CHAPTER 10 Effective AAD and Memory Management

In this chapter, we turn the ideas and toy code from the introductory chapters into a professional, efficient AAD library.

We have seen that, to differentiate calculation code with AAD, we instantiate the calculation code with a custom type for the representation of numbers, where all arithmetic operators and mathematical functions are overloaded, so that, in addition to being evaluated, all operations are recorded on a tape. After the calculation is complete and the entire sequence of its operations is recorded, adjoints are back-propagated over the tape to produce all its differentials in constant time.

The AAD library provides the custom number type, the tape data structure, and the object that represents an operation on tape, called *node*, together with back-propagation utilities to propagate adjoints over the nodes on tape in the reverse order from evaluation. Calculation code must be templated on its number type. Differentiation code executes the calculation code, instantiated with the AAD number type, which records the calculation on tape, calls the back-propagation utilities to accumulate adjoints, and reads differentials on the inputs' adjoints. This is called *instrumentation*. The AAD library is independent from any calculation code;

it is the instrumentation code that applies it to differentiate a particular calculation.

With AAD, more so than other algorithms, efficient implementation is crucial, and algorithmic correctness insufficient. Our toy code from the previous chapter implements the correct algorithm and computes differentials in constant time. However, if we tried to differentiate a complex calculation, implemented with sophisticated code, like the simulations of Part II, it would take a very long constant time to compute the differentials. In all likelihood, a naive AAD differentiation of such code would be in practice slower than finite differences. In addition, the recording of a large number of operations could saturate RAM and result in a system-wide slowdown or a crash.

For these reasons, it is sometimes heard that the benefits of AAD are purely theoretical and cannot be achieved in a practical context. Some results seem to indicate that AAD would only be applicable in restricted contexts, and often produce sensitivities slower than cleverly implemented bumps. All of this could not be more wrong. AAD is universally applicable; it produces derivatives with unthinkable speed, but only when both the AAD library and the instrumented calculation code are written efficiently. We discuss the AAD library here, and instrumentation in Chapter 12.

We work in sequence on the Node (object that stores one operation on tape), the Tape (data structure that holds the sequence of nodes), and the Number (custom number representation type that overloads operators and functions so they are recorded on tape), including operator overloading and adjoint propagation utilities. These are the same classes and routines we manipulated in the previous chapter, refactored for practicality and performance. The result is a self-contained AAD library, independent of calculation code and that may be used to instrument any kind of computation. The code is found in our repository. The AAD library consists in the AAD*.* files, with dependency on gaussians.h[1] and blocklist.h.[2]

An efficient AAD library is mainly about memory management. Everything that touches the tape is executed on every evaluation of a mathematical operation. On every sum, multiplication, square root, and so forth, an additional set of instructions are executed to put that operation on tape. AAD only performs when every aspect of that code is opti-

mized to the extreme. For instance, we cannot burden every mathematical operation with the significant overhead of a memory allocation, so the process that puts an operation on tape cannot involve allocation. Memory on tape must be preallocated so operations are recorded with minimal overhead.

What we develop here is a "traditional" AAD library, together with memory management techniques that allow it to perform with remarkable speed in contexts of practical relevance. We will revisit parts of the library and improve its performance in Chapter 15, resulting in an additional acceleration by a factor around two. We discuss some important characteristics and limitations in the next chapter, then, in Chapter 12, we instrument the parallel Monte Carlo library of Part II, like we instrumented our toy function $f()$ in the previous chapter, but on the larger scale of a financial library and in a parallel context.

10.1 THE NODE CLASS

In the toy implementation of the previous chapter, the partial derivatives of every operation were computed at back-propagation time. As a result, nodes were implemented as a polymorphic class, one concrete class for every mathematical operation, overriding the propagation method. In addition, the results of every operation were stored on the corresponding node, so that the partial derivatives could be computed at propagation time.

In the real code, we compute the local derivatives *eagerly* on evaluation and store them on the parent node. It follows that we no longer need polymorphic nodes, and we don't have to store the results of the operations. We have one single Node class to represent all operations, something like this:

```
1  //  Not acceptable code, for demonstration only
2  struct Node
3  {
4      //  0: leaf, 1: unary operation, 2: binary operation
5      size_t              n;
6
7      //  This node's adjoint, initialized to 0
8      double              adjoint = 0.0;
9      //  Local derivatives to arguments
10     vector<double>      derivatives;
11     //  Pointers to the adjoints of the arguments
12     vector<double*>     argAdjoints;
13
14     void propagate()
15     {
16         for (size_t i = 0; i < n; ++i)
17         {
18             *argAdjoints[i] += derivatives[i] * adjoint;
19         }
20     }
21 };
```

where the method $propagate()$ correctly implements the adjoint equation of the previous chapter. Note that we store pointers on the child adjoints in $argAdjoints$, not pointers on the child Nodes. Child adjoints are all we need for propagation.

There are a number of advantages to this approach:

- **Simplicity** The Node class is simpler and the code is considerably shorter. Adjoint propagation from node n_i adds $a_i \frac{\partial y_i}{\partial y_j}$ to the adjoints of its arguments $n_j \in C_i$. The knowledge of the specific operation and its arguments was only necessary to compute the partial derivatives $\frac{\partial y_i}{\partial y_j}$ at propagation time. With the partial derivatives known and stored on the parent node n_i, polymorphism and the storage of intermediate results are no longer necessary and we have a single Node type to represent all operations on tape.
- **Performance** Polymorphic classes are not free. They store hidden data members, called *vtable pointers*, necessary for the run-time resolution of virtual methods. Calls to virtual functions involve the additional overhead of run-time resolution, and the compiler cannot optimize and inline them as easily as nonvirtual methods. The single-Node design saves run-time polymorphism overhead at the cost of additional storage space on the Node for the partial derivatives. In addition, the partial derivatives stored on the Nodes can be computed once and reused for multiple back-propagations, for example, for the accumulation of the adjoints of multiple results.

It is also visible that this Node design is not limited to unary and binary operations. The data member n, which specifies the number of arguments to the operation on the node, may be larger than two. The adjoint propagation implemented in $propagate()$ remains correct with arbitrary n. This will alllow us to implement a substantial optimization in Chapter 15, where we represent entire *expressions* on a single multinomial node, resulting in around twice the speed in cases of practical relevance.

- **Convenience** It is no longer necessary to keep results on Nodes, so the value of a Number is stored on the Number itself. This makes it considerably simpler to reason about, design, and debug instrumented code.

There is also a drawback to this design. The storage of the derivatives on the Node increases its memory footprint. The benefits outweigh the drawbacks, but memory considerations are not to be underestimated with AAD. Every single mathematical operation is recorded on a node, so tapes store a large amount of nodes and consume vast amounts of RAM.

The code above correctly illustrates our desired *design* for the Node, but not the implementation. The data stored on the Node – the partial derivatives and the pointers to the arguments' adjoints – are stored in vectors. Vectors are dynamically sized data structures. They allocate, free, and manage dynamic memory. We can't have that on the Node. Dynamic memory management is expensive, and a Node is created every time a mathematical operation is executed.

A simplistic solution could be to store data in static arrays of size two instead, since we know that the mathematical operations recorded on tape have zero, one, or two arguments. Not only would that result in an unacceptable waste of memory in leaf and unary nodes, but it would forbid multinomial nodes, those with more than two arguments, necessary for the optimizations of Chapter 15.

