Tangent linear and Adjoint Model Compiler

Users manual

Ralf Giering

e-mail: giering@dkrz.de

Manual Version 1.4

TAMC Version 5.2

September 2, 1999

Contents

1 Introduction

The **T**angent linear and **A**djoint **M**odel **C**ompiler (TAMC) is an automatic differentiation tool implemented as a source-to-source translator.

Automatic differentiation or computational differentiation is a technic for computing derivatives of functions given by an algorithm [Gri89]. A numerical program is an algorithm that can be viewed as a composition of differentiable functions. The composition is differentiated by application of the chain rule. The resulting multiple product can be evaluated in arbitrary order. The two main orders are called the forward mode and the reverse mode.

For algorithms given as Fortran- or C-programs two different approaches exists, operator overloading and source-to-source translation. The former make use of a Fortran-90 (C++) feature (ADOL-F [Shi96], IMAS, AD01, OPTIMA90 [Chr92], ADOL-C [GJU96]). In the former and some of the later approaches (DAFOR [Ber87], GRESS [Hor91], JAKEF [Hil82], PADRE2 [Kub96]) a tape is written during the function computations and interpreted to compute derivatives. A, in most cases, more efficient technic is to generate derivative code directly (ADIC [BR97], ADIFOR [BCKM94], Odyssee [RDG93], TAMC). In this case tapes are only used in reverse mode and they contain only values and do not have to be interpreted. A corresponding program computing derivatives in forward (reverse) mode is also called tangent linear (adjoint) program.

Compared to hand coding, automatic differentiation reduces the time for coding and error debugging substantially. It also offers the possibility to change the code without adapting the derivative computing code manually.

TAMC generates code to compute derivatives in **forward** and **reverse** mode out of the function computing code. The given source code must conform to the FORTRAN-77/Fortran-90 standard with some restrictions as described in sec. 3.3. The generated code responds to FORTRAN-77/Fortran-90 standard. TAMC options allow for common but non standard Fortran extensions to be used (sec. 4).

Most of the syntax errors of Fortran source codes are reported by TAMC, but it does not contain a sophisticated Fortran parser. Thus, it is strongly recommended to check the source code using a Fortran compiler before applying TAMC.

TAMC has been used to generate derivative code of small [HHG$^+$97] and large programs [KGH96, OBV$^+$97, SWG$^+$97, MZG$^+$99].

1.1 Installation

The utility to use TAMC can be downloaded from:

http://puddle.mit.edu/˜ralf/tamc.

For installation a C-compiler, a FORTRAN-compiler, and a UNIX[1] operating system ir required. The following commands will install the utility:

```
gzip -d       tamc_utility.tar
tar xvf       tamc_utility.tar
cd            tamc_V.VV
./configure
make
```

[1]UNIX is a trademark of AT&T

where **V.VV** is the current version. The **INSTALL** file contains some information about installation. It might be necessary to make changes to the generated **Makefile** file. Up to now AIX, Cray-UNICOS, DEC-ULTRIX, Linux-86, SGI-IRIX, SOLARIS, and SUN-OS are supported.

1.2 Usage

TAMC is invoked through a command of the form:

tamc -toplevel name **-input** vars **-output** vars [*other options*] *file_names*

Each option begins with a '-', they are described in sec. 4. The command

tamc -help

gives a short explanation of the options and their default settings. File names must be given as a blank separated list. For each given file name corresponding files are generated:

<u>Input :</u>	file.f	:	FORTRAN source code containing the function computing code
<u>Output :</u>	file_ad.f	:	adjoint code and modified code
	file_ftl.f	:	tangent linear code

Generated file names have the same extension as the original file names. If no extension is given **.f** is choosen.

TAMC constructs derivative code for subroutines and functions. It is assumed that the output variables of a specific routine should be differentiated with respect to its input variables. This top-level routine is specified by the option **-toplevel**. The independent (input) and dependent (output) variables may be arguments of the top-level routine or entries of local common blocks. They are specified by the options -input and -output.

TAMC is invoked remotely. The source code is packed and transmitted to a server where it is processed. The packed result is transmitted back and unpacked on the local computer. TAMC is installed for remote access on several servers. Please choose an appropriate server, see the INSTALL file for details.

1.3 Compile and Link

The generated files and the driver source code should be compiled and linked by:

`f77 -o` *executable* `-L$TAMC_PATH` *driver*`.f` *file*`.f` *file_ad*`.f` *file_ftl*`.f -ltamc`

where `TAMC_PATH` is the path of the TAMC directory, i.e. the full path of tamc_V.VV. Other FORTRAN compilers may be called differently (f90, cft77). Please note, the extension depends on the extension of the original files. Alternatively the TAMC utility (see sec. 9) can be used to compile and link the generated code for several purposes.

1.4 Changes to previous versions

- A data dependence analysis is applied by default.

- New option -split for additional expression splitting.
- Some High Performance Extensions are handled.
- pure forward mode
- reduction detection
- free source form
- store-directive extension
- additional flow-directives
- directive for general assignments
- all Fortran-90 intrinsic functions (some with restriction to arguments)
- Fortran-90 modules and USE-statements
- extensive normalization phase
- control flow analysis
- option -module is replaced by -toplevel to avoid confusion with Fortran-90 modules

1.5 Planed extensions

- named arguments
- full Fortran-90 intrinsic support
- FORALL statement (Fortran-95)
- static pointer handling (partly implemented)
- storing/recomputation strategies, code optimized for speed, memory, or weighted combination of both (prototype exists)
- micro tasking directives
- OpenMP directives
- more HPF directives

2 Tutorial example

The use of TAMC is demonstrated by the simple subroutine `prod` shown in Fig. 1. `Prod` computes the product of three numbers. Assume we want the derivatives of the output argument `p` of subroutine `prod` with respect

```
CADJ  subroutine prod ADNAME = adprod
CADJ  subroutine prod FTLNAME = g_prod
C*******************************************************
      subroutine prod( a, x, y, p )
      real a, x, y, p
      p = a*x*y
      end
C*******************************************************
      program driver
      real       a
      parameter( a = 1.23 )
      integer    pmax
      parameter( pmax = 2 )
      real x, y
      real adx, ady, adp
      real g_p(pmax), g_x(pmax), g_y(pmax)
C--------------------------------------------------
C initialize independent variables
C--------------------------------------------------
      x = 2.4
      y = 1.7
C--------------------------------------------------
C initialize adjoint variables
C and call top-level adjoint routine
C--------------------------------------------------
      adx = 0.0
      ady = 0.0
      adp = 1.0
      call adprod( a, x, y, adx, ady, adp )
      print *,' adx = ', adx
      print *,' ady = ', ady
C--------------------------------------------------
C initialize tangent variables to identity matrix
C and call top-level tangent routine
C--------------------------------------------------
      g_x(1) = 1.0
      g_x(2) = 0.0
      g_y(1) = 0.0
      g_y(2) = 1.0
      call g_prod( a, x, y, p, g_x, g_y, g_p )
      print *,'    p = ', p
      print *,' g_p(1) = ', g_p(1)
      print *,' g_p(2) = ', g_p(2)
      end
```

Program 1: Simple example: product of arguments

to the input arguments `x` and `y`. Then the independent variables are `x` and `y` and the dependent variable is `p`. The top-level routine is `prod`.

The names of the tangent linear (`g_prod`) and adjoint subroutine (`adprod`) are defined by directives (sec. 5). This is necessary because these subroutines are called inside the driver and the driver is visible to TAMC. TAMC takes care of not reusing existing names of objects. The directives can be neglected if the driver is not visible to TAMC.

2.1 Invoke TAMC for reverse mode

Invoking the TAMC by the command

tamc -toplevel prod **-input** x,y **-output** p **-reverse -pure** prod.f

generates the adjoint code shown in Fig. 2. Please note, because of the option -pure, the adjoint code does not compute the dependent variable `p`. In the driver program the adjoint independent variables `adx` and `ady`

```
      subroutine adprod( a, x, y, adx, ady, adp )
C**************************************************************
C**************************************************************
C** This routine was generated by the                        **
C** Tangent linear and Adjoint Model Compiler,    TAMC 4.87   **
C**************************************************************
C**************************************************************
      implicit none
C==============================================
C define arguments
C==============================================
      real a
      real adp
      real adx
      real ady
      real x
      real y
C----------------------------------------------
C ROUTINE BODY
C----------------------------------------------
      adx = adx+adp*a*y
      ady = ady+adp*a*x
      adp = 0.
      end
```

Program 2: TAMC generated adjoint code to the example.

are initialized to zero and the adjoint dependent variable `adp` to one. On completion the adjoint independent variables `adx` and `ady` contain the derivatives of `p` with respect to `x` and `y`.

2.2 Invoke TAMC for forward mode

Invoking the TAMC by the command

tamc -toplevel prod **-input** x,y **-output** p **-forward -pure -jacobian** 2 prod.f

will generate the tangent linear code shown in Fig. 3. Please note, because of the option -pure, the code does not compute the dependent variable `p`. In the driver program the tangent linear independent variables `g_x` and `g_y` are initialized to form an identity matrix. On completion the tangent linear dependent variable `g_p` contains the derivatives of `p` with respect to `x` and `y`.

```
      subroutine g_prod( a, x, y, p, g_x, g_y, g_p )
C***************************************************************
C***************************************************************
C** This routine was generated by the                        **
C** Tangent linear and Adjoint Model Compiler,    TAMC 4.87   **
C***************************************************************
C***************************************************************
      implicit none
C==============================================
C define parameters
C==============================================
      integer g_pmax
      parameter ( g_pmax = 2 )
C==============================================
C define arguments
C==============================================
      real a
      real g_p(g_pmax)
      real g_x(g_pmax)
      real g_y(g_pmax)
      real p
      real x
      real y
C==============================================
C define local variables
C==============================================
      integer g_i_
C----------------------------------------------
C TANGENT LINEAR AND FUNCTION STATEMENTS
C----------------------------------------------
      do g_i_ = 1, g_pmax
        g_p(g_i_) = g_x(g_i_)*a*y+g_y(g_i_)*a*x
      end do
      p = a*x*y
      end
```

Program 3: TAMC generated tangent linear code to the example.

3 Description

A detailed description of adjoint code construction is given in [GK98a]. This section gives only a brief description of the algorithm for adjoint and tangent linear code generation used by the TAMC.

In the first step the FORTRAN code is parsed, identifiers are collected and an abstract representation of the program is generated (scanner and parser). The code is checked for internal consistency and most unsupported features are detected.

The code normalization phase removes saved variables and places them on common blocks, each for every routine containing saved variables. Entry statements are than removed and a clone of the function is generated having the entry statement name and a proper control flow. Return labels in call argument lists and dummy argument lists are removed and an integer argument is inserted instead. The value of the corresponding dummy argument is set by an assignment which replaces the return statement. A test of the values is inserted after the call in order to continue control at the right place. All return statements are replaced by goto statements, their target is the end of the routine. Computed goto, assigned goto, assign to, and arithmetic if statements are replaced by proper constructs using unconditional goto statements. Labels in I/O statements (ERR=,END=,EOR=) are removed and instead the value of the error status (IOSTAT=) is tested. Statement functions are replaced by an inline expansion of the function. Expressions in argument lists are replaced by auxiliary variables assigned the value of the expression beforehand.

The control flow analysis is applied to routines containing goto statements (may be as a result of the normalization). A directed graph(control flow graph) is build and reduced whenever constructs like if-blocks, dowhile-constructs, or sequences of statements are detected. If the remaining graph is acyclic (DAG) node splitting is applied to allow further reduction. The graph is irreducible if it contains complex cyclic subgraphs.

Then a intra-procedural data dependence analysis detects loop-carried data dependences. The loop category is set and the data dependences are stored.

The following inter-procedural data flow analysis determines all active subroutines, functions, and variables. In a backward sweep all variables required to compute the dependent variables are determined. In the following forward sweep all variables influenced by the independent variables are determined. Variables required and influenced are active variables and all subroutines and functions computing active variables are active themselves.

During the transformation phase a derivative holding common block is constructed for every common block containing at least one active variable. Then an adjoint and/or tangent linear routine is generated for every active routine. If necessary extra routines are generated for the bookkeeping of storing. No derivative code will be constructed for a main program and a block data routine. All common blocks will be included in the main program if the code of the main program is given to TAMC.

The last phase translates the internal representation of the modified function, the adjoint, and the tangent linear routines into FORTRAN code. The dialect of the generated FORTRAN code can be controlled by options (see sec. 4).

3.1 Derivative names

For every active variable, subroutine, function and common block, the TAMC constructs an adjoint and tangent linear name. The construction of derivative names depends on the given options and directives. By default adjoint (tangent linear) names consist of the two letter prefix **ad** (**g_**) followed by the original name. The prefixes can be changed by options (-admark, -ftlmark, see sec. 4). It is also possible to turn the prefix into a postfix (option -postfix). In addition an individual derivative name can be specified by directives (see sec. 5). The TAMC accepts 31 significant letters and ensures that a generated name does not already exist by changing the last letter until the name is unambiguous.

3.2 Derivative statements

A numerical program consists mainly of a few kinds of statements, among them are:

- assignment
- block if construct
- case construct
- DO loop
- procedure call
- function reference
- block of statements

For these kind of statements and for the Fortran-90 constructs:

- DO WHILE loop
- array assignment
- where construct

a short description of the construction of the tangent linear and adjoint statement is given.

3.2.1 Assignment

An assignment can be considered as an operator acting on the vector of active variables. This vector consists of the left hand side (LHS) variable and all active variables of the right hand side (RHS) expression except variables inside index expressions (such as `I` in `A(4*I)`). For a general assignment

$$y \; = \; f(x_1, .., x_n, y) \;,$$

where f is an expression containing the variables x_i and the LHS variable y.

