Computation Graphs for AAD and Machine Learning Part III: Application to Derivatives Risk Sensitivities

Article *in* Wilmott · March 2020

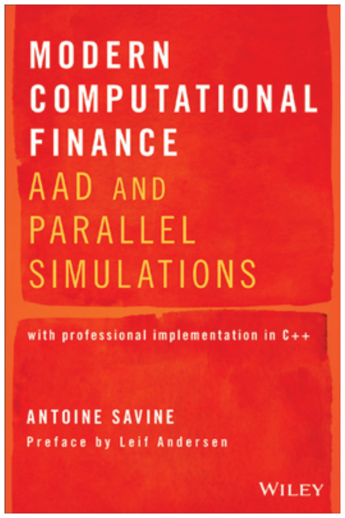
DOI: 10.1002/wilm.10830

Antoine Savine
DANSKE BANK
20 PUBLICATIONS 11 CITATIONS

Antoine Savine

Computation Graphs for AAD and Machine Learning Part III: Application to Derivatives Risk Sensitivities

Third in a series of three articles with code, exploring the notion of computation graph, with words, mathematics and code, and application in Machine Learning and finance to compute a vast number of derivative sensitivities with spectacular speed and accuracy.


MODERN COMPUTATIONAL FINANCE
AAD AND PARALLEL SIMULATIONS
with professional implementation in C++
ANTOINE SAVINE
Preface by Leif Andersen
WILEY

Abstract

The first two parts of this series of three introduced the main ideas of Adjoint Differentiation (AD) and its automatic implementation (Automatic AD or AAD) in a framework able to differentiate arbitrary calculation code in constant time (independent of the number of differentials). The framework we built in part II applies *operator overloading* to record the sequence of operations in a given calculation, while the calculation runs, on a data structure called *tape*, effectively building the computation graph in memory. After the tape is recorded in memory, all differentials are obtained with a single traversal, in the reverse order through the tape, by adjoint back-propagation.

The implementation code of part II was written to help the exposition and illustrate the concepts and ideas, and in the interest of clarity, we implemented AAD in its most simple form, at the expense of efficiency. This code is not meant to scale to more general, more complicated real-world problems like large Monte-Carlo simulations. We referred readers to the book [25] and its companion code `https://github.com/asavine/CompFinance` for a complete, professional, scalable and fully optimized implementation.

In this third and final part, we nonetheless apply the simple framework to the computation of risk sensitivities in the context of Monte-Carlo simulations. We write and differentiate simple code for the pricing of barrier options in Dupire's [8] local volatility model, implemented with Monte-Carlo.[1] Like the rest of the code in this series, our Monte-Carlo implementation is written in the simplest possible terms, for the purpose of demonstration of AAD over Monte-Carlo. Hence, we discuss the application of AAD in a context somewhat similar to real-world production, but we do so with an extremely simplified implementation.

We will see that even our simplistic implementation of AAD bears rather spectacular results, computing over a thousand sensitivities to *local volatilities*[2] in a few seconds, with 131,072 paths over 156 steps on a single CPU, a time similar to around 4 prices.[3]

The code is available on `https://github.com/asavine/WilmottArticle`. It has been tested to compile and run on Windows with Visual Studio 2017. We also included an xll wrapper to run it in 32bit Excel (see [7] or the tutorial `https://github.com/asavine/xlCppTutorial` for exporting C++ code to Excel) together with a demonstration spreadsheet.

1 A simple Monte-Carlo pricer

1.1 Dupire's local volatility model

We first develop a simple Monte-Carlo pricer, which we differentiate next. The pricer implements Dupire's local volatility model [8], a simple extension of the classic Black and Scholes model, where the spot price follows a one-factor diffusion with volatility dependent on spot and time:

$$\frac{dS_t}{S_t} = \sigma(S_t, t)\, dW_t$$

Dupire's triumph was to demonstrate that this model is consistent with the current prices of European calls $C(K,T)$ of all strikes K and maturities T if and only if the local volatility function satisfies Dupire's celebrated formula:

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

where subscripts denote partial derivatives.[4]

We don't worry about any of this here. In our simplistic implementation, we consider that a matrix of local volatilities is given, along with spot and time labels. In the Monte-Carlo simulations, volatility is bi-linearly interpolated in spot and time from this matrix:

```
vector < double >    spotLabels ;
vector < double >    timeLabels ;
matrix < double >    localVols ;
```

Here, the *matrix* class is a simple adapter over a vector with a matrix view, written in the source file *matrix.h*. This class stores data in a vector, with row major layout. A *matrix* also stores its number of rows and columns, which allows to view the data in matrix form. It offers a convenient interface where the element in row *r*, column *c* is accessed for read and write with the syntax *matrix[r][c]*. A *matrix* also exposes methods to access the number of *rows()* and *cols()*, and STL-like iterators *begin()* and *end()* for iterating over data, row major. Readers should easily make sense of the code in *matrix.h*. All the details are explained in chapters 1 and 2 of [25].

For interpolation, we apply the functions written in the file *interp.h*. This file contains functions to interpolate one and two-dimensional data, either linearly or with *smooth step interpolation*. We only use the 2D linear interpolator here. The (rather mundane) implementation is explained in the section 6.4 (1D linear) and 13.2 (1D and 2D, linear and smooth step) of [25]. We use it as follows for the interpolation of volatility:

```
# include  " interp . h "
//   ...
currentVol  =  interp_2d (
    spotLabels ,      //   x  labels
    timeLabels ,      //   y  labels
    localVols ,       //   z  values  for  xs  in  rows  and  ys  in  columns
    currentSpot ,     //   interpolate  across  xs  in  x0
    currentTime );    //   interpolate  across  ys  in  y0
```

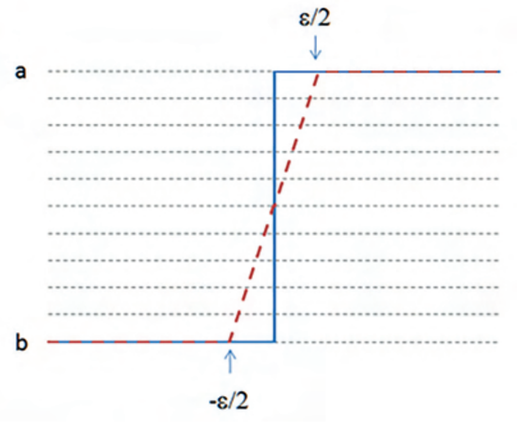
1.2 Barrier options

Barrier options are a rather simple variety of *exotic* derivatives. Barriers deliver at maturity the payoff of European options, but only if a specified barrier was never breached, at any prior time. Barrier options are actively traded in interbank equity and foreign exchange derivatives markets. We work with the example of a so-called *up and out call* (UOC), also called *reverse knock-out or RKO* on foreign exchange markets. RKOs are European calls subject to an *up and out* barrier. Formally, their payoff is of the form:

$$(S_T - K)^+ 1_{\{max(S_t,\, 0 \le t \le T) < B\}}$$

Barrier options are generally continuous, meaning that they die if the barrier is breached at *any* time. This causes difficulties with Monte-Carlo implementations, where paths are simulated in discrete time. A variety of methods have been developed to resolve the problem, see, for instance, the classic [11], but we don't worry about this in our simple implementation. Instead, we only consider *discretely monitored barrier options*, where the barrier is only active on a discrete schedule, for example daily or weekly fixings.

