The New Interpolative Trisection Root-Seeking Algorithms

by
Namir Shammas

Introduction

The concept of root-seeking using root-bracketing is not new. What is new in this article is combining root-bracketing and quadratic/cubic interpolation to ensure that the root-seeking iterations quickly achieve their goal. Simple root-seeking methods are guaranteed to work if the basic assumptions hold. By contrast, unconstrained interpolation may or may not work depending on the targeted function as well as the number and values of the points used in the interpolation. The two schemes together produce better result than each one by itself.

This article presents a new set of root-bracketing algorithms. The new algorithms, are named the Interpolative Trisection methods. They compete with and enhance the Bisection method which is the slowest root-seeking method. They also compete with the Trisection Plus algorithm[6] which I recently developed. I present two variants of the Interpolative Trisection algorithms—one that uses quadratic interpolation and the other applies cubic interpolation.

Since the Trisection Plus method[6] is based on the Bisection method, I will present a brief discussion for the Bisection method. In addition, since I am comparing the new algorithms with Newton's method and the Trisection Plus[6] algorithm, I will also discuss these two methods. If you are familiar with any or all of these algorithms, you can skip over the related sections.

The Bisection Algorithm

There are numerous algorithms that calculate the roots of single-variable nonlinear functions. The most popular of such algorithms is Newton's method. The slowest and simplest root seeking algorithm is the Bisection method. This method has the user select an interval that contains the sought root. The method iteratively shrinks the root-bracketing interval to zoom in on the sought root. Here is the pseudo-code for the Bisection algorithm:

```
Given f(x)=0, the root-bracketing interval [A,B], and the tolerance, Toler,
for the root of f(x):
   • Calculate Fa = f(A) and Fb = f(B).
   • Exit if Fa*Fb > 0.
   • Repeat
         ○ X =(A+B)/2
         ○ Fx = f(X)
         ○ If Fx*Fa > 0 then
               ▪ A = X
               ▪ Fa = Fx
         ○ Else
               ▪ B = X
               ▪ Fb = Fx
         ○ End
   • Until |A-B| < Toler
   • Return root as (A+B)/2
```

The above pseudo-code shows how the algorithm iteratively halves the root-bracketing until it zooms on the root. The Bisection method is the slowest converging method. It's main virtue is that it is guaranteed to work if f(x) is continuous in the interval [A, B] and f(A)×f(B) is negative.

Newton's Method

I will also compare the new algorithms with Newton's method. This comparison serves as an upper limit test. I am implementing Newton's method based on the following pseudo-code:

```
Given f(x)=0, the root-bracketing interval [A,B], and the tolerance, Toler,
for the root of f(x):
   • Calculate X = (A+B)/2
   • Repeat
         ○ h = 0.001 * (|X| + 1)
         ○ Fx = f(X)
         ○ Diff = h * Fx / (f(X+h) – Fx)
         ○ X = X – Diff
   • Until |Diff| < Toler
   • Return root as X
```

The above code shows that the implementation of Newton's method starts with the same interval [A, B] that is already available for the root-bracketing methods. Thus, the algorithm derives its single initial guess as the midpoint of that interval.

The Trisection Algorithm

The Trisection algorithm has each iteration divide the root-bracketing interval [A, B] into three parts, instead of two as does the Bisection. The algorithm chooses the first point X1 within the interval [A, B] closest to the end point A, or B, that has the smallest absolute function value (call this point Z). This strategy hopes that

f(X1) would have a sign opposite that of f(Z). If this condition is true, then the iteration has finished its task. If not, the algorithm calculates X2 which lies closer to the other interval end point (call it Y). The algorithm then determines whether the interval [X1, X2] or [X2, Y] is the new root-bracketing interval. The values of the interval [A. B] are then updated accordingly. Here is the pseudo-code for the Trisection algorithm:

```
Given f(x)=0, the root-bracketing interval [A,B], and the tolerance, Toler,
for the root of f(x):
   • Calculate Fa = f(A) and Fb = f(B).
   • Exit if Fa*Fb > 0.
   • Repeat
         ○ If |Fa| < |Fb| then
               ▪ X1 = A + (B-A)/3
               ▪ Fx1 = f(X1)
               ▪ If Fa*Fx1 < 0 then
                     • B = X1
                     • Fb = Fx1
               ▪ Else
                     • X2 = B – (B-A)/3
                     • Fx2 = f(X2)
                     • If Fx1*Fx2 < 0 then
                           ○ A = X1
                           ○ Fa = Fx1
                           ○ B = X2
                           ○ Fb = Fx2
                     • Else
                           ○ A = X2
                           ○ Fa = Fx2
                     • End
               ▪ End
         ○ Else
               ▪ X1 = B - (B-A)/3
               ▪ Fx1 = f(X1)
               ▪ If Fb*Fx1 < 0 then
                     • A = X1
                     • Fa = Fx1
               ▪ Else
                     • X2 = A + (B-A)/3
                     • Fx2 = f(X2)
                     • If Fx1*Fx2 < 0 then
                           ○ A = X2
                           ○ Fa = Fx2
                           ○ B = X1
                           ○ Fb = Fx1
                     • Else
                           ○ B = X2
                           ○ Fb = Fx2
                     • End
               ▪ End
```

- **Until |A-B| < Toler**
- **Return root as (A+B)/2**

The Trisection Plus Algorithm

I have used the same approach in my previous efforts[4][5] to enhance the Bisection method, with the Trisection Plus algorithm. This variant of the Trisection algorithm carries out the same basic steps with the added step of performing an inverse linear interpolation within the new root-bracketing interval. This additional step enhances significantly the convergence to the root.

