RWTH Aachen

Department of Computer Science
Technical Report

Adjoint Code by Source Transformation with OpenAD/F

Uwe Naumann et al.

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2006-05

RWTH Aachen · Department of Computer Science · May 2006

Adjoint Code by Source Transformation with OpenAD/F

Uwe Naumann*, Jean Utke†, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch††, Mike Fagan, Nathan Tallent†††, and Michelle Strout††††

*RWTH Aachen University
Department of Computer Science
52056 Aachen, Germany
naumann@stce.rwth-aachen.de

†Argonne National Laboratory
Mathematics and Computer Science Division
9700 South Cass Avenue, Argonne, IL 60439, USA
utke@mcs.anl.gov

††Massachusetts Institute of Technology
Department of Earth, Atmospheric and Planetary Sciences
77 Massachusetts Avenue, Cambridge, MA 02139, USA
heimbach@mit.edu

†††Rice University
Department of Computer Science
P.O. Box 1892, MS 132, Houston, TX 77251, USA
mfagan@cs.rice.edu

††††Colorado State University
Department of Computer Science
1873 Campus Delivery, Fort Collins, CO 80523, USA
mstrout@cs.colostate.edu

Abstract. This document reports on recent advances in the development of the adjoint code generator OpenAD/F. We give an overview of the software design, and we discuss case studies that illustrate the feasibility of adjoint code generation. Our main target application is the MIT General Circulation Model — a numerical model designed for study of the atmosphere, ocean, and climate.

1 INTRODUCTION

The "Adjoint Compiler Technology and Standards" (ACTS) project is a collaborative research and development effort in automatic differentiation (AD) [5, 10, 11, 16] focusing on its application and next-generation tool development. The main result of our work is a modular, open source AD infrastructure (OpenAD) that has been used to implement tangent-linear and adjoint code generators. Both algorithms were applied successfully to a number of practical problems. The specific objectives of the ACTS project are as follows:

1. Definition and implementation of a programming language independent intermediate representation
2. Provision of a platform for the development of AD algorithms for programs given in this intermediate format
3. Adaptation of an existing Fortran front- and back-end to parse into this intermediate representation as well as unparse from it into Fortran derivative code

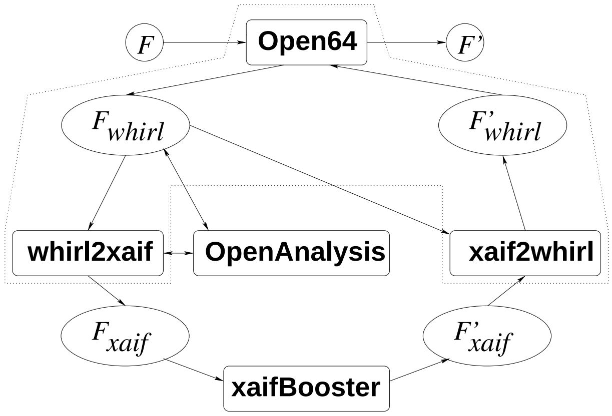


Fig. 1. The Open64 front-end performs a lexical, syntactic, and semantic analysis and produces an intermediate representation of F in the whirl format. OpenAnalysis is used to build call and control flow graphs, used by whirl2xaif to construct a representation of the numerical core of F in the xaif format. A differentiated version of F_{xaif} is derived by an algorithm that is built with xaifBooster. F'_{xaif} and F_{whirl} are combined by xaif2whirl to construct a whirl representation of the differentiated code. The unparser of Open64 transforms the latter into Fortran.

4. Automatic generation of tangent-linear and adjoint versions of the MIT general circulation model.

All objectives were met. Detailed information on the OpenAD framework and the Fortran tool OpenAD/F can be found on the web.

`http://www.mcs.anl.gov/openad` .

The basic architecture of this software package is illustrated in Figure 1 together with a brief description of the steps performed during the semantic transformation of a program F into a differentiated program F'. More details are presented in the following sections. The dotted line encloses the language-specific front-end that can potentially be replaced by front-ends for languages other than Fortran. For example, an effort running parallel to the ACTS project at Argonne National Laboratory is developing a new version of ADIC [22] by coupling a C/C++ front-end based on an EDG parser (`www.edg.com`) and uses ROSE in combination with SAGE 3 (`www.llnl.gov/CASC`) as internal representation with OpenAD.

The structure of this paper is as follows. In Section 2 we discuss the adjoint code that is generated automatically. Section 3 contains brief descriptions of the main components of OpenAD/F. Several case studies are discussed in Section 4. Conclusions and comments on ongoing and future work are presented in Section 5.

2 ADJOINT CODE BY OPENAD/F

In this section we discuss the mathematical and algorithmic basis for the code transformation algorithms within OpenAD/F that produce adjoint code automatically.

We consider computer programs that evaluate vector functions $\mathbf{y} = F(\mathbf{x})$ with

$$F : I\!R^n \rightarrow I\!R^m \quad .$$

We assume that F is once continuously differentiable. Hence, the Jacobian matrix $F' \equiv F'(\mathbf{x})$ of F exists and can be computed by AD.

AD expects the evaluation of F to decompose into a sequence of elemental computational operations

$$v_j = \varphi_j(v_i)_{i \prec j} \qquad \text{for } j = 1, \ldots, q \quad . \tag{1}$$

We adopt Griewank's notation [16]. Hence, $i \prec j$ if v_i is an argument of φ_j and $q = p + m$. We set $v_j = x_{j+n}$ for $j = 1 - n, \ldots, 0$ and $y_j = v_{p+j}$ for $j = 1, \ldots, m$. Refer to [16] for a comprehensive discussion of AD. A large number of successful applications of AD to real-world problems in science and engineering are described in [11, 5, 10, 7].

