Optimization Methods and Software

ISSN: 1055-6788 (Print) 1029-4937 (Online) Journal homepage: http://www.tandfonline.com/loi/goms20

Source-to-source adjoint Algorithmic Differentiation of an ice sheet model written in C

L. Hascoët & M. Morlighem

To cite this article: L. Hascoët & M. Morlighem (2017): Source-to-source adjoint Algorithmic Differentiation of an ice sheet model written in C, Optimization Methods and Software, DOI: 10.1080/10556788.2017.1396600

To link to this article: http://dx.doi.org/10.1080/10556788.2017.1396600

Published online: 08 Nov 2017.

Optimization Methods & Software, 2017
https://doi.org/10.1080/10556788.2017.1396600

Source-to-source adjoint Algorithmic Differentiation of an ice sheet model written in C

L. Hascoët[a*] and M. Morlighem[b]

[a]*Université Côte d'Azur, INRIA, Sophia-Antipolis, France;* [b]*Department of Earth System Science, University of California, Irvine, CA 92697-3100, USA*

(*Received 6 January 2017; accepted 19 October 2017*)

Algorithmic Differentiation (AD) has become a powerful tool to improve our understanding of the Earth System, because it can generate adjoint code which permits efficient calculation of gradients that are essential to sensitivity studies, inverse problems, parameter estimation and data assimilation. Most source-to-source transformation tools, however, have been designed for FORTRAN and support for C remains limited. Here we use the Adjoinable Land Ice Flow model (ALIF), a C clone of the C++ Ice Sheet System Model (ISSM) and employ source-to-source AD to produce its adjoint code. We present the first running source-to-source adjoint of ALIF, and its application to basal drag inversion under Pine Island Glacier, West Antarctica. ALIF brought several challenges to AD tool development, such as the correct treatment of the context code, which does not compute the differentiable function, but controls this computation through the setup of data structures, including possible aliasing, as well as data-flow reversal in the presence of pointers and dynamic memory, which are ubiquitous in codes such as ISSM and ALIF. We present the strategies we have developed to overcome these challenges.

Keywords: Ice sheet model; ISSM; Algorithmic Differentiation; adjoint methods; dynamic memory; Tapenade

AMS Subject Classification: 65K; 65H

1. Introduction

Algorithmic Differentiation (AD) has become a powerful tool to improve our understanding of the Earth system. It is 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 modelling community for almost 20 years, it is relatively new in the ice sheet modelling community [4,9]. The Ice Sheet System Model (ISSM) is a C++, object-oriented, massively parallelized, new generation ice sheet model that recently employed AD to improve its data assimilation capabilities [10]. ISSM currently provides gradient calculation using either of two methods:

- a 'manual' adjoint that computes the adjoint state from control theory, with its own discretization and implementation. Maintenance is obviously an issue there.

*Corresponding author. Email: laurent.hascoet@inria.fr

- an AD-based adjoint by Operator Overloading through Adol-C [18] and the 'Adjoinable MPI' AMPI [17]. However, Operator-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-generated 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 the *Adjoinable Land Ice Flow model (ALIF)*, in order to test source-to-source transformation and compare the performance of these two approaches to AD. ALIF is a clone of ISSM, the main difference with ISSM being that all objects are converted to C structures and some function names have been adapted in order to be unique, as C does not support overloaded functions. The code architectures are identical. Like ISSM, ALIF can be run in serial mode or in parallel using MPI. For adjoint communication in the parallel mode of ALIF, our choice is also to rely on AMPI as calls to AMPI are generated automatically by Tapenade. Likewise, to deal with parallel vectors and matrices and to solve linear systems, ALIF and ISSM rely on PETSc (the Portable, Extensible Toolkit for Scientific Computation).

The programming style of ALIF is a first attempt at defining a programming style of (or a sub-language of) C++ that source-to-source AD could handle. ALIF is designed as a C++-like C code. In other words, keeping in mind that one distant objective of source-to-source AD is to address C++ and therefore ISSM, the first step, which we present here, investigates source-to-source of a realistic C code that exhibits many of the difficulties that we foresee in C++. In particular, ALIF preserves the intensive use of dynamic memory of ISSM, in ways that are uncommon in pure C. Aliasing is also used extensively, making the code analysis challenging. We use Tapenade [7] to perform this source-to-source transformation on ALIF. Even though Tapenade officially supports C, differentiation of ALIF proved to be challenging. We present some of these challenges and the strategies that we adopted to overcome them.

The goal of this work is to produce an adjoint of ALIF by source-to-source AD, then to exercise this adjoint on a data assimilation problem on Pine Island Glacier, in West Antarctica. Section 2 focuses on the tangent-linear differentiation and discusses issues related to aliasing and to the sophisticated data structures in ALIF. Section 3 focuses on a specific challenge of the adjoint, which is the restoration of addresses needed by the control- and data-flow reversal. After an automated validation step described in Section 4, we discuss in Section 5 a realistic numerical experiment, where the AD-generated adjoint is used in a data assimilation problem. We give numerical results as well as an estimate of the costs and benefits of our AD adjoint. Finally, we discuss in Section 6 future research directions to further improve the performance of the address restoration mechanism.