The payoff of barrier options is (obviously) discontinuous, which causes unreliable risk sensitivities with Monte-Carlo. Discontinuous functions cannot be differentiated and the resulting risk sensitivities are unstable (with bumping) to downright wrong (with AAD). This is a well-known problem, and traders always *smooth* discontinuous payoffs of digital or barrier options, or other kinds of discontinuous exotics like the popular auto-callable notes. Smoothing consists in the approximation of the discontinuous profile with a close, continuous one. For example, digital options are typically represented as tight call spreads.



Barrier options, on the other hand, are typically represented as *smooth barriers*. Instead of knocking out the entire transaction when the spot price fixes above the barrier on a monitoring date, we only knock out a fraction of the notional, depending on where the spot price S_t fixes relative to the barrier B and a smoothing factor ϵ. If $S_t < B - \epsilon/2$, nothing knocks out. If $S_t > B + \epsilon/2$, the whole remaining notional knocks out. When the spot fixes in a region of size ϵ around the barrier, a part $(S_t - B + \epsilon/2)/\epsilon$ knocks out and the rest of the notional survives. The final payoff of the European call is delivered on the surviving notional at maturity.

Smoothing transforms discontinuous profiles into continuous ones, where risk sensitivities may be computed more reliably with Monte-Carlo. At the same time, the continuous approximation is close enough to the true (discontinuous) profile so that pricing remains virtually unbiased. The smoothing factor ϵ controls the trade-off. A small ϵ more accurately approximates the true profile, while a large ϵ makes the profile 'more continuous' and easier to differentiate. For digitals and barriers on single name equities or forex, it is common practice to set ϵ to 1 to 5% of the underlying asset price. For avoidance of doubt, it is common practice to use true profiles for pricing, and smoothed profiles for risk management.

Smoothing of digitals or barriers can be seen as particular cases of the more general idea to evaluate payoffs with *fuzzy logic*, which also establishes a general algorithm for the systematic smoothing of all exotic instruments, see [23]. We keep things simple in our toy implementation, but we do implement a smooth barrier so we can differentiate Monte-Carlo simulations.

1.3 Monte-Carlo simulations

Monte-Carlo is, conceptually, probably the simplest possible numerical algorithm. To estimate the expectation of a complicated function(al) of a stochastic process (in finance, the price of a derivative transaction is the expectation of its payoff, itself a function(al) of the path of the underlying asset price), randomly sample a large number of paths and approximate the expectation by the average payoff over the simulated scenarios. The Monte-Carlo estimate converges towards the true expectation by the Strong Law of Large Numbers as the number m of paths grows, the standard error vanishing in $1\sqrt{m}$ by the Central Limit Theorem. Furthermore, the average across simulated paths, of the differentials of the payoff to the model parameters (also called *path-wise derivatives*) also converge to the true risk sensitivities (which are the differentials of the true expectation) when mild regularity conditions are met (which is generally the case, provided prior smoothing of discontinuities). Finally,

Model and financial instrument logics must be properly encapsulated, and the communication protocol between models and instruments must be developed with special care

Monte-Carlo is applicable in virtually any context,[5] whereas faster methods, including closed-form formulas, analytic approximations, numerical integration and FDM are only viable in limited situations.

This conceptual simplicity makes it particularly easy to develop a self-contained implementation like the local volatility pricer we develop here for barrier options. A professional implementation is much more involved, because it must be general and efficient.

In a professional context, it is not acceptable to develop self contained pricers for different combinations of models and financial instruments. This would inevitably result in mistakes, inconsistencies and a maintenance nightmare. Simulation logic must be developed once and reused for multiple models and derivatives transactions. Model and financial instrument logics must be properly encapsulated, and the communication protocol between models and instruments must be developed with special care. This is achieved with best practice design patterns (see the classic [17] or the part II of the more recent [25]) and technologies like cash-flow scripting (see [24], [2] and the upcoming [4]).

In addition, the simulation of Monte-Carlo paths is time consuming and the convergence of the algorithm is particularly slow. For this reason, an *efficient* implementation of Monte-Carlo is a central concern in modern finance, including effective random sampling (see [16] for a discussion of Sobol numbers, now considered best practice in computational finance, in large part due to this work), various variance reduction techniques (see the classic [11]) and parallelism (see [25] for a complete implementation on CPU and [5] for a discussion of an implementation on GPU).

It is not the purpose of this article to cover the professional, efficient implementation of Monte-Carlo simulations. We only want to demonstrate how risk sensitivities are computed in constant time with AAD in the context of Monte-Carlo. We develop a self contained pricing function for barrier options in Dupire's local volatility model and focus on the simplicity of the code, somewhat at the expense of efficiency. The code presents evident optimization opportunities, which we leave for interested readers to identify and correct as an exercise.[6]

Monte-Carlo practically simulates paths in two steps. First, sample independent random numbers from a given distribution (generally, a standard Gaussian) and then, consume the random numbers to construct a path of the underlying stochastic process. This construction is called a *simulation scheme*. The simplest, and by far most widely used simulation scheme is *Euler's scheme*.

Recall the local volatility model is a one-factor diffusion:

$$\frac{dS_t}{S_t} = \sigma(S_t,t)dW_t \Longleftrightarrow d\log S_t = -\frac{\sigma(S_t,t)^2}{2}dt + \sigma(S_t,t)\,dW_t$$

and we simulate paths for the underlying asset price on a discrete set of simulation dates, corresponding to the payment date at maturity and the barrier monitoring dates.[7] Say, we have n simulation dates (in addition to today's T_0) $T_1, T_2, ..., T_n$. In order to generate a path, we first draw n independent standard Gaussian numbers $G_1, G_2, ..., G_n$. We call *Monte-Carlo dimension* the number of random numbers we need to generate one path, n in this case. Euler's scheme consumes these numbers to build a path with the intuitive construction:

$$\log S_{T_{i+1}} = \log S_{T_i} - \frac{\sigma(S_{T_i},T_i)^2}{2}(T_{i+1} - T_i) + \sigma(S_{T_i},T_i)\sqrt{T_{i+1} - T_i}\,G_{i+1}$$

Euler's scheme is well behaved, with convergence properties subject to mild technical conditions on the local volatility function σ. Other schemes exist, generally more complex, with convergence benefits depending on the context (see [11] or [3] for a recent and original take). Our code implements Euler's scheme.

