The New Bisection++ Algorithm

by

Namir Shammas

Introduction

The Bisection method is well known to be the slowest root-seeking method for single-variable non-linear functions. Numerical analysis books often cover it for historical purposes. Recently I have developed two variants of the Bisection method, namely the Bisection Plus and the Bisection++ methods. I have discussed the Bisection Plus method is another article[4]. In this article I will present the Bisection++ which improves on both of the Bisection and Bisection Plus methods.

This article discusses the algorithms for the following methods:

- The Bisection method. This method starts with a root-bracketing interval and systematically reduces that interval by half in each iteration. The method stops when the interval's limits are very close to a root.
- Newton's method.
- The Bisection Plus method. This method repeatedly halves the root-bracketing interval and then performs a linear interpolation to get a better guess for the root.
- The Bisection++ method. This method repeatedly performs the following subtasks:
    - Halves the root-bracketing interval.
    - Performs a linear interpolation to get a better guess for the root.
    - Performs additional interpolation to further refine the guess for the root.

Do not worry about the details for the above methods, since I will discuss each one and present an accompanying pseudo-code. You can easily use this pseudo-code in implementing the algorithms in your favorite programming language.

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 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| < tolerance
  • 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. Unlike Newton's method, the Bisection method is not affected by small slopes near the root.

Newton's Method

I will also compare the new algorithm 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 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| < tolerance
```

- `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 Bisection and Bisection++ methods. Thus, the algorithm derives its single initial guess from the midpoint of that interval.

The Bisection Plus Algorithm

In my effort to enhance the Bisection method, I started with several approaches that takes the midpoint in the original Bisection method and enhances it. I initially worked with adding some random fluctuation to the midpoint value in [A, B]. This effort showed little promise.

I went back to the proverbial drawing board. The essential part of Bisection method is that it limits itself to comparing the signs of f(x) values. A few years ago, I developed the Quartile Method[3], which improves on the Bisection method by comparing the absolute values of f(A) and f(B) in order to get a better "midpoint" value. In designing the Bisection Plus algorithm, I decided to up the ante and work with the values of f(A), f(B), and f((A+B/2)). The new algorithm has the following steps:

1. Each iteration of the Bisection Plus method calculates the midpoint, call it X_1, and its associated function value $f(x_1)$.
2. The new algorithm then calculates the slope and intercept passing through X_1 and either A or B. The method selects the interval endpoint whose function value has the opposite sign of $f(X_1)$.
3. Using this new line, the algorithm calculates X2--a better estimate for the root. The choice of using the proper endpoint and X_1 ensures that X_2 lies in the interval [A, B]. The algorithm also calculates $f(X_2)$.
4. Each iteration ends up with the original interval-defining values A and B, and two new values, X_1 and X_2, within that interval. The process of shrinking the root-bracketing interval involves two tests:
    a. If the product of $f(X_1)$ and $f(X_2)$ is negative, then the interval $[X_1, X_2]$ replaces the interval [A, B] causing a quick reduction in the root-bracketing interval.
    b. If the product of $f(X_1)$ and $f(X_2)$ is positive, then the method uses the value of $f(A)\times f(X_2)$ to determine which of A or B is replaced by X_2.

An additional improvement to the algorithm, inspired by the False-Position method, compares the newly calculated X_2 value with the one from the previous iteration. If the two values are within the tolerance limit, the algorithm stops iterating.

It is worth pointing out that while both Newton's method and the Bisection Plus calculate slope values, these values are different in context. Newton's method calculate the tangent of f(X) at X. By contrast, the Bisection Plus method calculate the slope of the straight line between X_1 and one of the endpoints. Thus, the Bisection Plus method is not vulnerable to values of X where the derivative of f(X) is near zero, as is the case with Newton's method.

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

```
Given f(x)=0, the root-bracketing interval [A,B], and the
tolerance for the root of f(x):
  • Calculate Fa = f(A) and Fb=f(B).
  • Exit if Fa*Fb > 0
  • LastX = A
  • Repeat
      ○ X1=(A+B)/2
      ○ Fx1 = f(X1)
      ○ If Fx1*Fa > 0 then
          ▪ Slope = (Fb – Fx1)/(B – X1)
          ▪ Intercept = Fb – Slope * B
      ○ Else
          ▪ Slope = (Fa – Fx1)/(A – 1)
          ▪ Intercept = Fa – Slope * A
      ○ End
      ○ X2=-Intercept / Slope
      ○ Fx2 = f(X2)
      ○ If Fx1*Fx2 < 0 then
          ▪ A = X1
          ▪ Fa = Fx1
          ▪ B = X2
          ▪ Fb = Fx2
      ○ Else
          ▪ If Fx2*Fa > 0 then
              • A=X2
              • Fa=Fx2
          ▪ Else
              • B=X2
              • Fb=Fx2
```

