

New Root-Seeking Algorithms

By

Namir C Shammas

This article presents a pair of new root-seeking algorithms that use an innovative approach. Testing these algorithms shows that they can reach a good approximation to the root in less iterations and/or fewer function calls than Newton's method.

Overview of Newton's Method

Newton's method starts with a guess for the root and iterates to refine that guess. Each iteration requires the calculations of the function and the slope at the current guess, and yields a new guess for the root. The algorithm discards the older guesses, replacing them with the new ones. Typically, the value of the slope is approximated by a finite difference method. Thus, each iteration in Newton's method requires two function calls. Other algorithms that converge at a faster rate use approximations to higher derivatives and thus make more function calls per iteration in order to approximate the higher derivatives. In the majority of the root-seeking methods, that do not bracket the root, the iterations refine the guess for the root until it converges to an acceptable value that is close enough to the actual root.

To solve for:

$$f(x^*) = 0$$

Newton's algorithm uses the following equation to refine the guess for the root:

$$x_1 = x_0 - f(x_0) / f'(x_0)$$

Where $f'(X)$ is the derivative of $f(X)$ with respect to X . You can evaluate an approximation to the derivative using, among others, the following equation:

$$f'(x) = [f(x + h) - f(x)] / h$$

Where h is a small increment and can be estimated as:

$$h = 0.01 * (|x| + 1)$$

Probing Algorithms

The two new algorithms that I designed share a common approach, yet differ in the details of their execution. The basic idea is to start with an initial guess which becomes a fixed vantage point. From this vantage point, the algorithm uses probes to find the root. Each probe generates a new value for the variable X and its corresponding function value. This function value gives an assessment of how good the

probed value of X is. The iterations of the probing algorithms strive to obtain better probed values. During the iterations, the initial guess remains fixed.

Here is the abstract version of the probing algorithms:

1. Given an initial guess X_0 .
2. Initialize an array of N probes. For each probe obtain a new value of $X(i)$ and $f(X(i))$.
3. Sort the first three probes using the absolute value of $f(X)$.
4. Repeat the next steps
 - a. Use the current array of N probes to interpolate a new probe value $X(N+1)$ and its function value $f(X(N+1))$.
 - b. Using the absolute value of $f(X)$, insert the new probe in the array of probes, such that the array of probes remains sorted. This step places the best probe at the array index 1. Likewise, the worst probe appears at the array index $N+1$.
 - c. If the probes show that the convergence criteria have been met, resume in step 4. Otherwise, perform another loop iteration.
5. The best guess for the root is $X(1)$

The Probing Slopes Algorithm

The first variant of the probing algorithms attempts to find the critical slope that would zoom in on the root. In other words, the algorithm attempts to find the slope of the straight line connecting the initial guess point with the root. The initial guess point is $(X_0, f(X_0))$, where initial guess X_0 . The root point is $(X^*, 0)$. The critical straight line passing through these two points is:

$$\text{Critical slope} = (f(X_0) - 0) / (X_0 - X^*)$$

Calculating the critical slope is not possible since we do not know the value of the root, X^* . The algorithm starts by calculating an array of three slopes, using the following equations:

$$S(1) = [f(X + h) - f(X)] / h$$

$$S(2) = (1 + r) S(1)$$

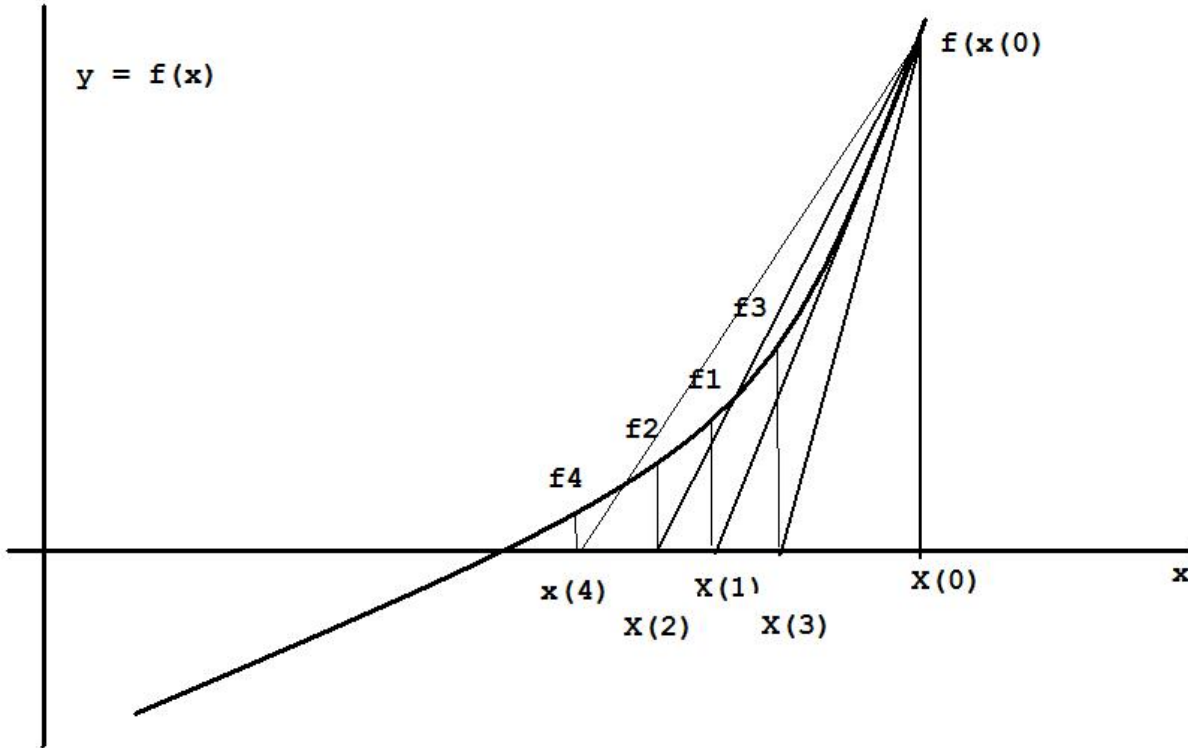
$$S(3) = (1 - r) S(1)$$

Where r is a small number like 0.15 and 0.10. Notice that calculating the values for slopes $S(2)$ and $S(3)$ does not require evaluating the function $f(x)$.

