A New Face of Romberg Integration
By
Namir Clement Shammas

Introduction
The Romberg method is among the popular numerical methods for integration. The algorithm is a composite one. It uses a basic integration method to give rough estimates for the integral. The method then performs extrapolations to significantly improve on these estimates. The algorithm runs in cycles, forming a lower triangular matrix, whose elements represent progressively refined values for the sought integral. The elements in the first column of the lower triangular matrix represent estimations of the integral using the trapezoidal rule—the basic integration algorithm. The rest of the matrix elements contain improvements for the integral values using Richardson's extrapolation. This extrapolation taps into elements in the neighboring matrix values. Each new matrix row brings with it better estimates for the sought integral. The number of rows in the matrix is related to the tolerance for the integral.

Article's Goal
This article answers the question, "Can we replace using the trapezoidal rule with other methods for estimating the integral? If so, what kinds of results do we get?" One may intuitively guess that any basic integration algorithm that is better than the trapezoidal rule. The article shows that such a guess may be true for certain basic integration algorithms and less true for others. It seems that the Richardson's extrapolation works well with certain, but not all, basic integration algorithms.

Instead of using extensive mathematical derivations, I will present the new modified Romberg methods using the listings of working Visual Basic code.

The Basic Romberg Method
To estimate the integral of function f(x) from a to b, the basic Romberg method uses the following general steps:

$$R(0,0) = \frac{(b-a)}{2}\,(f(b) + f(a))$$

$$R(n,0) = \frac{1}{2}\,R(n-1,0) + h_n \sum_{k=1}^{2^{n-1}} f(a + (2k-1)h_n$$

$$R(n,m) = \frac{1}{4^m-1}\,(4^m\,R(n,m-1) - R(n-1,m-1))$$

Where $h_n = (b-a)/2^n$
The step that calculates the element R(n,0) uses the trapezoidal rules to estimate the integral value. This article looks at other alternative for basic integration methods.

Implementation of the Basic Romberg Method

Let me present a Visual Basic (VBA Excel) function that implements the basic Romberg method:

```
Function MyFx(ByVal sExpress As String, ByVal sVarName As String, ByVal X As Double) As Double
  MyFx = Evaluate(Replace(sExpress, sVarName, "(" & CStr(X) & ")"))
End Function

Function RombergBasic(ByVal sExpress As String, ByVal sVarName As String, _
                      ByVal A As Double, ByVal B As Double, ByVal Toler As Double) As Double

  ' NOTE: The constants ROW0 and COL0 are offset values used to determine the lower values
  ' for the row and column indices of matrix R(,). These offsets are helpful when translating
  ' the VBA function into other BASIC dialects or languages that do not support zero indices
  ' for the rows and columns of a matrix.
  Const ROW0 = 1
  Const COL0 = 1
  Dim I As Integer, J As Integer, MaxCols As Integer, M As Long
  Dim R() As Double, h As Double, X As Double, Sum As Double
  Dim Func As String

  MaxCols = CInt(Abs(Log(Toler) / Log(10)))
  ReDim R(1 + ROW0, MaxCols + COL0)

  h = B - A
  R(ROW0, COL0) = h / 2 * (MyFx(sExpress, sVarName, A) + _
                     MyFx(sExpress, sVarName, B))
  For I = 1 To MaxCols
    h = h / 2
    Sum = 0
    For J = 1 To 2 ^ (I - 1)
      Sum = Sum + MyFx(sExpress, sVarName, A + (2 * J - 1) * h)
    Next J
    R(1 + ROW0, COL0) = R(ROW0, COL0) / 2 + h * Sum
    M = 1
    For J = 1 To I
      M = 4 * M
      R(1 + ROW0, J + COL0) = (M * R(1 + ROW0, J - 1 + COL0) - _
                              R(0 + ROW0, J - 1 + COL0)) / (M - 1)
    Next J
    For J = 0 To I
      R(0 + ROW0, J + COL0) = R(1 + ROW0, J + COL0)
    Next J
  Next I
  RombergBasic = R(0 + ROW0, MaxCols + COL0)
End Function
```

The function MyFX(ByVal sExpress As String, ByVal sVarName As String, ByVal X As Double) is a helper function. It evaluates the expression in parameter sExpress by replacing the variable names in that expression with the value of parameter X. The parameter sVarName specifies the name of the variable in the first parameter. Using this function, you can supply the expressions for the integrated function dynamically.

The function RombergBasic calculates the integral and has the following parameters:

- The parameter sExpress represents the expression for the function f(x).
- The parameter sVarName contains the name of the variable in the integrated function.
- The parameters A and B define the integral limits.
- The parameter Toler represents the tolerance. The function translates the value of this parameter into the number of columns, in the triangular Romberg matrix, required to calculate the integral.

Since the Romberg method uses data in the current and last matrix rows, all of the implemented functions maintain the data in only two rows of the matrix to reduce memory requirements.

The Romberg-Simpson Method

The first facelift that I will perform on the basic Romberg method is this. I will replace the trapezoidal methods with the Simpson one-third method. I will call this variant of the integration algorithm the Romberg-Simpson.

