Mixed-language automatic differentiation

Valérie Pascual, Laurent Hascoët

To appear in *Optimization Methods & Software*
Vol. 00, No. 00, Month 20XX, 1–16

Mixed-language Automatic Differentiation

Valérie Pascual* and Laurent Hascoët

[a]*Université Côte d'Azur, INRIA, Sophia-Antipolis, France*

(*December 2017*)

As Automatic Differentiation (AD) usage is spreading to larger and more sophisticated applications, problems arise for codes that use several programming languages. This work describes the issues involved in interoperability between languages and focuses on the main issue which is parameter passing. It describes the architecture of a source transformation AD tool and the algorithms used to differentiate mixed-language codes. A language-independent internal representation enables application of global analysis and strategies on the entire source code. Our goal is that the Tapenade AD tool differentiates codes that mix C and Fortran and generates efficient differentiated code using these strategies.

Keywords: automatic differentiation, program transformation, mixed-language, interoperability, program analysis

AMS Subject Classification: 68N20; 68N99

1. Automatic Differentiation and Language Interoperability

Many Automatic Differentiation tools have been designed with one application language in mind. Only a few use an internal representation that promotes language-independence, at least conceptually. When faced with the problem of building (with AD) the derivative code of a mixed-language application, end-users may consider using several AD tools, one per language. However, this leads to several problems:

- Different AD tools may implement very different AD concepts such as overloading-based versus source-transformation based transformations, or association-by-address versus association-by-name for storage of derivatives values. These concepts are generally not directly compatible.
- When selecting the source-transformation concept (for efficiency of the differentiated code), performance of the differentiated code strongly depends on the quality of data-flow analysis, which is improved if it is global on the code. A global analysis with separate AD tools would require inter-tool communication at the level of data-flow analysis, which does not exist at present.

It is possible to produce efficient derivative code for mixed-language applications using single-language AD tools by writing by hand dummy definitions of black-box procedures with the same data flow behavior. However, this method is prone to error and is tedious to apply for large mixed-language applications. It is also possible to automatically interface differentiated codes from different languages with a suitable high-level specification, for

*Corresponding author. Email: Valerie.Pascual@inria.fr

instance for MATLAB, and C or Fortran source code. [1].

Moreover, efficient AD using different tools would require interoperable data-flow analysis between these tools, implying that the tools share their analysis strategy, which is almost never the case. Consequently, we think a better approach is to use a single tool, with a single internal representation and data-flow analysis strategy, therefore converting each source file to this unique representation regardless of its original language. It turns out that Tapenade [2] provides such an internal representation, accessible at present from C [3] or Fortran [4] sources.

Other AD tools provide a language-independent internal representation. OpenAD provides such a representation based on the XAIF formalism. However, this gives birth to two separate tools, OpenAD/F [5] for Fortran, and ADIC2 [6] for C. Still, it seems that there is no deep reason to prevent OpenAD application to mixed-language codes. We are lacking information about a common architecture between TAF and TAC++ [7] that would allow such mixed-language AD.

Rapsodia [8] [9] was the first AD tool to support algorithmic differentiation in tangent mode of mixed-language components, specifically C++ and Fortran. As Rapsodia uses operator overloading, it performs no global analysis of the code. To our knowledge the extension of mixed-language differentiation with Rapsodia to adjoint mode is not yet provided.

2. Language standards and interoperability

Language standards often say little about interoperability with other languages, leaving much freedom to compilers. As interoperability conventions differ across compilers, mixed-language applications may use an include file such as cfortran.h [10] to provide a compiler independent interface between C and Fortran procedures and global data.

Still, usage, de facto standards, has progressively evolved, in particular between C and Fortran. The Fortran 2003 standard has specified its interaction with C in more detail. As far as AD is concerned, AD tools should not commit to any specific interoperability strategy, and in particular to parameter-passing behaviors. Those might change with new languages and versions of languages. Instead, an AD tool must be able to handle a small set of behaviors from which one can describe all reasonable ways of parameter-passing.

The main issue raised by analysis and transformation of mixed-language codes is parameter-passing. Other issues are related to matching elements across languages, mainly types and procedures. These matching issues seem less complex than parameter-passing, but are relevant for the further discussion and considered first.

2.1 *Interoperability of types*

Interoperability between types (and between variables of these types) across languages relies on identical memory representations built by compilers. Obviously interoperable types must match in the sense that they have the same structure, number of fields, and these fields must recursively be of interoperable types. Compilers often grant a natural interoperability between structured types. However Fortran 2003 provides the `bind` attribute to tell at compile time that a Fortran type has a C equivalent. It is essential to identify interoperable types in both languages, in particular because it may help to distinguish candidate interoperable procedures according to the types of their arguments.

2.2 *Interoperability of procedures*

Similarly, interoperability between procedures relies on the compiler identifying the called procedure from another language, usually comparing procedure names, arguments number, and interoperable arguments types. Interoperable procedures in Fortran 2003 are declared with an explicit interface. The `bind` attribute defines a binding label. It is the name by which the Fortran procedure is known to the C processor. By default, the binding label is the lower-case version of the Fortran name. With Fortran 77 and Fortran 90, different conventions exist to associate both names: by adding an underscore character at the end of the Fortran name, either with the same name, or with the name in uppercase, depending on the compilers.

