Compile Time Differentiation

Dominic P Jones[*]

August 2017

Abstract

To ease the burden of manually differentiating arithmetic expressions, typically required for implementing adjoint solvers, a methodology is presented which automatically performs the differentiation of an expression or a block of expressions and yields more efficient machine code than its equivalent source transformation implementation [2]. The methodology leverages template metaprogramming techniques to provide a means of generating the differentiated statements whilst faciliating perfect inlining of code. Whilst the the functionality of the methology is limited, it is sufficiently useful so as to be applied to loop code blocks typical of finite volume or finite element algorithms.

Keywords: Expression Templates, Operator Overloading, Algorithmic Differentiation, Adjoint, C++, Template Metaprogramming

1 Introduction

Two dominant methodologies exist for computing the differential of an algorithm without having to explicitly program its derivative: the first and more popular is to replace the floating point type of the algorithm with another type which augments the relational behaviour of the original floating point type with the side effects of recording every operation and associated state to some unique stack [1]. The derivative is evaluated by interpreting the stack after the algorithm has completed its execution. This method is commonly referred to, in context, as operator overloading. The second methodology is to use a tool to read the source code which implements the algorithm, and have the tool generate new source code which contains the implementation of the original algorithm and its derivative [2].

Both methods have inherent limitations with respect to their application to large industrial codes: the first method destroys compiler optimisation opportunities due to its side effects and its exclusively run-time governed interpretation of the heap-stored stack. The second potentially retains the high-performance characteristics associated to manually implemented differentiated code, but is generally difficult, if not impossible, to apply it to codes written modern system programming languages due to their expansive complexity.

Performance is a perennial issue in industrial numerical software, and especially in the field Computational Fluid Dynamics, where enormous computational resources are dedicated to solving simulations. It would not be unreasonable for the user of such software to expect the adjoint of a flow simulation to take about the same time as the flow simulation itself requiring proportional similar memory. To achieve this parity, both the flow algorithm and its adjoint would need to have similar implementation characteristics and compile to highly efficient machine code. A compromise on efficiency from the outset regarding how the adjoint is implemented would hinder the adoption of the feature, especially for industrial users to routinely simulate flow problems close to the maximum capacity afforded by their systems.

In addition to these difficulties, there is the often overlooked but critical issue of program build varieties. It is already common to have two versions of a numerical analysis software, one compiled with mixed precision floating point types and the other with double precision types. With a continuous delivery build system, every variety of the released program must be compiled and tested on a nightly basis. If another version was added, such as build using a differentiable double precision type, the resources required to absorb this extra work load may very well outweigh the perceived benefit in the added functionality being delivered.

With these constraints in mind, hand coding the required differentiated components appears to emerge as the path of least resistance for obtaining the desired adjoint solver. It is under this perspective that the following work pursued, whose aim is of offering some helping-hand to carrying out the task whilst minimising performance compromises.

2 Methodology

Consider the contrived function in Listing 1 which has two input fields and one output field. By templating this function on a differentiation mode and renaming `Field` and `double` as types dependent on the provided mode, the proposed methodology enables the function to compute its primal, tangent or adjoint. Evaluating any of the modes is as trivial as calling the function with the correct mode; no recording of operations is performed.

[*]Netherhall House, London NW3 5SA, `dominic.jones@gmx.co.uk`

```
void example(Terminal<A> const &a,
             Terminal<B> const &b,
             Terminal<R> &r)
{
  // unique types for passive values
  auto const c0 = UQ(5);
  auto const c1 = UQ(6);

  // generate expression tree
  auto const tmp0 = c0 * a * sin(a) / cos(b) * c1;
  auto const tmp1 = arg1 * c0 * sin(b) / UQ(7) * cos(a);
  auto const tmp2 = sin(tmp0) * tmp1 + sin(tmp1) * tmp0 * c1;
  auto const tmp3 = sin(tmp2) / cos(tmp2)  + cos(tmp0) / sin(tmp0);

  // evaluate upon assignment
  result = tmp3;
}
```

Listing 1: All terminal nodes require unique types, both for active and passive values.

2.1 Unique Types

`__COUNTER__` is a non-standard preprocessor (though is supported by GCC, Clang and MSVS) macro which is used in the implemention of `UQ`. This counter is incremented on every use, providing a sure mechanism to generate the unique types for the passive values.

2.2 Functional Purity

The dominant context of this methodology is the prerequisite of functional purity of the components of the program to be differentiated. This is a difficult prerequisite to honour, but is possible and is ultimately beneficial regardless of whether or not its pursuit is for the sake of adopting this methodology to compute derivatives. In the context of programming in C++, this can be summarised by the directives of: never passing an argument by non-`const` reference and always initializing class member data in the constructor member initialization list.

