The New Bisection Plus Algorithm
by
Namir Shammas

Introduction

This article presents a new variant for the root-bracketing Bisection algorithm. This new version injects a lot of punch to 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 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| < 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

Copyright © 2014 by Namir Clement Shammas

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 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
      o h = 0.001 * (|X| + 1)
      o Fx = f(X)
      o Diff = h * Fx / (f(X+h) - Fx)
      o 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 Plus methods. Thus, the algorithm derives its single initial guess as 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 $f(x_1)$.

Copyright © 2014 by Namir Clement Shammas

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 $f(X_1)\times f(X_2)$ 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 $f(X_1)\times f(X_2)$ 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.

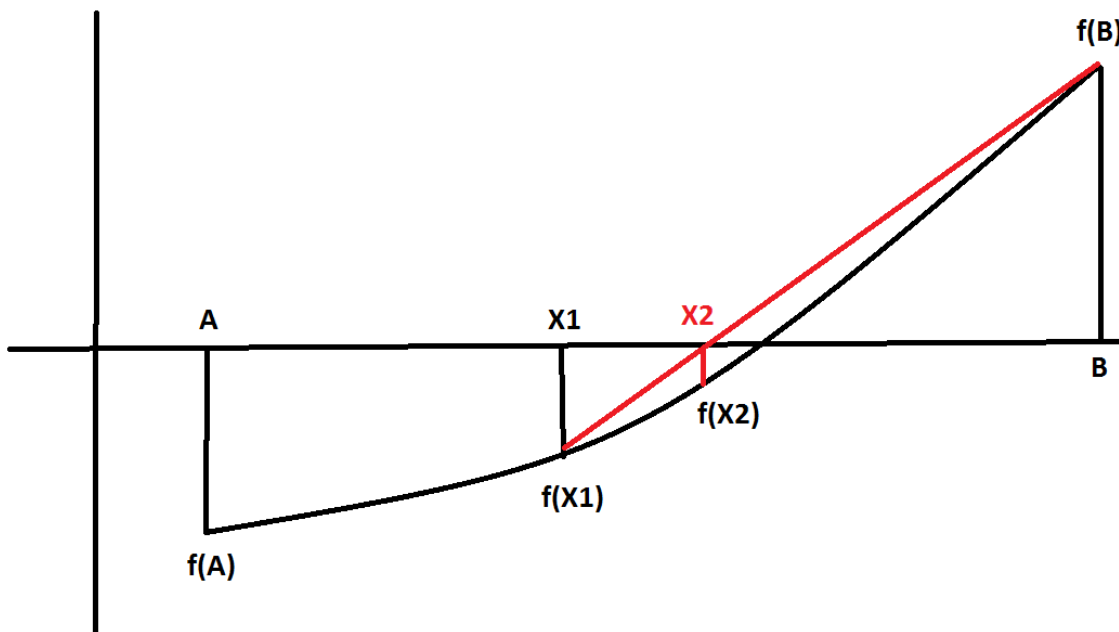
Let me present the pseudo-code for the Bisection Plus method:

```
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
      o X1=(A+B)/2
      o Fx1 = f(X1)
      o If Fx1*Fa > 0 then
          ▪ Slope = (Fb – Fx1)/(B – X1)
          ▪ Intercept = Fb – Slope * B
      o Else
```

Copyright © 2014 by Namir Clement Shammas

        - 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.



Copyright © 2014 by Namir Clement Shammas

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)|.

