CHAPTER 14 Multiple Differentiation in *Almost* Constant Time

14.1 MULTIDIMENSIONAL DIFFERENTIATION

In this chapter, we discuss the efficient differentiation of multidimensional functions. In general terms, we have a function:

$$F : \mathbb{R}^n \to \mathbb{R}^m$$

that produces m results $(F_k)_{1 \le k \le m}$ out of n inputs $(x_i)_{1 \le i \le n}$. The function F is implemented in code and we want to compute its $m \times n$ Jacobian matrix:

$$\frac{\partial F_k}{\partial x_i}$$

as efficiently as possible.

We have seen that AAD run time is constant in n, but linear in m, and that this is the case even when the *evaluation* of F is constant in m, because the adjoint propagation step is always linear in m. Finite differences, and most other differentiation algorithms, on the other hand, are constant in m and linear in n. As a result, AAD offers an order of magnitude acceleration in cases where the number of inputs n is significantly larger than the number of results m, as is typical in financial applications.

We have covered so far the case where $m = 1$, and AAD computes a large number n of differentials very efficiently in constant time. When $m > 1$, linear time is inescapable. The time spent in the computation of the Jacobian matrix is necessarily:

$$\Delta T = \alpha + \beta m$$

The theoretical, asymptotic differentiation time will remain linear irrespective of our efforts. In situations of practical relevance, however, when m is limited compared to n, and the marginal cost β is many times smaller than the fixed cost α, the Jacobian computes in "almost" constant time. This is the best result we can hope to achieve.

In the context of financial simulations, we have efficiently differentiated *one* result in Chapter 12. We pointed out that this is not limited to the value of one transaction. Our library differentiates, for example, the value of a portfolio of many European options, defined as a single product on page 238. We can differentiate the value of a portfolio. What we cannot do is differentiate separately every transaction in the portfolio, and obtain an *itemized* risk report, one per transaction, as opposed to one aggregated risk report for the entire portfolio. The techniques in this chapter permit to compute itemized risk reports in "almost" the same time it takes to compute an aggregated one.

The reason why we deferred this discussion to the penultimate chapter is not difficulty. What we cover here is not especially hard if the previous chapters are well understood. We opted to explain the core

mechanics of AAD in the context of a one-dimensional differentiation, in a focused manner and without the distraction of the notions and constructs necessary in the multidimensional case. In addition, one-dimensional differentiation, including the aggregated risk of a portfolio of transactions, constitutes by far the most common application in finance. For this reason, we take special care that the additional developments conducted here should not slow down the one-dimensional case.

14.2 TRADITIONAL MULTIDIMENSIONAL AAD

Traditional AAD does not provide specific support for multiple results in the AAD library, and leaves it to the client code to compute the Jacobian of F in the following, rather self-evident, manner:

1. Evaluate an instrumented instance of F, which builds its tape once and for all.
2. For each result F_k:
    1. *Reseed the tape*, setting the adjoint of F_k to one and all other adjoints on tape to zero.
    2. Back-propagate. Read the *k*th row of the Jacobian $\partial F_k / \partial x_i$ in the adjoints of the n inputs x_i. Store the results. Repeat.

Our code contains all the necessary pieces to implement this algorithm. The method:

$$resetAdjoints()$$

on the Tape and Number classes reset the value of all adjoints on tape to zero. The method overloads:

$$propagateAdjoints()$$

on the Number class set the caller Number's adjoint to one prior to propagation.

The complexity of this algorithm is visibly linear in m, back-propagation being repeated m times. It may be efficient or not depending on the instrumented code. In the context of our simulation code, it is *extremely* inefficient.

To see this, remember that we optimized the simulation code of Chapter 6 so that as much calculation as possible happens once on initialization and as little as possible during the repeated simulations. The resulting tape, illustrated on page 414, is mainly populated with the model parameters and the initial computations, the simulation part occupying a minor fraction of the space. Consequently, we implemented in Chapter 12 a strong optimization whereby adjoints are back-propagated through the minor simulation piece repeatedly during simulations, and only once after all simulations completed, through the initialization phase.