2.3 Location of State

Typical implementations of expression trees hold the intermediate state of the unary or binary operations [3]. However, they need not do so. The absolute minimum of state is simply the addresses of the terminals and the values of the passive terms. This is the approach used here, and the reason for taking this approach is to maximise the liklihood of the copy constructor being inlined when the tree is being built. If the nodes hold the state then as the tree gets larger the copying operation becomes on par with the computation. One way to assure such exccessive work is avoided is to not hold intermediate state in the first place. Storage of intermidate values will be requred at some point, but the (static) allocation is delayed until the result assignment is reached.

2.4 Metaprogramming Utilities

Brigand metaprogramming library was made use of as a foundational set of functions for implementing the algorithms needed to analyse and transform the expression tree into a list of operations.

3 Performance

The methodology outlined in this paper incurs a run-time cost of 1.34, relative to the primal, for the adjoint evaluation of Listing 2. To compare the run-time performance of the adjoint, the source code (after lowering it to C code) was supplied to Tapenade [2], a source transformation differentation tool. Its run-time cost was 1.51, relative to the primal.

Tests were compiled with g++ 6.2.0 (with the `-O3` flag) on a machine running Intel Xeon E5-2650 processors. Results presented are obtained by computing the median-average run-time of 50 sets of evaluations, where each set performs 100,000 invocations of the tested function.

```
void test(Terminal<A> const &a,
          Terminal<B> const &b,
          Terminal<R> &r)
{
  auto const c1 = UQ(1);
  auto const c2 = UQ(2);
  auto const c3 = UQ(3);
  auto const c4 = UQ(4);
  auto const c5 = UQ(5);
```

```
    auto const c6 = UQ(6);
    auto const c7 = UQ(7);
    auto const c8 = UQ(8);
    auto const c9 = UQ(9);
    auto const c10 = UQ(10);
    auto const c11 = UQ(11);
    auto const c12 = UQ(12);

    auto const tmp0 = c1 * a * sin(a) / cos(b) * c2;
    auto const tmp1 = b * c3 * sin(b) / cos(a);
    auto const tmp2 = sin(tmp0) * tmp1 + sin(tmp1) * tmp0;
    auto const tmp3 = sin(tmp2) / cos(tmp2) + cos(tmp0) * c4 / sin(tmp0);
    auto const tmp4 = sin(tmp2) / c5 * cos(tmp2) + cos(tmp1) / sin(tmp3);
    auto const tmp5 = sin(tmp2) * cos(tmp4) + cos(tmp0) * sin(tmp1);
    auto const tmp6 = c6 * sin(tmp2) / cos(tmp5) + cos(tmp0) / sin(tmp3);
    auto const tmp7 = sin(tmp4) + cos(tmp6) + cos(tmp0) * c7 / sin(tmp5);
    auto const tmp8 = sin(tmp1) / cos(tmp5) * cos(tmp0) * sin(tmp6);
    auto const tmp9 = sin(tmp6) + c8 * cos(tmp4) + cos(tmp0) / sin(tmp1);
    auto const tmp10 = sin(tmp5) / cos(tmp3) + cos(tmp7) + sin(tmp2);
    auto const tmp11 = sin(tmp8) * c9 * cos(tmp2) + cos(tmp0) / sin(tmp3);
    auto const tmp12 = c10 * sin(tmp10) / cos(tmp1) + cos(tmp0) / sin(tmp4);
    auto const tmp13 = sin(tmp12) + cos(tmp0) * c11 + cos(tmp0) / sin(tmp3);
    auto const tmp14 = cos(tmp13 / sin(tmp0) + tmp6 / sin(tmp11) * c12 * tmp13 *
        tmp11 * tmp9);
    r = tmp14;
}
```

Listing 2: The test function, making much use of transcendental functions

4 Conclusion

A high performance tool has been developed for aiding the implementon of adjoint code. Its functionality is limited, but what is does support, namely blocks of pure-functional numerical expressions, it supports very well. A better way of handling passive values is very much wanted, though seeking a more favourable solution may require a significantly different methodology to the one presented here. Futhermore, the limitation on single assignment is excessively stringent. Some way of being able to check-point the assignments in order to avoid duplicate evaluations is wanted. But again, probably a different methodology is required for this.

References

[1] Andreas Griewank, David Juedes, H. Mitev, Jean Utke, Olaf Vogel, and Andrea Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Institute of Scientific Computing, Technical University Dresden, 1999. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167.

[2] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39(3), 2013. URL http://dx.doi.org/10.1145/2450153.2450158.

[3] Robin J. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software*, 40(4):26:1–26:24, jun 2014. URL http://doi.acm.org/10.1145/2560359.