Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2010) 1–11
Procedia Computer Science

ADIC2: Development of a Component Source Transformation System for Differentiating C and C++

Sri Hari Krishna Narayanan, Boyana Norris

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

Beata Winnicka

Bloomington, IN, USA

Abstract

We present a new tool, ADIC2, for automatic differentiation (AD) of C and C++ code through source-to-source transformation. ADIC2 is the successor of the ADIC differentiation tool, which supports forward mode AD of C and a small subset of C++. ADIC2 was completely redesigned and reimplemented as part of the OpenAD software framework, resulting in a robust, flexible, and extensible tool for differentiating C and some features of C++, with plans for full support of C++ in the near future. We discuss some of the challenges in creating AD tools for C and C++ in general and describe the component approach employed in the design and implementation of ADIC2.

Keywords: automatic differentiation, source transformation, ADIC2
PACS: 02.60.Cb, 02.70.Wz
2010 MSC: 68N99

1. Introduction

Automatic differentiation (AD) [1] is a process for producing derivative computations from computer programs. The resulting derivatives are accurate to machine precision with respect to the original computation and can be used in many contexts, including uncertainty quantification, numerical optimization, nonlinear partial differential equation solvers or the solution of inverse problems using least squares. Many tools provide AD for different languages, including Fortran, C, Matlab, and C++ (e.g., [1, 2, 3, 4]). AD tools generally adopt one of two implementation

Email addresses: snarayan@mcs.anl.gov (Sri Hari Krishna Narayanan), norris@mcs.anl.gov (Boyana Norris)

approaches: operator overloading (in languages that support it) or source transformation. Operator overloading-based tools are generally easier to implement, but because they rely on runtime evaluation of partial derivatives, the ways in which the chain rule associativity can be exploited to attain better performing derivative code are limited. Source transformation approaches can exploit static analysis over an entire program and thus can potentially produce better performing derivative code. Source transformation-based AD has the same limitations as traditional compilers, namely the difficulty of implementing parsing and analysis infrastructure for complex languages, as well as reliance on necessarily conservative static analysis (e.g., alias analysis in C or C++), which may lead to generating suboptimal derivative code. In this paper we introduce ADIC2 [5], an AD tool that takes the source transformation approach to provide differentiation capabilities for C and C++. ADIC2 is part of the OpenAD [6] project, which includes the independently-developed XAIFBooster [7] and OpenAnalysis [8] components.

The first implementation of ADIC [3] generates derivatives for C code using the forward mode of AD, with support for reverse mode at the statement level. In cases when the number of dependent variables is much smaller than the number of independent variables, the reverse mode of differentiation can result in better-performing derivative code or enable the solution of problems for which forward-mode AD derivatives are infeasible due to excessive memory requirements or computational cost. Implementing reverse mode differentiation is significantly more involved, however, and was thus not completed in the first version of ADIC. Furthermore, C++ support was very limited due to the reliance on the Sage++ research compiler infrastructure, which did not provide robust support for C++.

ADIC2 builds on the successful design ideas of ADIC (e.g., the use of a language-independent intermediate program representation and multiple independent differentiation modules) and addresses the challenges of parsing C and C++ by leveraging the compiler infrastructure developed by the ROSE project [9, 10] at Lawrence Livermore National Laboratory. ROSE adopted and significantly extended the Sage++ research compiler infrastructure (used in the first version of ADIC) to create Sage III, a high-level abstract syntax tree (AST) representation that provides extensive support for traversals and transformations. ROSE relies on the commercial C and C++ parsers developed by the Edison Design Group (EDG) [11], which also underly a number of commercial C++ compilers. EDG is the only closed source ADIC2 prerequisite; however, ROSE includes a free binary distribution of EDG C/C++ parsers for a number of platforms, ensuring continuous and robust parsing support for these languages. While some of the design ideas are similar to those in ADIC, ADIC2 was completely reimplemented to use ROSE and interface to OpenAnalysis program analysis algorithms and OpenAD differentiation modules. Before we discuss the implementation in more detail, we present an example of derivative code produced with ADIC2.