It is this optimization, along with selective instrumentation, that allowed to differentiate simulations with remarkable speed, both against model parameters in Chapter 12 and against market variables in Chapter 13. But this is not scalable for a multidimensional differentiation. With the naive multidimensional AAD algorithm explained above, the adjoints of F_k are reset after they are computed, so the adjoints of F_{k+1} populate the tape next. We can no longer accumulate the adjoints of the parameters and initializers across simulations, because adjoints are overwritten by those of other results every time. To implement the traditional multidimensional algorithm would force us to repeatedly propagate adjoints, in every simulation, *back to the beginning of the tape*, resulting in a slower differentiation by orders of magnitude.

Traditional multidimensional AAD may be sufficient in some situations, but simulation is not one of them. We must find another way.

14.3 MULTIDIMENSIONAL ADJOINTS

After implementing and testing a number of possible solutions, we decided on a particularly efficient and rather simple one, although it cannot be implemented in client code without specific support from the AAD library: we work with not one, but *multiple adjoints*, one per result.

With the definitions and notations of Section 9.3, we define the instrumented calculation as a sequence $f_i, 1 \le i \le N$ of operations f_i, that produce the intermediate results y_i out of arguments y_j where $j \in C_i$, C_i being the set of indices of f_i's arguments in the sequence:

$$y_i = f_i(y_j, j \in C_i)$$

The first ys are the inputs to the calculation: for $1 \le i \le n$, $y_i = x_i$, and some ys, typically late in the sequence, are the results:

$$y_{N_k} = F_k$$

Like in Section 9.3, we define the adjoints A_i of all the intermediate results y_i, except that adjoints are no longer scalars, but m-dimensional vectors:

$$A_i = (a_i^1, ..., a_i^m)$$

with $a_i^k \equiv \frac{\partial F_k}{\partial y_i}$.

To every operation number i involved in the calculation, correspond not one adjoint a_i, the derivative of the final result to the intermediate result y_i, but m adjoints, the derivatives of the m final results F_k to the intermediate result y_i. This definition can be written in a compact, vector form:

$$A_i \equiv \frac{\partial F}{\partial y_i}$$

It immediately follows this definition that:

$$\frac{\partial F}{\partial x_i} = A_i$$

so the desired Jacobian, the differentials of results to inputs, are still the input's adjoints. It also follows that for every component k of F:

$$A_{N_k} = (1_{\{N_k = k'\}}, 1 \leq k' \leq m)$$

which specifies the boundary condition for adjoint propagation. The adjoint equation itself immediately follows from the chain rule:

$$a_j^k = \sum_{i/j \in C_i} \frac{\partial y_i}{\partial y_j} a_i^k$$

or, in compact vector form:

$$A_j = \sum_{i/j \in C_i} \frac{\partial y_i}{\partial y_j} A_i$$

and the back-propagation equation from node i to nodes $j \in C_i$ follows:

$$a_j^k += \frac{\partial y_i}{\partial y_j} a_i^k$$

or in compact form:

$$A_j += \frac{\partial y_i}{\partial y_j} A_i$$

Unsurprisingly, this is the exact same equation as previously, only it is now a vector equation in dimension m, no longer a scalar equation in dimension 1. The m adjoints Aj are propagated together from the m adjoints A_i, weighted by the same local derivative $\partial y_i / \partial y_j$. It follows that back-propagation remains, as expected, linear in m, but the rest is unchanged. For instance, in the case of simulations, we can propagate through a limited piece of the tape repeatedly during simulations, and through the initialization part, only once in the end. Therefore, the *marginal* cost β of additional differentiated results remains limited, resulting in an “almost” constant time differentiation, as will be verified shortly with numerical results. First, we implement these equations in code.

14.4 AAD LIBRARY SUPPORT

The manipulations above cannot be directly implemented in client code. We must develop support in the AAD library for the representation, storage, and propagation of multidimensional adjoints. Those manipulations necessarily carry overhead, and we don't want to adversely affect the speed of one-dimensional differentiations. For this reason, and at the risk of code duplication, we keep the one-dimensional and multidimensional codes separate.

