CHAPTER 13

Check-Pointing and Calibration

In the previous chapter, we learned to differentiate simulations against model parameters in constant time, and produced *microbuckets* (sensitivities to local volatilities) in Dupire's model with remarkable speed. In this chapter, we present the key check-pointing algorithm, apply it to differentiation through calibrations to obtain market risks out of sensitivities to model parameters, and implement *superbucket* risk reports (sensitivities to implied volatilities) in Dupire's model.

13.1 CHECK-POINTING

Reducing RAM consumption with check-pointing

We pointed out in Chapter 11 that, due to RAM consumption, AAD cannot efficiently differentiate calculations taking more than 0.01 seconds on a core. Longer calculations, which include almost all cases of practical relevance, must be divided into pieces shorter than 0.01 seconds.core, and differentiated separately over each piece, wiping RAM in between and aggregating sensitivities in the end.

How exactly this is achieved depends on the instrumented algorithm. This is particularly simple in the case of path-wise simulations, where sensitivities are computed path by path and averaged in the end. But this solution is specific to simulations. This chapter discusses a more general solution called *check-pointing*. Check-pointing applies to many problems of practical relevance, in finance and elsewhere. Huge successfully applied it to the differentiation of multidimensional FDM in [90]. Huge and Savine applied it to efficiently differentiate the LSM algorithm and produce a complete xVA risk in [31]. We explain check-pointing in general mathematical and programmatic terms and apply it to the differentiation of calibration.

Formally, we differentiate a two-step algorithm, that is, a scalar function $F : \mathbb{R}^n \to \mathbb{R}$ that can be written as:

$$F(X) = H[G(X)]$$

where $G : \mathbb{R}^n \to \mathbb{R}^m$ is a vector-valued function and $H : \mathbb{R}^m \to \mathbb{R}$ is a scalar function, assumed differentiable in constant time. We denote $Y = G(X)$ and $z = F(X) = H[G(X)]$.

In the context of a calibrated financial simulation model, H is the value of some transaction in some model with m parameters Y. We learned to differentiate H in constant time in the previous chapter. G is an explicit[1] calibration that produces the m model parameters Y out of n market variables X, with $m \leq n$. F computes the value z of the transaction from the market variables X. The differentials of F are the hedge coefficients, also called *market risks*, that risk reports are designed to produce.

Importantly, there are many good reasons to split differentiation this way, besides minimizing the size of the tape. In the context of financial simulations, intermediate differentials provide useful information for research and risk management, the sensitivity of transactions to model parameters and the sensitivity of model parameters to market variables aggregated into market risks, but also constitute interesting information in their own right. Besides, to differentiate calibration is fine when it is explicit, but to differentiate a numerical calibration may result in unstable, inaccurate sensitivities. We present in Section 13.3 a specific algorithm for the differentiation of numerical calibrations. But we can only do that if we split differentiation between calibration and valuation.

Another example is how we differentiated the simulation algorithm in the previous chapter. We split the simulation algorithm F into an initialization phase G, which pre-calculates m deterministic results Y from the model parameters X, and a simulation phase H, which generates and evaluates paths and produces a value z out of X and Y.[2] We implemented H in a loop over paths, and differentiated each of its iterations separately, before propagating the resulting differentials over G. This is a direct application of check-pointing, although we didn't call it by its name at the time. If we hadn't split the differentiation of F into the differentiation of G and the differentiation of H, we could not have implemented path-wise differentiation that efficiently. More importantly, we could not have conducted the differentiation of H in parallel. In order to differentiate in parallel a parallel calculation, we must first extract the parallel piece and differentiate it separately from the rest, applying check-pointing to connect the resulting differentials.

As a final example, we could split the processing F of every path into the generation G of the m-dimensional scenario Y and the evaluation H of the payoff. We differentiated F altogether in the previous chapter, but for a

[1]Implicit (numerical) calibration is discussed later in this chapter when we present the Implicit Function Theorem. For now, G is an explicit function that expresses the model parameters out of market prices, like Dupire's formula [12].

[2]So in this case $F(X) = H[X, G(X)]$ not $H[G(X)]$, but this doesn't change anything, as will be demonstrated shortly.

product with a vast number of cash-flows valued over a high-dimensional path, like an xVA, we would separate the differentiation of the two to limit RAM consumption.

In all these cases, H can be differentiated in constant time with AAD because it is a scalar function. In theory, F is also differentiable in constant time because it is also a scalar function. But we discussed some of the many reasons why it may be desirable to split its differentiation. Hence, the exercise is to split the differentiation of F into a differentiation of G and a differentiation of H while preserving the constant time property. The problem is that G is *not* a scalar function, hence it *cannot* be differentiated in constant time with straightforward AAD. To achieve this, we need additional logic, and it is this additional logic that is called check-pointing.

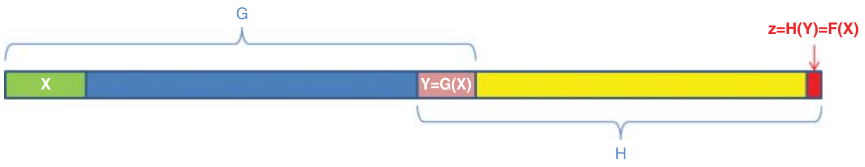
Formally, from the chain rule:

$$\frac{\partial F}{\partial X} = \frac{\partial G}{\partial X}\frac{\partial H}{\partial Y}$$

and our assumption is that we have a constant time computation for $\frac{\partial H}{\partial Y}$. But AAD cannot compute the Jacobian $\frac{\partial G}{\partial X}$ in constant time. We have seen in Chapter 11 that Jacobians take linear time in the number of results m. With bumping, it takes linear time in n. In any case, it cannot be computed in constant time. Furthermore, $\frac{\partial G}{\partial X}\frac{\partial H}{\partial Y}$ is the product of the m vector $\frac{\partial H}{\partial Y}$ by the $m \times n$ matrix $\frac{\partial G}{\partial X}$, linear in nm.

Check-pointing applies adjoint calculus to compute $\frac{\partial F}{\partial X}$ in constant time, in a sequence of steps where the adjoints of F and G are propagated separately, without ever computing a Jacobian or performing a matrix product.

If, hypothetically, we did differentiate F altogether with a single application of AAD, what would the tape look like?



It must be this way, because G is computed before H, and H only depends on the results of G.

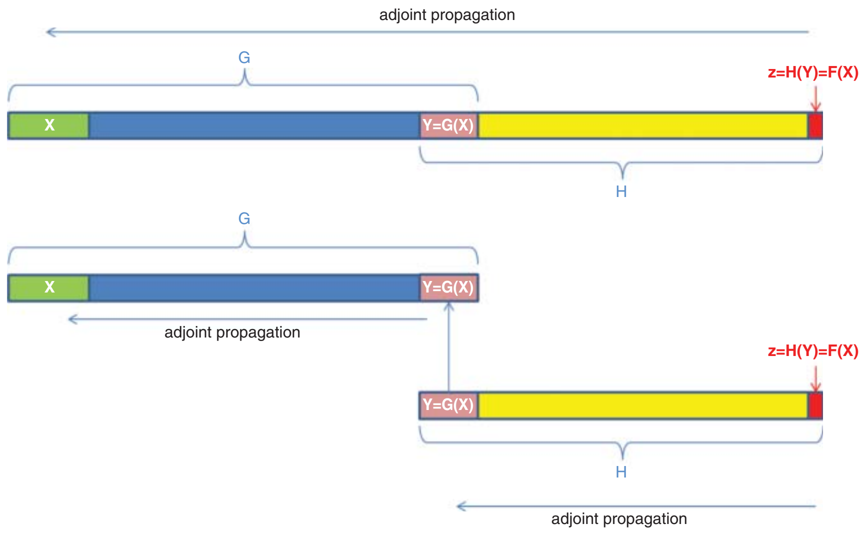
The part of the tape that belongs to H is self-contained for adjoint propagation, because H only depends on Y, and not on the details of its internal calculations within G.[3] So the arguments to all calculations within H must

[3]We assume that G and H are functional in the sense that they produce outputs out of inputs without modification of hidden state.

be located on H's part of the tape, including Y; they cannot belong to G's part of the tape.

The section of the tape that belongs to G (inclusive of inputs X and outputs Y) is also self-contained. Evidently, the calculations within G cannot depend on the calculations in H, which is evaluated *later*. And we have seen that the calculations within H cannot directly reference those of G, except through its outputs Y.

Hence, the tape for F is *separable* for the purpose of adjoint propagation: it can be split into two self-contained tapes, with a common section Y as the output to G's tape and the input to H's tape.