The tangent linear assignment is:

$$\delta y \; = \sum_{i=1}^{N} \frac{\partial f}{\partial x_i} \, \delta x_i \; + \; \frac{\partial f}{\partial y} \, \delta y \;,$$

where $\delta x_i, \delta y$ are small perturbations, the corresponding tangent linear variables.

Here, the adjoint matrix is the transposed Jacobian matrix. The coefficients of the variations δx_i and δy build the first line of the Jacobian matrix. The other lines consist of zeroes and ones in the diagonal. Out of this matrix the adjoint assignments are generated:

$$\begin{aligned} \delta^* x_i \;&=\; \delta^* x_i + \delta^* y \frac{\delta f}{\delta x_i} \\ \delta^* y \;&=\; \delta^* y \frac{\delta f}{\delta y} \;, \end{aligned}$$

where $\delta^* x_i$ and $\delta^* y$ are the corresponding adjoint variables. The assignment to the adjoint variable of the LHS variable must be the last one, because its old value is used by all previous adjoint assignments.

In the example given below, assume `X,Y,Z` are active and `A,B` are passive. Thus, the RHS expression depends non linearly on `Y`.

```
Z = A*X + B*Y*SIN(Y)
```

The tangent linear code is:

```
G_Z = A*G_X + B*SIN(Y)*G_Y + B*Y*COS(Y)*G_Y
Z   = A*X + B*Y*SIN(Y)
```

The adjoint code is:

```
ADY = ADY + ADZ*B*(SIN(Y) + Y*COS(Y))
ADX = ADX + ADZ*A
ADZ = 0.0
```

A more general rule is applied whenever TAMC detects a specific data dependence between the LHS variable and a RHS reference due to their repeated execution inside a loop. If both addresses may reference the same memory location for one loop path and different memory locations for another loop path, then an auxiliary variable is introduced for the RHS expression. In the assignment shown below the index expressions (I) and (2*I-1) have the same value only for I=1.

```
Y(I) = X(I) + Y(2*I-1)
```

Here an auxiliary adjoint variable `ADH` is introduced to hold the adjoint value of the LHS variable in order to prevent overwriting.

```
ADH = ADY(I)
ADY(2*I-1) = ADY(2*I-1) + ADH
ADX(I)     = ADX(I)     + ADH
ADY(I)     = 0.0
```

The rule is also applied if a specific directive is given (see sec. 5.2.4).

3.2.2 Array assignment

Tangent linear code to array assignments can be generated by the same rules as for regular assignments.

Generating adjoint code only one exception must be handled separately. If the RHS expression contains an active scalar variable it influences the whole array on the LHS of the assignment. Thus, all adjoint array components contribute to the adjoint scalar variable. (A broadcast in the code corresponds to a reduction in the adjoint code.)

TAMC generates a SUM function call for every active scalar variable occurring in the RHS expression of an array assignment. If the array assignment belongs to the body of a WHERE statement the WHERE condition becomes the mask argument of the SUM function and the whole statement is placed outside the WHERE statement (see also WHERE statement sec. 3.2.5).

code	adjoint code
`real x(nx,ny), z(nx,ny,nz), y`	`real x(nx,ny), y` `real adx(nx,ny), adz(nx,ny,nz), ady`

`where (x .gt. 0.0)`
`   x(:,:)  = y*z(:,:,2)`
 	
`end where`
	`ady = ady + SUM( x(:,:), x .gt.  0.0 )`
	`where (x .gt. 0.0)`
	`   adz(:,:,2) = adz(:,:,2) + adx(:,:)*y`
	`   adx(:,:)   = 0.0`
	`end where`

3.2.3 Loops

DO loop A loop is a composition of operators given by the loop kernel:

code	tangent linear code
`DO I =` low, up, step statement `END DO`	`DO I =` low, up, step tangent linear statement `END DO`

The tangent linear statement is a loop with the corresponding tangent linear kernel.

The same is true for the adjoint of loops having whether flow nor anti-flow dependences between different loop passes (see sec. 5.2.3). But in general, the adjoint statement is the reverse order composition of the adjoint kernel statement:

adjoint code

```
DO I = up, low, -step
   adjoint statement
END DO
```

If the value of the index variable in the last pass is not equal to the upper bound value, the first value of the adjoint index variable 'up' is replaced by the expression

$$\text{low} + \text{step} * \texttt{NINT}((\text{up} - \text{low})/\text{step} - 0.5) \ .$$

In case the adjoint loop kernel requires variables which are overwritten by the kernel, code is generated to recompute these variables in front of every adjoint loop kernel. Because this computation can be rather time consuming a warning is given. By inserting a store directive recomputations are replaced by storing and restoring values (see sec. 5.2.1).

In the example below a product of array components is computed.

code

```
FAC = 1.0
DO I = 1, N
   FAC = FAC * X(I)
END DO
```

adjoint code

```
DO I = N, 1, -1
   FAC = 1.0
   DO I1 = 1, I-1
      FAC = FAC * X(I)
   END DO
   ADX(I) = ADX(I) + ADFAC * FAC
   ADFAC  = ADFAC * X(I)
END DO
ADFAC = 0.0
```

The variable `FAC` holds an intermediate result whose actual value is required for the adjoint kernel computation. Since the adjoint kernel is computed in reverse order, the required value is not available and must be recomputed from prior intermediate results. This expensive recomputation can be avoided by inserting a store directive in front of the loop kernel (see sec. 5.2.1 and sec. 5.2.2).

Alternatively, the original code can be changed into code which does not overwrite required values by extending the dimension of the variable in question (scalar, expansion, array expansion).

code

```
FAC(0) = 1.0
DO I = 1, N
   FAC(I) = FAC(I-1) * X(I)
END DO
```

adjoint code

```
FAC(0) = 1.0
DO I = 1, N
   FAC(I) = FAC(I-1) * X(I)
END DO

DO I = N, 1, -1
   ADX(I) = ADX(I) + ADFAC(I) * FAC(I-1)
   ADFAC  = ADFAC(I) * X(I)
END DO
ADFAC = 0.0
```

In this case the required values are recomputed in the adjoint code in front of the adjoint loop. This recomputation is much cheaper than the previous one. Whenever TAMC detects only loop carried flow dependences adjoint code is generated by this rule.

For iterative loops see sec. 5.2.5.

DO WHILE loop Similar to a DO loop, the tangent linear statement to a DO-WHILE loop is a DO-WHILE loop with the same condition and the corresponding tangent linear kernel.

code

```
DO WHILE (condition)
   statement
END DO
```

tangent linear code

```
DO WHILE (condition)
   tangent linear statement
END DO
```

To compute the adjoint to a DO-WHILE loop the number of loop passes done must be known. Therefor the TAMC modifies the given DO-LOOP in order to compute this number and write it on a tape. The adjoint code reads the number of passes from tape and executes a DO loop with the adjoint kernel.

modified function code

```
COUNT = 0
DO WHILE (condition)
   COUNT = COUNT + 1
   statement
END DO
store COUNT
```

adjoint code

```
restore COUNT
DO I = 1, COUNT
   adjoint statement
END DO
```

In case of a conflict the same algorithm is applied as for a DO-loop (see above).

3.2.4 Block if construct

A conditional statement executes statements according to the logical value of a condition.

code

```
IF (condition A) THEN
   statement A
ELSE IF (condition B)
   statement B
ELSE
   statement C
END IF
```

The tangent linear code is simply the conditional statement executing the tangent linear code to every branch.

tangent linear code

```
IF (condition A) THEN
   tangent linear statement A
ELSE IF (condition B)
   tangent linear statement B
ELSE
   tangent linear statement C
END IF
```

adjoint code

```
IF (condition A) THEN
   adjoint statement A
ELSE IF (condition B)
   adjoint statement B
ELSE
   adjoint statement C
END IF
```

The adjoint code has to execute the adjoints of the statements which have been executed in the code. Consequently, the values of the conditions must be known in the adjoint code in order to decide which adjoint statement has to be executed. The condition is evaluated again in the adjoint code.

It is recommended to use ELSE branches whenever possible. If an ELSE branch exists, exactly one of the branches is executed. Otherwise, during the flow dependence analysis, it must be taken into account that no branch might be executed. In this case more recomputations might be necessary.

3.2.5 WHERE construct

A WHERE statement executes array assignments according to the logical value of a condition. The rank and shape of each LHS must conform to rank and shape of the array condition.

code

```
WHERE (array condition)
   array assignments
END WHERE
```

The tangent linear code is simply a WHERE statement having the same condition and executing the tangent linear array assignments.

tangent linear code

```
WHERE (array condition)
   tangent linear array assignments
END WHERE
```

The same is true for the adjoint code generated, but if the array assignments contain scalar active variables on the RHS expression then the adjoint code contains SUM function calls to update their scalar adjoint variables. These non array assignments (part 1) are not allowed inside a WHERE statement. Thus, they precede the WHERE statement, and the array condition of the WHERE statement becomes the mask of the SUM function (see also array assignment sec. 3.2.2). The part of the adjoint of an array assignment which consists of array assignments (part 2) builds the body of the adjoint WHERE statement.

adjoint code

```
adjoint array assignments, part 1
WHERE (array condition)
   adjoint array assignments, part 2
END WHERE
```

The example shown below contains the active scalar variables `y` and the two dimensional array `x`.

code

```
real x(nx,ny), y



WHERE (x(:,:) .gt. 0.0)
   x(:,:)  = 4*y/x(:,:)
END WHERE
```

adjoint code

```
real x(nx,ny), y
real adx(nx,ny), ady

ady = ady + SUM( 4*adx/x(:,:), x(:,:) .gt. 0.0)
WHERE (x(:,:)  .gt.  0.0)
   adx(:,:)  = -4*y*adx(:,:)/x(:,:)*x(:,:)
END WHERE
```

The adjoint array variable `adx` is updated inside the WHERE statement while the adjoint scalar variable `ady` is updated in front of it. The mask argument of the SUM function ensures that only adjoint values of used components of `x` are summed.

If the -jacobian option is given, an extra DO-loop over each component surrounds the tangent linear and adjoint WHERE statement:

tangent linear code

```
DO g_i_ = 1, g_pmax

   WHERE (array condition)
      tangent linear array assignments
   END WHERE
END DO
```

adjoint code

```
DO g_i_ = 1, g_pmax
   adjoint array assignments, part 1
   WHERE (array condition)
      adjoint array assignments, part 2
   END WHERE
END DO
```

3.2.6 Procedure call

The tangent linear (adjoint) statement of a procedure call is the call of the corresponding tangent linear (adjoint) procedure. The argument list of the tangent linear subroutine contains all variables of the corresponding subroutine and the tangent linear variables of the active variables. The argument list of the adjoint subroutine contains all required variables of the corresponding subroutine and the adjoint of the active variables.

An example will explain this. Assume the derivative code to the subroutine `SUB` has been constructed and the subroutine is called like:

```
CALL SUBR( N, A, B, X, Y, Z )
```

Assume furthermore the arguments `X`, `Y`, and `Z` are active, and the linearization of the subroutine depends on the input variables `N`, `A`, and `Y`. Then the tangent linear subroutine `G_SUBR` is called by:

```
CALL G_SUBR( N, A, B, X, Y, Z, G_X, G_Y, G_Z )
```

The adjoint subroutine `ADSUBR` needs the variables `N`, `A`, and `Y` it is called by:

```
CALL ADSUBR( N, A, Y, ADX, ADY, ADZ )
```

Expressions in the argument list will be replaced by auxiliary variables before generating the adjoint code.

Additional leading dimension arguments will be inserted after each derivative variable in the tangent linear and adjoint call if the -jacobian and -ldg options are given and the option -append is not used (see sec 4). An argument in front of the first adjoint argument specifies the number of Jacobian vector products requested.

```
CALL ADSUBR( N, A, Y, ADP_, ADX, LDX, ADY, LDY, ADZ, LDZ )
```

3.2.7 Function reference

For an active user defined function a tangent linear or adjoint subroutine is generated. Every user defined function reference in an expression list is replaced by an auxiliary variable to which the function result is assigned previously. The tangent linear (adjoint) statement to this assignment is the call of the corresponding tangent linear (adjoint) subroutine. The argument list of the tangent linear subroutine contains all variables of the corresponding subroutine, the LHS variable, the tangent linear variables of the active variables, and the tangent linear variables of the LHS variable. The argument list of the adjoint subroutine contains all required variables of the corresponding subroutine, the adjoint of the active variables, and the adjoint of the LHS variable. In the adjoint code the adjoint LHS variable is reset to zero after the call.

An example will explain this. Assume the derivative code to the function `FUNC` has been constructed and the function is referenced in an assignment:

```
Z = FUNC( N, A, X, Y )
```

Assume furthermore the arguments `X` and `Y` are active, and the linearization of the function depends on the input variables `N` and `X`. Then the tangent linear subroutine `G_FUNC` is called by:

```
CALL G_FUNC( N, A, X, Y, Z, G_X, G_Y, G_Z )
```

The adjoint subroutine `ADFUNC` requires the variables `N` and `Y` it is called by:

```
CALL ADFUNC( N, X, ADX, ADY, ADZ )
ADZ = 0
```

Expressions in the argument list will be replaced by auxiliary variables before generating derivative code. Additional leading dimension arguments will be inserted after each derivative variable in the tangent linear and adjoint call if the -jacobian and -ldg options are given and the option -append is not used (see sec 4).

3.2.8 Block of statements

A block of statements is a sequence of statements executed one after the other. An individual statement of a block can be a Fortran statement or a block itself. The tangent linear block of statements is simply the sequence of the tangent linear statements.