2.3 *Parameter-passing strategies*

The parameter-passing strategy is already a property of each given, single language. Mixed-language calls may be at the interface between two different parameter-passing strategies, which adds extra complexity.

Inside a given language, parameter-passing may use one of a few classical strategies: call by value, call by reference, call by value-result, call by sharing, call by name, etc. Call by value is the most common strategy. In call by value, the argument expression is evaluated, and the resulting value is copied to the corresponding variable in the function. If the called function overwrites one of its formal arguments, this affects only a local copy of the actual argument, so that the argument passed into the function call is unchanged when the function returns. In call by reference, a procedure receives a reference to a variable and can therefore modify the variable passed as actual argument. Call by value-result, also named call by copy-restore, is a special case of call by reference. It differs from call by reference when two arguments alias one another. Under call by reference, writing to one will affect the other. Call by value-result gives the function distinct copies, but the result in the callers environment depends on which of the aliased arguments is copied back first. Call by sharing is a terminology used by languages such as Python, Java and other object oriented languages. It is analogous to call by value where the passed value is either the argument when it is of a primitive type or its address when it is an object. Call by name is a strategy where each occurrence of the formal argument is replaced with the actual argument in the style of macro-expansion. This paper only focuses on the parameter-passing strategies used by Fortran and C.

2.4 *Fortran and C parameter-passing strategies*

Consider now the classical mix of Fortran and C. Parameter-passing mechanisms differ in the two languages. In Fortran, call by value-result and call by reference are used. Fortran 2003 introduces the `VALUE` attribute to specify call by value. In C, call by value is the only parameter-passing mechanism: all parameters are passed by value. In particular when an array is passed to a procedure, a pointer is passed: it is the address of the first element. In C, one simulates call by reference by passing a pointer to this parameter.

In mixed-language calls, the caller and the called procedures must agree on how parameters are passed, with explicit interfaces of the C procedures called from Fortran. The default parameter-passing mechanism between Fortran and C is call by reference. Inside a C procedure, all arguments of a Fortran call are represented by pointers, except the arguments corresponding to parameters with the `VALUE` attribute.

We believe that every mixed-language parameter-passing strategy used with Fortran

and C can boil down to a few simple behaviors at the time of entry into and return from the called procedure. At call time, we define what we call the *passed argument*, which may be the actual argument, or the memory pointed to by the actual argument, or conversely the address of the actual argument, depending on the mixed-language strategy that must be captured. Then the internal memory corresponding to this passed argument is copied into the internal memory corresponding to the called procedure's formal argument. At return time, the internal memory corresponding to the formal argument may be either copied back to the actual parameter, or not copied in which case it will vanish when the called procedure is popped from the call stack. When there is a back copy, it follows the link from the passed argument back to the actual argument: if they are the same, the copy is written into the actual argument, and if the passed argument is the destination of the actual, then the copy is written at the address designated by the actual argument.

Fig. 1 illustrates these behaviors for a few representative multi-language calls, and also for pure Fortran calls, distinguishing the scalar case from the array case, and for a C call using pointers simulating a call by reference. For each situation, we explicit the choice of the passed argument and the choice about back copy that implement the desired behavior. In anticipation on the next section, Fig. 1 also shows, for each situation, the `Translator` object used by Tapenade to specify these choices to the data-flow analyses.

3. Data-flow analysis of mixed-language applications

Static data-flow analysis [11] is an essential step to achieve efficient differentiation with a source-to-source AD tool. The goal of static data-flow analysis is to provide information on the data computed and returned by a program without knowing the values of the program's run-time inputs. In other words, static data-flow analysis extracts useful information on the program at compile-time, this information being thus valid for any run-time execution on any inputs. Obviously, such an information can only be partial and must often resort to the undecidable "I don't know" reply in addition to "yes" or "no". Abstract Interpretation [12] is a framework for static data-flow analysis in which the values computed in the original code are replaced with abstract values containing the propagated abstract information. One classical example of the abstract information that one may want to propagate is the interval in which the run-time value will range, or the set of possible destinations of each pointer variable. Starting from some abstract information on the inputs or outputs (which may be empty), abstract interpretation propagates it through the program, possibly guided by its control-flow structure. Instead of a true execution of the program, possible only at run-time, this propagation must stand for every possible execution path. Some data-flow analyses follow these paths forwards, others need to follow them backwards. As call graphs may be cyclic in general (recursivity), and flow graphs may be cyclic (loops), completion of the analysis requires the reaching of a fixed point both on the call graph and on each flow graph. The abstract domain in which the propagated information range is designed in such a way that this fixed point is reached in a finite number of iterations. Most of the classical data-flow analyses prove useful for AD as well as specific analyses such as activity and TBR analyses [13, 14]. In most AD-specific data-flow analysis the abstract information is, for each variable `v` a boolean value (e.g. does `v` influence the output in a differentiable way?) or a set of other variables (e.g. which input variables have a differentiable influence on `v`?).