Let me present the pseudo-code for the Trisection Plus method:

Given f(x)=0, the root-bracketing interval [A,B], the tolerance Toler for the root of f(x), and the function tolerance value FxToler:

- **Calculate Fa = f(A) and Fb = f(B).**
- **Exit if Fa*Fb > 0**
- **Repeat**
  - **LastA = A**
  - **LastB = B**
  - **If |Fa| < |Fb)| then**
    - **X1 = A + (B - A) / 3**
    - **Fx1 = f(X1)**
    - **Comment-- case 1: [A,X1] has the root**
    - **If Fx1 * Fa < 0 then**
      - **X3 = Interpolate2(A, X1, Fa, Fx1)**
      - **Fx3 = f(X3)**
      - **If Fa * Fx3 < 0 then**
        - **B = X3**
        - **Fb = Fx3**
      - **Else**
        - **A = X3**
        - **Fa = Fx3**
        - **B = X1**
        - **Fb = Fx1**
      - **End**
    - **Else**
      - **X2 = A + 2 * (B - A) / 3**
      - **Fx2 = f(X2)**
      - **Comment-- case 2: [X1,X2] has root**
      - **If Fx1 * Fx2 < 0 then**
        - **X3 = Interpolate2(X1, X2, Fx1, Fx2)**
        - **Fx3 = f(X3)**
        - **If Fx1 * Fx3 < 0 then**
          - **A = X1**
          - **Fa = Fx1**
          - **B = X3**
          - **Fb = Fx3**
        - **Else**

```
                        ▪ A = X3
                        ▪ Fa = Fx3
                        ▪ B = X2
                        ▪ Fb = Fx2
                    ○ End
                • Else
                    ○ Comment := case 2: [X2,B] has root
                    ○ X3 = Interpolate2(X2, B, Fx2, Fb)
                    ○ Fx3 = f(X3)
                    ○ If Fx2 * Fx3 < 0 then
                        ▪ A = X2
                        ▪ Fa = Fx2
                        ▪ B = X3
                        ▪ Fb = Fx3
                    ○ Else
                        ▪ A = X3
                        ▪ Fa = Fx3
                    ○ End
                • End
            ▪ End
        ○ Else
            ▪ X1 = A + 2 * (B - A) / 3
            ▪ Fx1 = f(X1)
            ▪ Comment-- case 4: [X1,B] has the root
            ▪ If Fx1 * Fb < 0 then
                • X3 = Interpolate2(X1, B, Fx1, Fb)
                • Fx3 = f(X3)
                • If Fx1 * Fx3 < 0 then
                    ○ A = X1
                    ○ Fa = Fx1
                    ○ B = X3
                    ○ Fb = Fx3
                • Else
                    ○ A = X3
                    ○ Fa = Fx3
                • End
            ▪ Else
                • X2 = A + (B - A) / 3
                • Fx2 = f(X2)
                • Comment-- case 5: [X1,X2] has root
                • If Fx1 * Fx2 < 0 then
```

```
                    X3 = Interpolate2(X1, X2, Fx1, Fx2)
                    Fx3 = f(X3)
                    If Fx1 * Fx3 < 0 then
                        A = X1
                        Fa = Fx1
                        B = X3
                        Fb = Fx3
                    Else
                        A = X3
                        Fa = Fx3
                        B = X2
                        Fb = Fx2
                    End
                Else
                    Comment-- case 6: [A,X2] has root
                    X3 = Interpolate2(A, X2, Fa, Fx2)
                    Fx3 = f(X3)
                    If Fa * Fx3 < 0 then
                        B = X3
                        Fb = Fx3
                    Else
                        A = X3
                        Fa = X3
                        B = X2
                        Fb = Fx2
                    End
                End
            End
        End
        If A > B then
            Swap A, B
            Swap Fa, Fb
            Swap LastA, LastB
        End
        If LastA <> A And |A - LastA| < Toler then exit loop
        If LastB <> B And |B - LastB| < Toler then exit loop
    Until |A - B| < Toler Or |Fa| < FxToler Or |Fb| < FxToler
    If |Fa| < |Fb| Then
        Return A
    Else
        Return B
    End
```

Despite the length of the pseudo-code, it is not really complicated. When the code is executed in an implementation, only a fraction of the statements are executed in each iteration. It's just there are many alternate sets of statements (or execution flow paths, if you prefer) to execute. The various segments of the pseudo-code perform basically the same tasks on different combinations of X values. The function **Interpolate2** in the above pseudo-code performs an inverse linear interpolation to calculate the value of X for f(X)=0. Here is the simple pseudo-code for function **Interpolate2**:

```
Function Interpolate2(X1, X2, Fx1, Fx2)
  Return(X1 * (Fx2 - 0) - X2 * (Fx1 - 0)) / (Fx2 - Fx1)
End Function
```

The iterations in the main loop first test if f(A) is smaller than f(B) in magnitude. The code contains two sets symmetrical statements. In each set, the code determines which of the three sub-intervals contain the root. The algorithm then performs an inverse linear interpolation to calculate a refined guess for the root within the new (and smaller) root-bracketing interval. The last step is to further shrink the root-bracketing interval. The interpolation step significantly accelerates the convergence to the root.

The Interpolative Trisection Algorithms

The basic idea of the Interpolative Trisection algorithms is to:

- Divide the root-bracketing interval into three equal parts. One of these three sub-intervals contains the root.
- Perform inverse interpolation to calculate a value for X as the root of the function.
- Determine if the interpolated value of X falls inside the root-bracketing sub-interval. If this condition is true, the method further narrows the sub-interval using the interpolated value, and uses the shrunk sub-interval for the next iteration. If not, the new root-bracketing sub-interval becomes the root-bracketing interval for the next iteration.

In the case of the Cubic Interpolative Trisection, each iteration has four points to work with. The inverse cubic interpolation uses all four points. In the case of the Quadratic Interpolative Trisection, the algorithm has to choose three out of the four points to use in an inverse quadratic interpolation. The algorithm selects certain points that have smaller absolute function values.

