1 SUMMARY

This Fortran 95 package provides automatic differentiation facilities for variables specified by Fortran code. Each independent variable and each variable whose value depends on the value of any independent variable must be declared to be of type AD02_REAL instead of default REAL (double precision REAL in the DOUBLE version). Note that Fortran variables of type default REAL (double precision REAL in the DOUBLE version) and default INTEGER may enter the computation provided their values do not vary with the values of the independent variables. Both the backward and the forward method are available.

ATTRIBUTES — **Version:** 1.1.0. **Types:** Forward (single, double), Backward (single, double). **Calls:** KB07. **Remark:** This is a threadsafe version of HSL_AD01 and supersedes it. **Origin:** J. K. Reid, Rutherford Appleton Laboratory and D. Cowey, RMCS Shrivenham. **Original date:** October 2001. **Language:** Fortran 95.

2 HOW TO USE THE PACKAGE

Access to the package requires a USE statement such as

Forward method, single precision version

```
USE HSL_AD02_FORWARD_SINGLE
```

Forward method, double precision version

```
USE HSL_AD02_FORWARD_DOUBLE
```

Backward method, single precision version

```
USE HSL_AD02_BACKWARD_SINGLE
```

Backward method, double precision version

```
USE HSL_AD02_BACKWARD_DOUBLE
```

If it is required to use more than one module at the same time, the types AD02_REAL and AD02_DATA (Section 2.1.1) and the generic procedures AD02_FLAGS (Section 2.5) and AD02_CONTROL (Section 2.6) must be renamed on all but one of the USE statements. The generic procedures described in Sections 2.1 to 2.4 do not need to be renamed.

Each module contains a derived type called AD02_REAL whose components are private. Dependent variables must be declared to be of this type instead of default REAL (double precision REAL in the DOUBLE version).

Information about each computation is held in a structure of type AD02_DATA, whose components are again private. Each such structure must be initialized by a call to AD02_INITIALIZE_DATA. Following this, it may be used for a sequence of computations, each starting with a call to AD02_INITIALIZE_COMP to say how many derivatives are wanted, declare the independent variables, and provide values for the independent variables. In each computation, the values of the dependent variables are calculated using the language features specified in Section 2.2, which include most of the Fortran 77 intrinsic functions. The name of an intrinsic function called with an AD02_REAL argument must not be given the INTRINSIC attribute. Procedure calls may be made to obtain derivative values. For cases where few of the independent variables affect a dependent variable, there are facilities to return the derivatives in packed form.

If derivatives are required for another set of values of the independent variables or for a different computation, a similar process must be executed. To preserve the old calculation, fresh calls of AD02_INITIALIZE_DATA and AD02_INITIALIZE_COMP must be made. If the old calculation is no longer needed, a fresh call to AD02_INITIALIZE_COMP with the same structure will ensure that its storage is reused. The procedure AD02_FINALIZE_DATA may be called when an AD02_DATA structure is no longer needed. It deallocates the pointer arrays associated with the problem.

An operation between values belonging to different computations or where an operand is undefined is regarded as

invalid and the result is treated as undefined. If an operand is associated with an AD02_DATA structure, the result is treated as an undefined value that is associated with the structure. We call this '**data-undefined**'. The procedure AD02_UNDEFINE (Section 2.2.1) may be called to give a variable the undefined or data-undefined value. The data-undefined value is needed for an assignment from a real or integer expression so that the assignment subroutine has a structure for the result.

A call to AD02_INITIALIZE_COMP gives the data-undefined value to all the AD02_REAL variables of the computation and a call to AD02_FINALIZE_DATA gives them the undefined value.

A facility is included to permit the user to add further unary functions to the set of functions supported. For a function $f(x)$, code must be provided that when given a real value x calculates $f(x)$, $f^{(1)}(x)$, ..., $f^{(r)}(x)$ where r is the order of derivatives wanted. The module itself may be used to calculate the derivatives of f (see Section 2.4).

For further explanation of how to convert codes, see Pryce and Reid (1998).

Pryce, J. D. and Reid, J. K. (1998). AD01, a Fortran 90 code for automatic differentiation. Report RAL-TR-1998-057, Rutherford Appleton Laboratory.

2.1 Argument lists and calling sequences of principal subroutines

There are seven principal subroutines:

1. The subroutine AD02_INITIALIZE_DATA must be called first for each AD02_DATA structure.
2. The subroutine AD02_INITIALIZE_COMP must be called prior to commencing a computation.
3. The subroutine AD02_VALUE provides the value of an independent or a dependent variable.
4. The subroutine AD02_GRAD provides the first derivatives of a dependent variable.
5. The subroutine AD02_HESSIAN provides the Hessian matrix (second derivatives) and optionally the first derivatives of a dependent variable.
6. The subroutine AD02_DERIVS provides derivatives of a dependent variable in packed form as Taylor coefficients (see Section 2.7).
7. The subroutine AD02_FINALIZE_DATA may be called when the results in an AD02_DATA structure are no longer needed.

