Chapter 4

Numerical Integration

An m-point *quadrature rule* Q for the definite integral

$$I(f,a,b) = \int_a^b f(x)dx \tag{4.1}$$

is an approximation of the form

$$I_Q(f,a,b) \;=\; (b-a)\sum_{k=1}^{m} w_k f(x_k). \tag{4.2}$$

The x_k are the *abscissas* and the w_k are the *weights*. The abscissas and weights define the rule and are chosen so that $I_Q(f,a,b) \approx I(f,a,b)$. *Efficiency essentially depends upon the number of function evaluations.* This is because the time needed to evaluate f at the x_i is typically *much* greater than the time needed to form the required linear combination of function values. Thus, a six-point quadrature rule is twice as expensive as a three-point rule.

We start by presenting the the *Newton-Cotes* family of quadrature rules. These rules are derived by integrating a polynomial interpolant of the integrand $f(x)$. Composite rules based on a partition of $[a,b]$ into subintervals are then discussed in §4.2. In a composite rule, a simple rule is applied to each subintegral and the result summed. The adaptive determination of the partition with error control is presented in §4.3. The partition is determined recursively using heuristic estimates of the integrand's behavior. In §4.4 we discuss the "super accuracte" Gauss quadrature idea and also how to approach the quadrature problem using splines when the integrand is only known through a discrete set of sample points.

4.1 The Newton-Cotes Rules

One way to derive a quadrature rule Q is to integrate a polynomial approximation $p(x)$ of the integrand $f(x)$. The philosophy is that $p(x) \approx f(x)$ implies

$$\int_a^b f(x)dx \approx \int_a^b p(x)dx.$$

(See Figure 4.1.) The *Newton-Cotes* quadrature rules are obtained by integrating uniformly spaced polynomial interpolants of the integrand. The m-point Newton-Cotes rule ($m \geq 2$) is defined by

$$Q_{\text{NC}(m)} = \int_a^b p_{m-1}(x)dx, \tag{4.3}$$

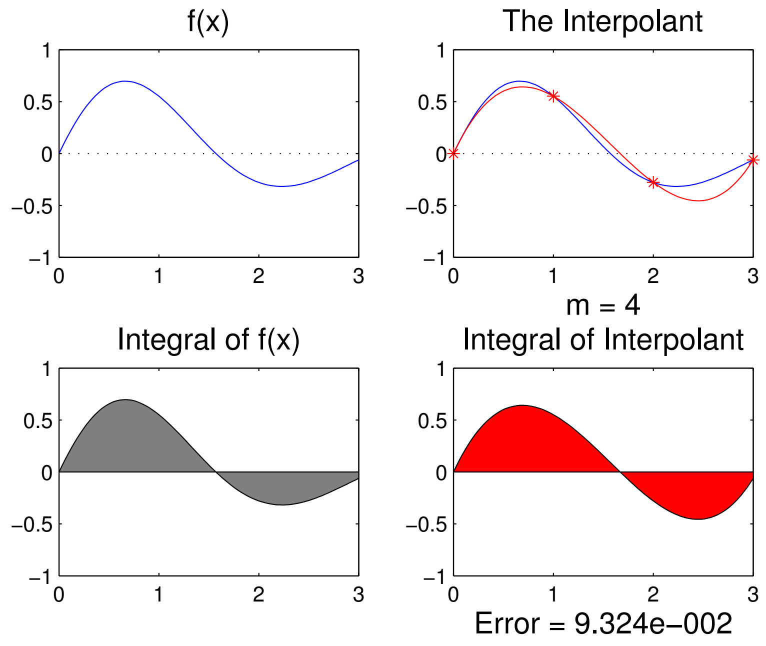


FIGURE 4.1 *The Newton-Cotes idea*

where $p_{m-1}(x)$ interpolates $f(x)$ at

$$x_i = a + \frac{i-1}{m-1}(b-a), \qquad i = 1{:}m.$$

If $m = 2$, then we obtain the *trapezoidal rule*:

$$\begin{aligned} Q_{\text{NC}(2)} &= \int_a^b \left(f(a) + \frac{f(b)-f(a)}{b-a}(x-a) \right) dx \\ &= (b-a)\left(\frac{1}{2}f(a) + \frac{1}{2}f(b) \right). \end{aligned}$$

If $m = 3$ and $c = (a+b)/2$, then we obtain the *Simpson rule*:

$$\begin{aligned} Q_{\text{NC}(3)} &= \int_a^b \left(f(a) + \frac{f(c)-f(a)}{c-a}(x-a) + \frac{\frac{f(b)-f(c)}{b-c} - \frac{f(c)-f(a)}{c-a}}{b-a}(x-a)(x-c) \right) dx \\ &= \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right). \end{aligned}$$

From these low-degree examples, it appears that a linear combination of f-evaluations is obtained upon expansion of the right-hand side in (4.3).

4.1.1 Derivation

For general m, we proceed by substituting the Newton representation

$$p_{m-1}(x) = \sum_{k=1}^{m} \left(c_k \prod_{i=1}^{k-1} (x - x_i) \right)$$

into (4.3):

$$Q_{\text{NC}(m)} = \int_a^b p_{m-1}(x)dx \; = \; \sum_{k=1}^{m} c_k \int_a^b \left(\prod_{i=1}^{k-1} (x - x_i) \right) dx.$$

If we set $x = a + sh$, where $h = (b - a)/(m - 1)$, then this transforms to

$$Q_{\text{NC}(m)} \; = \; \int_a^b p_{m-1}(x)dx \; = \; h\int_0^{m-1} p_{m-1}(a + sh)ds \; = \; \sum_{k=1}^{m} c_k h^k S_{mk},$$

where

$$S_{mk} \; = \; \int_0^{m-1} \left(\prod_{i=1}^{k-1} (s - i + 1) \right) ds.$$

The c_k are divided differences. Because of the equal spacing, they have a particularly simple form in terms of the f_i, as was shown in §2.4.1. For example,

$$\begin{aligned} c_1 &= f_1 \\ c_2 &= (f_2 - f_1)/h \\ c_3 &= (f_3 - 2f_2 + f_1)/(2h^2) \\ c_4 &= (f_4 - 3f_3 + 3f_2 - f_1)/(3!h^3). \end{aligned}$$

Recipes for the S_{mk} can also be derived. Here are a few examples:

$$\begin{aligned} S_{m1} &= \textstyle\int_0^{m-1} 1 \cdot ds &&= (m - 1) \\ S_{m2} &= \textstyle\int_0^{m-1} sds &&= (m - 1)^2/2 \\ S_{m3} &= \textstyle\int_0^{m-1} s(s - 1)ds &&= (m - 1)^2(m - 5/2)/3 \\ S_{m4} &= \textstyle\int_0^{m-1} s(s - 1)(s - 2)ds &&= (m - 1)^2(m - 3)^2/4 \end{aligned}$$

Using these tabulations we can readily derive the weights for any particular m-point rule. For example, if $m = 4$, then

$$S_{41} = 3 \quad S_{42} = 9/2 \quad S_{43} = 9/2 \quad S_{44} = 9/4.$$

Thus,

$$\begin{aligned} Q_{\text{NC}(4)} &= S_{41}c_1h + S_{42}c_2h^2 + S_{43}c_3h^3 + S_{44}c_4h^4 \\ &= 3f_1h + \frac{9}{2}\frac{f_2 - f_1}{h}h^2 + \frac{9}{2}\frac{f_3 - 2f_2 + f_1}{2h^2}h^3 + \frac{9}{4}\frac{f_4 - 3f_3 + 3f_2 - f_1}{6h^3}h^4 \\ &= \frac{3h}{8}(f_1 + 3f_2 + 3f_3 + f_4) \\ &= (b - a)(f_1 + 3f_2 + 3f_3 + f_4)/8 \end{aligned}$$

showing that [1 3 3 1]/8 is the weight vector for $Q_{\text{NC}(4)}$.