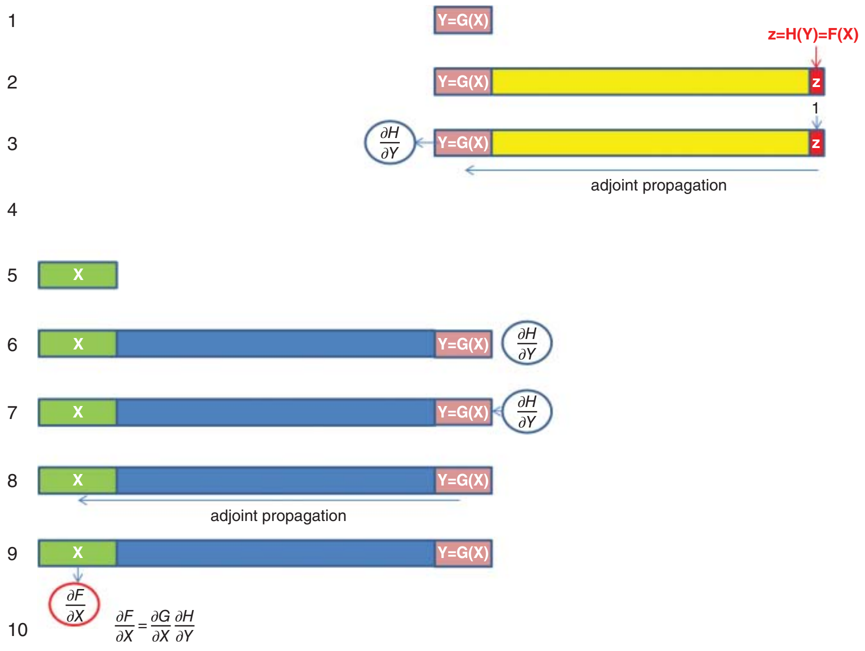
It should be clear that an overall back-propagation through the entire tape of F is equivalent, and produces the same results, as two successive back-propagations, first through the tape of H, and then through the tape of G. Note that the order is reversed from evaluation, where G is evaluated first and H is evaluated last.

It is this separation that allows the multiple benefits of separate differentiations, including a smaller RAM consumption. Only one of the two tapes of G and H is active at a time in memory, and the differentials of $z = F(X)$ are accumulated through adjoint propagation alone, hence, in constant time. The check-pointing algorithm is articulated below:

1. Starting with a clean tape, compute and store $Y = G(X)$ without AAD instrumentation. The only purpose here is to compute Y. Put Y on tape.
2. Compute the final result $z = H(Y) = F(X)$ with an *instrumented* evaluation of H. This builds the tape for H.

3. Back-propagate from z to Y, producing the adjoints of Y: $\frac{\partial H}{\partial Y}$. Store this result.
4. Wipe H's tape. It is no longer needed.
5. Put X on tape.
6. Recompute $Y = G(X)$ as in step 1, this time with AAD instrumentation. This builds the tape for G.
7. Seed that tape with the adjoints of Y, that is the $\frac{\partial H}{\partial Y}$, known from step 3. This is the defining step in the check-pointing algorithm. Instead of seeding the tape with 1 for the end result (and 0 everywhere else), seed it with the known adjoints for all the components of the vector Y.
8. Conduct back-propagation through the tape of G, from the known adjoints of Y to those, unknown, of X.
9. The adjoints of X are the final, desired result: $\frac{\partial F}{\partial X} = \frac{\partial G}{\partial X}\frac{\partial H}{\partial Y}$.
10. Wipe the tape.

The following figure shows the state of the tape after each step:


1
2
3
4
5
6
7
8
9
10
Y=G(X)
z=H(Y)=F(X)
z
1
$\frac{\partial H}{\partial Y}$
adjoint propagation
X
$\frac{\partial F}{\partial X}$
$\frac{\partial F}{\partial X} = \frac{\partial G}{\partial X}\frac{\partial H}{\partial Y}$

It should be clear that this algorithm guarantees all of the following:

Constant time computation since only adjoint propagations are involved. Note that the Jacobian of G is *never* computed. We don't know it at the term of the computation, and we don't need it to produce the end result. Also note that successive functions are propagated in the reverse order to their evaluation. Check-pointing is a sort of "macro-level" AAD where the nodes are not mathematical operations but steps in an algorithm.

Correct adjoint accumulation since it should be clear that these computations produce the exact same results as a full adjoint propagation throughout the entire tape for *F*. It is actually the same propagations that are executed, but through pieces of tape at a time.

Reduced RAM consumption since only the one tape for *G* or *H* lives in memory at a time.

In code, check-pointing goes as follows:

```
template<class T>
T H(const vector<T>& Y);

template<class T>
vector<T> G(const vector<T>& X);

//  Implements check-pointing
//  Takes input X
//  Computes and returns F(X) = H[G(X)] and its derivatives
inline pair<double, vector<double>> checkPoint(const vector<double>& X)
{
    //  Start with a clean tape
    auto* tape = Number::tape;
    tape->clear();

    //  1
    //  Compute Y
    vector<double> Y = G(X);
    //  Convert to numbers
    vector<Number> iY(Y.size());
    convertCollection(Y.begin(), Y.end(), iY.begin());
    //  Note that also puts iY on tape

    //  2
    Number z = H(iY);

    //  3
    //  Propagate
    z.propagateToStart();
    //  Store derivatives
    vector<double> dhdy(iY.size());
    transform(iY.begin(), iY.end(), dhdy.begin(),
        [](const Number& y) {return y.adjoint(); });

    //  4
    tape->clear();

    //  5
    vector<Number> iX(X.size());
    convertCollection(X.begin(), X.end(), iX.begin());

    //  6
    vector<Number> oY = G(iX);

    //  7
    for (size_t i = 0; i < oY.size(); ++i)
    {
        oY[i].adjoint() = dhdy[i];
    }

    //  8
    Number::propagateAdjoints(prev(tape->end()), tape->begin());

    //  9
    pair<double, vector<double>> results;
    results.first = z.value();
    results.second.resize(iX.size());
    transform(iX.begin(), iX.end(), results.second.begin(),
        [](const Number& x) {return x.adjoint(); });

    //  10
    tape->clear();

    return results;
}
```

We could write a generic higher-order function to encapsulate check-pointing logic. But we refrain from doing so. It is best left to client code to implement check-pointing at best in different situations. The AAD library provides all the basic constructs to implement check-pointing easily and in a flexible manner.

Note that the algorithm also works in the more general context where:

$$F(X) = H[X, G(X)]$$

because, then:

$$\frac{\partial F}{\partial X} = \frac{\partial H}{\partial X} + \frac{\partial H}{\partial Y}\frac{\partial G}{\partial X}$$

The left-hand side $\frac{\partial H}{\partial X}$ is computed in constant time by differentiation of H (we can do that: it has been our working hypothesis all along). The right-hand side $\frac{\partial H}{\partial Y}\frac{\partial G}{\partial X}$ is computed by check-pointing.

Alternatively, we may redefine G to return the n coordinates of X in addition to its result Y, and we are back to the initial case where $F(X) = H[G(X)]$.

This concludes our general discussion of check-pointing. Check-pointing applies in vast number of contexts, to the point that every nontrivial AAD instrumentation involves some form of check-pointing, including the instrumentation of our simulation library in the previous chapter, as pointed out earlier. In the next section, we apply check-pointing to calibration and the production of market risks. In the meantime, we quickly discuss application to black box code.

Check-pointing black box code

AAD instrumentation cannot be partial. The entire calculation code must be instrumented, and all the active code must be templated. A partial instrumentation would break the chain rule and prevent adjoints to correctly propagate through non-instrumented active calculations, resulting in wrong differentials. It follows that all the source code implementing a calculation must be available, and modifiable, so it may be instantiated with the Number type.

In a real-world production environment, this is not always the case. It often happens that part of the calculation code is a black box. We can call this code to conduct some intermediate calculations, but we cannot easily see or modify the source code. The routine may be part of third-party software with signatures in headers and binary libraries, but no source code. Or, the source code may be written in a different language. Or, the source is available but cannot be modified, for technical, policy, or legal reasons.

Or maybe we could instrument the code but we don't want to. An intensive calculation code with a low number of active inputs may be best differentiated either analytically or with finite differences. Or, as we will see in the case of a numerical calibration, some code must be differentiated in a specific manner, a blind differentiation, either with finite differences or AAD, resulting in wrong or unstable derivatives. This applies to many iterative algorithms, like eigenvalue decomposition, Cholesky decomposition, or SVD regression, as noted by Huge in [93].

In all these cases, we have an intermediate calculation that remains non-instrumented, and differentiated in its own specific way, which may or may not be finite differences.[4] Check-pointing allows to consistently connect this piece in the context of a larger differentiation, the rest of the calculation being differentiated in constant time with AAD.

