Debugging AAD Instrumentation

The AAD libraries of Chapters 10 and 15 were extensively tested for a long time in a vast number of contexts. There shouldn't be many bugs left there. In case readers do find bugs when experimenting with the libraries, we shall be most grateful if they kindly notify Wiley so we can correct the code in the online repository. Again, and following the many hours spent tracking and removing bugs and inefficiencies, we should expect the AAD libraries to be mostly clean.

This being said, the purpose of this publication is to let readers instrument their own code and differentiate it in constant time with the AAD libraries. It should be apparent from Chapters 12, 13, and 14 that a correct instrumentation is a long, difficult work, in our experience, more demanding than the development of the AAD libraries themselves. A lot may go wrong. This appendix offers general advice for debugging instrumented code, and reveals the one single tactic that saved us months of tedious debugging work while professionally developing AAD-powered systems, almost immediately identifying most AAD bugs in instrumented code. This debugging tactic is one of the key ingredients that allowed us to implement AAD throughout Danske Bank's production systems in a reasonably short time in the early 2010s.

First of all, it should be clear that the general advice of Section 3.20 for debugging and profiling serial and parallel code holds with AAD, and readers are encouraged to follow the steps articulated there in case their instrumented code doesn't produce the correct results, crashes, or takes too long. With AAD, however, a number of additional steps should be taken to improve debugging conditions:

- Switch to traditional AAD, without expression templates. Recall from Chapter 15 that this is done by changing the definition of the macro $AADET$ to $false$ in AAD.h. It is virtually impossible to debug expression templates, whereas traditional AAD provides human-readable information in the debugger's window. Incorrect instrumented code should be incorrect with both versions, and corrected code should remain correct in both cases. Traditional AAD is slower, but easier to debug, and it doesn't hurt switching off AADET for debugging as long as it is reestablished after the instrumented code is fixed.
- Switch off selective instrumentation and template the whole calculation when debugging. Reestablish selective instrumentation step by step after debugging, testing for correctness after de-templating every piece of code. Similarly, remove all explicit conversions while debugging and reestablish them afterwards, one by one.
- Wipe the tape instead of rewinding it in between calculations. To rewind and reuse the tape by overwriting is an optimization, but it makes it more difficult to assess and debug instrumented code. It is best always to wipe and start anew in between calculations for debugging. The easiest way to achieve this is to make $Tape::rewind()$ different for debug and release:

```
    //  Tape class, AADTape.h
    void rewind()
    {

#ifdef _DEBUG

        //  In debug mode, clear instead

        clear();

#else

        //  Normal code in rewind()

        if (multi)
        {
            myAdjointsMulti.rewind();
        }
        myDers.rewind();
        myArgPtrs.rewind();
        myNodes.rewind();

#endif

    }
```

Visual Studio automatically defines the macro _DEBUG for code compiled in the debug configuration. Release code is optimized for performance while debug code should maximize the clarity and relevance of the debugging information. For this reason, it is sometimes worth compiling different code for debug or release. With Visual Studio, this is easy to achieve with simple macros, as demonstrated above. When code is compiled in the debug configuration, every call to *rewind*() clears the tape instead. When compiled in release mode, the normal code for rewinding the tape is executed.

- Debug check-pointing with particular care. Inspect the results of partial differentiations separately. This may help identify which piece of the algorithm is incorrectly differentiated. When all pieces are correctly differentiated, the flaw may be in the check-pointing logic itself.
- Evidently, run algorithms over small data sets for debugging. AAD takes a long time in debug mode. Run no more than 100 simulations over big steps for Monte-Carlo simulations, or equivalently small data sets for other applications.

AAD is particularly hard to debug because errors typically manifest on adjoint propagation, whereas the cause always occurs at evaluation time, when operations are incorrectly recorded on tape. At back-propagation time when errors are revealed, however, all that remains is the flawed record on tape, and it is virtually impossible to identify the responsible operation in the code. For this reason, it may take hours and days, and a lot of guessing and logging to find the incorrect pieces of code that caused the flawed records. If only we could catch them at recording time.

In our many years of instrumenting AAD in professional software and academic presentations, the primary cause of bugs specific to AAD, in fact, almost every single AAD bug we ever encountered, was caused by operations with *arguments missing from the tape*. A line of code like:

```
Number z = x + y;
```

is executed, while x or y is not on tape. There are many reasons why it may be so. Maybe these variables are initial inputs, not the result of prior operations, and we forgot to put them on tape as discussed page 381? Maybe they were on tape, but the tape was wiped and they were not put back after that? Or maybe, in a multi-threaded environment, they are simply not on the right tape? Whatever the reason, it is very often the case that some arguments of a recorded operation are missing from the tape. This creates dangling pointers, in the record, to nonexistent argument adjoints, and invariably causes crashes or wrong results on back-propagation.

Once we realize that most AAD bugs are caused by arguments not on tape, it is very easy to catch those *at evaluation time*. Remember from Chapter 10 that the recording of every operation in the overloaded operators and functions calls the private method $node()$ on the active arguments to acquire a pointer to their node on tape. It follows that we can insert code in this method, compiled only in a debug configuration, that checks that the argument node is on tape, at the time where the operation is recorded:

```
1       //  Number class, AADNumber.h
2       Node& node() const
3       {
4
5   #ifdef _DEBUG
6
7           //  Check that node is on tape,
8           //      only in debug
9
10          //  Find node on tape
11          auto it = tape->find(myNode);
12
13          //  Not found
14          if (it == tape->end())
15          {
16              throw runtime_error("Put a breakpoint here");
17          }
18
19  #endif
20
21          //  Normal code, executed in debug or release
22
23          return const_cast<Node&>(*myNode);
24
25      }
```

Run the code in debug, putting a breakpoint on line 16. Execution will stop on the breakpoint whenever an operation is recorded with an argument not on tape, revealing the context of the operation in the debugger, which makes it easy to identify and fix the flaw.

For example, we put a bug in our Black and Scholes code of Chapter 6:

```
    //  BlackScholes class, mcMdlBs.h
    void setParamPointers()
    {
        myParameters[0] = &mySpot;
        myParameters[1] = &myVol;
        myParameters[2] = &myRate;
        myParameters[3] = &myRate;
    }
```

Can you see the bug? We wrongly set the pointer $myParameters[3]$ to the address of $myRate$. It should be the dividend $myDiv$. As a result, the differential to the dividend is always 0 but what is returned to client code, under the wrong label “div,” is the sensitivity to the interest rate. We have an incorrect differential, and it may be hard to identify the cause. There is a lot of code to go through, in the model, product, template algorithm, etc., and such typos are easily skipped. Running in debug mode with a breakpoint on line 16 in our $Number :: node()$ code, execution stops immediately and the call stack points to a subtraction in $BlackScholes :: init()$:

```
void init(
    const vector<Time>&         productTimeline,
    const vector<SampleDef>&    defline)
        override
{
    //  Pre-compute the standard devs and drifts over simulation timeline
                const T mu = myRate - myDiv;

    //  ...
```

which immediately tells us that mu does not record correctly, because either $myRate$ or $myDiv$ is not on tape. We know that parameters are put on tape altogether, so there must be a flaw in the registration of the parameters. It follows that we know to look into

$$BlackScholes :: setParamPointers()$$

at which point we find the bug immediately.

This is a trivial example, of course, where the bug could be identified by testing and reasoning without actual debugging. But the same tactic helped us identify a vast number of subtle hidden bugs in sophisticated code that would have been long and hard to find otherwise.