Term Graphs for Imperative Programming Languages

Paul Hovland [1] Boyana Norris [2]

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S Cass Ave, Argonne, IL 60439, USA

Jean Utke [3]

Computer Science
University of Chicago
Chicago, IL, USA

Michelle Mills Strout [4]

Computer Science
Colorado State University
Fort Collins, CO, USA

Abstract

Automatic differentiation is a technique for the rule-based transformation of a subprogram that computes some mathematical function into a subprogram that computes the derivatives of that function. Automatic differentiation algorithms are typically expressed as operating on a weighted term graph called a linearized computational graph. Constructing this weighted term graph for imperative programming languages such as C/C++ and Fortran introduces several challenges. Alias and definition-use information is needed to construct term graphs for individual statements and then combine them into one graph for a collection of statements. Furthermore, the resulting weighted term graph must be represented in a language-independent fashion to enable the use of AD algorithms in tools for various languages. We describe the construction and representation of weighted term graphs for C/C++ and Fortran, as implemented in the ADIC 2.0 and OpenAD/F tools for automatic differentiation.

Key words: automatic differentiation, computational graph, term graph

[1] Email: hovland@mcs.anl.gov
[2] Email: norris@mcs.anl.gov
[3] Email: utke@mcs.anl.gov
[4] Email: mstrout@cs.colostate.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

```
a = cos(x);         // statement 1
b = sin(y)*y*y;     // statement 2
f = exp(a*b);       // statement 3
```

Fig. 1. Pseudocode for a simple example.

1 Introduction

Automatic differentiation is a technique for the rule-based transformation of a subprogram that computes some mathematical function into a subprogram that computes the derivatives of that function [3,4]. Derivatives have a variety of uses in scientific computing, including the solution of nonlinear partial differential equations, function minimization, parameter identification, data assimilation, sensitivity analysis, and uncertainty quantification. Automatic differentiation algorithms typically operate on a directed acyclic graph referred to as a *computational graph* or, after edge weights corresponding to partial derivatives have been added, a *linearized computational graph*.

The computation graph represents each value in a computation as a vertex and represents value dependences between values as directed edges. Formally, the computation graph is a graph $G(V, E)$, where V is the set of vertices and E is the set of edges. Each vertex represents a value $val(v)$ and is either labeled with the variable that contains the value $store(v)$ and/or the operation that generates the value $op(v)$. If a vertex is labeled with an operation, then the vertex has incoming edges indicating the values that are inputs to the operation. For example, Fig. 2 shows the linearized computational graph for the simple function defined by the program segment in Fig. 1. Note that the computation in the first statement is modeled with the vertices labeled `x` and `cos` with a dependence edge from the value stored in `x` to the operation `cos`.

The linearized computational graph can be interpreted as a *weighted, acyclic term graph*. Construction of the linearized computational graph starts from the computational graph and then associates partial derivatives with the dependence edges. Formally, each edge e is associated with a partial derivative $p(e)$, where $e = (u, v)$ and $p(e) = \partial val(v)/\partial val(u)$. For example in Figure 2, the partial derivative of `cos(x)` with respect to `x` is `-sin(x)`. Given this linearized computational graph, the derivative of a dependent variable v_j with respect to an independent variable v_i is the sum over all paths from v_i to v_j of the product of the edge weights along that path [1].

To generate derivative code, the linearized computational graph is transformed via a sequence of vertex or edge eliminations into a bipartite graph whose edge weights correspond to these derivatives. Each elimination leads to the generation of code. A vertex is eliminated by multiplying the weight of each input edge by that of each output edge and adding the product to the edge whose source is that of the input edge and whose sink is that of the output edge, creating new edges from predecessor vertices to successor vertices where necessary. More formally, a vertex v_0 is eliminated using the rule: $\forall v_i \in \text{Pred}(v_0), v_j \in \text{Succ}(v_0)$, if $e_{ij} \in E$

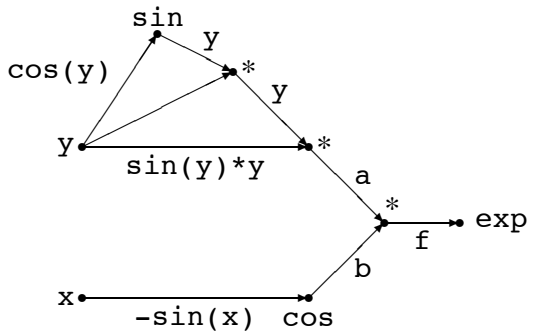

Fig. 2. Linearized computational graph for the simple example.

