Source-transformation Differentiation of a C++-like Ice Sheet model

Laurent Hascoët, Mathieu Morlighem

Source-transformation Differentiation of a C++-like Ice Sheet model

Laurent Hascoët*, Mathieu Morlighem†

March 2016

1 Introduction

Algorithmic Differentiation (AD) has become one of the most powerful tools to improve our understanding of the Earth System. It is routinely used to calculate model sensitivities to any model input, and to constrain numerical models using data assimilation techniques. If AD has been used by the ocean and atmospheric circulation modeling community for almost 20 years, it is relatively new in the ice sheet modeling community (e.g., [1]). The Ice Sheet System Model (ISSM) is a C++, object-oriented, massively parallelized, new generation ice sheet model that recently implemented AD to improve its data assimilation capabilities [2]. ISSM currently relies on Object Overloading through ADOL-C and AMPI. However, experience shows that Object Overloading AD on ISSM is significantly more memory intensive compared to the primal code. We want to investigate other AD approaches to improve the performance of the AD adjoint. Yet, to our knowledge, there is no source-to-source AD tool that supports C++.

To overcome this problem, we have developed a prototype of ISSM entirely in C, called *Boreas*, in order to test source-to-source transformation and compare the performance of these two approaches to AD. Boreas is a clone of ISSM, the main difference with ISSM is that all the objects are converted to C-structures and some function names have been adapted in order to be unique, but the code architectures are identical. The programming style of Boreas is a first attempt at defining a programming style of (or a sub-language of) C++ that source-transformation AD could handle. Boreas can be run in serial mode, or in parallel using MPI like ISSM. The adjoint of Boreas will also rely on AMPI for adjoint communication. To deal with parallel vectors and matrices and to solve linear systems, Boreas and ISSM rely on the Portable, Extensible Toolkit for Scientific Computation (PETSc).

The long term objective of this work is to use source-to-source transformation to differentiate ISSM. The first step, which we present here, consists in using Tapenade [3] to perform source-to-source transformation on Boreas, a C++-like C code. If Tapenade now officially supports C, differencing Boreas proved to be challenging. We present here some of these challenges and the strategies that we adopted to overcome them.

2 Extension of source transformation outside the differentiable code

File architecture of the source code, and in turn of the differentiated code, has been the first issue when applying AD to Boreas. Even though AD tools should ideally not be sensitive to the file architecture, we had to make some important changes in order for Tapenade to transform the primal code. For example, when the C source file is preprocessed before compilation (e.g. by `cpp`), it must also be preprocessed before differentiation. Consequently, the differentiated C code will be bound to one particular preprocessing output, coming from one set of preprocessing variable values. In other words, the differentiated C code will not contain `#ifdef` clauses, even if the source does. Similar issues apply to `#include` files. Even if we have reached today an acceptable strategy, it is still an open question whether the differentiated `include` files should contain definitions of differentiated objects only, or incorporate the original objects as well, and in this case should those be explicitly inlined or should they be added through an `#include` of the original `include` file.

We focus here on other issues that are more central to AD tool development. Namely, the automatic generation of a "calling context", the static analysis of destinations of pointers and its relation to aliasing issues.

2.1 Pointer aliasing

The calling context in which the differentiable code is used has a clear influence on differentiation results. The most obvious illustration is aliasing, i.e. the possibility that the same storage location is referenced by apparently different syntactic elements. Aliasing seriously restricts static code transformation. For instance, a possible aliasing between `X` and `Y` will forbid a loop nest parallelization tool to detect the code

```
for(i=2 ; i<size ; ++i) X[i] = Y[i-2];
```

as parallel, although without aliasing this is just a simple array copy.

*Corresponding Author, INRIA Sophia-Antipolis, `laurent.hascoet@inria.fr`
†University of California, Irvine, `mathieu.morlighem@uci.edu`

With former FORTRAN standards, aliasing was strongly discouraged, and AD tools often just required users to not use aliasing. The introduction of pointers in FORTRAN 90 makes this requirement more constraining. Forbidding aliasing becomes totally unrealistic in C, where aliasing is extremely common, if not encouraged. A source-transformation AD tool relies on accurate static data-flow analysis and, as such, needs a reliable detection of pointer aliasing. Tapenade is no exception and uses a static pointer destination analysis ("points-to" analysis) to determine whether two syntactic elements refer to the same memory location. This pointer analysis was already necessary to handle FORTRAN 90, and it becomes absolutely central for C/C++ [4]. The results of pointer analysis inform all following analyses of the function computation to handle correctly aliasing created during function computation.

