# CHAPTER 15 Acceleration with Expression Templates

In this final chapter, we rewrite the AAD library of Chapter 10 in modern C++ with the *expression template* technology. The application of expression templates to AAD was initially introduced in [89] as a means to accelerate AAD and approach the speed of manual AD (see Chapter 8) with the convenience of an automatic differentiation library.

The result is an overall acceleration *by a factor two to three* in cases of practical relevance, as measured in the numerical results of Chapters 12, 13, and 14. The acceleration applies to both single-threaded and multi-threaded instrumented code. It is also worth noting that we implemented a selective instrumentation in our simulation code of Chapter 12. It obviously follows that only the instrumented part of the code is accelerated with expression templates. It also follows that the instrumented part is accelerated by a very substantial factor, resulting in an overall acceleration by a factor two to three.

The resulting sensitivities are identical in all cases. The exact same mathematical operations are conducted, and the complexity is the same as before. The difference is a more efficient memory access resulting from smaller tapes, and the delegation of some *administrative* (not mathematical) work to *compile time*. The benefits are computational, not mathematical or algorithmic. Like in the matrix product of Chapter 1, the acceleration is nonetheless substantial.

The new AAD library exposes the exact same interface as the library of Chapter 10. It follows that there is no modification whatsoever in the instrumented code. The AAD library consists of three files: AADNode.h, AADTape.h, and AADNumber.h. We only rewrite AADNumber.h, in a new file AADExpr.h. AADNode.h and AADTape.h are unchanged. We put a convenient pragma in AAD.h so we can switch between the traditional implementation of Chapter 10 and the implementation of this chapter by changing a single definition:

```
1  //  AAD with expression templates
2  #define AADET   true
3
4  #if AADET
5
6  #include "AADExpr.h"
7
8  #else
9
10 #include "AADNumber.h"
11
12 #endif
```

although this is only intended for testing and debugging. The expression template implementation is superior by all metrics, so we never revert to the traditional implementation in a production context.

## 15.1 EXPRESSION NODES

At the core of AADET (AAD with Expression Templates) lies the notion that nodes no longer represent elementary mathematical operations

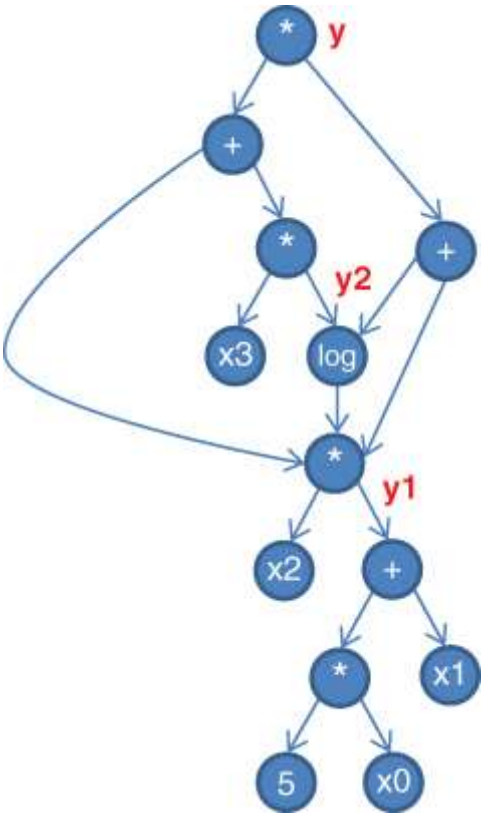
like $+$, $*$, $log()$ or $sqrt$, but entire expressions like our example of Chapter 9:

$$y = (y_1 + x_3 y_2)(y_1 + y_2), y_1 = x_2(5x_0 + x_1), y_2 = \log(y_1)$$

or, in (templated) code:

```
1  template <class T>
2  T f(T x[5])
3  {
4      auto y1 = x[2] * (5.0 * x[0] + x[1]);
5      auto y2 = log(y1);
6      auto y = (y1 + x[3] * y2) * (y1 + y2);
7      return y;
8  }
```

Recall that the DAG of this expression is:




We had 12 nodes on tape for the expression, 6 binaries, 2 unaries, and 4 leaves (a binary that takes a constant on one side is a unary, and a constant does not produce a node, by virtue of an optimization we put in the code in Chapter 10).

The size of a Node in memory is given by $sizeof(Node)$, which readers may verify easily, is 24 bytes in a 32-bit build. This is 288 bytes for the expression. Unary and binary nodes store additional data on tape: the local derivatives and child adjoint pointers, one double (8 bytes) and one pointer (4 bytes) for unaries (12 bytes), two of each (24 bytes) for binaries. For our expression, this is another 168 bytes of memory, for a total of 456 bytes.[1] In addition, the data is scattered in memory across different nodes.

The techniques in this chapter permit to represent the whole expression *in a single node*. This node does not represent an elementary mathematical operation, but the entire expression, and it has 4 arguments: the 4 leaves $x_0$, $x_1$, $x_2$, and $x_3$. The size of the node is still 24 bytes. It stores on tape the four local derivatives to its arguments (32 bytes) and the four pointers to the arguments adjoints (16 bytes) for a total of 72 bytes. The memory footprint for the expression is *six times* smaller, resulting in shorter tapes that better fit in CPU caches, explaining part of the acceleration.

In addition, the data on the single expression node is coalescent in memory so propagation is faster. The entire expression propagates on a single node, saving the cost of traversing 12 nodes, and the cost of working with data scattered in memory.

Finally, we will see shortly that the local derivatives of the expression to its inputs are produced very efficiently with expression templates, with costly work performed at compile time without run-time overhead. Those three *computational* improvements combine to substantially accelerate AAD, although they change nothing to the number or the nature of the mathematical operations.

With the operation nodes of Chapter 10, a calculation is decomposed in a large sequence of elementary operations with one or two arguments, recorded on tape for the purpose of adjoint propagation. With expression nodes, the calculation is decomposed in a smaller sequence

of larger expressions with an arbitrary number of arguments. Adjoint mathematics apply in the exact same manner. Our Node class from Chapter 10 is able to represent expressions with an arbitrary number of arguments, even though the traditional AAD code only built unary or binary operator nodes.

This is all very encouraging, but we must somehow build the expression nodes. We must figure out, from an expression written in C++, the number of its arguments, the location of the adjoints of these arguments, and the local derivatives of the expression to its arguments, so we can store all of these on the expression's node on tape. We know since Chapter 9 that this information is given by the expression's DAG, and we learned to apply operator overloading to build the DAG in memory.

However, the expression DAGs of Chapter 9 were built at run time, in dynamic memory and with dynamic polymorphism, at the cost of a substantial run-time overhead. To do it in the same way here would defeat the benefits of expression nodes and probably result in a *slower* differentiation overall. Expression templates allow to build the DAG of expressions *at compile time*, in static memory (the so-called stack memory, the application's working memory for the execution of its functions), so their run-time traversal to compute derivatives is extremely efficient.

The derivatives of an expression to its inputs generally depend on the value of the inputs, which is only known at run time. It follows that the derivatives cannot be computed at compile time. But the DAG only depends on the expression, and, to state the obvious, the expression, written in C++ code, is very much known at compile time. It follows that there must exist a way to build the DAG at compile time, saving the cost of DAG creation at run time, as long as the programming language provides the necessary constructs. It so happens that this can be achieved, very practically, with an application of the template metaprogramming facilities of standard C++. The only work that remains at

run time is to propagate adjoints to compute derivatives, in the reverse order, through the DAG, and this is very quick because compile-time DAGs are constructed on the stack memory, and the adjoint propagation sequence is generally inlined and further optimized by the compiler.