1.4 Random number generators

The generation of a Monte-Carlo path consumes a number n of random numbers, relying on Random Number Generators (or RNGs) for their production. The base *RNG* class in the file *random.h* establishes a unified API for RNGs.

```
class  RNG
{
public :

    //  Initialise  with  dimension  simDim
    virtual  void  init ( const  size_t  simDim )  =  0;

    //  Compute  the  next  vector [ simDim ]  of  independent  Uniforms  or  Gaussians
    //  The  vector  is  filled  by  the  function  and  must  be  pre - allocated
    virtual  void  nextU ( vector < double >&  uVec )  =  0;
    virtual  void  nextG ( vector < double >&  gaussVec )  =  0;

    virtual  unique_ptr < RNG >  clone ()  const  =  0;

    virtual  ~ RNG ()  {}

    //  Skip  ahead
    virtual  void  skipTo ( const  unsigned  b )  =  0;
 };
```

The five methods *init()*, *nextU()/nextG()*, *clone()* and *skipTo()* state the contract that concrete RNGs commit to fulfill. Their implementation is down to concrete RNGs. Their intent is as follows:

- **void init(const size_t simDim)** initializes the RNG for the generation of random vectors of size *simDim*. *simDim* is the Monte-Carlo dimension, the number of random numbers consumed for the generation of one Monte-Carlo path. In our example, this is the number n of time steps.
- **void nextU(vector<double>& uVec) / void nextG(vector<double>& gaussVec)** populate a vector given as argument with the next *simDim* random numbers, drawn either from a uniform (*nextU*) or Gaussian (*nextG*) distribution. The vector must be pre-allocated. Internally, RNGs are expected to generate Gaussian numbers by application of the inverse Normal Cumulative Distribution Function (CDF) to uniform numbers. Moro's classic approximation of the inverse Normal CDF [21] is given in the file *gaussians.h*. Importantly, these functions not only produce the next random vector, but also modify the internal state of the RNG to produce the *following* random vector when one of these functions is called next. Therefore, successive random vectors are obtained by sequential calls to these functions. Besides, these functions are not *const* since they modify the RNG's internal state. This is an important consideration for a parallel implementation.
- **unique_ptr<RNG> clone() const** creates an exact duplicate of the RNG and returns a unique base class pointer on the location of the copy in memory. Implementations are expected to follow the so-called *virtual copy constructor* idiom.[8]
- **void skipTo(const unsigned b)** places the RNG in the state where the next random vector obtained by call to *nextU()* or *nextG()* corresponds to the vector number b in the sequence. This method makes it possible to generate the random vector number b without having actually generated the $b - 1$ first random vectors. The implementation is down to concrete RNGs. Most RNGs permit a skip ahead of logarithmic complexity, meaning that the time to advance the RNG to vector number b is proportional to $\log b$. An efficient skip ahead is a key ingredient of a parallel implementation.

RNGs are not all born equal. Properties of random numbers affect in a major way the convergence of Monte-Carlo, hence, the number of simulations and the time it takes to achieve a desired accuracy. It is therefore not advisable to reuse standard library RNGs, but instead, carefully implement specific RNGs best suited for

>

Monte-Carlo simulations. Although the implementation of RNGs is out of the scope of this article, we provide an implementation of the two most common RNGs in finance, L'Ecuyer's MRG32K3A [18] in *mrg32k3a.h* and Sobol sequences in *sobol.h* and *sobol.cpp*. We refer to the chapters 5, 6 and 7 of [25] for a complete explanation of these codes.

1.5 Pricing code

Bringing it all together, we obtain the simplistic pricing code below, from the file *dupireBarrier.h*. This code is templated on the real number type, which is necessary for the application of AAD, as we have seen in parts I and II of this series. The function returns the average payoff over a batch of simulations, numbered from firstPath to *lastPath*. Simulating Monte-Carlo paths in batches of limited size bears many benefits, one of which is facilitate memory management for AAD as we will see shortly. The code should be self explanatory with the comments and the extensive explanations above.

```
template  < class  T >
inline  T  dupireBarrierMCBatch (
    //   Spot
    const  T             S0 ,
    //   Local  volatility
    const  vector < T >&    spots ,
    const  vector < T >&    times ,
    const  matrix < T >&    vols ,
    //   Product  parameters
     const  T             maturity ,
     const  T             strike ,
     const  T             barrier ,
     //   First  and  last  path
     const  int           firstPath ,
     const  int        lastPath ,
     //  Time  steps
     const  int           Nt ,
     //   Smoothing
     const  T             epsilon ,
     //   Random  number  generator
     RNG &                random )
{
     //   Initialize
     T  result  =  0;
     vector < double >  gaussianIncrements ( Nt );
     //  double  because  the  RNG  is  not  templated
     //   ( and  correctly  so ,  see  chapter  12)

     //  Set  RNG  state  to  the  first  path  in  the  batch
     random . skipTo ( firstPath );

     //   Loop  over  paths
     const  T  dt  =  maturity  /  Nt ,  sdt  =  sqrt ( dt );
     for  ( int  i  =  firstPath ;  i  <  lastPath ;  ++ i )
     {
         //   Generate  Nt  Gaussian  Numbers
         random . nextG ( gaussianIncrements );

         //  Inntialize  path
```

```
        T  spot  =  S0 ,  time  =  0;
        notionalAlive  =  1.0;

        //   Step  by  step
        for  ( size_t  j  =  0;  j  <  Nt ;  ++ j )
        {
            //   Interpolate  volatility
            const  T  vol  =  interp2D ( spots ,  times ,  vols ,  spot ,  time );

            //   Simulate  return
            spot  *=  exp (-0.5  *  vol  *  vol  *  dt
                +  vol  *  sdt  *  gaussianIncrements [ j ]);

              //  Increase  time
            time  +=  dt ;

            //   Monitor  barrier
            if  ( spot  >  barrier  +  epsilon )        //   definitely  dead
            {
                notionalAlive  =  0.0;
                break ;
            }
            else  if  ( spot  <  barrier  -  epsilon )   //   definitely  alive
            {
                /*  do  nothing  */
            }
            else                                //   in  between ,  interpolate
            {
                notionalAlive  *=
                    1.0  -  ( spot  -  barrier  +  epsilon )  /  (2  *  epsilon );
            }
        }

        //   Payoff ,  paid  on  surviving  notional
        if  ( spot  >  strike )  result  +=  notionalAlive  *  ( spot  -  strike );
    }

    return  result  /  ( lastPath  -  firstPath );
}
```