Under the usual assumptions about differentiability of the elemental functions we can compute local partial derivatives

$$c_{ji} \equiv \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \qquad \text{for } j = 1, \ldots, q \quad . \tag{2}$$

Forward-mode AD computes $\dot{\mathbf{y}} = F' \cdot \dot{\mathbf{x}}$ as

$$\dot{v}_j = \sum_{i \prec j} c_{ji} \cdot \dot{v}_i \qquad \text{for } j = 1, \ldots, q \quad . \tag{3}$$

Alternatively, the transposed product $\bar{\mathbf{x}} = (F')^T \cdot \bar{\mathbf{y}}$ can be computed in reverse-mode AD after initializing $\bar{v}_i = 0$ for $i = 1 - n, \ldots, p$ as

$$\bar{v}_i += c_{ji} \cdot \bar{v}_j \qquad \text{for } j = p, \ldots, 1 - n, \; i \prec j \quad . \tag{4}$$

Such adjoints are of particular interest in large-scale optimization because they allow for gradients to be computed with a computational complexity that is independent of n.

2.1 Preaccumulation and Reversal of Data Flow

In OpenAD/F we view all basic blocks (sequences of assignments that have a single control-flow entry and exit, respectively) as functions $\phi_j : I\!R^{n_j} \rightarrow I\!R^{m_j}$. Such local functions compute $\mathbf{y}^{(j)} = \phi_j(\mathbf{x}^{(j)})$. The local Jacobians $\phi'_j \in I\!R^{m_j \times n_j}$ are preaccumulated during a correspondingly augmented execution of the original code — the *augmented forward section* of the adjoint code — as described in [33]. Their entries are stored. The reverse section of the adjoint code restores the local Jacobians in reverse order (see Section 2.2) to compute $\bar{\mathbf{x}}^{(j)} += \phi_j^T \cdot \bar{\mathbf{y}}^{(j)}$ (compare with Equation (4)). The adjoints $\bar{\mathbf{y}} \in I\!R^m$ of the dependent output variables of F become inputs of the adjoint code to obtain $\bar{\mathbf{x}} = (F')^T \cdot \bar{\mathbf{y}}$.

We use *association by address* in order to attach adjoints to the original values of program variables. Therefore we define a new derived type `type(active)` that provides memory for both the value of the original variable (accessed via `x%v` for an active `x`) and its adjoint (`x%d`). All *active* [18] program variables that could potentially carry nonzero adjoint values are redeclared to be of this new type.

In general, the input vectors $\mathbf{x}^{(j)}$ may contain variables v also occurring in the output vectors $\mathbf{y}^{(j)}$ either by simply using the same variable or by *aliasing*; that is, some $x_k^{(j)}$ and some $y_l^{(j)}$ have different names but share the same address in memory. In practice this situation happens through the use of pointers, different dummy parameters bound to the same actual parameter or, for example, array dereferencing with syntactically different index expressions yielding the same value. Consider a simple example with a pointer variable `y2`, pointing to `x1`, that is, in Fortran `y2=>x1`. For two statements `y1=sin(x1); y2=x1*x2;` the naive application of Equation (4) yields the adjoints $\overline{\texttt{x2}}\texttt{=}\overline{\texttt{x2}}\texttt{+}c_{2,0}\texttt{*}\overline{\texttt{y2}}\texttt{;}$ $\overline{\texttt{x1}}\texttt{=}\overline{\texttt{x1}}\texttt{+}c_{2,-1}\texttt{*}\overline{\texttt{y2}}\texttt{;}$ $\overline{\texttt{x1}}\texttt{=}\overline{\texttt{x1}}\texttt{+}c_{1,-1}\texttt{*}\overline{\texttt{y1}}\texttt{;}$. In order to obtain correct adjoint values, however, $\overline{\texttt{x1}}$ needs to be reset to 0 prior to the third adjoint statement to reflect the implicit overwrite of `x1` via the assignment to `y2`.

In more general terms consider $\mathbf{z} = F(\mathbf{x}) \in I\!\!R^m$ such that $\mathbf{x} \in I\!\!R^n$, $\mathbf{y} = \phi_1(\mathbf{x}) \in I\!\!R^{n_y}$, and $\mathbf{z} = \phi_2(\mathbf{y})$. By the chain rule we get $F' = \phi_2'(\mathbf{y}) \cdot \phi_1'(\mathbf{x})$. Hence, $\bar{\mathbf{x}} = F'(\mathbf{x})^T \cdot \bar{\mathbf{z}}$, where $\bar{\mathbf{y}} = \phi_2'(\mathbf{y})^T \cdot \bar{\mathbf{z}}$ and $\bar{\mathbf{x}} = \phi_1'(\mathbf{x})^T \cdot \bar{\mathbf{y}}$.

Let $1 \le i, i' \le n$ and $0 \le j \le n_y$ such that $y_j = x_i$ share the same address. With $\phi_1'(\mathbf{x}) = A \equiv (a_{k,l})_{l=1,\ldots,n}^{k=1,\ldots,n_y}$ Equation (4) gives the incorrect

$$\begin{aligned} \bar{x}_i &= \bar{x}_i + a_{j,i}\bar{y}_j \\ \bar{x}_{i'} &= \bar{x}_{i'} + a_{j,i'}\bar{y}_j \end{aligned} \quad . \tag{5}$$

Unless we can prove that there is no aliasing (no sharing of addresses) between inputs and outputs of basic blocks, we need to accumulate the local adjoints in a compiler-generated temporary variable $\mathbf{t}$ followed by setting the adjoints of the outputs to zero and incrementing the adjoints of the inputs with the corresponding values in $\mathbf{t}$. A correct adjoint recurrence is the following.

$$\begin{aligned} \mathbf{t} &= A^T \cdot \bar{\mathbf{y}} \\ \bar{\mathbf{y}} &= \mathbf{0} \\ \bar{\mathbf{x}} &= \bar{\mathbf{x}} + \mathbf{t} \end{aligned} \tag{6}$$

Alternatively, aliasing can be resolved by the following adjoint recurrence.

$$\begin{aligned} \mathbf{t} &= \bar{\mathbf{y}} \\ \bar{\mathbf{y}} &= \mathbf{0} \\ \bar{\mathbf{x}} &= \bar{\mathbf{x}} + A^T \cdot \mathbf{t} \end{aligned} \tag{7}$$

The computational complexity of alias resolution is determined by the size of the vector $\mathbf{t}$ and the corresponding copy operation. In Equation (6) we have $\mathbf{t} \in I\!\!R^n$, whereas $\mathbf{t} \in I\!\!R^{n_y}$ in Equation (7).

