ADF95: Tool for automatic differentiation of a FORTRAN code designed for large numbers of independent variables

Christian W. Straka

Institut für Theoretische Astrophysik, Universität Heidelberg, Tiergartenstraße 15, 69121 Heidelberg, Germany

Abstract

ADF95 is a tool to automatically calculate numerical first derivatives for any mathematical expression as a function of user defined independent variables. Accuracy of derivatives is achieved within machine precision. ADF95 may be applied to any FORTRAN 77/90/95 conforming code and requires minimal changes by the user. It provides a new derived data type that holds the value and derivatives and applies forward differencing by overloading all FORTRAN operators and intrinsic functions. An efficient indexing technique leads to a reduced memory usage and a substantially increased performance gain over other available tools with operator overloading. This gain is especially pronounced for sparse systems with large number of independent variables. A wide class of numerical simulations, e.g., those employing implicit solvers, can profit from ADF95.

Key words: Automatic differentiation; Derivatives; FORTRAN 95; Implicit Solvers

PROGRAM SUMMARY

Nature of problem:

In many areas in the computational sciences first order partial derivatives for large and complex set of equations are needed with machine precision accuracy. For example, any implicit or semi-implicit solver requires the computation of the Jacobian matrix, which contains the first derivatives with respect to the independent variables. ADF95 is a software module to facilitate the automatic computation of the first partial derivatives of any arbitrarily complex mathematical FORTRAN expression. The program exploits the sparsity inherited

Email address: `cstraka@ita.uni-heidelberg.de` (Christian W. Straka).

by many set of equations thereby enabling faster computations compared to alternate [1] differentiation tools.

Solution method:

A class is constructed which applies the chain rule of differentiation to any FORTRAN expression, to compute the first derivatives by forward differencing. An efficient indexing technique leads to a reduced memory usage and a substantially increased performance gain when sparsity can be exploited. From a users point of view, only minimal changes to his/her original code are needed in order to compute the first derivatives of any expression in the code.

Restrictions:

Processor and memory hardware may restrict both the possible number of independent variables and the computation time.

Unusual features:

ADF95 can operate on user code that makes use of the array features introduced in FORTRAN 90. A convenient extraction subroutine for the Jacobian matrix is also provided.

Running time:

In many realistic cases, the evaluation of the first order derivatives of a mathematical expression is only six times slower compared to the evaluation of analytically derived and hard-coded expressions. The actual factor depends on the underlying set of equations for which derivatives are to be calculated, the number of independent variables, the sparsity and on the FORTRAN 95 compiler.

References:

[1] S.Stamatiadis, R.Prosmiti, S.C.Farantos, Comp. Phys. Commun. 127 (2000) 343.

LONG WRITE-UP

1   Introduction

ADF95 is a software module to facilitate the analytic computation of the *first* partial derivative of any arbitrarily complex mathematical FORTRAN expression including user defined and/or intrinsic functions and subroutines. The derivatives are computed with respect to independent variables which must be specified by the user. It must be emphasised that ADF95 does not provide the analytic derivative in functional form, rather it computes the numerical values of the analytic derivatives. ADF95 references its computed and internally stored derivatives with an indexing technique which results in reduced memory usage of sparse systems. Thereby it enables faster computations in many practical applications with large numbers of independent variables.

In many areas in the computational sciences the phenomena to be simulated can be approximated by solving systems of coupled differential equations. A quite general class of differential equations, e.g., is the following initial value problem:

$$B[\vec{y}(t), t] \cdot \vec{y}'(t) = \vec{f}[\vec{y}(t), t], \qquad \vec{y}(t_0) = \vec{y}_0, \tag{1}$$

where $\vec{y}(t)$ denotes a n-dimensional vector, $\vec{f}$ an arbitrary n-dimensional vector valued function and B a $n \times n$ matrix. $\vec{y}(t)$ is called a *solution* in the interval $I = [t_0, t_E]$ if Eq.(1) is fulfilled for all $t \in I$. Any *implicit* solution strategy requires the computation of the $n \times n$ Jacobian matrix of the residual:

$$J = \frac{\partial}{\partial \vec{y}}(\vec{f} - B \cdot \vec{y}'(t)). \tag{2}$$

Thus the Jacobian contains the first derivatives of the residual with respect to the independent vector variable $\vec{y}$. The need for an convenient albeit accurate determination of first derivatives for the class of implicit solvers has driven my motivation to develop ADF95. However, ADF95 may be useful in all instances where an automatic, efficient and to working precision accurate generation of first derivatives are needed. Only minimal changes in user code are required.

The functionality of ADF95 can only be achieved by making use of the new FORTRAN 95 (F95) features [1] that allow for object-oriented programming. By defining a new compound variable of *derived type*, and re-defining operators and functions that act on these types with the mechanism of *operator overloading* within the encapsulation mechanism provided by *modules* we

construct a class which applies the chain rule of differentiation to any FOR-TRAN expression to compute the first derivatives by *forward differencing.* All overloaded operators and functions are defined as *elemental* and can thus be called with array arguments of any rank. This is extremely useful for codes that make use of the array capabilities introduced in F90 [2] and may help compilers to vectorise or parallelise the code.