2.1.1 The derived data types

The independent and dependent variables must be declared to be of type AD02_REAL instead of default REAL (double precision REAL in the DOUBLE version). The initial value is always undefined.

Data about each computation is stored in a scalar pointer structure of type AD02_REAL, which is allocated on a call of AD02_INITIALIZE_DATA. Each AD02_REAL of the computation has a pointer component associated with the structure, which is sufficient to identify the computation on most procedure calls. However, some need the AD02_REAL structure to be passed as an argument. The user must not declare objects of this type other than scalar pointers and must not use them in any way other than as arguments to AD02 procedures.

If an actual argument is an expression of type AD02_REAL and the corresponding dummy argument is referenced more than once, it must be copied to a local variable, as in the following example

```
SUBROUTINE SUB(ARG_A,B)
   TYPE (AD02_REAL), INTENT(IN) :: ARG_A
   TYPE (AD02_REAL), INTENT(OUT) :: B

   A = ARG_A
   : ! Unchanged code. A was a dummy argument and is now local
```

For safety, this change may be made to all input arguments of type AD02_REAL.

If the user has a derived type with a component of type AD02_REAL, for example:

```
TYPE USER_TYPE
   TYPE (AD02_REAL) :: A
   INTEGER :: I
END TYPE USER_TYPE
```

an intrinsic assignment to an object of type USER_TYPE will involve an intrinsic assignment for the component of type AD02_REAL. Since this would be erroneous, the user must define assignment for USER_TYPE:

```
INTERFACE ASSIGNMENT(=)
   MODULE PROCEDURE ASSIGN
END INTERFACE
:
SUBROUTINE ASSIGN (L,R)
   TYPE (USER_TYPE), INTENT(OUT) :: L
   TYPE (USER_TYPE), INTENT(IN) :: R
   L%A = R%A
   L%I = R%I
END SUBROUTINE ASSIGN
```

This must be done for any type that has a component of type AD02_REAL at any level of component selection.

2.1.2 The storage mode (forward method only)

The forward method has two storage modes. Let the number of independent variables be n and suppose a variable depends on m of these variables. If the derivatives are held in packed storage, only the derivatives with respect to these m variables are held; otherwise, all derivatives are held. The packed mode saves storage for reals unless $m=n$, but needs integer storage for the list of dependencies. The full mode saves computing time when n is small or if m/n is reasonably large. A threshold t is maintained and packed storage is used if $m<t$. The default value of t is 0 if $n\leq 5$; otherwise, it is the smallest value of m for which the full mode doubles the required storage (which depends on the order of derivatives being found). However, the value may be set explicitly on a call to AD02_INITIALIZE_COMP (see next section). If the packed mode is required throughout, $t\geq n$ should be chosen. Note that this is necessary if it is desired to know exactly which are the dependent variables (see INDEX in Section 2.1.8). If the full mode is required throughout, $t\leq 0$ should be chosen.

2.1.3 To initialize an AD02_DATA structure

```
CALL AD02_INITIALIZE_DATA(DATA,ERROR)
```

DATA is scalar pointer of type AD02_DATA. If it has a target on entry, this is not altered. On return, it will have a target that will be used for computations. It must not be used other than as an argument to an AD02 procedure.

ERROR is of intent(out) and of type default INTEGER. It is given the value 0 after a successful entry or 3 if there is an allocation failure.

2.1.4 To start a computation

```
CALL AD02_INITIALIZE_COMP(DEGREE,X,VALUE,DATA,ERROR[,FULL_THRESHOLD])
```

DEGREE is scalar, intent(in), and of type default INTEGER. It specifies the order of the highest derivative required. **Restriction:** DEGREE $\geq$ 0. For backward differentiation (HSL_AD02_BACKWARD_SINGLE/DOUBLE), DEGREE $\leq$ 2.

X is scalar or rank-one, intent(out), and of type AD02_REAL. It identifies the independent variables and is given the value VALUE. The value of X must not be altered except by calling this subroutine afresh.

VALUE is of intent(in) and type default REAL (double precision REAL in the DOUBLE version). It must be scalar or have

the same shape as X and must be set by the user to the value for X.

DATA is scalar pointer of type AD02_DATA. It must have been initialized by a call to AD02_INITIALIZE_DATA. After a successful entry, it will be used for the computation to follow. Each of the AD02_REAL variables of the computation has a pointer component that will become associated with DATA. If DATA had been used for a previous calculation, any variable of that calculation starts as a data-undefined variables of the new calculation, even if it has the SAVE attribute.

ERROR is of intent(out) and of type default INTEGER. It is given the value 0 after a successful entry, 1 if DATA has not been initialized, 3 if there is an allocation failure, 4 if DEGREE < 0 or (backward differentiation only) DEGREE > 2, or 5 if X and VALUE have different shapes when VALUE is an array.