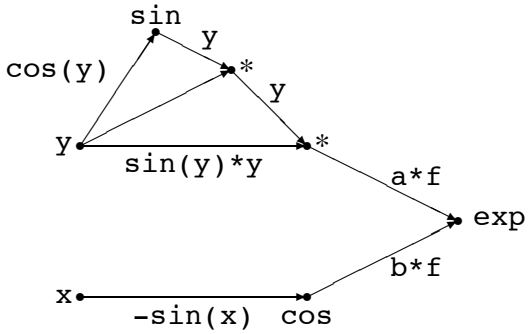

Fig. 3. Linearized computational graph of Fig. 2, after elimination of vertex a*b.

then $e_{ij} = e_{ij} + e_{i0}e_{0j}$ else $E = E \cup e_{ij}$ and $e_{ij} = e_{i0}e_{0j}$, where E is the set of all edges, e_{ij} denotes the edge from vertex v_i to vertex v_j, Pred(v_i) denotes the set of all predecessors to vertex v_i, and Succ(v_i) denotes the set of all successors to vertex v_i. Figure 3 shows the computational graph after eliminating the vertex corresponding to `a*b`. The associativity of the chain rule of differential calculus implies that vertices may be eliminated in any order. Because of fill-in, the elimination order impacts the computational cost. Finding an order that minimizes the number of multiplications is conjectured to be NP-hard. Many heuristics are used, however, including topological order (called the forward mode), reverse topological order (called the reverse mode), minimum Markowitz degree [5] [6], and relative Markowitz degree [4,8]. The number of multiplications can be further reduced by eliminating individual edges [8] or pairs of edges [9], rather than entire vertices, but such techniques are beyond the scope of this paper.

The rest of this paper is organized as follows. The next section sketches the construction of weighted term graphs for imperative programming languages. Section 3 describes an XML representation for term graphs. Section 4 describes some of the static analyses used in automatic differentiation tools. Section 5 discusses a procedure for merging the term graphs for individual statements into larger term graphs. We conclude with some thoughts about how ideas from term graphs could benefit the automatic differentiation community (and vice versa).

[5] The Markowitz degree is the product of the in degree and the out degree.

operation	cos	sin	*	*	*	exp
input value 0	1.00	2.00	0.91	1.82	0.54	1.97
input value 1	–	–	2.00	2.00	3.64	–
output value	0.54	0.91	1.82	3.64	1.97	7.17
input address 0	0	1	5	6	2	5
input address 1	–	–	1	1	3	–
output address	2	5	6	3	5	4

Fig. 4. A possible tape for the simple example of Fig. 1.

2 Weighted Term Graphs for Imperative Programming Languages

Automatic differentiation is used primarily in the domain of scientific computing, where the vast majority of programs are implemented in an imperative programming language such as C/C++ or Fortran. Because automatic differentiation algorithms operate on linearized computational graphs, mechanisms are needed for the construction of these weighted term graphs from programs written in imperative languages. We briefly describe two strategies, one suitable for the runtime construction of a weighted term graph and one suitable for compile-time construction, which requires static analysis and transformation of source code. The remainder of this paper discusses the second strategy in greater detail.

2.1 Runtime Construction of Weighted Term Graphs

In programming languages that support operator overloading, including C++, one can construct a term graph for a particular execution history by overloading the operators and intrinsic functions to record ("tape") the operation type as well as the values and addresses of the input and output arguments. When subprogram execution completes, the computation graph is constructed from this "tape," using each definition of an address as a vertex in the computational graph. The vertex identifier for any input address can be obtained by searching for the latest definition of that address in the tape. Given the operation type and input values, partial derivatives can be computed and assigned as edge weights to create a linearized computational graph. Figure 4 shows one possible tape for our simple example, using input values `x=1.0` and `y=2.0`. Figure 5 shows the weighted term graph constructed from this tape. Because the graph structure is not known until runtime, the vertex elimination order must be determined online, necessitating the use of simple, linear time heuristics. The ADOL-C automatic differentiation tool [5] uses runtime taping to construct weighted term graphs for C++ programs.