4.1.2 Implementation

For convenience in subsequent computations, we "package" the Newton-Cotes weight vectors in the following function:

```
  function w = NCWeights(m)
% w = NCWeights(m)
%
% w is a column m-vector consisting of the weights for the m-point Newton-Cotes rule.
% m is an integer that satisfies 2 <= m <= 11.

if m==2
   w=[1 1]'/2;
elseif m==3
   w=[1 4 1]'/6;
elseif m==4
   w=[1 3 3 1]'/8;
elseif m==5
   w=[7 32 12 32 7]'/90;
    :
end
```

Notice that the weight vectors are symmetric about their middle in that $w(1{:}m) = w(m{:}-1{:}1)$.

Turning now to the evaluation of $Q_{\text{NC}(m)}$ itself, we see from

$$Q_{\text{NC}(m)} = (b-a)\sum_{i=1}^{m} w_i f_i \;=\; (b-a)\left[\begin{array}{ccc} w_1 & \cdots & w_m \end{array}\right]\left[\begin{array}{c} f(x_1) \\ \vdots \\ f(x_m) \end{array}\right]$$

that it is a scaled inner product of the weight vector w and the vector of function values. Therefore, we obtain

```
  function numI = QNC(f,a,b,m)
% m-point Newton-Cotes quadrature across the interval [a b].
% f is a handle that points to a function of the form f(x) where x is a
% scalar. f must be defined on [a,b] and it must return a column vector if
% x is a column vector.
% m is an integer that satisfies 2 <= m <= 11.
% numI is the m-point Newton-Cotes approximation of the integral of f from
% a to b.
w = NCweights(m);
x = linspace(a,b,m)';
fvals = f(x);
numI = (b-a)*(w'*fvals);
```

We mention that $Q_{\text{NC}(2)}$ and $Q_{\text{NC}(3)}$ are referred to as the *trapezoidal rule* and *Simpson's rule* respectively.

Let us see how well `QNC` does when it is applied to the problems

$$I_1 \;=\; \int_0^1 e^{-x}dx \;=\; 1 - e^{-1}$$

and

$$I_2 \;=\; \int_0^1 e^{-20x}dx \;=\; (1 - e^{-20})/20$$

Setting `Q1 = QNC(@(x) exp(-x),0,1,m)` and `Q2 = QNC(@(x) exp(-20*x),0,1,m)` we find

```
 m       |Q1 - I1|            |Q2 - I2|
---------------------------------------------
 2   0.0518191617571635   0.4500000011336345
 3   0.0002131211751050   0.1166969337330916
 4   0.0000950324202655   0.0754778453850014
 5   0.0000003161797660   0.0301796546189490
 6   0.0000001782491539   0.0208012561376684
 7   0.0000000003894651   0.0080385105198381
 8   0.0000000002389524   0.0056365811921616
 9   0.0000000000003593   0.0019118765020265
10   0.0000000000002303   0.0013508599157407
11   0.0000000000000003   0.0003884845483225
```

We need a theory that explains why the results for I_2 are so inferior!

4.1.3 Newton-Cotes Error

How good are the Newton-Cotes rules? Since they are based on the integration of a polynomial interpolant, the answer clearly depends on the quality of the interpolant. Here is a result for Simpson's rule:

Theorem 4 *If $f(x)$ and its first four derivatives are continuous on $[a,b]$, then*

$$\left| \int_a^b f(x)dx - Q_{\mathrm{NC}(3)} \right| \le \frac{(b-a)^5}{2880} M_4,$$

where M_4 is an upper bound on $|f^{(4)}(x)|$ on $[a,b]$.

Proof Suppose

$$p(x) = c_1 + c_2(x-a) + c_3(x-a)(x-b) + c_4(x-a)(x-b)(x-c)$$

is the Newton form of the cubic interpolant to $f(x)$ at the points a, b, c, and d. If c is the midpoint of the interval $[a,b]$, then

$$\int_a^b (c_1 + c_2(x-a) + c_3(x-a)(x-b))\, dx = Q_{\mathrm{NC}(3)},$$

because the first three terms in the expression for $p(x)$ specify the quadratic interpolant of $(a, f(a))$, $(c, f(c))$, and $(b, f(b))$, on which the three-point Newton-Cotes rule is based. By symmetry we have

$$\int_a^b (x-a)(x-b)(x-c)dx = 0$$

and so

$$\int_a^b p(x)dx = Q_{\mathrm{NC}(3)}.$$

The error in $p(x)$ is given by Theorem 2,

$$f(x) - p(x) = \frac{f^{(4)}(\eta_x)}{24}(x-a)(x-b)(x-c)(x-d)$$

and thus,

$$\int_a^b f(x)dx - Q_{\mathrm{NC}(3)} = \int_a^b \left(\frac{f^{(4)}(\eta_x)}{24}(x-a)(x-b)(x-c)(x-d) \right) dx.$$

Taking absolute values, we obtain

$$\left| \int_a^b f(x)dx - Q_{\mathrm{NC}(3)} \right| \le \frac{M_4}{24} \int_a^b |(x-a)(x-b)(x-c)(x-d)|\, dx.$$

If we set $d = c$, then $(x-a)(x-b)(x-c)(x-d)$ is always negative and it is easy to verify that

$$\int_a^b |(x-a)(x-b)(x-c)(x-d)|\, dx = \frac{(b-a)^5}{120}$$

and so

$$\left| \int_a^b f(x)dx - Q_{\text{NC}(3)} \right| \le \frac{M_4}{24}\frac{(b-a)^5}{120} = \frac{M_4}{2880}(b-a)^5. \,\square$$

Note that if $f(x)$ is a cubic polynomial, then $f^{(4)} = 0$ and so Simpson's rule is exact. This is somewhat surprising because the rule is based on the integration of a *quadratic* interpolant.

In general, it can be shown that

$$\int_a^b f(x)dx \;=\; Q_{\text{NC}(m)} \;+\; c_m f^{(d+1)}(\eta)\left(\frac{b-a}{m-1}\right)^{d+2}, \tag{4.4}$$

where c_m is a small constant, η is in the interval $[a,b]$, and

$$d \;=\; \begin{cases} m-1 & \text{if } m \text{ is even} \\ m & \text{if } m \text{ is odd} \end{cases} .$$

Notice that if m is odd, as in Simpson's rule, then an extra degree of accuracy results. See P4.1.3 for details.

From (4.4), we see that knowledge of $f^{(d+1)}$ is required in order to say something about the error in $Q_{\text{NC}(m)}$. For example, if $|f^{(d+1)}(x)| \le M_{d+1}$ on $[a,b]$, then

$$\left| Q_{\text{NC}(m)} - \int_a^b f(x)dx \right| \;\le\; |c_m| M_{d+1}\left(\frac{b-a}{m-1}\right)^{d+2}. \tag{4.5}$$

The following function can be used to return this upper bound given the interval $[a,b]$, m, and the appropriate derivative bound:

```
  function error = QNCError(a,b,m,M)
% The error bound for the m-point Newton-Cotes rule when applied to
% the integral from a to b of a function f(x). It is assumed that
% a<=b and 2<=m<=11. M is an upper bound for the (d+1)-st derivative of the
% function f(x) on [a,b] where d = m if m is odd, and m-1 if m is even.
if     m==2,   d=1;  c = -1/12;
elseif m==3,   d=3;  c = -1/90;
elseif m==4,   d=3;  c = -3/80;
elseif m==5,   d=5;  c = -8/945;
elseif m==6,   d=5;  c = -275/12096;
elseif m==7,   d=7;  c = -9/1400;
elseif m==8,   d=7;  c = -8183/518400;
elseif m==9,   d=9;  c = -2368/467775;
elseif m==10,  d=9;  c = -173/14620;
else           d=11; c = -1346350/326918592;
end
error = abs( c*M*((b-a)/(m-1))^(d+2));
```