For example, consider the differentiation of a calculation F that is evaluated in three steps, $G : \mathbb{R}^n \to \mathbb{R}^m$, $BB : \mathbb{R}^m \to \mathbb{R}^p$, and $H : \mathbb{R}^p \to \mathbb{R}$:

$$F(X) = H\{BB[G(X)]\}$$

where BB is the black box. It is not instrumented, and its Jacobian $\partial BB/\partial G$ is computed by specific means, perhaps finite differences. The problem is to

[4]A third-party black box can always be differentiated with finite differences.

conduct the rest of the differentiation in constant time with AAD, and connect the Jacobian of the black box without breaking the derivatives chain. We have discussed a walkaround in Chapter 8 in the context of manual adjoint code. In the context of automatic adjoint differentiation, we have two choices.

We could *overload BB* and make it a building block in the AAD library, like we did for the Gaussian functions in Chapter 10. This solution invades and grows the AAD library. It is recommended when BB is a low-level, general-purpose algorithm, called from many places in the software.

In most situations, however, BB would be a necessary intermediate calculation in a specific context, which doesn't justify an invasion of the AAD library. All we need is a walkaround in the instrumentation of F, along the lines of Chapter 8, but with automatic adjoint propagation. We can implement such walkaround with check-pointing. Denote:

$$Y = G(X) \quad , \quad Z = BB(Y) \quad , \quad z = H(Z)$$

then, by the chain rule:

$$\frac{\partial F}{\partial X} = \frac{\partial H}{\partial Z}\frac{\partial BB}{\partial Y}\frac{\partial G}{\partial X}$$

We start with a non-instrumented calculation of $Y = G(X)$, as is customary with check-pointing. Next, we compute the value $Z = BB(Y)$ of BB, as well as its Jacobian $\partial BB/\partial Y$, computed, as discussed, by specific means.

Knowing Z, we compute the gradient $\partial H/\partial Z$, a row vector in dimension p, of H, in constant time, with AAD instrumentation. We multiply it on the right by $\partial BB/\partial Y$ to find:

$$\frac{\partial F}{\partial Y} = \frac{\partial H}{\partial Z}\frac{\partial BB}{\partial Y}$$

This row vector in dimension m is by definition the adjoint of Y in the calculation of F. We can therefore apply the check-pointing algorithm. Execute an instrumented instance of:

$$Y = G(X)$$

which builds the tape of G, seed the adjoints of the results Y with the known $\partial F/\partial Y$, and back-propagate to find the desired adjoints of X, that is $\partial F/\partial X$.

We successfully applied check-pointing to connect the specific differentiation of BB with the rest of the differentiated calculation. The differentiation of BB takes the times it takes, and a matrix-by-vector product is necessary for the connection, but the rest of the differentiation proceeds with AAD in constant time.

13.2 EXPLICIT CALIBRATION

Dupire's formula

We now turn toward financial applications of the check-pointing algorithm, more precisely, the important matter of the production of market risks.

So far in this book, we implemented Dupire's model with a given local volatility surface $\sigma(S, t)$, represented in practice by a bilinearly interpolated matrix. Its differentiation produced the derivatives of some transaction's value in the model with respect to this local volatility matrix.

But this is not the application Dupire meant for his model. Traders are not interested in risks to a theoretical, abstract local volatility. Dupire's model is meant to be *calibrated* to the market prices of European calls and puts, or, equivalently, market-implied Black and Scholes volatilities, such that its values are consistent with the market prices of European options, and its risk sensitivities are derivatives to *implied* volatilities, which represent the market prices of concrete instruments that traders may buy and sell to hedge the sensitivities of their transactions.

Dupire's model is unique in that its calibration is *explicit*. The calibrated local volatility is expressed directly as a function of the market prices of European calls by Dupire's famous formula [12]:

$$\sigma^2(S,t) = \frac{2C_T(S,t)}{S^2 C_{KK}(S,t)}$$

where $C(K, T)$ is today's price of the European call of strike K and maturity T, and subscripts denote partial derivatives.

Dupire's formula may be elegantly demonstrated in a couple of lines with Laurent Schwartz's generalized derivatives and Tanaka's formula (essentially an extension of Ito's lemma in the sense of distributions), following the footsteps of Savine, 2001 [44].

By application of Tanaka's formula to the function $f(x) = (x - K)^+$ under Dupire's dynamics $\frac{dS_t}{S_t} = \sigma(S_t, t)dW_t$, we find:

$$d(S_t - K)^+ = 1_{\{S_t > K\}} dS_t + \frac{1}{2}\delta(S_t - K)S_t^2\sigma^2(S_t, t)dt$$

where δ is the Dirac mass. Taking (risk-neutral) expectations on both sides:

$$dE(S_t - K)^+ = 0 + \frac{1}{2}\varphi_t(K)K^2\sigma^2(K, t)dt$$

where $\varphi_t(K) = C_{KK}(K, t)$ is the (risk-neutral) density of S_t in K, and since $E(S_t - K)^+ = C(K, t)$, we have Dupire's result:

$$\frac{dC(K,t)}{dt} = \frac{1}{2}\frac{d^2C(K,t)}{dK^2}K^2\sigma^2(K,t)$$

Similar formulas are found with this methodology in extensions of Dupire's model with rates, dividends, stochastic volatility, and jumps; see [44].

The Implied Volatility Surface (IVS)

Dupire's formula refers to today's prices of European calls of all strikes K and maturities T, or, equivalently, the continuous surface of Black and Scholes's market-implied volatilities $\hat{\sigma}(K, T)$. In Chapter 4, we pointed out that this is also equivalent to marginal risk-neutral densities for all maturities, and called this continuous surface of market prices an *Implied Volatility Surface* or IVS.

The IVS must satisfy some fundamental properties to feed Dupire's formula: it must be continuous, differentiable in T, and twice differentiable in K. C_{KK} must be strictly positive, meaning call prices must be convex in strike. We also require that $C_T > 0$, meaning call prices are increasing in maturity. $C_{KK} < 0$ or $C_T < 0$ would allow a static arbitrage (see for instance [46]) so any non-arbitrageable IVS guarantees that $C_{KK} \geq 0, C_T \geq 0$. But this is not enough. We need strict positiveness as well as continuity and differentiability.

We pointed out in Chapter 4 that the market typically provides prices for a discrete number of European options, and that to interpolate a complete, continuous, differentiable IVS out of these prices was not a trivial exercise. The implementation of the accepted solutions in the industry, including Gatheral's SVI ([40]) and Andreasen and Huge's LVI ([41]), are out of our scope here.

We circumvent this difficulty by *defining* an IVS from Merton's jump-diffusion model of 1976 [98]. Merton's model is an extension of Black and Scholes where the underlying asset price is not only subject to a diffusion, but also random discontinuities, or jumps, occurring at random times and driven by a Poisson process:

$$\frac{dS_t}{S_t} = \sigma dW + J_t dN_t - comp \cdot dt$$

where N is a Poisson process with intensity λ and the Js are a collection IID random variables such that $\log(1+J_t) \approx J_t \to N(m,v)$. The Poisson process and the jumps are independent from each other and independent from the Brownian motion. Jumps are roughly Gaussian with mean m and variance v. $comp = \lambda\left[\exp\left(m+\frac{v}{2}\right)-1\right] \approx \lambda m$ guarantees that S satisfies the martingale property so the model remains non-arbitrageable.

Merton demonstrated that the price of a European call in this model can be expressed explicitly as a weighted average of Black and Scholes prices:

$$C(K,T) = \sum_{n=0}^{\infty} \frac{\exp(-\lambda T)}{n!}(\lambda T)^n BS$$
$$\times \left(S_0 \exp\left[n\left(m+\frac{v}{2}\right) - comp \cdot T\right], \sqrt{\sigma^2 + \frac{nv}{T}}, K, T\right)$$

where $BS(S,\hat{\sigma},K,T)$ is Black and Scholes's formula. The model is purposely written so the distribution of S_T, *conditional to the number n of jumps*, is log-normal with known mean and variance. The conditional expectation of the payoff is therefore given by Black and Scholes's formula, with a different forward and variance depending on the number of jumps. It follows that the price, the *unconditional* expectation, is the average of the conditional expectations, weighted by the distribution of the number of jumps. The distribution of a Poisson process is well known, and Merton's formula follows.

The term in the infinite sum dies quickly with the factorial, so it is safe, in practice, to limit the sum to its first 5–10 terms. The formula is implemented in analytics.h, along with the Black and Scholes's formula.

We are using a continuous-time, arbitrage-free model to define the IVS; therefore, the properties necessary to feed Dupire's formula are guaranteed. In addition, Merton's model is known to produce realistic IVS with a shape similar to major equity derivatives markets.

We are using a fictitious Merton market in place of the "real" market so as to get around some technical difficulties unrelated to the purpose of this document. This is evidently for illustration purposes only and not for production.