The method uses the following basic calculations:

$R(0,0) = \frac{(b-a)}{3}\left(f(b) + 4f((a+b)/2) + f(a)\right)$

$Sum(a,b,n,h) = \sum\left(f(a) + 4f(a+h) + 2f(a+2h) + \cdots + 4f(b-h) + f(b)\right)$

$R(n,0) = (R(n-1,0) + h\,Sum(a,b,n,h)/4$

$R(n,m) = \frac{1}{4^m-1}\left(4^m R(n,m-1) – R(n-1,m-1)\right)$

Where $h = (b-a)/2^n$

Here is the Visual Basic implementation for the Romberg-Simpson method:

```
Function RombergSimpson(ByVal sExpress As String, ByVal sVarName As String, _
                ByVal A As Double, ByVal B As Double, ByVal Toler As Double) As Double

  ' Romberg's method variant that uses Simpson's rule instead of trapezoidal integration
  '
  ' Examples for calling this function are:
  '
  ' 1) RombergSimpson("1/X", "X", 1, 2, 1E-8) returns the value for ln(2)
  '
  ' 2) RombergSimpson("exp(X)", "X", 0, 1, 1E-8) returns the value for exp(1)-1
  '
  ' 3) RombergSimpson("EXP($X)", "$X", 0, 1, 1E-8) returns the value for exp(1)-1
  '
  ' Note that the second example uses the variable name of $X instead of X, because the
  ' letter X also appears in the name of the exponential function EXP. Thus, using the name
  ' $X yields the correct result.
  '

  ' NOTE: The constants ROW0 and COL0 are offset values used to determine the lower values
  ' for the row and column indices of matrix R(,). These offsets are helpful when translating
  ' the VBA function into other BASIC dialects or languages that do not support zero indices
```

```
  ' for the rows and columns of a matrix.
  Const ROW0 = 1
  Const COL0 = 1
  Dim I As Integer, J As Integer, MaxCols As Integer, M As Long, K As Integer
  Dim R() As Double, h As Double, X As Double, Sum As Double, h2 As Double

  MaxCols = CInt(Abs(Log(Toler) / Log(10)))
  ReDim R(1 + ROW0, MaxCols + COL0)

  h = (B - A) / 2
  R(ROW0, COL0) = h / 3 * (MyFx(sExpress, sVarName, A) + 4 * MyFx(sExpress, sVarName, A + h) + _
                     MyFx(sExpress, sVarName, B))
  For I = 1 To MaxCols
    h = h / 2
    ' Note: The next statement has a programming trick. It subtracts fx(B) from the sum so that
    ' when the loop adds 2*f(B) to the sum, the latter will have the correct value of:
    '
    ' Sum = fx(A) + 4 * fx(A+h) + 2 * fx(A+2h) + ... + 4 * fx(B-h) + fx(B)
    '
    Sum = MyFx(sExpress, sVarName, A) - MyFx(sExpress, sVarName, B)
    X = A + h
    Do While X < B
      Sum = Sum + 4 * MyFx(sExpress, sVarName, X) + 2 * MyFx(sExpress, sVarName, X + h)
      X = X + 2 * h
    Loop
    R(1 + ROW0, COL0) = (R(ROW0, COL0) + h * Sum) / 4
    M = 1
    For J = 1 To I
      M = 4 * M
      R(1 + ROW0, J + COL0) = (M * R(1 + ROW0, J - 1 + COL0) - _
                               R(0 + ROW0, J - 1 + COL0)) / (M - 1)
    Next J
    For J = 0 To I
      R(0 + ROW0, J + COL0) = R(1 + ROW0, J + COL0)
    Next J
  Next I
  RombergSimpson = R(0 + ROW0, MaxCols + COL0)
End Function
```

☛ In this article, I use red fonts to highlight relevant code in the various functions.

The function RombergSimpson has the same parameters as function RombergBasic. I use the constants ROW0 and COL0 as offsets for the row and column indices of the Romberg matrix. Since Visual Basic handles matrices that are either zero-based or one-based, setting these constants to 0 or 1 is fine. I put these index offset constants so that if you want to translate the code to other languages that support one-based array/matrix indexing (like Matlab) then you can easily translate the code. Simply replace ROW0 and COL0 with 1 in your translated code.