```
            ▪ End
        ○ End
        ○ If |X2 – LastX| < tolerance then exit loop
        ○ LastX = X2
• Until |A-B| < tolerance
• Return X2
```

Figure 1 depicts an iteration of the Bisection Plus algorithm.

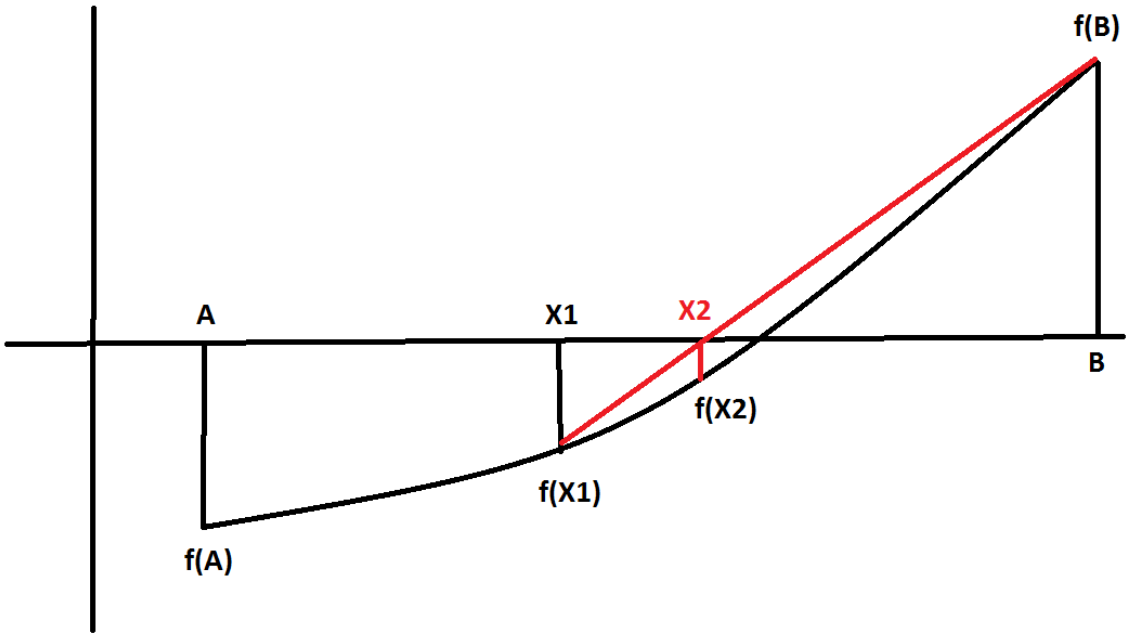

Figure 1. The steps within an iteration of the Bisection Plus Algorithm.

Figure 1 depicts an iteration that ends up replacing the initial root-bracketing interval [A, B] with [X_2, B]. If f(X_2) was positive, the iteration would have replaced the initial root-bracketing interval [A, B] with the narrower interval [X_1, X_2]. From the testing that I have done, the latter occurs frequently and helps to quickly narrow the root-bracketing interval around the targeted root value.

Figure 2 shows a case where some of the values of |f(X)| for X in [A, B] are greater than |f(A)| and |f(B)|. The figure illustrates why X_2 is calculated using X_1 and either endpoint whose function value has the opposite sign of f(X_1)—in the case of Figure 2, using X_1 and A. If X_2 is calculated using X_1 and B, then the value of X_2 lands outside the root-bracketing interval [A, B]. Using the scheme that I suggested to calculate X_2 simplify matters, because the algorithm needs not check if any X in

[A, B] has function values that exceed |f(A)| and/or |f(B)|.

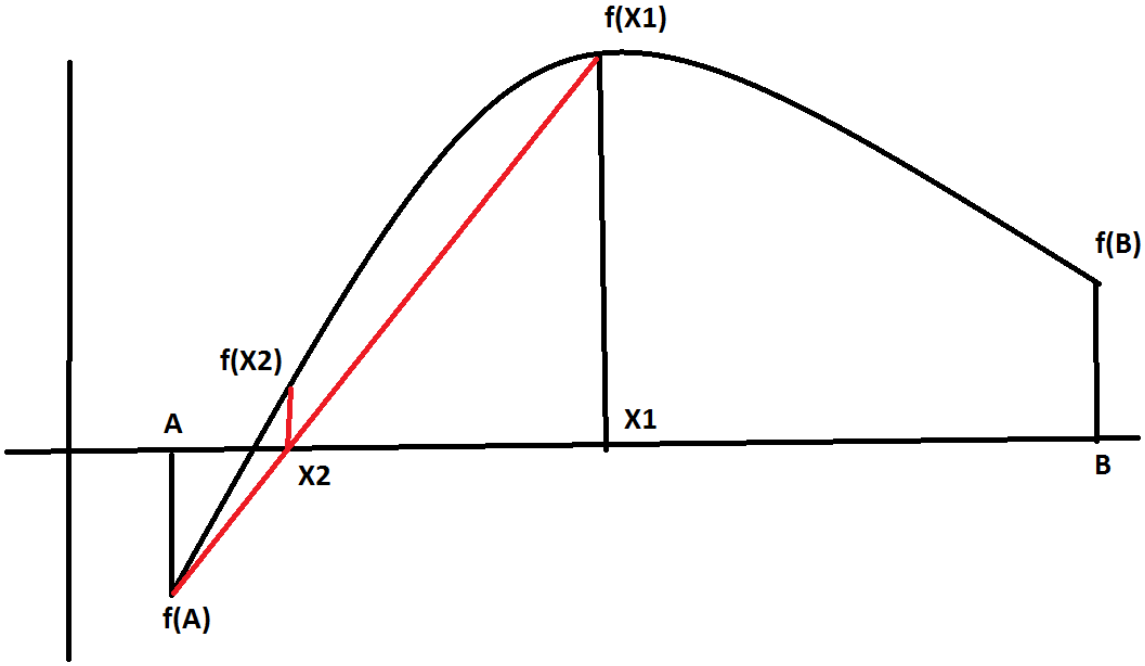

Figure 2. Special cases that dictates calculate X2 using the suggested scheme.

The Bisection++ Method

The Bisection++ method picks up where the Bisection Plus leaves off. In the last section I discussed how each iteration in the Bisection Plus method calculates two refined guesses, X_1 and X_2. The value of X1 is simply the midpoint of the root-bracketing interval [A, B]. The value of X_2 is calculated as the intersection of a straight with the X-axis, drawn between either (A, f(A)) and (X_1, f(X_1)) or between (B, f(B)) and (X_1, f(X_1)). Each iteration of the Bisection Plus ends up with four points at A, B, X_1, and X_2. The Bisection++ method uses these four points to perform an inverse quadratic Lagrangian interpolation to refine X_2. Since a quadratic interpolation requires three points and we have four, we need to choose three points and discard the fourth one. This choice creates two flavors of the Bisection++ method, which I will call version 1 and version 2:

- Version 1 uses the points at X_1 and X_2, and either at A or B, depending on which of these two points has a smaller absolute function value. Thus, version 1 is simple to code.
- Version 2 maps the four points to an array of X and an array of Y=f(X), sorts these arrays in ascending order using the absolute values of Y. The

method then uses the first three points in the sorted arrays to perform the inverse quadratic interpolation.

Either version calculates an improved value of X_2, call it X_3. The method ensures that the value of X_3 lies within the interval [A, B] before replacing X_2 with X_3. If not, the algorithm simply reuses the value of X_2. The algorithm also calculates the new value of $f(X_2)$ and tests if its absolute value is less than a function tolerance value. If it is smaller, the iteration stops.

Let me present the pseudo-code for the Bisection++ version 1 method. I will highlight in red the pseudo-code fragment that is specific to version 1:

```
Given f(x)=0, the root-bracketing interval [A,B], the root
tolerance Toler, and function tolerance FxToler:
  • Calculate Fa = f(A) and Fb=f(B).
  • Exit if Fa*Fb > 0
  • LastX = A
  • Repeat
      ○ X1=(A+B)/2
      ○ Fx1 = f(X1)
      ○ If Fx1*Fa > 0 then
          ▪ Slope = (Fb - Fx1)/(B - X1)
          ▪ Intercept = Fb - Slope * B
      ○ Else
          ▪ Slope = (Fa - Fx1)/(A - 1)
          ▪ Intercept = Fa - Slope * A
      ○ End
      ○ X2=-Intercept / Slope
      ○ Fx2 = f(X2)
      ○ If |Fa| < |Fb|then
          ▪ Calculate X3 using an inverse quadratic
            Lagrangian interpolation involving (A, Fa), (X1,
            Fx1), and (X2, FX2)
      ○ Else
          ▪ Calculate X3 using an inverse quadratic
            Lagrangian interpolation involving (B, Fb), (X1,
            Fx1), and (X2, FX2)
      ○ End
      ○ If X3>=A and X3<=B Then
          ▪ X2=X3
          ▪ Fx2=f(X2)
      ○ End
      ○ If |Fx2| < FxToler Then Exit Repeat loop
      ○ If Fx1*Fx2 < 0 then
          ▪ A = X1
```

```
            ▪ Fa = Fx1
            ▪ B = X2
            ▪ Fb = Fx2
        ○ Else
            ▪ If Fx2*Fa > 0 then
                • A=X2
                • Fa=Fx2
            ▪ Else
                • B=X2
                • Fb=Fx2
            ▪ End
        ○ End
        ○ If |X2 – LastX| < tolerance then exit loop
        ○ LastX = X2
  • Until |A-B| < tolerance
  • Return X2
```

Here is the pseudo-code for the Bisection++ version 2 method. I will highlight in red the pseudo-code fragment that is specific to version 2:

```
Given f(x)=0, the root-bracketing interval [A,B], the root
tolerance Toler, and function tolerance FxToler:
  • Calculate Fa = f(A) and Fb=f(B).
  • Exit if Fa*Fb > 0
  • LastX = A
  • Repeat
      ○ X1=(A+B)/2
      ○ Fx1 = f(X1)
      ○ If Fx1*Fa > 0 then
          ▪ Slope = (Fb – Fx1)/(B – X1)
          ▪ Intercept = Fb – Slope * B
      ○ Else
          ▪ Slope = (Fa – Fx1)/(A – 1)
          ▪ Intercept = Fa – Slope * A
      ○ End
      ○ X2=-Intercept / Slope
      ○ Fx2 = f(X2)
      ○ Map points (A,Fa),(B,Fb),(X1,Fx1), and (X2,Fx2) into
        an array of X and Y.
      ○ Sort the arrays of X and Y values in ascending order
        using the absolute values of Y.
      ○ Using the first three elements in the arrays X and Y,
        calculate X3 with an inverse quadratic Lagrangian
        interpolation.
      ○ If X3>=A and X3<=B Then
```

```
            ▪ X2=X3
            ▪ Fx2=f(X2)
        ○ End
        ○ If |Fx2| < FxToler Then Exit Repeat loop
        ○ If Fx1*Fx2 < 0 then
            ▪ A = X1
            ▪ Fa = Fx1
            ▪ B = X2
            ▪ Fb = Fx2
        ○ Else
            ▪ If Fx2*Fa > 0 then
                • A=X2
                • Fa=Fx2
            ▪ Else
                • B=X2
                • Fb=Fx2
            ▪ End
        ○ End
        ○ If |X2 – LastX| < tolerance then exit loop
        ○ LastX = X2
  • Until |A-B| < tolerance
  • Return X2
```

Testing with Excel VBA Code

I tested the new algorithm 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:

```
Option Explicit

Function MyFx(ByVal sFx As String, ByVal X As Double) As Double
  sFx = UCase(Trim(sFx))
  sFx = Replace(sFx, "$X", "(" & X & ")")
  MyFx = Evaluate(sFx)
End Function

Function QuadInterpolate(ByVal X1 As Double, ByVal X2 As Double,
ByVal X3 As Double, ByVal Y1 As Double, ByVal Y2 As Double,
ByVal Y3 As Double) As Double
  Dim Sum As Double

  Sum = X1 * (0 - Y2) * (0 - Y3) / (Y1 - Y2) / (Y1 - Y3)
  Sum = Sum + X2 * (0 - Y1) * (0 - Y3) / (Y2 - Y1) / (Y2 - Y3)
  Sum = Sum + X3 * (0 - Y1) * (0 - Y2) / (Y3 - Y1) / (Y3 - Y2)
  QuadInterpolate = Sum
```

```
End Function


Sub BisectionPlusPlusVer1()
  ' Bisection Plus Plus algorithm
  ' Version 2.00A  1/12/2014
  ' Copyright (c) 2014 Namir Clement Shammas
  '
  ' Perform:
  ' 1) Mid interval selection yo calculate X1
  ' 2) Linear interpolation between (X1,f(X1) and
  '    either end point to calculate X2
  ' 3) Quadratic interpolation involving points at
  '    X1, X2, and either A, or B to calculate
  '    a new X2.
  '
  Dim R As Integer, Count As Long, NumFxCalls As Integer
  Dim A As Double, B As Double, X As Double, LastX As Double, X3
As Double
  Dim X1 As Double, X2 As Double, FX1 As Double, FX2 As Double
  Dim FA As Double, FB As Double, FX As Double, Toler As Double
  Dim Slope As Double, Intercept As Double, FxToler As Double
  Dim h As Double, Diff As Double
  Dim sFx As String

  Range("B3:Z1000").Value = ""
  A = [A2].Value
  B = [A4].Value
  Toler = [A6].Value
  FxToler = [A8].Value
  sFx = [A10].Value

  FA = MyFx(sFx, A)
  FB = MyFx(sFx, B)
  If FA * FB > 0 Then
    MsgBox "F(A) & F(B) have the same signs"
    Exit Sub
  End If

  ' Bisection
  R = 3
  NumFxCalls = 2
  Do While Abs(A - B) > Toler
    NumFxCalls = NumFxCalls + 1
    X = (A + B) / 2
    FX = MyFx(sFx, X)
```

```
    If FX * FA > 0 Then
      A = X
      FA = FX
    Else
      B = X
      FB = FX
    End If
    Cells(R, 2) = A
    Cells(R, 3) = B
    R = R + 1
  Loop
  Cells(R, 2) = (A + B) / 2
  Cells(R, 3) = "FX Calls=" & NumFxCalls

  ' Bisection Plus Plus
  A = [A2].Value
  B = [A4].Value
  FA = MyFx(sFx, A)
  FB = MyFx(sFx, B)
  R = 3
  LastX = A
  NumFxCalls = 2
  Do
    X1 = (A + B) / 2
    FX1 = MyFx(sFx, X1)
    NumFxCalls = NumFxCalls + 1
    If FA * FX1 > 0 Then
      Slope = (FB - FX1) / (B - X1)
      Intercept = FB - Slope * B
    Else
      Slope = (FA - FX1) / (A - X1)
      Intercept = FA - Slope * A
    End If
    X2 = -Intercept / Slope
    FX2 = MyFx(sFx, X2)
    NumFxCalls = NumFxCalls + 1

    ' perform quadratic interpolation
    If Abs(FA) < Abs(FB) Then
      X3 = QuadInterpolate(A, X1, X2, FA, FX1, FX2)
    Else
      X3 = QuadInterpolate(B, X1, X2, FB, FX1, FX2)
    End If
    ' display intermediate values10
    Cells(R, 6) = X1
    Cells(R, 7) = X2
    Cells(R, 8) = X3
```

```
    ' make sure X3 is in the interval [A, B]
    If X3 >= A And X3 <= B Then
      X2 = X3
      FX2 = MyFx(sFx, X2)
    NumFxCalls = NumFxCalls + 1
    End If
    If Abs(FX2) < FxToler Then
      Cells(R, 4) = A
      Cells(R, 5) = B
      Cells(R, 9) = "FxToler Exit"
      R = R + 1
      Exit Do
    End If

    ' does [X1,X2] define a new root-bracketing interval?
    If FX1 * FX2 < 0 Then
      A = X1
      FA = FX1
      B = X2
      FB = FX2
      Cells(R, 9) = "New interval"
    Else
      If FA * FX2 > 0 Then
        A = X2
        FA = FX2
      Else
        B = X2
        FB = FX2
      End If
    End If
    Cells(R, 4) = A
    Cells(R, 5) = B
    If Abs(LastX - X2) < Toler Then Exit Do
    LastX = X2
    R = R + 1
  Loop Until Abs(A - B) < Toler
  Cells(R, 4) = X2
  Cells(R, 5) = "FX Calls=" & NumFxCalls

  ' Newton's method
  A = [A2].Value
  B = [A4].Value
  X = (A + B) / 2
  R = 3
  NumFxCalls = 2
  Diff = 10 * Toler
  Do
```