However, a simulation code often separates the initialization phase that builds and initializes the data structures (e.g. mesh elements and the links between them) possibly setting aliasing at that stage, and the computation phase that takes values from and rewrites values into these data structures, implicitly relying on this aliasing. The computation phase is also generally followed by a post-processing phase. We refer to the union of the initialization and post-processing phases as the "context".

Typically, AD should only focus on the actual computation phase. For instance, the end user specifies that they want the derivatives of the function implemented by the root procedure of the computation phase, more precisely of the *dependent* variables, with respect to the *independent*, these variables being among the arguments (formal parameters or globals) of that computation root. One could think that only the function computation code need to be passed to the AD tool. However, in doing so, the AD tool will be blind to aliasing created during the initialization phase. Figure 1 illustrates this pitfall for the case of Boreas. The computation phase corresponds to the function `FemModelSolve`, with independents `X` and dependents `J`. The computation copies `X` into the `Inputs` component of the central object `FemModel`, then deals only with the values attached to the mesh elements to compute `J` without ever reading `FemModel->Inputs` explicitly. As a result, the AD data-flow analysis and in particular *activity* analysis finds that there is no differentiable link from `X` to `J`. The link appears only through aliasing of `FemModel->Inputs` with deep components of each of the mesh elements `Element->Inputs`, which is done during the initialization phase by the function `LinkDataSets` (which specifies that `Element->Inputs = FemModel->Inputs` for each element of the mesh). The issue can be solved in two ways:

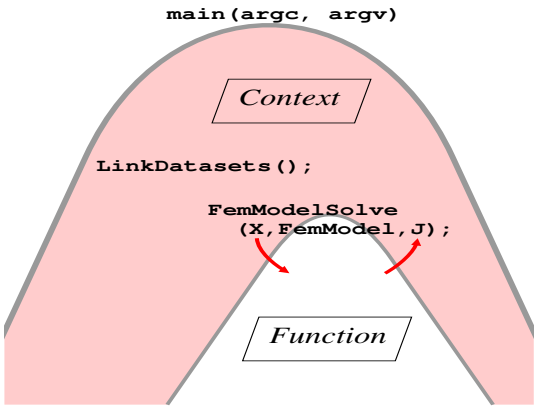

Figure 1: Call graph of the differentiable function inside its calling context call graph

- The AD tool can accept user directives that specify the existing aliasing upon entry of the computation root. Similarly, one can move the part of the initialization that creates the aliasing inside the computation phase. When running on the computation phase, the pointer analyzer then correctly detects the aliasing. Consequently, activity analysis detects the differentiable link from `X` to `J`. This is the solution we have implemented so far. However, it disturbs the original code either by adding a number of possibly complex directives or by moving a structural, non-differentiable piece of code that initializes the mesh structure at a place where it does not belong.

- One can extend the pointer analysis to the complete code, including the context. Aliasing information coming from the context would thus be exposed to AD additionally to aliasing coming from the computation code. This might imply some fine-tuning as the context contains more system calls, more problematic than what is usually found in the computation code, but this approach seems preferable in the long run. Still, this creates more complexity to the AD request: a given differentiation target (i.e. computation function plus dependent and independent variables) may produce different results for different contexts.

An immediate consequence of the second approach is that the complete simulation code must be exposed to the AD tool. Although this will certainly increase the memory size and run time of the AD tool, the benefits outweigh the cost. We will see in the next section that this is also beneficial to the initial execution, validation, and debugging of the differentiated code.

2.2 Generation of context code to call the differentiated code

The differentiated code of the computation root, obtained through AD, must be executed in an appropriate context. This context must call the differentiated code, providing it with the input derivative values in addition to the original inputs, and reading the output derivative values in addition to the original outputs. This context must also declare, allocate, and initialize the memory that hold these additional inputs and outputs, and must release this memory after differentiation. Although these tasks can be viewed as outside the realm of AD, it is not reasonable to leave them entirely to the end user. They are time-consuming, error-prone, and can be automated. Moreover, these tasks generally become manifest between the first successful run of the AD tool and the validation stage that should immediately follow. Starting the development of the calling context at that moment will handicap the user as they will lose the focus

on their primary objective. Finally, automated generation of this context can also automate the setup for derivatives validation.