2. Extending source transformation outside the differentiable code

The file architecture of the source code, and in turn of the differentiated code, has been the first issue when applying AD to ALIF. Even though AD tools should ideally not be sensitive to the file architecture, we had to make some 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 is 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. It seems this is the only approach that works in general, since the preprocessor clauses can be placed anywhere and sometimes do not even follow the syntactic nesting of constructs.

However, in the particular case of `#include` clauses and provided the include file does not contain executable code, we can often do a slightly better job placing corresponding `#include` clauses back into the differentiated code. We introduce the notion of a differentiated include file, which holds the declarations from the original include file plus the additional differentiated declarations, and differentiation produces these differentiated include files as well. Even if we have reached today an acceptable solution on this cosmetic issue, some marginal changes might still occur about the nesting level of include files. Consider an include file `incl.h`. Its differentiated `incl_d.h` currently contains the contents of `incl.h`. To reduce code duplication, `incl_d.h` could instead include `incl.h`.

We focus here on other issues that are more relevant to AD tool development. Namely, the automatic generation of a 'calling context', the static analysis of destinations of pointers and 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 any 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.

With former FORTRAN standards, aliasing was strongly restricted. Aliasing two formal arguments of a procedure is permitted when only one alias is possibly overwritten. However, even this restricted aliasing is adverse to adjoint AD as the adjoint of a read-only variable is in general an overwritten variable and therefore the adjoint code would violate the standard. Consequently, AD tools often just require users not to use aliasing [8]. The introduction of pointers in FORTRAN 90 opens the door to more aliasing, making this no-aliasing requirement more problematic today. Forbidding aliasing becomes totally unrealistic in C, where aliasing is extremely common. A source-to-source 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 may refer to the same memory location. This pointer analysis was already necessary to handle FORTRAN 90, and it becomes absolutely central for C/C++ [15]. The results of pointer analysis inform all following analyses in the AD work flow, so they can correctly handle aliasing created during function execution.

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*, and to the computation phase as the *Function* (see Figure 1). The users of AD typically specify that they want the derivatives of the mathematical function implemented by some procedure of the code. This is by definition the root procedure of the *Function*. The users may specify in addition that they want the derivatives of this function's *dependent* variables with respect to its *independent* variables, these two sets of variables being among the arguments (formal parameters or globals) of that computation root. Although one might think at first sight that only the call graph of the *Function* need to be passed to the AD tool, this would make the

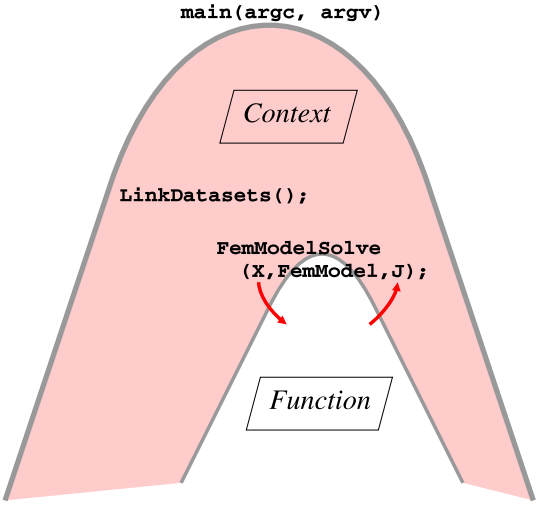

Figure 1. Call tree of the differentiable *Function* inside the call tree of its calling *Context*.

AD tool blind to aliasing created during the Context phase. Figure 1 illustrates this pitfall for the case of ALIF. The computation root is the function `FemModelSolve`, with independents `X` and dependents `J`. The *Function* itself copies `X` into the `Inputs` component of the 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 by calling `LinkDataSets` (which specifies that `Element->Inputs` = `FemModel->Inputs` for each element of the mesh). The call to `LinkDataSets` occurs in the *Context* phase, i.e. outside of the *Function* phase. This issue can be solved in two ways:

- 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 in addition to aliasing coming from the computation code. This approach fits with the view that AD should be provided with the complete code and not only with its computation phase. This might imply some fine-tuning as the *Context* contains more system calls, more problematic than what is usually found in the *Function*, but this approach seems preferable in the long run. Still, this has the subtle effect that the existing context strongly influences differentiation of the *Function*: the same differentiation target (i.e. computation function name plus dependent and independent variables) given by the end-user may produce radically different results in different contexts.
- The AD tool can still be run on the *Function* code only, with user directives that specify the existing aliasing upon entry into the *Function*. Similarly, one can move the part of the initialization that creates the aliasing inside the *Function* phase. Then the pointer analyzer can correctly detect and propagate 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.