A growing number of tools exist [3] for the task of automatically computing derivatives of FORTRAN expressions. Among them, two different approaches can be distinguished. The first method operates on the source code itself generating new source code for the derivatives. Both initial and generated code must be compiled in a subsequent invocation of the compiler. The advantage of this approach lies in the production of generally faster executables for the differentiation task. It is possible to use both the forward and the reverse mode of automatic differentiation. Disadvantages of the latter are that new code must be generated for any slight change in the parent code. Furthermore it is more difficult to pass the calculated derivatives of subroutines to the calling routine. Moreover, new language features are more difficult to add to these tools.

The second method makes use of operator overloading. The disadvantages of this method are the advantages of the source code approach and vice versa. NAG [4] is working on a solution that attempts to combine the advantages of source code transformation with operator overloading by adding new compiler functionality. While potentially exciting, code portability is lost. ADF95 is conceptually similar to AUTO_DERIV [5] which, in addition, can provide second derivatives. In contrast to AUTO_DERIV no modification of code utilising array notation is needed with ADF95. The main enhancement of ADF95 over existing approaches with operator overloading is its internal, indexed storage method that allows more efficient execution in case of sparse systems with large numbers of independent variables.

2   FORTRAN 90/95 concepts

A brief summary of concepts introduced in FORTRAN by the two major revisions [1,2] and used in ADF95 is given in this section. A thorough explanation of FORTRAN language usage can be found, e.g., in the books [6,7].

The current standard allows to define new data types in addition to the built-in ones (`integer`, `real`, etc.). These *derived types* constitute aggregates of built-in and/or other derived types. For example, the following derived type

```
type vector
```

4

```
      real :: x, y
end type vector
```

defines a new data structure that may represent a 2-dimensional vector. Whereas the compiler "knows" how to perform a mathematical operation on built-in types, it cannot possibly know how to apply those to derived types. The programmer can give a meaning to an operation between derived types by, first, defining a new function, and secondly, *overloading* the operator with this function. The following code provides the functionality for adding two variables of `type(vector)` employing the rules of vector calculus

```
function vadd(v, w)
    type(vector), intent(in) :: v, w
    type(vector)             :: vadd
    vadd%x = v%x + w%x
    vadd%y = v%y + w%y
end function vadd
```

The following interface construct overloads the *plus* symbol with the `vadd` function:

```
interface operator(+)
    module procedure vadd
end interface
```

The same mechanism can be useful for *intrinsic* functions and subroutine. For example, the intrinsic function `abs()` can be overloaded to calculate the norm of the `type(vector)`

```
function norm(v)
    type(vector), intent(in) :: v
    real                     :: norm
    norm = sqrt(v%x**2 + v%y**2)
end function norm
```

Note that the return value is of type `real`. Other functions may return the type `vector`. Again, an interface is needed to overload `abs()`

```
interface abs
    module procedure norm
end interface
```

For built-in data types FORTRAN 90 is instructed to perform array arithmetic, i.e.

```
integer, dimension(1:10) :: a, b
```

5

```
b = abs(a)
```

is a compact form equivalent to writing:

```
integer, dimension(1:10) :: a, b
do i=1, 10
    b(i) = abs(a(i))
enddo
```

If we want to do the same with a derived data type or a user defined function, the function must be given the keyword `elemental` introduced in FORTRAN 95

```
elemental function norm(v)
    type(vector), intent(in) :: v
    real                     :: norm
    norm = sqrt(v%x**2 + v%y**2)
end function norm
```

This enables the following notation, making array arithmetic available to the overloaded `abs()` function:

```
type(vector), dimension(1:10) :: a, b
b = abs(a)
```

3   Usage

ADF95 constitutes a module that is written in ISO FORTRAN 95 and should be compatible with any standard conforming compiler. A new derived type is introduced in ADF95, namely `type(ADF95_dpr)`, which lays out the memory structure to hold the value and its first derivatives. All FORTRAN 95 operators and intrinsic functions are implemented for this type. The user can choose a `kind` and must specify `LDsize` which is a number less or equal the number of independent variables. Some additional user functions are provided, to specify a variable as independent, to make extraction of values and derivatives easy and to find the optimal value for `LDsize`.

3.1   A first example

Consider we would like to calculate the first derivative with respect to the independent variable x of the following FORTRAN expression

```
real :: f, x
x = 1.0
f = sin(x**2)
```

The only changes required by the user are to make the module `mod_adf95` available, change the data type `real` to `type(ADF95_dpr)` and call the routine `ADF95_independent()` to set the independent variables (second argument) and initial values (third argument):

```
use mod_adf95
type(ADF95_dpr) :: f, x
call ADF95_independent(1,x,1.0)
f = sin(x**2)
```

Note that the code containing the mathematical evaluation is not changed. This convenient property is retained also for arrays, i.e.

```
use mod_adf95
type(ADF95_dpr), dimension(1:2) :: f, x
call ADF95_independent((/1,2/),x,(/1.0,5.0/))
f = sin(x**2)
```

Each independent variable must be given a unique index. User functions to extract the value and the derivatives from the last expression are provided and discussed in detail in Section 4.1.

3.2   A second example

A more comprehensive example demonstrates the changes to be made when function and subroutine calls are involved. Extensive use of array arithmetic is made to demonstrate this capability of ADF95. Consider we would like to calculate the derivatives of the original code segment shown in Fig. 1.

As before, the module `mod_adf95` must be made available within all scopes where derivatives should be calculated. Next, `ADF95_independent()` must be called to specify the independent variables and initial values. All independent and dependent variables must be changed to `type(ADF95_dpr)`. Since the function `my_func1` may also be called in a context in which the original version is expected, it is better to add a new `module procedure` to it (Fig. 2).