Notice the following statement located before the main For loop:

```
  R(ROW0, COL0) = h / 3 * (MyFx(sExpress, sVarName, A) + 4 * MyFx(sExpress, sVarName, A + h) + _
                     MyFx(sExpress, sVarName, B))
```


The above statement initializes the first element in the Romberg matrix by applying the basic Simpson's rule. The function uses the following statements to calculate better approximations for the integral using Simpson's rule:

```
    Sum = MyFx(sExpress, sVarName, A) - MyFx(sExpress, sVarName, B)
    X = A + h
    Do While X < B
      Sum = Sum + 4 * MyFx(sExpress, sVarName, X) + 2 * MyFx(sExpress, sVarName, X + h)
      X = X + 2 * h
    Loop
```

The above code implements a composite version of Simpson's rule that calculates the integral using more than three points. Finally, notice the following statement:

```
R(1 + ROW0, COL0) = (R(ROW0, COL0) + h * Sum) / 4
```

The above statement calculates the value for element R(1+ROW0, COL0) using the values of R(ROW0, COL0) and the calculated Simpson integral sum. The expression averages the two integrals (in R(ROW0,COL0) and h/3*Sum) using unequal weights. The expression assigns a weight of 1 to R(ROW0,COL0) and a weight of 3 to the Simpson's rule result (and that is why the statement shows h * Sum instead of h/3*Sum). The result is divided by 4 which is the sum of the weights. I have found that these weights give better results than using equal weights.

The Extended Romberg-Simpson Method

The second facelift that I will perform on the basic Romberg method is one that uses the basic Simpson's rule as well as the Alternative Extended Simpson's rule.

Here is the Visual Basic implementation for the Extended Romberg-Simpson method:

```
Function RombergSimpsonEx(ByVal sExpress As String, ByVal sVarName As String, _
                 ByVal A As Double, ByVal B As Double, ByVal Toler As Double) As Double

  ' Romberg's method variant that uses Simpson's rule and the Alternative extended Simpson's rule
  ' instead of trapezoidal integration
  '
  ' Examples for calling this function are:
  '
  ' 1) RombergSimpsonEx("1/X", "X", 1, 2, 1E-8) returns the value for ln(2)
  '
  ' 2) RombergSimpsonEx("exp(X)", "X", 0, 1, 1E-8) returns the value for exp(1)-1
  '
  ' 3) RombergSimpsonEx("EXP($X)", "$X", 0, 1, 1E-8) returns the value for exp(1)-1
  '
  ' Note that the second example uses the variable name of $X instead of X, because the
  ' letter X also appears in the name of the exponential function EXP. Thus, using the name
  ' $X yields the correct result.
  '

  ' NOTE: The constants ROW0 and COL0 are offset values used to determine the lower values
  ' for the row and column indices of matrix R(,). These offsets are helpful when translating
  ' the VBA function into other BASIC dialects or languages that do not support zero indices
  ' for the rows and columns of a matrix.
  Const ROW0 = 1
  Const COL0 = 1
```

```
  Dim I As Integer, J As Integer, MaxCols As Integer, M As Long, K As Integer
  Dim R() As Double, h As Double, X As Double, Sum As Double, h2 As Double
  Dim N As Integer

  MaxCols = CInt(Abs(Log(Toler) / Log(10)))
  ReDim R(1 + ROW0, MaxCols + COL0)

  N = 2
  h = (B - A) / 2
  R(ROW0, COL0) = h / 3 * (MyFx(sExpress, sVarName, A) + 4 * MyFx(sExpress, sVarName, A + h) + _
                     MyFx(sExpress, sVarName, B))
  For I = 1 To MaxCols
    N = 2 + N
    h = h / 2
    ' Note: The next statement has a programming trick. It subtracts fx(B) from the sum so that
    ' when the loop adds 2*f(B) to the sum, the latter will have the correct value of:
    '
    ' Sum = fx(A) + 4 * fx(A+h) + 2 * fx(A+2h) + ... + 4 * fx(B-h) + fx(B)
    '
    If CInt((B - A) / h - 0.5) < 8 Then
      Sum = MyFx(sExpress, sVarName, A) - MyFx(sExpress, sVarName, B)
      X = A + h
      Do While X < B
        Sum = Sum + 4 * MyFx(sExpress, sVarName, X) + 2 * MyFx(sExpress, sVarName, X + h)
        X = X + 2 * h
      Loop
      R(1 + ROW0, COL0) = (R(ROW0, COL0) + h * Sum) / 4
    Else
      ' use the alternative extended Simpson's rule
      Sum = 17 * (MyFx(sExpress, sVarName, A) + MyFx(sExpress, sVarName, B))
      Sum = Sum + 59 * (MyFx(sExpress, sVarName, A + h) + MyFx(sExpress, sVarName, B - h))
      Sum = Sum + 43 * (MyFx(sExpress, sVarName, A + 2 * h) + MyFx(sExpress, sVarName, B-2 * h))
      Sum = Sum + 49 * (MyFx(sExpress, sVarName, A + 3 * h) + MyFx(sExpress, sVarName, B-3 * h))
      X = 4 * h + A
      Do While X < (B - 3 * h)
        Sum = Sum + 48 * MyFx(sExpress, sVarName, X)
        X = X + h
      Loop
      R(1 + ROW0, COL0) = (R(ROW0, COL0) + h * Sum) / 49
    End If

    M = 1
    For J = 1 To I
      M = 4 * M
      R(1 + ROW0, J + COL0) = (M * R(1 + ROW0, J - 1 + COL0) - _
                              R(0 + ROW0, J - 1 + COL0)) / (M - 1)
    Next J
    For J = 0 To I
      R(0 + ROW0, J + COL0) = R(1 + ROW0, J + COL0)
    Next J
  Next I
  RombergSimpsonEx = R(0 + ROW0, MaxCols + COL0)
End Function
```

