Lesson 20: Newton-Cotes Rules

```
> restart;
  with(Student[Calculus1]):
```

Errors for Newton-Cotes rules with fixed n.

I want to look at the errors in Newton-Cotes rules with different orders, all using the same n, for some functions on the interval 0 .. 1.
I'll take n to be 36, so the order k can be any factor of 36. These are the possibilities.

```
> K := [1,2,3,4,6,9,12,18,36];
```

$$K := [1, 2, 3, 4, 6, 9, 12, 18, 36]$$

First I'll use our function $f(x) = \frac{1}{1+x}$.

```
> Digits:= 30:
  J:= int(1/(1+x),x=0..1):
  seq(evalf(J - ApproximateInt(1/(1+x),x=0..1,method=
  newtoncotes[K[j]],
  partition=36/K[j])), j=1..9);
```

$-0.000048220659074225432544874437, -1.8569675901583997777594\ 10^{-8}, -4.1701931418315024957474\ 10^{-8}, -1.1291685508126363460\ 10^{-10}, -1.782655156859319104\ 10^{-12}, -1.04704005629163311\ 10^{-13}, -1.47509332767563\ 10^{-16}, -1.24862983667\ 10^{-19}, -3.3836\ 10^{-26}$

For this function, the higher-order rules turned out to be better. But if we take an f whose higher derivatives grow faster, that might not be true.

```
> f:= x -> 1/(x^2 + 1/400):
  J:= int(f(x),x=0..1):
  seq(evalf(J - ApproximateInt(f(x),x=0..1,method=newtoncotes[K
  [j]],
  partition=36/K[j])), j=1..9);
```

$$-0.000641952269167915962147212 3, 0.0725441148897016001087090498, 0.184473443447492056222390601 2, 0.0091949923915456824776657040, -0.0365123673823126587492923243, -0.0518280077283736536549827 13, -0.0720167551380369955836442934, -0.0560179975592538805248659975, 0.0038520798802296260016578692 \tag{1.1}$$

Here the best answer was obtained with k = 1, i.e. the Trapezoid rule.

When higher order is worse

Here's an innocent-looking function where, taking **partition = 1** (so $n = k$), the errors tend to get worse as n and k increase.

```
> f:= x -> 1/(x^2+1); J:= int(f(x),x=-5..5);
  seq(evalf(J - ApproximateInt(f(x),x=-5..5,method=newtoncotes
  [k],
  partition=1)), k=1..30);
```