It is good practice to add a new function to every existing one that may be needed for differentiation and combine them in a `module procedure`. Thus, differentiation is only performed, when it is actually needed. Purely value oriented operations will choose the matching `module procedure` thereby omit-

ting unnecessary differentiations. Even more importantly, this approach omits time consuming memory allocations that would be otherwise necessary because of overloaded function calls with the data `type(ADF95_dpr)`. Thus, adding `module procedures` can save a lot of execution time, even more so if the data structure of `type(ADF95_dpr)` is large due to many independent variables. The authors of AUTO_DERIV implemented a switch which signals when derivatives are to be calculated. However, this approach is not very efficient compared to adding new `module procedures`, mainly because of the unnecessary memory allocations described above.

3.3 Full Description

The modifications needed for an existing FORTRAN program in order to evaluate first derivatives with ADF95 are as follows:

```
module my_module
    interface my_func
        module procedure my_func1
    end interface
contains
    elemental function my_func1(x, y) result(f)
        implicit none
        real, intent(in) :: x, y
        real             :: f
        f = sqrt(abs(x**2-y**2)) + 1.0
    end function my_func1
end module

program original
    use my_module
    implicit none
    real, dimension(1:10) :: fv, gv, xv, yv
    integer :: i

    xv(1:10) = real((/(i,i=1,10)/))**2
    yv(1:10) = 1.0 / real((/(i,i=1,10)/))

    fv(1:10)   = my_func(xv(1:10),yv(1:10))**2
    gv(1)      = sum(fv(1:10))
    gv(3: 9:2) = log(fv(4:10:2)**2)
    gv(2:10:2) = exp(1.0/(fv(1: 9:2)**2))
end program original
```

Fig. 1. Code segment to be changed to allow for automatic differentiation.

In module `mod_adf95.f90`:

(1) For a first guess, the the constant parameter `LDsize` should be set to

```fortran
module my_module
    interface my_func
        module procedure my_func1, my_func1_ADF
    end interface
contains
    elemental function my_func1(x, y) result(f)
        implicit none
        real, intent(in) :: x, y
        real             :: f
        f = sqrt(abs(x**2-y**2)) + 1.0
        end function my_func1

    elemental function my_func1_ADF(x, y) result(f)
        use mod_adf95
        implicit none
        type(ADF95_dpr), intent(in) :: x, y
        type(ADF95_dpr)             :: f
        f = sqrt(abs(x**2-y**2)) + 1.0
    end function my_func1_ADF
end module

program original
    use mod_adf95
    use my_module
    implicit none
    type(ADF95_dpr), dimension(1:10) :: fv, gv, xv, yv
    integer :: i

    call ADF95_independent((/(i,i =1,10)/),xv(1:10),real((/(i,i=1,10)/))**2)
    call ADF95_independent((/(i,i=11,20)/),yv(1:10),1.0/real((/(i,i=1,10)/))**2)

    fv(1:10)   = my_func(xv(1:10),yv(1:10))**2
    gv(1)      = sum(fv(1:10))
    gv(3: 9:2) = log(fv(4:10:2)**2)
    gv(2:10:2) = exp(1.0/(fv(1: 9:2)**2))
end program original
```

Fig. 2. Modifications of code presented in Fig. 1 to allow for automatic differentiation. Adding a new module procedure can save a lot of execution time when calculation of derivatives are not needed.

the number of independent variables. The best performance is achieved with the smallest `LDsize` possible for the problem to be differentiated. `LDsize` is the maximum number of dependencies from other independent variables. In many applications, this number is much smaller than the total number of independent variables themselves. To illustrate this point further, consider the following example:

```
call ADF95_independent((/(i,i=1,10)/),x(1:10),1.0)
f(2:9) = x(3:10) - 2 * x(2:9) + x(1:8)
```

where the x_i are 10 independent variables. Since all f_i are only functions of three independent variables, i.e. $f_i = f_i(x_{i-1}, x_i, x_{i+1})$, the best choice for `LDsize` is 3. Guessing the best value for `LDsize` is almost impossible for large and complex codes. Therefore, the user function `ADF95_fillin()` is provided to inquire about the optimal value for `LDsize`.

(2) If necessary, the *kind* parameter `dpr` needs to be changed to the appropriate value for the input variables. The default is to have `dpr = KIND(1.0D0)` which is `double precision`. If the code uses single precision only, one might like to change this kind to single precision. Other kind parameters provided for mixed mode arithmetic, i.e. `spr` and `ipr`, can also be changed. Currently, ADF95 supports all expressions among variables of types `real(dpr)`, `real(spr)`, and `integer(ipr)`.

In the user's code; in all scopes where derivatives should be calculated:

(1) Make `mod_adf95` accessible through `use`. Name clashes with local entities can be avoided by renaming the few public variables. For example, `use mod, newname => oldname` imports the variable `oldname` from module `mod` under the new name `newname`. The public entities of ADF95 are `ADF95_independent`, `ADF95_value`, `ADF95_deriv`, `ADF95_fillin` and `type(ADF95_dpr)`. In addition, all operators and many F95 intrinsic functions are public.