Static multidimensional indicators

First, we add some static indicators on the Node and Tape classes, that set the context to either single or multidimensional and specify its dimension. On the Node class, we add a static data member $numAdj$ ($= m$) that stores the dimension of adjoints, also the number of differentiated results. We keep it private, although it must be accessible from the Tape and Number classes, so we make these classes friends of the Node:

```
//  Node class in AADNode.h
class Node
{
    friend class Tape;
    friend class Number;

    //  Number of adjoints (results) to propagate, usually 1
    static size_t   numAdj;

//  ...
```

Similarly, we put a static boolean indicator $multi$ on the Tape class. It is somewhat redundant with $Node::numAdj$, although it is faster to test a boolean than compare a size_t to one. Since we test every time we record an operation, the performance gain justifies the duplication:

```
//  Tape class in AADTape.h
class Tape
{
    friend class Number;

    //      Working with multiple results / adjoints?
    static bool                                 multi;

//  ...
```

We also provide a function *setNumResultsForAAD()* in AAD.h so client code may set the context prior to a differentiation:

```
1  inline auto setNumResultsForAAD(
2      const bool multi = false,
3      const size_t numResults = 1)
4  {
5      Tape::multi = multi;
6      Node::numAdj = numResults;
7      return make_unique<numResultsResetterForAAD>();
8  }
```

This function returns a (unique) pointer on an RAII object that resets dimension to 1 on destruction:

```
1  //  Also in AAD.h
2  struct numResultsResetterForAAD
3  {
4      ~numResultsResetterForAAD()
5      {
6          Tape::multi = false;
7          Node::numAdj = 1;
8      }
9  };
```

Client code sets the context by a call like:

```
    //  Set context for differentiation in dimension 50
    auto resetter = setNumResultsForAAD(true, 50);
```

When *resetter* exits scope for whatever reason, its destructor is invoked and the static context is reset to dimension one. This guarantees that the static context is always reset, even in the case of an exception,

and that client code doesn't need to concern itself with the restoration of the context after completion.

The function and the RAII class must be declared friends to the Node and the Tape class to access the private indicators.

The Node class

The Node class is modified as follows:

```cpp
class Node
{
    friend class Tape;
    friend class Number;
    friend auto setNumResultsForAAD(const bool, const size_t);
    friend struct numResultsResetterForAAD;

    //  Number of adjoints (results) to propagate, usually 1
    static size_t   numAdj;

    //  Number of childs (arguments)
    const size_t n;

    //  The adjoint(s)
    //  in single case, self held
    double          mAdjoint = 0;
    //  in multi case, held separately and accessed by pointer
    double*         pAdjoints;

    //  Data lives in separate memory

    //  the n derivatives to arguments,
    double*         pDerivatives;

    //  the n pointers to the adjoints of arguments
    double**        pAdjPtrs;

public:

    Node(const size_t N = 0) : n(N) {}

    //  Access to adjoint(s)
    //  single
    double& adjoint() { return mAdjoint; }
    //  multi
    double& adjoint(const size_t n) { return pAdjoints[n]; }

    //  Back-propagate adjoints to arguments adjoints

    //  Single case
    void propagateOne()
    {
        //  Nothing to propagate
        if (!n || !mAdjoint) return;

        for (size_t i = 0; i < n; ++i)
        {
            *(pAdjPtrs[i]) += pDerivatives[i] * mAdjoint;
        }
    }

    //  Multi case
    void propagateAll()
    {
        //  No adjoint to propagate
        if (!n || all_of(pAdjoints, pAdjoints + numAdj,
            [](const double& x) { return !x; }))
            return;

        for (size_t i = 0; i < n; ++i)
        {
            double *adjPtrs = pAdjPtrs[i], ders = pDerivatives[i];

            //  Vectorized!
            for (size_t j = 0; j < numAdj; ++j)
            {
                adjPtrs[j] += ders * pAdjoints[j];
            }
        }
```

```
70      }
71  };
```

In addition to the static indicator and friends discussed earlier, we have three modifications:

1. The storage of adjoints on line 14. We keep a scalar adjoint $mAdjoint$ on the node for the one-dimensional case. In the multi-dimensional case, the $m = nAdj$ adjoints live in separate memory on tape, like the derivatives and child adjoint pointers of Chapter 10, and are accessed with the pointer $pAdjoints$.
2. Consequently, we must modify how client code accesses adjoints on line 32. In the one-dimensional case, the single adjoint is accessed, as previously, with the $adjoint()$ accessor without arguments. In the multidimensional case, adjoints are accessed with an overload that takes the index of the adjoint between 0 and $m - 1$ as an argument.
3. Finally, propagation on line 52 is unchanged in the one-dimensional case,[1] and we code another propagation function $propagateAll()$ for the multidimensional case. This function back-propagates the m adjoints altogether to the child adjoints, multiplied by the same local derivatives. Visual Studio 2017 vectorizes the innermost loop, another step on the way of an almost constant time differentiation.