To be accurate, data-flow analysis should be flow-sensitive and context-sensitive. In a context-sensitive analysis, each procedure uses a context that is built from the information available at its call sites. Even when choosing a generalization, i.e., using only

one context to summarize all call sites, this context allows the analysis to find more accurate results inside the called procedure. Flow-sensitive analysis operates at the flow graph level of a procedure. In a flow-sensitive analysis, the propagation of data-flow information follows an order compatible with the flow graph, thus respecting possible execution order. The present study does not interfere with flow-sensitivity of analysis, as flow-sensitivity operates at the level of an individual procedure, which is written in a single language. On the other hand, it is tightly related to context-sensitivity, as the frontier between two distinct implementation languages is precisely located at procedure calls.

In a context-sensitive analysis, each procedure is analyzed with one (or many) context that is built from the information available at its call sites. One can build one context that summarizes all call sites, through generalization, or one can build several contexts for each call site, making the choice of specialization. The AD tool Tapenade, for example, lets the user choose by adding directives in the source that activate specialization for a procedure [15], thus allowing the analysis to find more accurate results inside the called procedure.

Context-sensitive analysis has an essential role to play with mixed-language applications, as the context may belong to a different programming language with a different parameter-passing strategy. In the following sections, we will investigate how to extend our AD tool to analyze mixed-language codes, and how differentiation must be adapted.

4. Extension of Tapenade algorithms for interoperability

Tapenade was originally designed to support different imperative languages. The motivation was to share the concept of the tool and its implementation between these languages (at present, Fortran and C). We believe that this architecture also lets us deal with mixed-language source with a minimal implementation effort, affecting only a few components of the tool.

Tapenade represents a code as a call graph whose nodes represent procedures, and arrows represent calls. Each call graph node contains a flow graph, in which nodes are blocks of elementary instructions (in particular calls), and arrows represent control jumps. At present, Tapenade used on C or on Fortran source builds an internal representation of the same nature, using the same components for procedures, instructions, variables, types, etc. We exploit this common representation to deal with mixed-language codes, in particular mixing Fortran and C.

We first take a look at matching of types and procedures. Two interoperable entities are represented with the same internal representation in the AD tool. This representation distinguishes each component of structured types, and distinguishes pointer variables from their pointee destination variables. The representation of arrays, on the other hand, does not distinguish array elements, and therefore we need not worry about their possibly different memory layout in Fortran and C. Sometimes it is not possible to preserve this nice structural matching, for instance for the `complex` Fortran type which should match a 2-fields structure in C. Then the correspondence must be enforced by implementation, and possibly incurs some degradation of information.

The question of procedure matching amounts to finding, for a given procedure call, the node of the call graph that will be effectively called. This depends on the mixed-language conventions on procedure names and on type matching. Procedure name conventions may vary, and Tapenade offers parameterization to define one convention, using command-line arguments or directives at the call site. It also interprets attributes of Fortran 2003.

For instance, these parameters let us specify that inside a Fortran code all calls to a procedure named `FOO` will connect to a C procedure named `foo_`. Type matching may also play an important role here, when the number and types of argument may help disambiguate between several candidate procedure matches.

<table>
<tr><th>C calls Fortran</th><th>C calls Fortran, by value</th></tr>
<tr><td>

```
float *y;          SUBROUTINE BAR(V)
...                  REAL V
bar(y);
```

- Passed argument is *y
- Upon return, abstract information on V is copied back into *y

```
Translator:
     V -> *y  (Back copy)
```

</td><td>

```
float y;           SUBROUTINE BAR(V)
...                  REAL, VALUE:: V
bar(y);
```

- Passed argument is y
- Upon return, no copy takes place into y

```
Translator:
     V -> y  (No back copy)
```

</td></tr>
<tr><th>Fortran calls C</th><th>C calls C</th></tr>
<tr><td>

```
REAL X          void foo(float *a)
...
CALL FOO(X)
```

- Passed argument is address of X
- Upon return, no copy takes place into &X

```
Translator:
     a  -> &X  (No back copy)
     *a -> X   (Back copy)
```

</td><td>

```
float *y;       void bar(float *a)
...
bar(y);
```

- Passed argument is y
- Upon return, no copy takes place into y

```
Translator:
     a  -> y   (No back copy)
     *a -> *y  (Back copy)
```

</td></tr>
<tr><th>Fortran calls Fortran, scalars</th><th>Fortran calls Fortran, arrays</th></tr>
<tr><td>

```
REAL X              SUBROUTINE GEE(V)
...                   REAL V
CALL GEE(X)
```

- Passed argument is X
- Upon return, abstract information on V is copied back into X

```
Translator:
     V -> X (Back copy)
```

</td><td>

```
REAL Y(100)       SUBROUTINE GEE(B)
...                 REAL B(20)
CALL GEE(Y(10))
```

- Passed argument is address of Y(10)
- Upon return, no copy takes place into Y

```
Translator:
     B -> &(Y(10)) (No back copy)
```