Another solution could be to derive the Node type depending on the number of its arguments, but it would reintroduce dynamic polymorphism and associated overheads. The solution we implemented[3] is to store dynamically sized data *outside* of the Node, in a separate space in memory. Where and how exactly we store this data will be discussed in the next section, together with the Tape data structure and memory management

constructs. The Node doesn't store its data, but holds pointers on its place in memory.

```
1  //  Final code
2  class Node
3  {
4      //  Number of children (arguments)
5      const size_t n;
6
7      //  The adjoint
8          double      mAdjoint = 0;
9
10     //  Data lives in separate memory
11
12     //  the n derivatives to arguments
13     double*         pDerivatives;
14
15     //  the n pointers to the adjoints of the arguments
16     double**        pAdjPtrs;
17
18 public:
19
20     Node(const size_t N = 0) : n(N) {}
21
22     //  Access adjoint for read and write
23     double& adjoint()
24     {
25         return mAdjoint;
26     }
27
28     //  Back-propagate adjoints to arguments adjoints
29     void propagateOne()
30     {
31         //  Nothing to propagate
32         if (!n || !mAdjoint) return;
33
34         for (size_t i = 0; i < n; ++i)
35         {
36             *(pAdjPtrs[i]) += pDerivatives[i] * mAdjoint;
37         }
38     }
39 };
```

We named the back-propagation function $propagateOne()$, not simply $propagate()$, for a reason that will be made apparent in Chapter 14, where we upgrade the library and make minor additions to the Node class to efficiently differentiate not one, but multiple results in a calculation.

Note the check on line 32. We don't propagate anything unless there is a non-zero adjoint to propagate on the parent node. This is a simple, yet strong optimization. In cases of practical relevance, zero adjoints are frequent, due to control flow in the instrumented code. Empirically, this single optimization accelerates differentiation by an average 15% in practical situations.

The code is found in AADNode.h.

10.2 MEMORY MANAGEMENT AND THE TAPE CLASS

The toy implementation of the previous chapter stored Nodes in a vector. It had the merit of simplicity. Nodes were stored in the sequence where operations are evaluated, and the STL vector made it particularly simple and convenient to iterate over the sequence of the Nodes, either forward, or backwards for the purpose of adjoint propagation.

But this is both inefficient and impractical. It is inefficient because we don't know in advance the number of operations in a calculation. The size of the vector holding the Nodes grows during the calculation, causing it not only to reallocate larger chunks of memory, but also *to copy all its contents* from its previous memory. It is also impractical because the reallocation of a vector invalidates pointers and iterators to the elements in the vector. We have seen that Numbers hold a pointer to their Node on tape. All these pointers would be invalidated every time the vector holding the Nodes reallocates. The toy code walked around this particular difficulty by storing in vectors, not the Nodes themselves, but (unique) pointers on the Nodes. This way, the Nodes always remained in the same memory space, it is the (unique) pointers that were copied every time the vector reallocated, so the pointers on the Nodes held in the Numbers remained valid, at the cost of yet another overhead.

For this reason, we are not going to hold the Nodes in a vector. We are going to develop a custom container, purposely designed for maximum efficiency and practicality in the context of AAD. We will be following a pattern known as “memory pool” or “small object allocator,” whereby we will preallocate large blocks of memory and use them to store the sequence of Nodes side by side as operations in the instrumented code are recorded on tape. We baptized our data structure “blocklist” and implemented it in the file blocklist.h. Our implementation mainly follows the standard directives of STL container requirements, although we don't implement *all* of them.

The blocklist

The blocklist essentially follows the small object allocator pattern. It preallocates (large) blocks of memory and stores objects in the preallocated space so as to prevent costly allocations every time a new object is created. The design of such memory pools revolves around two considerations:

1. Manage what happens when a preallocated memory block fills out.
2. Handle the release of memory when individual objects are destroyed.

The second problem is typically hard to resolve in an efficient manner, but, luckily, it is not an issue for us. *We never need to release Nodes individually*. We don't free specific Nodes. We store the Nodes on tape in the sequence of the evaluation of their operations, and, when we no longer need the tape, we dispose of it and release all its Nodes simultaneously. In addition, Nodes don't manage external memory: they hold pointers to dynamic memory, but they don't own that memory, and it is not their responsibility to release it. It follows that Nodes don't need to be individually destroyed, either. They are just wiped out of memory altogether without any destruction overhead.

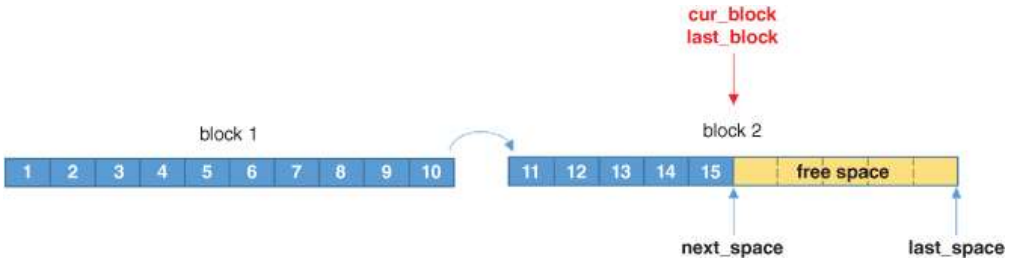
We store the Nodes in a sequence side by side in preallocated blocks of memory. When a block fills out, we preallocate another block for the storage of the subsequent Nodes. When this happens, the existing Nodes remain where they are. We don't copy anything, and, more importantly, all pointers and iterators to the existing Nodes remain valid.

The natural data structure for a memory block is a static array. C++11 introduced the standard array data structure, defined in the <array> header, as a thin, zero-cost wrapper around a static array, with the benefit of STL-compliant iterators. The ideal data structure to hold the multiple blocks recording a sequence of Nodes is a list: the standard guarantees that to insert a new element in a list does not affect the existing elements in any way, in particular, any pointer or iterator on existing elements remains valid. It follows that the container for the storage of the Nodes is a list of arrays, hence, “blocklist.” Our data structure wraps a list of arrays and provides functionality to completely encapsulate the underlying data structure and let client code navigate, read, and write its data *as if it was stored in a single container*.

```
#include <array>
#include <list>
#include <iterator>
using namespace std;

template <class T, size_t block_size>
class blocklist
{
    //  Container = list of blocks
    list<array<T, block_size>>  data;

    using list_iter = decltype(data.begin());
    using block_iter = decltype(data.back().begin());

    //  Current block
    list_iter           cur_block;

    //  Last block
    list_iter           last_block;

    //  Next free space in current block
    block_iter          next_space;

    //  Last free space (+1) in current block
    block_iter          last_space;

    //  ...
```

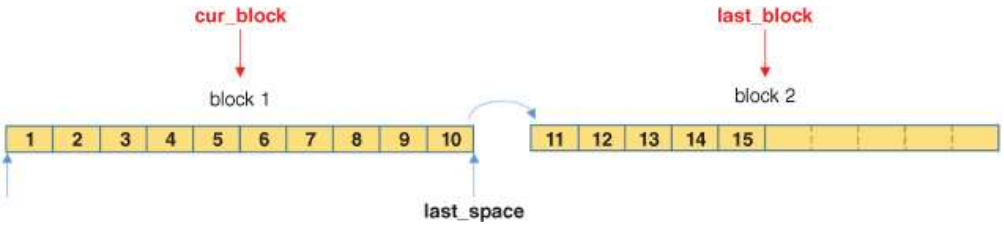
The iterator cur_block points to the current block, and the iterator next_space points to the first free space in that block. Together, these two iterators indicate the memory space where the next object is stored. The iterator last_space guards the end of the current block. When that space is reached, we jump to the next block. If the current block is the last block, as indicated by the iterator last_block, there is no next block, so we create another one at the back of the list.

For example, the following figure illustrates the state of a blocklist<Node, 10> after the storage of 15 Nodes:[4]



When a blocklist is reused repeatedly, we don't have to dispose of all the blocks, only to recreate them for the storage of subsequent objects. Instead, we *rewind* the blocklist with a reassignment of its iterators. The blocks remain in memory and their contents are overwritten with the storage of new objects.

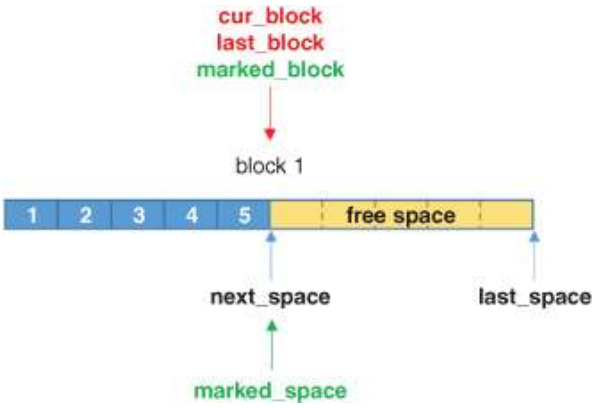
The following figure illustrates the state of the previous blocklist after a rewind. The next object will overwrite the first space in the first block, moving next_space to slot two. The storage of 10 objects will move the next_space to the first slot of the second block, without creation of a block, since we already have two. It is only after the storage of 20 objects, when the two blocks are full, that a third block will be created on the list.