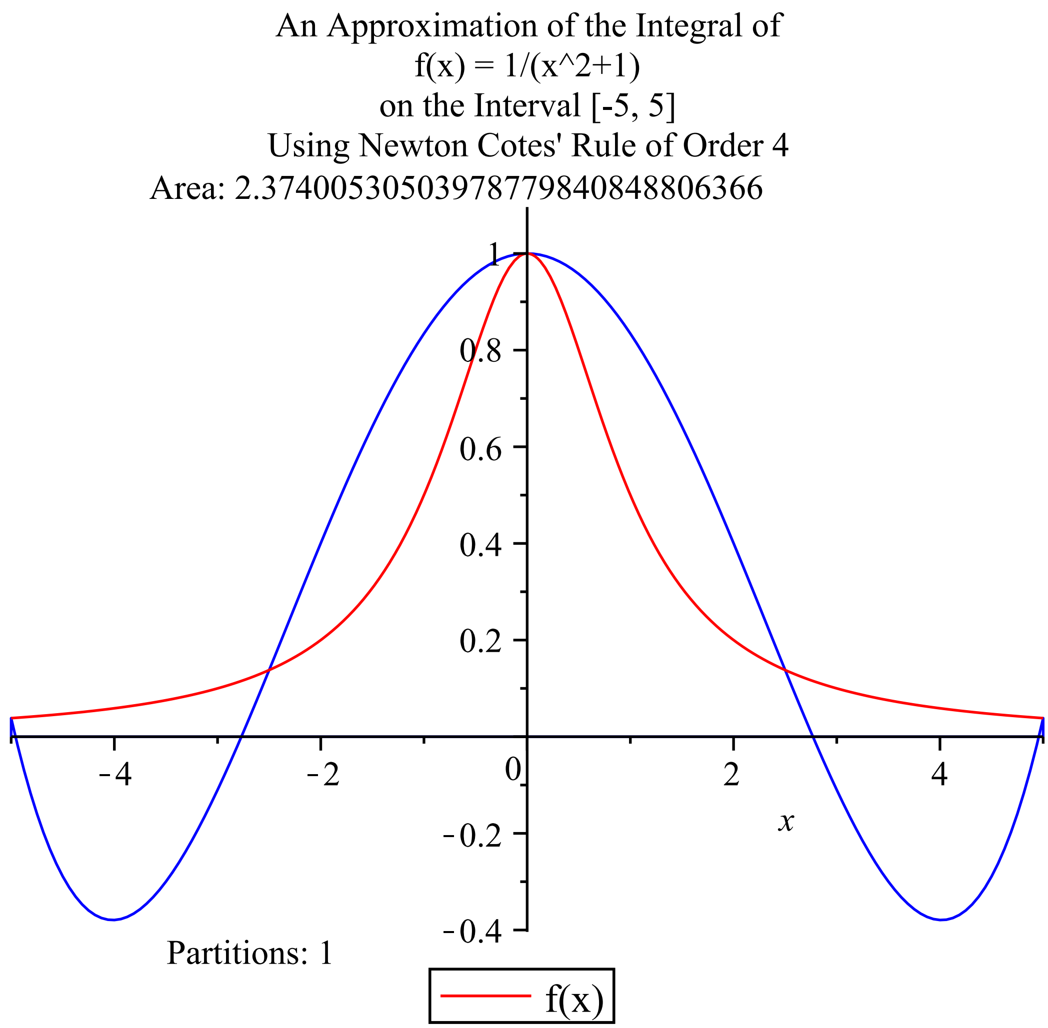
$$f := x \rightarrow \frac{1}{x^2 + 1}$$

$$J := 2 \arctan(5)$$

2.36218614927464710633792846826, -4.04807026098176315007232794199,
0.66535357008912674434697824202, 0.37279622885024392331405578922,
0.43910922619772402941485154519, -1.12364713958076803078255133975,
-0.15219287585834714013072723603, 1.24631262676212044443367320301,
0.34818363604819714486943740868, -1.92649902176346510824487434983,
-0.49797140638855296573330858472, 3.05973804964349848113316200580,
0.82700431705748154340555620008, -5.15274310696150529628600870298,
-1.40875745880984958108897822544, 8.98823884864786464489728400809,
2.48629209243840486298527849247, -16.1298197563551008484861397561,
-4.49922455162308315054814008638, 29.5963536204131431041723885357,
8.31123430387733930341693806385, -55.2971311248853536477397923599,
-15.6133901066921803820437948677, 104.874552571714577823391763842,
29.7461069336908708608183363509, -201.424202489608387121977833586,
-57.3494404584588707652736239613, 391.042086474141033243576598585,
111.701292496576266882524734562, -766.242852643854324635408907989

