The New Trisection and Trisection Plus Root-Seeking Algorithms

by

Namir Shammas

Introduction

This article presents a new root-bracketing algorithm and it's variant. The new algorithms, named the Trisection and Trisection Plus, compete with and enhance the Bisection method which is the slowest root-seeking method.

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
        o X =(A+B)/2
        o Fx = f(X)
        o If Fx*Fa > 0 then
             ▪ A = X
             ▪ Fa = Fx
        o Else
             ▪ B = X
             ▪ Fb = Fx
        o 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
      o h = 0.001 * (|X| + 1)
      o Fx = f(X)
      o Diff = h * Fx / (f(X+h) - Fx)
      o 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
      o 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
                o A = X1
                o Fa = Fx1
                o B = X2
                o Fb = Fx2
            • Else
                o A = X2
                o Fa = Fx2
            • End
        ▪ End
    o 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
                o A = X2
                o Fa = Fx2
                o B = X1
                o Fb = Fx1
            • Else
                o B = X2
                o 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
    o LastA = A
    o LastB = B
    o 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
                   o Else
                        ▪ A = X3
                        ▪ Fa = X3
                        ▪ B = X2
                        ▪ Fb = Fx2
                   o End
              • End
         ▪ End
    o End
    o If A > B then
         ▪ Swap A, B
         ▪ Swap Fa, Fb
         ▪ Swap LastA, LastB
    o End
    o If LastA <> A And |A – LastA| < Toler then exit loop
    o If LastB <> B And |B – LastB| < Toler then exit loop
• Until |A – B| < Toler Or |Fa| < FxToler Or |Fb| < FxToler
• If |Fa| < |Fb| Then
    o Return A
• Else
    o Return B
• End
```

Despite the length of the pseudo-code, it is not really complicated. When the code is executed in an implementation of the above pseudo-code, only a fraction of the statements are executed in each iteration. It's just there are many alternate sets of statements 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.

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:

```
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) - X2 * (Fx1 - 0)) / (Fx2 - Fx1)

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 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 = ""
  A = [A2].Value
  B = [A4].Value
  Toler = [A6].Value
  FxToler = [A8].Value
  sFx = [A10].Value


  ' Bisection
  Fa = MyFx(sFx, A)
  Fb = MyFx(sFx, B)
```

```
NumIters = 2
R = 3
Col = 2
Do
  X1 = (A + B) / 2
  Fx1 = MyFx(sFx, X1)
  NumIters = NumIters + 1
  If Fx1 * Fa > 0 Then
    A = X1
    Fa = Fx1
  Else
    B = X1
    Fb = Fx1
  End If
  Cells(R, Col) = A
  Cells(R, Col + 1) = B
  R = R + 1

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
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

  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
      B = X1
      Fb = Fx1
    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
        A = X1
        Fa = Fx1
        B = X2
        Fb = Fx2
      Else
```

```
          ' case 2: [X2,B] has root
          A = X2
          Fa = Fx2
        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
        A = X1
        Fa = Fx1
      Else
        X2 = B - 2 * (B - A) / 3
        Fx2 = MyFx(sFx, X2)
        NumIters = NumIters + 1
        ' case 5: [X1,X2] has root
        If Fx1 * Fx2 < 0 Then
          A = X1
          Fa = Fx1
          B = X2
          Fb = Fx2
        Else
          ' case 6: [A,X2] has root
          B = X2
          Fb = Fx2
        End If
      End If
    End If

