

DNAD, a Simple Tool for Automatic Differentiation of Fortran Codes Using Dual Numbers

Wenbin Yu*

Utah State University, Logan, Utah 84322-4130

and

Max Blair †

Air Force Research Laboratory, Wright-Patterson AFB, Ohio 45433-7542

The focus of this Note is to introduce DNAD (dual number automatic differentiation) as a simple, general-purpose tool to carry out automatic differentiation of Fortran codes including those written in F77. It is a F90/95 module developed using the arithmetic of dual numbers and F90/95 operator overloading feature. Very minimum changes of the source codes are needed to enable sensitivity calculation of existing Fortran programs. In comparison to existing Fortran AD tools, DNAD only differentiates the analysis code once and the designer does not have to access to the differentiated source codes. In comparison to the complex-step approximation method, DNAD does not require more changes to existing source codes and is more efficient and accurate.

Introduction

Optimization is concerned with finding the optimal system by exploring the feasible design space using analysis tools. Optimization methods can be generally classified as two categories: gradient free methods and gradient based methods. Gradient free methods rely only on the value of the objective function. Grid searching, genetic algorithm, neural networks, etc. fall into this category.¹ A great limitation of these methods is that the computation quickly become prohibitive as the number of design variables increases. Gradient based methods use not only the value of the objective and constraint functions but also its derivatives. Gradient based methods are only effective if the Lagrangian is smooth in the design space. The success of

*Associate Professor, Department of Mechanical and Aerospace Engineering. Senior Lifetime Member, AIAA; Member, ASME and AHS.

†Research Aerospace Engineer, Air Vehicles Directorate. Associate Fellow, AIAA.

these methods hinges on accurate and efficient sensitivity analysis. Sensitivity analysis is usually the most costly step in the optimization cycle, and the optimization process often fails due to inaccurate sensitivity calculations.

Sensitivity analysis focuses on evaluating derivatives² of outputs of the analysis tool with respect to inputs. There are several methods proposed for sensitivity analysis. Traditionally, finite differences are used to evaluate the sensitivity if the source codes of the analysis tools are not accessible. Basically, one perturbs each input and evaluates the difference of the outputs. The “step-size dilemma” forces a choice between a small step size to minimize truncation error and a step so small that errors due to subtractive cancellation become dominant.³

If the source codes are accessible, then one can use so-called automatic differentiation (AD)² or complex-step approximation⁴ to provide a more accurate and robust evaluation of the sensitivities. AD, also known as computational differentiation or algorithmic differentiation, is a well-established method based on the systematic application of the chain rule of differentiation to each operation in the program flow;² which can be carried out in a forward mode or a reverse mode. The forward mode can be easily implemented by a nonstandard interpretation of the program in which the real numbers are replaced by so-called dual numbers, the details of which are given in the next section. As far as programming concerned, the forward mode is usually implemented using two strategies: Source Code Transformation (SCT) and Operator Overloading (OO)⁵. A complete list of SCT or OO tools for automatic differentiation of Fortran codes can be found at www.autodiff.org. Using SCT, more source codes are generated automatically based on the original source for to evaluate the derivatives. Although SCT can be implemented for all programming language and it is easier for an optimized compilation of the code, the original source code is greatly enlarged and makes it difficult to debug the extended code and also it is very difficult to develop AD tool for automatic generation of the additional source codes.⁶ There are more than a dozen tools that use SCT for Fortan codes with ADIFOR⁷ being the most known one. OO is a more elegant approach if the source code is written in a language supporting it. In OO, one defines a new structure containing dual numbers and overloads elementary mathematical operations to this new data structure. The main changes to the source codes will be re-declaring real numbers with the new data structure. It is noted that OO is not only applicable to source codes written in a supporting language such as Fortran 90/95 and C++ but also applicable for the

