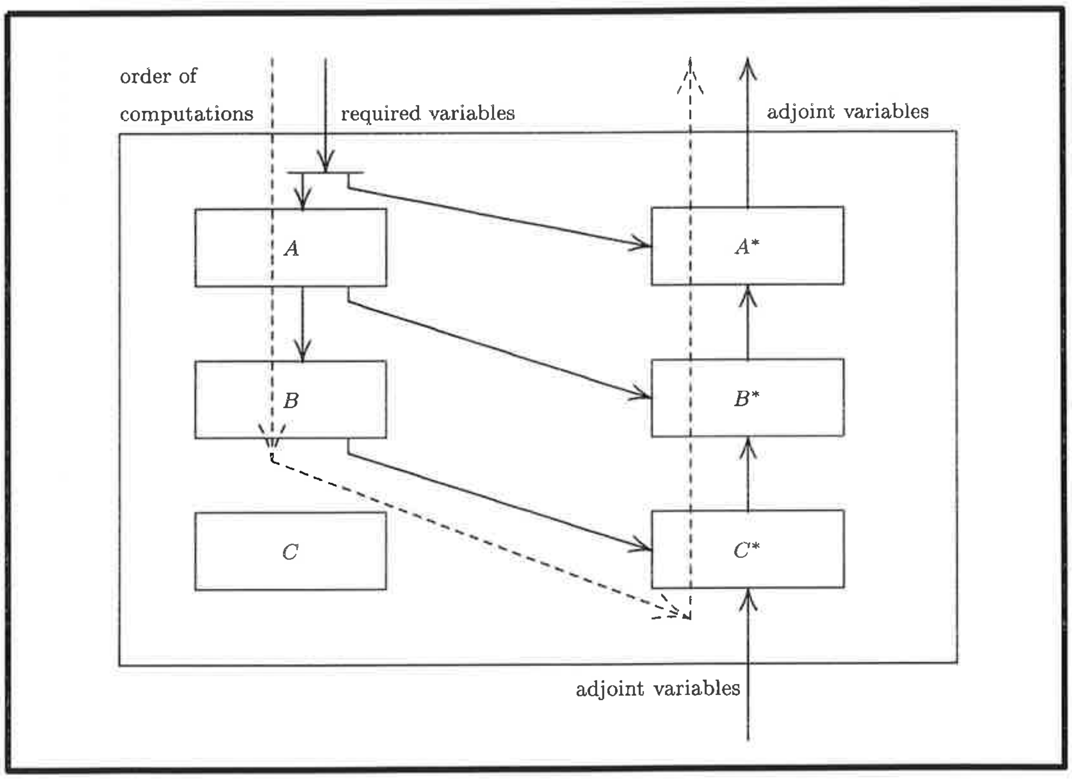
Max-Planck-Institut für Meteorologie

REPORT No. 212



RECIPES FOR ADJOINT CODE CONSTRUCTION

by

Ralf Giering • Thomas Kaminski

HAMBURG, August 1996

AUTHORS:

Ralf Giering
Thomas Kaminski

Max-Planck-Institut
für Meteorologie

MAX-PLANCK-INSTITUT
FÜR METEOROLOGIE
BUNDESSTRASSE 55
D - 20146 HAMBURG
GERMANY

Tel.:	+49-(0)40-4 11 73-0
Telefax:	+49-(0)40-4 11 73-298
E-Mail:	<name> @ dkrz.de

REPb 212

Recipes for Adjoint Code Construction

Ralf Giering
Max-Planck-Institut für Meteorologie
and
Thomas Kaminski
Max-Planck-Institut für Meteorologie

Adjoint models are increasingly being developed for use in meteorology and oceanography. Typical applications are data assimilation, model tuning, sensitivity analysis and determination of singular vectors. The adjoint model computes the gradient of a cost function with respect to control variables. Generation of adjoint code may be seen as the special case of differentiation of algorithms in reverse mode, where the dependent function is a scalar.

The described method for adjoint code generation is based on a few basic principles, which permits the establishment of simple construction rules for adjoint statements and complete adjoint subprograms. These rules are presented and illustrated with some examples. Conflicts that occur due to redefinition of variables and loops are also discussed.

Direct coding of the adjoint of a more sophisticated model is extremely time consuming and subject to errors. Hence, automatic generation of adjoint code represents a distinct advantage. An implementation of the method, described in this paper, is the Adjoint Model Compiler (AMC).

General Terms: adjoint model, differentiation of algorithms, inverse modeling, reverse mode
Additional Key Words and Phrases: adjoint operator, data assimilation, implicit functions, optimization, vectorization

1. INTRODUCTION

Adjoint models are tools developed for inverse modeling of physical systems. Inverse modeling is used in various fields of science such as geophysics and molecular physics. Among the applications of adjoint models in oceanography and meteorology are data assimilation, model tuning, sensitivity analysis, and determination of singular vectors.

In meteorology and oceanography, combining a model with data is a crucial task. Several methods have been developed for data assimilation. Sequential methods put a model in a state which is, in general, not consistent with its dynamics. The model is disturbed and needs some time to reach dynamic consistency. To keep the disturbance as small as possible, data are prepared and only a correction of the model state "in the direction" of the data takes place (Fig.1). Several sequential methods differ in the degree of consistency of the correction with model dynamics, e.g. Nudging, Successive Correction, Optimal Interpolation, Kalman Filter [Ghil 1989]. In contrast, the adjoint method always guarantees full consistency with the dynamics. By variation of control variables it is intended to adjust a model trajectory as close as possible to the data (Fig.2). To quantify the misfit of a model prediction, a cost function is introduced. This cost function is minimized by use of an iterative algorithm. Starting with a first guess, in each iteration step an improved vector of control variables is searched. Thereby the search direction is computed from the gradient of the cost function with respect to the control variables. The adjoint model computes this gradient vector. In data assimilation, the control variables typically determine the initial conditions or the forcing for the model [Talagrand and Courtier 1987; Courtier and Talagrand 1987; Giering and Maier-Reimer 1996]. The use of an adjoint model in an optimization procedure is

ISSN 0937-1060

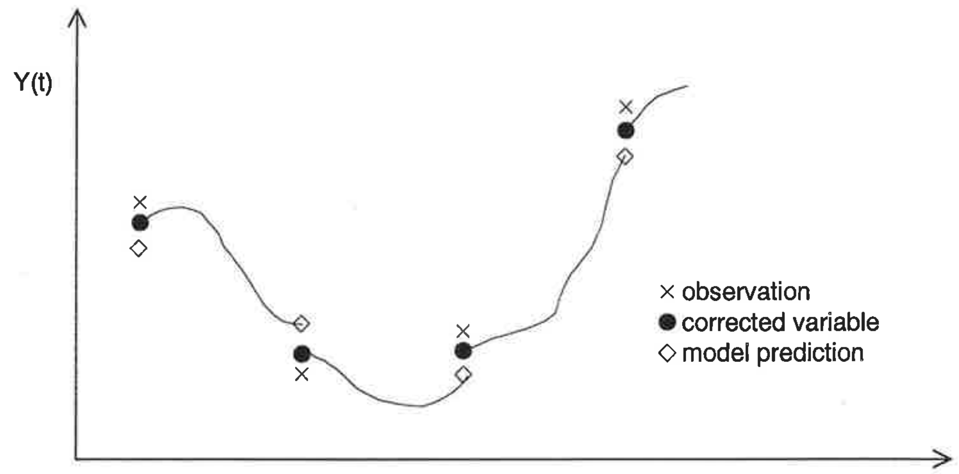


Fig. 1: Schematic representation of sequential methods. The model state is represented by the value on the Y-axis.

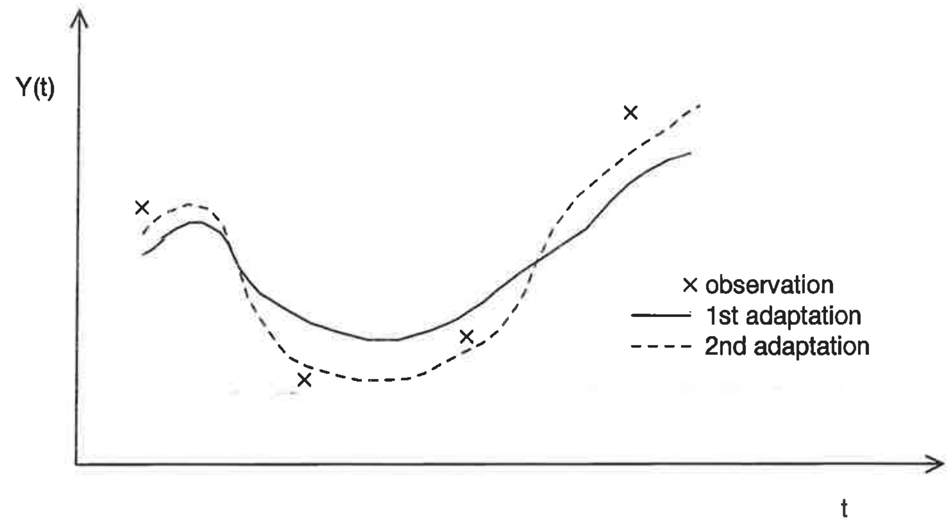


Fig. 2: Schematic representation of variational methods. Several trajectories differing in the respective value of the cost function are displayed.

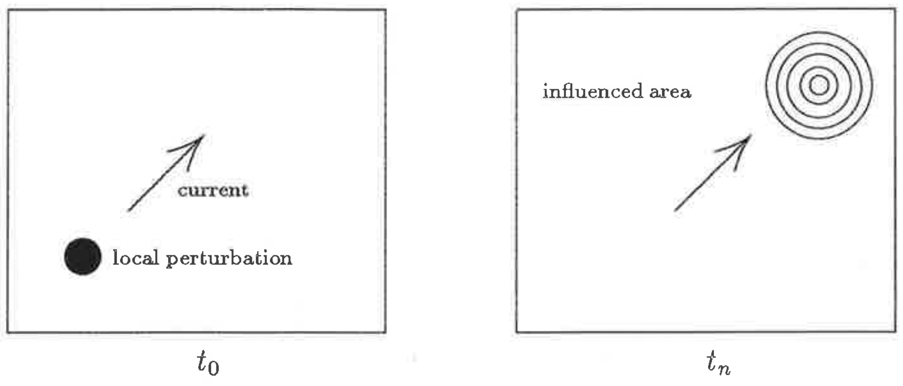


Fig. 3. Tangent linear model : advection and diffusion of perturbations

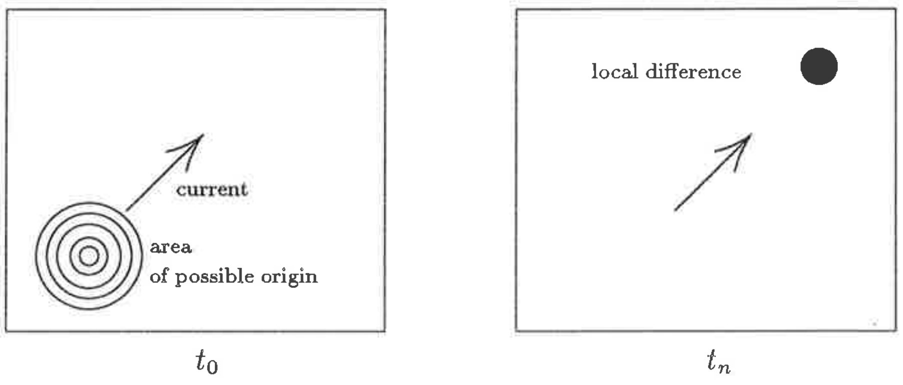


Fig. 4. Adjoint model : advection and diffusion of influence

described in section 2.

In case of model tuning, data are used to optimize the model equations themselves. Optimization is performed analogously to data assimilation, but the control variables are parameters in the underlying equations (for example numerical diffusion or coupling-constants) [Schröter 1992; Louis 1991].

In the context of inverse modeling it is useful to look at a model of a physical system as a mapping $\mathcal{H}$ of a vector of control variables X onto a vector of predictions Y. The aim is to infer information about the control variables X from the model prediction Y. Linearization of the model around a given point X_0 defines the tangent linear model, which is represented by the Jacobian matrix $A(X_0)$ of the mapping $\mathcal{H}$. The tangent linear model maps variations of the control variables δX onto variations of the model prediction δY. The adjoint model is represented by the adjoint $A^*(X_0)$ of the Jacobian. It maps in the reverse direction and computes the influence of the control variables on a given anomaly of the model prediction. A more detailed introduction to adjoint models is given in section 2.

