1);
    x2 = x(2);
    x3 = x(3);
    A1 = x(4);
    A2 = x(5);
    A3 = x(6);
    F = [(A1      + A2      + A3  -2.0);
         (A1*x1   + A2*x2   + A3*x3);
         (A1*x1^2 + A2*x2^2 + A3*x3^2 -2/3);
         (A1*x1^3 + A2*x2^3 + A3*x3^3);
         (A1*x1^4 + A2*x2^4 + A3*x3^4 -2/5);
         (A1*x1^5 + A2*x2^5 + A3*x3^5)];
```

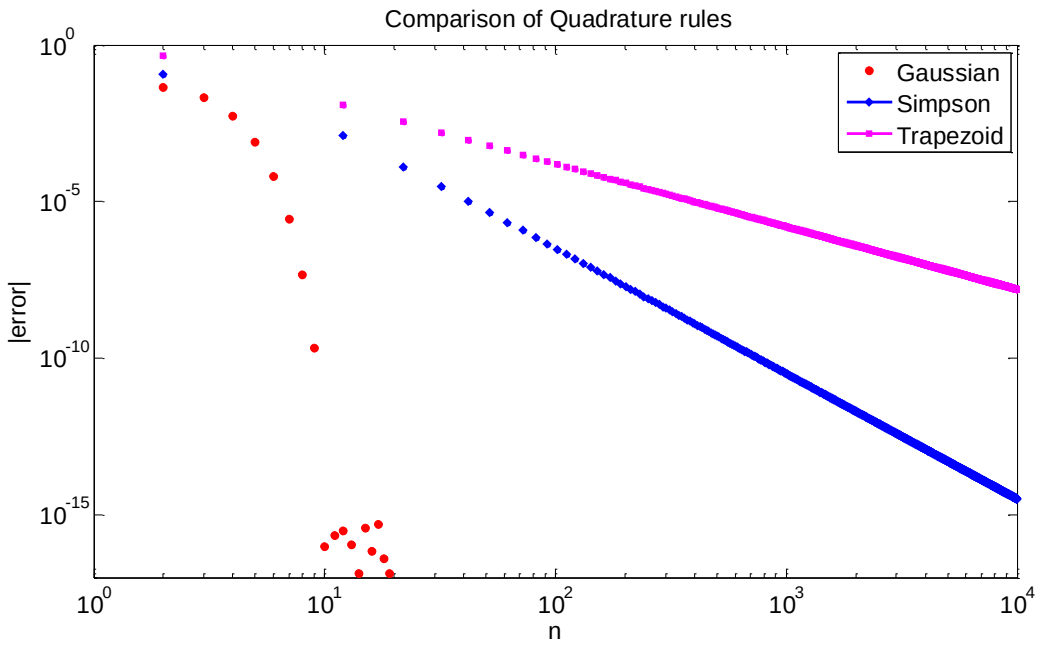
```
>> x0 = [-1 1 1 1 1 1]';
>> [r, fval, flag, output] = fsolve(@nonlinear_gauss3, x0, options);
>> flag
flag =  1
>> max(abs(fval))
ans =  5.55e-017
>> r
r =-0.774596669241483    % -(3/5)^0.5
    0.774596669241483    % +(3/5)^0.5
   -0.000000000000000    % + 0.0
   +0.555555555555555    % + 5/9
   +0.555555555555555    % + 5/9
   +0.888888888888889    % + 8/9
>> x = [ -0.774596669241483 +0.774596669241483 -0.000000000000000]';
>> w = [  0.555555555555555  0.555555555555555  0.888888888888889]';