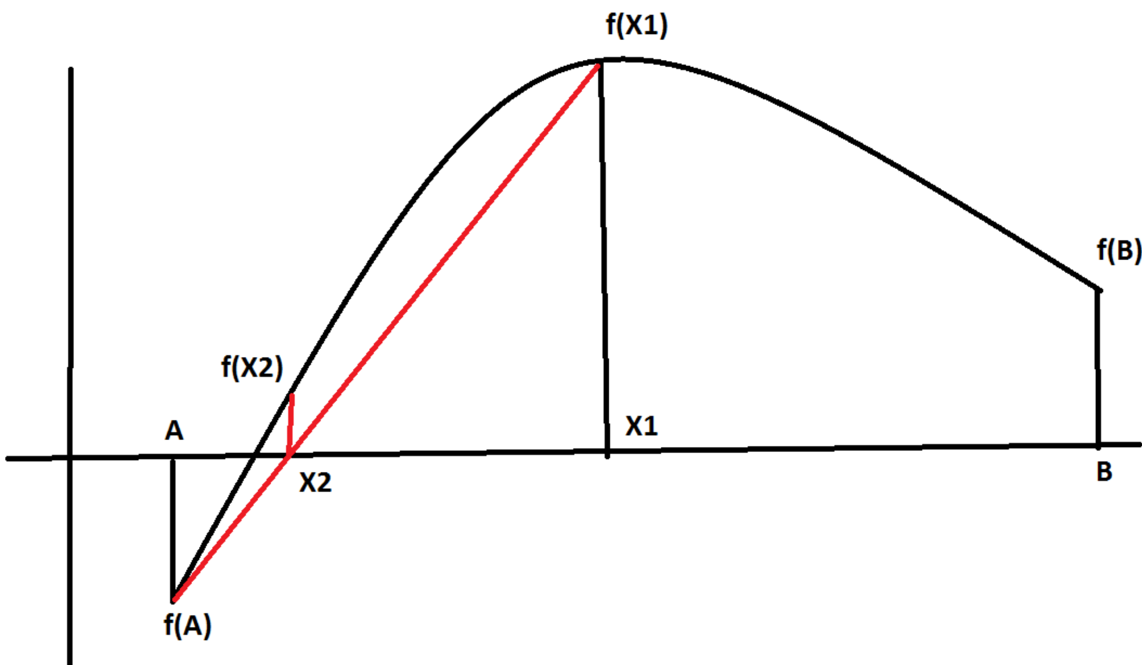


Figure 2. Special cases that dictates calculate X2 using the suggested scheme.

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
```

Copyright © 2014 by Namir Clement Shammas

```
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

Sub Go()
  ' Bisection Plus algorithm
  ' Version 1.00  1/9/2014
  ' Copyright (c) 2014 Namir Clement Shammas
  Dim R As Integer, Count As Long, NumFxCalls As Integer
  Dim A As Double, B As Double, X As Double, LastX 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
  Dim h As Double, Diff As Double
  Dim sFx As String

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

Copyright © 2014 by Namir Clement Shammas

```
    Cells(R, 3) = B
    R = R + 1
  Loop
  Cells(R, 2) = (A + B) / 2
  Cells(R, 3) = "FX Calls=" & NumFxCalls

  ' Bisection 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

    ' does [X1,X2] define a new root-bracketing interval?
    If FX1 * FX2 < 0 Then
      A = X1
      FA = FX1
      B = X2
      FB = FX2
      Cells(R, 6) = "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
```