The next two sections present the pseudo-ode for the two new algorithms.

The Quadratic Interpolative Trisection Algorithm

Here is the pseudo-code for the Quadratic Interpolative Trisection algorithm:

```
Given f(x)=0, the root-bracketing interval [A,B], the tolerance Toler for the
root of f(x), and the function tolerance value FxToler:
  • If A > B then Swap A, B
  • Calculate Fa = f(A) and Fb = f(B).
  • Exit if Fa*Fb > 0
  • Repeat
      ○ LastA = A
      ○ LastB = B
      • If |Fa| < |Fb| then
          ○ X1 = A + (B - A) / 3
          ○ Fx1 = f(X1)
          ○ Comment-- case 1: [A,X1] has the root
          ○ If Fx1 * Fa < 0 then
              ▪ X3 = QuadInterp(A, X1, B, Fa, Fx1, Fb)
              ▪ Fx3 = f(X3)
              ▪ If X3 > A And X3 < X1 then
                  • Comment-- X3 in [A, X1]
                  • If Fa * Fx3 > 0 then
                      ○ A = X3
                      ○ Fa = Fx3
                  • Else
                      ○ X1 = X3
                      ○ Fx1 = Fx3
                  • End
              ▪ End
              ▪ B = X1
              ▪ Fb = Fx1
          ○ Else
              ▪ X2 = B - (B - A) / 3
              ▪ Fx2 = f(X2)
              ▪ Comment-- case 2: [X1,X2] has root
              ▪ If Fx1 * Fx2 < 0 then
                  • Comment-- select A or B with the smaller
                    absolute function value
                  • If |Fa| < |Fb| then
                      ○ C = A
                      ○ Fc = Fa
                  • Else
                      ○ C = B
                      ○ Fc = Fb
                  • End
                  • X3 = QuadInterp(X1, X2, C, Fx1, Fx2, Fc)
                  • Fx3 = f(X3)
                  • If X3 > X1 And X3 < X2 then
                      ○ Comment-- X3 in [X1, X2]
                      ○ If Fx1 * Fx3 > 0 then
                          ▪ X1 = X3
```

```
                                    ▪ Fx1 = Fx3
                                o Else
                                    ▪ X2 = X3
                                    ▪ Fx2 = Fx3
                                o End
                        • End
                        • Comment-- update A and B
                        • A = X1
                        • Fa = Fx1
                        • B = X2
                        • Fb = Fx2
                    ▪ Else
                        • Comment-- case 2: [X2,B] has root
                        • If |Fa| < |Fx1| then
                            o C = A
                            o Fc = Fa
                        • Else
                            o C = X1
                            o Fc = Fx1
                        • End
                        • X3 = QuadInterp(C, X2, B, Fc, Fx2, Fb)
                        • Fx3 = f(X3)
                        • If X3 > X2 And X3 < B then
                            o Comment-- X3 in [X2, B]
                            o If Fx2 * Fx3 > 0 then
                                ▪ A = X3
                                ▪ Fa = Fx3
                            o Else
                                ▪ A = X2
                                ▪ Fa = Fx2
                            o End
                        • Else
                            o A = X2
                            o Fa = Fx2
                        • End
                    ▪ End
            o End
    • Else
            o X1 = B - (B - A) / 3
            o Fx1 = f(X1)
            o Comment-- case 4: [X1, B] has the root
            o If Fx1 * Fb < 0 then
                ▪ X3 = QuadInterp(A, X1, B, Fa, Fx1, Fb)
                ▪ Fx3 = f(X3)
                ▪ If X3 > X1 And X3 < B then
                        • Comment-- X3 in [X1, B]
                        • If Fb * Fx3 > 0 then
                            o B = X3
                            o Fb = Fx3
                            o A = X1
                            o Fa = Fx1
                        • Else
                            o A = X3
```

```
                    ○ Fa = Fx3
                • End
            ▪ Else
                • A = X1
                • Fa = Fx1
            ▪ End
    ○ Else
            ▪ X2 = A + (B - A) / 3
            ▪ Fx2 = f(X2)
            ▪ Comment-- case 5: [X1,X2] has root
            ▪ If Fx1 * Fx2 < 0 then
                • Comment-- select A or B with the smaller
                  absolute function value
                • If |Fa| < |Fb| then
                    ○ C = A
                    ○ Fc = Fa
                • Else
                    ○ C = B
                    ○ Fc = Fb
                • End
                • X3 = QuadInterp(X1, X2, C, Fx1, Fx2, Fc)
                • Fx3 = f(X3)
                • If X3 > X1 And X3 < X2 then
                    ○ Comment-- X3 in [X1, X2]
                    ○ If Fx1 * Fx3 > 0 then
                        ▪ X1 = X3
                        ▪ Fx1 = Fx3
                    ○ Else
                        ▪ X2 = X3
                        ▪ Fx2 = Fx3
                    ○ End
                • End
                • Comment-- update A and B
                • B = X1
                • Fb = Fx1
                • A = X2
                • Fa = Fx2
            ▪ Else
                • Comment-- case 6: [A, X2] has root
                • If |Fb| < |Fx1| then
                    ○ C = B
                    ○ Fc = Fb
                • Else
                    ○ C = X1
                    ○ Fc = Fx1
                • End
                • X3 = QuadInterp(C, X2, A, Fc, Fx2, Fa)
                • Fx3 = f(X3)
                • If X3 > A And X3 < X2 then
                    ○ Comment-- X3 in [A, X2]
                    ○ If Fx2 * Fx3 > 0 then
                        ▪ B = X3
```

```
                        ▪ Fb = Fx3
                    ○ Else
                        ▪ B = X2
                        ▪ Fb = Fx2
                    ○ End
                • Else
                    ○ B = X2
                    ○ Fb = Fx2
                • End
            ▪ End
        ○ End
    • End
    • If A <> LastA And |A – LastA| < Toler then Exit loop
    • If B <> LastB And |B – LastB| < Toler then Exit loop
    • Loop Until |A – B| < Toler Or |Fa| < FxToler Or |Fb| < FxToler
    • If |Fa| < |Fb| then
        ○ Return A
    • Else
        ○ Return B
    • End
```

The above pseudo-code mentions **QuadInterp** which represents a function that performs inverse quadratic interpolation. The VBA code shows you the implementation that uses Lagrangian interpolation.