FULL_THRESHOLD is optional, of intent(in), and of type default INTEGER. It is ignored when the backward method is in use. If it is present when the forward method is in use, the value of the threshold t that controls packed storage mode is used (see Section 2.1.2) is set to min (SIZE(X), FULL_THRESHOLD).

2.1.5 To obtain the real value of an independent or a dependent variable

```
CALL AD02_VALUE(A,VALUE,ERROR)
```

A is scalar or of rank one, of intent(in), and of type AD02_REAL. It specifies the independent or dependent variable whose value is required.

VALUE is of intent(out) and of type default REAL (double precision REAL in the DOUBLE version). It must have the same shape as A and returns the value of A or zero if it is undefined or data-undefined.

ERROR is of intent(out) and of type default INTEGER. It is given the value 0 after a successful entry, 1 if the value of A is undefined, and 2 if the value of A is data-undefined, and 5 if VALUE does not have the same shape as A.

2.1.6 To obtain the first derivatives of a dependent variable

```
CALL AD02_GRAD(A,GRAD,ERROR)
```

A is scalar, intent(in), and of type AD02_REAL. It specifies the dependent variable whose derivatives are required.

GRAD is intent(out) and of type default REAL (double precision REAL in the DOUBLE version) whose shape must be that of the argument X of the AD02_INITIALIZE_COMP call. It returns the gradient of A with respect to X or zero if the value of A is undefined or data-undefined.

ERROR is of intent(out) and of type default INTEGER. It is given the value 0 after a successful entry, 1 if the value of A is undefined, 2 if the value of A is data-undefined, 3 if there is an allocation failure, 4 if first derivatives are not being calculated, and 5 if GRAD has the wrong size.

Note: If the Hessian is wanted too and the backward method is in use, it is more efficient to call AD02_HESSIAN.

2.1.7 To obtain the second derivatives of a dependent variable

```
CALL AD02_HESSIAN(A,HESSIAN,ERROR[,GRAD])
```

A is scalar, intent(in), and of type AD02_REAL. It specifies the dependent variable whose derivatives are required.

HESSIAN is of intent(out) and type default REAL (double precision REAL in the DOUBLE version). It must be scalar if X is scalar and of shape (SIZE(X),SIZE(X)) if X is a rank-one array, where X is the argument X of the AD02_INITIALIZE_COMP call. It returns the Hessian (second derivative matrix) of A with respect to X or zero if the value of A is undefined or data-undefined.

ERROR is of intent(out) and of type default INTEGER. It is given the value 0 after a successful entry, 1 if the value of A is undefined, 2 if the value of A is data-undefined, 3 if there is an allocation failure, 4 if second derivatives are not being calculated, 5 if GRAD has the wrong size, and 6 if HESSIAN has the wrong shape.

GRAD is intent(out), optional, and of type default REAL (double precision REAL in the DOUBLE version) whose shape is

that of X. It returns the gradient of A with respect to X or zero if the value of A is undefined or data-undefined.

Note: If a single diagonal entry of the Hessian is wanted, we recommend that the calculation be performed separately with only one independent variable. The whole diagonal can be found by successive calculations of this kind.

2.1.8 To obtain the derivatives of a dependent variable in packed form

```
CALL AD02_DERIVS(A,R,INDEX,DERIVS,ERROR)
```

A is scalar, intent(in), and of type AD02_REAL. It specifies the dependent variable whose derivatives are required.

R is scalar, intent(in), and of type default INTEGER. It specifies the order of derivatives required. **Restriction:** $0 \leq$ R $\leq$ DEGREE, where DEGREE is the argument DEGREE of the AD02_INITIALIZE_COMP call.

INDEX is a rank-one array pointer of type default INTEGER. It is not altered if the value of A is undefined or data-undefined. Otherwise, it is altered as follows. For the forward method when the derivatives are held in packed storage (see Section 2.1.2), INDEX is allocated a target of size *m*, the number of elements of X upon which A is dependent, where X is the argument X of the AD02_INITIALIZE_COMP call. The indices of these elements are placed in INDEX. For example, if A has been calculated by the statement

```
A = X(1) + X(3)*X(7)/X(8)
```

INDEX may take the value (/1,3,7,8/). For full storage with the forward method or for the backward method, the value (/ (I,I=1,SIZE(X)) /) is returned.

