Utah State University
DigitalCommons@USU

Mechanical and Aerospace Engineering Faculty Publications | Mechanical and Aerospace Engineering

1-2013

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

Wenbin Yu
Utah State University

Maxwell Blair

Follow this and additional works at: http://digitalcommons.usu.edu/mae_facpub

Part of the Computational Engineering Commons, Computer Engineering Commons, and the Mechanical Engineering Commons

Recommended Citation

Yu, Wenbin and Blair, Maxwell, "DNAD, a Simple Tool for Automatic Differentiation of Fortran Codes Using Dual Numbers" (2013). *Mechanical and Aerospace Engineering Faculty Publications.* Paper 30.
http://digitalcommons.usu.edu/mae_facpub/30

This Article is brought to you for free and open access by the Mechanical and Aerospace Engineering at DigitalCommons@USU. It has been accepted for inclusion in Mechanical and Aerospace Engineering Faculty Publications by an authorized administrator of DigitalCommons@USU. For more information, please contact dylan.burns@usu.edu.

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

Wenbin Yu [a] and Maxwell Blair [b]

[a] *Utah State University, Logan, Utah 80322-4130*

[b] *Air Force Research Laboratory, Wright-Patterson AFB, Ohio 45433-7542*

Abstract

DNAD (dual number automatic differentiation) is a simple, general-purpose tool to automatically differentiate Fortran codes written in modern Fortran (F90/95/2003) or legacy codes written in previous version of the Fortran language. It implements the forward mode of automatic differentiation using the arithmetic of dual numbers and the operator overloading feature of F90/95/2003. Very minimum changes of the source codes are needed to compute the first derivatives of Fortran programs. The advantages of DNAD in comparison to other existing similar computer codes are its programming simplicity, extensibility, and computational efficiency. Specifically, DNAD is more accurate and efficient than the popular complex-step approximation. Several examples are used to demonstrate its applications and advantages.

PROGRAM SUMMARY

Program Title: DNAD
Journal Reference:
Catalogue identifier:
Licensing provisions: none
Programming language: F90/95/2003
Computer: all computers with a modern FORTRAN compiler
Operating system: all platforms with a modern FORTRAN compiler
Keywords: Automatic differentiation, Fortran, Sensitivity analysis
Classification: 4.12, 6.2
Nature of problem: Derivatives of outputs with respect to inputs of a Fortran code are often needed in physics, chemistry, and engineering. The author of the analysis code may no longer available and the user may not have deep knowledge of the

Email address: wenbin.yu@usu.edu (Wenbin Yu).
URL: http://www.mae.usu.edu/faculty/wenbin/ (Wenbin Yu).

Preprint submitted to Computer Physics Communications *5 February 2013*

code. Thus a simple tool is necessary to automatically differentiate the code through very minimum change of the source codes. This can be achieved using dual number arithmetic and operator overloading.
Solution method: A new data type is defined with the first scalar component holding the function value and the second array component holding the first derivatives. All the basic operations and functions are overloaded with the new definitions according to dual number arithmetic. To differentiate an existing code, all real numbers should be replaced with this new data type and the input/output of the code should also be modified accordingly.
Restrictions: None imposed by the program.
Unusual features: None.
Running time: For each additional independent variable, DNAD takes less time than than the running time of the original analysis code. However the actual running time depends on the compiler, the computer, and the operations involved in the code to be differentiated.

1 Introduction

In this age, most problems in physics, chemistry, and engineering are solved using computer codes. It is a frequent occurrence that we need to compute the derivatives of the outputs of the computer codes with respect to the input parameters. A very typical example is the gradient-based optimization commonly used in engineering design. Gradient-based methods use not only the value of the objective and constraint functions but also their derivatives. The success of these methods hinges on accurate and efficient evaluation of the derivatives, commonly called sensitivity analysis in the literature of engineering design. Sensitivity analysis is usually the most costly step in the optimization cycle, and the optimization process often fails due to inaccurate derivative calculations [1].

There are several methods proposed for evaluating derivatives [2] of outputs of a computer code with respect to its inputs. Traditionally, finite differences are used to evaluate the derivatives if the source codes are not accessible. Basically, one perturbs each input and evaluates the difference of the outputs. The "step-size dilemma" forces a choice between a step size small enough to minimize the truncation error and a step size large enough to avoid significant subtractive cancelation errors [3].

