CS 3430: S24: Lecture 14
Integration Approximation: Part 02: Romberg Lattices and Matrices

Vladimir Kulyukin
Department of Computer Science
Utah State University

Trapezoidal Rule of Definite Integral Approximation

Let us review the Trapezoidal Rule (T Rule) to approximate definite integrals of continuous functions. The formula for the T Rule is this.

$$\int_a^b f(x)dx \approx \left(\frac{b-a}{2n}\right)\left[f(a) + 2\sum_{i=1}^{n-1} f(a+ih) + f(b)\right] =$$

$$\frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(a+ih) + f(b)\right],$$

where $h = \frac{b-a}{n}$ and n is the number of the segments into which we divide the interval from a to b, i.e., $[a, b]$.

The Magnitude of the Error of the T Rule

Recall from Lecture 12 that $\left|E_t\right|$ is the absolute true relative error.

The formula of the magnitude of this error for the Trapezoidal Rule is as follows. Let us assume that $f''(x) \le K$, for some real constant K. Then

$$\left|E_t\right| \le \frac{K(b-a)^3}{12n^2} = O\left(\frac{1}{n^2}\right).$$

So, the magnitude of the error is proportional to $\frac{1}{n^2}$ or inversely proportional to n^2, i.e., the square of the number of segments into which we partition the interval $[a, b]$.

Some Notational Convention for the T Rule

We will use the following notation for the T Rule: $T_{k,1}$ is the application of T Rule on the interval $[a, b]$ partitioned into $n = 2^{k-1}$ segments. The meaning of the second subscript, i.e., 1 to the right of k in $T_{k,1}$, will become clear later in the lecture.

$T_{1,1}$ is the application of the T Rule on $[a, b]$ partitioned into $n = 2^{1-1} = 1$ segment, i.e., the segment contains the entire interval $[a, b]$.

$T_{2,1}$ is the application of the T Rule on $[a, b]$ partitioned into $n = 2^{2-1} = 2$ segments.

$T_{3,1}$ is the application of the T Rule on $[a, b]$ partitioned into $n = 2^{3-1} = 4$ segments.

$T_{4,1}$ is the application of the T Rule on $[a, b]$ partitioned into $n = 2^{4-1} = 8$ segments.

$T_{5,1}$ is the application of the T Rule on $[a, b]$ partitioned into $n = 2^{5-1} = 16$ segments.

And so on...

Some Notational Convention of the Trapezoidal Rule

If $f(x)$ is a continuous function that must be integrated on $[a, b]$, then, for $k = 1$, we have $n = 2^{1-1}$ segments and

$$\begin{aligned} h &= \tfrac{b-a}{1}; \\ T_{1,1} &= \left(\tfrac{h}{2}\right)\left[f(a) + \textstyle\sum_{i=1}^{1-1} f(a+ih) + f(b)\right] \\ &= \tfrac{b-a}{2}\Big(f(a) + f(b)\Big). \end{aligned}$$

For $k = 2$, we have $n = 2^{2-1} = 2$ segments and

$$\begin{aligned} h &= \tfrac{b-a}{2}; \\ T_{2,1} &= \left(\tfrac{h}{2}\right)\left[f(a) + 2\textstyle\sum_{i=1}^{2-1} f(a+ih) + f(b)\right]. \end{aligned}$$

For $k = 3$, we have $n = 2^{2} = 4$ segments and

$$\begin{aligned} h &= \tfrac{b-a}{4}; \\ T_{3,1} &= \left(\tfrac{h}{2}\right)\left[f(a) + 2\textstyle\sum_{i=1}^{4-1} f(a+ih) + f(b)\right]. \end{aligned}$$

Some Notational Convention of the Trapezoidal Rule

For $k = 4$, we have $n = 2^{4-1} = 8$ segments and

$$\begin{aligned} h &= \tfrac{b-a}{8}; \\ T_{4,1} &= \left(\tfrac{h}{2}\right)\left[f(a) + \textstyle\sum_{i=1}^{8-1} f(a+ih) + f(b)\right]. \end{aligned}$$

For $k = 5$, we have $n = 2^{5-1} = 16$ segments and

$$\begin{aligned} h &= \tfrac{b-a}{16}; \\ T_{5,1} &= \left(\tfrac{h}{2}\right)\left[f(a) + 2\textstyle\sum_{i=1}^{16-1} f(a+ih) + f(b)\right]. \end{aligned}$$

and so on...