We declare the IVS as a polymorphic class that provides a Black and Scholes market-implied volatility for all strikes and maturities. The implementation is simplified in that it ignores rates or dividends. The following code is found in ivs.h:

```
class IVS
{
    //  To avoid reference to a linear market
    double mySpot;
```

```
public:

    IVS(const double spot) : mySpot(spot) {}

    //  Read access to spot
    double spot() const
    {
        return mySpot;
    }

    //  Raw implied vol
    virtual double impliedVol(const double strike, const Time mat)
                const = 0;

    //  ...
```

where the concrete IVS derives *impliedVol*() to provide a volatility surface. The IVS also provides a method for the pricing of European calls:

```
    //  ...

    //  Call price
    template<class T = double>
    T call(
        const double strike,
        const Time mat) const
    {
        //  blackScholes is defined in analytics.h, templated
        return blackScholes<T>(
            mySpot,
            strike,
            impliedVol(strike, mat),
            mat);
    }

    //  ...
```

where the function *blackScholes* is implemented in analytics.h, templated. By application of Dupire's formula, the IVS also provides the local volatility for a given spot and time:

```
    //  ...

    //  Local vol, dupire's formula
    template<class T = double>
    T localVol(
        const double strike,
        const double mat) const
    {
        //  Derivative to time
        const T c00 = call(strike, mat, risk);
        const T c01 = call(strike, mat - 1.0e-04, risk);
        const T c02 = call(strike, mat + 1.0e-04, risk);
        const T ct = (c02 - c01) * 0.5e04;
        //  Second derivative to strike = density
        const T c10 = call(strike - 1.0e-04, mat, risk);
        const T c20 = call(strike + 1.0e-04, mat, risk);
        const T ckk = (c10 + c20 - 2.0 * c00) * 1.0e08;

        //  Dupire's formula
        return sqrt(2.0 * ct / ckk) / strike;
    }

    //  Virtual destructor needed for polymorphic class
    virtual ~IVS() {}
};
```

As discussed, we define a concrete IVS from Merton's model. All a concrete IVS must do is derive *impliedVol*():

```
class MertonIVS : public IVS
{
    double myVol;
    double myIntensity, myAverageJmp, myJmpStd;

public:

    MertonIVS(const double spot, const double vol,
        const double intens, const double aveJmp, const double stdJmp)
        : IVS(spot),
        myVol(vol),
        myIntensity(intens),
        myAverageJmp(aveJmp),
        myJmpStd(stdJmp)
    {}

    double impliedVol(const double strike, const Time mat) const override
    {
        //  Merton's formula is defined in analytics.h
        const double call
            = merton(
                spot(),
                strike,
                myVol,
                mat,
                myIntensity,
                myAverageJmp,
                myJmpStd);

        //  Implied volatility from price, also in analytics.h
        return blackScholesIvol(spot(), strike, call, mat);
    }
};
```

where *merton*() is an implementation of Merton's formula, and *blackScholesIvol*() implements a numerical procedure to find an implied volatility from an option price. Both are implemented in analytics.h.

We implemented a generic framework for IVS. Although we only implemented one concrete IVS, and a particularly simple one that defines the market from Merton's model, we could implement any other concrete IVS, including:

- Hagan's SABR [36] with parameters interpolated in maturity and underlying, as is market practice for interest rate options,
- Heston's stochastic volatility model [42] with parameters interpolated in maturity, as is market practice for foreign exchange options.[5]
- Gatheral's SVI implied volatility interpolation [40], as is market standard for equity derivatives, or
- Andreasen and Huge's recent award-winning LVI [41] argitrage-free interpolation.

Any concrete IVS implementation must only override the *impliedVol()* method to provide a Black and Scholes implied volatility for any strike and maturity. The rest, in particular the computation of Dupire's local volatility, is on the base IVS.

Calibration of Dupire's model

It is easy to calibrate Dupire's model to an IVS; all it takes is an implementation of Dupire's formula. The formula guarantees that the resulting local volatility surface in Dupire's model matches the option prices in the IVS. We write a free calibration function in mcMdlDupire.h. It accepts a target IVS, a grid of spots and times, and returns a local volatility matrix, calibrated sequentially in time:

```
#include "ivs.h"

#define ONE_HOUR 0.000114469

//  Returns a struct with spots, times and lVols
template<class T = double>
inline auto dupireCalib(
        //  The IVS we calibrate to
        const IVS& ivs,
        //  The local vol grid
        //  The spots to include
        const vector<double>& inclSpots,
        //  Maximum space between spots
```

[5]Some companies still use an approximation based on second-order sensitivities to volatility, a so-called "vega-volga-vanna" interpolation that vaguely mimics a stochastic volatility model, although it is hard to see any advantage over a correct implementation of Heston.

```
        const double maxDs,
        //  The times to include, note NOT 0
        const vector<Time>& inclTimes,
        //  Maximum space between times
        const double maxDt)
{
    //  Results
    struct
    {
        vector<double> spots;
        vector<Time> times;
        matrix<T> lVols;
    } results;

    //  Spots and times
    results.spots = fillData(inclSpots, maxDs, 0.01); //  min space = 0.01
    results.times = fillData(inclTimes, maxDt,
        ONE_HOUR,                //  min space = 1 hour
        &maxDt, &maxDt + 1       //  hack to include maxDt as first time
    );

    //  Allocate local vols, transposed maturity first
    matrix<T> lVolsT(results.times.size(), results.spots.size());

    //  Maturity by maturity
    const size_t n = results.times.size();
    for (size_t j = 0; j < n; ++j)
    {
        dupireCalibMaturity(
            ivs,
            results.times[j],
            results.spots.begin(),
            results.spots.end(),
            lVolsT[j]);
    }

    //  transpose is defined in matrix.h
    results.lVols = transpose(lVolsT);

    return results;
}
```

It is convenient to conduct the calibration sequentially in time, although our Dupire stores local volatility in spot major. For this reason, we calibrate a temporary volatility matrix in time major, and return its transpose (defined in matrix.h). We calibrate each time slice independently with the free function dupireCalibMaturity() defined in mcMdlDupire.h:

```
//  Calibrates one maturity
//  Main calibration function below
template <class IT, class OT, class T = double>
inline void dupireCalibMaturity(
    //  IVS we calibrate to
    const IVS& ivs,
    //  Maturity to calibrate
    const Time maturity,
    //  Spots for local vol
    IT spotsBegin,
    IT spotsEnd,
    //  Results, by spot
    //  With (random access) iterator, STL style
    OT  lVolsBegin)
{
    //  Number of spots
    IT spots = spotsBegin;
    const size_t nSpots = distance(spotsBegin, spotsEnd);

    //  Estimate ATM so we cut the grid 2 stdevs away to avoid instabilities
    const double atmCall = double(ivs.call(ivs.spot(), maturity));
    //  Standard deviation, approx. atm call * sqrt(2pi)
    const double std = atmCall * 2.506628274631;

    //  Skip spots below and above 2.5 std
    int il = 0;
    while (il < nSpots && spots[il] < ivs.spot() - 2.5 * std) ++il;
    int ih = nSpots - 1;
    while (ih >= 0 && spots[ih] > ivs.spot() + 2.5 * std) --ih;

    //  Loop on spots
    for (int i = il; i <= ih; ++i)
    {
        //  Dupire's formula
        lVolsBegin[i] = ivs.localVol(spots[i], maturity);
    }

    //  Extrapolate flat outside std
    for (int i = 0; i < il; ++i)
        lVolsBegin[i] = lVolsBegin[il];
    for (int i = ih + 1; i < nSpots; ++i)
        lVolsBegin[i] = lVolsBegin[ih];
}
```

Finally, we have the following higher-level function in main.h for our application:

```
//  Returns spots, times and lVols in a struct
inline auto
dupireCalib(
    //  The local vol grid
    //  The spots to include
    const vector<double>& inclSpots,
    //  Maximum space between spots
    const double maxDs,
    //  The times to include, note NOT 0
    const vector<Time>& inclTimes,
    //  Maximum space between times
    const double maxDt,
    //  The IVS we calibrate to
    //  'B'achelier, Black'S'choles or 'M'erton
    const double spot,
    const double vol,
    const double jmpIntens = 0.0,
    const double jmpAverage = 0.0,
    const double jmpStd = 0.0)
{
    //  Create IVS
    MertonIVS ivs(spot, vol, jmpIntens, jmpAverage, jmpStd);

    //  Go
    return dupireCalib(ivs, inclSpots, maxDs, inclTimes, maxDt);
}
```

It takes around 50 milliseconds to calibrate a local volatility grid of 30 spots between 50 and 200 and 60 times between now and 5 years, to a Merton IVS with spot 100, volatility 15, jump intensity 5, mean −15 and standard deviation 10. With 150 spots and 260 times, it takes 400 ms. Calibration is embarrassingly parallel and trivially multi-threadable across maturities. This is left as an exercise.