1.1. Example

Figure 1(a) shows an example input code that employs a pointer type, basic arithmetic, and a function call to an intrinsic function. Any valid C and certain C++ features are also supported. Figure 1(b) shows the forward-mode derivative code generated by ADIC2 (using the *BasicBlockPreaccumulation* XAIFBooster algorithm). A driver program (not shown) contains initializations including setting up the independent variable (in this case, x) and produces results shown in Fig. 1(c) comparing derivatives computed analytically, through divided differences with a δh value of 0.001, and by using the ADIC2-generated code in Fig. 1(b).

Analogously, Fig. 2 shows the reverse mode derivative code generated by ADIC2 (using the *BasicBlockPreaccumulationReverse* XAIFBooster algorithm). The driver for the reverse-

(a) Original code:

```
#include <math.h>

void foo(double *y, double x)
{
  *y = x + sin(x * x);
}
```

(b) Generated forward-mode AD code:

```
#include "ad_types.h"
#include <math.h>
void ad_foo(DERIV_TYPE *y,DERIV_TYPE *x)
{
  DERIV_TYPE ad_TempVarprop_0;
  double ad_TempVaracc_0;
  double ad_TempVarlin_2;
  double ad_TempVarlin_1;
  double ad_TempVarlin_0;
  double ad_TempVarSymbol_0;
  double ad_TempVartmp_0;
  ad_TempVartmp_0 = DERIV_val( *x) * DERIV_val( *x);
  ad_TempVarSymbol_0 = DERIV_val( *x) + sin(ad_TempVartmp_0);
  ad_TempVarlin_0 = DERIV_val( *x);
  ad_TempVarlin_1 = DERIV_val( *x);
  ad_TempVarlin_2 = cos(ad_TempVartmp_0);
  DERIV_val( *y) = ad_TempVarSymbol_0;
  ad_TempVaracc_0 = 1 + (ad_TempVarlin_0 + ad_TempVarlin_1)
                                                    * ad_TempVarlin_2;
  ADIC_SetDeriv(DERIV_TYPE_ref( *x),DERIV_TYPE_ref(ad_TempVarprop_0));
  ADIC_Sax(ad_TempVaracc_0,DERIV_TYPE_ref(ad_TempVarprop_0),
                                                DERIV_TYPE_ref( *y));
}
```

(c) Results:

```
Analytic result is:  [1.968912]
DD result is:  [1.968997] error {-8.45e-05}
AD result is:  [1.968912] error {0.00e+00}
```

Figure 1: (a) Example input code; (b) generated forward-mode differentiated code; (c) Results obtained through different differentiation approaches.

mode differs slightly from the one used for the results in Fig. 1 in that the user must specify the dependent variable and extract the result from the gradient portion of the independent variable. ADIC2 supports both scalar and vector differentiation modes.

