# CHAPTER 12 Differentiation of the Simulation Library

## 12.1 ACTIVE CODE

In this chapter, we apply the AAD library from the previous chapter to differentiate our simulation library from Part II. More precisely, we apply AAD to compute the differentials of prices to *model parameters*. In the next chapter, we discuss differentials of prices to *market variables*, also called market risks or *hedge coefficients*.

What will clearly appear is that an efficient AAD library is certainly necessary, but absolutely not sufficient for the efficient differentiation of complex valuation code. It takes work, art, and skill to correctly instrument calculation code. An efficient AAD library provides all the necessary pieces, but to articulate them together in an efficient instrumentation requires an intimate knowledge of the instrumented algorithms (known as *domain-specific knowledge*) and a comprehensive understanding of the inner mechanics of the AAD library. The notion that one can template calculation code, link to an efficient AAD library,

and obtain differentials with AAD speed is a myth. Instrumentation takes at least as much skill and effort as the production of the AAD library itself. A naive instrumentation does produce differentials in constant time, but it may be a very long constant time, often slower than finite differences. While the notion of an efficient instrumentation depends on the differentiated code, this chapter provides general advice and battle-tested methodologies while instrumenting the simulation code of Part II with remarkable results.

It is especially hard to instrument calculation code that was not written with AAD in mind to start with. We wrote our simulation library knowing that we would eventually instrument it for AAD. This will make its instrumentation substantially easier. In fact, we wrote templated code to start with throughout the simulation library. But not all of it. In particular, we did *not* template the code for the generation of uniform and Gaussian random numbers. All that part of the code uses doubles to represent numbers.

Why did we code it this way? We know that the final result, the average payoff across the simulated paths, depends on the random numbers. Those numbers have non-zero adjoints. However, our goal is to differentiate the final result with respect to the model parameters: spot, local volatilities, and so forth. We know that the random numbers do *not* depend on those parameters. Hence, their adjoints do not propagate all the way back to them.

Formally, with the notations of previous chapters, we differentiate:

$$V_0[(G_i)_{1\leq i\leq N}, a] = \frac{1}{N}\sum_{i=1}^{N} g[h(G_i, a)]$$

with respect to the vector of model parameters $a$. $G_i$ is the vector of Gaussian random numbers for path $i$, $h$ is the path generation function, and $g$ is the payoff function. It is clear from those notations that $G_i$ is another argument to $h$ and has no dependency on $a$. The derivatives we want to compute are:

$$\frac{\partial V_0[(G_i)_{1\le i\le N}, a]}{\partial a} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial g[h(G_i, a)]}{\partial h} \frac{\partial h(G_i, a)}{\partial a}$$

The $G_i$s do not depend depend on $a$. In particular,

$$\frac{\partial G_i}{\partial a} = 0$$

In AAD lingo, we call the $G_i$s *inactive*.

We know that because we know exactly how MC simulations work: this is domain-specific knowledge in action. Because we know that, we don't need to propagate from the random numbers at all. Hence, we don't need to propagate *to* them, either. Their adjoints don't contribute to the final derivatives, so we don't need to compute these adjoints at all. To instantiate the code that produces random numbers with the Number type would result in unnecessary AAD overhead at evaluation time (to put operations and local derivatives on tape) and propagation time (conducting unnecessary propagations to and from the random numbers). Because we know that, we can easily leave the random numbers and their generation outside of the adjoint logic. All it takes is use *doubles* for those calculations in place of a templated number type.

Code that uses the templated number type is said to be *instrumented*. This code is recorded on tape and participates in the adjoint propagation at the cost of AAD overhead. Code that uses native types like doubles is said to be noninstrumented. It is not recorded or propagated and free of AAD overhead. Calculations that depend directly or indirectly on the inputs and, therefore, must participate in adjoint propagation, are called *active*. Code that is independent of inputs and, therefore, does not contribute to their adjoints, is called *inactive*. We have a first and probably most important rule for the efficient differentiation of calculation code:

- *Only instrument active code.*

We call this major optimization *selective instrumentation*. On the other hand, if we forget to instrument active code, we produce wrong derivatives. Active adjoints are not propagated and their contribution is lost. If we instrument inactive code, like random number generators, we produce unnecessary AAD overhead. We must instrument all the active code, but only the active code. The identification of inactive code is a key step of an efficient differentiation.

Readers are encouraged to review our simulation code and verify that we correctly applied selective instrumentation.

## 12.2 SERIAL CODE

We start with the serial code. We have nothing to do about random numbers, and the models and products are already instrumented (although we will need minor extensions on models). Therefore, our work focuses on the template algorithm of Chapter 6, which we recall here:

```
//  MC simulator: free function that conducts simulations
//      and returns a matrix (as vector of vectors) of payoffs
//          (0..nPath-1 , 0..nPay-1)
inline vector<vector<double>> mcSimul(
    const Product<double>&      prd,
    const Model<double>&        mdl,
    const RNG&                  rng,
    const size_t                nPath)
{
    //  Work with copies of the model and RNG
    //      which are modified when we set up the simulation
    //  Copies are OK at high level
    auto cMdl = mdl.clone();
    auto cRng = rng.clone();

    //  Allocate results
    const size_t nPay = prd.payoffLabels().size();
    vector<vector<double>> results(nPath, vector<double>(nPay));
    //  Init the simulation timeline
    cMdl->allocate(prd.timeline(), prd.defline());
    cMdl->init(prd.timeline(), prd.defline());
    //  Init the RNG
    cRng->init(cMdl->simDim());
    //  Allocate Gaussian vector
    vector<double> gaussVec(cMdl->simDim());
    //  Allocate and initialize path
    Scenario<double> path;
    allocatePath(prd.defline(), path);
    initializePath(path);

    //  Iterate over paths
    for (size_t i = 0; i<nPath; i++)
    {
        //  Next Gaussian vector, dimension D
        cRng->nextG(gaussVec);
        //  Generate path, consume Gaussian vector
        cMdl->generatePath(gaussVec, path);
        //  Compute payoffs
        prd.payoffs(path, results[i]);
    }

    return results; //  C++11: move
}
```

We now differentiate this code, instantiating valuation code with the Number type in place of doubles and inserting additional logic for the back-propagation of adjoints and, more generally, the management of the tape. Following the discussion of the previous chapter, we back-propagate adjoints path-wise to avoid unreasonable tape growth.

```cpp
#include "AAD.h"

//  AAD instrumentation of mcSimul()

//  returns the following results:
struct AADSimulResults
{
    AADSimulResults(
        const size_t nPath,
        const size_t nPay,
        const size_t nParam) :
            payoffs(nPath, vector<double>(nPay)),
            aggregated(nPath),
            risks(nParam)
    {}

    //  matrix(0..nPath - 1, 0..nPay - 1) of payoffs
    //      same as mcSimul()
    vector<vector<double>>  payoffs;

    //  vector(0..nPath) of aggregated payoffs
    vector<double>          aggregated;

    //  vector(0..nParam - 1) of risk sensitivities
    //      of aggregated payoff, averaged over paths
    vector<double>          risks;
};

//  Default aggregator = 1st payoff = payoff[0]
const auto defaultAggregator = [](const vector<Number>& v) {return v[0]; };

template<class F = decltype(defaultAggregator)>
inline AADSimulResults
mcSimulAAD(
    const Product<Number>&  prd,
    const Model<Number>&    mdl,
    const RNG& rng,
    const size_t            nPath,
    const F&                aggFun = defaultAggregator)
{
    //  Work with copies of the model and RNG
    //      which are modified when we set up the simulation
    //  Copies are OK at high level
    auto cMdl = mdl.clone();
    auto cRng = rng.clone();

    //  Allocate path and model
    //      do not initialize yet
    Scenario<Number> path;
    allocatePath(prd.defline(), path);
    cMdl->allocate(prd.timeline(), prd.defline());

    //  Dimensions
    const size_t nPay = prd.payoffLabels().size();
    const vector<Number*>& params = cMdl->parameters();
    const size_t nParam = params.size();

    //  ...
```

