Interfacing source transformation AD with operator overloading libraries

Kshitij Kulshreshtha[*] and Sri Hari Krishna Narayanan[†]

March 2016

1 Introduction

Scientific applications are usually written in a single language such as C, C++, or a flavor of Fortran. Various algorithmic differentiation (AD) tools exist to differentiate these applications by using source transformation [1, 2, 3, 4] or operator-overloading [5, 6]. The language that an application is written in often dictates the approach and tool to be used for differentiation. Additionally, performance considerations, tool philosophy (recomputation vs. storage), and specific tool capabilities such as support for sparse Jacobians or fixed-point iterations may play a role in the tool that is ultimately used to differentiate an application.

Operator overloading and source transformation have their own strengths and weaknesses. The most important feature of operator-overloading-based AD tools is that they can be used in software regardless of its design complexity. Many large simulation tools (e.g., ISSM [7], SU2 [8, 9]) have been successfully differentiated in this manner. One drawback, however, is the large amount of memory required to store the tool-specific internal representation of the computation in order to run it in the reverse mode. The execution speed also suffers because of low compiler optimization potential. Both problems do not occur in source transformation tools. However, such tools cannot handle runtime features of C++ such as object inheritance, polymorphism, and templating.

Some scientific applications are coded in C++ but contain portions that are in C or are C like. Often, the global structure of the application is complex and requires the use of advanced C++ features, but the computationally intensive portions are C functions. In our previous work [10] we successfully demonstrated the interfacing of an application differentiated mainly by using operator overloading with a library that has been differentiated using source transformation. The implementation relies on the externally differentiated function feature of ADOL-C, where such functions have actually been differentiated by using ADIC. The interface works also well with Tapenade. We demonstrated that this mixed approach is able to amortize the memory requirement for the calculation of adjoints and Jacobians on two applications. By using both approaches in the same application wherever they are applicable and well suited, we were able to use the strengths of both approaches.

In this work we demonstrate the inverse interface. Here, the global application structure is a straightforward pure C (or Fortran) implementation; however, certain library calls may internally use complicated C++ features. We have created an entirely new interface keeping in mind the features of both ADOL-C and ADIC to support this setup.

2 Interfacing

When the C++ library is differentiated by using ADOL-C, an ADOL-C–specific internal representation of the computation, called a trace, will be generated. The interpretation of such a trace with the drivers provided in the ADOL-C library results in the computation of the derivatives for this function. To interface a C++ library function with an external application that is differentiated by ADIC, we annotate the C++ function in the library and use a preprocessor to generate the helper codes listed below.

- ADIC stub containing the same activity patterns as the C++ function
- Interfacing function that replaces the differentiated-stub
- Trace creation source and header

The ADIC stub is used in the outer code to allow ADIC to have access to a complete code base. By keeping the activity patterns consistent, ADIC creates an appropriate differentiated stub. However, this differentiated stub must be replaced in ADIC's output code with an interfacing function that contains calls to appropriate ADOL-C drivers.

Additionally, the trace creation source/header files are used to create the trace for the C++ function before computing the derivatives. This process occurs when the interfacing function is executed for the first time. This is true even if the outer code contains multiple calls to the C++ function, since ADOL-C traces are reusable at different evaluation points. An exception to this rule is when an ADOL-C branch switch warning occurs that requires setting up the trace again at an appropriate evaluation point.

[*]Paderborn University, Germany, `kshitij@math.upb.de`
[†]Argonne National Laboratory, IL, USA, `snarayan@mcs.anl.gov`

As is the norm in differentiating a library with ADOL-C, the source code of the C++ function must be manually instrumented for using operator overloading with ADOL-C. The ADIC-generated files and preprocessor-generated files are compiled and appropriately linked to the ADOL-C differentiated library. In this setup, the interfacing function copies primal data from ADIC's buffers, as well as pointers to tangent or adjoint data from ADIC's buffers, and passes them to the ADOL-C driver function for the appropriate mode of differentiation. The primal data is then copied back into ADIC's variables. The output adjoint/tangent data is implicitly written into ADIC's buffers because of the use of pointers.

The workflow for the C++ library developer involves instrumentating the library as usual for differentiation by ADOL-C. Additionally the developer must write annotations for each API function and run the preprocessor to generate files for each of them. The developer of the C application must use the stub function to the differentiate application using ADIC. The differentiated stub must then be discarded. The build process for the application must be modified to additionally compile and link the preprocessor-generated files.

