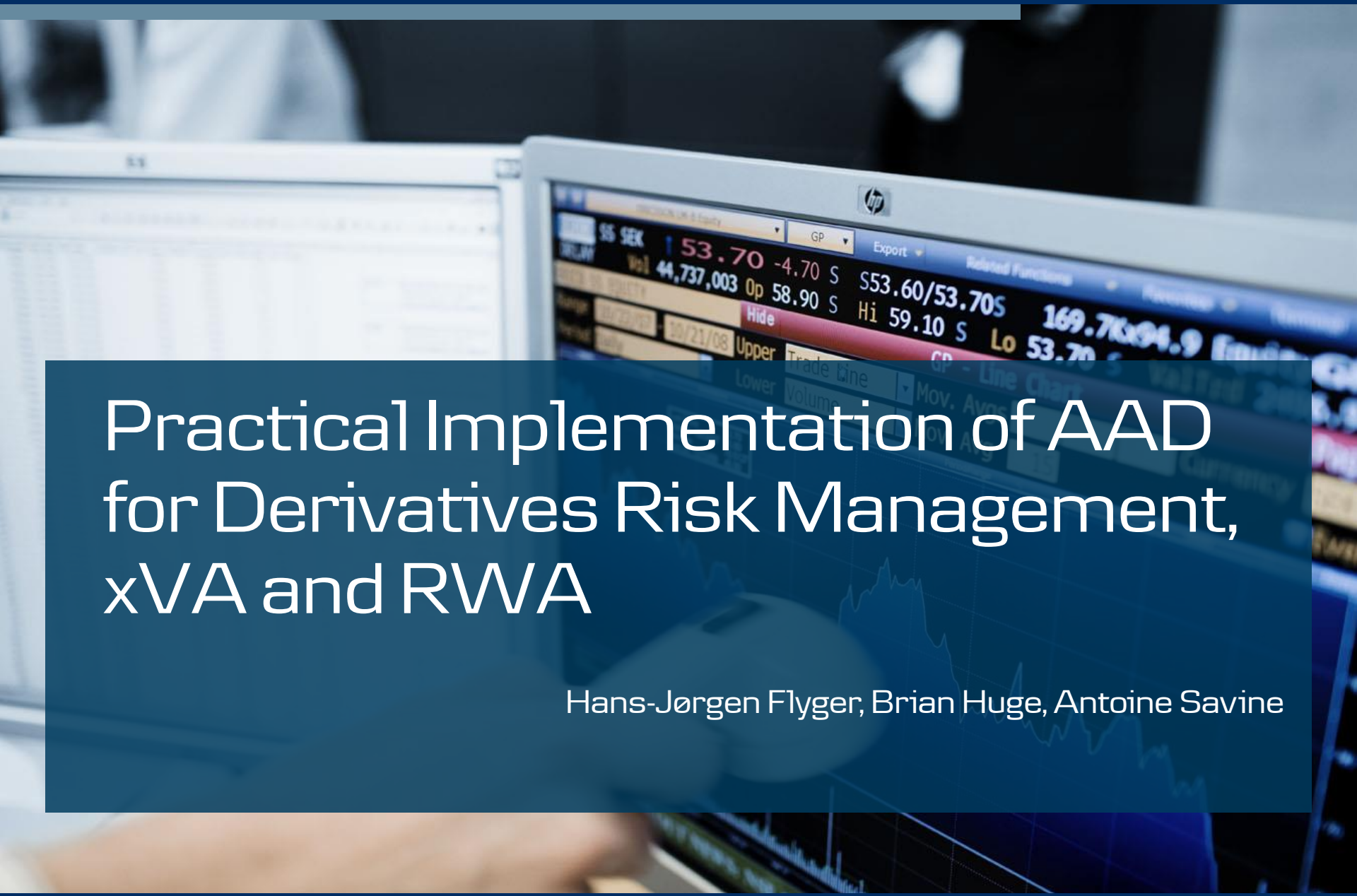
Danske Bank



Practical Implementation of AAD for Derivatives Risk Management, xVA and RWA

Hans-Jørgen Flyger, Brian Huge, Antoine Savine

Introduction

AAD in Finance

- AAD
  - A game changer for the computation of risk sensitivities
  - Applies to risks of derivatives books, xVA, capital charges, etc.
  - Useful beyond risk calculations: calibration, static hedge optimization, transaction costs, VAR, …
- AAD is easy in theory
  - Mathematically: an application of the chain rule $\left[f\left(x, g\left(x\right)\right)\right]_x = f_x + g_x f_g$
  - Implementation in C++ with operator overloading now well known
    See for instance, Savine, Global Derivatives 2014
    (We also quickly recap the main principles)
  - Quick and easy implementation with toy code, with "magical" results

- AAD is hard in practice
  - With real-life production systems, however, implementation of AAD is challenging
  - Quants in Danske Bank spent a couple of years implementing AAD across production systems (and were granted the Risk 2015 In-House System of the Year award for their efforts)
  - We expose the main problems and share some solutions

Introduction

The power of AAD

- Traditionally, with finite difference or ”bumping”
  - Bump inputs one by one and recalculate
  - **Sensitivity to n inputs costs n evaluations**
  - Even with smart caching, computation time remains proportional to n
- With AAD
  - All sensitivities are computed in one single sweep
  - **Sensitivity to n inputs takes constant time!**
  - In a given context, 5 sensitivities or 5,000 sensitivities are computed in the same time
  - The time is approx 4-8x a single evaluation
- Example: pricing takes 2sec, we produce 1,600 risks
  - Bumping: ~1 hour
  - AAD: 5sec
  - Multithreaded over 4 cores: bumping = 15min, AAD = 1.25sec

