

1. Newton-Cotes Quadrature

An entire class of these formulas can be derived by constructing interpolating polynomials of various degrees N and integrating the polynomials to get an approximation to the integral of the function. These are called Newton-Cotes formulas. If we incorporate the endpoints as nodes $a = x_1, b = x_n$, we call them closed Newton Cotes formulas.

In general, an n -point Newton-Cotes method is a quadrature rule that assumes there will be n nodes evenly distributed over the interval $[a,b]$. In Matlab if we want 6 nodes that are evenly distributed then we use `linspace(a,b,6)`.

Since the nodes are specified, all that remains is to find the weights. For simplicity in determining the weights we use the interval $[-1,1]$. (The weights will be the same regardless of the interval that is chosen).

To determine the weights, the Newton-Cotes method requires that an n -point method be exact in calculating

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

when $f(x)$ is a polynomial of up to order $n - 1$ (including constant functions this will give n equations):

$$\begin{aligned} f(x) = 1 & \rightarrow a_1 + a_2 + \dots + a_n & = \int_{-1}^1 1 dx = 2 \\ f(x) = x & \rightarrow a_1 x_1 + a_2 x_2 + \dots + a_n x_n & = \int_{-1}^1 x dx = 0 \\ f(x) = x^2 & \rightarrow a_1 x_1^2 + a_2 x_2^2 + \dots + a_n x_n^2 & = \int_{-1}^1 x^2 dx = \frac{2}{3} \\ & \vdots & & \vdots \\ f(x) = x^{n-1} & \rightarrow a_1 x_1^{n-1} + a_2 x_2^{n-1} + \dots + a_n x_n^{n-1} & = \int_{-1}^1 x^{n-1} dx \end{aligned}$$

Recall that x_1, x_2, \dots, x_n are known since they are chosen to be equally spaced across the interval $[-1,1]$. This amounts to solving a linear system of n equations for n unknowns where the unknowns are the weights, a_1, a_2, \dots, a_n .

2. Gaussian Quadrature

Gaussian quadrature takes a similar approach to Newton-Cotes method. The main difference is that the nodes are not chosen to be equally spaced across the interval. Instead the nodes are considered as unknowns along with the weights.

This gives $2n$ unknowns which means that we need $2n$ conditions. If it is required that the method is exact for $f(x)$ where f is a polynomial of up to order $2n - 1$ then this gives $2n$ equations.

Generally, only n equations need to be solved for the weights since it can be shown that the nodes x_i are the roots of the n -th order Legendre Polynomial on the interval $[-1,1]$. As it turns out, these nodes will be symmetrically spaced about the origin within the interval $[-1,1]$.

Example:

Let $Q[f]$ take the form $Q[f] = w_0 f(x_0) + w_1 f(x_1)$

So we have 4 free parameters, which will require 4 pieces of information. We will therefore require that the function be exact for $f(x) = 1, x, x^2, x^3$. The interval of choice for the derivation will be $[-1,1]$.

$$\begin{aligned} f(x) &= 1 \\ \int_{-1}^1 1 dx &= 2 \\ w_0 + w_1 &= 2 \end{aligned}$$

$$\begin{aligned} f(x) &= x \\ \int_{-1}^1 x dx &= 0 \\ w_0(x_0) + w_1(x_1) &= 0 \end{aligned}$$

$$\begin{aligned} f(x) &= x^2 \\ \int_{-1}^1 x^2 dx &= 2/3 \\ w_0(x_0)^2 + w_1(x_1)^2 &= 2/3 \end{aligned}$$

$$\begin{aligned} f(x) &= x^3 \\ \int_{-1}^1 x^3 dx &= 0 \\ w_0(x_0)^3 + w_1(x_1)^3 &= 0 \end{aligned}$$

The resulting system of nonlinear equations can be solved if we assume that none of w_0, w_1, x_0, x_1 are equal to 0. By combining equations, we get

$$\begin{aligned} w_0 &= 1 \\ w_1 &= 1 \\ x_0 &= -\sqrt{\frac{1}{3}} \\ x_1 &= \sqrt{\frac{1}{3}} \end{aligned}$$

$$Q[f, -1, 1] = f\left(-\sqrt{\frac{1}{3}}\right) + f\left(\sqrt{\frac{1}{3}}\right)$$

What we are getting here is maximum precision for the number of function evaluations. The other rule that used a two-node form was the trapezoid rule, which was exact only for straight lines. By solving for the nodes rather than selecting them, we've upped the degree of precision to 3 (cubic polynomials).