The function has the same inputs as the original, except they are now instantiated with Numbers instead of doubles and an additional "aggregator." Following the discussion of the previous chapter, we are dif-

ferentiating one result for now. In case the product has multiple payoffs, we differentiate an aggregate provided by client code as a function of the payoffs, represented by a lambda or another callable type. For example, the product may be a portfolio of transactions, and we may want to differentiate the value of the entire book, the sum of the values of the transactions weighted by the notionals. In this case, we would apply an aggregator that computes the "payoff" of the portfolio as the sum of the payoffs of the products, weighted by the notionals:

```
vector<double> notionals;
//  fill notionals
//  ...
const auto portfolioAggregator = [&notionals](const vector<Number>& v)
    {
        return inner_product(
            notionals.begin(),
            notionals.end(),
            v.begin(),
            Number(0.0));
    }
//  call AADSimulResults() with portfolioAggregator() as aggregator
```

where $inner\_product()$ is a standard C++ library algorithm from header <numeric> that does exactly what its names says. Note that it is not the values, but the payoffs that are aggregated. The aggregator effectively turns multiple payoffs into one aggregated payoff and it is this aggregated payoff that is differentiated. The default aggregator simply picks the first payoff.

The return type is different: we return a matrix of path-wise payoffs as previously, but we also return a vector of path-wise aggregated payoffs and the vector of all the sensitivities of its value (path-wise average) to all the model parameters.

Like in valuation, we start by cloning the model and the RNG, modified by the subsequent code. We allocate the path and the model's working memory *but we don't initialize them yet* and pick the dimensions: number of payoffs and number of parameters. We take a refer-

ence on the vector of *pointers to* the model parameters so we easily address it thereafter under the name $params$.

What follows is new and deserves an explanation:

```
1   //  ...
2
3   //  AAD - 1
4
5   //  Access tape
6   Tape& tape = *Number::tape;
7
8   //  Clear and initialize tape
9   tape.clear();
10
11  //  Put parameters on tape
12  //      note this also initializes all adjoints
13  cMdl->putParametersOnTape();
14
15  //  Init the model
16  //  CAREFUL: initialization is recorded on tape
17  //  Model parameters must be on tape prior to initialization
18  cMdl->init(prd.timeline(), prd.defline());
19
20  //  Initialize path
21  initializePath(path);
22
23  //  Mark the tape straight after initialization
24  tape.mark();
25
26  //
27
28  //  ...
```

We take a reference to the tape and initialize it.

The next thing we do (line 13) is put all the model parameters on tape. They are the inputs to our calculation with respect to which we compute derivatives. They must go on tape before any calculation is executed. See the discussion on page 391 for details and a simple example.

This is the modification we need in our simulation code: we must develop a method on the base Model class to put all the parameters on tape. The modification is minor, because we can implement it directly on the base class without modification to concrete models. The (small) difficulty is that the Model class is templated in its number type, but only Numbers go on tape. We must implement some simple template specialization magic so that

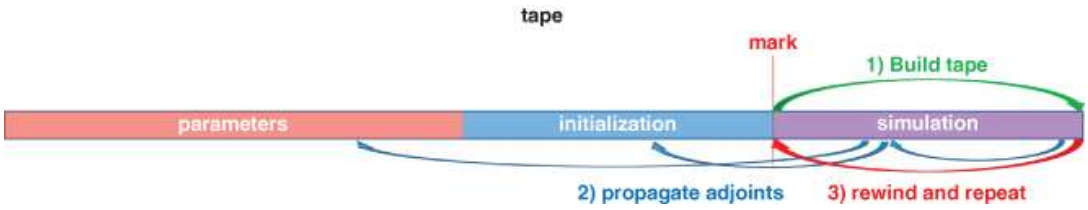
$Model<Number>::putParametersOnTape()$ puts its parameters on tape, whereas for any other type T, $Model<T>::putParametersOnTape()$ does nothing.[1] Readers unfamiliar with template specialization may learn all about it in [76].

The code below completes the base Model class in mcBase.h.

```
    //  Model base class
    //  ...

    //  Put parameters on tape, only valid for T = Number
    void putParametersOnTape()
    {
        putParametersOnTapeT<T>();
    }

private:

    //  Template specialization here

    //  If T is not Number : do nothing
    template<class U>
    void putParametersOnTapeT()
    {

    }

    //  If T is Number : put all parameters on tape
    template <>
    void putParametersOnTapeT<Number>()
    {
        for (Number* param : parameters()) param->putOnTape();
    }
```

Back to our simulation code, the next thing we do (lines 18 and 21) is initialize the model and the scenario. It is crucial to do this here and not before. Initialization must be conducted *after* the parameters are on tape. The initialization of a model typically conducts calculations with the model parameters. In Black and Scholes, we pre-calculate deterministic amounts for the subsequent generation of the paths. In Dupire, we pre-interpolate the local volatilities in time. So the parameters must be on tape, or these calculations result in nodes with dangling pointers to nonexisting parameter adjoints. We again refer to the detailed discussion on page 391.

At this point the parameters, and the initial calculations, those that are not repeated path-wise, are all on tape and correctly linked. We *mark* the tape there (line 24). Remember, we are going to compute path-wise derivatives. We will construct the tape for a path, propagate adjoints through it, and *rewind* it for the next path. We are going to rewind it *up to the mark* so that the inputs and the initialization remain on tape throughout the whole simulation, their adjoints accumulating contributions across simulations.




Next in our simulation code, we initialize the RNG, preallocate working memory for the simulations, and allocate results. This is admin code; it is not recorded and no different than valuation.

```
1     //  ...
2
3     //  Init the RNG
4     cRng->init(cMdl->simDim());
5
6     //  Allocate workspace
7     vector<Number> nPayoffs(nPay);
8     //  Gaussian vector
9     vector<double> gaussVec(cMdl->simDim());
10
11    //  Results
12    AADSimulResults results(nPath, nPay, nParam);
13
14    //  ...
```

The important initialization phase is complete; we now proceed with the simulations:

```
1     //  ...
2
3     //  Iterate through paths
4     for (size_t i = 0; i<nPath; i++)
5     {
6         //  AAD - 2
7         //  Rewind tape to mark
8         //      parameters and initializers stay on tape
9         //      but the simulation is overwritten
10        tape.rewindToMark();
11        //
12
13        //  Next Gaussian vector, dimension D
14        cRng->nextG(gaussVec);
15        //  Generate path, consume Gaussian vector
16        cMdl->generatePath(gaussVec, path);
17        //  Compute payofffs
18        prd.payoffs(path, nPayoffs);
19        //  Aggregate
20        Number result = aggFun(nPayoffs);
21
22        //  AAD - 3
23        //  Propagate adjoints
24        result.propagateToMark();
25        //  Store results for the path
26        results.aggregated[i] = double(result);
27        convertCollection(
28            nPayoffs.begin(),
29            nPayoffs.end(),
30            results.payoffs[i].begin());
31        //
32    }
33
34    //  ...
```

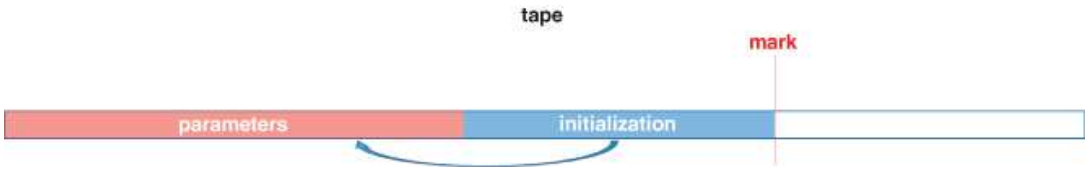
A path is processed in three stages:

1. Rewind the tape *to the mark* (line 10) so as to reuse the simulation piece of the tape across simulations and avoid growth while leaving the model parameters and initialization on tape throughout all simulations so that their adjoints correctly accumulate contributions from all the simulations.
2. Generate and evaluate the path (lines 13–20). This code is identical to the valuation code. It executes the three steps of the processing of a path: generate Gaussian numbers, simulate a path, evaluate payoffs over the path. The difference is that this code is now instantiated with Numbers. So it goes on tape when executed and builds the simulation piece of the tape, overwriting the previous simulation so that the size of the tape remains fixed to the size of one simulation; see figure on page 414.

