



AUTO_DERIV: Tool for automatic differentiation of a FORTRAN code

S. Stamatiadis¹, R. Prosmi, S.C. Farantos²

Department of Chemistry, University of Crete, Iraklion, Crete 711 10, Greece

Received 1 October 1999

Abstract

AUTO_DERIV is a module comprised of a set of FORTRAN 90 procedures which can be used to calculate the first and second partial derivatives of any continuous function with many independent variables. The function should be expressed as one or more FORTRAN 90 or FORTRAN 77 procedures. A new *type* of variables is defined and the *overloading* mechanism of functions and operators provided by the FORTRAN 90 language is extensively used to define the differentiation rules. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Automatic differentiation; Derivatives; FORTRAN 90; Potential energy surfaces; Molecular dynamics; Non linear mechanics

PACS: 02.70; 34.20.Gj; 34.20.Mq; 82.20.Kh; 45.10.Db; 05.45.Pq

PROGRAM SUMMARY

Title of program: AUTO_DERIV

Catalogue identifier: ADLS

Program Summary URL: <http://cpc.cs.qub.ac.uk/summaries/ADLS>

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland

Computer for which the program is designed: Any computer equipped with an ISO FORTRAN 90 conforming compiler

Computers on which it has been tested: Intel Pentium II PC, IBM RS/6000, HP Exemplar

Operating systems under which the program has been tested: GNU/Linux, AIX 4.3, HP-UX B.10.01

Programming language used: ISO FORTRAN 90

Compilers with which it has been tested: HP f90, NAGWare f95, IBM xlf90, Fujitsu F95, Absoft Pro Fortran F90, PGI Workstation pgf90

Memory required to execute with typical data: 680 KBytes

No. of bits in a word: 32

No. of processors used: 1

Has the code been vectorized or parallelized: No

No. of bytes in distributed program, including test data, etc.: 68 402

¹ Also at Department of Physics, University of Crete, Iraklion, Crete, GR-710 03, Greece.

² E-mail: farantos@iesl.forth.gr. Also at Institute of Electronic Structure and Laser, FORTH, Iraklion, Crete, 711 10, Greece.

Distribution format: ASCII

Keywords: Automatic differentiation, derivatives, FORTRAN 90, potential energy surfaces, Molecular Dynamics, non linear mechanics

Nature of physical problem

Complicated analytical functions of many independent variables often appear in molecular simulations. Particularly, in Molecular Dynamics the first and occasionally the second partial derivatives of the potential function are required which AUTO_DERIV evaluates to machine precision.

Method of solution

The mathematical rules for differentiation of sums, products, quotients, elementary functions in conjunction with the chain rule for compound functions are applied. The function should be expressed as one or more FORTRAN 90 or FORTRAN 77 procedures. A new *type* of variables is defined and the *overloading* mechanism of functions and operators provided by the FORTRAN 90 language is extensively used to implement the differentiation rules.

Restrictions on the complexity of the problem

None imposed by the program. There are certain limitations that may appear mostly due to the specific implementation chosen in the user code. They can always be overcome by recoding parts of the routines developed by the user or by modifying AUTO_DERIV according to specific instructions given below. The common restrictions of available memory and the capabilities of the compiler hold.

Typical running time

The typical running time for the program depends on the compiler and the complexity of the differentiated function. A rough estimate is that AUTO_DERIV is an order of magnitude slower than the evaluation of the analytical function value and derivatives (if they were available).

Unusual features of the program

None.

LONG WRITE-UP

1. Problem

AUTO_DERIV is a software module facilitating the analytical computation of the first and second partial derivatives of, in principle, an arbitrarily complex,³ continuous mathematical function of any number of independent variables. The mathematical function should be expressed as one or more FORTRAN procedures. It should be clarified that the package computes the numerical values of the derivatives at any given set of the independent variables and it does not deliver their analytical expressions.

Our motivation for developing AUTO_DERIV was the need to integrate and analyze classical trajectories in Molecular Dynamics simulations [1] by using realistic potential functions [2]. This requires the solution of Hamilton equations, and thus, the computation of the gradient of the Hamiltonian function, $\mathcal{H}(q_i, p_i)$, of the problem. q_i are the generalized coordinates and p_i their conjugate momenta for a system of n degrees of freedom. In mechanical systems, \mathcal{H} is comprised of the kinetic energy, T – having a straightforward dependence on p_i , and possibly, q_i – and the potential energy V , which, as usual in our field, is a function of q_i fitted to *ab initio* electronic structure calculations and/or empirical data, thus, lacking analytic expressions for the derivatives.

