

The New Ostrowski-Halley Root Seeking Algorithm

By

Namir Shammas

Introduction

Ostrowski was a Russian mathematician who taught for many years at the University of Basil, Switzerland. He proposed an enhancement to Newton's root seeking algorithm. Ostrowski suggested a new twist such that each iteration offers two refinements for the root—one of them being intermediate. The Ostrowski algorithm matches Halley's root seeking algorithm in its third order rate of convergence. Recently, the Ostrowski algorithm inspired many mathematicians to device root-seeking algorithms with two or more refinements to the root per iteration.

I recently applied Ostrowski's approach to the Illinois algorithm (an improved version of the False Position algorithm) and obtained better rates of convergence better than those of the Illinois algorithm. I was a little baffled as to why Ostrowski improved only the Newton's method and did not become more ambitious to enhance Halley's superior method! Moreover, there has been many articles on further improving Ostrowski's work, but none to improve Halley's method using the Ostrowski approach.

I decided to experiment with applying Ostrowski's approach to Halley's algorithm. Since the latter method is a bit more advanced than Newton's method (requiring the calculations of the first AND second derivatives), applying the Ostrowski approach was NOT trivial. I decided, nevertheless, to give it a go. I started with a simple improvement to Halley's method, but that did not yield better calculations. After two or three incarnations, I was able to find a satisfactory marriage between Ostrowski and Halley. This paper reports the algorithm details and also includes a comparison between the methods of Newton, Halley, Ostrowski, and my new Ostrowski-Halley algorithm. The results include testing these algorithms with two dozen functions and reporting the number of function calls AND iterations.

With the advance of computers in general and increasingly fast computers in particular, it has become more feasible to work with new non-legacy root seeking

algorithms like Ostrowski and the new method that I propose. These algorithms require a bit more computation effort (involving the basic mathematical operations) that does not impose significant burden on today's fast CPUs. This ease of automated calculation is far cry from the days of pre-20th centuries' manual calculations performed by humans working for the famous mathematicians. Manual calculations favored algorithms that required simpler calculations per iteration. Even when electronics firms, like HP and TI, launched programmable calculators in the seventies, the legacy algorithms were a perfect fit for these calculators with their limited memory and CPU speed. The advent of several generations of PCs, with their ever-increasing CPU speed and memory, made new root-seeking algorithms possible and feasible. For example, I recently developed smart enhancements to the Bisection method by adding more decision-making components. These variants of Bisection succeeded in improving on the number of iterations. The extra calculations and decision-making per iteration was, by no means, punishing the computer's CPU.

While the number of iterations in root-seeking is important, we must not totally disregard the number of function calls. They are the proverbial cost of doing business. This is very true when the target function required significant calculations/iterations, such as series or product involving many terms. Many of the new advanced root-seeking algorithms succeed in quickly converging to the desired answer. This achievement comes at the cost of making a relatively large number of function calls. There is also the issue of additional basic mathematical operations and decision-making needed by the more sophisticated root-seeking algorithms. Most mathematicians seem to ignore this kind of additional overhead, pointing out to the fact that we are using CPUs that are quite fast. The gain in CPU speed, they argue, can certainly more than make up for the additional CPU effort needed to handle the additional overhead operations. I would not be surprised if a few mathematicians consider the new CPU speeds as making the number of function calls less relevant.

Legacy Root-Seeking Algorithms

In this section we briefly discuss the root seeking methods of Newton, Halley, and Ostrowski.

The Newton Method

One of the most popular root-seeking algorithms is the Newton method (also called the Newton-Raphson method). While Isaac Newton had little to do with the

algorithm in its current form, it was Thomas Simpson (known for Simpson's rule for numerical integration) who gave it its name and homage to Isaac Newton.

The equation for Newton's method that refined a guess for the root is:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (1)$$

Equation 1 required evaluating the function $f(x)$ and its derivative $f'(x)$ which can be approximated using the forward difference approximation:

$$f'(x) = (f(x + h) - f(x))/h \quad (2)$$

Where $h = 0.02(1 + |x|)$. Newton's method usually converges at a second order rate.