</td></tr>
</table>

Figure 1. Mixed-language calls and the Translator that implements their behaviors

4.1 *Mixed-language data-flow analysis*

Parameter passing comes into play during data-flow analysis. In our tool's implementation, all data-flow analyses (e.g. in-out, activity, liveness) inherit from a base analysis class that provides primitives to transfer data-flow information between a caller procedure and a callee. Specific analyses only differ in the nature of the propagated information and in the way these propagated values behave for a handful of classical combination operations. For instance, in-out analysis finds in-out sets, which contain for each procedure the smallest possible sets of variables that may or must be read or overwritten by this procedure. Activity analysis propagates forward the set of the variables that depend on some independent input. It propagates backwards the set of the variables that influence some dependent output in a differentiable way. Most adaptions to mixed-language code must be done in the base analysis class. This information transfer is driven by an object we call a `Translator`, which describes how actual arguments are matched with formal arguments.

We consider scalars and arrays as elementary components. If we were to distinguish data-flow information of different cells of arrays, we would have to evaluate and compare all array indices in the code. This is, in general, out of reach of a static analysis. If an access to an element `A(i)` or to an array slice `A(lower:upper:stride)` of an array `A` modifies the data-flow information, our choice is to conservatively impact the complete array `A`. However, in Fortran `EQUIVALENCE` or `COMMON`, we consider each fragment as an elementary component.

Variables, and in particular those in arguments, may have a finer structure. Languages have introduced variables of structured type and pointers. Since the data-flow properties that we analyze may be different for each component of these structured objects, we distinguish all of their elementary components. Recursively, a structured type has one component per field, and a pointer type has two components corresponding to the pointer itself and to the pointer destination.

For example, a formal argument of type `mystruct *arg` where `mystruct` is a record:

```
struct mystruct {
   int numElems;
   float *elems;
}
```

is represented by four elementary formal arguments : the top-level `arg` which is a pointer to a structured type with two elementary components, `arg->numElems`, `arg->elems`. In addition, the fourth elementary argument represents the destination `*(arg->elems)`. While compilers do not require such a fine structure, abstract interpretation requires it. The data-flow information may then be more accurate on each component.

To each elementary formal argument of the called procedure, the `Translator` associates the corresponding elementary actual argument at the call site, which is either an expression or an elementary variable known to the calling procedure. In addition, the `Translator` associates to each elementary formal argument a `Back copy` boolean that specifies whether the corresponding data-flow information must be copied upon return. This boolean is set to "true" if the argument is passed by reference, or if the passed argument is the memory pointed to by the argument, or the address of the argument. For each example situation in Fig. 1, the `Translator` that implements the desired behavior is shown below the textual description of the behavior, as a set of arrows from formal elementary argument to actual elementary argument and back copy boolean. The rule of thumb is that the `Translator` associates the formal argument with the passed argument.

The "Fortran calls C" situation deserves further comment: since the C formal argument is a pointer to a float, there are in fact two elementary formal arguments, one for `a` and one for `*a`. The same happens for "C calls C". As `a` is associated with the passed argument `&X`, `*a` is naturally associated with `*(&X)` in other words with `X`. The actual value of (or information regarding) `X` is propagated to the callee through this second elementary argument. Consequently, even if no back copy is done upon return into `&X` itself, every write into `*a` in `foo` is automatically reflected into `X`.

The way each data-flow analysis, which computes a given data-flow value, uses the specification from the `Translator` can be sketched as follows: immediately before the call, and for each elementary formal argument (left column of `Translator`), we retrieve the corresponding elementary actual argument (right column), and we retrieve the current data-flow value for it. The initial data-flow value of the elementary formal argument is set to this retrieved value. Analysis can then run on the called procedure. Upon return from the call, the top-level elementary arguments that bear the "No back copy" retain the data-flow value they had before the call. For all other elementary arguments the data-flow value of the elementary formal argument upon return is copied back into the data-flow value of the elementary actual argument. This description applies to forward data-flow analyses. Adaption to backward analyses requires minor technical changes.

4.2 *Mixed-language differentiation*

The impact of mixed-language differentiation on the tangent mode of differentiation is quite limited. In tangent mode, the data-flow of derivatives follows closely the data-flow of primal variables. Therefore the strategy implemented in the primal code to pass primal variables can be simply reused for the tangent derivatives. There is a minor issue when differentiating a call to a function `F` which returns a value that has a derivative. Our convention is to pass this derivative as the return value of the differentiated `F_D`. Therefore `F_D` must be called with an extra argument to pass the primal result of `F`. This extra argument and its corresponding passed parameter must be declared and used according to the parameter-passing mechanism, e.g. by passing the address of the argument when `F` is a Fortran function and the call site is in C, as shown in Fig. 2. The issue is already present in mono-language C code.

```
      REAL FUNCTION F(t)
        REAL t
        ...
      END
```

```
      REAL FUNCTION F_D(t, td, f)
        REAL t, td, f
        ...
      END
```

```
extern float f_(float *t);

void foo(float *x) {
    *x = f_(x);
}
```

```
extern float f_d_(float *t, float *td,
                  float *f);
void foo_d(float *x, float *xd) {
    float tmpresult;
    *xd = f_d_(x, xd, &tmpresult);
    *x = tmpresult;
}
```

Figure 2. Tangent mixed-language differentiation with Tapenade

Problems arise for adjoint differentiation. Since a use of a variable in the primal code may become an overwrite of its adjoint variable, parameter passing method must be adapted. Consequently, the adjoint of an argument passed by value must sometimes be passed by reference. This is a general problem with pass-by-value parameters. It must be dealt with for mixed-language code as well as with mono-language code that uses pass-by-value.