Sensitivity analysis is another application of adjoint models [Cacucci 1981]. A tangent linear model can be used to analyze the impact of small disturbances. For instance, consider a tangent linear model of the advection of temperatures by horizontal currents. If the temperature at one point is changed, this anomaly is transported downstream and broadened by diffusion (Fig. 3). In contrast, the adjoint model can be used to analyze the origin of any anomaly. As shown in Figure 4, a difference at one location can be caused by propagation of an anomaly from upstream. Thereby, due to the effect of diffusion, the possible origin of the anomaly is located in a broader area.

In order to forecast the time development of a system, it is useful to know which initial perturbations amplify most rapidly [Webster and Hopkins 1994]. A perturbation δX implies the largest possible perturbation δY if it points in the direction associated with the dominant eigenvector of the operator A^*A. The dominant eigenvectors are called singular vectors or the most unstable modes.

Applications described above obviously require a numerical code of the model and its adjoint. The question is how practical coding of adjoint models can be done.

Suppose we want to simulate a dynamical system numerically. The development of a numerical simulation program is usually done in three steps. First, the analytical differential equations are formulated. Then a discretization scheme is chosen and the discrete difference equations are constructed. The last step is to implement an algorithm that solves the discrete equations in a programming language. The construction of the adjoint model code may be implemented after any of these three steps.

The analytical model equations are transformed into the adjoint equations by applying the rules for analytical adjoint operators. These equations subsequently are discretized and solved by use of a numerical algorithm. Since the product rule is not valid for discrete operators, one has to be careful in constructing the discrete adjoint operators. This method is mostly applied to box models having simple boundary conditions [Schröter 1989].

Constructing the adjoint model from the discrete model equations is usually done by defining a Lagrange Function. The derivatives of the Euler-Lagrange equations with respect to the model variables yield the discrete adjoint equations. Applying this method, no adjoint operators have to be constructed. However, extensive and cumbersome coding is necessary. The boundary conditions are handled separately in most cases. Thacker has introduced this method and applied it to simple models [Long and Thacker 1989a; Long and Thacker 1989b] . Also the adjoint code of the GFDL ocean model has been constructed this way [Tzipperman et al. 1992; Tzipperman et al. 1992].

The present article is concerned with the third method, where the adjoint code is developed directly from the numerical code of the model. A numerical model is an algorithm that can be viewed as a composition of differentiable functions, each representing a statement in the numerical code. Note that the order of evaluation of the individual functions is imposed by the algorithm. Differentiation of the composition can be done by applying the chain rule. The resulting multiple product can be computed in different ways:

Operating in forward mode, the intermediate derivatives are computed in the same order as the model computes the composition. In contrast, the adjoint model operates in reverse mode, i.e. the intermediate derivatives are computed in reverse order. A detailed introduction to differentiation of algorithms is given in section 3. This method is feasible even for highly sophisticated models with complicated boundary conditions. In this approach, a distinct adjoint model code fragment corresponds to each model code statement. The adjoint code fragments are composed in reverse order compared to the model code. For each kind of statement simple rules can be formulated for constructing adjoint statements [Talagrand 1991; Thacker 1991]. This simplifies considerably the adjoint code construction and debugging.

In section 4 some basic concepts for adjoint code generation from the model code are introduced, such as active and passive variables, locality, modularity, and readability. Following these concepts, simple rules for the adjoint of most types of statements are derived. The general rules are illustrated by some Fortran examples. A problem of the reverse mode is to provide required variables, i.e. variables computed by the model code and used by the adjoint code. Conflicts occurring due to redefinition of required variables are described and solutions are given. Programs written in modular languages like Fortran consist of procedures and functions. The generation of the corresponding adjoint structure is explained below. It consists of the argument list, declaration of all variables, initialization of adjoint variables, and the combination of the adjoint statements.

The existence of simple rules for the construction of adjoint code suggests performing this task automatically. Giering [Giering 1992] has developed a source transformation tool (Adjoint Model Compiler, AMC) based on most of these rules. It accepts Fortran-77 code for the computation of a function and generates Fortran-77 code for the computation of the derivative. Another system (Odyssée) has been developed by Rostaing [Rostaing et al. 1993] organized as a toolkit. The forward mode of automatic differentiation is implemented by ADIFOR [Bischof et al. 1994], another precompiler.

In many simulation programs an implicit or semi-implicit time integration scheme is applied. To perform one time step an implicit equation is solved using an iterative method. Applying the rules described in the previous sections, the adjoint code would also be an iteration, which requires variables for each iteration. In section 5 an alternative adjoint code for nonlinear implicit functions is presented. It avoids storing or recomputing required variables and thus saves memory resources or computation time.

An important aspect of computer programs is their performance, especially on vector machines. Occasionally, a formal application of the simple rules creates from a vectorizable loop an adjoint code fragment, which is not vectorizable. In section 6, we discuss how to overcome this problem with some typical examples.

2. ADJOINT MODELS

Consider a dynamical physical system and a model describing this system. Let $D \in \mathrm{R}^{\mathrm{m}}$ $(m \in \mathrm{N})$ be a set of observations and suppose that the model can compute the values $Y \in \mathrm{R}^{\mathrm{m}}$ corresponding to these observations. How can the model be manipulated in order to obtain an optimal fit between observations and corresponding model values?

To quantify the misfit we introduce a cost function

$$J := \frac{1}{2}(Y - D\,, Y - D) \tag{1}$$

by the choice of an appropriate inner product $(\cdot\,,\cdot)$. This implies that least-squares-fitting is intended: The smaller J is the better the model fits the data.

In order to manipulate the model, we specify a set of $n \in \mathrm{N}$ parameters X, which are called control variables in the following. The dependence of Y on X within the model is given by a mapping

$$\begin{aligned} \mathcal{H} : \mathrm{R}^{\mathrm{n}} &\longrightarrow \mathrm{R}^{\mathrm{m}} \\ X &\longmapsto Y \,. \end{aligned} \tag{2}$$

Thus, J can be expressed in terms of X by

$$\begin{aligned} J : \mathrm{R}^n &\rightarrow \mathrm{R} \\ X &\mapsto \frac{1}{2} (\mathcal{H}(X) - D \, , \mathcal{H}(X) - D) \ . \end{aligned} \tag{3}$$

The problem is to determine the set of control variables X that minimizes J. Effective minimization algorithms require the gradient $\nabla_X J(X_0)$ of J with respect to the control variables at a given point X_0. To first order we write the Taylor expansion of J:

$$J(X) \ = \ J(X_0) \ + \ (\nabla_X J(X_0) \, , X - X_0 \,) \ + \ o(|X - X_0|) \tag{4}$$

or, in short terms,

$$\delta J \ = \ (\nabla_X J(X_0) \, , \delta X \,) \ . \tag{5}$$

In the following we will use this shorthand notation whenever linear approximations are involved. Suppose $\mathcal{H}$ is sufficiently regular, then for each parameter vector X_0 a variation of Y can be approximated to first order by

$$\delta Y = A(X_0) \ \delta X \ , \tag{6}$$

where $A(X_0)$ denotes the Jacobian of $\mathcal{H}$ at X_0. Due to the symmetry of the inner product and the product rule the differentiation of (3) yields

$$\begin{aligned} \delta J \ &= \ \frac{1}{2} (\, A(X_0)\delta X \, , \mathcal{H}(X_0) - D \,) + \frac{1}{2} (\mathcal{H}(X_0) - D \, , A(X_0)\delta X \,) \\ &= \ \ (\mathcal{H}(X_0) - D \, , A(X_0)\delta X \,) \ . \end{aligned} \tag{7}$$

Using the definition of the adjoint operator A^*:

$$(\, v \, , A \, w \,) \ \ = \ \ (\, A^* \, v \, , w \,) \ , \tag{8}$$

we obtain

$$\delta J = \ \ (\, A^*(X_0)(\mathcal{H}(X_0) - D) \, , \delta X \,) \ . \tag{9}$$

Therefore, according to the definition of the gradient (5), the gradient of the cost function with respect to the control variables is

$$\nabla_X J(X_0) \ = \ A^*(X_0) \, (\mathcal{H}(X_0) - D) \ . \tag{10}$$

The linear operator $A(X_0)$ represents the tangent linear model. Its adjoint $A^*(X_0)$, which is linear as well, represents the adjoint model. Both operators depend on the point X_0 at which the model is linearized. According to (10) the difference $\mathcal{H}(X_0) - D$ can be interpreted as a forcing of the adjoint model.

The computation of the cost function and its gradient for a given vector of control variables is shown in Figure 5. A detailed analysis of required basic numerical operations yields that this computation takes only 2-5 times the computation of the cost function [Baur and Strassen 1983; Griewank 1989]. Alternatively, the gradient vector $\nabla_X J(X_0)$ could be approximated by finite differences, which needs at least $n + 1$ computations of the cost function. The ues of the adjoint model has two advantages compared to finite differences, especially for large n it saves a lot of run time and the computed gradient is exact.

The application of the adjoint model for optimization is illustrated by an example in appendix A. Here, the computation of the cost function and its gradient as shown in Figure 5 is performed by a module, which is called several times by the optimization procedure.

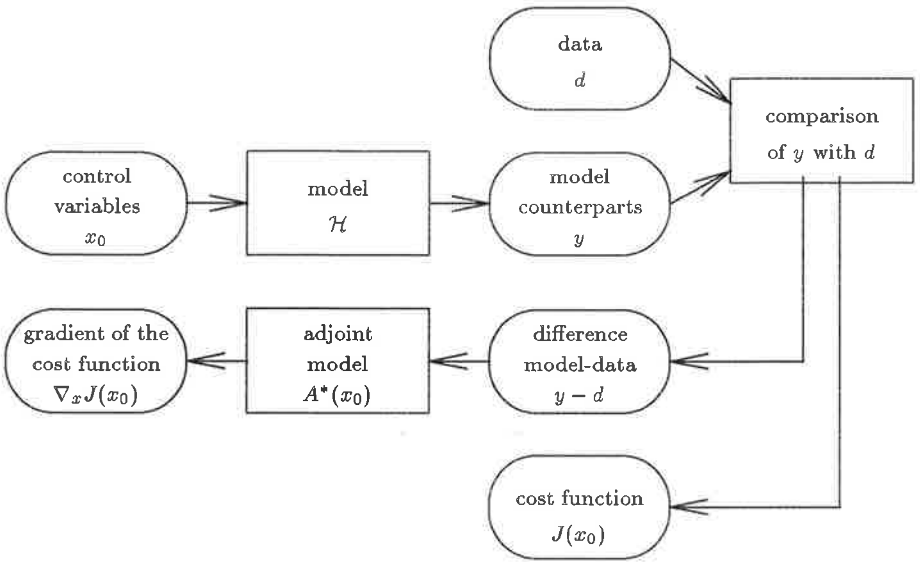


Fig. 5. Illustration of the evaluation of the cost function and its gradient vector

3. DIFFERENTIATION OF ALGORITHMS

In this section we show how a function $\mathcal{H}$, defined by a numerical algorithm, can be differentiated. Representing each step of the algorithm by a function, the composition of those functions is differentiated by use of the chain rule.

Evaluation of the derivative results in a multiple product of matrices each belonging to a particular step of the algorithm. For differentiation of scalar valued functions, in terms of run time, it is favorable to compute this matrix product in reverse order as compared to the original algorithm. This approach is called reverse mode. In the second part of this section the general rule for performing one step in reverse mode is derived for a scalar valued function.

3.1 Application of the chain rule

Let

$$\begin{aligned} \mathcal{H} : \mathrm{R}^{\mathrm{n}} &\rightarrow \mathrm{R}^{\mathrm{m}} \\ X &\mapsto Y \end{aligned} \tag{11}$$

be a function defined by a numerical algorithm. Since an algorithm can be long and complicated, it might be difficult to find an explicit representation of $\mathcal{H}$. However, a numerical algorithm can be decomposed into $K \in \mathrm{N}$ steps, each having an explicit representation

$$\begin{aligned} \mathcal{H}^l : \mathrm{R}^{\mathrm{n_{l-1}}} &\rightarrow \mathrm{R}^{\mathrm{n_l}} \qquad (\mathrm{l} = 1, ..., \mathrm{K}) \\ Z^{l-1} &\mapsto Z^l \ . \end{aligned} \tag{12}$$

In this mathematical representation the components of the variables Z^l are different from the variables in the numerical code. The numerical variables can change their values from step to step during one computation of $\mathcal{H}(X_0)$. In contrast, the vector Z^l holds all n_l intermediate results that are valid after the l-th step of the algorithm. In this context a result can be regarded as valid as long as it is kept on any memory unit of the computer. Thus, for $p \neq q$, components of Z^p and Z^q may be the values of the same variable at different steps of the algorithm.