codes written in a language compatible with a supporting language. For example, codes written in F77. One just need to compile the operator overloaded source codes using a Fortran 90/95 compiler. There are several automatic differentiation tools for Fortran codes using OO including AD01,⁸ AUTO_DERIV,⁹ and ADF.¹⁰ There are certain limitations with these tools. For example, AD01 does not support the array features of F90/95. ADF, although much faster than AUTO_DERIV, is about six times slower than analytic derivative calculation. Another limitation of these OO AD tools is that the independent variables and dependent variables are specified inside the code, which implies that the designer needs to recompile the differentiated version of the analysis code when the sensitivities of differen functions are needed or the design parameters are different. This demands the designer to know the meaning of variables inside the code and know how to compile the differentiated code. In other words, the analysis code has to be differentiated many times. Furthermore, the fact that often times the developer of the AD version of the analysis code is not the designer who is going to use the code discourages the use of such AD tools. For these types of AD, it is better to develop AD tools that can render the sensitivity analysis using the differentiated version of the analysis tools independent of the source codes.

The complex-step approximation method, publicized by Martins,¹ was originally proposed as a better alternative to replace finite difference as it can avoid the step-size dependency and the subtractive cancellation errors inherent in finite difference techniques. An extremely small number can be assigned to get a second-order approximation of the derivatives. Later the complex-step approximation is found similar to the automatic differentiation⁶ and it can be considered as an approximate implementation of automatic differentiation using the existing complex data structure in some computer languages. Because of its clear advantages over finite difference method and its easy implementation and the end use of differentiated version of the code is independent of the source codes, the complex-step approximation method is popular in the aerospace design community. However, some issues of the complex-step approximation can only be resolved using automatic differentiation.⁶ And it is less efficient than automatic differentiation using dual numbers.

SCOOT¹¹ is is an experimental variant of OO that is specialized to support sensitivities for vector calculus and geometric sensitivities in C++. With SCOOT, any number of sensitivities are computed in parallel with machine accuracy. The method is validated to support sensitivities geometric variables in a linear aerodynamic application.

The focus of this Note is to develop a simple, general-purpose F90/95 module, namely dual number automatic differentiation (DNAD), using the arithmetic of dual numbers and F90/95 OO features for automatic differentiation of Fortran codes. In comparison to existing Fortran AD tools using OO, DNAD only differentiates the analysis code once and the designer does not need to access to the differentiated source codes. A preprocessor is created to loop on the list of independent design variables, sequentially activating each independent variable and DNADS subsequently computes the sensitivities to machine accuracy. In comparison to the complex-step approximation method popular in the aerospace community, DNAD is more efficient and accurate, and does not require more changes to existing source codes.

Dual Number Arithmetic

The dual number arithmetic can be understood through an example. For example, we want to evaluate the derivative of $f(x_1, x_2) = x_1x_2 + \sin(x_1)$. If we extend the real numbers with an additional component, ie,

$$\langle x_1, x'_1 \rangle = x_1 + x'_1d_1 \quad \langle x_2, x'_2 \rangle = x_2 + x'_2d_2 \quad (1)$$

where x'_1 and x'_2 are real numbers but d_1 and d_2 are symbols, which is analogous to the imaginary unit I in the complex number arithmetic. However, we let all powers of d_1 and d_2 higher than one and d_1d_2 equal to zero in dual numbers, as opposed to $I^2 = -1$ in complex numbers. Substituting these new numbers into the function, we have

$$\begin{aligned} f(x_1 + x'_1d_1, x_2 + x'_2d_2) &= (x_1 + x'_1d_1)(x_2 + x'_2d_2) + \sin(x_1 + x'_1d_1) \\ &= x_1x_2 + x_1x'_2d_2 + x_2x'_1d_1 + \sin(x_1) + \cos(x_1)x'_1d_1 \\ &= x_1x_2 + \sin(x_1) + [x_2 + \cos(x_1)]x'_1d_1 + x_1x'_2d_2 \\ &= f(x_1, x_2) + \frac{\partial f}{\partial x_1}x'_1d_1 + \frac{\partial f}{\partial x_2}x'_2d_2 \end{aligned} \quad (2)$$

