Ostrowski's Method for Finding Roots

Namir Shammas

Introduction

Mathematicians, statisticians, engineers, and scientists frequently deal with problems that require the calculation of one or more roots of functions. When HP launched its first programmable calculator, the HP-65 in 1974, the accompanying Standard Pac included a root-seeking program. The HP-65 Math Pac 1 also included another root-finding program. Over the following years, HP released new programmable calculators; each included root-seeking programs in their standard and math pacs. In 1978, HP launched the HP-34C that contained the very first built-in root-finding Solver. The Solver was a new, powerful, and convenient tool for calculating the roots of single-variable nonlinear equations. The Solver implemented a clever version of the Secant method. This method along with Newton's method remained the two favorite root-seeking algorithms for many decades among users of programmable HP calculators. In this article I present the Ostrowski root-seeking method which I consider a gem of an algorithm that has not received much publicity until somewhat recently. I also present listings for Ostrowski's method to run on the WP34S calculator (using a repurposed HP 30b) and on the new HP 39gII. Before I discuss the algorithm and present the two listings, I would like to first shed some light on the person of Ostrowski.

Ostrowski: A Short Biography

Alexander Markowich Ostrowski (1893 to 1986) was a talented Russian mathematician who was gifted with a phenomenal memory. He was studying math at Marburg University in Germany, when World War I broke out. He was therefore interned as a hostile foreigner. During this period, he was able to obtain limited access to the university library. After the war he resumed his studies at the more prestigious University of Göttingen and was awarded a doctorate in mathematics in 1920. He graduated *summa cum laude* and worked in different universities. He eventually moved to Switzerland to teach at the University of Basel, before the outbreak of World War II. Ostrowski fared well living in that neutral country during the war. He taught in Basel until he retired in 1958. He remained very active in math until late in his life.

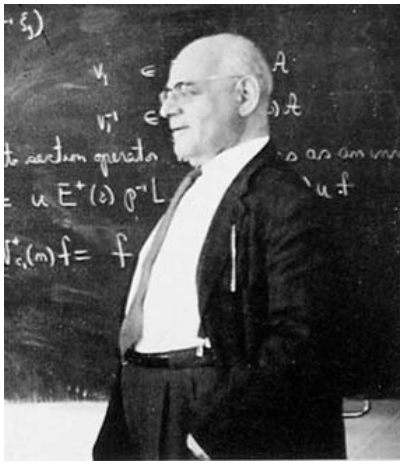

Fig. 1 – Alexander Ostrowski.

This gifted and prolific mathematician was engaged in various mathematical problems. The advent of computers catapulted Ostrowski to delve into numerical analysis. He studied iterative solutions of large linear systems. Ostrowski passed away in 1986 in Montagnola, Lugano, Switzerland. He had lived there with his wife during his retirement.

The Algorithm

Ostrowski tackled the problem of calculating a root for a single-variable non-linear function in a new way. Most of the methods we know perform a single refinement, for the guess of the root, in each iteration. Ostrowski's novel approach was to obtain an *interim refinement* for the root and then further enhance it by the end of each iteration. The two-step iterations in Ostrowski's method use the following two equations:

$$y_n = x_n - f(x_n) / f'(x_n) \quad \textbf{(1)}$$

$$x_{n+1} = y_n - f(y_n)(x_n-y_n) / (f(x_n) - 2f(y_n)) \quad \textbf{(2)}$$

Equation (1) applies the basic Newton algorithm as the first step to calculate y_n which Ostrowski uses as an interim refinement for the root. The iteration's additional refinement for the root comes by applying equation (2). Ostrowski's algorithm has a fourth-order convergence, compared to Newton's algorithm which has a second-order convergence. Thus Ostrowski's method converges to a root faster than Newton's method. There is the extra cost of evaluating an additional function call to calculate $f(y_n)$ in Ostrowski's method. This extra cost is well worth it, since, in general, the total number of function calls for Ostrowski's method is less than that for Newton's method.