The composition $\mathcal{H}$ of differentiable functions $\mathcal{H}^l$

$$\mathcal{H} = \mathcal{H}^K \circ \ldots \circ \mathcal{H}^1 =: \bigodot_{l=1}^{K} \mathcal{H}^l , \tag{13}$$

can be differentiated according to the chain rule. For a differentiable function $\mathcal{H}$ the Jacobian is defined by

$$A_{ij}(X_0) := \left. \frac{\partial \mathcal{H}_i(X)}{\partial X_j} \right|_{X=X_0} \qquad (i = 1, ..., m\,;\; j = 1, ..., n)\ . \tag{14}$$

Applying the chain rule to (13) yields:

$$A(X_0) = \left. \frac{\partial \mathcal{H}^K}{\partial Z^{K-1}} \right|_{Z^{K-1} = \bigodot_{l=1}^{K-1} \mathcal{H}^l(X_0)} \cdot \ldots \cdot \left. \frac{\partial \mathcal{H}^1}{\partial Z^0} \right|_{Z^0 = X_0} . \tag{15}$$

Since matrix multiplication is associative, at least two strategies for evaluation of the right hand side of (15) exist. Operating in forward mode the multiple product is evaluated in the same order as the composition in (13), i.e. first $\frac{\partial \mathcal{H}^2}{\partial Z^1} \cdot \frac{\partial \mathcal{H}^1}{\partial Z^0}$ is computed, then $\frac{\partial \mathcal{H}^3}{\partial Z^2}$ is multiplied by the result and so on. In contrast, the reverse mode starts with the evaluation of $\frac{\partial \mathcal{H}^K}{\partial Z^{K-1}} \cdot \frac{\partial \mathcal{H}^{K-1}}{\partial Z^{K-2}}$. In the former case all intermediate results have n columns and in the latter case they have m rows. Thus, for $n < m$, the forward mode needs less numerical computations, whereas for $n > m$ the situation is the other way around[1]. In general, the intermediate results of the preceding step are required for evaluation of the Jacobian (see eq. 15). This causes an essential difference between the two methods. While in the forward mode the intermediate results are required in the same order as computed, in the reverse mode they are required in reverse order.

By rigorous application of this concept, differentiation of an algorithm can be performed automatically.

3.2 Differentiation of a scalar-valued function

In the context of optimization a scalar valued function has to be differentiated, i.e. $n \geq m = 1$. Thus, the reverse mode is preferable. For $m = 1$ operating in reverse mode is called adjoint method and the algorithm for computing the gradient is called adjoint model. The rest of the paper is concerned with this case.

Let the decomposition of $\mathcal{H}$ be

$$\mathcal{H} = \bigodot_{l=1}^{K} \mathcal{H}^l , \tag{16}$$

where

$$\mathcal{H}^K : R^{n_{K-1}} \rightarrow R \tag{17}$$

and thus $n_K = m = 1$. For an intermediate result

$$Z_0^l := \bigodot_{i=1}^{l} \mathcal{H}^i(X_0) \qquad (1 \leq l \leq K) \tag{18}$$

[1]The sparsity of Jacobians can be used to reduce the number of computations. In this case the total number of computations in forward and reverse mode depends on additional criteria.

a variation δZ^l depends on a variation of the control variables δX, and can be written as

$$\delta Z^l = \left. \frac{\partial \left(\bigodot_{i=1}^{l} \mathcal{H}^i(X) \right)}{\partial X} \right|_{X=X_0} \delta X \ , \tag{19}$$

where $\delta Z^0 := \delta X$. The intermeditate variation depends on the previous intermediate variation by:

$$\delta Z^l = \left. \frac{\partial \mathcal{H}^l(Z^{l-1})}{\partial Z^{l-1}} \right|_{Z^{l-1}=Z_0^{l-1}} \delta Z^{l-1} \ . \tag{20}$$

The adjoint of an intermediate result is defined as the gradient of $\mathcal{H}$ with respect to the intermediate result:

$$\delta^* Z^l := \nabla_{Z^l} \bigodot_{i=l+1}^{k} \mathcal{H}^i(Z^l)|_{Z^l=Z_0^l} \ . \tag{21}$$

By the definition of the gradient (5) we obtain

$$\delta \mathcal{H} = \langle \delta^* Z^l , \delta Z^l \rangle \ , \tag{22}$$

where $\langle \cdot , \cdot \rangle$ denotes the Euclidean inner product. This shows that an adjoint value can be interpreted as the influence of the corresponding intermediate results on the cost function. Since (22) holds for every l, we obtain by using (20):

$$\begin{aligned} \langle \delta^* Z^{l-1} , \delta Z^{l-1} \rangle &= \langle \delta^* Z^l , \delta Z^l \rangle \\ &= \left\langle \delta^* Z^l , \left. \left(\frac{\partial \mathcal{H}^l(Z^{l-1})}{\partial Z^{l-1}} \right) \right|_{Z^{l-1}=Z_0^{l-1}} \delta Z^{l-1} \right\rangle \\ &= \left\langle \left. \left(\frac{\partial \mathcal{H}^l(Z^{l-1})}{\partial Z^{l-1}} \right)^* \right|_{Z^{l-1}=Z_0^{l-1}} \delta^* Z^l , \delta Z^{l-1} \right\rangle \ . \end{aligned}$$

This holds for all δZ^{l-1}, so that

$$\delta^* Z^{l-1} = \left. \left(\frac{\partial \mathcal{H}^l(Z^{l-1})}{\partial Z^{l-1}} \right)^* \right|_{Z^{l-1}=Z_0^{l-1}} \delta^* Z^l \ . \tag{23}$$

Equation (23) is the general rule to perform one step in the reverse mode. According to

$$\delta^* Z^0 = \delta^* X = \nabla_X \mathcal{H}, \tag{24}$$

the gradient of $\mathcal{H}$ with respect to the control variables is evaluated in the last step.

Since for the Euclidean inner product the adjoint operator is the transposed matrix, (23) can be written as:

$$\delta^* Z_i^{l-1} = \sum_{j=1}^{n_l} \left. \frac{\partial \mathcal{H}_j^l(Z^{l-1})}{\partial Z_i^{l-1}} \right|_{Z^{l-1}=Z_0^{l-1}} \delta^* Z_j^l \ . \tag{25}$$

Equation (25) is the basic equation for adjoint code generation.

4. ADJOINT CODE

This section is concerned with the implementation of the general rule (25) for adjoint code construction. Among the various ways to implement (25) a scheme of adjoint code construction should guarantee that the adjoint code is good to survey, efficient, easy to debug, and quickly adaptable to changes in the code that computes the cost function. For this purpose, some basic concepts are presented (see [Talagrand and Courtier 1987]). Following these concepts, simple rules for the adjoint of most types of statements are derived. For each kind of statement an example illustrates the general rules for construction of adjoint statements. The code fragments shown in the examples are written in the Fortran-77 programming language with some Fortran-90 extensions. Nevertheless, they may be easily translated into other languages. Finally, the construction of a complete adjoint subprogram from the individual adjoint statements will be discussed.

For convenience, we refer to the code, which computes the cost function, simply as code. The code computing the adjoint model is denoted by adjoint code.

4.1 Basic concepts

4.1.1 *Adjoint variables.* The intermediate results Z_i^l of section 3 denote the values of variables in the code. In the adjoint code we compute the adjoints $\delta^* Z_i^l$ of these values. In order to hold those adjoint values, adjoint variables have to be defined. Since the periods of validity of the values of one variable do not overlap and there is a one to one mapping between values and adjoint values, the periods of validity of the adjoint values do not overlap either. Thus, to hold these adjoint values, it is sufficient to define one adjoint variable for each variable.

4.1.2 *Active and passive variables.* Variables depending on the control variables and having an influence on the cost function are called active. An inter-procedural data dependence analysis has to be applied to determine the active variables. Since we deal with differentiation, only variables characterized by real numbers can be active.

A constant does not depend on an intermediate result except for intermediate results which are constants as well. In the former case the Jacobian corresponding to the definition of the constant has a column consisting of zeros. Thus, according to (23) the corresponding component of the adjoint intermediate result is lost. In the latter case the adjoint intermediate result is also lost, because it is only used to compute adjoint intermediate results, which will be lost later.

A diagnostic value does not influence any other intermediate result except for other diagnostic values. In the former case the Jacobian corresponding to the step where the diagnostic value disappears has a row consisting of zeros. Thus, according to (25) the component of the adjoint intermediate result is zero. In the latter case the adjoint intermediate result is also zero, because it is a linear combination of adjoints of diagnostic values, which are zero.

Therefore, for constants and diagnostic values no corresponding adjoint values have to be computed and thus no corresponding adjoint variables are needed. In context of differentiation of algorithms they are called passive variables.

To each statement computing one or more active variables a corresponding adjoint statement must be constructed. All the remaining statements only change values of passive variables and thus do not need an adjoint statement.

4.1.3 *Locality.* The position of an adjoint statement within the adjoint code is determined by the order of statements in the code, if the adjoint code is a strict implementation of the reverse mode. It is consistent and useful to construct an adjoint subprogram for every subprogram computing active variables. This concept makes it easier to adapt the adjoint code to changes in the cost function computing code. The adjoint code, constructed this way, is safer and errors will be found much easier, although in some cases unnecessary statements will be constructed following this concept.

4.1.4 *Modularity.* According to (23), for the computation of the l-th statement of a program, basically all intermediate results Z^{l-1} should be available. In general, however, the adjoint code of a statement does not need all results. All variables holding results needed are called required variables. Modularity is given, if for each adjoint statement the correct values of all required variables are available. Extra statements must be included in some cases to meet this demand. In case a required variable changes its value during code execution, conflicts in the recomputation of its values can arise, since in the adjoint code the values are required in reverse order (see sec. 4.2.3 and sec. 4.2.4). Following the concept of modularity each part of the adjoint code can be developed and maintained, i.e. adapted to changes in the code, independently from the rest of the adjoint code.

4.1.5 *Readability.* It is strongly recommended to follow a mnemonical convention for generating adjoint names. Considering the number of significant characters of a name, the new adjoint name must be distinguishable from all other valid names denoting the same structure.

In the examples given below, the adjoint names consist of the original name preceded by a short string: The generated adjoint name of a variable called `X` is `ADX`. Variables of the code required for the adjoint code computations have the same name in both codes. Hence, statements computing required variables can be copied directly from the code into the adjoint code. In addition the adjoint code is easy to understand.

4.2 Statements

The code of a numerical model consists mainly of only a few elements:

—assignments
—conditional statements
—loops
—sequences of statements (blocks)
—procedure calls
—input-, output statements (I/O-statements)

The following sections show the construction of the corresponding adjoint statements.

4.2.1 *Assignment.* Only assignments to active variables do have corresponding adjoint assignments (concept of active variables, see sec. 4.1.2). An assignment can be considered as an operator acting on the vector of active variables. In general, not all active variables are involved in an assignment. Hence, for representation of the assignment, it is sufficient to use a restricted operator acting only on the subset of involved active variables. The restricted vector of active variables consists of the left hand side (LHS) variable and all active variables of the right hand side (RHS) expression except variables inside subscript expressions (such as `I` in `A(4*I)`).

In order to construct the adjoint statement we determine the Jacobian of the operator. This is equivalent to constructing the tangent linear assignment. The variation coefficients form the first row of the Jacobian. The other rows consist of zeros and ones in the diagonal elements. The adjoint matrix is the transposed Jacobian. From this matrix the adjoint assignments are formulated.