If the source codes are accessible, then one can use so-called automatic differentiation (AD) [2] or complex-step approximation [4] to provide a more accurate and robust evaluation of the derivatives. AD, also known as computational differentiation or algorithmic differentiation, is a well-established method based

on a 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 evaluating 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 which makes it difficult to debug the extended code. Furthermore, it is very difficult to develop the AD tool for automatic generation of the additional source codes [6]. There are more than a dozen tools that use SCT for Fortran codes, with ADIFOR [7] 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 [5] 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 F90/95/2003 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 needs to compile the operator overloaded source codes using an F90/95/2003 compiler. There are several automatic differentiation tools for Fortran codes using OO, including HSL_AD02 [8], AUTO_DERIV [9,10], and ADF95 [11]. There are certain limitations with these tools. For example, HSL_AD02 does not support the array features of F90/95/2003. AUTO_DERIV [9] was very slow, and even the most recent release in [10] is still about ten times slower than analytic derivative calculation. Analytic derivative calculation means that a Fortran code containing the analytic expressions for the derivatives is used for computation. ADF95, although it is much faster than AUTO_DERIV for the cases tested in [11], 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 end user needs to modify the code and recompile the differentiated version of the analysis code when the derivatives of different outputs are needed or the independent variables are different. This requires the end user to know the meaning of variables inside the code and to know how to compile the differentiated codes. Furthermore, the fact that oftentimes the developer of the differentiated version of the analysis code is not the end user who is going to use the code discourages the use of such AD tools. For this very reason, it is better to develop AD tools that can minimize and simplify the changes needed by the end user to the source codes of the analysis 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 cancelation 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 was found to be similar to the automatic differentiation [6], 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 the finite difference method, its easy implementation, and the fact that the end use of the 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 complex-step approximation can only be resolved using automatic differentiation [6].

SCOOT [12] is an experimental variant of OO that is specialized to support differentiation of vector calculus and derivatives of geometric variables in C++. With SCOOT, any number of derivatives are computed in parallel with machine accuracy. The method is validated to support the derivatives of geometric variables in a linear aerodynamic application.

We have developed a simple general-purpose module, namely dual number automatic differentiation (DNAD), using the arithmetic of dual numbers and the OO feature of F90/95/2003 for automatic differentiation of Fortran codes. In comparison to existing Fortran AD tools using OO, DNAD only requires the end user to specify the number of independent variables. It is simple and can be easily extended as needed. For the examples we have tested, it is much more efficient than other tools. In comparison to the complex-step approximation method popular in the aerospace design community, DNAD is more efficient and accurate, and it avoids the pitfalls associated with the complex-step approximation method as pointed out in [6].

2 Dual Number Arithmetic

Dual number arithmetic can be readily illustrated through an example. Consider the derivative of function $f(x_1, x_2) = x_1 x_2 + \sin(x_1)$. The variables x_1 and x_2 are independent variables. We wish to use dual number arithmetic to compute the value for the output $y = f(x_1, x_2)$ and the associated derivatives $\partial f/\partial x_1$ and $\partial f/\partial x_2$. A dual number is formed by two real numbers such that

$$< x_1, x_1' >= x_1 + x_1' d_1 \qquad < x_2, x_2' >= x_2 + x_2' d_2, \tag{1}$$

where x_1' and x_2' are real numbers and d_1 and d_2 are two nilpotent symbols, which are analogous to the imaginary unit i in the complex number arithmetic. We let all powers of d_1 and d_2 higher than one and their products equal zero in dual numbers, as opposed to $\mathrm{i}^2 = -1$ in complex numbers. Substituting these new numbers into the first term of the function, we have

$$\begin{aligned}(x_1 + x_1'd_1)(x_2 + x_2'd_2) &= x_1x_2 + x_1x_2'd_2 + x_2x_1'd_1 + x_1'x_2'd_1d_2 \\ &= x_1x_2 + x_1x_2'd_2 + x_2x_1'd_1.\end{aligned} \tag{2}$$

Note that here $x_1'x_2'd_1d_2$ are dropped in the final expression of the above equation, not because x_1' and x_2' are small but because d_1 and d_2 are nilpotent symbols, and thus $d_1d_2 = 0$.