From this we see that if you are contemplating an even m rule, then the $(m-1)$-point rule is probably just as good and requires one less function evaluation. The following table summarizes the error when the m-point Newton-Cotes rule is applied to

$$I \;=\; \int_0^{\pi/2} \sin(x)dx.$$

m	`QNC(@sin,0,pi/2,m)`	Actual Error	Error Bound
2	0.7853981633974483	2.146e-01	3.230e-01
3	1.0022798774922104	2.280e-03	3.321e-03
4	1.0010049233142790	1.005e-03	1.476e-03
5	0.9999915654729927	8.435e-06	1.219e-05
6	0.9999952613861667	4.739e-06	6.867e-06
7	1.0000000258372355	2.584e-08	3.714e-08
8	1.0000000158229039	1.582e-08	2.277e-08
9	0.9999999999408976	5.910e-11	8.466e-11
10	0.9999999999621675	3.783e-11	5.417e-11
11	1.0000000000001021	1.021e-13	1.460e-13

Problems

P4.1.1 Let $C(x)$ be the cubic Hermite interpolant of $f(x)$ at $x = a$ and b. Show that

$$\int_a^b C(x)dx = \frac{h}{2}(f(a)+f(b)) + \frac{h^2}{12}(f'(a)-f'(b)).$$

This is sometimes called the *corrected trapezoidal rule*. Write a function `CorrTrap(f,fp,a,b)` that computes this value. Here, `f` and `fp` are handles that reference the integrand and its derivative respectively. The error in this rule has the form $ch^4 f^{(4)}(\eta)$. Determine c (approximately) through experimentation.

P4.1.2 This problem is about the computation of the closed Newton-Cotes weights by solving an appropriate linear system. Observe that the m-point rule should compute the integral

$$\int_0^1 x^{i-1}dx = \frac{1}{i}$$

exactly for $i = 1{:}m$. For this calculation, the abscissas are given by $x_j = (j-1)/(m-1)$, $i = 1{:}m$. Thus the weights $w_1, \ldots, w_m$ satisfy

$$w_1 x_1^{i-1} + w_2 x_2^{i-1} + \cdots + w_m x_m^{i-1} = \frac{1}{i}$$

for $i = 1{:}m$. This defines a linear system whose solution is the weight vector for the m-point rule. Write a function `MyNCweights(m)` that computes the weights by setting up the preceding linear system and solving for w using the backslash operation. Compare the output of `NCweights` and `MyNCweights` for $m = 2{:}11$.

P4.1.3 (a) Suppose m is odd and that $c = (a+b)/2$. Show that $Q_{\mathrm{NC}(m)}$ is exact if applied to

$$I = \int_a^b (x-c)^k dx$$

when k is odd. (b) If $p(x)$ has degree m, then it can be written in the form $p(x) = q(x) + \alpha(x-c)^m$ where q has degree $m-1$ and α is a scalar. Use this fact with $c = (a+b)/2$ to show that if m is odd, then $Q_{\mathrm{NC}(m)}$ is exact when applied to

$$I = \int_a^b p(x)dx.$$

P4.1.4 Augment `ShowQNCError` so that it also prints a table of errors and error bounds for the integral

$$I = \int_0^1 \frac{dx}{1+10x}.$$

Explain clearly the derivative bounds that are used.

4.2 Composite Rules

We will not be happy with the error bound (4.5) unless $b-a$ is sufficiently small. Fortunately, there is an easy way to organize the computation of an integral so that small-interval quadratures prevail.

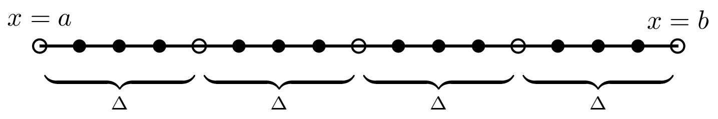


FIGURE 4.2 *Function evaluations in* $Q^{(4)}_{\text{NC}(5)}$

4.2.1 Derivation

If we have a partition

$$a = z_1 < z_2 < \cdots < z_{n+1} = b,$$

then

$$\int_a^b f(x)dx \;=\; \sum_{i=1}^{n} \int_{z_i}^{z_{i+1}} f(x)dx.$$

If we apply $Q_{\text{NC}(m)}$ to each of the subintegrals, then a *composite quadrature rule* based on $Q_{\text{NC}(m)}$ results. For example, if $\Delta_i = z_{i+1} - z_i$ and $z_{i+1/2} = (z_i + z_{i+1})/2$, $i = 1{:}n$, then

$$Q = \sum_{i=1}^{n} \frac{\Delta_i}{6}\left(f(z_i) + 4f(z_{i+1/2}) + f(z_{i+1})\right) \tag{4.6}$$

is a composite Simpson rule. In general, if `z` houses a partition of $[a, b]$ and `f` is a handle that references a function, then

```
numI=0
for i=1:length(z)-1
   numI = numI + QNC(@f,z(i),z(i+1),m);
end
```

assigns to `numI` the composite m-point Newton-Cotes estimate of the integral based on the partition housed in `z`.

In §4.4 we will show how to automate the choice of a good partition. In the remainder of this section, we focus on composite rules that are based on uniform partitions. In these rules, $n \geq 1$,

$$z_i = a + (i-1)\Delta, \qquad \Delta = \frac{b-a}{n}$$

for $i = 1{:}n+1$, and the composite rule evaluation has the form

```
numI = 0;
Delta=(b-a)/n;
for i=1:n
   numI = numI + QNC(@f,a+(i-1)*Delta,a+i*Delta,m);
end
```

We designate the estimate produced by this quadrature rule by $Q^{(n)}_{\text{NC}(m)}$. The computation is a little inefficient because it involves $n-1$ extra function evaluations and a `for`-loop. The rightmost f-evaluation in the ith call to `QNC` is the same as the leftmost f-evaluation in the $i+1$st call. Figure 4.2 depicts the situation in the four-subinterval, five-point rule case.

To avoid redundant f-evaluation and a `for`-loop with repeated function calls, it is better not to apply `QNC` to each of the n subintegrals. Instead, we precompute *all* the required function evaluations and store them in a single column vector `fval(1:n(m-1)+1)`. The linear combination that defines the composite rule is then calculated. In the preceding $Q^{(4)}_{\text{NC}(5)}$ example, the 17 required function evaluations are assembled in `fval(1:17)`. If w is the weight vector for $Q_{\text{NC}(5)}$, then

$$Q^{(4)}_{\text{NC}(5)} = \Delta\left(w^T fval(1{:}5) + w^T fval(5{:}9) + w^T fval(9{:}13) + w^T fval(13{:}17)\right).$$

From this we conclude that $Q^{(n)}_{\mathrm{NC}(m)}$ is a summation of n inner products, each of which involves the weight vector w of the underlying rule and a portion of the $fval$-vector. The following function is organized around this principle:

```
  function numI = CompQNC(f,a,b,m,n)
% Composite Newton-Cotes rule for the integral of f from a to b.
% f is a handle that points to a function of the form f(x) where x is a
% scalar. f must be defined on [a,b] and it must return a column vector if x is a
% column vector.
% m is an integer that satisfies 2 <= m <= 11.
% n is a positive integer.
% numI is the composite m-point Newton-Cotes approximation of the integral of f
% from a to b with n equal length subintervals.

w = NCweights(m);
x = linspace(a,b,n*(m-1)+1)';
f = f(x);
numI = 0; first = 1; last = m;
for i=1:n
   %Add in the inner product for the i-th subintegral.
   numI = numI + w'*f(first:last);
   first = last;
   last = last+m-1;
end
numI = Delta*numI;
```