The Halley Method

Halley devised a method for calculating roots that has a third order convergence rate. The root-refining equation for this algorithm is:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \left[1 - \frac{f(x_i)}{f'(x_i)} \frac{f''(x_i)}{2f'(x_i)} \right]^{-1} \quad (3)$$

Or,

$$x_{i+1} = x_i - \frac{2 f(x_i) f'(x_i)}{2[f'(x_i)]^2 - f(x_i) f''(x_i)} \quad (3b)$$

The first and second derivatives are calculated using the following central difference approximations:

$$f'(x) = (f(x + h) - f(x - h))/2h \quad (4)$$

$$f''(x) = (f(x + h) - 2f(x) + f(x - h))/h^2 \quad (5)$$

The Ostrowski Method

While relatively newer than the previous algorithms, I am including the Ostrowski method in this section, since it is a few decades old. The Ostrowski method generates two refinements for the root in each iteration, the first refinement is an intermediate one. The method uses the following two equations:

$$y_i = x_i - \frac{f(x_i)}{f'(x_i)} \quad (6)$$

$$x_{i+1} = y_i - \frac{f(y_i)(x_i - y_i)}{f(x_i) - 2f(y_i)} \quad (7)$$

The Ostrowski method has a convergence rate resembling that of Halley's method. Both the Halley and Ostrowski methods require three function calls per iteration. This number is compared to two function calls (when using the forward or backward difference approximation to the first derivative) for the Newton method. If you use the central difference approximation for the first derivative (which is a bit more accurate than the forward or backward difference) then each iteration in Newton's method makes three function calls. In this case, you already have the basic information that makes it easy to calculate the second derivative and graduate to using Halley's method with a small extra computational effort.

The New Ostrowski-Halley Algorithm

Let me present the pseudo-code for the new Ostrowski-Halley method. Given the function $f(x)=0$, an initial guess, x , and a tolerance $Toler$ for the guess:

```

Do
  h = 0.01 * (1 + |x|)
  F0 = f(x)
  Fp = f(x + h)
  Fm = f(x - h)
  Deriv1 = (Fp - Fm) / 2 / h
  Deriv2 = (Fp - 2 * F0 + Fm) / h / h
  Diff = F0 / Deriv1 / (1 - F0 * Deriv2 / Deriv1 / 2 / Deriv1)
  z = x - Diff
  Fz = f(z)
  If |x - z| < h Then h = x - z
  Deriv1b = (F0 - 2 * Fz) / (x - z)
  Deriv2b = (Fp - 2 * Fz + Fm) / h / h
  Diff2 = Fz / Deriv1b / (1 - Fz * Deriv2b /
    Deriv1b / 2 / Deriv1b)
  x = z - Diff2
Loop Until |Diff2| < Toler
Return X as the refined guess for the root.

```

The above pseudo-code shows how the method calculates an intermediate refinement for the root, z , then recalculates new versions of the first and second derivatives using z and $f(z)$, and then obtains a second value for the refined root that is stored in x . The expressions that assign values to variables $Deriv1b$, $Deriv2b$, and $Diff2$ are the heart of the Ostrowski modification to Halley's method.

Excel VBA Code

I present Excel VBA code that calculates roots using the methods of Newton, Halley, Ostrowski, and the new Ostrowski-Halley algorithm. Figure 1 shows a sample worksheet. You can download the Excel file that contains all the VBA code and the worksheets for the various tested functions.

	A	B	C	D	E	F	G	H	I	J	K	L
1	X	Newton		Halley		Ostrowski		Ostrowski-Halley				
2	1	0.915001	-0.0149	0.9103	-0.000870597	0.90997263	0.000104	0.9103	0.910002	1.53E-05		
3	Toler	0.910077	-0.00021	0.910008	4.41757E-08	0.910007577	-1.3E-08	0.910008	0.910008	-3.6E-14		
4	1.00E-09	0.910008	-2.3E-06	0.910008	-2.24043E-12	0.910007572	1.65E-12					
5	Fx	0.910008	-2.6E-08	0.910008	0	0.910007572	0		Fx Calls=	8		
6	$\exp(x)-3*x^2$	0.910008	-2.8E-10									
7		0.910008	-3.2E-12	Fx Calls=	12	Fx Calls=	12					
8												
9		Fx Calls=	12									
10												
11												
12												
13												

Figure 1. Sample Worksheet.