The above pseudo-code appears long, but keep in mind that steps executed in each iteration are but a small fraction. In other words, the pseudo-code shows multiple alternate program execution flow paths. The code handles the cases for the left, middle, and right sub-intervals. The code also deal with symmetric braches of code that are executed depending on which interval endpoint has the smaller absolute function value.

The Cubic Interpolative Trisection Algorithm

Here is the pseudo-code for the Cubic Interpolative Trisection algorithm:

```
Given f(x)=0, the root-bracketing interval [A,B], the tolerance Toler for the
root of f(x), and the function tolerance value FxToler:
    • If A > B then Swap A, B
    • Calculate Fa = f(A) and Fb = f(B).
    • Exit if Fa*Fb > 0
    • Repeat
        ○ LastA = A
        ○ LastB = B
        ○ X1 = A + (B - A) / 3
        ○ Fx1 = f(X1)
        ○ X2 = B - (B - A) / 3
        ○ Fx2 = f(X2)
        ○ X3 = CubicInterp(A, X1, X2, B, Fa, Fx1, Fx2, B)
        ○ Fx3 = f(X3)
        ○ If Fa * Fx1 < 0 then
```

```
        ▪ Comment-- case 1: root in [A, X1]
        ▪ If X3 > A And X3 < X1 then
            • If Fa * Fx3 > 0 then
                ○ Comment-- root in [X3, X1]
                ○ A = X3
                ○ Fa = Fx3
                ○ B = X1
                ○ Fb = Fx1
            • Else
                ○ Comment-- root in [A, X3]
                ○ B = X3
                ○ Fb = Fx3
            • End
        ▪ Else
            • B = X1
            • Fb = Fx1
        ▪ End
    ○ Else If Fx1 * Fx2 < 0 then
        ▪ Comment-- case 2: root in [X1, X2]
        ▪ If X3 > X1 And X3 < X2 then
            • If Fx1 * Fx3 > 0 then
                ○ Comment-- root in [X3, X2]
                ○ A = X3
                ○ Fa = Fx3
                ○ B = X2
                ○ Fb = Fx2
            • Else
                ○ Comment-- root in [X1, X3]
                ○ A = X1
                ○ Fa = Fx1
                ○ B = X3
                ○ Fb = Fx3
            • End
        ▪ Else
            • A = X1
            • Fa = Fx1
            • B = X2
            • Fb = Fx2
        ▪ End
    ○ Else
        ▪ Comment-- root in [X2, B]
        ▪ If X3 > X2 And X3 < B then
            • If Fx2 * Fx3 > 0 then
                ○ Comment-- root in [X3, B]
                ○ A = X3
                ○ Fa = Fx3
            • Else
                ○ Comment-- root in [X2, X3]
                ○ A = X2
                ○ Fa = Fx2
                ○ B = X3
                ○ Fb = Fx3
            • End
        ▪ Else
```

```
                    • A = X2
                    • Fa = Fx2
                ▪ End
        o End
        o If A <> LastA And |A – LastA| < Toler then Exit loop
        o If B <> LastB And |B – LastB| < Toler then Exit loop
  • Loop Until |A - B) < Toler Or |Fa| < FxToler Or |Fb| < FxToler
  • If |Fa| < |Fb| then
        o Return A
  • Else
        o Return B
  • End
```

The above pseudo-code mentions **CubicdInterp** which represents a function that performs inverse cubic interpolation. The VBA code shows you the implementation that uses Lagrangian interpolation.

The pseudo-code for the Cubic Interpolative Trisection is shorter than its quadratic counterpart. This is true, because the cubic version of the Trisection method performs fewer choices since it uses all four points in the three sub-intervals. The algorithm determines which sub-interval brackets the root. It also determines if the interpolated value is inside that sub-interval. If so, the method further shrinks the root-bracketing sub-interval and uses it for the next iteration. If the interpolated value is outside the root-bracketing sub-interval, it simply uses that sub-interval as the root-bracketing interval in the next iteration.