In our example, the value and the partial derivatives of $y$ depend on the array $x$, which is unknown at compile time, and may depend on user input. The value and derivative of the expression may only be computed at run time. But we do have valuable information at compile time: we know the number of inputs (five, out of which one, $x[4]$, is inactive) and we know *the sequence of operations* that produce $y$ out of $x$, so we can produce its DAG, illustrated above, while the code is being compiled.

The technology that implements these notions is known as *expression templates*. This is a well-known idiom in advanced C++ that has been successfully applied to accelerate mathematical code in many contexts, like high-performance linear algebra libraries.

It follows that the differentials of the expressions that constitute a calculation are still recorded on tape, and propagated in reverse order, to accumulate the derivatives of the calculation to its inputs in constant time. But the tape doesn't record the individual operations that constitute the expressions; it records the expressions themselves, which results in a shorter tape storing a lower number of records. What is new is that the differentials of the expressions themselves, also computed automatically and in constant time with a reverse adjoint propagation over their constituent operations, are accumulated without a tape and without the related overhead, with the help of expression templates. The next section explains this technology and its application to a tapeless, constant time differentiation of the expressions.

## 15.2 EXPRESSION TEMPLATES

## Template meta-programming

C++ templates offer vastly more than their common use as a placeholder for types in generic classes and methods. C++ templates may be applied to meta-programming, that is, very literally, code that generates code. Templates are resolved into template-free code *before compilation*, when all templates are *instantiated*, so it is the instantiated code which is ultimately translated into machine language by the compiler. Because the translation from the templated code into the instantiated code occurs at compile time, that transformation is free from any run-time overhead. This application of C++ templates to write general code that is transformed, at instantiation time, into specific code to be compiled into machine code, is called *template meta-programming*. Well-designed meta-programming can produce elegant, readable code where the necessary boilerplate is generated on instantiation. It can also produce extremely efficient code, where part of the work is conducted on instantiation, hence, at compile time.

The classic example is the compile-time factorial:

```
1  template <unsigned n>
2  struct factorial
3  {
4      enum { value = n * factorial<n - 1>::value };
5  };
6
7  template <>
8  struct factorial<0>
9  {
10     enum { value = 1 };
11 };
12
13 int main()
14 {
15     cout << factorial<5>::value << endl;   // 120
16 }
```

The factorial is computed at compile time. The instantiated code that is compiled and executed is simply:

```
int main()
{
    cout << 120 << endl;    // 120
}
```

This traditional C++ code uses the fact that enums are resolved at compile time, like templates, sizeof(), and functions marked constexpr. We will use these to figure the number of the active arguments of an expression at compile time.

Template meta-programming is probably the most advanced form of C++ programming. It is a fascinating area, only available in C++. The most complete reference in this field is Vandevoorde et al.'s C++ Templates – The Complete Guide, recently updated in modern C++ in [76]. Motivated readers may find extensive information on template meta-programming, including expression templates, in this publication. Our introduction barely scratches the surface.

## Static polymorphism and CRTP

Expression templates implement the CRTP (curiously recursive template pattern) idiom, so to understand expression templates, we must introduce CRTP first.

Classical object-oriented polymorphism offers a neat separation between interface and implementation, at the cost of run-time overhead, a textbook example being:

```
class Animal
{
public:

    virtual void makeNoise() const = 0;
    virtual ~Animal() {}
};

class Duck : public Animal
{
public:

    void makeNoise()  const override
    {
        cout << "Quack!" << endl;
    }
};

class Dog : public Animal
{
public:

    void makeNoise() const override
    {
        cout << "Woof!" << endl;
    }
};

inline void speakTo(const Animal& animal)
{
    animal.makeNoise();
}

int main()
{
    unique_ptr<Animal> donald = make_unique<Duck>();
    unique_ptr<Animal> goofy = make_unique<Dog>();

    speakTo(*donald);    //  Quack
    speakTo(*goofy);     //  Woof

}
```

We applied this pattern to build DAGs with polymorphic nodes in Chapter 9. Edges were represented by base node pointers, and concrete nodes represented the operators that combine nodes to produce an expression: $+$, $*$, $log()$, $sqrt()$, etc. Concrete nodes overrode methods to evaluate or differentiate operators, providing the means of recursively evaluating or differentiating the whole expression. To apply polymorphism to graphs in this manner is a well-known design pattern, applicable in many contexts and identified in GOF [25] as the *composite pattern*.

What is perhaps less well known is that polymorphism can be also be achieved at compile time, saving run-time allocation, creation, and resolution overhead, as long as the concrete type is known at compile time, as opposed to, say, dependent on user input:

```
template <class A>
class Animal
{
public:
    void makeNoise() const
    {
        //  Downcast, we know it's a A
        const A& a = static_cast<const A&>(*this);
        //  Call makeNoise on class A
        a.makeNoise();
    }
};

//  CRTP
class Duck : public Animal<Duck>
{
public:
    void makeNoise() const
    {
        cout << "Couac!" << endl;
    }
};

class Dog : public Animal<Dog>
{
public:
    void makeNoise() const
    {
        cout << "Waf!" << endl;
    }
};

template <class A>
inline void speakTo(const Animal<A>& animal)
{
    animal.makeNoise();
}

int main()
{
    Duck donald;
    Dog goofy;

    speakTo(donald);     // Couac
    speakTo(goofy);      //  Waf
}
```

We have the same results as before, without vtable pointers or allocations. Nothing is virtual in the code above; execution is conducted on the stack with resolution at compile time. The base class must know the derived class so it can downcast itself in the “virtual” function call

on line 8 and call the correct function defined on the “concrete” class, not with run-time *overriding*, but with compile time *overloading*. It follows that the base class must be templated on the derived class, hence the curious syntax on line 15 where a class derives another class templated on itself. This syntax is what permits compile-time polymorphism, and it may appear awkward at first sight, hence the name “curiously recursive template pattern,” or CRTP.

Static polymorphism can achieve many behaviors of classical run-time polymorphism, without the associated overhead. Our animals bark and quack all the same, only faster: the identification of animals as ducks or dogs (resolution) occurs at compile time, and they live on the stack (static memory), not on the heap (dynamic memory), so they are typically accessed faster. A compiler will generally inline the call on lines 44 and 45, whereas it is rarely able to inline true virtual calls. This being said, static polymorphism cannot do everything dynamic polymorphism does. If it were the case, and since static polymorphism is always faster, language support for dynamic polymorphism would be unnecessary. In particular, resolution may only occur at compile time if the concrete type is known at compile time. Our simulation library of Chapter 6 implemented virtual polymorphism so users can mix and match models, products, and RNGs at run time. This cannot be achieved with CRTP. To apply compile time polymorphism, the concrete type of the objects must be static and cannot depend on user input.

As far as C++ expressions are concerned, concrete expression types are compile time constants. For example, in the expression code on page 504, the operation on line 5 is a $log()$. The sequence of operations on line 4 is $*$, $+$, $*$. The concrete types of the operations is known at compile time, so we can construct the DAG with CRTP in place of virtual mechanisms. As a counterexample, in our publication [11], we build and evaluate DAGs that determine a product's cash-flows from a user-supplied script. In this case, the sequence of the concrete operations is not known at compile time, so CRTP cannot be applied.