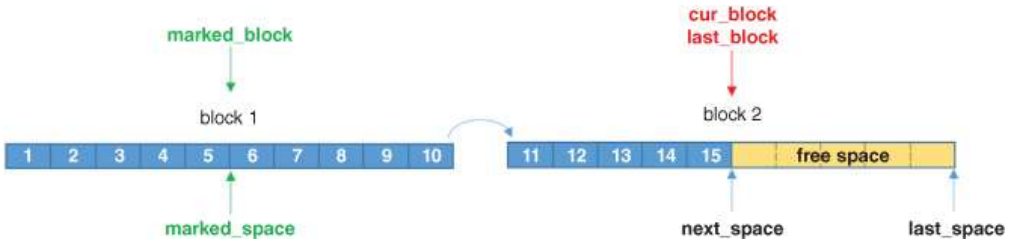
A very useful feature is to rewind a blocklist, not to the beginning, but to a mark left at a previous stage. We add a couple of iterators to the data structure so we can mark its state and subsequently revert to that marked state:

```
//  ...
//  Mark
list_iter           marked_block;
block_iter          marked_space;
//  ...
```

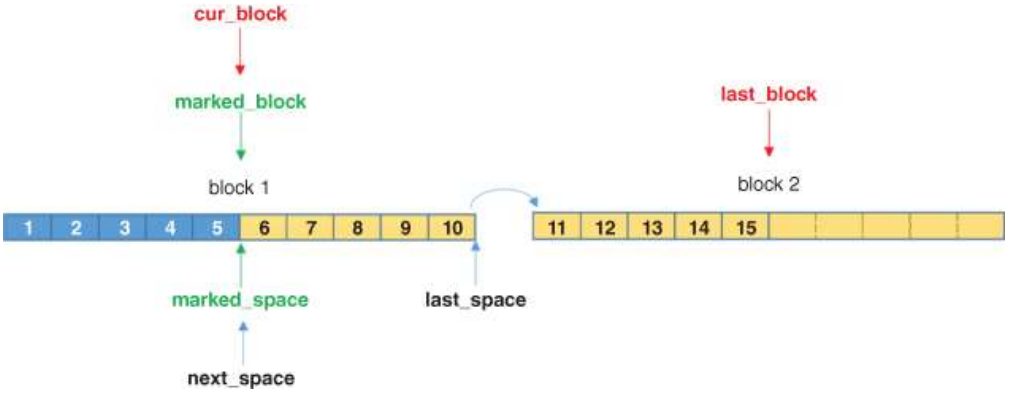
The following figure illustrates the state of the blocklist when the mark is set after the storage of five elements:



After the storage of another 10 elements, the state is illustrated below:



At this point, we rewind to the mark. Again, we just reassign iterators. Nothing is deleted; no memory is allocated or deallocated. The next stored object will overwrite the sixth slot of the first block:



These iterator gymnastics permit us to navigate the blocklist for reading and writing. The methods implemented in the blocklist essentially manipulate these iterators to store objects side by side in the preallocated memory blocks, and rewind the blocklist in various ways by reassignment of the iterators and without any modification to the contents or memory. First, we have a private method to create a new block and position the iterators for subsequent storage:

```
    //  ...
private:
    //  Create new array
    void newblock()
    {
        //  Create array
        data.emplace_back();

        //  Re-assign iterators to the created array
        cur_block = last_block = prev(data.end());
        next_space = cur_block->begin();
        last_space = cur_block->end();
    }
    //  ...
```

The C++11 *emplace_black()* method on the standard list (and other STL data structures) inserts a new slot in the back of the list and creates a new

object in place in that memory space. This method may take parameters, which are forwarded to the constructor of the created element. It is similar to $push\_back()$, except $push\_back()$ inserts a copy of its argument into the data structure, whereas $emplace\_black()$ creates a new element in place, saving a copy. In the blocklist code, $emplace\_black()$ creates a new array on the list $data$. The blocklist's iterators are assigned to the newly created array. We have another method to move to the next block, and create a new one if necessary:

```
    //  ...
    //  Move on to next array
    void nextblock()
    {
        //  This is the last array: create new
        if (cur_block == last_block)
        {
            newblock();
        }

        //  This is not the last array: move to next
        else
        {
            ++cur_block;
            next_space = cur_block->begin();
            last_space = cur_block->end();
        }
    }
    //  ...
```

If there is no next block, we create one and reassign iterators to the newly created array. Otherwise, we reassign the iterators to the next array on the list.

Next, we have the public interface, starting with the constructor (which creates the first block and sets iterators) and a method $clear()$ to factory reset the blocklist, as is customary for STL containers:

```
    //  ...
public:
    //  Create first block on construction
    blocklist()
    {
        newblock();
    }

    //  Factory reset
    void clear()
    {
        data.clear();
        newblock();
    }
    //  ...
```

We have methods to set a mark, rewind to the mark, or rewind to the beginning, by reassignment of the iterators, as explained earlier:

```
//  ...
//  Rewind but keep all blocks
void rewind()
{
    cur_block = data.begin();
    next_space = cur_block->begin();
    last_space = cur_block->end();
}

//  Set mark on current position
void setmark()
{
    marked_block = cur_block;
    marked_space = next_space;
}

//  Rewind to mark
void rewind_to_mark()
{
    cur_block = marked_block;
    next_space = marked_space;
    last_space = cur_block->end();
}
//  ...
```

The main purpose of the blocklist is to store objects. We implement a method *emplace_back()*, with STL consistent semantics to create a new object of type T in the next available memory space, as indicated by the iterators, and return a pointer on the object:

```
//  ...
//  Overload for default constructed
T* emplace_back()
{
    //  No more space in current array
    //      move to next (if none, create)
    if (next_space == last_space)
    {
        nextblock();
    }

    //  Current space
    auto old_next = next_space;

    //  Advance
    ++next_space;

    //  Return pointer
    //      on PREVIOUS space where the new object is stored
    return &*old_next;
}
//  ...
```

This overload is for default constructed objects. We don't even need to explicitly create the object, since arrays are populated with default objects on construction. We simply advance the iterator $next\_space$ and skip over the current slot. Note that in case new objects overwrite existing ones, they are not reset to default state. This is contrary to STL standard, but it prevents wasting CPU time resetting state, something we don't need. Also note the check on the first line: if we stand at the end of a block, we move on to the next one, which also creates a new block if necessary. The method returns a pointer on the memory space where the new object lives, as specified by the *previous* state of the iterator $next\_space$.

The whole purpose of the blocklist is to store objects in memory with minimum overhead. The method $emplace\_back()$ visibly achieves this goal. The method $nextblock()$ is only called in the rare situations where the current block fills out (one operation out of 13,384 by default). Besides, $nextblock()$ only involves an expensive allocation when a new block is created. When the same blocklist is repeatedly reused and rewound, $nextblock()$ typically involves a cheap reassignment of pointers. In the vast majority of situations where there is space available on the block, the storage of an object involves a check (next_space == last_space), an iterator assignment (old_next = next_space), and a skip (++next_space), three insignificantly cheap low-level instructions. The method $emplace\_back()$, and its overloads below, are a crucial factor in AAD performance, because they record Nodes in memory virtually for free.