>> a = 0.0; b = 1.0; f = @(t) t.^5;
```

```
>> t = ((b - a)*x + (b + a))/2.0;
>> sum(w.*f(t))*0.5*(b - a)
ans = 0.166666666666666            % 3-pt Gaussian solves exactly a quintic (5th)
```

Note that in practice Gaussian abscissas and weights are computed using zeros of orthogonal polynomials ([p. 143] Press et al. 1992).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        function result = gauss_quadrature(f, a, b, n)
      % function result = gauss_qyadrature(f, a, b, n)
      %  INPUT:
      %         f      = 1D function
      %         a      = minimum dependent value
      %         b      = maximum dependent value
      %         n      = number of dependent values
      %  OUTPUT:
      %         result = integral

       [t, c]  = gauss_coef(n - 1);
        x      = (t*(b-a)  + a + b)/2.0;
        s      = sum(c.*f(x));
        result =(b - a)*s/2.0;
       end
       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       function [t, c] = gauss_coef(n)
      % Trefethen, L N (2008)
      % SIAM Review, 50, 67-87
      % (n+1)-pt Gauss quadrature of f
      beta    = .5./sqrt(1-(2*(1:n)).^(-2)); % 3-term recurrence coeffs
      T       = diag(beta,1) + diag(beta,-1); % Jacobi matrix
      [V, D]  = eig(T); % eigenvalue decomposition
      x       = diag(D); [x,i] = sort(x); % nodes (= Legendre points)
      w       = 2*V(1,i).^2; % weights
      t       = x'; c = w;
       end

      >> gauss_quadrature(@sin, 0.0, pi, 2)
      ans =  1.935819574651136
      >> gauss_quadrature(@sin, 0.0, pi, 4)
      ans =  1.999984228457722
      >> gauss_quadrature(@sin, 0.0, pi, 6)
      ans =  1.999999999477271
      >> gauss_quadrature(@sin, 0.0, pi, 8)
      ans =  1.999999999999996
```

The following plot is the comparison of three quadrature rules, calculated in **integral_comparison.m**, The efficiency of Gaussian quadrature is evident.



Usually Gaussian quadrature is the preferred numerical method for smooth functions, as seen by its use in the IMSL & Nag procedural libraries:

Why use Gaussian quadrature algorithms?

- Weights are always positive → stable method.
- Optimal accuracy for a chosen number of points.
- Works well with non-removable singularities.
- Are open rules, *without* integrand evaluations at boundaries. (Integration difficulties frequently occur at end points.)

9.5.2 Semi-infinite Domain [0, ∞]

Special forms of Gaussian quadrature have been developed for semi-infinite domains –

$$\int_0^\infty e^{-x} g(x)dx = \sum_{k=1}^{N} \omega_k g(x_k),$$

where an exponential weighting term is employed, which is termed Gauss-Laguerre quadrature. Here a 2-point Gaussian quadrature rule

$$I_2(g) = \omega_1 g(x_1) + \omega_2 g(x_2),$$

is derived by the method of undetermined coefficients. Such a quadrature rule must integrate exactly cubic equations. Thus the four variables, the two weights and two nodes, are determined from a system of nonlinear equations fitting the first four monomial bases, 1, x, x^2, & x^3.

$$g(x) = 1: \quad w_1 + w_2 = \int_0^\infty e^{-x}dx = 1,$$

$$g(x) = x: \quad w_1x_1 + w_2x_2 = \int_0^\infty xe^{-x}dx = 1,$$

$$g(x) = x^2: \quad w_1x_1^2 + w_2x_2^2 = \int_0^\infty x^2e^{-x}dx = 2,$$

$$g(x) = x^3: \quad w_1x_1^3 + w_2x_2^3 = \int_0^\infty x^3e^{-x}dx = 6.$$

Mathematica 8.0:

```
Solve[{w1 + w2 == 1, w1*x1 + w2*x2 == 1, w1*x1^2 + w2*x2^2 == 2,
w1*x1^3 + w2*x2^3 == 6}]
```