Finally, CRTP is type safe, like run-time polymorphism, in the sense that $speakTo()$ only accepts Animals as arguments. CRTP is different from simple templates, like:

```
1  template <class T>
2  inline void speakTo(const T& t)
3  {
4      t.makeNoise();
5  }
```

This template function catches any argument and compiles as long as its type implements a method $makeNoise()$. This may be convenient in specific cases, but this is not CRTP. This is actually not any kind of polymorphism. And this is not what we need for expression templates. In order to build a compile-time DAG, like the run-time DAGs of Chapter 9, we need operator overloads that catch all kinds of expressions, *but only expressions and nothing else*. This cannot be achieved with simple templates; it requires a proper type hierarchy, either virtual or CRTP.

## Building expressions

Expression templates are a modern counterpart of the composite pattern. They represent expressions in a CRTP hierarchy, which, combined with operator overloading, turns expressions into DAGs at compile time.

An expression is an essentially recursive thing: by definition, an expression consists in one or multiple expressions combined with an operator. A number is an elementary expression. For example, in the last line of the code on page 504:

$$y = (y_1 + x_3 y_2)(y_1 + y_2)$$

$y_1$, $y_2$, and $x_3$ are (leaf) expressions, $x_3 y_2$ is the expression that applies the operator $*$ to the expressions $x_3$ and $y_2$. The left-hand-side factor is the expression that applies the operator $+$ to the sub-expres-

sions $y_1$ and $x_3y_2$, and $y$ is the expression that applies $*$ to the sub-expressions $y_1 + x_3y_2$ and $y_1 + y_2$, the latter being an expression that applies $+$ to the (leaf) expressions $y_1$ and $y_2$.

We used this recursive definition in Chapter 9 to develop a classical object-oriented hierarchy of expressions (which we called nodes). We can do the same here with CRTP:

```
template <class E>
class Expression
{};

//  Times

//  CRTP
template <class LHS, class RHS>
class ExprTimes : public Expression<ExprTimes<LHS, RHS>>
{
    LHS lhs;
    RHS rhs;

public:

    //  Constructor
    explicit ExprTimes
        (const Expression<LHS>& l, const Expression<RHS>& r)
        : lhs(static_cast<const LHS&>(l)),
        rhs(static_cast<const RHS&>(r)) {}

    double value() const
    {
        return lhs.value() * rhs.value();
    }
};

//  Operator overload for expressions
template <class LHS, class RHS>
inline ExprTimes<LHS, RHS> operator*(
    const Expression<LHS>& lhs, const Expression<RHS>& rhs)
{
    return ExprTimes<LHS, RHS>(lhs, rhs);
}
```

The base class is empty. We need it to build a CRTP hierarchy, so the overloaded operators catch all expressions and nothing else. The concrete *ExprTimes* expression represents the multiplication of two expressions. It is constructed from two expressions, storing them in accordance with their concrete type after a static cast in the constructor. The overloaded operator $*$ takes two expressions (all types of expres-

sions but nothing other than expressions) and constructs the corresponding *ExprTimes* expression.

We must code expression types and operator overloads for all standard mathematical functions. We will not do this in this section, where the code only demonstrates and explains the expression template technology. Our actual AADET code does implement all standard functions, including the Gaussian density and cumulative distribution, as discussed in Chapter 12. For now, we only consider multiplication and logarithm:

```
1  //  Log
2
3  template <class ARG>
4  class ExprLog : public Expression<ExprLog<ARG>>
5  {
6      ARG arg;
7
8  public:
9
10     //  Constructor
11     explicit ExprLog(const Expression<ARG>& a)
12         : arg(static_cast<const ARG&>(a)) {}
13
14     double value() const
15     {
16         return log(arg.value());
17     }
18 };
19
20 //  Operator overload for expressions
21 template <class ARG>
22 inline ExprLog<ARG> log(const Expression<ARG>& arg)
23 {
24     return ExprLog<ARG>(arg);
25 }
```

Finally, we have our custom number type, which is also an expression, although a special one because it is a leaf in the DAG:

```
// Number type, also an expression
class Number : public Expression<Number>
{
    double val;

public:

    // Constructor
    explicit Number(const double v) : val(v) {}

    double value() const
    {
        return val;
    }

};
```

We can test our toy code and implement a lazy evaluation:

```
template <class T>
auto calculate(const T t1, const T t2)
{
    return t1 * log(t2);
}

int main()
{
    Number x1(2.0), x2(3.0);

    auto e = calculate(x1, x2);

    cout << e.value() << endl;  // 2.19722 = 2 * log(3)
}
```

We achieved the exact same result as in Chapter 9. The call to $calculate()$ did not calculate anything. It built the DAG for the calculation. The code in $calculate()$, instantiated with the Number type, is:

```
auto calculate(const Number t1, const Number t2)
{
    return t1 * log(t2);
}
```

which is nothing else than syntactic sugar for:

```
ExprTimes<Number,ExprLog<Number>> calculate(
    const Number t1,
    const Number t2)
{
    return ExprTimes<Number,ExprLog<Number>>(t1, ExprLog<Number>(t2));
}
```

The reason is that $t2$ is a Number, hence an Expression. Therefore, the resolution of $log(t2)$ is caught in the $log()$ overload for expressions, producing $ExprLog < Number > (t2)$, which is itself an Expression, as is $t_1$, so the multiplication is also caught in the overload for expressions, resulting in a type $ExprTimes < Number, ExprLog < Number >>$, constructed with the left-hand-side $t_1$ and the right-hand-side $ExprLog < Number > (t2)$.

Provided the compiler correctly inlines the calls to the overloaded operators,[2] the code that is effectively compiled is:

```
1  int main()
2  {
3      Number x1(2.0), x2(3.0);
4
5      //  auto e = calculate(x1, x2);
6      ExprTimes<Number,ExprLog<Number>>
7          e  =  ExprTimes<Number,ExprLog<Number>>(
8               x1,
9               ExprLog<Number>(x2));
10
11     //  cout << e.value() << endl;
12     cout << 2.0 * log(3.0) << endl;
13 }
```

where it is apparent that line 5, which originally said “e = calculate(x1, x2),” does not calculate anything, but builds an *expression tree*, which is another name for the expression's DAG. Furthermore, as evident in the instantiated code above, and provided that the compiler performs the correct inlining, this DAG is built at compile time and stored on the stack. Finally, the compiler always applies operators, overloaded or not, in the correct order, respecting conventional precedence, as well as parentheses, so the resulting DAG is automatically correct.

The (lazy) evaluation is conducted on line 11, from the DAG. The call to $value()$ on the top node evaluates the DAG in postorder, as in Chapter 9. This evaluation happens at run time, but over a DAG prebuilt at compile time, and stored on the stack. No time is wasted constructing the DAG at run time, and evaluation on the stack is typically faster than on the heap. In addition, the compiler should inline and optimize the nested sequence of calls to $value()$ over the expression tree, something the compiler could not do in Chapter 9, where the DAG was not available at compile time.

We applied expression templates to build an expression DAG, like in Chapter 9, but at compile time, on the stack, without allocation or resolution overhead. We effectively achieved the same result while cutting most of the overhead involved, hence the performance gain.