2.2 Static Construction of Weighted Term Graphs

One can also construct the term graph using static analysis of the source code. In this case, the edge weights are not known until runtime, but the structure of the term

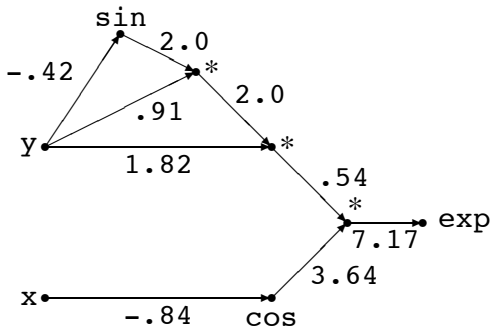

Fig. 5. Linearized computational graph constructed from the tape of Fig. 4.

graph is known at compile time. Since the analysis is performed offline, polynomial time algorithms can be used to select a vertex elimination order. Global search algorithms such as simulated annealing are also tractable [10]. Static source transformation offers several other opportunities for efficiency improvements in automatic differentiation. However, it also presents many challenges: a robust compiler infrastructure is needed for parsing and unparsing; many types of static analysis are required; and a mechanism for handling control structures such as loops and branches is essential.

For programs with complex control flow, static construction of the complete and correct term graph is not possible. Instead, one typically constructs a term graph for each basic block (a sequence of statements with no intervening control constructs) and applies arbitrary vertex elimination strategies only within individual basic blocks. The derivatives of basic blocks are combined using either the forward mode or reverse mode. The forward mode requires no additional runtime information, since the derivative computation follows the same control flow as the original function evaluation. The reverse mode must reverse the control flow. Therefore, at runtime a record of control flow decisions (such as basic block identifiers or branch conditions and loop bounds) must be stored.

Even constructing a single term graph for each basic block may not be possible. Consider the simple example of Fig. 1. If `a` is aliased to `b`, then the computational graph is the term graph shown in Fig. 6. If one cannot statically determine whether a and b are aliased, separate term graphs are needed for each statement, as depicted in Fig. 7. In practice, separate term graphs are used by default and these separate graphs are merged into larger term graphs only when static analysis guarantees correctness. This process, called flattening, is described in more detail in Section 5.

3 An XML Schema for Term Graphs

To facilitate software reuse, the ADIC 2.0 and OpenAD/F [12] automatic differentiation tools are constructed in a modular fashion, as depicted in Fig. 8. Language-specific frontends communicate with a differentiation module using an XML representation of the mathematically-relevant elements of a program. The XML Abstract Interface Form (XAIF) [7] provides a language-independent representation of con-

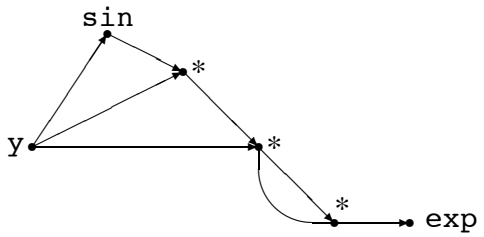

Fig. 6. Computational graph when a is aliased to b.

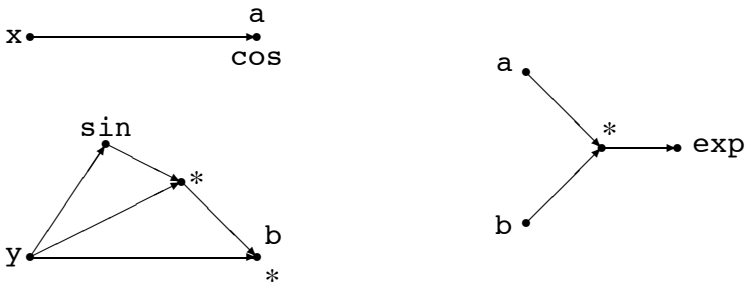

Fig. 7. Term graphs for individual statements.