DERIVS is a rank-one array pointer of type default REAL (double precision REAL in the DOUBLE version). It is not altered if the value of A is undefined or data-undefined. Otherwise, it is altered as follows. For the If X is scalar, DERIVS is allocated a target of size 1. If X is an array, DERIVS is allocated a target of size ${}^{m+r-1}C_r$ if *m* = SIZE(INDEX) and *r* is the order of derivatives required. Its value is the vector of all Taylor coefficients (scaled derivatives, see Section 2.7) of order exactly *r* of A with respect to X(INDEX). Only one copy of each Taylor coefficient is held, namely the one with ordered multi-index

$$\mathbf{i}_r = (i_1, i_2, ..., i_r),\ m \geq i_1 \geq i_2 \geq ... \geq i_r \geq 1$$

and they are ordered lexicographically. For example, when $r = 3$, the ordering corresponds to the multi-index ordering

(1 1 1), (2 1 1), (2 2 1), (2 2 2), (3 1 1), (3 2 1), (3 2 2), (3 3 1), (3 3 2), (3 3 3), (4 1 1), ...

In the case $r=2$, it corresponds to holding the lower triangle of the Hessian by rows, in packed form suppressing the values that are bound to be zero and with the diagonal entries halved.

ERROR is of intent(out) and of type default INTEGER. It is given the value 0 after a successful entry, 1 if the value of A is undefined, 2 if the value of A is data-undefined, 3 if there is an allocation failure, and 4 if the value of R is outside the range 0 $\leq$ R $\leq$ DEGREE.

Note: If a single diagonal entry of the tensor of derivatives is wanted, we recommend that the calculation be performed separately with only one independent variable. The whole diagonal can be found by successive calculations of this kind.

2.1.9 To finalize a computation

```
CALL AD02_FINALIZE_DATA(DATA,ERROR)
```

DATA is a scalar pointer of type AD02_DATA that has been initialized by a call to AD02_INITIALIZE_DATA. Any of its pointer array components with a target is deallocated and it is given a special value. On return, any variable of type AD02_REAL that is associated with DATA is treated as data-undefined, even if it has the SAVE attribute.

ERROR is of intent(out) and of type default INTEGER. It is given the value 0 after a successful entry, 1 if DATA has not been initialized, or 3 if there is a failure of a deallocate statement for a component that was associated with a target.

2.2 The language supported

All dependent variables (of type AD02_REAL) are given values by assignments from expressions involving independent variables (also of type AD02_REAL), dependent variables whose values have been previously found, and default REAL (double precision REAL in the DOUBLE version) or INTEGER data objects that are treated as invariants for the purpose of differentiation.

The following operations and procedures for the type AD02_REAL are supported:

1. The operators +, –, *, /, and ** for two scalars of types
   (i) both AD02_REAL,
   (ii) one AD02_REAL and one default REAL (default REAL or double precision REAL in the DOUBLE version), or
   (iii) one AD02_REAL and one default INTEGER,

   and the result is of type AD02_REAL.
2. The operator – for a scalar of type AD02_REAL. The result is of type AD02_REAL.
3. The operators ==, /=, >, >=, <, and <= for two scalars of types
   (i) both AD02_REAL,
   (ii) one AD02_REAL and one default REAL (default REAL or double precision REAL in the DOUBLE version), or
   (iii) one AD02_REAL and one default INTEGER,

   and the result is of type default LOGICAL.
4. Assignment to a scalar of type AD02_REAL from a scalar of type AD02_REAL. The previous value of the left-hand-side is completely ignored; if it is associated with another AD02_DATA structure, the association is lost but other associations are not altered.
5. Assignment to a scalar of type AD02_REAL from a scalar of type default REAL (default REAL or double precision REAL in the DOUBLE version), or default INTEGER. The previous value of the left-hand-side must be associated with an AD02_DATA structure and the association is not altered.
6. Assignment to a scalar of type default INTEGER from a scalar of type AD02_REAL. Note that assignment to REAL is not provided since such an assignment is likely to be erroneous.
7. The functions ABS, ACOS, ASIN, ATAN, COS, COSH, EXP, LOG, LOG10, SIN, SINH, SQRT, TAN, and TANH for a scalar of type AD02_REAL. The result is of type AD02_REAL.
8. The functions AINT and ANINT for a scalar of type AD02_REAL. The result is of type AD02_REAL.
9. The functions DABS, DACOS, DASIN, DATAN, DBLE, DCOS, DCOSH, DEXP, DLOG, DLOG10, DSIN, DSINH, DSQRT, DTAN, and DTANH for a scalar of type AD02_REAL (DOUBLE version only). The result is of type AD02_REAL.
10. The functions DINT and DNINT for a scalar of type AD02_REAL (DOUBLE version only). The result is of type AD02_REAL.
11. The functions ALOG and ALOG10 for a scalar of type AD02_REAL (single precision version only). The result is of type AD02_REAL.
12. The functions ATAN2, MAX, MIN, and SIGN for two scalars of type AD02_REAL or one scalar of type AD02_REAL and one scalar of type default REAL (default REAL or double precision REAL in the DOUBLE version). The result is of type AD02_REAL.
13. The functions DATAN2, DMAX1, DMIN1, and DSIGN for two scalars of type AD02_REAL or one scalar of type AD02_REAL and one scalar of type double precision REAL (DOUBLE version only). The result is of type

AD02_REAL.

