CHAPTER 11 Discussion and Limitations

Before we apply our AAD library to instrument our simulation code from Part II, we briefly discuss some important aspects and limitations of AAD that will guide our work.

11.1 INPUTS AND OUTPUTS

We minimized AAD overhead as much as possible thanks to efficient memory management, and we minimize it further in Chapter 15 with expression templates, but AAD overhead cannot be eliminated. Even if we did not apply operator overloading or a tape, and manually coded adjoint computations along with the calculation code, AAD differentiation would still take around three times the cost of one calculation, as seen in Chapter 8. We come close to this theoretical limit with an automatic approach in Chapter 15.

AAD is only relevant when the number of inputs is large (certainly larger than four), as is almost always the case for financial valuation. With a low number of inputs, bumping is generally faster.

Further, AAD computes the differentials of *one* scalar result in constant time in the number of differentials, but the differentials of *multiple* results take linear time in the number of results. This is the case even when the calculation itself takes constant time or close in the number of the results, when all results are computed together and share a vast fraction of the computations, like when pricing multiple products in a simulation over the same paths. Differentiation still takes linear time, because the back-propagation of adjoints must be conducted separately for each result.

In the context of financial simulations, we should therefore expect AAD to produce constant time risk reports *for one transaction*, or one aggregated value of multiple transactions, but, when we need an itemized risk report for every transaction in a book, we should expect linear time in the number of transactions.

This is somewhat frustrating, because we designed our simulation library in such a way that multiple transactions may be valued simultaneously in virtually constant time by sharing the simulated paths. This being said, AAD's linear time in the number of outputs is theoretical and *asymptotic*. While it is inescapable to spend time $\alpha + \beta m$ to differentiate m results with AAD, if we manage to make β orders of magnitude smaller than α, we will be done in "almost constant time" in situations of practical relevance. How to do this, and whether it is at all possible, depends on the differentiated code. In the context of financial simulations, we achieve such "almost constant time" in Chapter 14 with the development of special support in the AAD code. In the meantime, we only consider the differentiation of a single result.[1]

11.2 HIGHER-ORDER DERIVATIVES

AAD also computes first-order derivatives. To compute second-order derivatives would take "AAD over AAD," where the adjoint calculation of first-order derivatives becomes part of the calculation code that is

itself instrumented with AAD to produce second-order derivatives. That is not viable in practice.

Despite active research in the production of higher-order derivatives with AAD, and the claim from some commercial AAD frameworks to produce second- and higher-order AAD risk efficiently, there exists, to date and to our knowledge, no solution of practical relevance. It is generally admitted among AAD academics and professionals that second derivatives are best produced by "bumping over AAD," where the first derivatives are produced with AAD but higher-order derivatives are produced by bumping.

This means that AAD provides acceleration by one order in the production of derivatives, but no more: constant instead of linear time for the production of first-order derivatives, linear instead of quadratic for the second-order derivatives, and so forth.

11.3 CONTROL FLOW

We have already pointed out that AAD differentiates mathematical calculations *along a fixed control path*. It does not differentiate control flow: "if this then that" statements and friends. Control flow introduces discontinuities, which cannot be differentiated with AAD or otherwise, so this is not necessarily a problem, but something that must be kept in mind when instrumenting code.

One example is how we defined the payoff of a barrier option in Section 6.3. The payoff of a barrier is discontinuous, so the MC estimate of its price is not differentiable. We smoothed its payoff with a well-known "smooth barrier approximation" to remove the discontinuity.

The differentiation of discontinuous functions, including control flow, is not resolved in the differentiation library with differentiation methods, but in the valuation library with smoothing methods. To smooth a

function means to approximate it by a close continuous function. The work relates to the differentiated function, not its differentiation. Smoothing in general terms, and smoothing of financial cash-flows in particular, is introduced and discussed in [77] and [78].

11.4 MEMORY

Finally, and most importantly, it should be clear that AAD may consume an *insane* amount of RAM. *Every single mathematical operation*, every sum, difference, product, division, and so forth, is recorded on tape. The billions of such operations involved in a practical valuation take considerable space in memory. It has been estimated through both theoretical and empirical means (see [96] and [90]) that tapes consume around 5 GB of RAM per second per core in the non-instrumented calculation. That would limit AAD to calculations faster than around 12 seconds of CPU time on a 64 GB workstation, and 2 seconds.cpu on a 16 GB laptop (we must leave *some* RAM for the OS).