Example 1

Enough theory, time for an example. So, let us approximate $\int_{0.1}^{1.3} 5xe^{-2x}dx$ with different numbers of segments for the T Rule.

$$\int_{0.1}^{1.3} 5xe^{-2x}dx = \left(\frac{1.3-0.1}{6}\right)\left[f(0.1)+2\sum_{i=1}^{2} f(0.1+i0.4)+f(1.3)\right]$$

On the next slide we will get the ground truth (at least, a ground truth estimate) with sympy. It will give us the value to compare our approximations against.

Example 1

```
import sympy as sp
from sympy import symbols
from sympy.utilities import lambdify

x = symbols('x')
intf =  sp.integrate(5*x*sp.exp(-2.0*x), x)

lintf = lambdify((x), intf)
### this is the value of the integral
sp_int_val = lintf(1.3) - lintf(0.1)
print('sympy_integral_value = {}'.format(sp_int_val))
```

The above code segment prints out

`sp_int_val = 0.8938650276524703`.

Example 1

Let us assume that that we have implemented the computation of $T_{k,1}$ with the T Rule formula with $n = 2^{k-1}$ intervals as a static method in the rmb.py. The name of this module abbreviates Romberg (more on Romberg later). The method rmb.T_k_1(f, a, b, k) takes a function, the lower and upper ends of the interval, i.e., a and b, and the value of k. When we run the following code segment

```
f = lambda x: 5.0*x*math.exp(-2.0*x)
a, b = 0.1, 1.3
for k in range(1, 9):
    T_k_1 = rmb.T_k_1(f, a, b, k)
    n = 2**(k-1)
    ### tsum is the internal sum in the T Rule formula.
    tsum = sum(f(a+i*(b-a)/n) for i in range(1, n))
    print('T_{}_1 = {}'.format(k, T_k_1))
    assert np.allclose(T_k_1, np.longdouble((b - a)/(2*n)*(f(a) + 2*tsum + f(b))))
```

we get the following output. So, it looks like $T_{8,1}$ puts us pretty close to sympy's 0.8938650276524703.

```
T_1_1 = 0.5352861809592966
T_2_1 = 0.7854967147570219
T_3_1 = 0.865348660703763
T_4_1 = 0.8866421503679945
T_5_1 = 0.8920533685405028
T_6_1 = 0.8934117404006319
T_7_1 = 0.8937516825405087
T_8_1 = 0.8938366899179881
```

Richardson Extrapolation (RE) for the T Rule

RE can be applied not only for differentiation but also for integration. Recall that TV stands for "true value," AV for "approximate value," and that, in general, $TV = AV + Error$. So, in case of the T Rule, we have two initial equations, where n is the number of the segments into which the interval $[a, b]$ is partitioned.

1. $TV = AV_n + \frac{C}{n^2}$;
2. $TV = AV_{2n} + \frac{C}{(2n)^2} = AV_{2n} + \frac{C}{4n^2}$

If we multiply both sides of Equation 2 by 4, we have

2. $4TV = AV_{2n} + \frac{C}{n^2}$.