In general, adjoint differentiation of a formal parameter that is passed by value requires *two* adjoint variables. A simple way to justify that is to view the formal parameter inside the called procedure as a new copy of the passed parameter. Consider for illustration the parameter `u` of procedure `BAR` in Fig. 3: overwrites of `u` in `BAR` do not affect the actual argument `*x` in `foo`. In the body of `BAR`, we will consider that `u` is now a local variable, whereas the first formal argument is renamed as `u0`. We conceptually add an initial instruction :

u = u0 ;

at the beginning of `BAR`. Applying the standard adjoint method, we introduce `ub`, the adjoint variable of `u`, initialized to zero and then used and updated in the body of the adjoint `BAR_B`. At the end of `BAR_B`, `ub` is added to `ub0` (the adjoint of the first formal argument), as the contribution to the adjoint of `ub0` (i.e. equivalently `*xb`) of the adjoint of its copy `u`. This is nothing but the standard adjoint of the initialization instruction of `u`. In order to propagate the resulting value of `u0b` to the calling subroutine, the passed actual argument `xb` must be a reference, and the argument `ub0` of `BAR_B` must not inherit the `VALUE` attribute from `u`. Because of the order of assignments, increments and updates of `ub` and `ub0`, there is in general no way of implementing this mechanism with a single adjoint variable `ub`. In the particular case where the passed-by-value argument is not modified in `BAR`, a single adjoint variable is enough, which must be passed by reference.

We focused on the main issue with mixed-language codes, which is parameter passing. We did not discuss interoperability of global variables. A Fortran variable that interoperates with a C variable is represented internally in the same way as a Fortran variable that is equivalenced with another Fortran variable with an `equivalence` declaration. Interoperable global variables, for instance using a `common` declaration or the `bind` attribute of Fortran, and an `extern` C declaration, share the same memory location. Inside the Tapenade internal representation these variables occupy the same memory location.

5. Application of Automatic Differentiation to CalculiX

To validate our extension of Tapenade for interoperability, we differentiate a real size mixed-language application, the CalculiX finite element library [16], written in C and Fortran77/Fortran90, which does not use the Fortran 2003 interoperability standard. A C include file contains the prototypes of the Fortran subroutines that are called from C. The main procedure is in C. The analysis of the source code with Tapenade detects 719 mixed-language calls of 178 different Fortran subroutines from C. No C procedure is called by a Fortran procedure. All data transfer is through arguments of procedure calls. No global variable or common is used to transfer data between C and Fortran. In our test case, the head procedure for differentiation is `linstatic`, which is called by the `main` procedure.

In order to measure the improvements that mixed-language differentiation brings, we will consider three differentiation approaches.

- Separate AD (Sec. 5.1) is the only approach available at a reasonable cost using only

```
void bar(float a, float *b);

void foo(float *x, float *y) {
  bar(*x, y);
}
```

```
void bar_b(float a, float *ab,
           float *b, float *bb);

void foo_b(float *x, float *xb,
           float *y, float *yb) {
    bar_b(*x, xb, y, yb);
}
```

```
SUBROUTINE BAR(u, v) BIND(C)
  IMPLICIT NONE
  REAL, VALUE :: u
  REAL :: v
  u = 2 * u
  v = u * u
END SUBROUTINE
```

```
SUBROUTINE BAR_B(u, ub0, v, vb)
                 BIND(C)
  IMPLICIT NONE
  REAL, VALUE :: u
  REAL :: ub, ub0
  REAL :: v
  REAL :: vb
  u = 2*u
  ub = 2*u*vb
  *vb = 0.0
  ub = 2*ub
  ub0 = ub0 + ub
END SUBROUTINE BAR_B
```

Figure 3. Mixed-language differentiation with Tapenade, C calling Fortran case

single-language AD tools. It will serve as a reference.
- True mixed-language AD can be achieved with single-language AD tools at a significant development cost. We describe this in Sec. 5.2.
- We automate mixed-language AD to achieve the same performance with a much lower development cost. We describe this in Sec. 5.3

With all methods, we use the option `-context` to generate automatically the context code to call the differentiated procedure `linstatic_d`. This option extends the data-flow analysis to the complete code passed to the differentiation command. It also declares, allocates, and initializes the input and output derivatives values [17]. Using this option requires that all source code is given to the AD tool, including not-to-differentiate code from the `main` procedure to the call to `linstatic`. Therefore 79 C source files (42 000 lines of code) and 627 Fortran source files (160 000 lines of code) are passed to the differentiation commands. We choose the default behavior of Tapenade, which is generalization, to analyze the procedures with different contexts at different call sites.