The higher level pricing function below, also from the file *dupireBarrier.h*, provides an entry point for pricing. Note that it does nothing but call *dupireBarrierMCBatch()* repeatedly, and that it is not templated on the real number type. The reason is that we will call this higher level function only for pricing, not for risk sensitivities, for reasons explained shortly.

```
inline  double  dupireBarrierPricer (
    //   Spot
    const  double        S0 ,
    //   Local  volatility
    const  vector < double >&    spots ,
```

>

```
  const  vector < double >&    times ,
  const  matrix < double >&    vols ,
  //   Product  parameters
  const  double              maturity ,
  const  double              strike ,
  const  double              barrier ,
  //   Number  of  paths
  const  int           Np ,
  // Number  of  simulations  in  every  batch
  const  int           Nb ,
  // Time  steps
  const  int           Nt ,
  //   Smoothing
  const  double              epsilon ,
  //   Random  number  generator
  RNG &              random )
{
 random . init ( Nt );
 double  result  =  0.0;
 int  firstPath  =  0;

 while  ( firstPath  <  Np )
 {
    int  lastPath  =  firstPath  +  Nb ;
    lastPath  =  min ( lastPath ,  Np );
    double  batchPrice  =  dupireBarrierMCBatch (
      S0 ,
      spots ,
      times ,
      vols ,
      maturity ,
      strike ,
      barrier ,
      firstPath ,
      lastPath ,
      Nt ,
      epsilon ,
      random );
    result  +=  batchPrice  *  ( lastPath - firstPath )  /  Np ;
 }

 return  result ;
}
```

We also wrote an xll wrapper in the file *xlExport.cpp*. Building the project *wArticle.vcxproj* on the 32bit platform in Visual Studio 2017 or above, builds the xll *wArticle.xll*, a pre-built version of which is also provided in the GitHub repo.[9] After opening the xll in Excel 32bit, you may check that the function *xDupireBarrierPricer* is available, along with Excel's native functions, and test it either on your own, or with the provided demonstration spreadsheet *xlTest.xlsx*.

The demonstration sheet runs the pricer with a matrix of 1,860 local volatilities (31 spots from 50 to 200 every 5 by 60 times from 0 to 5 years every month) to price a 3 years' up-and-out call with weekly monitoring, hence 156 time steps. With 131,072 paths (16,384 batches of 8 paths each) and Sobol numbers, it takes around 1.5 seconds to compute a price on a recent workstation.[10] This is far from optimal. Speed could be doubled at the expense of some complexity in the code. Most of the time is spent interpolating volatility in spot and time on every path and every time step. We could instead pre-interpolate in time and interpolate in spot only during simulations, resulting in a significant speed-up without affecting results. We could also pre-interpolate in spot over an even grid, so we don't need binary searches during simulations. This is left as an exercise for interested readers, who will find answers in the section 6.4 of [25]. Note that the most significant acceleration is obtained from processing paths in parallel, on CPU or GPU, see the 7 of [25] for details and code.

2 Adjoint Differentiation of the Monte-Carlo pricer

2.1 Implementation

Now we have a pricing function, we want to compute risk sensitivities. In the terms of the demonstration spreadsheet, we want to produce the 1,860 differentials of the price to local volatilities (microbucket/local vegas), and its differential to the spot (delta). This is 1,861 differentials. To be fair, we have a barrier option with 3 years expiry, sensitive to local volatilities only up to 3 years. So we really have 1,116 local volatilities to care about, for a total of 1,117 differentials. With 1.5 seconds for one price, conventional bumping would take around half an hour to produce a complete risk report. We are going to do it in just 6 seconds (1.5 seconds in parallel).

Recall from the second part of this series, that the simple framework we built permits to differentiate arbitrary calculation code, instrumented by templating the code on the real number type. We have our pricing function *dupireBarrierPricer()*, which takes a matrix of local volatilities and a spot as parameters (among others) and returns a price. It would seem that we could differentiate it with AAD just as easily as we differentiated the Black and Scholes pricing function in part II.

Unfortunately, it doesn't work this way with Monte-Carlo. We have a little additional work before we see the magic results. Recall that adjoint differentiation requires the entire computation graph in memory to back-propagate derivatives. This is impractical to downright impossible with hundreds of thousands of Monte-Carlo paths. Empirical and theoretical estimates of memory footprint point to around 5GB of tape storage per second of calculation on one core. In our example where a price takes 1.5 seconds, this is a tape of 7.5GB. Excel 32bits limits xll memory to around 1GB, so we could not possibly store the tape in memory. Even if we could, to back-propagate through a 7.5GB large tape would be slow (slow enough to defeat the benefits of AAD) because only small parts would fit in the CPU cache, causing constant moves of slices of the tape between RAM and cache.

The solution is simple with Monte-Carlo simulations. Compute differentials batch by batch and average in the end. The differentials of the average are the averages of the differentials. With batches of, say, 8 paths, we only store the computation of 8 paths in memory, this is around 0.1ms, or 0.5MB, which easily fits in a modern CPU cache. We can repeatedly compute batch-wise derivatives and wipe the tape between batches. We could also differentiate batches of one path, something called *path-wise differentiation*. We found it best to differentiate in small batches, to dilute the overhead of additional work per batch. We found a batch size of 8 to produce satisfactory results in most cases.[11]

Batch-wise (or path-wise) differentiation of Monte-Carlo is a particular case of a general technique called *check-pointing*. Check-pointing reduces the memory footprint of AAD by differentiating a long computation one slice at a time, wiping the tape between slices and stitching differentials together in the end. Check-pointing can be seen as a back-propagation through a meta-graph, where nodes are not elementary operations, but slices of the calculation. Applications in finance include the differentiation of multi-dimensional FDM, one time step at a time, and the differentiation of calibration, resulting in the so-called *market risks*, risk sensitivities to market variables (like implied volatilities), by opposition to *model risks*, risk sensitivities to model parameters (like local volatilities). Check-pointing and its applications are discussed in the chapter 13 of [25].