Let $1 \le i \le n$ and $0 \le j \le m$ such that $z_j = x_i$ share the same address. With $A \equiv \phi_1'(\mathbf{x})$ defined as above and $\phi_2'(\mathbf{y}) = B \equiv (b_{k,l})_{l=1,\ldots,n_y}^{k=1,\ldots,m}$, naively Equation (4) yields

$$\begin{aligned} \bar{y}_k &= \bar{y}_k + b_{j,k}\bar{z}_j \\ &\vdots \\ \bar{x}_i &= \bar{x}_i + b_{k,i}\bar{y}_k \end{aligned} \tag{8}$$

for $k = 1, \ldots, n_y$. With z_j and x_i being mutual aliases, the increment of $\bar{x}_i$ implies the increment of $\bar{z}_j$. This value is obviously not what we want, since prior to the first increment we expect the adjoint of x_i to be zero. Hence the value of $\bar{z}_j$

needs to be set to zero after its use for the increment of the adjoints of all y_k, $k = 1, \ldots, n_y$. The correct adjoint recurrence is the following.

$$\begin{aligned} \bar{\mathbf{y}} &= \bar{\mathbf{y}} + B^T \cdot \bar{\mathbf{z}} \\ \mathbf{t} &= \bar{\mathbf{y}} \\ \bar{\mathbf{z}} &= \mathbf{0} \\ \bar{\mathbf{x}} &= \bar{\mathbf{x}} + A^T \cdot \bar{\mathbf{y}} \\ \bar{\mathbf{y}} &= \mathbf{0} \end{aligned} \tag{9}$$

The last statement results from the recursive application of the above argument when considering the given code fragment in a larger context.

In Equation (9) we assume that aliasing exists neither between $\mathbf{y}$ and $\mathbf{z}$ nor between $\mathbf{x}$ and $\mathbf{y}$. The most general conservative adjoint recurrence is the following combination of Equation (6) and Equation (9).

$$\begin{aligned} \mathbf{t}_1 &= B^T \cdot \bar{\mathbf{z}} \\ \bar{\mathbf{z}} &= \mathbf{0} \\ \bar{\mathbf{y}} &= \bar{\mathbf{y}} + \mathbf{t}_1 \\ \mathbf{t}_2 &= A^T \cdot \bar{\mathbf{y}} \\ \bar{\mathbf{y}} &= \mathbf{0} \\ \bar{\mathbf{x}} &= \bar{\mathbf{x}} + \mathbf{t}_2 \end{aligned} \tag{10}$$

2.2 Reversal of the Control Flow

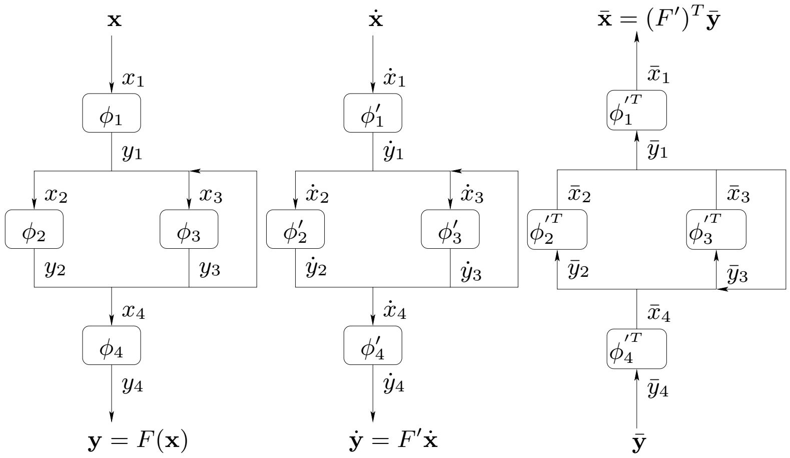
The flow of control within subroutines is reversed by memorizing branches and counting loop iterations while executing the augmented forward section of the adjoint code as described in [33]. All values are pushed onto a stack. The reverse section restores them (using pop) and, hence, executes the products of the transposed local Jacobians with the adjoints of the outputs of the respective basic block in reverse order. This method is illustrated in Figure 2. It also shows how the same local Jacobians are used in the tangent-linear model.

2.3 Reversal of the Call Tree

The interprocedural flow of control can be viewed as a call tree where each node represents a particular call made at run time; see Figure 3(b) for the example code shown in Figure 3(a). The call tree can be reversed by a combination of *split* and *joint* reversal modes [16]. For each subroutine in the original program, OpenAD/F generates distinct versions of the subroutine body that can be used to accomplish the following.

- Evaluate the original subroutine ($\rightarrow$ on top of $\square$);
- Store all inputs (argument checkpoint, $\downarrow$ on the left of $\square$);
- Restore all inputs (argument checkpoint, $\uparrow$ on the left of $\square$);
- Execute the forward section of the adjoint code ($\rightarrow$ on top and bottom of $\square$);
- Execute the reverse section of the adjoint code ($\leftarrow$ on top of $\square$).