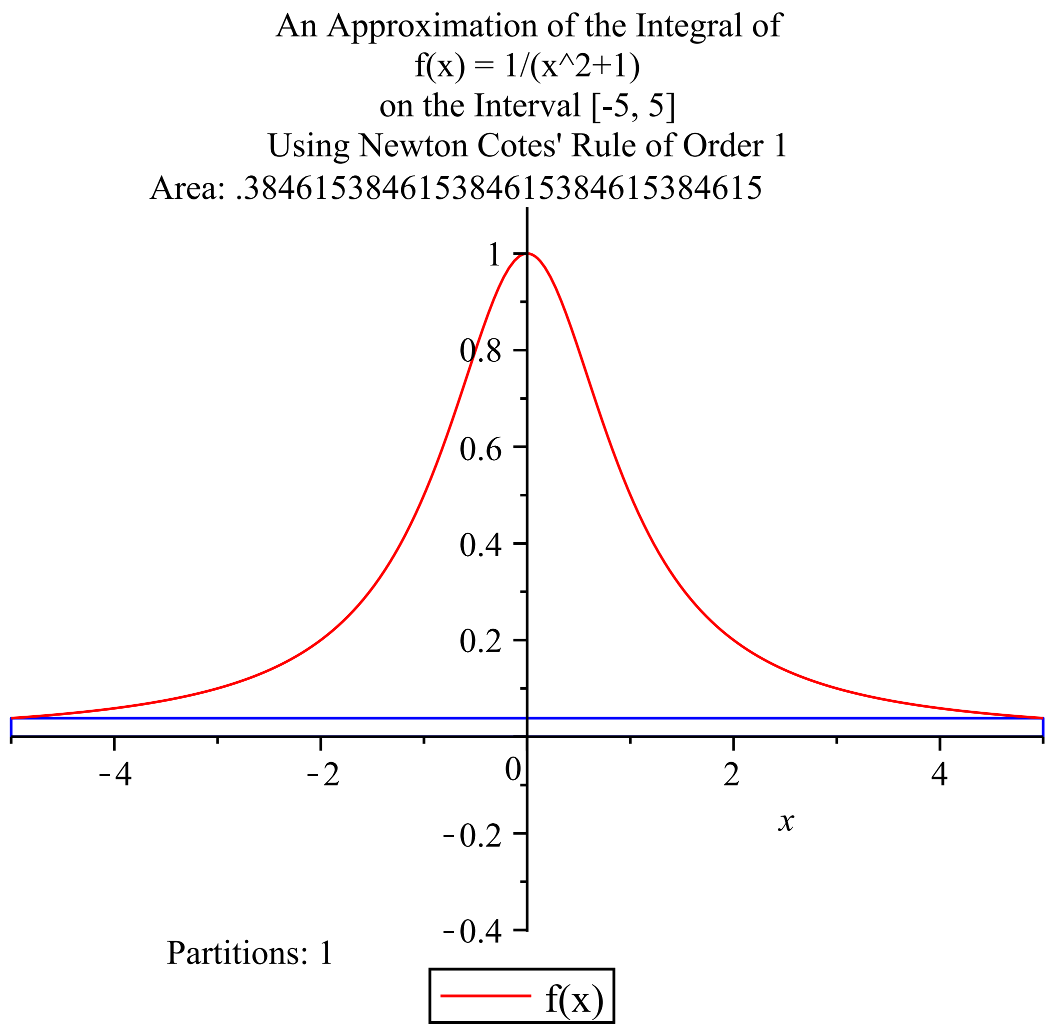
The Newton-Cotes rule has a close connection to **interpolation** by polynomials. Consider the Newton-Cotes rule of order k with $n = k$. The result depends on the value of your function f at the $k + 1$ equally spaced points $x_0, ..., x_k$. If f was a polynomial of degree at most k, this result would be correct. So what the rule gives you is the integral for a polynomial of degree at most k that agrees with f at those points. We say that polynomial interpolates the values of f at $x_0, ..., x_k$. If partition > 1, the Newton-Cotes rule does this on each partition of k intervals. **ApproximateIn**t with the option **output=plot** shows you your function f and the interpolating polynomials on each interval.

```
> ApproximateInt(1/(x^2+1),x=-5..5,method=newtoncotes[4],
  partition=1,output=plot);
```

