Chapter 8

Algorithmic Adjoint Differentiation

In this chapter, we introduce and explain AAD in a general context with the help of basic mathematics and toy code. This is a general chapter not related to Finance in any way. It is also purely pedagogical, and the toy code is for demonstration purpose only. Professional, efficient AAD code is delivered in the next chapter.

We consider a mathematical calculation that takes scalar inputs $(x_i)_{1 \leq i \leq I}$ and produces a scalar output y, and introduce the theory and practice of *adjoint mathematics* as a means to compute all the $\frac{\partial y}{\partial x_i}$ in constant time.

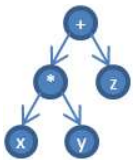
8.1 Calculation graphs

8.1.1 Definition and examples

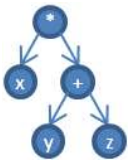
The first notion we must understand to make sense of AAD is that of a *calculation graph*. Every calculation defines a graph. The *nodes* are all the elementary mathematical calculations involved: $+$, $-$, $*$, $/$, *pow*, *exp*, *log*, *sqrt* and so forth. The *vertices* that join 2 nodes represent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

[1] Not all leafs are inputs. Leafs may also be constants. But that does not matter for now. We will compute the partial derivatives of the final result to all the leafs here, and optimize the constants away in the next chapter.

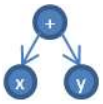
The structure of the graph reflects precedence among operators: for instance the graph for $xy + z$ is



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 that code is, the compiler having made a choice among the equivalent orders [2]. 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:

[2] 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.

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, according to the formula:

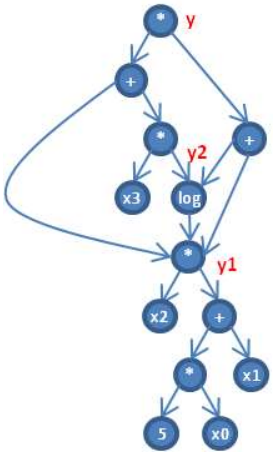
$$y = (y_1 - x_3 y_2)(y_1 + y_2), y_1 = x_2(5x_0 + x_1), y_2 = \log(y_1)$$

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

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

int main()
{
   double x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
   double y = f(x);
   cout << y << endl; // 797.751
   for (size_t i = 0; i < 5; ++i)
   {
      x[i] += 1.e-08;
      cout << 1.0e+08 * (f(x) - y) << endl;
      x[i] -= 1.e-08;
   } // 950.736, 190.147, 443.677, 73.2041, 0
}
```

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



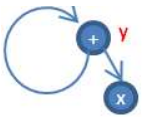
It is important to understand that complicated, compound calculations, even those that use OO architecture and nested function calls, like the ones that compute the value of an exotic option with MC simulations or FDM, or even the value of the xVA on a large netting set, or any kind of valuation or other 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 vertices refer to their arguments.

8.1.2 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*



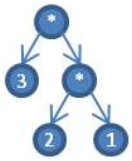
but

Although the variable y is reused and overwritten in the calculation of the sum, that 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:

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

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.

8.1.3 Directed Acyclic Graphs (DAGs)

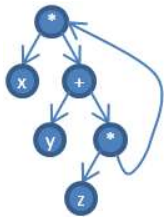
Computer programs do not define just any type of graph.

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 father [3]. In our previous example, the node y_2 has 2 parents and the node y_1 has 3. Calculation graphs are not trees, so reasoning and algorithms that apply to trees don't necessarily apply to them.

Secondly, note that vertices are *directed*, calculation to arguments. A vertice 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 vertices 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:

[3] We are not talking about recombining trees here.



is *not* a calculation graph, and no code could ever generate a graph like that. 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 using graph theory shortly, so we must identify exactly the class of graphs that qualify as calculation graphs.

8.1.4 DAGs and control flow

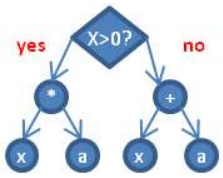
The DAG of a calculation only represents the mathematical operations involved, not the control flow. There are no nodes for if, for, $while$ and friends. A DAG represents the chain of calculations under one particular control branch. For instance, the code

```
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



The graph above is *not* a calculation DAGs.