4.2.2 Error

Let us examine the error. Suppose Q_i is the m-point Newton-Cotes estimate of the ith subintegral. If this rule is exact for polynomials of degree d, then using (4.4) we obtain

$$\int_a^b f(x)dx \;=\; \sum_{i=1}^n \int_{z_i}^{z_{i+1}} f(x)dx = \sum_{i=1}^n \left(Q_i + c_m f^{(d+1)}(\eta_i) \left(\frac{z_{i+1}-z_i}{m-1}\right)^{d+2}\right).$$

By definition

$$Q^{(n)}_{\mathrm{NC}(m)} = \sum_{i=1}^n Q_i$$

and

$$z_{i+1} - z_i = \Delta = \frac{b-a}{n}.$$

Moreover, it can be shown that

$$\frac{1}{n}\sum_{i=1}^n f^{(d+1)}(\eta_i) = f^{(d+1)}(\eta)$$

for some $\eta \in [a,b]$ and so

$$\int_a^b f(x)dx \;=\; Q^{(n)}_{\mathrm{NC}(m)} + c_m \left(\frac{b-a}{n(m-1)}\right)^{d+2} n f^{(d+1)}(\eta). \tag{4.7}$$

If $|f^{(d+1)}(x)| \le M_{d+1}$ for all $x \in [a,b]$, then

$$\left| Q^{(n)}_{\mathrm{NC}(m)} - \int_a^b f(x)dx \right| \;\le\; \left[|c_m| M_{d+1} \left(\frac{b-a}{m-1}\right)^{d+2}\right] \frac{1}{n^{d+1}}. \tag{4.8}$$

Comparing with (4.5), we see that the error in the composite rule is the error in the corresponding "simple" rule divided by n^{d+1}. Thus, with m fixed it is possible to exercise error control by choosing n sufficiently large. For example, suppose that we want to approximate the integral with a uniformly spaced composite Simpson rule so that the error is less than a prescribed tolerance *tol*. If we know that the fourth derivative of f is bounded by M_4, then we choose n so that

$$\frac{1}{90} M_4 \left(\frac{b-a}{2}\right)^5 \frac{1}{n^4} \le tol.$$

To keep the number of function evaluations as small as possible, n should be the smallest positive integer that satisfies

$$n \ge (b-a)\sqrt[4]{\frac{M_4(b-a)}{2880 \cdot tol}}.$$

The script file `ShowCompQNC` displays the error properties of the composite Newton-Cotes rules for three different integrands. (See Figure 4.3.)

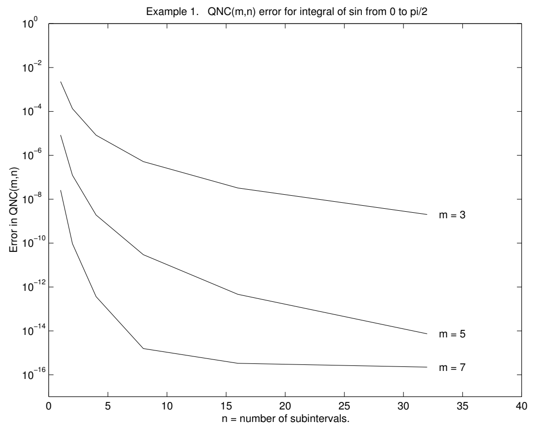


FIGURE 4.3 *Error in composite Newton-Cotes rules*

Problems

P4.2.1 Write a function `error = CompQNCerror(a,b,m,DerBound,n)` that returns an upper bound for the error in the uniformly spaced composite m-point Newton-Cotes quadrature rule applied to the integral of $f(x)$ from a to b. Use `errNC`.

P4.2.2 Rewrite `CompQNC` so that only one call to the integrand function is required.

P4.2.3 Write a function: `n = nBest(a,b,m,DerBound,tol)` that returns an integer n such that the error bound for $Q^{(n)}_{\mathrm{NC}(m)}$ is less than *tol*.

P4.2.4 Let $C(x)$ be the piecewise cubic Hermite interpolant of of $f(x)$ on $[a,b]$. Develop a uniformly spaced composite rule based on this interpolant.

P4.2.5 What can you say about the *approximate* value of T_2/T_1, where T_1 is the time required to compute a certain integral using a composite m-point Newton-Cotes rule with n subintervals, and T_2 is the time required to compute the same integral using a composite $2m$-point Newton-Cotes rule with $10n$ subintervals.

4.3 Adaptive Quadrature

Uniformly spaced composite rules that are exact for degree d polynomials are efficient if $f^{(d+1)}$ is uniformly behaved across $[a,b]$. However, if the magnitude of this derivative varies widely across the interval of integration, then the error control process discussed in §4.2 may result in an unnecessary number of function evaluations. This is because n is determined by an interval-wide derivative bound M_{d+1}. In regions where $f^{(d+1)}$ is small compared to this value, the subintervals are (possibly) much shorter than necessary. *Adaptive quadrature* methods address this problem by "discovering" where the integrand is ill behaved and shortening the subintervals accordingly.

4.3.1 An Adaptive Newton-Cotes Procedure

To obtain a good partition of $[a, b]$, we need to be able to estimate error. That way the partition can be refined if the error is not small enough. One idea is to use two different quadrature rules. The difference between the two predicted values of the integral could be taken as a measure of their inaccuracy:

```
function numI = AdaptQNC(f,a,b,...)
   Compute the integral from a to b in two ways.  Call the values $A_1$ and $A_2$
      and assume that $A_2$ is better.
   Estimate the error in $A_2$ based on $|A_1 - A_2|$.
   If the error is sufficiently small, then
      numI = $A_2$;
   else
      mid = (a+b)/2;
      numI = AdaptQNC(f,a,mid,...)  + AdaptQNC(f,mid,b,...);
   end
```

This divide-and-conquer framework is similar to the one we developed for adaptive piecewise linear approximation.

The filling in of the details begins with the development of a method for estimating the error. Fix m and set $A_1 = Q^{(1)}_{\mathrm{NC}(m)}$ and $A_2 = Q^{(2)}_{\mathrm{NC}(m)}$. Thus A_1 is the "simple" m-point rule estimate and A_2 is the two-interval, m-point rule estimate. If these rules are exact for degree d polynomials, then it can be shown that

$$I \;=\; A_1 + \left[c_m f^{(d+1)}(\eta_1) \left(\frac{b-a}{m-1} \right)^{d+2} \right] \tag{4.9}$$

$$I \;=\; A_2 + \left[c_m f^{(d+1)}(\eta_2) \left(\frac{b-a}{m-1} \right)^{d+2} \right] \frac{1}{2^{d+1}} \tag{4.10}$$

where η_1 and η_2 are in the interval $[a, b]$. We now make the assumption $f^{(d+1)}(\eta_1) = f^{(d+1)}(\eta_2)$. This is reasonable if $f^{(d+1)}$ does not vary much on $[a, b]$. (The shorter the interval, the more likely this is to be the case.) Thus

$$I = A_1 + C$$

and

$$I = A_2 + C/2^{d+1},$$

where

$$C = \left[c_m f^{(d+1)}(\eta_1) \left(\frac{b-a}{m-1} \right)^{d+2} \right].$$

By subtracting these two equations for I from each other and solving for C, we get

$$C = \frac{A_2 - A_1}{1 - \frac{1}{2^{d+1}}}$$

and so

$$|I - A_2| \;\approx\; \frac{|A_1 - A_2|}{2^{d+1} - 1}.$$

Thus, the discrepancy between the two estimates divided by $2^{d+1} - 1$ provides a reasonable estimate of the error in A_2. If our goal is to produce an estimate of I that has absolute error *tol* or less, then the recursion may be organized as follows:

```
  function numI = AdaptQNC(f,a,b,m,tol)
% f is a handle that points to a function of the form f(x) where x is a
% scalar. f must be defined on [a,b] and it must return a column vector if x is a
% column vector.
% a,b are real scalars, m is an integer that satisfies 2 <= m <=11, and
% tol is a positive real.
% numI is a composite m-point Newton-Cotes approximation of the
% integral of f(x) from a to b, where the subinterval partition is
% determined adaptively.

% Estimates based on composite rule with 1 and 2 subintervals...
A1 = CompQNC(f,a,b,m,1);
A2 = CompQNC(f,a,b,m,2);
% The error estimate...
d = 2*floor((m-1)/2)+1;
error = (A2-A1)/(2^(d+1)-1);
% Accept of reject A2?
if abs(error) <= tol
   % A2 is acceptable
   numI = A2+error;
else
   % Subdivide the problem...
   mid = (a+b)/2;
   numI = AdaptQNC(f,a,mid,m,tol/2) + AdaptQNC(f,mid,b,m,tol/2);
end
```