We can derive higher order formulas using the above procedure, but it gets tricky -- we have to solve systems of non-linear equations. An alternate procedure involves constructing Legendre polynomials -- a sequence of polynomial that have the nodes as their roots. The most common way to obtain the weights and nodes, however, is to look them up. They are typically published to 15 or so decimal places in assorted handbooks, or stored in the software package you are using.

The basic rule for Gaussian quadrature uses the same computational formula as Newton-Cotes rules. For n nodes,

$$I = \int_a^b f(x)dx \approx \sum_{i=1}^n c_i f(x_i)$$

Algorithm for Basic Gauss Quadrature on [-1,1]

```

specify n, the number of nodes
obtain n node locations,  $x_i$ , and weights,  $c_i$ , for  $-1 \leq x \leq 1$ 
evaluate  $f(x_i), i = 1, \dots, n$ 
initialize:  $I = 0$ 
for  $i = 1:n$ 
     $I = I + c_i f(x_i)$ 
end

```

As with the Newton-Cotes rules, Gaussian quadrature can be applied to a large interval that is divided into several subintervals or panels. To obtain a numerical approximation to

$$(1) \quad I = \int_a^b f(x)dx,$$

divide the interval $[a,b]$ into N subintervals of equal size $H = \frac{b-a}{N}$. N is the number of subintervals and n is the number of nodes in each subinterval. The subinterval width is H and should not be confused with h , the spacing between nodes used in Newton-Cotes rules. The starting point of each panel is

$$x_i = a + (i-1)H, \quad i = 1, \dots, N.$$

A Gauss quadrature rule is applied to each subinterval to approximate

$$(2) \quad I_i = \int_{x_i}^{x_{i+1}} f(x) dx$$

Note that in order to apply a Gauss quadrature rule, the limits of the integral must be transformed from $[x_i, x_{i+1}]$ to $[-1, 1]$. i.e $x_i^* = x_i + \left(\frac{h}{2}\right)(z+1)$ where the x_i 's are the endpoints of the subintervals, the x_i^* 's are the nodes within that interval, $h = x_{i+1} - x_i$, and z is the gauss nodes on the interval $[-1, 1]$.

So equation (2) becomes $I_i = \frac{h}{2} \int_{-1}^1 f(x) dx$

The value of the integral in equation (1) is obtained by adding together the contributions from each subinterval:

$$I = \sum_{i=1}^N I_i$$

The quadrature rule may be applied to each subinterval with any number of nodes.

Algorithm for Composite Gauss Quadrature on [a,b]

```

input: f(x), a, b, N, n
% N is the number of subintervals
% n is the number of nodes in each subinterval

x = linspace(a,b,N+1) %subinterval endpoints
compute (look up) the nodes and weights for the interval [-1,1] using n nodes
I = 0
for i = 1:N           %loop over subintervals
    I_i = 0
    for j = 1:n       %loop over nodes

        h = x(i+1) - x(i)
        x_j^* = x(i) + (h/2)*(z+1)
        %the jth node within the ith subinterval

        I_i = I_i + \frac{h}{2} c_j f(x_j^*)   %the gauss quadrature for the ith subinterval
    end
    I = I + I_i
end
end

```

3. Commands in Matlab

There are several commands in Matlab (other than the symbolic toolbox) which can be used to approximate the values of integrals.

- **quad**
- **quadl**

All of these are used in similar fashion:

```
>> quad(funname, a, b)
>> quadl(funname, a, b)
```

It should be pointed out that these methods are all adaptive methods which means that they try to intelligently choose the size of each subinterval so that the integral can be more efficiently and accurately calculated.

Type **help quad** (etc.) to find out about other options such as using a specific error tolerance for each method.

A word of caution with these methods is that they are black boxes. Since they are adaptive, you do not know going in just how much work the function is going to do.