CHAPTER 9 Algorithmic Adjoint Differentiation

In this second introductory chapter, we discuss in detail how to *automatically* generate calculation graphs and conduct adjoint calculations through them to produce all the differentials of any calculation code in constant time, automatically and in a noninvasive manner.

We consider a mathematical calculation that takes a number of scalar inputs $(x_i)_{1 \leq i \leq I}$ and produces a scalar output y. We assume that the calculation is written in C++ (including perhaps nested function calls, overwriting of variables, control flow, and object manipulation). Our goal is to differentiate this calculation code in constant time, like we did in the previous chapter, but automatically. For this purpose, we develop *AAD code*, that is, code that runs adjoint differentiation over any calculation code. In order to do that, the AAD code must understand the sequence of operations involved in the calculation code, a notion known to programmers as "meta-programming" or "introspection." It is achieved through *operator overloading*, whereby all mathematical operators and functions are replaced by custom code when applied to specific types, as opposed to native C++ types like $double$. So the calculation code must be *templated* on the number type, the type used for the representation of numbers. Safe for templating, the calculation

code remains untouched. All this will be clarified throughout the chapter.

The differentiation code in this chapter is only meant for pedagogical purposes. The implementation is incomplete, not optimal, and most of the code and concepts explored here are unnecessary for the implementation of AAD. Its purpose is to understand what exactly is going on inside AAD. We will also see that calculation graphs turn out to be unnecessary for adjoint differentiation, and that, instead, an optimal implementation uses "tapes," data structures that "record" all the mathematical operations involved in the calculation as they are executed. However, it is only by exploring calculation graphs in detail that we understand exactly how AAD works its magic and why tapes naturally appear as the better solution.

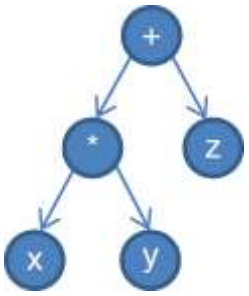
A complete, optimal AAD implementation is delivered in the next chapter, and further improved in the rest of the book. The resulting AAD library is available from our online repository.

9.1 CALCULATION GRAPHS

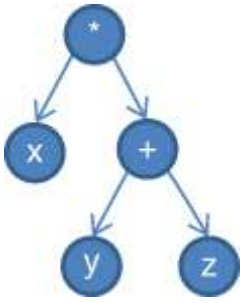
Definition and examples

The first notion we must understand to make sense of AAD is that of a *calculation graph*. Every calculation defines a graph. The *nodes* are all the elementary mathematical calculations involved: $+$, $-$, $*$, $/$, pow, exp, log, $sqrt$, and so forth.[1] The *edges* that join two nodes represent operation to argument relations: *child* nodes are the arguments to the calculation in the *parent* node. The $leaves$ (childless nodes) are scalar numbers and inputs to the calculation.[2]

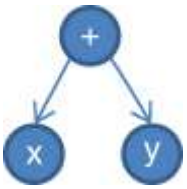
The structure of the graph reflects precedence among operators: for instance the graph for $xy + z$ is:



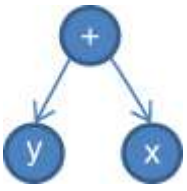
because multiplication has precedence, unless explicitly overwritten with parentheses. The graph for $x(y+z)$ is:



The graph also reflects the order of calculation selected by the compiler among equivalent ones. For instance, the graph for $x+y$ could be:

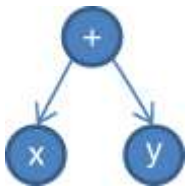


or:

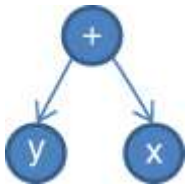


The compiler selects one of the possible calculation orders. Hence, the graph for the calculation *code* is not unique, but the graph for a given execution of the code is, the compiler having made a choice among the equivalent orders.[3] We work with the *unique* graphs that reflect the

compiler's choice of the calculation order, and always represent calculations left to right. Hence, the graph:



represents $x + y$ and:



represents $y + x$.

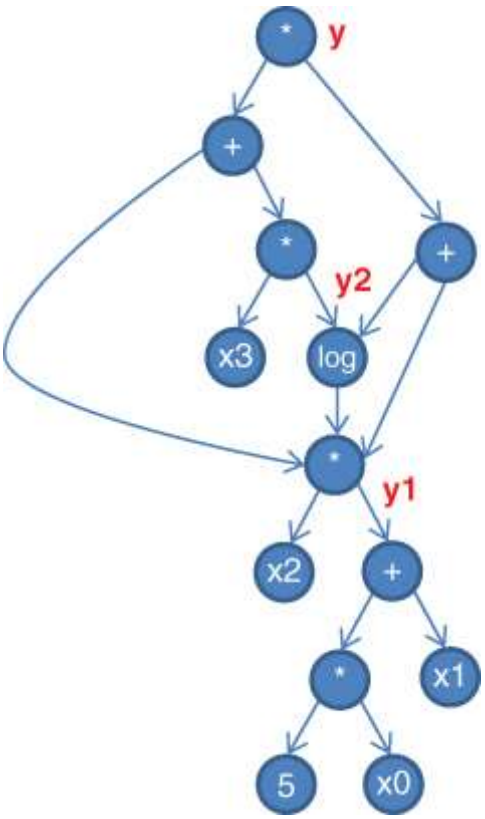
The following example computes a scalar y out of 5 scalars $(x_i)_{0 \leq i \leq 4}$ (where x_4 is *purposely* left out of the calculation) with the formula:

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

We also compute and export the partial derivatives by finite differences for future reference. We will be coming back to this code throughout the chapter.[4]

```
1  double f(double x[5])
2  {
3      double y1 = x[2] * (5.0 * x[0] + x[1]);
4      double y2 = log(y1);
5      double y = (y1 + x[3] * y2) * (y1 + y2);
6      return y;
7  }
8
9  int main()
10 {
11     double x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
12     double y = f(x);
13     cout << y << endl;  //  797.751
14     for (size_t i = 0; i < 5; ++i)
15     {
16         x[i] += 1.e-08;
17         cout << 1.0e+08 * (f(x) - y) << endl;
18         x[i] -= 1.e-08;
19     }   //  950.736, 190.147, 443.677, 73.2041, 0
20 }
```

Its graph (assuming the compiler respects the left-to-right order in the code) is:



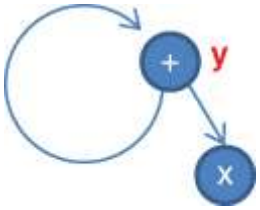
It is important to understand that complicated, compound calculations, even those that use OO (object-oriented) architecture and nested function calls, like the ones that compute the value of an exotic option with MC simulations or FDM, or even the value of the xVA on a large netting set, or any kind of valuation or other mathematical or numerical calculation, all define a similar graph, perhaps with billions of nodes, but a calculation graph all the same, where nodes are elementary operations and edges refer to their arguments.