We have two overloads to store more than one object, following the exact same logic. The first overload may be used when the number of objects is known at compile time and passed as a template parameter. When this is not the case, the second overload must be used. Both overloads implement the same code, virtually identical to the first overload above, and also with virtually zero run-time overhead, returning a pointer on the first created element.

```
//  ...
//  Stores n default constructed elements
//      and returns a pointer on the first

//      Version 1: n known at compile time
template <size_t n>
T* emplace_back_multi()
{
    //  No more space in current array
    if (distance(next_space, last_space) < n)
    {
        nextblock();
    }

    //  Current space
    auto old_next = next_space;

    //  Advance next
    next_space += n;

    //  Return
    return &*old_next;
}

//      Version 2: n unknown at compile time
//          same code
T* emplace_back_multi(const size_t n)
{
    //  No more space in current array
    if (distance(next_space, last_space) < n)
    {
        nextblock();
    }

    //  Current space
    auto old_next = next_space;

    //  Advance next
    next_space += n;

    //  Return
    return &*old_next;
}
//  ...
```

We mentioned that default constructed objects are not necessarily reset to factory state with our implementation of $emplace\_back()$. We find it more efficient to provide instead a method $memset()$ that resets all the bytes in all the arrays to some value, usually zero, simultaneously.

```
//  ...
//  Memset
void memset(unsigned char value = 0)
{
    for (auto& arr : data)
    {
        std::memset(&arr[0], value, block_size * sizeof(T));
    }
}
//  ...
```

All it does is call the standard $memset()$ repeatedly on all the blocks. Finally, we have an overload for the creation of objects that are not default constructible. This overload takes arguments and forwards them to the constructor of the new element:

```
//  ...
//  Construct object of type T in place
//      in the next free space and return a pointer on it
//  Implements perfect forwarding of constructor arguments
template<typename ...Args>
T* emplace_back(Args&& ...args)
{
    //  No more space in current array
    if (next_space == last_space)
    {
        nextblock();
    }
    //  Placement new, construct in memory pointed by next

    //  memory pointed by next as T*
    T* emplaced = new (&*next_space)
        //  perfect forwarding of ctor arguments
        T(forward<Args>(args)...);

    //  Advance next
    ++next_space;

    //  Return
    return emplaced;
}
//  ...
```

This overload follows the same logic, except it doesn't only skip over a slot in the array; instead, it does construct a new object in place, forwarding its arguments to the object's constructor, with the *placement new* syntax. Contrary to what the operator new may suggest, no allocation takes place. Placement new is a traditional C++ construct, albeit a rarely used one, with a perhaps peculiar syntax: $new(mem)T(params...)$ that creates an object of type T *in place* in the memory location mem with a call to T's constructor with arguments $params...$. The arguments to $emplace\_back()$ are *forwarded* to the object's constructor.

In addition, the number of arguments to $emplace\_back()$ is not fixed and depends on the number of arguments of the object's constructor. C++11 deals with variable number of arguments with a construct called *variadic templates*, which offers a specific syntax to deal with arguments not only of unknown type, like all templates, but also of unknown number. We don't discuss variadic templates in detail here, referring to any up-to-date C++ textbook instead. The arguments to $emplace\_back()$ are

forwarded to the object's constructor with the standard C++11 function $forward()$, which, as the name indicates, passes the arguments through, in the same number, order, type, and conserving properties like const-ness or rvalue and lvalue reference, something known in C++11 as *perfect forwarding*.

Blocklist iterators

Blocklists encapsulate block creation and navigation logic, so that client code may act *as if* its elements were stored in one contiguous array, and seamlessly navigate the data, the blocklist correctly and automatically jumping from block to block, creating new blocks when necessary behind the scenes. For this purpose, we develop an *iterator* on the blocklist as a public nested class within the blocklist class. Like all iterators, the blocklist::iterator acts as a pointer on an element in the blocklist. It can be dereferenced to access the element with the usual prefix (*) or post-fix (->). Like any STL *bidirectional* iterator, a blocklist iterator can be skipped to the next element with the prefix (++) or reverted to the previous element with the prefix (–), its internal mechanics taking care of the blocklist logic and seamlessly jumping from block to block when necessary.

A blocklist iterator has an internal state that specifies one slot in the blocklist:

```
//  ...
//  Iterator
class iterator
{
    //  List and block
    list_iter   cur_block;   // current block
    block_iter  cur_space;   // current space
    block_iter  first_space; // first space in block
    block_iter  last_space;  // last (+1) space in block
//  ...
```

The iterator's state consists of a current block and a current space in that block, as explained earlier in detail. It also holds iterators on the first and last space in its block so it can jump blocks when necessary. Although we don't implement all the features of a fully STL-compliant iterator, we write a few definitions, called *iterator traits*, that identify the iterator as bidirectional, so it may be used with basic STL algorithms:

```
// ...
public:

    // iterator traits
    using difference_type = ptrdiff_t;
    using reference = T&;
    using pointer = T*;
    using value_type = T;
    using iterator_category = bidirectional_iterator_tag;
// ...
```

We implement a default constructor for the iterator, and another constructor that sets its state to a specific slot in the blocklist:

```
// ...
    // Default constructor
    iterator() {}

    // Constructor
    iterator(list_iter cb, block_iter cs, block_iter fs, block_iter ls) :
        cur_block(cb), cur_space(cs), first_space(fs), last_space(ls) {}
// ...
```

Next, we have the prefix (++) operator, skipping the iterator to the next slot, jumping to the next block when necessary:

```
// ...
    // Pre-increment (we do not provide post)
    iterator& operator++()
    {
        // Next slot
        ++cur_space;

        // Skip over block?
        if (cur_space == last_space)
        {
            // Next block
            ++cur_block;
            // Reset block iterators
            first_space = cur_block->begin();
            last_space = cur_block->end();
            cur_space = first_space;
        }

        return *this;
    }
// ...
```

and the prefix (–) that implements the same logic to point to the previous element:

```
    //  ...
        //  Pre-decrement
        iterator& operator--()
        {
            //  Jump to previous block?
            if (cur_space == first_space)
            {
                //  Previous block
                --cur_block;
                //  Reset block iterators
                first_space = cur_block->begin();
                last_space = cur_block->end();
                cur_space = last_space;
            }

            --cur_space;

            return *this;
        }
    //  ...
```

Finally, we have the dereference operators expected from any pointer or iterator to access its element:

```
    //  ...
        //  Access elements
        T& operator*()
        {
            return *cur_space;
        }
        const T& operator*() const
        {
            return *cur_space;
        }
        T* operator->()
        {
            return &*cur_space;
        }
        const T* operator->() const
        {
            return &*cur_space;
        }
    };
```

and some comparison operators:

```
    //  ...
        //  Check equality
        bool operator ==(const iterator& rhs) const
        {
            return (cur_block == rhs.cur_block && cur_space == rhs.cur_space);
        }
        bool operator !=(const iterator& rhs) const
        {
            return (cur_block != rhs.cur_block || cur_space != rhs.cur_space);
        }
    };

    //  End of iterator code, blocklist code continues
    //  ...
```

so we can use these iterators in a loop like:

```
blocklist<T> b;
//...

for(auto it = b.begin(); it != b.end(); ++it)
{
    do_something_with_element(*it);
}
```

where we note that we implemented (a sufficient part of) the bidirectional iterator requirements so we can use STL algorithms like $for\_each()$, and rewrite the loop above in a more proper manner:

```
for_each(b.begin(), b.end(), do_something_with_element);
```

or even use the C++11 range syntax:

```
for (T& elem : b) do_something_with_element(elem);
```

Our blocklist data structure is *almost* complete. We must provide methods for client code to access iterators on the first and (one after) last element in the blocklist, like all STL containers:

```
    //  blocklist code continues
    //  ...
    //  Access to iterators

    iterator begin()
    {
        return iterator(data.begin(), data.begin()->begin(),
            data.begin()->begin(), data.begin()->end());
    }

    iterator end()
    {
        auto last_block = prev(data.end());
        return iterator(cur_block, next_space,
            cur_block->begin(), cur_block->end());
    }
    \\  ...
```

begin() returns an iterator on the first slot of the first block; *end*() returns an iterator on the *next available slot for the storage of an element*.[5] We also provide an iterator on the mark:

```
    //  ...
    //  Iterator on mark
    iterator mark()
    {
        return iterator(marked_block, marked_space,
            marked_block->begin(), marked_block->end());
    }
    \\  ...
```

and a method *find*() that locates an element in the blocklist *by address* and returns an iterator on this element if found, *end*() otherwise, as is customary with STL containers:

```
    //  ...
    //  Find element, by pointer, searching sequentially from the end
    iterator find(const T* const element)
    {
        //  Search from the end
        iterator it = end();
        iterator b = begin();

        while (it != b)
        {
            --it;
            if (&*it == element) return it;
        }

        if (&*it == element) return it;

        return end();
    }
};
```

Note that the semantics of $find()$ differ from usual STL semantics, where elements are found by value. It so happens that it best suits the needs of AAD to find elements by address. Also note that the search starts *from the end*, which is also AAD specific.