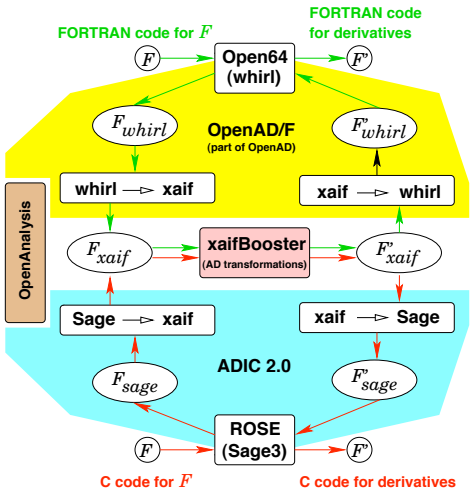

Fig. 8. Schematic of the OpenAD/F architecture.

structs common in imperative languages, such as C, C++, and Fortran. The program is represented as a sequence of nested graphs: a call graph that contains a scope tree and one or more control flow graphs, whose vertices are basic blocks. Each basic block contains assignment statements whose right-hand sides are expression term graphs. At the assignment statement level, imperative languages are not very different from other types of languages, making the XAIF useful for representing term graphs for expressions in non-imperative languages. The XAIF schemas are also designed with extensibility in mind, allowing easy customization of the contents of

```
<xaif:Assignment statement_id="139561992">
  <xaif:AssignmentLHS>
   <xaif:SymbolReference annotation="139557224" scope_id="4"
                         symbol_id="b" vertex_id="1"/>
   </xaif:SymbolReference>
  </xaif:AssignmentLHS>
  <xaif:AssignmentRHS>
    <xaif:VariableReference vertex_id="1">
      <xaif:SymbolReference annotation="139554184" scope_id="3"
                            symbol_id="y" vertex_id="1"/>
    </xaif:VariableReference>
    <xaif:Intrinsic name="sin_scal" vertex_id="2"/>
    <xaif:Intrinsic name="mul_scal_scal" vertex_id="3"/>
    <xaif:Intrinsic name="mul_scal_scal" vertex_id="4"/>
    <xaif:VariableReference vertex_id="2">
      <xaif:SymbolReference annotation="139554184" scope_id="3"
                            symbol_id="y" vertex_id="1"/>
    </xaif:VariableReference>
    <xaif:ExpressionEdge edge_id="1" position="1" source="1" target="2"/>
    <xaif:ExpressionEdge edge_id="2" position="1" source="2" target="3"/>
    <xaif:ExpressionEdge edge_id="3" position="2" source="1" target="3"/>
    <xaif:ExpressionEdge edge_id="4" position="1" source="3" target="4"/>
    <xaif:ExpressionEdge edge_id="5" position="2" source="1" target="4"/>
  </xaif:AssignmentRHS>
</xaif:Assignment>
```

Fig. 9. XAIF representation of the statement `b=sin(y)*y*y`.

graph, vertex, and edge elements.

Figure 9 shows an XAIF fragment describing the computational graph for the second statement in the simple example in Fig. 1. Each assignment statement consists of a `AssignmentLHS` and `AssignmentRHS` elements. The expression graph in the `AssignmentRHS` element can contain vertices corresponding to variable references, constants, binary, and unary operators, as illustrated in Fig. 7 (excluding the edge weights). In the automatic differentiation context, first the XAIF representation of a program is generated by a language-specific frontend, then the XAIF is transformed by a language-independent differentiation module, and finally the resulting new XAIF is parsed by the language-specific backend and merged with the original language-specific AST representation.

In addition to acyclic term graphs for expressions, the XAIF representation includes elements for expressing scope hierarchies as trees whose vertices are the symbol tables for each scope. Each symbol reference vertex contains `scope_id` and `symbol_id` attributes, which refer to the scope and symbol element definitions contained in the scope hierarchy. This provides the connection between the abstract expression term graph representation and the actual program elements.

4 Representation-Independent Static Analysis

Implementation of efficient automatic differentiation tools requires various types of static analysis. Rather than implement these analyses twice (once for the Open64/SL infrastructure used by OpenAD/F and again for the ROSE/Sage infrastructure used by ADIC 2.0), we have implemented them within the OpenAnalysis framework [11]. OpenAnalysis seeks to decouple compiler analyses from specific intermediate rep-

resentations by introducing analysis-specific interfaces. This facilitates the use of multiple analysis algorithms with a single compiler infrastructure as well as the use of a single analysis implementation with multiple compiler infrastructures. OpenAnalysis provides algorithms for call graph construction, control flow graph construction, alias analysis, and interprocedural data-flow analysis. We have implemented OpenAnalysis interfaces for the Open64/SL (for Fortran) and ROSE/Sage (for C/C++) infrastructures.