It gets worse: in order to traverse the tape *efficiently* for back-propagation, it should fit in the L3 cache, that is around 50 MB for the best mainstream CPUs available. That would limit AAD to calculations under 0.01 seconds.cpu and preclude it for pretty much all cases of practical relevance.

Path-wise AAD simulations

The solution consists in computing derivatives separately on parts of the calculation and aggregating them in the end. In the case of MC simulations, the obvious solution is to differentiate path-wise. The differential of the average payoff among N paths is the average of the differentials over each path. Path-wise differentials can be computed by conducting both the computation and the adjoint differentiation for each path separately. The final differentials are averages of path-wise differentials.

Mathematically: one path number i consists in the vector $(S^i_{T_j})_{1\leq j\leq J}$; see 5.2. This vector is a function of the Gaussian random vector G_i for the path, and a set of model parameters a: initial asset prices, volatility, and so forth:

$$(S^i_{T_j})_{1\leq j\leq J} = h(G_i, a)$$

A payoff is a function g of the path. The MC estimate of the value of the corresponding instrument is:

$$\frac{1}{N}\sum_{i=1}^{N} g(S^i_{T_j})$$

Hence:

$$\frac{\partial}{\partial a}\left(\frac{1}{N}\sum_{i=1}^{N} g(S^i_{T_j})_{1\leq j\leq J}\right)$$

$$= \frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial a}\left[g\left(S^i_{T_j}\right)_{1\leq j\leq J}\right]$$

$$= \frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial a}\{g[h(G_i, a)]\}$$

The generation and evaluation of one path takes a few microseconds.cpu in general; see our results from Chapter 6. Even in the case of an xVA on a monster netting set, it takes less than a millisecond, and, in extreme cases, perhaps up to 0.01s. Hence, memory consumption for path-wise derivatives is limited to 50 MB in the worse case, and always fits in the L3 cache of high-end modern CPUs.

Practically, we evaluate path-wise derivatives

$$\frac{\partial}{\partial a}\{g[h(G_i, a)]\}$$

with AAD over one path, then we wipe the tape for the processing on the next path. The size of the tape is related to the processing time of one single path, hence, always below 50 MB and generally substan-

tially less than that. The path-wise derivatives are stored and averaged in the end.

Introduction to check-pointing

As a side note, we separated the generation h from the evaluation g of the paths in our notations, and we can further split the expression of our path-wise derivative into:

$$\frac{\partial}{\partial a}\{g[h(G_i, a)]\} = \frac{\partial h(G_i, a)}{\partial a}\frac{\partial g[h(G_i, a)]}{\partial h}$$

This means that we *could* reduce RAM usage even further by computing the derivatives for the generation and the evaluation of the path separately, and then use this formula for the computation of the complete path-wise derivative. This is a multidimensional formula, $\frac{\partial h(G_i,a)}{\partial a}$ is the Jacobian of the components of the path to model parameters, and the whole operation is expensive. The check-pointing methodology discussed in Chapter 13 conducts this calculation *in constant time*, without the production of a Jacobian, or a matrix product, while limiting RAM usage to recording h and g separately.

In practice, there is no need to check-point this calculation or split path processing into generation and evaluation for the purpose of differentiation. Tapes for the full processing of a path are small enough. We only made this comment to show that we *could* further split the production of derivatives, and introduce the important check-pointing technique, which we develop in the dedicated Chapter 13. Check-pointing is also applied to its full extent in our publication [31] in order to efficiently differentiate through the Longstaff-Schwartz algorithm.

MC is a "lucky" case where the RAM consumption problem is easily overcome with path-wise differentiation. In addition, such path-wise computation of derivatives may be conducted in parallel over paths, as we demonstrate in the next chapter.

Evidently, outside of MC, it is not always possible to separate computations in such a trivial manner. For example, multidimensional FDM is too slow for a self-contained AAD differentiation, but not as easily separable as MC. FDM is step-wise, generally back to front, and the calculations for each time step depend on the results for the previously calculated time step.

Check-pointing provides solutions for all such cases. For the AAD differentiation of FDM with check-pointing, see [90]. In the next chapter, we differentiate our simulation code with path-wise AAD without the need for check-pointing. In Chapter 13, we implement check-pointing and apply it to the problem of differentiating through the *calibration* of the local volatilities of Dupire's model to market data and discuss the general case of the differentiation of any model through its calibration.

NOTE

1 Even though that single result may be the aggregated value of a large number of transactions.