Sometimes, the location of a special type of trajectories, such as periodic orbits, and the computation of their stability properties (Lyapunov exponents), require the integration of the variational equations, and thus, the Hessian of the Hamiltonian function [3]. Hence, the need for an easy and accurate way of computing the first and second partial derivatives of multidimensional functions is a prerequisite in Molecular Dynamics simulations. Naturally, the need of accurate to machine precision derivatives instead of approximate ones arises in many fields of chemistry [4], mathematics, physics and engineering.

The potential functions and, therefore the Hamiltonians we use, are coded in FORTRAN and, thus, the language we chose to implement this tool was the one described in [5,6], commonly known as FORTRAN 90 (F90). By this standard, certain features and language constructs were introduced in FORTRAN, providing, among others,

³ Depending on the available memory and the capabilities of the compiler.

primitive support for *value-oriented programming*. Using *derived types*, function and operator *overloading*, and the encapsulation mechanism of *modules*, we constructed a *concrete class* which “instructs” the compiler how to apply the chain rule of differentiation to any mathematical function comprised of elementary, intrinsic FORTRAN 90 functions and to interpret appropriately all operators in order to compute the derivatives.

Commercial and Open Source symbolic algebra packages exist, which can evaluate derivatives of any order of relatively straightforward functions, or, rather, complex expressions of elementary functions and, even, encode them in a variety of programming languages [7–9]. They cannot, however – to the best of our knowledge – input a complex FORTRAN 90, or, even a FORTRAN 77, piece of code implementing a mathematical function and compute the derivatives; at least, we are not aware of any publicly available software that does not require translation from FORTRAN to another programming language and a significant amount of modifications to the user code in order to accomplish the differentiation (with the partial exception of ADIFOR).

ADIFOR [10] is a, free for educational and non-profit research, tool for automatic differentiation, which, according to the user guide: “accepts Fortran 77 source code (with a few common extensions) and the user’s specification of dependent and independent variables, and generates an augmented derivative code that computes the partial derivatives of all of the specified dependent variables with respect to all of the specified independent variables in addition to the original result”. It can also report arithmetic exceptions. The main drawbacks of this package are that it does not accept F90/95 codes and that it computes only first derivatives making cumbersome the derivation of higher ones.

2. Certain FORTRAN 90 concepts

A brief presentation of the concepts introduced in FORTRAN by the major revision of [5] and used by AUTO_DERIV, will be given in this section. For a detailed explanation of them the reader can consult any book on FORTRAN 90 [11].

By the new standard, the programmer can use not only the original data types (REAL, INTEGER, etc.) but also, can define and use aggregates of them. These structures can in turn serve as the building blocks of more complex entities. Such a *derived type* is defined through the keyword, TYPE. For example:

```
TYPE vector
  REAL :: x, y, z
END TYPE vector
```

defines a structure holding three real numbers (which can be the coordinates of a vector in 3D space).

The user can declare the type of a variable to be a derived type (for example, TYPE (vector) :: a, b, c), address its components via % (for example, a%x = 1.0) and pass them as arguments to a subroutine, effectively packing related information in one variable.

The programmer can also define operations involving them. For example, the cross product of two *vectors* can be computed via the * operator, as in c = a * b, provided that a suitable definition for the intended action of this operator is made known to the compiler. This is done by first defining a function in the same module as TYPE (vector) being something like:

```
FUNCTION cross(a, b)
  TYPE (vector), INTENT (in) :: a, b
  TYPE (vector) :: cross

  cross%x = a%y * b%z - a%z * b%y
  cross%y = a%z * b%x - a%x * b%z
  cross%z = a%x * b%y - a%y * b%x
END FUNCTION cross
```

An interface should exist to *overload* the operator `*`

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

In addition to operator overloading, FORTRAN 90 supports function overloading; one could use the notation `ABS(a)`, to compute the absolute value (norm) of a *vector* `a`. This requires a function

```
FUNCTION absvec(a)
  TYPE (vector), INTENT (in) :: a
  REAL :: absvec

  absvec = sqrt(a%x**2 + a%y**2 + a%z**2)
END FUNCTION absvec
```

and an interface declaration to overload the generic name `ABS` with a new function

```
INTERFACE ABS
  MODULE PROCEDURE absvec
END INTERFACE
```

Given these declarations, the same function (`ABS`) applied to a `REAL` returns the absolute value, and applied to a *vector* returns its norm.