The function RombergSimpsonEx has the same parameters as the first two Romberg functions that I presented. This function is an extension of RombergSimpson. When the number of rows reaches 8, the function switches from using the basic Simpson's one-third rule to the Alternative Extended Simpson's rule. Once the function calculates the area sum for this algorithm, it calculates the new value in the Romberg matrix using:

```
R(1 + ROW0, COL0) = (R(ROW0, COL0) + h * Sum) / 49
```


Notice that the value for the Romberg matrix element is calculated using R(ROW0, COL0) and the calculated Simpson integral sum. The above equation assigns a weight of 1 to R(ROW0, COL0) and a weight of 48 to the calculated sum. As with function RombergSimpson, I have found that these weights give better results than using equal weights.

The Romberg-Gauss Method

Another variant of the Romberg method that I studied is one where I replaced the trapezoidal rule with Legendre-Gaussian quadrature. I will call this method Romberg-Gauss. Combining two versatile methods should yield interesting results. You may say that the algorithm is an overkill, since the Gaussian quadrature by itself can do a good job in calculating an integral. My motive for the Romberg-Gauss method is mainly driven by curiosity.

The first step in using Gaussian quadrature involves writing a function that performs the integration for a varying number of points. Currently, the function supports up to 11 points, which should be adequate for our purposes. Here is the listing of the function GaussQuad:

```
Function GaussQuad(ByVal sExpress As String, ByVal sVarName As String, _
                   ByVal N As Integer, ByVal A As Double, ByVal B As Double) As Double
  Const MAX = 11
  Dim Wt(MAX) As Double, X(MAX) As Double
  Dim Sum As Double, h1 As Double, h2 As Double
  Dim I As Integer

  If N > MAX Then N = MAX
  h1 = (B - A) / 2
  h2 = (B + A) / 2
  Select Case N
    Case 2
      Wt(1) = 1
      Wt(2) = 1
      X(1) = -0.577350269189626
      X(2) = -X(1)

     Case 3
       Wt(1) = 0.888888888888889
       X(1) = 0
       Wt(2) = 0.555555555555556
       X(2) = -0.774596669241483
       Wt(3) = Wt(2)
       X(3) = -X(2)

     Case 4
       Wt(1) = 0.652145154862546
       X(1) = 0.339981043584856
       Wt(2) = 0.347854845137454
       X(2) = 0.861136311594053
       For I = 3 To 4
         Wt(I) = Wt(I - 2)
         X(I) = -X(I - 2)
       Next I


     Case 5
       Wt(1) = 0.568888888888889
```

```
    X(1) = 0
    Wt(2) = 0.478628670499367
    X(2) = 0.538469310105683
    Wt(3) = 0.236926885056189
    X(3) = 0.906179845938664
    For I = 4 To 5
      Wt(I) = Wt(I - 2)
      X(I) = -X(I - 2)
    Next I

  Case 6
    Wt(1) = 0.360761573048139
    X(1) = 0.661209386466264
    Wt(2) = 0.467913934572691
    X(2) = 0.238619186083197
    Wt(3) = 0.17132449237917
    X(3) = 0.932469514203152
    For I = 4 To 6
      Wt(I) = Wt(I - 3)
      X(I) = -X(I - 3)
    Next I

  Case 7
    Wt(1) = 0.417959183673469
    X(1) = 0
    Wt(2) = 0.381830050505119
    X(2) = 0.405845151377397
    Wt(3) = 0.279705391489277
    X(3) = 0.741531185599394
    Wt(4) = 0.12948496616887
    X(4) = 0.949107912342758
    For I = 5 To 7
      Wt(I) = Wt(I - 3)
      X(I) = -X(I - 3)
    Next I

  Case 8
    Wt(1) = 0.362683783378362
    X(1) = 0.18343464249565
    Wt(2) = 0.313706645877887
    X(2) = 0.525532409916329
    Wt(3) = 0.222381034453375
    X(3) = 0.796666477413627
    Wt(4) = 0.101228536290376
    X(4) = 0.960289856497536
    For I = 5 To 8
      Wt(I) = Wt(I - 4)
      X(I) = -X(I - 4)
    Next I

  Case 9
    Wt(1) = 0.33023935500126
    X(1) = 0
    Wt(2) = 0.180648160694857
    X(2) = 0.836031107326636
    Wt(3) = 8.12743883615744E-02
    X(3) = 0.968160239507626
    Wt(4) = 0.312347077040003
    X(4) = 0.324253423403809
    Wt(5) = 0.260610696402935
    X(5) = 0.61337143270059
    For I = 6 To 9
      Wt(I) = Wt(I - 4)
      X(I) = -X(I - 4)
    Next I

  Case 10
    Wt(1) = 0.295524224714753
    X(1) = 0.148874338981631
    Wt(2) = 0.269266719309996
    X(2) = 0.433395394129247
```



```
      Wt(3) = 0.219086362515982
      X(3) = 0.679409568299024
      Wt(4) = 0.149451349150581
      X(4) = 0.865063366688985
      Wt(5) = 6.66713443086881E-02
      X(5) = 0.973906528517172
      For I = 6 To 10
        Wt(I) = Wt(I - 5)
        X(I) = -X(I - 5)
      Next I

    Case 11
      Wt(1) = 0.272925086777901
      X(1) = 0
      Wt(2) = 0.262804544510247
      X(2) = 0.269543155952345
      Wt(3) = 0.233193764591991
      X(3) = 0.519096129206812
      Wt(4) = 0.186290210927734
      X(4) = 0.730152005574049
      Wt(5) = 0.125580369464905
      X(5) = 0.887062599768095
      Wt(6) = 5.56685671161737E-02
      X(6) = 0.978228658146057
      For I = 7 To 11
        Wt(I) = Wt(I - 5)
        X(I) = -X(I - 5)
      Next I

  End Select

  Sum = 0
  For I = 1 To N
    Sum = Sum + Wt(I) * MyFx(sExpress, sVarName, h1 * X(I) + h2)
  Next I
  GaussQuad = h1 * Sum
End Function
```