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 that calculation.

AAD works with DAGs. It follows that AAD differentiates mathematical operations over the specific realization of the control flow during some evaluation, and does not differentiate through the control flow itself: since there are no "if" nodes in the 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 10. For now, we focus on building DAGs and using them for the computation of differentials.

8.2 Building and using DAGs

8.2.1 Building a DAG

A DAG is not just a theoretical representation of some calculation. We can use *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 purpose 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 a OO hierarchy:

```
#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)
   {
      myValue = val;
   }
};
```

We use a classical composite pattern (see GOF, [27]) 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 parents are gone, the node is released automatically.

Next, we design a custom number type that refers to the node on the graph corresponding to the operation last assigned to it. It also creates a leaf on initialization. We will be using that type in place of doubles.

```
class Number
{

   shared_ptr<Node> myNode;

public:

   Number(double val)
      : myNode(new Leaf(val)) {}

   Number(shared_ptr<Node> node)
      : myNode(node) {}

   shared_ptr<Node> node()
   {
      return myNode;
   }

   void setVal(double val)
   {
      // Cast to leaf, only leafs can be changed
      dynamic_pointer_cast<Leaf>(myNode)->setVal(val);
   }

   double getVal()
   {
      // Same comment here, only leafs can be read
      return dynamic_pointer_cast<Leaf>(myNode)->getVal();
```

```
    }
};
```

Then, we *overload* mathematical operators and functions for that number type so that when those operators and functions are executed with that type, the program does not conduct an evaluation, 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 framework, we can now build graphs *automatically*. This is where the first "A" in AAD comes from. We don't need to build the DAG manually. Our framework ensures that the code does that for us when we instantiate it with our number type. We *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:

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

    Number y = f(x);
}
```

Because we called f with Numbers in place of doubles, the line

```
Number y = f(x);
```

does *not* conduct any calculation whatsoever. All mathematical operations within f execute our overloaded operators and functions. So, in the end, what that 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, along with other preparations, is called *instrumentation*. In chapter 11, we instrument our simulation code.

8.2.2 Traversing the DAG

What can we do with that DAG?

First, we can calculate it. We call that an *evaluation* of the DAG. Note that we are going to evaluate the DAG itself, without further reference to the function that constructed it or the calculation code in that function.

How are we going to proceed exactly? The first comment is that arguments to an operation must be computed before the operation. That means that the processing of a node starts with the processing of its children.

Secondly, 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 that reason, we start the processing of a node with a check that this node was not processed previously. If that is verified, then, we process its children, and then, 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. In this algorithm, the only one 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 the node was already processed, we simply pick its cached result.

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.

So 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 implemented here is well known to the graph theory. It is called "Depth first postorder" or simply *postorder*. Depth first because it implements bottom-up visits, starting with leafs and ending with the top node. Postorder because children (arguments) of a node are always visited (evaluated) before the the node (operation). Postorder guarantees that each node in a DAG is processed exactly once, and that children are visited before 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 that a node was processed, and caches the result of its evaluation. For our information only, the order of calculation for that node is also stored. 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)
   {
      // Allready processed -> do nothing
      if (myProcessed == false)
      {
         // Process children
         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;
   }
};
```

Now 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. We remind that an object of the Number type stores a reference to the node that was last assigned to it.

```
// 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
}
```

We remind that the evaluation, the execution of the instructions, the mathematical operations in the calculation, did not happen on

```
    Number y = f(x);
```

That just constructed the DAG. No mathematics whatsoever were executed. The actual calculation happened in

```
    y.evaluate()
```

directly from the DAG. That it returns the exact same result than a direct function call (with double as the number type) is simply indicates that our code is correct.

To see that 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:

```
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

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

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

This pattern to store a calculation in the form a DAG, or another form that lends itself to evaluation, instead of conducting the evaluation straight away (or *eagerly*), and conduct 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 13) and scripting (see our publication [14]).

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. That is useful information for what follows. We identify the nodes by their calculation order and store that id on the node. We reuse our postorder code, we just need a new visit function. Since those visits don't depend on the concrete type of the node, there is no need to make them virtual and override them for the concrete node types.