(a) Generated adjoint code:

```
#include "ad_types.h"
#include <math.h>
void ad_foo(DERIV_TYPE *y,DERIV_TYPE *x)
{
  if ((our_rev_mode.plain) == 1) {
    DERIV_val( *y) = DERIV_val( *x) + sin(DERIV_val( *x)
                                          * DERIV_val( *x));
}
  if ((our_rev_mode.tape) == 1) {
    double ad_TempVartmp_0;
    double ad_TempVarSymbol_0;
    double ad_TempVarlin_0;
    double ad_TempVarlin_1;
    double ad_TempVarlin_2;
    double ad_TempVaracc_0;
    ad_TempVartmp_0 = DERIV_val( *x) * DERIV_val( *x);
    ad_TempVarSymbol_0 = DERIV_val( *x) + sin(ad_TempVartmp_0);
    ad_TempVarlin_0 = DERIV_val( *x);
    ad_TempVarlin_1 = DERIV_val( *x);
    ad_TempVarlin_2 = cos(ad_TempVartmp_0);
    DERIV_val( *y) = ad_TempVarSymbol_0;
    ad_TempVaracc_0 = 1 + (ad_TempVarlin_0 + ad_TempVarlin_1)
                                           * ad_TempVarlin_2;
    ADIC_push(ad_TempVaracc_0);
  }
  if ((our_rev_mode.adjoint) == 1) {
    double ad_TempVarSymbol_1;
    DERIV_TYPE ad_TempVarprop_0;
    ADIC_ZeroDeriv(DERIV_TYPE_ref(ad_TempVarprop_0));
    ADIC_pop(ad_TempVarSymbol_1);
    ADIC_Saxpy(ad_TempVarSymbol_1,DERIV_TYPE_ref( *y),
                                  DERIV_TYPE_ref(ad_TempVarprop_0));
    ADIC_ZeroDeriv(DERIV_TYPE_ref( *y));
    ADIC_IncDeriv(DERIV_TYPE_ref(ad_TempVarprop_0),DERIV_TYPE_ref( *x));
    ADIC_ZeroDeriv(DERIV_TYPE_ref(ad_TempVarprop_0));
  }
}
```

(b) Results:

```
Analytic result is:  [1.968912]
DD result is:  [1.968997] error {-8.45e-05}
AD result is:  [1.968912] error {0.00e+00}
```

Figure 2: Generated reverse-mode differentiated code (a) and comparison with the analytic derivative and divided differences (b).

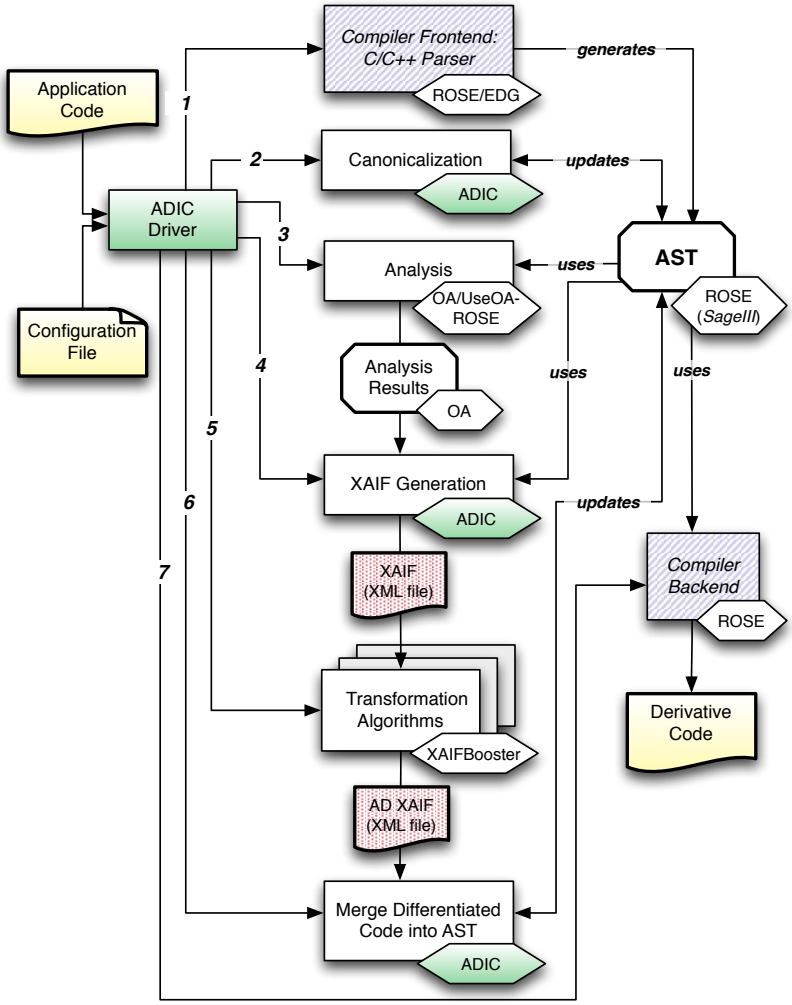

Figure 3: Overview of the ADIC2 differentiation process. The sequence of steps is indicated by numbers embedded in arrows connected to the corresponding components implementing them.

