In: Recent advances in algorithmic differentiation, Berlin Heidelberg: Springer, p. 285-295.
Part of the Lecture Notes in Computational Science and Engineering book series (LNCSE) Volume 87
DOI:10.1007/978-3-642-30023-3_26

An AD-Enabled Optimization ToolBox in LabVIEW^{TM}

Abhishek Kr. Gupta and Shaun A. Forth

Abstract LabVIEW^{TM} is a visual programming environment for data acquisition, instrument control and industrial automation. This article presents LVAD, a graphically programmed implementation of forward mode Automatic Differentiation for LabVIEW. Our results show that the overhead of using overloaded AD in LabVIEW is sufficiently low as to warrant further investigation and that, within the graphical programming environment, AD may be made reasonably user friendly. We further introduce a prototype LabVIEW Optimization Toolbox which utilizes LVAD's derivative information. Our toolbox presently contains two main LabVIEW procedures `fzero` and `fmin` for calculating roots and minima respectively of an objective function in a single variable. Two algorithms, Newton and Secant, have been implemented in each case. Our optimization package may be applied to graphically coded objective functions, not the simple string definition of functions used by many of the optimizers of LabVIEW's own optimization package.

Key words: Forward mode AD, LabVIEW, graphical programming, optimization

Abhishek Kr. Gupta
Department of Electrical Engineering, IIT Kanpur, Kanpur 208016, India, g.kr.abhishek@gmail.com

Shaun A. Forth
Applied Mathematics and Scientific Computing, Cranfield University, Shrivenham, Swindon SN6 8LA, UK, S.A.Forth@cranfield.ac.uk

1 Introduction

LabVIEW[1] is a programming environment for data acquisition, instrument control and industrial automation [7]. LabVIEW programs are written in the visual programming language *G* [7]. Visual programming languages facilitate writing complex codes as flow diagrams by dragging and dropping inbuilt graphical icons representing an instrument, module or a subprogram and wiring icons together to establish the program's data flow. Visual programming eliminates the writing of programs as a collection of text commands making it popular among non-programming engineers and scientists. LabVIEW programs are called virtual instruments (VI) as they typically represent actual laboratory equipment. Each LabVIEW VI has two components: the *Front Panel* containing the VI's controls (inputs) and displays (outputs); and the *Block Diagram* which defines the VI's data flow.

Many scientific and engineering applications involve optimization so necessitating an optimization package in LabVIEW. LabVIEW provides a handful of optimization algorithms but many of these are limited to optimizing objective functions coded as simple equation expressions within a string [7]. Further, there appears to be no way for the user to provide derivative information.

AD tools exist for a wide number of programming languages e.g., C, C++, Fortran, MATLAB, Python[2]. However, AD of *visual programming languages*, such as LabVIEW, appears under-researched. The equation-based simulation language Modelica[3] is frequently programmed via a visual programming environment. Elsheikh et al. [2] considered AD of Modelica by source transformation of the model's representation in the Modelica programming language and also by a symbolic approach [1]; differentiation of the visual program was not considered.

Our LVAD package implements forward mode AD using operator overloading in LabVIEW's visual programming language G as described in Sect. 2. This is the first presentation of LVAD outside of the student competition paper [6]. Section 3 describes the implementation of our Optimization toolbox with results presented in Sect. 4 and conclusions in Sect. 5.

2 Implementation of AD in LabVIEW: the LVAD Package

Our LVAD package's forward mode AD [5, Chap. 3] differs from that for standard operator overloading [5, Chap. 6] since it is implemented in LabVIEW's visual programming paradigm. In Sect. 2.1 we define an LVAD class whose objects posses

[1] LabVIEW^{TM} is a trademark of National Instruments. This publication is independent of National Instruments, which is not affiliated with the publisher or the author, and does not authorize, sponsor, endorse or otherwise approve this publication.

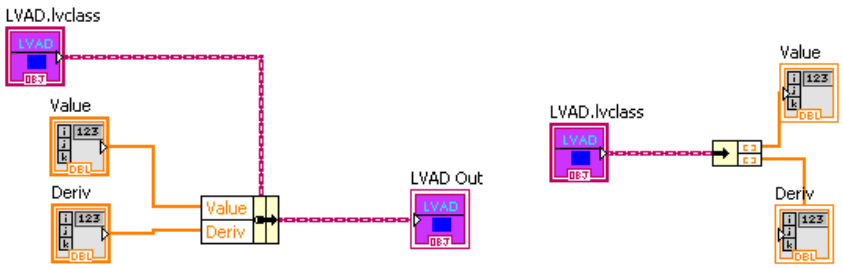
[2] See `www.autodiff.org` for a list of such tools.