The function GaussQuad has parameters similar to the previous Romberg functions. The Toler parameter does not exist in function GaussQuad. Conceptually, this missing parameter is replaced with the integer-typed parameter, N, that specifies the number of integration points. The function uses a Select statement to zoom in on the sought weights and abscissa points used in Gaussian quadrature.

The function that implements the Romberg-Gauss method is:

```
Function RombergGauss(ByVal sExpress As String, ByVal sVarName As String, _
                ByVal A As Double, ByVal B As Double, ByVal Toler As Double) As Double

  ' Romberg's method variant that uses Simpson's rule and the Alternative extended Simpson's rule
  ' instead of trapezoidal integration
  '
  ' Examples for calling this function are:
  '
  ' 1) RombergGauss("1/X", "X", 1, 2, 1E-8) returns the value for ln(2)
  '
  ' 2) RombergGauss("exp(X)", "X", 0, 1, 1E-8) returns the value for exp(1)-1
  '
  ' 3) RombergGauss("EXP($X)", "$X", 0, 1, 1E-8) returns the value for exp(1)-1
  '
  ' Note that the second example uses the variable name of $X instead of X, because the
  ' letter X also appears in the name of the exponential function EXP. Thus, using the name
  ' $X yields the correct result.
```

```
  '
  ' NOTE: The constants ROW0 and COL0 are offset values used to determine the lower values
  ' for the row and column indices of matrix R(,). These offsets are helpful when translating
  ' the VBA function into other BASIC dialects or languages that do not support zero indices
  ' for the rows and columns of a matrix.
  Const ROW0 = 1
  Const COL0 = 1
  Dim I As Integer, J As Integer, MaxCols As Integer, M As Long, K As Integer
  Dim R() As Double, h As Double, X As Double, Sum As Double, h2 As Double
  Dim N As Integer

  MaxCols = CInt(Abs(Log(Toler) / Log(10)))
  ReDim R(1 + ROW0, MaxCols + COL0)

  N = 2
  h = (B - A) / 2
  R(ROW0, COL0) = GaussQuad(sExpress, sVarName, N, A, B)

  For I = 1 To MaxCols
    N = N + 1
    R(1 + ROW0, COL0) = GaussQuad(sExpress, sVarName, N, A, B)
    M = 1
    For J = 1 To I
      M = 4 * M
      R(1 + ROW0, J + COL0) = (M * R(1 + ROW0, J - 1 + COL0) - R(0 + ROW0, J - 1 + COL0)) / (M -
1)
    Next J
    For J = 0 To I
      R(0 + ROW0, J + COL0) = R(1 + ROW0, J + COL0)
    Next J
  Next I
  RombergGauss = R(0 + ROW0, MaxCols + COL0)
End Function
```

The function RombergGauss calls the function GaussQuad to obtain very good estimates of the integral. Notice that the statement which calculates R(1 + ROW0, COL0) simply calls the function GaussQuad. It does not use other elements in the Romberg matrices to calculate R(1 + ROW0, COL0). I have found this approach to yield better results.

Test Scores!

It's time to separate the men from the boys, the winners from the losers! So how do the different methods stack up against each other? Table 1 shows the test results for various integrations. The table also includes columns that show the exact solution and the results of using 11-point and 6-point Legendre-Gauss quadrature (by calling function GaussQuad). The results are color coded—yellow for first place, green for second, and blue for third.