We subtract 2 from 1 to get

$$TV = AV_{2n} + \frac{AV_{2n} - AV_n}{3}.$$

Example 1

Let AV_n denote the approximate value computed with the T Rule with n segments and we take TV to be sympy's 0.8938650276524703. AV_n is computed with the appropriate value of k in $T_{k,1}$. I computed the table below in Python 3.6.7 on Ubuntu 18.04 LTS. Ubuntu is a Linux distribution. 18.04 LTS is also named "Bionic Beaver." Your floats may be slightly different.

k	n	$AV_n = T_{k,1}$	$E_t = TV - AV_n$
1	$2^{1-1} = 1$	0.5352861809592966	0.3585788466931737 5
2	$2^{2-1} = 2$	0.7854967147570219	0.1083683128954484 6
3	$2^{3-1} = 4$	0.865348660703763	0.0285163669487072 95
4	$2^{4-1} = 8$	0.8866421503679945	0.0072228772844757 93
5	$2^{5-1} = 16$	0.8920533685405028	0.0018116591119675 673
6	$2^{6-1} = 32$	0.8934117404006319	0.0004532872518384 634
7	$2^{7-1} = 64$	0.8937516825405087	0.0001133451119615 9358
8	$2^{8-1} = 128$	0.8938366899179881	2.833773448218668 3e-05
9	$2^{9-1} = 256$	0.8938579431278147	7.084524655587288e-06

Note that the number of intervals n doubles at *every* iteration.

Example 1

The RE values are computed from AV's. So, since we have computed AV's, let us compute the corresponding RE's for the first three rows of the table on the previous slide.

k	n	$AV_n = T_{k,1}$	RE
1	$2^{1-1} = 1$	0.5352861809592966	
2	$2^{2-1} = 2$	0.7854967147570219	0.8689002260229303
3	$2^{3-1} = 4$	0.865348660703763	0.8919659760193435

How did we get these values? If AV_n is the approximate value of the T Rule for n segments and the number of the segements doubles every iteration, the RE value, after the first row, is computed from the current row and the previous row, i.e., from AV_{2n} and AV_n.

Thus, after AV_1 is computed with $T_{1,1}$ for the row with $k = 1$, $n = 1$ (row 1), the next AV_{2n} is $AV_{2n} = AV_{2 \cdot 1} = AV_2$. The RE in the row $k = 2, n = 2$ is computed from AV_2 and AV_1 as $AV_2 + (AV_2 - AV_1)/3 =$ $0.7854967147570219 + (0.7854967147570219 +$ $0.5352861809592966)/3.0 = 0.8689002260229303$.

Example 1

After we computed AV_2 on the previous slide, we proceed to the row $k = 3, n = 4$.

k	n	$AV_n = T_{k,1}$	RE
1	$2^{1-1} = 1$	0.5352861809592966	
2	$2^{2-1} = 2$	0.7854967147570219	0.8689002260229303
3	$2^{3-1} = 4$	0.865348660703763	0.8919659760193435

In this row, the RE is computed from AV_4 and AV_2. Why? Since the number of segments in the previous iteration was $n = 2$, i.e., AV_2. The number of segments in the next iteration is double the number of segments in the previous iteration. Thus, $AV_{2n} = AV_{2\cdot 2} = AV_4$. So, we have
$AV_4 + (AV_4 - AV_2)/3 =$
$0.865348660703763 + (0.865348660703763 - 0.7854967147570219)/3 =$
0.8919659760193435.
And so on.

Example 1

Proceeding downward in this fashion from row to row, we get the following table.

k	n	$AV_n = T_{k,1}$	RE
1	$2^{1-1} = 1$	0.5352861809592966	
2	$2^{2-1} = 2$	0.7854967147570219	0.8689002260229303
3	$2^{3-1} = 4$	0.865348660703763	0.8919659760193435
4	$2^{4-1} = 8$	0.8866421503679945	0.8937399802560717
5	$2^{5-1} = 16$	0.8920533685405028	0.8938571079313389
6	$2^{6-1} = 32$	0.8934117404006319	0.8938645310206749
7	$2^{7-1} = 64$	0.8937516825405087	0.8938649965871344
8	$2^{8-1} = 128$	0.8938366899179881	0.8938650257104813
9	$2^{9-1} = 256$	0.8938579431278147	0.8938650275310903

Observe that we already have a decent approximation in the RE column for the row with $k = 7$, $n = 64$, so no need to continue after this point.