The code is found in blocklist.h in our repository.

The Tape class

We have encapsulated all the memory management difficulties in a custom data structure, the blocklist, which stores data in an extremely efficient manner, never invalidates its iterators (unless, of course, the blocklist is erased or rewound), and exposes a simple, STL-style interface to navigate data. The blocklist data structure was purposely designed to provide maximum efficiency and convenience, specifically in the context of AAD, so that the development of the tape presents no additional difficulty. Its code is in AADTape.h.

The tape mainly wraps a blocklist of Nodes. Remember from Section 10.1 that Nodes don't store their dynamically sized data (local derivatives and pointers on arguments adjoints). They only store the size of the data and pointers on its memory location. The actual Node data lives *in its own dedicated blocklists on the tape*:

```cpp
#include "blocklist.h"
#include "AADNode.h"

constexpr size_t BLOCKSIZE  = 16384;   //  Number of Nodes in a block
constexpr size_t DATASIZE   = 65536;   //  Size of data in a block

class Tape
{
    //  Storage for the Nodes
    blocklist<Node, BLOCKSIZE>          myNodes;

    //  Storage for Node data:
    //      derivatives and child adjoint pointers
    blocklist<double, DATASIZE>          myDers;
    blocklist<double*, DATASIZE>         myArgPtrs;

    //  Padding so tapes in a vector don't interfere
    char                                 myPad[64];

public:

    //  Build Node in place and return a pointer
    //  N : number of children (arguments)
    template <size_t N>
    Node* recordNode()
    {
        //  Construct the Node in place on the blocklist
        Node* node = myNodes.emplace_back(N);

        //  Store Node data unless leaf
        //      note constexpr if
        if constexpr(N)
        {
            //  derivatives
            node->pDerivatives = myDers.emplace_back_multi<N>();
            //  child adjoints
            node->pAdjPtrs = myArgPtrs.emplace_back_multi<N>();
        }

        //  Return pointer on the new node
        return node;
    }

    void resetAdjoints()
    {
        for (Node& node : myNodes)
        {
            node.mAdjoint = 0;
        }
    }

    //  Clear, all blocklists
    void clear()
    {
        myDers.clear();
        myArgPtrs.clear();
        myNodes.clear();
    }

    //  Rewind, all blocklists together
    void rewind()
```

```
62      {
63          myDers.rewind();
64          myArgPtrs.rewind();
65          myNodes.rewind();
66      }
67
68      //  Set mark, all blocklists in sync
69      void mark()
70      {
71          myDers.setmark();
72          myArgPtrs.setmark();
73          myNodes.setmark();
74      }
75
76      //  Rewind to mark, all blocklists
77      void rewindToMark()
78      {
79          myDers.rewind_to_mark();
80          myArgPtrs.rewind_to_mark();
81          myNodes.rewind_to_mark();
82      }
83
84      //  Iterators
85
86      using iterator = blocklist<Node, BLOCKSIZE>::iterator;
87
88      auto begin()
89      {
90          return myNodes.begin();
91      }
92
93      auto end()
94      {
95          return myNodes.end();
96      }
97
98      auto markIt()
99      {
100         return myNodes.mark();
101     }
102
103     auto find(Node* node)
104     {
105         return myNodes.find(node);
106     }
107 };
```

The code should be mostly self-explanatory, since it mainly wraps blocklists. The padding on line 17 avoids false sharing when multiple tapes stored in a vector are manipulated concurrently from different threads, as explained in Section 3.5.

The node recorder on line 22 takes the number of child nodes as a template parameter, not as an argument. Leaf nodes, by definition, don't have children. They represent the inputs to a calculation. Unary functions like $log()$ or $sqrt()$ have one child. Binary operators like $+$ or $*$ have two. All this is always known at compile time. We will see in Chapter 15 that the number of inputs to an entire *expression* can also be determined at compile time with expression templates. It follows that we *can* pass it as a

template parameter instead of an argument, which may reduce run-time overhead.

To record a node means store it in the node blocklist. The local derivatives and child adjoint pointers are stored in their own blocklists, where their node can find them by pointer, set on lines 32 and down. There is nothing to set when the node is a leaf, and this is something known at compile time. C++17 implements a new construct for these situations, called *constexpr if*, which we use on line 32. A constexpr if, as its name indicates, evaluates at compile time. A call to $recordNode()$ instantiated with $N = 0$ does not *compile* the block of code starting line 32, in a similar manner as if it was surrounded by pragmas like "#if 0 … #endif." This code is executed every time an operation happens, so we must optimize it to the extreme, evaluating all we can at compile time to eliminate as much run-time overhead as possible.

We have a method to reset all adjoints to zero (which we will not use), one to clear the tape (by clearing all its blocklists), followed by methods to rewind the tape, set a mark, and rewind to the mark. We explained these notions in detail earlier, and their implementation is entirely delegated to the underlying blocklists. The only comment is that all the blocklists must be rewound and marked synchronously.

Finally, the tape provides iterators to the usual first and (one after) last records, as well as the marked record and a record found by address. This is all entirely delegated to the blocklist iterator, explained in detail earlier.

10.3 THE NUMBER CLASS

The last component of the AAD library is the custom Number class, including operator overloading and adjoint propagation utilities. Remember its purpose: when calculation code instantiated with this number type is executed, its mathematical operations are not only evaluated, but also recorded on tape, so that adjoints may be back-propagated through the tape at a later stage to compute differentials in constant time. The code is found in AADNumber.h.

Data members

The code mainly follows the logic of the toy code from the previous chapter. One difference, explained in Section 10.1, is that the toy code stored values on the Node for simplicity, whereas it is both more convenient and more efficient to store it on the Number. It follows that a Number stores its value in addition to a pointer on its Node. A Number doesn't have any other nonstatic data members.

A Number also provides accessors on its value and adjoint.

```
class Number
{
    double myValue;
    Node* myNode;

    //  ...

public:

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

    //  ...
```

For the convenience of (friend) operator and mathematical function overloads, we also provide some (private) shortcut accessors to the data stored on a Number's Node:

```
    //  ...

private:

    //  Convenient access for friends

    //  Access to local derivatives

    //  For unary functions
    double& derivative() { return myNode->pDerivatives[0]; }
    //  For binary operators and functions
    double& lDer() { return myNode->pDerivatives[0]; }
    double& rDer() { return myNode->pDerivatives[1]; }

    //  Access to child adjoints

    //  For unary functions
    double*& adjPtr() { return myNode->pAdjPtrs[0]; }
    //  For binary operators and functions
    double*& leftAdj() { return myNode->pAdjPtrs[0]; }
    double*& rightAdj() { return myNode->pAdjPtrs[1]; }

    //  ...
```

Thread local tape

Like in the toy code, the tape is a static member of the Number class, but with two twists. First, the tape itself is a global instance in the application. What the Number class (statically) holds is a *pointer* on it.