2. Design and Implementation

ADIC2 is part of the OpenAD [6] project, which encompasses multiple independently developed software components, including XAIFBooster [7], OpenAnalysis [8], and parsing and code generation support for Fortran, C, and C++. Figure 3 illustrates the portions of the OpenAD infrastructure that are integrated within ADIC2. In the remainder of this section we describe the ADIC2 implementation in sequence of the typical differentiation workflow.

2.1. Compiler Frontend: Parser and Analysis

As indicated in the first step in Fig. 3, ADIC2 relies on the ROSE compiler framework to parse the input source code and generate an *abstract syntax tree* (AST) consisting of Sage III nodes [9, 10]. ROSE provides access to AST nodes through hierarchical traversals including *TopDown*, *BottomUp* and *TopDownBottomUp*. Modifications to existing nodes and addition of new nodes are possible during traversals. Nodes can be decorated with attributes which persist between visits in a single traversal or multiple separate traversals. ADIC2 first identifies nodes of interest and sets relevant attributes for them during a *TopDownBottomUp* traversal. For example, ADIC2 is only concerned with nodes representing floating-point values. Therefore, it identifies such nodes and sets an attribute to designate them as *differentiable* As another example, each scope statement (code contained within curly braces in C/C++) is assigned a unique number to be used for identification in later steps. The last task in this step is to add certain data structures that are not provided by ROSE but are convenient to have with ADIC2 to support later differentiation stages. One of the development challenges faced by ADIC2 is the close coupling with the ROSE interfaces and implementation. Changes within ROSE can result in failures in ADIC2 that can only be fixed by pervasive updates. As ROSE matures, we expect that such updates will become easier and less frequent.

ADIC2 also reads one or more configuration files, in which users can specify various settings such as differentiation module parameters, inactive symbols (e.g., types, variables, or functions that do not lie on the computational path between independent and dependent variables), and other parameters that are used to fine-tune the differentiation process. For example, users can supplement the analysis capabilities in ADIC2 (see Sec. 2.3) by explicitly indicating that certain functions or types are inactive and thus improving the performance of the differentiated code.

2.2. Canonicalization

The principal function of the second phase (see arrow labeled with 2 in Fig. 3) of the ADIC2 workflow is to modify *differentiable* variable declarations and references such that the Sage III AST can be interfaced efficiently with other components of ADIC2. In addition, complex control flow are simplified for later stages. This is achieved through the traversal of all the statements present in the top-level *SgProject* AST node which represents the entire source code. The following list outlines the actions taken for a subset of the nodes of interest.

- **Variable declaration.** The type of all differentiable (i.e., floating-point) variables is converted to a new type, *DERIV_TYPE*, which is defined in a header file automatically included in the differentiated code. For example, one commonly used type is a struct containing a scalar field for the original value and an array to hold the computed partial derivative values. For forward mode, the size of the array depends on the number of independent variables in the code. The names of differentiable variables are prefixed with a user-defined prefix which is usually *ad_* but can be customized by the user. Some of the challenges in canonicalizing declarations include proper handling of typedef statements and traversing type hierarchies, correctly identifying those that are relevant to differentiation.

- **Function definition.** Because C does not support struct return types, if a function returns a value of a differentiable type, a special variable *ad_var_ret* of type *DERIV_TYPE* * is added to the function's argument list and all return statements in the function definition are converted to assignments to **ad_var_ret*.

- **Function Call.** If return value of a function is differentiable and assigned to a variable x, then $\&x$ is added to the argument list of the function call to accommodate the function declaration return value change described earlier. Scalar arguments of differentiable types are converted to references to pointers of *DERIV_TYPE*.

- **For loop.** In order to simplify further phases, a dummy basic block is added immediately after the for loop statement for use in later steps.