$$\left\{\left\{w1 \to \frac{1}{4}\left(2-\sqrt{2}\right),\ w2 \to \frac{1}{4}\left(2+\sqrt{2}\right),\ x2 \to 2-\sqrt{2},\ x1 \to 2+\sqrt{2}\right\},\right.$$
$$\left.\left\{w1 \to \frac{1}{4}\left(2+\sqrt{2}\right),\ w2 \to \frac{1}{4}\left(2-\sqrt{2}\right),\ x2 \to 2+\sqrt{2},\ x1 \to 2-\sqrt{2}\right\}\right\}$$

```
N[%]
{{w1→0.146447,w2→0.853553,x2→0.585786,x1→3.41421},{w1→0.853553,w2→0
.146447,x2→3.41421,x1→0.585786}}
```

```
  function result = gauss_laguerre(f, n)
% function result = gauss_laguerre(f, n)
%  INPUT:
%         f      = 1D function
%         n      = number of dependent values
%  OUTPUT:
%         result = integral

[t, c] = coefficient_gauss_laguerre(n);
c      = c.*exp(t);
  s = 0;
  for k =1 :n
    x=t(k); y = f(x);
    s=s + c(k)*y;
  end
result = s;

function [x, w] = coefficient_gauss_laguerre(n)
% Numerical Mathematics, 2000, (Springer-Verlag: Berlin)
% A Quateroni, R Sacco, & F Saleri  p. 430
if (n <= 1), disp(' n must be > 1 '); return; end
[a,b] = ceofficient_laguerre_poly(n);
JacM=diag(a)+diag(sqrt(b(2:n)),1)+diag(sqrt(b(2:n)),-1);
[w,x]=eig(JacM); x=diag(x); w=w(1,:)'.^2;

function [a, b] = ceofficient_laguerre_poly(n)
% Numerical Mathematics, 2000, (Springer-Verlag: Berlin)
% A Quateroni, R Sacco, & F Saleri  p. 430
```

```
if (n <= 1), disp(' n must be > 1 '); return; end
a=zeros(n,1); b=zeros(n,1); a(1)=1; b(1)=1;
for k=2:n, a(k)=2*(k-1)+1; b(k)=(k-1)^2; end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

>> h = @(x) log(1 + exp(-x));
>> result = gauss_laguerre(h, 12)
result = 0.822467025596490
```

Mathematica 8.0:

```
Integrate[Log[1 + Exp[-x]], {x, 0, Infinity}]
```

$\frac{\pi^2}{12}$

```
N[%]
0.822467
>> result - pi^2/12    % from above
ans = 1.739362209818296e-008
```

9.5.3 Infinite Domain [-∞, ∞]

Another special form of Gaussian quadrature has been developed for infinite domains –

$$\int_{-\infty}^{+\infty} e^{-x^2} g(x)dx = \sum_{k=1}^{N} \omega_k h(x_k),$$

where yet another Gaussian weighting term is employed. This method is called Gauss-Hermite quadrature. Here a 2-point Gaussian quadrature rule

$$I_2(h) = \omega_1 h(x_1) + \omega_2 h(x_2),$$

is derived by the method of undetermined coefficients. Such a quadrature rule must integrate exactly cubic equations. Thus the four variables, the two weights and two nodes, are determined from a system of nonlinear equations fitting the first four monomial bases, 1, x, x^2, & x^3.

$$h(x) = 1:\ w_1 + w_2 = \int_{-\infty}^{+\infty} e^{-x^2} dx = \sqrt{\pi},$$

$$h(x) = x:\quad w_1 x_1 + w_2 x_2 = \int_{-\infty}^{+\infty} x e^{-x^2} dx = 0,$$

$$h(x) = x^2:\quad w_1 x_1^2 + w_2 x_2^2 = \int_{-\infty}^{+\infty} x^2 e^{-x^2} dx = \frac{\sqrt{\pi}}{2},$$

$$h(x) = x^3:$$

$$w_1 x_1^3 + w_2 x_2^3 = \int_{-\infty}^{+\infty} x^3 e^{-x^2} dx = 0,$$

Mathematica 8.0:

```
s = Solve[ {w1 + w2 ==Sqrt[Pi], w1*x1 + w2*x2 == 0, w1*x1^2 + w2*x2^2
== Sqrt[Pi]/2, w1*x1^3 + w2*x2^3 ==0}]
```