Problem	*Exact Value*	*RombergBasic*	*RombergSimpson*	*RombergSimpsonEx*	*RombergGauss*	*Legendre-Gauss-11*	*Legendre-Gauss-6*	Color Scheme
1/x from 1 to 2	0.693147181	0.693147181	0.693147181	0.693147181	0.693147181	0.693147181	0.69314718	1st rank
1/x from 1 to 10	2.302585093	2.302585093	2.302585093	2.302585094	2.30258458	2.302583355	2.301408084	2nd Rank
1/x from 1 to 100	4.605170186	4.605320986	4.605173699	4.605070339	4.539591105	4.550142068	4.230412779	3rd Rank
ln(x)/x from 1 to 10	2.650949055	2.650949055	2.650949055	2.650949055	2.650949055	2.650949055	2.650949055	
ln(x)/x from 1 to 100	10.603796221	10.60378807	10.60398559	10.68648172	10.67441869	10.83360554	10.60378807	
sin(x) from 0 to 1	0.459697694	0.459697694	0.459697694	0.459697694	0.459697694	0.459697694	0.459697694	
sin(x)/x from 1e-10 to pi/4	0.758975881	0.758975881	0.758975881	0.761068293	0.758975881	0.758975881	0.758975881	

Problem	Exact Value	RombergBasic	RombergSimpson	RombergSimpsonEx	RombergGauss	Legendre-Gauss-11	Legendre-Gauss-6	Color Scheme
sin(x)cos(x) from 0 to 1	0.354036709	0.354036709	0.354036709	0.354036709	0.354036709	0.354036709	0.354036709	
ln(x)/x^2 from 1 to 2	0.15342641	0.15342641	0.15342641	0.15342641	0.15342641	0.15342641	0.153426421	
ln(x)/x^2 from 1 to 10	0.669741491	0.669741491	0.669741491	0.669741485	0.66975646	0.669761624	0.674774153	
ln(x)/x^2 from 1 to 100	0.943948298	0.943066528	0.943917809	0.94437022	1.008005008	1.002756148	0.919835967	

Table 1. The test results.

Examining Table 1 we observe the following:

1. The function RombergSimpson has performed the best in most cases.
2. The function RombergBasic comes second rank overall.
3. The function RomberSimpsonEx comes third rank overall.
4. The Gaussian quadrature also performed well. This should not come as a total surprise as the Legendre-Gauss quadrature is one of the main rivals of the Romberg integration method.

We conclude that:

1. Incorporating the Simpson’s one-third rule with the Romberg method enhances the algorithm in general. The improvement comes as a moderate additional effort in computing.
2. The basic Romberg method is still a viable method since it has outdone most of the variant algorithms.
3. The Extended Romberg-Simpson also shows some promise, albeit at some additional computing effort.
4. Incorporating the Gaussian quadrature with the Romberg algorithm does not give the result method sustainable advantage. My hunch as to why this lack of advantage occurs is that the quadrature results do not work well with the Richardson extrapolation.

Prologue

In 2013 I was in communication with Graeme Dennes, of Melbourne, Australia. He offered several suggestions to improve the Romberg-Simpson method. Graeme focused on including VBA statements that detect the conditions for earlier termination of the Romberg iterations. In addition, he implemented a general-purpose API-based Timer in VBA and used it to time the integration calculations. In March 2014, Graeme submitted an new improved version of his code which included the following changes:

- The exit criteria has been enhanced to achieve further improvements in speed and accuracy.
- The outputs now show the number of function evaluations as a more useful performance metric for comparison purposes.

- The Excel file has 200 test functions for even more diversity for comparison purposes

Table 1 shows a partial view of Graeme's latest Excel file that performs integration on many functions using his code.

ROMBERG QUADRATURE
AND FUNCTION PLOTTING CHART
Finite Interval (a,b)

UDF Name: QUAD_ROMBERG_GD

CLICK THE ORANGE BUTTON	
CLICK BUTTON TO SWITCH BETWEEN "SHOW RESULTS" AND "HIDE RESULTS"	SHOW /HIDE CLICK TO HIDE RESULTS

FUNCTION PLOTTER							
CLICK THE "CLICK TO SHOW CHART" BUTTON, THEN SELECT FUNCTION WITH UP/DOWN BUTTONS						SELECT FUNCTION	SHOW / HIDE
No.	Cell Function Plotted	Name	Variable	a	b	▲ ▼	CLICK TO SHOW CHART
4							

OVERALL PROGRAM PERFORMANCE			
Total Func Evals:	Total Time in Seconds:	Total True Error:	Total Number of Correct Digits:
542,624	48.185	0.0128	2475 of 3000 (82.5%)
↓↓↓	↓↓↓	↓↓↓	↓↓↓