[3] `https://modelica.org/`

value and derivative components, in Sect. 2.2 we use visual programming to overload arithmetic operations and we present an example of use in Section 2.3.

2.1 LVAD Class

Figure 1(a) shows the visual programming of a LVAD object's `set` operation that takes as input the `Value` and `Deriv` supplied by a front panel, say, and assigns them to the appropriate components of an LVAD object. The attributes of `Value` and `Deriv` indicate that these are both arrays (the `i,j,k` attribute) and contain numeric data (the `1 2 3` attribute) of type double (the `DBL` attribute). Figure 1(b) shows the `get` method for extracting an object's two components.



(a) The LVAD set method (b) The LVAD get method

Fig. 1 Visual programming of an LVAD object's `set` and `get` methods

2.2 Operator Overloading

The usual arithmetic operations for forward mode AD [5, Sec. 3.1] are defined by overloading LVAD objects. Figure 2 implements the product rule for multiplication of two LVAD objects. Note how the `get` operation of Fig. 1(a) is used to access the `Value` and `Deriv` components of `X` and `Y` and then the intrinsic addition and multiplication operations form the product's value and derivatives before `set` assigns them to the product object's components.

Similarly, other arithmetic operations and intrinsic functions (e.g., sin) may be overloaded [6].

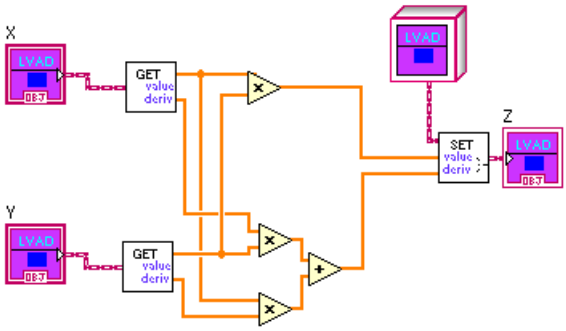


Fig. 2 Overloaded multiplication of two LVAD objects $Z = X * Y$

2.3 Examples

Consider obtaining the derivatives $\partial f/\partial x$ and $\partial f/\partial c$ of the scalar function,

$$f(x) = (x-c)^2 \sin \frac{4\pi e^{\cos(x)}}{\sqrt{(x-a)}}, \tag{1}$$

with constant $a = 2$ and variable $c = 2$. The front panel of Fig. 3 permits the user

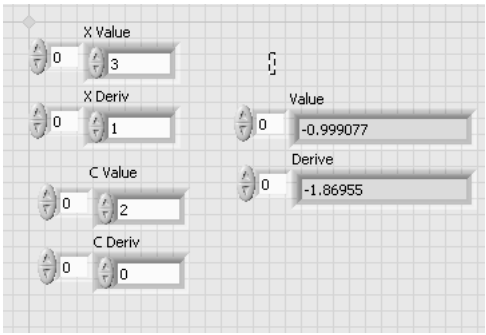


Fig. 3 Front panel to enable differentiation of the function (1)

to set the value and derivative of both x and c and observe the computed function's value and derivative. The function value and derivative are correct for $\partial f/\partial x$ at $x = 3$ and $c = 2$. To perform this overloaded AD computation the function was programmed using the arithmetic operations and functions of the LVAD class by standard LabVIEW techniques as seen in Fig. 4.

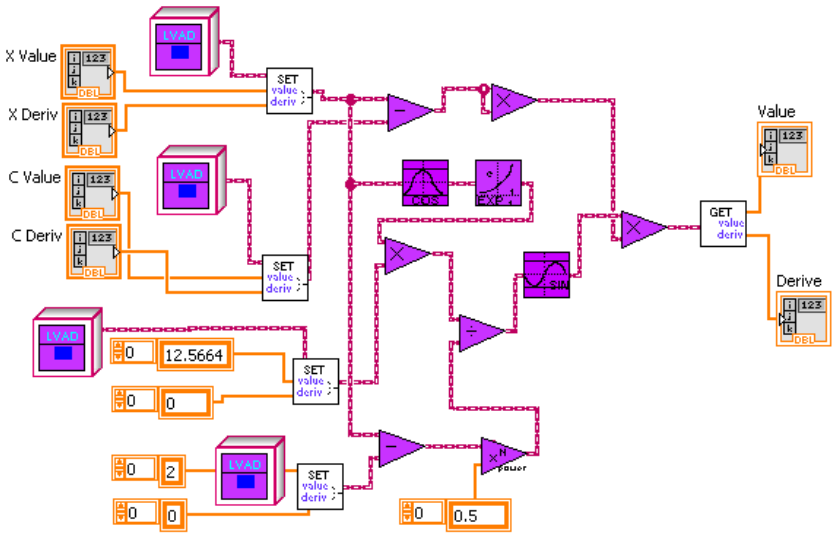


Fig. 4 Visual programming to differentiate function (1)