Introduction

Overview

1) Recap AAD 101: mathematics and implementation

2) Efficient differentiation of main financial algorithms
   - Multi-dimensional PDEs with check-pointing
   - Monte-Carlo simulations with multi-threading
   - Calibrations with the Implicit Function Theorem (IFT)

3) Application to CVA and RWA

Reverse Adjoint Propagation

Decompose into a sequence of elementary mathematical operations

- Consider a scalar function $f : \mathbb{R}^n \to \mathbb{R}$

  with value $y = f(x_1, \ldots, x_n)$

  where we are interested in calculating derivatives for the *n* independent variables

  $$\frac{\partial y}{\partial x_i}, \; i = 1, \ldots, n$$

- Any such function evaluated on a computer is – for a given control flow – decomposed into a sequence of elementary mathematical operations: +, -, *, /, exp, log, pow, ...
- The elementary operations f_i are unary or binary and the sequence can be written as

  $$x_k = f_k(x_i) \;\; \text{or} \;\; x_k = f_k(x_i, x_j), \;\; k > i, j, n$$

  and the result $y = x_N$
- The order of the operations is given by the compiler

Reverse Adjoint Propagation

Reverse adjoint propagation

- For each intermediate operation/variable we define the associated ***adjoint***

$$A_i \equiv \frac{\partial y}{\partial x_i}$$

- We have $A_N = 1$ and from the chain rule

$$A_i = \sum_{j \in E_i} \frac{\partial y}{\partial x_j}\frac{\partial f_j}{\partial x_i} = \sum_{j \in E_i} A_j \frac{\partial f_j}{\partial x_i}, \quad E_i = \left\{ j > i \mid \frac{\partial f_j}{\partial x_i} \neq 0 \right\}$$

- I.e. for each elementary operation f_j ($j>i$) having argument(s) x_i
  - we calculate the analytically known $\frac{\partial f_j}{\partial x_i}$
  - And add $A_j \frac{\partial f_j}{\partial x_i}, \; j > i$ to the adjoint A_i for argument x_i!
  - … adjoints are propagated from result to the argument(s)

- All adjoints are calculated by going through the same sequence of operations as for the (usual fwd) value calculation ***once***
  - But calculating *adjoint contributions* instead of results at each node
  - … in reverse order starting from A_N with initial condition $A_N = 1, \; A_{i \neq N} = 0$

Reverse Adjoint Propagation

Simple example

Calculate value and derivatives for

$$y = 2x_1 + \log(x_1 x_2 + x_3) \quad x_1 = 2,\ x_2 = 3,\ x_3 = 4$$

Operation	Val	Adjoint contrib	Aggregated adjoint
$x_1=2$	2		
$x_2=3$	3		
$x_3=4$	4		
$x_4=x_1*x_2$	6	$A_1+=A_4*x_2$	$A_1=2+3/10$
		$A_2+=A_4*x_1$	$A_2=2/10$
$x_5=x_4+x_3$	10	$A_4+=A_5*1$	$A_4=1/10$
		$A_3+=A_5*1$	$A_3=1/10$
$x_6=\log(x_5)$	$\log(10)$	$A_5+=A_6/x_5$	$A_5=1/10$
$x_7=2*x_1$	4	$A_1+=A_7*2$	$A_1=2$
$x_8=x_7+x_6$	$4+\log(10)$	$A_7+=A_8*1$	$A_7=1$
		$A_6+=A_8*1$	$A_6=1$
			$A_1=\ldots=A_7=0, A_8=1$

Reverse Adjoint Propagation

Implementation

- I.e. to calculate the derivatives $\frac{\partial y}{\partial x_i}$ of a function $y = f(x_1, \ldots, x_n)$:
  1. Do the usual forward calculation to get *y*

     … keeping track of the entire sequence of operations f_j and argument(s) x_i
  2. Calculate adjoints by running the sequence backwards, starting at $A_N = 1$

- This may be implemented by
  1. Source transformation (e.g. tapenade)
  2. Templates and operator overloading in a modern language

- We use templates and operator overloading in C++

  … but in all cases is the computational time independent of the number of sensitivities

AAD with Operator Overloading in C++

The 4 ingredients

- To implement AAD using templates and operator overloading, we need 4 basic ingredients
  1. A customized number type
  2. Operator nodes
  3. A ”tape” for storing operator nodes
  4. An AD evaluator that calculate adjoints by running the tape backwards

- Since extra work is introduced for each single mathematical operation, optimizing this part of the code is essential!

AAD with Operator Overloading in C++

First ingredient: kDoubleAd

1. Customized number type
   - A class for representing numbers (we call it kDoubleAd)
   - Overload all elementary mathematical operations for kDoubleAd : +, -, *, /, exp, log, pow, ...
   - As a side effect in the overload, the performed operation is stored as a node on "the tape"
   - The kDoubleAd only needs to contain a pointer to the operator node
     - ... e.g. in our earlier example, *y* contains a pointer to operator node 8 ("+" operator)

```
class kDoubleAd
{
public:
    kDoubleAd() : myOper(nullptr){}
    kDoubleAd(const double& val);
    ...
    kDoubleAd& operator*=(const kDoubleAd& rhs);
    kDoubleAd& operator*=(const double&    rhs);
    ...
    friend kDoubleAd operator*(const kDoubleAd& lhs, cons
    friend kDoubleAd operator*(const kDoubleAd& lhs, cons
    friend kDoubleAd operator*(const double&    lhs, cons
    ...
protected:
    //  pointer to the operator node
    kDoubleAdOper*  myOper;
};
```

- Perform the calculation (e.g. $y = f(x_1, \ldots, x_n)$) using kDoubleAds instead of native doubles to store the operations
  - ... by templatizing calculation code

```
double f(
    double x1,
    ...
    double xn);
```