If the heuristic estimate of the error is greater than *tol*, then two recursive calls are initiated to obtain estimates

$$Q_L \approx \int_a^{mid} f(x)dx = I_L$$

and

$$Q_R \approx \int_{mid}^{b} f(x)dx = I_R$$

that satisfy

$$|I_L - Q_L| \le tol/2$$

and

$$|I_R - Q_R| \le tol/2.$$

Setting $Q = Q_L + Q_R$, we see that

$$|I - Q| = |(I_L - Q_L) + (I_R - Q_R)| \le |I_L - Q_L| + |I_R - Q_R| \le (tol/2) + (tol/2) = tol.$$

Insight into the economies that are realized by the adaptive framework can be obtained by applying `AdaptQNC` to the integral of the built-in function

$$\texttt{humps}(x) \;=\; \frac{1}{0.01 + (x - 0.3)^2} \;+\; \frac{1}{0.04 + (x - 0.9)^2} \;-\; 6$$

from 0 to 1. The tables in FIGURE 4.4 and FIGURE 4.5 report on the number of required function evaluations associated with the call `AdaptQNC(@humps,0,1,m,tol)` for various choices of `m` and `tol`. These values would be much higher if we used `CompQNC(f,a,b,m,n)` to attain the same level of accuracy. This is because higher derivatives of `humps` are modest in size except near $x = .3$ and $x = .9$. To handle these "rough spots" we would need a large number of subintervals, i.e., a large value for `n` in the call to `CompQNC`.

	m = 3	m = 5	m = 7	m = 9
tol = .01	26	14	6	2
tol = .001	54	22	6	2
tol = .0001	94	30	14	10
tol = .00001	174	46	26	14

FIGURE 4.4 *Number of Scalar f-evaluations required by* `QNC(@humps,0,1,m,tol)`

	m = 3	m = 5	m = 7	m = 9
tol = .01	104	98	60	26
tol = .001	216	154	60	26
tol = .0001	376	210	140	130
tol = .00001	696	322	260	182

FIGURE 4.5 *Number of Vector f-evaluations required by* `QNC(@humps,0,1,m,tol)`

Problems

P4.3.1 The one-panel midpoint rule Q_1 for the integral

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

is defined by

$$Q_1 \;=\; (b-a)f\left(\frac{a+b}{2}\right).$$

The two-panel midpoint rule Q_2 for I is given by

$$Q_2 \;=\; \frac{b-a}{2}\left(f\left(\frac{3a+b}{4}\right)+f\left(\frac{a+3b}{4}\right)\right).$$

Using the heuristic $|I-Q_2| \le |Q_2-Q_1|$, write an efficient MATLAB adaptive quadrature routine of the form `Adapt(f,a,b,tol,...)` that returns an estimate of I that is accurate to within the tolerance given by *tol*. You may extend the parameter list, and you may use `nargin` as required. You may ignore the possibility of infinite recursion.

P4.3.2 A number of efficiency improvements can be made to `AdaptQNC`. A casual glance at `AdaptQNC` reveals two sources of redundant function evaluations: First, each function evaluation required in the assignment to `A1` is also required in the assignment to `A2`. Second, the recursive calls could (but do not) make use of previous function evaluations. In addressing these deficiencies, you are to follow these ground rules:

- A call of the form `AdaptQNC1(@f,a,b,m,tol)` must produce the same value as a call of the form `AdaptQNC(@f,a,b,m,tol)`.
- No global variables are allowed.

To "transmit" appropriate function values in the recursive calls, you will want to design `AdaptQNC1` so that it has an "optional" sixth argument `fValues`. By making this argument optional, the same five-parameter calls at the top level are permitted.

P4.3.3 An implementation `y = MyF(x)` of the function $f(x)$ has the property that it returns $f(x_i)$ in y_i for $i = 1{:}n$ where n is the length of x. Assume that the cost of a `MyF` evaluation is constant and independent of the length of the input vector `x`. We want to compute

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

with specified accuracy. Explain why it might be more efficient to use a composite trapezoidal rule with uniform length subintervals than an adaptive trapezoidal rule *if* we have information about the second derivative of f.

P4.3.4 Assume that `MyF` is a given implementation of the function $f(x)$ and that f has positive period T. Write an efficient MATLAB script for computing the integral

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

with absolute error $\le 10^{-6}$. Assume that a and b are given and make effective use of the Matlab quadrature function `quad`. The absolute error is no bigger than `tol`.

P4.3.5 Let Q_n be the equal spacing composite trapezoidal rule:

$$Q_n = h\left(\frac{1}{2}f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{1}{2}f(x_n)\right) \qquad h = \frac{b-a}{n-1},$$

where $x = \mathit{linspace}(a, b, n)$ and we assume that $n \geq 2$. Assume that there is a constant C (independent of n), such that

$$I = \int_a^b f(x)fx = Q_n + Ch^2.$$

(a) Give an expression for $|I - Q_{2n}|$ in terms of $|Q_{2n} - Q_n|$. (b) Write an *efficient* function `Q = TrapRecur(f,a,b,tol` that returns in `Q` the value of Q_{2^k+1}, where k is the smallest positive integer so that $|I - Q_{2^k+1}|$ is smaller than the given positive tolerance `tol`.

P4.3.6 Assume that the function $f(x)$ is available and define

$$\phi(z) = \int_{-z}^{z} f(x)dx.$$

Using `quad`, show how to compute an array `phiVals(1:100)` with the property that $\phi(k)$ is assigned to `phiVals(k)` for `k=1:100`.

P4.3.7 Give a solution procedure for computing

$$I = \int_a^b \left(\int_a^x f(x,y)dy\right) dx,$$

where `f(x,y)` is a given. All integrals in your method must be computed using `quad`. Clearly define the functions that are required by your method. Note: The built-in MATLAB function `dblquad` can be used to evaluate double integrals of the form

$$I = \int_a^b \int_c^d f(x,y)dxdy,$$

but this does not help in this problem.

4.4 Gauss Quadrature and Spline Quadrature

We discuss two other approaches to the quadrature problem. Gauss quadrature rules are of great interest because they optimize accuracy for a given number of f-evaluations. They also have merit in certain problems where the integrand has singularities. In situations where the function evaluations are experimentally determined, spline quadrature has a certain appeal.

4.4.1 Gauss Quadrature

In the Newton-Cotes framework, the integrand is sampled at regular intervals across $[a, b]$. In the *Gauss quadrature* framework, the abscissas are positioned in such a way that the rule is correct for polynomials of maximal degree.