Here, we have extended the generation of the differentiated code to also create a calling context for the actual derivative code. This extension is triggered by adding the single command-line option `-context` to the AD tool invocation. This extension requires that all the original files that define the calling context of the computation root, up to and including the `main` procedure, are passed to the differentiation command. In other words, in addition to the code of the differentiable function, that must be passed to the AD tool in all cases, one must pass the context code that prepares for and calls the differentiable function. As a result, a new "differentiated" context code is produced that prepares for and calls the differentiated function. The new context code follows very closely the structure of the original context, performing no derivative computation, as it is outside the call tree of the differentiable function. Still, the context code declares, allocates and initializes all the data structures that will later hold the derivatives, and propagates them to the entry of the differentiated function. Upon return from the differentiated function, the context code cleans up and releases these data structures. These creation and destruction operations take place by mimicking the corresponding operations on the original data structures. These differentiated data structures and differentiated variables follow Tapenade's *association by name* approach, but can be adapted quite easily to follow the alternative *association by address* where derivative containers are attached close to the original containers, deep at the level of the leaves of the data structures, therefore requiring no extra derivative variable names.

The main ingredient of this "context" functionality is a static data-flow analysis that runs over the complete code to find all allocation, initialization, and release operations of the original context that must have a differentiated counterpart. Insertion of appropriate declarations follows naturally from that. At each point in the code, we call **Req** the set of all variables for which the derivative variable is required downstream of that point, and that must therefore have been allocated and initialized upstream that point. The **Req** sets are computed by a data-flow analysis that runs backwards on the flow graph, i.e. in the direction opposite of execution. Similarly, we call **Avl** the set of all variables for which the derivative variables is available (i.e. allocated) and therefore for which the derivative variables must be released whenever they are released. The **Avl** sets are computed by a data-flow analysis that runs forwards on the flow graph, i.e. in the direction of execution. Like any data-flow analysis, both **Req** and **Avl** analyses deal with cycles in the flow-graph by running repeatedly until reaching a fixed point. They also propagate interprocedurally on the call graph, using a fixed point search to deal with recursive programs. On the call graph, both analyses first run a bottom-up sweep to compute summaries for each procedure, then run top-down, using these summaries when encountering a procedure call.

Let us focus here on the **Req** analysis, as the **Avl** analysis is straightforward. Regardless of the context functionality, the **Req** analysis is already needed for differentiation of the differentiable function code, because the AD model leads to introducing differentiated pointer variables. While activity analysis applies to variables of differentiable type, **Req** analysis extends it to pointers. An instruction I will be differentiated into some I' not only if its outputs are active but also if they intersect the **Req** set immediately after I. Only the differentiation rules are simpler. For instance, if I is the pointer arithmetic assignment:

```
p = &A[2] + offset;
```

and assuming that the derivative of `p` is required downstream, one must generate the "derivative" assignment I':

```
p' = &A'[2] + offset;
```

where `p'` and `A'` are the differentiated variables of `p` and `A`. This defines the required `p'`, and in turn requires anterior definition of `A'`. Therefore, the data-flow equation of the **Req** analysis (backwards) across a statement I computes **Req** before I (noted $\mathbf{Req}^-(I)$) from **Req** after I (noted $\mathbf{Req}^+(I)$) as:

$$\mathbf{Req}^-(I) = \begin{cases} (\mathbf{Req}^+(I) \setminus \mathbf{kill}'(I')) \cup \mathbf{use}'(I') & \text{if } \mathbf{out}(I) \text{ is active or if } \mathbf{out}(I) \cap \mathbf{Req}^+(I) \neq \emptyset \\ \mathbf{Req}^+(I) & \text{otherwise} \end{cases}$$

where $\mathbf{use}'(I')$ (resp. $\mathbf{kill}'(I')$) is the set of variables whose derivative variable is used (resp. totally overwritten) by the derivative instruction I' of I.

The novelty brought by the "context" functionality is that the **Req** and **Avl** analyses are now run on the context code as well. As there is no activity in these regions of the code, the data-flow equation above applies only when $\mathbf{out}(I) \cap \mathbf{Req}^+(I) \neq \emptyset$, in other words, when the statement defines or modifies a pointer that may eventually be used in the differentiated part. The code generated for the context part is essentially the original code where every declaration, initialization, and propagation of a variable belonging to **Req** is followed by the same operation on the derivative variable. Similarly, propagation and release of a variable belonging to **Avl** is followed by the same operations on the derivative variable.