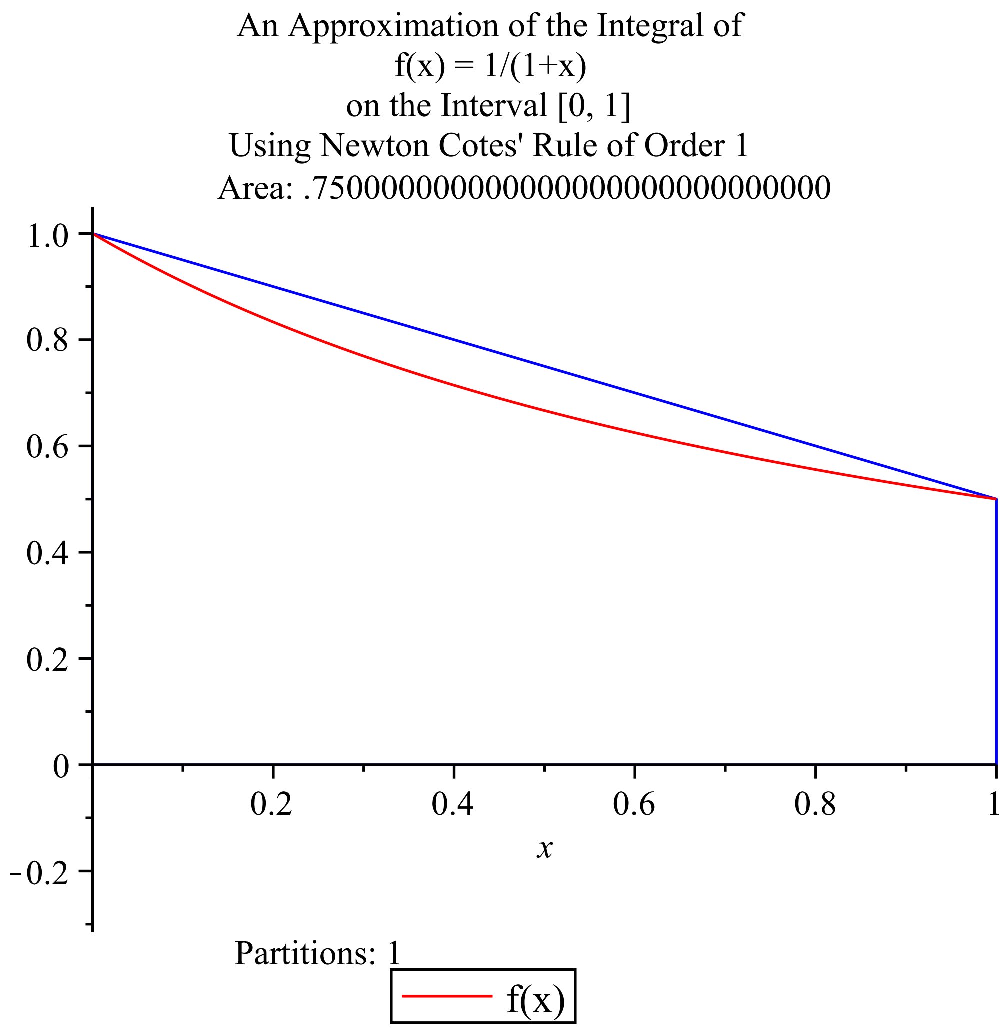


Here's an animation of this for different k with partition=1. For technical reasons, the **animate** command doesn't work here. However, as we saw in Lesson 15 there's another way to produce an animation, using the **display** command: you give it a list of plots (one for each frame), and the option **insequence=true**.

```
> with(plots):
  display([seq(ApproximateInt(1/(x^2+1),x=-5..5,method=
  newtoncotes[k],partition=1, output=plot),k=1..20)],
  insequence=true);
```




```
> display([seq(ApproximateInt(1/(x+1),x=0..1,method=newtoncotes
  [k],partition=1, output=plot),k=1..20)],insequence=true);
```



Richardson extrapolation

Suppose J is some quantity you want to calculate, and you have available some approximations $A(n)$. Often you know something about how well $A(n)$ approximates J, e.g. $J = A(n) + O(n^{-p})$. That is, the error in approximating J by $A(n)$ is less than some constant times n^{-p}. But let's suppose you have more, say $J = A(n) + C\,n^{-p} + O(n^{-p-\varepsilon})$ for some $\varepsilon > 0$ (and typically $\varepsilon = 1$ or 2). Thus the error is approximately some constant times n^{-p}. Unfortunately you don't know the constant C. If you did know it, you could make a better approximation by using $A(n) + C\,n^{-p}$ instead of $A(n)$.