3 Implementation of a LabVIEW Optimization Toolbox

The two main Subroutine Virtual Instruments (subVIs) of our optimization toolbox are Sect. 3.1's `fzero` (1-D root finding) and Sect. 3.2's `fmin` (1-D minimization).

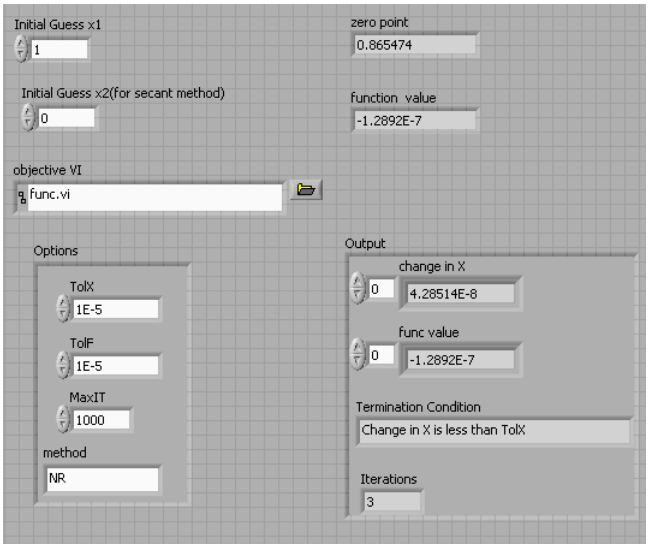
3.1 Root finding - the `fzero` subVI

Figure 5(a) depicts the `fzero` front panel and Fig. 5(b) the corresponding subVI in the case of Newton iteration; derivative-free Secant iteration may also be employed. The front panel allows the user to: nominate an objective function VI which must have a single LVAD input and single LVAD output; set an initial value for the iteration, or two such values for the Secant method; set solver options (tolerances, maximum iterations, method) within the LabVIEW equivalent of a structure termed a *cluster*[4]. On completion the calculated root x^*, function value $f(x^*)$ and, via a cluster, iteration summary outputs are displayed.

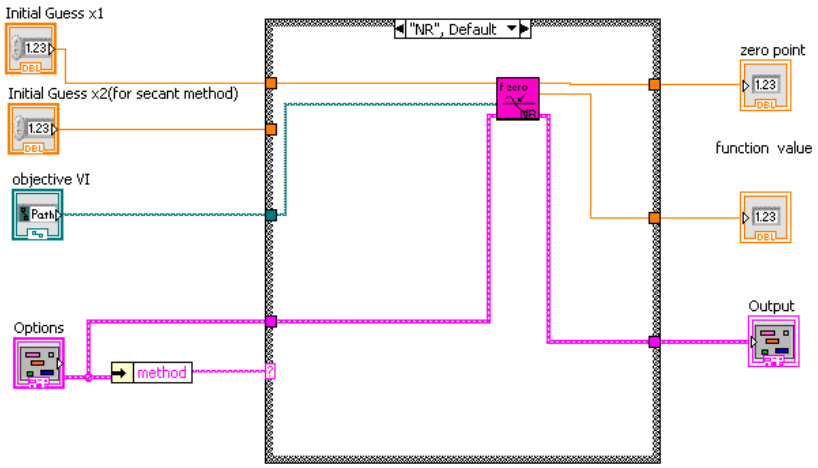
The subVI of Fig. 5(b) shows that, depending on the method selected, `fzero` calls either the `fzeroNR` or the `fzerosecant` VI[5]. For this proof-of-concept work we adopt iteration without any global convergence enhancements [8]. Iteration

[4] A further VI (details omitted for brevity) is supplied to set these options within the cluster.

[5] Both block diagrams omitted for brevity.



(a) `fzero` front panel



(b) `fzero` block diagram when Newton-Raphson selected

Fig. 5 The `fzero` subVI

continues until simple convergence conditions are met ($|x| <$ `tolX`, $|f(x)| <$`tolF`, or maximum iterations exceeded).