A Reflective Pause

Let us pause for a moment and reflect on what we have done. What is the point of this computation? We have `sympy`, we can use it, why bother with the T Rule and RE?

Because the T Rule and RE allow us to approximate definite integrals without the computation of the antiderivative.

Every symbolic mathematics engine such as `sympy` (and it is a true work of art!) has its limits. At some point, if you stay with scientific computing and need to integrate various functions for you research or practice, you will come over a function that your engine will not be able to integrate. Then you will need to use the T Rule and RE or another approximation technique, e.g., Riemann sums.

Example 2

Let us apply the combination of the T Rule and RE on the previous to this function.

$$f(t) = \int_8^{30} \left[2000 ln \left[\frac{140,000}{140,000 - 2100t} \right] - 9.8t \right] dt.$$

The book from which I took this function gives the true value at 11061.

Example 2

Let us get the TV assessment from `sympy`, because the function is easy enough. Since the function involved the natural logartihm, `sympy` may return a complex number (and my version does), but we can convert it into a real with `numpy.longdouble`.

```
t = symbols('t')
intf2 =  sp.integrate(2000*sp.ln(140000.0/(140000 - 2100*t)) - 9.8*t)
lintf2 = lambdify((t), intf2)
sp_int_val = lintf2(30) - lintf2(8)
print('sp_int_val = {}'.format(sp_int_val))
print('sp_int_val = {}'.format(np.longdouble(sp_int_val)))
```

The output of the above code segment (on my OS (Bionic Beaver) and my version of Python) is as follows.

```
sp_int_val = (11061.335535081103+0j)
ComplexWarning: Casting complex values to real discards the imaginary part
sp_int_val = 11061.335535081103
```

We can ignore the warning, because the imaginary part is 0. So, sympy's estimate is 11061.335535081103.

Example 2

Here is the table of AV's and RE's. I skipped the relative error column. Rest assured though that the error does go down pretty fast.

k	n	AV_n	RE
1	1	11868.34818984112	
2	2	11266.374293259403	11065.716327732165
3	4	11112.820676369294	11061.636137405925
4	8	11074.221297660055	11061.354838090308
5	16	11064.557886992881	11061.336750103823
6	32	11062.141180115508	11061.335611156384
7	64	11061.53694990726	11061.335539837844
8	128	11061.385889010568	11061.335535378337
9	256	11061.348123577327	11061.33553509958

The book's TV is 11061 and the sympy's TV is 11061.335535081103. The table tells us that we can stop at the row $k = 1, n = 8$, i.e., after 3 REs, because 11061.354838090308 is in the ballpark of both TVs. So, 3 REs of the T Rule give us a decent estimate of the definite integral *without having to compute any antiderivative*.

Romberg Integration

Romberg Integration is a method to approximate values of definite integrals by combining the T Rule and RE.

A key idea of the Romberg Integration is to comute as many T Rule approximations as needed and then repeatedly apply the RE to compute more accurate definite integral estimates.

How many times do we need to apply the T Rule initially? That depends on how much precision we would like to get. This is a classical tradeoff between precision and computation.

Romberg Lattice

The conceptual data structure in terms of which Romberg Integration can be visualized is called the *Romberg Lattice*. For example, suppose that we have decided to compute five T Rule approximations $T_{1.1}$, $T_{2,1}$, $T_{3,1}$, $T_{4,1}$, and $T_{5,1}$. The complete Romberg lattice will look as follows. Note that in the lattice T(1,1) = $T_{1,1}$, T(2,1) = $T_{2,1}$, etc. Now the meaning of the second subscript in $T_{k,1}$ should become clear: it is at the bottom level (1st level) of the Romberg lattice.

```
Level 5:                   R(5,5)
                          /      \
Level 4:              R(4,4)     R(5,4)
                      /       \   /      \
Level 3:          R(3,3)    R(4,3)     R(5,3)
                  /     \   /      \    /    \
Level 2:      R(2,2)   R(3,2)     R(4,2)    R(5,2)
              /     \   /    \    /     \    /   \
Level 1:    T(1,1) T(2,1)   T(3,1)     T(4,1)   T(5,1)
```