→

```
template<class T>
T f(
    T x1,
    ...
    T xn);
```

AAD with Operator Overloading in C++

Second ingredient: operator nodes

2. Operator node

- The operator nodes are stored on the “tape” and contains what is needed to calculate the adjoint contributions:
  - The type of the operation (+, -, *, /, exp, log, pow, …)
  - The result of the operation
  - Pointer to the argument node(s)
  - Placeholder for the adjoint

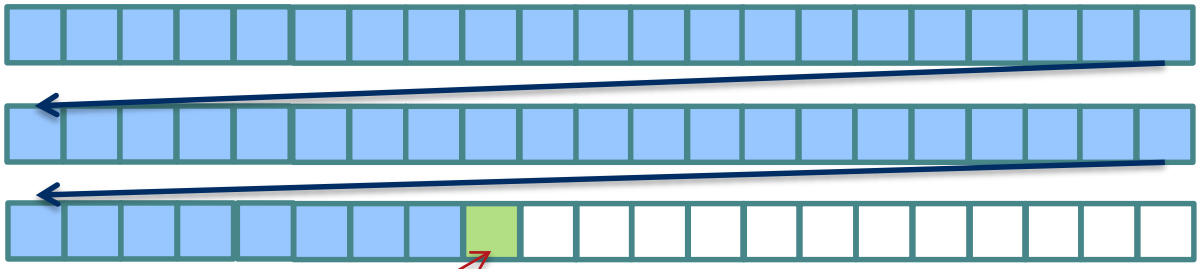
```
class kDoubleAdOper
{
    ...
protected:
    //  the operator type
    unsigned char   myType;

    //  the value (of the operation)
    double          myVal;

    //  lhs oper
    kDoubleAdOper*  myLhs;

    //  rhs oper
    kDoubleAdOper*  myRhs;

    //  adjoint value calculated by evaluator
    double          myAdjoint;
```

AAD with Operator Overloading in C++

Third ingredient: The tape

3. The tape

- The tape is the memory for the operator nodes and contains the entire sequence of operator nodes
- The tape is static (thread_local) to make it accessible for all calculation code
- Memory is allocated as a list of (big) blocks of contiguous memory
  - … to limit expensive memory allocations
  - … to help the computer's memory prefetcher (more about that later)



```
inline kDoubleAd
operator*(const kDoubleAd& lhs, const kDoubleAd& rhs)
{
    return kDoubleAd(new(tapeAllocate()) kDoubleAdOper(Mult, lhs.val()*rhs.val(), lhs.oper(), rhs.oper()));
}
```

Return constructed kDoubleAd using RVO

Construct new operator node using placement new

Node type

Node result

LHS argument node

RHS argument node

AAD with Operator Overloading in C++

Fourth ingredient: The AD RAP evaluator

4. The AD RAP evaluator

- Set the initial condition for the adjoints and run backwards through the operator nodes on the tape to calculate all adjoints
- At each operator node, adjoints are propagated from result to argument node(s)

```
//  f=l*r; df/dl=r; df/dr=l
case kDoubleAdOper::Mult:
{
    oper->lhs()->adjoint() += oper->adjoint()*oper->rhs()->val();
    oper->rhs()->adjoint() += oper->adjoint()*oper->lhs()->val();
    break;
}
```

$$A_{i_L} \mathrel{+}= A_j * \frac{\partial f_j}{\partial x_{i_L}}$$

$$A_{i_R} \mathrel{+}= A_j * \frac{\partial f_j}{\partial x_{i_R}}$$

- When done, the derivative $A_i = \frac{\partial y}{\partial x_i}$ can be picked from the operator node i (e.g. through the pointer to the operator node sitting on the relevant kDoubleAd variable)

```
//  generate AD tape
kDoubleAd x1, ..., xi, ..., xn;
...
kDoubleAd y = f<kDoubleAd>(x1, ..., xi, ..., xn);

kAd eval;
//  initialise adjoints to 0
eval.initAdjoints();
//  ... and the adjoint for the result to 1
y.oper()->adjoint() = 1.0;

//  run bwd AD evaluator
eval.evalBwd();

//  pick derivative(s)
double dydxi = xi.oper()->adjoint();
```

AAD with Operator Overloading in C++

Complexity

AAD complexity

- Fwd calculation: ~ 1 * (non-AD) fwd calculation
  - Operator overload side effect
  - Memory allocation
- Bwd calculation: ~ 1 * (non-AD) fwd calculation
  - ~2 times the (AD) fwd calculation
  - Operator node traversal
  - More calculations at each node
    - E.g. 2 multiplications and 2 additions for the multiplication node
- Expect 4–8 * (non-AD) fwd calculation
- … but constant in number of sensitivities!

Memory Usage

Memory usage and memory cache

- Using kDoubleAds quickly fills the tape with operator nodes
  - Each core typically produces around 10^8 operator nodes per sec … corresponding to 4 GB/sec
- … using 64 bit code is a necessity for production scale calculations
- Keep tapes small …
  - When the tape is traversed backwards to calculate adjoints, each node is read from the CPUs memory cache.
  - Due to the contiguous memory structure of the tape, the memory pre-fetcher can (and will) load subsequent nodes into the memory cache before the CPU needs them
  - BUT it can not predict and pre-fetch the argument nodes
    - … resulting in a high number of cache misses for long tapes (where the CPU has to wait for data to be loaded from the slow main memory)
- Keeping tapes small – and ideally small enough to fit into the cache – is important for performance!
  - … a **large number of small tapes is much better** than a small number of large tapes!