Calculation graphs and computer programs

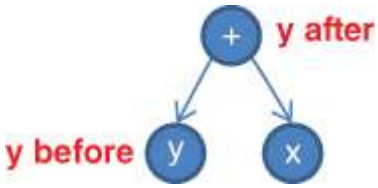
It is also important to understand that calculation graphs refer to *mathematical operations*, irrespective of how computer programs store them in *variables*. For instance, the calculation graph for:

```
y = y + x;
```

is most definitely *not*:



but:

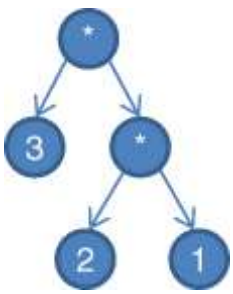


Although the variable y is reused and overwritten in the calculation of the sum, the sum defines a new calculation of its own right in the graph. Graph nodes are calculations, not variables.

Similarly, recursive function calls define new calculations in the associated graph: the following recursive implementation for the factorial:

```
1  unsigned fact(const unsigned n)
2  {
3      return n == 1 ? 1 : n * fact(n - 1);
4  }
```

defines the following graph when called with 3 as argument:



The fact that the function is defined recursively is irrelevant; the graph is only concerned with mathematical operations and their order of execution.

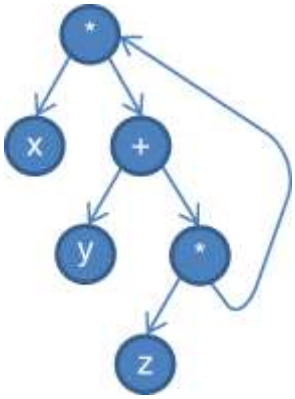
Directed Acyclic Graphs (DAGs)

Computer programs do not define just any type of graph.

First, notice that a calculation graph is not necessarily a *tree*, a particularly simple type of graph most people are used to working with, where every child has exactly one parent.[5] In the example on page 323, the node y_2 has two parents and the node y_1 has three. Calculation graphs are not trees, so reasoning and algorithms that apply to trees don't necessarily apply to them.

Second, the edges are *directed*, calculation to arguments. An edge from y to x means that x is an argument to the calculation of y, like in $y = \log(x)$, which is of course not the same as $x = \log(y)$. Graphs where all edges are directed are called directed graphs. Calculation graphs are always directed.

Finally, in a calculation graph, a node cannot refer to itself, either directly or indirectly. The following graph, for instance:



is *not* a calculation graph, and no code could ever generate a graph like this. The reason is that the arguments to a calculation must be evaluated prior to the calculation.

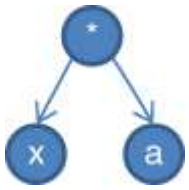
Graphs that satisfy such property are called *acyclic* graphs. Therefore, calculation graphs are *directed acyclic graphs* or DAGs. We will be applying some basic results from the graph theory shortly, so we must identify exactly the class of graphs that qualify as calculation graphs.

DAGs and control flow

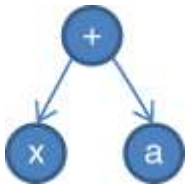
The DAG of a calculation only represents the mathematical operations involved, not the control flow. There are no nodes for if, for, $while$, and friends. A DAG represents the chain of calculations under one particular control branch. For instance, the code

```
y = x > 0 ? x * a : x + a;
```

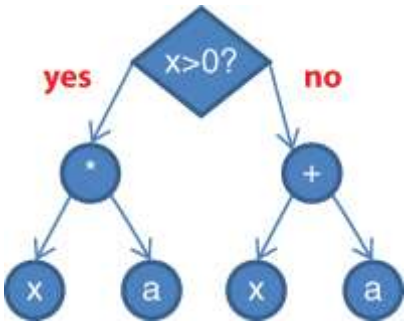
has the DAG:



when executed with a positive input x and the different DAG:



when executed with a negative input. It doesn't have a composite dag of the type:



This is *not* a calculation DAG.

We work with *calculation* DAGs, not *code* DAGs. Calculation DAGs refer to one particular execution of the code, with a specific set of inputs, and a unique control branch, resulting in a specific output. They don't refer to the *code* that produced the calculation. AAD differentiates calculations, not code. It follows that AAD does not differentiate through the control flow itself: since there are no "if" nodes in the calculation DAG, the AAD "derivative" of an "if" statement is 0.

This may look at first sight like a flaw with AAD, but how is any type of differentiation supposed to differentiate through control flow, something that is by nature discontinuous and not differentiable?

We will discuss control flow and discontinuities further in Chapter 11. For now, we focus on building DAGs and applying them for the computation of differentials in constant time.

9.2 BUILDING AND APPLYING DAGS

Building a DAG

A DAG is not just a theoretical representation of some calculation. We apply *operator overloading* to explicitly build a calculation DAG in memory at run time, as follows. This code is incomplete, we only instantiate the $+$, $*$ and log nodes, and *extremely* inefficient because every operation allocates memory for its node on the DAG. We also don't implement const correctness and focus on functionality. This code is

for demonstration purposes only. We suggest readers take the time to fully understand it. This inefficent code nonetheless demonstrates the DNA of AAD.

First, we create classes for the various types of nodes, $+$, $*$, log, and leaf, in an object-oriented hierarchy:

```cpp
#include <memory>
#include <string>
#include <vector>

using namespace std;

class Node
{
protected:

    vector<shared_ptr<Node>> myArguments;

public:

    virtual ~Node() {}
};

class PlusNode : public Node
{
public:

    PlusNode(shared_ptr<Node> lhs, shared_ptr<Node> rhs)
    {
        myArguments.resize(2);
        myArguments[0] = lhs;
        myArguments[1] = rhs;
    }
};

class TimesNode : public Node
{
public:

    TimesNode(shared_ptr<Node> lhs, shared_ptr<Node> rhs)
    {
        myArguments.resize(2);
        myArguments[0] = lhs;
        myArguments[1] = rhs;
    }
};

class LogNode : public Node
{
public:

    LogNode(shared_ptr<Node> arg)
    {
        myArguments.resize(1);
        myArguments[0] = arg;
    }
};

class Leaf: public Node
{

    double myValue;

public:

    Leaf(double val)
        : myValue(val) {}

    double getVal()
    {
        return myValue;
    }

    void setVal(double val)
    {
```

```
70          myValue = val;
71      }
72  };
```

We apply a classical composite pattern (see GOF, [25]) for the nodes in the graph. Concrete nodes are derived for each operation type and hold their arguments by base class pointers. The systematic use of smart pointers guarantees an automatic (if inefficient) memory management. It has to be *shared* pointers because multiple parents may share a child. We can't have a parent release a child while other nodes are still referring to it. When all its parents are gone, the node is released automatically.