5.1 *Separate AD*

Separate AD is done in two steps. First we differentiate only the C source files, then we differentiate the Fortran source files. As the top procedure to differentiate is in C, we differentiate the C files with the command line:
`tapenade -head "linstatic(co maxvm_pnorm)>(co maxvm_pnorm)" -context *.c`
and consider all the Fortran subroutines as external subroutines.

When differentiating only the C code, the analyses of external procedure calls make

conservative assumptions and so are less accurate. The external procedure is supposed to communicate with the calling code only through the calling arguments. All actual arguments are supposed to be read inside the external procedure, and all actual arguments that are variables are supposed to be overwritten. Any variable argument of a differentiable type is assumed to return a value that depends on the input value of every argument of a differentiable type. Therefore a differentiated argument is created for any argument of a differentiable type. The differentiated C code that we obtain calls 28 differentiated external subroutines.

We then differentiate the Fortran files, given the list of these 28 Fortran subroutines called from C, with the command line:

```
tapenade -fixinterface -head "..." -head "..." ... *.f
```

To be consistent with differentiation of the C code, each of these Fortran subroutines is differentiated in the most general way, i.e. all outputs with respect to all inputs. We use Tapenade option `-fixinterface` to prevent it from applying any reduction to these output and input sets.

An extra step is needed to create the executable differentiated code. With separate AD, the Fortran procedure "foo" is differentiated as "foo_d" in a file "foo_d.f". The called name of "foo" from C is "foo_" and the name of the differentiated procedure is "foo__d" in the differentiated call. It must be changed to "foo_d_". The C calls of the 28 differentiated Fortran procedures must be changed. Finally we compile and link the differentiated object files and the necessary non differentiated object files to obtain the executable differentiated code.

In the performance comparisons of Sec. 5.4, this differentiated code is referred to as "Separate AD". It sets the reference to appreciate the improvements brought by mixed-language AD. The next two sections describe how to achieve better performance through mixed-language AD, either manually (a tedious and error-prone approach) or automatically through our new developments.

5.2 *Tedious manual mixed-language AD*

To achieve true efficient mixed-language AD with an AD tool that supports only one language, we must provide some knowledge of the external procedures that are called from the C source code. For each of the Fortran subroutines called from C, we must provide a stub which is a dummy definition written in C. This stub must embody the differentiable data dependences of the actual Fortran procedure, i.e. which outputs depend on which inputs. The first step then differentiates the complete C code, together with all these stubs. In a second step, we must differentiate the Fortran code, according to the activity context discovered by the first step for the Fortran subroutines represented by their stubs. This is made somewhat easier by Tapenade, that emits during differentiation messages of the kind:

```
(AD09) Please provide a differential of function foo
  for arguments Arg1=(in;out) Result=(out)
```

Following these messages we build a differentiation command on the Fortran code with as many differentiation heads as needed with the correct dependent and independent parameters. This two-steps method is only applicable when the call tree is easily split in two layers. More steps are required if for instance the Fortran code calls C back. Things may get even worse in the case of recursion. In our application, 28 Fortran subroutines are differentiated and called from C. These 28 Fortran subroutines need a particular attention to define a correct stub for each of them: each parameter must be carefully

checked, in particular for subroutines with many parameters. Notice that the stubs must be rewritten if these 28 Fortran procedures are modified, for instance if we want to differentiate a new version of Calculix.

The goal of the present section is to show how tedious this manual approach to mixed-language AD can be. As our purpose is to improve the AD tool Tapenade to automate mixed-language AD, we did not invest too much time applying it. We only checked that it eventually provides a differentiated code that is very similar and has the same performance as our new automated mixed-language AD, described in the next section.

5.3 *Automated mixed-language AD*

The different steps above are now replaced with just one step with mixed-language AD. All the C and Fortran source files are analyzed and differentiated at the same time, with the command line:

```
tapenade -head "linstatic(co maxvm_pnorm)>(co maxvm_pnorm)"
       -context *.c *.f
```

As we differentiated all the C and Fortran files at the same time, we add the option `-java "-mx51200m -Xss2280k"` to the Tapenade command to increase the Java process size. This single Tapenade command generates a differentiated code composed of the C `linstatic_d` differentiated procedure and its C calling context, and of all the C and Fortran differentiated procedures. All the generated files are compiled and linked together with the necessary non-differentiated files to build an executable, without any further manual modification. This approach produces a differentiated code which is efficient, i.e. as efficient as the one produced by the approach of Sec. 5.2, at a development cost which is much less than in Sec. 5.2.

5.4 *Experimental result*

Considering run-time and memory use of the differentiated program, there is no reason to expect a significant improvement from approach described in Sec. 5.2 to approach described in Section 5.3. The stub-based approach (5.2) is in principle able to express finely the dependence information, so that performance of the differentiated codes are similar. Approach 5.2 may even perform slightly better, like any approach that involves a great deal of hand modifications. The difference between approaches 5.2 and 5.3 is mainly about comparing a partly manual differentiation process that may take a few work days, with an automated process that takes minutes. In contrast, we provide execution comparisons between separate AD (5.1) and our new proposed mixed-language AD (5.3).