The algorithm uses the array of slopes to calculate the array of probed root values. The algorithm uses the following equation:

$$X(i) = X_0 - f(X_0) / S(i), \text{ for } i = 1, 2, \text{ and } 3$$

Next, the algorithm calculates the array of function values $Fx(i) = f(X(i))$. Finally, sort the first three probes using the absolute values of the array $Fx()$ as the sort key.



The steps up till now supply the algorithm with its initial probes. Each one of the three probes is made up of the values of a slope $S(i)$, probed root value $X(i)$, and probed function value $F_x(i)$. Armed with the probes the algorithm starts its main loop. Each loop iteration performs the following tasks:

1. Interpolate the first three elements of arrays $S()$ and $F_x()$ to calculate the slope $S(4)$ for when the function value is 0. The calculated slope is designated as $S(4)$ and represents a new element of the array of slopes $S()$.
2. Calculate the probed root value, $X(4) = X_0 - f(X_0) / S(4)$, and then the probed function value, $F_x(4) = f(X(4))$.
3. Using the absolute value of $F_x(X)$, insert the new probe in the array of probes, such that the array of probes remains sorted. This step places the best probe at the array index 1. Likewise, the worst probe appears at the array index 4.

The above loop ends when one of the following conditions is true:

- The value $|X(1) - X(2)|$ fall below an acceptable tolerance limit.
- The absolute value of $F_x(1)$ is less than the function tolerance value.
- The sorting step in task 3 fails to rearrange the array.

If the convergence conditions fail, the next iteration occurs. Since task 1 uses the first three elements of arrays $S()$, $X()$, and $Fx()$, the values of the elements at index 4 are ignored. In fact, tasks 1 and 2 overwrite these values. As you can see in task 2, the algorithm retains the value of the initial guess $X0$. Why retain this value and not update it with the value of $X(1)$? The answer lies with the fact that replacing $X0$ with a new value requires calculating a new set of values for $S(1)$, $S(2)$, $S(3)$, $X(1)$, $X(2)$, $X(3)$, $Fx(1)$, $Fx(2)$, and $Fx(3)$. Calculating these values comes at a price of making more evaluations of function $f(X)$. This rise in the number of function evaluation makes the algorithm less efficient than Newton's algorithm. By keeping the original guess for the root, the algorithm economizes on the number of invocation of $f(x)$ without compromising on the speed of convergence. Since each iteration in the main loop requires only one invocation of $f(x)$, compared to two for Newton's method, the Probing Slopes Algorithm tends to have an advantage. Using an efficient interpolation method, like the Lagrangian interpolation, adds to the overall efficiency of the implementation of the algorithm. Thus, the Probing Slopes Algorithm gains more advantage over Newton's method.

The Probing Steps Algorithm

The second variant of the probing algorithms attempts to find the critical step that would zoom in on the root in one swoop. Of course, such a magical step does not exist. Instead, we can modify the Probing Slopes Algorithm to work with steps of X instead of function slopes.

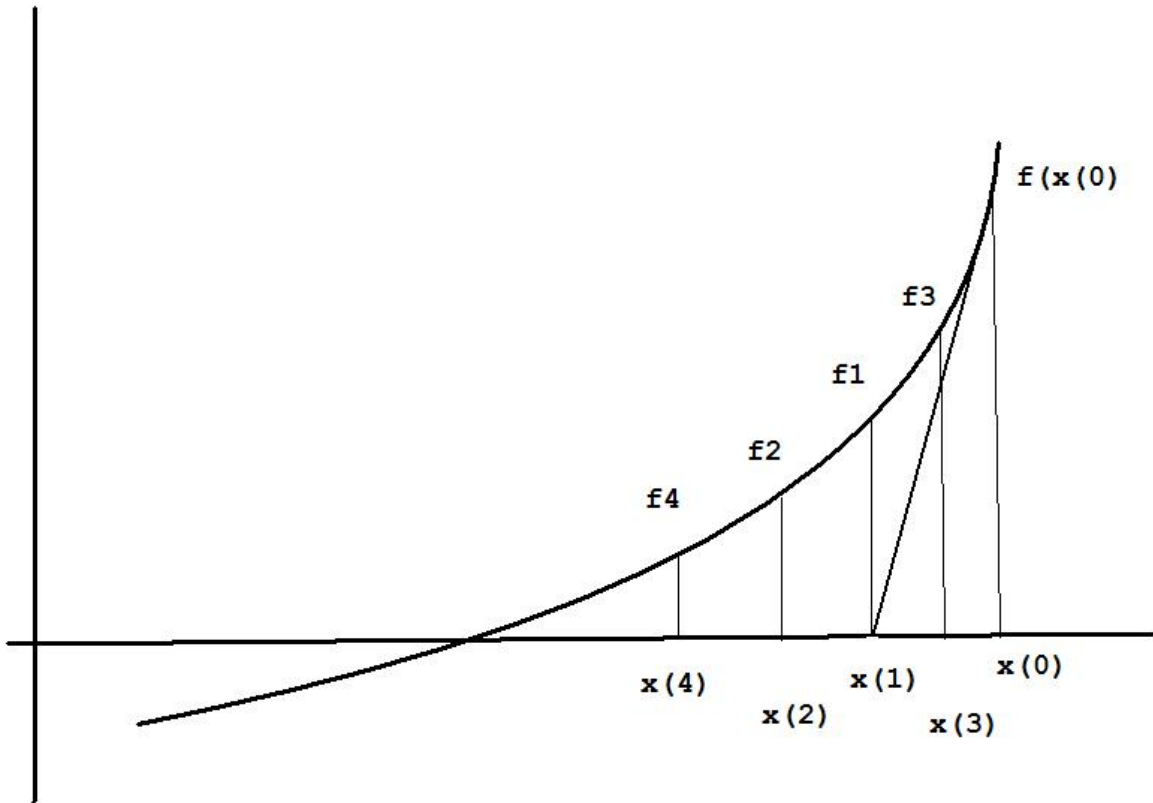
Calculating the critical step is not possible since we do not know the value of the root, X^* . The algorithm starts by calculating an array of three steps, using the following equations:

$$St(1) = h * f(x) / [f(x + h) - f(x)]$$

$$St(2) = (1 + r) St(1)$$

$$St(3) = (1 - r) St(1)$$

Where r is a small number like 0.15 and 0.1. Notice that calculating the values for steps $St(2)$ and $St(3)$ does not require evaluating the function $f(x)$.