Computation of Individual Nodes in Romberg Lattice

The next question is, how do we compute the individual nodes above the level 1 nodes that can be computed with the T Rule. Answer: with the RE applied to the two nodes on the previous level. For example, how do we compute $R_{2,2}$, which is notationally equivalent to $R(2,2)$ in the lattice? Let us set up the two equations. The second term in each equation is an error estimate, which is a function of h in the T Rule.

1. $R_{2,2} = T_{1,1} + Ch^2$
2. $R_{2,2} = T_{2,1} + C\left(\frac{h}{2}\right)^2 = T_{2,1} + C\frac{h^2}{4}$.

From these two equations, we can derive the following equation by subtracting 2 from 1:

$$R_{2,2} = T_{2,1} + \frac{T_{2,1} - T_{1,1}}{3}.$$

Computation of Individual Nodes in Romberg Lattice

How do we compute $R_{3,2}$? We have two equations.

1. $R_{3,2} = T_{2,1} + C\frac{h^2}{4}$
2. $R_{3,2} = T_{3,1} + C\frac{h^2}{16}$.

From these two equations, we derive

$$R_{3,2} = T_{3,1} + \frac{T_{3,1} - T_{2,1}}{3}.$$

Computing Nodes in Romberg Lattice

After playing with individual nodes above level 1, we can derive a general formula. Suppose that we want to compute $R_{j,l}$, i.e., the j-th Romberg's approximation at level l.

When $l = 1$, we apply the T rule for a given value of j, i.e., if $j = 1$, then $T_{1,1}$, if $j = 2$, then $T_{2,1}$, etc.

When $l > 1$, $R_{j,l}$ is

$$R_{j,l} = R_{j,l-1} + \frac{R_{j,l-1} - R_{j-1,l-1}}{4^{l-1} - 1}.$$

Examples

$$R_{2,2} = R_{2,1} + \frac{R_{2,1} - R_{1,1}}{3}$$

$$R_{3,3} = R_{3,2} + \frac{R_{3,2} - R_{2,2}}{15}$$

$$R_{4,3} = R_{4,2} + \frac{R_{4,2} - R_{3,2}}{15}$$

$$R_{5,4} = R_{5,3} + \frac{R_{5,3} - R_{4,3}}{63}$$

Computing Nodes in Romberg Lattice with Romberg Matrix

We can compute $R_{j,l}$ using the recursive formula. However, this is slow, especially for larger values of j and l, because we have to re-compute the same nodes in the lattice multiple times.

If we want to be more efficient, we can use a dynamic programming trick: to compute $R_{j,l}$, we allocate a $j \times l$ matrix, a Romberg matrix, and compute the values in the first column of the matrix, i.e., $R_{1,1} = T_{1,1}$, $R_{2,1} = T_{2,1}$, $R_{3,1} = T_{3,1}$, ..., $R_{j,1} = T_{j,1}$.

For the subsequent columns 2, 3, . . ., l, we compute the values of the current column from the values of the previous column. In particular, $R_{j,l}$ is computed from $R_{j,l-1}$ and $R_{j-1,l-1}$, which are already computed and saved in the previous column (i.e., column $l - 1$).

Example of a Romberg Matrix

Let us compute $R_{5,5}$ with the Romberg Matrix. Here is the full matrix.

	1	2	3	4	5
1	T(1,1)				
2	T(2,1)	R(2,2)			
3	T(3,1)	R(3,2)	R(3,3)		
4	T(4,1)	R(4,2)	R(4,3)	R(4,4)	
5	T(5,1)	R(5,2)	R(5,3)	R(5,4)	R(5,5)

Let us compute it column by column.