The Tape class

The Tape class is modified in a similar manner:

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

constexpr size_t BLOCKSIZE = 16384;        //  Number of nodes
constexpr size_t ADJSIZE   = 32768;        //  Number of adjoints
constexpr size_t DATASIZE  = 65536;        //  Data in bytes

class Tape
{
    //  Working with multiple results / adjoints?
    static bool                            multi;

    //  Storage for adjoints
    blocklist<double, ADJSIZE>             myAdjointsMulti;

    //  Storage for the nodes
    blocklist<Node, BLOCKSIZE>             myNodes;

    //  Storage for 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];

    friend auto setNumResultsForAAD(const bool, const size_t);
    friend struct numResultsResetterForAAD;
    friend class Number;

public:

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

        //  Store and zero the adjoint(s)
        if (multi)
        {
           node->pAdjoints = myAdjointsMulti.emplace_back_multi(Node::numAdj);
           fill(node->pAdjoints, node->pAdjoints + Node::numAdj, 0.0);
        }

        //  Store the derivatives and child adjoint pointers unless leaf
        if constexpr(N)
        {
            node->pDerivatives = myDers.emplace_back_multi<N>();
            node->pAdjPtrs = myArgPtrs.emplace_back_multi<N>();

        }

        return node;
    }

    void resetAdjoints()
```

```
59      {
60          if (multi)
61          {
62              myAdjointsMulti.memset(0);
63          }
64          else
65          {
66              for (Node& node : myNodes)
67              {
68                  node.mAdjoint = 0;
69              }
70          }
71      }
72
73      //  Clear
74      void clear()
75      {
76          myAdjointsMulti.clear();
77          myDers.clear();
78          myArgPtrs.clear();
79          myNodes.clear();
80      }
81
82      //  Rewind
83      void rewind()
84      {
85          if (multi)
86          {
87              myAdjointsMulti.rewind();
88          }
89          myDers.rewind();
90          myArgPtrs.rewind();
91          myNodes.rewind();
92      }
93
94      //  Set mark
95      void mark()
96      {
97          if (multi)
98          {
99              myAdjointsMulti.setmark();
100         }
101         myDers.setmark();
102         myArgPtrs.setmark();
103         myNodes.setmark();
104     }
105
106     //  Rewind to mark
107     void rewindToMark()
108     {
109         if (multi)
110         {
111             myAdjointsMulti.rewind_to_mark();
112         }
113         myDers.rewind_to_mark();
114         myArgPtrs.rewind_to_mark();
115         myNodes.rewind_to_mark();
116     }
117
118     //  Iterators, etc. unmodified
119     //  ...
120 };
```

1. We have an additional blocklist for the multidimensional adjoints on line 14.

2. The method $recordNode()$ performs, in the multidimensional case, the additional work following line 41: initialize the m adjoints to zero, store them on the adjoint blocklist, and set the Node's pointer $pAdjoints$ to access their memory space in the blocklist, like local derivatives and child adjoint pointers.
3. The method $resetAdjoints()$ is upgraded to handle the multidimensional case with a global memset in the adjoint blocklist, line 60.
4. Finally, all the methods that clean, rewind, or mark the tape, like $clear()$ on line 76 or $rewind()$ on line 85, are upgraded to also clean, rewind, or mark the adjoint blocklist so it is synchronized with the rest of the data on tape.

The Number class

Finally, we need some minor support in the Number class. First, we need some accessors for the multidimensional adjoints so client code may seed them before propagation and read them to pick differentials after propagation:

```
//  Number class in AADNumber.h
//  ...