We execute the differentiated codes in tangent mode and measure the run-time and memory on examples provided with CalculiX. We have validated tangent mixed-language AD by comparison with divided differences. Table 1 shows a significant improvement both in run-time and memory usage. In average, the total run-time and the time spent in the `linstatic_d` differentiated procedure is divided by two with mixed-language AD compared with separate AD. This brings the slowdown factor of the differentiated code obtained by mixed-language AD down to 1.5 compared to the primal code.

Table 2 shows some time measurements for AD analyses and differentiation with both methods. Time spent during analyses is 3 to 6 times longer with mixed-language AD because we analyze the complete call graph. This gives more accurate results, as the conservative assumptions for the external procedures with separate AD are replaced with the results of the analyses on the complete Fortran source code of these procedures.

Calculix differentiated code execution	Separate AD	Mixed-language AD
linstatic_d run-time	0.104 s	0.057 s
total run-time	0.122 s	0.067 s
peak stack size	66.10 KB	48.66 KB

Table 1. Run-time and memory performance of the tangent differentiated code using on one hand separate AD and on the other hand mixed-language AD

Calculix C + Fortran code	Separate AD (C + Fortran)	Mixed-language AD
Pointer destinations analysis	25 + 0.02 s	144 s
In-out analysis	9 + 0.18 s	37 s
Activity analysis	102 + 0.9 s	440 s
Differentiation	8.5 + 1.5 s	5.5 s
Total time	242 + 11 s	750 s

Table 2. Time measurements of analyses and differentiations of Calculix C and Fortran code

The activity analysis through the complete call graph finds the variables that do not need to be differentiated. It detects the variables that do not depend on the independent input variables, and the variables that do not influence the dependent output variables. This is done for each mixed-language call of a procedure, whatever the language of the caller and the callee, and it takes into account the parameter passing strategy for each parameter. This leads to fewer differentiated variables and simplifies the differentiated code, through slicing and partial evaluation. Fewer instructions need to be differentiated as a result. If no formal parameter of a procedure is differentiated, the procedure is not differentiated, so fewer procedures need to be differentiated. Therefore the differentiation itself takes less time. Table 3 shows some measurements on the differentiated code we obtain with both methods.

Calculix C code differentiation	Separate AD	Mixed-language AD
# differentiated files	28	12
# non differentiated procedures	50	79
# context procedures	19	6
# differentiated procedures	21	5
# lines of differentiated code	42000	19400
# calls to differentiated Fortran subroutines	28	5

Table 3. Comparison of the C part of the differentiated code, using on one hand separate AD and on the other hand mixed-language AD. The original C code consists of 79 files, 90 procedures, and 40000 lines of code. The call graph under the differentiation root contains 80 C procedures.

The first benefit that we notice in Table 3 is the reduction of the number of differentiated files and procedures. Let us detail the contents of these generated files. In C, a given source file may define several static variables. The scope of these variables is shared by all the procedures of this file and not by others outside. Consequently, it is not possible to split these files into many, nor to merge them into a single file. The consequence for AD is that we cannot use the Tapenade option that puts all differentiated code into one file, nor can we create one file per differentiated procedure. Differentiated code must

follow the file structure of the source code. If a static variable is used in both a source procedure and in a differentiated procedure, it must be defined only once, which means the differentiated file must also contain the source procedure and the definition of this static variable. In CalculiX source code, only one differentiated file contains a copy of a source procedure to respect the file scope of global variables.

With mixed-language AD, the total number of lines is divided by more than half. The accurate analyses reduce the number of differentiated procedures. Only 5 C procedures are differentiated, instead of 21 on a total of 90 in the source code. Among the 12 generated files, 5 files contain differentiated procedures, 6 files contain procedures used in the context call of the `linstatic_d` procedure. The last generated file contains the prototypes of differentiated procedures and is included in the other generated files in order to compile them correctly.

The second benefit is observed inside each differentiated procedure. Several C differentiated procedures have fewer differentiated parameters. The last line in Table 3 summarizes the benefit of mixed-language AD: the number of Fortran differentiated subroutines called from C is reduced by a factor of more than 5. Instead of the 28 Fortran subroutines that separate AD has differentiated, or the 28 that manual mixed-language AD have required a stub for, we observe that only 5 are effectively differentiated by our automated approach.

To summarize, mixed-language AD and its accurate analyses on the complete call graph produce a more efficient code, with fewer differentiated procedures, parameters, fewer calls to external Fortran procedures, and fewer differentiated instructions. All these improvements result from the accurate interprocedural analyses of the complete source code. Furthermore no additional hand-written stubs or code is required. Time spent to obtain and validate a mixed-language differentiated code is drastically reduced. Mixed-language AD of CalculiX with Tapenade generates a code that compiles without modification, the interoperability convention is automatically used in the differentiated code.