Checkpointing

Reduce tape size by checkpointing

- Checkpointing is a general technique for splitting the AAD calculation into a number of smaller slices/tapes
  - Split the algorithm into *M* slices
    - and calculate (non-AD) values for the first *M-1* slices
  - A tape is generated for the final slice *M*
    - and adjoints are calculated using the usual initial conditions $A_N = 1,\ A_{i \neq N} = 0$
  - The tape for slice *M* is wiped and a tape is generated for slice *M*-1
    - This time adjoints are calculated using the calculated derivatives from slice *M* as initial condition
    - i.e. we "glue" tapes together via the adjoints
- This is a general method and works well with finite difference grids

Checkpointing

The simple example extended to use checkpointing

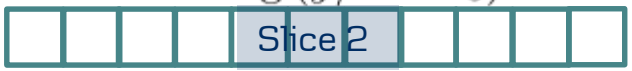
Calculate derivatives $\frac{\partial z}{\partial x_i}$ for $z = \log\left(\left(2x_1 + \log(x_1 x_2 + x_3)\right)/2 - x_3\right)$

y

Split in 2 slices:

$y = 2x_1 + \log(x_1 x_2 + x_3)$

$z = \log(y/2 - x_3)$



1. Calc value of slice 1 (use double – no AD)

2. Generate tape for slice 2 (use kDoubleAd)

3. Set initial conditions $A_z = 1,\ A_{i \neq z} = 0$
   Run slice 2 bwds to calculate adjoints

4. Store derivatives and wipe tape

5. Generate tape for slice 1 (use kDoubleAd)

6. Set initial conditions $A_y = \frac{\partial z}{\partial y},\ A_3 = \frac{\partial z}{\partial x_3},\ A_{i \notin \{y,3\}} = 0$
   Run slice 1 bwds to calculate adjoints

7. Pick derivatives $\frac{\partial z}{\partial x_i}$ from the x_i nodes and wipe tape

Linear Algebra of AAD

Is AAD magic?

- Consider the matrix multiplication $Z = X^T B_1 B_2 Y$

  where $B_1, B_2 \in \mathbb{R}^{M \times M}, X, Y \in \mathbb{R}^M, Z \in \mathbb{R}$

$$[Z] = \begin{bmatrix} X^T \end{bmatrix} \begin{bmatrix} B_1 \end{bmatrix} \begin{bmatrix} B_2 \end{bmatrix} \begin{bmatrix} Y \end{bmatrix}$$

- If we need $Z_1, \cdots, Z_L$ for different $Y_1, \cdots, Y_L$ then we do the $X^T B_1 B_2$ multiplication first and apply the result to each of the Y's
- Instead, if we need $Z_1, \cdots, Z_L$ for different $X_1, \cdots, X_L$ then we do the $Y^T B_2^T B_1^T$ multiplication first and apply the result to each of the X's
- Remember, the adjoint of a matrix is the transpose

Linear Algebra of AAD

Toy example again

$$y = 2x_1 + \log\left(x_1 x_2 + x_3\right)$$

$$
\begin{aligned}
& & x_4 &= x_1 x_2 \\
(n=1) & & \dot{x}_4 &= x_1 \dot{x}_2 + \dot{x}_1 x_2 \\
& & x_5 &= x_3 + x_4 \\
(n=2) & & \dot{x}_5 &= \dot{x}_3 + \dot{x}_4 \\
& & x_6 &= \log x_5 \\
(n=3) & & \dot{x}_6 &= \frac{\dot{x}_5}{x_5} \\
& & x_7 &= 2x_1 \\
(n=4) & & \dot{x}_7 &= 2\dot{x}_1 \\
& & x_8 &= x_6 + x_7 \\
(n=5) & & \dot{x}_8 &= \dot{x}_6 + \dot{x}_7
\end{aligned}
$$

Linear Algebra of AAD

Calculations on vector form

- Define $\dot{X}(n)=\begin{bmatrix}\dot{x}_1(n)\\ \vdots \\ \dot{x}_8(n)\end{bmatrix}$ and let the matrix $A(n)$ be the calculation for step $n=1,\cdots,N$
- Since differentiation is a linear operation each calculation can be it written in matrix form as

$$\dot{X}(n)=A(n)\dot{X}(n-1)$$

or starting from the boundary condition $\dot{X}(0)$ we have

$$\dot{X}(N)=A(N)\cdots A(1)\dot{X}(0)=\prod_{i=1}^{N}A(i)\dot{X}(0)$$

Linear Algebra of AAD

Calculation step 1

- First calculation step $n=1$ is $\dot{x}_4(1) = x_1\dot{x}_2(0) + \dot{x}_1(0)x_2 + \dot{x}_4(0)$

$$= \begin{bmatrix} x_2 & x_1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \dot{X}(0)$$

- Hence $\dot{X}(1) = A(1)\dot{X}(0)$

where

$$A(1) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ x_2 & x_1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Linear Algebra of AAD

Calculation steps 2, 3, 4 and 5

$$A(2)=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&1&1&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&1\end{bmatrix}$$

$$A(3)=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&\frac{1}{x_5}&1&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&1\end{bmatrix}$$

$$A(4)=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\2&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&1\end{bmatrix}$$

$$A(5)=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&1&1&1\end{bmatrix}$$