14. The functions AMAX1 and AMIN1 for two scalars of type AD02_REAL or one scalar of type AD02_REAL and one scalar of type default REAL (single precision version only). The result is of type AD02_REAL.
15. The functions INT and NINT for a scalar of type AD02_REAL. The result is of type AD02_REAL.
16. The functions IDINT and IDNINT for a scalar of type AD02_REAL (DOUBLE version only). The result is of type AD02_REAL.

2.2.1 To construct an undefined or data-undefined value

```
CALL AD02_UNDEFINE(X,[DATA])
```

X is of any rank with intent(out) and of type AD02_REAL. If DATA is absent or is present but has not been initialized by a call to AD02_INITIALIZE_DATA or has been finalized by an AD02_FINALIZE_DATA call, X is given the undefined value. Otherwise, X is given the data-undefined value associated with DATA.

DATA is an optional scalar pointer of type AD02_DATA.

2.2.2 To test for an undefined value

```
LOGICAL FUNCTION AD02_UNDEFINED(X)
```

X is scalar, intent(in), and of type AD02_REAL. If X is undefined, the result is true. Otherwise, the result is false.

2.2.3 To test for a data-undefined value

```
LOGICAL FUNCTION AD02_DATA_UNDEFINED(X)
```

X is scalar, intent(in), and of type AD02_REAL. If X is data-undefined, the result is true. Otherwise, the result is false.

2.2.4 To provide an additional unary function

To provide an additional unary function $f(x)$ the user must write a function of the form:

```
FUNCTION F(X)
   USE HSL_AD02_FORWARD_SINGLE
   TYPE(AD02_REAL) :: F
   TYPE(AD02_REAL), INTENT(IN) :: X
   INTEGER, PARAMETER :: DEGREE=2
   REAL D(0:DEGREE),VALUE_X
   CALL AD02_VALUE (X,VALUE_X)
   .... ! Code to evaluate f(x) in D(0) and its derivatives
        ! in D(1), .... , D(DEGREE)
   CALL AD02_USER(D, X, F, 'F') ! The final argument is optional
END FUNCTION F
```

The subroutine AD02_USER has the form

```
CALL AD02_USER(D, X, F [, NAME])
```

D is of intent(in) and type default REAL (double precision REAL in the DOUBLE version). It is a rank-one array with bounds 0:DEGREE, where DEGREE has the same value as the argument DEGREE of the AD02_INITIALIZE_COMP call. It must be set to hold the function value and the values of its derivatives.

X is a scalar, intent(in), and of type AD02_REAL. It specifies the dependent or independent variable whose function value is being found.

F is a scalar, intent(out), and of type AD02_REAL. It is set by the subroutine to the required function value.

NAME is optional. It is scalar, intent(in), and of type CHARACTER with assumed length. If present, it is included in any warning messages that result from calling the function with an undefined argument value. If absent, AD02_USER is used in the message.

2.3 Improving the performance of the forward method

The performance of the forward method can sometimes be dramatically improved if the value of a variable of type AD02_REAL is repeatedly updated, as in the example

```
DO I = 1, N-1
  S1 = 10.0D0*(X(I+1)-X(I)**2)
  S2 = 1.0D0 - X(I)
  F = F + S1**2 + S2**2
END DO
```

by inserting, ahead of the statement that performs the update, the call

```
CALL AD02_TEMP(F)
```

F is scalar, intent(inout), and of type AD02_REAL. It is given the status of 'temporary', which means that its value is used at most once before it is changed.

2.4 More than one computation

There are situations that make it appropriate to have several computations active at the same time. One example is when the user adds a unary function to the set of functions supported. For a function $f(x)$, code must be provided that when given a real value x calculates $f(x)$, $f^{(1)}(x)$, ..., $f^{(r)}(x)$ where r is the order of derivatives wanted. The module itself may be used to calculate the derivatives of f as a subsidiary calculation. This process will be more efficient than calling the inner procedure as a part of the main calculation.

2.5 Error handling

Associated with each calculation is an integer array holding error and warning flags and there is a similar array for computations not associated with a calculation. Component 3 is for an error and the rest are for warnings. The array initially has the the value 0. In the event of an error or warning, the corresponding component is incremented by one. The components are for the following:

1. Unused.
2. Unused.
3. Error: insufficient storage (failure of an ALLOCATE statement).
4. Unused.
5. SQRT(A) when A is of type AD02_REAL and has value 0.
6. A**B when A and B are of type AD02_REAL, and A has a non-positive value.
7. A**B when A is of type AD02_REAL, B is of type REAL (double precision REAL in the DOUBLE version), A has value 0, and B has a non-integer value less than DEGREE.
8. SIGN(A,B) when B is of type AD02_REAL and has value 0.
9. ABS(A) or SIGN(A,B) when A is of type AD02_REAL and has value 0.
10. INT(A) when A is of type AD02_REAL and has an integer value.
11. AINT(A) when A is of type AD02_REAL and has an integer value.
12. NINT(A) when A is of type AD02_REAL and has a value $n + 0.5$, where n is an integer.
13. ANINT(A) when A is of type AD02_REAL and has a value $n + 0.5$, where n is an integer.
14. MAX(A,B) when at least one of A and B is of type AD02_REAL and A has the same value as B.