The algorithm uses the array of slopes to calculate the array of probed root values. The algorithm uses the following equation:

$$x(i) = x_0 - St(i), \text{ for } i = 1, 2, \text{ and } 3$$

Next, the algorithm calculates the array of function values $F_x(i) = f(X(i))$. Finally, sort the first three probes using the absolute values of the array $F_x()$ as the sort key.

The steps up till now supply the algorithm with its initial probes. Each one of the three probes is made up of the values of a step $St(i)$, probed root value $X(i)$, and probed function value $F_x(i)$. Armed with the probes the algorithm starts its main loop. Each loop iteration performs the following tasks:

1. Interpolate the first three elements of arrays $St()$ and $F_x()$ to calculate the step $St(4)$ for when the function value is 0. The calculated step is designated as $St(4)$ and represents a new element of the array of steps $St()$.
2. Calculate the probed root value, $X(4) = X_0 - St(4)$, and then the probed function value, $F_x(4) = f(X(4))$.
3. Using the absolute value of $F_x(X)$, insert the new probe in the array of probes, such that the array of probes remains sorted. This step places the best probe at the array index 1. Likewise, the worst probe appears at the array index 4.

The above loop ends when one of the following conditions is true:

- The value $|X(1) - X(2)|$ fall below an acceptable tolerance limit.
- The absolute value of $Fx(1)$ is less than the function tolerance value.
- The sorting step in task 3 fails to rearrange the array.

If the convergence conditions fail, the next iteration occurs. Since task number 1 uses the first three elements of arrays $St()$, $X()$, and $Fx()$, the values of the elements at index 4 are not used. In fact, tasks 1 and 2 overwrite these values. As you can see in task 2, the algorithm retains the value of the initial guess $X0$, for the same reason as the Probing Slopes Algorithm.

Testing the Probing Slopes Algorithm

This section presents the tests results that compare the Probing Slopes Algorithm with Newton’s method. The test uses an Excel spreadsheet that has the following visual interface:

	A	B	C	D	E	F	G
1	X	Slope	New X	Fx		New X	Fx
2	5.5	120.66868	4.22425941636	14.79078241		4.7990677737815	52.30404061
3	X Toler	95.09317731	3.88114640167	3.289868331		4.2543350693861	16.11188322
4	1.00E-08	88.06838164	3.75201815450	3.74E-01		3.9086529665032	3.999077626
5	Fx Toler	87.15135825	3.73362555267	1.06E-02		3.7631043999835	0.599098064
6	1.00E-08	87.12444275	3.73307986372	1.62E-05		3.7351130194856	0.039552189
7		87.12440158	3.73307902864	1.02E-10		3.7331692037868	0.00175037
8						3.7330828753212	7.47E-05
9						3.7330791924378	3.18E-06
10		Num Fx Calls				3.7330790356077	1.35376E-07
11		11				3.733079029	5.76E-09
12							
13							
14						Num Fx Calls	
15						21	
16							

The VBA code used for this test is:

```
Option Explicit
Option Base 0
```

```
Function F(X As Double) As Double
    F = Exp(X) - 3 * X ^ 2
End Function
```

```
Sub RootByProbingSlopes()
    Const MAXITER = 100
    Const NUM_SLOPES = 3
    Const SLOPE_FACTOR = 0.15

    Dim XToler As Double, FxToler As Double
    Dim R As Integer, NumCalls As Integer, Iters As Integer
    Dim X As Double, S(NUM_SLOPES + 1) As Double, Fx(NUM_SLOPES + 1) As Double
    Dim Xnew(NUM_SLOPES + 1) As Double, Xold As Double
    Dim I As Integer, J As Integer
    Dim Sum As Double, Prod As Double, FxVal As Double, Diff As Double
    Dim h As Double, Buff As Double
    Dim Fxtmp As Double, Stemp As Double, Xtem As Double
    Dim bNoSwap As Boolean
```

```

' method of shift slopes
X = Cells(2, 1)
XToler = Cells(4, 1)
FxToler = Cells(6, 1)
R = 2
FxVal = F(X)
NumCalls = 1
Range("B2:Z10000").Value = ""

h = 0.01 * (1 + Abs(X))
S(1) = (F(X + h) - FxVal) / h
NumCalls = NumCalls + 1
S(2) = S(1) * (1 + SLOPE_FACTOR)
S(3) = S(1) * (1 - SLOPE_FACTOR)
For I = 1 To 3
    Xnew(I) = X - FxVal / S(I)
    Fx(I) = F(Xnew(I))
    NumCalls = NumCalls + 1
Next I

' Sort first three probes
For I = 1 To NUM_SLOPES - 1
    For J = I + 1 To NUM_SLOPES
        If Abs(Fx(I)) > Abs(Fx(J)) Then

            Buff = Fx(I)
            Fx(I) = Fx(J)
            Fx(J) = Buff

            Buff = S(I)
            S(I) = S(J)
            S(J) = Buff

            Buff = Xnew(I)
            Xnew(I) = Xnew(J)
            Xnew(J) = Buff

        End If
    Next J
Next I

Iters = 0

Do
    Iters = Iters + 1
    If Iters > MAXITER Then Exit Do
    Sum = 0
    For I = 1 To NUM_SLOPES