Richardson extrapolation remedies this difficulty by looking at two different $A(n)$. We'll get both a better approximation for J and some idea of the error in $A(n)$. Suppose we calculate $A(n)$ and $A\left(\frac{n}{2}\right)$.

```
> J:= 'J':
  e1:= J = A(n) + C*n^(-p) + O(n^(-p-epsilon));
```

$$e1 := J = A(n) + C\,n^{-p} + O(n^{-p-\varepsilon}) \quad \textbf{(3.1)}$$

```
> e2:= eval(e1,n=n/2);
```

$$e2 := J = A\left(\frac{1}{2}n\right) + C\left(\frac{1}{2}n\right)^{-p} + \mathrm{O}\left(\left(\frac{1}{2}n\right)^{-p-\varepsilon}\right) \qquad \textbf{(3.2)}$$

Think of these as two equations in the two unknowns J and C.

```
> S:= solve({e1,e2},{J, C});
```

$$S := \left\{C = \frac{A(n) - \mathrm{O}\left(\left(\frac{1}{2}n\right)^{-p-\varepsilon}\right) + \mathrm{O}\left(n^{-p-\varepsilon}\right) - A\left(\frac{1}{2}n\right)}{\left(\frac{1}{2}n\right)^{-p} - n^{-p}}, J = \frac{A(n)\left(\frac{1}{2}n\right)^{-p} - n^{-p}\,\mathrm{O}\left(\left(\frac{1}{2}n\right)^{-p-\varepsilon}\right) - n^{-p} A\left(\frac{1}{2}n\right) + \mathrm{O}\left(n^{-p-\varepsilon}\right)\left(\frac{1}{2}n\right)^{-p}}{\left(\frac{1}{2}n\right)^{-p} - n^{-p}}\right\} \qquad \textbf{(3.3)}$$

If we neglect the O terms:

```
> SR := simplify(eval(S,O=0));
```

$$SR := \left\{C = -\frac{\left(-A(n) + A\left(\frac{1}{2}n\right)\right) n^{p}}{2^{p} - 1}, J = \frac{A(n)\,2^{p} - A\left(\frac{1}{2}n\right)}{2^{p} - 1}\right\}$$

```
> CR:= eval(C,SR); JR:= eval(J,SR);
```

$$CR := -\frac{\left(-A(n) + A\left(\frac{1}{2}n\right)\right) n^{p}}{2^{p} - 1}$$

$$JR := \frac{A(n)\,2^{p} - A\left(\frac{1}{2}n\right)}{2^{p} - 1}$$

```
> simplify(CR-eval(C,S));
```

$$\frac{n^{p}\left(\mathrm{O}\left(2^{p+\varepsilon} n^{-p-\varepsilon}\right) - \mathrm{O}\left(n^{-p-\varepsilon}\right)\right)}{2^{p} - 1}$$

So the difference between the Richardson value C_R and the true C is $\mathrm{O}\left(n^{-\varepsilon}\right)$.

```
> simplify(JR-eval(J,S));
```

$$-\frac{-\mathrm{O}\left(2^{p+\varepsilon} n^{-p-\varepsilon}\right) + \mathrm{O}\left(n^{-p-\varepsilon}\right) 2^{p}}{2^{p} - 1}$$

The difference between the Richardson value J_R and the true J is $\mathrm{O}\left(n^{-p-\varepsilon}\right)$.

When n is large, the main contribution to the error in $A(n)$ is $C\,n^{-p}$. If we approximate that error as $C_R\,n^{-p}$, how far off are we (i.e. what is the error in our approximation of the error in our approximation)?

```
> simplify(eval(J - A(n) - CR*n^(-p),S));
```

$$\frac{-\mathrm{O}\left(2^{p+\varepsilon} n^{-p-\varepsilon}\right) + \mathrm{O}\left(n^{-p-\varepsilon}\right) 2^{p}}{2^{p} - 1}$$

This is $O(n^{-p-\varepsilon})$. As long as $C \neq 0$, that's much smaller than the actual error when n is large. So this should be a good approximation for the error in $A(n)$.

```
> simplify(CR*n^(-p));
```

$$-\frac{-A(n)+A\left(\frac{1}{2}n\right)}{2^p-1} \tag{3.4}$$

We don't know a good approximation for the error in our improved approximation J_R, only that it is $O(n^{-p-\varepsilon})$. But $C_R n^{-p}$ should be a fairly conservative estimate for it, at least if n is large.