```
    h = 0.001 * (Abs(X) + 1)
    FX = MyFx(sFx, X)
    Diff = h * FX / (MyFx(sFx, X + h) - FX)
    X = X - Diff
    NumFxCalls = NumFxCalls + 2
    Cells(R, 10) = X
    Cells(R, 11) = Diff
    R = R + 1
  Loop Until Abs(Diff) <= Toler
  Cells(R, 10) = X
  Cells(R, 11) = "FX Calls=" & NumFxCalls
End Sub

Function QuadInterp(ByVal X1 As Double, ByVal X2 As Double,
ByVal X3 As Double, ByVal X4 As Double, ByVal Y1 As Double,
ByVal Y2 As Double, ByVal Y3 As Double, ByVal Y4 As Double) As
Double
  Const MAX = 4
  Dim X(MAX) As Double, Y(MAX) As Double, Xint As Double, Yint
As Double
  Dim I As Integer, J As Integer, Prod As Double, Buffer As
Double
  Dim bInOrder As Boolean

  ' map parameters to local arrays X() and Y()
  X(1) = X1
  X(2) = X2
  X(3) = X3
  X(4) = X4
  Y(1) = Y1
  Y(2) = Y2
  Y(3) = Y3
  Y(4) = Y4

  ' perform a simple Bubble sort
  For I = 1 To MAX - 1
    bInOrder = True
    For J = I + 1 To MAX
      If Abs(Y(I)) > Abs(Y(J)) Then
        Buffer = Y(I)
        Y(I) = Y(J)
        Y(J) = Buffer
        Buffer = X(I)
        X(I) = X(J)
        X(J) = Buffer
        bInOrder = False
      End If
```

```
    Next J
    ' exit outer For loop if all elements were in order
    If bInOrder Then Exit For
  Next I

  ' Perform (inverse) Lagrangian interpolation using arrays X()
and Y()
  Yint = 0 ' target value
  Xint = 0
  For I = 1 To MAX - 1
    Prod = X(I)
    For J = 1 To MAX - 1
      If I <> J Then
        Prod = Prod * (Yint - Y(J)) / (Y(I) - Y(J))
      End If
    Next J
    Xint = Xint + Prod
  Next I
  QuadInterp = Xint

End Function

Sub BisectionPlusPlusVer2()
  ' Bisection Plus algorithm
  ' Version 2.00B  1/12/2014
  ' Copyright (c) 2014 Namir Clement Shammas
  '
  ' Perform:
  ' 1) Mid interval selection yo calculate X1
  ' 2) Linear interpolation between (X1,f(X1) and
  '    either end point to calculate X2
  ' 3) Sort the four points at A, B, X1, and X2 and
  '    select the best three to use in quadratic
  '    interpolation to calculkate new X2.
  '
  Dim R As Integer, Count As Long, NumFxCalls As Integer
  Dim A As Double, B As Double, X As Double, LastX As Double, X3
As Double
  Dim X1 As Double, X2 As Double, FX1 As Double, FX2 As Double
  Dim FA As Double, FB As Double, FX As Double, Toler As Double
  Dim Slope As Double, Intercept As Double, FxToler As Double
  Dim h As Double, Diff As Double
  Dim sFx As String

  Range("B3:Z1000").Value = ""
  A = [A2].Value
  B = [A4].Value
```

```
Toler = [A6].Value
FxToler = [A8].Value
sFx = [A10].Value

FA = MyFx(sFx, A)
FB = MyFx(sFx, B)
If FA * FB > 0 Then
  MsgBox "F(A) & F(B) have the same signs"
  Exit Sub
End If

' Bisection
R = 3
NumFxCalls = 2
Do While Abs(A - B) > Toler
  NumFxCalls = NumFxCalls + 1
  X = (A + B) / 2
  FX = MyFx(sFx, X)
  If FX * FA > 0 Then
    A = X
    FA = FX
  Else
    B = X
    FB = FX
  End If
  Cells(R, 2) = A
  Cells(R, 3) = B
  R = R + 1
Loop
Cells(R, 2) = (A + B) / 2
Cells(R, 3) = "FX Calls=" & NumFxCalls

' Bisection Plus Plus
A = [A2].Value
B = [A4].Value
FA = MyFx(sFx, A)
FB = MyFx(sFx, B)
R = 3
LastX = A
NumFxCalls = 2
Do
  X1 = (A + B) / 2
  FX1 = MyFx(sFx, X1)
  NumFxCalls = NumFxCalls + 1
  If FA * FX1 > 0 Then
    Slope = (FB - FX1) / (B - X1)
    Intercept = FB - Slope * B
```

```
Else
  Slope = (FA - FX1) / (A - X1)
  Intercept = FA - Slope * A
End If
X2 = -Intercept / Slope
FX2 = MyFx(sFx, X2)
NumFxCalls = NumFxCalls + 1

' perform quadratic interpolation
X3 = QuadInterp(A, B, X1, X2, FA, FB, FX1, FX2)
' display intermediate values10
Cells(R, 6) = X1
Cells(R, 7) = X2
Cells(R, 8) = X3
' make sure X3 is in the interval [A, B]
If X3 >= A And X3 <= B Then
  X2 = X3
   FX2 = MyFx(sFx, X2)
    NumFxCalls = NumFxCalls + 1
End If
If Abs(FX2) < FxToler Then
  Cells(R, 4) = A
  Cells(R, 5) = B
  Cells(R, 9) = "FxToler Exit"
  R = R + 1
  Exit Do
End If