A simple example clarifies the main idea. Let us try to determine weights w_1 and w_2 and abscissas x_1 and x_2 so that

$$w_1 f(x_1) + w_2 f(x_2) = \int_{-1}^{1} f(x)dx$$

for polynomials of degree 3 or less. This is plausible since there are four parameters to choose (w_1, w_2, x_1, x_2) and four constraints obtained by forcing the rule to be exact for the functions 1, x, x^2, and x^3:

$$\begin{aligned} w_1 + w_2 &= 2 \\ w_1x_1 + w_2x_2 &= 0 \\ w_1x_1^2 + w_2x_2^2 &= 2/3 \\ w_1x_1^3 + w_2x_2^3 &= 0 \end{aligned}$$

By multiplying the second equation by x_1^2 and subtracting it from the fourth equation we get $w_2x_2(x_1^2 - x_2^2) = 0$, and so $x_2 = -x_1$. It follows from the second equation that $w_1 = w_2$ and thus, from the first equation, $w_1 = w_2 = 1$. From the third equation, $x_1^2 = 1/3$ and so $x_1 = -1/\sqrt{3}$ and $x_2 = 1/\sqrt{3}$. Thus, for any $f(x)$ we have

$$\int_{-1}^{1} f(x)dx \;\approx\; f(-1/\sqrt{3}) + f(1/\sqrt{3}).$$

This is the two-point *Gauss-Legendre* rule.

The m-point Gauss-Legendre rule has the form

$$Q_{\mathrm{GL}(m)} = w_1 f(x_1) + \cdots + w_m f(x_m),$$

where the w_i and x_i are chosen to make the rule exact for polynomials of degree $2m-1$. One way to define these $2m$ parameters is by the $2m$ *nonlinear* equations

$$w_1 x_1^k + w_2 x_2^k + \cdots + w_m x_m^k = \frac{1-(-1)^{k+1}}{k+1}, \qquad k = 0{:}2m-1.$$

The kth equation is the requirement that the rule

$$w_1 f(x_1) + \cdots + w_m f(x_m) \;=\; \int_{-1}^{1} f(x)dx$$

be exact for $f(x) = x^k$. It turns out that this system has a unique solution, which we encapsulate in the following function for the cases $m = 2{:}6$:

```
  function [w,x] = GLweights(m)
% [w,x] = GLWeights(m)
% w is a column m-vector consisting of the weights for the m-point Gauss-Legendre rule.
% x is a column m-vector consisting of the abscissae.
% m is an integer that satisfies 2 <= m <= 6.
w = ones(m,1);
x = ones(m,1);
if m==2
   w(1) =  1.000000000000000; w(2) =  w(1);
   x(1) = -0.577350269189626; x(2) = -x(1);
elseif m==3
    :
end
```

The Gauss-Legendre rules

$$Q_{\mathrm{GL}(m)} = w_1 f(x_1) + \cdots + w_m f(x_m) \;\approx\; \int_{-1}^{1} f(x)dx$$

are not restrictive even though they pertain to integrals from -1 to 1. By a change of variable, we have

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} g(x)dx,$$

where

$$g(x) = f\left(\frac{a+b}{2} + \frac{b-a}{2}x\right),$$

and so

$$\frac{b-a}{2}\left(w_1 f\left(\frac{a+b}{2} + \frac{b-a}{2}x_1\right) \;+\cdots+\; w_m f\left(\frac{a+b}{2} + \frac{b-a}{2}x_m\right)\right) \;\approx\; \int_a^b f(x)dx.$$

This gives

```
  function numI = QGL(f,a,b,m)
% f is a handle that references a function of the form f(x) that
% is defined on [a,b]. f should return a column vector if x is a column vector.
% a,b are real scalars.
% m is an integer that satisfies 2 <= m <= 6.
% numI is the m-point Gauss-Legendre approximation of the
% integral of f(x) from a to b.
[w,x] = GLWeights(m);
fvals = f((b-a)/2)*x + ((a+b)/2)*ones(m,1));
numI = ((b-a)/2)*w'*fvals;
```

It can be shown that

$$\left| \int_a^b f(x)dx - Q_{\mathrm{GL}(m)} \right| \;\le\; \frac{(b-a)^{2m+1}(m!)^4}{(2m+1)[(2m)!]^3} M_{2m},$$

where M_{2m} is a constant that bounds $|f^{2m}(x)|$ on $[a,b]$. The script file GLvsNC compares the $Q_{\mathrm{NC}(m)}$ and $Q_{\mathrm{GL}(m)}$ rules when they are applied to the integral of $\sin(x)$ from 0 to $\pi/2$:

```
   m          NC(m)                     GL(m)
------------------------------------------------------
   2    0.7853981633974483       0.9984726134041148
   3    1.0022798774922104       1.0000081215555008
   4    1.0010049233142790       0.9999999771971151
   5    0.9999915654729927       1.0000000000395670
   6    0.9999952613861668       0.9999999999999533
```

Notice that for this particularly easy problem, $Q_{\mathrm{GL}(m)}$ has approximately the accuracy of $Q_{\mathrm{NC}(2m)}$.

It is possible to formulate an adaptive quadrature procedure that is based on a Gauss-Legendre rule. However, the "weird" location of the abscissae creates a problem. The f-evaluations that are required when we apply an m-point rule across $[a,b]$ are not shared by the m-point rules applied to the half-interval problems. The *Gauss-Kronrod* framework circumvents this problem. The basic idea is to work with a pair of rules that share f-evaluations. The (15,7) Gauss-Kronrod procedure, works with a 15-point rule

$$\int_{-1}^{1} f(x)dx \;\approx\; Q_{\mathrm{GK}(15)} \;=\; \sum_{k=1}^{15} \omega_k^{(15)} f(x_k^{(15)})$$

and a 7-point rule,

$$\int_{-1}^{1} f(x)dx \;\approx\; Q_{\mathrm{GK}(7)} \;=\; \sum_{k=1}^{7} \omega_k^{(7)} f(x_k^{(7)}).$$

The key connection between $x^{(15)}$ and $x^{(7)}$ is this:

$$x^{(7)} \;=\; x^{(15)}(2{:}2{:}15).$$

See FIGURE 4.x. Moreover, there is a heuristic argument that says

$$\left| \int_{-1}^{1} f(x)dx - Q_{\mathrm{GK}(15)} \right| \;\approx\; 200|Q_{\mathrm{GK}(15)} \;-\; Q_{\mathrm{GK}(7)}|^{1.5}. \qquad (4.11)$$

The demo function ShowGK affirms this result.

One can formulate an adaptive procedure based on these two rules that use these facts. We compute $Q_{\mathrm{GK}(15)}$ and get $Q_{\mathrm{GK}(7)}$ "for free" because of the shared f-evaluations. If the discrepancy between the two rules is too large, then we subdivide the problem and repeat the process on each half-interval. The MATLAB procedure quadgk is based on this idea.

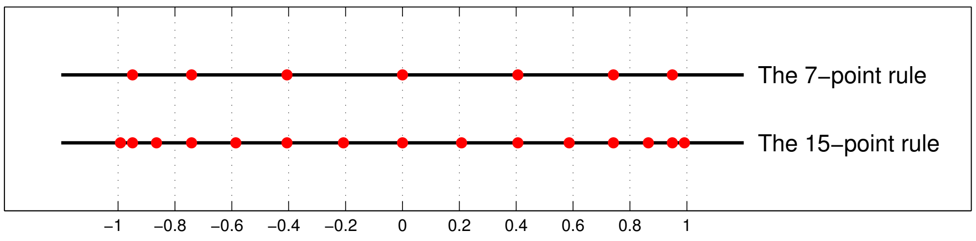


FIGURE 4.6 *Abscissa location for the (15,7) Gauss-Kronrod Pair*

Problems

P4.4.1 If $Q_{\mathrm{GL}(m)}$ is the m-point Gauss-Legendre estimate for

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

then it can be shown that

$$|I - Q_{\mathrm{GL}(m)}| \;\le\; \frac{(b-a)^{2m+1}(m!)^4}{(2m+1)[(2m)!]^3} M_{2m} \;\equiv\; E_m,$$

where the constant M_{2m} satisfies $|f^{(2m)}(x)| \le M_{2m}$ for all $x \in [a,b]$. The following questions apply to the case when $f(x) = e^{cx}$, where $c > 0$. Assume that $a < b$. (a) Give a good choice for M_{2m}. (b) Give an expression for E_{m+1}/E_m. (c) Write a MATLAB script that determines the smallest positive integer m so that E_m is less than `tol`.