3 Annotations and Preprocessing

The annotation used in the C++ library is itself a description of the arguments of the C++ function, in that it specifies the active input and output variables, as well as the inactive indices, sizes, counters, and so forth. The input and output variables can be scalars, vectors, matrices, or slices of vectors or matrices, or even lower, or upper triangular matrices, or slices thereof. The annotation describes the dimensions associated with each variable name as well as its position in the formal argument list. This annotation is written by using Python syntax inside a specialized C++ comment. It is therefore ignored by a C/C++ compiler, and the Python-based preprocessor does not bother with the actual C/C++ code. One such annotation for a simple case can be seen in Fig. 1

```
/*@ stad_export interface
name = 'k_eval'
iarr = [ ('n', 2), ('m', 4), ('j', 6) ]
input = [ ('y', [ 'iArr[0]' ], 1), ('u', [ 'iArr[1]' ], 3) ]
output =[ ('k', [ 's', 'iArr[2]', 'iArr[2]+1', '0', 'iArr[0]' ], 5 )
     ]
  @*/
void k_eval(double *y, int n, double *u, int m, double **k,
        int j)
```

Figure 1: Annotations describing input and output variables with their dimensions

Writing the annotation is intuitive for anyone familiar with the semantics of the API function being annotated. The annotation is written as a list of tuples for each of input, output, and integer data (iArr). The tuple consists of the formal argument name, its dimension if it is an active input or output, and its position in the formal parameter list. For inactive integers the dimension is not required because they are all scalars. The numbers in the list of inactive integers may be referenced to provide the dimensions of the input or output variables by indexing them starting at 0. For example, in Fig. 1 `iArr[0]` is a reference to the parameter named `n` mentioned as the first element of the `iarr` list. Dimensions may also be given in terms of global variables or expressions. The dimension itself is a list of length up to 6. Length 0 implies scalars, length 1 vectors, and length 2 matrices. Additionally, one may specify slices of vectors and matrices by putting the character "`s`" as the first element of the dimension list, followed by the first index and past-end-of-slice index for each dimension respectively. Upper or lower triangular matrices may be specified by putting the character "`u`" or "`l`" at the end of the dimension list.

In order to generate C code for the interfacing function described in the preceding section, as well as the code to set up the trace properly and a stub to be processed by ADIC, the annotation is extracted and processed by a Python-based preprocessing script. Processing the information about the formal argument names, their dimensions, and the position in the parameter list allows one to use a generic loop structure to copy required data from one data structure to another and call either the ADOL-C–instrumented API function or the driver functions of ADOL-C in an application-agnostic way. The preprocessor relies mainly on regular expressions and string concatenation in Python. Only the portion between `/*@` and `@*/` is parsed in Python. The skeleton of the interface code generated from the annotation in Fig. 1 is shown in Fig. 2.

4 Validation

We have tested the mixed approach on a simple scalable test case of moderate code size modeling a periodic adsorption process used in an optimal control setting. This code is used to compute the derivatives required by the optimization algorithm, namely, gradients and Jacobians.

The periodic adsorption process was studied from an optimization point of view in [11, 12]. A system of PDAEs in time and space with periodic boundary conditions models the cyclic steady state of a process, where a fluid is preferentially absorbed on the surface of a sorbent bed. This leads to dense Jacobians that dominate the computation time (see [11]). Therefore, previous works have used inexact Jacobians (for example, [12]). Using AD, however, we

```
void ad_k_eval(DERIV_TYPE *y, int n, DERIV_TYPE *u, int m, DERIV_TYPE **k, int j){
  static char firsttime = 1;
  if (firsttime) {
    setup_tape_k_eval(y, n, u, m, k, j);
    firsttime = 0;
  }
  int iArr[3];
  double *invec, *outvec;
  if (our_rev_mode.adjoint ==TRUE)
     //Pop iArr
  } else {
    iArr[0] = n; iArr[1] = m; iArr[2] = j;
  }
  invec = (double *) malloc((((iArr[0])-(0))+ ((iArr[1])-(0))) * sizeof(double));
  outvec = (double *) malloc(((((iArr[2]+1)-(iArr[2]))*((iArr[0])-(0)))) * sizeof(double));
  //Store DERIV_val(y, u) in invec
  if (our_rev_mode.plain==TRUE){
    zos_forward(1,(((iArr[2]+1)-(iArr[2]))*((iArr[0])-(0))),((iArr[0])-(0))+ ((iArr[1])-(0)),0,invec,outvec);
     //Store outvec DERIV_val(k)
  } else if (our_rev_mode.tape == TRUE) {
     //Push invec
     //Push iArr
    zos_forward(1,(((iArr[2]+1)-(iArr[2]))*((iArr[0])-(0))),((iArr[0])-(0))+ ((iArr[1])-(0)),0,invec,outvec);
    //Store outvec DERIV_val(k)
  } else if (our_rev_mode.adjoint ==TRUE) {
    //Pop invec
    double **inbar, **outbar;
    inbar = (double **) malloc((((iArr[0])-(0))+ ((iArr[1])-(0))) * sizeof(double*));
    outbar = (double **) malloc(((((iArr[2]+1)-(iArr[2]))*((iArr[0])-(0)))) * sizeof(double*));
     //Point inbar to DERIV_grad(y) and DERIV_grad(u)
     //Point outbar to DERIV_grad(k)
    zos_forward(1,(((iArr[2]+1)-(iArr[2]))*((iArr[0])-(0))),((iArr[0])-(0))+ ((iArr[1])-(0)),1,invec,outvec);
    set_nested_ctx(1,1);
    fov_reverse(1,(((iArr[2]+1)-(iArr[2]))*((iArr[0])-(0))),((iArr[0])-(0))+ ((iArr[1])-(0)),__ADIC_GradSize(),outbar,inbar);
    set_nested_ctx(1,0);
    free(inbar);
    free(outbar);
  }
  free(invec);
  free(outvec);
}
```