Next, we design a custom number type. This type holds a reference to the node on the graph corresponding to the operation last assigned to it. When such a number is initialized, it creates a leaf on the graph. We use this custom type in place of doubles in our calculation code.

```
1   class Number
2   {
3       shared_ptr<Node> myNode;
4
5   public:
6
7       Number(double val)
8           : myNode(new Leaf(val)) {}
9
10      Number(shared_ptr<Node> node)
11          : myNode(node) {}
12
13      shared_ptr<Node> node()
14      {
15          return myNode;
16      }
17
18      void setVal(double val)
19      {
20          //  Cast to leaf, only leaves can be changed
21          dynamic_pointer_cast<Leaf>(myNode)->setVal(val);
22      }
23
24      double getVal()
25      {
26          //  Same comment here, only leaves can be read
27          return dynamic_pointer_cast<Leaf>(myNode)->getVal();
28      }
29  };
```

dynamic_pointer_cast() is a C++11 standard function associated with smart pointers that returns a copy of the smart pointer of the proper type with its stored pointer casted dynamically.[6] This is necessary because only leave nodes store a value.

Next, we *overload* the mathematical operators and functions for the custom number type so that when these operators and functions are executed with this type, the program does not evaluate the operator as normal, but instead constructs the corresponding node on the graph.

```
shared_ptr<Node> operator+(Number lhs, Number rhs)
{
    return shared_ptr<Node>(new PlusNode(lhs.node(), rhs.node()));
}

shared_ptr<Node> operator*(Number lhs, Number rhs)
{
    return shared_ptr<Node>(new TimesNode(lhs.node(), rhs.node()));
}

shared_ptr<Node> log(Number arg)
{
    return shared_ptr<Node>(new LogNode(arg.node()));
}
```

This completes our “AAD” code; we can now build graphs *automatically*. This is where the first A in AAD comes from. We don't build the DAG by hand. The code does it for us when we instantiate it with our number type. We could hard replace “double” by “Number” in the calculation code, or make two copies, one with doubles, one with Numbers. The best practice is *template* the calculation code on its number type, like this:

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

and call it with our custom type like that:

```
1  int main()
2  {
3      Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
4
5      Number y = f(x);
6  }
```

Because we called $f()$ with Numbers in place of doubles, line 5 does *not* calculate anything. All mathematical operations within $f()$ execute our overloaded operators and functions. So, in the end, what this line of code does is construct in memory the DAG for the calculation of $f()$.

We templated the code of $f()$ for its number representation type. To template calculation code in view of running of it with AAD is called *instrumentation*. In Chapter 12, we instrument our simulation code.

Traversing the DAG

Lazy evaluation What can we do with the DAG we just built?

First, we can calculate it. We evaluate the DAG itself, without reference to the function $f()$, the execution of which code built the DAG in the first place.

How do we proceed exactly? First, note that arguments to an operation must be computed before the operation. This means that the processing of a node starts with the processing of its children.

Second, we know that nodes may have more than one parent in a DAG. Those nodes will be processed multiple times if we are not careful. For this reason, we start the processing of a node with a check that this node was not processed previously. If this is verified, then we process its children, and then, and only then, we evaluate the operation on the node itself, mark it as processed, and cache the result of the operation on the node in case it is needed again from a different parent. If the node was previously processed, we simply pick its cached result.

In this sequence, the only step that is specific to evaluation is the execution of the operation on the node. All the rest is graph traversal logic and applies to evaluation as well as other forms of *visits* we explore next.

If we follow these rules, starting with the top node, we are guaranteed to visit (evaluate) each node exactly once and in the correct order for the evaluation of the DAG.

Hence, we have the following graph *traversal* and *visit* algorithm: starting with the top node,

1. Check if the node was already processed. If so, exit.
2. Process the child nodes.
3. Visit the node: conduct the calculation, cache the result.
4. Mark the node as processed.

The third line of the algorithm is the only one specific to evaluation. The rest only relates to the order of the visits. More precisely, the traversal strategy implemented here is well known to the graph theory. It is called “depth-first postorder” or simply *postorder*. Depth-first because it follows every branch down to a leaf before moving on to the next branch. Postorder because the children (arguments) of a node are always visited (evaluated) before the node (operation). It follows that visits start on the leaf nodes and end on the top node. Postorder implements a bottom-up traversal strategy and guarantees that every node in a DAG is processed exactly once, where children are always visited before their parents. It is the natural order for an evaluation. But we will see that other forms of visits may require a different traversal strategy.

We implement our evaluation algorithm, separating traversal and visit logic into different pieces of code. The traversal logic goes to the base class, which also stores a flag that tells if a node was processed and caches the result of its evaluation. For our information only, the order

of calculation is also stored on the (base) nodes. Finally, we code a simple recursive method to reset the processed flags on all nodes.

```
class Node
{
protected:

    vector<shared_ptr<Node>> myArguments;

    bool        myProcessed = false;
    unsigned    myOrder = 0;
    double      myResult;

public:

    virtual ~Node() {}

    //  visitFunc:
    //  a function of Node& that conducts a particular form of visit
    //  templated so we can pass a lambda
    template <class V>
    void postorder(V& visitFunc)
    {
        //  Already processed -> do nothing
        if (myProcessed == false)
        {
            //  Process children first
            for (auto argument : myArguments)
                argument->postorder(visitFunc);

            //  Visit the node
            visitFunc(*this);

            //  Mark as processed
            myProcessed = true;
        }
    }

    //  Access result
    double result()
    {
        return myResult;
    }

    //  Reset processed flags
    void resetProcessed()
    {
        for (auto argument : myArguments) argument->resetProcessed();
        myProcessed = false;
    }
};
```

The $resetProcessed()$ algorithm isn't optimal, as most of the code in this chapter. Next, we code the specific evaluation visitor as a virtual method: evaluation is different for the different concrete nodes (sum in a $+$ node, product in a $*$ node, and so on):

```
//  On the base Node
    virtual void evaluate() = 0;

//  On the + Node
    void evaluate() override
    {
        myResult = myArguments[0]->result() + myArguments[1]->result();
    }

//  On the * Node
    void evaluate() override
    {
        myResult = myArguments[0]->result() * myArguments[1]->result();
    }

//  On the log Node
    void evaluate() override
    {
        myResult = log(myArguments[0]->result());
    }

//  On the Leaf
    void evaluate() override
    {
        myResult = myValue;
    }
```

We start the evaluation of the DAG from the Number that holds the result and, therefore, refers to the top node of the DAG. Recall that Numbers contain a shared pointer to the last assignment.

```
//  On the Number class
    double evaluate()
    {
        myNode->resetProcessed();
        myNode->postorder([](Node& n) {n.evaluate(); });
        return myNode->result();
    }
```

Now we can evaluate the DAG:

```
int main()
{
    Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };

    Number y = f(x);

    cout << y.evaluate() << endl;   // 797.751
}
```

The evaluation of the mathematical operations in the calculation did not happen on line 5. That just constructed the DAG. Nothing was calculated. The calculation happened on line 7 in the call to $evaluate()$, directly from the DAG after it was built. That it returns the exact same result as a direct function call (with double as the number type) indicates that our code is correct.