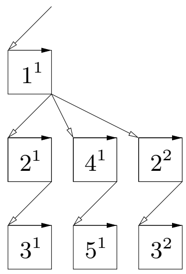
The two fundamental reversal modes are illustrated in Figure 3. In split mode a single run of the augmented forward section is performed followed by executing



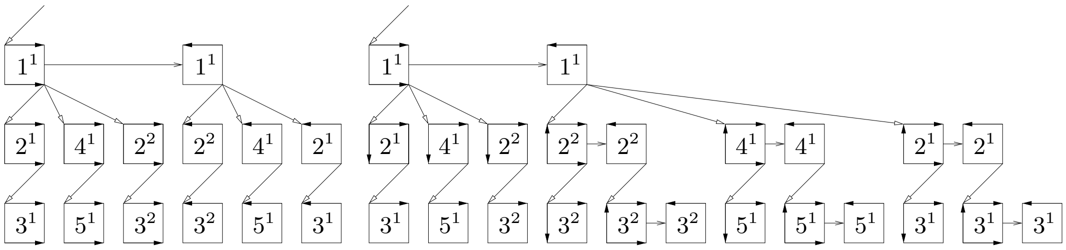
(a) (b) (c)

Fig. 2. Control flow graphs: original code (a), tangent linear model (b), adjoint model (c)

```
subroutine 1
  call 2; call 4; call 2
end subroutine 1
subroutine 2
  call 3
end subroutine 2
subroutine 4
  call 5
end subroutine 4
```



(a) (b)



(c) (d)

Fig. 3. Call tree reversal: code (a), call tree (b), adjoint call tree in split (c) and joint (d) reversal

the reverse section. All local Jacobians of the entire forward execution need to be stored. As a result, memory requirements may quickly become prohibitive for large-scale problems such as the MIT General Circulation Model used in high-resolution setups. Joint reversal mode represents a tradeoff between memory consumption and the number of arithmetic operations performed. It computes the adjoints of a callee only when they are required by the adjoint of the caller. Joint reversal requires a recomputation from stored checkpoints. Subroutine-level granularity used here together with side-effect analysis (see Section 3.4) allows for an easy, automatic checkpoint code generation. Split and joint reversal modes can be combined to achieve a feasible balance between memory requirement and operations count. Refer to [16] for further details on these reversal modes.

3 OPENAD/F

In this section we briefly describe the main components of OpenAD/F. References are provided for further details.

3.1 xaif

An XML-based (`www.w3c.org/XML`) hierarchy of directed graphs, referred to as xaif [21], is used for the internal representation of the numerical core of the implementation of a given vector function. This format is well suited to represent the results of semantic transformations including preaccumulation [6, 9, 17] and program reversal [15, 35] at various levels (call graph, control flow graphs, basic blocks, expressions). The main idea behind xaif is to provide a language-independent exchange format that separates language-specific from transformation-related algorithmic issues. Potentially, front-ends for various programming languages can utilize xaif, for example, Open64 and EDG/SAGE 3, as pointed out before.

3.2 xaifBooster

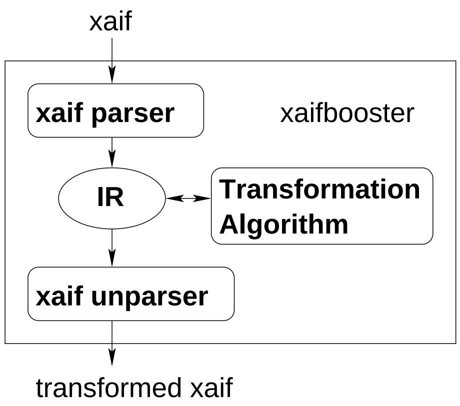


Fig. 4. xaifBooster parses xaif code into an internal representation (IR). It provides an API for transformation algorithms to modify the IR. An unparser returns the transformed xaif code.

The C++ software xaifBooster is a collection of utilities and routines for the (semantic) transformation of programs given in xaif. Its architecture is illustrated in Figure 4. One of the major concerns during the development of xaifBooster has been the clean separation of the internal representation (an enhanced object image of xaif) from algorithms that operate on this data structure. This goal has been achieved by applying the design patterns [12] factory, visitor, and decorator as described in [34]. The result is an API that gives AD developers the opportunity to implement new algorithms in a source transformation environment without having to implement full compiler front- and back-ends. Building on this API, we have implemented tangent-linear and adjoint algorithms that use statement- and basic-block-level preaccumulation of local gradients or Jacobians as described in Section 2. Near-optimal face elimination [27] sequences are computed by the software tool ANGEL [3, 28] (`angellib.sourceforge.net`) and transformed into Jacobian code by an xaifBooster algorithm. A simple adjoint version of the code is obtained by taping the local Jacobians and by computing the corresponding "transposed Jacobian-vector" products during an interpretive reverse sweep through the tape. This approach is essentially equivalent to split program reversal [16] and allows for an easy coupling of tangent-linear and adjoint versions of small to medium-sized codes as described in [29]. Our main adjoint code generation strategy uses combinations of split and joint reversal in a hierarchical fashion as described in Section 2.

3.3 OpenADFortTk

The main target application of the ACTS project is the MIT General Circulation Model (MITgcm) [24, 25]. It is implemented mostly in Fortran 77 to permit maintenance of an efficient and correct adjoint as the code evolves [19]. Future development will increasingly add Fortran features. We use the Fortran front-end that is part of the Open64 compiler suite maintained by the Center for High Performance Software Research at Rice University (`hipersoft.cs.rice.edu/Open64/`). It parses Fortran codes into an intermediate representation called whirl and provides an unparser from whirl back to Fortran. The focus of the ACTS project is on the transformation of whirl into xaif (OpenADFortTk's component whirl2xaif) and, conversely, on the back-translation of differentiated xaif code into whirl (xaif2whirl). OpenADFortTk uses the call graph and control flow graph builders provided by OpenAnalysis.

3.4 OpenAnalysis

The OpenAnalysis toolkit[1] separates program analysis from language-specific or front-end-specific intermediate representations. This separation enables a single implementation of domain-specific analyses such as activity analysis, to-be-recorded analysis, and linearity analysis in OpenAD. Also available via OpenADFortTk are standard analyses implemented within OpenAnalysis, such as CFG construction, call graph construction, alias analysis, reaching definitions, ud- and du-chains, and side-effect analysis.