' does [X1,X2] define a new root-bracketing interval?
If FX1 * FX2 < 0 Then
  A = X1
  FA = FX1
  B = X2
  FB = FX2
  Cells(R, 9) = "New interval"
Else
  If FA * FX2 > 0 Then
    A = X2
    FA = FX2
  Else
    B = X2
    FB = FX2
  End If
End If
Cells(R, 4) = A
Cells(R, 5) = B
```

```
    If Abs(LastX - X2) < Toler Then Exit Do
    LastX = X2
    R = R + 1
  Loop Until Abs(A - B) < Toler
  Cells(R, 4) = X2
  Cells(R, 5) = "FX Calls=" & NumFxCalls

  ' Newton's method
  A = [A2].Value
  B = [A4].Value
  X = (A + B) / 2
  R = 3
  NumFxCalls = 2
  Diff = 10 * Toler
  Do
    h = 0.001 * (Abs(X) + 1)
    FX = MyFx(sFx, X)
    Diff = h * FX / (MyFx(sFx, X + h) - FX)
    X = X - Diff
    NumFxCalls = NumFxCalls + 2
    Cells(R, 10) = X
    Cells(R, 11) = Diff
    R = R + 1
  Loop Until Abs(Diff) <= Toler
  Cells(R, 10) = X
  Cells(R, 11) = "FX Calls=" & NumFxCalls
End Sub

Function QuadInterp2(ByVal X1 As Double, ByVal X2 As Double,
ByVal X3 As Double, ByVal X4 As Double, ByVal Y1 As Double,
ByVal Y2 As Double, ByVal Y3 As Double, ByVal Y4 As Double) As
Double
  Const MAX = 4
  Dim X(MAX) As Double, Y(MAX) As Double, Xint As Double, Yint
As Double
  Dim I As Integer, J As Integer, Prod As Double, Buffer As
Double
  Dim bInOrder As Boolean

  ' map parameters to local arrays X() and Y()
  X(1) = X1
  X(2) = X2
  X(3) = X3
  X(4) = X4
  Y(1) = Y1
  Y(2) = Y2
  Y(3) = Y3
```

```
  Y(4) = Y4


  ' Perform (inverse) Lagrangian interpolation using arrays X()
and Y()
  Yint = 0 ' target value
  Xint = 0
  For I = 1 To MAX
    Prod = X(I)
    For J = 1 To MAX
      If I <> J Then
        Prod = Prod * (Yint - Y(J)) / (Y(I) - Y(J))
      End If
    Next J
    Xint = Xint + Prod
  Next I
  QuadInterp2 = Xint