    //  Accessors: value and adjoint

    double& value()
    {
        return myValue;
    }
    double value() const
    {
        return myValue;
    }

    //  1D case
    double& adjoint()
    {
        return myNode->adjoint();
    }
    double adjoint() const
    {
        return myNode->adjoint();
    }

    //  m-dimensional case
    double& adjoint(const size_t n)
    {
        return myNode->adjoint(n);
    }
    double adjoint(const size_t n) const
    {
        return myNode->adjoint(n);
    }

//  ...
```

We also coded some methods to facilitate back-propagation on the Number type in Chapter 10. For multidimensional propagation, we only provide a general static method, leaving it to the client code to seed the tape and provide the correct start and end propagation points:

```
//  Number class in AADNumber.h
//  ...

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

//  ...
```

14.5 INSTRUMENTATION OF SIMULATION ALGORITHMS

With the correct instrumentation of the AAD library, we can easily develop the multidimensional counterparts of the template algorithms of Chapter 12 in mcBase.h. The code is mostly identical to the one-dimensional code, so the template codes could have been merged. We opted for duplication in the interest of clarity.

The nature of the results is different: we no longer return the *vector* of the risk sensitivities of one aggregate of the product's payoffs to the model parameters. We return the *matrix* of the risk sensitivities of all the product's payoffs to the model parameters:

```
struct AADMultiSimulResults
{
    AADMultiSimulResults(
        const size_t nPath,
        const size_t nPay,
        const size_t nParam) :
            payoffs(nPath, vector<double>(nPay)),
            risks(nParam, nPay)
    {}

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

    //  matrix(0..nParam - 1, 0..nPay - 1) of risk sensitivities
    //      of all payoffs, averaged over paths
    matrix<double>      risks;
};
```

The signature of the single-threaded and multi-threaded template algorithms is otherwise unchanged, with the exception that they no longer require a payoff aggregator from the client code, since they produce an itemized risk for all the payoffs. They are listed in mcBase.h, like the one-dimensional counterparts. The single-threaded template algorithm is listed below with differences to the one-dimensional version highlighted and commented:

```
inline AADMultiSimulResults
mcSimulAADMulti(
        const Product<Number>&  prd,
        const Model<Number>&    mdl,
        const RNG&              rng,
        const size_t            nPath)
{
    auto cMdl = mdl.clone();
    auto cRng = rng.clone();

    Scenario<Number> path;}
    allocatePath(prd.defline(), path);
    cMdl->allocate(prd.timeline(), prd.defline());

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

    Tape& tape = *Number::tape;
    tape.clear();

    //  Set the AAD environment to multi-dimensional with dimension nPay
    //  Reset to 1D is automatic when resetter exits scope
          auto resetter = setNumResultsForAAD(true, nPay);

    cMdl->putParametersOnTape();
    cMdl->init(prd.timeline(), prd.defline());
    initializePath(path);
    tape.mark();

    cRng->init(cMdl->simDim());

    vector<Number> nPayoffs(nPay);
    vector<double> gaussVec(cMdl->simDim());

    //  Allocate multi-dimensional results
    //      including a matrix(0..nParam - 1, 0..nPay - 1)
    //      of risk sensitivities
          AADMultiSimulResults results(nPath, nPay, nParam);

    for (size_t i = 0; i<nPath; i++)
    {
        tape.rewindToMark();

        cRng->nextG(gaussVec);
        cMdl->generatePath(gaussVec, path);
        prd.payoffs(path, nPayoffs);

        //  Multi-dimensional propagation
        //      client code seeds the tape
        //      with the correct boundary conditions
          for (size_t j = 0; j < nPay; ++j) nPayoffs[j].adjoint(j) = 1.0;
        //      multi-dimensional propagation over simulation,
        //      end to mark
          Number::propagateAdjointsMulti(prev(tape.end()), tape.markIt());

        convertCollection(
            nPayoffs.begin(),
            nPayoffs.end(),
            results.payoffs[i].begin());
    }

    //  Multi-dimensional propagation over initialization, mark to start
      Number::propagateAdjointsMulti(tape.markIt(), tape.begin());

    //  Pack results
      for (size_t i = 0; i < nParam; ++i)
          for (size_t j = 0; j < nPay; ++j)
              results.risks[i][j] = params[i]->adjoint(j) / nPath;
```