Linear Algebra of AAD

Going forward

- From this we get $\dot{y} = e_8^T \dot{X}(N) = e_8^T \prod_{i=1}^{N} A(i) \dot{X}(0)$

$$\frac{\partial y}{\partial x_1} = e_8^T \prod_{i=1}^{N} A(i) e_1$$
$$\frac{\partial y}{\partial x_2} = e_8^T \prod_{i=1}^{N} A(i) e_2$$
$$\frac{\partial y}{\partial x_3} = e_8^T \prod_{i=1}^{N} A(i) e_3$$

  where e_i is the i'th unit vector

- We need to calculate the system with 3 different boundary conditions e_1, e_2, e_3
- The easiest way is to calculate it forward as

$$\left[\frac{\partial y}{\partial x_1} \; \frac{\partial y}{\partial x_2} \; \frac{\partial y}{\partial x_3} \; \frac{\partial y}{\partial x_4} \; \frac{\partial y}{\partial x_5} \; \frac{\partial y}{\partial x_6} \; \frac{\partial y}{\partial x_7} \; \frac{\partial y}{\partial x_8} \right] = e_8^T \prod_{i=1}^{N} A(i)$$

  and then apply the 3 different boundary conditions

- This is equivalent to pick the first 3 elements

$$\left[\frac{\partial y}{\partial x_1} \; \frac{\partial y}{\partial x_2} \; \frac{\partial y}{\partial x_3} \right]$$

Linear Algebra of AAD

Sub sequences

- Checkpointing is just a sub sequence

$$\left[\frac{\partial y}{\partial X(0)}\right]^T = \left[\frac{\partial y}{\partial X(N)}\right]^T \prod_{n=1}^{N} A(n) = \left[\frac{\partial y}{\partial X(K)}\right]^T \prod_{n=1}^{K} A(n)$$

$$\left[\frac{\partial y}{\partial X(N)}\right]^T \prod_{n=K+1}^{N} A(n)$$

$$\left[\frac{\partial y}{\partial X(K)}\right]^T \prod_{n=1}^{K} A(n)$$

- Also notice that any sub sequence of calculations is a Jacobian
- I.e. for any $K \geq k$ we have

$$\left[\frac{\partial y}{\partial X(k)}\right]^T = \left[\frac{\partial y}{\partial X(K)}\right]^T \prod_{n=k+1}^{K} A(n) = \left[\frac{\partial y}{\partial X(K)}\right]^T \left[\frac{\partial X(K)}{\partial X(k)}\right]$$

- This can be a useful reference when we discuss risk propagation later

Checkpointed AAD for PDE finite difference grids

Example: SLV – 2D PDE

- Arbitrage free volatility surface
  - Andreasen and Huge: Volatility interpolation (Risk March 2011)

- Stochastic volatility model
  - Andreasen and Huge: Random Grids (Risk July 2011)

$$ds = \sigma(t,s)\sqrt{z}dW$$

$$dz = \theta(1-z)dt + \varepsilon z^{\gamma} dZ, \qquad z(0) = 1$$

$$dWdZ = 0$$

Checkpointed AAD for PDE finite difference grids

Grid setup

- Discrete time and state space (FD)

$$v_i = \prod_{j=i+1}^{n} A_j d_n = A_{i+1} v_{i+1} \quad , \qquad i = 0,1,\ldots,n$$

- where d_n is the cashflow at some future time
- v_0 is the present value
- A_i is a matrix with the finite differences