For illustration, consider the following assignment performing the l-th step of a numerical algorithm:

```
Z = X * SIN(Y**2)    .
```

Assuming that `X`, `Y` and `Z` are active variables, the vector of involved active variables consists of these three variables. The tangent linear statement of the assignment is

$$\delta Z^l \;=\; [SIN(Y^{l-1}**2)]*\delta X^{l-1}+[X^{l-1}*COS(Y^{l-1}**2)*2*Y^{l-1}]*\delta Y^{l-1} \;.$$

Using the Jacobian, this can be expressed by the matrix-vector expression:

$$\begin{pmatrix}\delta Z\\ \delta Y\\ \delta X\end{pmatrix}^{l}=\begin{pmatrix}0 & X^{l-1}*COS(Y^{l-1}**2)*2*Y^{l-1} & SIN(Y^{l-1}**2)\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}\delta Z\\ \delta Y\\ \delta X\end{pmatrix}^{l-1},$$

where the index $l-1$ (l) denotes the values of the variables just before (after) the execution of the assignment. The adjoint operator is the transposed matrix acting on the adjoint variables (see eq. 23).

$$\begin{pmatrix}\delta^* Z\\ \delta^* Y\\ \delta^* X\end{pmatrix}^{l-1}=\begin{pmatrix}0 & 0 & 0\\ X^{l-1}*COS(Y^{l-1}**2)*2*Y^{l-1} & 1 & 0\\ SIN(Y^{l-1}**2) & 0 & 1\end{pmatrix}\begin{pmatrix}\delta^* Z\\ \delta^* Y\\ \delta^* X\end{pmatrix}^{l}.$$

Using the notation of section 4.1.5 this is translated to the assignments:

```
ADY = ADY + ADZ * X*COS(Y**2)*2*Y
ADX = ADX + ADZ * SIN(Y**2)
ADZ = 0.0    .
```

The adjoint assignments refer to the variables `X` and `Y`. For the execution their values just before the execution of the original assignment are required. The previous value of `Z` is overwritten by executing the assignment. Consequently, the previous value has no influence on the cost function. This is reflected by setting the adjoint variable `ADZ` to zero. An expression is added to the two other adjoint variables `ADX` and `ADY` denoting the additional influence of `X` and `Y` through `Z` on the cost function by the assignment to `Z`.

The assignment to the adjoint variable of the LHS variable must be the last one, because its previous value is used by all other corresponding adjoint assignments. The previous value must not be overwritten by one of the other adjoint assignments. Such an error could result from not recognizing that the LHS variable is referenced in the RHS expression, e.g. in case those variables are components of an array determined by subscript expressions. Whenever it is not possible to decide whether the same variable is used, the adjoint code has to be constructed according to a more general rule. This more general rule can be derived by introducing an auxiliary variable to hold the result of the assignment. The adjoint of this hypothetical code is then constructed and simplified.

This strategy is illustrated by the example in Figure 6. The subscript expressions `I` and `2*I-1` yield the same component of `X` for `I=1`. The hypothetical code and its adjoint are shown in Figure 7. Assuming that the auxiliary variable `ADH` is not used by other statements in the adjoint code we can simplify the adjoint hypothetical code and obtain the adjoint code on the right hand side of Figure 6.

code	adjoint code
`X(I) = X(2*I-1) + Y(I)`	`ADH = ADX(I)`
	`ADX(I)      = 0.0`
	`ADX(2*I-1) = ADX(2*I-1) + ADH`
	`ADY(I)      = ADY(I)      + ADH`

Fig. 6. Adjoint statements of an assignment (X and Y are active variables)

hypothetical code	adjoint hypothetical code
`H(I) = X(2*I-1) + Y(I)`	`ADH         = ADH + ADX(I)`
`X(I) = H(I)`	`ADX(I)      = 0.0`
	`ADX(2*I-1) = ADX(2*I-1) + ADH`
	`ADY(I)      = ADY(I)      + ADH`
	`ADH         = 0.0`

Fig. 7. Adjoint statements of an assignment with auxiliary variable

If the values of computed indices are known, the auxiliary adjoint variable can be ommitted by simplifying the adjoint code in Figure 6, which yields the adjoint code in Figure 8. In this case the same code would have been constructed according to the less general rule.

Function calls in the RHS expression will be discussed in section 4.2.6.

4.2.2 *Conditional statement.* A conditional statement executes statements according to the value of a condition (Fig. 9).

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 actual boolean value of the condition may be stored during the model code execution and restored in the adjoint code. Otherwise the condition must be evaluated again in the adjoint code (right hand side of Fig. 9).

4.2.3 *Loops.* In Fortran-77 one major application of loop constructs is assigning values to arrays. For construction of adjoint code, it is important to know, whether the result of one loop pass depends on the results of another. Such dependence analysis is very similar to the dependence analysis performed for vectorization or parallelization of loops.

4.2.3.1 *Parallel loops.* If there are no dependencies between different loop passes, the adjoint of the loop is a loop with the same bounds but the adjoint kernel. (Fig. 10).

4.2.3.2 *Sequential loops.* If the result of a loop pass depends on a result of a previous pass, the order of the loop passes is important. The adjoint loop has to compute the adjoint kernel in reverse order. The upper and the lower bound have to be exchanged and the negative step size has to be used (Fig. 11). If, in the loop,

adjoint code for I $\neq$ 2*I-1	adjoint code for I = 2*I-1, i.e I = 1
`ADX(2*I-1) = ADX(2*I-1) + ADX(I)`	`ADY(I)      = ADY(I)      + ADX(I)`
`ADY(I)      = ADY(I)      + ADX(I)`	
`ADX(I)      = 0.0`	

Fig. 8. Adjoint statements of an assignment (X and Y are active variables)

```
code                          adjoint code

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

Fig. 9. Adjoint conditional statement

```
code                          adjoint code

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

Fig. 10. Adjoint DO-loop

the upper bound is not reached, the lower bound of the adjoint is no longer up but the expression

$$\text{low} + \text{step} * \texttt{INT}((\text{up} - \text{low})/\text{step}) ,$$

where the notation of Fig. 11 is used.

Since the adjoint kernel needs required variables in reverse order of computation, a conflict occurs whenever the loop kernel overwrites required variables (see section 4.1.4). In the example in Fig. 12 the variable FAC holds an intermediate result which is overwritten in every loop pass. On the other hand the current values of FAC are required by the adjoint of the loop kernel.

For this conflict three solutions are suggested in the following:

(1) The values of FAC can be stored during every execution of the kernel and read before execution of the adjoint kernel.
(2) The variable FAC can be expanded by one dimension, so that no value is overwritten during execution of the loop. Thus for the adjoint of the loop the values of FAC can be provided either by a single read operation or by a single loop for recalculation (see Fig. 13).
(3) The required value of FAC can be recomputed before every pass of the adjoint kernel. For recomputation an inner loop is inserted (see Fig. 14). This is the most expensive solution in terms of run time but it needs neither additional memory nor additional I/O.

An important application of sequential loops is the computation of the limit of a converging sequence. For construction of the adjoint of such loops, an alternative scheme avoiding conflicts is described in section 5.

4.2.4 *Block of statements.* In order to obtain the adjoint of a block of statements, the adjoint of each statement must be constructed and arranged in reverse order

```
code                          adjoint code

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

Fig. 11. Adjoint DO-loop

```
Code

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

Fig. 12. Example of a conflict, FAC is overwritten (FAC and X are activ)

```
Code                        adjoint code

H(0) = 1.0                  ADH(N) = ADFAC
                            ADFAC = 0.0
DO I = 1, N                 DO I = N,1,-1
   H(I) = H(I-1) * X(I)        ADX(I)   = ADX(I)   + ADH(I)*H(I-1)
                               ADH(I-1) = ADH(I-1) + ADH(I)*X(I)
                               ADH(I)   = 0.0
END DO                      END DO
FAC = H(N)                  ADH(0) = 0.0
```

Fig. 13. Solution of a conflict by introducing an auxiliary array H(0:N)

(right hand side in Fig. 15). These adjoint statements may depend on variables of the original code defined inside the block or required by the block (arrows between left and right hand side in Fig. 15). 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. 15). A data flow analysis determines these statements. This might be done by computing the sets of input and output variables of each statement and including statements, which define a required variable. A more sophisticated data flow analysis would take array indices into consideration which can be arbitrary complex.

The set of required variables for the adjoint block consists of variables directly used by an adjoint statement and those needed for computing intermediate variables. Thus, 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

$$AB\ C^*\ B^*\ A^* \tag{26}$$

(the dashed line in Fig. 15 denotes this order of computations). The automatic adjoint code generation tool AMC [Giering 1992] applies this method.

The example in Figure 16 illustrates the method applied to a block of three statements. Variables located on the left hand side of an arrow denote referenced

```
code

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

Fig. 14. Solution of a conflict by recomputation of the value of the required variable FAC

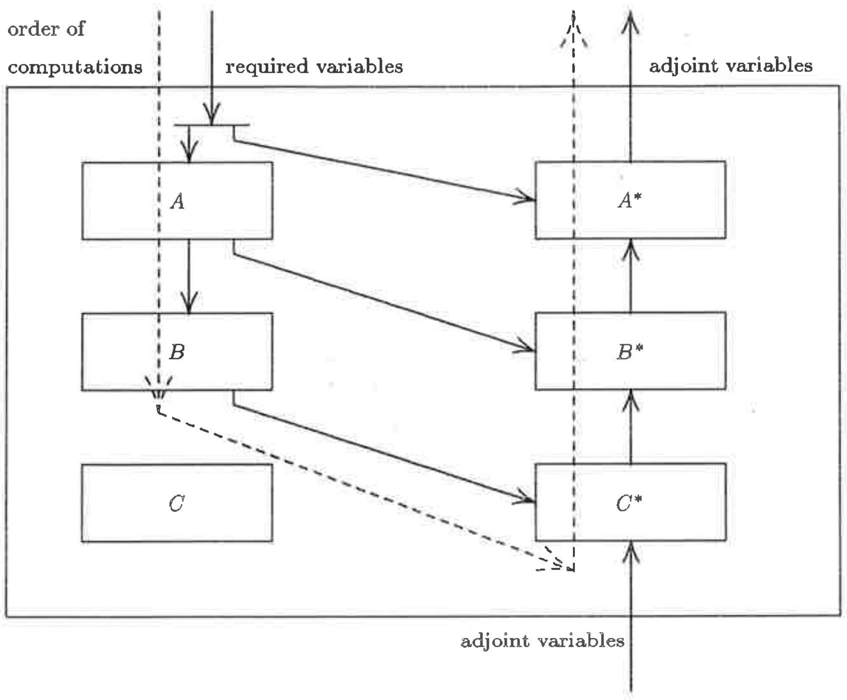


Fig. 15: Graph of an adjoint block of statements (A,B, and C are statements, A^*, B^*, and C^* are the corresponding adjoint statements)

variables, while those located on the right hand side denote defined variables. The constructed adjoint block is shown in Figure 17. This block requires the values of the variables `X`, `OM`, `PI`, and `RHOW`. They must be provided in order to obey modularity (see sec. 4.1.4).

A conflict occurs whenever a variable of the code is referenced by two or more adjoint statements requiring different values, because the variable is overwritten inside the block. This is often the case with auxiliary variables, which are used more than once to hold intermediate results.

The terms on the RHS of the first and third assignment of the code in Figure 18 depend in a nonlinear way on the active variable `X`. Thus, the corresponding adjoint statements both reference `X`. But different values of `X` are required because the second statement changes the value of `X`. Therefore, the block obtained by applying scheme (26) to this example is not the adjoint block.

In order to solve this conflict, i.e. to ensure that the right value will be used by the adjoint statements, as in the case of conflicts caused by loops, there are different possibilities:

(1) The required value is stored during the code execution and then can be used in the adjoint code.

(2) The required value is assigned to an auxiliary variable and the adjoint statements use the auxiliary variable.

(3) The required value is recomputed. For a block of statements ABC, in contrast to the scheme (26), this yields the following scheme of adjoint block construction

$$AB\,C^* \quad A\,B^* \quad A^* \quad . \tag{27}$$

By this scheme an additional recomputation (A) of required variables is in-

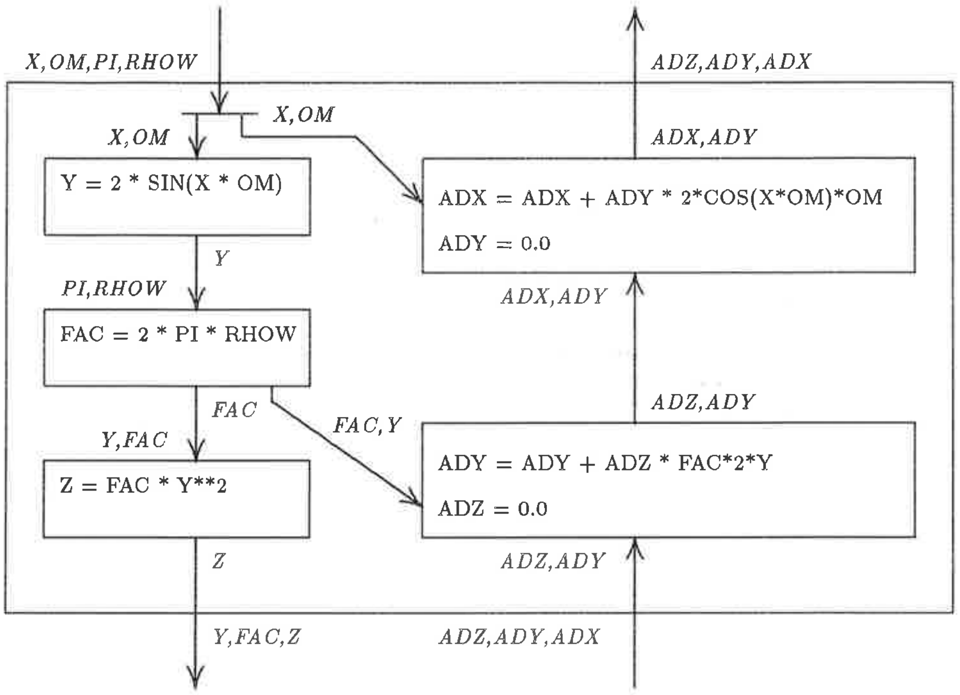


Fig. 16: Examples of an adjoint block of statements (X, Y, and Z are active variables, OM, FAC, PI, and RHOW are passive variables). The order of execution inside the blocks on the right hand side is top down.