```
70
71      tape.clear();
72
73      return results;
74  }
```

The same modifications apply to the multi-threaded code, which is therefore listed below without highlighting or comments:

```
1   inline AADMultiSimulResults
2   mcParallelSimulAADMulti(
3       const Product<Number>&  prd,
4       const Model<Number>&    mdl,
5       const RNG& rng,
6       const size_t            nPath)
7   {
8       const size_t nPay = prd.payoffLabels().size();
9       const size_t nParam = mdl.numParams();
10
11      Number::tape->clear();
12      auto resetter = setNumResultsForAAD(true, nPay);
13
14      ThreadPool *pool = ThreadPool::getInstance();
15      const size_t nThread = pool->numThreads();
16
17      vector<unique_ptr<Model<Number>>> models(nThread + 1);
18      for (auto& model : models)
19      {
20          model = mdl.clone();
21          model->allocate(prd.timeline(), prd.defline());
22      }
23
24      vector<Scenario<Number>> paths(nThread + 1);
25      for (auto& path : paths)
26      {
27          allocatePath(prd.defline(), path);
28      }
29
30      vector<vector<Number>> payoffs(nThread + 1, vector<Number>(nPay));
31
32      vector<Tape> tapes(nThread);
33
34      vector<int> mdlInit(nThread + 1, false);
35
36      initModel4ParallelAAD(prd, *models[0], paths[0]);
37
38      mdlInit[0] = true;
39
40      vector<unique_ptr<RNG>> rngs(nThread + 1);
41      for (auto& random : rngs)
42      {
43          random = rng.clone();
44          random->init(models[0]->simDim());
45      }
46
47      vector<vector<double>> gaussVecs
48      (nThread + 1, vector<double>(models[0]->simDim()));
49
50      AADMultiSimulResults results(nPath, nPay, nParam);
51
```

```
    vector<TaskHandle> futures;
    futures.reserve(nPath / BATCHSIZE + 1);

    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]()
        {
            const size_t threadNum = pool->threadNum();

            if (threadNum > 0) Number::tape = &tapes[threadNum - 1];

            if (!mdlInit[threadNum])
            {
                initModel4ParallelAAD(
                     prd,
                     *models[threadNum],
                     paths[threadNum]);

                mdlInit[threadNum] = true;
            }

            auto& random = rngs[threadNum];
            random->skipTo(firstPath);

            for (size_t i = 0; i < pathsInTask; i++)
            {

                Number::tape->rewindToMark();
                random->nextG(gaussVecs[threadNum]);
                models[threadNum]->generatePath(
                    gaussVecs[threadNum],
                    paths[threadNum]);
                prd.payoffs(paths[threadNum], payoffs[threadNum]);

                const size_t n = payoffs[threadNum].size();
                for (size_t j = 0; j < n; ++j)
                {
                    payoffs[threadNum][j].adjoint(j) = 1.0;
                }
                Number::propagateAdjointsMulti(
                    prev(Number::tape->end()),
                    Number::tape->markIt());

                convertCollection(
                    payoffs[threadNum].begin(),
                    payoffs[threadNum].end(),
                    results.payoffs[firstPath + i].begin());
            }

            return true;
        }));

```

```
        pathsLeft -= pathsInTask;
        firstPath += pathsInTask;
    }

    for (auto& future : futures) pool->activeWait(future);

    Number::propagateAdjointsMulti(
        Number::tape->markIt(),
        Number::tape->begin());

    for (size_t i = 0; i < nThread; ++i)
    {
        if (mdlInit[i + 1])
        {
            Number::propagateAdjointsMulti(
                        tapes[i].markIt(),
                        tapes[i].begin());
        }
    }