We have an additional line 20, where the payoffs for the current path are aggregated in one result, with an aggregator provided by the client code, as explained earlier.

3. Propagate adjoints from the result (which adjoint is seeded to 1) *to the mark*. We back-propagate the adjoints of the current simulation, but only those. As a result, the adjoints on the persistent piece of tape, where the parameters and initialization were recorded, accumulate their contribution to the current simulation.

After the simulation loop completes, the adjoints on the persistent piece of tape accumulated their contribution to all the simulations. What is left is back-propagate adjoints through the persistent piece so that the adjoints of the entire Monte-Carlo valuation accumulate into the model parameters.




This takes *one* line of code:

```
1     //  ...
2
3     //  AAD - 4
4     //  Mark = limit between pre-calculations and path-wise operations
5     //  Operations above mark have been propagated and accumulated
6     //  We conduct one propagation mark to start
7     Number::propagateMarkToStart();
8     //
9
10    //  ...
```

It follows that we back-propagate simulation adjoints repeatedly, but initialization adjoints only once. This is another one of the strongest optimizations in our code. If we propagated adjoints all the way, repeatedly over each simulation, to differentiate a Monte-Carlo valuation would be *many times* slower. This optimization, to a large extent, is the dual of the strongest optimization we included in our simulation

code of Chapter 6. We gathered as much work as possible into the initialization, conducted once, so as to minimize work during the repeated simulations. It follows that we have large a initialization tape and small simulation tapes. In the differentiation algorithm, we back-propagate repeatedly over the small tapes and once over the large tape, reaping the benefits of our optimization of Chapter 6 for differentiation, too.

All that remains is pick sensitivities to parameters on the parameters adjoints, pack,[2] clean, and return:

```
    //  ...

    //  Pick sensitivities, summed over paths, and normalize
    transform(
        params.begin(),
        params.end(),
        results.risks.begin(),
        [nPath](const Number* p) {return p->adjoint() / nPath; });

    //  Clear the tape
    tape.clear();

    return results;
}
```

The code is found in mcBase.h.

## 12.3 USER INTERFACE

We must upgrade our user interface in main.h to run the differentiation algorithm. We develop two functions: $AADriskOne()$, which differentiates one selected payoff in a product, and $AADriskAggregate()$, which differentiates a portfolio of payoffs with given notionals. Both functions take a model and a product in the memory store, developed in Section 6.5, run the differentiation algorithm, and return:

1. The labels of the payoffs in a vector<string>
2. The corresponding values in a vector<double>

3. The value of the differentiated aggregate, as a double
4. The names of all the parameters in a vector<string>
5. The corresponding risk sensitivities of the aggregate to parameters in a vector<double>

```
    struct
    {
        vector<string>  payoffIds;
        vector<double>  payoffValues;
        double          riskPayoffValue;
        vector<string>  paramIds;
        vector<double>  risks;
    } results;
```

The differentiation algorithm works with instrumented models and products, Model <Number> and Product <Number>. We upgrade the memory store in store.h so we can store and retrieve both instrumented and non-instrumented models and products. In practice, we store a pair of models or products, one instrumented and one not:

```
#include "mcBase.h"
#include "mcMdl.h"
#include "mcPrd.h"
#include <unordered_map>
#include <memory>
using namespace std;

using ModelStore =
unordered_map<string,
    pair<unique_ptr<Model<double>>, unique_ptr<Model<Number>>>>;
using ProductStore =
unordered_map<string,
    pair<unique_ptr<Product<double>>, unique_ptr<Product<Number>>>>;

ModelStore modelStore;
ProductStore productStore;
```

We upgrade the functions to initialize models and products in memory accordingly:

```
void putBlackScholes(
    const double            spot,
    const double            vol,
    const bool              qSpot,
    const double            rate,
    const double            div,
    const string&           store)
{
    //  We create 2 models, one for valuation and one for risk
    unique_ptr<Model<double>> mdl = make_unique<BlackScholes<double>>(
        spot, vol, qSpot, rate, div);
    unique_ptr<Model<Number>> riskMdl = make_unique<BlackScholes<Number>>(
        spot, vol, qSpot, rate, div);

    //  And move them into the map
    modelStore[store] = make_pair(move(mdl), move(riskMdl));
}

//  Same pattern for putDupire

void putEuropean(
    const double            strike,
    const Time              exerciseDate,
    const Time              settlementDate,
    const string&           store)
{
    //  We create 2 products, one for valuation and one for risk
    unique_ptr<Product<double>> prd = make_unique<European<double>>(
        strike, exerciseDate, settlementDate);
    unique_ptr<Product<Number>> riskPrd = make_unique<European<Number>>(
        strike, exerciseDate, settlementDate);

    //  And move them into the map
    productStore[store] = make_pair(move(prd), move(riskPrd));
}

//  Same pattern for other products
```

We template the functions to retrieve models and products, so client codes calls $getModel < double > ()$ for pricing or $getModel < Number > ()$ for risk:

```
template<class T>
const Model<T>* getModel(const string& store);

template<>
const Model<double>* getModel(const string& store)
{
    auto it = modelStore.find(store);
    if (it == modelStore.end()) return nullptr;
    else return it->second.first.get();
}

template<>
const Model<Number>* getModel(const string& store)
{
    auto it = modelStore.find(store);
    if (it == modelStore.end()) return nullptr;
    else return it->second.second.get();
}

//  Same pattern for products
```

The entire code is found in store.h. With the store upgraded to deal with instrumented objects, we develop the user interface for the differentiation of simulations in main.h below. We write two functions, one for the differentiation of one named payoff, and one for the differentiation of a portfolio. The project in our repository exports these functions to Excel.

```
//  Generic risk
inline auto AADriskOne(
    const string&           modelId,
    const string&           productId,
    const NumericalParam&   num,
    const string&           riskPayoff = "")
{
    //  Get instrumented model and product
    const Model<Number>* model = getModel<Number>(modelId);
    const Product<Number>* product = getProduct<Number> (productId);

    if (!model || !product)
    {
        throw runtime_error(
            "AADrisk() : Could not retrieve model and product");
    }

    //  Random Number Generator
    unique_ptr<RNG> rng;
    if (num.useSobol) rng = make_unique<Sobol>();
    else rng = make_unique<mrg32k3a>(num.seed1, num.seed2);

    //  Find the payoff to differentiate
    size_t riskPayoffIdx = 0;
    if (!riskPayoff.empty())
    {
        const vector<string>& allPayoffs = product->payoffLabels();
        auto it = find(allPayoffs.begin(), allPayoffs.end(), riskPayoff);
        if (it == allPayoffs.end())
        {
            throw runtime_error("AADriskOne() : payoff not found");
        }
        riskPayoffIdx = distance(allPayoffs.begin(), it);
    }

    //  Simulate
    const auto simulResults = num.parallel
        ? mcParallelSimulAAD(*product, *model, *rng, num.numPath,
            [riskPayoffIdx](const vector<Number>& v)
            {return v[riskPayoffIdx]; })
        : mcSimulAAD(*product, *model, *rng, num.numPath,
            [riskPayoffIdx](const vector<Number>& v)
            {return v[riskPayoffIdx]; });

    //  We return: a number and 2 vectors :
    //  -   The payoff identifiers and their values
    //  -   The value of the aggreagte payoff
    //  -   The parameter idenitifiers
    //  -   The sensititivities of the aggregate to parameters
    struct
    {
        vector<string>  payoffIds;
        vector<double>  payoffValues;
        double          riskPayoffValue;
        vector<string>  paramIds;
        vector<double>  risks;
    } results;