## Traversing expressions

The DAG may have been built at compile time on the stack; it still does everything the run time DAG of Chapter 9 did. We already demonstrated lazy evaluation. We can also count the active numbers in an expression, at compile time, using enums and constexpr functions (functions evaluated at compile time):

```
//  class ExprTimes
    enum { numNumbers = LHS::numNumbers + RHS::numNumbers };

//  class Log
    enum { numNumbers = ARG::numNumbers };

//  class Number
    enum { numNumbers = 1 };

//  constexpr function evaluates at compile time
template <class E>
constexpr auto countNumbersIn(const Expression<E>&)
{
    return E::numNumbers;
}

int main()
{
    Number x1(2.0), x2(3.0);

    auto e = calculate(x1, x2);

    cout << e.value() << endl;  // 2.19722 = 2 * log(3)

    cout << countNumbersIn(e) << endl;    // 2
}
```

The active Numbers are counted at compile time in the exact same way that we computed a compile-time factorial earlier. This information gives us, at compile time, the number of arguments we need for the expression's node on tape.

We can also reverse engineer the original program from the DAG, like we did in Chapter 9:

```
1
2   //  class ExprTimes
3       string writeProgram(
4           //  On input, the number of nodes processed so far
5           //  On return the total number of nodes processed on exit
6           size_t& processed)
7       {
8           //  Process the left sub-DAG
9           const string ls = lhs.writeProgram(processed);
10          const size_t ln = processed - 1;
11
12          //  Process the right sub_DAG
13          const string rs = rhs.writeProgram(processed);
14          const size_t rn = processed - 1;
15
16          //  Process this node
17          const string thisString = ls + rs +
18              "y" + to_string(processed)
19              + " = y" + to_string(ln) + " * y" + to_string(rn) + "\n";
20
21          ++processed;
22
23          return thisString;
24      }
25
26  //  class ExprLog
27      string writeProgram(
28          //  On input, the number of nodes processed so far
29          //  On return the total number of nodes processed on exit
30          size_t& processed)
31      {
32          //  Process the arg sub-DAG
33          const string s = arg.writeProgram(processed);
34          const size_t n = processed - 1;
35
36          //  Process this node
37          const string thisString = s +
38              "y" + to_string(processed)
39              + " = log(y" + to_string(n) + ")\n";
40
41          ++processed;
42
43          return thisString;
44      }
45
46  //  class Number
47      string writeProgram(
48          //  On input, the number of nodes processed so far
49          //  On return the total number of nodes processed on exit
50          size_t& processed)
51      {
52          const string thisString = "y" + to_string(processed)
53              + " = " + to_string(val) + "\n";
54
55          ++processed;
56
57          return thisString;
58      }
59
60
61  int main()
62  {
63      Number x1(2.0), x2(3.0);
64
65      auto e = calculate(x1, x2);
66
67      cout << e.value() << endl;  // 2.19722 = 2 * log(3)
68
69      cout << numNumbersIn(e) << endl;    // 2
```

```
70
71      size_t processed = 0;
72      cout << e.writeProgram(processed);
73  }
```

We get:

- y0 = 2.000000
- y1 = 3.000000
- y2 = log(y1)
- y3 = y0 * y2

### Differentiating expressions in constant time

More importantly, we can apply adjoint propagation through the compile-time DAG to compute in constant time the derivatives of an expression to its active inputs. The mathematics are identical to classical adjoint propagation. Adjoints still accumulate in reverse order over the operations that constitute the expression. The difference is that this reverse adjoint propagation is conducted over the expression's DAG built at compile time, saving the cost of building and traversing a tape in dynamic memory. It is only the resulting expression derivatives that are stored on tape, so that the adjoints of the entire calculation, consisting of a sequence of expressions, may be accumulated over a shorter tape in reverse order at a later stage.

At this point, we must examine the expression's DAG in further detail. This DAG is actually different in nature to the DAGs of Chapter 9. Our DAG here is a *tree*, in the sense that every node has only one parent. When an expression uses the same number multiple times, as in:

```
Number x, y;
//  ...
Number z = x * x * y;
```

each instance (in the sense that there are two instances of $x$ in the expression above) is a different node in the expression tree, although both point to the same node on tape. When we compute the expression's derivatives, we produce separate derivatives for the two $x$s. But on tape, the two child adjoint pointers refer to the same adjoint. Therefore, at propagation time, the two are summed up into the adjoint on $x$'s node, so in the end the adjoint for $x$ accumulates correctly.

The expression's DAG being a tree makes it easier to “push” adjoints top down, applying basic adjoint mathematics as in the previous chapters, from the top node, whose adjoint is one, to the leaves (Numbers), which store the adjoints propagated through the tree.

```
//  class ExprTimes
    //  Input: accumulated adjoint for this node or 1 if top node
    void pushAdjoint(const double adjoint)
    {
        lhs.pushAdjoint(adjoint * rhs.value());
        rhs.pushAdjoint(adjoint * lhs.value());
    }

//  class Log
    //  Input: accumulated adjoint for this node or 1 if top node
    void pushAdjoint(const double adjoint)
    {
        arg.pushAdjoint(adjoint / arg.value());
    }


//  class Number
class Number : public Expression<Number>
{
    double val;
    double adj;

public:

    //  Constructor
    explicit Number(const double v) : val(v), adj(0.0) {}

    double value() const
    {
        return val;
    }

    double adjoint() const
    {
        return adj;
    }

//  ...

    void pushAdjoint(const double adjoint)
    {
        adj = adjoint;
    }

int main()
{
    Number x1(2.0), x2(3.0);

    auto e = calculate(x1, x2);

    cout << e.value() << endl;  // 2.19722 = 2 * log(3)

    cout << numNumbersIn(e) << endl;    // 2

    size_t processed = 0;
    cout << e.writeProgram(processed);

    e.pushAdjoint(1.0);
    cout << "x1 adjoint = " << x1.adjoint() << endl;    //  1.09861 = log(3)
    cout << "x2 adjoint = " << x2.adjoint() << endl;    //  0.666667 = 2/3
}
```

Note that $pushAdjoint()$ is *overloaded* for different expression types. Contrarily to virtual overriding, overloading is resolved at compile

time, without run-time overhead. This function accepts the node's adjoint as argument, implements the adjoint equation to compute the adjoints of the child nodes, and recursively pushes them down the child expression sub-trees by calling $pushAdjoint()$ on the child expressions. The recursion starts on the top node of the expression tree, the result of the expression, with argument 1, and stops on Numbers, leaf nodes without children, which register their adjoint, given as argument, in their internal data member. The actual AADET code implements the same pattern, where Number leaves register adjoints on the expression's node on tape.

## Flattening expressions

This tutorial demonstrated how expression templates work, how they are applied to produce compile-time expression trees, and how these trees are traversed at run time to compute the derivatives of an expression to its active numbers so we can construct the expression's node on tape.

We know how to efficiently compute the derivatives of an *expression*, but a *calculation* is a *sequence* of expressions. All expressions are still recorded on tape, and their adjoints are still back-propagated through the tape to compute the differentials of the whole calculation.