```



```

    Prod = S(I)
    For J = 1 To NUM_SLOPES
      If I <> J Then
        Prod = Prod * (0 - Fx(J)) / (Fx(I) - Fx(J))
      End If
    Next J
    Sum = Sum + Prod
  Next I

  S(4) = Sum
  Xnew(4) = X - FxVal / S(4)
  Cells(R, 2) = S(4)
  Cells(R, 3) = Xnew(4)
  Fx(4) = F(Xnew(4))
  NumCalls = NumCalls + 1
  Cells(R, 4) = Fx(4)
  R = R + 1

  ' store last probe in index 0
  S(0) = S(4)
  Fx(0) = Fx(4)
  Xnew(0) = Xnew(4)

  ' remove worst fx() value
  bNoSwap = True
  For I = 1 To NUM_SLOPES - 1
    If Abs(Fx(0)) < Abs(Fx(I)) Then
      bNoSwap = False
      For J = NUM_SLOPES To I Step -1
        Fx(J) = Fx(J - 1)
        S(J) = S(J - 1)
        Xnew(J) = Xnew(J - 1)
      Next J
      ' restore latest problem
      S(I) = S(0)
      Fx(I) = Fx(0)
      Xnew(I) = Xnew(0)
    End If
  Exit For
Next I

Loop Until bNoSwap Or Abs(Xnew(1) - Xnew(2)) <= XToler Or Abs(Fx(1))
<= FxToler

Cells(R + 2, 2) = "Num Fx Calls"
Cells(R + 3, 2) = NumCalls
' flag end of iteration because the last guess failed to improve
Cells(R + 4, 2) = IIf(bNoSwap, "Iter Halted", "")

' now test Newton's method

```

```

X = Cells(2, 1)
FxVal = F(X)
R = 2
NumCalls = 1

Iters = 0

Do
  Iters = Iters + 1
  If Iters > MAXITER Then Exit Do
  h = 0.01 * (1 + Abs(X))
  Diff = h * FxVal / (F(X + h) - FxVal)
  NumCalls = NumCalls + 1
  X = X - Diff
  FxVal = F(X)
  NumCalls = NumCalls + 1
  Cells(R, 6) = X
  Cells(R, 7) = FxVal
  R = R + 1
Loop Until Abs(Diff) <= XToler Or Abs(FxVal) < FxToler

Cells(R + 2, 6) = "Num Fx Calls"
Cells(R + 3, 6) = NumCalls

End Sub

```

The above code shows the function F which contains a sample test f(x) function. The code for function F changes as the test function f(x) changes.

The Function $f(X)=\text{Exp}(X)-3*X^2$

The first test function is $f(X) = \text{Exp}(X)-3*X^2$. The following figure shows a graph for this function.

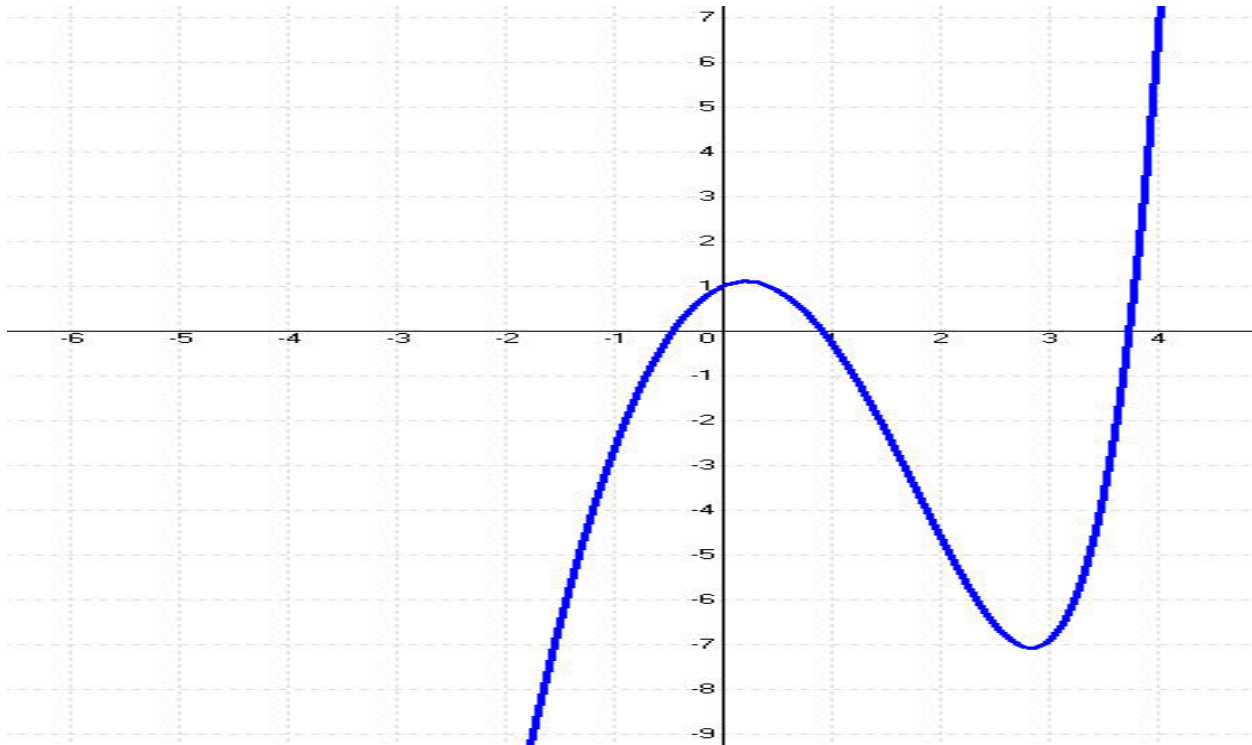


Table 1. Test results for $f(x)=\text{Exp}(x)-3*x^2$ for $X_{\text{toler}} = 1\text{E}-8$ and $F_{\text{xToler}} = 1\text{E}-8$.

Initial Guess	PSA Iterations	PSA Fx Calls	PSA Result/Fx Value	Newton Iterations	Newton Fx Calls	Newton Result/Fx Value
7	8	13	3.73307902 7.658E-12	12	25	3.73307902 3.0184E-09
6	7	12	3.73307902 1.758E-13	11	23	3.73307902 1.7948E-9
5	6	11	3.73307902 -7.105E-15	10	21	3.73307902 6.4740E-10
4	4	9	3.73307902 2.0605E-13	8	17	3.73307902 7.0187E-10
3	10	15	3.73307902 6.7501E-14	12	25	3.73307902 1.2888E-09
1	3	8	0.91000757 9.9503E-11	5	11	0.91000757258 -2.8452E-10
0	6	11	-0.4589622 -3.538E-10	7	15	-0.4589622 -1.9023E-10
-1	4	9	-0.4589622 -3.2224E-14	6	13	-0.4589622 -2.187558E-10
-2	5	10	-0.4589622 -1.4488E-14	7	15	-0.4589622 -2.041214E-10
-3	5	10	-0.4589622 -6.7522E-10	5	15	-0.4589622 7.5226654E-09

Table 1 shows the results for function $f(X) = e^X - 3X^2$ for X tolerance of $1E-8$ and function tolerance of $1E-8$. The table shows a set of initial guesses supplied to the VBA subroutine RootByProbingSlopes. The tested function has three roots near 3.733, 0.9100, and -0.4589. The table shows that for each initial guess, the Probing Slopes Algorithm required less iterations and functions calls than Newton’s method.