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

  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) = Fx()
    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 Bisection method, 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	Bisection		Trisection		Trisection Plus		Newton	
1	A	B	A	B	A	B	X	
B	1.5	2	1.666666667	2	1.840376801	2	1.986679131	0
2	1.75	2	1.888888889	1.777777778	1.856820732	1.893584534	1.865982452	0
Toler	1.75	1.875	1.888888889	1.851851852	1.857181284	1.869075332	1.857244749	0
1.00E-10	1.8125	1.875	1.864197531	1.851851852	1.857183854	1.861145967	1.857183967	0
FxToler	1.84375	1.875	1.855967078	1.860082305	1.857183854	Fx Calls=10	1.85718386	0
1.00E-07	1.84375	1.859375	1.855967078	1.85733882			1.85718386	0
Function	1.8515625	1.859375	1.856881573	1.85733882			1.85718386	0
EXP(X)-X^3	1.85546875	1.859375	1.857186405	1.857033989			1.85718386	Fx Calls=14
	1.85546875	1.857421875	1.857186405	1.857135599				
	1.856445313	1.857421875	1.857186405	1.857169469				
	1.856933594	1.857421875	1.857186405	1.857180759				
	1.857177734	1.857421875	1.857184523	1.857182641				
	1.857177734	1.857299805	1.857183896	1.857183268				
	1.857177734	1.85723877	1.857183896	1.857183687				
	1.857177734	1.857208252	1.857183896	1.857183826				
	1.857177734	1.857192993	1.857183849	1.857183872				
	1.857177734	1.857185364	1.857183849	Fx Calls=24				
	1.857181549	1.857185364						
	1.857183456	1.857185364						
	1.857183456	1.85718441						
	1.857183456	1.857183933						
	1.857183695	1.857183933						

Figure 1. The Excel spreadsheet used to compare the Bisection, 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 contents of 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 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 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

My aim is to significantly accelerate the Trisection method compared to the Bisection method. I was also hoping that the Trisection Plus method perform comparable to Newton's method. The results proved my optimism to be well founded. 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	Bisec= 24 Trisec= 16 Trisec+ = 4 Newton= 7	Bisec= 26 Trisec = 24 Trisec+= 10 Newton= 14
Exp(X) – 3*X^2	[3, 4]	1E–10 1E–7	3.73307	Bisec= 26 Trisec= 17 Trisec+ = 5 Newton= 7	Bisec= 28 Trisec= 24 Trisec+ = 12 Newton= 14
Cos(X) – X	[0, 1]	1E–10 1E–7	0.73908	Bisec= 23 Trisec= 14 Trisec+ = 4 Newton= 5	Bisec= 25 Trisec= 21 Trisec+ = 10 Newton= 10

Function	[A, B]	Toler / FxToler	Root	Iterations	Num Fx Calls
(X–1.234) * (X–5.678) * (X+12.345)	[5, 6]	1E–10 1E–7	5.678	Bisec= 28 Trisec= 18 Trisec+ = 4 Newton=6	Bisec= 30 Trisec= 23 Trisec+ = 10 Newton=12
(X–1.234) * (X–5.678) * (X+12.345)	[1, 2]	1E–10 1E–7	1.234	Bisec= 28 Trisec= 17 Trisec+ = 4 Newton= 5	Bisec= 30 Trisec= 23 Trisec+ = 10 Newton= 10
(X–1.234) * (X–5.678) * (X+12.345)	[5,11]	1E–10 1E–7	5.678	Bisec= 29 Trisec= 19 Trisec+ = 6 Newton= 7	Bisec= 31 Trisec= 32 Trisec+ = 14 Newton= 14
(X–1.234) * (X–5.678) * (X+12.345)	[–8, –15]	1E–10 1E–7	–12.345	Bisec= 33 Trisec= 20 Trisec+ = 5 Newton= 5	Bisec= 35 Trisec= 30 Trisec+ = 13 Newton= 10

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

The above table shows that the Trisection method performs better than the Bisection method, but not as good as Newton's method. This is within my initial expectations. I am glad to see that, on the other hand, the Trisection Plus performs as good as or 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.

Conclusion

The Trisection Plus algorithm offers significant improvement over the Bisection method. The new algorithm has an efficiency that is somewhat comparable to that of Newton's method.

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

Document Information

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