Testing with Excel VBA Code

I tested the new algorithms using Excel taking advantage of the application's worksheet for easy input and the display of intermediate calculations. The following listing shows the Excel VBA code used for testing. It implements the Quadratic Interpolative Trisection, Cubic Interpolative Trisection, Trisection Plus, and Newton's methods:

```
Option Explicit

Function MyFx(ByVal sFx As String, ByVal X As Double) As Double

  sFx = UCase(sFx)
  sFx = Replace(sFx, "EXP(", "!!")
  sFx = Replace(sFx, "X", "(" & X & ")")
  sFx = Replace(sFx, "!!", "EXP(")
  MyFx = Evaluate(sFx)
End Function

Private Sub Swap(ByRef A As Double, ByRef B As Double)
  Dim Buff As Double
```

```
  Buff = A
  A = B
  B = Buff
End Sub

Function Interpolate2(ByVal X1 As Double, ByVal X2 As Double, _
                      ByVal Fx1 As Double, ByVal Fx2 As Double) As Double

  ' Interpolate2 = X1 * (Fx2 - 0) / (Fx2 - Fx1) + X2 * (Fx1 - 0) / (Fx1 -
Fx2)
  Interpolate2 = (X1 * (Fx2 - 0) - X2 * (Fx1 - 0)) / (Fx2 - Fx1)

End Function


Function QuadInterp(ByVal X1 As Double, ByVal X2 As Double, _
                    ByVal X3 As Double, ByVal Fx1 As Double, _
                    ByVal Fx2 As Double, ByVal Fx3 As Double) As Double
  Dim X(3) As Double, Fx(3) As Double, Sum As Double, Prod As Double
  Dim I As Integer, J As Integer, N As Integer
  X(1) = X1
  X(2) = X2
  X(3) = X3
  Fx(1) = Fx1
  Fx(2) = Fx2
  Fx(3) = Fx3

  Sum = 0
  N = 3
  For I = 1 To N
    Prod = X(I)
    For J = 1 To N
      If I <> J Then
        Prod = Prod * (0 - Fx(J)) / (Fx(I) - Fx(J))
      End If
    Next J
    Sum = Sum + Prod
  Next I
  QuadInterp = Sum
End Function

Function CubicInterp(ByVal X1 As Double, ByVal X2 As Double, _
                      ByVal X3 As Double, ByVal X4 As Double, _
                      ByVal Fx1 As Double, ByVal Fx2 As Double, _
                      ByVal Fx3 As Double, ByVal Fx4 As Double) As Double
  Dim X(4) As Double, Fx(4) As Double, Sum As Double, Prod As Double
  Dim I As Integer, J As Integer, N As Integer
  X(1) = X1
  X(2) = X2
  X(3) = X3
  X(4) = X4
  Fx(1) = Fx1
  Fx(2) = Fx2
  Fx(3) = Fx3
  Fx(4) = Fx4

  Sum = 0
```

```
  N = 4
  For I = 1 To N
    Prod = X(I)
    For J = 1 To N
      If I <> J Then
        Prod = Prod * (0 - Fx(J)) / (Fx(I) - Fx(J))
      End If
    Next J
    Sum = Sum + Prod
  Next I
  CubicInterp = Sum
End Function


Sub Go()
  Dim R As Integer, Col As Integer
  Dim A As Double, B As Double, Fa As Double, Fb As Double
  Dim C As Double, Fc As Double
  Dim X1 As Double, X2 As Double, Fx1 As Double, Fx2 As Double
  Dim X3 As Double, Fx3 As Double, Toler As Double, FxToler As Double
  Dim LastA As Double, LastB As Double, h As Double, Diff As Double
  Dim sFx As String, NumIters As Integer

  Range("B3:Z10000").Value = ""
  Toler = [A6].Value
  FxToler = [A8].Value
  sFx = [A10].Value

  ' Quadratic Trisection
  A = [A2].Value
  B = [A4].Value
  If A > B Then Swap A, B
  Fa = MyFx(sFx, A)
  Fb = MyFx(sFx, B)
  NumIters = 2
  R = 3
  Col = 2
  Do
    LastA = A
    LastB = B
    If Abs(Fa) < Abs(Fb) Then
      X1 = A + (B - A) / 3
      Fx1 = MyFx(sFx, X1)
      NumIters = NumIters + 1
      ' case 1: [A,X1] has the root
      If Fx1 * Fa < 0 Then
        X3 = QuadInterp(A, X1, B, Fa, Fx1, Fb)
        Fx3 = MyFx(sFx, X3)
        NumIters = NumIters + 1
        If X3 > A And X3 < X1 Then ' X3 in [A, X1]
          If Fa * Fx3 > 0 Then
            A = X3
            Fa = Fx3
          Else
            X1 = X3
            Fx1 = Fx3
          End If
```

```
    End If
    B = X1
    Fb = Fx1
  Else
    X2 = B - (B - A) / 3
    Fx2 = MyFx(sFx, X2)
    NumIters = NumIters + 1
    ' case 2: [X1,X2] has root
    If Fx1 * Fx2 < 0 Then
      ' select A or B with the smaller absolute funciton value
      If Abs(Fa) < Abs(Fb) Then
        C = A
        Fc = Fa
      Else
        C = B
        Fc = Fb
      End If
      X3 = QuadInterp(X1, X2, C, Fx1, Fx2, Fc)
      Fx3 = MyFx(sFx, X3)
      NumIters = NumIters + 1
      If X3 > X1 And X3 < X2 Then ' X3 in [X1, X2]
        If Fx1 * Fx3 > 0 Then
          X1 = X3
          Fx1 = Fx3
        Else
          X2 = X3
          Fx2 = Fx3
        End If
      End If
      ' update A and B
      A = X1
      Fa = Fx1
      B = X2
      Fb = Fx2
    Else
      ' case 2: [X2,B] has root
      If Abs(Fa) < Abs(Fx1) Then
        C = A
        Fc = Fa
      Else
        C = X1
        Fc = Fx1
      End If
      X3 = QuadInterp(C, X2, B, Fc, Fx2, Fb)
      Fx3 = MyFx(sFx, X3)
      NumIters = NumIters + 1
      If X3 > X2 And X3 < B Then ' X3 in [X2, B]
        If Fx2 * Fx3 > 0 Then
          A = X3
          Fa = Fx3
        Else
          A = X2
          Fa = Fx2
        End If
      Else
        A = X2
        Fa = Fx2
```

```
      End If
    End If
  End If
Else
  X1 = B - (B - A) / 3
  Fx1 = MyFx(sFx, X1)
  NumIters = NumIters + 1
  ' case 4: [X1, B] has the root
  If Fx1 * Fb < 0 Then
    X3 = QuadInterp(A, X1, B, Fa, Fx1, Fb)
    Fx3 = MyFx(sFx, X3)
    NumIters = NumIters + 1
    If X3 > X1 And X3 < B Then ' X3 in [X1, B]
      If Fb * Fx3 > 0 Then
        B = X3
        Fb = Fx3
        A = X1
        Fa = Fx1
      Else
        A = X3
        Fa = Fx3
      End If
    Else
      A = X1
      Fa = Fx1
    End If
  Else
    X2 = A + (B - A) / 3
    Fx2 = MyFx(sFx, X2)
    NumIters = NumIters + 1
    ' case 5: [X1,X2] has root
    If Fx1 * Fx2 < 0 Then
      ' select A or B with the smaller absolute funciton value
      If Abs(Fa) < Abs(Fb) Then
        C = A
        Fc = Fa
      Else
        C = B
        Fc = Fb
      End If
      X3 = QuadInterp(X1, X2, C, Fx1, Fx2, Fc)
      Fx3 = MyFx(sFx, X3)
      NumIters = NumIters + 1
      If X3 > X1 And X3 < X2 Then ' X3 in [X1, X2]
        If Fx1 * Fx3 > 0 Then
          X1 = X3
          Fx1 = Fx3
        Else
          X2 = X3
          Fx2 = Fx3
        End If
      End If
      ' update A and B
      B = X1
      Fb = Fx1
      A = X2
      Fa = Fx2
```

```
        Else
          ' case 6: [A, X2] has root
          If Abs(Fb) < Abs(Fx1) Then
            C = B
            Fc = Fb