Second, the static Tape pointer is *thread local*, see Section 3.4 in Chapter 3. This means that every thread has *its own copy* of it. Where exactly Number::tape points depends on the thread who's asking. We need this feature to differentiate parallel code in Chapter 12. More generally, this is what makes AAD applicable to parallel algorithms. All the operations conducted on a given thread are recorded *on that thread's tape*. In particular, concurrent threads don't interfere when recording operations.

```
class Number
{
    //  ...

public:

    //  Static access to tape
    static thread_local Tape* tape;

//  ...
```

The main thread's tape is global to the application and declared in AAD.cpp. The static pointer on the Number type is initialized *on the main*

thread to refer to that global tape. The following code is in AAD.cpp:

```
#include "AADnumber.h"

//  Declaration of the main thread's tape, global to the application
Tape globalTape;
//  Set the Number's static thread-local pointer to the global tape
//  from the main thread
thread_local Tape* Number::tape = &globalTape;
```

It is important to understand that the result of the instruction:

```
#include "AADnumber.h"

Tape& useThatTape = *Number::tape;
```

depends on the thread who executes it. If it is executed on the main thread, it returns a reference to globalTape, because the main thread's pointer is initialized to point to globalTape in AAD.cpp when the application opens. On another thread, Number::tape refers to a different tape, whichever was assigned *on that thread* by the execution of:

```
#include "AADnumber.h"

//  Called on some thread
Number::tape = &tapeForThatThread;
```

This point will be further clarified in Chapter 12, when we instrument our parallel simulation code. In the meantime, we continue with the Number class.

Node creation

We put a private method $createNode()$ on the Number class to create a new Node on a tape for a Number:

```
    //  ...

private:

    //  Create Node on tape,
    //      with the right number of children (arguments)
    template <size_t N>
    void createNode()
    {
        myNode = tape->recordNode<N>();
    }

//  ...
```

and use it to create Nodes instead of calling $recordNode()$ on the tape directly. This clarifies code and highlights the situations where a Node is created.

Leaf nodes Leaf nodes are the inputs to a calculation. They belong to Numbers that are not themselves the result of an operation involving other Numbers. These Numbers are the initial values of a calculation out of which the calculation is conducted and with respect to which the sensitivities of the final result are computed. It follows that leaf nodes are created in three situations, listed below:

```
    //  ...

public:

    Number() {}

    //  Construction out of a numeric value
    //      produces a leaf record on tape
    explicit Number(const double val) :
        myValue(val)
    {
        createNode<0>();
    }

    //  Assignment of a numeric value
    //  (i.e. overwriting)
    //      also produces a leaf record on tape
    Number& operator=(const double val)
    {
        myValue = val;
        createNode<0>();

        return *this;
    }

    //  putOnTape() explicitly called
    //      to produce a leaf record on tape
    void putOnTape()
    {
        createNode<0>();
    }

//  ...
```

where the *explicit* qualifier in the constructor will be explained shortly. Leaf nodes are created in three circumstances: when a Number is constructed with an initial value, when a Number is reinitialized with an initial value, and when the client code explicitly invokes $putOnTape()$ on a Number instance. These situations are illustrated in the client example below:

```
#include "AAD.h"

int main()
{
    //  Creates a leaf on tape for x
    //      from Number's constructor
    Number x(1.0);

    //  Use x in a calculation
    //      OK: x is on tape
    Number y = sqrt(x);

    //  Now y is also on tape as a unary node
    //      whose child is x's leaf node

    //  ...

    //  Starting another calculation

    //  Wipe or rewind tape first

    Number::tape->rewind();

    //  Reuse the variable x
    x = 2.0;
    //  This call also creates a leaf on tape for x
    //      from Number's assignment operator

    //  Use x in a calculation
    //      this works because x is on tape
    //      from the previous line
    Number z = 2 * x;

    //  ...

    //  Starting a third calculation
    //  Wipe or rewind tape first
    Number::tape->rewind();

    //  This calculation reuses x without changing its value
    //  Number t = exp(x);
    //  This can't work: x is not on tape,
    //      t's Node would have a dangling pointer

    //  In case we reuse an existing variable with its existing value,
    //      we must put it on tape explicitly

    x.putOnTape();

    //  Now x is on tape (as a leaf), we can use it in calculations

    Number t = exp(x);

    //  ...
}
```

This code illustrates the situations where a Number goes on tape as a leaf. Operator overloading records all mathematical operations on tape as unary or binary nodes with pointers on the arguments of the operation on tape. It follows that the arguments to any operation must be on tape when the operation is evaluated, or we have dangling pointers. When the arguments are themselves the results of previous operations, they would

be on tape automatically, as unary or binary nodes. But it is the client code's responsibility to put initial values on tape, explicitly, as leaf nodes.

This is performed in the constructor when a Number instance is created and given a numeric value. When an existing Number instance is reinitialized with another numeric value, this is performed in the assignment operator.

But it also happens frequently that a calculation would reuse existing Numbers as inputs, as opposed to creating or reinitializing Numbers purposely. For instance, in the simulation code of Chapter 6, an existing model's parameters are the initial values to a pricing. Those parameters were created and assigned a value on the construction of the model; they are not recreated or reinitialized for the purpose of the simulation. Before the calculation starts, all its inputs must be on tape. Numbers newly created or overwritten for the occasion are recorded as leaves from the constructor or assignment operator. But existing Numbers input to the calculation, like the parameters of a model prior to simulation, must be recorded manually, explicitly, with a call to $putOnTape()$.

We also develop a free function to put a collection of Numbers on tape (in AAD.h):

```
1  template <class IT>[numbers = none]
2  inline void putOnTape(IT begin, IT end)
3  {
4      for_each(begin, end, [](Number& n) {n.putOnTape(); });
5  }
```

Operations on Numbers with arguments not recorded on tape are by far the primary cause of bugs in AAD instrumented code. Tapes are wiped or rewound in between calculations and existing inputs are frequently reused. When an operation is evaluated with Number arguments not on tape, the resulting Node gets dangling pointers on the arguments' adjoints. As a result, the back-propagation step from that node writes into random memory instead of correctly accumulating adjoints. The result is random: the application can crash, corrupt memory, or return incorrect results. All inputs to a calculation must be recorded as leaf nodes on tape before the first operation in the calculation is evaluated. When the tape is wiped or rewound, all the inputs must be put back on tape. When the in-

puts are not created for the occasion, it is the responsibility of the client code to put them on tape with a call to $putOnTape()$, either individually or as a collection.

This is fundamental and a major design concern for instrumented code, as we will see when we instrument simulations in Chapter 12. Furthermore, with multi-threaded code, we have one tape *per thread*. The inputs must be registered *on all the tapes*.

Note that we need not implement custom copy, assignment, or move semantics on the Number class. The default ones do the right thing. When a variable of type Number is assigned to another:

```
Number y;
//  ...
Number x = y;
```

the value of y is copied into x, but x does not get a *new* Node. The pointer y.myNode is copied into x.myNode, so that x refers to y's Node. This is the desired behavior: a Number refers to the last mathematical operation that produced its value. Assignment means that the same operation that created y's Node is responsible for the value of x.

This is worth repeating: *Assignment is not a mathematical operation, and it is not recorded on tape*. It simply redirects the pointer of the left-hand side to the Node of the right-hand side.

Operator nodes Operator nodes record operations on tape: unary functions like $log()$ or $sqrt()$, binary operators like $+$ or $*$, or binary functions like $pow()$. Operations are not limited to standard C++ mathematical functions and operators. For example, the Gaussian (exact) density and (approximate) cumulative distribution are building blocks in financial libraries. We consider them as standard mathematics and create a Node for them, even though they are not part of C++ standard and are defined in our header gaussians.h.[6] It is advisable to overload all frequently called, low-level mathematical functions in the same manner. It results in a more efficient, more accurate differentiation.

This is best seen in the case of the cumulative Gaussian distribution, which does not admit a closed-form and is typically computed with polynomial approximations.[7] Our code in gaussians.h implements Zelen and Severo's approximation of 1964:

```
1  //  Normal density
2  inline double normalDens(const double x)
3  {
4      return x<-10.0 || 10.0<x ? 0.0 : exp(-0.5*x*x) / 2.506628274631;
5  }
6
7  //  Normal CDF (N in Black-Scholes)
8  //  Zelen and Severo's approximation (1964)
9  inline double normalCdf(const double x)
10 {
11     if (x < -10.0) return 0.0;
12     if (x > 10.0) return 1.0;
13     if (x < 0.0) return 1.0 - normalCdf(-x);
14
15     static constexpr double p = 0.2316419;
16     static constexpr double b1 = 0.319381530;
17     static constexpr double b2 = -0.356563782;
18     static constexpr double b3 = 1.781477937;
19     static constexpr double b4 = -1.821255978;
20     static constexpr double b5 = 1.330274429;
21
22     const auto t = 1.0 / (1.0 + p*x);
23
24     const auto pol = t*(b1 + t*(b2 + t*(b3 + t*(b4 + t*b5))));
25
26     const auto pdf = normalDens(x);
27
28     return 1.0 - pdf * pol;
29 }
```

The normal density involves four operations, the cumulative distributions 18 (including the call to $normalDens()$). If we simply instrumented $normalCdf()$ by templating its code, its evaluation with the Number type would create 18 Nodes on tapes. Its differentiation would propagate adjoints backward through these 18 Nodes. The result would not necessarily be accurate:

The differential of an accurate approximation is not necessarily an accurate approximation of the differential.