Ostrowski's method has recently inspired quite a number of new algorithms that speed up convergence. These algorithms calculate two and even three interim root refinements in each iteration. While these methods succeed in reducing the number of iterations, they fail to consistently reduce the total number of function calls, compared to the basic Ostrowski's method. For example, Grau and Diaz-Barrero proposed the following equations for their algorithm:

$$y_n = x_n - f(x_n) / f'(x_n) \quad \textbf{(3)}$$

$$\mu = (x_n-y_n) / (f(x_n) - 2f(y_n)) \quad \textbf{(4)}$$

$$z_n = y_n - \mu f(y_n) \quad \textbf{(5)}$$

$$x_{n+1} = z_n - \mu f(z_n) \quad \textbf{(6)}$$

The algorithm starts with a Newton step using equation (3) to calculate y_n as the first interim refinement for the root. Equations (4) and (5) yield a second interim refinement, z_n. Equation (6) provides the iteration's final refinement for the root. The above equations show that each iteration requires two additional function calls needed to calculate $f(y_n)$ and $f(z_n)$. I compared the total number of function calls for Ostrowski's method with the Grau and Diaz-Barrero method for different test functions. The result is a mixed one, as neither method consistently performed better.

In the reference section you will find a reference to one of many books written by Ostrowski. The second cited reference is a paper by Walter Gautschi about the life, works, and students of Ostrowski. Gautschi himself is a mathematician and one of the last students of Ostrowski. Gautschi has published books in the field of numerical analysis. The third cited reference points to a recently published book written by him. I also included references to several articles, I found on the Internet, which present variants for the Ostrowski's method. I encourage you to search for these articles and learn more about these new algorithms.

The HP 39gII Listing

Table 1 contains the commented listing for the HP 39GII calculator. The table shows the following functions:

- The function **MYFX** which implements the code for calculating the mathematical function f(x).
- The function **OST** which performs the calculations for the Ostrowski method. This function makes several calls to function **MYFX**.
- The function **GO** which uses forms and message boxes to interact with the user. This function also calls function **OST** to obtain the number of iterations and the root value.

Table 1 – The HP 39gII Listing

Statement	*Comment*
`EXPORT MYFX(X)`	Define function for f(x)=0.
`BEGIN`	
`X-10*LN(1+4*X2+2*X^4);`	
`END;`	
`EXPORT OST(X,T)`	Define the function for the Ostrowski method. The parameter X is the initial guess for the root. The parameter T is the tolerance value.
`BEGIN`	
`LOCAL DIFF,FX,X0,FY;`	Declare multi-character local variables.
`0→I;`	Initialize the iteration counter.
`REPEAT`	Start the main iteration.
`I+1→I;`	Increment the iteration counter.
`X→X0;`	Assign a copy of X to X0.
`MYFX(X)→FX;`	Calculate and store f(X).
`0.001*(ABS(X)+1)→H;`	Calculate and store the increment h used to numerical evaluate the derivative.
`H*FX/(MYFX(X+H)-FX)→DIFF;`	Calculate and store the refinement for the guess.
`X-DIFF→Y;`	Calculate and store the value for y.
`MYFX(Y)→FY;`	Calculate and store the value for f(y).
`Y-FY*(X-Y)/(FX-2*FY)→X;`	Calculate and store the updated guess for the root.
`UNTIL ABS(X-X0) <= T;`	Did the solution converge?
`{X,I};`	Create a list containing the refined guess for the root and the iteration counter. This list is the result of the function OST.
`END;`	
`EXPORT GO()`	Function that drives the root-seeking program
`BEGIN`	
`LOCAL LST;`	Declare a local variable.
`INPUT(T,"Tolerance","Tolerance = ","Enter tolerance for root",1E-8);`	Prompt user with a form to enter the tolerance for the root. Store the tolerance value in variable T.
`INPUT(X,"Initial Guess","Guess = ","Enter initial guess for the root", 50);`	Prompt user with a form to enter the initial guess for the root. Store the initial guess in variable X.
`OST(X,T)→LST;`	Call function OST with the arguments X and T. Store the resulting list in list variable LST.
`MSGBOX("Number of iterations = " + LST(2));`	Display the number of iterations in a message box. Uses expression LST(2) to obtain the number of iterations.
`MSGBOX("Root = " + LST(1));`	Display the root in a message box. Uses expression LST(1) to obtain the value for the refined root.
`LST(1);`	Return the root.
`END;`	

The code for function **MYFX** is simple and requires a single statement for the tested function. The function **MYFX** has the parameter X to pass the value needed to calculate the mathematical function f(x).