$$\left\{\left\{w1 \to \frac{\sqrt{\pi}}{2},\ w2 \to \frac{\sqrt{\pi}}{2},\ x2 \to -\frac{1}{\sqrt{2}},\ x1 \to \frac{1}{\sqrt{2}}\right\},\ \left\{w1 \to \frac{\sqrt{\pi}}{2},\ w2 \to \frac{\sqrt{\pi}}{2},\ x2 \to \frac{1}{\sqrt{2}},\ x1 \to -\frac{1}{\sqrt{2}}\right\}\right\}$$

```
N[%]
{{w1→0.886227,w2→0.886227,x2→-0.707107,x1→0.707107},
 {w1→0.886227,w2→0.886227,x2→0.707107,x1→-0.707107}}

>> [x, w] = coefficent_gauss_hermite(2)  % MATLAB
x = -0.70710678118655                % Inside gauss_hermite.m
     0.70710678118655
w =  0.88622692545276
     0.88622692545276

  function result = gauss_hermite(f, n)
% function result = gauss_hermite(f, n)
%  INPUT:
%         f      = 1D function
%         n      = number of dependent values
%  OUTPUT:
%         result = integral

[t,c] = coefficent_gauss_hermite(n);
      c     = c.*exp(t.^2);
        s=0;
        for k=1:n
          x1 = t(k);
          y  = f(x1);
          s  = s + c(k)*y;
        end
      result = s;

function [x,w]= coefficent_gauss_hermite(n)
% Numerical Mathematics, 2000, (Springer-Verlag: Berlin)
% A Quateroni, R Sacco, & F Saleri  p. 430
if (n <= 1), disp(' n must be > 1 '); return; end
[a,b] = coefficient_hermite_poly(n);
```

```
JacM=diag(a)+diag(sqrt(b(2:n)),1)+diag(sqrt(b(2:n)),-1);
[w,x]=eig(JacM); x=diag(x); scal=sqrt(pi); w=w(1,:)'.^2*scal;
[x,ind]=sort(x); w=w(ind);

function [a, b] = coefficient_hermite_poly(n)
% Numerical Mathematics, 2000, (Springer-Verlag: Berlin)
% A Quateroni, R Sacco, & F Saleri  p. 430
if (n <= 1), disp(' n must be > 1 '); return; end
a=zeros(n,1); b=zeros(n,1); b(1)=sqrt(4.*atan(1.));
for k=2:n, b(k)=0.5*(k-1); end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Mathematica 8

```
Integrate[1/(1 + x^2)^2, {x, -Infinity, Infinity}]
```

$$\frac{\pi}{2}$$

```
 f = @(x)1/(1 + x^2)^2;
>> gauss_hermite(@f7, 64)
ans =  1.57029022883813
>> ans - pi/2
ans = -5.060979567648172e-004
```

9.6 ADAPTIVE GAUSSIAN QUADRATURE -- GAUSS-KRONROD RULES

Usually error estimates for quadrature rules are obtained by determining the difference between results of two *different* rules applied to the *same* integral. For the sake of efficiency it is useful to evaluate the integrand at nodes common to both rules. This is simple for Newton-Cotes rules, but not for Gaussian quadrature. However, Gaussian rules have no common points, except for the midpoint found in all odd-order rules. Nonetheless, Gauss-like, Kronrod rules have been developed which share some common nodes, leading to the development of very accurate error estimates.

Kronrod (K) rules occur in pairs; thus an N-point Gaussian (G) rule is related to a (2N+1)-point Kronrod rule. The fundamental principle of a Kronrod rule is that the K_{2N+1} nodes includes all the nodes of G_N; the left over N + 1 nodes as well as all the (2N+1) weights remain to be optimized. Thus the K_{2N+1} rule can reproduce exactly a 3N+1 polynomial. Note that a true 2N+1 Gaussian rule would reproduce exactly a 4N+1 polynomial. The Gauss-Kronrod rules are used to obtain accurate error estimates. The K_{2N+1} rule is used to approximate the integral, but the error estimate, ε, is estimated heuristically by

$$\varepsilon \approx (200\left|G_N - K_{2N+1}\right|)^{1.5},$$

based primarily on experience. In the words of Michael Heath (2002) "*Gauss-Kronrod rules are among the most effective quadrature rules available, and they form the basis for many of the quadrature routines in major software libraries*". The pair rule (G_7, K_{15}), in particular, has become a commonly used standard. MATLAB implements the 7-15 pair in **quadgk.m**.

The calling sequence for **quadgk.m** is the following:

```
r           = quadgk(f, a, b)
[r, errbnd] = quadgk(f, a, b)
[r, errbnd] = quadgk(f, a, b, 'AbsTol', 1.0e-12, 'RelTol',1.0e-09)
[r, errbnd] = quadgk(f, a, b, 'Params, values)
```

Observe that the integrand function, f(x), must be in vector format!
Singular points should be handled by making them endpoints of *separate* integrations and adding results.

r	approximates the value of the integral.
errbnd	**estimate of the** upper bound on the absolute error, **|r – exact|.**
a & b	are integration limits. Limits a and b can be **–inf** and **+inf.** If limits are finite, they can be complex valued.
'AbsTol'	default value is 1.0×10^{-10}.
'RelTol'	default value is 1.0×10^{-6}. Minimum value is 100***eps**.
'Waypoints'	If f(x) has discontinuities in the integration interval, locations should be supplied as a **Waypoints** vector.