To work with expression trees is faster than working with the tape: DAGs are constructed at compile time, saving run-time overhead; computations are conducted in faster, static memory on the stack, and optimized by the compiler because they are known at compile time. In principle, it is best working as much as possible with expression trees and as little as possible with the tape. In practice, it is generally impossible to represent a whole calculation as a single expression. Variables of expression types cannot be overwritten, since different expressions are instances of different types. Expressions cannot grow through control flow like loops. In general, an expression is contained in one line

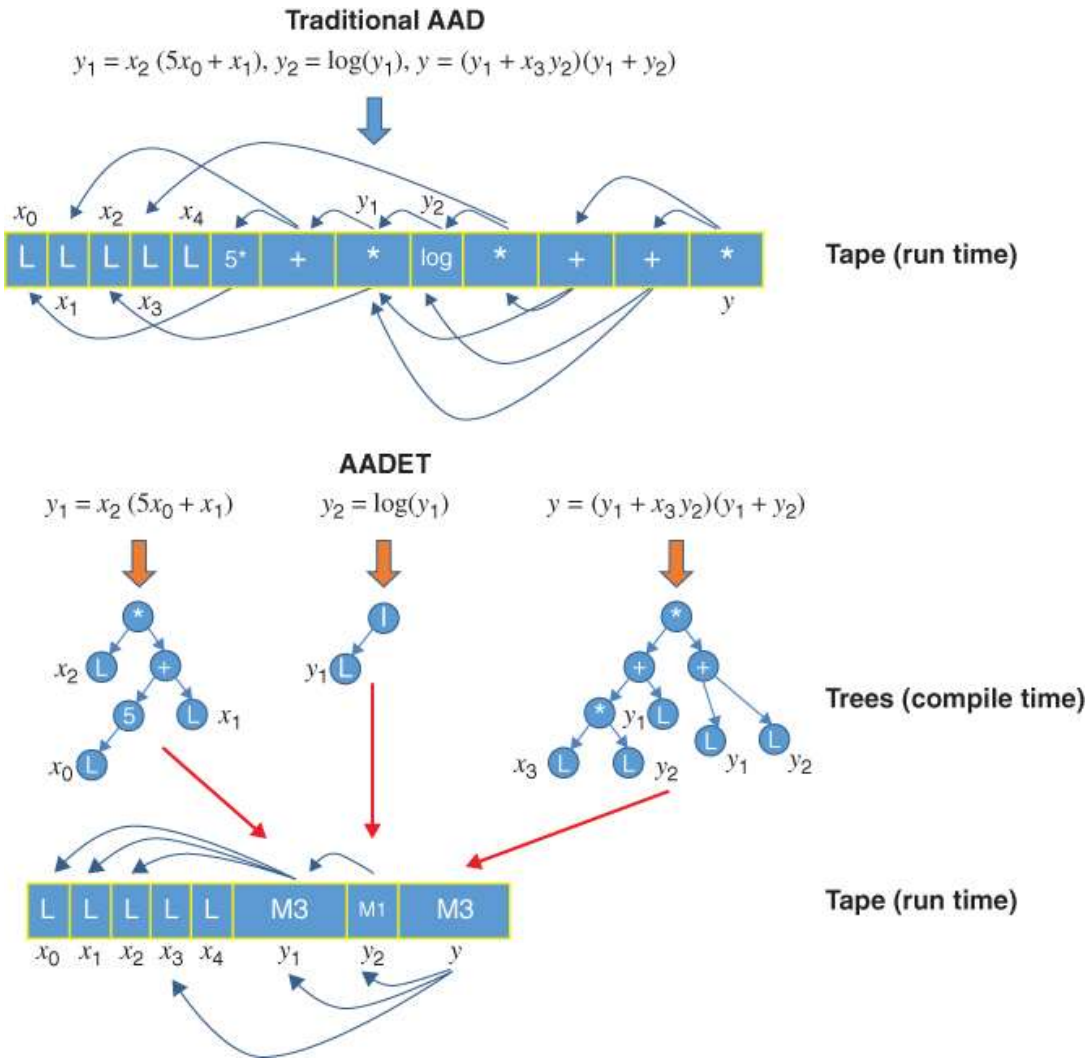
of code. For instance, the example we used earlier (here instantiated with the Number type):

```
1  Number f(Number x[5])
2  {
3      Number y1 = x[2] * (5.0 * x[0] + x[1]);
4      Number y2 = log(y1);
5      Number y = (y1 + x[3] * y2) * (y1 + y2);
6      return y;
7  }
```

consists in three expressions. The three lines assign expressions of different types on the right-hand side, into Numbers on the left-hand side. From the moment an expression is assigned to a Number (which is itself a leaf expression), the expression is *flattened* into a single Number and no longer exists as a compound expression. This is when the expression and its derivatives are evaluated, and where the expression is recorded on tape, on a node that is assigned to the left-hand-side Number's $myNode$ pointer. The flattening code will be discussed shortly, with the AADET library. It is implemented in the Number's assignment operator with an Expression argument:

```
1  //  On the Number class
2      template <class E>
3      Number& operator=
4          (const Expression<E>& e)
5      {
6          myNode = evaluateFlattenAndRecord(e);
7          return *this;
8      }
```

AADET in general, and flattening in particular, are illustrated in the figure below:




This graphical representation should clarify how AADET works and why it produces such a considerable acceleration. Expressions adjoints are back-propagated over trees that live on the stack and are constructed at compile time, so only their results live on tape. A large part of the adjoint propagation occurs on faster grounds. We want as much propagation as possible to be performed over expression trees and as little as possible on tape.

Hence, we want expressions to grow. In order to keep the expression growing as much as possible, the instrumented code should use *auto* as the result type for expressions, so the resulting variable is of type Expression, not Number, no flattening occurs, and the expression keeps growing. It is only on assignment *into a Number* that the expression is flattened and a multinomial node is created on tape. The same code rewritten with auto:

```
Number f(Number x[5])
{
    auto y1 = x[2] * (5.0 * x[0] + x[1]);
    auto y2 = log(y1);
    auto y = (y1 + x[3] * y2) * (y1 + y2);
    return y;
}
```

creates only one node on tape (with four arguments) and therefore differentiates faster. Flattening occurs on return when the expression $y$ is assigned into a temporary Number returned by $f()$. Expression templates offer an additional optimization opportunity to instrumented code by systematically using *auto* as the type to hold expression results. Importantly, calls to functions with templated arguments and auto return type also result in expressions, so yet another version of $f()$:

```
auto f(Number x[5])
{
    auto y1 = x[2] * (5.0 * x[0] + x[1]);
    auto y2 = log(y1);
    auto y = (y1 + x[3] * y2) * (y1 + y2);
    return y;
}
```

is even more efficient because $f()$ returns an expression that may further grow outside of $f()$, in the caller code.

What we cannot do is traverse control flow without flattening expressions, because the overwritten variables would be of different expression types, as mentioned earlier. For instance, the following code would not compile:

```
Number f(Number x[5])
{
    auto y = x[2] * (5.0 * x[0] + x[1]);
    y = log(y1);                          //  error: different type
    y = (y1 + x[3] * y2) * (y1 + y2);     //  error: different type
    return y;
}
```

We cannot overwrite variables of expression types with different expression types. Every auto variable must be a new variable. In particular, variables overwritten in a loop must be of Number type, which means we cannot grow expressions through loops:

```
Number f(Number x[5])
{
    /*  Error

    auto s = x[0] + x[1];   //  type = +(Number, Number)

    for (size_t i = 2; i < 5; ++i)
    {
        s += x[i]; //  on 1st iter type = +((+Number,Number),Number) etc.
    }

    */

    //  Correct

    Number s = x[0] + x[1];   //  flattened into Number

    for (size_t i = 2; i < 5; ++i)
    {
        s += x[i]; //  flattened into Number on every iteration
    }

    return y;
}
```

The flattening code in $evaluateFlattenAndRecord()$ (which is not actually called that) is given, along with the rest of the AADET code, in the next section.