OpenADFortTk interfaces with OpenAnalysis as a producer and a consumer. A description of alias analysis illustrates this interaction. Xaif requires an alias

[1] see `http://www.mcs.anl.gov/OpenAnalysisWiki/moin.cgi`

map data structure, in which each variable reference is mapped to a set of virtual locations that it may or must reference. For example, if a global variable `g` is passed into subroutine `foo` through the dummy parameter `p`, variables `g` and `p` will reference the same memory address within `foo` and therefore be aliased to each other. OpenAnalysis determines the aliasing relationships by calling methods of an abstract *alias IR interface*. This is a language-independent interface between OpenAnalysis and any intermediate representation for an imperative programming language. OpenADFortTk implements the alias IR interface for the Fortran intermediate representation given in whirl. The interface includes iterators over all the procedures, statements in those procedures, memory references in each statement, and memory reference expression and location abstractions that provide further information about memory references and symbols. The results of the alias analysis are returned to OpenADFortTk through a special alias results interface.

OpenAnalysis also performs activity analysis. For activity analysis the independent and dependent variables of interest are communicated to the front-end through the use of pragmas. The results of the analysis are then encoded by the Fortran front-end into xaif. The analysis indicates which variables are *active*(i.e., have nonzero derivatives) at any time, which memory references are active, and which statements are active (activity determined by the variable on the left-hand side).

The activity analysis itself is based on the formulation in [18]. The main difference is that the data-flow framework in OpenAnalysis does not yet take advantage of the structured data-flow equations. Activity analysis is implemented in a context-insensitive, flow-sensitive interprocedural fashion.

4 CASE STUDIES

Below we discuss briefly four applications for which OpenAD/F has successfully generated adjoint code. Details including the generated codes and driver routines can be found on the OpenAD website.

First we illustrate the transformation procedure and the resulting code with a simple toy problem. Consider the following Fortran code that implements a univariate scalar function $y = f(x)$.

```
          subroutine head(x,y)
            double precision,intent(in) :: x
            double precision,intent(out) :: y
c$openad INDEPENDENT(x)
            y=sin(x*x)
c$openad DEPENDENT(y)
          end subroutine
```

Two pragmas are used to indicate to the system what the independent (x) and the dependent (y) variables are. Our aim is to generate an adjoint subroutine to compute $\bar{x} = (f')^T \cdot \bar{y}$. Obviously, $(f')^T = f'$ as $f' \in I\!R$ for this simple example.

Augmented Forward Section All *active* program variables are redeclared to be of type `active` as introduced in Section 2.1. The augmented forward section of the adjoint code evaluates the function (lines 1 and 2 in the listing below) and the local partial derivatives labeling the edges in the linearized computational graph

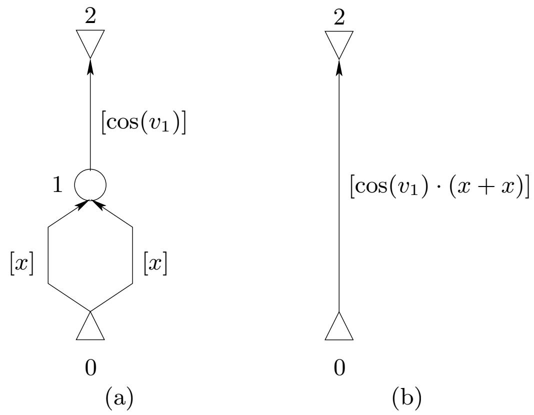


Fig. 5. Preaccumulation for toy problem

as shown in Figure 5 (a) (lines 3–5). The local Jacobian (a scalar partial derivative in this case) is accumulated by transformation of the linearized computational graph into bipartite form as shown in Figure 5 (b) (line 6). The preaccumulated local Jacobian is then stored for later use by the reverse section of the adjoint code.

```
1         OpenAD_Symbol_0 = (X%v*X%v)
2         Y%v = SIN(OpenAD_Symbol_0)
3         OpenAD_Symbol_2 = X%v
4         OpenAD_Symbol_3 = X%v
5         OpenAD_Symbol_1 = COS(OpenAD_Symbol_0)
6         OpenAD_Symbol_5 = ((OpenAD_Symbol_3 + OpenAD_Symbol_2) *
              OpenAD_Symbol_1)
7         double_tape(double_tape_pointer) = OpenAD_Symbol_5
8         double_tape_pointer = double_tape_pointer+1
```

Reverse Section The reverse section of the adjoint code simply restores the previously preaccumulated local Jacobian (lines 1 and 2) followed by the computation of the product of the transposed local Jacobian with the vector of adjoints of the outputs of the current basic block (line 3). In our simple case the above is reduced to the scalar product $\frac{\partial y}{\partial x} \cdot \bar{y} = \bar{x}$. The adjoints of the outputs are set to zero for potential further use in the incremental adjoint code, as outlined in Section 2 (line 4).

```
1         double_tape_pointer = double_tape_pointer-1
2         OpenAD_Symbol_7 = double_tape(double_tape_pointer)
3         X%d = X%d+Y%d*OpenAD_Symbol_7
4         Y%d = 0.0d0
```

Through interprocedural alias analysis one can determine if X and Y are aliased. Here this is not the case and therefore OpenAD/F uses no temporary variable in the adjoint accumulation (refer to Section 2.1).