The first approach requires that the complete simulation code is exposed to the AD tool. Although this certainly increases 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 holds these additional inputs and outputs, and must release this memory after differentiation. Although these tasks can be considered 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, users often postpone these tasks until they become unavoidable, which is 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 users 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 generated, which sets up all required data structures and calls the differentiated function. The new context code follows 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 mimic the corresponding operations on the original data structures. With this process, the pointer chains of the original data structures and the resulting aliasing are naturally reflected in the pointer chains of the differentiated data structures. These differentiated data structures and differentiated variables follow Tapenade's *association by name* approach, but can be adapted easily to follow the alternative *association by address* (see [1]) 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 are available (i.e. allocated). When a variable that belongs to **Avl** is released, its derivative variable must be released too. 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. 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

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

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

The specific differentiation rules for instructions whose output are in **Req** are quite simple. For instance, if I is some 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` (actual C syntax will obviously require another naming convention). This defines the required `p'`, and in turn requires anterior definition of `A'`. Similarly, in the case of memory allocation, if I is

```
A = (MyStruct*)malloc(n * sizeof(MyStruct));
```

and assuming that the derivative of `A` is required downstream, one must generate the additional allocation I':

```
A' = (MyStruct'*)malloc(n * sizeof(MyStruct'));
```

where `MyStruct'` is the 'derivative' type of `MyStruct`, built automatically from `MyStruct` by removing all fields containing values that never have derivatives, and by recursively replacing fields of structured type with the corresponding derivative type.

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 be used in the differentiated part. The code generated for the context part is essentially the original code where every declaration, allocation, initialization, and propagation of a variable belonging to **Req** is directly followed by the same operation on the derivative variable. Similarly, propagation and release of a variable belonging to **Avl** is followed by the same operation on the derivative variable.

3. Restoring addresses during the adjoint

Section 2 dealt with issues that show up in any mode of AD. On the other hand, the issue of address restoration arises only for adjoint AD. Classically, the code produced by adjoint AD propagates the derivative of the (scalar) output with respect to the intermediate variables at each location in the code. Therefore, this propagation must be done backwards with respect to the original code's execution order. In addition to that, the derivative computations may use elements of the original computation such as intermediate values, indices, or addresses. Consequently, an adjoint code typically consists of two successive sweeps:

(1) a forward sweep, that executes the original program and stores the elements computed during this execution that will be needed during the derivatives computation, and
(2) a backward sweep, that computes and propagates the derivatives, in an order inverse to the original program, and retrieves the elements it needs from where the forward sweep stored them.

Some AD models save a portion of this storage at the cost of recomputation [2], but this recomputation should start from a saved state, and efficient AD of large codes usually requires to save many of these re-starting states. Therefore the storage issue remains, merely shifted to a different time. Obviously, the best structure to hold the stored elements is a stack. Consider for example instruction `y = y*x;`. The forward sweep will in general contain the sequence

```
push(y); y = y*x;
```

that saves `y` before it is lost by overwriting, whereas the backward sweep will contain

```
pop(&y); x' = x'+y*y'; y' = x*y';
```

that retrieves `y` before using it in the derivative instructions. Adjoint variables `x'` and `y'` hold the derivatives of `x` and `y`. These adjoint derivative instructions, and the related storage issues, are discussed in [6].

The elements that need to be stored are not necessarily limited to numerical values, they also include memory addresses. Consider the original instruction `*z = sin(*p);` which involves pointers `z` and `p`. By applying the classical adjoint differentiation rules, the backward sweep will contain the sequence of derivative instructions:

```
*p' = *p'+cos(*p)*(*z'); *z' = 0.0
```

in which we notice the use of addresses `p`, `p'`, and `z'`. If, later in the forward sweep, the address contained in `p` is overwritten, then it must be stored before this overwrite, then restored during the backward sweep before it is used. In addition to that, the addresses in `p'` and `z'` must be managed by the backward sweep to maintain semantic correspondence between `z` and `z'`, and between `p` and `p'`. For the sake of simplicity, we will focus in the remainder of this article on the issue of restoring `p`. Restoring `z'` and `p'` is done in a similar manner.

We also note that `p` contains an address, either in the stack or in the heap, of a memory chunk that may be of limited time-span. It may point inside a local variable of some procedure, that will be freed at procedure exit, or inside dynamically allocated memory, which may be freed later in the forward sweep. Although the address in `p` is easy to store and restore, and although the backward sweep can organize corresponding re-allocation of the freed objects, there is no guarantee that the re-allocated objects lie at the same addresses in memory. The restored addresses may then be wrong and will cause unpredictable behaviours.

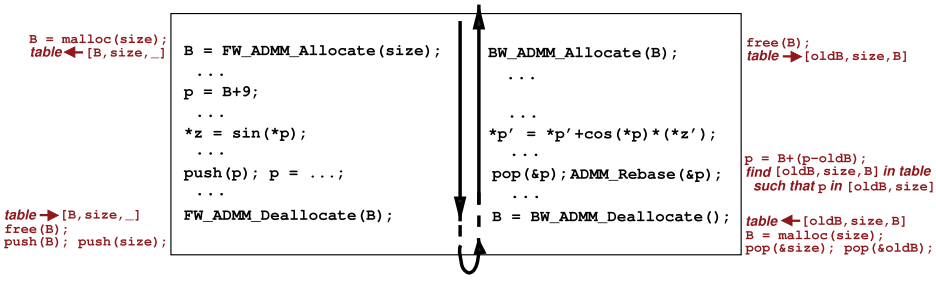

Figure 2. The elementary stitch of adjoint address management. Forward sweep on the left is executed downwards, backward sweep on the right is executed upwards, so that original instructions and adjoint instructions face each other. Execution detail of each ADMM library call is sketched in red.