Checkpointed AAD for PDE finite difference grids

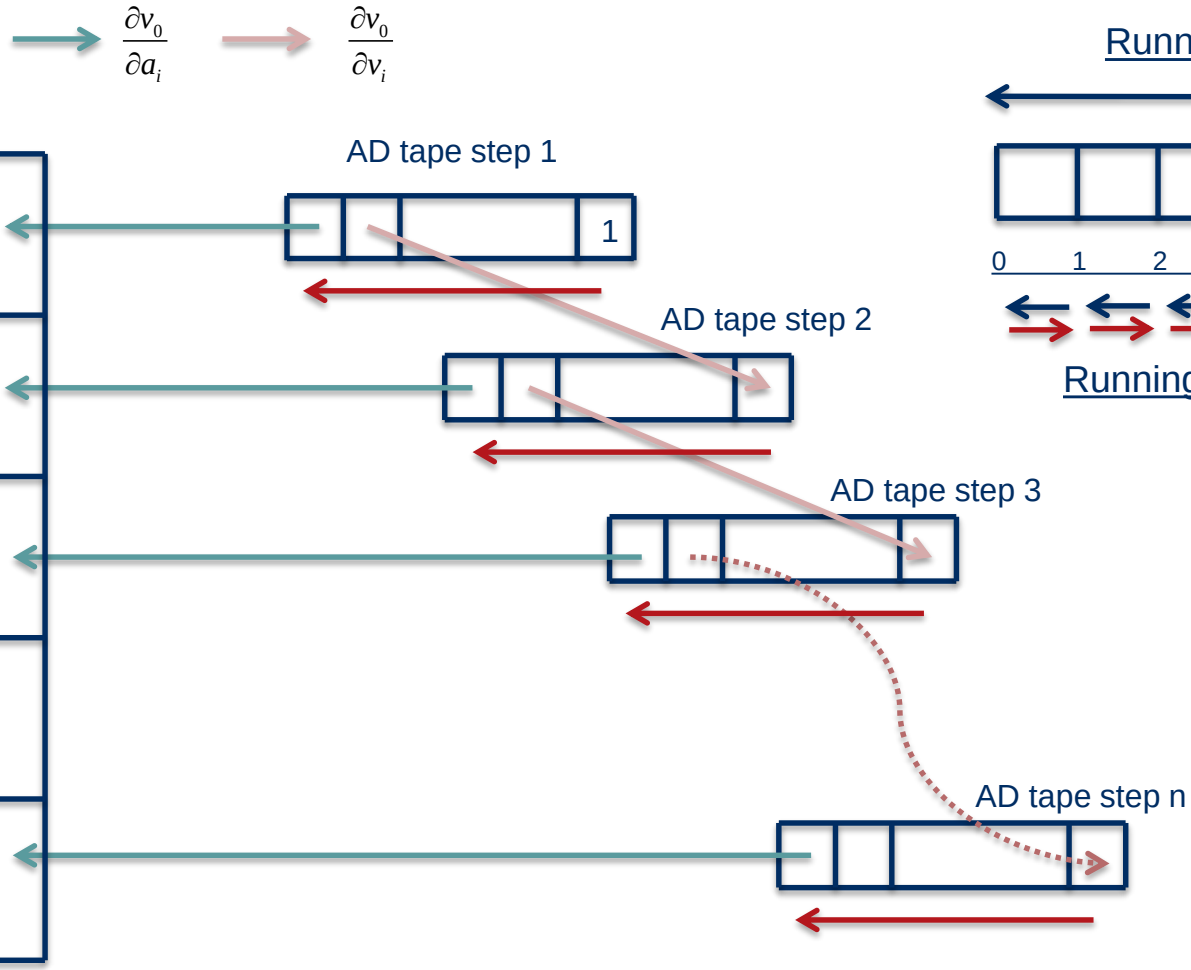
Algorithm

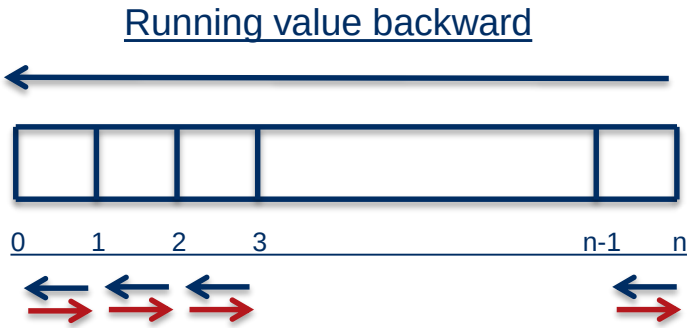
- A_i is a function of parameters a_i to which we want risks
- To find $\frac{\partial v_0}{\partial a_j}$ we run the following checkpointed procedure

1. Run the whole FD without AD and record all the vectors v_i
2. Run the step $v_0 = A_1 v_1$ with AD, record a tape
3. Run RAP over the tape to get $\frac{\partial v_0}{\partial a_1}, \frac{\partial v_0}{\partial v_1}$
4. Record $\frac{\partial v_0}{\partial v_1}$ to be used as boundary for the next RAP over $v_1 = A_2 v_2$ then wipe the tape
5. Repeat from step 2 $v_i = A_{i+1} v_{i+1}$

Checkpointed AAD for PDE finite difference grids

Algorithm





AAD for Monte-Carlo simulations

The problem

- AAD for Monte Carlo may consume a lot of memory
  - 10sec MC run ➔ around 40GB for the tape
- Monte Carlo simulation suggests multi threading
  - Each path can be handled independently and run in parallel
- Running each path separately would also reduce the need for memory
  - Similar to check pointing in the sense it creates more manageable calculations
- AAD creates dependency between paths
  - Variables/cashflows are averages across paths
- How do we combine AAD and multi threading?

AAD for Monte-Carlo simulations

Pathwise derivatives to the rescue

- Giles and Glasserman: ”Smoking Adjoints: fast Monte Carlo Greeks” introduced adjoint differentiation in Finance. They used pathwise derivatives
- Essential idea is that under suitable conditions

$$\frac{\partial E\left[Cashflow\right]}{\partial a} = \mathrm{E}\left[\frac{\partial Cashflow}{\partial a}\right]$$

- Derivatives can be calculated as Monte Carlo estimates on a different payoff
- New ”payoff” can easily be calculated with AAD
- We are again in a situation where Monte Carlo paths can be treated independently

AAD for Monte-Carlo simulations

Pathwise derivatives with AAD

- The derivatives are calculated as $\frac{\partial E\left[Cashflow\right]}{\partial a} \approx \frac{1}{n}\sum_{i=1}^{n}\frac{\partial Cashflow_i}{\partial a} \equiv \frac{1}{n}\sum_{i=1}^{n} X_i$

- Each X_i is calculated with AAD over path i
  - One tape per simulation, maximum time 1/100sec ➔ up to 40MB, almost fits in CPU cache
  - Record X_i and wipe the tape between simulations

- Result = average of X_i

- Remember that the output from the tapes is ALL the derivatives so we only need to run this procedure only once

AAD for Monte-Carlo simulations

Multi-threading

- Multithreading
  - Run Monte-Carlo paths in parallel
  - Compute pathwise derivatives with AAD in parallel
- Problem:
  - Templated code writes to a global/static tape
  - Writing to the tape from parallel threads leads to race conditions
- Solution: give each thread its own tape
  - **thread_local** objects are like global/static except each thread has its own copy
  - Support for thread local in C++11 standard, Boost, Windows API, …
- Hence what it takes for AD to work with multi-threaded Monte-Carlo:
  - Create a tape for each thread, access it through thread local variables
  - Obviously, work with per thread/task copies of variables written into
  - In addition, copy inputs into custom number types per thread/task so they land on the right tape!