(2) All independent and dependent, as well as any intermediate (dependent) variables must be declared as `type(ADF95_dpr)`. If the mathematical expressions are provided in functions and subroutines, it is advisable to construct an interface and add a new module procedure to the existing function or subroutine only with different input and output variables of `type(ADF95_dpr)`. For codes with many expressions, the *include* statement can be used to omit extensive code repetition. Implicit typing is permissible, but highly discouraged since it has the side-effect of declaring constants and other variables as `type(ADF95_dpr)` that are not related to the differentiation process, thereby wasting memory and execution speed.

(3) The independent variables must be declared in the parent scope of the differentiation process. This is easily done by calling the user function `ADF95_independent` which provides a method to assign an index and a value to each independent variable. All indices must be unique, the lowest index must be one and subsequent indices should not differ by more than one with respect to the previous index. However, the index order is arbitrary.

(4) After the final assignment to the dependent variable, the real value of it can be extracted by calling `ADF95_value(f)` where f is a variable of `type(ADF95_dpr)`. The first derivative of f with respect to the independent variable with index i can be extracted by calling `ADF95_deriv(f,i)`.

Following these rules, changes are neither required in the argument list of any function or subroutine nor in any statement or mathematical expression. Almost all modifications can be constrained to interfaces and the declarations of variables within the interface block and/or the module procedures.

3.4 Special Cases

Some potential difficulties may arise from old FORTRAN 77 programming style and from kind conversions.

- The use of `common` blocks and `equivalence` statements is still widespread, although their use is discouraged by the current standard and should be replaced by automatic arrays, allocatable arrays, pointers and the `transfer` statement. Passive variables, such as constants, pose no problems. However, any active variable, that is passed through a `common` block or that is `equivalence`d should be renamed and duplicated as follows:

```
real :: constant ! no problems
real :: x, y, z  ! active variables

equivalence(y,z)
common /block/ constant, x

!-------------
! use x, y, z
!-------------
```

should become

```
real :: constant    ! no problems
```

```
real :: x_, y_, z_   ! rename variables
type(ADF95_dpr) :: x, y, z

equivalence(y_,z_)
common /block/ constant, x_

call ADF95_independent(1, x, x_)
call ADF95_independent(2, y, y_)

! y and z not independent
z = y

!-------------
! use x, y, z
!-------------

! at the end of the routine
x_ = ADF95_value(x)
y_ = ADF95_value(y)
z_ = ADF95_value(z)
```

- In FORTRAN 77 the use of double precision versions of trigonometric and other mathematical functions was encouraged, i.e. `dsin(x)` was used for `double precision` types. In ADF95 only the standard conforming generic names are implemented. The user must therefore change all occurrences, for example, of `dsin(x)` to `sin(x)`.

- Type conversions from one kind to another is not permissible for variables of `type(ADF95_dpr)`, since only one type is implemented. Actually, it is not possible to construct a user defined function in FORTRAN 95 that can return values with different kinds. Thus, expressions such as `y=real(x,1.d0)` or the obsolete FORTRAN 77 expression `y=dble(x)` must be omitted.

3.5   Output Verification

To be able to test for successful compilation of ADF95 and verify the correct solution I provide a simple example in Fig. 3. Running the executable should yield the following output:

```
x array =     1.000000000000000E+00     5.000000000000000E+00
f array =     8.414709848078965E-01    -1.323517500977730E-01
 ***ADF95:
df/dx1  =     1.080604611736280E+00     0.000000000000000E+00
```

12

```
program verify_out
    use mod_adf95
    implicit none
    type(ADF95_dpr), dimension(1:2) :: f , x
    real(dpr)       , dimension(1:2) :: fp, xp

    xp = (/1.0,5.0/)
    call ADF95_independent((/1,2/),x,xp)

    f  = sin(x**2)
    fp = 2.0_dpr*xp*cos(xp**2)

    write(*,'(A,2(ES25.15))') "x array =", ADF95_value(x)
    write(*,'(A,2(ES25.15))') "f array =", ADF95_value(f)
    write(*,*) "***ADF95:"
    write(*,'(A,2(ES25.15))') "df/dx1  =", ADF95_deriv(f,1)
    write(*,'(A,2(ES25.15))') "df/dx2  =", ADF95_deriv(f,2)
    write(*,*) "***Analytic:"
    write(*,'(A,2(ES25.15))') "df/dx1  =", fp(1)  , 0.0_dpr
    write(*,'(A,2(ES25.15))') "df/dx2  =", 0.0_dpr, fp(2)
end program verify_out
```

Fig. 3. Code segment to verify correct compilation of ADF95.

```
df/dx2  =     0.000000000000000E+00     9.912028118634735E+00
 ***Analytic:
df/dx1  =     1.080604611736280E+00     0.000000000000000E+00
df/dx2  =     0.000000000000000E+00     9.912028118634735E+00
```

The last digits may vary depending on the system architecture, but outputs from ADF95 when compared to the analytic approach (last two lines of output) must be identical.