- **If statement.** The conditional expression is hoisted out of the statement and is assigned to a new boolean variable. In order to simplify further phases, a dummy basic block is added immediately after the conditional statement.

2.3. Analysis

In the next step (see arrow labeled with 3 in Fig. 3) the canonicalized AST serves as input to OpenAnalysis, a representation -independent program analysis toolkit [8, 12]. ADIC2 uses OpenAnalysis through the interface UseOA-ROSE [13], which provides program information to analysis algorithms without exposing them to the particulars of the Sage III data structures. The analysis results are used to reduce the amount of code that is passed to the differentiation algorithms later and to ensure the accuracy of differentiated code. ADIC2 currently applies the following analyses from OpenAnalysis: call graph and control flow graph (CFG) construction, side effects, def-use and use-def chains, def-overwrite chains, and alias analysis. Activity analysis will also be integrated in the near future.

UseOA-ROSE connects ROSE and OpenAnalysis and thus its implementation must track changes within both packages. Until recently, OpenAnalysis had been evolving rapidly in several development branches. In order to enable support for reverse mode differentiation in ADIC2, we successfully merged some of these branches and continue to participate in OpenAnalysis development, with the eventual goal to have the same OpenAnalysis algorithm implementations used in the different parts of the OpenAD framework.

2.4. XAIF Generation

In this phase (see arrow labeled with 4 in Fig. 3), the OpenAnalysis results and AST information are merged and expressed in the XML Abstract Interface Form (XAIF) [14, 15]. XAIF provides a language-independent representation of constructs common in imperative languages, such as C, C++, and Fortran. It also allows the embedding of annotations or pointers back to the corresponding language-dependent AST node from which they were created. Constructs that are not used for the differentiation (transformation) can be included and designated as *marked statements*, which are subsequently ignored (and preserved) by the transformation algorithms. These statements are also used to delimit sequences of statements that must be transformed. Special constructs exist within XAIF to express OpenAnalysis results.

The merging process involves iterating through OpenAnalysis data results and converting the associated Sage III *SgNode*s into XAIF nodes. Special treatment of iterative and conditional statements is required because Sage III, OpenAnalysis and XAIF have different representations for these structures. Expression statements in particular require a traversal through the Sage III subtree rooted at the expression statement. The XAIF file is not generated directly during the traversals; instead, a graph-based representation is first created in memory and later traversed to produce the XAIF files that serve as input to transformation algorithms.

2.5. Transformation Algorithms

ADIC2 next invokes XAIFBooster (or potentially any other transformation module that supports XAIF) to produce a differentiated XAIF file (see arrow labeled with 5 in Fig. 3). Both forward and reverse modes of differentiation are supported.

2.6. Merging of Differentiated Code into AST

The XAIF output generated by the transformation algorithms is parsed and a corresponding graph-based representation is created using an event-based (SAX2) XML parser (see arrow labeled with 6 in Fig. 3). The graph is traversed once in a depth-first manner and at each node or edge, visitor methods are used to integrate the transformed XAIF nodes into the Sage III AST. For example, in ADIC2, we employ two visitors: one that converts the XAIF to Sage III node and another to output the nodes for debugging purposes.

In forward-mode differentiation, the output of the transformation algorithms contains transformed expression statements and marker statements that point to where the expression statements should be inserted into the ROSE AST. Often the transformed expression statements require the creation of new Sage III nodes that do not exist in the original AST. In reverse-mode differentiation marker statements are not available in the reversed computation generated by the transformation algorithm. Therefore, ADIC2 has to create complete control flow structures, including basic blocks, if statements, and loop statements. One of the challenges is that the CFG elements in the XAIF schema do not directly correspond to the data structures in the Sage III AST. Specifically, the XAIF does not have a concept of loop body, conditional statement body or scope statement. Therefore, ADIC2 has to create and maintain additional information and use logic outside that provided by the XAIF to create these structures correctly.