P4.4.2 Write a function `numI = CompQGL(f,a,b,m,n)` that approximates the integral of a function from `a` to `b` by applying the m-point Gauss-Legendre rule on n equal-length subintervals of $[a,b]$.

P4.4.3 Develop an addaptive quadrature procedure `numI = AdapkGK(f,a,b,tol)` that is based on the (15,7) Gauss-Kronrod pair and the error heuristic (4.11).

4.4.2 Spline Quadrature

Suppose $S(x)$ is a cubic spline interpolant of (x_i, y_i), $i = 1{:}n$ and that we wish to compute

$$I = \int_{x_1}^{x_n} S(x)dx.$$

If the ith local cubic is represented by

$$q_i(x) = \rho_{i4} + \rho_{i,3}(x - x_i) + \rho_{i,2}(x - x_i)^2 + \rho_{i,1}(x - x_i)^3,$$

then

$$\int_{x_i}^{x_{i+1}} q_i(x)dx = \rho_{i,4}h_i + \frac{\rho_{i,3}}{2}h_i^2 + \frac{\rho_{i,2}}{3}h_i^3 + \frac{\rho_{i,1}}{4}h_i^4,$$

where $h_i = x_{i+1} - x_i$. By summing these quantities from $i = 1{:}n-1$, we obtain the sought-after spline integral:

```
  function numI = SplineQ(x,y)
% Integrates the spline interpolant of the data specified by the
% column n-vectors x and y. It is a assumed that x(1) < ... < x(n)
% and that the spline is produced by the Matlab function spline.
% The integral is from x(1) to x(n).
```

```
S = spline(x,y);
[x,rho,L,k] = unmkpp(S);
sum = 0;
for i=1:L
   % Add in the integral from x(i) to x(i+1).
   h = x(i+1)-x(i);
   subI = h*(((rho(i,1)*h/4 + rho(i,2)/3)*h + rho(i,3)/2)*h + rho(i,4));
   sum = sum + subI;
end
numI = sum;
```

The script file `ShowSplineQ` uses this function to produce the following estimates for the integral of sine from 0 to $\pi/2$:

```
   m     Spline Quadrature
----------------------------
   5    1.0001345849741938
  50    0.9999999990552404
 500    0.9999999999998678
```

Here, the spline interpolates the sine function at `x = linspace(0,pi/2,m)`.

Problems

P4.4.4 Modify `SplineQ` so that a four-argument call `SplineQ(x,y,a,b)` returns the integral of the spline interpolant from a to b. Assume that $x_1 \le a \le b \le x_n$.

P4.4.5 Let $a(t)$ denote the acceleration of an object at time t. If v_0 is the object's velocity at $t = 0$, then the velocity at time t is prescribed by

$$v(t) = v_0 + \int_0^t a(\tau)d\tau.$$

Likewise, if x_0 is the position at $t = 0$, then the position at time t is given by

$$x(t) = x_0 + \int_0^t v(\tau)d\tau.$$

Now suppose that we have snapshots $a(t_i)$ of the acceleration at times t_i, $i = 1{:}m$, $t_1 = 0$. Assume that we know the initial position x_0 and velocity v_0. Our goal is to estimate position from this data. Spline quadrature will be used to approximate the preceding integrals. Let $S_a(t)$ be the not-a-knot spline interpolant of the acceleration data $(t_i, a(t_i))$, $i = 1{:}m$, and define

$$\tilde{v}(t) = v_0 + \int_0^t S_a(\tau)d\tau.$$

Let $S_v(t)$ be the not-a-knot spline interpolant of the data $(t_i, \tilde{v}(t_i))$, $i = 1{:}m$, and define

$$\tilde{x}(t) = x_0 + \int_0^t S_v(\tau)d\tau.$$

The spline interpolant $S_x(t)$ of the data $(t_i, \tilde{x}(t_i))$ is then an approximation of the true position. Write a function

```
  function Sx = PosVel(a,t,x0,v0)}
%
% t is an m-vector of equally spaced time values with t(1) = 0, m>=2.
% a is an m-vector of accelerations, a(i) = acceleration at time t(i).
% x0 and v0 are the  position and velocity  at t=0
%
% Sx the pp-representation of a spline that approximates position.
```

Try it out on the data `t = linspace(0,50,500)`, with $a(t) = 10e^{-t/25}\sin(t)$. However, before you turn the `a` vector over to `PosVel`, contaminate it with noise: `a = a + .01*randn(size(a))`. Produce a plot of the exact and estimated positions across [0,50] and a separate plot of $x(t) - S_x(t)$ across $[0, 50]$. Also print the value of $S_x(t)$ at $t = 50$. Repeat with $m = 50$ instead of 500. Use the MATLAB `spline` function.

P4.4.6 Assume that we have a vectorized implementation `f.m` of a positive-valued function $f(x)$ and that `x` is a given column n-vector with $x_1 < ... < x_n$. (a) Write a MATLAB fragment that sets up a column n-vector `q` with the property that

$$\left| q_i - \int_{x_1}^{x_i} f(x)dx \right| \leq \texttt{tol}$$

for $i = 1{:}n$. Assume that `tol` is a given positive tolerance. Make effective use of `quad`. (By setting the relative error tolerance to zero, `quad` will return an approximation of the integral that satisfies the absolute error tolerance.) (b) Assume that the array `q` has been successfully computed in (a). Making effective use of `spline`, `ppval`, and the idea of inverse interpolation, show how to estimate x_* so that

$$\int_{x_1}^{x_*} f(x) = \frac{1}{2}\int_{x_1}^{x_n} f(x)dx.$$

P4.4.7 Let $(x_1, y_1), \ldots, (x_n, y_n)$ be given points in the plane. Let d_i be the straight-line distance between (x_i, y_i) and (x_{i+1}, y_{i+1}), $i = 1{:}n-1$. Set $t_i = d_1 + \cdots + d_{i-1}$, $i = 1{:}n$. Suppose $S_x(t)$ is a spline interpolant of $(t_1, x_1), \ldots, (t_n, x_n)$ and that $S_y(t)$ is a spline interpolant of $(t_1, y_1), \ldots, (t_n, y_n)$. It follows that the curve $\Lambda = \{(S_x(t), S_y(t)) : t_1 \leq t \leq t_n\}$ is smooth and passes through the n points. Write a MATLAB function `[Sx,Sy,L] = Arc(x,y)` that returns the two splines interpolants (in pp-form) and the length of Λ, i.e.,

$$L = \int_{t_1}^{t_n} \sqrt{[S_x'(t)]^2 + [S_y'(t)]^2}dt.$$

Use `quad` for the integral with the default tolerance. You will have to set up an integrand function that accesses the piecewise quadratic functions $S_x'(t)$ and $S_y'(t)$. Write a script that displays the curve Λ where the input points are prescribed by

```
  x = [ 3  2  1  2  4  5  4  3  2  4  5  5  3];
  y = [ 7  6  5  4  3  2  1  1  2  4  5  6  7];
```

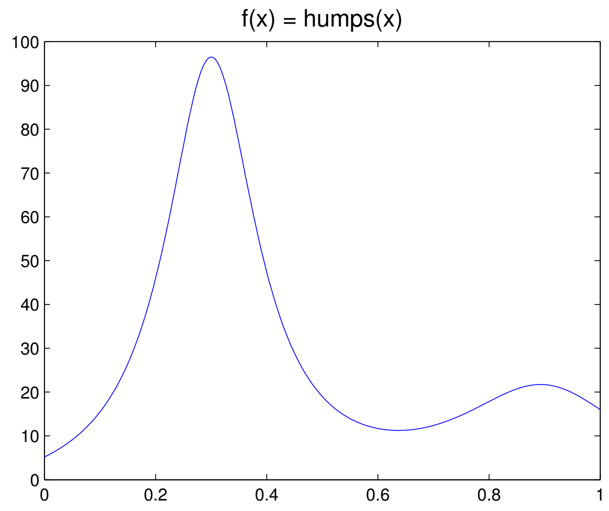
Print the curve length in the title of the plot.