To see this more clearly, we can change the inputs on the DAG and evaluate it again, without any further call to the function that built the DAG. The new result correctly reflects the change of inputs, as readers may easily check:

```
1  int main()
2  {
3      Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
4
5      Number y = f(x);
6
7      cout << y.evaluate() << endl;    // 797.751
8
9      //  Change x0 on the DAG
10     x[0].setVal(2.5);
11
12     //  Evaluate the dag again
13     cout << y.evaluate() << endl;    // 2769.76
14 }
```

This pattern to store a calculation in the form a DAG, or another form that lends itself to evaluation, instead of conducting the evaluation straightaway (or *eagerly*), and conducting the evaluation at a later time, directly from the DAG, possibly repeatedly and possibly with different inputs set directly on the DAG, is called *lazy evaluation*. Lazy evaluation is what makes AAD automatic; it is also the principle behind, for example, expression templates (see Chapter 15) and scripting (see our publication [11]).

Calculation order There is more we can do with a DAG than evaluate it lazily. For instance, we can store the calculation order number on the node, which will be useful for what follows. We identify the nodes by their calculation order and store it on the node. We reuse our pos-

torder code; we just need a new visit function. In this case, the visits don't depend on the concrete type of the node; we don't need a virtual visit function, and we can code it directly on the base node.

```
//  On the base Node

    void setOrder(unsigned order)
    {
        myOrder = order;
    }

    unsigned order()
    {
        return myOrder;
    }

//  On the Number class

    void setOrder()
    {
        myNode->resetProcessed();
        myNode->postorder(
            [order = 0](Node& n) mutable {n.setOrder(++order); });
    }
```

The lambda defines a data member $order$ that starts at 0. Since C++14, lambdas not only capture environment variables, but can also define other internal variables in the capture clause. The evaluation of the lambda modifies its $order$ member, so the lambda must be marked mutable.

```
int main()
{
    //  Inputs
    Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };

    //  Build DAG
    Number y = f(x);

    //  Number the nodes
    y.setOrder();
}
```

Our DAG is numbered after the execution of the code above, and each node is identified by its postorder. We can log the intermediate results in the evaluation as follows:

```
//  On the Number class

    void logResults()
    {
        myNode->resetProcessed();
        myNode->postorder([] (Node& n)
        {
            cout << "Processed node "
                << n.order() << " result = "
                << n.result() << endl;
        });
    }
```

after evaluation of the DAG. When we evaluate the DAG repeatedly with different inputs, we get different logs accordingly:

```
int main()
{
    //  Set inputs
    Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };

    //  Build the dag
    Number y = f(x);

    //  Set order on the dag
    y.setOrder();

    //  Evaluate on the dag
    cout << y.evaluate() << endl;   // 797.751

    //  Log all results
    y.logResults();

    //  Change x0 on the dag
    x[0].setVal(2.5);

    //  Evaluate the dag again
    cout << y.evaluate() << endl;   // 2769.76

    //  Log results again
    y.logResults();

}
```

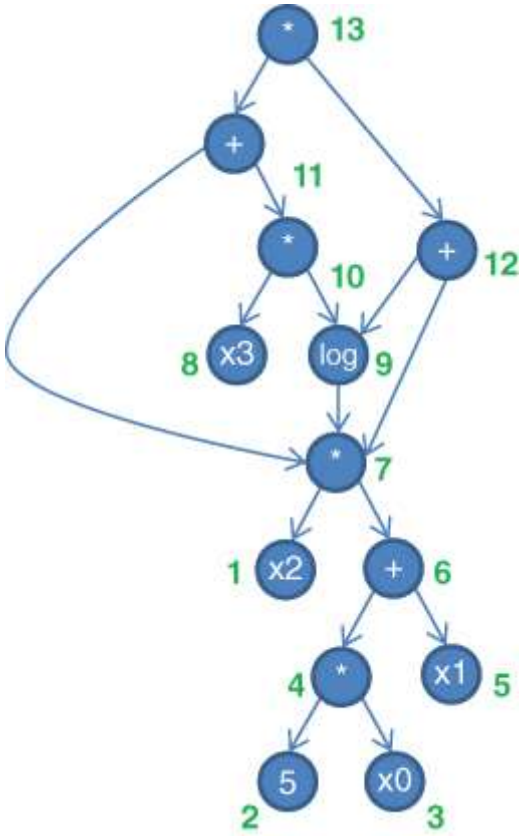
For the first evaluation we get the log:

- Processed node 1 result = 3
- Processed node 2 result = 5
- Processed node 3 result = 1
- Processed node 4 result = 5
- Processed node 5 result = 2
- Processed node 6 result = 7

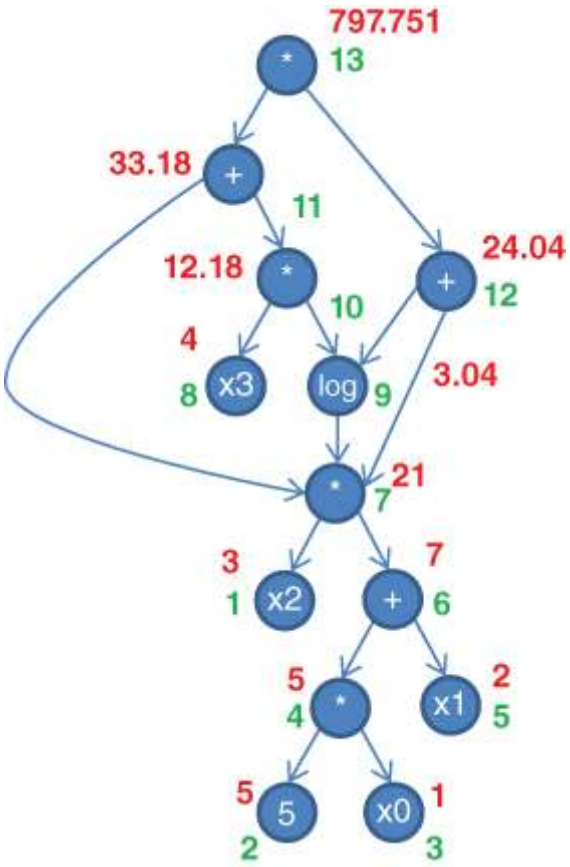
- Processed node 7 result = 21
- Processed node 8 result = 4
- Processed node 9 result = 3.04452
- Processed node 10 result = 12.1781
- Processed node 11 result = 33.1781
- Processed node 12 result = 24.0445
- Processed node 13 result = 797.751

For the second evaluation, we get different logs reflecting $x_0 = 2.5$.