In order to obtain the adjoint of a block of statements, the adjoint of each statement is constructed and arranged in reverse order (right hand side in Fig. 1). These adjoint statements may require intermediate results of the original code defined inside the block or previously in preceding parts of the code (bold arrows between left and right hand side in Fig. 1). All statements within the block, which are needed for providing intermediate values, are included in front of the adjoint statements (left hand side in Fig. 1). A backward data flow analysis determines these statements. This is done by calculating the sets of input and output variables of each statement and including statements, which define a required variable.

The required variables of the adjoint block are variables directly referenced by an adjoint statement and variables needed for recomputing required intermediate variables. The adjoint block is a composition of statements defining intermediate variables followed by the adjoint statements. The scheme of the adjoint of three statements is

$$A'B' \;\; C^* \, B^* \, A^* \tag{1}$$

where the single quote denotes the subset of statements really needed. (the dashed line in Fig. 1 denotes this order of computations).

A conflict occurs whenever a required variable is redefined later in the block and the new value is also required for the adjoint code computations. E.g., if B^* requires a variable which is overwritten by the computations of

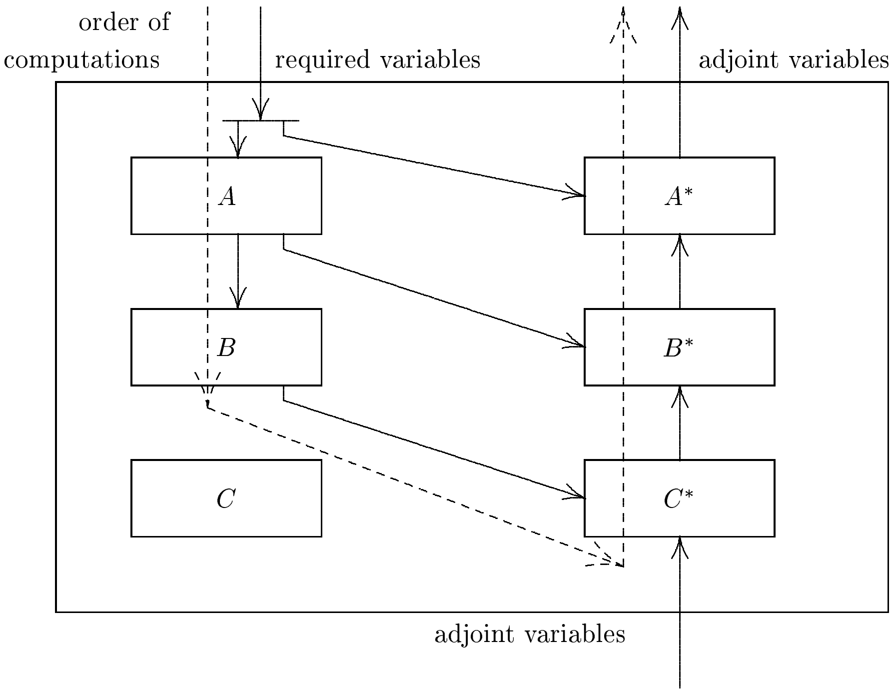

Figure 1: Graph of an adjoint block of statements (A,B, and C are statements, A^*, B^*, and C^* are the corresponding adjoint statements)

$B'C^*$. In this case the scheme given above is not suitable and instead a more general scheme is applied:

$$A'B' \;\; C^* \, R''A''B^* \, R'''A^* \tag{2}$$

where R recomputes overwritten input variables of the whole block which are required for the following adjoint statements. TAMC inserts these extra recomputations whenever necessary and issues a level-1 warning.

The recomputations R consist of statements executed before the block. Two exceptions exist. If the block is the body of a routine, R consists of statements to restore values from dynamic memory. The values are stored at the very beginning of the adjoint routine. If the block is the kernel of a loop for which a recomputation warning was given then R consists of statements to restore values from dynamic memory. The values are stored automatically at the beginning of the adjoint loop.

The algorithm is applied recursively if the statements inside the block contain blocks themselves.

3.3 Restrictions

It is intended to support the full Fortran-90 standard. But some constructs declared as obsolete in Fortran-90 or Fortran-95 will not be supported.

3.3.1 Parser and Scanner

The TAMC scanner is sensitive to white space. Thus, no blanks must occur inside identifiers (names of routines and variables), reserved words (eg. DO, FUNCTION, REAL), and operators (eg. .eq.). Identifiers and reserved words must be separated by blanks. TAMC does not always accept reserved words as identifiers.

Fortran-90 `INCLUDE` statements are not handled yet. It is recommended to use the preprocessor include (cpp or FORTRAN preprocessor). A script called change_include to replace FORTRAN-`INCLUDE` by the cpp preprocessor #include is provided in the utility.

A main program must start with an explicit PROGRAM statement.

3.3.2 Transformations

For adjoint code generation the control flow has to be reversed. This reversal becomes extremely difficult if the sequential flow is changed by specific statements. TAMC normalize the code and applies a control flow analysis to remove statements changing the control flow (other then structured). However, certain cyclic structures cannot be replaced by TAMC. In that case TAMC gives a warning. Then the adjoint code generated might be wrong. The data flow analysis may fail even if these structures occur only in a passive routines.

TAMC expects each common block to have the same number and kind of entries in each subroutine and function. A warning will be given if this condition is not fulfilled.

Recursion (Fortran-90) is not supported by TAMC.

3.3.3 Fortran-90

The restrictions on Fortran-90 code are:

- INCLUDE-statement
- named arguments
- derived types
- interfaces
- operator and function overloading
- kind parameter of constants

3.4 Known problems

3.4.1 All modes

For both modes the problems are:

1. Intrinsics function names passed as procedure parameters will not be handled.
2. Comments in front of continuation lines are not accepted in fixed source format
3. Common block variables which are passed as arguments to a called routine may establish an indirect aliasing. The data dependence analysis may fail in this case. Note, the Fortran-90 standard restricts the variable to be only used in this case.
4. The generic intrinsic functions `MIN` and `MAX` and its specific versions must have two arguments.

3.4.2 Reverse mode

1. Assignment to a part of an array is handled as to the whole array. This may cause neglecting necessary statements.

code	wrong recalculation part:
`A(1) = ...`	
`B(1) = ...  A(1) ...`	
`A(2) = ...`	`A(2) = ...`
`FC = A(1)**2 + A(2)**3 + B(1)`	

   Exchanging statement 2 and 3 yields:

code	correct recalculation part:
`A(1) = ...`	
`A(2) = ...`	`A(1) = ...`
`B(1) = ...  A(1) ...`	`A(2) = ...`
`FC = A(1)**2 + A(2)**3 + B(1)`	

   As an exemption TAMC will produce correct code whenever an array is not referenced between two definitions of different components of this array.

2. TAMC keeps track of potentially partly defined arrays inside each routine but assumes each routine defines array completely. If a routine defines an array only partly the dependence analysis may fail. In order to avoid this problem one can insert a dummy assignment. The example below shows a subroutine which defines `A` only partly.

   ```
   SUBROUTINE PART( N, A )
   INTEGER N
   REAL A(N)
   A(1) = A(1) !  dummy
   A(4) = 3.0
   END
   ```

   By inserting the first statement, TAMC recognizes that the array `A` must still be set in front of a call to the subroutine `PART`, because then `A` also appears as input.

3. I/O-statements may require a given `CLOSE` statement. This is not always included in the adjoint code Moving the close statement behind the last read or write statement of the unit ensures it will be included.

4. Routines declared selfadjoint by a directive (see sec. 5.1) can not be handled if the '-jacobian' option is used.

5. Assumed size arrays cannot be stored, because their length cannot be computed during compile time.

6. Adjoint code for DOWHILE loops is not correct in some cases.

4 Options

4.1 Overview

The options of TAMC can be divided in five classes. Remote access options (Tab. 1) are only excepted by the script managing the communication between the local host and the server.

NAME	TYPE	DEFAULT
-server	string	klima47.dkrz.de
-ssh	boolean	off
-mail	boolean	off
-version	string	NONE

Table 1: Remote access options

Many common FORTRAN-77 options (Tab. 2) are also accepted by TAMC to ease the use of TAMC.

NAME	TYPE	DEFAULT
-l	string	NONE
-u	boolean	off
-i2	boolean	off
-i4	boolean	on
-i8	boolean	off
-r4	boolean	on
-r8	boolean	off
-e	boolean	off
-N	integer	72
-U	boolean	off
-V	boolean	off
-free	boolean	off
-fixed	boolean	off

Table 2: Common Fortran options accepted by TAMC

Output options (Tab. 3) can be specified for changing output of TAMC.

NAME	TYPE	DEFAULT
-maxerrors	integer	30
-info	boolean	on
-warning	boolean	on
-warn_step	boolean	off
-direc	string	NONE
-help	-	

Table 3: Output options

The most important options are the transformational options (Tab. 4) They specify the kind of transformation to be applied, information needed for the transformation, and details about the code being produced. Some of the transformational options must be given (toplevel,input,output).

NAME	TYPE	DEFAULT
-toplevel	string	model
-input	string	[]
-output	string	[]
-reverse	boolean	off
-adjoint	boolean	off
-forward	boolean	off
-pure	boolean	off
-jacobian	integer	1
-append	boolean	off
-ldg	boolean	off
-arglist	boolean	off
-unit	integer	60
-closetapes	boolean	off
-split	boolean	off
-auxmark	string	h
-admark	string	ad
-modmark	string	md
-ftlmark	string	g_
-postfix	boolean	off

Table 4: Transformation options

The source and target code options (Tab. 5) allow for specifying the Fortran dialect of the sources and the generated code.

NAME	TYPE	DEFAULT
-f90	boolean	off
-hpf	boolean	off
-dynamic	boolean	off
-vector	boolean	off
-doenddo	boolean	on
-dowhile	boolean	on
-ivdep	boolean	off
-keep	boolean	off

Table 5: Source and target code options

Boolean type options are enabled by the option itself and disabled by a 'no' in front of the option name. E.g. -nowarning will suppress all warnings. A string must be surrounded by quotation marks if it contains blanks. The dummy file name 'NONE' means that no file will be generated. Options may be given in any order.

4.2 Explanation

-server *address* Specifies the server for remote access.

-ssh Use secure shell (ssh) instead of remote shell (rsh) access. Use the script setup_ssh to submit your public key. (Only available for server trough.mit.edu .)

-mail Use email access instead of remote shell. The email will be replied automatically. The script rtamc should be used to extract the generated files out of the reply.

-version *number* Use a specific version instead of the current one.

-l *file* This option sets the name of a listing file on which TAMC will write flow information about each routine.

-u Set all variables undefined by default.

-i2 -i4 -i8 Make integers be 2 (4,8) bytes by default. This value is only used to store integer variables on memory or user defined tapes.

-r4 -r8 Make reals be 4 (8) bytes by default. This value is only used to store real variables on memory or user defined tapes.

-e Accept and write extended source lines up to 132 columns long.

-N *col* Accept and write source lines up to *col* columns long.

-U Distinguish between upper and lower case letters.

-V Print TAMC version and date.

-fixed Specifies that all source files are in fixed source format.

-free Specifies that all source files are in free source format. Enables option -f90.

-toplevel *name* This option defines the name of the subroutine to be differentiated (top-level routine).

-input *list* This option defines the independent variables (blank or comma separated list of names, given as a string). The variables must be arguments or entries of local common blocks of the top-level routine. If this option is missing all real, double precision and complex arguments of the top-level routine are assumed to be independent. The corresponding adjoint variables must be initialized before calling the generated derivative code.

-output *list* This option defines the dependent variables (blank or comma separated list of names, given as a string). The variables must be arguments or entries of local common blocks of the top-level routine. If this option is missing all real, double precision and complex arguments of the top-level routine are assumed to be dependent. The corresponding adjoint variables must be initialized before calling the generated adjoint code.

-reverse Construct code to compute derivatives in reverse mode. The code computes the dependent variables and the derivatives of the independent variables. If the option -pure is given only derivatives are computed.

-adjoint This option is a synonym for -reverse (obsolete).

-forward Construct code to compute derivatives in forward mode. The code computes the dependent variables and the derivatives of the independent variables (tangent linear and function code).

-pure By default the adjoint and tangent linear code computes the dependent variables and their derivatives. If this option is given only derivatives will be computed. In reverse mode the dependent variables and tapes are then computed by a modified top-level routine.

-jacobian *int* By default TAMC generates code, which computes the product of the Jacobian times a vector from the left (reverse mode) or the right (forward mode). This option sets the number of vectors to be multiplied. If reverse mode is specified and this number is set to the number of dependent variables the full Jacobian will be computed. By specifying forward mode and supplying the number of independent variables, the full Jacobian will be computed. In both cases the derivative variables must be initialized as the identity matrix.

-append Append index of jacobian dimension. Per default the dimension denoting the different vectors to be multiplied are the first dimension of each derivative variable. Please note, this option cannot be used if an assumed size array is active.

-ldg Add leading dimension arguments for each derivative argument. The argument follows directly the derivative argument. This option has no effect if the option -append is given or the option -jacobian is missing.

-arglist If set all arguments of the given top-level routine will be arguments of the adjoint top-level routine. Thus, the form of the constructed argument list does not depend on which of the arguments is required by the adjoint code.

-unit *int* This option defines the lowest unit number used for file tapes (60 by default). If the option -closetapes is given this unit is used for all file tapes.

-closetapes If a tape is implemented by files, each file will only be opened at the beginning of the routine (modified forward, adjoint) in which the tape is defined and closed at its end. If this option is given each file will be opened before every read and write, and closed afterwards. This saves buffer memory, but may be slower.