Our solution to this issue was developed jointly with Argonne National Laboratory. It has been implemented into a library called ADMM for 'Adjoinable Dynamic Memory Management', which can be invoked from the adjoint code created by Tapenade as well as by OpenAD [16]. The idea is to dynamically track the base address of every chunk of memory created during the forward sweep, then on the backward sweep track the base address of every re-allocated chunk, thus maintaining a correspondence from old to new memory chunks. Given this correspondence, every address restored from the stack can be 'rebased', i.e. modified so that it points towards the new chunk instead of the old. Figure 2 illustrates the mechanism on the elementary stitch of dynamic memory management where a memory chunk is allocated, then a pointer points to it, is used in some differentiable instruction, later is overwritten or falls out of scope, and the memory chunk is finally freed. The adjoint code behaves as follows:

- During the forward sweep (left of Figure 2), one task is to track memory chunks: upon dynamic memory allocation, the memory chunk's base address `B` and size are stored into a dedicated table. Upon deallocation, both base address and size are retrieved (and removed) from the table, then are pushed onto the AD stack. Note that this pushes only two integers, and does not push the contents of the deallocated memory chunk. In addition, each time an address (e.g. `p`) is overwritten, and the AD tool has detected that this address is needed in the backward sweep, then the address is pushed onto the AD stack.
- During the backward sweep (right of Figure 2), tracking maintains correspondence: the backward correspondent of the `free` retrieves the old base `oldB` and size from the AD stack and re-allocates a new chunk of same size, yielding a new base address `B`. Old and new bases, plus sizes, are stored again into the table. Similarly, this new table entry will be removed by the backward correspondent of the original allocation. In between, each time an address is restored from the AD stack, the table is searched to look for the old chunk that contains it, and the address is updated to point to the corresponding new chunk instead.

Since it may happen that the overwrite of `p` occurs *after* the freeing of `B`, the backward sweep may want to rebase `p` *before* the new `B` is known. Therefore, an internal waiting list is used to make sure that `p` gets rebased as soon as the backward deallocation `BW_ADMM_Deallocate` determines the new `B`. This behavior is implemented in ADMM with a handful of library primitives. Namely, the forward and backward counterparts of `malloc` and `free`, plus the pointer rebasing operation. For clarity, Figure 2 shows only the actions that ensure a correct value in `p`. Other actions that, for instance, store and restore the contents of memory chunk `B`, may be present if needed but are not shown.

Both Tapenade and OpenAD [16] generate adjoint code that makes calls to the ADMM library. This approach was already successfully tested on three small- to medium-size applications, one of them with both AD tools. ALIF code is one order of magnitude larger, and with a coding

style 'inherited' from C++, which is definitely challenging. In particular, ALIF completely disconnects allocation/deallocation pairs from the syntactic structure of the code: contrary to recommended coding style for AD [8], there is no guarantee that the routine that does the `malloc` also does the `free`. Quite the contrary in fact, often a memory chunk is freed just before a newly allocated one is about to replace it. Being itself disconnected from the syntactic structure, ADMM was able to handle the dynamic memory management of ALIF, so that Tapenade could produce a correct adjoint code.

4. Validation and overhead analysis

The `-context` extension of Tapenade automates the process of validation of the differentiated code. Every call to the differentiated function gets surrounded by special initialization and termination code. To validate tangent differentiation, the tangent differentiated code is executed in two different manners, according to the value of some conventional Unix environment variable. In the first manner, initialization modifies the vector of inputs X by adding to it some $\epsilon\dot{X}$, where ϵ is user-defined, and $\dot{X}$ is a pseudo-random vector determined by a user-given seed. The termination code computes and displays the dot product of the resulting vector of outputs, *Yeps*, with another pseudo-random vector $\bar{Y}$. In the second manner, initialization leaves X unmodified and provides the same $\dot{X}$ as the tangent derivative of X. The termination code computes and displays on the one hand the dot product of the resulting vector of outputs, Y, with the same pseudo-random $\bar{Y}$, and on the other hand computes and displays the dot product of the resulting tangent derivative of Y, called $\dot{Y}$, with the same $\bar{Y}$. The tangent code passes the test if

$$\frac{(\bar{Y} \mid Yeps) - (\bar{Y} \mid Y)}{\epsilon} = (\bar{Y} \mid \dot{Y}).$$

To validate adjoint differentiation, the adjoint differentiated code is executed only once. The initialization code sets $\bar{Y}$ to the same pseudo-random vector, and the termination code computes and displays the dot product of the resulting adjoint derivative of X, called $\bar{X}$, with the same pseudo-random $\dot{X}$. The adjoint code passes the test if

$$(\bar{Y} \mid \dot{Y}) = (\bar{X} \mid \dot{X}).$$

On the particular test case that we used, we obtained:

$$\begin{aligned}((\bar{Y} \mid Yeps) - (\bar{Y} \mid Y))/\epsilon &= 1.391674587\\ (\bar{Y} \mid \dot{Y}) &= 1.3916746070155939\\ (\bar{X} \mid \dot{X}) &= 1.3916746070155936\end{aligned}$$