The programmer can define functions accepting *vectors* and returning a *vector*; for example, rotation can be performed through such a function.

The access attribute (`PRIVATE` or `PUBLIC`) can be specified for an entity in a module. The former indicates that the entity can be used only inside the module, while the latter attribute “exports” it to any program unit which uses the module.

3. Usage

`AUTO_DERIV` is written in strict ISO FORTRAN 90 and should be accepted by any F90 conforming compiler. It is encoded in the form of a Fortran *module*, which, when used, provides the definition of a derived type, `TYPE (func)`, which lays out the memory for the value, the first and the second partial derivatives of any variable declared as such. Functions for the manipulation and extraction of the numerical values are also available. The user should define the kind, the number and the values of the independent variables and the order (first or/and second) of the derivatives required.

3.1. Example

An example of how to use `AUTO_DERIV` is the following:

Let us assume that we want to evaluate the first and (possibly) the second derivatives of the mathematical function f with respect to three variables x , y , and z encoded in the subroutine in Fig. 1. Note that it is not necessary to use a function or, even, only one procedure, as we will see.

The modifications which are needed in this subroutine in order to evaluate the derivatives are the following:

We should augment the argument list with additional variables to hold the first and (if required) the second derivatives and, also, we should specify, either in the procedure or through an argument, the order of the derivatives we want. In order to use the same subroutine for both first and second order differentiation let us pass an integer

```

SUBROUTINE a(x, y, z, f)
  IMPLICIT NONE

  INTEGER, PARAMETER :: dpk = KIND(1.d0) ! double precision kind
  REAL (dpk), INTENT (in)  :: x, y, z
  REAL (dpk), INTENT (out) :: f

  REAL (dpk) :: g

  g = y**2 * EXP(z)
  f = SIN(x) * g

END SUBROUTINE a

```

Fig. 1. Routine implementing a test function to be differentiated.

(*deriv*); different values of it will choose different order. Another way to do this is to use the facilities of the optional and keyword arguments of F90: over-simplifying, we can pass only those variables we need to be filled. The former approach enables us to call the subroutine from a F77 code or other programming languages lacking the concept of optional arguments; in the example we will follow this method.

There is no requirement imposed by AUTO_DERIV for the type, the number or, even, the existence of these additional variables. For simplicity we will pass a rank-1 array (Df) with n real elements to hold the first derivatives, and another rank-1 array (DDf) with $n(n + 1)/2$ elements which will hold the upper triangle of the Hessian matrix (H), where n is the number of independent variables. In our example $n = 3$ so, in F90 notation the array DDf is:

$$DDf = (/H_{11}, H_{12}, H_{13}, H_{22}, H_{23}, H_{33}/).$$

Additionally, we will indicate with *deriv* = 1 that we need only the first derivatives and with *deriv* = 2 that we require *both* first and second derivatives.

The definition of the subroutine incorporating these changes and the necessary calls from *deriv_class* are presented in Fig. 2.

In order to preserve the same form of the statements in the main part of the subroutine, we reserve the names x, y, z, f for the new variables of TYPE (func), and we use other names, ($x_, y_, z_, f_$) for their values.

The programmer should make the following modifications before applying AUTO_DERIV:

In module *deriv_class*:

- (1) Assign the number of independent variables to the variable n .
- (2) Change, if necessary, the *kind* variable dpk to the appropriate value for the input variables. The default is to have double precision. If required, the other *kind* variables spk and ik , provided to cope with mixed mode arithmetic, can be changed. Currently AUTO_DERIV supports all expressions among variables of the types REAL (dpk), REAL (spk), and INTEGER (ik).

In the user's subroutine or function:

- (1) Change the names of the input and output variables in the argument list and declare them as REAL having the same kind as in *deriv_class*. Their previous names should be used for the variables declared as TYPE (func).
- (2) Make *deriv_class* accessible through USE. If necessary, rename the few public variables provided by *deriv_class* to avoid name clashes with local entities. For example, USE mod, newname => oldname imports the variable oldname from module mod, but with the name newname. The public entities of *deriv_class* are: the subroutines *derivative*, *independent*, *extract*, the type (func), and, of course, all operators and many intrinsic F90 functions.

```

SUBROUTINE a(x_, y_, z_, f_, Df, DDf, deriv)
  USE deriv_class      ! make the module accessible
  IMPLICIT NONE

  INTEGER, PARAMETER :: dpk = KIND(1.d0) ! double precision kind

  REAL (dpk), INTENT (in)   :: x_, y_, z_
  REAL (dpk), INTENT (out)  :: f_
  REAL (dpk), INTENT (out)  :: Df(3), DDf(3 * (3 + 1) / 2)
  INTEGER, INTENT (in)     :: deriv

  TYPE (func) :: x, y, z, f

  TYPE (func) :: g

  CALL derivative(deriv) ! declare the order of the derivative.

  ! declare as independent the variables (x, y, z)
  ! and assign them their values (x_, y_, z_)

  CALL independent(1, x, x_)
  CALL independent(2, y, y_)
  CALL independent(3, z, z_)

  g = y**2 * EXP(z)
  f = SIN(x) * g

  ! from f extract value and derivatives.

  CALL extract(f, f_, Df, DDf)

END SUBROUTINE a

```

Fig. 2. Modification of the routine presented in Fig. 1 for computing the first and second derivatives of function f by incorporating *deriv_class*.

- (3) Declare as TYPE (func) all variables corresponding to mathematical functions; that is, both dependent and independent, and also, all intermediate (dependent) variables. Note, that the use of IMPLICIT typing is permissible; this practice, however, has the side-effect of declaring constants and other variables not related to the differentiation as variables in the mathematical sense. This is not wrong as their derivatives are zeroed, it is only a waste of memory and triggers unnecessary computations. It is regarded as “good programming style” to avoid implicit declarations.
- (4) Define the order of derivatives by calling the subroutine *derivative*. The integer argument passed (*deriv*) should have the value of 0, 1, or 2 indicating that we require the computation of the value of the function (included for symmetry and testing), the value and the first derivatives or the value, the first and the second derivatives respectively.
- (5) Declare the independent variables, their order and value, by calling the subroutine *independent* separately for each variable.

(6) After the final assignment to the dependent variable, extract the derivatives and the value of the function. The subroutine *extract*, will return the derivatives in rank-1 arrays holding the gradient and the upper triangle of the Hessian.

Note, that the statements in the main body of the subroutine need not be altered. In fact, AUTO_DERIV was designed in such a way that almost no modification of the existing code is required.

3.2. Special cases

There are certain cases which the potential user should bear in mind:

- If the subroutine or function calls other subroutines or functions, their definition should be changed as follows:
 - (i) They must USE the module *deriv_class*.
 - (ii) All dependent and independent variables have to be declared as TYPE (func).

No change is required in the argument list.
- Use of COMMON blocks and EQUIVALENCE statements, although discouraged by the introduction of modules and pointers in FORTRAN 90, is widespread. Transferred constants pose no problem. On the other hand, the awkward programming style of passing input and returning results from a subroutine or function through COMMON blocks or using EQUIVALENCED variables, requires the user to treat them as if they were declared in the argument list; that is, their names must be changed, and other variables should be declared, as described above. The subroutine *independent* should be called for the input variables and their values should be extracted and update the COMMON blocks at the end. For example,

```
REAL :: constant
REAL :: x           ! input variable

COMMON/block/constant, x
.....
! use x
.....
```

should become

```
REAL :: constant
REAL :: x_         ! input variable
TYPE (func) :: x

COMMON/block/constant, x_

CALL independent(i, x, x_)      ! i : unique index
.....
! use x
.....

! at the end of the routine
CALL extract(x, x_) ! assign to x_ the expected value
```

- The user must ensure, and modify the subroutine or function if necessary, that the generic names of trigonometric and other mathematical functions are used; that is, for example, SIN(x) is used instead of DSIN(x) irrespectively of the type of x, as encouraged by FORTRAN 77. Also, the user should eliminate all transformations from one kind to another between variables of TYPE (func). That is, statements such as $y = \text{DBLE}(x)$ works for x, y

if they are real (with the appropriate kinds), but as *deriv_class* provides a TYPE (func) with only one kind, such transformations have no meaning. Likewise, assignments of a TYPE (func) to a real are not supported.

4. Implementation

deriv_class is a collection of functions overloading all operators and all appropriate FORTRAN 90 functions and subroutines to accept not only real values but also variables of a derived type (TYPE (func)) comprised of the value, the first and the second derivatives of the corresponding mathematical quantity.