We can display the DAG, this time identifying nodes by their evaluation order:



and intermediate results (for the first evaluation):



Reverse engineering There is even more we can do with the DAG. For instance, we can reverse engineer an equivalent calculation program:

```
//  On the base Node
    virtual void logInstruction() = 0;

//  On the + Node
    void logInstruction() override
    {
        cout << "y" << order()
            << " = y" << myArguments[0]->order()
            << " + y" << myArguments[1]->order()
            << endl;
    }

//  On the * Node
    void logInstruction() override
    {
        cout << "y" << order()
            << " = y" << myArguments[0]->order()
            << " * y" << myArguments[1]->order()
            << endl;
    }

//  On the log Node
    void logInstruction() override
    {
        cout << "y" << order() << " = log("
            << "y" << myArguments[0]->order() << ")" << endl;
    }

//  On the Leaf
    void logInstruction() override
    {
        cout << "y" << order() << " = " << myValue << endl;
    }

//  On the number class
    void logProgram()
    {
        myNode->resetProcessed();
        myNode->postorder([](Node& n) {n.logInstruction(); });
    }

int main()
{
    //  Inputs
    Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };

    //  Build the DAG
    Number y = f(x);

    //  Set order
    y.setOrder();

    //  Log program
    y.logProgram();
}
```

We get the log:

- y1 = 3
- y2 = 5
- y3 = 1

- y4 = y2 * y3
- y5 = 2
- y6 = y4 + y5
- y7 = y1 * y6
- y8 = 4
- y9 = log(y7)
- y10 = y8 * y9
- y11 = y7 + y10
- y12 = y7 + y9
- y13 = y11 * y12

We reconstructed an equivalent computer program from the DAG. Of course, we can change the inputs and generate a new computer program that reflects the new values of the leaves and produces different results.

Note that we managed to make visits (evaluation, instruction logging) depend on both the visitor and the concrete type of the visited node. Different things happen according to the visitor (evaluator, instruction logger) and the node (+, − and so forth). We effectively implemented GOF's *visitor* pattern; see [25]. This pattern is explained in more detail and systematically applied to scripts in our publication [11] (where we study trees, not DAGs, but the logic is the same).

We learned to automatically generate DAGs from templated calculation code with operator overloading. We learned to traverse DAGs in a specific order called postorder and visit its nodes in different ways: lazily evaluate the DAG or reverse engineer an equivalent sequence of instructions.

This covered the “Automatic” in “Automatic Adjoint Differentiation.”[7] We now move on to “Adjoint Differentiation” and use the DAG to calculate *differentials*, all of them, in a single traversal; hence the key constant time property.

Before we do this, we need a detour through basic formalism and adjoint mathematics.

9.3 ADJOINT MATHEMATICS

We must now further formalize and generalize the adjoint mathematics introduced in the previous chapter.

In what follows, we identify the nodes n_i in the DAG by their postorder traversal number. We call y_i the result of the operation on n_i.

We denote $y = y_N$ the final result on the top node n_N.

We denote C_i the set of children (arguments) nodes of n_i.

Mathematical operations are either unary (they take one argument like log or $sqrt$ or the unary $-$ that changes a number into its opposite) or binary (two arguments, like pow or the operators $+$, $-$, $*$, $/$). Hence C_i is a set of either 0, 1, or 2 nodes. Besides, because of postorder, the numbers of the nodes in C_i are always less than i. The only nodes n_i for which C_i is empty are by definition the leaves.

The leaf nodes are processed first in postorder traversal, so their node number i is typically low. Leaves represent the inputs to the calculation.

We define the adjoint of the result y_i on the node n_i and denote a_i the partial derivative of the final result $y = y_N$ to y_i:

$$a_i \equiv \frac{\partial y}{\partial y_i} = \frac{\partial y_N}{\partial y_i}$$

Then, obviously:

$$a_N = \frac{\partial y_N}{\partial y_N} = 1$$

and, directly from the derivative chain rule, we have the fundamental adjoint equation:

$$a_j = \sum_{i/n_j \in C_i} \frac{\partial y_i}{\partial y_j} a_i$$

The adjoint of a node is the sum of the adjoints of its parents (calculations for which that node is an argument) weighted by the partial derivatives of parent to child, operation to argument. The operations on the nodes are elementary operations and their derivatives are known analytically.

Note the reverse order to evaluation: in an evaluation, the result of an operation depends on the values of the arguments. In the adjoint equation, it is the adjoint of the argument that depends on the adjoints of its operations.

Hence, just like evaluation computes results bottom-up, children first, adjoint computation propagates top down, parents first.

More precisely, the adjoint equation turns into the following algorithm that computes the adjoints for all the nodes in the DAG (all the differentials of the final result) and is a direct translation, in graph terms, of the back-propagation algorithm of the previous chapter.

Traversal starts on the top node, which adjoint is known to be 1, and makes its way toward the leaf nodes, which adjoints are the derivatives of the final result to the inputs.

The adjoints of every node are seeded to 0 (except the top node, which is seeded to 1) and the *visit* to node n_i consists in the propagation of its adjoint a_i:

$$a_j += \frac{\partial y_i}{\partial y_j} a_i$$

to the adjoints a_j of all its arguments (child nodes) $n_j \in C_i$. In the terms of the previous chapter, the visit of a node implements the adjoint of the operation represented on the node. After every node has been processed, exactly once and in the correct order, the adjoints of all nodes have correctly accumulated in accordance with the adjoint equation.

As we observed in the previous chapter, $\frac{\partial y_i}{\partial y_j}$ is known analytically, but depends on all the $(y_j)_{n_j \in C_i}$. This means that we must know all the intermediate results before we conduct adjoint propagation. Adjoint propagation is a *post-evaluation* step. It is only after we built the DAG and evaluated it that we can propagate adjoints through it. At this point, all the intermediate results are cached on the corresponding nodes, so the necessary partial derivatives can be computed.[8]

We may now formalize and code adjoint propagation: we set $a_N = 1$ and start with all other adjoints zeroed: for $i < N$, $a_i = 0$. We process all the nodes in the DAG, top to leaves. When visiting a node, we must know its adjoint a_i, so we can implement:

$$a_j += \frac{\partial y_i}{\partial y_j} a_i$$

for all its kids $j \in C_i$. This effectively computes the adjoints of all the nodes in the DAG, hence all the partial derivatives of the final result, with a single DAG traversal hence, in constant time.

This point is worth repeating, since the purpose of AAD is constant time differentiation. All the adjoints of the calculation are accumulated from the top-down traversal of its DAG. In this traversal, every node is visited once and its adjoints are propagated to its child nodes, weighted by local derivatives. Hence, the adjoint accumulation, like the evaluation, consists in one visit to every node (operation) on the calculation DAG. Adjoint accumulation, which effectively computes all

derivative sensitivities, is of the same order of complexity as the evaluation and constant in the number of inputs.

