AD2016 - Block Scope Differentiation

Dominic P Jones*

March 2016

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. 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.

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.

Since the operator overloading approach recasts the fundamental numerical data type, the primal data must always be accessed in tandem with the derivative data. In the case of adjoint evaluation, there is a conflict of use: the primal will only be read, whilst the adjoint will be modifiable. Binding the data prevents any distinction. Consequently, ambiguity of `const` correctness is introduced, and, in the case of parallel computation, inter-partition exchanges are unnecessarily doubled in size.

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.

```
1  void evaluate(Region const &region)
2  // becomes: template<DrvMode mode> void evaluate(Region const &region)
3  {
4    using double_t = double;
5    // becomes: using double_t = Drv<mode, TypeValue<double, 7> >;
6
7    Field<const Pressure, Cell> fp(region);
8    Field<const Volume, Cell> fv(region);
```

*CD-adapco, London W6 7NL, dominic.jones@cd-adapco.com

```
 9    Field<Temperature, Cell> ft(region);
10    // becomes: Drv<mode, Field<const Pressure, Cell> > fp(region); ... etc
11
12    FieldIndex<Cell> it(region);
13    for (; it.test(); it.increment())
14    {
15      /* cache the inputs */
16      double_t const A(fp[it]);
17      double_t const B(fv[it]);
18
19      /* compute some complicated terms... */
20      double_t const t(A * B + sin(B / A));
21      double_t const u(cos(A / B) - t * A + B / t);
22      double_t const v(-B / A + u / t - B * u / t);
23      double_t const w(A * (t + u / v) * B / t);
24      double_t const x(v > w? (double_t(w)): double_t(-w));
25
26      /* write the result */
27      ft[it] = A > B? double_t(x * sin(w) / A): double_t(x * cos(w) * B);
28    }
29  }
```

Listing 1: The common task of looping over all elements, filling the output field.

2.1 Destructors

Adjoint evaluation requires the reversing of the sequence of operations. This is triggered by the destructors of the local objects within the loop block scope. Expressions are captured and stored by the local objects during their construction. These captured expressions are retained in an opaque manner (i.e. as an array of `char`s) along with a pointer to its `adjoint` method, which has an assumed signature. Allied with these two pieces of data, the object triggers the execution of the expression adjoint when it goes out of scope.

Since the result is assigned rather than constructed and returned, its expression adjoint is evaluated immediately. This is a reasonable solution provided the state is not later mutated with anything other than accumulation operations.

The use of the block scope as a trigger mechanism for adjoint implies that nested blocks cannot be permitted around local objects. Whilst this is unenforceable, this methodology has the prerequisite that code must be functionally pure, ruling out the possibility of block scopes in correctly written code.

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 Referenced State

When an expression is evaluated in the adjoint mode, the ultimate data storing the derivative accumulation of the terms need to be written to. To achieve this, the state of all adjoint expression objects contains the primal value, the derivative value, and a reference which, upon construction, holds the address of the derivative value passed in (Listing 2). If no derivative parameter is supplied, as in the case when it is constructed from an expression (via a sub-class), the reference holds the address of its own derivative value.

```
 1  template<typename double_t>
 2  class Drv<
 3    DrvMode::Adjoint,
 4    double_t,
 5    typename std::enable_if<std::is_same<double, double_t>::value>::type>
 6    : public DrvExpression<DrvMode::Adjoint, double, Drv<DrvMode::Adjoint,
         double_t> >
 7  {
 8  public:
 9    Drv(double const &primal)
10      : _primal(primal)
11      , _derivative(0)
12      , _adjoint(_derivative)
13    {}
14
15    Drv(double const &primal, double &derivative)
16      : _primal(primal)
17      , _derivative(derivative)
18      , _adjoint(derivative)
```

```
19    {}
20
21    double primal() const { return _primal; }
22
23    void adjoint(double const &derivative) const { _adjoint += derivative; }
24
25  protected:
26    double const _primal;
27    double _derivative;
28
29  private:
30    double &_adjoint;
31  };
```

Listing 2: The adjoint type for a `double`

In the context of evaluating the adjoint of expressions, copies of adjoint objects are made. When the default copy constructor is invoked, all member data is literally copied so that the new object's adjoint reference holds the address of the copied derivative value, not to its own, enabling adjoint results to be correctly accumulated to its ultimate sources.