We can also evaluate the dual version of $\sin(x_1)$ as follows:

$$\begin{aligned}\sin(x_1 + x_1'd_1) &= \sin(x_1) + \cos(x_1)x_1'd_1 - \frac{\sin(x_1)}{2}(x_1'd_1)^2 + \ldots \\ &= \sin(x_1) + \cos(x_1)x_1'd_1.\end{aligned} \tag{3}$$

Again, all the terms after $\cos(x_1)x_1'd_1$ are dropped, not because x_1' is small but because d_1 is a nilpotent symbol, and thus $d_1^2 = d_1^3 = \ldots = 0$.

The dual version of the function $y = f(x_1, x_2)$ can be expressed as

$$\begin{aligned}< y, y' > &= x_1x_2 + x_1x_2'd_2 + x_2x_1'd_1 + \sin(x_1) + \cos(x_1)x_1'd_1 \\ &= x_1x_2 + \sin(x_1) + [x_2 + \cos(x_1)]\, x_1'd_1 + x_1x_2'd_2 \\ &=< f(x_1, x_2), \frac{\partial f}{\partial x_1}x_1'd_1 + \frac{\partial f}{\partial x_2}x_2'd_2 > .\end{aligned} \tag{4}$$

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 they can be arbitrary. However, we choose them to be 1 so that the outputs will be the derivatives themselves. This implies that we calculate derivatives with respect to each independent input separately. In other words, the mathematical process is repeatedly run, once for every independent variable. Nevertheless, it is possible to address multiple independent variables simultaneously: we just need to replace the second component of the dual number with an array with the size the same as the number of independent variables to hold corresponding derivatives, as will be shown in the next section.

The dual number arithmetic for a general function is as follows:

$$g(< u, u' >, < v, v' >) = \langle g(u, v), g_{,u}u' + g_{,v}v' \rangle \tag{5}$$

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:

$$< u, u' > + < v, v' > = < u + v, u' + v' > \tag{6}$$

$$< u, u' > - < v, v' > = < u - v, u' - v' > \tag{7}$$

$$< u, u' > \times < v, v' > = < uv, u'v + uv' > \tag{8}$$

$$< u, u' > / < v, v' > = < \frac{uv}{v^2}, \frac{u'v - uv'}{v^2} > . \tag{9}$$

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

$$(u + u'\mathrm{i}) \times (v + v'\mathrm{i}) = (uv - u'v') + \mathrm{i}(u'v + uv') \tag{10}$$

$$(u + u'\mathrm{i})/(v + v'\mathrm{i}) = \frac{uv - u'v'}{v^2 + v'^2} + \mathrm{i}\frac{u'v - uv'}{v^2 + v'^2}. \tag{11}$$

Clearly, complex multiplication, Eq. (10), has one more multiplication operation $u'v'$ and one more minus operation $uv - u'v'$ than multiplication using dual number arithmetic in Eq (8). Complex division, Eq. (11), has two more multiplication operations $u'v', v'^2$, one more minus operation $uv - u'v'$, and one more add operation $v^2 + v'^2$ than division using dual number arithmetic in Eq (9). Multiplication and division of complex numbers will be numerically the same as that of 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, logarithmic functions, and others, the calculations of the complex-step approximation method will be more involved than with dual numbers. For example, complex arithmetic defines the arcsin function as

$$\arcsin(u + u'\mathrm{i}) = -\mathrm{i}\ \log(\mathrm{i}u - u' + \sqrt{1 + u'^2 - u^2 - 2\mathrm{i}uu'}) \tag{12}$$

while dual number arithmetic defines it as

$$\arcsin(< u, u' >) = < \arcsin(u), \frac{u'}{\sqrt{1 - u^2}} > \tag{13}$$

Certainly, the computation of the complex arithmetic will be much more involved than that of dual number arithmetic. In fact, for the arcsin function with extremely small imaginary parts, subtractive cancelation error will occur, as we shall see later. This has been realized in [6], and some of the complex functions are replaced with a definition similar to that of dual number arithmetic. Dual number automatic differentiation (DNAD) has the following advantages compared to complex-step approximation.

- DNAD is more efficient as the number of calculations of dual numbers in-

volved in computing derivatives is never more than and mostly less than that for complex numbers.
- DNAD will be more accurate as 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 in dual number arithmetic, some cancelation and truncation errors might occur.
- DNAD can deal with multiple independent variables at the same time, while complex-step approximation can only compute sensitivities with respect to one variable.