However we still need to improve the generated code if the source code uses an include file such as cfortran.h or the one in CalculiX, and function-like macros to call Fortran subroutines from C, as shown here:

```
FORTRAN(foo,(co, ...));
```

With the GNU compiler, this call is expanded in:

```
foo_(co, ...);
```

As Tapenade differentiates the C source files after using the C preprocessor, all the macro definitions are lost in the differentiated code, as shown in this piece of differentiated code of CalculiX:

```
foo_(co, ...);
...
foo_d_(co, cod, ...);
```

Tapenade offers parameterization to describe the association of Fortran and C procedure names that respects the compiler convention. If the C source files contain macro-definitions for calling Fortran subroutines, we would like to regenerate a code after post-processing with the same convention with macro-definitions such as:

```
FORTRAN(foo,(co, ...));
...
FORTRAN(foo_d,(co, cod, ...));
```

In the current implementation, the differentiated code only respects the interoperability convention of the current compiler.

6. Conclusion and further work

We extended Tapenade to differentiate codes that mix C and Fortran, with calls in either direction and validated this extension on the CalculiX finite element library. The architecture of Tapenade, with a language-independent internal representation, allowed us to implement this functionality quite easily, with modifications in only a few component of the tool. Most adaptions to mixed-language code concern the data-flow analyses used by Tapenade and the data-flow information transferred between a caller procedure and a callee. This information transfer is driven by a translator which describes the parameter-passing strategy.

Measurements on the generated code of CalculiX show significant improvements with mixed-language AD. As the analyses are more efficient, the differentiated code contains fewer differentiated procedures, variables, and instructions and is only half as long. From a user point-of-view, mixed-language AD avoids a lot of time consuming manual handling to obtain correct differentiated code, and allows a more "automatic" differentiation.

Concerning the mixed-language AD of CalculiX, only the tangent code has been validated. The next step will be the validation of the adjoint code that now uses the adjoinable dynamic memory management library ADMM.

Future improvements may concern the differentiation of very large mixed-language codes that take too much space and time, in particular during pointer destination analysis and activity analysis. It turns out that mixed-language applications are often larger than single-language. The natural answer to size problems used in compilers is separate compilation. The question then arises whether we can provide a sort of separate AD. This is a difficult issue as AD heavily relies on global data-flow analysis. This would require a convenient way to provide data-flow information on procedures that are either external or written in a different language. This question would in turn ask for a standardization effort among AD tools and perhaps with compilers.

References

[1] H. M. Bücker, A. Elsheikh, and A. Vehreschild. A system for interfacing MATLAB with external software geared toward automatic differentiation. In A. Iglesias and N. Takayama, editors, *Mathematical Software – ICMS 2006, Proceedings of the Second International Congress on Mathematical Software, Castro Urdiales, Spain, September 1–3, 2006*, volume 4151 of *Lecture Notes in Computer Science*, pages 373–384, Berlin, 2006. Springer.

[2] Laurent Hascoët and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3):20:1–20:43, 2013.

[3] Valérie Pascual and Laurent Hascoët. TAPENADE for C. In Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 199–209. Springer, 2008.

[4] Valérie Pascual and Laurent Hascoët. Extension of TAPENADE toward Fortran 95. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering, pages 171–179. Springer, 2005.

[5] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4):18:1–18:36, 2008.

[6] 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.

[7] Michael Voßbeck, Ralf Giering, and Thomas Kaminski. Development and first applications of TAC++. In Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke,

editors, *Advances in Automatic Differentiation*, pages 187–197. Springer, 2008.

[8] Isabelle Charpentier and Jean Utke. Fast higher-order derivative tensors with Rapsodia. *Optimization Methods & Software*, 24(1):1–14, 2009.

[9] Jean Utke, Bradley T. Rearden, and Robert A. Lefebvre. Sensitivity analysis for mixed-language numerical models. *Procedia Computer Science*, 18:1794 – 1803, 2013. 2013 International Conference on Computational Science.

[10] B. D. Burow. Mixed Language Programming. In R. Shellard and et al., editors, *Computing in High Energy Physics: CHEP '95 - Proceedings of the International Conference. Edited by Shellard Ronald et al. Published by World Scientific Publishing Co. Pte. Ltd., 1996. ISBN #9789814447188, pp. 610-614*, pages 610–614, 1996.

[11] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[12] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[13] Laurent Hascoët, Uwe Naumann, and Valérie Pascual. "To be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8):1401–1417, 2005.

[14] Valérie Pascual and Laurent Hascoët. Native handling of message-passing communication in data-flow analysis. In Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther, editors, *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*, pages 83–92. Springer, Berlin, 2012.

[15] Jan Christian Hueckelheim, Laurent Hascoët, and Jens-Dominik Müller. Algorithmic differentiation of code with multiple context-specific activities. *ACM Transactions on Mathematical Software*, 2016.

[16] G. Dhondt. *The Finite Element Method for Three-Dimensional Thermomechanical Applications*. Wiley, 2004.

[17] L. Hascoët and M. Morlighem. Source-to-source adjoint algorithmic differentiation of an ice sheet model written in C. *Optimization Methods and Software*, 0(0):1–15, 2017.