Risk propagation

- Methods explained so far produce derivatives of result to *model* params like *local* volatilities in Dupire

- We want derivatives to *market* variables, like *implied* volatilities in Dupire
- Model parameters are derived from market variables in one or more steps that may involve *calibrations*
- To Naively run AAD over this process may be unstable and memory consuming

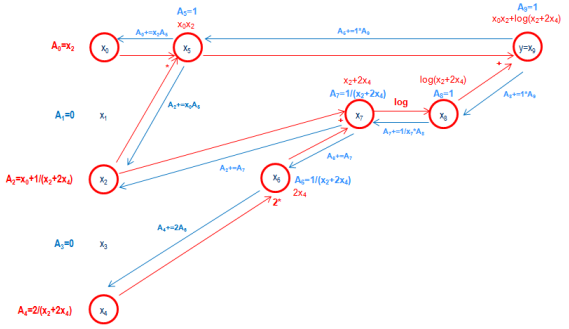
- **FAQ: how to run AAD through calibration?**
- Solution: first run model with AAD to get sensitivities to model parameters

➔Then propagate sensitivities *backwards* to market variables

Backward Risk Propagation and RAP

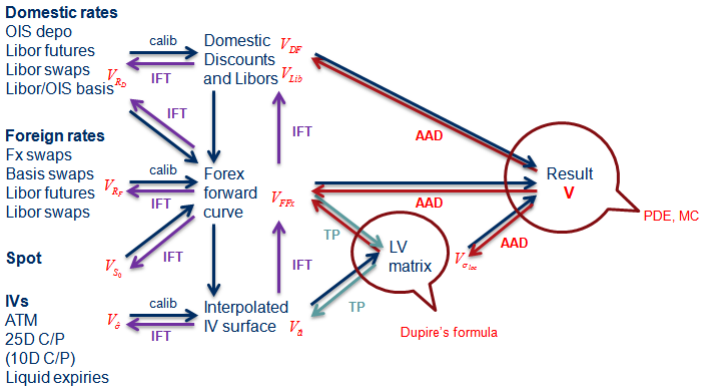
- RAP: Backward propagation of sensitivities at micro (math operation) level

$$y = x_0 x_2 + \log\left(x_2 + 2x_4\right)$$



- Risk propagation: Backward propagation of sensitivities at macro (param) level

Market data -> Interpolated IV -> Calculated LV -> Price



Risk Propagation by example: Dupire's model

$$\vec{a} = h(\hat{\sigma}, S_0)$$

$$h : \min_{\vec{a}} \|e\|^2, e = [e_i(\vec{a}, \hat{\sigma}, S_0)]$$

$$e_i = w_i \left[C_{K_i,T_i}(\vec{a}, S_0) - BS_{K_i,T_i}(\hat{\sigma}, S_0) \right]$$

$$\sigma(K,T) = \sqrt{\frac{2C_T(K,T)}{K^2 C_{KK}(K,T)}} = g_{K,T}(\vec{a}, S_0)$$

observed market implied BS volatilities

$\hat{\sigma} = (\hat{\sigma}_i)$

$\hat{\sigma}_i = \hat{\sigma}(K_i, T_i)$

$1 \le i \le n$

calibration of **a**

step 1

spot S_0

Implicit function

IV surface

= all vanilla option prices

$C(K,T) = f_{\vec{a},S_0}(K,T)$

$\vec{a} = (a_j)$

$1 \le j \le m \le n$

Explicit function (Dupire)

computation of LV

step 2

spot S_0

Local Volatility

$\sigma(S_t, t)$

$\sigma = (\sigma_k)$

$\sigma_k = \sigma(S_k, t_k)$

$1 \le k \le p$

Monte-Carlo, PDE

step 3

spot S_0

Result **V**

Pricing

Risk

$$\frac{\partial V}{\partial S_0} += \left(\frac{\partial V}{\partial a_j}\right)\left(\frac{\partial a_j}{\partial S_0}\right)$$

$$\frac{\partial V}{\partial S_0} += \left(\frac{\partial V}{\partial \sigma_k}\right)\left(\frac{\partial \sigma_k}{\partial S_0}\right)$$