The Function $f(X) = \exp(-X) - \exp(-3)$

The second function is $f(X) = \exp(-X) - \exp(-3)$ whose graph appears in the next figure:

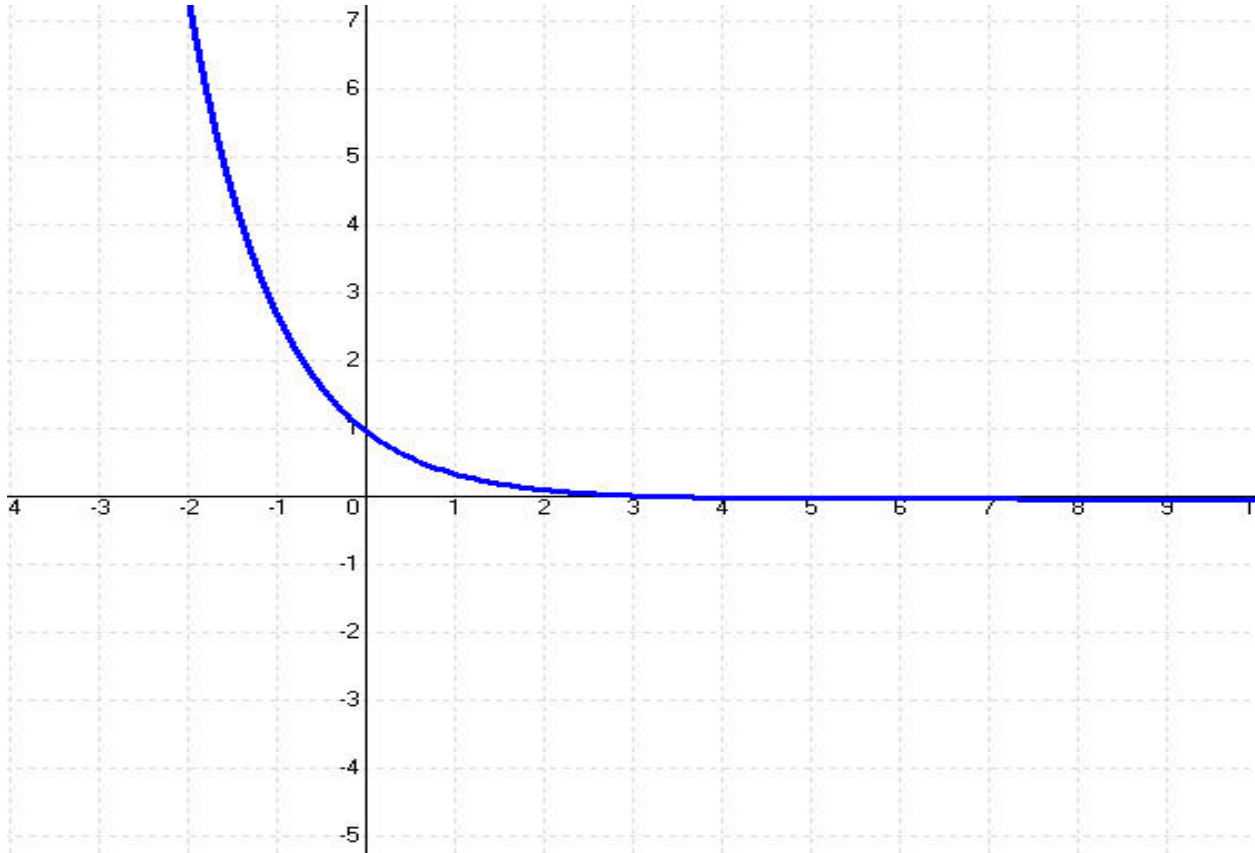


Table 2. Test results for $f(X)=\exp(-X)- \exp(-3)$ for $Xtoler =1E-8$ and $FxToler = 1E-8$

Initial Guess	PSA Iterations	PSA Fx Calls	PSA Result/Fx Value	Newton Iterations	Newton Fx Calls	Newton Result/Fx Value
-2	8	13	2.9999999998 7.62157E-12	10	21	3.000000016057 -7.994606E-10
-1	7	12	2.99999999987 6.11515E-12	9	19	3.00000001687 -8.402010E-10
0	6	11	2.99999999984	8	17	3.00000001601

Initial Guess	PSA Iterations	PSA Fx Calls	PSA Result/Fx Value	Newton Iterations	Newton Fx Calls	Newton Result/Fx Value
			7.8517956E-12			-7.973450E-10
1	4	9	2.9999999967 1.6284092E-11	7	15	3.0000001166 -5.805927E-10
2	5	10	2.9999999918 4.0644182E-11	5	11	2.9999987482 6.2321192E-09
4	5	10	2.9999992977 3.4963635E-09	6	13	3.0000008045 -4.005808E-09
5	11	16	2.9999999999 3.5638159E-14	10	21	3.00000065888 -3.280370E-09
6	24	29	2.999999950 2.4448264E-10	22	45	3.0000004812 -2.395785E-09

Table 2 shows the results for function $f(X) = e^{-X} - e^{-3}$ for X tolerance of 1E-8 and function tolerance of 1E-8. The table shows a set of initial guesses supplied to the VBA subroutine RootByProbingSlopes. The tested function has a root of 3. The table shows that for each initial guess below the root value, the Probing Slopes Algorithm required less iterations and functions calls than Newton’s method. The value of X = 6 is an exception since Newton’s method required 2 iterations less, but made 16 additional function calls.

The Test Polynomial

The third test function is:

$$f(x) = 0.005 * (x+5) * (x+3) * (x+1) * (x-5) * (x-3) * (x-1)$$

Which has the following figure shows a graph of the test polynomial.