We can easily test the quality of the calibration. Initialize Dupire's model with the result of the calibration. Price a set of European options of different strikes and maturities (developed as a single product with multiple payoffs on page 238) by simulation in this model, and compare with Merton's price as implemented in the *merton*() function in analytics.h in closed-form. In our tests, Dupire and Merton prices match within a couple of basis points over a wide range of strikes and maturities (with 500,000 paths, weekly time steps, where a parallel Sobol pricing of 20 European calls with maturities up to three years takes 400 milliseconds).

Risk views

The process calibration + simulation produces the value of a transaction *out of market-implied volatilities*. Its differentials are sensitivities to market-traded variables, more relevant for trading and hedging than sensitivities to model parameters:

$$\begin{tikzcd}\hat{\sigma}(K,T) \arrow[r, "\text{calibration}", "G"'] & \sigma(S,t) \arrow[r, "\text{simulation}", "H"'] & V_0\end{tikzcd}$$

Model parameters are obtained from market variables with a prior calibration step. Model sensitivities are obtained with AAD as explained and developed in Chapter 12. We can therefore obtain the market sensitivities by *check-pointing the model sensitivities into calibration*.

We developed, in the previous chapter, functionality to obtain the *micro-bucket* $\frac{\partial V_0}{\partial \sigma(S,t)}$ in constant time. We check-point this result into calibration to obtain $\frac{\partial V_0}{\partial \hat{\sigma}(K,T)}$, what Dupire calls a *superbucket*.

We are missing one piece of functionality: our IVS $\hat{\sigma}(K,T)$ is defined in derived IVS classes, from a set of parameters, which nature depends on the concrete IVS. For instance, the Merton IVS is parameterized with a continuous volatility, jump intensity, and the mean and standard deviation of jumps. The desired derivatives are not to the parameters of the concrete IVS, but to a discrete set of implied Black and Scholes market-implied volatilities, *irrespective* of how these volatilities are produced or interpolated.

To achieve this result, we are going to use a neat technique that professional financial system developers typically apply in this situation: we are going to define a *risk surface*:

$$s(K, T)$$

such that if we denote $\hat{\sigma}(K, T)$ the implied volatilities given by the concrete IVS, our calculations will not use these original implied volatilities, but implied volatilities *shifted by the risk surface*:

$$\sum(K, T) = \hat{\sigma}(K, T) + s(K, T)$$

Further, we interpolate the risk surface $s(K, T)$ from a discrete set of knots:

$$s_{ij} = s(K_i, T_j)$$

that we call the *risk view*. All the knots are set to 0, so:

$$\sum(K, T) = \hat{\sigma}(K, T)$$

so the results of all calculations remain evidently unchanged by shifting implied volatilities by zero, but in terms of risk, we get:

$$\frac{\partial}{\sigma(K, T)} = \frac{\partial}{\partial s(K, T)}$$

The risk view does not affect the value, and its derivatives exactly correspond to derivatives to implied volatilities, irrespective of how these implied volatilities are computed.

We compute sensitivities to implied volatilities as sensitivities to the risk view:

$$\frac{\partial V_0}{\partial s_{ij}}$$

Risk views apply to bumping as well as AAD and are extremely useful, in many contexts, to aggregate risks over selected market instruments.

In the context of Dupire's model, we apply a risk view over an IVS fed to Dupire's formula. Dupire's formula depends on the first- and second-order derivatives of call prices, so the risk view must be differentiable. (Bi-)linear interpolation is not an option. We must implement a smooth interpolation.

A vast amount of smooth interpolations exist in literature, but what we need is a *localized* one, otherwise the resulting risk spills over the volatility surface. For these reasons, we implement a well-known, simple, localized and efficient smooth interpolation algorithm called *smoothstep*, presented in many places, including Wikipedia's "Smoothstep" article. Like linear interpolation, smoothstep interpolation finds x_i such that $x_i < x_0 \leq x_{i+1}$, and, unlike linear interpolation, which returns:

$$y_0 = y_i + (y_{i+1} - y_i)t$$

where $t = (x_0 - x_i)/(x_{i+1} - x_i)$, smoothstep returns:

$$y_0 = y_i + (y_{i+1} - y_i)t^2(3 - 2t)$$

Practically, we upgrade the *interp*() function of Chapter 6 to implement either linear or smoothstep interpolation. We also produce a two-dimensional variant:

```
//  interp.h

#include <algorithm>
using namespace std;

//  Utility for interpolation
//  Interpolates the vector y against knots x in value x0
//  Interpolation is linear or smooth, extrapolation is flat
template <bool smoothStep=false, class ITX, class ITY, class T>
inline auto interp(
    //  sorted on xs
    ITX                           xBegin,
    ITX                           xEnd,
    //  corresponding ys
    ITY                           yBegin,
    ITY                           yEnd,
    //  interpolate for point x0
    const T&                      x0)
    ->remove_reference_t<decltype(*yBegin)>
{
    //  STL binary search, returns iterator on 1st no less than x0
    //  upper_bound guarantees logarithmic search
    auto it = upper_bound(xBegin, xEnd, x0);

    //  Extrapolation?
    if (it == xEnd) return *(yEnd - 1);
    if (it == xBegin) return *yBegin;

    //  Interpolation
    size_t n = distance(xBegin, it) - 1;
    auto x1 = xBegin[n];
    auto y1 = yBegin[n];
    auto x2 = xBegin[n + 1];
    auto y2 = yBegin[n + 1];

    auto t = (x0 - x1) / (x2 - x1);

    //  Note constexpr if
    if constexpr (smoothStep)
    {
        //  smoothstep
        return y1 + (y2 - y1) * t * t * (3.0 - 2 * t);
    }

    else
    {
        //  linear
        return y1 + (y2 - y1) * t;
    }
}

//  2D
template <bool smoothStep=false, class T, class U, class V, class W, class X>
inline V interp2D(
    //  sorted on xs
    const vector<T>&            x,
    //  sorted on ys
    const vector<U>&            y,
    //  zs in a matrix
    const matrix<V>&            z,
    //  interpolate for point (x0,y0)
    const W&                    x0,
    const X&                    y0)
{
    const size_t n = x.size();
    const size_t m = y.size();

    //  STL binary search, returns iterator on 1st no less than x0
    //  upper_boung guarantees logarithmic search
    auto it = upper_bound(x.begin(), x.end(), x0);
    const size_t n2 = distance(x.begin(), it);

    //  Extrapolation in x?
    if (n2 == n)
        return interp<smoothStep>(
            y.begin(),
            y.end(),
            z[n2 - 1],
            z[n2 - 1] + m,
            y0);
    if (n2 == 0)
        return interp<smoothStep>(
            y.begin(),
            y.end(),
            z[0],
            z[0] + m,
            y0);

    //  Interpolation in x
    const size_t n1 = n2 - 1;
    auto x1 = x[n1];
    auto x2 = x[n2];
    auto z1 = interp<smoothStep>(
                    y.begin(),
                    y.end(),
                    z[n1],
                    z[n1] + m,
                    y0);
    auto z2 = interp<smoothStep>(
                    y.begin(),
                    y.end(),
                    z[n2],
                    z[n2] + m,
                    y0);

    auto t = (x0 - x1) / (x2 - x1);
    if constexpr (smoothStep)
    {
        //  Smooth step
        return z1 + (z2 - z1) * t * t * (3.0 - 2 * t);
    }
    else
    {
        //  linear
        return z1 + (z2 - z1) * t;;
    }
}
```

Armed with smooth interpolation, we can define the RiskView object in ivs.h. Note that (contrarily to the IVS), the risk view is templated since we will be computing derivatives to its knots, and we want to do that with AAD:

```
//  ivs.h

#include "interp.h"

//  Risk view
template <class T>
class RiskView
{
    bool                myEmpty;

    vector<double>  myStrikes;
    vector<Time>    myMats;
    matrix<T>       mySpreads;

public:

    //  Default constructor, empty view
    RiskView() : myEmpty(true) {}

    //  Intializes risk view AND put on tape
    //  Sets all spreads to 0
    RiskView(const vector<double>& strikes, const vector<Time>& mats) :
        myEmpty(false),
        myStrikes(strikes),
        myMats(mats),
        mySpreads(strikes.size(), mats.size())
    {
        for (auto& spr : mySpreads) spr = T(0.0);
    }

    //  Get spread
    T spread(const double strike, const Time mat) const
    {
        return myEmpty
            ? T(0.0)
            : interp2D<true>(myStrikes, myMats, mySpreads, strike, mat);
    }

    //  Accessors by const ref
    bool empty() const { return myEmpty; }
    size_t rows() const { return myStrikes.size(); }
    size_t cols() const { return myMats.size(); }
    const vector<double>& strikes() const { return myStrikes; }
    const vector<Time>& mats() const { return myMats; }
    const matrix<T>& risks() const { return mySpreads; }

    //  Iterators
    typedef typename matrix<T>::iterator iterator;
    typedef typename matrix<T>::const_iterator const_iterator;
    iterator begin() { return mySpreads.begin(); }
    iterator end() { return mySpreads.end(); }
    const_iterator begin() const { return mySpreads.begin(); }
    const_iterator end() const { return mySpreads.end(); }

    //  For bump risk
    void bump(const size_t i, const size_t j, const double bumpBy)
    {
        mySpreads[i][j] += bumpBy;
    }
};
```

This code should be self-explanatory; the risk view is nothing more than a two-dimensional interpolation object with convenient accessors and iterators and knots set to zero. The method *bump*() modifies one knot by a small amount *bumpBy*, from zero to *bumpBy*, so we can apply bump risk.