One of the most important analyses implemented in OpenAnalysis is alias analysis. Alias analysis is used to identify whether two inputs to a statement should share a vertex or be treated as separate variables. In addition, alias analysis is needed for other static analyses, including data-flow analyses such as reaching definitions. Reaching definitions is a static analysis used to determine which definitions of a variable can possibly reach a particular use of a variable. It can be used to construct the du- and ud-chains used in the flattening procedure described in Section 5.

While the precise flow of information cannot always be determined statically, one can often determine that certain variables, no matter what control path is taken, will never lie along the paths between the independent variables and the dependent variables of interest. Such variables are called passive and do not need to have their derivatives computed. Thus, these variables can be ignored in the construction of term graphs. The OpenAnalysis infrastructure implements an interprocedural data-flow analysis called activity analysis to identify the set of passive variables.

5 Merging Term Graphs by Flattening

In Section 1 we mentioned the possibility of minimizing the cost of computing derivatives using automatic differentiation by searching for an optimal elimination order in the term graphs. For many existing tools the default scope for constructing term graphs is the assignment statement as explained in Section 2.2. It is clear that this limits the improvements one can gain from optimizing the elimination order. Consequently we prefer to construct term graphs that cover a larger scope. On the other hand, due to the complexity of the optimization problem, we must avoid graphs that grow proportionally with the run time of the program as done in Section 2.1. The unrolling of loop bodies in this fashion is a good example of bloating the term graph with repetitive structures that should be avoided. Furthermore, if the control flow contains branches, a unified term graph for these branches requires a transformation that makes the computations in the branches mutually independent.

Therefore, we consider consecutive sequences of assignment statements within basic blocks to be reasonable scopes for constructing term graphs. The example in Section 2.2 illustrates the principal problem that arises due to aliasing. The most familiar form of aliasing occurs with arrays when for example we consider `v[i]` and `v[j]` and we cannot tell at compile time if `i` and `j` will always or never be the same, i.e. `v[i]` and `v[j]` refer to the same address in memory. We can use refined code analyses for the purpose of constructing semantically correct term graphs in

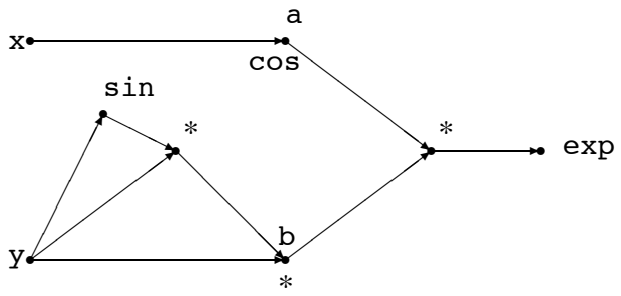

Fig. 10. Merging vertices a and b

the presence of aliasing. The so called *use/define-* or *ud-chains* are a suitable representation for the combined results of alias and dependency analysis. In essence, each use of a variable in the code is associated with a ud-chain that contains a location list of possible definitions. We start out with the term graphs of the individual assignment statements. The left-hand side defined in the assignment is represented by the maximal vertex (we assume side-effect-free expressions) in the term graph. We iterate through all statements in execution order. For each statement we consider each variable use in the right-hand side. If the ud-chain associated with that use contains exactly one element then we can merge the vertex representing the use with the vertex representing the definition. If there is no definition within the scope of the merging process than we retain the vertex as is. If none of the variables from the example in Figure 1 are aliased then the ud-chain for the use of `a` in statement 3 will contain only statement 1 as the definition point. Similarly, the ud-chain for the use of `b` in statement 3 will point only to statement 2 as the definition point and we merge the respective vertices as shown in Figure 10. If `a` is aliased to `b` then their respective ud-chains both point to `b` defined by statement 2 and we obtain the graph as shown in Figure 6.