```

```
    const size_t nPayoffs = product->payoffLabels().size();
    results.payoffIds = product->payoffLabels();
    results.payoffValues.resize(nPayoffs);
    for (size_t i = 0; i < nPayoffs; ++i)
    {
        results.payoffValues[i] = accumulate(
            simulResults.payoffs.begin(),
            simulResults.payoffs.end(),
            0.0,
            [i](const double acc, const vector<double>& v)
                { return acc + v[i]; }
        ) / num.numPath;
    }
    results.riskPayoffValue = accumulate(
        simulResults.aggregated.begin(),
        simulResults.aggregated.end(),
        0.0) / num.numPath;
    results.paramIds = model->parameterLabels();
    results.risks = move (simulResults.risks);

    return results;
}

inline auto AADriskAggregate(
    const string&           modelId,
    const string&           productId,
    const map<string, double>&   notionals,
    const NumericalParam&   num)
{
    //  Get instrumented model and product
    const Model<Number>* model = getModel<Number>(modelId);
    const Product<Number>* product = getProduct<Number> (productId);

    if (!model || !product)
    {
        throw runtime_error(
        "AADriskAggregate() : Could not retrieve model and product");
    }

    //  Random Number Generator
    unique_ptr<RNG> rng;
    if (num.useSobol) rng = make_unique<Sobol>();
    else rng = make_unique<mrg32k3a>(num.seed1, num.seed2);

    //  Vector of notionals
    const vector<string>& allPayoffs = product->payoffLabels();
    vector<double> vnots(allPayoffs.size(), 0.0);
    for (const auto& notional : notionals)
    {
        auto it = find(allPayoffs.begin(), allPayoffs.end(), notional.first);
        if (it == allPayoffs.end())
        {
            throw runtime_error("AADriskAggregate() : payoff not found");
        }
        vnots[distance(allPayoffs.begin(), it)] = notional.second;
    }

```

```
    //  Aggregator
    auto aggregator = [&vnots](const vector<Number>& payoffs)
    {
        return inner_product(
            payoffs.begin(),
            payoffs.end(),
            vnots.begin(),
            Number(0.0));
    };

    //  Simulate
    const auto simulResults = num.parallel
        ? mcParallelSimulAAD(*product, *model, *rng, num.numPath, aggregator)
        : mcSimulAAD(*product, *model, *rng, num.numPath, aggregator);

    //  We return: a number and 2 vectors :
    //  -   The payoff identifiers and their values
    //  -   The value of the aggreagte payoff
    //  -   The parameter idenitifiers
    //  -   The sensititivities of the aggregate to parameters
    struct
    {
        vector<string>  payoffIds;
        vector<double>  payoffValues;
        double          riskPayoffValue;
        vector<string>  paramIds;
        vector<double>  risks;
    } results;

    const size_t nPayoffs = product->payoffLabels().size();
    results.payoffIds = product->payoffLabels();
    results.payoffValues.resize(nPayoffs);
    for (size_t i = 0; i < nPayoffs; ++i)
    {
        results.payoffValues[i] = accumulate(
            simulResults.payoffs.begin(),
            simulResults.payoffs.end(),
            0.0,
            [i](const double acc, const vector<double>& v)
                { return acc + v[i]; }
        ) / num.numPath;
    }
    results.riskPayoffValue = accumulate(
        simulResults.aggregated.begin(),
        simulResults.aggregated.end(),
        0.0) / num.numPath;
    results.paramIds = model->parameterLabels();
    results.risks = move(simulResults.risks);

    return results;
}
```

where $mcParallelSimulAAD()$ , the parallel version of the differentiation algorithm, is developed next.

For test purposes, and to assess the speed and the accuracy of our derivatives, we also implement bump risk. Bumps are implemented in a trivial manner to produce itemized risk reports[3] in a trivial manner. Bump risk is up to thousands of times slower than AAD, and the result-

ing risk sensitivities are expected to be virtually identical. The following self–explanatory function is found in main.h:

```
//  Returns a vector of values and a matrix of risks
//      with payoffs in columns and parameters in rows
//      along with ids of payoffs and parameters

struct RiskReports
{
    vector<string> payoffs;
    vector<string> params;
    vector<double> values;
    matrix<double> risks;
};

inline RiskReports bumpRisk(
    const string&           modelId,
    const string&           productId,
    const NumericalParam&   num)
{
    auto* orig = getModel<double>(modelId);
    const Product<double>* product = getProduct<double> (productId);

    if (!orig || !product)
    {
        throw runtime_error(
            "bumpRisk() : Could not retrieve model and product");
    }

    RiskReports results;

    //  Base values
    auto baseRes = value(*orig, *product, num);
    results.payoffs = baseRes.identifiers;
    results.values = baseRes.values;

    //  Make copy so we don't modify the model in memory
    auto model = orig->clone();

    results.params = model->parameterLabels();
    const vector<double*> parameters = model->parameters();
    const size_t n = parameters.size(), m = results.payoffs.size();
    results.risks.resize(n, m);

    //  Bump parameters one by one
    for (size_t i = 0; i < n; ++i)
    {
        //  Bump
        *parameters[i] += 1.e-08;
        //  Reval
        auto bumpRes = value(*model, *product, num);
        //  Unbump
        *parameters[i] -= 1.e-08;

        // Compute finite differences for all payoffs
        for (size_t j = 0; j < m; ++j)
        {
            results.risks[i][j] = 1.0e+08 *
                (bumpRes.values[j] - baseRes.values[j]);
        }
    }

    return results;
}
```