```
Y   = 2 * SIN(X * OM)
FAC = 2 * PI * RHOW
ADY = ADY + ADZ * FAC*2*Y
ADZ = 0.0
ADX = ADX + ADY * 2*COS(X*OM)*OM
ADY = 0.0
```

Fig. 17. Adjoint code of a block of statements

cluded. The execution of $AB\ C^*$ might change variables used for computation of $A\ B^*$ and A^*. Therefore, to ensure that all required variables have correct values, some variables possibly have to be reset before execution of AB^*, or A^*, or both. Applied to the example in Figure 18, the variable X has to be reset before the second execution of the first assignment.

4.2.5 *Procedure call.* Procedures computing active variables are called active and a corresponding adjoint procedure has to be constructed. The adjoint statement of a procedure call is the call of the adjoint procedure, in some cases followed by some additional statements. The adjoint procedure itself contains the adjoint block of statements (see section 4.3.1). The variables required by the adjoint procedure are not known until the adjoint procedure has been constructed. Since the adjoint procedure call contains required variables as arguments, the formulation of the call may depend on details of the adjoint procedure. Thus, the construction of the adjoint procedure must precede the construction of its call[2]. When the adjoint procedure is called, the required variables must be provided (modularity, see section 4.1.4).

[2]This strategy is known as bottom up.

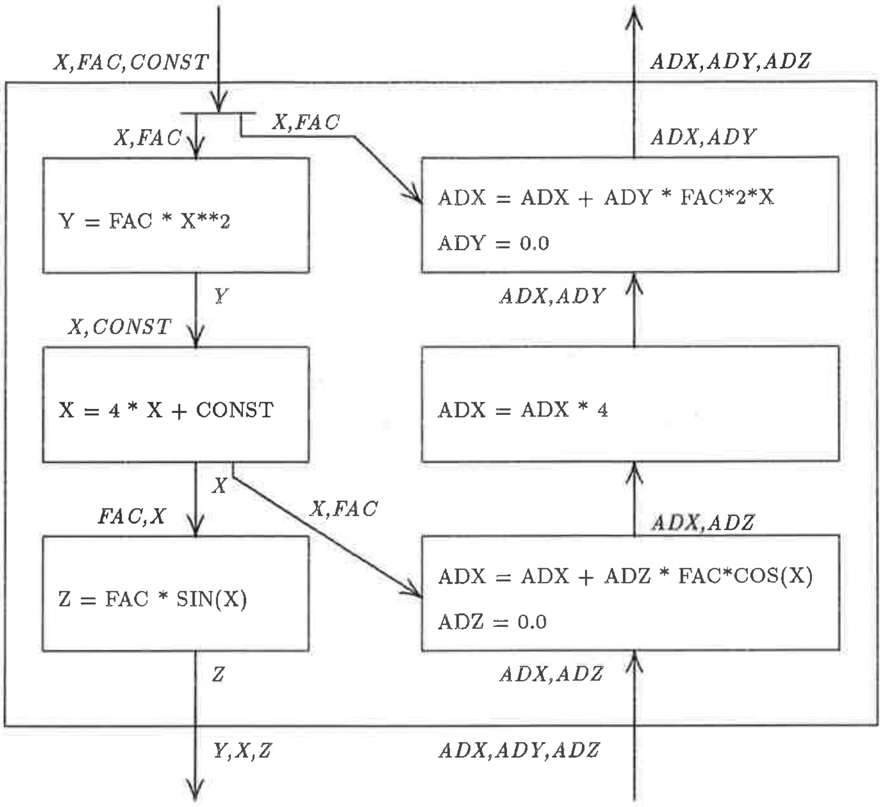


Fig. 18: Example of a conflict (X, Y, and Z are active variables, FAC and CONST are passive variables)

<u>code</u>	<u>adjoint code</u>
CALL SUB($\underline{\bar{X}}$, $\bar{Y}$, $\bar{A}$)	CALL ADSUB($\bar{X}$, $\bar{Y}$, $\bar{A}$, $\underline{\overline{ADX}}$, $\underline{\overline{ADY}}$)

Fig. 19: Example of an adjoint procedure call (A is a passive variable, X and Y are active variables)

Required arguments are passed as arguments as well. Global variables belonging to COMMON-blocks are taken into account by including the COMMON-block in the adjoint procedure. The argument list of the adjoint procedure consists of the required variables of the original argument list and the adjoint variables corresponding to the active variables of the original argument list. Obviously, the number and types of arguments in the call must correspond to the argument list of the adjoint procedure.

A simple example illustrating the call of an adjoint procedure is shown in Figure 19. Arguments used inside the procedure have a bar, while underlined arguments are computed by the procedure. The latter must be called by reference (a pointer to the argument is passed), the former, which only have a bar, may be called by value (the value of the argument is computed and passed).

In order to fulfill modularity we assume that each adjoint variable in an argument list will be used and computed inside the adjoint procedure. This is similar to the adjoint of an assignment (see section 4.2.1) where we regarded all involved active variables as input and output.

According to the Fortran-77 standard, expressions can also be an argument of a code

$\texttt{CALL SUB (}\overline{\texttt{X}}\texttt{, }\overline{\texttt{4*X+2*Y}}\texttt{, }\overline{\texttt{A}}\texttt{)}$

adjoint code

$\texttt{CALL ADSUB(}\overline{\texttt{X}}\texttt{, }\overline{\texttt{4*X+2*Y}}\texttt{, }\overline{\texttt{A}}\texttt{, }\underline{\overline{\texttt{ADX}}}\texttt{, }\underline{\overline{\texttt{ADP2}}}\texttt{)}$

```
ADX  = ADX + ADP2 * 4
ADY  = ADY + ADP2 * 2
ADP2 = 0.0
```

Fig. 20: Example of an adjoint procedure call having an expression in its argument list (A is a passive variable, X and Y are active variables)

hypothetical code

$\texttt{P2 = 4*X + 2*Y}$

$\texttt{CALL SUB(}\overline{\texttt{X}}\texttt{, }\overline{\texttt{P2}}\texttt{, }\overline{\texttt{A}}\texttt{)}$

adjoint hypothetical code

$\texttt{P2 = 4*X + 2*Y}$

$\texttt{CALL ADSUB(}\overline{\texttt{X}}\texttt{, }\overline{\texttt{P2}}\texttt{, }\overline{\texttt{A}}\texttt{, }\underline{\overline{\texttt{ADX}}}\texttt{, }\underline{\overline{\texttt{ADP2}}}\texttt{)}$

```
ADX  = ADX + ADP2 * 4
ADY  = ADY + ADP2 * 2
ADP2 = 0.0
```

Fig. 21: Substitution of a procedure call without expressions within the argument list (A is a passive variables, X, Y, and P2 are active variables)

procedure call. This saves introducing auxiliary variables. Assume the expression is substituted by an auxiliary variable and the expression is assigned to this variable just before the call. The call of the adjoint procedure is constructed as described in the previous section using the adjoint of the auxiliary variable. After this call, the adjoint of the assignment to the auxiliary variable is generated by applying the rules of section 4.2.1. If the expression is a required argument of the adjoint procedure, the auxiliary variable is replaced by the expression it stands for. Figure 20 shows an example. The hypothetical code avoiding expressions as arguments is shown in Figure 21. The adjoint auxiliary variable **ADP2** represents the indirect influence of the second argument, i.e. the expression **4*X+2*Y**, on the cost function by means of the procedure[3]. The expression itself depends on the active variables **X** and **Y**.

An example of the procedure **SUB** is given in section 4.3.1.

4.2.6 *Function call.* Functions return a result and are called only inside an expression. For user defined functions an adjoint procedure should be generated. Intrinsic functions can be directly differentiated. The adjoint of call to a user defined function is the call of the corresponding adjoint procedure. The argument list is constructed according to the rules for an adjoint procedure call (see above). This adjoint procedure has one more argument, namely the adjoint variable corresponding to the function result. In the example shown in Figure 22 this is the adjoint variable **ADZ**, the last argument in the list. If the LHS variable of the assignment is an argument of the function call, then the variable appears also on the RHS and an auxiliary variable must be introduced according to the general rule for an adjoint assignment. Otherwise two arguments of the adjoint procedure would be the same variable which is not allowed according to the Fortran-77 standard.

[3]Adjoint auxiliary variables must be initialized as local adjoint variables (see sec. 4.3.1).

code

$\texttt{Z = FCT(}\overline{\texttt{X}}\texttt{, }\overline{\texttt{Y}}\texttt{)}$

adjoint code

$\texttt{CALL ADFCT(}\overline{\texttt{X}}\texttt{, }\overline{\texttt{Y}}\texttt{, }\underline{\overline{\texttt{ADX}}}\texttt{, }\underline{\overline{\texttt{ADY}}}\texttt{, }\overline{\texttt{ADZ}}\texttt{)}$

```
ADZ = 0.0
```

Fig. 22. Example of an adjoint function call (X, Y and Z are active variables)

```
code                  adjoint code

Y = SIN(X)            ADX = ADX + ADY * COS(X)
                      ADY = 0.0
```

Fig. 23. Example of an adjoint predefined function call (X and Y are active variables)

```
code                              adjoint code

Z = 4*X + Y*FCT( X**2, 4*Y )      CALL ADFCT( X**2, 4*Y, ADP1, ADP2, ADZ*Y )
                                  ADX = ADX + ADZ*4                + ADP1*2*X
                                  ADY = ADY + ADZ*FCT( X**2, 4*Y ) + ADP2*4
                                  ADZ = 0.0
                                  ADP1 = 0.0
                                  ADP2 = 0.0
```

Fig. 24: Example of an adjoint function call having more than one active expression in its argument list (X, Y, and Z are active variables)

In case of an assignment containing a predefined function, the derivative of the function with respect to the argument is inserted in the RHS of the adjoint assignment, constructed according to the rule for the adjoint of an assignment (see sec. 4.2.1). Figure 23 shows an example.

An expression, which is the argument of a function call, is handled as described previously for a procedure call.

The additional influence of an active variable as an argument of a function is taken into account by an additional term inside the RHS expression of the assignment to the corresponding adjoint variable. This is shown in Figure 24. On the left hand side an assignment to the variable `Z` is shown. The expression on the RHS contains the call of the function `FCT`. The actual arguments of the function are expressions depending on active variables.

The left hand side of Figure 25 shows the hypothetical code avoiding expressions as arguments. The auxiliary variables `P1` and `P2` have been introduced to substitute these expressions and `FCTH` to hold the function result. The corresponding adjoint code is shown on the right hand side of Figure 25. The adjoint subprogram call is constructed by applying the rules described previously. The adjoint auxiliary variable `ADFCTH` might be replaced by the expression `ADZ*Y`, since this argument is called by value. Thus, this auxiliary variable is not needed. The assignments to the adjoint variable `ADX` (`ADY`) can be combined to a single assignment. Replacing the auxiliary variables `P1`, `P2`, and `FCTH` by the expressions they substitute, the adjoint code on the right hand side of Figure 24 is created.

4.2.7 *Input and output statements.* In terms of data flow, writing a value into a file and reading it is equivalent to assigning the value to a variable and referencing the variable. The only difference is the internal organization of how values are stored.

If active variables are written into a file and read from it, this file is called "active file" and a corresponding "adjoint file" has to be introduced. All I/O-statements effecting this file have corresponding adjoint statements. The values in the adjoint file represent the influence of the corresponding values in the active file on the cost function.

When a variable is read, the adjoint variable has to be set to zero, since its old value is lost and has no influence on the cost function. The value which is read affects the cost function by means of the variable. Hence, the value of the

```
hypothetical code             adjoint hypothetical code

P1   = X**2                   P1   = X**2
P2   = 4*Y                    P2   = 4*Y
FCTH = FCT( P1, P2 )          FCTH = FCT( P1, P2 )

Z = 4*X + Y*FCTH              ADX    = ADX    + ADZ*4
                              ADY    = ADY    + ADZ*FCTH
                              ADFCTH = ADFCTH + ADZ*Y
                              ADZ    = 0.0

                              CALL ADFCT( P1, P2, ADP1, ADP2, ADFCTH )
                              ADFCTH = 0.0
                              ADY  = ADY + ADP2*4
                              ADP2 = 0.0
                              ADX  = ADX + ADP1*2*X
                              ADP1 = 0.0
```

Fig. 25: Substitution of a function call without expressions within the argument list (X, Y, Z, P1 , P2, and FCTH are active variables). The lower group of statements on the right hand side is the adjoint to the upper group of statements on the left hand side.