It is easy enough to code the visit routines. We store adjoints on the nodes and write methods to set and access them, like we did for evaluation results. We also need a method to reset all adjoints to 0, like we did for the processed flag. Finally, we log the visits so we can see what is going on.

```
//  On the base Node
    double      myAdjoint = 0.0;

    //  ...

public:

    //  ...

    //  Note: return by reference
    //  used to get and set
    double& adjoint()
    {
        return myAdjoint;
    }

    void resetAdjoints()
    {
        for (auto argument : myArguments) argument->resetAdjoints();
        myAdjoint = 0.0;
    }

    virtual void propagateAdjoint() = 0;

//  On the + Node
    void propagateAdjoint() override
    {
        cout << "Propagating node " << myOrder
            << " adjoint = " << myAdjoint << endl;

        myArguments[0]->adjoint() += myAdjoint;
        myArguments[1]->adjoint() += myAdjoint;
    }

//  On the * Node
    void propagateAdjoint() override
    {
        cout << "Propagating node " << myOrder
            << " adjoint = " << myAdjoint << endl;

        myArguments[0]->adjoint() += myAdjoint * myArguments[1]->result();
        myArguments[1]->adjoint() += myAdjoint * myArguments[0]->result();
    }

//  On the log Node
    void propagateAdjoint() override
    {
        cout << "Propagating node " << myOrder
            << " adjoint = " << myAdjoint << endl;

        myArguments[0]->adjoint() += myAdjoint / myArguments[0]->result();
    }

//  On the Leaf
    void propagateAdjoint() override
    {
        cout << "Accumulating leaf " << myOrder
            << " adjoint = " << myAdjoint << endl;
    }

//  On the number class

    //  Accessor/setter, from the inputs
    double& adjoint()
    {
        return myNode->adjoint();
    }

    //  Propagator, from the result
```

```
70      void propagateAdjoints()
71      {
72          myNode->resetAdjoints();
73          myNode->adjoint() = 1.0;
74          //  At this point, we must propagate sensitivities,
75          //      but how exactly do we do that?
76      }
```

Notice how the propagation code uses the intermediate results stored on the child nodes to compute partial derivatives, as mentioned earlier. The propagation of adjoints on the unary and binary operator nodes implements the adjoint equation. Leaf nodes don't propagate anything; by the time propagation hits them, their adjoints are entirely accumulated, so their override of $propagateAdjoints()$ simply logs the results.

We could easily code the visits but not start the propagation. It is clear that doing the same as for evaluation, something like:

```
myNode->postorder([](Node& n) {n.propagateAdjoint(); });
```

is not going to fly. Postorder traverses the DAG arguments first, but for adjoint propagation it is the other way around. Adjoints are propagated parent to child. We must find the correct traversal order.

9.4 ADJOINT ACCUMULATION AND DAG TRAVERSAL

Can we find a traversal strategy for adjoint propagation such that each node is visited exactly once and all adjoints accumulate correctly?

Preorder traversal

An intuitive candidate is *preorder*. Like postorder, preorder is a depth-first traversal pattern, but one that visits each node *before* its children. We reprint the postorder code and write the preorder code below (on

the base Node class). We also add a method propagateAdjoints() on the Number class to start adjoint propagation visits in preorder from the result's node:

```cpp
//  On the base Node
    template <class V>
    void postorder(V& visitFunc)
    {
        //  Already processed -> do nothing
        if (myProcessed == false)
        {
            //  Process children first
            for (auto argument : myArguments)
                argument->postorder(visitFunc);

            //  Visit the node
            visitFunc(*this);

            //  Mark as processed
            myProcessed = true;
        }
    }

    template <class V>
    void preorder(V& visitFunc)
    {
        //  Visit the node first
        visitFunc(*this);

        //  Process children
        for (auto argument : myArguments)
            argument->preorder(visitFunc);
    }

//  On the Number class
    //  Propagate from the result
    void propagateAdjoints()
    {
        //  Reset all to 0
        myNode->resetAdjoints();
        //  See result to 1
        myNode->adjoint() = 1.0;
        //  Propagate from top
        myNode->preorder([](Node& n) {n.propagateAdjoint(); });
    }

int main()
{
    //  Set inputs
    Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };

    //  Build the DAG
    Number y = f(x);

    //  Set order on the dag
    y.setOrder();

    //  Evaluate
    y.evaluate();

    //  Propagate adjoints through the dag, from the result
    y.propagateAdjoints();

    //  Get derivatives
    for (size_t i = 0; i < 5; ++i)
    {
        cout << "a" << i << " = " << x[i].adjoint() << endl;
    }
}
```

Here is the result:

- Propagating node 13 adjoint = 1
- Propagating node 11 adjoint = 24.0445
- Propagating node 7 adjoint = 24.0445
- Accumulating leaf 1 adjoint = 168.312
- Propagating node 6 adjoint = 72.1336
- Propagating node 4 adjoint = 72.1336
- Accumulating leaf 2 adjoint = 72.1336
- Accumulating leaf 3 adjoint = 360.668
- Accumulating leaf 5 adjoint = 72.1336
- Propagating node 10 adjoint = 24.0445
- Accumulating leaf 8 adjoint = 73.2041
- Propagating node 9 adjoint = 96.1781
- Propagating node 7 adjoint = 28.6244
- Accumulating leaf 1 adjoint = 368.683
- Propagating node 6 adjoint = 158.007
- Propagating node 4 adjoint = 230.14
- Accumulating leaf 2 adjoint = 302.274
- Accumulating leaf 3 adjoint = 1511.37
- Accumulating leaf 5 adjoint = 230.14
- Propagating node 12 adjoint = 33.1781
- Propagating node 7 adjoint = 61.8025
- Accumulating leaf 1 adjoint = 801.3
- Propagating node 6 adjoint = 343.414
- Propagating node 4 adjoint = 573.555
- Accumulating leaf 2 adjoint = 875.829
- Accumulating leaf 3 adjoint = 4379.14
- Accumulating leaf 5 adjoint = 573.555
- Propagating node 9 adjoint = 129.356
- Propagating node 7 adjoint = 67.9623
- Accumulating leaf 1 adjoint = 1277.04
- Propagating node 6 adjoint = 547.301
- Propagating node 4 adjoint = 1120.86
- Accumulating leaf 2 adjoint = 1996.69

- Accumulating leaf 3 adjoint = 9983.43
- Accumulating leaf 5 adjoint = 1120.86
- a0 = 9983.43
- a1 = 1120.86
- a2 = 1277.04
- a3 = 73.2041
- a4 = 0

We conducted 35 visits through a DAG with 13 nodes. All nodes were visited, but many were visited multiple times. For example, node 7 propagated 4 times.

However, the derivatives are wrong. We recall that we computed them with finite differences earlier, and the correct values are: 950.736, 190.147, 443.677, 73.2041, and 0.