Before discussing performance, let us point out that the derivative code for external primitives (e.g. `memcpy`) as well as derivative code for the linear solver were replaced by hand-written optimized code [3]. Performance is in par with what is generally expected from source-to-source AD: compared to the run-time of the primal, non-differentiated code, execution of the tangent differentiated code is an average of 1.6 times slower (1.5 on another test case), and execution of the adjoint differentiated code is an average of 4.3 times slower (2.6 on another test case). In our application, the derivative that we need is the gradient of a scalar cost function J with respect to a vector of inputs X. Since it takes only one execution of the adjoint code to produce

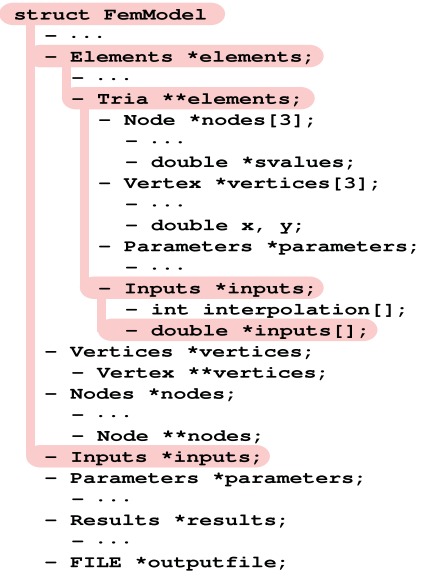

Figure 3. The main data structure, with its differentiated parts.

the complete $\mathrm{d}J/\mathrm{d}X$, whereas this would take n executions of the tangent code (where n is the length of X), these figures demonstrate again the superiority of adjoint AD for optimization and inverse problems.

The 1.6 slowdown ratio for the tangent code simply reflects the presence of additional instructions that compute the derivatives, accounting for the extra 0.6. This is quite good compared to 'theoretical' estimates that generally give an upper bound of this slowdown between 3 and 5, depending on the approximation chosen to estimate run-time. We believe that one reason for this good ratio is the effectiveness of activity analysis: the better the activity analysis, the fewer original instructions need to be differentiated. A poorer activity analysis will lead to more derivative instructions, which, at run-time, will in turn lead to computing and propagating zero-valued derivatives. To demonstrate this, we have counted the number of assignments on `double` values in the primal source code, and in the tangent differentiated source code. The latter is only 50% higher, reflecting that many assignments on real values are detected as passive. Figure 3 shows another sign of the benefit of activity analysis: it shows the structure of the `FemModel`, which holds all the data related to the simulated test case. Highlighted in red are the only components for which activity analysis, together with **Req** analysis, detected that a corresponding derivative component must be created. One can see that the size of the structure holding the derivative is significantly smaller.

The 4.3 slowdown ratio (2.6 on another test case) for the adjoint does not correspond a priori to additional computations. In fact, one run of the adjoint code performs roughly the same operations as one run of the tangent code, only in a different order and on different inputs. The additional cost comes from orchestrating the control flow a second time for the backward sweep, mainly implying storing and restoring the needed intermediate values through the AD stack. However, the use of the ADMM library for addresses adds its specific overhead. The most part of the run-time cost in using ADMM comes from the rebasing operations, done during `ADMM_rebase` and also during `BW_ADMM_Deallocate` for the delayed rebase's placed in the waiting list: the expensive part is to look up in the current list of memory chunks for the one that contains a given address. The following table splits up the adjoint run-time (approximately) into its components namely: the primal computations (normalized to one), the derivative

computations, the control-flow and data-flow reversal including the `push` and `pop` of intermediate values, and the cost associated with ADMM address management. We did this for two test cases, with different cost functions. These figures strongly depend on the test case. However, they show that the address lookup cost is significant and could benefit from an adapted algorithm such as a binary search tree instead of the present plain ordered list. On one test case the chunks table reached a peak of 1282 chunks, and the address rebasing operation itself was called 12,694 times.