A more serious disadvantage of complex-step approximation is that it is not just as simple as replacing all real numbers with complex numbers; overloading (redefining) some intrinsic functions/operators such as ABS, conditional operators, is also needed, as the original definition in complex arithmetic is not valid for computing derivatives. It is hard to know, a priori, which functions/operations need to be redefined, and hard to identify, as compilers will not flag such functions/operations because they are perfectly legal complex operations. However, the compiler will flag every undefined operation for DNAD as dual number is a data type new to the compiler. The only disadvantage using dual numbers is that we need to overload all the operations and functions to this new data type while 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 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-step approximation does not exist. And also the development effort of DNAD is not that significant, as shown in the next section.

3 Implementation of Dual Number Automatic Differentiation

To use dual number arithmetic for automatic differentiation, we need to first define a new data type `DUAL_NUM` as follows:

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

where `DBL_AD` is an integer parameter indicating the precision used for the code, `NDV_AD` is an integer parameter indicating the number of independent variables, `x_ad_` is the function value and `xp_ad_` is the corresponding first

derivatives (the second component of the dual number data type). What one needs to do next is to overload the functions and operations needed for computing such as relational operators, arithmetic operators and functions. To define a function of a dual number, say $F(< u, u' >)$, we just need to assign the function evaluated at the first component of the input dual number, $F(u)$, to the first component of the output dual number, and the derivative of the function with respect to the independent variable u multiplying the second component of the input dual number u', $\frac{\partial F(u)}{\partial u}u'$, to the second component of the output dual number. For example, one can overload the sine function as follows:

```
INTERFACE SIN
     MODULE PROCEDURE SIN_D
END INTERFACE

ELEMENTAL FUNCTION SIN_D(u) RESULT(res)
         TYPE (DUAL_NUM), INTENT(IN)::u
         TYPE (DUAL_NUM)::res
         REAL (DBL_AD):: tmp

         res%x_ad_ = SIN(u%x_ad_)
         tmp=COS(u%x_ad_)
         res%xp_ad_= u%xp_ad_*tmp
END FUNCTION SIN_D
```

All the other functions can be defined similarly. Note that, if there are multiple independent variables, the additional effort for computing the derivatives is just a simple multiplication. The current version of DNAD has all the common relational operators, arithmetic operators and functions defined. And if it happens that some functions/operations are not defined, DNAD can be easily extended to include such definitions. For example, in the process of differentiating an F77 code for turbulence modeling, it was found that x^y with both x and y as real numbers cannot be overloaded by the original release of DNAD [13,14], as raising a dual number to a dual-number power is not defined; but we can easily insert the following segment code into the DNAD module for its definition:

```
INTERFACE OPERATOR(**)
      MODULE PROCEDURE POW_D
END INTERFACE

ELEMENTAL FUNCTION POW_D(u,v) RESULT(res)
         TYPE (DUAL_NUM), INTENT(IN)::u
         TYPE (DUAL_NUM), INTENT(IN)::v
```

```
        REAL(DBL_AD)::uf,vf
        TYPE (DUAL_NUM)::res

        uf=u%x_ad_
        vf=v%x_ad_
        res%x_ad_ =uf**vf
        res%xp_ad_=res%x_ad_*(vf/uf*u%xp_ad_+LOG(uf)*v%xp_ad_)
END FUNCTION POW_D
```

as we know that $< u, u' >^{<v,v'>} = \left\langle u^v, u^v \left(\frac{v}{u} u' + \log(u) v' \right) \right\rangle$.

4 How to Use DNAD

To use DNAD to automatically differentiate an existing Fortran code, one needs to carry out the following steps.

- Specify the number of independent variables. For example, if there are two independent variables, we need to set `INTEGER(2), PUBLIC,PARAMETER:: NDV_AD=2`.
- Replace all the declarations of real numbers in the existing code with new declarations 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.D0,0.0D0/))`.
- 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 independent variable and 0 after all other real numbers in the inputs. In the outputs, the number right after the function value indicates the derivative of this function with respect to the given independent design variable. The number of numbers inserted after the real numbers in the inputs and the number of derivatives in the outputs is `NDV_AD`.
- Recompile all the source codes and link with the DNAD module to generate the executable.

5 Examples

In this section, we are going to use a few examples to demonstrate the applications and advantages of DNAD.