Note the following cells and columns in Figure 1:

- Cell A2 has the initial guess for the root.
- Cell A4 has the tolerance value.
- Cell A6 has the expression for $f(x)$.
- Columns B and C show the output for the refined root values and their function values for Newton's method. The bottommost items in these two columns display the number of function calls for Newton's method.
- Columns D and E show the output for the refined root values and their function values for Halley's method. The bottommost items in these two the number of function calls for Halley's method.
- Columns F and G show the output for the refined root values and their function values for Ostrowski's method. The bottommost items in these two columns display the number of function calls for Ostrowski's method.
- Columns H, I, and J show the output for intermediate refined root values, the refined root values, and their function values for the new Ostrowski-Halley method. The bottommost items in these two columns display the number of function calls for the Ostrowski-Halley method.

Here is the VBA code listing for version A:

Option Explicit

```
Function Fx(ByVal sFx As String, ByVal X As Double) As Double
    sFx = Replace(sFx, "EXP(", "!")
    sFx = Replace(sFx, "X", "(" & X & ")")
    sFx = Replace(sFx, "!", "EXP(")
    Fx = Evaluate(sFx)
End Function
```

```
Sub Go()
    Dim R As Long, C As Double
    Dim X As Double, h As Double, Diff As Double
    Dim F0 As Double, Deriv1 As Double, Deriv1b As Double
    Dim Deriv2 As Double, Deriv2b As Double
    Dim Fp As Double, Fm As Double
    Dim Z As Double, Fz As Double, LastX As Double
    Dim Toler As Double
    Dim sFx As String
```

```
X = [A2].Value
Toler = [A4].Value
sFx = [A6].Value
sFx = UCase(Replace(sFx, " ", ""))
```

```
Range("B2:z1000").Clear
```

```
' Newton's method
R = 2
C = 2
Do
    h = 0.01 * (1 + Abs(X))
    F0 = Fx(sFx, X)
    Diff = h * F0 / (Fx(sFx, X + h) - F0)
    X = X - Diff
    Cells(R, C) = X
    Cells(R, C + 1) = Fx(sFx, X)
    R = R + 1
Loop Until Abs(Diff) < Toler Or R > 1000
Cells(R + 1, C) = "Fx Calls="
Cells(R + 1, C + 1) = 2 * (R - 2)
```

```
' Halley
R = 2
C = C + 2
X = [A2].Value
Do
    h = 0.01 * (1 + Abs(X))
```

```

F0 = Fx(sFx, X)
Fp = Fx(sFx, X + h)
Fm = Fx(sFx, X - h)
Deriv1 = (Fp - Fm) / 2 / h
Deriv2 = (Fp - 2 * F0 + Fm) / h / h
Diff = F0 / Deriv1 / (1 - F0 * Deriv2 / Deriv1 / 2 / Deriv1)
X = X - Diff
Cells(R, C) = X
Cells(R, C + 1) = Fx(sFx, X)
R = R + 1
Loop Until Abs(Diff) < Toler
Cells(R + 1, C) = "Fx Calls="
Cells(R + 1, C + 1) = 3 * (R - 2)

```

```

' Ostrowski
R = 2
C = C + 2
X = [A2].Value
Do
  LastX = X
  h = 0.01 * (1 + Abs(X))
  F0 = Fx(sFx, X)
  Fp = Fx(sFx, X + h)
  Deriv1 = (Fp - F0) / h
  Z = X - F0 / Deriv1
  Fz = Fx(sFx, Z)
  X = Z - Fz * (X - Z) / (F0 - 2 * Fz)
  Cells(R, C) = X
  Cells(R, C + 1) = Fx(sFx, X)
  R = R + 1
Loop Until Abs(X - LastX) < Toler Or R > 1000
Cells(R + 1, C) = "Fx Calls="
Cells(R + 1, C + 1) = 3 * (R - 2)