Driver OpenAD/F inserts the automatically generated derivative code into a source code template as described in detail in [30]. For example, in split reversal the following template is used.

```
subroutine template()
 use OpenAD_tape
```

```
3    use OpenAD_rev
4  !$TEMPLATE_PRAGMA_DECLARATIONS
5    if (rev_mode%tape) then
6  !$PLACEHOLDER_PRAGMA$ id=2
7    end if
8    if (rev_mode%adjoint) then
9  !$PLACEHOLDER_PRAGMA$ id=3
10   end if
11 end subroutine template
```

The augmented forward and reverse sections of the adjoint code replace the pragmas in lines 6 and 9, respectively. Additional declarations replace the pragma in line 4. Depending on the values of the Boolean flags `rev_mode%tape` and `rev_mode%adjoint` the augmented forward and reverse sections are executed, respectively. The template for joint reversal is slightly more complicated. It can be found on the OpenAD website. The various reversal modes are represented by the derived type `our_rev_mode` that is defined in the module `OpenAD_rev`. The runtime support module `OpenAD_active` contains the definition of the active data type `active`.

The following driver program calls the adjoint version of `head` to compute the derivative of its output `y` with respect to its input `x`.

```
1  program driver
2    use OpenAD_active
3    use OpenAD_rev
4    external head
5    type(active):: x, y
6    read *, x%v
7    y%d=1.0
8    our_rev_mode%tape=.TRUE.
9    our_rev_mode%adjoint=.TRUE.
10   call head(x,y)
11   write (*,*) "J(1,1)=",x%d
12 end program driver
```

Both `x` and `y` are declared as active, and the value of `x`, that is, `x%v`, is read in. The adjoint of the output `y` is set to one. In lines 8 and 9 we indicate that the adjoint version of head is supposed to execute the augmented forward section followed by the reverse section. The desired partial derivative is stored in `x%d`.

4.1 Flow in a Driven Cavity

The driven cavity problem was taken from the MINPACK-2 test problem collection [4]. The 2D (x-y-plane) flow in a driven cavity is formulated as a boundary value problem, which is discretized by standard finite difference approximations to obtain a system of nonlinear equations. We choose equal numbers of steps in both the x- and y-directions. The number of independent variables is equal to the number of mesh points that are not part of the boundary.

The whole Jacobian matrix was accumulated in both tangent-linear and adjoint mode. The results were verified against each other as well as finite difference approximation and the hand-written Jacobian supplied as part of the MINPACK-2 test problem collection.

4.2 Box Model for Thermohaline Circulation

In physical oceanography, researchers are expending substantial effort in understanding the ocean circulation's role in the variability of the climate system, on time scales of decades to millennia and beyond. Of especial interest is the so-called thermohaline circulation — the contribution to the ocean circulation that is driven by density gradients and thus controlled by temperature and salinity properties and its associated fluxes. It plays a crucial role in connecting the surface to the deep ocean through deep-water formation, which occurs at some isolated convection sites at high latitudes mainly in the subpolar Atlantic ocean, such as the Labrador Sea and the Greenland-Irminger-Norwegian Seas. The box model is used to solve the (generalized) eigenvalue problem arising in the study of thermohaline circulation. Further details on the problem itself and ideas on how to couple tangent-linear and adjoint evaluations in one and the same computation can be found in [29].

4.3 Shallow Water Model

The third test case is a shallow water model used in an earlier study by Losch and Wunsch [23] on bottom topography as a control variable in ocean models. This code was originally written in Fortran 77 and is differentiable with the commercial adjoint model compiler for Fortran codes TAF [14]. There have been a few Fortran extensions to the source and it contains some of the basic language features also used in the MITgcm (see Section 4.4).

In Figure 6 we show as an example output of a map of sensitivities of zonal volume transport through the Drake Passage to changes in bottom topography everywhere in a barotropic ocean model computed by P. Heimbach. An adjoint model generated by OpenAD/F was used for the gradient calculation. Both the split and joint modes exhibit rather large storage requirements. Improvements result from two-level nested checkpointing as described in [30]. Moreover, a special reversal technique was used for *explicit* loops as outlined in [33].

4.4 MITgcm

Adjoint modeling (i.e., the reverse mode of AD) plays an ever increasing role in ocean and climate modeling, and with it the use of AD tools to generate efficient, exact, and up-to-date adjoint code of the underlying parent models. One of the most prominent examples in which AD plays a crucial role comes from the ECCO (Estimating the Circulation and Climate of the Ocean) Consortium.[2] It treats ocean state estimation as an optimal control problem (see, e.g., [36]). Variously called the method of Lagrange multipliers, the adjoint method, 4D-Var, and the Pontryagin principle, it ultimately reduces to a form of least squares in which a fully fledged ocean general circulation model, here the MITgcm [25, 24], is fitted, through the adjustment of initial conditions and time-dependent air-sea momentum and buoyancy fluxes, to a variety of observations ranging from global satellite altimetry available continuously since 1992 (TOPEX/Poseidon, ERS-1/2, Jason-1, ENVISAT), to in-site measurements from the World Ocean Circulation Experiment, and recently the ARGO programme (see [37] for