The compiler is “taught” how to handle expressions involving variables of this type – addition, subtraction, multiplication, etc. between them – and, also, how to apply all meaningful FORTRAN 90 functions and subroutines on them in order to compute not only the value of the expression but, in addition, the numerical value of the first and second derivatives. *deriv_class* also provides functions to interpret mixed mode expressions between variables of TYPE (func), reals, and integers. Using this module, the compiler is able to apply the usual rules of differentiation (including the chain rule for cumbersome expressions) on any statement. That is, the compiler parsing a statement involving the TYPE (func) variables *a*, *b*, *c*, *d* like

$$d = a * b + c$$

will generate a code to evaluate the following mathematical expressions:

$$d = ab + c,$$

$$\frac{\partial d}{\partial q_i} = \frac{\partial a}{\partial q_i} b + a \frac{\partial b}{\partial q_i} + \frac{\partial c}{\partial q_i},$$

$$\frac{\partial^2 d}{\partial q_i \partial q_j} = \frac{\partial^2 a}{\partial q_i \partial q_j} b + \frac{\partial a}{\partial q_i} \frac{\partial b}{\partial q_j} + \frac{\partial a}{\partial q_j} \frac{\partial b}{\partial q_i} + a \frac{\partial^2 b}{\partial q_i \partial q_j} + \frac{\partial^2 c}{\partial q_i \partial q_j}.$$

Allocatable components of a *derived type* are not yet part of FORTRAN. It is necessary, therefore, to define prior to compiling, the number of independent variables (in the mathematical sense); *n* is the integer which holds it.

The components of a TYPE (func) variable are a REAL variable for the value (*value*) of the function, and two rank-1 arrays of *n* and $n(n+1)/2$ REAL elements (*x*, *xx*), for the first and second derivatives, respectively. The arrays hold the gradient and the upper triangle of the Hessian; it is stored in the format:

$$xx(i + n(j - 1) - j(j - 1)/2) = H_{ji} \quad (i \geq j). \quad (1)$$

A few parameters are defined in the module; they are the kind of the components of TYPE (func), (*dpr*), and the kind for the reals (*spk*) and the integers (*ik*) that can appear in the same expression. Currently, FORTRAN provides no mechanism to utilize implicit promotions in expressions involving derived types or define conversions of one derived type to another. It was necessary, therefore, to write all procedures into supported types.

The default values for the kinds yield double precision, default real, and default integer numbers. They can be tailored to extend the precision. Note, however, that *dpr*, and *spk* must correspond to different precisions to avoid clashes in overloading resolution. The FORTRAN 90 standard ensures that at least two different kinds of reals are provided by the compiler. Multiple definitions of these kinds can be eliminated by modifying the module to inherit the kinds from another or export them outside *deriv_class* by changing the access attribute. By default, these and all “internal” variables and functions are of PRIVATE access.

Note, that *complex variables are not supported*.

There is a number of variables to be specified at run-time, before the computation is enabled. These are the required order of differentiation (*drvrv*) and the independent variables. Two subroutines are provided, *derivative* and *independent*, to manipulate them from the user’s routine.

The subroutine *derivative* accepts an integer argument and assigns it to *drviv*. It should be either 0, 1 or 2. All derivatives with order less or equal to this number will be evaluated (considering the value of the function as the zeroth derivative). An argument different than 0, 1 or 2 results in computing only the value. The interface is:

```
SUBROUTINE derivative(der)
  INTEGER, INTENT (in) :: der
END SUBROUTINE derivative
```

The subroutine *independent* is used for declaring a TYPE (func) variable as mathematically independent. The routine accepts three arguments, the variable, a REAL (*dpk*) value and an integer *i*. It zeroes the derivatives of the independent variable except the *i*th component of the first derivative, which is set to unity. It also assigns to the appropriate component of the TYPE (func) variable the supplied real value. This routine should be repeatedly called for all independent variables. Its interface is:

```
SUBROUTINE independent(i, x, val)
  INTEGER, INTENT (in) :: i
  TYPE (func), INTENT (out) :: x
  REAL (dpk), INTENT (in) :: val
END SUBROUTINE independent
```

The module *deriv_class* provides the subroutine *extract* to “decode” a TYPE (func) variable. It breaks it up into a real number (for the value), and two optional rank-1 arrays for the derivatives. There is a number of routines to extract them separately (under the generic names *value*, *FD*, *SD*) but normally they need not be used; therefore, they are private to the module. The interface of *extract* is:

```
SUBROUTINE extract(x, val, Dx, DDx)
  TYPE (func), INTENT (in) :: x
  REAL (dpk), INTENT (out) :: val
  REAL (dpk), INTENT (out), OPTIONAL :: Dx(n), DDx(n*(n+1)/2)
END SUBROUTINE extract
```