```
code           hypothetical code    adjoint hypothetical code    adjoint code

OPEN(8)                             ADXD = 0.0                   OPEN(9)
   :              :                    :                            :
WRITE(8) X     XD = X               ADXD = ADXD + ADZ            WRITE(9) ADZ
                                    ADZ = 0.0                    ADZ = 0.0
   :              :                    :                            :
READ(8) Z      Z = XD               ADX = ADX + ADXD             READ(9) ADXD
                                    ADXD = 0.0                   ADX = ADX + ADXD
                                                                 ADXD = 0.0
   :              :                    :                            :
CLOSE(8)                                                         CLOSE(9)
```

Fig. 26: Example of adjoint I/O operations. Please note that the adjoint code shown on the right hand side is executed top down. Thus, the first adjoint statement corresponds to the last code statement.

corresponding adjoint variable has to be written into the adjoint file.

On the other hand, by writing a variable into the active file, the value in the file depends on the control variables. Thus, the corresponding value in the adjoint file has to be added to the adjoint variable.

The adjoint statement of an OPEN-statement is a CLOSE-statement and vice versa. Since the adjoint statements are combined in reverse order, the same order of I/O-operations will be applied to the adjoint file.

Figure 26 illustrates the construction of adjoint I/O-operations. On the left hand side a simple sequence of I/O-operations is given and in the middle the hypothetical code is shown.

Whenever a value of an active variable is read more than once, a modified algorithm of adjoint code construction must be used. The value in the adjoint file has to be changed as an adjoint variable due to a reference of the corresponding active variable. By a READ statement followed by a WRITE statement concerning the same file position a value is added to the value in the file. Hence, the adjoint file has to be a direct access file, as indicated by the additional **REC=** in Figure 27.

code	hypothetical code	adjoint hypothetical code	adjoint code
`OPEN(8)`		`ADXD = 0.0`	`OPEN(9,'ACCESS=DIRECT')`
⋮	⋮	⋮	⋮

`WRITE(8) X`	`XD = X`	`ADXD = ADXD + ADZ`
		`ADZ = 0.0`
			`WRITE(9,REC=1) ADZ`
			`ADZ = 0.0`

⋮	⋮	⋮	⋮

`READ(8) Y`	`Y = XD`	`ADXD = ADXD + ADY`
		`ADY = 0.0`
			`READ(9,REC=1) ADXD`
			`WRITE(9,REC=1) ADXD+ADY`
			`ADY = 0.0`

⋮	⋮	⋮	⋮

`REWIND(8)`
`READ(8) Z`
	`Z = XD`	`ADX = ADX + ADXD`
		`ADXD = 0.0`
			`READ(9,REC=1) ADXD`
			`ADX = ADX + ADXD`
			`ADXD = 0.0`

⋮	⋮	⋮	⋮
`CLOSE(8)`			`CLOSE(9)`

Fig. 27. Example of adjoint I/O operations of multiple input statements

The first write operation of the adjoint file is not preceded by a read operation, since the file is empty. Thus, the last read operation of the active file has to be identified in order to construct the adjoint code. Another method would be to initialize the adjoint file with zeroes in the same manner as global adjoint variables are initialized (see sec. 4.3.1) and always changing the values in the file. But this is not a safe approach, since the length of the active file is not known a priori and such an initialization takes additional run time.

In case an expression is written into an active file, additional adjoint code must be inserted after reading a value of the adjoint file in the same manner as for expressions in an argument list (see sec. 4.2.5).

4.3 Subprogram

4.3.1 *Procedures (subroutines).* A procedure uses and defines variables. Except for local variables, they are passed as arguments of the argument list or as global variables by a COMMON block. As a first step, the active variables are determined (for data dependence analysis see sec. 4.1.2). The adjoint variables of active variables of the argument list are included in the argument list of the adjoint procedure.

Once all active variables of the procedure are known, the adjoint code can be constructed as a block of statements. The required variables of the adjoint block must be provided by the calling subprogram. All required arguments of the original argument list should be passed to the adjoint procedure as arguments as well. Thus, the argument list of the adjoint procedure contains adjoint variables and required variables. How to handle required variables of COMMON blocks is explained below.

Many programming languages offer the possibility to pass names of subprograms by the argument list. Using Fortran-77, such a name must be characterized by an EXTERNAL declaration. If this subprogram is active and called, an adjoint name must be generated, declared as EXTERNAL, and included in the adjoint argument list.

Fortran-77 permits to pass return addresses as arguments of the argument list. In this case recognizing the structure of the whole program is very difficult and

```
SUBROUTINE SUB( X, Y, A )
IMPLICIT NONE

INTEGER N, I, J
PARAMETER( N = 100 )
REAL X(N), Y(N), A
REAL SUMX, SUMY, FAC

REAL XNORM
COMMON /COM1/ XNORM

SUMX = 0.0
SUMY = 0.0
DO I = 1,N
   SUMX = SUMX + X(I)
   SUMY = SUMY + Y(I)
END DO
FAC  = SUMX * SUMY * XNORM / A
DO I = 1,N
   J = N+1-I
   X(I) = FAC * (X(I)**2 + Y(J)**2)
END DO

END
```

Fig. 28. Example of a procedure

the construction of the adjoint code becomes a very complicated task. Thus, such construction of code should be avoided.

The declaration part of an adjoint procedure consists of the declaration of the required and the adjoint variables. An adjoint COMMON block is formed for every COMMON block containing at least one active variable. The adjoint COMMON block holds the corresponding adjoint variables. We assume that all variables in COMMON blocks are global variables. Thus, the adjoint COMMON blocks must be initialized with zeros before the adjoint code is executed. For this purpose a special procedure must be constructed. Usually, memory will be allocated dynamically for local variables whenever a procedure is called. A SAVE declaration changes this proceeding, the local variable will be static and keeps its value as if it was a global variable only used by the specific procedure containing the SAVE declaration. Thus a static active variable can be handled as a global active variable and a special adjoint COMMON block should be constructed containing all adjoint variables corresponding to the active variables in the SAVE declaration. This COMMON block can be initialized in the same procedure as the other adjoint COMMON blocks.

The statement part of the adjoint procedure starts with the initialization of the local adjoint variables. This is necessary because usually adjoint variables will be changed by addition of other values. This part is followed by the adjoint block of statements.

The construction of an adjoint procedure is illustrated by the example in Figure 28. Assume that the arguments `X` and `Y` are active variables as well as the global variable `XNORM` of the COMMON block `COM1`. Thus the local variables `SUMX`, `SUMY`, and `FAC` are active and the adjoint COMMON block `ADCOM1` is constructed.

The adjoint code shown in Figure 29 requires in addition to the arguments X, Y, and A also the global variable XNORM. Thus, the COMMON block COM1 containing XNORM is included in the declaration. The adjoint argument list contains the required variables X, Y, A and the adjoint variables ADX and ADY.

After the declaration part, the local adjoint variables ADSUMX, ADSUMY and ADFAC are initialized with zero. This is followed by the recomputation of SUMX, SUMY, and FAC since their values are required by the adjoint statements of the block. The last loop kernel defines and uses the variable J. Since J is also required by the adjoint loop kernel, its definition precedes this kernel.

The last two assignments and the last assignment to ADFAC are not necessary, since these adjoint local variables are not used afterwards. Nevertheless, they should remain in the adjoint code keeping the adjoint code quickly and savely adaptable to changes in the code (see modularity, sec. 4.1.4). Furthermore, code optimizing compilers would recognize these redundant statements and ignore them.

4.3.2 *Functions.* Functions are handled in the same way as procedures. The argument list is formed by applying the rules described in the preceding section. An additional argument holds the adjoint variable of the result of the function. As an example, the function FCT shown in Figure 30 consists only of one assignment. Since the RHS expression depends in a nonlinear way on the active variables X and Y, these variables must be passed through the adjoint argument list. The adjoint variables ADX and ADY and the adjoint variable of the result are also part of the list.

4.4 Storing of required variables

In case a required value of a variable ought not to be recomputed in the adjoint code, its value must be stored during the code execution. Therefore an additional statement must follow the computation of the required variable in the code. This should be the call of a special procedure. Just before execution of the adjoint statement requiring this variable, another procedure, which restores its value, should be called.

The values can either be stored on a temporary file, in dynamic memory, or in a global variable. The choice depends on the access time and the size of available memory.

The values are usually required in reverse order of computation. Thus, the values must be accessible independently. If a file is used, this should be a direct access file. The records of values can be accessed by a key composed of the name of the variable and the subprogram computing it. In the case the subprogram is called several times, the key must also contain the actual number of the call. Whenever the variable is stored inside one or more loops, all loop index variables must be taken into account additionally.

4.5 Problematic code-structures

Our recipes to construct the adjoint code directly from the programming code of the model do not apply for a few specific structures of the code. A short, probably incomplete list of such structures is given.

Since the order of the execution of statements in the code is important for the generation of adjoint code, statements, which may define a complicated order, do not have simple rules for adjoint code generation. This problem may arise in Fortran-77 when one of the statements ENTRY, RETURN, and GOTO is used. Fortran-77 permits to pass return addresses as arguments of the argument list. In this case

```
subroutine adsub(x, y, a, adx, ady)

integer n
parameter (n = 100)
common /com1/ xnorm
real xnorm
common /adcom1/ adxnorm
real adxnorm
real x(n), y(n), a
integer i, j
real sumx, sumy, fac
real adx(n), ady(n)
real adsumx, adsumy, adfac

adsumx = 0.0
adsumy = 0.0
adfac = 0.0
sumx = 0.0
sumy = 0.0
do i = 1,n
  sumx = sumx+x(i)
  sumy = sumy+y(i)
end do
fac = sumx*sumy*xnorm/a
do i = n,1,-1
  j = n+1-i
  adfac = adfac+adx(i)*(x(i)**2+y(j)**2)
  ady(j) = ady(j)+adx(i)*fac*2*y(j)
  adx(i) = adx(i)*fac*2*x(i)
end do
adsumx = adsumx+adfac*sumy*xnorm/a
adsumy = adsumy+adfac*sumx*xnorm/a
adxnorm = adxnorm+adfac*sumx*sumy/a
adfac = 0.0
do i = n,1,-1
  ady(i) = ady(i)+adsumy
  adx(i) = adx(i)+adsumx
end do
adsumy = 0.0
adsumx = 0.0