4 Implementation

ADF95 is a FORTRAN 95 module containing functions that overload all operators and all appropriate FORTRAN 90/95 intrinsic functions for the new derived data `type(ADF95_dpr)`. The data structure of `type(ADF95_dpr)` is simple: it holds one entry for the value, `LDsize` entries for the values of derivatives and `LDsize+1` values for indices:

```
type ADF95_dpr
    real   (dpr)                      :: value  = 0.0_dpr
    real   (dpr), dimension(1:LDsize) :: deriv  = 0.0_dpr
```

13

```
      integer(ipr), dimension(0:LDsize) :: index   = 0_ipr
end type ADF95_dpr
```

The entry for `index(0)` is reserved for the current number of non-zero deriva-
tives. The values for the indices correspond to the index of the independent
variable with respect to which the derivative is taken. For illustration, consider
that the variable f is a function of the independent variable x and further that
$f(x) = 1, f'(x) = 2$. The representation on `type(ADF95_dpr)` would be:


```
f%value    = 1.0  ! f(x) = 1
f%index(0) = 1    ! number of derivatives
f%index(1) = 1    ! unique index of x
f%deriv(1) = 2.0  ! f'(x) = 2
```

This indexing technique leads to compact storage of derivatives and — since
`LDsize` is in many applications much smaller than the total number of in-
dependent variables — to an economical memory use which is rewarded by
faster execution speeds.

Note that `LDsize` must be chosen before the compilation of the program and
that all variables of `type(ADF95_dpr)` allocate the same amount of memory.
Since not all of those variables actually need `LDsize` entries, memory and exe-
cution speed is wasted. Dynamic memory allocation could be used through the
`allocatable` keyword which is nowadays supported also for derived types by
many FORTRAN 95 compilers and that is part of the new FORTRAN 2003
standard [8]. However, all my actual implementations resulted in considerably
slower execution speeds in practical applications. This is probably due to the
overhead needed to decide when new memory must be allocated/deallocated
and, more likely, because of the time needed for the allocation process and for
the access to the resulting scattered memory locations. These findings may
well change with future compilers [1] and further research in this direction is
needed.

Due to the overloading of operators and intrinsic functions the compiler gen-
erates code for the evaluation of the value and the numerical derivatives
according to the chain rule of differentiation whenever operations between
`type(ADF95_dpr)` are encountered. Mixed mode arithmetic is also supported
through additional module procedures provided in ADF95. For example, with
variable `a` of type `real(dpr)` and variables `b, c` of `type(ADF95_dpr)` the com-
piler parsing the statement

$$c = a \cdot b \tag{3}$$

_____
[1] Tests were only performed with the Lahey/Fujitsu F95 compiler.

generates code for the value and its derivatives:

$$c = a \cdot b \tag{4}$$

$$\frac{\partial c}{\partial q_i} = a \, \frac{\partial b}{\partial q_i} \tag{5}$$

This is accomplished technically by the following function that overloads `operator(*)`:

```
elemental function multiply(a, b) result(f)
    use mod_precision
    implicit none
    real(dpr)       , intent(in) :: a
    type(ADF95_dpr), intent(in) :: b
    type(ADF95_dpr)             :: f
    integer(ipr)                :: lenb

    lenb    = b%index(0)
    f%value = a * b%value

    f%deriv(1:lenb) = a * b%deriv(1:lenb)
    f%index(0:lenb) =     b%index(0:lenb)
end function multiply
```

The only parameters defined in the module are the kinds of the components in `type(ADF95_dpr)`, i.e. `dpr`, and those needed for mixed mode arithmetic, i.e. `spr` and `ipr`. These parameters can be changed to extend the precisions. In order to avoid clashes in overloading resolution, `dpr` and `spr` must have different values. Currently, FORTRAN 95 does not provide a mechanism to utilise implicit promotions from one derived type to another nor does it allow to define conversions between derived types. Therefore, all procedures had to be written into supported types. Also note, that complex variables are not supported.

4.1   User functions

The mathematically independent variables must be specified at run-time, therefore ADF95 provides the user function `ADF95_independent`. The routine accepts three arguments, the variable, a value and an integer index that must be unique. This routine assigns the value and sets the derivative with respect to itself to `1.0_dpr`. Its interface is:

elemental subroutine `ADF95_independent(i,x,val)`

```fortran
    integer(ipr)    , intent(in)    :: i
    type(ADF95_dpr), intent(inout) :: x
    real(dpr)       , intent(in)    :: val
end subroutine ADF95_independent
```

Three different versions are overloaded such that **ADF95_independent** accepts values of the types `real(dpr)`, `real(spr)` and `integer(ipr)`. The value of the variable of `type(ADF95_dpr)` can be extracted by calling **ADF95_value**. Its function interface is:

```fortran
elemental function ADF95_value(x) result(f)
    type(ADF95_dpr), intent(in) :: x
    real(dpr)                    :: f
end function ADF95_value
```

Similarly, the function **ADF95_deriv** is provided to extract the derivatives. In addition to the `type(ADF95_dpr)` a second argument is expected, the index of the independent variable to which respect the derivative is taken:

```fortran
elemental function ADF95_deriv(x, i) result(df)
    type(ADF95_dpr), intent(in) :: x
    integer(ipr)   , intent(in) :: i
    real(dpr)                    :: df
end function ADF95_deriv
```

Two additional user routines are provided for convenience. The first subroutine, **ADF95_jacobian**, expects an array of `type(ADF95_dpr)` and returns three arrays containing derivatives and indices. For example, `df(k)` is the derivative of $\partial f(ir(k))/\partial x(ic(k))$. The integer return value `nz`, contains the number of non-zero entries in `df` or a negative value if the array size of `df`, `ic` or `ir` is not sufficiently large:

```fortran
pure subroutine ADF95_jacobian(f, df, ir, ic, nz)
    type(ADF95_dpr), dimension(:), intent(in)  :: f
    real(dpr)      , dimension(:), intent(out) :: df
    integer(ipr)   , dimension(:), intent(out) :: ir, ic
    integer(ipr)                 , intent(out) :: nz
end subroutine ADF95_jacobian
```