```
>> f = @(x) x.^x;
>> [r, errbnd] = quadgk(f, 0.0, 1.0, 'AbsTol', 1.0e-12, 'RelTol', 1.0e-09)
r      = 0.783430510712433
errbnd = 3.730721259698199e-010
```

Mathematica 8.0: % Also computational result but with 32 digits of precision

```
NIntegrate[x^x, {x, 0, 1}, WorkingPrecision →32]
0.78343051071213440705 93
```

```
>> h           = @(x) log(1 + exp(-x));    % from gauss_laguerre example
>> [r, errbnd] = quadgk(h, 0, inf);
>> r
r      =   0.822467033424113
>> errbnd
errbnd =   1.601561923265661e-011
>> r - pi^2/12
ans = -6.661338147750939e-016
```

```
>> g = @(x) 1.0./(1.0 + x.^2).^2;          % from gauss_hermite example
>> [r, errbnd] = quadgk(g, -inf, inf);
>> r
r      = 1.570796326794897
>> errbnd
errbnd = 1.000921567850810e-013
>> r - pi/2
ans    =  2.220446049250313e-016
```

9.8 ADAPTIVE SIMPSON ROUTINES

As you saw earlier error estimates for single steps of the Simpson rule are proportional to h^5. If the same region is approximated as the sum of two regions each of with h/2, the error in each of them is ~ $(h/2)^5 f^4(\xi)/16$, but the total error, associated with n steps, is estimated to be twice this, 1/16 E(h), since there exists two equispaced regions. If Q and Q2 are Simpson approximations, and I_{exact} is the actual integral,

$$
\begin{aligned}
Q2 - Q &= (I_{exact} - E(h/2)) - (I_{exact} - E(h)) \\
&= -E(h/2) + E(h) \\
&= -E(h/2) + 16E(h/2) \\
&= 15E(h/2).
\end{aligned}
$$

These simple approximations provide a mechanism by which one can judge whether the two-panel Simpson rule is superior to the single-panel Simpson rule at given tolerance, *tol*,

$$
\begin{aligned}
E(h/2) &\approx \frac{1}{15}[Q2 - Q], \\
&\approx \frac{1}{15}[Q2 - Q] \leq \text{tol}.
\end{aligned}
$$

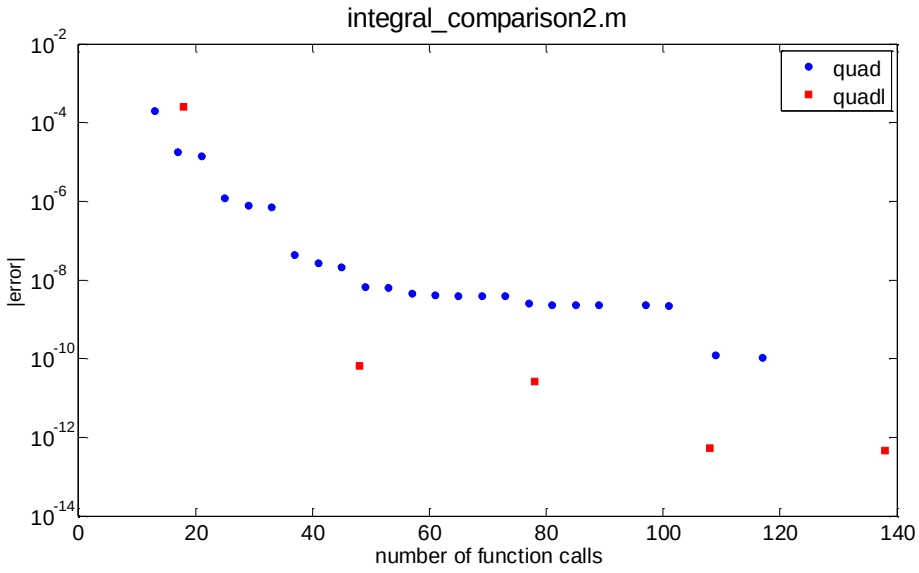
If the tolerance relation is not satisfied, a region is again split and the adaptive Simpson's rule is called again to estimate the integral of half the region. However, note that only regions which do not satisfy the error tolerance are split.