The next step is to effectively incorporate the risk view in the calculation. We extend the methods *call*() and *localVol*() on the base IVS so they may be called with a risk view.

```
//  IVS base class
//  ...

    //  Call price
    template<class T = double>
    T call(
        const double strike,
        const Time mat,
        const RiskView<T>* risk = nullptr) const
    {
        //  blackScholes is defined in analytics.h, templated
        return blackScholes<T>(
            mySpot,
            strike,
            impliedVol(strike, mat)
                + (risk ? risk->spread(strike, mat) : T(0.0)),
            mat);
    }

    //  Local vol, dupire's formula
    template<class T = double>
    T localVol(
        const double strike,
        const double mat,
        const RiskView<T>* risk = nullptr) const
    {
        //  Derivative to time
        const T c00 = call(strike, mat, risk);
        const T c01 = call(strike, mat - 1.0e-04, risk);
        const T c02 = call(strike, mat + 1.0e-04, risk);
        const T ct = (c02 - c01) * 0.5e04;

        //  Second derivative to strike = density
        const T c10 = call(strike - 1.0e-04, mat, risk);
        const T c20 = call(strike + 1.0e-04, mat, risk);
        const T ckk = (c10 + c20 - 2.0 * c00) * 1.0e08;

        //  Dupire's formula
        return sqrt(2.0 * ct / ckk) / strike;
    }

//  ...
```

The modification is minor. A shift, interpolated from the risk view, is added to the implied volatility for the computation of call prices, hence local volatilities. Since the risk view is set to zero, this doesn't modify results, but it does produce risk with respect to the risk view's knots.

Finally, we apply the same minor modification to the calibration functions in McMdlDupire.h so they accept an optional risk view:

```
//  mcMdlDupire.h

//  Calibrates one maturity
//  Main calibration function below
template <class IT, class OT, class T = double>
inline void dupireCalibMaturity(
    //  IVS we calibrate to
    const IVS& ivs,
    //  Maturity to calibrate
    const Time maturity,
    //  Spots for local vol
    IT spotsBegin,
    IT spotsEnd,
    //  Results, by spot
    //  With (random access) iterator, STL style
    OT  lVolsBegin,
    //  Risk view
    const RiskView<T>& riskView = RiskView<double>())
{
    //  Number of spots
    IT spots = spotsBegin;
    const size_t nSpots = distance(spotsBegin, spotsEnd);

    //  Estimate ATM so we cut the grid 2 stdevs away to avoid instabilities
    const double atmCall = double(ivs.call(ivs.spot(), maturity));
    //  Standard deviation, approx. atm call * sqrt(2pi)
    const double std = atmCall * 2.506628274631;

    //  Skip spots below and above 2.5 std
    int il = 0;
    while (il < nSpots && spots[il] < ivs.spot() - 2.5 * std) ++il;
    int ih = nSpots - 1;
    while (ih >= 0 && spots[ih] > ivs.spot() + 2.5 * std) --ih;

    //  Loop on spots
    for (int i = il; i <= ih; ++i)
    {
        //  Dupire's formula
        lVolsBegin[i] = ivs.localVol(spots[i], maturity, &riskView);
    }

    //  Extrapolate flat outside std
    for (int i = 0; i < il; ++i)
        lVolsBegin[i] = lVolsBegin[il];
    for (int i = ih + 1; i < nSpots; ++i)
        lVolsBegin[i] = lVolsBegin[ih];
}

#define ONE_HOUR 0.000114469

//  Returns a struct with spots, times and lVols
template<class T = double>
inline auto dupireCalib(
        //  The IVS we calibrate to
        const IVS& ivs,
        //  The local vol grid
        //  The spots to include
        const vector<double>& inclSpots,
        //  Maximum space between spots
        const double maxDs,
        //  The times to include, note NOT 0
        const vector<Time>& inclTimes,
        //  Maximum space between times
        const double maxDt,
        //  Risk view if required
        //  omitted: T = double , no risk view
        const RiskView<T>& riskView = RiskView<double>())
{
    //  Results
    struct
    {
        vector<double> spots;
        vector<Time> times;
        matrix<T> lVols;
    } results;

    //  Spots and times
    results.spots = fillData(inclSpots, maxDs, 0.01); //  min space = 0.01
    results.times = fillData(inclTimes, maxDt,
        ONE_HOUR,                //  min space = 1 hour
        &maxDt, &maxDt + 1       //  dirty trick to include maxDt
    );

    //  Allocate local vols, transposed maturity first
    matrix<T> lVolsT(results.times.size(), results.spots.size());

    //  Maturity by maturity
    const size_t n = results.times.size();
    for (size_t j = 0; j < n; ++j)
    {
        dupireCalibMaturity(
            ivs,
            results.times[j],
            results.spots.begin(),
            results.spots.end(),
            lVolsT[j],
            riskView);
    }

    //  transpose is defined in matrix.h
    results.lVols = transpose(lVolsT);

    return results;
}
```

Superbuckets

We now have all the pieces to compute superbuckets by check-pointing. We build a higher-level function in main.h that executes the steps of our check-pointing algorithm:

```
struct SuperbucketResults
{
    double value;
    double delta;
    vector<double> strikes;
    vector<Time> mats;
    matrix<double> vega;
};

//  Returns value, delta, strikes, maturities
//      and vega = derivatives to implied vols = superbucket
inline auto
    dupireSuperbucket(
    //  Model parameters that are not calibrated
    const double            spot,
    const double            maxDt,
    //  Product
    const string&           productId,
    const map<string, double>&   notionals,
    //  The local vol grid
    //  The spots to include
    const vector<double>&   inclSpots,
    //  Maximum space between spots
    const double            maxDs,
    //  The times to include, note NOT 0
    const vector<Time>&     inclTimes,
    //  Maximum space between times
    const double            maxDtVol,
    //  The IVS we calibrate to
    //  Risk view
    const vector<double>&   strikes,
    const vector<Time>&     mats,
    //  Merton params
    const double            vol,
    const double            jmpIntens,
    const double            jmpAverage,
    const double            jmpStd,
    //  Numerical parameters
    const NumericalParam&   num)
{
    //  Results
    SuperbucketResults results;

    //  ...
```

The first check-pointing step is compute local volatilities by calibration (after initialization of the tape) and store the calibrated model in memory.

```
    //  ...

    //  Start with a clean tape
    auto* tape = Number::tape;
    tape->rewind();

    //  Calibrate the model
    auto params = dupireCalib(
        inclSpots,
        maxDs,
        inclTimes,
        maxDtVol,
        spot,
        vol,
        jmpIntens,
        jmpAverage,
        jmpStd);
    const vector<double>& spots = params.spots;
    const vector<Time>& times = params.times;
    const matrix<double>& lvols = params.lVols;

    //  Put in memory
    putDupire(spot, spots, times, lvols, maxDt, "superbucket");

    //  ...
```

Next, we compute the microbucket:

```
    //  ...

    //  Find delta and microbucket
    auto mdlDerivs = dupireAADRisk(
        "superbucket",
        productId,
        notionals,
        num);
    results.value = mdlDerivs.value;
    results.delta = mdlDerivs.delta;
    const matrix<double>& microbucket = mdlDerivs.vega;

    //  ...
```

where *dupireAADRisk*() is essentially a wrapper around *AADrisk-Aggregate*() from the previous chapter, with an interface specific to Dupire that returns a delta and a microbucket matrix:

```
//  Returns a struct with price, delta and vega matrix
inline auto dupireAADRisk(
    //  model id
    const string&           modelId,
    //  product id
    const string&           productId,
    const map<string, double>&   notionals,
    //  numerical parameters
    const NumericalParam&   num)
{
    //  Check that the model is a Dupire
    const Model<Number>* model = getModel<Number>(modelId);
    if (!model)
    {
        throw runtime_error("dupireAADRisk() : Model not found");
    }
    const Dupire<Number>* dupire
        = dynamic_cast<const Dupire<Number>*>(model);
    if (!dupire)
    {
        throw runtime_error("dupireAADRisk() : Model not a Dupire");
    }

    //  Results
    struct
    {
        double value;
        double delta;
        matrix<double> vega;
    } results;

    //  Go
    auto simulResults = AADriskAggregate(
                            modelId,
                            productId,
                            notionals,
                            num);

    //  Find results

    //  Value
    results.value = simulResults.riskPayoffValue;

    //  Delta

    results.delta = simulResults.risks[0];

    //  Vegas

    results.vega.resize(dupire->spots().size(), dupire->times().size());
    copy(
        next(simulResults.risks.begin()),
        simulResults.risks.end(),
        results.vega.begin());

    return results;
}
```