End Function
```

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. 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 with flexibility. However, with most of today's PCs you will hardly notice the difference in execution speeds.

The subroutine **BisectionPlusPlusVer1** performs the root-seeking calculations that compare the Bisection method, Bisection++ (version 1) method, and Newton's method. Figure 2 shows a snapshot of the Excel spreadsheet used in the calculations for the Bisection, Bisection++, and Newton's method.

The function **QuadInterpolate** performs the inverse quadratic Lagrangian interpolation using thee points.

The subroutine **BisectionPlusPlusVer2** performs the root-seeking calculations that compare the Bisection method, Bisection++ (version 2) method, and Newton's method.

The function **QuadInterp** performs the following tasks:

- Maps the parameters onto arrays of four X and four Y values.
- Sort the arrays X and Y in ascending order using the absolute values of Y.

- Using the first three elements of arrays X and Y, calculate the interpolated X (for a Y=0) using inverse quadratic Lagrangian interpolation.

	A	B	C	D	E	F	G	H	I	J	K
1	A	Bisection		BisectionPlusPlus						Newton's Method	
2	11	A	B	A	B	X1	X2	X3	Status	X	Diff
3	B	11	16.5	16.5	12.33967815	16.5	12.43924387	12.33967815	New interval	10.51459583	5.98540417
4	22	11	13.75	12.345097	12.33967815	14.41983907	12.345097	12.34499999		12.50827242	-1.9936766
5	Toler	11	12.375	12.345097	12.33967815	12.34238757	12.345	12.345	FxToler Exit	12.34519975	0.16307267
6	1.00E-08	11.6875	12.375	12.345	FX Calls=9					12.34500003	0.00019972
7	Fx Toler	12.03125	12.375							12.345	2.6711E-08
8	1.00E-04	12.203125	12.375							12.345	3.499E-12
9	F(X)	12.2890625	12.375							12.345	FX Calls=14
10	($X-2.345)*($X-12.345)*($X-23.456)	12.33203125	12.375								
11		12.33203125	12.35351563								
12		12.34277344	12.35351563								
13		12.34277344	12.34814453								
14		12.34277344	12.34545898								
15		12.34411621	12.34545898								
16		12.3447876	12.34545898								
17		12.3447876	12.34512329								
18		12.34495544	12.34512329								
19		12.34495544	12.34503937								
20		12.34499741	12.34503937								
21		12.34499741	12.34501839								
22		12.34499741	12.3450079								
23		12.34499741	12.34500265								
24		12.34499741	12.34500003								
25		12.34499872	12.34500003								
26		12.34499937	12.34500003								
27		12.3449997	12.34500003								
28		12.34499986	12.34500003								
29		12.34499995	12.34500003								
30		12.34499999	12.34500003								
31		12.34499999	12.34500001								
32		12.345	12.34500001								
33		12.345	12.345								
34		12.345	FX Calls=33								

Figure 2. The Excel spreadsheet used to comparing the Bisection, Bisection++, and Newton's method.

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 root tolerance value.
- Cell A8 contains the function tolerance value.
- Cell A10 contains the expression for f(X)=0. Notice that the contents of cell A10 use $X as the variable name. The expression is case insensitive.

Output

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

- Columns B and C display the updated values for the root-bracketing interval [A, B] for the Bisection method. This interval shrinks with each iteration until the Bisection 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 Bisection++ method. Columns F, G, and H display the values of X_1, X_2, and X_3, respectively. Column I displays comments made by the

code that point out when X_1 and X_2 form a new root-bracketing interval. The column also displays a message when the iteration stops because $|f(X_2)|$ is less than the function tolerance value. 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 J and K display the refined guess for the root and the refinement value, respectively, using Newton's method. The bottom most value, in column J, is the best estimate for the root. To its right is the total number of function calls made during the iterations.

The Results

My aim is to significantly accelerate the Bisection++ method compared to the Bisection method. Table 1 shows a summary of the results for version 1 of the Bisection++. 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 that 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.

Function	[A, B]	Toler/FxToler	Root	Iterations	Num Fx Calls
Exp(x)-4*x^2	[3, 5]	1E-8 1E-8	4.30658	Bisec= 28 Bisec++ =4 Newton=6	Bisec= 30 Bisec++ = 14 Newton=14
Exp(x)-3*x^2	[1, 4]	1E-8 1E-8	3.73307	Bisec= 29 Bisec++ = 4 Newton=13	Bisec= 31 Bisec++ = 18 Newton=22
Exp(x)-3*x^2	[3, 4]	1E-8 1E-8	3.73307	Bisec= 27 Bisec++ = 3 Newton=6	Bisec= 29 Bisec++ = 11 Newton=14
(X-2.345) * (X-12.345) * (X-23.456)	[1, 11]	1E-8 1E-8	2.345	Bisec= 30 Bisec++ = 4 Newton=9	Bisec= 32 Bisec++ = 12 Newton=20
(X-2.345) * (X-12.345) * (X-23.456)	[11, 22]	1E-8 1E-8	12.345	Bisec= 29 Bisec++ = 4 Newton=6	Bisec= 31 Bisec++ = 11 Newton=14

Function	[A, B]	Toler/FxToler	Root	Iterations	Num Fx Calls
LN(X^4)-X	[8,9]	1E-8 1E-8	8.613169	Bisec= 27 Bisec++ = 2 Newton=4	Bisec= 29 Bisec++ = 8 Newton=10
Cos(X)-X	[0.5,1]	1E-8 1E-8	0.73908	Bisec= 26 Bisec++ = 2 Newton=4	Bisec= 28 Bisec++ = 8 Newton=10

Table 1. Summary of the results comparing the Bisection, Bisection++ version 1, and Newton's method, with function tolerance value of 1E-8.