A closer look at the error in Trapezoid

Applying Richardson to Trapezoid

I want to apply Richardson extrapolation to the Trapezoid rule.

```
> h := n -> (b-a)/n:
> X := (k,n) -> a + k*h(n):
  a:= 0: b:= 1:
> T := n -> add((f(X(k-1,n)) + f(X(k,n)))/2 * h(n), k=1..n);
  J := int(f(x),x=a..b);
```

$$T := n \to add\left(\frac{1}{2}\left(f(X(k-1,n))+f(X(k,n))\right)h(n), k=1..n\right)$$

$$J := \int_0^1 f(x)\,\mathrm{d}x \tag{5.1}$$

The Trapezoid Rule $T(n)$ has error $\frac{c_2}{n^2}+O\left(\frac{1}{n^4}\right)$, so the improved approximation using Richardson extrapolation would be

```
> TR[1] := n -> (2^2*T(n) - T(n/2))/(2^2-1);
```

$$TR_1 := n \to \frac{4}{3}T(n)-\frac{1}{3}T\left(\frac{1}{2}n\right) \tag{5.2}$$

I'm calling it TR_1 instead of just TR because, as we'll see, this will be the start of a sequence TR_k

```
> TR[1](2);
```

$$\frac{1}{6}f(0)+\frac{2}{3}f\left(\frac{1}{2}\right)+\frac{1}{6}f(1) \tag{5.3}$$

That should look familiar. TR_1 is Simpson's rule.

If T(n) has error $\sum_{k=1}^{N}\frac{c_{2k}}{n^{2k}}+O\left(\frac{1}{n^{2N+2}}\right)$, what about TR_1? It's not hard to see that this will have error $\sum_{k=2}^{N}\frac{c'_{2k}}{n^{2k}}+O\left(\frac{1}{n^{2N+2}}\right)$ (where c'_{2k} are other constants). So Richardson extrapolation improves TR_1 to this:

```
> TR[2]:= n -> (2^4*TR[1](n)-TR[1](n/2))/(2^4-1);
```

$$TR_2 := n \to \frac{16}{15}\, TR_1(n) - \frac{1}{15}\, TR_1\left(\frac{1}{2}\, n\right) \tag{5.4}$$

```
> TR[2](4);
```

$$\frac{7}{90}\, f(0) + \frac{16}{45}\, f\left(\frac{1}{4}\right) + \frac{2}{15}\, f\left(\frac{1}{2}\right) + \frac{16}{45}\, f\left(\frac{3}{4}\right) + \frac{7}{90}\, f(1) \tag{5.5}$$

This turns out to be the same as Newton-Cotes rule of order 4.

```
> with(Student[Calculus1]):
  ApproximateInt(f(x), x=a..b, partition=1,
  method=newtoncotes[4]);
```

$$\frac{7}{90}\, f(0) + \frac{16}{45}\, f\left(\frac{1}{4}\right) + \frac{2}{15}\, f\left(\frac{1}{2}\right) + \frac{16}{45}\, f\left(\frac{3}{4}\right) + \frac{7}{90}\, f(1) \tag{5.6}$$