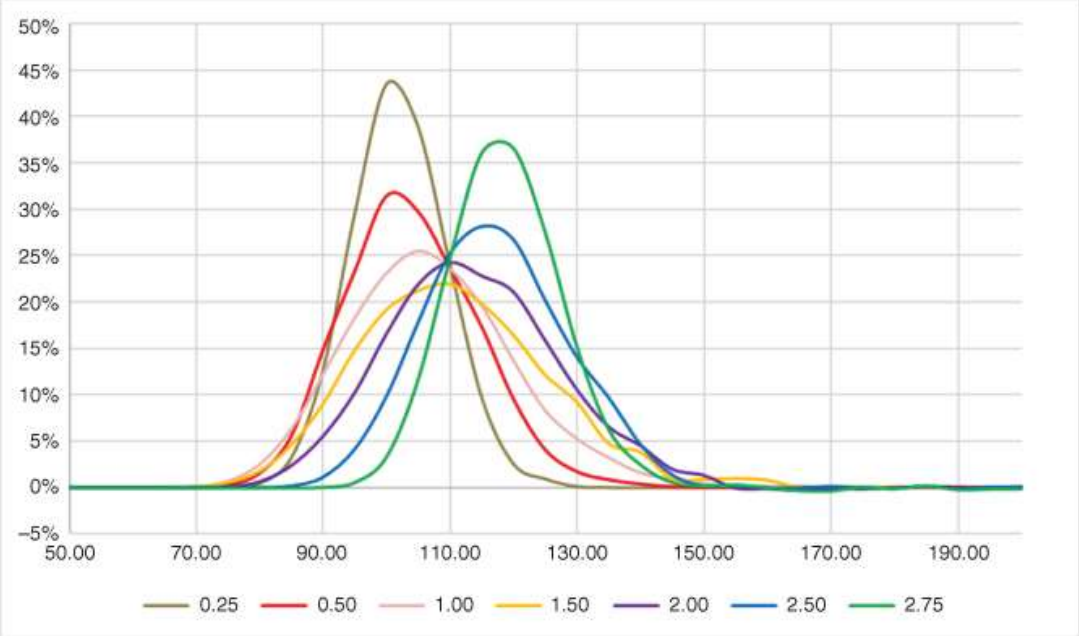
## 12.4 SERIAL RESULTS

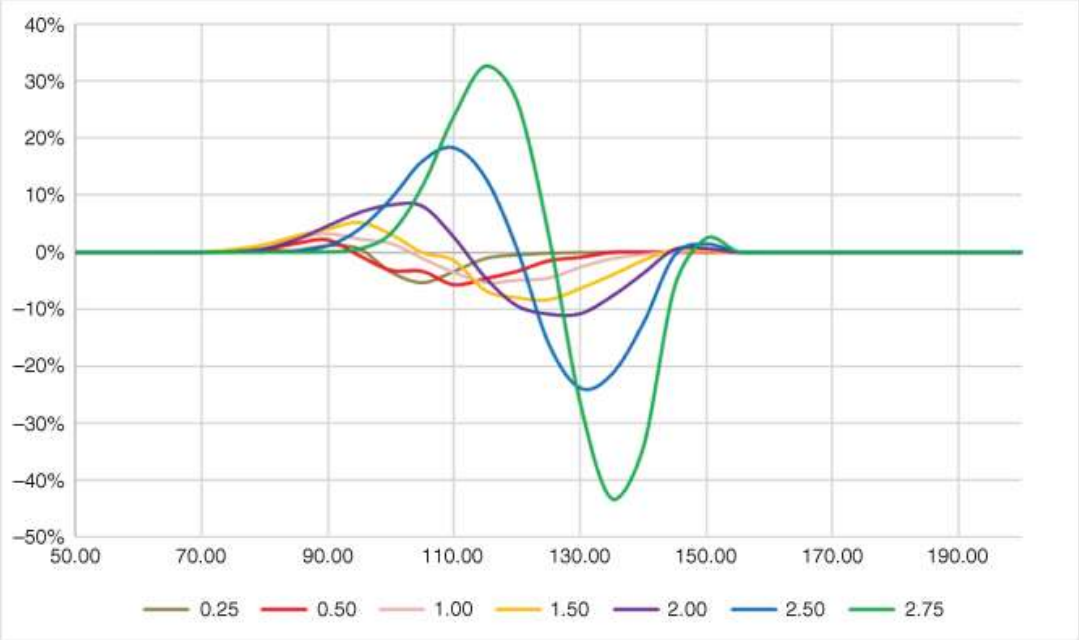
We can now differentiate the results of our simulation library with respect to model parameters. For instance, we compute the sensitivities of a European or barrier option to all local volatilities. Sensitivities to local volatilities, what Dupire calls a *microbucket*, is valuable information for research purpose. For the purpose of trading, however, we really need the sensitivities to tradable market variables, not model parameters. In the context of Dupire's model, we want the sensitivities to the *market-implied* volatilities, what Dupire calls a *superbucket*, because those are risks that may be directly hedged by trading European options in the market. Local volatilities are calibrated to implied volatilities, so superbuckets can be computed from microbuckets, as discussed in detail in Chapter 13. In this chapter, we focus on the production of microbuckets.

We differentiate the example of Section 6.6: a 3y European call of strike 120, together with a 150 barrier, in Dupire's model with a local volatility of 15% over 60 times and 30 spots, simulating 500,000 paths over 156 weekly steps, where a pricing took around 3 seconds with Sobol. With 1,801 sensitivities to compute, we need 1,802 prices with finite differences, so the bump risk takes around 1.5 hours.[4]

Our AAD implementation produces the entire risk in around 8 seconds (8,250 ms), less than three times a single valuation. With the improvements of Chapter 15, when we further accelerate AAD by factor around 2, we produce this report in less than 5 seconds (4,500 ms). With the parallel algorithm, it takes *half a second* (575 ms).

The results are displayed on the chart below, first for the European, then for the barrier, with a smoothing of 5. The charts display the differentials to local volatilities on seven different times, as functions of the spot.




The speed is remarkable and may look suspect. Didn't we establish the theoretical maximum speed for AAD at three times a valuation? With the improvements of Chapter 15, our example computes in the time of 1.5 valuations, *half the theoretical minimum*.

This is mainly due to our selective code instrumentation. A lot of the valuation time is spent generating random numbers and turning them into Gaussians with the not-so-cheap inverse Gaussian, executed $ND = 78M$ times. All those calculations are inactive. We did not instrument them. They execute without ADD overhead, in the exact

same time in differentiation or evaluation mode. Only the instrumented code bears AAD overhead. The remarkable speed is not only due to AAD's constant time, and the implementation of the AAD library, but also to the optimizations in our instrumented code, mainly the selective instrumentation and the partial back-propagation.

Risk sensitivities are almost identical to bumping. Sensitivities to local volatilities match to many significant figures. Difference in delta is around $2.5e-05$. The reason is subtle. In the barrier payoff code, we used a smoothing factor of $x\%$ of the current spot. When the spot is bumped, the smoothing factor changes and affects the price. But this is numerical noise, not market risk. Hence, we made the smoothing factor a *double* in our code so that it does not contribute to derivatives, and the result we get is market risk alone. Our delta is not off; on the contrary, it is more accurate.

## 12.5 PARALLEL CODE

We now move on to the AAD instrumentation of our *parallel* code from Chapter 7, combining the idioms of that section for parallelism with the techniques of this chapter for differentiation. The instrumentation of the parallel code is more complicated, but well worth the effort, since it is really the combination of parallel simulations and AAD (and expression templates, see Chapter 15) that gives us this “magic” performance. The AAD library helps. The static tape pointer on the Number type is thread local, so all the calculations executed on a thread land on that thread's own tape. But it doesn't mean our work is done. First, the tape *pointer* is thread local, but we still must declare all the tapes for all the threads and set the thread local pointer to each thread's tape *from that thread*. More importantly:

1. With AAD, the model is a *mutable* object. Even though $Model::generatePath()$ is marked const, it is not really the case any more; see the discussion on page 391. When the Numbers stored on the model (parameters and initialized amounts) are *read*,

their adjoints on tape are *written* into during back-propagation. So, we must make copies of the model for every thread like other mutable objects.[5]

2. In the single-thread version, we were careful to put the model parameters on tape and initialize the model so the initialized amounts are recorded, too. In the parallel version, we must put the parameters and initializers, not on one tape, but on every thread's tape. The only way to do this is call $Model::putParametersOnTape()$ and $Model::init()$ from *each thread* before the first simulation on that thread.
3. Adjoints are accumulated on thread local tapes, so after the simulation loop, the adjoints of the parameters and initializers are scattered across different tapes. We must back-propagate over initialization on multiple tapes and aggregate all the sensitivities to parameters.

The code is essentially a combination of the parallel code of Section 7 with the differentiation code of this chapter, with a special response to these specific challenges.

To address number 2, we separate the initialization in a dedicated function:

```
//  Put parameters on tape and init model
inline void initModel4ParallelAAD(
    //  Inputs
    const Product<Number>&      prd,
    //  Cloned model, must have been allocated prior
    Model<Number>&              clonedMdl,
    //  Path, also allocated prior
    Scenario<Number>&           path)
{
    //  Access tape
    Tape& tape = *Number::tape;

    //  Rewind tape
    tape.rewind();

    //  Put parameters on tape
    //      note this also initializes all adjoints
    clonedMdl.putParametersOnTape();

    //  Init the model
    //      not before the parameters are on tape:
    //      initialization uses parameters as arguments
    clonedMdl.init(prd.timeline(), prd.defline());

    //  Path
    initializePath(path);

    //  Mark the tape straight after parameters
    tape.mark();
    //
}
```

The template algorithm has the same signature as the single-threaded one and returns results in the same format:

```
#include "ThreadPool.h"

//  Parallel version of mcSimulAAD()
template<class F = decltype(defaultAggregator)>
inline AADSimulResults
mcParallelSimulAAD(
    const Product<Number>&  prd,
    const Model<Number>&    mdl,
    const RNG&              rng,
    const size_t            nPath,
    const F&                aggFun = defaultAggregator)
{
    const size_t nPay = prd.payoffLabels().size();
    const size_t nParam = mdl.numParams();

    //  Allocate results
    AADSimulResults results(nPath, nPay, nParam);

    //  Clear and initialise tape
    Number::tape->clear();
//  ...
```

and the entry-level function in main.h is already set up to fire it.