Finally, the function **ADF95_fillin** inquires about the optimal value for `LDsize`. Its input argument is the the (array of) variables of the final assignment statement. Two optional integer arguments `ml` and `mu` are returned with the number of non-zero sub-diagonals and/or super-diagonals, respectively.

```fortran
pure subroutine ADF95_fillin(f, LDsize_opt, ml, mu)
    type(ADF95_dpr), dimension(:) , intent(in)    :: f
```

```
    integer(ipr)                      , intent(out)  :: LDsize_opt
    integer(ipr)    , optional    , intent(out)  :: ml, mu
end subroutine ADF95_fillin
```

It must be stressed that `ADF95_fillin` returns only the correct number for
`LDsize` if ADF95 was compiled with a sufficiently large `LDsize` in the first
place. If a sensible initial guess for `LDsize` is not possible, `LDsize` should be
set to the total number of independent variables before compiling ADF95.
Next inquire about the best value for `LDsize` by calling `ADF95_fillin` and set
it to the inquired value. Finally re-compile ADF95.

4.2 Supported FORTRAN 90/95 intrinsics

Great care has been taken to overload all FORTRAN 90/95 intrinsic func-
tions and built-in operators for the new data `type(ADF95_dpr)` whenever
meaningful. Fully supported are the following routines including the capability
to accept and return conformable arrays: `abs`, `atan`, `cos`, `cosh`, `digits`, `dim`,
`dot_product`, `epsilon`, `exp`, `exponent`, `fraction`, `huge`, `kind`, `log`, `log10`,
`matmul`, `maxexponent`, `minexponent`, `mod`, `modulo`, `nearest`, `precision`, `radix`,
`range`, `rrspacing`, `scale`, `set_exponent`, `sign`, `sin`, `sinh`, `spacing`, `tan`,
`tanh`, `tiny`. For some others, exactly the same behaviour as for built-in func-
tions cannot be overloaded. These limitations are described in the next section.

4.3 Implementation details of `tanh`

The derivative of `tanh(x)` with respect to `x` is given by `1.0/cosh(x)**2`.
For increasing `x` the hyperbolic cosine grows beyond all limits. Thus, `cosh`
produces a floating point exception for large `x`. To circumvent this situation,
the derivative could be calculated from the mathematically equivalent form
`1.0-tanh(x)**2` as done in [5]. This formula avoids floating point exceptions
but due to finite computer precision the result is resolved to zero for relatively
small `x` rather than to a finite number.

A better implementation is chosen in ADF95. The formula `1.0/cosh(x)**2`
is used for `abs(x) < 2.0*range(x)` in which case cosh can be calculated. For
larger `abs(x)` the derivative is approximated with `4.0*exp(-2.0*abs(x))`.
Thus a finite number is returned which can be as low as the smallest number
that is representable in the current data model without being hampered by
finite precision. For even larger `x` the derivative is resolved to zero.

4.4 Limitations

The built-in intrinsic functions `aint`, `anint`, `ceiling`, `floor`, `int` and `nint` can be called with an optional *kind* parameter such that the return value has the same kind. Since FORTRAN 95 does not allow the kind of a derived type's component to be chosen when the derived type is used, this functionality cannot be implemented. However, this situation will change in the near future with the advent of FORTRAN 2003 [8].

A similar problem arises with functions that accept arrays of different rank. Since the rank cannot be chosen dynamically in user defined functions, a new module procedure must be added for all possible ranks. ADF95 accepts only arrays of rank one in these instances. The functions `maxloc`, `maxval`, `minloc`, `minval`, `product` and `sum` are affected by this restriction. For the same reason, the optional parameter `dim` is not supported for these functions.

The functions `max` and `min` accept a variable number of arguments for built-in types. This cannot be implemented either. A simple work-around to this deficiency is to change all instances in which more than two arguments are used from `max(v1, v2,..., vn)` to `max(...max(v1,v2),..., vn)`.

4.5 Undefined derivatives

Any mathematical operation between values of `type(ADF95_dpr)`, that is forbidden (e.g., division by zero) is treated exactly the same as for built-in types and produces floating point exceptions. No additional coding is needed in these instances. However, in some functions a situation can occur where the operation on the value is permissible while the derivative is not defined.

Serious problems of this kind arise in cases where the function is not mathematically differentiable. For example, the derivative of `abs(f(x))` at `f(x)=0`, `f'(x)`$\neq$`0` is not defined, likewise the derivative of `sqrt(f(x))` at `f(x)=0`. Undefined situations as such can occur in the functions `acos`, `asin`, `atan2`, `max`, `maxval`, `min`, `minval` and `sqrt`. Divisions by zero return `Inf` (Infinity) or, depending on the compiler options, a floating point exception. In all other cases ADF95 is instructed to return `-sqrt(-1.0)` which yields, depending on the system and compiler options, either `NaN` (Not a Number) or a again a floating point exception. Note that computing the analytic derivatives by other means would lead to the same undefined situations.

In AUTO_DERIV, these occurrences are arbitrarily resolved to zero which is mathematically incorrect. The approach of ADF95 has the advantage that the user is being notified that an illegal mathematical operation has been

performed, pointing him to the location where his code needs rethinking.