Differentiation code is listed below. The function *dupireBarrierRisks()* is also implemented in the file *dupireBarrier.h* and exported to Excel. The implementation is similar to the pricing function *dupireBarrierPricer()*, in that it repeatedly calls *dupireBarrierMCBatch()*, but with arguments instantiated with the *Number* type, causing batch-wise pricing to be recorded on tape. We subsequently differentiate the computation by back-propagation, as usual, wipe the tape between batches, and average differentials in the end.

```
inline  void  dupireBarrierRisks (
    //   Spot
    const  double        S0 ,
    //   Local  volatility
    const  vector < double >&    spots ,
    const  vector < double >&    times ,
    const  matrix < double >&    vols ,
    //   Product  parameters
    const  double             maturity ,
    const  double             strike ,
    const  double             barrier ,
    //   Number  of  paths
```

>

```
    const  int          Np ,
    //  Number  of  simulations  in  every  batch
    const  int          Nb ,
    //  Time  steps
    const  int          Nt ,
    //   Smoothing
    const  double             epsilon ,
    //   Random  number  generator
    RNG &             random ,
    //  Results
    double &            price ,
    double &            delta ,
    matrix < double >&       vegas )
{
    //  Allocate  and  initialize  results
    price  =  0.0;
    delta  =  0.0;
    vegas . resize ( spots . size (),  times . size ());
    for  ( auto &  vega  :  vegas )  vega  =  0.0;

    //  Temporary  storage  for  batch  risks
    double  batchPrice ,  batchDelta ;
    matrix < double >  batchVegas ( vegas . rows (),  vegas . cols ());

    //  Temporary  storage  for  parameters  as  Number  types
    Number  nS0 ,  nMaturity ,  nStrike ,  nBarrier ,  nEpsilon ;
    vector < Number >  nSpots ( spots . size ()),  nTimes ( times . size ());
    matrix < Number >  nVols ( vols . rows (),  vols . cols ());

    //  Initialize  the  RNG
    random . init ( Nt );

    //  Loop  over  batches
    int  firstPath  =  0;
    while  ( firstPath  <  Np )
    {
      int  lastPath  =  firstPath  +  Nb ;
      lastPath  =  min ( lastPath ,  Np );

      //  Wipe  the  tape
      tape . clear ();

      //  Put  parameters  on  tape  by  initialization  of  Number  types
      //  Note  this  is  inefficient  in  many  ways  but  it  keeps  things  simple
      //  and  doesn ’ t  matter  too  much  with  batch - wise  differentiation
      nS0  =  S0 ;
      nMaturity  =  maturity ;
      nStrike  =  strike ;
      nBarrier  =  barrier ;
    nEpsilon  =  epsilon ;
    copy ( spots . begin (),  spots . end (),  nSpots . begin ());
    copy ( times . begin (),  times . end (),  nTimes . begin ());
    copy ( vols . begin (),  vols . end (),  nVols . begin ());

    //  Compute  the  batch
    Number  nBatchPrice  =  dupireBarrierMCBatch (
      nS0 ,
      nSpots ,
      nTimes ,
      nVols ,
      nMaturity ,
      nStrike ,
      nBarrier ,
      firstPath ,
      lastPath ,
      Nt ,
      nEpsilon ,
      random );

    //  Back - propagate  derivatives
    vector < double >  adjoints  =  calculateAdjoints ( nBatchPrice );

    //  Pick  results
    batchPrice  =  nBatchPrice . value ;
    batchDelta  =  adjoints [ nS0 . idx ];
    transform ( nVols . begin (),  nVols . end (),  batchVegas . begin (),
      [&]( const  Number &  vol )  {  return  adjoints [ vol . idx ];  });

    //  Accumulate
    int  paths  =  lastPath  -  firstPath ;
    double  w  =  double ( paths )  /  Np ;
    price  +=  batchPrice  *  w ;
    delta  +=  batchDelta  *  w ;
    transform ( vegas . begin (),  vegas . end (),  batchVegas . begin (),  vegas . begin (),
      [&]( const  double &  vega ,  const  double &  batchVega )
      {  return  vega  +  batchVega  *  w ;  });

     //  Next  batch
     firstPath  =  lastPath ;
  }
}
```

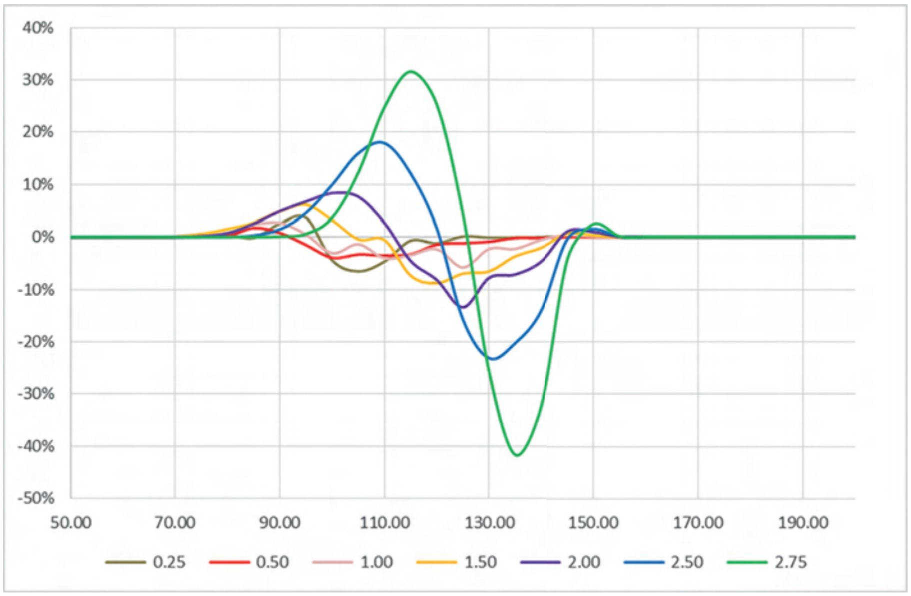
Note that there is some inefficiency in this design, in terms of unnecessary work performed repeatedly for every batch. For instance, because the tape is cleared between batches, we must put the model parameters on tape again for every batch. What we could do instead is *rewind* the tape between batches, so that the computation is wiped but the parameters remain and accumulate differentials across batches. This, and other important optimizations, require an AAD framework significantly more complex than the simplistic implementation of this series. Part III of [25] explains in deep detail the implementation of all these optimizations, resulting in a significant speed-up of the computation of a complete risk report. Numbers are given in the next section.