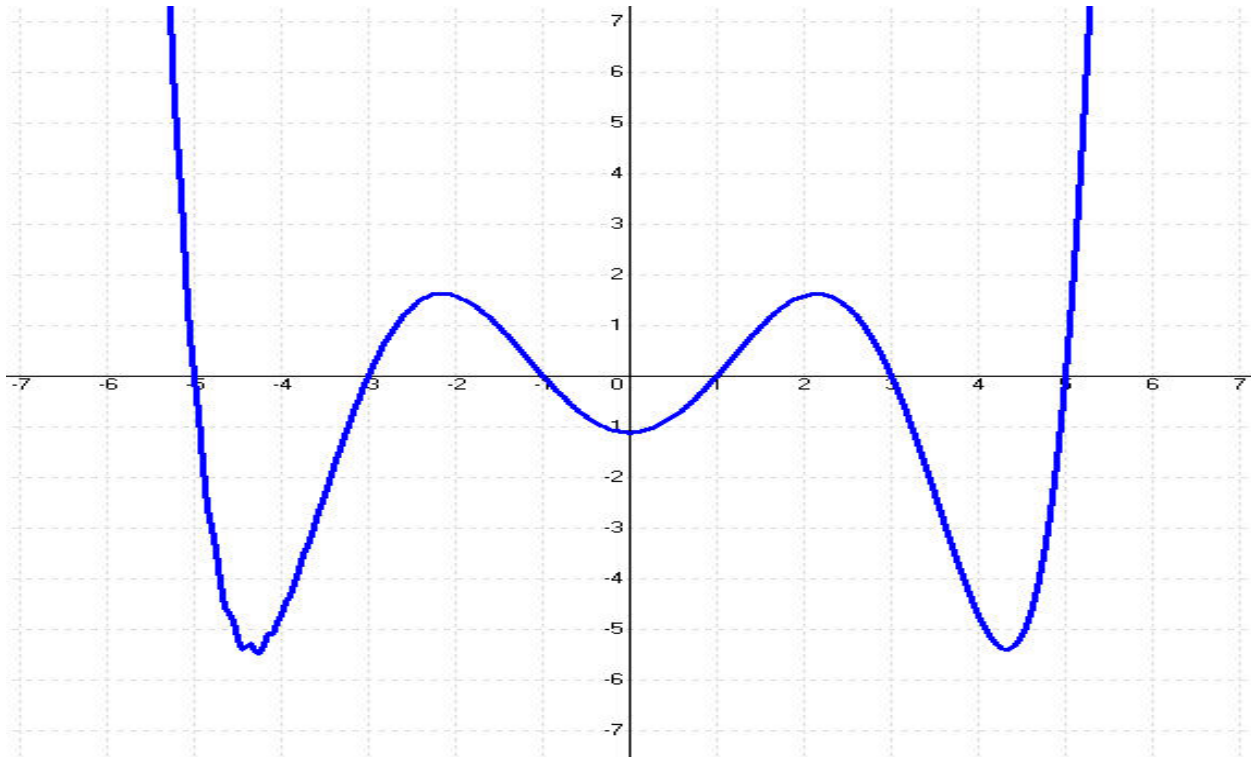


Table 3. Test results for $f(X) = 0.005 * (X+5) * (X+3) * (X+1) * (X-5) * (X-3) * (X-1)$ for $Xtoler = 1E-8$ and $FxToler = 1E-8$.

Initial Guess	PSA Iterations	PSA Fx Calls	PSA Result/Fx Value	Newton Iterations	Newton Fx Calls	Newton Result/Fx Value
7	7	12	5.0000000000000000 0	12	35	5.0000000000005101 9.7944052873E-10
6	5	10	5.000000000000042 8.20591594E-10	10	21	5.000000000031603 6.0677564328E-09
4	4	9	2.999999999928 2.73922751E-10	7	15	3.00000000140738 -5.404344282E-09
2	4	9	-0.999999999999 -3.82030407E-12	5	11	-0.9999999999541 -8.7987856999E-11
0	28	33	5.0000000000068 1.31813067E-09	28	57	5.00000000005315 1.0205553736E-09

Table 3 shows the results for the test polynomial for X tolerance of 1E-8 and function tolerance of 1E-8. The table shows a set of initial guesses supplied to the VBA subroutine RootByProbingSlopes. The tested function has roots at 5, 3, 1, -1, -3, and -5. The table shows that for each initial guess below the root value, the Probing Slopes Algorithm required less iterations and functions calls than Newton’s method. The value of X = 0 is an exception since both algorithms required the same number of iterations to reach the root of 5.

Testing the Probing Steps Algorithm

This section presents the tests results that compare the Probing Steps Algorithm with Newton's method. The test uses an Excel spreadsheet that has the following visual interface:

	A	B	C	D	E	F	G
1	X	Step	New X	Fx		New X	Fx
2	4	0.266741023	3.733258977	0.003493		3.792747	1.223341
3	XToler	0.266920588	3.733079412	7.44E-06		3.738517	0.106075
4	1.00E-08	0.266920971	3.733079029	5.02E-11		3.733336	0.004988
5	FxToler					3.73309	0.000213
6	1.00E-08					3.733079	9.09E-06
7		Num Fx Calls				3.733079	3.87E-07
8		8				3.733079	1.65E-08
9						3.733079	7.02E-10
10							
11							
12						Num Fx Calls	
13						17	
14							