	Primal	*Derivatives*	*Flow reversal*	*Addresses*	*Total*
Test case 1	1	0.47	0.27	0.87	2.61
Test case 2	1	0.61	0.09	2.56	4.26

5. Application to basal friction inversion on Pine Island Glacier

We apply here ALIF and its adjoint capability to one of the most common inverse problems in ice sheet modelling: inferring basal friction from satellite-derived surface velocities (e.g. [12]). Ice is modelled as an incompressible viscous fluid that deforms under its own weight, that can be approximated by a Stokes flow:

$$\nabla \cdot \boldsymbol{\sigma} + \rho\, \mathbf{g} = \mathbf{0}, \quad \nabla \cdot \mathbf{v} = 0 \tag{1}$$

where $\boldsymbol{\sigma}$ is the Cauchy Stress tensor, ρ is the ice density, $\mathbf{g}$ is the acceleration due to gravity, and $\mathbf{v}$ is the ice velocity. Here, we employ an approximation of these equations that is known as the Shelfy Stream Approximation (SSA) [11]. The velocity of the ice is the solution of an elliptic problem that consists of two equations in the horizontal plane. One of the model inputs required to solve for the ice velocity is the basal friction coefficient, α, that governs the amount of stress exerted by the bedrock on the ice. Since basal friction cannot be measured remotely, it is generally inferred from surface velocities derived from satellite observations [12]. Namely, we find the pattern of basal friction by minimizing the following cost function [14]:

$$\begin{aligned}\mathcal{J}(\boldsymbol{v}, \alpha) = {} & \gamma_1 \frac{1}{2} \int_\Omega (v_x - v_x^{\text{obs}})^2 + (v_y - v_y^{\text{obs}})^2 \, \mathrm{d}\Omega \\ & + \gamma_2 \frac{1}{2} \int_\Omega \ln \left(\frac{\sqrt{v_x^2 + v_y^2} + \varepsilon}{\sqrt{{v_x^{\text{obs}}}^2 + {v_y^{\text{obs}}}^2} + \varepsilon} \right)^2 \mathrm{d}\Omega \\ & + \gamma_t \frac{1}{2} \int_\Omega \nabla\alpha \cdot \nabla\alpha \, \mathrm{d}\Omega \end{aligned} \tag{2}$$

where Ω is the two-dimensional model domain, (v_x, v_y) and $(v_x^{\text{obs}}, v_y^{\text{obs}})$ are the modelled and measured surface velocities respectively, ε is a minimum velocity used to avoid singularities, and γ_i are non-dimensionalizing weighing constants. The first two terms are the classical $\mathcal{L}^2$ misfit and a logarithmic misfit, respectively, and the third term is Tikhonov regularization term, which penalizes uncontrolled oscillations of α and stabilizes the inversion.

Here, we apply this inverse method to Pine Island glacier, one of the major ice streams of the

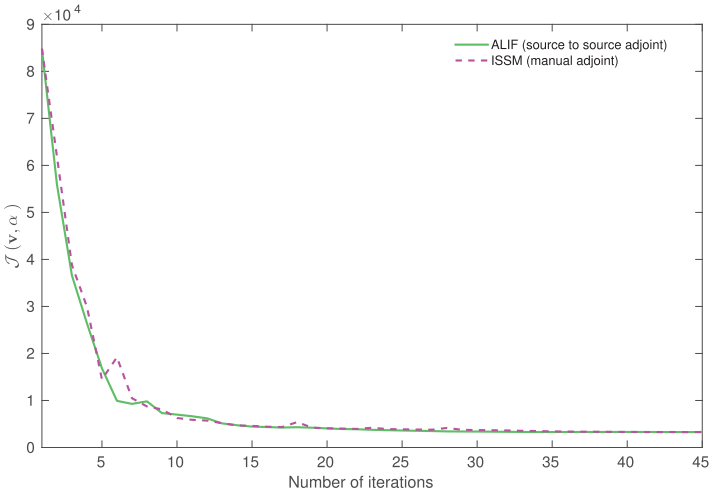

Figure 4. Convergence of the optimization using ISSM (magenta dots) and ALIF (solid green).

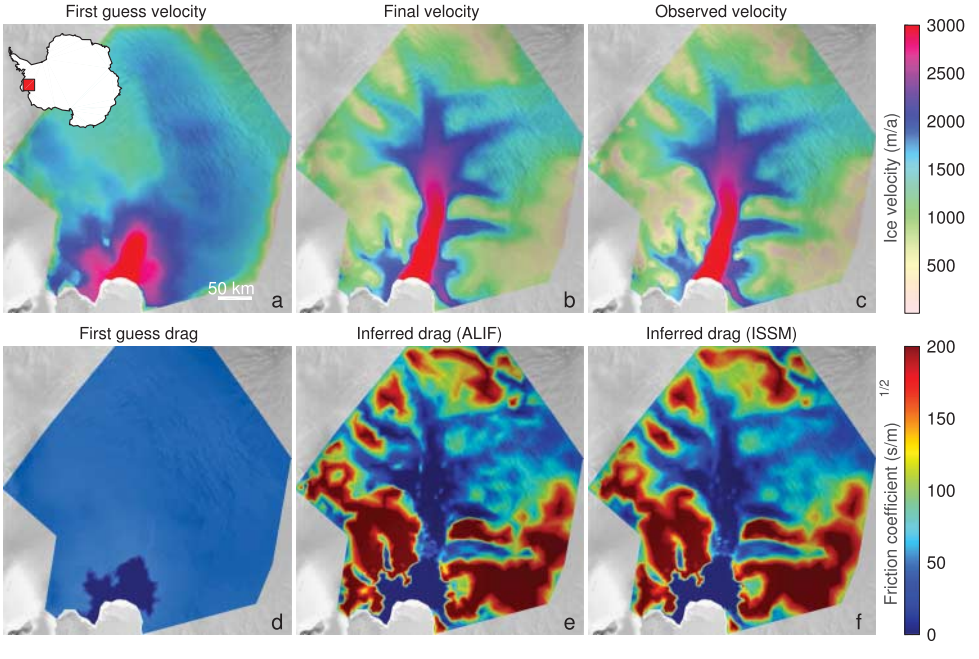

Figure 5. First guess for velocity (a) and for basal friction (d). Final modelled velocity (b) after optimization of the basal friction from ALIF (e). Observed velocities (c). Inferred basal friction (f) using ISSM and a manual adjoint.