15. MIN(A,B) when at least one of A and B is of type AD02_REAL and A has the same value as B.
16. A==B when at least one of A and B is of type AD02_REAL and A has the same value as B.
17. A/=B when at least one of A and B is of type AD02_REAL and A has the same value as B.
18. A>B when at least one of A and B is of type AD02_REAL and A has the same value as B.
19. A<B when at least one of A and B is of type AD02_REAL and A has the same value as B.
20. A>=B when at least one of A and B is of type AD02_REAL and A has the same value as B.
21. A<=B when at least one of A and B is of type AD02_REAL and A has the same value as B.

22-32. Unused.

33. Execution of function for value undefined or data-undefined.
34. Assignment to an array of type AD02_REAL.

35-40. Unused.

By default, execution continues following a warning and stops following an error, but other choices are available (see Section 2.6). If execution continues following a warning, an appropriate default action is taken and subsequent execution is normal. If execution of a computation continues following an error, no further changes are made to the computation data except by a call of AD02_UNDEFINE or AD02_FINALIZE_DATA. While running in this mode, any function result or defined assignment of type AD02_REAL has value undefined, any of type REAL or INTEGER has value zero, and any of type LOGICAL has value true.

2.5.1 To access the error flags

```
CALL AD02_FLAGS(DATA,L,  FLAG,ERROR[,RESET])
CALL AD02_FLAGS(DATA,L,U,FLAG,ERROR[,RESET])
CALL AD02_FLAGS(     L,  FLAG,ERROR[,RESET])
CALL AD02_FLAGS(     L,U,FLAG,ERROR[,RESET])
```

DATA is a scalar pointer of type AD02_DATA. If DATA is present, it must have been initialized by a call to AD02_INITIALIZE_DATA and not have been finalized by a call to AD02_FINALIZE_DATA. FLAG is given the values of the flags associated with the computation stored in DATA. Otherwise, FLAG is given the values of the flags associated with computations on undefined data.

L is an intent(in) default INTEGER scalar that specifies the first error flag required.

U is an intent(in) default INTEGER scalar that specifies the last error flag required.

FLAG is an intent(out) default INTEGER variable. If U is present it must be a rank-one array of size at least U-L+1; otherwise is must be scalar. On return it holds the requested flags.

ERROR is of intent(out) and of type default INTEGER. It is given the value 0 after a successful entry, 1 if L or U is outside the range [1,40], 2 if DATA is present but has not been initialized by a call to AD02_INITIALIZE_DATA, or 3 if DATA is present and has been finalized by a call to AD02_FINALIZE_DATA.

RESET is optional, of intent(in) and of type default LOGICAL. If present, the flags that are accessed are reset to zero.

2.6 Error control

By default, execution continues following a warning and stops after an error; error and warning messages are printed on unit 6. These choices may be altered for a computation by:

```
CALL AD02_CONTROL(DATA,ERROR,[,LP][,MP][,PRINT_LEVEL][,STOP_LEVEL])
CALL AD02_CONTROL(           [,LP][,MP][,PRINT_LEVEL][,STOP_LEVEL])
```

DATA is a scalar pointer of type AD02_DATA. If DATA is present, it must have been initialized by a call to AD02_INITIALIZE_DATA and the choices for the associated computations are affected. Otherwise, the choices for computations on undefined data are affected.

ERROR is of intent(out) and of type default INTEGER. It is given the value 0 after a successful entry or 2 if DATA is

present but has not been initialized by a call to AD02_INITIALIZE_DATA.

LP is optional, scalar, intent(in), and of type default INTEGER. If it is present, subsequent error messages will be sent to unit LP.

MP is optional, scalar, intent(in), and of type default INTEGER. If it is present, subsequent warning messages will be sent to unit MP.