2.7. Compiler Backend

ADIC2 uses ROSE unparsing functionality to create the output code from the modified AST (see arrow labeled with 7 in Fig. 3). ADIC2 provides runtime libraries and header files required for compiling the transformed code.

3. Related Work

As mentioned in Sec. 1, AD tools for C/C++ can be classified according to their implementation approach. One method is to use C++ operator overloading, an approach adopted by a number of tools, including ADOL-C [16], CppAD [17], FAD [18], FADBAD/TADIFF [19], FFADLib [20], and Rapsodia [21].

The differentiation approach used in ADIC2 is source-to-source transformation, i.e., the input source code is transformed into a new code augmented with derivative computations. TAPENADE [22] is another source transformation-based AD tool that differentiates C and Fortran programs using a tool-specific language-independent internal representation. Handling object oriented input is stated as one of the eventual goals. It performs its parsing and source analysis internally. TAC++ [23] is a source-to-source translator that supports both forward and reverse mode differentiation and is proposed as the C++ language counterpart of TAF. It can currently handle a number of C features (e.g., variables of the type as scalar int and scalar double, arrays, typedef types and pointer types). It also supports basic arithmetic operations, intrinsic functions, control flow statements and function calls. It is not clear to us at the time of this writing what C++ features are handled by TAC++.

ADIC2 differs from other C/C++ source transformation-based AD tools in several ways.

- ADIC2 consists of a number of loosely coupled components, some of which are also used by OpenAD/F for support of Fortran. The component design enables independent development of AD algorithms, language-specific support, and program analyses.
- ADIC2 uses the XAIF language-independent intermediate representation which is not tool-specific, and thus can be used to leverage any differentiation approach that can process XAIF as input and produce XAIF output.
- ADIC2 is based on robust compiler infrastructure, including the commercial EDG C/C++ frontend, which ensures that all language features are parsed correctly.

While ADIC2's support for C++ is not complete at present, it is under active development, with a focus on C++-specific canonicalization and correct handling of templates.

4. Conclusions and Future Work

We have described the design and implementation of the ADIC2 system for differentiating C and C++ codes. As part of the OpenAD project, this tool is the first C/C++ AD tool that shares a number of components with its independently developed Fortran counterpart, OpenAD/F. The modular design of the OpenAD infrastructure allows the reuse of program analysis and differentiation modules. This enables access to new analysis and differentiation methods from different languages without having to reimplement them. Furthermore, relying on robust external compiler infrastructure ensures that few if any manual code modifications are required before a code can be differentiated.

ADIC currently handles one input file at a time, but we plan to enable simultaneous differentiation of projects containing multiple source code files as input (this capability is supported by ROSE). This requires careful handling of shared data structures as well as the ability to recognize the portions of the project relevant to ADIC2 based on minimal user input. ADIC2 is also continuously being extended to handle more complex features of C++ (e.g., user-defined templates and everything in the standard template library). In the long term we also plan to explore hybrid source transformation/operator overloading strategies for supporting C++ templated codes for which it may not be possible to determine active types statically.

We are working on integrating activity analysis results into the XAIF input provided to transformation modules. We will continue to explore different analyses that can help generate more efficient derivative code. For example, we plan to integrate more accurate array section analysis (currently OpenAnalysis does not take array reference indices into account in data flow analyses).

In the next few months, we will begin applying ADIC2 to applications and compare the performance of the generated code to that produced by other AD tools when possible. We are in the process of integrating ADIC2 into libraries such as PETSc [24], which currently uses the first version of ADIC to automatically provide Jacobians for applications involving the nonlinear solution of partial differential equations [25].

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. The authors would like to thank Jean Utke of the University of Chicago for his invaluable suggestions and help.

References

[1] M. Berz, C. Bischof, G. Corliss, A. Griewank (Eds.), Computational Differentiation: Techniques, Applications, and Tools, SIAM, Philadelphia, PA, 1996.

[2] J. Utke, OpenAD: Algorithm implementation user guide, Technical Memorandum ANL/MCS–TM–274, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (2004).