```
// 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. This is a new syntax in C++14 that works with VS15 and allows not only to capture environment variables in the lambda, but also, define new variables internal to the lambda in the capture clause. The evaluation of the lambda modifies the order, so the lambda must be marked as mutable.

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

   Number y = f(x);

   y.setOrder();
}
```

Our DAG is now numbered, and each node is identified by its postorder in the DAG. We can log the intermediary 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;
      });
   }
```

Evidently, we must evaluate the DAG before we log results. 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();

}
```

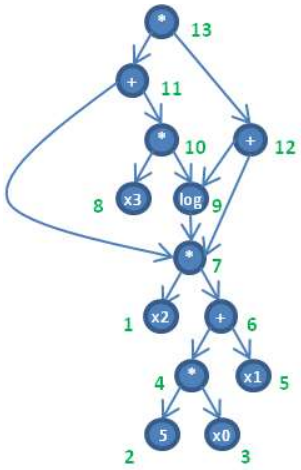
For the first evaluation we get the log:

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

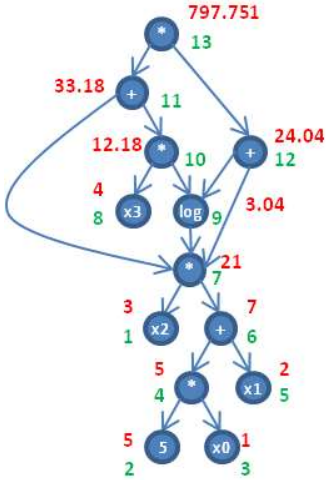
Processed node 11 result = 33.1781
Processed node 12 result = 24.0445
Processed node 13 result = 797.751

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

We can display the DAG (for the first evaluation), this time identifying nodes by their evaluation order:



and intermediary results:



There is more we can do with the DAG. We can reverse the DAG creation process and reconstruct an equivalent calculation program out of the DAG:

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

   Number y = f(x);

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

   // Log program from the dag
   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 leafs 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 [27]. That pattern is explained in more detail and systematically applied to scripts in our publication [14] (where we study trees, not DAGs, but the logic is the same).

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

That covered the "Automatic" in "Automatic Adjoint Differentiation" [4]. 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 that, we need a detour through basic formalism and adjoint mathematics.

[4]The first A in AAD sometimes stands for Automatic and sometimes for Algorithmic. Algorithmic sounds better but Automatic better describes the method.

8.3 Adjoint mathematics

In what follows, we identify the 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 (2 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 leafs.

The leafs are processed first in postorder traversal, so their node number i is typically low. Leafs 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. Those operations on the nodes are elementary operations and their derivatives are known analytically.

Note the reverse order to that of 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, that is all the differentials of the final result.

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

The *visit* to node n_i, consists in *adding*

$$\frac{\partial y_i}{\partial y_j} a_i$$

to the adjoints of its arguments (childs) $n_j \in C_i$. After every node has been processed, the adjoint of all nodes will have correctly accumulated in accordance to the adjoint equation.

That algorithm is called *adjoint propagation*.

Note that $\frac{\partial y_i}{\partial y_j}$ is known analytically, but depends on all the $(y_j)_{n_j \in C_i}$. That means that we must know all the intermediary results of the evaluation 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.

In addition, if the variables that referred to the child nodes nodes n_js are overwritten or out of scope at propagation time, we don't know the y_js when we propagate adjoints from node i, so we can't compute those partial derivatives. For that reason, during the evaluation, we must store either the temporary results y_j on the nodes j, or all the partial derivatives $\frac{\partial y_i}{\partial y_j}$ on the node i.

Note that our DAG evaluation code already implements the first solution. Temporary results y_j are stored on the nodes j as they are evaluated.

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 leafs. When visiting a node, we must know its adjoint a_i, so we can add

$$\frac{\partial y_i}{\partial y_j} a_i$$

to the adjoints a_j of all its child nodes $j \in C_i$. That 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 cnstant time.

It is easy enough to code the visit routines. Note we store adjoints on the nodes and write methods to set and access them, like we did for 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
   void propagateAdjoints()
   {
      myNode->resetAdjoints();
      myNode->adjoint() = 1.0;
      // ???
   }
```

We could 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 cut it. Postorder traverses the DAG arguments first, but for adjoint propagation, it is the other way around. Adjoints are propagated parent to child. We need to find the correct traversal order for that.