4.1. Supported F90 intrinsics

The module was designed in such a way that any function or subroutine in standard F90 that can be applied to a real number can also accept a TYPE (func) variable and give the expected result. For example, a statement

$$f = \text{SIN}(y),$$

where *f* and *y* are of TYPE (func), amounts to three statements, when *drviv* = 2 (the FORTRAN 90 array notation is used)

```
f%value = SIN(y%value)
f%x     = COS(y%value) * y%x
f%xx    = -SIN(y%value) * y%x ⊗ y%x + COS(y%value) * y%xx.
```

The product of two arrays, $y\%x \otimes y\%x$, in the last expression is the tensor product. It produces a higher order tensor (the Hessian), which is stored in a rank-1 array as in Eq. (1). As operations involving intrinsic types can not be redefined, we should use a function (*tensor*) or define a new operator (*.tensor.*), as we chose in *deriv_class*, to compute this special product.

In order to localize all references to the exact representation of TYPE (func), we introduce several functions under the generic names *value*, *FD*, *SD*, and *val_assign*, *FD_assign*, *SD_assign*. They provide the value, the first

and the second derivatives, and assign them to the components of a TYPE (func) variable. They are not meant to be used outside the module so they are declared as PRIVATE. Extensive use of pointers keep them very simple and it should be a relatively easy task for a compiler with inlining capabilities to integrate them in a code without the overhead of a function call. We anticipate that no loss in efficiency is introduced by this practice while it adds considerably to the maintainability of the code. The only binding to the internal representation of TYPE (func), remains the assumption that the derivatives are stored in rank-1 arrays; all pointers are declared as such. Unless type aliasing is introduced in FORTRAN, this limitation can not be waived. We could have declared the derivatives as new user-defined types to solve it but by this we would make the code more complex and, possibly, less efficient.

Care was taken that all meaningful for numerical calculations intrinsic functions and every built-in operator of FORTRAN 90 are overloaded appropriately. Fully supported are all expressions involving the following routines: ABS, ATAN, CEILING, COS, COSH, DIGITS, DIM, DOT_PRODUCT, EPSILON, EXP, EXPONENT, FLOOR, FRACTION, HUGE, KIND, LOG, LOG10, MATMUL, MAXEXPONENT, MINEXPONENT, MOD, MODULO, NEAREST, PRECISION, RADIX, RANGE, RRSPPACING, SCALE, SET_EXPONENT, SIGN, SIN, SINH, SPACING, TAN, TANH, TINY.

Certain others are not fully overloaded; details on them are presented in the following paragraph.

4.2. Limitations

As we can not implement the *kind* as an argument, required by AINT, ANINT, INT, NINT, their overloaded functions simply do not accept this argument. They can transform only to the default kinds defined by *dpk* and *ik*.

A very serious limitation stems from the lack in FORTRAN 90 of user-defined elemental functions. An expression like $y = \text{SIN}(x)$ where y, x are conformable arrays, is accepted by FORTRAN 90 if the array elements are of a built-in type. Unfortunately, this does not hold for user-defined types. The multitude of definitions that must be added to AUTO_DERIV in order to provide a similar behavior for them, would make the code very complicated. The user is, therefore, required either to make explicit the loop or loops implied in his/her code or, a better solution, to enhance *deriv_class* by adding to it only the exact instance of the function needed. This normally means that the programmer should copy the function accepting scalars, rename it, change the type of the arguments to make them arrays of the required rank, and add the function name to the public interface provided by the module.

A similar problem arises for intrinsic functions accepting arrays (SUM, PRODUCT, MAXLOC/MINLOC, and MAXVAL/MINVAL). AUTO_DERIV overloads these functions to accept arrays of rank-1 only of TYPE (func) elements. If the user wishes to apply any of the above functions to arrays of different rank, he/she should add to the module the appropriate version of the function with the dimensions of the array arguments altered.

FORTRAN does not provide functions a mechanism to accept variable number of arguments. Therefore, the implementation of overloaded functions to MAX, MIN is inherently incomplete. The number of arguments must be fixed, while the not overloaded versions can have arbitrary. *deriv_class* provides MAX/MIN that accept two or three arguments. If the user needs a MAX/MIN with more, the appropriate instance has to be added and overload MAX/MIN by including its name to the relevant interface.