```

```

' Ostrowski-Halley
R = 2
C = C + 2
X = [A2].Value
Do
  h = 0.01 * (1 + Abs(X))
  F0 = Fx(sFx, X)
  Fp = Fx(sFx, X + h)
  Fm = Fx(sFx, X - h)
  Deriv1 = (Fp - Fm) / 2 / h
  Deriv2 = (Fp - 2 * F0 + Fm) / h / h
  Diff = F0 / Deriv1 / (1 - F0 * Deriv2 / Deriv1 / 2 / Deriv1)

```

```

Z = X - Diff
Fz = Fx(sFx, Z)
If Abs(X - Z) < h Then h = X - Z
Deriv1b = (F0 - 2 * Fz) / (X - Z)
Deriv2b = (Fp - 2 * Fz + Fm) / h / h
Diff = Fz / Deriv1b / (1 - Fz * Deriv2b / _
      Deriv1b / 2 / Deriv1b)
X = Z - Diff
Cells(R, C) = Z
Cells(R, C + 1) = X
Cells(R, C + 2) = Fx(sFx, X)
R = R + 1
Loop Until Abs(Diff) < Toler Or R > 1000
Cells(R + 1, C + 1) = "Fx Calls="
Cells(R + 1, C + 2) = 4 * (R - 2)
End Sub

```

Testing and Comparing the Algorithms

Table 1 shows the list of test functions. The first two functions are ones that I have chosen. The remaining functions come from the Table II in the article by Galdino, Sérgio (2011). "A family of regula falsi root-finding methods". Proceedings of 2011 World Congress on Engineering and Technology. 1. Retrieved 9 September 2016. I am using the same function numbers in Table 1 as in Table II in the article by Sérgio. I skipped functions 16 and 17 in Table II. I would like to point out that Table II, in the article by Sérgio, erroneously replicates function number 16 and 17 as function number 19 and 20, respectively. Table 1 shows the corrected form of function number 19 and 20, which are variants of function number 18.

Function Number	F(x)=
Custom 1	$\sin(x-1)/(x-1)-1$
Custom 2	$\exp(x)-3*x^2$
2	$x^2*(x^2/3+\sqrt{2}*\sin(x))-\sqrt{3/18}$
3	$11*x^{11}-1$
4	x^3+1
5	$x^3-3*x-5$
6	$2*x*\exp(-5)+1-2*\exp(-5*x)$
7	$2*x*\exp(-10)+1-2*\exp(-10*x)$
8	$2*x*\exp(-20)+1-2*\exp(-20*x)$
9	$(1+(1-5)^2)*x^2-(1-5*x)^2$
10	$(1+(1-10)^2)*x^2-(1-10*x)^2$
11	$(1+(1-20)^2)*x^2-(1-20*x)^2$

Function Number	F(x)=
12	$x^2-(1-x)^5$
13	$x^2-(1-x)^{10}$
14	$x^2-(1-x)^{20}$
15	$(1+(1-5)^4)*x-(1-5*x)^4$
18	$\exp(-5*x)*(x-1)+x^5$
19	$\exp(-10*x)*(x-1)+x^{10}$
20	$\exp(-20*x)*(x-1)+x^{20}$
21	$x^2+\sin(x/5)-0.25$
22	$x^2+\sin(x/10)-0.25$
23	$x^2+\sin(x/20)-0.25$

Table 1. List of test functions.

Table 2 shows the results that compare the efficiency of the various algorithms. The comma-delimited results report the number of iterations and the number of function calls. Remember that the Newton, Halley, Ostrowski, and the new algorithm use 2, 3, 3, and 4 function calls, per iterations, respectively. The table shows one, two, and three different guesses for various test functions. The tolerance value for all the calculations is 1E-9.