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

    Number::tape->clear();

    return results;
}
```

14.6 RESULTS

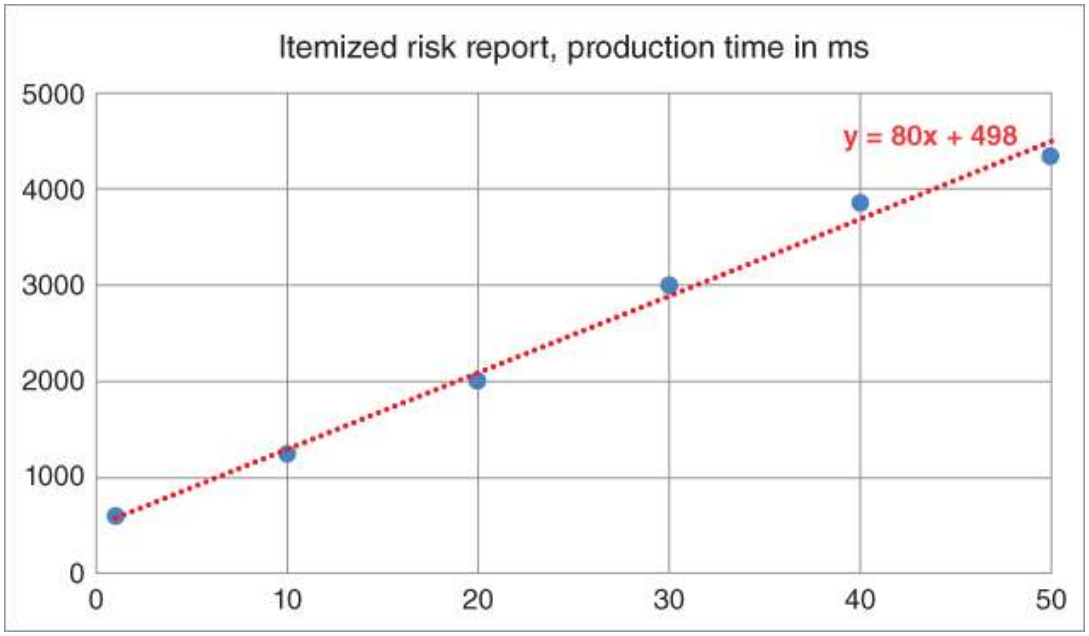
We need an entry-level function $AADriskMulti()$ in main.h, almost identical to the $AADriskOne()$ and $AADriskAggregate()$ of Chapter 12:

```
inline RiskReports AADriskMulti(
    const string&           modelId,
    const string&           productId,
    const NumericalParam&   num)
{
    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");
    }

    RiskReports results;

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

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

    results.params = model->parameterLabels();
    results.payoffs = product->payoffLabels();
    results.risks = move(simulResults.risks);

    //  Average values across paths
    const size_t nPayoffs = product->payoffLabels().size();
    results.values.resize(nPayoffs);
    for (size_t i = 0; i < nPayoffs; ++i)
    {
        results.values[i] = accumulate(
            simulResults.payoffs.begin(),
            simulResults.payoffs.end(),
            0.0,
            [i](const double acc, const vector<double>& v)
                {
                    return acc + v[i];
                }
        ) / num.numPath;
    }

    return results;
}
```

so we can produce an itemized risk report. To produce numerical results, we set Dupire's model in the same way as for the numerical results of Chapter 12, where it took a second (half a second with the improvements in the next chapter) to produce the accurate microbucket of a barrier option of maturity 3y (with Sobol numbers, 500,000 paths over 156 weekly steps, in parallel over 8 cores). With a European call, without barrier monitoring, it takes around 850 ms (450 ms with expression templates).

To obtain the exact same microbucket for a single European call, but in a portfolio of one single transaction, with the multidimensional algorithm, takes 600 ms with expression templates. Irrespective of the number of itemized risk reports we are producing, multidimensional differentiation carries a measurable additional fixed overhead of around 30%. This is why we kept two separate cases instead of programming the one-dimensional case as multidimensional with dimension one.

To produce 50 microbuckets for 50 European options, with different strikes and maturity 3y or less, takes only 4,350 ms with expression templates, around seven times one microbucket. The following chart shows the times, in milliseconds, to produce itemized risk reports, with expression templates, as a function of the number of microbuckets:



As promised, the time is obviously linear in the number of transactions, but almost constant in the sense that the marginal time to produce an additional microbucket is only 80 ms, compared to the 600 ms for the production of one microbucket.

We produced 50 microbuckets in less than 5 seconds, but this is with the improvements of the next chapter. With the traditional AAD library, it takes around 15 seconds. The expression template optimization is particularly effective in this example, with an acceleration by a factor of three. Expression templates are introduced in the next and final chapter.

NOTE

1 And we understand now why we called the function "propagateOne()".