Next, we allocate working memory (without initializing anything quite yet), one mutable object for every thread (including the main thread, so number of worker threads + 1):

```
1   //  ...
2
3   //  We need one of all these for each thread
4   //  0: main thread
5   //  1 to n : worker threads
6
7   ThreadPool *pool = ThreadPool::getInstance();
8   const size_t nThread = pool->numThreads();
9
10  //  Allocate workspace, do not initialize anything yet
11
12  //  One model clone per thread
13  //      models are mutable with AAD
14  vector<unique_ptr<Model<Number>>> models(nThread + 1);
15  for (auto& model : models)
16  {
17      model = mdl.clone();
18      model->allocate(prd.timeline(), prd.defline());
19  }
20
21  //  One scenario per thread
22  vector<Scenario<Number>> paths(nThread + 1);
23  for (auto& path : paths)
24  {
25      allocatePath(prd.defline(), path);
26  }
27
28  //  One vector of payoffs per thread
29  vector<vector<Number>> payoffs(nThread + 1, vector<Number>(nPay));
30
31  //  ~workspace
32
33  //  Tapes for the worker threads
34  //  The main thread has one of its own
35  vector<Tape> tapes(nThread);
36
37  //  Model initialized on this thread?
38  vector<int> mdlInit(nThread + 1, false);
39
40  //  ...
```

We also initialized a vector of tapes on line 35, one for each *worker* thread. The main thread already owns a tape: the static tape declared on the Number type instantiates of copy for the main thread when the application starts, as for any static object.

Finally, we have a vector of booleans on line 38, one for every thread, tracking what threads completed the initialization phase, starting with false for everyone.[6]

We must also initialize the RNGs and preallocate the Gaussian vector, but for that we need the simulation dimension $D$, and only an initialized model can provide this information. So the next step is initialize the main thread's model (the main thread always has index 0, the worker threads indices being 1 to nThread):

```
    //  ...

    //  Initialize main thread
    initModel4ParallelAAD(prd, *models[0], paths[0]);

    //  Mark main thread as initialized
    mdlInit[0] = true;

    //  Init the RNGs, one per thread
    //  One RNG per thread
    vector<unique_ptr<RNG>> rngs(nThread + 1);
    for (auto& random : rngs)
    {
        random = rng.clone();
        random->init(models[0]->simDim());
    }

    //  One Gaussian vector per thread
    vector<vector<double>> gaussVecs
        (nThread + 1, vector<double>(models[0]->simDim()));

    //  ...
```

Next, we dispatch the simulation loop for parallel processing to the thread pool, applying the same techniques as in Chapter 7:

```
    //  ...

    //  Reserve memory for futures
    vector<TaskHandle> futures;
    futures.reserve(nPath / BATCHSIZE + 1);

    //  Start

    size_t firstPath = 0;
    size_t pathsLeft = nPath;
    while (pathsLeft > 0)
    {
        size_t pathsInTask = min<size_t>(pathsLeft, BATCHSIZE);

        futures.push_back(pool->spawnTask([&, firstPath, pathsInTask]()
        {

    //  ...
```

The body of the lambda executes on different threads. This is the concurrent code. Here, the tape is the executing thread's tape and $pool->threadNum()$ returns the index of the executing thread. The first thing we do is note this index, and set the thread local tape accordingly:

```
1     //  ...
2
3             const size_t threadNum = pool->threadNum();
4
5             //  Use this thread's tape
6             //  Thread local magic: each thread has its own pointer
7             //  Note main thread = 0 is not reset
8             if (threadNum > 0) Number::tape = &tapes[threadNum - 1];
9
10    //  ...
```

This is the core of parallel AAD right here. The tape is accessed through the *thread local* pointer $Number::tape$. We set it *from the executing thread* to the tape we allocated for that thread in our vector of tapes. From there on, all the subsequent calculations coded in the lambda and executed on this thread are recorded on that tape. The main thread (0) already correctly refers to its global tape. It does not need resetting.

Only at this stage can we put the parameters on tape and initialize the model (other than allocation, we allocated all models earlier, on the main thread, knowing from Chapter 3 that we must avoid concurrent allocations):

```
// ...

        // Initialize once on each thread
        if (!mdlInit[threadNum])
        {
            // Initialize and put on the
            //      executing thread's tape
            initModel4ParallelAAD(
                prd,
                *models[threadNum],
                paths[threadNum]);

            // Mark as initialized
            mdlInit[threadNum] = true;
        }

// ...
```

Next, we pick this thread's RNG and skip it ahead, like in Chapter 7:

```
// ...

        // Get a RNG and position it correctly
        auto& random = rngs[threadNum];
        random->skipTo(firstPath);

// ...
```

Finally, the rest of the concurrent code executes the inner batch loop of Chapter 7 with the same contents as the serial differentiation code: rewind tape (to mark), generate random numbers, simulate scenario, compute payoffs, aggregate them, back-propagate adjoints (to mark), store results, repeat):

```
    //  ...

            //  And conduct the simulations, exactly same as sequential
            for (size_t i = 0; i < pathsInTask; i++)
            {
                //  Rewind tape to mark
                //  Notice : this is the tape for the executing thread

                Number::tape->rewindToMark();
                //  Next Gaussian vector, dimension D
                random->nextG(gaussVecs[threadNum]);
                //  Path
                models[threadNum]->generatePath(
                    gaussVecs[threadNum],
                    paths[threadNum]);
                //  Payoffs
                prd.payoffs(paths[threadNum], payoffs[threadNum]);

                //  Propagate adjoints
                Number result = aggFun(payoffs[threadNum]);
                result.propagateToMark();
                //  Store results for the path
                results.aggregated[firstPath + i] = double(result);
                convertCollection(
                    payoffs[threadNum].begin(),
                    payoffs[threadNum].end(),
                    results.payoffs[firstPath + i].begin());
            }

            //  Remember tasks must return a bool
            return true;
        }));

        pathsLeft -= pathsInTask;
        firstPath += pathsInTask;
    }

    //  Wait and help
    for (auto& future : futures) pool->activeWait(future);

    //  ...
```

Like for serial differentiation, all adjoints are accumulated on the persistent piece of the tape at this stage, and we must propagate the initializers' adjoints back to the parameters' adjoints; see figure on page 416. The difference is that the adjoints are now spread across the multiple tapes belonging to multiple threads, but we are out of the concurrent code and back on the main thread. Never mind; this is one propagation conducted one time; we propagate all the threads' tapes from the main thread:

```
1   //  ...
2
3   //  Mark = limit between pre-calculations and path-wise operations
4   //  Operations above mark have been propagated and accumulated
5   //  We conduct one propagation mark to start
6   //  On the main thread's tape
7   Number::propagateMarkToStart();
8   //  And on the worker thread's tapes
9   Tape* mainThreadPtr = Number::tape;
10  for (size_t i = 0; i < nThread; ++i)
11  {
12      if (mdlInit[i + 1])
13      {
14          //  Set tape pointer
15          Number::tape = &tapes[i];
16          //  Propagate on that tape
17          Number::propagateMarkToStart();
18      }
19  }
20  //  Reset tape to main thread's
21  Number::tape = mainThreadPtr;
22
23  //  ...
```

We first propagate over the main thread's tape on line 7. Then, we sequentially propagate the other $n$ tapes, from the main thread. In order to propagate the tape number $i$ from the main thread, we temporarily set the main thread's tape as the tape $i$ (line 15), and propagate (line 17).