The first example is to differentiate the following simple Fortran code for computing the area of a circle.

```
PROGRAM CircleArea
    REAL(8):: PI=4.0D0*ATAN(1.0D0)
    REAL(8)::radius, area

    READ(*,*) radius
    Area=PI*radius**2
    WRITE(*,*) "AREA=", Area
END PROGRAM CircleArea
```

The differentiated version using DNAD is as follows:

```
PROGRAM CircleArea
    USE Dual_Num_Auto_Diff
    TYPE (DUAL_NUM)::PI=DUAL_NUM(4.0D0*ATAN(1.0D0),(/0.D0/))
    TYPE (DUAL_NUM)::radius,area

    READ(*,*) radius
    Area=PI*radius**2
    WRITE(*,*) "AREA=",Area
END PROGRAM CircleArea
```

where all the changes happen on the second, third and fourth lines. Change `NDV_AD` to be 1, recompile the code and link with the DNAD module. One also needs to make a simple change to the inputs: add 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 the derivative of the area with respect to radius, we just need to input `5.0, 1.0`. The output will be `AREA= 78.5398163397448 31.4159265358979`, where the first output is the function value and the second is the corresponding derivative.

The second example is to evaluate the derivative of $\arcsin(x)$ at x=0.5 to demonstrate the loss of accuracy of complex-step approximation.

```
PROGRAM TestComplex
    USE complexify
    USE Dual_Num_Auto_Diff
    REAL(8)::x
    COMPLEX(8)::xc
    TYPE(DUAL_NUM)::xd

    x=0.5D0
    WRITE(*,*)ASIN(x),1.0D0/SQRT(1.0D0-x*x)
```

```
    xd=DUAL_NUM(0.5D0,(/1.0D0/))
    WRITE(*,*)ASIN(xd)

    xc=(0.5D0,1.0D-20)
    WRITE(*,*)ASIN(xc)
END PROGRAM
```

where `complexify` is the module for computing sensitivities using complex-step approximation, directly downloaded from Prof. Martins website [15]. The code is compiled using gfortran 4.7.0 with `-O3 -static` option. The outputs are

$$\begin{array}{ll} 0.52359877559829893 & 1.1547005383792517 \\ 0.52359877559829893 & 1.1547005383792517 \\ (0.52359879016876221, & 1.1547005142714115E-020) \end{array}$$

Clearly, DNAD computes the derivatives the same as the analytical expression up to machine precision, while the result of complex-step approximation is only accurate up to the seventh digit after the decimal point. The derivative of the complex-step approximation is the imaginary part of the result divided by the small imaginary part of the independent variable (1.0D-20 for this case).

The third example is used to compare the efficiency of different AD tools.

```
PROGRAM TestSpeed
    IMPLICIT NONE
    INTEGER::i
    REAL(8)::x,y,z,ftot,f
    INTEGER::nloops=50000000
    REAL:: start_time, end_time

    call CPU_TIME(start_time)

    x=1.0D0; y=2.0D0; z=3.0D0
    ftot=0.0D0

    DO i=1,nloops
       f=x*y-x*SIN(y)*LOG(i*z)
       ftot= (ftot- f)/EXP(z)
    ENDDO

    CALL CPU_TIME(end_time)

    WRITE(*,*) ftot, end_time-start_time,"seconds"
```

Table 1
Running time of the sample code differentiated by different AD tools (s)

AD Tools	1 variable (x)	2 variables (x, y)	3 variables (x, y, z)
DNAD	1.732	1.747	1.966
Complex	2.901	5.772	8.954
AUTO_DRIV	9.968	12.854	15.226
ADF95	9.750	15.475	15.553

```
END PROGRAM
```