Actually, the values are incorrect *because* nodes are visited multiple times. Every time, their adjoint accumulated so far is propagated so their arguments accumulate adjoints multiple times. This is easily remedied by setting the adjoint to 0 after its propagation to child nodes. This way, only the part of the adjoint *accumulated since the previous propagation* is propagated the next time around:

```
//  On the + Node
    void propagateAdjoint() override
    {
        cout << "Propagating node " << myOrder
            << " adjoint = " << myAdjoint << endl;

        myArguments[0]->adjoint() += myAdjoint;
        myArguments[1]->adjoint() += myAdjoint;

        myAdjoint = 0.0;
    }

//  On the * Node
    void propagateAdjoint() override
    {
        cout << "Propagating node " << myOrder
            << " adjoint = " << myAdjoint << endl;

        myArguments[0]->adjoint() += myAdjoint * myArguments[1]->result();
        myArguments[1]->adjoint() += myAdjoint * myArguments[0]->result();

        myAdjoint = 0.0;
    }

//  On the log Node
    void propagateAdjoint() override
    {
        cout << "Propagating node " << myOrder
            << " adjoint = " << myAdjoint << endl;

        myArguments[0]->adjoint() += myAdjoint / myArguments[0]->result();

        myAdjoint = 0.0;
    }

//  On the Leaf
    void propagateAdjoint() override
    {
        cout << "Accumulating leaf " << myOrder
            << " adjoint = " << myAdjoint << endl;

        //  No reset here, no propagation is happening, just accumulation
    }
```

Now we get the correct values, *exactly* the same derivatives we had with finite differences, but we still visited 35 nodes out of 13, and the remedy we implemented is really a hack: if every node was visited once, as it should, such remedy would not be necessary.

Note that AAD and finite differences agree on differentials, at least to the fourth significant figure. This illustrates that analytic derivatives produced by AAD are not different, in general, from the numerical derivatives produced by bumping. They are just computed a lot faster.

At least AAD is faster when implemented correctly. A poor implementation may well perform slower than finite differences. Our implementation so far certainly does. Efficient memory management and friends are addressed in the next chapter, but DAG traversal is addressed now. Our implementation will not fly unless we visit each node in the DAG once; 35 visits to 13 nodes is not acceptable. Preorder is not the correct traversal strategy.

Breadth-first traversal

Preorder is still a depth-first strategy in the sense that it sequentially follows paths on the DAG down to a leaf.

We may try a breadth-first strategy instead, where visits are scheduled by level in the hierarchy: top node first, then its children, left to right, then all the children of the children, and so forth.

Breadth-first traversal cannot be implemented recursively.[9] It is implemented as follows: start on the top node with an empty queue of (pointers on) nodes. Process a node in the following manner:

1. Send the children to the back of the queue.
2. Visit the node.
3. Process all nodes in the queue from the front until empty.

Easy enough. The implementation takes a few lines:

```
#include <queue>

//  On the base Node

    template <class V>
    void breadthFirst(
        V& visitFunc,
        queue<shared_ptr<Node>>& q = queue<shared_ptr<Node>>())
    {
        //  Send kids to queue
        for (auto argument : myArguments) q.push(argument);

        //  Visit the node
        visitFunc(*this);

        //  Process nodes in the queue until empty
        while (!q.empty())
        {
            //  Access the front node
            auto n = q.front();

            //  Remove fom queue
            q.pop();

            //  Process it
            n->breadthFirst(visitFunc, q);
        }   //  Finished processing the queue: exit
    }

//  On the Number class

    //  Propagator, from the result
    void propagateAdjoints()
    {
        myNode->resetAdjoints();
        myNode->adjoint() = 1.0;
        myNode->breadthFirst([](Node& n) {n.propagateAdjoint(); });
    }
```

The $main()$ caller is unchanged.

Unfortunately, breadth-first is no better than preorder: it computes the correct results (providing we still zero adjoints after propagation) but still conducts 35 visits, and node 7, for instance, still propagates four times. The order is not the same, but the performance is identical.

Breadth-first is not the answer.

9.5 WORKING WITH TAPES

We believe it was useful to experiment with different unsuccessful traversal strategies to earn a good understanding of DAGs and how they

work, and we took the time to introduce some essential pieces of graph theory and put them in practice in modern, modular (although inefficient) C++ code.

Now is the time to provide the correct answer.

What we need is a traversal strategy that guarantees that the entire, correct adjoints accumulate after a single visit to every node. The only way this is going to happen is that every node must have completed accumulating its adjoint *before* it propagates. In other terms, *all* the parents of a node must be visited *before* the node.

This particular order is well known to graph theory. It is called the *topological order*. In general, it takes specialized algorithms to sort a graph topologically. But for DAGs we have a simple result:

The topological order for a DAG is the reverse of its postorder.

This is simple enough, and the demonstration is intuitive: postorder corresponds to the evaluation order, where arguments are visited before calculations. Hence, in its reverse order, all the parents (operations) of a node (argument) are visited before the node. In addition, postorder visits each node once, hence, so does the reverse. Therefore, the reverse postorder *is* the topological order.[10]

The correct order of traversal in our example is simply: 13, 12, 11,..., 1.

What this means is that we don't need DAGs after all. We need *tapes*. We don't need a graph structure for the nodes; we need a *sequence* of nodes in the order they evaluate (postorder). After we completed evaluation and built the tape, we traversed it in the reverse order to propagate adjoints. One single (reverse) traversal of the tape correctly computes the adjoints for all the nodes. We computed all the derivative sensitivities in constant time.

We propagate adjoints once through each node just as we evaluate each node once. The complexity of the propagation in terms of visits is therefore the same as for the evaluation. Both conduct the same number of visits, corresponding to the number of operations. Propagation visits are slightly more complex, though; for example, to propagate through a multiplication takes two multiplications. So the total complexity of AAD is around three evaluations (depending on the code): one evaluation + (one propagation = two evaluations). In addition, we have a significant *administrative* overhead: with our toy code, allocations alone could make AAD slower than bumping. We will reduce admin costs to an absolute minimum in the next chapter, and further in Chapter 15.

For now, we refactor our code to use a tape.

Conceptually, a tape is a DAG sorted in the correct order.

Programmatically, this simplifies data structures. The tape is a sequence of nodes, so it is natural to store the nodes in a vector of (unique pointers). That way, a node's lifespan is the same as the tape's. We also save the overhead of shared pointers. Nodes still need references to their arguments to propagate adjoints, but they don't need to own their memory: we can use dumb pointers for that.

We also simplify the code and remove lazy evaluation and numbering: these were for demonstration and we don't need them anymore. We need the tape only for adjoint propagation. Therefore, we modify the code so it evaluates calculations (eagerly) as it builds the tape, and, when the calculation is complete, propagates adjoints through the tape in the reverse order.