end
```

Fig. 29: Example of an adjoint procedure (this code was constructed by the AMC [Giering 1992])

<u>code</u>	<u>adjoint code</u>
`REAL FUNCTION FCT(` $\overline{\texttt{X}}$, $\overline{\texttt{Y}}$ `)`	`SUBROUTINE ADFCT(` $\overline{\texttt{X}}$, $\overline{\texttt{Y}}$, $\overline{\underline{\texttt{ADX}}}$, $\overline{\underline{\texttt{ADY}}}$, $\overline{\texttt{ADFCT}}$ `)`
`REAL X, Y`	`REAL X, Y, ADX, ADY, ADFCT`
`FCT = X*X + 2*Y*Y`	`ADX   = ADX + ADFCT*2*X`
	`ADY   = ADY + ADFCT*4*Y`
	`ADFCT = 0.0`
`END`	`END`

Fig. 30. Example of an adjoint function (X, Y, and FCT are active)

recognizing the structure of the whole program is very difficult and construction of the adjoint code becomes a very complicated task.

Fortran-77 permits static aliasing, i.e. different variables can share the same memory location. Usually this is realized by EQUIVALENCE statements. But in some constructs aliasing can not be detected only by analyzing the EQUIVALENCE statements. The dependency analysis may become very difficult in such cases.

In most cases we are aware of, such structures are not essential. We recommend to replace them by other elements of Fortran, which are consistent with the concept of structured programming.

5. ADJOINT OF CONVERGING SEQUENCES

In many simulation programs an implicit equation is solved [4] which has the form

$$x - f(x,p) = 0 \ , \tag{28}$$

where $p \in \mathrm{R^m}$ is given and $x \in \mathrm{R^n}$ is unknown and $f : \mathrm{R^n} \times \mathrm{R^m} \to \mathrm{R^n}$. If a pair (x,p) fulfills (28), f is sufficiently regular, and

$$1 - \frac{\partial f}{\partial x}(x,p) =: 1 - A(x,p) \tag{29}$$

is invertible the implicit function theorem applies. It yields the existence of a neighborhood $U \in \mathrm{R^m}$ of p and of a differentiable function $g : U \to \mathrm{R^n}$ so that for $\tilde{p} \in U$ the pair $(\tilde{x} = g(\tilde{p}), \tilde{p})$ fulfills (29) and the derivative of g is

$$\frac{\partial g}{\partial p}(p) = (1 - A(g(p),p))^{-1} \cdot B(g(p),p) \ , \tag{30}$$

where

$$B(x,p) := \frac{\partial f}{\partial p}(x,p) \ . \tag{31}$$

Assuming equation (28) in the code is solved by an iterative method:

$$\begin{array}{lrcll}
\mathbf{S1}: & x_0 & = & \text{first guess} & \\
\mathbf{S2}: & x_n & = & f(x_{n-1},p) & (n = 1, ..., N) \\
\mathbf{S3}: & x & = & x_N \ , &
\end{array}$$

where and N is the number of steps performed to reach an appropriate accuracy, the adjoint algorithm is also an iteration consisting of the corresponding adjoint operators:

$$\begin{array}{lrcll}
\mathbf{S3^*}: & \delta^* x_N & = & \delta^* x & \\
 & \delta^* x & = & 0 & \\
 & & & & \\
\mathbf{S2^*}: & \delta^* x_{n-1} & = & A^*(x_{n-1},p)\,\delta^* x_n & (n = 1, ..., N) \\
 & \delta^* p & = & B^*(x_{n-1},p)\,\delta^* x_n & \\
 & \delta^* x_n & = & 0 & \\
 & & & & \\
\mathbf{S1^*}: & \delta^* x_0 & = & 0 \ . &
\end{array}$$

If the function f is nonlinear, A or B depend on the intermediate result x_{n-1}. In this case, the adjoint iteration requires all intermediate results. Thus, they must be stored during the iteration and restored during the adjoint iteration.

[4] When integrating dynamical systems, implicit equations occur due to an implicit or semi implicit time step.

```
code                          adjoint code

X = first guess
DO I = 1, N                   DO I = N, 1, -1
   store X                       restore X
   CALL FCT( X, P )              CALL ADFCT( X, P, ADX, ADP )
END DO                        END DO
                              ADX = 0.0
```

Fig. 31: Iterative algorithm for solving an implicit equation and the exact adjoint code (P and X are active variables)

Figure 31 shows on the left hand side a simple example, where the iteration is implemented as a DO-loop and X is computed depending on P. Suppose the nonlinear function FCT is given by a subprogram and its adjoint ADFCT has been constructed according to the rules described in section (4). Since **S2*** is the adjoint operation to **S2**, the adjoint subprogram ADFCT is an implementation of **S2***. The corresponding adjoint code is shown on the right hand side of Figure 31, where the required intermediate results are restored before execution of the adjoint loop kernel.

Depending on the number of iterations, storing might require a huge amount of memory. Hence, it would be advantageous to construct adjoint code, which does not require intermediate results. Assuming that in the code the exact solution $x = g(p)$ of (28) has been computed this is possible. Equation (30) can be used to compute a variation of x that results from from a variation of p:

$$\delta x = (1 - A(g(p), p))^{-1} \cdot B(g(p), p)\delta p \ . \tag{32}$$

Introducing

$$\delta z := B \, \delta p \tag{33}$$

according to (23) for the adjoints one obtains

$$\delta^* x = [(1 - A)^{-1}]^* \, \delta^* z \ . \tag{34}$$

Using basic properties of the adjoint yields an implicit equation for $\delta^* z$:

$$\delta^* z = A^* \, \delta^* z \, + \, \delta^* x \ . \tag{35}$$

Christianson [Christianson 1993] has shown that this equation can be solved by an iteration that determines $\delta^* z$ from $\delta^* x$:

$$\begin{aligned} \delta^* z_N &= \delta^* x \\ \\ \delta^* z_{n-1} &= A^*(x,p)\, \delta^* z_n \, + \, \delta^* x \qquad (n = 1, ..., N) \\ \delta^* z_n &= 0 \\ \\ \delta^* z &= \delta^* z_0 \ . \end{aligned}$$

and that the convergence is as fast as for the iteration solving (28).

Using (33) the adjoint variable $\delta^* p$ can be computed from the solution $\delta^* z$ by

$$\delta^* p = B^*(x,p)\, \delta^* z \ . \tag{36}$$

Since A and thus A^* do not depend on the intermediate results x_{n-1}, only the solution x of the iteration and p must be provided for the adjoint iteration. Computation of (36) after the iteration saves run time compared to the exact adjoint code. However, for the implementation the aim is to transform this algorithm into

```
code                          adjoint code

X = first guess               restore X
                              ADZ = ADX
                              ADPH = ADP
DO I = 1, N                   DO I = N, 1, -1
                                 ADP = 0.0
   CALL FCT( X, P )              CALL ADFCT( X, P, ADX, ADP )
                                 ADX = ADX + ADZ
END DO                        END DO
store X                       ADX = 0.0
                              ADP = ADP + ADPH
```

Fig. 32: Iterative algorithm for solving an implicit equation and the adjoint code without requiring intermediate results (P and X are active variables)

an adjoint algorithm in which the adjoint operators A^* and B^* can be replaced by the call of the adjoint procedure **ADFCT**.

In a first step the algorithm can be transformed by splitting the assignment to $\delta^* z_{n-1}$ into two assignments and including the computation of $B^*\delta^* z$ in the iteration:

$$\begin{aligned}
\delta^* z_N &= \delta^* x \\[1ex]
\delta^* z'_{n-1} &= A^* \, \delta^* z_n \qquad\qquad (n = 1, ..., N) \\
\delta^* p &= B^* \, \delta^* z_n \\
\delta^* z_n &= 0 \\[1ex]
\delta^* z_{n-1} &= \delta^* z'_{n-1} + \delta^* x \\[1ex]
\delta^* z &= \delta^* z_0 \ .
\end{aligned}$$

Observing that **S2*** appears in this iteration operating on $\delta^* z_n$ instead of $\delta^* x_n$, the adjoint procedure **ADFCT** can be used again for implementation. In order to keep the argument names for **ADFCT** the variables **ADX** and **ADZ** are exchanged. The resulting adjoint code is shown in Figure 32.

The adjoint auxiliary variable **ADZ** is initialized with the value of the adjoint variable **ADX**. During the iteration this variable is added to the intermediate result **ADX**. The previous value of **ADP** is saved in an auxiliary variable **ADPH**. After the loop the saved previous value of **ADP** is added. Inside the adjoint procedure **ADFCT** values are always added to **ADP**. To guarantee that only the last iteration determines **ADP**, it is initialized with zero before the call of **ADFCT**. Since the iteration converges, this has the same effect as computing $B^*\delta^* z$ after the iteration.

6. VECTORIZATION OF ADJOINT CODE

Numerical simulations run mostly on high-performance computer systems characterized by vector pipes or processors working parallel. In order to take the best advantage of these features, optimizing compilers are used. In addition to translating the source program into machine language, they analyze it and apply various transformations to it.

Among the various forms of analysis used by optimizing compilers, we make use of dependence analysis in this section (a detailed description is given in [Banerjee 1988a; Banerjee 1988b; Bacon et al. 1993]). Two statements have a data dependence if they cannot be executed simultaneously due to conflict uses of the same variable. Loop-carried data dependences are dependences between statements due to their

```
code                                  non vectorizable adjoint code

                                      ADHSCAL = 0.0
                                         :
                                         :
DO I = 1, N                           DO I = N, 1, -1
   HSCAL = Z(I)                          ADHSCAL = ADHSCAL + ADX(I)
   X(I)  = X(I) + HSCAL                  ADZ(I)  = ADZ(I) + ADHSCAL
                                         ADHSCAL = 0.0
END DO                                END DO
```

Fig. 33: Example of adjoint code generated by described recipes, which will not be vectorized (**X**, **Z**, **HSCAL** are active variables)

```
code after scalar expansion           code after loop distribution

DO I = 1, N                           DO I = 1, N
   HVECT(I) = Z(I)                       HVECT(I) = Z(I)
                                      END DO
                                      DO I = 1, N
   X(I) = X(I) + HVECT(I)                X(I) = X(I) + HVECT(I)
END DO                                END DO
HSCAL = Z(N)                          HSCAL = Z(N)
```

Fig. 34. Transformed loop

repeated execution in a loop kernel. There are three kinds of data dependences: flow dependence, anti dependence, and output dependence.

It turns out that vectorization of loops applied to the code by the optimizing compiler will not be applied in some cases to the corresponding adjoint code generated according to the rules described previously. The structure of the adjoint code often has additional data dependences, that eventually prevent vectorization.

At present optimizing compilers are not yet able to detect the correctness of transformations in all cases. Thus, the compiler needs support. Two cases will be described in which the adjoint code can be modified in order to allow vectorization.

6.1 Temporary variables

In case a temporary variable holds an intermediate result inside a loop kernel and its actual value does not depend on results of the previous loop passes, a transformation called scalar expansion[5] of the temporary variable can be applied to the loop. This transformation deletes the loop-carried dependence between the statement using the temporary variable and the statement defining it. Thus, a second transformation called loop distribution is permitted, that splits the loop in two loops.

Figure 33 shows an example, where **HSCAL** is the temporary variable. The transformed code after applying scalar expansion to **HSCAL** is shown on the left hand side of Figure 34. Substituting the scalar **HSCAL** by the array **HVECT**, the variable **HSCAL** is not computed anymore. Hence an additional statement is inserted after the loop by the optimizing compiler, which assigns to **HSCAL** the value of **HSCAL** after the last pass of the original loop (left hand side of Fig. 34). After that, the loop is split into the loops as shown on the right hand side. These loops can be vectorized.

Since, in the adjoint code, values are added to adjoint variables, the actual value of the corresponding adjoint temporary variable **ADHSCAL** depends on its value after

[5]The variable is replaced by another variable having an additional dimension, which denotes the loop index.

vectorizable adjoint code

```
ADHSCAL = 0.0
   :
ADZ(N) = ADZ(N) + ADHSCAL
DO I = N, 1, -1
   ADHSCAL = ADX(I)
   ADZ(I)  = ADZ(I) + ADHSCAL
END DO
ADHSCAL = 0.0
```

Fig. 35. Modified adjoint code which can be vectorized

code

```
DO I = 1, N
   Z(I) = X(I) + X(I-1)