-split Common subexpressions in adjoint and tangent linear code will be avoided if this option is present. The derivative code is speed up if compiled by a non optimizing compiler.

-auxmark mark The names of auxiliary variables, common blocks, and routines consist of an existing name and this mark. E.g. the default setting 'h' means, the auxiliary variable to the variable x will be xh. Please note, the mark will always be appended to an existing name.

-admark mark The names of adjoint variables, common blocks, and routines consist of the original name followed by this mark. E.g. the default setting 'ad' means, the adjoint variable to the variable x will be adx. The mark will be appended if the option -postfix is given.

-modmark mark The names of modified routines consist of the original name followed by this mark. E.g. the default setting 'md' means, the modified routine to the routine `sub` will be `mdsub`. The mark will be appended if the option -postfix is given.

-ftlmark mark The names of tangent linear variables, common blocks, and routines consist of the original name followed by this mark. E.g. the default setting 'g_' means, the tangent linear variable to the variable x will be g_x. The mark will be appended if the option -postfix is given.

-postfix The mark of derivative names leads the original name by default. If set, the mark will be appended, e.g. the adjoint variable of x will be xad.

-f90 The generated code may contain Fortran-90 extensions to FORTRAN-77. This option enables -vector, -doendo, -dowhile, and -dynamic. In addition most Fortran-90 intrinsic functions will be accepted and used. If any of the source files is in free source format this option will be on automatically.

-hpf TAMC reads and writes High Performance Fortran extensions. This options enables -f90. Right now only INDEPENDENT, DISTRIBUTION, and SHADOW directives are handled.

-dynamic If the FORTRAN compiler supports automatic arrays TAMC may generate local automatic arrays, i.e. arrays with at least one dimension given as non constant specification expression.

-vector TAMC may generate Fortran-90 array assignments.

-doendo TAMC may generate Fortran-90 DO ENDDO statements.

-dowhile TAMC may generate Fortran-90 DO WHILE ENDDO statements. If this option is off DO-WHILE constructs in the source code will be translated into IF THEN GOTO constructs.

-ivdep Write 'CDIR$ IVDEP' directives in front of generated loops which do not have a vector dependency.

-keep Evaluating the data dependence between independent and dependent variables, TAMC detects statements which are not required to compute the dependent variables. These might be statements like `PRINT` but also those to compute diagnostic variables used only for monitoring. If this option is given TAMC will include these statements. This option has no effect in reverse mode.

-maxerrors *int* Sets the maximal number of syntax errors reported, 30 by default.

-info If this option is turned off information will no longer be given.

-warning If this option is turned off warnings will be suppressed.

-warn_step Warnings about non differentiable mapping between independent and dependent variables are given if switched on. Active calls of intrinsic functions non differentiable at some points will be noticed. If a condition depends on the independent variables directly or indirectly, the mapping to the dependent variables may be also be non differentiable at some points.

-direc *file* Directives readable by TAMC are written to the specified file.

-help A short explanation of each option and its default setting is given.

5 Directives

TAMC accepts directives, there are two classes:

- directives outside subroutines and functions
- directives inside the statement part of subroutines and functions.

A TAMC directive begins with 'CADJ' in the first four columns of a line. Lower or upper case letters might be used. Continuation lines of TAMC directives start with 'CADJ &' in the first six columns of a line. The syntax and the semantic of these directives are described in the following sections.

5.1 Outside subroutines and functions

The following directives specify information about a routine not included in the source code. They have to be positioned outside the code of a routine.

The first directive reserves the subroutine name.

```
CADJ SUBROUTINE name
CADJ SUBROUTINE name INPUT    = list_of_integers
CADJ SUBROUTINE name OUTPUT   = list_of_integers
```

The second directive sets the input arguments of the routine argument list, while the third sets the output arguments. The list of integers must be a comma separated list. Each number specifies the position of an argument.

The next directives are only required for an active subroutine.

```
CADJ SUBROUTINE name MODNAME  = modified_name
CADJ SUBROUTINE name ADNAME   = adjoint_name
CADJ SUBROUTINE name FTLNAME  = tangent_linear_name
CADJ SUBROUTINE name ACTIVE   = list_of_integers
CADJ SUBROUTINE name DEPEND   = list_of_integers
CADJ SUBROUTINE name ADKILL   = list_of_integers
```

The routine will be considered as active if its adjoint or tangent linear name has been specified using the keyword `ADNAME` or `FTLNAME`. If the tangent linear name is the same as the routine name, then the routine is assumed to be linear and the routine itself is called to compute derivatives in forward mode. If the adjoint name is the same as the routine name the routine is assumed to be linear and self adjoint and the routine itself is called to compute derivatives in reverse mode.

Active arguments are set using the keyword `ACTIVE`. If the adjoint routine requires arguments of the original routine they have to be specified using the keyword `DEPEND`. The keyword `ADKILL` specifies arguments overwritten by the adjoint routine but not used later, eg. working arrays.

To set information about the flow through a common block *common_name* of a routine *name* similar directives can be used:

```
CADJ SUBROUTINE name COMMON common_name INPUT   = list_of_integers
CADJ SUBROUTINE name COMMON common_name OUTPUT  = list_of_integers
CADJ SUBROUTINE name COMMON common_name ACTIVE  = list_of_integers
CADJ SUBROUTINE name COMMON common_name DEPEND  = list_of_integers
CADJ SUBROUTINE name COMMON common_name ADKILL  = list_of_integers
```

In this case the numbers specify the entries of the common block counted from left to right.

Similar directives exists for functions, i.e.:

```
CADJ FUNCTION name
CADJ FUNCTION name INPUT    = list_of_integers
CADJ FUNCTION name OUTPUT   = list_of_integers
CADJ FUNCTION name MODNAME  = modified_name
CADJ FUNCTION name ADNAME   = adjoint_name
CADJ FUNCTION name FTLNAME  = tangent_linear_name
CADJ FUNCTION name ACTIVE   = list_of_integers
CADJ FUNCTION name DEPEND   = list_of_integers
CADJ FUNCTION name ADKILL   = list_of_integers
```

In case a subroutine does not have data flow dependences which can be specified by variables, the two directives

```
CADJ SUBROUTINE name REQUIRED
CADJ SUBROUTINE name INFLUENCED
```

can be used to declare a subroutine to be required for the computation of the dependent variables or influenced by the independent variables. If both are specified for a subroutine and an adjoint name has been specified too then this adjoint subroutine will be called in the adjoint code.

A common block name is reserved by the directive:

```
CADJ COMMON common_name
```

The name of an adjoint (tangent linear) common block can be specified explicitly by:

```
CADJ COMMON common_name ADNAME   = adjoint_name
CADJ COMMON common_name FTLNAME  = tangent_linear_name
```

5.2 Inside statement part

5.2.1 Storing variables

If a variable of the forward code is required for the adjoint code TAMC will by default construct code to recompute it. Alternatively, TAMC will generate code to store and restore values if a STORE-directive has been inserted. The syntax of a STORE-directive is:

```
CADJ STORE lhs_list = tape [, KEY = expr] [, BYTE = const_int_expr] [, SHAPE = const_array_expr]
```

where *lhs_list* is a comma separated list of variable names or array segments (Fortran-90 syntax) and *tape* is a storage tape defined by a directive (see next section). The optional `KEY` extension defines the record number. You are responsible to define a unique record number. Using the optional `BYTE` extension values can be stored in a precision different from the declared precision of the variables. This is useful to save memory resources, i.e. by storing a `REAL*8` variable on a `REAL*4` variable. Please note, the `BYTE` extension can only be used for common block tapes right now. By specifying the shape of an array (keyword `SHAPE`) by a constant array expression (Fortran-90 syntax) TAMC can handle automatic arrays.

TAMC will generate code to store and to restore every given variable or array segment. If the variable is not required the directive is ignored and a warning is given. Whenever the variable to be stored is an array with no segment specification the whole array is stored. The store-directive should be positioned directly after the last statement which changes the stored variable and before the statement which references the variable at first. Inside a loop, the directive should occur in front of the loop kernel or inside the kernel code Store-directives at the end of a loop kernel will be ignored.

The example below shows the DO-loop of section 3.2.3 which produces a conflict because the required variable `FAC` is overwritten:

<ins>code a</ins>

```
CADJ INIT mytape = 'myfile'
      FAC = 1.0
      DO I = 1, N
CADJ STORE FAC = mytape
         FAC = FAC * X(I)
      END DO
```

<ins>code b</ins>

```
CADJ INIT mytape = COMMON, N
      FAC = 1.0
      DO I = 1, N
CADJ STORE FAC = mytape, KEY=I
         FAC = FAC * X(I)
      END DO
```

Here, the STORE-directive at the beginning of the loop kernel solves the conflict (see next section for description of INIT-directives). The tape is defined as a file-tape (code a) or a common block tape (code b). Instead of recomputing code, a READ statement (adjoint code a) or an assignment (adjoint code b) is generated to provide the required value of `FAC`.

<ins>adjoint code a</ins>

```
open(60,file='myfile_1_model_fac',access='direct',recl=8)
⋮
adfac = 0
do i = n, 1, -1
   call getkey( 4,1,i-1,0,adsubcl(1),nrec )
   read(60,REC=nrec) fac
   adx(i) = adx(i)+adfac*fac
   adfac = adfac*x(i)
end do
⋮
close(60)
```

<ins>adjoint code b</ins>

```
common /cadfac/ fac_h(n)
⋮
adfac = 0
do i = n, 1, -1

   fac = fac_h(i)
   adx(i) = adx(i)+adfac*fac
   adfac = adfac*x(i)
end do
⋮
```

The tape is written by a corresponding WRITE statement (modified code a) or an assignment (modified code b) inside the adjoint or modified code.

modified code a

```
open(60,file='myfile_1_model_fac',access='direct',recl=8)
⋮
fac = 1.
do i = 1, n
   call getkey( 4,1,i-1,0,adsubcl(1),nrec )
   write(60,REC=nrec) fac
   fac = fac*x(i)
end do
⋮
close(60)
```

modified code b

```
common /cadfac/ fac_h(n)
⋮
fac = 1.
do i = 1, n

   fac_h(i) = fac
   fac = fac*x(i)
end do
⋮
```

By adding a `KEY=I` argument to the STORE-directive in code b the call of `getkey` is avoided and instead the given record number is used (adjoint and modified code b). Please note as stated above, the record number must be unique. In this case meaning, the piece of code must be executed only once inside the function computing code. If it is part of another loop or the routine containing the code is called multiple times during function evaluation, another record number must be used. Nevertheless, under specific circumstances this extension can be used to fool TAMC by using non unique record numbers and thus overwriting tape values and restoring wrong values This save memory resources at the cost of inexact adjoint values due to inexact required variables.

5.2.2 Initialize storing tapes

Values are stored on a tape. Tapes are defined by INIT-directives with different attributes.

```
CADJ INIT tape = MEMORY
CADJ INIT tape = COMMON, const_int_expr
CADJ INIT tape = const_string_expr
CADJ INIT tape = USER, const_string_expr
```

Variables stored on a `MEMORY` tape will be held in dynamic memory. The memory will be allocated (deallocated) from the heap by the library routine `DMPUT` (`DMFREE`) (see appendix A).

Static memory is used for variables stored on a `COMMON` block tape. For each variable stored, a common block having only one entry will be generated. To use this kind of tape you have to know the maximal number of records written for one variable. This number has to be specified as a constant integer expression[2] (*const_int_expr*).

The third alternative defines a tape realized by files on the local disk. For each variable stored the name of the file holding the records of values consists of the given string (*const_string_expr*), a number identifying the store-directive, the subroutine name, and the variable name (see OPEN statement in example given above). Storing is implemented by `READ` and `WRITE` statements using a direct access file. TAMC tries to keep track of all used unit numbers in the code and generates only not used numbers. But unit numbers might be assigned dynamically or used inside code not visible to TAMC. To take this into account an option -unit exists to ensure generating only unit numbers greater or equal than a given number (60 per default). If the option -closetapes was given OPEN and CLOSE statements will be inserted for every I/O operation, and their unit numbers will always be the one given by the option -unit.

[2]The value of a constant integer expression can be evaluated at compile time. See Fortran manual for details.

External storage handling routines will be used if a tape has the attribute USER. A call to the subroutine ADWRITE will be generated to store the variable, and a call to ADREAD to restore the variable. Calls to ADOPEN and ADCLOSE are also generated. These routines must be provided by the user. Examples can be found in the directory **storage**. The argument lists of these routines are explained in appendix B. If a `const_string_expr` is specified this string will replace the string AD in the name of user defined tapes, allowing different tapes to have different interfaces.

A tape from which required variables are read during adjoint computations must first be written. The code to compute and write these values (tape computations) is inserted into the adjoint code. (If -pure is given these tape computations might be done in the modified top-level routine.) Tape computations start at the position of the tape-init-directive, or more precisely: they start from the intermediate results being valid as the function computation reaches this statement. If the tape is initialized at the very beginning of the top-level routine then recomputations start from the initial values of the independent variables. By positioning tape initialization directives checkpoints can be arbitrary set. The only limitation is, that a tape initialization should not occur before a store-directive to a previously initialized tape.

