

Newton-Cotes Quadrature

Mth 351 Summer 1999

Bent E. Petersen

Filename: newton_cotes.mws

We investigate various compound (closed) Newton-Cotes quadrature rules.

Newton-Cotes quadrature proceeds as follows - we wish to approximate an integral

```
> Int(f(x), x=a..b);
```

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

We subdivide the interval $[a,b]$ into m (the *degree*) subintervals of equal length. The corresponding $m+1$ endpoints are called the *nodes*. Now let p be the unique interpolation polynomial of degree at most m through the points on the graph of f corresponding to the nodes. Then the integral of p is our approximation to the integral of f . The Lagrange formula for the interpolation polynomial, a change of variable and a quick calculation show that

```
> Int(p(x), x=a..b) =  
(b-a) * Sum(f(a+k*(b-a)/m), k=0..m) * Int(p[k](x), x=0..1);
```

$$\int_a^b p(x) dx = (b-a) \left(\sum_{k=0}^m f\left(a + \frac{k(b-a)}{m}\right) \right) \int_0^1 p_k(x) dx$$

where $p[k]$ is the interpolation polynomial which is 1 at $\frac{k}{m}$ and 0 at the other multiples of $\frac{1}{m}$. The integrals of the $p[k]$ are called the Newton-Cotes coefficients of degree n . Here's a routine to compute these coefficients:

```
> NCC:=proc(m)  
> local k,h,xx,y,yy,coef,q,w;  
> xx:= [seq(k/m,k=0..m)];  
> coef:=[];  
> y:= [seq(0,k=0..m)];  
> for k from 0 to m do  
> yy:=subsop(k+1=1,y); # make kth component in y 1 in yy  
> q:=interp(xx,yy,x);  
> w:=int(q,x=0..1);  
> coef:= [op(coef),w]; # push onto list of weights  
> od;  
> coef;  
> end;
```

Let's check a few.

Tapezoidal rule:

> **NCC(1);**

$$\left[\frac{1}{2}, \frac{1}{2} \right]$$

Simpson's rule:

> **NCC(2);**

$$\left[\frac{1}{6}, \frac{2}{3}, \frac{1}{6} \right]$$

Simpson's 3/8 rule:

> **NCC(3);**

$$\left[\frac{1}{8}, \frac{3}{8}, \frac{3}{8}, \frac{1}{8} \right]$$

Boole's rule:

> **NCC(4);**

$$\left[\frac{7}{90}, \frac{16}{45}, \frac{2}{15}, \frac{16}{45}, \frac{7}{90} \right]$$

> **NCC(5);**

$$\left[\frac{19}{288}, \frac{25}{96}, \frac{25}{144}, \frac{25}{144}, \frac{25}{96}, \frac{19}{288} \right]$$

> **NCC(6);**

$$\left[\frac{41}{840}, \frac{9}{35}, \frac{9}{280}, \frac{34}{105}, \frac{9}{280}, \frac{9}{35}, \frac{41}{840} \right]$$

> **NCC(7);**

$$\left[\frac{751}{17280}, \frac{3577}{17280}, \frac{49}{640}, \frac{2989}{17280}, \frac{2989}{17280}, \frac{49}{640}, \frac{3577}{17280}, \frac{751}{17280} \right]$$

> **NCC(8);**

$$\left[\frac{989}{28350}, \frac{2944}{14175}, \frac{-464}{14175}, \frac{5248}{14175}, \frac{-454}{2835}, \frac{5248}{14175}, \frac{-464}{14175}, \frac{2944}{14175}, \frac{989}{28350} \right]$$

Something important happened here. For the case of degree 8 we have some negative weights. When the weights are all positive we can integrate nonnegative functions without worrying about loss of significance errors. When some weights are negative this is no longer the case. As a result the degree 8 method is not popular.

> **NCC(9);**

$$\left[\frac{2857}{89600}, \frac{15741}{89600}, \frac{27}{2240}, \frac{1209}{5600}, \frac{2889}{44800}, \frac{2889}{44800}, \frac{1209}{5600}, \frac{27}{2240}, \frac{15741}{89600}, \frac{2857}{89600} \right]$$

> **NCC(10);**

$$\left[\frac{16067}{598752}, \frac{26575}{149688}, \frac{-16175}{199584}, \frac{5675}{12474}, \frac{-4825}{11088}, \frac{17807}{24948}, \frac{-4825}{11088}, \frac{5675}{12474}, \frac{-16175}{199584}, \frac{26575}{149688}, \frac{16067}{598752} \right]$$

Degree 9 is well-behaved but degree 10 again has a sign problem.

This problem with the sign of the weights makes it useless to increase the degree in an effort to increase accuracy. We may see a decrease in truncation error, but an increase in roundoff error. Thus compound methods are usually used.