Hence the derivative $\frac{\partial f}{\partial x_1}$ can be obtained by letting $x'_1 = 1, x'_2 = 0$ and $\frac{\partial f}{\partial x_2}$ can be obtained by letting $x'_1 = 0, x'_2 = 1$. If x_1, x_2 are independent variables, x'_1, x'_2 are commonly called seeds and can be arbitrary. However, we choose them to be the unit number so that the outputs will be the derivatives themselves. Also here we restrict to one independent variable for easy implementation. For sensitivity analysis with respect

to multiple independent variables, we can run multiple analyses with different seeds. The dual number arithmetic for a general function is as follows:

$$g(\langle u, u' \rangle, \langle v, v' \rangle) = \langle g(u, v), g_u u' + g_v v' \rangle \quad (3)$$

with $g_u = \frac{\partial g}{\partial u}$ and $g_v = \frac{\partial g}{\partial v}$. With this general formula, we can easily derive the following:

$$\langle u, u' \rangle + \langle v, v' \rangle = \langle u + v, u' + v' \rangle \quad \langle u, u' \rangle - \langle v, v' \rangle = \langle u - v, u' - v' \rangle \quad (4)$$

$$\langle u, u' \rangle \times \langle v, v' \rangle = \langle uv, u'v + uv' \rangle \quad \langle u, u' \rangle / \langle v, v' \rangle = \langle \frac{uv}{v^2}, \frac{u'v - uv'}{v^2} \rangle \quad (5)$$

Addition and subtraction are the same as the complex arithmetic. However, multiplication and division operations by the complex numbers involve more calculations as disclosed by the following formulas:

$$(u + u'I) \times (v + v'I) = (uv - u'v') + I(u'v + uv') \quad (u + u'I)/(v + v'I) = \frac{uv - u'v'}{v^2 + v'^2} + I \frac{u'v - uv'}{v^2 + v'^2} \quad (6)$$

And multiplication and division of complex number will be numerically the same as dual numbers if the seeds u' and v' used in the complex arithmetic are extremely small numbers. For more sophisticated functions such as trigonometric functions, exponential functions, logarithm and others, the calculations of the complex-step approximation method will be more involved than dual numbers. It has been realized in Ref. [6] and some of the complex functions are replaced with a definition similar to dual numbers. However, it is impossible to redefine the functions and operations if they are intrinsic in a programming language, such as Fortran. Hence, it is obvious the dual number automatic differentiation (DNAD) has the following advantages comparing to the complex-step approximation:

- DNAD is more efficient as the calculations of dual numbers involved in computing derivatives are never more than and mostly less than complex numbers.
- DNAD will be more accurate as the complex-step approximation is only valid for extremely small imaginary parts. For some functions and operations, if it is impossible to redefine them the same as dual number arithmetic, some cancelation and truncation errors might occur.

Of course, the disadvantage using dual numbers is that we need to overload all the operations and functions to this new data structures while the complex-step approximation only needs to overload part of these for some languages such as Fortran having complex arithmetic as one of intrinsic data types. However, as long as a general-purpose module or class is written, the efforts for differentiating an existing source code will be similar. Of course, for computer languages which do not have the complex number as an intrinsic data type, such advantage of complex number does not exist. For this purpose, DNAD (Dual Number Automatic Differentiation) is developed as a general-purpose module using the operator overloading feature of F90/F95 to differentiate Fortran codes including those written in Fortran 77. The same approach can be easily applied to develop automatic differentiation tools for codes written in other language.