5 Tests

5.1 Verifying the solution

In order to test the correctness of the solution calculated with ADF95, numerous comparisons between ADF95 and AUTO_DERIV including *all* overloaded operators and functions (as well as combinations among them) were performed. These comparisons revealed an error in the function `fraction` of AUTO_DERIV: the return value must be of type `real` and not of type `integer`.

The results of all other operations and functions turned out to be *identical*. Since both modules were developed completely independently this result is a strong indication for the correctness of both packages. Nevertheless, it is almost impossible to cover and compare all possible code branches of both routines, therefore all tests are inherently incomplete.

As expected, different results were encountered with `tanh` and in situations in which undefined derivatives occur. Those are set to zero in AUTO_DERIV whereas ADF95 sets them to `NaN` (see Section 4.5).

5.2 Performance and Compiler Comparison

A number of tests have been performed in order to measure the efficiency of ADF95 in comparison to AUTO_DERIV and in comparison to analytically computed and hard-coded derivatives. Five up-to-date FORTRAN compilers for *Linux* must demonstrate their efficiency: *Absoft*, *Intel*, *Lahey/Fujitsu*, *NAG* and *PGI*. The compiler options have been chosen to give maximum execution performance (Table 1). All tests were performed on a Mobile Intel(R) Pentium(R) 4 processor at 2.5 GHz, 1 GB of memory, running on a *Linux/RedHat 8* operating system.

As a first example in [5], the performance of AUTO_DERIV is benchmarked by calculating the derivatives of the Potential Energy Surface (PES) for the HCP molecule, described in [9]. The code for the calculation of the PES is available from [10]. This is a realistic example, but only with three independent variables and its main purpose is to test ADF95 with the exact same piece of code on which AUTO_DERIV was tested. The PES code is simple enough that the calculation of first derivatives "by hand" is still feasible and I have

19

Table 1
FORTRAN 95 compilers and compiler options used for test runs.

Compiler	Options
Absoft Pro FORTRAN Version 8.2a	–O3 –cpu:p7
Intel(R) Fortran Compiler for 32-bit applications, Version 8.0 (Package ID: l_fc_pc_8.0.046)	–O3 –ipo –static [a]
Lahey/Fujitsu Fortran 95 Compiler Release L6.20b	–O –tp4 –trap –staticlink
NAGWare Fortran 95 compiler Release 5.0(322)	–O4 –Bstatic –unsharedf95
Portland Group, Inc. pgf90 5.1-6 [b]	–fast –tp piv

[a] Omitted in third example: generated code causes segmentation fault.
[b] Note: pgf90 is not a FORTRAN 95 compiler.

done this in order to allow comparison with the automatic differentiation approach. Table 2 summarises the results of the HCP example for different methods and compilers. The variable `LDsize` had to be set to 3 in ADF95. The time was measured with the FORTRAN routine `system_clock`.

For three out of five compilers ADF95 is only a factor of 3–6 slower compared to the direct analytic computation. Furthermore, ADF95 is about a factor of four faster than AUTO_DERIV regardless of the compiler chosen. This is quite surprising, because the advantages of the indexing method do not show up in systems where `LDsize` is small or where it is equal to the number of independent variables. One reason might stem from the additional memory that must be allocated in AUTO_DERIV to hold the second derivatives. Extensive use of function calls in AUTO_DERIV may also produce additional

Table 2
Derivatives of the Potential Energy Surface of a HCP molecule. Time averaged over 10^6 evaluations and quoted in μs.

Compiler	Execution time [μs]			Memory usage [kBytes]		
	analytic	ADF95	AUTO_DERIV	analytic	ADF95	AUTO_DERIV
Absoft	3.6	32	108			
Intel	3.7	11	42			
Lahey/Fujitsu	1.5	19	93	2.0	3.3	5.5
NAG	5.3	30	159			
PGI	7.3	19 [a]	714			

[a] ADF95 was stripped of its FORTRAN 95 features in order to make it run with the PGI compiler.

Table 3
Derivatives for nuclear reaction network example. The simulation consists of 10 shells with 14 nuclei each which amounts to 140 independent variables.

Compiler	Execution time [s]			Memory usage [kBytes]		
	analytic	ADF95	AUTO_DERIV	analytic	ADF95	AUTO_DERIV
Absoft	1.6	7.3 [a]	5090			
Intel	1.0	5.7	103	48	172	74800
Lahey/Fujitsu	1.2	7.0	2530			
NAG	1.4	7.1	1139			
PGI [b]	—	—	—			

[a] The Absoft compiler does not behave conforming to ISO FORTRAN 95 . ADF95 had to be altered to make it run with this compiler.
[b] Excluded from test since PGI does not support FORTRAN 95.

overhead, unless the compiler is capable of inlining code properly.

My second example is one from astrophysics. A nuclear fusion network with 14 nuclei is operating within every 10 different temperature/density shells. This corresponds to 140 independent variables altogether, but since only the network is coupled, `LDsize` can be set to 14. Thus this example exploits the advantages of ADF95 as can be seen in Table 3.

Depending on the compiler AUTO_DERIV is a factor of 20–700 slower and uses 400 times more memory than ADF95. Also note that the performance of AUTO_DERIV is extremely compiler dependent whereas ADF95 is about a factor of 6 slower compared to the analytic computation of derivatives regardless of the compiler.