Note, that in the implementation of MAX, MIN and MAXVAL, MINVAL we use MAXLOC, MINLOC; finding only the maximum/minimum *value* does not suffice. As MAXLOC, MINLOC do not accept the optional argument *DIM* in FORTRAN 90, the four functions above can not be implemented efficiently to accept it either. We chose not to support the optional argument in them.

A serious mathematical problem arises in cases where we have discontinuity or when 0/0 is encountered; for example, in $\text{abs}(x)$ at $x = 0$, or when the derivatives of $\sqrt{f(x)}$ at $x = 0$ are required and $f(0) = f'(0) = 0$. Care was taken when only the denominator in a fraction, equals to zero; this case is avoided, unless the user in the code performs an illegal operation (which would show up even without using *deriv_class*). But “undefined” situations as the above, may appear in ACOS, ASIN, ATAN2, SQRT. In such cases, 0/0 is arbitrarily resolved to 0; admittedly this is not correct. The user should not depend on AUTO_DERIV in these occasions. Note, that even computing by other means the analytic expressions for the derivatives would lead to this problem. In such a situation, usually, the user code needs rethinking.

5. Tests

We conducted certain tests to measure the efficiency of `AUTO_DERIV`. We have calculated the elapsed time during the computation of the value and the derivatives of functions for which the corresponding analytic expressions were known. In each test, we use a version of the FORTRAN 90 code with the derivatives computed analytically, “by hand”, and another incorporating `AUTO_DERIV`.

The first mathematical function we used is the Potential Energy Surface for the HCP molecule, described in [12]. It is a realistic example of a function depending on three variables and it is sufficiently complex for the purpose of exhibiting the capabilities of `AUTO_DERIV`. Another test was made with the potential for the HF-dimer [13], a function of six variables. For this we were able to calculate analytically the first derivatives only.

In Table 1 we present the elapsed time, averaged over 1000 evaluations, during the computation of each potential and for the two versions (analytic derivatives and using `AUTO_DERIV`). The results are tabulated for the compilers we had available. The standard `DATE_AND_TIME` subroutine was employed.

The tests were performed on an Intel Pentium II processor at 450 MHz, running the GNU/Linux operating system. We have also compiled and run the programs on IBM RS/6000 and on HP-PA 2.0 processors with the system’s compilers; these results are not presented here. In the test we used the F90/95 compilers shown in Table 2; the options chosen should be the optimal ones.

Table 1
Elapsed times during the calculations of the derivatives of Potential Energy Surfaces for the molecular systems HCP and HF-dimer. Time quoted in ms

Potential	Compiler	Order of derivatives		
		0	1	2
HCP (analytic)	NAG	0.626	1.155	2.612
HCP (<code>AUTO_DERIV</code>)	NAG	8.465	38.147	83.975
HCP (analytic)	PGI	0.712	1.217	2.451
HCP (<code>AUTO_DERIV</code>)	PGI	9.105	355.795	807.796
HCP (analytic)	Fujitsu	0.690	1.270	2.703
HCP (<code>AUTO_DERIV</code>)	Fujitsu	10.279	66.731	142.812
HCP (analytic)	Absoft	0.667	1.288	2.973
HCP (<code>AUTO_DERIV</code>)	Absoft	16.075	68.494	156.458
HF dimer (analytic)	NAG	0.071	0.752	–
HF dimer (<code>AUTO_DERIV</code>)	NAG	1.807	5.302	14.250
HF dimer (analytic)	PGI	0.152	0.518	–
HF dimer (<code>AUTO_DERIV</code>)	PGI	1.142	39.513	103.397
HF dimer (analytic)	Fujitsu	0.075	0.596	–
HF dimer (<code>AUTO_DERIV</code>)	Fujitsu	1.903	9.335	23.231
HF dimer (analytic)	Absoft	0.086	1.315	–
HF dimer (<code>AUTO_DERIV</code>)	Absoft	3.174	10.434	22.812

Table 2
FORTRAN 90/95 compilers used for the test runs

Compiler	Options
NAGWare f95 Rel. 4.0(185)	–Ounroll=1 –O4
gcc 2.95.1	–Wc, –funroll-loops, –O3, –fforce-mem –Wc, –fforce-addr, –march=i686
Portland Group, Inc. pgf90 v3.0	–fast –tp p6
Fujitsu F90	–O3 –Kfast,eval,PENTIUM_PRO –AR
Absoft Pro Fortran 6.0 f90 v2.1	–B100 –O

The required memory during the tests, naturally, depends directly on the specific function differentiated. As we use a TYPE (func) variable holding $1 + n + n(n + 1)/2$ REAL(dpk) numbers for each REAL(dpk) mathematical variable in the original code we expect a roughly proportional increase in memory taken by the program.