The code template given below shows the setup for a two level checkpointing. Here, the loop used previously as

code

```
CADJ  INIT outer_tape = 'out'
      fac = 1.0
      DO i = 1, n, nc
CADJ     STORE fac = outer_tape
CADJ     INIT inner_tape = common, nc
         DO ic = 1, nc
CADJ        STORE fac = inner_tape
            fac = fac * x(i+ic-1)
         END DO
      END DO
```

an example is split into two nested loops. Two tapes are used, both define checkpoints from which intermediate results are recomputed. For the outer tape (outer_tape) computations start all over, while for the inner tape (inner_tape) they start at the checkpoints defined by its init-directive. The generated adjoint code contains two main loops. The first loop computes the outer tape by storing intermediate result every nc loop path. The second loop is the reverse order adjoint outer loop. Inside this loop, starting from the previously stored values, intermediate results are computed again and stored on the inner tape. Then the reverse order adjoint inner loop is computed using the last intermediate results.

The adjoint code shown above uses $\frac{n}{nc}$ records of the outer tape and nc records of the inner tape. In Tab. 6 memory and run time requirements for the three different strategies are listed. For $n = nc * nc \gg 1$, two level checkpointing requires $2\sqrt{n}$ records and the additional time of almost one function computation. An additional level of checkpointing would reduce memory requirements to $3\sqrt[3]{n}$ and would cost one more function computation [3].

For most nonlinear large scale applications with $n > 100$ adjoint code without recomputations would require a prohibitively huge tape. On the other hand recomputing every intermediate result takes endless if the function computation itself already requires significant computer time. The solution is multi-level checkpointing. Even

[3]Logarithmic growth of memory and run time can be archived in theory [Gri92].

adjoint code

```
OPEN(60,file='myfile_1_model_fac',access='direct',recl=8)
fac = 1.0
DO i = 1, n, nc
   WRITE(60,nrec=1+(i-1)/nc) fac
   DO ic = 1, nc
      fac = fac * x(i+ic-1)
   END DO
END DO

DO i = 1+nc*NINT((n-1)/nc-0.5), 1, -nc
   READ(60,nrec=1+(i-1)/nc) fac
   DO ic = 1, nc-1
      fac_h(ic) = fac
      fac = fac * x(i+ic-1)
   END DO
   ic = nc
   fac_h(ic) = fac
   DO ic = nc, 1, -1
      fac = fac_h(ic)
      adx(i+ic-1) = adx(i+ic-1) + adfac * fac
      adfac = adfac * x(i+ic-1)
   END DO
END DO
CLOSE(60)
```

strategy	records	time
no recomputations	n	$T_{ad} + T_f$
two level checkpointing	$\frac{n}{nc} + nc$	$T_{ad} + T_f + \frac{nc-1}{nc} T_f$
no storing	0	$T_{ad} + \frac{n-1}{2} T_f$

Table 6: Relative time and storage requirements (T_f : run time of function code, T_{ad} : run time of pure adjoint code)

for 3 levels the additional computer time is acceptable. But the reduction in storage requirements is substantial compared to the 'no recomputation strategy'.

5.2.3 Specifying loop data dependence

Two statements have a data dependence if they cannot be executed simultaneously due to conflict uses of the same variable. There are three kinds of data dependences: flow dependence, anti flow dependence, and output dependence. Two statements have a flow dependence if the first statement defines a variable which is used by the second statement. An anti flow dependence exists if the first statement uses a variable which is redefined later by the second statement. If both statements define the same variable then an output dependence exists between them. Loop-carried data dependence are dependences between statements due to their repeated execution in a loop kernel.

The necessary recomputations inside a loop depend very much on its loop-carried data dependences. The following directives specify the flow dependence of loops.

If TAMC gives a recomputation warning about a loop, but the loop kernels can be executed independently from each other, insert the following HPF directive in front of the loop:

`!HPF$ INDEPENDENT [, NEW (` *variable_list* `)]`

TAMC will then construct adjoint code taking the non existing dependency into account and generates an adjoint loop which computes the adjoint kernel in the original order. The optional variable list should contain all variables and arrays define inside the kernel and being local to each loop path (see HPF specification for details). If the loop has only flow dependences and no anti flow dependences the following directive can be used to suppress the warning and the generation of unnecessary recomputations.

```
CADJ LOOP = SEQUENTIAL
```

5.2.4 Specifying general assignments

TAMC can be explicitly forced to generate adjoint code for a general assignment by inserting the directive

```
CADJ GENERAL
```

in front of an assignment. For the code generated see sec. 3.2.1.

5.2.5 Converging iterations

Alternative adjoint code can be generated by TAMC for loops doing an iteration to solve an equation for variables. By inserting the directive:

`CADJ LOOP = ITERATION` *variables* `[=` *tape*`]`

where *variables* is a comma separated list of variable names, TAMC assumes the equation is solved exactly with respect to machine precision. The adjoint code generated does also an iteration to solve the corresponding adjoint equation. The advantage of the alternative code is, that no variables must be restored or recomputed during the iteration. To compute the adjoint code only the solution of the original iteration must be available. By default recomputation of the solution is inserted. By specifying a *tape* the solution computed in the function code is stored in the modified code and restored in front of the alternative adjoint code.

Please note, if the equation is not solved exactly the computed derivatives are biased.

6 Intrinsic functions

6.1 Elemental intrinsic procedures

6.1.1 Generic and single precision intrinsics

All standard FORTRAN-77, most Fortran-90, and some common intrinsic functions are implemented. The generic and single precision intrinsic functions and their first derivatives are shown in table 7.

intrinsic function	fkt. not def.	derivative arg. 1	derivative arg. 2	deriv. not def.
SIN(x)		COS(x)		
COS(x)		- SIN(x)		
TAN(x)	$x = \frac{\pi}{2} + n\pi, n \in I\!N$	1./(COS(x)**2)		
SINH(x)		COSH(x)		
COSH(x)		SINH(x)		
TANH(x)		1./COSH(x)**2		
ASIN(x)	$\lvert x\rvert > 1$	1./SQRT(1.-x**2)		$x = 1$
ACOS(x)	$\lvert x\rvert > 1$	- 1./SQRT(1.-x**2)		$x = 1$
ATAN(x)		1./(1.+x**2)		
ATAN2(x,y)	$x, y = 0$	y / (x**2+y**2)	- x / (x**2+y**2)	
EXP(x)		EXP(x)		
LOG(x)	$x \leq 0$	1./x		
ALOG(x)	$x \leq 0$	1./x		
LOG10(x)	$x \leq 0$	ALOG(EXP(1.))/x		
ALOG10(x)	$x \leq 0$	ALOG(EXP(1.))/x		
SQRT(x)	$x < 0$	0.5 /SQRT(x)		$x = 0$
ABS(x)		SIGN(1.,x)		$x = 0$
FLOAT(x)		1.		
REAL(d)		1.		
SNGL(d)		1.		
MAX(x,y)		0.5+SIGN(0.5,x-y)	0.5-SIGN(0.5,x-y)	$x = y$
AMAX0(x,y)		0.	0.	
AMAX1(x,y)		0.5+SIGN(0.5,x-y)	0.5-SIGN(0.5,x-y)	$x = y$
MIN(x,y)		0.5+SIGN(0.5,y-x)	0.5-SIGN(0.5,y-x)	$x = y$
AMIN0(x,y)		0.	0.	
AMIN1(x,y)		0.5+SIGN(0.5,y-x)	0.5-SIGN(0.5,y-x)	$x = y$
AINT(x)		0.		$x = \pm n, n \in I\!N$
ANINT(x)		0.		$x = \frac{1}{2} \pm n, n \in I\!N$
AMOD(x,y)		0.	0.	$x = ny, n \in I\!N$
MOD(x,y)		0.	0.	$x = ny, n \in I\!N$
SIGN(x,y)		SIGN(1.,x)*SIGN(1.,y)	0.	$x = 0 \vee y = 0$
DIM(x,y)		- DIM(x,y)	DIM(x,y)	$x = y$
RAN(x)		0.		
RANF(x)		0.		
SECOND(x)		0.		

continued on next page

continued from previous page				
intrinsic function	fkt. not def.	derivative arg. 1	derivative arg. 2	deriv. not def.

Table 7: Implemented generic and single precision intrinsic functions. The ranges where the function is not defined, their derivatives with respect to argument 1 and 2, and the points of non differentiability are shown.

The second row gives the range of the arguments for which the intrinsic function is not defined. The derivatives with respect to argument 1 and 2 are shown in the next rows. In the last row the points of non differentiability are given which are inside the range of definition.

6.1.2 Double precision result intrinsics

The double precision intrinsics are shown in table 8.

intrinsic funct.	fkt. not def.	derivative arg. 1	derivative arg. 2	deriv. not def.
DSIN(x)		DCOS(x)		
DCOS(x)		- DSIN(x)		
DTAN(x)	$x = \frac{\pi}{2} + n\pi$	1.d/(DCOS(x)**2)		
DSINH(x)		DCOSH(x)		
DCOSH(x)		DSINH(x)		
DTANH(x)		1.d/DCOSH(x)**2		
DASIN(x)	$\|x\| > 1$	1.d/DSQRT(1.d-x**2)		$x = 1$
DACOS(x)	$\|x\| > 1$	-1.d/DSQRT(1.d-x**2)		$x = 1$
DATAN(x)		1.d/(1.d+x**2)		
DATAN2(x,y)	$x, y = 0$	y / (x**2+y**2)	- x / (x**2+y**2)	
DEXP(x)		DEXP(x)		
DLOG(x)	$x \leq 0$	1.d/x		
DLOG10(x)	$x \leq 0$	DLOG(DEXP(1.d))/x		
DSQRT(x)	$x < 0$	0.5d /DSQRT(x)		$x = 0$
DABS(x)		DSIGN(1.d,x)		$x = 0$
DBLE(x)		1.d		
DPROD(x,y)		y	x	
DMAX1(x,y)		0.5d+DSIGN(0.5d,x-y)	0.5d-DSIGN(0.5d,x-y)	$x = y$
DMIN1(x,y)		0.5d+DSIGN(0.5d,y-x)	0.5d-DSIGN(0.5d,y-x)	$x = y$
DINT(x)		0.		$x = \pm n, n \in I\!N$
DNINT(x)		0.		$x = \frac{1}{2} \pm n, n \in I\!N$
DMOD(x,y)		0.	0.	$x = ny, n \in I\!N$
DSIGN(x,y)		DSIGN(1.,x)*DSIGN(1.,y)	0.	$x = 0 \vee y = 0$
DDIM(x,y)		- DDIM(x,y)	DDIM(x,y)	$x = y$
RAND(x)		0.		

Table 8: Implemented double precision intrinsic functions and their first derivatives

6.1.3 Complex intrinsics

The intrinsic functions having a complex argument or returning a complex result are shown in table 8. The adjoint matrix to a complex matrix is the conjugate complex transposed matrix. Therefor, if the LHS variable

intrinsic function	fkt. not def.	derivative arg. 1	derivative arg. 2	deriv. not def.
REAL(z)		REAL($\cdots$)		
AIMAG(z)		AIMAG($\cdots$)		
CMPLX(x,y)		CMPLX($\cdots$,0.)	CMPLX(0.,$\cdots$)	
CONJG(z)		CONJG($\cdots$)		
CABS(z)		(z/CABS(z)) $\circ$		$z = 0$
CSIN(z)		CCOS(z)		
CCOS(z)		- CSIN(z)		
CEXP(z)		CEXP(z)		
CLOG(z)	$z = 0$	1./z		
CSQRT(z)		0.5/CSQRT(z)		$z = 0$

Table 9: Implemented complex intrinsic functions and their first derivatives

of an assignment is complex and a derivative of the RHS is complex, then the adjoint assignment is generated by using the conjugate complex of the derivative. This is implemented by a call of the intrinsic function `CONJG`.

6.1.4 Integer and character result intrinsics

The implemented intrinsic functions returning an integer or character result are listed in table 10. Their

intrinsic function	
ACHAR	IACHAR
ADJUSTL	ADJUSTR
BTEST	CEILING
INT	NINT
IDINT	IDNINT
IFIX	IABS
IDIM	MOD
LEN	INDEX
CHAR	ICHAR
MAX0	MAX1
MIN0	MIN1

Table 10: Implemented integer and character result intrinsic functions

derivatives are always zero.

6.1.5 MERGE

The merge intrinsic function merges two array expression subject to a mask. Each element of the output array expression contains the value of the first array expression if the element of the mask argument is true and otherwise the second array expression. Consider the following assignment:

```
z = MERGE(x,y,mask=m)
```

The tangent linear and the adjoint statements generated are:

tangent linear code	adjoint code

`g_z = MERGE(g_x,g_y,m)`	`where (m)`
	`   adx += adz`
	`else`
	`   adx += adz`
	`end where`
	`adz = 0.0`

6.2 Transformational intrinsic functions

6.2.1 COUNT, MAXLOC, MINLOC, SELECT_INT_KIND, SELECT_REAL_KIND

These intrinsic functions return an integer result and are always passive.

6.2.2 ALL, ANY

These intrinsic functions return a logical result and are always passive.

6.2.3 REPEAT, TRIM

These intrinsic functions return a character result and are always passive.

6.2.4 CSHIFT

Assume the following array assignment is given, where `x` and `y` are active arrays:

```
y = CSHIFT(x,shift=s,dim=d)
```

The CSHIFT intrinsic function performs a circular shift of an array expression. The shift-argument specifies the number of positions each index is shifted to the left (s positive) or right (s negative). Elements shifted out at one end are shifted in at the other end. In general `s` can be a rank 1 or a rank 0 integer expression. Please note, named arguments are not supported yet by TAMC.

The generated tangent linear code depends on the given options, it is shown in Tab. 11. If the option -jacobian

options	tangent linear code
	`g_y = CSHIFT(g_x,s,d)`
-jacobian	`g_y = CSHIFT(g_x,s,d+1)`
-jacobian, -append	`g_y = CSHIFT(g_x,s,d)`

Table 11: Tangent linear code to CSHIFT

is given each derivative variable gets an additional dimension. If at the same time -append is not given this dimension is the first one. Thus, in order to shift the same dimension of the derivative array expression the dimension argument has to be increased by one in the derivative code.