>

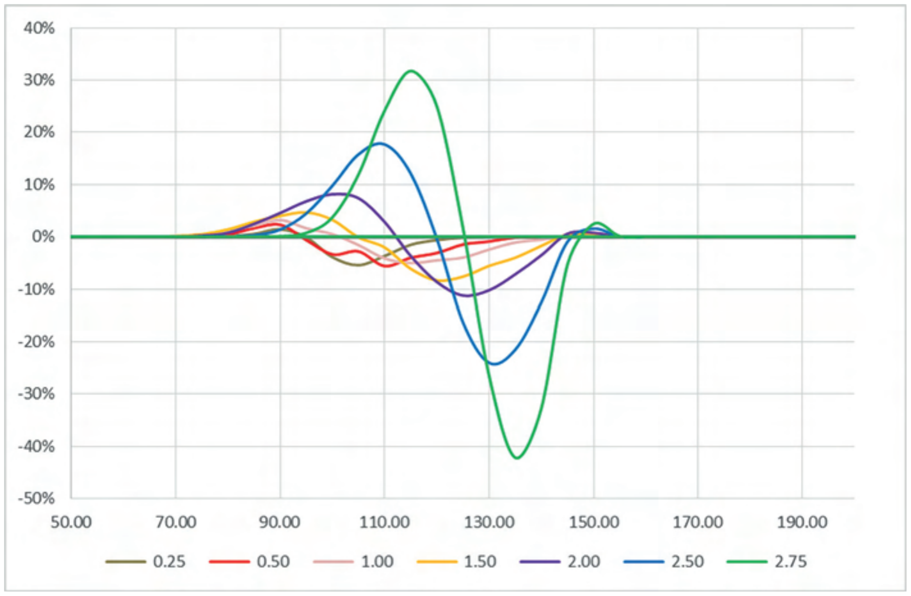
2.2 Results

Our simplistic code computes the risk report in around 6.5 seconds, just over 4 times one price to compute 1,117 differentials, a speed-up by a factor over 250 compared to bumping! This is a remarkable result. Real-time production of complex risk reports was unthinkable before AAD. What is perhaps even more remarkable is that we managed to achieve this speed with such a simplistic implementation.

Results match bumping to many decimals, as readers may check for themselves, bumping local volatilities manually on the spreadsheet. The chart below displays some local vegas, obtained with a smoothing factor of 5%.



Those results are just decent, as seen in the wiggly shapes of local vegas of near maturities. This is because 131,072 paths are not sufficient to obtain a correct risk report, even with Sobol sampling. With 655,360 paths, we obtain the much better looking report below:



We still see small wiggles around mid-maturity, which eventually vanish with a few million paths. Of course, with 5 times the number of paths, the risk report now takes half a minute (more precisely, 32 seconds) to compute, which is still kind of real-time, but just so.

It is also interesting to compare results and speed with the professional, fully optimized implementation of [25]. We reproduced the exact same results, but in only 1.15 seconds on a single core for 131,072 paths and around 6 seconds for 655,360 paths. The collective benefit of the many optimizations developed in [25] results in a speed-up of around times 5-6 for differentiation. The production of the risk report feels almost instantaneous. This is very significant, and demonstrates the point of investing in a professional, optimized implementation despite the 250x speed-up offered by even a simplistic implementation of AAD. Besides, the implementation of [25] scales to more general, larger problems like the risk sensitivities of the Counterparty Value Adjustment (CVA) of a large book of complex trades. It also scales much better in parallel.

2.3 Parallel implementation

Besides an optimized implementation of AAD, the most significant speed-up is obtained by the distribution of Monte-Carlo paths across multiple CPUs for parallel processing. We could do it with *distributed memory parallelism*, whereby different batches are processed simultaneously on different processes, potentially located on different machines, and results are communicated by messaging protocols and aggregated. Standard C++ does not support distributed parallelism, so third party software and platform specific libraries are necessary. Another option is *shared memory parallelism*, where different batches are processed *concurrently* on the different CPU cores of a given machine, and share memory. Concurrency is a light-weight form of parallelism, and it is fully supported by the standard C++ threading library. It is therefore a preferred option for the implementation of parallel algorithms, with the caveat that shared memory parallelism must be developed with extreme care, or different threads may end up simultaneously using the same resources, causing subtle and hard to debug flaws known as *race conditions.*[12] See [29] for a complete reference on C++ concurrency. Finally, *GPU* or *TPU* parallelism offers a massive speed-up, in return for significant development costs, where the entire implementation must be refactored and rewritten in a specialized language like CUDA, see [27] for a discussion.

Although we will not discuss the details of a parallel implementation of Monte-Carlo pricing and risks in this article, we do provide parallel code in the functions *dupireBarrierPricerMT()* and *dupireBarrierRisksMT()*, located in the file *dupireBarrier.h*. You may run them from Excel by setting the 'parallel' flag to 'True' on the Excel functions. The flag is also implemented on the demonstration spreadsheet. We implemented a particularly simple form of concurrency, setting the outer loop over batches to run in parallel with an OpenMP directive,[13] and making batch-wise copies of mutable objects and working memory to avoid races. The only modification on the AAD framework itself is to mark the global tape with the *thread_local* keyword, so that every thread works with its own copy of the tape, and concurrent back-propagations don't interfere. The code is simple enough and extensively commented, so that readers should figure it out easily. The chapters 7 and 12 of [25] discuss the parallel implementation of pricing and differentiation in deep detail, with optimized code.

New implementations of AAD are pushing the limits of its efficiency, while quantitative analysts are leveraging them in unexpected ways, besides the evident application to risk sensitivities or calibration

Our parallel implementation of pricing produces the exact same results as the serial implementation, in just 0.15 seconds for 131,072 paths and 0.75 seconds for 655,360 paths, a speed-up by factor of around 10 on an octo-core workstation, close than 120% parallel efficiency! This is due to *hyper-threading*, a technology implemented on Intel's modern chips, known to boost parallel efficiency by up to 20-25% depending on the situation.

The performance of our parallel implementation of risks is much more nuanced. We still produce the exact same risks as before, in 1.1 seconds for 131,172 paths and 5.8 seconds for 655,360 paths. This is around 5.75x parallel speed-up on an octo-core machine, around 70% parallel efficiency. It has been stated that an implementation of AAD in parallel is easy: just mark the tape to be *thread_local*. This not quite true. An efficient parallel implementation of AAD is hard, and involves things like custom memory management and custom thread pools. The implementation explained in [25] comes close to 100% efficiency, causing the efficiency lag of our simplistic implementation to widen in the parallel context. The professional parallel implementation produces the risk reports in 0.17 seconds for 131,172 paths and 0.75 seconds for 655,360 paths, faster by a factor around 7.

Conclusion: the future of AAD

AAD took quantitative finance by storm, following the ground-breaking work of [10], winner of the prestigious Quant of the Year 2007 Risk award; and the large scale implementation in production pioneered by Superfly Analytics at Danske Bank, which was followed by a public demonstration of the results at Global Derivatives 2014, and earned the In-House System of the Year 2015 Risk award. Today, AAD is widely considered one of the most powerful algorithms in quantitative finance and every decent derivatives system, either in-house or from external vendors, includes at least some flavour of it. Hopefully, this series of three dedicated articles succeeded to demonstrate the power of AAD, and at the same time, demystify this technology and explain exactly how it works in words, mathematics and simple C++ code.

>

Today, AAD is evolving in new and exciting ways, in finance and elsewhere. New implementations of AAD are pushing the limits of its efficiency, while quantitative analysts are leveraging them in unexpected ways, besides the evident application to risk sensitivities or calibration.

This series of articles presented an implementation of AAD based on run-time operator overloading. This may have been cutting-edge just five years ago, but today it is considered a 'conventional implementation'. Major progress was recently made and new, considerably more efficient implementations, have been produced. The main milestones are listed below.