6. Discussion

AUTO_DERIV was designed to make the evaluation of analytic first and second derivatives feasible; computing them “by hand” is, in realistic cases, too time consuming and highly error prone. The emphasis was not on implementing it optimally. Efficiency is, for the most part, hampered by the primitive support FORTRAN 90 has for value-oriented programming. Most important was to make it work, with as little change in user’s code as possible. We can see from the tests in the last section that AUTO_DERIV should not be considered as an option when the derivatives are, or can easily be, available analytically. Instead, it is a valuable tool when we deal with complicated mathematical functions with no derivatives available. We anticipate that as the compilers evolve and support more efficiently the advanced features of FORTRAN 90 this disproportionality in time performance between AUTO_DERIV and analytic derivatives shrinks and researchers will not be discouraged for using it.

The current standard [6] of FORTRAN 95 (F95) includes a few extra facilities (ELEMENTAL attribute of subroutines and functions) which would enable us to provide a much “cleaner” and more powerful implementation. However, the relative scarcity of FORTRAN 95 compilers led us to postpone these modifications for future upgrades.

As we have mentioned before, the applications that we have in mind are mainly from the area of Molecular Dynamics. Since we want to integrate the Hamilton equations and the variational equations one could implement AUTO_DERIV to compute the derivatives of the Hamiltonian function itself. However, the kinetic part can in most cases easily be differentiated and AUTO_DERIV is only applied on the potential function.

We may ask whether it is possible to compute higher order derivatives from the first and second ones. Recursive functions could be used in order to compute derivatives of arbitrarily high order. We leave such extensions for future upgrades.

Care was taken that the array notation provided by F90 is used throughout the code. It might assist a compiler to optimize, or, even, parallelize it. Unfortunately, the current definition of High Performance Fortran (v2.0) does not support distribution onto multiple processors of arrays of a derived type; otherwise, compiler directives could be inserted to indicate to the HPF compiler possible parallelisms.

When F95 compilers become widespread, and, most importantly, when the expected major revision in the year 2000 is standardized, some of the above limitations in the package will be overcome. We plan to follow closely all enhancements in the language and incorporate them to AUTO_DERIV.

Acknowledgement

We thank P. Maragakis for critically reading the manuscript and commenting on it.

References

- [1] R.E. Wyatt, J.Z.H. Zhang (Eds.) (Marcel Dekker, New York, 1996).
- [2] J.N. Murrell, S. Carter, S.C. Farantos, P. Huxley, A.J.C. Varandas, *Molecular Potential Energy Functions* (Wiley, New York, 1984).
- [3] R. Seydel, *From Equilibrium to Chaos: Practical Bifurcation and Stability Analysis* (Elsevier, Amsterdam, 1988).
- [4] S.C. Farantos, *Comp. Phys. Commun.* 108 (1998) 240.
- [5] International Organization for Standardization, ISO/IEC 1539-1:1991, *Information technology–Programming languages–Fortran*, 1991.
- [6] International Organization for Standardization, ISO/IEC 1539-1:1997, *Information technology–Programming languages–Fortran*, 1997.
- [7] Wolfram Research, Inc., *Mathematica*, Version 4.0 (1999).
- [8] Waterloo Maple, Inc., *Maple V*, Release 5.1 (1999).
- [9] The Mathworks, Inc., *Matlab 5.3*, 24 Prime Park Way Natick, MA, USA.
- [10] C. Bischof, A. Carle, P. Hovland, P. Khademi, A. Mauer, *The ADIFOR 2.0 System for the Automatic Differentiation of Fortran 77 Programs*. Technical Memorandum 192, Mathematics and Computer Science Division, Argonne National Laboratory, June 1998. URL address: <http://www.mcs.anl.gov/adifor>.
- [11] M. Metcalf, J. Reid, *Fortran 90/95 Explained* (Oxford University Press, Oxford, 2nd edn., 1996).
- [12] H. Ishikawa, R.W. Field, S.C. Farantos, M. Joyeux, J. Koput, C. Beck, R. Schinke, *Ann. Rev. Physical Chemistry* 50 (1999) 443.
- [13] W. Klopper, M. Quack, M.A. Suhm, *J. Chem. Phys.* 108 (1998) 10096.