Figure 2: Interfacing function with ADIC data and ADOL-C drivers

compute the equality and inequality constraint Jacobians as well as the objective gradient exactly up to machine precision. The PDAE system is discretized in space by using a finite-volume approach, and the resulting system of ODEs is then integrated in time by using a Runge-Kutta method.

The code of the original application was written completely in C. This application has been separately differentiated by using both ADIC and ADOL-C in the past. Therefore, to test the approach, we arbitrarily divided the call graph of the application into two portions: one to be differentiated by ADIC and one to be differentiated by ADOL-C. We added annotations to the ADOL-C portion and used the preprocessor to generate the interfacing function, stub, and tape creation header and source. We then used ADIC to generate derivative code and modified the ADOL-C portion by transforming all the double variables to adoubles. We then modified the build process of the original application and generated an executable. The resulting values for the objective gradient and the equality and inequality constraint Jacobians matched the values computed by a purely ADOL-C application.

5 Conclusion and Future Work

We have developed a method to interface a Fortran or C application differentiated by source transformation AD with a C++ library differentiated by operator overloading using ADOL-C. The method requires the use of hand written annotations to automatically generate additional files. We have shown that the method works on a medium sized application. In the future, we would like to study the approach on a large scale application differentiated by ADIC or Tapenade that must be interfaced with a large library differentiated by ADOL-C. We would also like to study the exploitation of structure such as sparsity within the combined application in such a framework.

Acknowledgments. This work was funded in part by a grant from DAAD Project Based Personnel Exchange Programme and by a grant from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] Jean Utke, Uwe Naumann, Mike Fagan, et al. OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes. *ACM Trans. Math. Softw.*, 34(4):18:1–18:36, 2008.

[2] Laurent Hascoet and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, May 2013.

[3] Ralf Giering and Thomas Kaminski. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. In *Proceedings of GAMM 2002, Augsburg, Germany*, 2002.

[4] Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. ADIC2: Development of a component source transformation system for differentiating C and C++ . *Procedia Computer Science*, 1(1):1845–1853, 2010. ICCS 2010.

[5] Andrea Walther and Andreas Griewank. Getting started with ADOL-C. In Uwe Naumann and Olaf Schenk, editors, *Combinatorial Scientific Computing*. Chapman-Hall, 2012.

[6] Eric T. Phipps, Roscoe A. Bartlett, David M. Gay, and Robert J. Hoekstra. Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation. In Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 351–362. Springer, 2008.

[7] E. Larour, J. Utke, B. Csatho, A. Schenk, H. Seroussi, M. Morlighem, E. Rignot, N. Schlegel, and A. Khazendar. Inferred basal friction and surface mass balance of the northeast Greenland ice stream using data assimilation of ICEsat (ice cloud and land elevation satellite) surface altimetry and ISSM (ice sheet system model). *The Cryosphere*, 8(6):2335–2351, 2014.

[8] Francisco Palacios, Juan Alonso, Karthikeyan Duraisamy, et al. Stanford University Unstructured (SU2): An open-source integrated computational environment for multi-physics simulation and design. In *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*. American Institute of Aeronautics and Astronautics, Jan. 2013.

[9] Tim A Albring, Max Sagebaum, and Nicolas R Gauger. Development of a consistent discrete adjoint solver in an evolving aerodynamic design framework. In *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. American Institute of Aeronautics and Astronautics, June 2015.

[10] Kshitij Kulshreshtha, Sri Hari Krishna Narayanan, and Tim Albring. A mixed approach to adjoint computation with algorithmic differentiation. In Lorena Bociu, J.A. Desideri, and A. Habbal, editors, *System Modeling and Optimization (CSMO 2015), Proceedings of the IFIP TC7 Conference 2015*, IFIP Advances in Information and Communication Technology. Springer, 2016. submitted, available as preprint ANL/MCS-P5426-1015.

[11] L. Jiang, L.T. Biegler, and G. Fox. Optimization of pressure swing adsorption systems for air separation. *AIChE Journal*, 49:1140–1157, 2003.

[12] S. R. Vetukuri, L. T. Biegler, and A. Walther. An inexact trust-region algorithm for the optimization of periodic adsorption processes. *I & EC Research*, 49:12004–12013, 2010.