The above table shows that the Bisection++ method consistently outperforms the Bisection and Newton's methods. The table highlights in red the cases where the Bisection++ method performs better than Newton's method. 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.

Table 2 shows the same tests but with a function tolerance value of 1E-4.

Function	[A, B]	Toler/FxToler	Root	Iterations	Num Fx Calls
Exp(x)-4*x^2	[3, 5]	1E-8 1E-8	4.30658	= 28 Bisec++ =3 Newton=6	Bisec= 30 Bisec++ = 11 Newton=14
Exp(x)-3*x^2	[1, 4]	1E-8 1E-4	3.73307	Bisec= 29 Bisec++ = 3 Newton=13	Bisec= 31 Bisec++ = 10 Newton=22
Exp(x)-3*x^2	[3, 4]	1E-8 1E-4	3.73307	Bisec= 27 Bisec++ = 2 Newton=6	Bisec= 29 Bisec++ = 8 Newton=14
(X-2.345) * (X-12.345) * (X-23.456)	[1, 11]	1E-8 1E-4	2.345	Bisec= 30 Bisec++ = 3 Newton=9	Bisec= 32 Bisec++ = 10 Newton=20
(X-2.345) * (X-12.345) * (X-23.456)	[11, 22]	1E-8 1E-4	12.345	Bisec= 29 Bisec++ = 3 Newton=6	Bisec= 31 Bisec++ = 9 Newton=14
LN(X^4)-X	[8,9]	1E-8 1E-4	8.613169	Bisec= 27 Bisec++ = 1 Newton=4	Bisec= 29 Bisec++ = 5 Newton=10

Function	[A, B]	Toler/FxToler	Root	Iterations	Num Fx Calls
Cos(X)-X	[0.5,1]	1E-8 1E-4	0.73908	Bisec= 26 Bisec++ = 1 Newton=4	Bisec= 28 Bisec++ = 5 Newton=10

Table 2. Summary of the results comparing the Bisection, Bisection++ version 1, and Newton's method, with function tolerance value of 1E-4.

Table 2 shows a typical reduction of one iteration for the Bisection++ when the function tolerance value goes from 1E-8 to its square root value of 1E-4.

Table 3 shows the tests for version 2 of Bisection++ with a function tolerance value of 1E-8.

Function	[A, B]	Toler/FxToler	Root	Iterations	Num Fx Calls
Exp(x)-4*x^2	[3, 5]	1E-8 1E-8	4.30658	Bisec= 28 Bisec++ =4 Newton=6	Bisec= 30 Bisec++ = 14 Newton=14
Exp(x)-3*x^2	[1, 4]	1E-8 1E-8	3.73307	Bisec= 29 Bisec++ = 4 Newton=13	Bisec= 31 Bisec++ = 13 Newton=22
Exp(x)-3*x^2	[3, 4]	1E-8 1E-8	3.73307	Bisec= 27 Bisec++ = 3 Newton=6	Bisec= 29 Bisec++ = 11 Newton=14
(X-2.345) * (X-12.345) * (X-23.456)	[1, 11]	1E-8 1E-8	2.345	Bisec= 30 Bisec++ = 3 Newton=9	Bisec= 32 Bisec++ = 10 Newton=20
(X-2.345) * (X-12.345) * (X-23.456)	[11, 22]	1E-8 1E-8	12.345	Bisec= 29 Bisec++ = 4 Newton=6	Bisec= 31 Bisec++ = 11 Newton=14
LN(X^4)-X	[8,9]	1E-8 1E-8	8.613169	Bisec= 27 Bisec++ = 2 Newton=4	Bisec= 29 Bisec++ = 8 Newton=10
Cos(X)-X	[0.5,1]	1E-8 1E-8	0.73908	Bisec= 26 Bisec++ = 2 Newton=4	Bisec= 28 Bisec++ = 8 Newton=10

Table 3. Summary of the results comparing the Bisection, Bisection++ version 2, and Newton's method, with function tolerance value of 1E-8.

Table 4 shows the tests for version 2 of Bisection++ with a function tolerance value of 1E-4.

Function	[A, B]	Toler/FxToler	Root	Iterations	Num Fx Calls
Exp(x)-4*x^2	[3, 5]	1E-8 1E-4	4.30658	Bisec= 28 Bisec++ =3 Newton=6	Bisec= 30 Bisec++ = 11 Newton=14
Exp(x)-3*x^2	[1, 4]	1E-8 1E-4	3.73307	Bisec= 29 Bisec++ = 3 Newton=13	Bisec= 31 Bisec++ = 10 Newton=22
Exp(x)-3*x^2	[3, 4]	1E-8 1E-4	3.73307	Bisec= 27 Bisec++ = 2 Newton=6	Bisec= 29 Bisec++ = 8 Newton=14
(X-2.345) * (X-12.345) * (X-23.456)	[1, 11]	1E-8 1E-4	2.345	Bisec= 30 Bisec++ = 3 Newton=9	Bisec= 32 Bisec++ = 10 Newton=20
(X-2.345) * (X-12.345) * (X-23.456)	[11, 22]	1E-8 1E-4	12.345	Bisec= 29 Bisec++ = 4 Newton=6	Bisec= 31 Bisec++ = 11 Newton=14
LN(X^4)-X	[8,9]	1E-8 1E-4	8.613169	Bisec= 27 Bisec++ = 1 Newton=4	Bisec= 29 Bisec++ = 5 Newton=10
Cos(X)-X	[0.5,1]	1E-8 1E-4	0.73908	Bisec= 26 Bisec++ = 1 Newton=4	Bisec= 28 Bisec++ = 5 Newton=10

Table 4. Summary of the results comparing the Bisection, Bisection++ version 2, and Newton's method, with function tolerance value of 1E-4.

Tables 5 and 6 compare the results of the two versions of Bisection++ with the function tolerance value of 1E-8 and 1E-4, respectively.

Function	[A, B]	Version 1 Iterations	Version 1 Fx Calls	Version 2 Iterations	Version 2 Fx Calls
Exp(x)-4*x^2	[3, 5]	4	14	3	11
Exp(x)-3*x^2	[1, 4]	4	18	3	10
Exp(x)-3*x^2	[3, 4]	3	11	2	8
(X-2.345) * (X-12.345) * (X-23.456)	[1, 11]	4	12	3	10

Function	[A, B]	Version 1 Iterations	Fx Calls	Version 2 Iterations	Fx Calls
(X-2.345) * (X-12.345) * (X-23.456)	[11, 22]	4	11	3	9
LN(X^4)-X	[8,9]	2	8	1	5
Cos(X)-X	[0.5,1]	2	8	1	5

Table 5. Comparing the results of the two versions of Bisection++ with the function tolerance value of 1E-8.

Function	[A, B]	Version 1 Iterations	Fx Calls	Version 2 Iterations	Fx Calls
Exp(x)-4*x^2	[3, 5]	3	11	3	11
Exp(x)-3*x^2	[1, 4]	3	10	3	10
Exp(x)-3*x^2	[3, 4]	2	8	2	8
(X-2.345) * (X-12.345) * (X-23.456)	[1, 11]	3	10	3	10
(X-2.345) * (X-12.345) * (X-23.456)	[11, 22]	3	9	4	11
LN(X^4)-X	[8,9]	1	5	1	5
Cos(X)-X	[0.5,1]	1	5	1	5

Table 6. Comparing the results of the two versions of Bisection++ with the function tolerance value of 1E-4.

Looking at Tables 5 and 6, you can draw the conclusion that version 2 of the Bisection++ performs better that version 1, when the function tolerance is 1E-8. This edge is lost when function tolerance is 1E-4.

Conclusion

The Bisection++ algorithm offers significant improvement over the Bisection, Bisection Plus, and even Newton's method. I recommend version 1 of the Bisection++ method since it does not require sorting the interpolation data. The price for this approach may add one iteration to version 1, compared to version 2.

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 Algorithm*, http://www.namirshammas.com/NEW/BisPls.pdf.

Document Information

Version	*Date*	*Comments*
1.0.0	1/18/2014	Initial release.