PRINT_LEVEL is optional, scalar, intent(in), and of type default INTEGER. If it is present, subsequent error and warning messages are controlled according to the value of PRINT_LEVEL:

0. No printing.
1. Printing of error messages.
2. Printing of error and warning messages.

STOP_LEVEL is optional, scalar, intent(in), and of type default INTEGER. If it is present, subsequent errors are controlled according to the value of STOP_LEVEL:

0. Continue in execution.
1. Continue after a warning, but stop after an error.
2. Stop after a warning or an error.

2.7 The Taylor expansion in many variables

When working with high-order derivatives, it is convenient to use vector subscripts, or *multi-indices*. If the variable a depends on the distinct independent variables $x_{\texttt{INDEX}(i)}$, $i = 1, 2, ..., m$, a multi-index of order r is

$$\mathbf{i}_r = (i_1, i_2, ..., i_r),$$

where the indices lie in the range

$$1 \le i_k \le m, k = 1, 2, ..., r$$

and need not be distinct. The multi-index is *ordered* if the inequalities

$$i_1 \ge i_2 \ge ... \ge i_r$$

hold. Let $\sigma(\mathbf{i}_r)$ be the number of different multi-indices that are permutations of $\mathbf{i}_r$. The single ordered multi-index is a representative of this set. For example, (2 1 1) represents the set {(2 1 1), (1 2 1), (1 1 2)} and $\sigma(\,(2\,1\,1)\,)$ has the value 3.

For a derivative, we use the notation

$$D_{\mathbf{i}_r} a = \frac{\partial^r a}{\partial x_{\texttt{INDEX}(i_1)} \ldots \partial x_{\texttt{INDEX}(i_r)}},$$

and the $D_{\mathbf{i}_r} a$ with ordered $\mathbf{i}_r$ is a single representative of $\sigma(\mathbf{i}_r)$ identical derivatives. To save storage, the package stores derivatives only for ordered multi-indices. It scales them to the *Taylor coefficients*

$$T_{\mathbf{i}_r} a = \frac{\sigma(\mathbf{i}_r)}{r!} D_{\mathbf{i}_r} a$$

for convenience in Taylor expansions. This makes the Taylor expansion of order R have the form

$$a(\mathbf{x}+\mathbf{h}) = a(\mathbf{x}) + \sum_{r=1}^{R} \sum_{\substack{\text{ordered} \\ \mathbf{i}_r}} \left(T_{\mathbf{i}_r} a \prod_{k=1}^{r} h_{\texttt{INDEX}(i_k)} \right)$$

It can be shown that if an ordered $\mathbf{i}_r$ has l groups of identical indices and the number of indices in the groups are n_1,

n_2, ..., n_l, the relation

$$\frac{\sigma(\mathbf{i}_r)}{r!} = \frac{1}{n_1!\, n_2! \ldots n_l!}$$

is true.

3 GENERAL INFORMATION

Use of common: None.

Other routines called directly: calls KB07AI (forward method only).

Restrictions:

DEGREE ≥ 0, DEGREE ≤ 2 (HSL AD02 BACKWARD SINGLE/DOUBLE), 0 ≤ R ≤ DEGREE.

4 METHOD

In the forward method, for each independent variable and each dependent variable, the module holds a representation of the values of the variable and all the desired derivatives of it. For each elementary operation, the desired derivatives of the result are calculated from those of the primaries by the chain rule. For example, if

$$a = b*c,$$

we have

$$\frac{\partial a}{\partial x} = \frac{\partial b}{\partial x} c + b \frac{\partial c}{\partial x}.$$

All the derivatives are calculated at the same time as the values.

In the backward method, a tree is constructed to represent the whole computation, with a node i for each independent variable, $i = 1, 2, ..., m$, and a node i for the result of each elementary operation, $i = m+1, 2, ...$, with links to nodes for the primaries of the operation. The nodes are in execution-order sequence, so the links are always to nodes with lesser indices. Only the values are constructed initially in the forward pass that constructs the tree. Let us use the notation x_i for the value at node i and suppose that derivatives of $f = x_n$ are required. As well as x_i, $\frac{\partial f}{\partial x_i}$ is held at node i, $i = 1, 2, ..., n$. The derivatives are calculated when one of the subroutines AD02_GRAD, AD02_HESSIAN, or AD02_DERIVS is called. Initially, all the variables are regarded as independent so that all the derivatives are zero except at node n where the derivative value is 1. One by one, from n backwards to $m+1$, the variables are changed to be dependent and the derivatives updated by the chain rule. For example, if

$$a = g(b,c),$$

when a is changed to dependent, we have

$$\frac{\partial f}{\partial b}^{new} = \frac{\partial f}{\partial b}^{old} + \frac{\partial f}{\partial a}^{old} \frac{\partial g}{\partial b}.$$

The forward method is likely to be best if the number of independent variables is small, since then the extra work and storage to compute and hold all the derivatives as the computation proceeds is modest. For a large number of independent variables, the forward method becomes impractical, but the work of the backward method is bounded by a small fixed multiple of the work needed for the values themselves. The disadvantage of the backward method is that the whole computational tree has to be stored, which is not practical for very long computations.

For further details, see Pryce and Reid (1998).

Pryce, J. D. and Reid, J. K. (1998). AD01, a Fortran 90 code for automatic differentiation. Report RAL-TR-1998-

057, Rutherford Appleton Laboratory.

4.1 Acknowledgement

This work is based on that of Stephens (1991) for Fortran 77. A Sun version is available from Nag on

http://www.nag.co.uk:70/1/public/DAPRE

Stephens, B. R. (1991). Automatic differentiation as a general purpose numerical tool. Faculty of Science Ph. D. Thesis, University of Bristol.