The VBA code used for this test is:

```
Option Explicit
```

```
Option Base 0
```

```
Function F(X As Double) As Double
```

```
    F = Exp(X) - 3 * X ^ 2
```

```
End Function
```

```
Sub RootByProbingSteps()
```

```
    Const NUM_STEPS = 3
```

```
    Const SLOPE_SHIFT = 5
```

```
    Const STEP_FACTOR = 0.15
```

```
    Dim XToler As Double, FxToler As Double
```

```
    Dim R As Integer, NumCalls As Integer
```

```
    Dim X As Double, St(NUM_STEPS + 1) As Double, Fx(NUM_STEPS + 1) As Double
```

```
    Dim Xnew(NUM_STEPS + 1) As Double, Xold As Double
```

```
    Dim I As Integer, J As Integer
```

```
    Dim Sum As Double, Prod As Double, FxVal As Double, Diff As Double
```

```
    Dim h As Double, Buff As Double
```

```
    Dim bNoSwap As Boolean
```

```
    X = Cells(2, 1)
```

```
    XToler = Cells(4, 1)
```

```
    FxToler = Cells(6, 1)
```

```

Range("B2:Z10000").Value = ""

R = 2
FxVal = F(X)
NumCalls = 1

h = 0.01 * (1 + Abs(X))
St(1) = h * FxVal / (F(X + h) - FxVal)
NumCalls = NumCalls + 1
St(2) = St(1) * (1 + STEP_FACTOR)
St(3) = St(1) * (1 - STEP_FACTOR)
For I = 1 To 3
    Xnew(I) = X - St(I)
    Fx(I) = F(Xnew(I))
    NumCalls = NumCalls + 1
Next I

' remove worst fx() value
For I = 1 To NUM_STEPS - 1
    For J = I + 1 To NUM_STEPS
        If Abs(Fx(I)) > Abs(Fx(J)) Then

            Buff = Fx(I)
            Fx(I) = Fx(J)
            Fx(J) = Buff

            Buff = St(I)
            St(I) = St(J)
            St(J) = Buff

            Buff = Xnew(I)
            Xnew(I) = Xnew(J)
            Xnew(J) = Buff

        End If
    Next J
Next I

Do
    Sum = 0
    For I = 1 To NUM_STEPS
        Prod = St(I)
        For J = 1 To NUM_STEPS
            If I <> J Then
                Prod = Prod * (0 - Fx(J)) / (Fx(I) - Fx(J))
            End If
        Next J
        Sum = Sum + Prod
    Next I

```



```

St(4) = Sum
Xnew(4) = X - St(4)
Cells(R, 2) = St(4)
Cells(R, 3) = Xnew(4)
Fx(4) = F(Xnew(4))
NumCalls = NumCalls + 1
Cells(R, 4) = Fx(4)

' store last probe in index 0
St(0) = St(4)
Fx(0) = Fx(4)
Xnew(0) = Xnew(4)

' remove worst fx() value
bNoSwap = True
For I = 1 To NUM_STEPS - 1
  If Abs(Fx(0)) < Abs(Fx(I)) Then
    bNoSwap = False
    For J = NUM_STEPS To I Step -1
      Fx(J) = Fx(J - 1)
      St(J) = St(J - 1)
      Xnew(J) = Xnew(J - 1)
    Next J
    ' restore latest problem
    St(I) = St(0)
    Fx(I) = Fx(0)
    Xnew(I) = Xnew(0)
  End If
Exit For
Next I

R = R + 1

Loop Until bNoSwap Or Abs(Xnew(1) - Xnew(2)) <= XToler Or Abs(Fx(1))
<= FxToler

Cells(R + 2, 2) = "Num Fx Calls"
Cells(R + 3, 2) = NumCalls
' flag end of iteration because the last guess failed to improve
Cells(R + 4, 2) = IIf(bNoSwap, "Iter Halted", "")

' now test Newton's method
X = Cells(2, 1)
FxVal = F(X)
R = 2
NumCalls = 1

Do
  h = 0.01 * (1 + Abs(X))

```

```

    Diff = h * FxVal / (F(X + h) - FxVal)
    NumCalls = NumCalls + 1
    X = X - Diff
    FxVal = F(X)
    NumCalls = NumCalls + 1
    Cells(R, 6) = X
    Cells(R, 7) = FxVal
    R = R + 1
Loop Until Abs(Diff) <= XToler Or Abs(FxVal) < FxToler

Cells(R + 2, 6) = "Num Fx Calls"
Cells(R + 3, 6) = NumCalls

End Sub

Sub RootByMovingSteps()
    Const NUM_STEPS = 3
    Const SLOPE_SHIFT = 5
    Const STEP_FACTOR = 0.15

    Dim XToler As Double, FxToler As Double, CHS As Double
    Dim R As Integer, NumCalls As Integer
    Dim X As Double, St(NUM_STEPS + 1) As Double, Fx(NUM_STEPS + 1) As
Double
    Dim Xnew(NUM_STEPS + 1) As Double, Xold As Double
    Dim I As Integer, J As Integer
    Dim Sum As Double, Prod As Double, FxVal As Double, Diff As Double
    Dim h As Double, Buff As Double

    X = Cells(2, 1)
    XToler = Cells(4, 1)
    FxToler = Cells(6, 1)
    Range("B2:Z10000").Value = ""

    R = 2
    FxVal = F(X)
    NumCalls = 1

    h = 0.01 * (1 + Abs(X))
    St(1) = h * FxVal / (F(X + h) - FxVal)
    NumCalls = NumCalls + 1
    St(2) = St(1) * (1 + STEP_FACTOR)
    St(3) = St(1) * (1 - STEP_FACTOR)
    For I = 1 To 3
        Xnew(I) = X - St(I)
        Fx(I) = F(Xnew(I))
        NumCalls = NumCalls + 1
    Next I

Do

```

```

Sum = 0
For I = 1 To NUM_STEPS
  Prod = Xnew(I)
  For J = 1 To NUM_STEPS
    If I <> J Then
      Prod = Prod * (0 - Fx(J)) / (Fx(I) - Fx(J))
    End If
  Next J
  Sum = Sum + Prod
Next I

Xnew(4) = Sum
Cells(R, 3) = Xnew(4)
Fx(4) = F(Xnew(4))
NumCalls = NumCalls + 1
Cells(R, 4) = Fx(4)

' remove worst fx() value
For I = 1 To NUM_STEPS
  For J = I + 1 To NUM_STEPS + 1
    If Abs(Fx(I)) > Abs(Fx(J)) Then
      Buff = Fx(I)
      Fx(I) = Fx(J)
      Fx(J) = Buff

      Buff = Xnew(I)
      Xnew(I) = Xnew(J)
      Xnew(J) = Buff
    End If
  Next J
Next I

R = R + 1
If R > 100 Then Exit Do

Loop Until Abs(Xnew(1) - Xnew(2)) <= XToler Or Abs(Fx(1)) <= FxToler

Cells(R + 2, 3) = NumCalls

' now test Newton's method
X = Cells(2, 1)
FxVal = F(X)
R = 2
NumCalls = 1

Do
  h = 0.01 * (1 + Abs(X))
  Diff = h * FxVal / (F(X + h) - FxVal)

```

```

    NumCalls = NumCalls + 1
    X = X - Diff
    FxVal = F(X)
    NumCalls = NumCalls + 1
    Cells(R, 6) = X
    Cells(R, 7) = FxVal
    R = R + 1
    Loop Until Abs(Diff) <= XToler Or Abs(FxVal) < FxToler

    Cells(R + 2, 6) = NumCalls

```

End Sub

The above code shows the function F which contains a sample test f(x) function. The code for function F changes as the test function f(x) changes.