Romberg Matrix: Computation of Column 1

We compute the column 1 entries by applying the T Rule at the corresponding numbers of segments according to the general formula.

	1	2	3	4	5
1	T(1,1)				
2	T(2,1)				
3	T(3,1)				
4	T(4,1)				
5	T(5,1)				

On to column 2.

Romberg Matrix: Computation of Column 2

Column 2 is computed by the RE of the corresponding values in column 1, i.e., R(2,2) is computed as the RE of T(2,1) and T(1,1), R(3,2) as the RE of T(3,1) and T(2,1), etc.

	1	2	3	4	5
1	T(1,1)				
2	T(2,1)	R(2,2)			
3	T(3,1)	R(3,2)			
4	T(4,1)	R(4,2)			
5	T(5,1)	R(5,2)			

On to column 3.

Romberg Matrix: Column 3

Column 3 is computed by the RE of the corresponding values in column 2, i.e., R(3,3) is computed as the RE of R(3,2) and R(2,2), R(4,3) as the RE of R(4,2) and R(3,2), etc.

	1	2	3	4	5
1	T(1,1)				
2	T(2,1)	R(2,2)			
3	T(3,1)	R(3,2)	R(3,3)		
4	T(4,1)	R(4,2)	R(4,3)		
5	T(5,1)	R(5,2)	R(5,3)		

On to column 4.

Romberg Matrix: Column 4

Column 4 is computed by the RE of the corresponding values in column 3, i.e., R(4,4) is computed as the RE of R(4,3) and R(3,3), R(5,4) as the RE of R(4,3) and R(5,3), etc.

	1	2	3	4	5
1	T(1,1)				
2	T(2,1)	R(2,2)			
3	T(3,1)	R(3,2)	R(3,3)		
4	T(4,1)	R(4,2)	R(4,3)	R(4,4)	
5	T(5,1)	R(5,2)	R(5,3)	R(5,4)	

On to column 5.

Romberg Matrix: Column 5

The only entry in column 5 is computed by the RE of the corresponding values in column 4, i.e., R(5,5) is computed as the RE of R(4,4) and R(5,4).

	1	2	3	4	5
1	T(1,1)				
2	T(2,1)	R(2,2)			
3	T(3,1)	R(3,2)	R(3,3)		
4	T(4,1)	R(4,2)	R(4,3)	R(4,4)	
5	T(5,1)	R(5,2)	R(5,3)	R(5,4)	R(5,5)

And (mental drum roll!) we are done.

Some Implementational Observations on Romberg Matrices

We need a lot of precision when computing Rombergs. One way to get more precision is to use the C++ long double type, which is available with `np.longdouble`. You probably noticed that we used this type in the code segments on the previous slides when comparing our integral approximations with sympy's.

The T Rule should return `np.longdouble`. The matrix should also be of type `np.longdouble`. Here is an example of allocating a `numpy` matrix of `np.longdouble`'s.

```
>>> m = np.zeros((5, 5), dtype=np.longdouble)
>>> m.dtype
dtype('float128')
>>> m[1,1].dtype
dtype('float128')
```

Some Implementational Observations on Romberg Matrices

Another implementational observation is that we do not really need a full matrix. If the function is involved and we need large Rombers, we will waste a lot of space above the main diagonal.

A moment's reflection will tell us that we only need 2 arrays of long doubles. If we go column by column, the arrays should have a length of j each, i.e., the number of rows in the Romberg matrix. If we go row by row, i.e., the arrays should have a length of l, i.e., the number of columns in the Romberg matrix.

Computing Rombergs with Column Arrays

Let us compute R(5,5) by using two column arrays of length 5. Initially, the contents of the arrays are as follows. Array 1 is filled with $T_{j,1}$ values, $1 \leq j \leq 5$. Array 2 is empty.