The code for the function **OST** has the parameters X and T which pass the values for the initial guess for the root and the tolerance, respectively. Notice the following aspects about this function:

- The **LOCAL** statement that declares a number of local variables that have multi-character names.
- The **REPEAT-UNTIL** loop that implements the iterations needed to refine the guess for the root. The **UNTIL** clause tests the convergence criterion. The test determines if the absolute difference between the old guess for the root and the new one falls at or below the tolerance value.
- The function's return value which is the list **{X,I}**. This list contains the refined root value, X, and the number of iterations, I.

The code for the parameter-less function **GO** has the following three parts:

- Two **INPUT** statements that display input forms to obtain the values for the tolerance and the initial guess for the root.
- A call to function **OST**. The function **GO** stores the result of **OST** in the local list variable **LST**.
- Two **MSGBOX** statements to display message boxes showing the number of iterations and the refined guess for the root. The function uses the expression **LST(1)** and **LST(2)** to access the root value and number of iterations, respectively.

The function **GO** returns the refined root value. This approach allows you to store the root for additional inspection. You can invoke functions **MYFX**, **OST**, and **GO** from the command input line.

After you key in the three functions in Table 1, you can run the program by performing the following tasks:

1. Type **GO** at the command input line. The function **GO** first displays the **Tolerance** input form, shown in Figure 2. You can enter a new value for the tolerance and then press the **OK** menu option (which is associated with the F6 menu key), or simply accept the current tolerance value and press the **OK** menu option.

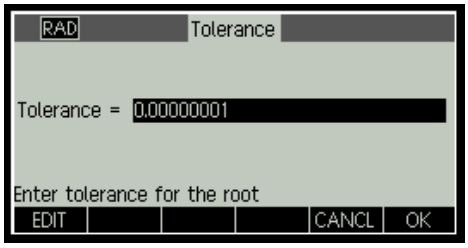

Fig. 2 – The prompt for the tolerance value.

2. The function displays the **Initial Guess** input form, shown in Figure 3. If the current guess is not 50, type in 50 and then press the **OK** menu option. Otherwise, just press the **OK** menu option.

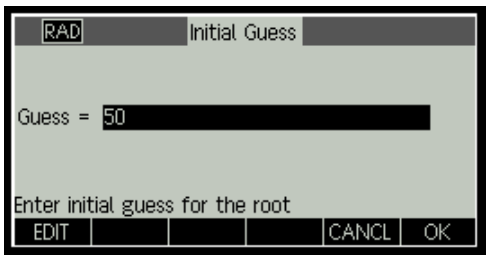

Fig. 3 – The prompt for the initial guess for the root.

3. The function displays the number of iterations, using the message box shown in Figure 4. Click the **OK** menu option to resume program execution,

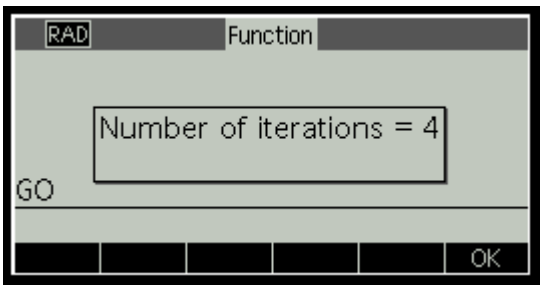

Fig. 4 – The number of iterations.

4. The function displays the value for the root, using the message box shown in Figure 5. Press the **OK** menu option to end the program.

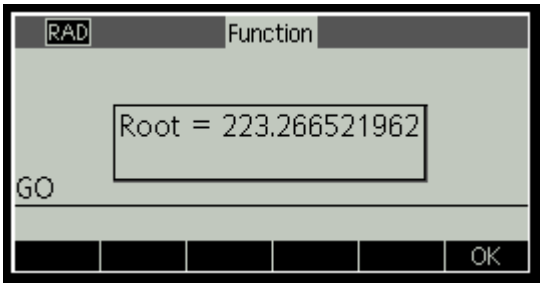

Fig. 5 – The refined root value.

I encourage you to edit function **MYFX** in Table 1 and replace the currently coded function with a different one. Table 2 shows a list of additional test functions. The table also includes columns that show the recommended initial root and the final root you get. Using the tolerance value of 1E-8 should be adequate for the test functions in Table 2.