The complete refactored code is listed below:

```cpp
#include <memory>
#include <string>
#include <vector>
#include <queue>

using namespace std;

class Node
{
protected:

    vector<Node*> myArguments;

    double      myResult;
    double      myAdjoint = 0.0;

public:

    //  Access result
    double result()
    {
        return myResult;
    }

    //  Access adjoint
    double& adjoint()
    {
        return myAdjoint;
    }

    void resetAdjoints()
    {
        for (auto argument : myArguments) argument->resetAdjoints();
        myAdjoint = 0.0;
    }

    virtual void propagateAdjoint() = 0;
};

```

```
class PlusNode : public Node
{
public:

    PlusNode(Node* lhs, Node* rhs)
    {
        myArguments.resize(2);
        myArguments[0] = lhs;
        myArguments[1] = rhs;

        //  Eager evaluation
        myResult = lhs->result() + rhs->result();
    }

    void propagateAdjoint() override
    {
        myArguments[0]->adjoint() += myAdjoint;
        myArguments[1]->adjoint() += myAdjoint;
    }
};

class TimesNode : public Node
{
public:

    TimesNode(Node* lhs, Node* rhs)
    {
        myArguments.resize(2);
        myArguments[0] = lhs;
        myArguments[1] = rhs;

        //  Eager evaluation
        myResult = lhs->result() * rhs->result();
    }

    void propagateAdjoint() override
    {
        myArguments[0]->adjoint() += myAdjoint * myArguments[1]->result();
        myArguments[1]->adjoint() += myAdjoint * myArguments[0]->result();
    }
};

class LogNode : public Node
{
public:

    LogNode(Node* arg)
    {
        myArguments.resize(1);
        myArguments[0] = arg;

        //  Eager evaluation
        myResult = log(arg->result());
    }
```

```
94
95      void propagateAdjoint() override
96      {
97          myArguments[0]->adjoint() += myAdjoint / myArguments[0]->result();
98      }
99  };
100
101 class Leaf: public Node
102 {
103 public:
104
105     Leaf(double val)
106     {
107         myResult = val;
108     }
109
110     double getVal()
111     {
112         return myResult;
113     }
114
115     void setVal(double val)
116     {
117         myResult = val;
118     }
119
120     void propagateAdjoint() override {}
121 };
122
123 class Number
124 {
125     Node* myNode;
126
127 public:
128
129     //  The tape, as a public static member
130     static vector<unique_ptr<Node>> tape;
131
132     //  Create node and put it on tape
133     Number(double val)
134         : myNode(new Leaf(val))
135     {
136         tape.push_back(unique_ptr<Node>(myNode));
137     }
138
139     Number(Node* node)
140         : myNode(node) {}
141
142     Node* node()
143     {
144         return myNode;
145     }
146
```

```
    void setVal(double val)
    {
        //  Cast to leaf, only leaves can be changed
        dynamic_cast<Leaf*>(myNode)->setVal(val);
    }

    double getVal()
    {
        //  Same comment here, only leaves can be read
        return dynamic_cast<Leaf*>(myNode)->getVal();
    }

    //  Accessor for addjoints
    double& adjoint()
    {
        return myNode->adjoint();
    }

    //  Adjoint propagation
    void propagateAdjoints()
    {
        myNode->resetAdjoints();
        myNode->adjoint() = 1.0;

        //  Find my node on the tape, searching from last
        auto it = tape.rbegin();    //  last node on tape
        while (it->get() != myNode)
            ++it;   //  reverse iter: ++ means go back

        //  Now it is on my node
        //  Conduct propagation in reverse order
        while (it != tape.rend())
        {
            (*it)->propagateAdjoint();
            ++it;   //  Really means --
        }
    }
};

vector<unique_ptr<Node>> Number::tape;

Number operator+(Number lhs, Number rhs)
{
    //  Create node: note eagerly computes result
    Node* n = new PlusNode(lhs.node(), rhs.node());
    //  Put on tape
    Number::tape.push_back(unique_ptr<Node>(n));
    //  Return result
    return n;
}

Number operator*(Number lhs, Number rhs)
{
    //  Create node: note eagerly computes result
    Node* n = new TimesNode(lhs.node(), rhs.node());
    //  Put on tape
    Number::tape.push_back(unique_ptr<Node>(n));
```

```
204     //  Return result
205     return n;
206 }
207
208 Number log(Number arg)
209 {
210     //  Create node: note eagerly computes result
211     Node* n = new LogNode(arg.node());
212     //  Put on tape
213     Number::tape.push_back(unique_ptr<Node>(n));
214     //  Return result
215     return n;
216 }
217
218 template <class T>
219 T f(T x[5])
220 {
221     auto y1 = x[2] * (5.0 * x[0] + x[1]);
222     auto y2 = log(y1);
223     auto y = (y1 + x[3] * y2) * (y1 + y2);
224     return y;
225 }
226
227 int main()
228 {
229     //  Set inputs
230     Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
231
232     //  Evaluate and build the tape
233     Number y = f(x);
234
235     //  Propagate adjoints through the tape
236     //  in reverse order
237     y.propagateAdjoints();
238
239     //  Get derivatives
240     for (size_t i = 0; i < 5; ++i)
241     {
242         cout << "a" << i << " = " << x[i].adjoint() << endl;
243     }
244
245     //  950.736, 190.147, 443.677, 73.2041, 0
246 }
```

This code returns the correct derivatives, having traversed the 13 nodes on the tape exactly once.

We just completed our first implementation of AAD.

This code is likely slower than bumping, the Number type is incomplete: it only supports sum, product, and logarithm. The code is not optimal in many ways, and uses shortcuts such as a static tape and unsafe casting. In the next chapter, we produce professional code, work on memory management, and implement various optimizations and

conveniences. We make AAD orders of magnitude faster than bumping.

But the *essence* of AAD lies in this code and the explanations of this chapter.

NOTES

1 We define elementary operations as the building blocks of any calculation: scalar numbers, unary functions like exp or $sqrt$, and binary arithmetic operators and functions like pow. In addition to the operators and mathematical functions available as standard in C++, we consider as building blocks all the fundamental unary or binary functions, repeatedly called in calculation code, and which derivatives are known analytically, for example, the Gaussian density $n()$ and cumulative distribution $N()$ from the previous chapter.

2 Not all leaf nodes are inputs. Leaf nodes may also be constants. But that does not matter for now. We will compute the partial derivatives of the final result to all the leaves here, and optimize the constants away in the next chapter.

3 In all that follows, we neglect rounding errors that could, for example, make $x(y+z)$ different from $xy+xz$ and only consider *mathematical* equivalence.

4 We use a particularly simple and practically irrelevant example in this chapter, in an attempt to demonstrate the main notions behind AAD with maximum simplicity and minimum code. In particular, this calculation only uses the operators $+$, $*$ and log, so we can simplify the demonstration code with only four node types. The input $x[4]$ does not participate in the calculation; hence the sensitivity of $f()$ to $x[4]$ is 0, something our algorithm must reproduce correctly.

5 We are not talking about recombining trees here.

6 See http://www.cplusplus.com/reference/memory/dynamic_pointer_cast/.

7 The first *A* in *AAD* sometimes stands for Automatic and sometimes for Algorithmic. Algorithmic sounds better but Automatic better describes the method.

8 An alternative solution is to compute the partial derivatives *eagerly* during evaluation and store them on the parent node. Once the parent node caches all its partial derivatives, the intermediate results are no longer needed for back-propagation. This solution is actually the one implemented in Chapters 10 and 15.

9 To be honest, *nothing* should be implemented recursively in C++. It is best for performance to implement everything without recursion. But this is tedious; performance is not the concern of this chapter, and recursive code is terse and elegant.

10 Actually, this proves that reverse postorder is *a* topological order but let us put uniqueness considerations aside.