	1	2
1	T(1,1)	
2	T(2,1)	
3	T(3,1)	
4	T(4,1)	
5	T(5,1)	

Computing Rombergs with Column Arrays

Then the values of Array 2 are filled with the Rombergs computed from the appropriate values of column 1, i.e., R(2,2) from T(2,1) and T(1,1), R(3,2) from T(3,1) and T(2,1), etc.

```
  |    1   |    2   |
---------------------
1 | T(1,1) |        |
---------------------
2 | T(2,1) | R(2,2) |
---------------------
3 | T(3,1) | R(3,2) |
---------------------
4 | T(4,1) | R(4,2) |
---------------------
5 | T(5,1) | R(5,2) |
---------------------
```

Computing Rombergs with Column Arrays

Then the appropriate cells of Array 2 are placed into the corresponding cells of Array 1. We can zero out T(1,1), because it has served its purpose, but it is harmless and faster to let it be. So we put R(2,2) into cell 2 of Array 1, R(3,2) into cell 3 of Array 1, R(4,2) into cell 4, and R(5,2) into cell 5.

	1	2
1	T(1,1)	
2	R(2,2)	R(2,2)
3	R(3,2)	R(3,2)
4	R(4,2)	R(4,2)
5	R(5,2)	R(5,2)

Computing Rombergs with Column Arrays

Then the cells of Array 2 are filled with the appropriate Rombergs computed from the appopriate cells of Array 1, i.e., R(3,3) from R(3,2) and R(2,2), R(4,3) from R(4,2) and R(3,2), and R(5,3) from R(5,2) and R(4,2). R(2,2) and T(1,1) are left alone.

	1	2
1	T(1,1)	
2	R(2,2)	R(2,2)
3	R(3,2)	R(3,3)
4	R(4,2)	R(4,3)
5	R(5,2)	R(5,3)

Computing Rombergs with Column Arrays

It is time for another cell value replacement. This time we put R(3,3) into cell 3 of Array 1, R(4,3) into cell 4 of Array 1, and R(5,3) into cell 5.

	1	2
1	T(1,1)	
2	R(2,2)	R(2,2)
3	R(3,3)	R(3,3)
4	R(4,3)	R(4,3)
5	R(5,3)	R(5,3)

Computing Rombergs with Column Arrays

We compute R(4,4) in Array 2 from R(4,3) and R(3,3) in Array 1. Then we compute R(5,4) in Array 2 from R(5,3) and R(4,3) in Array 1.

	1	2
1	T(1,1)	
2	R(2,2)	R(2,2)
3	R(3,3)	R(3,3)
4	R(4,3)	R(4,4)
5	R(5,3)	R(5,4)

Computing Rombergs with Column Arrays

Another cell value replacement. This time we put R(4,4) into cell 4 in Array 1 and R(5,4) into cell 5 in Array 1.

	1	2
1	T(1,1)	
2	R(2,2)	R(2,2)
3	R(3,3)	R(3,3)
4	R(4,4)	R(4,4)
5	R(5,4)	R(5,4)

Computing Rombergs with Column Arrays

Finally, we compute R(5,5) in Array 2 from R(4,4) and R(5,4) in Array 1.

	1	2
1	T(1,1)	
2	R(2,2)	R(2,2)
3	R(3,3)	R(3,3)
4	R(4,4)	R(4,4)
5	R(5,4)	R(5,5)

And, we are done. My mental drummer has fallen asleep, so no drum roll.

How Many Rombergs to Compute?

One final touch on Romberg Integration. There is no need to compute all nodes in the Romberg Lattice. These lattices are beautiful, but computing their nodes can get expensive and numerically unstable. Beauty vs. Efficiency. Deep sigh...

As we compute the values in the two arrays (or the values in the Romberg Matrix if space is not an issue for us), we can compute the relative and/or absolute errors and, if the error level is acceptable, stop the computation and return the appropriate Romberg.

References

1. www.numpy.org
2. www.sympy.org
3. L. Goldstein, D. Lay, D. Schneider, N. Asmar. *Calculus and its Applications*.

P.S. Typos to Vladimir Kulyukin in Canvas.