2.3 Validation and semi-automatic debugging of the differentiated code

When combined with the `-debugTGT` or the `-debugADJ` option of Tapenade, the context extension also adds into the differentiated context the infrastructure to validate the derivatives and to narrow down the faulty instructions if the derivatives are not valid. Here, we only discuss the validation of the tangent code.

In the differentiated context code, the call to the differentiated function is instrumented to implement two behaviors, depending on the value of a Unix shell variable. In the first behavior, the context perturbs the input of the differentiated function as $X + \epsilon\dot{X}$ for each independent input X, with a random $\dot{X}$. In the second behavior, the differentiated function

is given the normal input X and the same random $\dot{X}$ for the tangent derivative. In the derivative of the function computation part, test points are inserted by default upon entry into and exit from every procedure call. More test points can be inserted by the end user through directives. At each test point, the code that follows the first behavior writes into a FIFO file the value of the perturbed value for each active variable. Meanwhile for the second behavior, and for each active variable, the code reads the perturbed value from the FIFO file, computes the classical finite difference and compares it with the tangent derivative. Thus, running in parallel two instances of the differentiated executable, one for each behavior, executes the finite differences test to validate the tangent code. In case of an error, the test points prove very useful to narrow down the wrongly differentiated code.

3 Results and further work

We validate our tangent derivatives by comparison with centered finite differences with a well chosen ϵ. We consistently obtain seven matching digits, which is satisfactory for a computation on `double` values. Some C primitives (e.g. `memcpy`) are "differentiated" by hand, as well as, classically, the calls to the linear solver. The tangent code runs approximately 1.7 times slower than the primal code.

To highlight the benefits of the AD tool's activity analysis, we observe that the number of assignments in the tangent code is higher than in the original code by only 50%: many assignments on real values are detected as passive. To visualize this in another way, figure 2 shows the nesting of the original data structures, highlighting the only components that need a derivative in red. These are the only components that need to be computed, and that need storage space to hold the derivatives.

We have now started the next step, which is to use reverse AD to create the adjoint of Boreas. The programming style of ISSM and Boreas uses many intermediate arrays that are allocated, then released quickly after. This will raise the problem of storage of addresses in the adjoint code, i.e. addresses stored might point to released memory and thus may be no longer valid. To address this issue, we will use the experimental ADMM library, developed jointly with Argonne. It will then be meaningful to compare the overhead of overloading-based adjoint code of ISSM with the overhead of a source-transformation-based adjoint of Boreas, which might suffer from this intensive use of dynamic memory management. In particular, it is still unclear what the most efficient design options for ADMM should be. For instance, should ADMM recompute stored addresses when their underlying memory chunks were released then re-allocated at a different location, or should it implement its own `malloc` to make sure that re-allocating returns the same memory chunk? Only experiments on a large code such as Boreas/ISSM will tell us which option is best.

```
struct FemModel
  - ...
  - Elements *elements;
    - ...
    - Tria **elements;
      - Node *nodes[3];
        - ...
        - double *svalues;
      - Vertex *vertices[3];
        - ...
        - double x, y;
      - Parameters *parameters;
        - ...
      - Inputs *inputs;
        - int interpolation[];
        - double *inputs[];
  - Vertices *vertices;
    - Vertex **vertices;
  - Nodes *nodes;
    - ...
    - Node **nodes;
  - Inputs *inputs;
  - Parameters *parameters;
    - ...
  - Results *results;
    - ...
  - FILE *outputfile;
```

Figure 2: The main data structure, with its differentiated parts

References

[1] Patrick Heimbach and Veronique Bugnion. Greenland ice-sheet volume sensitivity to basal, surface and initial conditions derived from an adjoint model. *Ann. Glaciol.*, 50(52):67–80, 2009.

[2] 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). *Cryosphere*, 8(6):2335–2351, 2014.

[3] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39(3), 2013.

[4] V. Pascual and L. Hascoët. TAPENADE for C. In *Advances in Automatic Differentiation*, Lecture Notes in Computational Science and Engineering, pages 199–210. Springer, 2008. Selected papers from AD2008 Bonn, August 2008.