```
  function result = adaptive_simpson(f, a, b, tol)
% function result = adaptive_simpson(f, a, b, tol)
%  INPUT:
%         f      = 1D function
%         a      = minimum dependent value
%         b      = maximum dependent value
%         tol    = required absolute (NOT relative) accuracy of the solution
%  OUTPUT:
%         result = integral
%       .. Adaptive Simpson's rule ..

    mid = (b + a)/2.0;
    one_simpson_area = simpson_approx (f, a, b);
    two_simpson_area = simpson_approx (f, a, mid) + simpson_approx (f, mid, b);
    if( abs(one_simpson_area - two_simpson_area) < 15.0*tol )
              result = two_simpson_area;
    else
         left_area   = adaptive_simpson (f, a, mid, tol);
         right_area  = adaptive_simpson (f, mid, b, tol);
         result      = left_area + right_area;
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [integral_estimate, m] = simpson_approx (f, a, b)
     h = (b - a)/2.0;
     integral_estimate = h*(f(a) + 4.0*f(a + h) + f(b) )/3.0;
end

>> adaptive_simpson(@sin, 0, pi, 100*eps)
ans =  2.000000000001096
```

9.8 MATLAB INTRINSICS: **quad and quadl**

```
>> help quad  (MATLAB intrinsic)
QUAD Numerically evaluate integral, adaptive Simpson quadrature. Q = quad(FUN,A,B)
tries to approximate the integral of function FUN from A to B to within an error
of 1.e-6 using recursive adaptive Simpson quadrature.


>> help quadl
quadl    Numerically evaluate integral, adaptive Lobatto quadrature. Q =
quadl(FUN,A,B) tries to approximate the  integral of function FUN from A to B to
within an error of 1.e-6 using high order recursive adaptive quadrature. [Lobatto
quadrature is a variant of Gaussian quadrature where the end points are included
in the calculation (Mathworld.wolfram.com).]
```

The following plot is the comparison of MATLAB's **quad** & **quadl** –



9.9 EXAMPLES

```
>> f = @(x)  x.*sin(30.*x)./(sqrt(1 - (x/(2*pi)).^2));
>> ezplot(f, [0.0, 2*pi])

>> [r, errbnd] = quadgk(f, 0.0, 2.0*pi, 'AbsTol', 1.0e-12, 'RelTol', 1.0e-09)
r      = -2.543259618893549
errbnd =  1.484547218662257e-010

>> r =  quadl(f, 0.0, 2.0*pi, 100*eps)
Warning: Maximum function count exceeded; singularity likely.
> In quadl at 106
r =   3.080871010854835
>> r =  quadl(f, 0.0, 2.0*pi, 1.0e-12)
r = -2.543259618890357
```

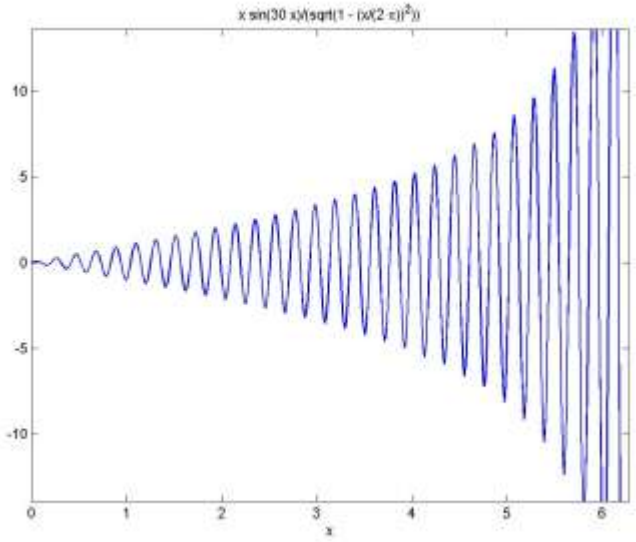
```
>> r =  quad(f, 0.0, 2.0*pi, 1.0e-12)
Warning: Maximum function count exceeded; singularity likely.
> In quad at 107
r = -7.350958103166474
>> r =  quad(f, 0.0, 2.0*pi, 1.0e-10)
r =  -2.543259618582663
```

Analytic Mathematica 8.0

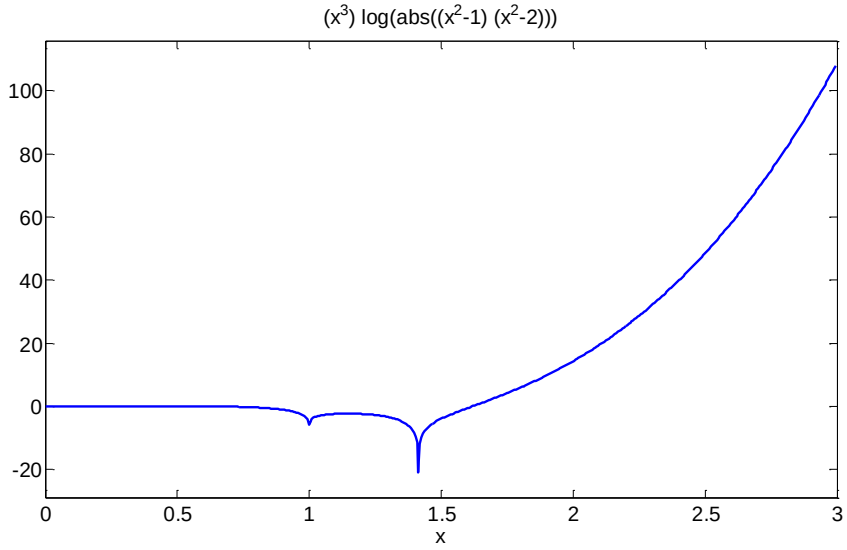
```
Integrate[x*Sin[30*x]/Sqrt[1 - (x/(2*Pi))^2], {x, 0, 2*Pi} ]
```

$$2\pi^3 \, \text{BesselJ}[1, 60\pi]$$