The idea in the compound methods is we break the interval into n subintervals where n is divisible by the degree m that we plan to use. Then we use the Newton-Cotes method on the first set of m subintervals, then on the second set, and so on, and then add the results together. Let's write a routine to do it for us:

```
> NCCint:=proc(f,r::range,m,n)
>   local k,j,a,b,q,C,h,hh,S;
>   if type(n, numeric) then
>     if not type(n, posint) then ERROR("n must be positive"); fi;
>     if not type(m, posint) then ERROR("m must be positive"); fi;
>     if not irem(n,m)=0 then ERROR("m (degree) must divide n
      (order)"); fi;
>   fi;
>   a:=op(1,r); b:=op(2,r); q:=n/m; h:=(b-a)/n; hh:=(b-a)/q;
>   C:=NCC(m); S:=0;
>   for j from 0 to q-1 do
>     for k from 0 to m do
>       S:=S+C[k+1]*f(a+j*hh+k*h);
>     od; od;
>   S:=hh*S; # rescale, recall intervals here have length (b-a)/q
> end;
```

For example here is the (compound) Simpson's rule with 6 subintervals on [a,b].

```
> NCCint(f,a..b,2,6);
```

$$\left(\frac{1}{3}b - \frac{1}{3}a \right) \left(\frac{1}{6}f(a) + \frac{2}{3}f\left(\frac{5}{6}a + \frac{1}{6}b\right) + \frac{1}{3}f\left(\frac{2}{3}a + \frac{1}{3}b\right) + \frac{2}{3}f\left(\frac{1}{2}a + \frac{1}{2}b\right) + \frac{1}{3}f\left(\frac{1}{3}a + \frac{2}{3}b\right) + \frac{2}{3}f\left(\frac{1}{6}a + \frac{5}{6}b\right) + \frac{1}{6}f(b) \right)$$

If we multiply by 6 and simplify we see the familiar 1,4,2,4,2,...,2,4,1 pattern.

Now let's compare some order 36 methods for

```
> rng:=2..5: ex:=Int(exp(x),x=rng);
```

$$ex := \int_2^5 e^x dx$$

```

> `(1,36) rel_error (trapezoidal)`;
evalf((ex-NCCint(exp,rng,1,36))/ex,40);
(1,36) rel_error (trapezoidal)
-.0005786367351811138195055696543412576333731
> `(2,36) rel_error (Simpson's)`;
evalf((ex-NCCint(exp,rng,2,36))/ex,40);
(2,36) rel_error (Simpson's)
-.2676970493645123463788605006041036146537 10-6
> `(3,36) rel_error (Simpson 3/8)`;
evalf((ex-NCCint(exp,rng,3,36))/ex,40);
(3,36) rel_error (Simpson 3/8)
-.6018212100510396036642660688234988688149 10-6
> `(4,36) rel_error (Boole)`;
evalf((ex-NCCint(exp,rng,4,36))/ex,18);
(4,36) rel_error (Boole)
-.706202385824962271 10-9
> `(6,36) rel_error`; evalf((ex-NCCint(exp,rng,6,36))/ex,40);
(6,36) rel_error
-.2470100308872566719802249620241372804574 10-11
> `(9,36) rel_error`; evalf((ex-NCCint(exp,rng,9,36))/ex,40);
(9,36) rel_error
-.2083586305707669970039883286859413078756 10-13
> `(12,36) rel_error`; evalf((ex-NCCint(exp,rng,12,36))/ex,40);
(12,36) rel_error
-.2149309937627081215600849772199922166976 10-18
> `(18,36) rel_error`; evalf((ex-NCCint(exp,rng,18,36))/ex,40);
(18,36) rel_error
-.2967743044315975885121890984893149445570 10-25

```

Note each of these methods required 37 function evaluations.

In computing the error we use the actual value of the integral and both ex and $NCCint()$ are computed *symbolically*. The difference is found and divided by ex , also *symbolically*. We then convert to floating point at a prescribed precision. By using a very high precision we are able to compute the error with minimal roundoff - so we are seeing mostly truncation error here.