We already mentioned that quite often the alias analysis does not yield such clear cut results. If we replace `b` in statement 2 by `b[i]` and in statement 3 by `b[j]` then in many cases the ud-chain for the use of `b[j]` in statement 3 will not only point to statement 2 but also to some preceding statement 0, e.g. `b[k]=2*x`, as possible point of definition. While the `a` vertices might still be merged the proper definition point of `b[j]` will only be known at run time. The easiest (but not the only) way to enforce semantical correctness is to organize the merging such that a given statement term graph can either be merged completely or in the case of ambiguous defines we start a new merge. This results in a sequence of merged graphs and semantical correctness is ensured if the derivative accumulation is executed in the order implied by the statement subsequences that make up the merged graphs. In practical applications we observe graph sizes with a few hundred vertices which is reasonable for the elimination heuristics we employ.

6 Conclusions

While we have focused on static techniques in this paper, our automatic differentiation tools often employ hybrid static-dynamic techniques. For example, the determination of whether a variable is active or passive can be deferred until runtime, at least for those variables where static analysis is inconclusive. Future work will examine similar techniques for the situation where static alias analysis is ambiguous.

We have only recently become aware of the connection between term graphs and what the automatic differentiation community refers to as (linearized) computational graphs. It seems likely that the transformations used for vertex and edge elimination in automatic differentiation can be recast as term graph rewrite rules. Furthermore, while computational graphs are always acyclic, term graphs are not. It may be possible to use techniques from term graph rewriting to handle cyclic computational graphs, providing a mechanism to include loops and other control flow constructs in linearized computational graphs.

Conversely, it appears that the automatic differentiation community may be able to contribute technologies to the term graph rewriting community. While the latter community focuses on functional and declarative programming languages, automatic differentiation tools typically target imperative and object-oriented languages. It may be possible to use much of the existing infrastructure to implement term graph rewrite techniques, thus making them available to a broader user community. Even if this sort of technology transfer proves impossible, it is our hope that the automatic differentiation and term graph rewriting research communities can learn from one another.

7 Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy under Contract W-31-109-Eng-38 and by the National Science Foundation under Grant No. OCE-020559. Michelle Strout leads the OpenAnalysis project and implemented the reaching definitions and alias analysis algorithms. Uwe Naumann contributed to the design and implementation of the XAIF.

References

[1] Baur, W. and V. Strassen, *The complexity of partial derivatives*, Theoretical Computer Science **22** (1983), pp. 317–330.

[2] Bischof, C. H., P. D. Hovland and B. Norris, *Implementation of automatic differentiation tools*, in: *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Portland , Oregon, January 14–15, 2002* (2002), pp. 98–107.

[3] Bischof, C. H., P. D. Hovland and B. Norris, *Implementation of automatic differentiation tools*, Higher-Order and Symbolic Computation (2006), to appear.

[4] Griewank, A., "Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation," Number 19 in Frontiers in Appl. Math., SIAM, Philadelphia, PA, 2000.

[5] Griewank, A., D. Juedes and J. Utke, *ADOL–C, a package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Software **22** (1996), pp. 131–167.

[6] Griewank, A. and S. Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, in: A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, PA, 1991 pp. 126–135.

[7] Hovland, P. D., U. Naumann and B. Norris, *An XML-based platform for semantic transformation of numerical programs*, in: M. Hamza, editor, *Software Engineering and Applications* (2002), pp. 530–538.

[8] Naumann, U., "Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs," Ph.D. thesis, Technical University of Dresden (1999).

[9] Naumann, U., *Optimal accumulation of jacobian matrices by elimination methods on the dual computational graph*, Math. Program. **99** (2004), pp. 399–421.

[10] Naumann, U. and P. Gottschling, *Simulated annealing for optimal pivot selection in Jacobian accumulation*, in: A. Albrecht and K. Steinhöfel, editors, *Stochastic Algorithms: Foundations and Applications*, Lecture Notes in Computer Science **2827** (2003), pp. 83–97.

[11] Strout, M. M., J. Mellor-Crummey and P. Hovland, *Representation-independent program analysis*, in: *Proceedings of the Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2005.

[12] Utke, J., *OpenAD web site*, http://www.mcs.anl.gov/openad.