Table 2 – A List of Additional Test Functions

Mathematical Function	*Initial Guess for the Root*	*Root*
$f(x) = x^3 + 4x^2 - 15$	3.0	1.6319808055661
$f(x) = x \exp(x^2) - \sin^2(x) + 3\cos(x) + 5$	-2.0	-1.2076478271309
$f(x) = \sin(x) - x/2$	1.1	1.8954942670340
$f(x) = 10 x \exp(-x^2) - 1$	1.0	1.6796306104284
$f(x) = \cos(x) - x$	10	0.73908513321516
$f(x) = \sin^2(x) - x^2 + 1$	0.1	1.4044916482153
$f(x) = \exp(-x) + \cos(x)$	0.1	1.7461395304080

The WP34S Listing

Let me present the listing, in Table 3, for my implementation of the Ostrowski method on a WP34S calculator. The program labels appear in red characters so they are easier to locate. The emulator I used for this article is running version 3 of the calculator software.

Table 3 – The WP34S Listing

Program Step	*Comment*	*Program Step*	*Comment*
LBL'OSTR'	Start and initialize the program.	x	
CLREG	Clear all the registers.	STO 02	Calculate small increment, h, used to numerically evaluate the slope.
EEX		RCL 00	
+/-		XEQ D	Calculate f(x).
8		STO 03	Store f(x).
STO 05	Store 1E-8 in register R05 as the initial tolerance value.	RCL 00	
RTN		RCL 02	
LBL D	Label used to code f(x)=0.	+	
LocR 003	Declare three local registers-R.00, R.01, and R.02.	XEQ D	Calculate f(x+h).
STO .00	R.00 = x	RCL 03	
x^2		-	
STO .01	R.01 = x^2	RCL 02	
x^2		/	
STO .02	R.02 = x^4	1/x	
RCL .00	Start evaluating f(x).	RCL 03	
1		x	Calculate the slope at x as (f(x+h)-f(x))/h.
RCL .01		STO- 01	Calculate y = x - f(x)/f ′(x).
4		RCL 01	
*		XEQ D	Calculate f(y).
+		STO 04	
RCL .02		RCL 00	
2		RCL 01	
*		-	
+		x=0?	
LN		GTO 01	
1		x	
0		RCL 03	
x		RCL 04	
-		STO+ X	
PopLR	Remove local registers.	-	
RTN		/	
LBL A	Label used to rerun the program.	RCL 01	
CLα	Clear the alpha register to start building new text.	x⇆Y	
α'TOL'		-	Calculate y-f(y)(x - y)/(f(x) - 2 f(y)).
α'ER?'		RCL 00	Recall older x.
RCL 05	Push current tolerance value in the stack.	x⇆Y	
PROMPT	Prompt for the tolerance value.	STO 00	Store new x.
STO 05		-	

Program Step	*Comment*	*Program Step*	*Comment*
LBL B	Label used to simply enter a new guess for the root.	ABS	
CLα	Clear the alpha register to start building new text.	RCL 05	
α'GUE'		x⇆Y	
α'SS?'		x>Y?	|new x – old x| > tolerance?
PROMPT	Prompt user to enter the guess for the root.	GTO 00	Resume iterations.
STO 00		LBL 01	
0		CLα	Clear the alpha register to start building new text.
STO 06	Initialize the loop counter.	α'ITE'	
LBL 00	Start of the main loop.	α'R='	
INC 06	Increment iteration counter.	RCL 06	
RCL 00		PROMPT	Display the number of iterations.
PSE 10	Display intermediate guess for the root.	CLα	Clear the alpha register to start building new text.
STO 01	Initialize y = x.	α'ROO'	
ABS		α'T='	
1		RCL 00	
+		PROMPT	Display the refined guess for the root.
EEX		GTO B	
3		END	
+/-			

If you are familiar with programming the HP-41C (and to a lesser extent other RPN programmable calculators like the HP-11C, HP-15C, and HP 35s) the above listing should be somewhat familiar to you. You will still find in Table 1 a number of new and different programming commands and features that are special to the WP34S. These new and different features are:

1. The program supports alphanumeric labels that are up to three characters long. The first command is **LBL 'OST'** which complies with this feature. This feature differs from the six character limit of alphanumeric labels in the HP-41C.
2. The WP34S calculator has four user defined keys labeled **A**, **B**, **C**, and **D**. They help you to easily execute code that you place after **LBL A**, **LBL B**, **LBL C**, and **LBL D**, respectively. The program in Table 1 uses the labels for keys **A**, **B** and **D**. By comparison, the HP-41C has more than four user defined keys.
3. The program uses **LBL D** to code the target mathematical function f(x). You can evaluate the function at any value of your choice by entering that value and pressing the key **D**. The code for calculating f(x) uses the local variables feature. The command **LocR *n*** dynamically allocates n local registers. The command also makes available 16 local flags, regardless of the value of n. The WP34S allows you to directly access the first 16 local registers, using the dot as a prefix followed by two digits. Thus to store a value in the first local register, you use **STO .00**. Likewise, to recall a value from the second local register, you use **RCL .01**. Beyond the 16th register, you must use indirect addressing. The above code uses the command **LocR 3** to

dynamically allocate three local memory registers, R.00, R.01, and R.02. The command **PopLR** de-allocates the local registers. It is optional and I am including it for the sake of demonstration. By default, the local variables and flags in a subroutine are automatically removed when the subroutine reaches a **RTN** or **END** statement.

4. The program uses the alpha register to prompt the user for input and to display tagged/commented output values. By default, the WP34S simply appends characters to the alpha register. This feature is the reverse of HP41C's support for the alpha register, where inserted text automatically overwrites existing text in that register. Appending text in the HP-41C requires that you start with the special append character. Thus, to start inserting new text in the alpha register of the WP43S, you must first use the command **CLα** to explicitly clear the alpha register. Each program step can take up to three characters. So, for example to build the prompt text **TOLER?** I need two program steps. The first step inserts the first three characters **TOL**, while the second step inserts the last three characters **ER?**. The WP34S displays the character **α** to the left of the inserted text.

   You can insert one character per program step, but that is both wasteful and makes the listing harder to read. While entering letters in the alpha register is easy, entering the space, the question mark, the equal sign, and other punctuation characters requires some practice. The WP34S calculator stores various non-alpha characters in different menus and key combinations. I would like to point out that you can also append numbers to the alpha register. This feature, which I chose not to use in this article, can help to prompt program users in the input and display of array and matrix elements.

5. The program uses the **INC** command to increment the value of a memory (and also stack) register by one. The above listing uses the command **INC 06** to increment memory register 06 that stores the iteration counter. The WP34S has the **DEC** command to similarly decrement the value of a register by one. The **INC** and **DEC** commands are very handy in directly adding or subtracting 1 from a memory or a stack register without pushing 1 into the stack. This direct action does not disturb the stack. When all of the stack's contents are relevant, using **INC** or **DEC** is a welcome feature.

6. The **PSE *n*** command pauses the program for a specified number of tenths of a second. The above listing uses the command **PSE 10** to pause for one second.

The above listing has **LBL D** coded for the following function:

$f(x) = x - 10 \ln(1 + 4x^2 + 2x^4)$

Let's run the program to calculate the root with the initial guess of 50 and a tolerance value of 1E-8. The root is approximately 223.226. To run the program, perform the following steps:

1. Execute the command **XEQ 'OST'** to start and initialize the program. To rerun the program, simply press the key **A**.

2. The program prompts you for the tolerance value, as shown in Figure 6. The display shows the current tolerance value of 0.00000001. This is the initial value set by the program in step 1. If you enter a different value and rerun the program, you will see your most recent tolerance value. In the case of our example, accept the current tolerance value and press the **R/S** key.

Fig. 6 – The prompt for the tolerance value.

3. The program prompts you to enter the initial guess for the root, as shown in Figure 7. The value in the X register is a leftover from step 2. Just ignore it! Enter the value of 50 and then press the **R/S** key.

Fig.7– The prompt for the initial guess for the root.

4. The program pauses to display the series of improved guesses for the root. When it finds a refined root guess that is within the tolerance you specified, it displays the number of iterations, as shown in Figure 8. The figure shows how the alpha and X registers display a tagged (or commented) value. Simply press the **R/S** key to resume program execution.