[2] `http://www.ecco-group.org`

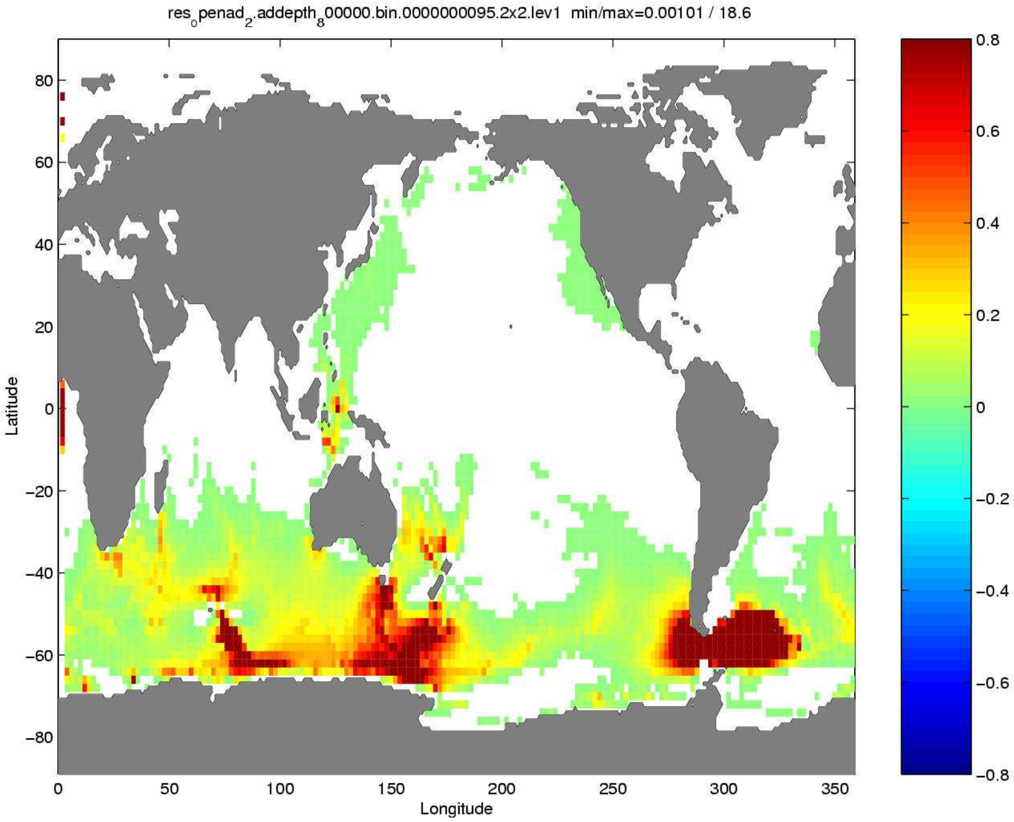


Fig. 6. Sensitivities of zonal volume transport through the Drake Passage to changes in bottom topography computed by an adjoint model that was generated with OpenAD/F

an overview and references). The feasibility of this approach was demonstrated by [31, 32] with their landmark publication of the first multiyear, quasi-global, dynamically consistent model vs. data synthesis. Dynamic consistency is crucial when attempting to compute and interpret property budgets and trends such as decadal changes in heat and mass transports transport or global sea-level rise (e.g., [38]). The adjoint of the MITgcm was first generated (semi-)automatically by using the software tool TAF [13, 20]. With a 10^8-dimensional control space, the ECCO problem constitutes, to our knowledge, the biggest gradient-based optimization problem undertaken so far.

The MITgcm, when run using height as the vertical coordinate, solves the Boussinesq form of the Navier-Stokes equations for an incompressible fluid, hydrostatic or fully non-hydrostatic, in a curvilinear framework (the model can also be run in pressure vertical coordinates, in which case it is fully mass-conserving and the Boussinesq-approximation can be relaxed). The horizontal assembly of the finite volume grid cells is based on a domain decomposition to enable efficient parallelization across a variety of high-performance computer architectures. The model is endowed with state-of-the art physical parameterization schemes for subgrid-scale horizontal and vertical mixing of momentum and tracer properties, as well as a sophisticated dynamic/thermodynamic sea-ice model, plus an atmospheric boundary layer scheme over the open ocean. A suite of higher-order advection schemes and time-stepping algorithms enables flexible configuration for a large variety of applications under the tight constraint to meet the criteria for numerical stability. It is currently being used for high-resolution global-scale ocean simulations [26].

The model is continuously undergoing vigorous development to incorporate novel physical and numerical schemes and approaches for treating the horizontal and vertical grid (e.g., [1, 2, 8]). In this environment the ability to (re-)generate derivative code such as the adjoint by means of AD becomes a crucial element to ensure that novel aspects and features of the parent model can be carried over to the adjoint model. Altogether, the model consists of roughly 40k lines of mostly Fortran77 code.

In a first OpenAD/F milestone, the model is configured in a single-layer setup to closely mimic the shallow water setup of the previous section over the exact same quasi-global $2^\circ \times 2^\circ$ domain. The full momentum and tracer advection code is thus subject to differentiation. The semi-implicit treatment of the surface involves solving an elliptic problem using a two-dimensional preconditioned conjugate gradient (CG) algorithm. Taking advantage of the self-adjointness of the elliptic operator enables us to exclude this iterative solver from the differentiation procedure and instead rely on the original code. The only code not currently differentiated pertains to baroclinic elements of the simulation, such as the equation of state and the parameterization schemes for subgrid-scale mixing. Nevertheless, these codes do not contain any aspects that would differ structurally from the code now handled by OpenAD/F, and no obstacles are expected when expanding to a fully baroclinic configuration.

5 CONCLUSIONS

We have successfully generated and tested adjoint code of a quasi-global configuration of the MIT general circulation model by means of OpenAD/F for a roughly 10^5-dimensional control problem. This work clearly demonstrates the ability of the tool to treat complex codes such as those used in earth and climate sciences. The main work now will consist in improving the flow dependency analysis (OpenAnalysis) to achieve adjoint code that is efficient enough to be applied in an operational setting.

ACKNOWLEDGMENTS

Funding for the ACTS project was initially provided by NSF under ITR contract OCE-0205590 for a three-year period that ended in August 2005. The project is currently supported in part by NASA, NOAA, and NSF through the National Ocean Partnership Program (NOPP), with additional support through NSF's CMG program. Thanks are due to our many MITgcm, ECCO, and ECCO-GODAE partners.