Copyright © 2014 by Namir Clement Shammas

```
    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, 7) = X
    Cells(R, 8) = Diff
    R = R + 1
  Loop Until Abs(Diff) <= Toler
  Cells(R, 7) = X
  Cells(R, 8) = "FX Calls=" & NumFxCalls
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. 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 **Go** performs the root-seeking calculations that compare the Bisection method, Bisection Plus method, and Newton's method. Figure 2 shows a snapshot of the Excel spreadsheet used in the calculations for the Bisection, Bisection Plus, and Newton's method.

Copyright © 2014 by Namir Clement Shammas

	A	B	C	D	E	F	G	H
1	A	Bisection		BisectionPlus			Newton's Method	
2	3	A	B	A	B	Status	X	Diff
3	B	4	5	4.162619536	5		4.413907218	-0.4139072
4	5	4	4.5	4.581309768	4.275781869	New interval	4.315740113	0.09816711
5	Toler	4.25	4.5	4.428545819	4.303585013	New interval	4.30669389	0.00904622
6	1.00E-08	4.25	4.375	4.366065416	4.306439906	New interval	4.306585219	0.00010867
7	F(X)	4.25	4.3125	4.336252661	4.306581198	New interval	4.30658473	4.8899E-07
8	EXP($X)-4*$X^2	4.28125	4.3125	4.32141693	4.306584685	New interval	4.306584728	2.1561E-09
9		4.296875	4.3125	4.314000807	4.306584728	New interval	4.306584728	FX Calls=14
10		4.3046875	4.3125	4.306584728	FX Calls=18	New interval		
11		4.3046875	4.30859375					
12		4.3046875	4.306640625					
13		4.305664063	4.306640625					
14		4.306152344	4.306640625					
15		4.306396484	4.306640625					
16		4.306518555	4.306640625					
17		4.30657959	4.306640625					
18		4.30657959	4.306610107					
19		4.30657959	4.306594849					
20		4.30657959	4.306587219					
21		4.306583405	4.306587219					
22		4.306583405	4.306585312					
23		4.306584358	4.306585312					
24		4.306584358	4.306584835					
25		4.306584597	4.306584835					
26		4.306584716	4.306584835					
27		4.306584716	4.306584775					
28		4.306584716	4.306584746					
29		4.306584716	4.306584731					
30		4.306584723	4.306584731					
31		4.306584727	FX Calls=30					

Figure 2. The Excel spreadsheet used to comparing the Bisection, Bisection Plus, 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 tolerance value.
- Cell A8 contains the expression for f(X)=0. Notice that the contents of cell A8 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 Plus method. Column F displays comments made by the code that point out when X_1 and X_2 form a new root-bracketing interval. 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.

Copyright © 2014 by Namir Clement Shammas

- Columns G and H display the refined guess for the root and the refinement value, respectively, using Newton's method. The bottom most value, in column G, 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 Plus method compared to the Bisection method. I do not expect the Bisection Plus method to outperform Newton's method. The results proved me partial wrong regarding the last statement. 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 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	Root	Iterations	Num Fx Calls
Exp(x)-4*x^2	[3, 5]	1E-8	4.30658	Bisec= 28 Bisec+ = 7 Newton=6	Bisec= 30 Bisec+ = 18 Newton=14
Exp(x)-3*x^2	[1, 4]	1E-8	3.73307	Bisec= 29 Bisec+ = 7 Newton=10	Bisec= 31 Bisec+ = 18 Newton=22
Exp(x)-3*x^2	[3, 4]	1E-8	3.73307	Bisec= 27 Bisec+ = 6 Newton=6	Bisec= 29 Bisec+ = 16 Newton=14
(X-2.345) * (X-12.345) * (X-23.456)	[1, 11]	1E-8	2.345	Bisec= 30 Bisec+ = 5 Newton=9	Bisec= 32 Bisec+ = 14 Newton=20
(X-2.345) * (X-12.345) * (X-23.456)	[11, 22]	1E-8	12.345	Bisec= 29 Bisec+ = 4 Newton=6	Bisec= 31 Bisec+ = 12 Newton=14
LN(X^4)-X	[8,9]	1E-8	8.66125	Bisec= 27 Bisec+ = 4 Newton=4	Bisec= 29 Bisec+ = 12 Newton=10
Cos(X)-X	[0.5,1]	1E-8	0.73908	Bisec= 26 Bisec+ = 3 Newton=4	Bisec= 28 Bisec+ = 10 Newton=10

Copyright © 2014 by Namir Clement Shammas

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

The above table shows that the Bisection Plus method consistently outperforms the Bisection method. The table also highlights in red the cases where the Bisection Plus 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.

Conclusion

The Bisection 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.

Document Information

Version	***Date***	***Comments***
1.0.0	1/9/2014	Initial release.
1.0.1	1/10/2014	Edit of general description of algorithm
1.0.2	1/18/2014	Minor edits.

Copyright © 2014 by Namir Clement Shammas