## 15.3 EXPRESSION TEMPLATED AAD CODE

In this final section, we deliver and discuss the AADET code in AADExpr.h. It is hoped that the extended tutorial of the previous section helps readers read and understand this code. Template meta-programming in general, and expression templates in particular, are advanced C++ idioms, notoriously hard to understand or explain. It is our hope that we have given enough background to clarify what follows.

We start with a base expression class that implements CRTP for evaluation:

```
1  //  Base CRTP class so operators catch expressions
2  template <class E>
3  struct Expression
4  {
5      double value() const
6      {
7          return static_cast<const E*>(this)->value();
8      }
9
10     explicit operator double() const
11     {
12         return value();
13     }
14 };
```

Next, we have a single CRTP expression class to handle *all binary operations*:

```
1  //  Binary expression
2  //  LHS : the expression on the left
3  //  RHS : the expression on the right
4  //  OP : the binary function
5  template <class LHS, class RHS, class OP>
6  class BinaryExpression
7      //  CRTP
8      : public Expression<BinaryExpression<LHS, RHS, OP>>
9  {
10     const double myValue;
11
12     const LHS lhs;
13     const RHS rhs;
14
15 public:
16
17     //  Constructor out of 2 expressions
18     //  Note : eager evaluation on construction
19     explicit BinaryExpression(
20         const Expression<LHS>& l,
21         const Expression<RHS>& r)
22         : myValue(OP::eval(l.value(), r.value())),
23         lhs(static_cast<const LHS&>(l)),
24         rhs(static_cast<const RHS&>(r))
25     {}
26
27     //  Value accessors
28     double value() const { return myValue; }
29
30     //  Expression template magic
31     //  Expressions know
32     //      AT COMPILE TIME
33     //      the count of Numbers in their sub-DAG
34     enum { numNumbers = LHS::numNumbers + RHS::numNumbers };
35
36     //  Push adjoint down the DAG
37     //  And write information into the node for the expression
38     //  N : total number of numbers in the expression
39     //  n : numbers already processed
40     template <size_t N, size_t n>
41     void pushAdjoint(
42         //  Node for the complete expression being processed
43         Node&       exprNode,
44         //  Adjoint accumulated for this binary node
45         const double    adjoint)
46         const
47     {
48         //  Push on the left, if numbers there
49         if (LHS::numNumbers > 0)
50         {
51             lhs.pushAdjoint<N, n>(
52                 exprNode,
53                 adjoint * OP::leftDerivative(
54                                 lhs.value(),
55                                 rhs.value(),
56                                 value()));
57         }
58
59         //  Push on the right
60         if (RHS::numNumbers > 0)
61         {
62             //  Note left push processed LHS::numNumbers numbers
63             //  So the next number to be processed is n + LHS::numNumbers
64             rhs.pushAdjoint<N, n + LHS::numNumbers>(
65                 exprNode,
66                 adjoint * OP::rightDerivative(
67                                 lhs.value(),
68                                 rhs.value(),
69                                 value()));
```

```
70          }
71      }
72  };
```

This single class handles all binary operations, and delegates the evaluation of the value and derivatives of specific binary operators to its template parameter OP. This is an example of the so-called *policy design*, promoted by Andrei Alexandrescu in [100]. This is a very neat idiom that allows to minimize code duplication and gather the common parts to all binary expressions in one place. It is a modern counterpart to GOF's template pattern in [25].

The CRTP design was discussed earlier. The implementation of $pushAdjoint()$ is more complicated than earlier, and deserves an explanation. It is templated on two numbers: $N$ and $n$. $N$ is the total count of active Numbers in the expression, and $n$ is the count of these active Numbers processed so far. Hence, we start by pushing adjoints onto the left sub-tree with the same $n$, increase $n$ by the number of active Numbers on the left, and process the right sub-tree.

The method takes as an input a reference to the current expression's node on the tape, where the resulting derivatives and pointers are stored when the traversal of the expression is complete. Concretely, the traversal stops on leaves, which are Numbers. The implementation of $pushAdjoint()$ on the Number class, which has nothing to "push," having no children, stores the resulting derivatives on the expression's Node.

Next, we define the different OPs for different binary operations. Their only responsibility is to provide values and derivatives for specific binary operations:

```
//  "Concrete" binaries, we only need to define operations and derivatives
struct OPMult
{
    static const double eval(const double l, const double r)
    {
        return l * r;
    }

    static const double leftDerivative
        (const double l, const double r, const double v)
    {
        return r;
    }

    static const double rightDerivative
        (const double l, const double r, const double v)
    {
        return l;
    }
};

struct OPAdd
{
    static const double eval(const double l, const double r)
    {
        return l + r;
    }

    static const double leftDerivative
        (const double l, const double r, const double v)
    {
        return 1.0;
    }

    static const double rightDerivative
        (const double l, const double r, const double v)
    {
        return 1.0;
    }
};

//  ...
```

Subtraction, division, power, maximum, and minimum are defined identically. Next, we have the binary operator overloads, which build the binary expressions and are no different from our toy code earlier in this chapter:

```
1  //  Operator overloading for binary expressions
2  //  So DAG is built on the stack at compile time
3  //  And traversed at run time for evaluation and propagation
4
5  template <class LHS, class RHS>
6   BinaryExpression<LHS, RHS, OPMult> operator*(
7      const Expression<LHS>& lhs, const Expression<RHS>& rhs)
8  {
9      return BinaryExpression<LHS, RHS, OPMult>(lhs, rhs);
10 }
11
12 template <class LHS, class RHS>
13  BinaryExpression<LHS, RHS, OPAdd> operator+(
14     const Expression<LHS>& lhs, const Expression<RHS>& rhs)
15 {
16     return BinaryExpression<LHS, RHS, OPAdd>(lhs, rhs);
17 }
18
19 //  ...
```

We also have $-$, $/$, $pow()$, $max()$, and $min()$.

Next, with the exact same design, we have the unary expressions:

```
//  Unary expressions : Same logic with one argument

//  The CRTP class
template <class ARG, class OP>
class UnaryExpression
    //  CRTP
    : public Expression<UnaryExpression<ARG, OP>>
{
    const double myValue;

    const ARG arg;
    //  For binary operators with a double on one side
    const double dArg = 0.0;

public:

    //  Constructor
    //  Note : eager evaluation on construction
    explicit UnaryExpression(
        const Expression<ARG>& a) :
            myValue(OP::eval(a.value(), 0.0)),
            arg(static_cast<const ARG&>(a))
        {}

    //  Constructor for binary expressions with a double on one side
    explicit UnaryExpression(
        const Expression<ARG>& a,
        const double b) :
            myValue(OP::eval(a.value(), b)),
            arg(static_cast<const ARG&>(a)), dArg(b)
        {}

    //  Value accessors
    double value() const { return myValue; }

    //  Expression template magic
    enum { numNumbers = ARG::numNumbers };

    //  Push adjoint down the expression DAG
    template <size_t N, size_t n>
    void pushAdjoint(
        //  Node for the complete expression being processed
        Node&       exprNode,
        const double    adjoint)       //  Adjoint accumulated on the node
        const
    {
        //  Push to argument, if numbers there
        if (ARG::numNumbers > 0)
        {
            arg.pushAdjoint<N, n>(
                exprNode,
                adjoint * OP::derivative(arg.value(), value(), dArg));
        }
    }
};
```

```
57  //  The unary operators
58
59  struct OPExp
60  {
61      static const double eval(const double r, const double d)
62      {
63          return exp(r);
64      }
65
66      static const double derivative
67          (const double r, const double v, const double d)
68      {
69          return v;
70      }
71  };
72
73  struct OPLog
74  {
75      static const double eval(const double r, const double d)
76      {
77          return log(r);
78      }
79
80      static const double derivative
81          (const double r, const double v, const double d)
82      {
83          return 1.0 / r;
84      }
85  };
86
87  //  OPSqrt and OPFabs are defined in the same manner
88
89  //  And overloading
90
91  template <class ARG>
92   UnaryExpression<ARG, OPExp> exp(const Expression<ARG>& arg)
93  {
94      return UnaryExpression<ARG, OPExp>(arg);
95  }
96
97  template <class ARG>
98   UnaryExpression<ARG, OPLog> log(const Expression<ARG>& arg)
99  {
100     return UnaryExpression<ARG, OPLog>(arg);
101 }
102
103 //  sqrt() and fabs() are overloaded in the same manner
```