While working at Argonne National Laboratory, Strout and Naumann were supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

1. A. Adcroft, J.-M. Campin, C. Hill, and J. Marshall. Implementation of an atmosphere-ocean general circulation model on the expanded spherical cube. *Mon. Wea. Rev.*, 132(12):2845–2863, 2004.
2. A. Adcroft and J.M. Campin. Rescaled height coordinates for accurate representation of free-surface flows in ocean circulation models. *Ocean Modelling*, 7(3-4):269–284, 2004.
3. A. Albrecht, P. Gottschling, and U. Naumann. Markowitz-type heuristics for computing Jacobian matrices efficiently. In *Computational Science – ICCS 2003*, volume 2658 of *LNCS*, pages 575–584. Springer, 2003.
4. B. Averik, R. Carter, and J. Moré. The MINPACK-2 test problem collection (preliminary version). Technical Memorandum ANL/MCS-TM-150, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.
5. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, Philadelphia, 1996. SIAM.
6. C. Bischof and M. Haghighat. Hierarchical approaches to automatic differentiation. In *[5]*, pages 82–94, 1996.
7. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, volume 50 of *LNCSE*, Berlin, 2005. Springer.
8. J.M. Campin, A. Adcroft, C. Hill, and J. Marshall. Conservation of properties in a free surface model. *Ocean Modelling*, 6:221–244, 2004.
9. B. Christianson, L. Dixon, and S. Brown. Sharing storage using dirty vectors. In *[5]*, pages 107–115. SIAM, Philadelphia, 1996.
10. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.
11. G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, 1991. SIAM.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

13. R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24:437–474, 1998.
14. R. Giering and T. Kaminski. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. In *Proceedings of GAMM 2002*, 2002.
15. A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992.
16. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.
17. A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markovitz rule. In *[11]*, pages 126–135, 1991.
18. L. Hascoët, U. Naumann, and V. Pascual. TBR analysis in reverse-mode automatic differentiation. Preprint MCS-P936-0202, Argonne National Laboratory, 2002. Under Review for Elsevier Science FGCS.
19. P. Heimbach, C. Hill, and R. Giering. Automatic generation of efficient adjoint code for a parallel Navier-Stokes solver. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 1019–1028, Berlin, 2002. Springer.
20. P. Heimbach, C. Hill, and R. Giering. An efficient exact adjoint of the parallel mit general circulation model, generated via automatic differentiation. *Future Generation Computer Systems*, 21(8):1356–1371, 2005.
21. P. Hovland, U. Naumann, and B. Norris. An XML-based platform for semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications*, pages 530–538. ACTA Press, 2002.
22. P. Hovland and B. Norris. Users' guide to ADIC 1.1. Technical Memorandum ANL/MCS-TM-225, Mathematical and Computer Science Division, Argonne National Laboratory, 2001.
23. M. Losch and C. Wunsch. Bottom topography as a control variable in an ocean model. *Journal of Atmospheric and Oceanic Technology*, 20(11):1685–1696, 2003.
24. J. Marshall, A. Adcroft, C. Hill, L. Perelman, and C. Heisey. Hydrostatic, quasi-hydrostatic and nonhydrostatic ocean modeling. *J. Geophysical Research*, 102, C3:5,753–5,766, 1997b.
25. J. Marshall, C. Hill, L. Perelman, and A. Adcroft. Hydrostatic, quasi-hydrostatic and nonhydrostatic ocean modeling. *J. Geophysical Research*, 102, C3:5,733–5,752, 1997.
26. D. Menemenlis, C. Hill, A. Adcroft, J.M. Campin, B. Cheng, B. Ciotti, I. Fukumori, A. Koehl, P. Heimbach, C. Henze, T. Lee, D. Stammer, J. Taft, and J. Zhang. Towards eddy permitting estimates of the global ocean and sea-ice circulations. *EOS Transactions AGU*, 86(9):89, 2005.
27. U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.*, 3(99):399–421, 2004.
28. U. Naumann and P. Gottschling. Simulated annealing for optimal pivot selection in Jacobian accumulation. In A. Albrecht and K. Steinhöfel, editors, *Stochastic Algorithms: Foundations and Applications*, volume 2827 of *LNCS*, pages 83–97. Springer, 2003.
29. U. Naumann and P. Heimbach. Coupling Tangent-Linear and Adjoint Models. In V. Kumar, M. Gavrilova, C. Tan, and P. L'Ecuyer, editors, *Computational Science and Its Applications – ICCSA 2003, Proceedings of the International Conference on Computational Science and its Applications, Montreal, Canada, May 18–21, 2003. Part II*, volume 2668 of *LNCS*, pages 95–104, Berlin, 2003. Springer.
30. U. Naumann and J. Utke. Source templates for the automatic generation of adjoint code through static call graph reversal. In V. Sunderam, G. van Albada, P. Sloot, and J. Dongarra, editors, *Computational Science - ICCS 2005, Proceedings of the International Conference on Computational Science, Atlanta, GA, USA, May 22-25, 2005, Part I*, volume 3514 of *LNCS*, pages 338–346, Berlin, 2005. Springer.
31. D. Stammer, C. Wunsch, R. Giering, C. Eckert, P. Heimbach, J. Marotzke, A. Adcroft, C.N. Hill, and J. Marshall. The global ocean circulation and transports during 1992 – 1997, estimated from ocean observations and a general circulation model. *J. Geophysical Research*, 107(C9):3118, 2002a.
32. D. Stammer, C. Wunsch, R. Giering, C. Eckert, P. Heimbach, J. Marotzke, A. Adcroft, C.N. Hill, and J. Marshall. Volume, heat and freshwater transports of the global ocean circulation 1993 – 2000, estimated from a general circulation model constrained by WOCE data. *J. Geophysical Research*, 108(C1):3007, 2003.

33. J. Utke, A. Lyons, and U. Naumann. Efficient reversal of the intraprocedural flow of control in adjoint computations. *J. Systems and Software*, 2006. To appear.
34. J. Utke and U. Naumann. Software technological issues in automatizing the semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications*, pages 417–422. ACTA Press, 2003.
35. A. Walther and A. Griewank. New results on program reversals. In *[10]*, pages 237–243. Springer, New York, 2001.
36. C. Wunsch. *The ocean circulation inverse problem.* Cambridge University Press, Cambridge (UK), 1996.
37. C. Wunsch and P. Heimbach. Practical global oceanic state estimation. *Physica D*, under revision, 2005.
38. C. Wunsch and P. Heimbach. Estimated decadal changes in the north atlantic meridional overturning circulation and heat flux. *J. Phys. Oceanogr.*, in press, 2006.