4.5 Matlab's Quadrature Tools

Consider the function $f(x) = \texttt{humps}(x)$ where `humps` is the built-in MATLAB function

$$\texttt{humps}(x) = \frac{1}{(x-.3)^2 + .01} + \frac{1}{(x-.9)^2+.04} - 6.$$

This function's higher derivatives are large near $x = .3$ and $x = .9$:



The function `quad` can be used to approximate the integral of this function from 0 to 1:

```
>>    Q = quad(@humps,0,1)
      Q = 29.858326128427638
```

The number of f-evaluations can be obtained by supplying a second output parameter:

```
>>   [Q,fevals] = quad(@humps,0,1)
     Q = 29.858326128427638
     fevals = 145
```

Unless it is told otherwise, `quad` aims to compute the required integral with absolute error bounded by .000001. The error tolerance can be modified:

```
>>  [Q,fevals] = quad(@humps,0,1,10^-12)
    Q = 29.858325395498067
    fevals = 2321
```

The function `quad` implements an adaptive version of the *composite Simpson rule.* See §4.4. If high accuracy is required, then it is sometimes more economical to use the MATLAB quadrature function `quadl`:

```
>>  [Q,fevals] = quadl(@humps,0,1,10^-12)
    Q = 29.858325395498671
    fevals = 1608
```

The function `ShowQUADs(f,a,b)` approximates $I(f,a,b)$ and can be used to experiment with these two quadrature procedures for various choices of error tolerance. `ShowQUADs(@sin,0,pi)` tells us that `quadl` is to be preferred for very smooth integrands like $f(x) = \sin(x)$:

```
                        quad                          quadl
   tol      Approximation   f-evals      Approximation   f-evals
-----------------------------------------------------------------
1.0e-003    1.999993496535     13        1.999999977471     18
1.0e-006    1.999999996398     33        1.999999977471     18
1.0e-009    1.999999999999    129        2.000000000000     48
1.0e-012    2.000000000000    497        2.000000000000     48
```

On the other hand, `ShowQuads(@(x) sin(1./x),.01,1)` reveals that for nasty integrands like $\sin(1/x)$ it is better to use a low-order rule like `quad`, especially for modest tolerances:

```
                       quad                          quadl
   tol      Approximation   f-evals      Approximation   f-evals
-----------------------------------------------------------------
1.0e-003    0.463673444706     25        0.504011796906    138
1.0e-006    0.504041285733    237        0.503981893171    558
1.0e-009    0.503981892714    981        0.503981893175   1338
1.0e-012    0.503981893175   3985        0.503981893175   3648
```

The function `quadgk` offers greater control over error (absolute or relative) and can report back an estimate of the error if required. A call of the form

```
[Q,est] = quadgk(@f,a,b,'AbsTol',tol1,'RelTol',tol2)
```

attempts to return a value in `Q` that satisfies

$$|I(f,a,b) - \texttt{Q}| \;\le\; \max\{\texttt{AbsTol},\texttt{RelTol}\}.$$

If relative error is critical, then set `tol1`=0. If absolute error is the concern, set `tol2` = 0. In either case, the estimate returned in `est` is an estimate of the absolute error. If `quadgk` spots a problem with its error control, then it may suggest an increase in the value of `MaxIntervalCount` which permits the procedure to get a more accurate answer by evaluating f and more points. In this case you can try again with a response of t he form

```
[Q,est] = quadgk(@f,a,b,'AbsTol',tol1,'RelTol',tol2,'MaxIntervalCount',BiggerValue)
```

Here are some results when `quadgk` is used to compute

$$I = \int_0^1 100\sin\left(\frac{1}{x}\right)dx$$

with `BiggerValue = 100000`:

```
  Result Via       Error          AbsTol  RelTol
    quadgk        Estimate
------------------------------------------------
  50.40654795    0.00061442      0.0010  0.0000
  50.40465490    0.03100950      0.0000  0.0010
  50.40670252    0.00007224      0.0001  0.0000
  50.40658290    0.00430233      0.0000  0.0001
```

In some cases, `quadgk` can handle endpoint singularities. For example,

```
Q = quadgk(@(x) 1./sqrt(x),0,1)
Q = 1.999999999999763
```

The procedure can also accommodate infinite endpoints as in

$$I \;=\; \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{+\infty} e^{-(x-\mu)^2/(2\sigma^2)}\,dx.$$

Thus,

```
mu = 1;
sigma = 3;
Q = quadgk(@(x) exp(-((x-mu).^2/(2*sigma^2)))/(sigma*sqrt(2*pi)),-inf,inf)
Q = 1.000000000146726
```

thereby affirming that the area under the normal distribution $N(\mu,\sigma)$ equals one.

Problems

P4.5.1 Consider the function

$$I(\alpha) \;=\; (2+\sin(10\alpha))\int_0^2 x^\alpha \sin\left(\frac{\alpha}{2-x}\right)dx$$

Write a script that confirms the fact that

$$\max_{0\le\alpha\le5} \; I(\alpha) \;=\; I(.7859336743...)$$

Make effective use of MATLAB's quadrature software.

P4.5.2 It turns out that

$$\lim_{\epsilon\to 0}\int_\epsilon^1 \frac{1}{x}\cdot\cos\left(\frac{\ln(x)}{x}\right)dx \;=\; .3233674316...$$

Write the most efficient script you can that confirms this result. Make effective use of MATLAB's quadrature software.

Script Files

`ShowNCError`	Illustrates `NCerror`.
`ShowCompQNC`	Illustrates `CompQNC` on three examples.
`ShowAdapts`	Illustrates `AdaptQNC`.
`GLvsNC`	Compares Gauss-Legendre and Newton-Cotes rules.
`ShowSplineQ`	Illustrates `SplineQ`.
`ShowGK`	Illustrates the (15,7) Gauss-Kronrod rule.

Function Files

`ShowQuads`	Illustrates `quad`, `quadl`, and `quadgk`.
`ShowNCIdea`	Displays the idea behind the Newton-Cotes rules.
`NCWeights`	Constructs the Newton-Cotes weight vector.
`QNC`	The simple Newton-Cotes rule.
`NCError`	Error in the simple Newton-Cotes rule.
`CompQNC`	Equally-spaced, composite Newton-Cotes rule.
`AdaptQNC`	Adaptive Newton-Cotes quadrature.
`SpecHumps`	The humps function with function call counters.
`GLWeights`	Constructs the Gauss-Legendre weight vector.
`QGL`	The simple Gauss-Legendre rule.
`SplineQ`	Spline quadrature.

References

P. Davis and P. Rabinowitz (1984). *Methods of Numerical Integration, 2nd Ed.*, Academic Press, New York.

G.H. Golub and J.M. Ortega (1993). *Scientific Computing: An Introduction with Parallel Computing,* Academic Press, Boston.

A. Stroud (1972). *Approximate Calculation of Multiple Integrals*, Prentice Hall, Englewood Cliffs, NJ.