The next part is the crucial one for the check-pointing algorithm: we clean the tape, build the risk view (which puts it on tape), and conduct calibration again, this time in instrumented mode:

```
    //  dupireSuperbucket()
    //  ...

    //  Clear tape
    //  tape->rewind();
    tape->clear();

    //  Convert market inputs to numbers, put on tape

    //  Create IVS
    MertonIVS ivs(spot, vol, jmpIntens, jmpAverage, jmpStd);

    //  Risk view --> that is the AAD input
    //  Note: that puts the view on tape
    RiskView<Number> riskView(strikes, mats);

    //  Calibrate again, in AAD mode, make tape
    auto nParams = dupireCalib(
        ivs,
        inclSpots,
        maxDs,
        inclTimes,
        maxDtVol,
        riskView);
    matrix<Number>& nLvols = nParams.lVols;

    //  ...
```

We seed the calibration tape with the microbucket obtained earlier:

```
    //  ...

    //  Seed local vol adjoints on tape with microbucket results
    for (size_t i = 0; i < microbucket.rows(); ++i)
    {
        for (size_t j = 0; j < microbucket.cols(); ++j)
        {
            nLvols[i][j].adjoint() = microbucket[i][j];
        }
    }

    //  ...
```

and propagate back to the risk view:

```
    //  ...

    //  Propagate
    Number::propagateAdjoints(prev(tape->end()), tape->begin());

    //  ...
```

This completes the computation. We pick the desired derivatives as the adjoints of the knots in the risk view, clean the tape, and return the results:

```
    //  ...

    //  Results: superbucket = risk view

    //  Copy results
    results.strikes = strikes;
    results.mats = mats;
    results.vega.resize(riskView.rows(), riskView.cols());
    transform(riskView.begin(), riskView.end(), results.vega.begin(),
        [](const Number& n)
    {
        return n.adjoint();
    });

    //  Clear tape
    tape->clear();

    //  Return results
    return results;
}
```

We illustrated two powerful and general techniques for the production of financial risk sensitivities: the risk view, which allows to aggregate risks over instruments selected by the user, irrespective of calibration or the definition of the market; and check-pointing, which separates the differentiation of simulations from the differentiation of calibration, so that the differentiation of simulations may be performed with path-wise AAD, with limited RAM consumption, in parallel and in constant time.

Finally, note that we left delta unchanged through calibration. We are returning the Dupire delta, the sensitivity to the spot with *local* volatility unchanged. With the superbucket information, we could easily adjust delta for any smile dynamics assumption (sticky, sliding, . . .). There is no strong consensus within the trading and research community as to what the "correct" delta is, but most convincing research points to the Dupire delta as the correct delta within a wide range of models; see, for instance, [99].

Finite difference superbucket risk

As a reference, and in order to test the results of the check-pointing algorithm, we implement a superbucket bump risk, where we differentiate, with finite differences, the whole process calibration + simulation in a trivial manner in main.h:

```
//  Returns value, delta, strikes, maturities
//      and vega = derivatives to implied vols = superbucket
inline auto
    dupireSuperbucketBump(
        //  Model parameters that are not calibrated
        const double             spot,
        const double             maxDt,
        //  Product
        const string&            productId,
        const map<string, double>&   notionals,
        //  The local vol grid
        //  The spots to include
        const vector<double>&    inclSpots,
        //  Maximum space between spots
        const double             maxDs,
        //  The times to include, note NOT 0
        const vector<Time>&      inclTimes,
        //  Maximum space between times
        const double             maxDtVol,
        //  The IVS we calibrate to
        //  Risk view
        const vector<double>&    strikes,
        const vector<Time>&      mats,
        //  Merton params
        const double             vol,
        const double             jmpIntens,
        const double             jmpAverage,
        const double             jmpStd,
        //  Numerical parameters
        const NumericalParam&    num)
{
    //  Results
    SuperbucketResults results;

    //  Calibrate the model
    auto params = dupireCalib(
        inclSpots,
        maxDs,
        inclTimes,
        maxDtVol,
        spot,
        vol,
        jmpIntens,
        jmpAverage,
        jmpStd);
    const vector<double>& spots = params.spots;
    const vector<Time>& times = params.times;
    const matrix<double>& lvols = params.lVols;

    //  Create model
    Dupire<double> model(spot, spots, times, lvols, maxDt);

    //  Get product
    const Product<double>* product = getProduct<double>(productId);

    //  Base price
    auto baseVals = value(model, *product, num);

    //  Vector of notionals
    const vector<string>& allPayoffs = baseVals.identifiers;
    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(
                "dupireSuperbucketBump() : payoff not found");
        }
        vnots[distance(allPayoffs.begin(), it)] = notional.second;
    }

    //  Base book value
    results.value = inner_product(
                        vnots.begin(),
                        vnots.end(),
                        baseVals.values.begin(),
                        0.0);

    //  Create IVS
    MertonIVS ivs(spot, vol, jmpIntens, jmpAverage, jmpStd);

    //  Create risk view
    RiskView<double> riskView(strikes, mats);

    //  Bumps

    //  bump, recalibrate, reset model, reprice, pick value, unbump

    //  Delta

    //  Recreate model
    Dupire<double> bumpedModel(
                        spot + 1.0e-08,
                        spots,
                        times,
                        lvols,
                        maxDt);
    //  Reprice
    auto bumpedVals = value(bumpedModel, *product, num);
    //  Pick results and differentiate
    results.delta = (
        inner_product(
            vnots.begin(),
            vnots.end(),
            bumpedVals.values.begin(),
            0.0)
        - results.value) * 1.0e+08;

    //  Vega

    const size_t n = riskView.rows(), m = riskView.cols();
    results.vega.resize(n, m);
    for (size_t i = 0; i < n; ++i) for (size_t j = 0; j < m; ++j)
    {
        //  Bump
        riskView.bump(i, j, 1.0e-05);
        //  Recalibrate
        auto bumpedCalib = dupireCalib(
                            ivs,
                            inclSpots,
                            maxDs,
                            inclTimes,
                            maxDtVol,
                            riskView);
        //  Recreate model
        Dupire<double> bumpedModel(
                            spot,
                            bumpedCalib.spots,
                            bumpedCalib.times,
                            bumpedCalib.lVols,
                            maxDt);
        //  Reprice
        auto bumpedVals = value(bumpedModel, *product, num);
        //  Pick results and differentiate
        results.vega[i][j] = (
            inner_product(
                    vnots.begin(),
                    vnots.end(),
                    bumpedVals.values.begin(),
                    0.0)
            - results.value) * 1.0e+05;
        //  Unbump
        riskView.bump(i, j, -1.0e-05);
    }

    //  Copy results and strikes
    results.strikes = strikes;
    results.mats = mats;

    //  Return results
    return results;
}
```

Results

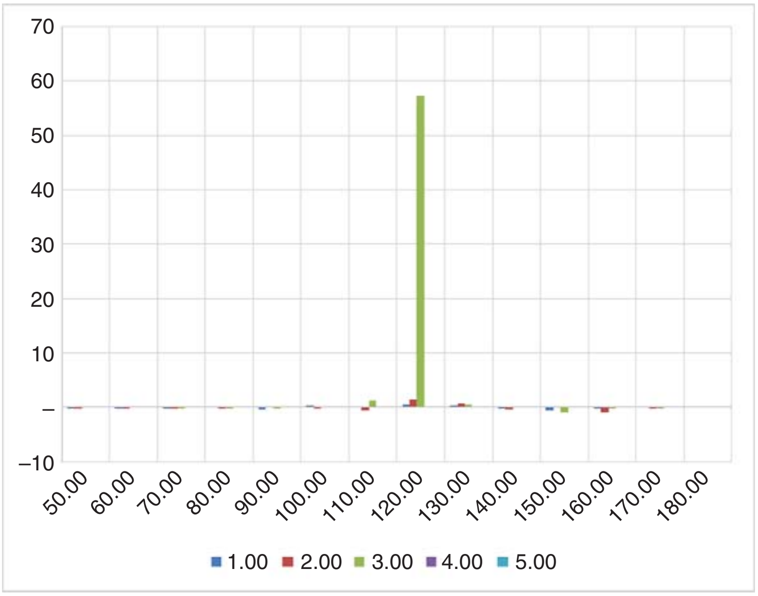
We start with the superbucket of a European call of maturity 3 years, strike 120, over a risk view with 14 knot strikes, every 10 points between 50 and 180, and 5 maturities every year between 1y and 5y.