In the end, we reset the main thread's tape to its static tape on line 21.[7]

We are almost home safe. All the adjoints are correctly accumulated, but still across multiple tapes and multiple models. The tape of each thread contains the sum of the adjoints over the paths processed on that thread. The adjoints are accessible for every thread's copy of the model. We sum-up the adjoints across models, hence across tapes, normalize, pack, and return:

```
    //  ...

    //  Sum sensitivities over threads
    for (size_t j = 0; j < nParam; ++j)
    {
        results.risks[j] = 0.0;
        for (size_t i = 0; i < models.size(); ++i)
        {
            if (mdlInit[i]) results.risks[j] +=
                    models[i]->parameters()[j]->adjoint();
        }
        results.risks[j] /= nPath;
    }

    //  Clear the main thread's tape
    //  The other tapes are cleared when the vector of tapes exists scope
    Number::tape->clear();

    return results;
}
```

The code is found in mcBase.h.

## 12.6 PARALLEL RESULTS

We reproduce the example on page 424, this time in parallel: a 3y call 120 with a 150 barrier, in Dupire's model 1,800 local volatilities (of which 1,080 active) of 15% over 60 times and 30 spots, simulating 500,000 paths over 156 weekly steps with Sobol. With the serial implementation, it took around 8 seconds to compute the microbuckets (precisely 8,250 ms). With the improvements of Chapter 15, it takes less than 5 seconds (precisely 4,500 ms).

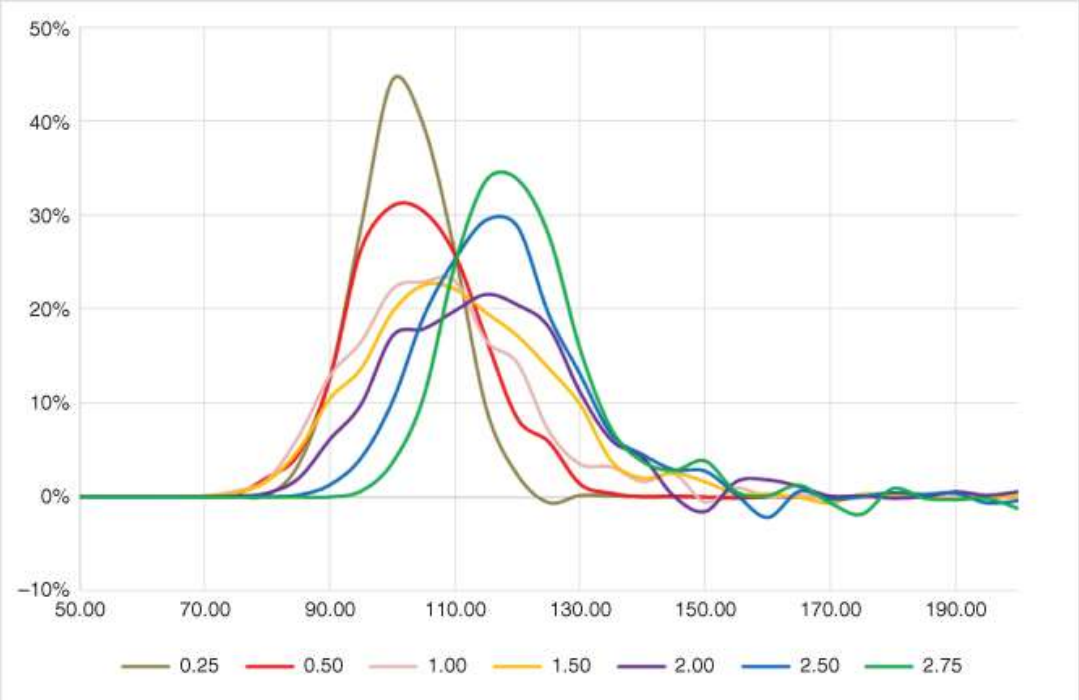
With the parallel implementation, we get the exact same results in exactly one second, with perfect parallel efficiency. With the improvements of Chapter 15, we get them in half a second (precisely 575 ms).

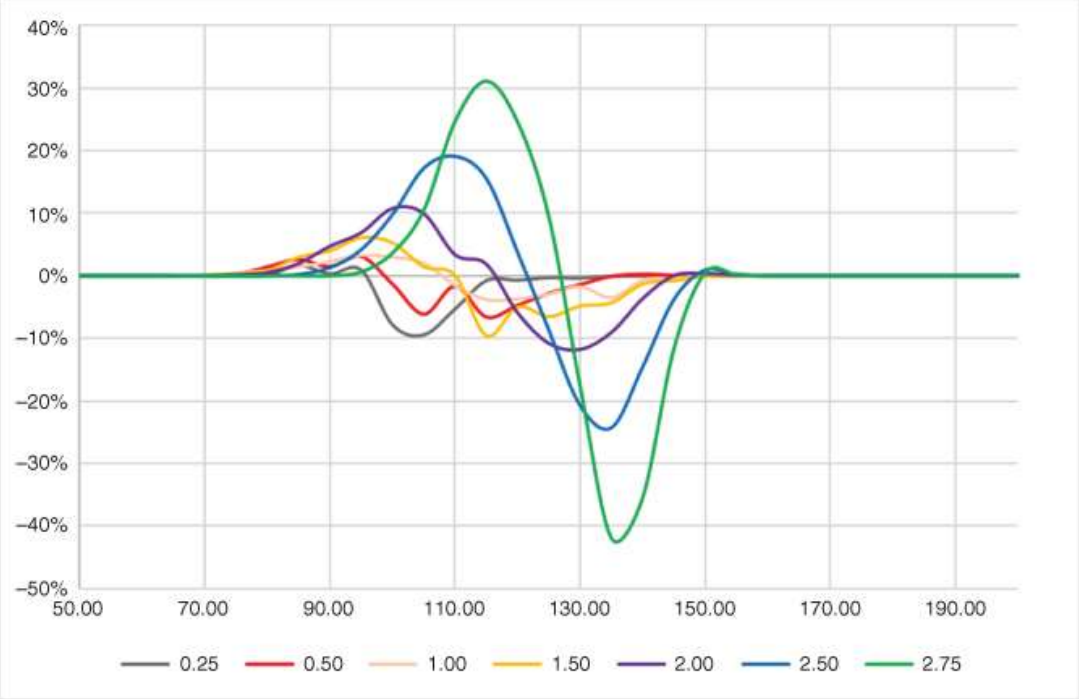
*We produce more than 1,000 active sensitivities over 500,000 paths, 156 steps, in half a second.*

With an efficient AAD library, coupled with an effective instrumentation of the simulation code, we computed derivatives over simulations with unthinkable speed. Not only can we produce lightning-fast risk

reports on workstations and laptops, including for xVA, we can also conduct research that would have been impossible otherwise.

For instance, a parallel price with Sobol takes 350 ms in our example, that is, more than five minutes for a finite difference risk over the 1,080 active local volatilities (which perhaps may be improved to two to three minutes with substantial effort and invasive code). This is too slow. It is therefore common practice to reduce accuracy to produce risk reports in a manageable time. With 36 monthly time steps and 100,000 paths, a parallel price takes 20 ms, so a finite difference risk report could perhaps be produced in as little as ten seconds. The results, however, are not pretty, as seen in the charts below, first for the European, then for the barrier, to be compared to the charts on page.