We also implement the same optimization as in Chapter 10: we consider a binary expression with a constant (double) on one side as a unary expression: if $d$ is a double (not a Number, so a "constant" as far as AAD is concerned), then $xd$ or $dx$ are not binary $*$ but a unary $(* \ d)$. Hence, the unary expression class stores that constant, and we have additional OPs and operator overloads for these expressions:

```
//  Binary operators with a double on one side

//  * double or double *
struct OPMultD
{
    static const double eval(const double r, const double d)
    {
        return r * d;
    }

    static const double derivative
    (const double r, const double v, const double d)
    {
        return d;
    }
};

//  double -
struct OPSubDL
{
    static const double eval(const double r, const double d)
    {
        return d - r;
    }

    static const double derivative
    (const double r, const double v, const double d)
    {
        return -1.0;
    }
};

//  - double
struct OPSubDR
{
    static const double eval(const double r, const double d)
    {
        return r - d;
    }

    static const double derivative
    (const double r, const double v, const double d)
    {
        return 1.0;
    }
};

//  We have (many) identical structs for all the binary operators

//  And overloading
```

```
51  //  Binary operators with a double on one side
52
53  template <class ARG>
54   UnaryExpression<ARG, OPMultD> operator*(
55      const double d, const Expression<ARG>& rhs)
56  {
57      return UnaryExpression<ARG, OPMultD>(rhs, d);
58  }
59
60  template <class ARG>
61   UnaryExpression<ARG, OPMultD> operator*(
62      const Expression<ARG>& lhs, const double d)
63  {
64      return UnaryExpression<ARG, OPMultD>(lhs, d);
65  }
66
67  template <class ARG>
68   UnaryExpression<ARG, OPSubDL> operator-(
69      const double d, const Expression<ARG>& rhs)
70  {
71      return UnaryExpression<ARG, OPSubDL>(rhs, d);
72  }
73
74  template <class ARG>
75   UnaryExpression<ARG, OPSubDR> operator-(
76      const Expression<ARG>& lhs, const double d)
77  {
78      return UnaryExpression<ARG, OPSubDR>(lhs, d);
79  }
80
81  //  We have (many) identical overloads for all the binary operators
```

It may happen that we want to include another function or operator to be considered as an elementary building block. For instance, the Normal density and cumulative distribution functions from gaussians.h should be considered as elementary building blocks, even though they are not standard C++ mathematical functions, as discussed in Chapter 10. We pointed out that it is a fair optimization to overload all those frequently called, low-level functions, and consider them as elementary building blocks, rather than instrument them and record their expressions on tape. We saw that in the case of the cumulative normal distribution, in particular, the results are both faster and more accurate. This is how we proceed:

1. Code the corresponding unary or binary operator, with the evaluation and differentiation methods. For the Gaussian functions, operators are unary:

```
struct OPNormalDens
{
    static const double eval(const double r, const double d)
    {
        return normalDens(r);
    }

    static const double derivative
    (const double r, const double v, const double d)
    {
        return - r * v;
    }
};

struct OPNormalCdf
{
    static const double eval(const double r, const double d)
    {
        return normalCdf(r);
    }

    static const double derivative
    (const double r, const double v, const double d)
    {
        return normalDens(r);
    }
};

```

2. Code the overload for the function, which builds a unary or binary expression with the correct operator:

```

template <class ARG>
UnaryExpression<ARG, OPNormalDens>
    normalDens(const Expression<ARG>& arg)
{
    return UnaryExpression<ARG, OPNormalDens>(arg);
}

template <class ARG>
UnaryExpression<ARG, OPNormalCdf>
    normalCdf(const Expression<ARG>& arg)
{
    return UnaryExpression<ARG, OPNormalCdf>(arg);
}

```

This is it. The two snippets of code above effectively made $normalDens()$ and $normalCdf()$ AADET building blocks. There is no need to template the definitions of the original functions (no harm, either; the expression overloads have precedence, being the more specific).

Next, we have comparison operators for expressions:

```
//  Comparison, as normal

template<class E, class F>
 bool operator==(const Expression<E>& lhs, const Expression<F>& rhs)
{
    return lhs.value() == rhs.value();
}
template<class E>
 bool operator==(const Expression<E>& lhs, const double& rhs)
{
    return lhs.value() == rhs;
}
template<class E>
 bool operator==(const double& lhs, const Expression<E>& rhs)
{
    return lhs == rhs.value();
}

template<class E, class F>
 bool operator!=(const Expression<E>& lhs, const Expression<F>& rhs)
{
    return lhs.value() != rhs.value();
}
template<class E>
 bool operator!=(const Expression<E>& lhs, const double& rhs)
{
    return lhs.value() != rhs;
}
template<class E>
 bool operator!=(const double& lhs, const Expression<E>& rhs)
{
    return lhs != rhs.value();
}

template<class E, class F>
 bool operator<(const Expression<E>& lhs, const Expression<F>& rhs)
{
    return lhs.value() < rhs.value();
}
template<class E>
 bool operator<(const Expression<E>& lhs, const double& rhs)
{
    return lhs.value() < rhs;
}
template<class E>
 bool operator<(const double& lhs, const Expression<E>& rhs)
{
    return lhs < rhs.value();
}

//  And all the combinations of <, >, <=, >=, ==, != with either
//      two Numbers
//      a double and a Number
//      a Number and a double
```

and the unary +/- operators:

```
template <class RHS>
UnaryExpression<RHS, OPSubDL> operator-
(const Expression<RHS>& rhs)
{
    return 0.0 - rhs;
}

template <class RHS>
Expression<RHS> operator+
(const Expression<RHS>& rhs)
{
    return rhs;
}
```

This completes all binary and unary operators. Finally, we have the leaf expression, the custom number type:

```
//  The Number type, also an expression

//  CRTP
class Number : public Expression<Number>
{
    //  The value and node for this number, as normal
    double myValue;
    Node* myNode;

    //  Node creation on tape, as normal

    template <size_t N>
    Node* createMultiNode()
    {
        return tape->recordNode<N>();
    }

//  ...
```

The private helper createMultiNode(), templated on the number of arguments to an expression, creates the expression's node on tape. Another private helper, fromExpr(), flattens an expression assigned into a Number, computing derivatives through its DAG and storing them on the expression's multinomial node:

```
//  ...

    //  This is where, on assignment or construction from an expression,
    //      derivatives are pushed through the expression's DAG
    //      into the node
    template<class E>
    void fromExpr(const Expression<E>& e)
    {
        //  Build node
        auto* node = createMultiNode<E::numNumbers>();

        //  Push adjoints through expression DAG from 1 on top
        static_cast<const E&>(e)
            .pushAdjoint<E::numNumbers, 0>(*node, 1.0);

        //  Set my node
        myNode = node;
    }
//  ...
```

We create the node on tape with a call to $createMultiNode()$ , and populate it with a call to $pushAdjoint()$ on the top node of the expression's tree. Note the CRTP style call to $PushAdjoint$ . It is given the template parameters $E::numNumbers$ , the count of the active inputs to the expression, known at compile time as explained earlier, and also given as a template parameter to $createMultiNode()$ , and the count 0 of inputs already processed. It is also given as arguments the address of the expression's node on tape, where derivatives are stored once computed, along with the address of the adjoints of the expression's arguments, and 1.0, the adjoint (in the expression tree, not on tape) of the top node of the expression. Finally, we register the node as this number's node on tape.

