

# Activity Analysis: Algorithms and Effectiveness

Alan Carle

Mike Fagan

## 1 Introduction/Overview

In order to improve the performance of derivative code, many modern automatic differentiation tools use a source code analysis technique called *activity analysis*. Activity analysis seeks to discover which variables in the program actually contribute to the sensitivity calculation. Program variables that provably do not affect the sensitivity calculation need not:

- have derivative code generated for their part in assignment statements
- have space allocated for their derivative values.

In essence, activity analysis is dead code (and space) elimination for derivative computation.

To understand the remainder of this paper requires (informal) definitions of 'active' variable, as well as 'independent' variable and 'dependent' variable.

### Definition of 'independent', 'dependent', and 'active' variable

To begin, the notion of 'independent' and 'dependent' variable correspond closely to the mathematical notion. Mathematically speaking, given a functional definition  $(u, w) = f(x, y, z)$  where  $f : R^3 \rightarrow R^2$ , then  $x, y, z$  are the potential independent variables, and  $u, w$  are the potential dependent variables. The 'potential' qualifier indicates that for a particular problem, not all of the variables in question may be important. For example, sensitivity of  $w$  may not be of interest, and the sensitivity of  $u$  with respect to  $y$  may not be of interest. In this case, the dependent variable would be  $u$  alone, and the independent variable set would be  $x, z$ .

For computer codes, the notion of independent/dependent is similar. Given a prototypical computer code like:

```
!   x, y, z   inputs
!   u, w     outputs
      subroutine f(x, y, z, u, w)
```

the potential independent variables would be  $x, y, z$  and the potential dependent variables would be  $u, w$ . Given a sound grasp of independent and dependent variables, the definition of active variable goes as follows:

**Definition:** A variable  $V$  is *active* if there is a computation path from some independent variable to  $V$  *and* there is a computation path from  $V$  to some dependent variable.

Note that the set of active variables depends on the both the set of independent variables and the set of dependent variables.

## Organization of the rest of the paper

The remainder of this paper is organized as follows:

**section 2** outlines the Adifor 3.0 static activity analysis algorithm. The static algorithm approximates true activity information.

**section 3** outlines the Adifor 3.0 special purpose dynamic activity analysis method. The dynamic method determines the *exact* activity status of a variable at run time.

**section 4** compares the static approximation with the true activity status for some large codes we have done in the past.

## 2 Adifor static activity analysis

To determine the set of active variables, Adifor uses compiler analysis techniques to analyze a user's code. The analysis technique is both *conservative*, and *static*. Being conservative means that any variable that is not *provably* inactive will be treated as active. Being static means that no activity information is determined at run time.

In the abbreviated description of the analysis algorithm that follows, we assume that some classical compiler analyses are available as building blocks. Notably, we assume that we have available:

- A call graph, with edges of the call graph holding actual-to-formal parameter binding information.
- Abstract syntax tree
- Symbol table, holding various information determined from declarations

The general mechanism that Adifor uses for solving interprocedural problems uses the call graph as the driving data structure. Generically, the mechanism looks like:

```
For each node N in the call graph:
    determine parameter binding information
    use node N intra-procedural info with binding info to
    determine node N annotation.
    pass binding information (on the call graph edges) to
    the next iteration.
end
```

To (approximately) determine the active variables for a given program, Adifor uses the generic interprocedural propagation method to compute 2 bits of information for every variable  $V$  in the program:

**pathFromIVARS** is true if there is a potential computation path from any independent variable to  $V$

**pathFromDVARs** is true if there is a potential computation path from  $V$  to any dependent variable.

Once these 2 bits of information are computed, then the conjunction of the 2 bits by definition determines whether or not  $V$  is active.

The computation of activity, then depends on efficiently computing appropriate intra-procedural information. To simplify this intra-procedural computation, Adifor makes these simplifying assumptions:

- Arrays and common blocks are treated as scalars: if any component of an array/common block is referenced, then the whole array/common block is assumed to be referenced.
- Variable information is flow insensitive. That is, both sides of a branch are presumed to be taken.
- Only variables of type `real`, `double precision`, `complex` or `double complex` are relevant in sensitivity computations.

In addition, Adifor also takes advantage of Fortran 77's lack of recursion, so the call graph is a DAG.

The intraprocedural information that Adifor computes is (floating point) variable dependence information. This dependence information is calculated in 2 stages. First, Adifor computes a *simple* dependence relation using this algorithm:

```
for each assignment statement A in the procedure:
  D = lhs(A)
  for each V in rhs(A):
    dependence_matrix(D,V) = 1
  end
end
```

. The 2nd step of the dependence calculation is to compute the transitive closure of the simple dependence relation. Given this (closed) dependence matrix, both path relations of the activity definition can be propagated. If we know from parameter binding information that there is a path from  $D$  to some dependent variable, then for every  $V$  on the row `dependence_matrix(D,V)`, there is a path from  $V$  to some dependent variable. Similarly, if we know that there is path from some independent variable to  $V$ , then for every  $D$  in the column `dependence_matrix(D,V)`, there is a path from an independent variable to  $D$ .

In order to make calling interfaces uniform, Adifor makes 1 more interprocedural pass to produce the 'forced' active variables. [ Note: this abstract does not describe 'forced'. The paper will explain this concept ].

### 3 Adifor run-time activity analysis

To compute the exact activity status of any variable at run time requires both forward and reverse propagation of information. Fortunately, the Adifor 3.0 differentiation engine is flexible enough to permit the generation of special accumulation code in the forward and reverse pass. In the forward pass, this code is inserted:

```
lhs = OP(rhs1,rhs2,...)
! or ivar path bits from a_rhs1,... into a_lhs
ivar_path_accum(a_lhs,a_rhs1,a_rhs2,...)
```

Similarly, in the reverse pass, this code is inserted:

```
! lhs = OP(rhs1,rhs2,...) the forward assignment statement
! or dvar path bits from a_lhs into a_rhs1,...
dvar_path_accum(a_lhs,a_rhs1,a_rhs2,...)
```

### 4 Comparison of run-time versus static activity analysis

The surprising fact appears to be that for most complicated codes in design contexts, almost all floating point variables are truly active. The static analysis and the dynamic analysis match closely.

The results of the dynamic activity status of variables with the static activity analysis are shown below:

Code	Static	Dynamic
Modtran	97%	96.9%
Spurc	99%	98.7%
STAGS	100%	100.0%
CFL3d	99%	99.0%
Overflow	100%	100.0%
USSAERO	99%	99.0%