2.4 Capturing Expressions

The expression template technique relies on the ability to assign one object to another without the latter having inherent knowledge of the former's type (i.e. the latter could never own an instance of the former within the constraints of the type system). For the evaluation of the tangent derivative this does not present a problem since the primal and derivative can be evaluated at the same time. For the evaluation of adjoints, somehow the expression received as the argument of the constructor must be stored by the object so as to be later evaluated in an adjoint manner when its destructor is called, i.e. somehow the host must own a copy of the assigned object *and* be able to make use of it.

Since the type of the expression object is lost once the constructor is evaluated, only raw data can be retained. Alone, nothing useful can be done with the raw data, but a delegate to a member function associated to the raw data and the original type of the expression enables the host object to evaluate the delegate. This is what is done in the destructor of Listing 3 to trigger the adjoint evaluation of the captured expression.

Two important details are noted here: the first is that despite the standard library offering tools for creating and using delegates (`std::bind` and `std::function`), their performance is suboptimal as they are general purpose tools (typically invoking heap allocations) so a method-to-function trait class is used instead. The second point is that in order to capture the expression, a sufficiently large block of raw (stack) storage is required. The required size cannot be known up front, so templating the host class on a suitable integral value to dictate the maximum size is resorted to.

```
1   template<typename double_v_t>
2   class Drv<
3     DrvMode::Adjoint,
4     double_v_t,
5     typename std::enable_if<std::is_same<double_v<double_v_t::value()>, double_v_t
          >::value>::type>
6     : public Drv<DrvMode::Adjoint, double>
7   {
8   public:
9     // 'adjoint' method signature
10    using Function = void(*)(void const * const this_, double const &derivative_);
11
12    template<typename Expr_t>
13    Drv(Expr_t const &expr)
14      : Drv<DrvMode::Adjoint, double>(expr.primal())
15      , _object(this->object(expr))
16      , _function(this->function<Expr_t>())
17    {}
18
19    ~Drv() { this->adjoint(_derivative); }
20
21    void adjoint(double const &derivative) const
22    {
23      if (_function) { (*_function)(_object, derivative); }
24    }
25
26  private:
27    template<typename Expr_t>
28    void const * object(Expr_t const &expr)
29    {
30      static_assert(sizeof(_stack) >= sizeof(expr), "");
31      std::memcpy(_stack.data(), &expr, sizeof(expr));
32      return _stack.data();
33    }
```

```
34
35    template<typename Expr_t>
36    static Function function()
37    {
38      using Traits = FunctionTraits<decltype(&Expr_t::adjoint)>;
39      return &Traits::template function<&Expr_t::adjoint>;
40    }
41
42    static constexpr std::size_t size()
43    {
44      return {sizeof(double) * double_v_t::value()};
45    }
46
47    std::array<char, size()> _stack;
48    void const * const _object;
49    Function const _function;
50  };
```

Listing 3: Capturing an expression for deferred evaluation

The reason for splitting the adjoint form of the `double` class into a super-class and and sub-class is to enable the latter to capture constructs built from the former. In other words, the the object being captured cannot be some product of other terms of the same type of that which is capturing the expression. The size of that captured must be less than the cache size of the object doing the capturing. This would be impossible if all adjoint `double` types were identical. Instead, when building up the expression type, the operands must be type-sliced to its super-class (and owned by-value rather than by-reference).

2.5 Optimisation

To obtain the run-time performance achieved in this work, an optimisation must be made concerning what a given operator (such as a binary operator) holds as member data. Ordinarily, these operator classes hold copies of their operands, as oppose to `const`-qualified references, since any of the received operators may be temporaries, i.e. previous expression results. However, if either of them are not expression types, then they necessarily must be named objects, meaning they can be safely referenced. Doing so improves run-time performance by as much as 20%.

3 Performance

The methodology outlined in this paper incurs a run-time cost of 2.7, relative to the primal, for the adjoint evaluation of Listing 1. The size of the `std::array` of `char`s used to capture assigned expressions (Listing 3) is set to the minimum compilable value, though the effect of doubling, trebling and quadrupling the array size showed no meaningful trend in the effect on execution time. To compare the run-time performance of the adjoint, the Adept library [1] was used to evaluate the adjoint of the same function. Its normalised execution time was 3.3.

Tests were compiled with g++ 5.1.0 (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 executions, where the field size is 100,000 elements.

4 Conclusion

A highly performant 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. No change of fundamental numerical type is required at the lowest level, nor an external language parsing tool, nor any post-evaluation stack interpretation.

References

[1] 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`.