The computing times (seconds) for the code differentiated by complex-step approximation [15], DNAD, AUTO_DERIV [10], and ADF95 [11] are reported in Table 1. The tests are performed on a Dell Latitude laptop with 2.70 GHz Intel Core i7-2620 CPU, running on a Windows 7 operating system. The code is compiled with gfortran 4.7.0 with options `-O3 -static`. To allow maximum optimization, the automatic differentiation module and the differentiated code are put into the same file for compiling. What is surprising is that the original undifferentiated code runs for 1.981 seconds, which seems impossible as the DNAD differentiated codes run even faster. Theoretically, the differentiated codes require more calculations than the original code. After consulting the gfortran developers, this mystery can be explained by the fact the compiler optimizes the differentiated codes in a different and more efficient way because the type `DUAL_NUM` is a new data type. We can clearly observe that DNAD is several times more efficient than other AD tools for this example. It is noted that DNAD, AUTO_DERIV, and ADF95 provide the same results up to double precision while complex-step approximation is slightly inaccurate. For unknown reasons, HSL_AD02 [8] cannot provide correct results for the function value and the derivatives of this code. Hence, its running time is not reported. It is also pointed out here that, although it was shown that ADF95 is more efficient than AUTO_DERIV for the examples tested in [11], the performance of both codes is similar for this example, and AUTO_DERIV is even faster than ADF95 for this problem with two independent variables. It is not clear to us why there is a large jump between one variable and two variables for ADF95. We tested the differentiated code in several different computers, and the same phenomenon was observed. This reveals that the relative efficiency of automatic differentiation tools is not only computer and compiler dependent but also problem dependent. Interested readers are recommended to test the AD tools themselves to find the best one to differentiate their own codes. The reason for this significant advantage of DNAD with respect to other tools is its simple data and code structures, which enable gfortran to compile more optimized differentiated codes.

To test the performance of the AD tools for computing derivatives with respect

to a relatively large number of independent variables, we slightly modified the previous sample code we used for obtaining results in Table 1 by changing all the input variables and function variables `x, y, z, ftot, f` to be arrays of 10, and also reduced `nloops` ten times so that the original code runs for 1.977 seconds, similar to the previous code. The code used for this test is entitled `TestSpeed2` and is shown below:

```
PROGRAM TestSpeed2
    IMPLICIT NONE
    INTEGER, PARAMETER::N=10
    INTEGER::i,j
    REAL(8)::x(N),y(N),z(N),ftot(N),f(N)
    INTEGER::nloops=5000000
    REAL:: start_time, end_time

    call CPU_TIME(start_time)

    x=1.0D0;  y=2.0D0; z=3.0D0
    ftot=0.0D0

    DO j=1,N
        DO i=1,nloops
            f(j)=x(j)*y(j)-x(j)*sin(y(j))*log(i*z(j))
            ftot(j)= (ftot(j)- f(j))/exp(z(j))
        ENDDO
    ENDDO

    CALL CPU_TIME(end_time)

    WRITE(*,*) end_time-start_time,"seconds"
    DO i=1,N
        WRITE(*,*)ftot(i)
    ENDDO
END PROGRAM TestSpeed2
```

This code can be correctly differentiated using DNAD, ADF95 [11], and complex-step approximation [15]; and their running times with respect to different numbers of independent variables are plotted in Fig. 1. Note that `3 variables` implies we use `x(1), y(1), z(1)`, `6 variables` implies we use `x(1), y(1), z(1), x(2), y(2), z(2)`, and so on We can observe from the plot that the running time of complex-step approximation is linearly proportional to the number of independent variables, and that this approach is less efficient in comparison to both DNAD and ADF95, particularly, when there is a large number of independent variables. DNAD is much more efficient than both complex-step approximation and ADF95 when the number of indepen-

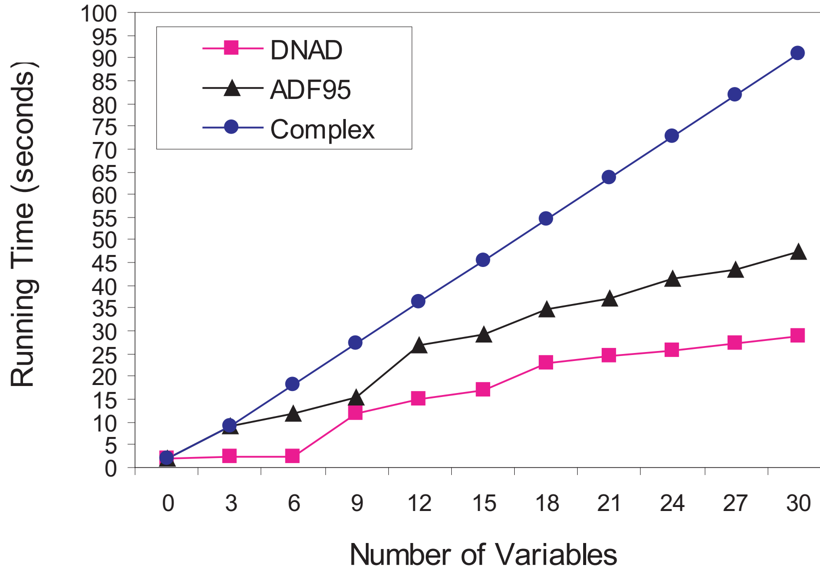

Fig. 1. Running time of a sample code differentiated by DNAD and ADF95 with respect to number of independent variables