	ROMBERG PROGRAM INPUTS					TRUE	ROMBERG PROGRAM OUTPUTS						
No.	Cell Function	Name	Variable	a	b	INTEGRAL	Integral	Est. Error	Func Evals	Time (secs)	True Error	Correct Digits	Message
1	1/SQRT(x)	FINITE_1	x	0	1	2	**2.00000000000000**	1.02E-13	255	0.015332	1.11E-16	15	
2	SQRT(4-x^2)	FINITE_2	x	0	2	3.14159265358979	**3.14159265358979**	7.69E-14	255	0.015685	1.41E-16	15	
3	LN(x)	FINITE_3	x	0	1	-1	**-1.00000002065983**	6.20E-08	8191	0.487040	2.07E-08	7	
4	x*LN(x)	FINITE_4	x	0	1	-0.25	**-0.250000000000032**	6.22E-13	4095	0.319614	1.26E-13	12	
5	LN(x)/SQRT(x)	FINITE_5	x	0	1	-4	**-3.99964289999447**	8.93E-05	8191	0.814806	8.93E-05	4	
6	4/(1+x^2)	FINITE_6	x	0	1	3.14159265358979	**3.14159265358979**	7.07E-16	511	0.050082	5.65E-16	15	
7	SIN(x)^4*COS(x)^2	FINITE_7	x	0	1.57079632679490	0.0981747704246810	**0.0981747704246809**	9.66E-14	511	0.033268	1.27E-15	14	
8	COS(x)	FINITE_8	x	0	3.14159265358979	0	**-3.67602986263905E-16**	7.75E+00	8191	0.734519	3.68E-16	15	
9	COS(LN(x))	FINITE_9	x	0	1	0.5	**0.500000120614689**	7.86E-08	8191	0.715191	2.41E-07	6	
10	SQRT(4*x-x^2)	FINITE_10	x	0	2	3.14159265358979	**3.14159265358979**	7.69E-14	255	0.030071	2.83E-16	15	
11	5*x^2	FINITE_11	x	0	10	1666.66666666667	**1666.66666666667**	0.00E+00	63	0.008770	0.00E+00	15	
12	x^0.125	FINITE_12	x	0	1	0.888888888888889	**0.888888888635416**	1.07E-09	8191	0.705251	2.85E-10	9	
13	1/x	FINITE_13	x	1	10	2.30258509299405	**2.30258509299404**	3.45E-14	1023	0.077601	9.64E-16	15	
14	LN(x)/(1-x)	FINITE_14	x	0.5	1	-0.582240526465013	**-0.582240526465013**	4.98E-14	255	0.017951	0.00E+00	15	
15	EXP(-1/COS(x))	FINITE_15	x	0	1.04719755119660	0.307694394903451	**0.307694394903450**	2.71E-15	511	0.039261	2.89E-15	14	
16	(x*(x+88)*(x-88)*(x+47)*(x-47)*(x+117)*(x-117))^2	FINITE_16	x	0	128	6.55134477611335E+27	**6.55134477611343E+27**	8.11E-14	1023	0.055648	1.26E-14	13	
17	EXP(-(x^2))	FINITE_17	x	0	100	0.886226925452758	**0.886226925452758**	2.34E-15	4095	0.334456	5.01E-16	15	
18	2*x^2/(x+1)/(x-1)-x/LN(x)	FINITE_18	x	0	1	0.0364899739785776	**0.0364899739785732**	4.78E-13	2047	0.268353	1.18E-13	12	
19	x*LN(1+x)	FINITE_19	x	0	1	0.25	**0.250000000000002**	1.48E-13	255	0.028800	7.77E-15	14	
20	x^2*ATAN(x)	FINITE_20	x	0	1	0.210657251225807	**0.210657251225807**	2.02E-15	511	0.040849	5.27E-16	15	
21	EXP(x)*COS(x)	FINITE_21	x	0	1.57079632679490	1.90523869048268	**1.90523869048268**	5.83E-16	511	0.045262	1.05E-15	14	
22	ATAN(SQRT(x^2+2))/(1+x^2)/SQRT(x^2+2)	FINITE_22	x	0	1	0.514041895890071	**0.514041895890071**	6.48E-16	511	0.035798	6.48E-16	15	
23	LN(x)*SQRT(x)	FINITE_23	x	0	1	-0.444444444444444	**-0.444444444440236**	6.63E-11	8191	0.533775	9.47E-12	11	
24	SQRT(1-x^2)	FINITE_24	x	0	1	0.785398163397448	**0.785398163397447**	7.66E-14	255	0.016194	1.27E-15	14	
25	SQRT(x)/SQRT(1-x^2)	FINITE_25	x	0	1	1.19814023473559	**1.19814023473560**	2.38E-13	255	0.048248	4.26E-15	14	
26	LN(x)^2	FINITE_26	x	0	1	2	**2.00000074421026**	1.00E-06	8191	0.666670	3.72E-07	6	

Table 2. Graeme Graeme's latest Excel test file showing the results of integration of several test functions.

Graeme includes an orange-colored button that allows you to toggle between performing the calculations and showing them, and hiding the results. You can of course change some of the input values in the leftmost columns. The tested functions start appearing in row 16. Graeme's code contains the following modules:

- The **m_High_Res_Timer** module.
- The **Quad_QUAD_ROMBERG** module.