5 EXAMPLES OF USE

5.1 Simple example

To calculate the derivatives and Hessian of the function $(x_1^4 - 3)^2 + x_2^3$:

```
PROGRAM TEST
   USE HSL_AD02_FORWARD_DOUBLE
   INTEGER :: DEGREE=2
   DOUBLE PRECISION :: VALUE(2),FUN,GRAD(2),HESSIAN(2,2)
   TYPE (AD02_REAL) :: X(2),F
   TYPE (AD02_DATA),POINTER :: DATA
   INTEGER :: ERROR

   READ(5,*) VALUE
   CALL AD02_INITIALIZE_DATA(DATA,ERROR)
   CALL AD02_INITIALIZE_COMP(DEGREE,X,VALUE,DATA,ERROR)
   WRITE(6,'(A,2ES12.4)') 'At X =',VALUE
   F = (X(1)**4 - 3.0D0)**2 + X(2)**3
   CALL AD02_VALUE(F,FUN,ERROR)
   WRITE(6,'(A,2ES12.4)') 'F =',FUN
   CALL AD02_GRAD(F,GRAD,ERROR)
   WRITE(6,'(A,2ES12.4)') 'Grad(F) =',GRAD
   CALL AD02_HESSIAN(F,HESSIAN,ERROR)
   WRITE(6,'(A/(T12,2ES12.4))') 'Hessian(F) =',HESSIAN

END PROGRAM TEST
```

Given the input

```
     2.0 3.0
```

the output would be

```
At X =  2.0000E+00  3.0000E+00
F =  1.9600E+02
Grad(F) =  8.3200E+02  2.7000E+01
Hessian(F) =
             3.2960E+03  0.0000E+00
             0.0000E+00  1.8000E+01
```

5.2 Use of two modules at once

Here is the example of Section 5.1 using both methods.

```
PROGRAM TEST
   USE HSL_AD02_FORWARD_DOUBLE
   USE HSL_AD02_BACKWARD_DOUBLE,AD02B_DATA=>AD02_DATA,&
                AD02B_REAL=>AD02_REAL
   INTEGER :: DEGREE=2
   DOUBLE PRECISION VALUE(2),FUN,GRAD(2),HESSIAN(2,2)
   TYPE (AD02_REAL) :: X(2),F
   TYPE (AD02B_REAL) :: XB(2),FB
   TYPE (AD02_DATA),POINTER :: DATA
   TYPE (AD02B_DATA),POINTER :: DATAB
   INTEGER :: ERROR

   READ(5,*) VALUE

! Forward method
   CALL AD02_INITIALIZE_DATA(DATA,ERROR)
   CALL AD02_INITIALIZE_COMP(DEGREE,X,VALUE,DATA,ERROR)
   WRITE(6,'(A,2ES12.4)') 'At X =',VALUE
   F = (X(1)**4 - 3D0)**2 + X(2)**3
   CALL AD02_VALUE(F,FUN,ERROR)
   WRITE(6,'(A,2ES12.4)') 'F =',FUN
   CALL AD02_GRAD(F,GRAD,ERROR)
   WRITE(6,'(A,2ES12.4)') 'Grad(F) =',GRAD
   CALL AD02_HESSIAN(F,HESSIAN,ERROR)
   WRITE(6,'(A/(T12,2ES12.4))') 'Hessian(F) =',HESSIAN

! Backward method
   CALL AD02_INITIALIZE_DATA(DATAB,ERROR)
   CALL AD02_INITIALIZE_COMP(DEGREE,XB,VALUE,DATAB,ERROR)
   WRITE(6,'(A,2ES12.4)') 'At XB =',VALUE
   FB = (XB(1)**4 - 3D0)**2 + XB(2)**3
   CALL AD02_VALUE(FB,FUN,ERROR)
   WRITE(6,'(A,2ES12.4)') 'FB =',FUN
   CALL AD02_HESSIAN(FB,HESSIAN,ERROR,GRAD)
   WRITE(6,'(A,2ES12.4)') 'Grad(FB) =',GRAD
   WRITE(6,'(A/(T12,2ES12.4))') 'Hessian(FB) =',HESSIAN

END PROGRAM TEST
```

Given the input

```
     2.0 3.0
```

the output would be

```
At X =  2.0000E+00  3.0000E+00
F =  1.9600E+02
Grad(F) =  8.3200E+02  2.7000E+01
Hessian(F) =
             3.2960E+03  0.0000E+00
             0.0000E+00  1.8000E+01
At XB =  2.0000E+00  3.0000E+00
FB =  1.9600E+02
Grad(FB) =  8.3200E+02  2.7000E+01
Hessian(FB) =
             3.2960E+03  0.0000E+00
             0.0000E+00  1.8000E+01
```