dent variables is small, which confirms what we have observed in Table 1. Even if the number of independent variables becomes relatively large, DNAD only takes about 60% of the time ADF95 takes. Neither HSL_AD02 [8] nor AUTO_DERIV [10] can correctly differentiate this code. Hence, their running time is not available.

DNAD has been used to differentiate several engineering analysis codes including GEBT [16], a general-purpose nonlinear composite beam analysis code, VABS [17], an internationally known cross-sectional analysis code for modeling composite blades, VAMUCH [18], a general-purpose, multiphysics micromechanics code, JET [13,14], a computational fluid dynamics code accompanying a popular turbulence modeling textbook, and ZEUS [19], an industrial CFD-based aerodynamic solver. These codes are used in real-world settings, and they contain thousands of lines of F77/90/95/2003 codes along with several supporting libraries including BLAS, LAPACK, ARPACK, and HSL libraries. Very small efforts, usually a couple of hours, are needed to differentiate these codes. Very satisfactory results have been obtained as far as efficiency and accuracy are concerned.

6 Conclusion

Exploiting the operator overload feature of F90/95/2003, we have developed DNAD as a simple, general-purpose, automatic differentiation tool for Fortran

codes. It is based on dual number arithmetic and achieves the same accuracy as analytic differentiation. In comparison to existing Fortran AD tools, DNAD minimizes the changes to the source codes and is more efficient than existing tools. 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 several analysis codes used in the real world, and very satisfactory results have been obtained.

7 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

[1] J. R. R. A. Martins. *A Coupled-Adjoint Method for High-Fidelity Aero-Structural Optimization.* PhD thesis, Aerospace Engineering, Stanford University, October 2002.

[2] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, 2000.

[3] J. 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.

[4] J. 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.

[5] Automatic differentiation. Technical Report `http://en.wikipedia.org/wiki/Automatic_differentiation`, Wikipedia, 2009.

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

[7] C. 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.

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

[9] S. Stamatiadis, R. Prosmiti, and S. C. Farantos. AUTO_DERIV: Tool for automatic differentiation of a FORTRAN code. *Computer Physics Communications*, 127:343– 355, 2000.

[10] S. Stamatiadis and S. C. Farantos. AUTO_DERIV: Tool for automatic differentiation of a FORTRAN code. *Computer Physics Communications*, 181:1818–1819, 2010.

[11] C. 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.

[12] M. 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.

[13] R. E. Spall and W. Yu. Imbedded dual-number automatic differentiation for cfd sensitivity analysis. In *Proceedings of the Fluids Engineering Division Summer Conference*, San Juan, Puerto Rico, July, 08-12 2012.

[14] R. E. Spall and W. Yu. Imbedded dual-number automatic differentiation for cfd sensitivity analysis. *Journal of Fluids Engineering*, Jan. 2012. in press.

[15] J. R. R. A. Martins. Complex-step approximation. Technical Report `http://mdolab.engin.umich.edu/sites/default/files/complexify.f90.txt`, University of Michigan, 2012.

[16] Wenbin Yu 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.

[17] W. Yu, D. H. Hodges, V. V. Volovoi, and C. E. S. Cesnik. On Timoshenko-like modeling of initially curved and twisted composite beams. *International Journal of Solids and Structures*, 39(19):5101 – 5121, 2002.

[18] W. Yu and T. Tang. Variational asymptotic method for unit cell homogenization of periodically heterogeneous materials. *International Journal of Solids and Structures*, 44:3738–3755, 2007.

[19] Z. Wang, P.C.and Sarhaddi D Zhang, Z.and Chen, and W. Yu. Enabling sensitivity analysis capability for a cfd-based unsteady aerodynamic/aeroelastic solver. In *Proceedings of the 53rd Structures, Structural Dynamics, and Materials Conference*, Honolulu, Hawaii, Apr. 23-26 2010.