          Else
            C = X1
            Fc = Fx1
          End If
          X3 = QuadInterp(C, X2, A, Fc, Fx2, Fa)
          Fx3 = MyFx(sFx, X3)
          NumIters = NumIters + 1
          If X3 > A And X3 < X2 Then ' X3 in [A, X2]
            If Fx2 * Fx3 > 0 Then
              B = X3
              Fb = Fx3
            Else
              B = X2
              Fb = Fx2
            End If
          Else
            B = X2
            Fb = Fx2
          End If
        End If
      End If
    End If
    Cells(R, Col) = A
    Cells(R, Col + 1) = B
    R = R + 1

    If A <> LastA And Abs(A - LastA) < Toler Then Exit Do
    If B <> LastB And Abs(B - LastB) < Toler Then Exit Do

  Loop Until Abs(A - B) < Toler Or Abs(Fa) < FxToler Or Abs(Fb) < FxToler

  If Abs(Fa) < Abs(Fb) Then
    Cells(R, Col) = A
  Else
    Cells(R, Col) = B
  End If
  Cells(R, Col + 1) = "Fx Calls=" & NumIters


' Cubic Trisection
  A = [A2].Value
  B = [A4].Value
  If A > B Then Swap A, B
  Fa = MyFx(sFx, A)
  Fb = MyFx(sFx, B)
  NumIters = 2
  R = 3
  Col = Col + 2
  Do
    LastA = A
    LastB = B
```

```
X1 = A + (B - A) / 3
Fx1 = MyFx(sFx, X1)
X2 = B - (B - A) / 3
Fx2 = MyFx(sFx, X2)
X3 = CubicInterp(A, X1, X2, B, Fa, Fx1, Fx2, B)
Fx3 = MyFx(sFx, X3)
NumIters = NumIters + 3

If Fa * Fx1 < 0 Then ' case 1: root in [A, X1]
  If X3 > A And X3 < X1 Then
    If Fa * Fx3 > 0 Then ' root in [X3, X1]
      A = X3
      Fa = Fx3
      B = X1
      Fb = Fx1
    Else ' root in [A, X3]
      B = X3
      Fb = Fx3
    End If
  Else
    B = X1
    Fb = Fx1
  End If
ElseIf Fx1 * Fx2 < 0 Then ' case 2: root in [X1, X2]
  If X3 > X1 And X3 < X2 Then
    If Fx1 * Fx3 > 0 Then ' root in [X3, X2]
      A = X3
      Fa = Fx3
      B = X2
      Fb = Fx2
    Else ' root in [X1, X3]
      A = X1
      Fa = Fx1
      B = X3
      Fb = Fx3
    End If
  Else
    A = X1
    Fa = Fx1
    B = X2
    Fb = Fx2
  End If
Else ' root in [X2, B]
  If X3 > X2 And X3 < B Then
    If Fx2 * Fx3 > 0 Then ' root in [X3, B]
      A = X3
      Fa = Fx3
    Else ' root in [X2, X3]
      A = X2
      Fa = Fx2
      B = X3
      Fb = Fx3
    End If
  Else
    A = X2
    Fa = Fx2
  End If
```

```
    End If

    Cells(R, Col) = A
    Cells(R, Col + 1) = B
    R = R + 1

    If A <> LastA And Abs(A - LastA) < Toler Then Exit Do
    If B <> LastB And Abs(B - LastB) < Toler Then Exit Do


  Loop Until Abs(A - B) < Toler Or Abs(Fa) < FxToler Or Abs(Fb) < FxToler

  If Abs(Fa) < Abs(Fb) Then
    Cells(R, Col) = A
  Else
    Cells(R, Col) = B
  End If
  Cells(R, Col + 1) = "Fx Calls=" & NumIters

  ' Trisection Plus
  A = [A2].Value
  B = [A4].Value
  If A > B Then Swap A, B
  Fa = MyFx(sFx, A)
  Fb = MyFx(sFx, B)
  NumIters = 2
  R = 3
  Col = Col + 2
  Do
    LastA = A
    LastB = B

    If Abs(Fa) < Abs(Fb) Then
      X1 = A + (B - A) / 3
      Fx1 = MyFx(sFx, X1)
      NumIters = NumIters + 1
      ' case 1: [A,X1] has the root
      If Fx1 * Fa < 0 Then
        X3 = Interpolate2(A, X1, Fa, Fx1)
        Fx3 = MyFx(sFx, X3)
        NumIters = NumIters + 1
        If Fa * Fx3 < 0 Then
          B = X3
          Fb = Fx3
        Else
          A = X3
          Fa = Fx3
          B = X1
          Fb = Fx1
        End If
      Else
        X2 = A + 2 * (B - A) / 3
        Fx2 = MyFx(sFx, X2)
        NumIters = NumIters + 1
        ' case 2: [X1,X2] has root
        If Fx1 * Fx2 < 0 Then
          X3 = Interpolate2(X1, X2, Fx1, Fx2)
```

```
        Fx3 = MyFx(sFx, X3)
        NumIters = NumIters + 1
        If Fx1 * Fx3 < 0 Then
          A = X1
          Fa = Fx1
          B = X3
          Fb = Fx3
        Else
          A = X3
          Fa = Fx3
          B = X2
          Fb = Fx2
        End If
      Else
        ' case 2: [X2,B] has root
        X3 = Interpolate2(X2, B, Fx2, Fb)
        Fx3 = MyFx(sFx, X3)
        NumIters = NumIters + 1
        If Fx2 * Fx3 < 0 Then
          A = X2
          Fa = Fx2
          B = X3
          Fb = Fx3
        Else
          A = X3
          Fa = Fx3
        End If
      End If
    End If
  Else
    X1 = A + 2 * (B - A) / 3
    Fx1 = MyFx(sFx, X1)
    NumIters = NumIters + 1
    ' case 4: [X1,B] has the root
    If Fx1 * Fb < 0 Then
      X3 = Interpolate2(X1, B, Fx1, Fb)
      Fx3 = MyFx(sFx, X3)
      NumIters = NumIters + 1
      If Fx1 * Fx3 < 0 Then
        A = X1
        Fa = Fx1
        B = X3
        Fb = Fx3
      Else
        A = X3
        Fa = Fx3
      End If
    Else
      X2 = A + (B - A) / 3
      Fx2 = MyFx(sFx, X2)
      NumIters = NumIters + 1
      ' case 5: [X1,X2] has root
      If Fx1 * Fx2 < 0 Then
        X3 = Interpolate2(X1, X2, Fx1, Fx2)
        Fx3 = MyFx(sFx, X3)
        NumIters = NumIters + 1
        If Fx1 * Fx3 < 0 Then
```

```
          A = X1
          Fa = Fx1
          B = X3
          Fb = Fx3
        Else
          A = X3
          Fa = Fx3
          B = X2
          Fb = Fx2
        End If
      Else
        ' case 6: [A,X2] has root
        X3 = Interpolate2(A, X2, Fa, Fx2)
        Fx3 = MyFx(sFx, X3)
        NumIters = NumIters + 1
        If Fa * Fx3 < 0 Then
          B = X3
          Fb = Fx3
        Else
          A = X3
          Fa = X3
          B = X2
          Fb = Fx2
        End If
      End If
    End If
  End If

  If A > B Then
    Swap A, B
    Swap Fa, Fb
    Swap LastA, LastB
  End If
  Cells(R, Col) = A
  Cells(R, Col + 1) = B
  R = R + 1
  If LastA <> A And Abs(A - LastA) < Toler Then Exit Do
  If LastB <> B And Abs(B - LastB) < Toler Then Exit Do
Loop Until Abs(A - B) < Toler Or Abs(Fa) < FxToler Or Abs(Fb) < FxToler

If Abs(Fa) < Abs(Fb) Then
  Cells(R, Col) = A
Else
  Cells(R, Col) = B
End If
Cells(R, Col + 1) = "Fx Calls=" & NumIters


' Newton's method
A = [A2].Value
B = [A4].Value
X1 = (A + B) / 2
NumIters = 0
R = 3
Col = Col + 2
Do
  h = 0.001 * (1 + Abs(X1))
```

```
    Fx1 = MyFx(sFx, X1)
    NumIters = NumIters + 2
    Diff = h * Fx1 / (MyFx(sFx, X1 + h) - Fx1)
    X1 = X1 - Diff
    Cells(R, Col) = X1
    Cells(R, Col + 1) = Fx1
    R = R + 1
  Loop Until Abs(Diff) < Toler
  Cells(R, Col) = X1
  Cells(R, Col + 1) = "Fx Calls=" & NumIters
End Sub
```