This being said, we *know* that the derivative of $normalCdf()$, by definition, is $normalDens()$. By considering it as a standard mathematical function, we create on overload, which, when evaluated, like all other standard functions and operators, creates a single Node on tape with the exact derivative. As a result, the differentiation is both measurably faster and significantly more accurate.

The code of the overloaded operators and functions will be discussed shortly. Functions and operators are overloaded so they compute not only a result, but also its derivatives to the arguments, and create an operator node on tape. All frequently called, low-level functions should be overloaded along with the standard C++ operators and functions. The standard functions are all unary or binary. The Gaussian functions are also unary, so all our overloads admit one or two arguments. It doesn't have to be the case. There is nothing in the code that prevents overloading functions with three or more arguments. To some extent, the expression templates of Chapter 15 overload an entire *expression* and store it on a single Node with an arbitrary number of arguments.

To help overloading, we write two private constructors on the Number class for the results of the overloads. These constructors are only called from the overloads and all overloads are friends of the Number class; hence, we declare them private to prevent accidental calls. The unary operator constructor is listed below:

```
//  ...
private:

    //  Unary
    Number(Node& arg, const double val) :
        myValue(val)
    {
        createNode<1>();
        myNode->pAdjPtrs[0] = &arg.mAdjoint;
    }
//  ...
```

It sets the created Number's value, constructs a unary node on tape for it, and sets the pointer on the Node to correctly refer to the adjoint of the argument. Note that the private constructor does *not* set or calculate the derivative. This is done directly in the different overloads. The binary constructor is very similar:

```
//  ...
    //  Binary
    Number(Node& lhs, Node& rhs, const double val) :
        myValue(val)
    {
        createNode<2>();
        myNode->pAdjPtrs[0] = &lhs.mAdjoint;
        myNode->pAdjPtrs[1] = &rhs.mAdjoint;
    }

//  ...
```

Similar constructors could be developed for n-ary operators if required.

Conversions

We now address the *explicit* qualifier in the Number constructor. It prevents *implicit* conversion, like in:

```
    Number g(const Number x);
    double y;
    //  ...
    Number z = g(y);
```

If the Number's constructor was not marked explicit, this code would compile and silently convert the double y into a Number x when g is invoked. The explicit qualifier causes a compilation error, attracting attention to the conversion and preventing accidental conversions. When a conversion is indeed desired, it compiles when made explicit:

```
    Number z = g(Number(y));
```

Accidental conversions are the second-most-common cause of bugs and inefficiencies in instrumented code. It is best forcing explicit conversions on client code. For the same reason, conversions from Numbers to doubles is implemented with explicit conversion operators:

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

When doubles are used in place of Numbers, operations are not recorded on tape on evaluation, and adjoints are not propagated on back-propagation. The chain rule is broken, resulting in wrong sensitivities.

When Numbers are used in place of doubles, their evaluation is subject to AAD overhead: nodes are created on tape and local derivatives are computed and stored with the Node. In the situations where Numbers are not necessary, they cause substantial and unnecessary overhead.

Instrumented code must be very clearly written to use doubles or Numbers where relevant. We will see in Chapter 12 that this is even a key design consideration for an efficient instrumentation. Explicit constructors and conversion operators force the client code to express these choices explicitly. They prevent compilation of nonexplicit conversions, attracting developers' attention to the places where conversions occur, hence mitigating a frequent cause of bugs and inefficiencies.

We also provide (in AAD.h) a function that converts a *collection* of Numbers into a collection of doubles and vice versa, with STL iterator syntax:

```
template<class It1, class It2>
inline void convertCollection(It1 srcBegin, It1 srcEnd, It2 destBegin)
{
    using destType = remove_reference_t<decltype(*destBegin)>;
    transform(srcBegin, srcEnd, destBegin,
        [](const auto& source) { return destType(source); });
}
```

where $decltype(expression)$ is a standard C++11 compile-time utility that gives the type of the expression, so $decltype(*destBegin)$ is the type of the elements in the destination collection pointed by the iterator $destBegin$. The compile-time function $remove\_reference\_t()$, as the name indicates, gives the raw type, turning $T\&$ or $T\&\&$ into T. We can apply this conversion utility, for example, to convert a vector of templated types into a vector of other templated types, and back:

```
template<class FROM, class TO>
vector<TO> convertVector(const vector<FROM>& src)
{
    //  Allocate
    vector<TO> dest(src.size());

    //  Convert
    //  If TO = FROM, we get a copy
    //  If not, we get the correct conversion
    convertCollection(src.begin(), src.end(), dest.begin());

    return dest;
}
```

Back-propagation

Numbers store a pointer to their Node on tape. Back-propagation consists in the propagation of adjoints through the tape in reverse order from the final result's Node to the first Node on tape. Prior to back-propagation, all adjoints are set to 0 (although adjoints are always initialized to zero, so it is generally not necessary to reset them), and the adjoint of the final result is set to one. This is called *seeding the tape*. We write a convenient method on the Number class to propagate adjoints from the caller Number's Node:

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

    //  Propagation

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

    //  Convenient overloads

    //  Set the adjoint on this Node to 1,
    //      then propagate from this Node to propagateTo
    void propagateAdjoints(
        //  We start on this number's Node
        Tape::iterator propagateTo)
    {
        //  Set this adjoint to 1
        adjoint() = 1.0;
        //  Find node on tape
        auto propagateFrom = tape->find(myNode);
        //  Propagate
        propagateAdjoints(propagateFrom, propagateTo);
    }

    //  Other overloads to propagate
    //      from this Node, after setting its adjoint to 1,
    //      to either start or mark
    void propagateToStart()
    {
        propagateAdjoints(tape->begin());
    }
    void propagateToMark()
    {
        propagateAdjoints(tape->markIt());
    }
//  ...
```

The first method $resetAdjoints()$ resets all adjoints on tape to zero. We don't use this method; it is only provided for completeness. The second is a static method that propagates adjoints backwards through the tape, from a starting point to an endpoint, both represented by tape iterators given as arguments. The third *nonstatic* method propagates adjoints from the Number instance's Node, which adjoint it first initializes to one, to a specified endpoint. The last two convenient overloads propagate in the same manner, to, respectively, the beginning or the mark on the tape. The

code uses tape iterators and the Node's propagation method and should be self-explanatory.

Const incorrectness

Before we discuss the code of the operator and function overloads, we illustrate how they work on a simple example and expose one of the main inherent flaws of AAD. Consider the following code:

```
#include "AAD.h"

//  A simple Multiplier class
template <class T>
struct Multiplier
{
    const T multiplyBy;

    Multiplier(const double m) : multiplyBy (m) {}

    T operator () (const T y) const
    {
        return multiplyBy * y;
    }
};

//  Instrumented code

//  Initialize tape first
Number::tape->rewind();

//  Initialize inputs, including data members
//  Note that puts inputs on tape, as leave nodes
Multiplier<Number> doubler(2.0);
Number y(5.0);

//  The calculation itself, in this case, a simple product
Number z = doubler(y);

//  Adjoint propagation
z.propagateToStart();