- In 2014, Hogan [14] proposed to partly resolve computation graphs at compile-time with a technique known as *expression templates*. Besides delegating part of the work to compile-time, expression templates may substantially reduce the size of tapes, resulting in a lower memory footprint and a more efficient back-propagation. For this reason, the application of expression templates is sometimes called 'tape compression' [28]. Expression templates have been demonstrated to substantially accelerate differentiation in cases of practical relevance, in return for a (much) more sophisticated implementation in code, expression templates being a notoriously difficult C++ construct. Expression templates are now considered best practice and implemented in production in the best Derivatives systems. AAD with expression templates is implemented and explained in deep detail in the chapter 15 of [25].
- In 2017, Naumann presented the concept of a fully compile-time, tape-less AAD, and demonstrated its suitability for implementation on GPU, opening a new world of possibilities, despite some limitations and constraints [19].
- In September 2019, Goloubentsev and Lakshtanov published in Wilmott [12] a radical approach to AAD, implemented in their commercial software *Matlogica*, where, instead of building a tape, they *compile* evaluation and differentiation functions at run-time. Demonstrations point to another leap in speed and efficiency compared to all known implementations, but this is brand new technology, yet to be battle tested, and its implementation requires specialized, platform dependent compiler development skills.
- Finally, Google implemented its own flavour of AAD in TensorFlow and substantially improved it in its recent 2.0 release. TensorFlow's AAD is tensor-based, meaning that it records and back-propagates tensor operations, not scalar operations, and works with vectorized code that transforms tensors into tensors, by opposition to classic, scalar code that transforms numbers into numbers. To this extent, TensorFlow's implementation of AAD is unusual: it cannot efficiently differentiate arbitrary code, its main interface is in Python, and it is heavily geared towards machine learning applications. On the other hand, it runs natively on GPU with spectacular speed. In its current form, TensorFlow is not suitable for the implementation of general purpose calculations, like financial risks, but it has all the right building blocks of the next generation AAD engine.

While increasingly sophisticated frameworks are pushing the boundaries of the speed and efficiency of AAD, the quantitative finance community is realizing that differentials contain a vast amount of useful information, and that the availability of a large number of accurate, cheap to compute differentials may be leveraged in many ways. The natural application of fast and accurate differentials is in the production of risk sensitivities, as explained in this article. Another evident application is fitting model parameters, like training a machine learning model to learn its weights from a training set, or calibrating the parameters of a financial model to market data. In both cases, fitting generally involves an iterative optimization, where most of the time is spent computing a vast amount of differentials on every iteration. By computing differentials an order of magnitude faster, AAD provides a massive speed-up for model fitting. Deep learning is basically based on back-propagation, a form of AAD, and this is how deep neural networks learn their weights in reasonable time. Similarly, AAD massively accelerates the calibration of financial models to market data. In particular, the application of AAD to the real-time construction of interest rate surfaces was demonstrated in the pioneering work of Scavenius in 2012 [26]. In the context of dynamic models, where calibration is generally about finding a set of parameters that makes the model hit the price of some European instruments of known market value, AAD, again, massively accelerates iterative optimization and permits calibration by Monte-Carlo in reasonable time. With bumping, calibration by Monte-Carlo is too slow, forcing practitioners to implement less accurate calibration by analytic approximation instead, see for instance [1]. The application of AAD to calibration is discussed in [13].

Even more exciting is recent research where differentials, computed with AAD, are applied to resolve other problems than calibration or risks. Among other applications, differentials have been applied to resolve the persisting problem of the efficient computation of future values of Derivatives transactions and trading books in various scenarios. This is arguably the most significant problem of modern finance, because:

1. The knowledge of the values and risks of trading books, not only today, but also in the future, depending on the scenario, is priceless information for a better understanding and hedging of exotic risks.
2. The computation of these dynamic risks constitutes the main bottleneck for regulations like XVA, CCR, SIMM-MVA, FRTB, etc.

The following recent research leveraged in various ways the differentials produced with AAD to resolve the problem of future values and risks, in different contexts and from different angles.

- In 2015, Reghai and al. published in Risk an innovative take [22], where future values are reconstructed as path integrals, by virtue of the Martingale Representation Theorem, the integrand being expressed in terms of differentials, computed with AAD.
- In 2018, Andreasen presented on QuantMinds a new approach to Value at Risk (VAR) and Expected Loss (EL), including for FRTB, where the near future value of a trading book is approximated in different scenarios with a Taylor expansion based on AAD differentials. The expansion is only used to order scenarios by increasing value of the trading book. In the relevant scenarios, as identified by ordering, the trading book is fully and accurately evaluated. Hence, the expensive complete evaluation only happens in a low number of relevant scenarios, which effectively resolves the computational bottleneck.
- Finally, Huge and Savine presented on QuantMinds and RiskMinds 2019, a method based on neural networks, trained on simulated data, for the estimation of pricing and risk as functions of a future state [15]. The main innovation is that those networks learn, not only from payoffs, but also from path-wise derivatives (the gradient of the payoff to the state, obtained by AAD). It is demonstrated that training with path-wise differentials massively improves the ability of neural networks to learn accurate pricing, order-

ing and risk functions for complex trading books, from a reasonable amount of simulated data, quickly, in a stable manner and without manual tinkering. In fact, it is the path-wise differentials that enable networks to learn meaningful prices for real-world Derivatives books, textbook deep learning failing to learn from a realistically sized data-set.

As a last example, Andreasen implemented, in Danske Bank's Superfly risk management platform,[14] the differentials of XVA and capital charges to *the notionals of all the trades in a given netting set*. These differentials are valuable information. First, they directly provide the marginal contribution of every transaction to the XVA charge, allowing to distribute it across trading desks. In addition, they make it possible to add to the netting set a dummy collection of standard trades with zero notional, letting the differentials to these notionals tell traders what possible transactions with the counterparty increase or reduce the XVA charge. All of these are obtained virtually for free with AAD.

About the Author
Antoine Savine is a French mathematician and risk management practitioner with Superfly Analytics in Danske Bank. He has held multiple leading roles in the derivatives industry in the past 20 years, and presently also teaches Volatility and Computational Finance at Copenhagen University.