Let's consider another example:

```

[ >
[ > rng:=0..Pi: ex:=Int(sin(x),x=rng);
      
$$ex := \int_0^{\pi} \sin(x) dx$$

[ > `(1,36) rel_error (trapezoidal)`;
      evalf((ex-NCCint(sin,rng,1,36))/ex,40);
      (1,36) rel_error (trapezoidal)
      .0006347001875770279795828014456345840185000
[ > `(2,36) rel_error (Simpson's)`;
      evalf((ex-NCCint(sin,rng,2,36))/ex,40);
      (2,36) rel_error (Simpson's)
      -.3224859886427327311172174433400355000000 10-6
[ > `(3,36) rel_error (Simpson 3/8)`;
      evalf((ex-NCCint(sin,rng,3,36))/ex,40);
      (3,36) rel_error (Simpson 3/8)
      -.7262524560995178254906258891912465000000 10-6
[ > `(4,36) rel_error (Boole)`;
      evalf((ex-NCCint(sin,rng,4,36))/ex,18);
      (4,36) rel_error (Boole)
      .9384712700000000000 10-9
[ > `(6,36) rel_error`; evalf((ex-NCCint(sin,rng,6,36))/ex,40);
      (6,36) rel_error
      -.363855925882545611794575984750000000000 10-11
[ > `(9,36) rel_error`; evalf((ex-NCCint(sin,rng,9,36))/ex,40);
      (9,36) rel_error
      .344842766022348820525975645000000000000 10-13
[ > `(12,36) rel_error`; evalf((ex-NCCint(sin,rng,12,36))/ex,40);
      (12,36) rel_error
      .444038994579830459114500000000000000000 10-18
[ > `(18,36) rel_error`; evalf((ex-NCCint(sin,rng,18,36))/ex,40);
      (18,36) rel_error
      -.898763353542815000000000000000000000000 10-25

```

One more set of examples:

```

[ > f:=x->exp(x): rng:=3..8: ex:=Int(f(x),x=rng);
      
$$ex := \int_3^8 e^x dx$$


```

```

> `(1,36) rel_error (trapezoidal)`;
evalf((ex-NCCint(f,rng,1,36))/ex,40);
          (1,36) rel_error (trapezoidal)
          -0.01606993707454761673510197708252287088319
> `(2,36) rel_error (Simpson's)`;
evalf((ex-NCCint(f,rng,2,36))/ex,40);
          (2,36) rel_error (Simpson's)
          -.2062533686062336605453489969246517737922 10-5
> `(3,36) rel_error (Simpson 3/8)`;
evalf((ex-NCCint(f,rng,3,36))/ex,40);
          (3,36) rel_error (Simpson 3/8)
          -.4630090957512773713123467870871570605425 10-5
> `(4,36) rel_error (Boole)`; evalf((ex-NCCint(f,rng,4,36))/ex,18);
          (4,36) rel_error (Boole)
          -.150390032384600922 10-7
> `(6,36) rel_error`; evalf((ex-NCCint(f,rng,6,36))/ex,40);
          (6,36) rel_error
          -.1448094774390365865743323428367052296037 10-9
> `(9,36) rel_error`; evalf((ex-NCCint(f,rng,9,36))/ex,40);
          (9,36) rel_error
          -.3325923996548388730008774697695650508581 10-11
> `(12,36) rel_error`; evalf((ex-NCCint(f,rng,12,36))/ex,40);
          (12,36) rel_error
          -.2568836808982809598089232253667627777154 10-15
> `(18,36) rel_error`; evalf((ex-NCCint(f,rng,18,36))/ex,40);
          (18,36) rel_error
          -.7011852462380051678651793139847097342379 10-21
> f:=evaln(f): # unassign

```

For our (nice and smooth) examples the higher degree methods are much superior. The composite Boole's method is no harder to apply than Simpson's rule, even for hand calculations, but has smaller error (for smooth functions).

Compound Boole, 12 subintervals

```

> NCCint(f,a..b,4,12): algsubs(b-a=12*h,%): (1/90)*simplify(90*%);

$$\frac{2}{45} (7 f(a) + 32 f(a+h) + 12 f(a+2h) + 32 f(a+3h) + 14 f(a+4h) + 32 f(a+5h) + 12 f(a+6h) + 32 f(a+7h) + 14 f(a+8h) + 32 f(a+9h) + 12 f(a+10h) + 32 f(a+11h))$$


```

$$+ 7 f(a + 12 h)) h$$

Compound Boole, 8 subintervals

```
> NCCint(f,a..b,4,8): alsubs(b-a=8*h,%): (1/90)*simplify(90*%);
```

$$\frac{2}{45} (7 f(a) + 32 f(a + h) + 12 f(a + 2 h) + 32 f(a + 3 h) + 14 f(a + 4 h) + 32 f(a + 5 h) + 12 f(a + 6 h) + 32 f(a + 7 h) + 7 f(a + 8 h)) h$$

We see we have a coefficient of $\frac{2h}{45}$ and then the sequence 7 32 12 32 14 32 12 32 14 14 32 12 32 7. The 14's come from two 7's at the points where two of the interpolation intervals meet.

Since Boole's rule depends on interpolation of degree 4 it is obviously exact for polynomials of degree at most 4. However, just as for Simpson's rule, it is actually exact for one degree more, that is, for polynomials of degree at most 5.

```
> collect(NCCint(x->x^5,a..b,4,4),b);
```

$$\frac{1}{6} b^6 - \frac{1}{6} a^6$$

The degree 9 rule is one of the higher degree rules which has positive weights. We can expect it to be fairly well-behaved. Why don't you experiment a bit with compound degree 9 rule?