3.2 Minimization - the `fmin` subVI

Minimizing a function in a single variable is performed by Newton or Secant iteration on the stationary equation $f'(x) = 0$ by the `fmin` VI. The user must provide the objective function $f(x)$ in the form of a subVI. The program calculates $f'(x)$ from $f(x)$ by overloaded AD for both the Secant and Newton methods. For the Newton method, the second derivative $f''(x)$ is calculated by one-sided differencing of $f'(x)$.

4 Results

We present a simple example of our package use for root finding in Sect. 4.1 and then present performance testing in Sect. 4.2.

4.1 Simple Example

The objective VI of Fig. 6 corresponds to the objective function

$$f(x) = \cos(x) - x^3, \tag{2}$$

The input, output and all arithmetic operations and function calls are of LVAD class.

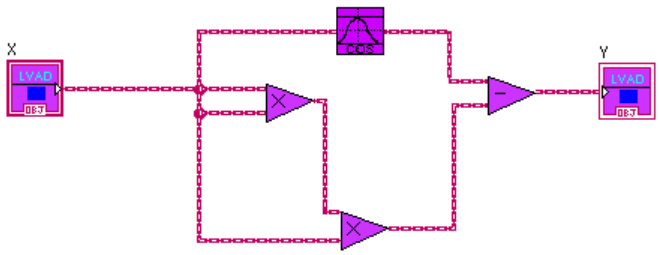
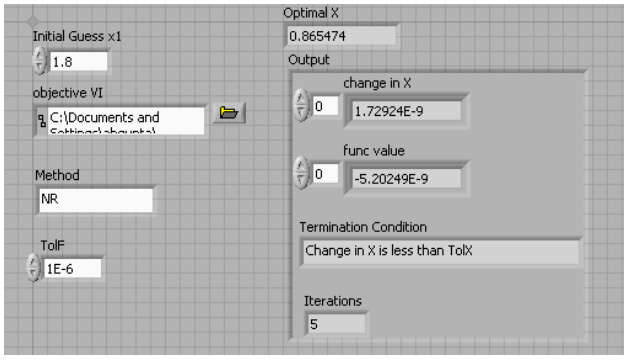
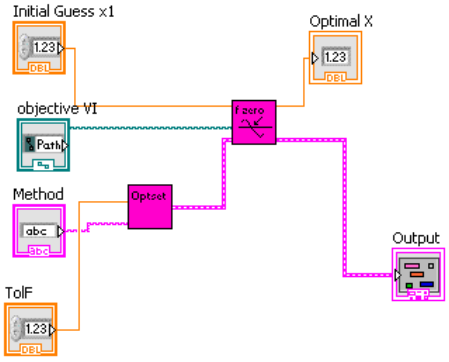


Fig. 6 Objective function VI for $f(x) = \cos(x) - x^3$

Figure 7 shows a suitable front panel and block diagram for use of Fig. 5's `fzero` to find a zero of (2); the zero is found in 5 iterations at $x = .865474$. The function may similarly be minimized using `fmin` of Sect. 3.2.



a) Front Panel



b) subVI

Fig. 7 Front panel and subVI for function zero example

4.2 Performance Testing

For objective function (2) and a tolerance of 1.0×10^{-6}, Table 1 compares performance of: our Secant `fzero` method without overloading the objective; our Secant and Newton Raphson methods when overloaded with LVAD; and the inbuilt LabVIEW Newton Raphson zero finder [7]. Our Newton method used an initial $x = 1$ and all others used the initial pair $x = 1, -3$. As the root is located at $x \approx 0.865$ this avoids giving an *unfair advantage* to the latter three methods. The timing difference between the two Secant methods is due to the overhead of overloaded AD (n.b., the derivatives computed are not used). The improved convergence rate of Newton makes up some of this overhead by using one less iteration. The inbuilt method has least execution time owing to its optimized implementation as an executable.