The corresponding adjoint operation is a circular shift in opposite direction. The generated adjoint is shown in Tab. 12[4].

options	adjoint code
	`adx += CSHIFT(ady,-s,d)`
-jacobian	`adx += CSHIFT(ady,-s,d+1)`
-jacobian, -append	`adx += CSHIFT(ady,-s,d)`

Table 12: Adjoint code to CSHIFT

6.2.5 EOSHIFT

Assume the following array assignment is given, where `x` and `y` are active arrays:

```
y = EOSHIFT(x,shift=s,bound=b,dim=d)
```

The EOSHIFT intrinsic function performs a end-off shift of an array expression. The shift-argument specifies the number of positions each index is shifted to the left (s positive) or right (s negative). Elements are shifted out at one end and the value `b` given by the bound argument is shifted in at the other end. In general `s` can be a rank 1 or a rank 0 integer expression. Please note, named arguments are not supported yet by TAMC.

The generated tangent linear code depends on the given options, it is shown in Tab. 13. If the option -jacobian is given each derivative variable gets an additional dimension. If at the same time -append is not given this dimension is the first one. Thus, in order to shift the same dimension of the array expression the dimension argument has to be increased by one in the derivative code. The new boundary argument `bn` is the neutral element of addition with respect to the type of the array argument (0.0 for REAL, 0.d0 for DOUBLE PRECISION and (0.0,0.0) for COMPLEX expression).

options	tangent linear code
	`g_y = EOSHIFT(g_x,s,bn,d)`
-jacobian	`g_y = EOSHIFT(g_x,s,bn,d+1)`
-jacobian, -append	`g_y = EOSHIFT(g_x,s,bn,d)`

Table 13: Tangent linear code to EOSHIFT

The corresponding adjoint operation is an end-out shift in opposite direction. The generated adjoint code for a passive boundary argument is shown in Tab. 14 Again the new boundary argument `bn` is zero.

If the boundary argument is active the value shifted out must be added to the corresponding adjoint variable. This can be realized by accessing the out-shifted adjoint value prior to the end-out shift.

[4] a += b is a shortcut for a = a + b.

options	adjoint code
	`adx += EOSHIFT(ady,-s,bn,d)`
-jacobian	`adx += EOSHIFT(ady,-s,bn,d+1)`
-jacobian, -append	`adx += EOSHIFT(ady,-s,bn,d)`

Table 14: Adjoint code to EOSHIFT

```
adb += out_shifted_value(ady,s,d)
adx += EOSHIFT(ady,-s,bn,d)
```

Because there is no obvious way to implement out_shifted_value TAMC is not yet able to generate this code.

6.2.6 SUM

Assume the following assignment is given, where `x` and `y` are active:

```
y = SUM(x,dim=d,mask=m)
```

Assume further the argument `dim=d`, if present, denotes the d'th dimension of the array expression with shape `dshape`. The SUM intrinsic function sums all the elements of the array `x` (along dimension `d`) that correspond to true elements of mask `m`.

The generated tangent linear code depends on the given options, it is shown in Tab. 15.

options	dim given	tangent linear code
	no	`g_y = SUM(g_x,mask=m)`
	yes	`g_y = SUM(g_x,d,m)`

-jacobian	no	`do i = 1, g_pmax`
		`   g_y(i,...) = SUM(g_x(i,...),mask=m)`
		`end do`

-jacobian	yes	`do i = 1, g_pmax`
		`   g_y(i,...) = SUM(g_x(i,...),d,m)`
		`end do`

-jacobian, -append	no	`do i = 1, g_pmax`
		`   g_y(...,i) = SUM(g_x(...,i),mask=m)`
		`end do`

-jacobian, -append	yes	`do i = 1, g_pmax`
		`   g_y(...,i) = SUM(g_x(...,i),d,m)`
		`end do`

Table 15: Tangent linear code to intrinsic function SUM

The generated adjoint is shown in Tab. 16.

options	dim given	mask given	adjoint code
	no	no	`adx(...) += ady(...)`

	no	yes	`where (m)`
			`   adx(...) += ady(...)`
			`end where`

	yes	no	`do id = 1, dshape`
			`   adx(...,ed,...) += ady(...)`
			`end do`

	yes	yes	`do id = 1, dshape`
			`   where (m')`
			`      adx(...,ed,...) += ady(...)`
			`   end where`
			`end do`

-jacobian	no	no	`do i = 1, g_pmax`
			`   adx(i,...) += ady(i,...)`
			`end do`

-jacobian	no	yes	`do i = 1, g_pmax`
			`   where (m)`
			`      adx(i,...) += ady(i,...)`
			`   end where`
			`end do`

-jacobian	yes	no	`do i = 1, g_pmax`
			`   do id = 1, dshape`
			`      adx(i,...,ed,...) += ady(i,...)`
			`   end do`
			`end do`

-jacobian	yes	yes	`do i = 1, g_pmax`
			`   do id = 1, dshape`
			`      where (m')`
			`         adx(i,...,ed,...) += ady(i,...)`
			`      end where`
			`   end do`
			`end do`

continued on next page

continued from previous page			
options	dim given	mask given	adjoint code

-jacobian, -append	no	no	`do i = 1, g_pmax`
			`   adx(...,i) += ady(...,i)`
			`end do`

-jacobian, -append	no	yes	`do i = 1, g_pmax`
			`   where (m)`
			`      adx(...,i) += ady(...,i)`
			`   end where`
			`end do`

-jacobian, -append	yes	no	`do i = 1, g_pmax`
			`   do id = 1, dshape`
			`      adx(...,ed,...,i) += ady(...,i)`
			`   end do`
			`end do`

-jacobian, -append	yes	yes	`do i = 1, g_pmax`
			`   do id = 1, dshape`
			`      where (m')`
			`         adx(...,ed,...,i) += ady(...,i)`
			`      end where`
			`   end do`
			`end do`

Table 16: Adjoint code to intrinsic function SUM

The reduction in the code becomes an array assignment in the adjoint code. A WHERE statement selects the used array components if a `mask` argument is given. The condition of the generated WHERE statement is the mask of the SUM function call. If the `dim` argument is given a reduced rank array mask `m'` is used and an additional loop is generated. The mask and the adjoint LHS variable are generated by replacing each vector expression with shape $(s_i)_{i=1,k}$ by one with shape $(s_i)_{i=1,k,k\neq d}$. The d'th component is replaced by an expression `ed` depending on the surrounding DO-loop variable.

Example:

code

```
real x(nx,ny,nz), y(nx,nz)



y = sum( x, 2, (x .gt. 0.0) )
```

adjoint code

```
real x(nx,ny,nz), adx(nx,ny,nz), ady(nx,nz)

do id = 1, ny
   where (x(:,id,:) .gt. 0.0)
      adx(:,id,:) = adx(:,id,:)  + ady
   endwhere
end do
```

6.2.7 MINVAL, MAXVAL

Consider the following assignment:

```
y = MINVAL(x,mask=m,dim=d)
```

where x is an array and $k := rank(x)$ is the rank of x.

The tangent linear and the adjoint statements generated are:

<u>tangent linear code</u>

```
INTEGER L(k)
⋮
L = MINLOC(x,m,d)
g_y = g_x(L(1),...,L(k))
y = x(L(1),...,L(k))
```

<u>adjoint code</u>

```
INTEGER L(k)
⋮
L = MINLOC(x,m,d)
adx(L(1),...,L(k)) += ady
```

Derivative code for a MAXVAL intrinsic function call is generated accordingly.

6.2.8 DOT_PRODUCT

Consider the following assignment:

```
res = DOT_PRODUCT(x,y)
```

The tangent linear and the adjoint statements generated are:

<u>tangent linear code</u>

```
g_res = DOT_PRODUCT(g_x,y) + DOT_PRODUCT(x,g_y)
```

<u>adjoint code</u>

```
adx += adres * y
ady += adres * x
adres = 0.0
```

If x is of type complex the adjoint statements generated are:

<u>adjoint code</u>

```
adx += CONJG(adres * y)
ady += CONJG(adres * x)
adres = (0.0,0.0)
```

6.2.9 MATMUL

Consider the following assignment:

```
res = MATMUL(x,y)
```

If $rank(x) = 2$ and $rank(y) = 2$ then the tangent linear and the adjoint statements generated are:

tangent linear code

```
g_res = MATMUL(g_x,y) + MATMUL(x,g_y)
```

adjoint code

```
adx += MATMUL( adres, TRANSPOSE(y) )
ady += MATMUL( TRANSPOSE(x), adres )
adres = 0.0
```

If $rank(x) = 1$ or $rank(y) = 1$ then the adjoint statements generated are:

adjoint code ($rank(x) = 1$)

```
adx += MATMUL( adres, TRANSPOSE(y) )
do i = 1, imax
   ady(i,:)  += x(i)*adres(:)
end do
adres = 0.0
```

adjoint code ($rank(y) = 1$)

```
do i = 1, imax
   adx(i,:)  += adres(i)*y(:)
end do
ady += MATMUL( TRANSPOSE(x), adres )
adres = 0.0
```

The adjoint to a matmul intrinsic function is not implemented correctly if $rank(x) = 1$ or $rank(y) = 1$.

6.2.10 PACK

Consider the following assignment:

```
res = PACK( x, mask )
```

The tangent linear and the adjoint statements are:

tangent linear code

```
g_res = PACK( g_x, mask )
```

adjoint code

```
adx += UNPACK( adres, mask, SPREAD(zero,1,SIZE(mask) )
adres = 0.0
```

Here zero is the neutral value of the data typ of the variable x, i.e '0' for INTEGER, '0.0' for REAL, and '0.d0' for DOUBLE PRECISION variables.

Consider the following assignment:

```
res = PACK( x, mask, vector )
```

The tangent linear statements are

tangent linear code

```
g_res = PACK( g_x, mask, SPREAD(zero,1,SIZE(vector) )
      + PACK( SPREAD(zero,1,SIZE(mask)), mask, g_vector )
```

and the adjoint statements are

adjoint code

```
adx += UNPACK( adres, mask, SPREAD(zero,1,SIZE(vector) )
advector += PACK( SPREAD(zero,1,COUNT(mask)), .true., adres)
adres = 0.0
```

6.2.11 UNPACK

Consider the following assignment:

```
res = UNPACK( x, mask, field )
```

The tangent linear statements are

tangent linear code

```
g_res = UNPACK(SPREAD(zero,1,n),mask,g_field) + UNPACK(g_x,mask,zero)
```

and the adjoint statements are

adjoint code

```
adx += PACK( adres, mask )
adfield += merge(adres,SPREAD(zero,1,SIZE(mask)), .not.  mask)
adres = 0.0
```

Here zero is the neutral value of the data typ of the variable `x`, i.e '0' for INTEGER, '0.0' for REAL, and '0.d0' for DOUBLE PRECISION variables.

6.2.12 PRODUCT

Although the product intrinsic function is very similar to the sum function the adjoint code is quite different. A very efficient but less accurate implementation has been chosen.

```
res = PRODUCT( x, dim, mask )
```

The tangent linear statements are

tangent linear code

```
res = PRODUCT( x, dim, mask )
g_res = MERGE( zero, SUM(g_x/x,dim,mask), res .eq.  zero )
```

and the adjoint statements are

adjoint code

```
res = PRODUCT( x, dim, mask )
adx += MERGE( zero, adres*res/x, res .eq.  zero )
```

Here zero is the neutral value of the data typ of the variable `x`, i.e '0' for INTEGER, '0.0' for REAL, and '0.d0' for DOUBLE PRECISION variables.

The adjoint product intrinsic function is not implemented yet for given `dim` or `mask` argument.

6.2.13 RESHAPE

Consider the following assignment:

```
res = RESHAPE( x, shape, pad, order )
```

If the optional `pad` and `order` arguments are not given the tangent linear and adjoint statements are:

tangent linear code

```
g_res = RESHAPE( g_x, shape )
```

adjoint code

```
adx += RESHAPE( adres, SHAPE(x) )
adres = 0.0
```

The reshape intrinsic function is not implemented for given `pad` or `order` argument.

6.2.14 SPREAD

Consider the following assignment:

```
res = SPREAD( x, dim, ncopies )
```

The tangent linear and the adjoint statements are:

tangent linear code

```
g_res = SPREAD( g_x, dim, ncopies )
```

adjoint code

```
adx += SUM( adres, dim )
adres = 0.0
```

6.2.15 TRANSFER

The transfer intrinsic function performs a low-level conversion of types. If this is an active statement the derivative is not defined.

6.2.16 TRANSPOSE

Consider the following assignment:

```
res = TRANSPOSE( x )
```

The tangent linear and the adjoint statements are:

tangent linear code	adjoint code
`g_res = TRANSPOSE( g_x )`	`adx += TRANSPOSE( adres )` `adres = 0.0`

6.3 Intrinsic subroutines

All Fortran-90 intrinsic subroutines (DATE_AND_TIME, MVBITS, RANDOM_NUMBER, RANDOM_SEED, SYSTEM_CLOCK) are passive routines.

7 Generating derivative code for various applications

The generation and embedding of derivative code for some common applications is described. It is assumed that standard options (*standard_options*) are given which set the top-level routine, the independent, and the dependent variables (if not described explicitly) or other machine or compiler dependent options.

7.1 Forward sensitivity

Derivative code is generated to compute the sensitivity of the dependent variables with respect to the independent variables by:

tamc *standard_options* **-forward** *file_names*

The generated tangent linear top-level routine computes the independent variables and the product of the Jacobian times a vector simultaneously. The tangent linear variables of the independent variables must be initialized to some perturbations of the independent variables.