[3] C. H. Bischof, L. Roh, A. Mauer, ADIC — An extensible automatic differentiation tool for ANSI-C, Software–Practice and Experience 27 (12) (1997) 1427–1456.

[4] A. Griewank, On automatic differentiation, in: M. Iri, K. Tanabe (Eds.), Mathematical Programming: Recent Developments and Applications, Kluwer Academic Publishers, Dordrecht, 1989, pp. 83–108.

[5] ADIC2 Web Page, `http://trac.mcs.anl.gov/projects/ADIC/`.

[6] OpenAD Web Page, `http://www.mcs.anl.gov/OpenAD/`.

[7] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, C. Wunsch, OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes, ACM Trans. Math. Softw. 34 (4) (2008) 1–36. doi:`http://doi.acm.org/10.1145/1377596.1377598`.

[8] M. M. Strout, J. Mellor-Crummey, P. Hovland, Representation-independent program analysis, SIGSOFT Softw. Eng. Notes 31 (1) (2006) 67–74. doi:`http://doi.acm.org/10.1145/1108768.1108810`.

[9] D. Quinlan, ROSE Web Page, `http://rosecompiler.org`.

[10] M. Schordan, D. Quinlan, A source-to-source architecture for user-defined optimizations, in: JMLC'03: Joint Modular Languages Conference, Vol. 2789 of Lecture Notes in Computer Science, Springer Verlag, 2003, pp. 214–223.

[11] EDG C++ Front End, `http://www.edg.com/index.php?location=c_frontend`.

[12] D. Quinlan, ROSE web page, `http://rosecompiler.org`.

[13] UseOA-ROSE Web Page, `http://developer.berlios.de/projects/useoa-rose/`.

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

[15] XAIF Web Page, `http://www-unix.mcs.anl.gov/xaif/`.

[16] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, A. Walther, ADOL-C: A package for the automatic differentiation of algorithms written in C/C++, Tech. rep., Technical University of Dresden, Institute of Scientific Computing and Institute of Geometry, updated version of the paper published in it ACM Trans. Math. Software 22, 1996, 131–167 (1999).

[17] B. M. Bell, cppAD Web Page, `http://www.coin-or.org/CppAD/`.

[18] P. Aubert, N. Di Césaré, Expression templates and forward mode automatic differentiation, in: G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (Eds.), Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science, Springer, New York, NY, 2001, Ch. 37, pp. 311–315.

[19] C. Bendtsen, O. Stauning, FADBAD, a flexible C++ package for automatic differentiation, Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark (Aug. 1996).

[20] I. Tsukanov, M.Hall, Fast forward automatic differentiation library (FFADLib): A user manual, Technical report 2000-4, Spatial Automation Laboratory, University of Wisconsin-Madison, `http://sal-cnc.me.wisc.edu/publications/auto_diff.html` (December 2000).

[21] I. Charpentier, J. Utke, Fast higher-order derivative tensors with Rapsodia, Optimization Methods Software 24 (1) (2009) 1–14. doi:http://dx.doi.org/10.1080/10556780802413769.

[22] V. Pascual, L. Hascoët, TAPENADE for C, in: Advances in Automatic Differentiation, Lecture Notes in Computational Science and Engineering, Springer, 2008, pp. 199–210, selected papers from AD2008 Bonn, August 2008.

[23] M. Voßbeck, R. Giering, T. Kaminski, Development and first applications of TAC++, in: C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Springer, 2008, pp. 187–197. doi:10.1007/978-3-540-68942-3_17.

[24] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. M. cInnes, B. F. Smith, H. Zhang, PETSc Web page, `http://www.mcs.anl.gov/petsc` (2009).

[25] P. Hovland, B. Norris, B. Smith, Making automatic differentiation truly automatic: Coupling PETSc with ADIC, in: P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, A. G. Hoekstra (Eds.), Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II, Vol. 2330 of Lecture Notes in Computer Science, Springer, Berlin, 2002, pp. 1087–1096.