Antarctic Ice Sheet that has been experiencing dramatic changes for the past four decades. We use a similar model as the one described in [13]. Our triangle mesh comprises 2700 elements and the SSA equations of stress balance are solved using linear P1 finite elements.

Figure 4 shows the rate of convergence of the cost function from ALIF, that relied on the adjoint generated by Tapenade, as well as ISSM's manual adjoint (that does not rely on Adol-C).

Figures 5(a,d) shows the velocity and basal friction coefficient for the first guess of the inverse problem. Figure 5(b) shows the modelled velocity after the inversion, that is in excellent agreement with the measurements (Figure 5(c)). Figure 5(e,f) shows the inferred basal friction coefficient from ALIF, using that adjoint code generated from Tapenade, and from ISSM, using

a manual adjoint. We observe that both fields converged to a similar solution that is comparable with previously published results [13].

6. Outlook

We have presented the main Algorithmic Differentiation issues that we faced while producing an adjoint for the ice-sheet model ALIF, a clone of ISSM in plain C that still exhibits a strong relationship with the original C++ code, with similar structures and coding style. We have discussed developments made to overcome these issues, in particular the automatic generation of context code that prepares for the call to the adjoint code, and the automated management of addresses and dynamic memory through a library called ADMM. This work certainly does not promote the strategy of rewriting a code from C++ to C: we see it instead as a proof of concept for future Source-Transformation AD of C++, as C++ code intensively uses the problematic usage patterns that we are addressing here. It is informative to compare the overhead of the existing Adol-C Operator-Overloading-based adjoint code of ISSM with the overhead of the present Source-Transformation-based adjoint of ALIF. Unfortunately a detailed run-time comparison is still not possible at present, in part because the current Adol-C adjoint (temporarily) involves some manual steps. Roughly, run-time of the Tapenade adjoint is at least two to three times shorter than the Adol-C adjoint. This is a significant improvement, but still not as good as what was observed on other applications. Source-Transformation AD, when applicable, can often claim larger improvements compared to Operator-Overloading AD. Our first measurements suggest that the overhead incurred by our management of addresses and of dynamic memory explains this not-as-good speed-up. We believe we can reduce this overhead through data-flow analysis detecting that some pointer operations do not need this address management.

More importantly the comparison on memory usage is clearer: the size of the 'tape' used by the Adol-C adjoint is about one order of magnitude larger than the stack size used by Source-Transformation adjoint. On our test case and for one gradient computation, Adol-C builds a tape of 5.5 Gb, whereas the Tapenade adjoint uses a peak stack size of 1.1 Gb. As we did not yet apply the usual customizations of Tapenade differentiation (adaption of the checkpointing strategy, detection of parallel loops ...), we believe we can reduce this peak size further. The observed memory overhead incurred by the ADMM mechanism is negligible.

On the other hand, the manual adjoint used by ISSM does not lend itself to performance comparison with AD adjoint. The manual adjoint approach consists in deriving the adjoint equations, then to discretize and solve them in a new code. There are similar storage issues since the manual adjoint uses some primal values in reversed order. The needed primal values are stored and the manual adjoint retrieves them. However, only those values are concerned: the control-flow decisions, the dynamic memory management, and the management of pointers result from implementation choices that are specific to the adjoint solver and do not necessarily mimic the choices made in the primal solver.

It turns out that the test problem we have chosen repeatedly allocates and deallocates memory, making it a good test-bed for the ADMM mechanism. It is true that this problem is steady-state, whereas time-stepping is another case where dynamic memory is used intensively. In the present application, this dynamic memory management occurs at a deeper level. Vectors and matrices prepared for the solver are systematically replaced with new copies, and old copies are deallocated, thus mimicking the behaviour of the original C++ code. Test cases for future applications should also include a time-stepping phase. In particular, this would allow us to compare the results with previous work on time-evolving simulations [5,10].