7.2 Full Jacobian by forward mode or Jacobian times seed matrix

Code is generated to compute the full Jacobian of a vector-valued function in forward mode by:

tamc *standard_options* **-forward -jacobian** *number file_names*

If the full Jacobian is requested the number of independent variables must by given by the '-jacobian' option. Otherwise if the product of the Jacobian times a seed matrix is requested then the number of columns of the seed matrix must be given by the '-jacobian' option.

7.3 Backward sensitivity

Derivative code to compute the sensitivity of an independent variable with respect to many dependent variables is generated by:

tamc *standard_options* **-reverse** *file_names*

The generated adjoint top-level routine computes the product of the transposed Jacobian times a vector. If the -pure option was given the modified top-level routine may have to be called before to compute tapes which are read during the adjoint code computations. The adjoint independent variables must be initialized to zero. The adjoint dependent variables must be initialized with suitable values.

7.4 Full Jacobian by reverse mode or seed matrix times Jacobian

Code is generated to compute the full Jacobian of a vector-valued function in reverse mode by:

tamc *standard_options* **-reverse -jacobian** *number file_names*

If the full Jacobian is requested the number of dependent variables must by given by the '-jacobian' option. Otherwise, if the product of a seed matrix times the Jacobian is requested then the number of rows of the seed matrix must be given by the '-jacobian' option.

7.5 Optimization

If a scalar-valued function has to be optimized then its Jacobian has only one row and the function gradients can be computed efficiently in reverse mode. The same derivative code than for backward sensitivity should be generated.

tamc *standard_options* **-reverse** *file_names*

7.6 Singular vector determination

Singular vectors are the leading eigenvectors of the transposed Jacobian times the Jacobian. The singular vectors can be computed without forming the whole Jacobian. This is useful if the Jacobian is very large. The required code is the combination of forward sensitivity code and backward sensitivity code. Both parts can be generated simultaneously by:

tamc *standard_options* **-reverse -forward -pure** *file_names*

7.7 Hessian times a vector

The Hessian matrix of a scalar valued function is a symmetric matrix. Code to compute the product of the Hessian times an arbitrary vector is generated in two steps. First adjoint code computing the gradient of the function at a given point is generated by applying TAMC in reverse mode. Then TAMC is applied again to the whole generated code generating derivative code in forward mode this time. The whole procedure is called forward over reverse mode.

tamc *standard_options* **-toplevel** toplevel **-input** x **-output** fc **-reverse -special** *file_names*

tamc *standard_options* **-toplevel** adtoplevel **-input** x **-output** adx **-forward** *file_names.f file_names_ad.f*

Please note some restrictions apply. The directives

```
CADJ INIT tape = MEMORY
CADJ INIT tape = USER
```

can not be used, because the subroutine calls, generated for the store directives and using these tapes, do not have a corresponding tangent linear version yet.

7.8 Full Hessian

Code to compute the full Hessian matrix is generated as explained above. In the second sweep the **-jacobian** option has to be used. Its argument must be the number of independent variables, e.g. 20.

tamc *standard_options* **-toplevel** toplevel **-input** x **-output** fc **-reverse -special** *file_names*

tamc *standard_options* **-toplevel** adtoplevel **-input** x **-output** adx **-forward** **-jacobian** 20 *file_names.f* *file_names_ad.f*

Please note, if the number of independent variables is large, huge memory and computer time resources might be necessary.

8 Incorporate TAMC generated routines

The correct incorporation of the generated routines requires some objects and there proper initialization in the driver routine. TAMC generated routines can be easily compiled and linked with the tamlink utility (see next section). However, the requirements are described in this section for the various kinds of generated code. The code must be linked with the TAMC library if it contains calls to library routines.

8.1 Tangent linear code

The incorporation of TAMC generated tangent linear code into a program requires the following steps:

1. Allocate gradient objects in the calling routine.
2. Initialize all tangent linear variables of variables given as independent (option -input) to the identity or seed matrix, or to some given perturbations.
3. Call the tangent linear top-level routine.
4. Write out or use independent variables and their tangent linear correspondent.

8.2 Adjoint code

The incorporation of TAMC generated adjoint code into a program requires the following steps:

1. Allocate gradient objects in the calling routine.
2. If the option -pure was given call the modified top-level routine, if it exists, in order to write required tapes and to compute the cost function.
3. Initialize the adjoint of variables given as independent (option -input) to zero.
4. Initialize the adjoint of variables given as dependent (option -output) to the identity or seed matrix, or to some given sensitivity.
5. Call the adjoint top-level routine.
6. Write out or use sensitivities, i.e. the adjoint independent variables.
7. Link with TAMC library (-ltamc).

9 Utility

A software package has been developed in order to simplify the embedding of derivative code for several purposes. Among them are gradient check, sensitivity analysis, optimization, singular vector determination, and hessian computation. If you want to use this package, you have to split your function computing code into parts and construct subroutines having specific names. If the function is scalar valued (cost function) these subroutines should be:

- NUMBMOD
- INITMOD
- MODEL
- POSTMOD

For a vector valued function the subroutines are:

- SETFUNC
- INITFUNC
- FUNC
- POSTFUNC

After construction of the derivative code by

tamc -toplevel MODEL **-input** X **-output** FC **-arglist -reverse -forward** *file_names*

or by

tamc -toplevel FUNC **-input** X **-output** Y **-arglist -reverse -forward** *file_names*

a script "tamlink" will apply compile and link the function code, the adjoint code, the tangent linear code, and a library and write an executable file. The syntax is :

tamlink [options] filenames

Input :	filename_ad.f	:	adjoint and modified function code
	filename_ftl.f	:	tangent linear code
Output :	last_file_name	:	executable file

The file names are given by a blank separated list. They may be given with or without the extension **f**. The executable name is build out of the last given file name.

9.1 Options of tamlink

-opti	:	link code for cost function minimization (default)
-check	:	link code to check the gradients provided by the adjoint code.
-checktl	:	link code to check the gradients provided by the tangent linear code.
-time	:	link code to compute a timing statistic
-optlib 'libname'	:	link the library for minimization (default: c_cgopt, other libraries:c_e04dge, c_e04uce, c_umcgg, c_m1qn3)
-l 'libname'	:	link an additional library
-L 'path'	:	set an additional search path for libraries

9.2 Scalar valued function interfaces

9.2.1 Subroutine NUMBMOD

Specification

```
SUBROUTINE NUMBMOD( N )
INTEGER N
```

Purpose

The subroutine NUMBMOD must set the number of independent variables.

Arguments

`N` INTEGER
On exit N must be the number of independent variables.

9.2.2 Subroutine INITMOD

Specification

```
SUBROUTINE INITMOD( N, X )
INTEGER N
REAL X(N)
```

Purpose

The subroutine INITMOD must initialize the vector of independent variables. All kinds of initialization should be done inside this subroutine or NUMBMOD.

Arguments

`N` INTEGER
On entry N is the number of independent variables.

`X` REAL array of dimension N.
On exit, X must contain the initial values of the independent variables.

9.2.3 Subroutine MODEL

Specification

```
SUBROUTINE MODEL( N, X, FC )
INTEGER N
REAL X(N), FC
```

Purpose

The subroutine MODEL must compute the cost function to the given independent variables. It is called by the optimization procedure.

Arguments

`N`	INTEGER On entry N is the number of independent variables.
`X`	REAL array of dimension N. On entry, X contains the values of the independent variables. X must not be changed.
`FC`	REAL On exit FC must contain the value of the function.

9.2.4 Subroutine POSTMOD

Specification

```
SUBROUTINE POSTMOD( N, X, FC, ADX )
INTEGER N
REAL X(N), FC, ADX(N)
```

Purpose

The subroutine POSTMOD is called at the end. It may contain any post processing.

Arguments

`N`	INTEGER On entry N is the number of independent variables.
`X`	REAL array of dimension N. On entry, X contains the values of the independent variables.
`FC`	REAL On entry FC contains the value of the function at the point X.
`ADX`	REAL array of dimension N. On entry, ADX contains the gradient vector of the cost function with respect to the independent variables.

9.3 Vector valued function interfaces

9.3.1 Subroutine SETFUNC

Specification

```
SUBROUTINE SETFUNC( N, M )
INTEGER N, M
```

Purpose

The subroutine SETFUNC must set the number of independent and dependent variables.

Arguments

`N`	INTEGER On exit N must be the number of independent variables.
`M`	INTEGER On exit M must be the number of dependent variables.

9.3.2 Subroutine INITFUNC

Specification

```
SUBROUTINE INITFUNC( N, X )
INTEGER N
REAL X(N)
```

Purpose

The subroutine INITFUNC must initialize the independent variables. All kinds of initialization should be done inside this subroutine or SETFUNC.

Arguments

`N` INTEGER
On entry N is the number of independent variables.

`X` REAL array of dimension N.
On exit, X must contain the initial values of the independent variables.

9.3.3 Subroutine FUNC

Specification

```
SUBROUTINE FUNC( N, X, M, Y )
INTEGER N, M
REAL X(N), Y(M)
```

Purpose

The subroutine FUNC must compute the dependent variables to the given independent variables.

Arguments

`N` INTEGER
On entry N is the number of independent variables.

`X` REAL array of dimension N.
On entry, X contains the values of the independent variables. X must not be changed.

`M` INTEGER
On entry M is the number of dependent variables. N must not be changed.

`Y` REAL array of dimension M.
On exit Y must contain the values of the dependent variables.

9.3.4 Subroutine POSTFUNC

Specification

```
SUBROUTINE POSTFUNC( N, X, M, Y, JAC, LDJ )
INTEGER N, M, LDJ
REAL X(N), Y(M), JAC(LDJ,M)
```

Purpose

The subroutine POSTFUNC is called at the end and should store the gradient values. It may also contain additional post processing.

Arguments

`N` INTEGER
On entry N is the number of independent variables.

`X` REAL array of dimension N.
On entry X contains the values of the independent variables.

`M` INTEGER
On entry N is the number of dependent variables.

`Y` REAL array of dimension M.
On entry Y contains the values of the dependent variables.

`JAC` REAL array of dimension N*M.
On entry, JAC contains the Jacobian matrix.

`LDJ` INTEGER
On entry LDJ is the leading dimension of JAC.

References

[BCKM94] Christian Bischof, Alan Carle, Payvand Khademi, and Andrew Maurer. The ADIFOR 2.0 System for the Automatic Differentiation of Fortran 77 Programs. Argonne Preprint ANL-MCS-P981-1194, Mathematics and Computer Science Division,Argonne National Laboratory, 9700 South Cass Ave., Argonne, 1994.

[Ber87] Martin Berz. The differential algebra FORTRAN precompiler DAFOR. Tech. Report AT–3: TN–87–32, Los Alamos National Laboratory, Los Alamos, N.M., 1987.

[BR97] Christian Bischof and Lucas Roh. ADIC: An Extensible Automatic Differentiation Tool for ANSI-C. Argonne Preprint ANL-MCS-P626-1196, Mathematics and Computer Science Division,Argonne National Laboratory, 9700 South Cass Ave., Argonne, 1997.

[Chr92] Bruce Christianson. Automatic Hessians by reverse accumulation. *IMA J. Numerical Anal.*, 12:135–150, 1992.

[Gie96] Ralf Giering. *AMC: Ein Werkzeug zum automatischen Differenzieren von Fortran-Programmen*, volume 42 of *Forschung und wissenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 1995*, pages 11–27. Gesellschaft für wissenschaftliche Datenverarbeitung mbH, Göttingen, Germany, 1996.

[GJU96] Andreas Griewank, David Juedes, and Jean Utke. ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, 22(2):131–167, 1996.

[GK98a] Ralf Giering and Thomas Kaminski. Recipes for Adjoint Code Construction. *ACM Trans. On Math. Software*, 24(4):437–474, 1998.

[GK98b] Ralf Giering and Thomas Kaminski. Using tamc to generate efficient adjoint code: Comparison of hand written and automatically generated code for evaluation of first and second derivatives. In C. Faure, editor, *Automatic differentiation for adjoint code generation*, pages 31–37. INRIA, Rapport de recherche 3555, November 1998.

[Gri89] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Akedemic Publishers, Dordrecht, 1989.

[Gri92] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[HHG+97] Klaus Hasselmann, Susanne Hasselmann, Ralf Giering, Victor Ocaña, and Hans von Storch. Sensitivity Studdy of Optimal CO_2 Emission Paths using a Simplified structural Integrated Assesment Model (SIAM). *Climatic Change*, 37:345–386, 1997.

[Hil82] Kenneth E. Hillstrom. JAKEF – A portable symbolic differentiator of functions given by algorithms. Tech. Report ANL–82–48, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1982.

[Hor91] Jim E. Horwedel. GRESS: A preprocessor for sensitivity studies on Fortran programs. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 243–250. SIAM, Philadelphia, Penn., 1991.

[KGH96] T. Kaminski, R. Giering, and M. Heimann. Sensitivity of the seasonal cycle of CO_2 at remote monitoring stations with respect to seasonal surface exchange fluxes determined with the adjoint of an atmospheric transport model. *Physics and Chemistry of the Earth*, 21(5–6):457–462, 1996.

[Kub96] Koichi Kubota. PADRE2 - Fortran precompiler for automatic differentiation and estimates of rounding error. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 367–374. SIAM, Philadelphia, Penn., 1996.

[MZG+99] Jochem Marotzke, Q. Kate Zhang, Ralf Giering, Detlef Stammer, Chris Hill, and Tong Lee. The Linearization and Adjoint of the MIT Ocean General Circulation Model, 1999. JGR in press.