```
N[%,30]
-2.54325961889353148987086198257
```




```
>> f = @(x) (x.^3) .*log(abs( (x.^2 - 1).* (x.^2 - 2)));  % two singularities
>> ezplot(f, [0, 3])
```

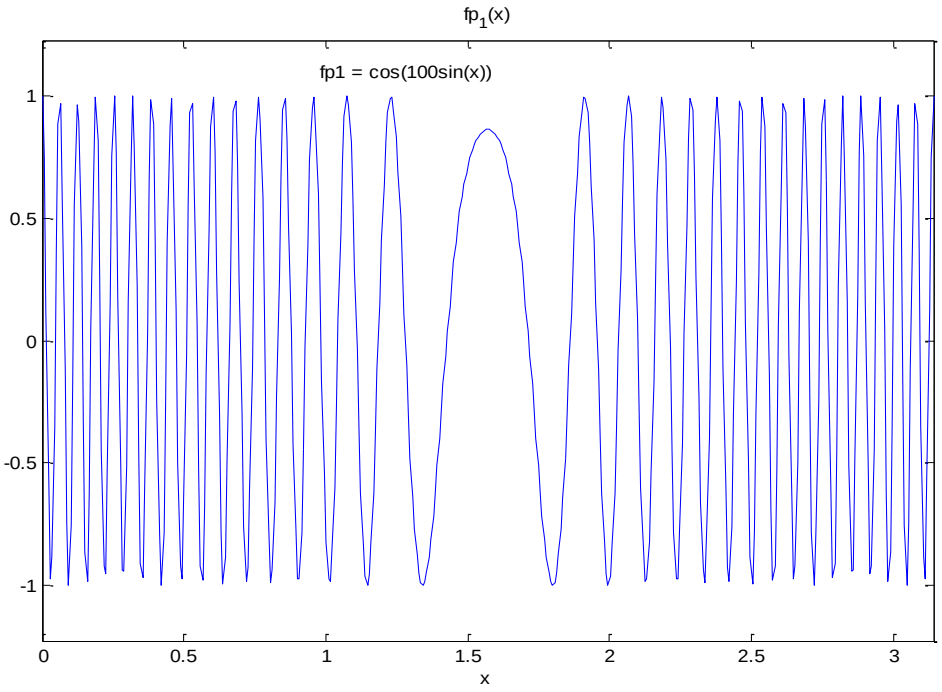



```
>> [r, errbnd] = quadgk(f, 0.0, 3.0, 'AbsTol', 1.0e-12, 'RelTol', 1.0e-10);
>> r
r      = 52.740748385962839
>> errbnd
errbnd =  3.084765620227532e-009