First, we define a new data type *DUAL_NUM* as follows

```
TYPE,PUBLIC:: DUAL_NUM
    REAL(DBL_AD)::x_ad_
    REAL(DBL_AD)::xp_ad_
END TYPE DUAL_NUM
```

where *DBL_AD* is an integer indicating the precision used for the code, *x_ad_* is the function value and *xp_ad_* is the corresponding derivative (the second component of the dual number). What one needs to do next is to overload the functions and operations needed for computing such as relational operators, arithmetic operators and functions. For example, one can overload the ABS function as follows:

```
INTERFACE ABS
    MODULE PROCEDURE ABS_D
END INTERFACE

ELEMENTAL FUNCTION ABS_D(u) RESULT(res)
    TYPE (DUAL_NUM), INTENT(IN)::u
    TYPE (DUAL_NUM)::res
    res%x_ad_ = ABS(u%x_ad_)
    IF(u%x_ad_>=0) THEN
        res%xp_ad_ = u%xp_ad_
    ELSE
        res%xp_ad_ = -u%xp_ad_
    END IF
END FUNCTION ABS_D
```

ELSE

res%xp_ad_ = -u%xp_ad_

ENDIF

END FUNCTION ABS_D

All the other functions can be defined similarly. The current version of DNAD has all the common relational operators, arithmetic operators and functions defined and it is ready to be used to differentiate existing Fortran code.

How to Use DNAD

To use DNAD to automatically differentiate a Fortran code, one needs to carry out the following steps:

- Replace all the definitions of real numbers in the existing code with the new definition of the dual numbers. If the real number is also initialized along with the declaration, the initialization should be changed correspondingly. For example, *REAL(8),PARAMETER:: ONE=1.0D0* should be changed to *TYPE(DUAL_NUM),PARAMETER::ONE=DUAL_NUM(1.0D0,0.D0)*.
- Insert *Use Dual_Num_Auto_Diff* right after *Module, Function, Subroutine* statements.
- Change IO commands correspondingly. If the code uses free formatting read and write, no changes are needed. The developer of the differentiated code only needs to instruct the end user to insert 1 after the real number representing the design parameter and 0 after all other real numbers in the input file. In the output file, the number right after the function value indicates the sensitivity of this function with respect to the given design parameter.
- Recompile all the source codes and link with *Dual_Num_Auto_Diff.o* to generate the executable.

For example for the following Fortran code

PROGRAM CircleArea

REAL(8),PARAMETER:: PI=3.141592653589793D0

REAL(8)::radius, area

READ(,*) radius*

*Area=PI*radius**2*

```
WRITE(*,*) "AREA=", Area
END PROGRAM CircleArea
```

should be changed to be the following for sensitivity analysis using DNAD:

```
PROGRAM CircleArea

USE Dual_Num_Auto_Diff

TYPE (DUAL_NUM),PARAMETER:: PI=DUAL_NUM(3.141592653589793D0,0.D0)

TYPE (DUAL_NUM)::radius,area

READ(*,*) radius

Area=PI*radius**2

WRITE(*,*) "AREA=",Area

END PROGRAM CircleArea
```

where the bold text indicates the changes. One also needs to make a simple change to the inputs: adding one more real number into any real number as its seed. For example, for the above program, to calculate the area of a circle with radius 5.0 and its sensitivity, we just need to input *5.0,1.0*. The output will be *AREA=78.5398163397448 31.4159265358979*, where the first output is the functional value and the second is the corresponding sensitivity.

We have used DNAD to differentiate GEBT¹², a Fortran code implementing the mixed formulation of a geometrically nonlinear beam theory.¹³ The current version of GEBT, GEBT 2.0, has more than 10,000 lines coding mostly written using F90/95. It also invokes a F77 Harwell Subroutine Library (HSL-MA48) for solving sparse linear systems and other supporting F77 libraries. We obtained a correctly differentiated version of GEBT (GEBT-AD) by simply replacing declarations of all real numbers with *TYPE(DUAL_NUM)* and inserting *USE Dual_Num_Auto_Diff* right after all Module/Function/Subroutine statements, and recompiling GEBT source along with *Dual_Num_Auto_Diff.o*. Although this process can be easily automated, it only takes less than five minutes to manually modify the source codes. We have tested various problems and find out that the running time of the differentiated code (GEBT-AD) is about three times of the corresponding undifferentiated code (GEBT). Note even with knowing the appropriate step size a priori, one needs to run the analysis code at least twice to calculate the sensitivity. This efficiency is believed to be much better than ADF as it is about six times slower than analytic derivative calculation, which will be around 12 times slower