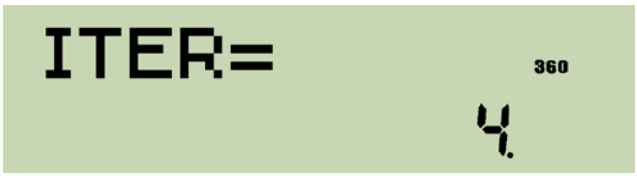

Fig. 8 – The number of iterations.

5. The program displays the improved guess for the root, as shown in Figure 9. Again, the figure shows the alpha and X registers displaying a tagged root value.

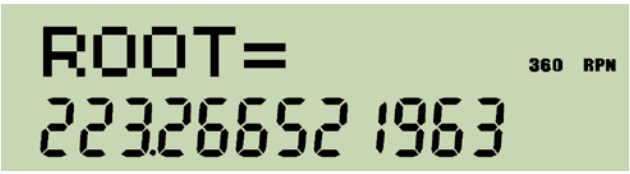

Fig. 9 – The refined root value.

You can press the **R/S** key and resume at step 3. Enter the initial guess of 1 to locate the other root for the function. The calculator displays the refined root of 0.025023481 in 4 iterations. Remember that you can evaluate the currently coded function f(x) by entering a value for x and then pressing the key **D**.

I encourage you to edit the code in **LBL D** in Table 3 to replace the currently coded function with the ones that appear in Table 2.

Observations and Conclusions

The article introduced you to Ostrowski's root-finding method. This fourth-order method excels over Newton's second-order method and Halley's third-order method. While Ostrowski's method has inspired new algorithms that further reduce the number of iterations, the basic Ostrowski method remains the most optimum as far as the total number of function calls. In addition to introducing you to the Ostrowski method, the article also presented two example listings: the BASIC-like program for the new HP 39gII and the RPN program for the repurposed HP 30b - WP34S. My hope is that the article will encourage you to use the efficient Ostrowski's method in future programs that you write.

References

1. A. M. Ostrowski, "Solution of Equations and Systems of Equations", Academic Press; second edition (1966).
2. Walter Gautschi, "Alexander M. Ostrowski (1893–1986): His life, work, and students", can be downloaded from http://www.cs.purdue.edu/homes/wxg/AMOengl.pdf.
3. Walter Gautschi, "Numerical Analysis", Birkhäuser Boston; 2nd ed. 2012 edition (December 6, 2011).
4. W. Kahan, "Personal Calculator Has Key to Solve Any Equation f(X)=0", Hewlett-Packard Journal, December 1979.
5. F. Soleymani and M. Sharifi, "A New Derivative-Free Quasi-Secant Algorithm For Solving Non-Linear Equations", World Academy of Science, Engineering and Technology 55 2009.
6. Xia Wang and Liping Liu, "New eighth-order iterative methods for solving nonlinear equations", Journal of Computational and Applied Mathematics 234 (2010) 1611-1620.
7. Reza Ezzati and Elham Azadegan, "A simple iterative method with fifth-order convergence by using Potra and Pták's method", Mathematical Sciences Vol. 3, No. 2 (2009) 191-200.
8. Guofeng Zhang, Yuxin Zhang, and Hengfei Ding, "New family of eighth-order methods for nonlinear equation", COMPEL volume 28, issue 6, 2009.
9. M. Heydari, S. M. Hosseini, and G. B. Loghmani, "ON TWO NEW FAMILIES OF ITERATIVE METHODS FOR SOLVING NONLINEAR EQUATIONS WITH OPTIMAL ORDER", Applicable Analysis and Discrete Mathematics 5 (2011), 93-109.

About the Author

Namir Shammas is a native of Baghdad, Iraq. He resides in Richmond, Virginia, USA. Namir graduated with a degree in Chemical Engineering from the University of Baghdad. He also received a master degree in Chemical Engineering from the University of Michigan, Ann Arbor. He worked for a few years in the field of water treatment before focusing for 17 years on writing programming books and articles. Later he worked in corporate technical documentation. He is a big fan of HP calculators and collects many vintage models. His hobbies also include traveling, music, movies (especially French movies), chemistry, cosmology, Jungian psychology, mythology, statistics, and math. As a former PPC and CHHU member, Namir enjoys attending the HHC conferences. Email him at: nshammas@aol.com