Finally I use the exact same application again, but now with 1000 temperature/density shells which amounts to 14000 independent variables. This problem cannot be handled with AUTO_DERIV any more (Table 4). It can be seen, that as long as `LDsize` is not changed, the execution time scales simply with the number of derivatives to be calculated. Again, computation of hardwired analytic derivatives are by a factor of 6 faster. The memory requirement of ADF95 can be easily calculated: A variable of `type(ADF95_dpr)` holds $(1 + \mathtt{LDsize}) * (\mathtt{real(dpr)} + \mathtt{integer(ipr)})$ numbers. Taking default parameters for `dpr` and `ipr` this amounts to $12 \cdot (1 + \mathtt{LDsize})$ bytes multiplied by the number of `type(ADF95_dpr)` variables in the program.

Table 4
Derivatives for nuclear reaction network example consisting of 1000 shells with 14 nuclei each which amounts to 14000 independent variables.

Compiler	Execution time [s]		Memory usage [kBytes]	
	analytic	ADF95	analytic	ADF95
Absoft	159	715 [a]		
Intel	107	566		
Lahey/Fujitsu	109	575	3838	7643
NAG	131	706		
PGI [b]	—	—		

[a] The Absoft compiler does not behave conforming to ISO FORTRAN 95. ADF95 had to be altered to make it run with this compiler.
[b] Excluded from test since PGI does not support FORTRAN 95.

6 Discussion

If there is need for numerical first derivatives, accurate to machine precision, which is the case, e.g., for implicit solvers employed for simulations in all computational sciences, the use of ADF95 should be seriously considered. As demonstrated for realistic examples in this paper, this method is still about a factor of 6 slower compared to the method of hard-wiring the analytically derived first derivatives. Thus, if maximum performance is demanded, ADF95 should be employed only if the part for calculating first derivatives is not limiting the performance of the entire program. The latter situation in which the differentiation part is not crucial to the overall program performance does indeed occur in state-of-the-art implicit solvers [11] and little compromise has to be made when employing ADF95.

Apart from these performance considerations, ADF95 can reduce code development considerably. In the case of large systems the analytic differentiation combined with the need of extra coding is an error-prone task which easily introduces difficult to find bugs into the program thereby slowing down the development process. Furthermore, successfully implemented systems are difficult to change, since it usually requires to alter many equations for calculating derivatives. Even if one insists on this approach in view of its performance benefits, ADF95 can be a convenient tool to find bugs or verify the solution for calculating derivatives more quickly. It can be also used to inquire about the structure of the Jacobian matrix and also to search for non-differentiable situations within the coded systems of equations which can lead to the detection of spurious convergence problems.

The disproportionality in performance between the hard-wiring approach and

ADF95 may well be reduced with better compiler technology in the future. Although FORTRAN 95 has been standardised seven years ago, many compilers are still lacking reliable support for it. *PGI* does not provide FORTRAN 95 and the one from *Absoft* is extremely buggy on the new features. To a substantially lesser degree, this finding is also true for the *Intel* compiler. Throughout the whole study the *Intel* compiler produces the fastest executables but best support for ISO FORTRAN 95 is provided by the compilers from *NAG* and *Lahey/Fujitsu*. The latter offers the best compromise between stable language support and execution speed. It can be suspected that there is still room for compiler optimisations when FORTRAN 95 constructs are involved.

On the other hand, the design of ADF95 may also be improved upon. The approach of allocating memory statically leads to some waste of memory. Dynamic memory allocation might improve on the performance of ADF95 but my first tests on this showed, unfortunately, the opposite effect. Also, the default initialisation of all entries within `type(ADF95_dpr)` is not needed but the code for the overloaded functions would have been more complicated otherwise.

With ADF95 an easy to use automatic differentiation tool is now available efficient enough worth being employed in many realistic applications.

Acknowledgements

References

[1] International Organization for Standardization, ISO/IEC 1539-1:1997, Information technology-Programming languages-Fortran, 1997.

[2] International Organization for Standardization, ISO/IEC 1539-1:1991, Information technology-Programming languages-Fortran, 1991.

[3] http://www.autodiff.org.

[4] http://www.nag.co.uk/nagware/research/ad_overview.asp.

[5] S. Stamatiadis, R. Prosmiti, S.C. Farantos, Comp. Phys. Commun. 127 (2000) 343.

[6] J.C. Adams, W.S. Brainerd, J.T. Martin, B.T. Smith, J.L. Wagener, Fortran 95 Handbook — Complete ISO/ANSI Reference (The MIT Press, Cambridge, Massachusetts, 1997).

[7] M. Metcalf, J. Reid, Fortran 90/95 explained (Oxford University Press, Oxford, 2nd edn., 2000).

[8] International Organization for Standardization, ISO/IEC 1539-1:2004(E), Information technology-Programming languages-Fortran, 2003.

[9] C. Beck, H.-M. Keller, S.Yu. Grebenshchikov, R. Schinke, S.C. Farantos, K. Yamashita, K. Morokuma, J. Chem. Phys., 107 (1997) 9818.

[10] S.C. Farantos, H.-M. Keller, R. Schinke, K. Yamashita, K. Morokuma, J. Chem. Phys., 104 (1996) 10055, Software Archive: E-JCPSA-104-10055-0.005MB.

[11] R. Ehrig, U. Nowak, LIMEX Version 4.3B, Konrad-Zuse-Zentrum für Informationstechnik (ZIB), Berlin, 2003, http://www.zib.de/ehrig/software.html