Here is the listing for the **m_High_Res_Timer** module:

```
Option Explicit


Private Declare PtrSafe Function QueryPerformanceCounter Lib "kernel32" (ByRef x As Currency) As
Long

Private Declare PtrSafe Function QueryPerformanceFrequency Lib "kernel32" (ByRef x As Currency)
As Long


Public Function MicroTimer() As Double
```

```
    Dim Frequency As Currency, Counter As Currency

    MicroTimer = 0

    If Not Frequency Then QueryPerformanceFrequency Frequency

    If Frequency Then QueryPerformanceCounter Counter Else Exit Function

    MicroTimer = Counter / Frequency

End Function
```

Here is the listing for the **Quad_QUAD_ROMBERG** module:

```
Option Explicit


' Romberg integrator by Graeme Dennes (Melbourne, Australia). Released 2014-03-31
'
' Based on the Composite Midpoint Rule (the end points are not used), enabling
' the end points, (a,b), to be located at discontinuities without causing problems.
'
' Func is the string variable with the function to be integrated
' intvar is the string holding the integrating variable
' a and b are the lower and upper limits, respectively, of the integration interval.



Function QUAD_ROMBERG_GD(Func As String, intvar As String, a As Double, b As Double) As Variant

    Dim M(0 To 12, 0 To 12) As Double, Result(1 To 4) As Variant, Row_Index As Long
    Dim Col_Index As Long, j As Long, k As Long, MaxLoops As Long
    Dim errval As Double, h As Double, LowestErr As Double, parm1 As Double
    Dim parm2 As Double, sum As Double, tol As Double, u As Double, x As Double
    Dim ExitFlag As Boolean, one As Double, three As Double

    ' Error handling
    On Error Resume Next

    ' Get the start time
    Result(4) = MicroTimer

    ' Set program tolerance
    tol = 10 ^ -12

    ' Set some values
    MaxLoops = 12
    LowestErr = 1
    ExitFlag = False
    one = 1
    three = 3

    'Start the process
    parm1 = (b - a) / 4
    parm2 = (b + a) / 2
    k = -1
    h = 4
    Do
        Do
            k = k + 1
            h = h / 2
            u = -1 + h / 2
            sum = 0
            Do
                x = parm2 + (parm1 * u * (three - (u * u)))
                sum = sum + (one - (u * u)) * Evaluate(Replace(Func, intvar, x))
                u = u + h
            Loop While u < one

            M(k, 0) = 3 * parm1 * h * sum

        Loop While k = 0

        For j = 1 To k
            M(k, j) = M(k, j - 1) + (M(k, j - 1) - M(k - 1, j - 1)) / (4 ^ j - 1)
        Next j
```

```
        ' EXIT TEST 1:   For k = 2-4, check for diagonal match.
        '                Exit with diagonal element
        If k >= 2 And k <= 4 Then
            LowestErr = Abs((M(k, k) - M(k - 1, k - 1)) / M(k, k))
            If LowestErr <= tol Then
                Row_Index = k
                Col_Index = k                    ' return diagonal element and error
                Exit Do                          ' and exit
            End If
        End If


        ' EXIT TEST 2:   For k >= 5, check for weighted error match.
        '                Exit with the column/diagonal element with best match
        If k >= 5 Then
            For j = 1 To k
                errval = Abs(((M(k - 1, j - 1) + 2 * M(k, j - 1)) / 3 - M(k, j)) / M(k, j))
                If errval <= tol Then
                    ExitFlag = True              ' so set the flag
                    If errval < LowestErr Then
                        LowestErr = errval       ' save smallest error value from all matches in the row
                        Row_Index = k
                        Col_Index = j            ' save the column index of lowest error in the row
                    End If
                End If                           ' check for further matches, just in case they exist,
            Next                                 ' then exit the loop with the best match in the row
            If ExitFlag Then Exit Do             ' if flag = true then exit with element and its error
        End If


        ' EXIT TEST 3:  For k >= 8, check for column match.
        '               Exit with column element with best match
        If k >= 8 Then
            For j = 0 To k - 1
                errval = Abs((M(k, j) - M(k - 1, j)) / M(k, j))        ' check column convergence
                If errval <= tol Then
                    ExitFlag = True              ' so set the flag
                    If errval < LowestErr Then
                        LowestErr = errval       ' save smallest error value from all matches in the row
                        Row_Index = k
                        Col_Index = j            ' save the column index of lowest error in the row
                    End If
                End If                           ' check for further matches, just in case they exist,
            Next                                 ' then exit the loop with the best match in the row
            If ExitFlag Then Exit Do             ' if flag = true then exit with element and its error
        End If


        ' EXIT TEST 4:  For k = maxloops, exit with final diagonal element

        If k = MaxLoops Then
            LowestErr = Abs((M(k, k) - M(k - 1, k - 1)) / M(k, k))
            Row_Index = k
            Col_Index = k                    ' select the diagonal element
        End If

    Loop While k < MaxLoops                  ' else try next loop

    Result(1) = M(Row_Index, Col_Index)      ' so load the integral result
    Result(2) = LowestErr                    ' and its error value
    Result(3) = 2 ^ (k + 1) - 1              ' fx evals
    Result(4) = MicroTimer - Result(4)

    QUAD_ROMBERG_GD = Result

End Function
```

Conclusion

Replacing the trapezoidal rule with the simplest version of Simpson's rule gives the Romberg method an added advantage in many cases. Romberg's method now has a new face to go by—the Romberg-Simpson method!