Table 2 gives CPU times to minimize $f(x) = x^2 - \sin(x)$ to a tolerance of 1.0×10^{-6} with: our package's Secant and Newton methods (with overloaded AD); and LabVIEW's inbuilt Quasi-Newton and Brent's methods. We indicate whether

Table 1 Performance testing for `fzero`: variation in number of iterations per solution and mean CPU time (ms) per solution with method for 1,000 repeated solutions. A dash indicates unavailable information

Method	AD	iterations	CPU time (ms)
LVAD:Secant	no	4	2.232
LVAD:Secant	yes	4	14.52
LVAD:Newton	yes	3	9.670
inbuilt:Newton	no	-	0.199

the objective is supplued as a VI or string. For the (Quasi) Newton methods an initial value of $x = 1$ was used; for Secant the pair $x = -1.2, 1$; and for Brent's the triplet $x = -1.2, 1, 2.2$ (n.b., the minimum is located at $x \approx 0.45$). The inbuilt functions have least execution time owing to their optimized implementation as executables; the large number of function evaluations is possibly due to poor derivative accuracy preventing asymptotic superlinear convergence. Our LVAD:Secant method is, encouragingly for a one-dimensional optimization, only some 30% slower. We are currently unable to explain the high CPU time for LVAD:Newton..

Method	AD	objective supplied as	function evaluations	CPU time (ms)
LVAD:Secant	yes	VI	19	0.531
LVAD:Newton	yes	VI	6	3.620
inbuilt:Quasi-Newton	no	string	47	0.447
inbuilt:Quasi-Newton	no	VI	47	0.400
inbuilt:Brent's	no	string	-	0.433

Table 2 Performance testing for `fmin`: variation in number of iterations per solution and mean CPU time (ms) per solution with method for 1,000 repeated solutions. A dash indicates unavailable information

5 Conclusions

Following Sect. 2's description of our overloaded forward mode AD package, Sect. 3 detailed how we utilized AD in Newton methods for single variable root finding and minimization in a prototypical LabVIEW optimisation package. Our package accepts graphically coded definitions of the objective function, an advantage over the restrictive string definitions of many of LabVIEW's inbuilt optimization functions. Section 4's performance testing showed that overloading overheads, though noticeable, are not sufficiently large to warrant discarding our approach.

A disadvantage of our LabVIEW approach, compared to AD in compiled languages, is that the objective function must be re-coded by replacing all the subVI's

inbuilt class function calls and arithmetic operations by those of the LVAD class. In compiled languages one simply changes, perhaps automated by scripting or templating, the class or type of the objects [4, 9]. This task is unnecessary in MATLAB as objects acquire the class of the result of the assignment that creates them [3].

The LVAD class might be extended to vector forward mode, perhaps by utilizing a specialized vector derivative storage and linear combination class [3]. Then we might extend our optimization toolbox for functions with $x \in \mathbb{R}^n$.

Finally, we note that LabVIEW saves VIs in a propriety format making source-transformation AD approaches almost impossible without the cooperation of LabVIEW's owner National Instruments.

6 Acknowledgements

The authors thank National Instruments for permission to include LabVIEW screenshots.

References

1. Elsheikh, A., Noack, S., Wiechert, W.: Sensitivity analysis of Modelica applications via automatic differentiation. In: 6th International Modelica Conference, vol. 2, pp. 669–675. Bielefeld, Germany (2008)
2. Elsheikh, A., Wiechert, W.: Automatic sensitivity analysis of DAE-systems generated from equation-based modeling languages. In: C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, J. Utke (eds.) Advances in Automatic Differentiation, pp. 235–246. Springer (2008). DOI 10.1007/978-3-540-68942-3_21
3. Forth, S.A.: An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. ACM Transactions on Mathematical Software **32**(2), 195–222 (2006). URL http://doi.acm.org/10.1145/1141885.1141888
4. Griewank, A., Juedes, D., Utke, J.: Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. ACM Transactions on Mathematical Software **22**(2), 131–167 (1996). URL ftp://info.mcs.anl.gov/pub/tech_reports/reports/TM162.ps
5. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd edn. No. 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA (2008). URL http://www.ec-securehost.com/SIAM/OT105.html
6. Gupta, A.K., Agrahari, A.: LVAD package: Implementation of forward mode automatic differentiation in LabVIEW using operator overloading. VI Mantra Technical Paper Writing Contest, National Instuments (2008). URL https://sites.google.com/site/gkrabhishek/projects/publications
7. LabVIEW 2011 Help (2011). URL http://digital.ni.com/manuals.nsf/websearch/7C3F895E4B50A03D862578D400575C01
8. Nocedal, J., Wright, S.J.: Numerical Optimization, 2nd edn. Springer, New York (2006)
9. Pryce, J.D., Reid, J.K.: ADO1, a Fortran 90 code for automatic differentiation. Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England (1998). URL ftp://ftp.numerical.rl.ac.uk/pub/reports/prRAL98057.pdf