This should have error $\displaystyle\sum_{k=3}^{N} \frac{c''_{2k}}{n^{2k}} + \mathrm{O}\left(\frac{1}{n^{2N+2}}\right)$.

```
> TR[3] := n -> (2^6*TR[2](n)-TR[2](n/2))/(2^6-1);
```

$$TR_3 := n \to \frac{64}{63}\, TR_2(n) - \frac{1}{63}\, TR_2\left(\frac{1}{2}\, n\right) \tag{5.7}$$

```
> TR[3](8);
```

$$\frac{31}{810}\, f(0) + \frac{512}{2835}\, f\left(\frac{1}{8}\right) + \frac{176}{2835}\, f\left(\frac{1}{4}\right) + \frac{512}{2835}\, f\left(\frac{3}{8}\right) + \frac{218}{2835}\, f\left(\frac{1}{2}\right) + \frac{512}{2835}\, f\left(\frac{5}{8}\right) + \frac{176}{2835}\, f\left(\frac{3}{4}\right) + \frac{512}{2835}\, f\left(\frac{7}{8}\right) + \frac{31}{810}\, f(1) \tag{5.8}$$

This one is **not** the same as the Newton-Cotes rule of order 8, although they evaluate f at the same points.

```
> ApproximateInt(f(x), x=a..b, partition=1,
  method=newtoncotes[8]);
```

$$\frac{989}{28350}\, f(0) + \frac{2944}{14175}\, f\left(\frac{1}{8}\right) - \frac{464}{14175}\, f\left(\frac{1}{4}\right) + \frac{5248}{14175}\, f\left(\frac{3}{8}\right) - \frac{454}{2835}\, f\left(\frac{1}{2}\right) + \frac{5248}{14175}\, f\left(\frac{5}{8}\right) - \frac{464}{14175}\, f\left(\frac{3}{4}\right) + \frac{2944}{14175}\, f\left(\frac{7}{8}\right) + \frac{989}{28350}\, f(1) \tag{5.9}$$

We could go farther with these "TR rules", but we won't. The correct name is "Romberg Integration".

TR_3 versus $newtoncotes_8$

TR_1 was Simpson's Rule (the Newton-Cotes rule of order 2), and TR_2 was the Newton-Cotes rule of order 4, but TR_3 is not a Newton-Cotes rule. Which is better, TR_3 or the Newton-Cotes rule of order 8?

On the one hand, TR_3 should have error $\mathrm{O}(n^{-8})$, while $newtoncotes_8$ should have $\mathrm{O}(n^{-10})$. So $newtoncotes_8$ should be better for large n. On the other hand, if n is fairly small TR_3 might be as good or better. Here is our function from last time that was bad for the Newton-Cotes rules with partition=1.

```
> f := x -> 1/((8*x-4)^2+1);
  evalf(J-TR[3](8));
  evalf(J-ApproximateInt(f(x),x=0..1,method=newtoncotes[8],
  partition=1));
```

$$f := x \to \frac{1}{(8\,x-4)^2+1}$$

0.0085039023782592833983298773053

0.088817627868455361829702426324 **(6.1)**

So in this particular case TR_3 is much better. If we used a larger n, Newton-Cotes might win.

```
> seq([evalf(J-TR[3](8*k)),
  evalf(J-ApproximateInt(f(x),x=0..1,method=newtoncotes[8],
  partition=k))], k=1..10);
```

[0.0085039023782592833983298773053, 0.088817627868455361829702426324], [**(6.2)**
-0.000285624142523695725955841187, -0.000623764984166393543784739424], [
-0.000204883478822395979713756790, 0.000572494430453664642795487508], [
-0.000029727163046928359760280158, 0.000009360917377241639161280457], [
-0.000014334144057919530076745403, 0.000011222845549916056796665309],
[$6.35591878056944622021186\ 10^{-7}$, $9.98784313900962525216486\ 10^{-7}$], [
-0.000001520348681035074090446262, $5.01169142327219992932987\ 10^{-7}$],
[$4.37170182374160065048074\ 10^{-7}$, $-1.588471646387363761220\ 10^{-9}$], [
$-2.52547826016599896396935\ 10^{-7}$, $4.5158941472082878856508\ 10^{-8}$],
[$1.04081643684555957157445\ 10^{-7}$, $-1.1096585987219955417995\ 10^{-8}$]

Maple objects introduced in this lesson

ApproximateInt(..., output=plot) in **Student[Calculus1]** package
Rule[parts,...] in **Student[Calculus1]** package
op