The VBA function **MyFX** calculates the function value based on a string that contains the function's expression. This expression must use **X** as the variable name. Note that the implementation of **MyFX** differs from previous ones (the Bisection Plus and Bisection++ methods) in that the name of the variable is **X** and not **$X**. Using function **MyFX** allows you to specify the function f(X)=0 in the spreadsheet and not hard code it in the VBA program. Granted that this approach trades speed of execution for flexibility. However, with most of today's PCs you will hardly notice the difference in execution times.

The subroutine **Go** performs the root-seeking calculations that compare the Quadratic Interpolative Trisection method, Cubic Interpolative Trisection method, Trisection Plus method, and Newton's method. Figure 1 shows a snapshot of the Excel spreadsheet used in the calculations for the methods mentioned above.

A	Quadratic Trisetion		Cubic Trisection		Trisection Plus		Newton	
1	A	B	A	B	A	B	X	
B	1.666666667	1.905177377	1.666666667	2	1.840376801	2	1.986679131	1.10668907
2	1.825673807	1.85743265	1.777777778	1.862098591	1.856820732	1.893584534	1.865982452	-0.54993176
Toler	1.846846369	1.857183922	1.856377262	1.862098591	1.857181284	1.869075332	1.857244749	-0.03486494
1.00E-10	1.853738071	1.85718386	1.857183843	1.858284372	1.857183854	1.861145967	1.857183967	-0.00024001
FxToler	1.85718386	Fx Calls=10	1.857183843	Fx Calls=14	1.857183854	Fx Calls=10	1.85718386	-4.2009E-07
1.00E-07							1.85718386	-7.1997E-10
Function							1.85718386	-1.2399E-12
EXP(X)-X^3							1.85718386	Fx Calls=14

Figure 1. The Excel spreadsheet used to compare the quadratic Trisection, Cubic Trisection, Trisection Plus, and Newton's methods.