We proposed a working solution to the problem of the adjoint of dynamic memory. Still, this leaves several questions open, which we will describe from the angle of the so-called *save-on-kill*

approach. The save-on-kill approach (used by Tapenade) chooses to save a value, in provision for the future derivative computations, only when this value is about to be killed by some overwriting. This applies in particular to addresses in pointers. However, it may happen that this address is always used/dereferenced after receiving a certain offset, whereas the address itself does not point to allocated storage. This sort of contrived code would make our approach fail. We could mention other similar situations, hopefully quite rare, where the 'too late' save-on-kill will cause problems when rebasing the saved address. One possible strategy is to replace save-on-kill for pointers with *save-on-use*, which saves the address that is effectively dereferenced, repeatedly for each instruction that uses it. This strategy would fit OpenAD well, as it already applies save-on-use for the partial derivatives of differentiable instructions (but not for addresses). Another possible strategy could be that each pointer carries along a unique identifier of the memory chunk it points into. This is demanding in terms of AD tool development, but it would solve the problem. Moreover, this strategy would eliminate the need to search addresses in the chunks table, and we saw this search incurs a significant run-time overhead. Finally, the problem would be solved even more radically if we could make sure that memory chunks in the backward sweep are always the same as in the forward sweep. This would require ADMM to provide its own dynamic memory management, and also to make some hypotheses on how the compiler allocates static stack memory, which would make the strategy more language- and platform-dependent. Each of these options would result in different implementations of the ADMM library. In any case, only experiments on a large code such as ALIF/ISSM will tell us which option is best.

Disclosure statement

No potential conflict of interest was reported by the authors.

References

[1] M. Fagan, L. Hascoët and J. Utke, *Data representation alternatives in semantically augmented numerical models*, 6th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2006, Philadelphia, PA, USA, 2006.

[2] R. Giering and T. Kaminski, Generating recomputations in reverse mode AD, in *Automatic Differentiation of Algorithms: From Simulation to Optimization*, G. Corliss, A. Griewank, C. Faure, L. Hascoet, and U. Naumann, eds., Chapter 33, Springer, Heidelberg, 2002, pp. 283–291. Available at http://www.springer.de/cgi-bin/search_book.pl?isbn = 0-387-95305-1.

[3] M.B. Giles, Collected matrix derivative results for forward and reverse mode algorithmic differentiation, in *Advances in Automatic Differentiation*, C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, and J. Utke, eds., Springer, 2008, pp. 35–44.

[4] Goldberg D.N. and P. Heimbach, *Parameter and state estimation with a time-dependent adjoint marine ice sheet model*, Cryosphere 7 (2013), pp. 1659–1678.

[5] D.N. Goldberg, P. Heimbach, I. Joughin and B. Smith, *Committed retreat of smith, pope, and kohler glaciers over the next 30 years inferred by transient model calibration*, Cryosphere 9 (2015), pp. 2429–2446. Availale at http://www.the-cryosphere.net/9/2429/2015/.

[6] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed., No. 105 in Other Titles in Applied Mathematics, SIAM, Philadelphia, PA, 2008. Available at http://www.ec-securehost.com/SIAM/OT105.html.

[7] L. Hascoët and V. Pascual, *The Tapenade Automatic Differentiation tool: Principles, Model, and Specification*, ACM Trans. Math. Softw. 39 (2013). doi:10.1145/2450153.2450158

[8] L. Hascoët and J. Utke, *Programming language features, usage patterns, and the efficiency of generated adjoint code*, Optim. Methods Softw. 31 (2016), pp. 885–903.

[9] P. Heimbach and V. Bugnion, *Greenland ice-sheet volume sensitivity to basal, surface and initial conditions derived from an adjoint model*, Ann. Glaciol. 50 (2009), pp. 67–80.

[10] 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 (2014), pp. 2335–2351. Available at http://www.the-cryosphere.net/8/2335/2014/.

[11] D.R. MacAyeal, *Large-scale ice flow over a viscous basal sediment: Theory and application to Ice Stream B, Antarctica*, J. Geophys. Res. 94 (1989), pp. 4071–4087.

[12] D.R. MacAyeal, *A tutorial on the use of control methods in ice-sheet modeling*, J. Glaciol. 39 (1993), pp. 91–98.
[13] M. Morlighem, E. Rignot, H. Seroussi, E. Larour, H. Ben Dhia and D. Aubry, *Spatial patterns of basal drag inferred using control methods from a full-Stokes and simpler models for Pine Island Glacier, West Antarctica*, Geophys. Res. Lett. 37 (2010), pp. 1–6.
[14] M. Morlighem, H. Seroussi, E. Larour and E. Rignot, *Inversion of basal friction in Antarctica using exact and incomplete adjoints of a higher-order model*, J. Geophys. Res. 118 (2013), pp. 1746–1753.
[15] V. Pascual and L. Hascoët, TAPENADE for C, in *Advances in Automatic Differentiation*, C. Bischof, M. Buecker, P. Hovland, U. Naumann, and J. Utke, J, eds., Lecture Notes in Computational Science and Engineering, selected papers from AD2008, August 2008, Bonn, Springer, 2008, pp. 199–210.
[16] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill and C. Wunsch, *OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes*, ACM Trans. Math. Softw. 34 (2008), pp. 18:1–18:36.
[17] J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland and U. Naumann, *Toward Adjoinable MPI*, in *Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering, PDSEC'09*, Rome, Italy, 2009.
[18] A. Walther and A. Griewank, Getting started with ADOL-C, in *Combinatorial Scientific Computing*, U. Naumann and O. Schenk, eds., Chapter 7, Chapman-Hall CRC Computational Science, 2012, pp. 181–202.