//  Display results
cout << z.value() << endl;                      //  10.0
cout << y.adjoint() << endl;                    //  2.0
cout << doubler.multiplyBy.adjoint() << endl;   //  5.0
```

This code illustrates, in a simple example, the general pattern for AAD instrumentation.

The Multiplier class is a simple function object that multiplies numbers by a constant. It is templated on its number type, but its constructor takes a double to initialize its templated data member $myltiplyBy$. Its operator (), marked const, returns the product of the argument by its data member $myltiplyBy$, without apparent modification of the Multiplier's internal state or the argument.

The rest of the code is a simple instrumentation that evaluates and differentiates a product. Line 20 initializes the tape: it is faster to rewind than wipe it. Line 25 initializes a Multiplier<Number>, baptized “doubler,” with a multiplicative constant 2. This invokes the Multiplier's constructor, which in turn invokes the Number's constructor on line 9 to build *doubler*'s *multiplyBy* data member. *This puts multiplyBy on tape.* Line 25 initializes a Number with value 5, which puts on tape.

We see that the order of the instructions is important here: first, initialize the tape, then, initialize the inputs, including data members, so they are recorded on tape, and only then perform the calculation.

Line 28 conducts the calculation. The call to 's operator () evaluates the product on line 13. The operator , like all other arithmetic operators and mathematical functions, is overloaded for the Number type, as we will see shortly, so line 13 not only evaluates the product, it also compute its derivatives to the arguments and , and builds a binary node on tape, correctly linking it to the arguments' Nodes. The result is held by the Number variable , which node is the binary node created by the overloaded product.

Line 31 propagates adjoints, from the result , which adjoint is set to one, to the start of the tape. After the propagation completes, the adjoint of every node on tape correctly accumulated to . We can read the final result , as well as in 's adjoint and in 's adjoint.

This works nicely and illustrates the typical instrumentation pattern: initialize the tape, put inputs on tape, calculate with the Number type, propagate, pick differentials.

But it should be apparent that the instrumented Multiplier's operator on line 11 *is no longer correctly const*, and neither is its argument . As a result of the product on line 13, the adjoints of and 's Nodes are held *by non-const pointers* on 's Node, and during the adjoint propagation on line 31 the adjoints of and are *modified.* Although the Multiplier's operator and its argument are marked const on line 11, its data member and argument's adjoints are modified as an indirect result of its evaluation.

AAD is inherently const incorrect.

This rather subtle characteristic is not without consequence for the design of instrumented code, especially in a parallel context. The Multiplier's operator is const. It may be called safely from concurrent threads for evaluation when instantiated with the double type, but it cannot be called concurrently for differentiation when instantiated with the Number type. Contrary to a Multiplier<double>, a Multiplier<Number> is a *mutable object* in the sense of Section 3.11. For the exact same reason, a model is mutable in the context of an instrumented simulation. We have seen in Chapter 7 that it is safe to simulate paths concurrently with the same Model<double> for evaluation. It is no longer the case for differentiation with a Model<Number>. An instrumented model is mutable, so each thread must use its own copy of it, or data races will occur. Our instrumented simulation code of Chapter 12 is designed accordingly.

It is very annoying that we can no longer trust const qualifiers in the context of AAD. We must constantly remember that AAD modifies the adjoints of all Numbers involved in calculations, so all Numbers, even the ones that are only read, never explicitly written into, are always mutable when instrumented. An intrusive alternative would be to modify instrumented code and remove all const qualifiers of all Numbers involved in calculations and all objects that store such Numbers. This solution is impractical and we don't recommend it. One of the difficulties of programming with AAD is not being able to trust constness.

In practice, this means that we need another private accessor on the Number class to access a Number's Node, by non-const reference, even for a const Number. Overloads use it to set the child adjoint pointers on the operator nodes:

Operator overloading

Finally, we implement operator overloading for instances of the Number class. Remember that every mathematical operation that involves Numbers should implement all of the following:

1. Evaluate the operation and store its result on the resulting Number.

2. Create an operator node on the tape, unary or binary depending on the operation. Set the child adjoint pointers on the Node to the adjoints of the operation's arguments.
3. Evaluate the derivatives of the operation to its arguments, store them on tape, and link the pointers on the operator node.

We implemented private unary and binary constructors on the Number class to facilitate overloading and implement the second item on the list. This considerably simplifies the development of the overloads. For instance, we list below the overloaded operator :

This code is executed every time two Numbers are multiplied with the operator . Line 8 computes the result. Line 10 calls the private binary constructor on the Number class, constructing a Number to hold the result of the multiplication together with its binary node on tape. The constructor also correctly sets the child adjoints' pointers on the operator node to the adjoints or and , which Node is accessed (and const-casted) by calls to the private method . Finally, lines 12 and 13 set the two local derivatives on the operator node:

and return the result Number. This Number stores the result of the multiplication and points to the newly created operator node on tape. When this Number, at a later stage, becomes an argument to a subsequent operation, the operator node of that operation will point to its adjoint on tape. As the calculation evaluates its sequence of operations, a network of nodes is recorded on tape, linked to one another with the argument adjoint pointers. These pointers are the edges of the DAG held on tape, the veins where adjoint propagation flows in reverse after the calculation completes.

This overload is called when two Numbers are multiplied. What happens when a Number is multiplied by a double, say a constant, as here?

Note than any double is a constant as far as AAD is concerned. It doesn't matter if the double is the result of a calculation. That calculation is out of the tape. Only the operations involving at least one Number, and resulting in a Number, are *active*. The multiplication above is not a binary operation anymore, not in the sense of AAD. This is the *unary* operation (*2), an operation that doubles its argument. We must implement that properly for the sake of performance, with a different overload for the operator :

Note that although the code is similar, it is the unary private Number constructor that is invoked now on line 6 so a unary node is recorded on tape, linked to Number but not the constant. We even need a third overload when the double is on the left:

All the binary operators and functions must implement the three overloads. This is a strong optimization with measurable impact in cases of practical relevance. It is substantially faster to construct a unary node and propagate through one than with binary nodes. Although we don't list the code here in the interest of conciseness, AADNumber.h implements the binary operators and functions , , , , , and in the exact same way.

We must also overload the operators a la , both with Number of double arguments. This is easy; the code for the overload, for example, is listed below. The others are found in AADNumber.h.

Unary operators and functions follow the same pattern, although (evidently) they don't need an overload for doubles. We list below the overloads for , , , and , as well as the trivial unary and :

Following the earlier discussion regarding the Gaussian density and cumulative distribution, we overload these in the same manner:

Finally, we need comparison operators since calculation code often compares Numbers by value:

We have ==, !=, >, >=, <, and <= to overload. Each overload in fact consists of three overloads in case we compare Numbers to doubles, where the double may be on the left or right of the comparison. This is a total of 18 overloads in AADNumber.h.

This concludes the design of the Number type and our AAD library. The code is in our repository, files AADNode.h, AADTape.h, AADNumber.h., and, of course, blocklist.h. The static variables are defined in AAD.cpp. All the headers are listed in AAD.h, so client code may include that header alone. With the exception of a dependency on gaussians.h, these files form a self-contained, professional AAD library in C++. Its performance is excellent and even impressive compared to traditional differentiation, although somewhat short of handwritten adjoint code. Performance will be further improved in Chapter 15.

10.4 BASIC INSTRUMENTATION

The library may be used identically to the toy code in the previous chapter. Include AAD.h, template calculation code, initialize the tape, put the inputs on tape, evaluate with Numbers, propagate adjoints, and pick derivatives. We described instrumentation in deep detail in a simple example on page 391. We provide another example here, and, of course, we fully instrument our simulation code in Chapter 12.

The full definition of the Number class in AADNumber.h contains 650 lines of code. We don't list it here, referring to our repository for reference instead.

NOTES

1 For the instrumentation of the Gaussian density and cumulative distribution, which we consider as standard mathematical functions, even though they are not part of standard C++.

2 For memory management.

3 After testing many possible solutions and carefully measuring their performance.

4 Of course, we have much larger blocks in practice, default being 13,384 Nodes in a block.

5 When the blocklist is being overwritten, points to the *current* slot, not necessarily the last one in memory.

6 This is the reason why the AAD library, despite being self-contained, has a dependency to gaussians.h.

7 See https://en.wikipedia.org/wiki/Normal_distribution#Numerical_approximations_for_the_normal_(