than the corresponding undifferentiated version of the code. Unfortunately, ADF is a commercial product and the authors do not have access to this tool for a benchmark study.

Conclusion

Exploiting the operator overload feature of F90/95, we have developed DNAD as a simple, general-purpose, automatic differentiation tool for Fortran codes written in Fortran 77/90/95. It is based on the dual number arithmetic and achieves the same accuracy as analytic differentiation. In comparison to existing Fortran AD tools, DNAD only differentiates the analysis code once and the designer does not have to access to the differentiated source codes. In comparison to the complex-step approximation method, DNAD is more efficient and accurate, and does not require more changes to existing source codes. DNAD has been used to differentiate GEBT, a geometrically exact beam analysis code, very satisfactory results have been obtained.

Acknowledgements

The development of DNAD is supported, in part, by the Chief Scientist Innovative Research Fund at AFRL/RB WPAFB. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the funding agency.

References

- ¹J. R. R. A. Martins. *A Coupled-Adjoint Method for High-Fidelity Aero-Structural Optimization*. PhD thesis, Aerospace Engineering, Stanford University, October 2002.
- ²A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, 2000.
- ³J. R. R. A. Martins, P. Sturdza, and J. J. Alonso. The connection between the complex-step derivative approximation and algorithmic differentiation. In *Proceedings of the 39th Aerospace Sciences Meeting*, Reno, NV, Jan. 2001. AIAA Paper 2001-0921.
- ⁴J. R. R. A. Martins, I. M. Kroo, and J. J. Alonso. An automated method for sensitivity analysis using complex variables. In *Proceedings of the 38th Aerospace Sciences Meeting*, Reno, NV, Jan. 2000. AIAA Paper 2000-0689.
- ⁵TBD. Automatic differentiation. Technical Report http://en.wikipedia.org/wiki/Automatic_differentiation,

Wikipedia, 2009.

⁶J. R. R. A. Martins, P. Sturdza, and J. J. Alonso. The complex-step derivative approximation. *ACM Transactions on Mathematical Software*, 29:245– 262, 2003.

⁷C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3:18– 32, 1996.

⁸J. D. Pryce and J. K. Reid. ADO1, a Fortran 90 code for automatic differentiation. Technical Report <ftp://matisa.rl.ac.uk/pub/reports/prRAL98057.ps.gz>, Rutherford Appleton Laboratory, 1998.

⁹S. Stamatiadis, R. Prosimiti, and S. C. Farantos. AUTO-DERIV: Tool for automatic differentiation of a Fortran code. *Computer Physics Communications*, 127:343– 355, 2000.

¹⁰C. W. Straka. ADF95: Tool for automatic differentiation of a Fortran code designed for large numbers of independent variables. *Computer Physics Communications*, 168:123– 139, 2005.

¹¹M. Blair. SCOOT: sensitivity class with operator overloaded types. In *Proceedings of the 51st AIAA AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, Orlando, Florida, April. 2010. AIAA Paper 2010-2918.

¹²Y. Wenbin and M. Blair. GEBT: a general-purpose tool for non-linear analysis of composite beams. In *Proceedings of the 51st AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, Orlando, Florida, April. 2010. AIAA Paper 2010-3019.

¹³D. H. Hodges, X. Shang, and C. E. S. Cesnik. Finite element solution of nonlinear intrinsic equations for curved composite beams. *Journal of the American Helicopter Society*, 41(4):313 – 321, Oct. 1996.