>

ENDNOTES

1. The prices of barrier options in Dupire's model can be computed (much) more efficiently with finite difference methods (FDM), but the (much) less efficient Monte-Carlo algorithm applies to a (much) wider range of financial models and instruments, including XVA and capital regulations. This is why the efficient implementation of Monte-Carlo is a central problem of modern computational finance. This article focuses on the efficient *differentiation* of Monte-Carlo, although we happen to be working with a particularly simple example where FDM offers a viable and far superior alternative.
2. What Bruno dupire calls a *microbucket*, by opposition to a *superbucket*, which is the collection of sensitivities to *implied* volatilities. We don't cover superbuckets here, and refer interested readers to the chapter 13 of [25].
3. Although the full risk report may be produced *in a fraction of a second* on a standard workstation with the more efficient, parallel implementation of [25].
4. Dupire later extended his result to stochastic volatility [9] and demonstrated that a general model is consistent with the current prices of European calls if and only if:

$$E\left[\sigma_t^2 \,|\, S_t = x\right] = \frac{2C_T(x,t)}{C_{KK}(x,t)}$$

where the right hand side corresponds to a *conditional forward variance contract*, which can be locked with a trading strategy on European calls. For an overview, see the talk https://www.youtube.com/watch?v=-YiAMx-jOKHg recorded in Rio for Dupire's 60th birthday in November 2018. The slides of the talk may be downloaded from https://www.slideshare.net/AntoineSavine.
5. For exotics with early exercises, a prior LSM pass is necessary to first turn early exercises into path dependencies, see founding papers [20] and [6].
6. Answers are found in [25], which introduces a scalable, modular and efficient approach to Monte-Carlo simulations.
7. In principle, we should also include additional dates to 'fill' the simulation schedule and avoid large gaps between simulation dates, which may cause large errors in Euler's simulation scheme. In our simple implementation, we just assume that the monitoring schedule is tight enough.
8. See how some concrete RNGs implement those concepts in *mrg32k3a.h* and *sobol.h*.
9. To run the xll on a machine without an installation of Visual Studio, it may be necessary to install the redistributable files *VC_redist.x64.exe* and *VC_redist.x86.exe*, also provided in the repo for convenience.
10. We have observed that timings vary significantly depending on the chip and architecture of the workstation. This is especially true of AAD, and even more so of its parallel implementation. AAD is memory intensive and its performance highly depends on memory bandwidth, cache size etc. All the timings exposed in this article were measured on an octo-core iMac Pro, running Windows 10, obviously.
11. The amount of additional work performed for every batch depends on the implementation. Our simplistic implementation performs significant administrative work per batch, therefore differentiation works better batch-wise than path-wise. With a more sophisticated implementation, additional work may be reduced to virtually zero, in which case it is best to differentiate path-wise to reduce the size of tapes as much as possible. The optimal batch size also depends on the size of the problem. A single barrier option in Dupire's model is a small problem, which we differentiate in batches of 8 paths. For a larger problem, like a Counterparty Value Adjustment (CVA) on a large book of complex trades, we would differentiate path-wise, the batch-wise administrative work being negligible compared to the complexity of processing a single path.
12. Even writing simultaneously into different regions of memory close to one another causes what is known as *false sharing*, a flaw resulting not in wrong results, but a significantly reduced performance.
13. OpenMP is not part of C++ standard strictly speaking, but it is implemented in all major compilers, including Visual Studio. OpenMP instructs the compiler to run loops in parallel with simple pragmas and constitutes by far the simplest means of running code in parallel. This simplicity has a cost in terms of efficiency and control. For something as sophisticated as AAD over multi-threaded Monte-Carlo, it is advisable to develop custom thread pools, as explained in detail in the part I of [25].
14. Winner of the "Excellence in Risk Management and Modelling" RiskMinds 2019 award.

REFERENCES

[1] J. Andreasen. Back to the future. *Risk*, 2005.
[2] J. Andreasen. Payoff scripting languages: Sung and unsung glories and the next generation. QuantMinds, Vienna, 2019.
[3] J. Andreasen and B. Huge. Random grids. *Risk*, 2011.
[4] J. Andreasen and A. Savine. *Modern Computational Finance: Scripting for Derivatives and xVA*. Wiley Finance, 2018.
[5] G. Blacher and R. Smith. Leveraging gpu technology for the risk management of interest rates derivatives. Global Derivatives, 2015.
[6] J. F. Carriere. Valuation of the early-exercise price for options using simulations and nonparametric regression. *Insurance: Mathematics and Economics*, 19(1):19–30, 1996.
[7] S. Dalton. *Financial Applications using Excel Add-in Development in C / C++*. Wiley Finance, 2007.
[8] B. Dupire. Pricing with a smile. *Risk*, 7(1):18–20, 1994.
[9] B. Dupire. A unified theory of volatility. Working Paper, 1996.
[10] M. Giles and P. Glasserman. Smoking adjoints: Fast evaluation of greeks in monte carlo calculations. *Risk*, 2006.
[11] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2003.
[12] D. Goloubentsev and E. Lakshtanov. 2019. Aad: Breaking the primal barrier. *Wilmott* 2019:8–11. Available at https://wilmott.com/aad-breaking-the-primal-barrier.
[13] M. Henrard. *Algorithmic Differentiation in Finance Explained*. Palgrave Macmillan, 2017.
[14] R. J. Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Trans. Math. Softw*, 40(4), 2014.
[15] B. Huge and A. Savine. Deep analytics. QuantMinds, Vienna — slides posted on www.deep-analytics.org, 2019.
[16] P. Jaeckel. *Monte Carlo Methods in Finance*. Wiley Finance, 2002.
[17] M. S. Joshi. *C++ Design Patterns and Derivatives Pricing*. Cambridge University Press, 2008.
[18] P. L'Ecuyer. Combined multiple recursive number generators. *Operations Research*, 1996.
[19] K. Leppkes, J. Lotz, U. Naumann, and J. du Toit. Meta adjoint programming in c++. *AIB, Dept. of Computer Science, RWTH Aachen University*, 07 2017.
[20] F. A. Longstaff and E. S. Schwartz. Valuing american options by simulation: A simple least-squares approach. *The Review of Financial Studies*, 14(1):113–147, 2001.
[21] B. Moro. The full monte. *Risk*, 1995.
[22] A. Reghai, O. Kettani, and M. Messaoud. Cva with greeks and aad. *Risk*, 2015.
[23] A. Savine. Stabilize risks of discontinuous payoffs with fuzzy logic. Global Derivatives, 2016.
[24] A. Savine. Financial cash-flow scripting: Beyond valuation. Available at SSRN: https://ssrn.com/abstract=3281884 or http://dx.doi.org/10.2139/ssrn.3281884, 2018.
[25] A. Savine. *Modern Computational Finance, Volume 1: AAD and Parallel Simulations*. Wiley, 2018.
[26] O. Scavenius. Rate surfaces calibration and risk using auto differentiation. WBS Interest Rate Conference, London, 2012.
[27] N. Sherif. Aad vs gpus: banks turn to maths trick as chips lose appeal. *Risk Magazine*, 2015.
[28] A. Sokol. Retrofitting aad to your existing c++ library using tape compression. QuantsHub Webinar, 2015.
[29] A. Williams. *C++ Concurrency in Action*. Manning, 2012.