The Function $f(x)=\text{Exp}(x)-3*X^2$

The first test function is $f(X) = \text{Exp}(X)-3*X^2$. The following figure shows a graph for this function.

Table 4. Test results for $f(X)=\text{Exp}(X)-3*X^2$ for $Xtoler = 1E-8$ and $FxToler = 1E-8$.

Initial Guess	PSA Iterations	PSA Fx Calls	PSA Result/Fx Value	Newton Iterations	Newton Fx Calls	Newton Result/Fx Value
7	8	13	3.73307902 7.090203E-10	12	25	3.73307902 3.01841573E-09
6	7	12	3.73307902 2.623679E-12	11	23	3.73307902 1.79482562E-09
5	5	10	3.73307902 3.932557E-09	10	21	3.73307902 6.47405684E-10
4	3	8	3.73307902 5.015543E-11	8	17	3.73307902 7.01874114E-10
3	8	13	3.73307902 2.646771E-13	12	25	3.73307902 1.2888E-09
1	2	7	0.91000757 -5.476774E-11	5	11	0.91000757258 -2.8452E-10
0	5	10	-0.4589622 -1.033062E-13	7	15	-0.4589622 -1.9023E-10
-1	3	8	-0.4589622 -2.575483E-09	6	13	-0.4589622 -2.187558E-10
-2	5	10	-0.4589622 -7.580047E-14	7	15	-0.4589622 -2.041214E-10
-3	5	10	-0.4589622 -6.848205E-09	5	15	-0.4589622 7.5226654E-09

Table 4 shows the results for function $f(X) = e^X - 3X^2$ for X tolerance of 1E-8 and function tolerance of 1E-8. The table shows a set of initial guesses supplied to the VBA subroutine RootByProbingSteps. The tested function has three roots near 3.733, 0.9100, and -0.4589. The table shows that for each initial guess, the Probing Steps Algorithm required less iterations and functions calls than Newton’s method.

The Function $f(X) = \exp(-X) - \exp(-3)$

The second function is $f(X) = \exp(-X) - \exp(-3)$ whose graph appears in the next figure:

Table 5. Test results for $f(X)=\exp(-X)- \exp(-3)$ for $Xtoler =1E-8$ and $FxToler = 1E-8$.

Initial Guess	PSA Iterations	PSA Fx Calls	PSA Result/Fx Value	Newton Iterations	Newton Fx Calls	Newton Result/Fx Value
-2	9	14	2.999999999 1.3929135E-13	10	21	3.00000016057 -7.994606E-10
-1	7	12	2.9999990245 4.8563936E-09	9	19	3.0000001687 -8.402010E-10
0	6	11	2.9999999045 4.7515362E-10	8	17	3.0000001601 -7.973450E-10
1	5	10	2.9999999969 1.5068467E-11	7	15	3.0000001166 -5.805927E-10
2	4	9	2.999999999 1.3974932E-14	5	11	2.9999987482 6.2321192E-09
4	4	9	2.9999992977 6.3310566E-10	6	13	3.0000008045 -4.005808E-09
5	9	14	2.9999999999 9.9669231E-12	10	21	3.00000065888 -3.280370E-09
6	22	27	2.9999892082 3.3237995E-13	22	45	3.0000004812 -2.395785E-09

Table 5 shows the results for function $f(X) = e^{-X} - e^{-3}$ for X tolerance of 1E-8 and function tolerance of 1E-8. The table shows a set of initial guesses supplied to the VBA subroutine RootByProbingSteps. The tested function has a root of 3. The table shows that for each initial guess below the root value, the Probing Steps Algorithm required less iterations and functions calls than Newton’s method. The value of X = 6 is an exception since Newton’s method required the same number of iterations, but made 18 additional function calls.

The Test Polynomial

The third test function is:

$$f(X) = 0.005 * (X+5) * (X+3) * (X+1) * (X-5) * (X-3) * (X-1)$$

Table 6. Test results for $f(X) = 0.005 * (X+5) * (X+3) * (X+1) * (X-5) * (X-3) * (X-1)$ for $Xtoler = 1E-8$ and $FxToler = 1E-8$.

Initial Guess	PSA Iterations	PSA Fx Calls	PSA Result/Fx Value	Newton Iterations	Newton Fx Calls	Newton Result/Fx Value
7	7	12	5 8.526512829E-14	12	35	5.0000000005101 9.7944052873E-10
6	5	10	5.000000953793 2.595470505E-10	10	21	5.0000000031603 6.0677564328E-09
4	5	10	2.99999999928 3.410605131E-15	7	15	3.0000000140738 -5.404344282E-09
2	4	9	-1.0000000000067 1.290914042E-12	5	11	-0.999999999541 -8.7987856999E-11
0	25	30	5.0000000002342 4.49603021491E-10	28	57	5.0000000005315 1.0205553736E-09

Table 6 shows the results for the test polynomial for X tolerance of 1E-8 and function tolerance of 1E-8. The table shows a set of initial guesses supplied to the VBA subroutine RootByProbingSteps. The tested function has roots at 5, 3, 1, -1, -3, and -5. The table shows that for each initial guess below the root value, the Probing Slope Algorithm required less iterations and functions calls than Newton’s method.

Conclusions

The Probing Slopes Algorithm and the Probing Steps Algorithms demonstrated in the three examples that they outperform Newton’s method. The tests also show that the Probing Steps Algorithm slightly outperforms the Probing Slopes Algorithm. The recommendation is to use the Probing Steps Algorithm, with the Probing Slopes Algorithm as plan B.