[OBV+97] Geert Jan van Oldenborgh, Gerrit Burgers, Stephan Venzke, Christian Eckert, and Ralf Giering. Tracking down the delayed ENSO oscillator with an adjoint OGCM. Technical Report 97-23, Royal Netherlands Meteorological Institute, P.O. Box 201, 3730 AE De Bilt, The Netherlands, 1997. Monthly Weather Review, in press.

[RDG93] Nicole Rostaing, Stéphane Dalmas, and André Galligo. Automatic differentiation in Odyssée. *Tellus*, 45A:558–568, 1993.

[Shi96] Dmitri Shiriaev. ADOL–F automatic differentiation of Fortran codes. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 375–384. SIAM, Philadelphia, Penn., 1996.

[Sou91] Edgar J. Soulié. User's experience with Fortran precompilers for least squares optimization problems. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 297–306. SIAM, Philadelphia, Penn., 1991.

[SWG+97] Detlef Stammer, Carl Wunsch, Ralf Giering, Qian Zhang, Jochem Marotzke, John Marshall, and Chris Hill. The Global Ocean Circulation estimated from TOPEX/POSEIDON Altimetry and a General Circulation Model. Technical Report 49, Center for Global Change Science, Massachusetts Institute of Technology, 1997.

A TAMC library

A.1 Dynamic memory

The library libtamc.a contains subroutines for allocating dynamic memory and for storing and restoring variables.

- `MYALLOC` Allocate dynamic memory
- `ADSTORE` Write values to a dynamic memory tape
- `ADRESTO` Read values from a dynamic memory tape
- `GETKEY` Get a unique key
- `DMALLOC` Allocate dynamic memory
- `DMPUT` Allocate dynamic memory and write values to it
- `DMGET` Read values from dynamic memory
- `DMFREE` Deallocate dynamic memory

A.1.1 Subroutine GETKEY

Specification

```
SUBROUTINE GETKEY( ITAPE, IVAR, IKEY1, IKEY2, IKEY3, NREC )
INTEGER ITAPE, IVAR
INTEGER IKEY1, IKEY2, IKEY3
INTEGER NREC
```

Purpose

Compute a unique record number for storing.

Arguments

`ITAPE` INTEGER
On entry ITAPE contains a number indicating the tape.

`IVAR` INTEGER
On entry IVAR contains a number indicating the variable to be stored.

`IKEY1` INTEGER
On entry IKEY1 contains the pass number of surrounding unlimited loops.

`IKEY2` INTEGER
On entry IKEY2 contains the pass number of surrounding limited loops.

`IKEY3` INTEGER
On entry IKEY3 contains the number of subroutine calls.

`NREC` INTEGER
On exit NREC contains the unique record number.

A.1.2 Subroutine DMALLOC

Specification

```
SUBROUTINE DMALLOC( IPA, ISIZE, IFAIL )
INTEGER IPA
INTEGER ISIZE, IFAIL
```

Purpose

Allocate dynamic memory.

Arguments

`IPA` INTEGER
On exit IPA contains a pointer to the dynamic memory.

`ISIZE` INTEGER
On entry ISIZE contains the number of bytes to be allocated.

`IFAIL` INTEGER
On exit IFAIL=0 means successful allocation otherwise an error has occurred.

A.1.3 Subroutine DMPUT

Specification

```
SUBROUTINE DMPUT( IPA, X, ISIZE, N )
INTEGER IPA
INTEGER ISIZE, N
REAL X(N)
```

Purpose

Allocate dynamic memory and store a variable on it.

Arguments

`IPA` INTEGER
On exit IPA contains a pointer to the dynamic memory.

`X` REAL array of dimension N.
On entry, X is the variable to be stored.

`ISIZE` INTEGER
On entry ISIZE contains the precision in bytes of X.

`N` INTEGER
On entry N is the length of X.

A.1.4 Subroutine DMGET

Specification

```
SUBROUTINE DMGET( IPA, X, ISIZE, N )
INTEGER IPA
INTEGER ISIZE, N
REAL X(N)
```

Purpose

Restore a variable from dynamic memory.

Arguments

`IPA`	INTEGER On entry IPA contains a pointer to the dynamic memory.
`X`	REAL array of dimension N. On exit, X contains the stored values.
`ISIZE`	INTEGER On entry ISIZE contains the precision in bytes of X.
`N`	INTEGER On entry N is the length of X.

A.1.5 Subroutine DMFREE

Specification

```
SUBROUTINE DMFREE( IPA )
INTEGER IPA
```

Purpose

Deallocate dynamic memory allocated by DMALLOC or DMPUT.

Arguments

`IPA`	INTEGER On entry IPA contains a pointer to the dynamic memory.

A.2 Auxiliary subroutines

In addition some subroutines to initialize gradient objects are included.

A.2.1 Subroutine SZERO1, DZERO1

Specification

```
SUBROUTINE SZERO1( N, X )
INTEGER N
REAL X(N)

SUBROUTINE DZERO1( N, X )
INTEGER N
DOUBLE PRECISION X(N)
```

Purpose

Initialize an one dimensional array to zero.

Arguments

`N`	INTEGER On entry N is the length of X
`X`	REAL [DOUBLE PRECISION] array of dimension N. On exit, X contains zeros.

A.2.2 Subroutine SZERO2, DZERO2

Specification

```
SUBROUTINE SZERO2( N, M, X, LDX )
INTEGER N, M
REAL X(LDX,M)

SUBROUTINE DZERO2( N, M, X, LDX )
INTEGER N, M
DOUBLE PRECISION X(LDX,M)
```

Purpose

Initialize a two dimensional array to zero.

Arguments

`N` INTEGER
On entry N is the number of rows.

`M` INTEGER
On entry M is the number of columns

`X` REAL [DOUBLE PRECISION] array of dimension (LDX,M).
On exit, X contains zeros.

`LDX` INTEGER
On entry LDX is the first dimension of the array X as declared in the calling routine.
Constraint: LDX $\geq$ N

A.2.3 Subroutine SIDENT, DIDENT

Specification

```
SUBROUTINE SIDENT( N, X, LDX )
INTEGER N
REAL X(LDX,N)

SUBROUTINE DIDENT( N, X, LDX )
INTEGER N
DOUBLE PRECISION X(LDX,N)
```

Purpose

Initialize a two dimensional array to the identity matrix.

Arguments

`N` INTEGER
On entry N is the number of rows and columns.

`X` REAL [DOUBLE PRECISION] array of dimension (LDX,N).
On exit, X contains zeros.

`LDX` INTEGER
On entry LDX is the first dimension of the array X as declared in the calling routine.
Constraint: LDX $\geq$ N

A.2.4 Subroutine STRANS, DTRANS

Specification

```
SUBROUTINE STRANS( N, M, X, LDX, Y, LDY )
INTEGER N, M
REAL X(LDX,M)
REAL Y(LDY,N)

SUBROUTINE DTRANS( N, M, X, LDX, Y, LDY )
INTEGER N, M
DOUBLE PRECISION X(LDX,M)
DOUBLE PRECISION Y(LDY,N)
```

Purpose

Set the transposed to a two dimensional array.

Arguments

`N` INTEGER
On entry N is the number of rows.

`M` INTEGER
On entry M is the number of columns

`X` REAL [DOUBLE PRECISION] array of dimension (LDX,M).
On exit, X contains zeros.

`LDX` INTEGER
On entry LDX is the first dimension of the array X as declared in the calling routine.
Constraint: LDX $\geq$ N

`Y` REAL [DOUBLE PRECISION] array of dimension (LDY,N).
On exit, X contains zeros.

`LDY` INTEGER
On entry LDY is the first dimension of the array Y as declared in the calling routine.
Constraint: LDY $\geq$ M

B User defined storage subroutines

B.0.1 Subroutine ADOPEN

Specification

```
SUBROUTINE ADOPEN( NAME, LEN, TID, VID, SIZE, LENGTH )
INTEGER LEN, TID, VID, SIZE, LENGTH
CHARACTER*(*) NAME
```

Purpose

The subroutine ADOPEN must initialize a user defined tape for a specific variable stored on that tape.

Arguments

`NAME`	CHARACTER On entry NAME is a string consisting of the tape name, a number identifying the variable, the subroutine name and the variable name.
`LEN`	INTEGER On entry, LEN is the characters length of NAME.
`TID`	INTEGER On entry, TID identifies the tape.
`VID`	INTEGER On entry, VID identifies the variable to be stored on that tape.
`SIZE`	INTEGER On entry, SIZE is the size in bytes of the type stored on that tape.
`LENGTH`	INTEGER On entry, LENGTH is the length of the variable to be stored on the tape.

B.0.2 Subroutine ADCLOSE

Specification

```
SUBROUTINE ADCLOSE( NAME, LEN, TID, VID, SIZE, LENGTH )
INTEGER LEN, TID, VID, SIZE, LENGTH
CHARACTER*(*) NAME
```

Purpose

The subroutine ADCLOSE must close a user defined tape.

Arguments

See description of ADOPEN.

B.0.3 Subroutine ADWRITE

Specification

```
SUBROUTINE ADWRITE( NAME, LEN, TID, VID, VAR, SIZE, LENGTH, IREC )
INTEGER LEN, TID, VID, SIZE, LENGTH, IREC
CHARACTER*(*) NAME
REAL VAR(LENGTH)
```

Purpose

The subroutine ADWRITE must store the variable VAR on the tape NAME.

Arguments

`NAME` CHARACTER
On entry NAME is the extended tape name.

`LEN` INTEGER
On entry, LEN is the characters length of NAME.

`TID` INTEGER
On entry, TID identifies the tape.

`VID` INTEGER
On entry, VID identifies the variable to be stored on that tape.

`VAR` REAL array of dimension LENGTH
On entry, VAR contains the values to be stored.
VAR must not be changed.

`SIZE` INTEGER
On entry, SIZE is the size in bytes of the type of variable VAR.

`LENGTH` INTEGER
On entry, LENGTH is the dimension of the variable stored on tape.

`IREC` INTEGER
On entry, IREC is the record number to be written.

B.0.4 Subroutine ADREAD

Specification

```
SUBROUTINE ADREAD( NAME, LEN, TID, VID, VAR, SIZE, LENGTH, IREC )
INTEGER LEN, TID, VID, SIZE, LENGTH, IREC
CHARACTER*(*) NAME
REAL VAR(LENGTH)
```

Purpose

The subroutine ADREAD must restore the variable VAR from the tape NAME.

Arguments

`NAME` CHARACTER
On entry NAME is the extended tape name.

`LEN` INTEGER
On entry, LEN is the characters length of NAME.

`TID` INTEGER
On entry, TID identifies the tape.

`VID` INTEGER
On entry, VID identifies the variable to be stored on that tape.

`VAR` REAL array of dimension LENGTH
On exit, VAR must contain the restored values.

`SIZE` INTEGER
On entry, SIZE is the size in bytes of the type of variable VAR.

`LENGTH` INTEGER
On entry, LENGTH is the dimension of the variable stored on tape.

`IREC` INTEGER
On entry, IREC is the record number to be read.

statement		_file_	_unit_	_ptr_
OPEN (write)	ref			
	def		x	x
OPEN (read/write)	ref	x		
	def		x	x
INQUIRE	ref		x	x
	def			
REWIND	ref		x	
	def			x
BACKSPACE	ref		x	x
	def			x
ENDFILE	ref		x	x
	def		x	x
WRITE	ref		x	x
	def		x	x
READ	ref		x	x
	def			x
CLOSE (read/write)	ref		x	
	def	x		
	kill		x	x
CLOSE (read)	ref		x	
	def			
	kill		x	x

Table 17: Phantom variable dependences introduced to detect active files and to handle required file accesses.

C Active files

A file is active if an expression, influenced by the independent variables, is written on it and at least one variable read from the file influences the dependent variables. All records of an active file will be handled as active. To detect active files and to generate derivative code phantom global variables are introduced. For each unit three different variables are used, if the unit number is 10 these are _file_10, _unit_10, and _ptr_10. If the unit number can not be determined from the source code TAMC uses the first variable name in the unit expression instead. Consequently, it should be made sure that there is an unambiguous mapping between unit numbers and expressions representing unit numbers. If a corresponding OPEN-statement is found and a file name argument is present TAMC replaces the characters 'file' in the phantom name by the file name.

Table 17 shows the data flow dependences introduced for every FORTRAN I/O-statement. OPEN- and CLOSE-statements, are divided into two classes. The separation depends on whether the unit is read or written. In addition to the flow shown, a READ-statement defines all variables read and a write statement references all variables in expressions written. Variables of indices of defined or referenced variables are referenced as well. Implied DO-loop variables are defined by the READ- or WRITE-statement. I/O statements do not reference or define the phantom _ptr_ variable if the unit is a direct access file.

It can be verified that, for simple I/O-sequences, accessing one unit, like [OPEN, WRITE, WRITE, CLOSE, OPEN, READ, READ, CLOSE] or [OPEN, WRITE, READ, CLOSE], the phantom variable _unit_ will reflect whether the file is active or not. Furthermore, if a variable read from a file is required for derivative computations, all necessary I/O-statements will be detected thanks to the phantom variable dependences.

Handling of required and/or active files may fail if

- complicated sequences of I/O-statements are used
- units can not be distinguished
- a CLOSE-statement follows the use of a variable read from the unit closed
- the OPEN-statement of a unit can not be found

In reverse mode, recomputations can not be generated in some cases if the file accesses are spread over several routines. These pitfalls should be checked carefully before using the derivative code. If TAMC can not generate correct derivative code because of I/O-statements, these statements should be separated in extra subroutines. Instead of the code, only flow directives can then be given to TAMC.