We define the European option market as a Merton market with volatility 15, jump intensity 5, mean jump −15 and jump standard deviation 10. We calibrate a local volatility matrix with 150 spots between 50 and 200, and 60 times between now and 5 years.[6] We simulate with 300,000 Sobol points in parallel over 312 (biweekly) times steps.

The Merton price is 4.25. Dupire's price is off two basis points at 4.23. The corresponding Black and Scholes implied volatility is 15.35. The Black and Scholes vega is 59. We should expect a superbucket with 59 on the strike 120, maturity 3 years, and zero everywhere else. The superbucket is obtained in around two seconds. Almost all of it is simulation. Calibration and check-pointing are virtually free.

With the improvements of Chapter 15, the superbucket is produced in one second.

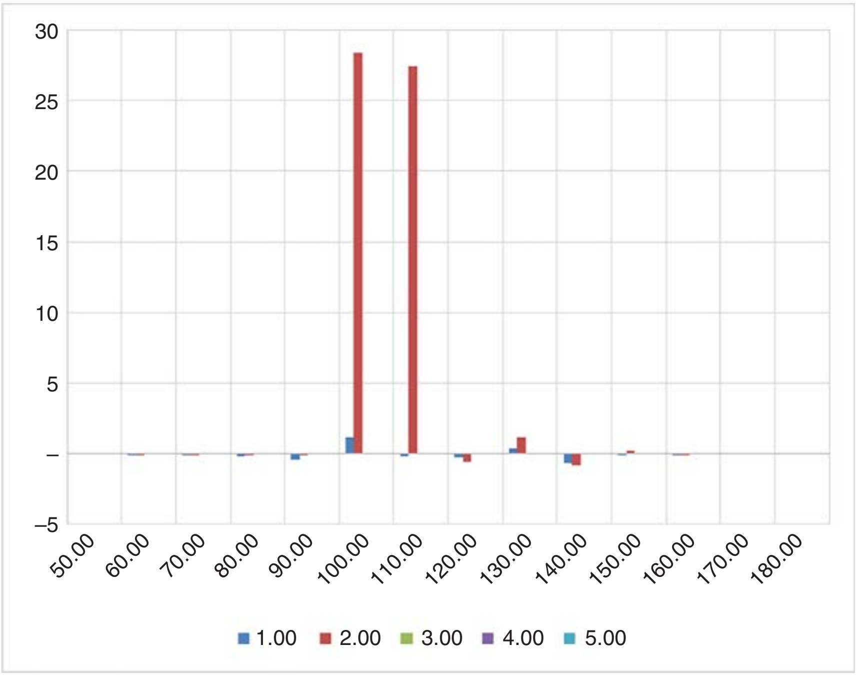
The resulting superbucket is displayed on the chart below.

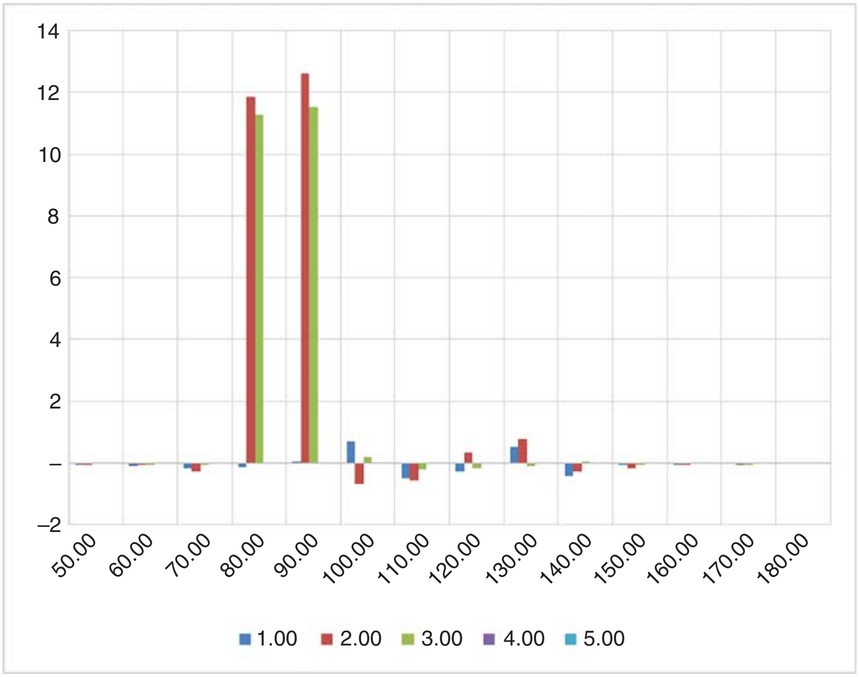


[6]To obtain a stable superbucket takes a fine-grained microbucket and simulation timeline, many paths, and a sparse risk view.

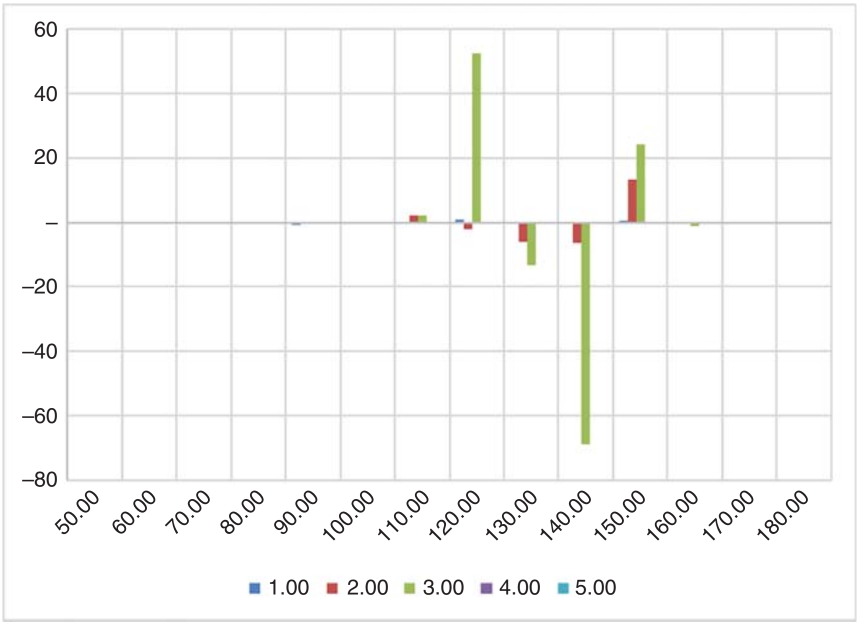
This is a good-quality superbucket, especially given it was obtained with simulations in just over a second. Superbuckets traditionally obtained with FDM are of similar quality and also take around a second to compute. Once again, we notice that AAD and parallelism bring FDM performance to Monte-Carlo simulations.

With the same settings (10 points spacing on the risk view), we compute the superbucket for a 2 years 105 call (first chart) and a 2.5 years 85 call (second chart). The results are displayed below. The calculation is proportionally faster for lower maturities, linearly in the total number of time steps. We see that vega is correctly interpolated over the risk view (with limited spilling that eventually disappears as we increase the number of paths and time steps).





Finally, with a maturity of 3 years, strike 120, and a (biweekly monitored) barrier of 150 (with a barrier smoothing of 1), we obtain the following. The calculation time is virtually unchanged, the barrier monitoring cost being essentially negligible.



This barrier superbucket has the typical, expected shape for an up-and-out call: positive vega concentrated at maturity on the strike, negative vega, also concentrated at maturity (with some spilling over the preceding maturity on the risk view from the interpolation), *below* the barrier, partly unwound by (perhaps counterintuitive but a systematic observation nonetheless) *positive* vega on the barrier.

Comparing with a bump risk for performance and correctness, we find that finite differences produce a very similar risk report in 45 around seconds. For a 3y maturity, it could be reduced to 30 seconds by only bumping active volatilities. We are computing "only" 42 risk sensitivities (14 strikes and 3 maturities up to 3y on the risk view), so AAD acceleration is less impressive here: times 30, probably down to times 20 with a smarter implementation of the bump risk.

However, in this particular case, AAD risk is also much more stable. The results of the bumped superbucket depend on the size of the bumps and the spacing of the local volatility and the risk view, in an unstable, explosive manner. It frequently produces results in the thousands in random cells where vega is expected in the tens. AAD superbuckets are resilient and stable, because derivatives are computed analytically, without ever actually changing the initial conditions.

Finally, the quality of the superbucket is dependent on how sparse is the risk view, and rapidly deteriorates when more strikes and maturities are added to it. This is a problem with superbuckets known in the industry: to obtain decent superbuckets over a thinly spaced risk view forces to increase the time steps at the expense of speed. It helps to implement simulation schemes more sophisticated than Euler's.