>> r = quadl(f, 0.0, 3.0, 1.0e-12)
r =  52.740748383482881

>> r = quad(f, 0.0, 3.0, 1.0e-12)
r =  52.740748383457429
```

Analytic Mathematica 8.0

```
      Integrate[Cos[100*Sin[x]], {x, 0, Pi}]
           π BesselJ[0,100]
      N[%, 30]
0.062787400491492695655032824056
```

```
>> ezplot('fp1', [0, pi])
```




```
>> f = @(x)  cos(100.0.*sin(x));
>> [r, errbnd] = quadgk(f, 0.0, pi, 'AbsTol', 1.0e-12, 'RelTol', 1.0e-10);
>> r
r =      0.062787400491493
>> errbnd
errbnd = 4.561113205875376e-013

>> r = quad(f, 0.0, pi, 1.0e-12);
Warning: Maximum function count exceeded; singularity likely.
> In quad at 107

>> r = quad(f, 0.0, pi, 1.0e-10);
>> r
r = 0.062787400504379

>> r = quadl(f, 0.0, pi, 1.0e-12);
Warning: Maximum function count exceeded; singularity likely.
> In quadl at 106

>> r = quadl(f, 0.0, pi, 1.0e-11);
>> r
r = 0.062787400491493
```

9.10 REFERENCES

Abramowitz, M., & Stegun, I. A. 1965, *Handbook of Mathematical Functions* (Dover: NY, NY).

Heath, M. T. 2002, *Scientific Computing: An Introductory Survey, 2nd Ed.,* (McGraw-Hill: NY).

Krommer, A. R. and Ueberhuber, C. W. 1998, *Computational Integratio*n, (SIAM: Philadelphia, PA)

Lindfield, G., & Penny, J. 2000, *Numerical Methods Using MATLAB*, 2nd Ed (Prentice-Hall: Saddle River, NJ).

Neumaier, A. 2001, *Introduction to Numerical Analysis* (Cambridge: Cambridge, UK).

Piessens, R, de Doncker-Kapenga, E, Uberhuber, C W & Kahaner D K. 1983, *QUADPACK A Subroutine Package for Automatic Integration* (Springer-Verlag: Berlin).

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 1992, *Numerical Recipes in Fortran, The Art of Scientific Computing,* 2nd Ed. (Cambridge: Cambridge UK).

Quarteroni, A., Sacco, R., & Saleri, F. 2000, *Numerical Mathematics*, (Springer-Verlag: Berlin).

van Loan, C. 2000, *Introduction to Scientific Computing, 2nd Ed.* , (Prentice-Hall: Saddle River, NJ).

9.12 QUESTIONS

1. True or false. Evaluating a definite integral is always a well-conditioned problem.

2. True or false: Because it is based on polynomial interpolation of degree one higher, the trapezoid rule is generally more accurate than the midpoint rule

3. True or false: The degree of a quadrature rule is the degree of the interpolating polynomial quadrature rule on which the rule is based.

4. True or false. An n-point Newton-Cotes rule is always of degree of n – 1.

5. True or false. Gaussian quadrature rules of different order never have any points in common.

6. What conditions are both necessary and sufficient for a Riemann integral to exist?

7. (a) Under what conditions is a definite integral likely to be sensitive to small perturbations in the integrand? (b) Under what conditions is a definite integral likely to be sensitive to small perturbations in the limits of integration?

8. What is the difference between an open quadrature rule and a closed quadrature rule?

9. Name two different methods for computing the weight corresponding to a given set of nodes of a quadrature rule.

10. How can you estimate the error in a quadrature rule without computing the derivative of the integrand function that would be required by a Taylor series expansion?

12. (a) How does the node placement differ between Newton-Cotes quadrature and Gaussian quadrature? (b) Which would you expect to be more accurate for the same number of nodes? Why?

13. (a) If a quadrature rule for an interval [a, b] is based on polynomial interpolation at n equally spaced points in the interval, what is the highest degree such that the rule integrates all polynomials of that degree exactly? (b) How would your answer change if the points were optimally placed to integrate the highest possible degree polynomials exactly?

15. (a) What is the degree of Simpson's rule for numerical quadrature? (b) What is the degree of an n-point Gaussian quadrature rule?