We have seen the implementation of $pushAdjoint()$ on the binary and unary expressions, but it is really on the leaves, that is, on the Number class, that $pushAdjoint()$ populates the multinomial tape node. The execution of $pushAdjoint()$ on the Number class occurs on the bottom of the expression tree. In particular, the argument $adjoint$ accumulated the differential of the expression to this $Number$ , so it may be stored on the expression's node:

```
//  ...

public:

    //  Expression template magic
    enum { numNumbers = 1 };

    //  Push adjoint
    //  This is where the derivatives and pointers are set on the node
    //  Push adjoint down the expression DAG

    //  N: count Numbers in the expression
    //  n: index of this Number in the expression
    template <size_t N, size_t n>
    void pushAdjoint(
        //  Node for the complete expression
        Node&           exprNode,
        //  Adjoint of this Number in the expression
        const double    adjoint)
        const
    {
        //  Register child adjoint pointer on the expression's node
        exprNode.pAdjPtrs[n] = Tape::multi
            ? myNode->pAdjoints
            : &myNode->mAdjoint;

        //  Register derivative on the expression's node
        exprNode.pDerivatives[n] = adjoint;
    }

//  ...
```

Next, we have (thread local) static tape access and constructors similar to the code of Chapter 10:

```
//  ...

public:

    //  Static access to tape, as normal
    static thread_local Tape* tape;

    //  Constructors

    Number() {}

    explicit Number(const double val) : myValue(val)
    {
        //  Create leaf
        myNode = createMultiNode<0>();
    }

    Number& operator=(const double val)
    {
        myValue = val;
        myNode = createMultiNode<0>();
        return *this;
    }

    //  No need for copy and assignment
    //  Default ones do the right thing:
    //      copy value and pointer to node

//  ...
```

The new thing is the construction or assignment from an expression, which calls $fromExpr()$ above to flatten the assigned expression and create and populate the multinomial node by traversal of the expression's tree, whenever an expression is assigned to a Number:

```
//  ...

    //  Construct or assign from expression

    template <class E>
    Number(const Expression<E>& e) : myValue(e.value())
    {
        fromExpr<E>(static_cast<const E&>(e));
    }

    template <class E>
    Number& operator=
        (const Expression<E>& e)
    {
        myValue = e.value();
        fromExpr<E>(static_cast<const E&>(e));
        return *this;
    }

//  ...
```

Next, we have the same explicit conversion operators, accessors, and propagators as in traditional AAD code:

```
//  ...

    //  Explicit coversion to double
    explicit operator double& () { return myValue; }
    explicit operator double () const { return myValue; }

    //  All the normal accessors and propagators, as in normal AAD code

    //  Put on tape
    void putOnTape()
    {
        myNode = createMultiNode<0>();
    }

    //  Accessors: value and adjoint

    double& value()
    {
        return myValue;
    }
    double value() const
    {
        return myValue;
    }
    double& adjoint()
    {
        return myNode->adjoint();
    }
    double adjoint() const
    {
        return myNode->adjoint();
    }
    double& adjoint(const size_t n)
    {
        return myNode->adjoint(n);
    }
    double adjoint(const size_t n) const
    {
        return myNode->adjoint(n);
    }

    //  Reset all adjoints on the tape
    //      note we don't use this method
    void resetAdjoints()
    {
        tape->resetAdjoints();
    }

    //  Propagation

    //  Propagate adjoints
    //      from and to both INCLUSIVE
    static void propagateAdjoints(
        Tape::iterator propagateFrom,
        Tape::iterator propagateTo)
    {
        auto it = propagateFrom;
        while (it != propagateTo)
        {
            it->propagateOne();
            --it;
        }
        it->propagateOne();
    }
```

```
    //  Convenient overloads

    //  Set the adjoint on this node to 1,
    //  Then propagate from the node
    void propagateAdjoints(
        //  We start on this number's node
        Tape::iterator propagateTo)
    {
        //  Set this adjoint to 1
        adjoint() = 1.0;
        //  Find node on tape
        auto it = tape->find(myNode);
        //  Reverse and propagate until we hit the stop
        while (it != propagateTo)
        {
            it->propagateOne();
            --it;
        }
        it->propagateOne();
    }

    //  These 2 set the adjoint to 1 on this node
    void propagateToStart()
    {
        propagateAdjoints(tape->begin());
    }
    void propagateToMark()
    {
        propagateAdjoints(tape->markIt());
    }

    //  This one only propagates
    //  Note: propagation starts at mark - 1
    static void propagateMarkToStart()
    {
        propagateAdjoints(prev(tape->markIt()), tape->begin());
    }

    //  Multi-adjoint propagation

    //  Propagate adjoints
    //      from and to both INCLUSIVE
    static void propagateAdjointsMulti(
        Tape::iterator propagateFrom,
        Tape::iterator propagateTo)
    {
        auto it = propagateFrom;
        while (it != propagateTo)
        {
            it->propagateAll();
            --it;
        }
        it->propagateAll();
    }

//  ...
```

and, finally, the unary on-class operators:

```
1   //  ...
2
3       //  Unary operators
4
5       template <class E>
6       Number& operator+=(const Expression<E>& e)
7       {
8           *this = *this + e;
9           return *this;
10      }
11
12      template <class E>
13      Number& operator*=(const Expression<E>& e)
14      {
15          *this = *this * e;
16          return *this;
17      }
18
19      template <class E>
20      Number& operator-=(const Expression<E>& e)
21      {
22          *this = *this - e;
23          return *this;
24      }
25
26      template <class E>
27      Number& operator/=(const Expression<E>& e)
28      {
29          *this = *this / e;
30          return *this;
31      }
32  };
```

which completes the expression-templated AAD code.

As mentioned earlier, we get the exact same results as previously, two to three times faster. Since a small portion of the instrumented code is templated, we actually accelerated AAD by a substantial factor.

Finally, we reiterate that this AAD framework encourages further improvements, both in the AAD library and instrumented code, with potential further acceleration. A systematic application of the *auto* type in the instrumented code delays the flattening of expressions, resulting in a smaller number of larger expressions on tape and a faster overall processing, expressions being processed faster than nodes on tape. It is also worth overloading more functions as expression building blocks, like we did for the Gaussian functions. The AADET framework is meant to evolve with client code and be extended to improve the speed of the entire application.

# NOTES

1 Or more in the multidimensional case of Chapter 14, which requires additional memory for the storage of $m$ adjoints.

2 Which Visual Studio 2017 always does; for avoidance of doubt, we set the project property C/C++ / Optimization / Inline Function Expansion to Any Suitable.