The Input Cells

The VBA code relies on the following cells to obtain data:

- Cells A2 and A4 supply the values for the root-bracketing interval [A, B].
- Cell A6 contains the tolerance value.
- Cells A8 contains the function tolerance value.
- Cell A10 contains the expression for f(X)=0. Notice that the expression in cell A10 use **X** as the variable name. The expression is case insensitive.

Output

The spreadsheet displays output in the following four sets of columns:

- Columns B and C display the updated values for the root-bracketing interval [A, B] for the Quadratic Interpolative Trisection method. This interval shrinks with each iteration until the method zooms on the root. The bottom most value, in column B, is the best estimate for the root. To its right is the total number of function calls made during the iterations.
- Columns D, and E display the updated values for the root-bracketing interval [A, B] for the Cubic Interpolative Trisection method. The bottom most value, in column D, is the best estimate for the root. To its right is the total number of function calls made during the iterations.
- Columns F, and G display the updated values for the root-bracketing interval [A, B] for the Trisection Plus method. The bottom most value, in column F, is the best estimate for the root. To its right is the total number of function calls made during the iterations.
- Columns H and I display the refined guess for the root and the refinement value, respectively, using Newton's method. The bottom most value, in column H, is the best estimate for the root. To its right is the total number of function calls made during the iterations.

The Results

I really had no preconceived expectations as to how the Interpolative Trisection algorithms stack up against the Trisection Plus algorithm and Newton's method. The results surprised me. Table 1 shows a summary of the results. The metrics for comparing the algorithms include the number of iterations and, perhaps more importantly, the number of function calls. I consider the number of function calls as the underlying *cost of doing business*, so to speak. I have come across new root-seeking algorithms that require fewer iterations than popular algorithms like Newton's method and Halley's method. However, these new algorithms require more function calls to zoom in on the root in fewer iterations. The best results in Table 1 appear in red.

Function	[A, B]	Toler / FxToler	Root	Iterations	Num Fx Calls
Exp(X) – X^3	[1, 2]	1E–10 1E–7	1.857183	QuadTri= 4 CubTri= 4 Trisec+ = 4	QuadTri= 10 CubTri = 14 Trisec+= 10

Function	[A, B]	Toler / FxToler	Root	Iterations	Num Fx Calls
				Newton= 7	Newton= 14
Exp(X) – 3*X^2	[3, 4]	1E–10 1E–7	3.73307	QuadTri= 3 CubTri = 5 Trisec+ = 5 Newton= 7	QuadTri= 8 CubTri = 17 Trisec+ = 12 Newton= 14
Cos(X) – X	[0, 1]	1E–10 1E–7	0.73908	QuadTri= 3 CubTri = 4 Trisec+ = 4 Newton= 5	QuadTri= 8 CubTri = 14 Trisec+ = 10 Newton= 10
(X–1.234) * (X–5.678) * (X+12.345)	[5, 6]	1E–10 1E–7	5.678	QuadTri= 3 CubTri = 3 Trisec+ = 4 Newton=6	QuadTri= 8 CubTri = 11 Trisec+ = 10 Newton=12
(X–1.234) * (X–5.678) * (X+12.345)	[1, 2]	1E–10 1E–7	1.234	QuadTri= 3 CubTri = 7 Trisec+ = 4 Newton= 5	QuadTri= 8 CubTri = 23 Trisec+ = 10 Newton= 10
(X–1.234) * (X–5.678) * (X+12.345)	[5,11]	1E–10 1E–7	5.678	QuadTri= 5 CubTri = 6 Trisec+ = 6 Newton= 7	QuadTri= 12 CubTri = 20 Trisec+ = 14 Newton= 14
(X–1.234) * (X–5.678) * (X+12.345)	[–8, –15]	1E–10 1E–7	–12.345	QuadTri= 5 CubTri = 8 Trisec+ = 5 Newton= 5	QuadTri= 14 CubTri = 26 Trisec+ = 13 Newton= 10

Table 1. Summary of the results comparing the Quadratic Trisection, Cubic Trisection, Trisection Plus, and Newton's methods.

The above table shows that the Quadratic Interpolative Trisection method performs better in most test cases than the other algorithms. The Cubic Interpolative Trisection did fine in most cases, and faltered in a few. Of course there is a huge number of test cases that vary the tested function and root-bracketing range. Due to time limitation, I have chosen the above few test cases which succeeded in proving my goals.

Conclusion

The Quadratic Interpolative Trisection algorithm offers improvement over the Trisection Plus and Newton's method. The new algorithm has an efficiency that competes with Newton's method.

The Trisection Plus, Quadratic Interpolative Trisection, and Cubic Interpolative Trisection algorithms, use inverse linear interpolation, inverse quadratic interpolation, and inverse cubic interpolation, respectively. This small study shows that the inverse quadratic interpolation to be optimum. The inverse quadratic interpolation is best, followed by the inverse linear interpolation.

References

1. William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3rd edition, Cambridge University Press; 3rd edition, September 10, 2007.
2. Richard L. Burden, J. Douglas Faires, *Numerical Analysis*, Cengage Learning, 9th edition, August 9, 2010.
3. Namir Shammas, *Root-Bracketing Quartile Algorithm*, http://www.namirshammas.com/NEW/quartile.htm.
4. Namir Shammas, *The New Bisection Plus root-seeking algorithm*, http://www.namirshammas.com/NEW/BisPls.pdf
5. Namir Shammas, *The New Bisection++ root-seeking algorithm*, http://www.namirshammas.com/NEW/BisPls2.pdf
6. Namir Shammas, *The New Trisection and Trisection Plus root-seeking algorithm*, http://www.namirshammas.com/NEW/Tri1.pdf

Document Information

Version	*Date*	*Comments*
1.0.0	3/8/2014	Initial release.