<i>Function Number</i>	<i>Initial Guess</i>	<i>Newton</i>	<i>Halley</i>	<i>Ostrowski</i>	<i>Ostrowski-Halley</i>
Custom 1	0	Failed	16, 48	Failed	10, 40
Custom 2	3	13, 26	6, 18	6, 18	5, 20
Custom 2	5	10, 20	6, 18	6, 18	4, 16
Custom 2	-1	7, 14	4, 12	4, 12	3, 12
Custom 2	1	6, 12	4, 12	4, 12	2, 8
2	1	8, 16	5, 15	5, 15	3, 12
3	1	12, 24	7, 21	7, 21	4, 16
4	-1.8	8, 16	5, 15	5, 15	4, 16
5	3	8, 16	5, 15	5, 15	3, 12
6	0	8, 16	5, 15	5, 15	3, 12
6	1	59, 118	6, 18	18, 54	6, 24
7	0	8, 16	5, 15	5, 15	4, 16
8	0	8, 16	5, 15	5, 15	3, 12
9	0	6, 12	3, 9	4, 12	3, 12
9	1	7, 14	4, 12	4, 12	3, 12
10	0	6, 12	3, 9	4, 12	2, 8
10	1	6, 12	3, 9	4, 12	2, 8

Function Number	Initial Guess	Newton	Halley	Ostrowski	Ostrowski-Halley
11	0	5, 10	3, 9	3, 9	2, 8
11	1	6, 12	3, 9	4, 12	2, 8
12	0	8, 16	5, 15	4, 12	3, 12
12	1	6, 12	5, 15	5, 15	3, 12
13	0	9, 18	5, 15	5, 15	4, 16
13	1	9, 18	6, 18	5, 15	4, 16
14	0	11, 22	7, 21	6, 18	5, 20
14	1	11, 22	7, 21	6, 18	5, 20
15	0	4, 8	3, 9	3, 9	2, 8
15	1	6, 12	3, 9	4, 12	2, 8
18	0	9, 18	6, 18	5, 15	5, 20
18	1	9, 18	5, 15	5, 15	4, 16
19	0	12, 24	8, 24	7, 21	6, 24
19	1	14, 28	7, 21	7, 21	6, 24
20	0	19, 38	12, 36	11, 33	9, 36
20	1	24, 48	13, 39	8, 24	10, 40
21	0	9, 18	5, 15	5, 15	4, 16
21	1	8, 16	4, 12	5, 15	3, 12
22	0	10, 20	6, 18	6, 18	5, 20
22	1	8, 16	4, 12	5, 15	3, 12
23	0	11, 22	6, 18	6, 18	6, 24
23	1	8, 16	4, 12	5, 15	3, 12

Table 2. Test functions for different algorithms showing the initial guesses, and the number of iterations and total function calls for each algorithm.

Looking at Table 2, you can see that the new Ostrowski-Halley algorithm does very well. I use red fonts to indicate the minimum number of function calls and the minimum number of iterations. The competition between Halley's method, Ostrowski's method and the new algorithm is stiff. The Ostrowski method seems to be in second place, followed by Halley's method, and ending with Newton's method. The new algorithm does very well in having less function calls and/or less iterations.

Note

If you download the ZIP file containing the Excel that contains the test functions, you can use either Excel file in the following ways:

1. Single-sheet mode. Select (or even create a copy of) a worksheet for a function you want to test. Optionally update all or some of the input parameters in cells A2, A4, and/or A6. Execute the macro **Go()** to perform the calculations on the tested root-seeking algorithms.
2. Multiple-sheets mode. You can optionally update all or some the input parameters in cells A2, A4, and/or A6, in all or some of the worksheets. To recalculate the roots in ALL of the worksheets execute macro **doAll()**. This macro will display a prompt message asking you to verify if you wish to recalculate the roots in ALL of the worksheets. Click the **Yes** button to proceed or click the **No** button to exit. The macro will quickly visit each worksheet containing the word “Roots” in its tab name and perform the calculations. If there are no runtime errors, this macro will perform its task very quickly.