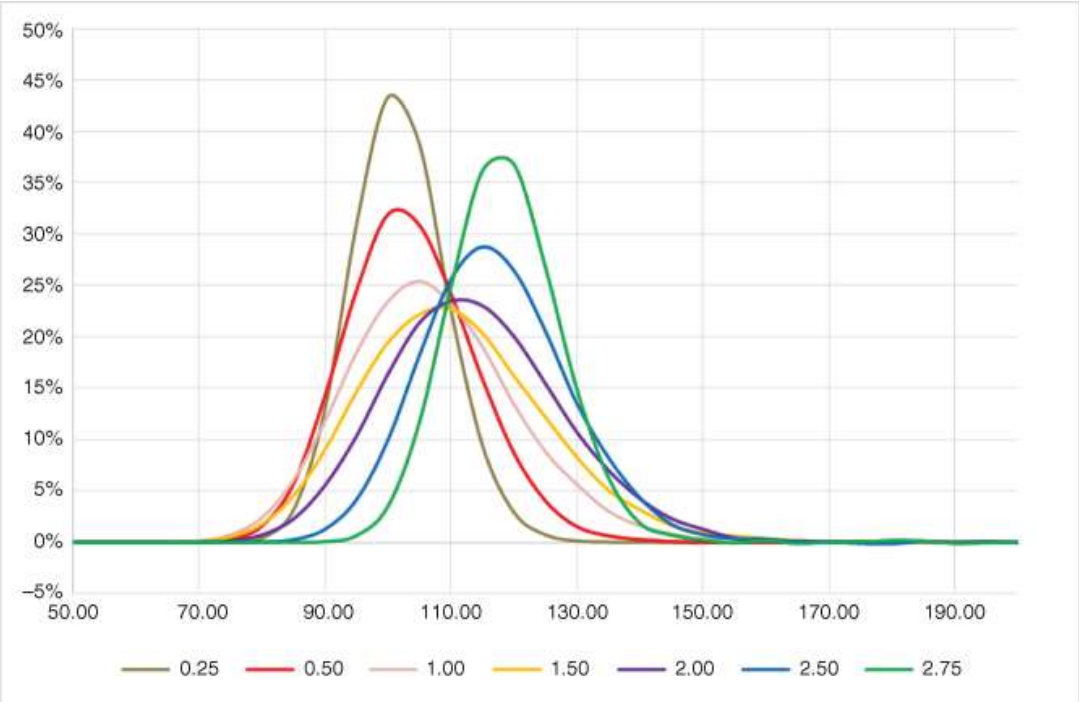


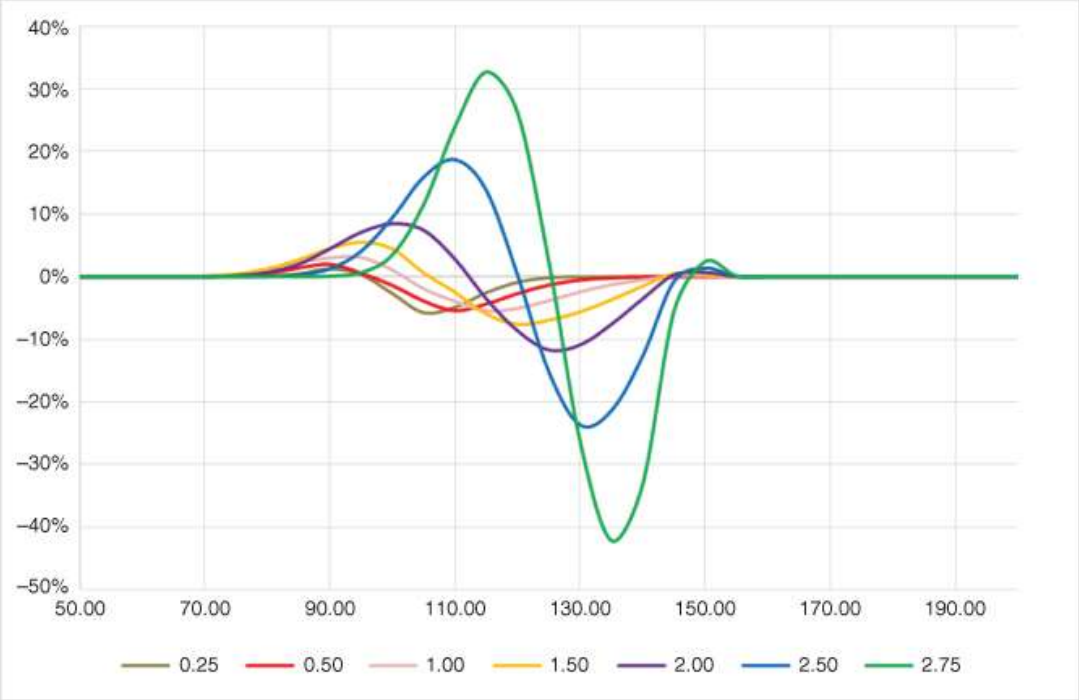



Slow financial software has the unfortunate consequence that users are encouraged to settle for approximate, inaccurate risk computed in reasonable time. With AAD, we produce the accurate charts on page 425 in half a second. We could run 1 M path to produce even better, smoother risk reports in a second.

Interestingly perhaps, one second is approximately the time it takes to produce a similar risk report with comparable accuracy, with FDM in place of Monte-Carlo, on a single core with finite differences. The combination of AAD and parallelism roughly brings FDM speed to Monte-Carlo simulations.

With 3M paths, we produce the much smoother charts below in around three seconds (that is three seconds *each*, although we produce them both in around three seconds in Chapter 14, and three seconds is with the improvements of Chapter 15):




It would have taken at least half an hour with smart, multi-threaded bumping, at least four hours in serial mode. For all intents and purposes, to obtain such precise microbuckets with Monte-Carlo simulations is not feasible without AAD. AAD opens new research possibilities into financial risk management. One example is that it produces accurate microbuckets in a few seconds. For other, perhaps unex-

pected applications, see, for example, Reghai's [97] and Andreasen's [82].

We reiterate that the main purpose of derivatives systems is not to produce sensitivities to model parameters, but risks to market instruments. For this reason, this comparison between AAD and bumping is somewhat invidious. With bumping, sensitivities to hedge instruments could be produced faster by directly bumping a smaller number of market prices and repeating the whole valuation process, including the calibration of the model parameters. However, to produce market risks in this manner could lead to instabilities from differentiating through calibration. Further, the sensitivity to model parameters would not be computed, and that is valuable information in its own right. Finally, it would not offer the flexibility to compute risks with different instruments than those used to calibrate model parameters.

We have seen that AAD produces sensitivities to a large number of model parameters extremely fast. With the concern for costs of computing a large number of sensitivities out of the way, the better strategy for producing risks is to first compute sensitivities to model parameters, as explained in this chapter, and next turn them into sensitivities to hedge instruments, as explained in the next chapter, where we calibrate Dupire's model to market data and produce risks, not to local volatilities, but to implied volatilities, which directly reflect the market prices of European options.

## NOTES

1 Or we could throw an exception, or even implement a static assert that would result in a compilation error when code attempts to call the method on a Model<T> other than Model <Number>, although we implement the simple solution here.

2 The parameters' adjoint accumulated the *sum* of contributions over all the paths; we must divide by $N$ to obtain the sensitivities of values,

which are averages of payoffs across paths.

3 One risk report for each payoff, as opposed to one risk report for an aggregate–we produce itemized risk reports with AAD in Chapter 14.

4 To be fair, with a 3y maturity, we “only” have 1,080 active local volatilities, so we would be done in an hour. More generally, our bump risk implementation is particularly dumb, only for reference against the AAD code. Before AAD made its way to finance, derivatives systems typically implemented a “smart bump” risk, a conscientious, invasive work on models to recalculate as little as possible in the bumped scenarios. We don't do that here because it is not the subject, but we acknowledge that a smart bump risk could complete in this example in around 20 minutes. With multi-threading over the 8 cores of our iMac Pro, it could complete in less than 3 minutes. As a comparison, AAD risk computes in around 8 seconds with the current implementation, 5 seconds with the improvements of Chapter 15, and *half a second* with multi-threading. This is still at least 300 times faster than the smartest possible implementation with bumps.

5 Products are unaffected because they are not supposed to store Numbers in their state, although this is not enforced, so we must be very careful there.

6 We used a vector<int> in place of a vector<bool>. For legacy reasons, vector<bool> is not thread safe in standard C++ for concurrent writes into separate entries, and should be avoided.

7 This code is not exception safe. If an exception occurs here, the main thread's tape is never reset. Ideally, we would code some RAII logic to reset the main thread's tape no matter what.