END DO
```

adjoint code

```
DO I = N, 1, -1
   ADX(I)   = ADX(I)   + ADZ(I)
   ADX(I-1) = ADX(I-1) + ADZ(I)
   ADZ(I)   = 0.0
END DO
```

Fig. 36. Example of adjoint code which will not be vectorized (X and Z are active variables)

the previous loop pass (right hand side of Fig. 33). This additional dependence hinders the loop distribution of **ADHSCAL**. However, this dependence can be removed by simple modifications to the adjoint code. Observing that **ADHSCAL** is reset to zero at the end of the loop kernel, the addition of an expression to **ADHSCAL** has the same effect as assigning the expression to it. A possible non zero value of **ADHSCAL** before the loop execution can be considered by an additional statement in front of the loop. By changing the code in this way, the code shown in Figure 35 is obtained. Since **ADHSCAL** is zero after the execution of the adjoint loop it must be reset to zero after the modified adjoint loop.

Scalar expansion of **ADHSCAL** and loop distribution will be applied to the modified adjoint loop by an optimizing compiler. We emphasize that inside the adjoint loop kernel modularity is lost by these modifications.

6.2 Simultaneous references to different array elements

Suppose inside a loop kernel an active variable appears several times on the RHS of an assignment. If the variable is an array and the subscript expressions reference different components, then the corresponding adjoint array components occur on the LHS and the RHS of the adjoint assignments. In this case the adjoint loop has additional loop-carried data dependences hindering vectorization.

Figure 36 shows an example, where the code on the left hand side is a loop, whose kernel is an assignment. The array **X** is appears twice on the RHS with different subscript expressions. The adjoint loop kernel on the right hand side of Figure 36 consists of three statements. The statement **S1** depends on **S2** of the previous loop pass, because **S2** computes a component of the array **ADX** and **S1** references the same component in the next loop pass. Loop distribution (see section 6.1) is not allowed in this case, since all statements **S2** would be executed after all statements **S1**, which violates the loop-carried dependence (Fig. 37). However, because adding is a commutative operation, any order of execution of the statements yields the same result. Thus, the transformation would be correct.

The order of the statements **S1** and **S2** is not laid down by the rule for an

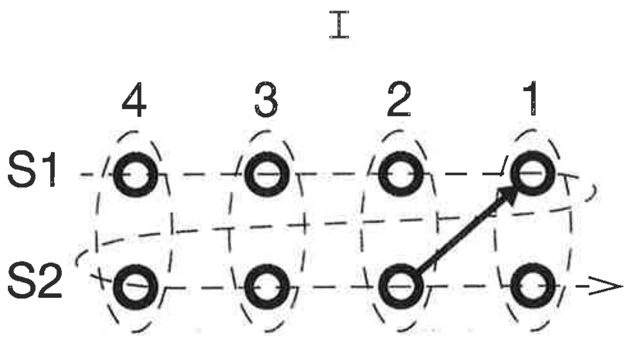


Fig. 37: A loop containing S1 and S2 cannot be distributed when S1 depends on S2. The dashed arrow denotes the execution order after loop distribution. This is a modified picture from [Bacon et al. 1993] page 24.

new adjoint code

```
DO I = N, 1, -1
   ADX(I-1) = ADX(I-1) + ADZ(I)
   ADX(I)   = ADX(I)   + ADZ(I)
   ADZ(I)   = 0.0
END DO
```

Fig. 38. Modified adjoint code which will be vectorized (X and Z are active variables)

adjoint assignment. By exchanging **S1** and **S2**, the adjoint code in Figure 38 is retrieved. The loop-carried dependence of this adjoint loop is not violated by a loop distribution, because all statements **S2** are executed before the statements **S1**. Hence, this loop will be vectorized by an optimizing compiler.

In general, the adjoint assignments to variables, which have the latest subscript value with respect to the execution order of the loop kernel, should occur first inside the adjoint loop kernel.

7. SUMMARY AND CONCLUSIONS

Simple rules for adjoint code generation have been deduced for the most important elements of numerical programs. The construction of the adjoint code of a Fortran-77 subprogram formed by these elements has been described.

It has been shown that alternative adjoint code can be constructed for iterative procedures solving implicit equations. If the equations are non linear this alternative code needs less memory resources. Modifications to the adjoint code have been described in order to allow vectorization of adjoint loops, which correspond to loops that can be vectorized. In summary, these general directions allow for construction of adjoint code, which is easy to maintain, efficient, and fast.

For the execution of the adjoint code many intermediate results are required from the code that computes the cost function. Problems arise whenever these results are not accessible. Thus, the code must be analyzed to detect loss of values. For loops a data dependence analysis similar to that for vectorization of loops is required. In order to provide the intermediate results, one has to choose between recomputation or recording the values into memory or file. Large memory or disc resources might be necessary to record all required variables. On the other hand recomputation consumes additional run time. We recommend a mixed strategy, which combines recomputation and recording into memory and file. The best combination depends on details of the application and on features of the computer which executes the adjoint code. Further research is necessary in this direction.

Most of the described rules are implemented in the Adjoint Model Compiler [Giering 1992]. Given the toplevel routine to be differentiated and its independent and dependent variables, the AMC determines all active variables and subroutines by an interprocedural data dependence analysis. The adjoint routines are constructed bottum up, in which recomputations are inserted wherever required. Alternatively, directives can be inserted into the source code. In this case statements to store (restore) will be inserted in the automatically generated code (adjoint code).

Using the AMC several adjoint codes have been constructed. Among them are some complex models designed to integrate components of the climate system: Hamburg Large Scale Ocean General Circulation Model (LSG) [Maier-Reimer and Miokolajewicz 1992] , Hamburg Ocean Primitive Equation model (HOPE) [Latif and Barnett 1984] , Tracer Model (TM2) [Heimann 1995] , and MIT ocean model [Marshall 1995]. Basically, large disc or memory resources are required for storing the trajectory of non linear models. However, resources were reduced considerable by introducing additional checkpoints [Griewank 1992] with the cost of an additional code run during adjoint code computation. Finally, those adjoint models need between 2.5 and 4 times the execution time of the respective model. Some Fortran extensions have been implemented by the AMC in order to read and generate code for a parallel machine (Connection Machine CM5). Further extensions especially of High Performance Fortran are planed.

REFERENCES

BACON, D., GRAHAM, S., AND SHARP, O. 1993. Compiler Transformation for High-Performance Computing. Technical Report UCB/CSD-93-781, Computer Science Division, University of California, Berkley, California 94720.

BANERJEE, U. 1988a. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Massachusetts.

BANERJEE, U. 1988b. An introduction to a formal theory of dependence analysis. *Journal Supercomputing 2*, 133–149.

BAUR, W. AND STRASSEN, V. 1983. The complexity of partial derivatives. *Theoretical Computer Science 22*, 317–330.

BISCHOF, C., CARLE, A., KHADEMI, P., AND MAURER, A. 1994. 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.

CACUCCI, D. 1981. Sensitivity theory for nonlinear systems. Part I: Nonlinear functional analysis approach. *Journal of Mathematical Physics 22*, 2794–2812.

CHRISTIANSON, B. 1993. Reverse accumulation and attractive fixed points. Internal report, School of Information Sciences, University of Hertfordshire, England.

COURTIER, P. AND TALAGRAND, O. 1987. Variational assimilation of meteorological observations with the adjoint vorticity equation Part II : Numerical results. *Quarterly Journal of the Royal Meteorological Society 113*, 1329–1347.

GHIL, M. 1989. Meteorological data assimilation for oceanographers. Part I: Description and theoretical framework. *Dynamics of Atmospheres and Oceans 13*, 171–218.

GIERING, R. 1992. *Adjoint Model Compiler, users manual.* Max-Planck-Institut für Meteorologie.

GIERING, R. AND MAIER-REIMER, E. 1996. Data assimilation into the Hamburg LSG OGCM with its adjoint model. in prep.

GRIEWANK, A. 1989. On automatic differentiation. In M. IRI AND K. TANABE (Eds.), *Mathematical Programming: Recent Developments and Applications*, pp. 83–108. Dordrecht: Kluwer Akedemic Publishers.

GRIEWANK, A. 1992. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software 1*, 35–54.

HEIMANN, M. 1995. The Global Atmospheric Tracer Model TM2. Technical Report 10, Deutsches Klima-Rechenzentrum, Hamburg, Germany.

LATIF, M. AND BARNETT, T. 1984. Causes of decadel climate variability over the North Pacific and North america. *Science 266*, 634–637.

LONG, R. AND THACKER, W. 1989a. Data assimilation into a numerical equatorial ocean model. Part I : The model and assimilation algorithm. *Dynamics of Atmospheres and Oceans 13*, 379–412.

LONG, R. AND THACKER, W. 1989b. Data assimilation into a numerical equatorial ocean model. Part II : Assimilation experiments. *Dynamics of Atmospheres and Oceans 13*, 413–440.

LOUIS, J.-F. 1991. Use of the adjoint to optimize model parameters. In *Supplement to vol.9, EGS 16th General Assembly*, Wiesbaden, Germany, pp. C107.

MAIER-REIMER, E. AND MIOKOLAJEWICZ, U. 1992. The Hamburg Larg Scale Geostrophic Ocean Genearl Circulation Model (cycle 1). Technical Report 2, Deutsches Klima-Rechenzentrum, Hamburg, Germany.

MARSHALL, J. 1995. A nonhydrostatic ocean model. Technical Report 2, Messachusetss Institut of Technology, Cambridge, USA.

NAGLIB 1987. *Fortran Library Manual - Mark 13.* Numerical Algorithms Group.

ROSTAING, N., DALMAS, S., AND GALLIGO, A. 1993. Automatic differentiation in Odyssée. *Tellus A,* 45, 558–568.

SCHRÖTER, J. 1989. Driving of non-linear time dependent ocean models by observations of transient tracer - a problem of constrained optimization. In D. ANDERSON AND J. WILLEBRAND (Eds.), *Ocean Circulation Models: Combining Data and Dynamics*, pp. 257–285. Kluwer Academic Publishers.

SCHRÖTER, J. 1992. Variational assimilation of GEOSAT data into an eddy-resolving model of the gulf-stream extension area. *Journal of Physical Oceanography 23*, 925–953.

TALAGRAND, O. 1991. The use of adjoint equations in numerical modeling of the atmospheric circulation. In A. GRIEWANK AND G. CORLIESS (Eds.), *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pp. 169–180. Philadelphia, Penn: SIAM.

TALAGRAND, O. AND COURTIER, P. 1987. Variational assimilation of meteorological observations with the adjoint vorticity equation Part I : Theory. *Quarterly Journal of the Royal Meteorological Society 113*, 1311–1328.

THACKER, W. 1991. Automatic differentiation from an oceanographer's perspective. In A. GRIEWANK AND G. CORLIESS (Eds.), *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pp. 191–201. Philadelphia, Penn: SIAM.

TZIPPERMAN, E. AND THACKER, W. 1989. An optimal control/adjoint equation approach to studying the ocean general circulation. *Journal of Physical Oceanography 19*, 1471–1485.

TZIPPERMAN, E., THACKER, W., LONG, R., AND HWANG, S.-M. 1992. Ocean data analysis using a general circulation model, I, simulations. *Journal of Physical Oceanography 22*, 1434–1457.

TZIPPERMAN, E., THACKER, W., LONG, R., HWANG, S.-M., AND RINTOUL, S. 1992. Ocean data analysis using a general circulation model, II, North Atlantic model. *Journal of Physical Oceanography 22*, 1458–1485.

WEBSTER, S. AND HOPKINS, B. 1994. Adjoints and singular vectors in a barotropic model. In *Workshop on Adjoint Applications in Dynamic Meteorology.*

ACKNOWLEDGMENTS

We thank Christian Bischof, Gerd Bürger, Christian Eckert , Klaus Hasselmann, Martin Heimann, Frank Kauker, Axel Kleidon, Ernst Maier-Reimer, Matthias Münnich, and Andreas Schiller for helpful discussions and Marko Laudam and Norbert Noreiks for preparing the figures.

The work was supported in part by the Bundesminister für Forschung und Technologie (grant no. 03F0157A).

```
PROGRAM OPTIM
EXTERNAL QGBFUN
INTEGER N
PARAMETER ( N = ..  )
REAL X(N), FC, ADX(N)

   :

!  initialization of X() and the model

   :

!  start of optimization
CALL E04DGF( N, QGBFUN, ..., FC, ADX, X, ...)

   :

END

SUBROUTINE QGBFUN ( N, X, FC, ADX )
INTEGER N, I
REAL X(N), FC, ADX(N), ADFC

!  computation of the cost function
CALL COST ( N, X, FC )

!  initialization of the local adjoint variables
ADFC = 1.0
DO I = 1,N
   ADX(I) = 0.0
ENDDO

!  initialization of the global adjoint variables
CALL ADZERO

!  computation of the gradient vector by the adjoint model
CALL ADCOST (N, X, ADX, ADFC )
END
```

Fig. 39: Example of an optimization program (Arguments used inside the procedure have a bar, while underlined arguments are computed by the procedure)

APPENDIX A. OPTIMIZATION

In this section we describe how the adjoint model is effectively used for minimization of a cost function (see eq. 4).

The structure of an optimization program is shown in Figure 39. In this example the routine E04DGF of the NAG library [NAGLIB 1987] is used.

The array **X** holds the values of the control variables, and the integer variable **N** denotes its length. The control variables must be initialized by a first guess before the start of the optimization. In order to reduce run time, the initialization of the model is separated and performed only once before the optimization.

The usage of **E04DGF** requires a subroutine **QGBFUN**, which computes the cost function and its gradient vector. In the example **QGBFUN** calls the subroutine **COST** to compute the cost function **FC** for a given vector of control variables **X**. The local adjoint variables **ADX** and **ADFC** are initialized. The global adjoint variables are reset to zero by the subroutine **ADZERO**. Finally, the adjoint model subroutine **ADCOST** is called to compute the gradient vector **ADX**.