8.4 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?

8.4.1 Preorder traversal

The natural candidate is *preorder*. Like postorder, preorder is a depth-first traversal pattern, but one that visits the node *before* its children. We repeat 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 number's node:

```
// On the base Node
   template <class V>
   void postorder(V& visitFunc)
   {
      // Allready processed -> do nothing
      if (myProcessed == false)
      {
         // Process children
         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
      visitFunc(*this);

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

// On the Number class
   // Propagator, from the result
   void propagateAdjoints()
   {
      myNode->resetAdjoints();
      myNode->adjoint() = 1.0;
      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
    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.

Secondly, the derivatives are wrong. We remind 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. 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, the remedy is not needed.

Note that AAD and finite differences agree on differentials, at least to the forth decimal. That 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.

8.4.2 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 [5].

[5] To be honest, *nothing* should be implemented recursively in C++. It is best for performance to implement everything without recursion, with the explicit use of a stack over non-recursive code. But this is tedious, performance is not the concern of this chapter, and recursive code is terse and elegant.

This 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 4 times. The order is not the same, but the performance is identical.

Breadth-first in not the answer.

8.5 Working with tapes

We believe it is useful that we experimented with different unsuccessful traversal strategies to earn a good understanding of DAGs and how they operate, and we took the time to introduce some essential pieces of graph theory and put them in practice in modern, modular C++.

Now is the time to provide the correct answer.

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

That 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.

That 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, the parents (operations) of a node (argument) are visited before that node. In addition, postorder visits each node once, hence, so does the reverse. Therefore, the reverse postorder *is* the topological order [6].

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

What that 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 traverse 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

[6] Actually, that proves that reverse postorder is *a* topological order but let us put uniqueness considerations aside.

same as for the evaluation. Both conduct the same number of visits, and that is the number of nodes. Propagation visits are slightly more complex though, for example, to propagate through a multiplication takes 2 multiplications. So the total complexity of AAD is around 3 evaluations (depending on the code): one evaluation + (one propagation = 2 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 13.

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

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

Programmatically, that simplifies our data structures. The tape is a sequence of nodes, so it is natural to use a vector of (unique pointers) of nodes as a data structure. The way, the nodes lifespan is the same than the tape. 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:

```
#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());
   }

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

class Leaf: public Node
{

   double myValue;

public:

   Leaf(double val)
      : myValue(val)
   {
      // Eager evaluation
      myResult = val;
   }

   double getVal()
   {
      return myValue;
   }

   void setVal(double val)
   {
      myValue = val;

      // Eager evaluation
      myResult = val;
   }

   void propagateAdjoint() override {}
};

class Number
{
   Node* myNode;

public:

   // The tape, as a public static member
```

```
   static vector<unique_ptr<Node>> tape;

   // Create node and put it on tape
   Number(double val)
      : myNode(new Leaf(val))
   {
      tape.push_back(unique_ptr<Node>(myNode));
   }

   Number(Node* node)
      : myNode(node) {}

   Node* node()
   {
      return myNode;
   }

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

   double getVal()
   {
      // Same comment here, only leafs 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 it: ++ 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));
   // Return result
   return n;
}

Number log(Number arg)
{
   // Create node: note eagerly computes result
   Node* n = new LogNode(arg.node());
   // Put on tape
   Number::tape.push_back(unique_ptr<Node>(n));
   // Return result
   return n;
}

template <class T>
T f(T 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;
}

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

   // Evaluate and build the tape
   Number y = f(x);

   // Propagate adjoints through the tape
   // in reverse order
   y.propagateAdjoints();

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

```
    // 950.736, 190.147, 443.677, 73.2041, 0
}
```

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

And just like that, we implemented AAD.

Make no mistake: this code is likely slower than bumping, the Number type is incomplete: it only supports sum, product and logarithm. In the forthcoming chapter, we work on memory management, as well as various optimizations and conveniences. We make AAD orders of magnitude faster than bumping.

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