$$\frac{\partial V}{\partial S_0}$$

$\left(\frac{\partial V}{\partial \hat{\sigma}_i}\right)$

step 3

Implicit Function Theorem

$\left(\frac{\partial V}{\partial a_j}\right)$

step 2

Tape Pasting

$\left(\frac{\partial V}{\partial \sigma_k}\right)$

step 1

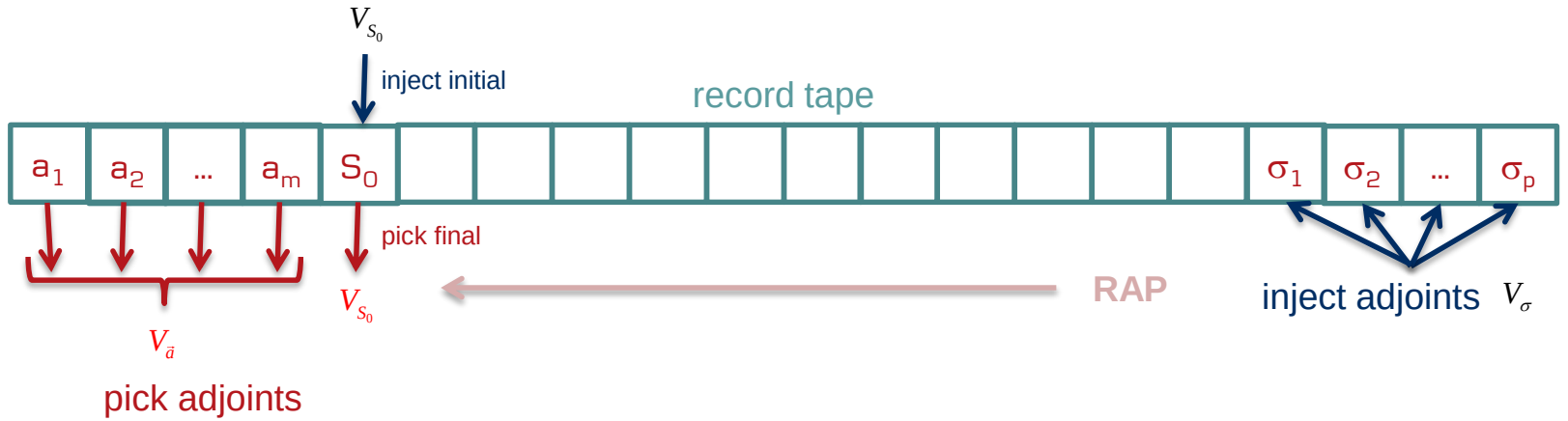
AAD

Result **V**

Propagation through explicit functions: Tape Pasting

- Step 2 in our example: computation of local volatilities $\sigma = g(\vec{a}, S_0)$
  - We know sensitivities to LVs V_σ from step 1
  - We want to propagate them to sensitivities $V_{\vec{a}}$ to interpolation parameters
  - We could compute the Jacobian $\sigma_{\vec{a}}$ of g and apply the chain rule $V_{\vec{a}} = V_\sigma \sigma_{\vec{a}}$
  - We can do this *orders of magnitude* faster with tape pasting (but only for explicit functions)

run explicit function $\left(\sigma_k^{\,2}\right) = \left[\sigma^2(K_k, T_k)\right] = \left[\dfrac{2C_T(K_k, T_k)}{K^2 C_{KK}(K_k, T_k)}\right] = \left[\dfrac{2 f_T(\vec{a}, S_0, K_k, T_k)}{K^2 f_{KK}(\vec{a}, S_0, K_k, T_k)}\right] = \left[g_k(\vec{a}, S_0)\right]$ in AAD mode



Propagation through implicit functions

- Step 3 in our example $\vec{a} = h(\hat{\sigma}, S_0)$, where h is implicitly defined by a calibration
  - We know the sensitivities to interpolation parameters $V_{\vec{a}}$ from step 2
  - We want to propagate them to sensitivities to IVs $V_{\hat{\sigma}}$ and spot $V_{S_0}{}^{\hat{\sigma}}$
  - We need the Jacobians $h_{\hat{\sigma}}$ and h_{S_0} of h so we can apply the chain rule $\begin{cases} V_{\hat{\sigma}} = V_{\vec{a}} h_{\hat{\sigma}} \\ V_{S_0}{}^{\hat{\sigma}} = V_{S_0}{}^{\vec{a}} + V_{\vec{a}} h_{S_0} \end{cases}$
  - h is implicitly defined by calibration to n instruments: $h : \min_{\vec{a}} \|e\|^2, e = \left[e_i(\vec{a}, \hat{\sigma}_i, S_0) \right] = \left\{ w_i \left[C_{K_i,T_i}(\vec{a}, S_0) - BS_{K_i,T_i}(\hat{\sigma}_i, S_0) \right] \right\}$
  - Where the error functions e_i are explicit and we easily compute their Jacobians $e_{\vec{a}}, e_{\hat{\sigma}}, e_{S_0}$
  - We need the Jacobians $h_{\hat{\sigma}}$ and h_{S_0} of the implicit function out of these and we're home safe

Implicit Function Theorem

- Theorem: $\begin{cases} h_{\hat{\sigma}} = -Ie_{\hat{\sigma}} \\ h_{S_0} = -Ie_{S_0} \end{cases}$

- Where $I = (e_{\vec{a}}{}' e_{\vec{a}})^{-1} e_{\vec{a}}{}'$ is the "pseudo-inverse" of $e_{\vec{a}}$ the Jacobian of calibration errors to the results of h = interpolation parameters
- Note that I is the projection operator ➔ propagation = risk projection
- In a perfect calibration context, $e_{\vec{a}}$ is square of full rank, and in this case $I = e_{\vec{a}}^{-1}$

➔ propagation = Jacobian inversion

- Finally, the general formula for propagation through calibrations:

$$\begin{cases} V_{\hat{\sigma}} = -V_a (e_{\vec{a}}{}' e_{\vec{a}})^{-1} e_{\vec{a}}{}' e_{\hat{\sigma}} \\ V_{S_0}{}^{\vec{a}} = -V_a (e_{\vec{a}}{}' e_{\vec{a}})^{-1} e_{\vec{a}}{}' e_{S_0} \end{cases}$$

Proof

$\vec{a}^* = h(\hat{\sigma}, S_0)$ is the solution of $\min_{\vec{a}} \|e\|^2, e = \left[e_i(\vec{a}, \hat{\sigma}_i, S_0) \right]$

At the optimum, we have $e_i(\vec{a}^*, \hat{\sigma}, S_0) = e_i\left[h(\hat{\sigma}, S_0), \hat{\sigma}, S_0 \right] = \varepsilon_i(\hat{\sigma}_i, S_0)$

and we have the 1st order condition $e_{\vec{a}}' e = 0_m = e_{\vec{a}}' \varepsilon$

Deriving the 1st order condition with respect to $\hat{\sigma}$ and S_0 :

$$\begin{cases} (e_{\vec{a}}' \varepsilon)_{\hat{\sigma}} = 0_{mn} = e_{\vec{a}\hat{\sigma}}' \varepsilon + e_{\vec{a}}' \varepsilon_{\hat{\sigma}} \\ (e_{\vec{a}}' \varepsilon)_{S_0} = 0_m = e_{\vec{a}S_0}' \varepsilon + e_{\vec{a}}' \varepsilon_{S_0} \end{cases}$$

At the optimum, errors are negligible compared to derivatives, hence:

$$\begin{cases} e_{\vec{a}}' \varepsilon_{\hat{\sigma}} = 0_{mn} = e_{\vec{a}}' \left(e_{\vec{a}} h_{\hat{\sigma}} + e_{\hat{\sigma}} \right) \\ e_{\vec{a}}' \varepsilon_{S_0} = 0_m = e_{\vec{a}}' \left(e_{\vec{a}} h_{S_0} + e_{S_0} \right) \end{cases}$$

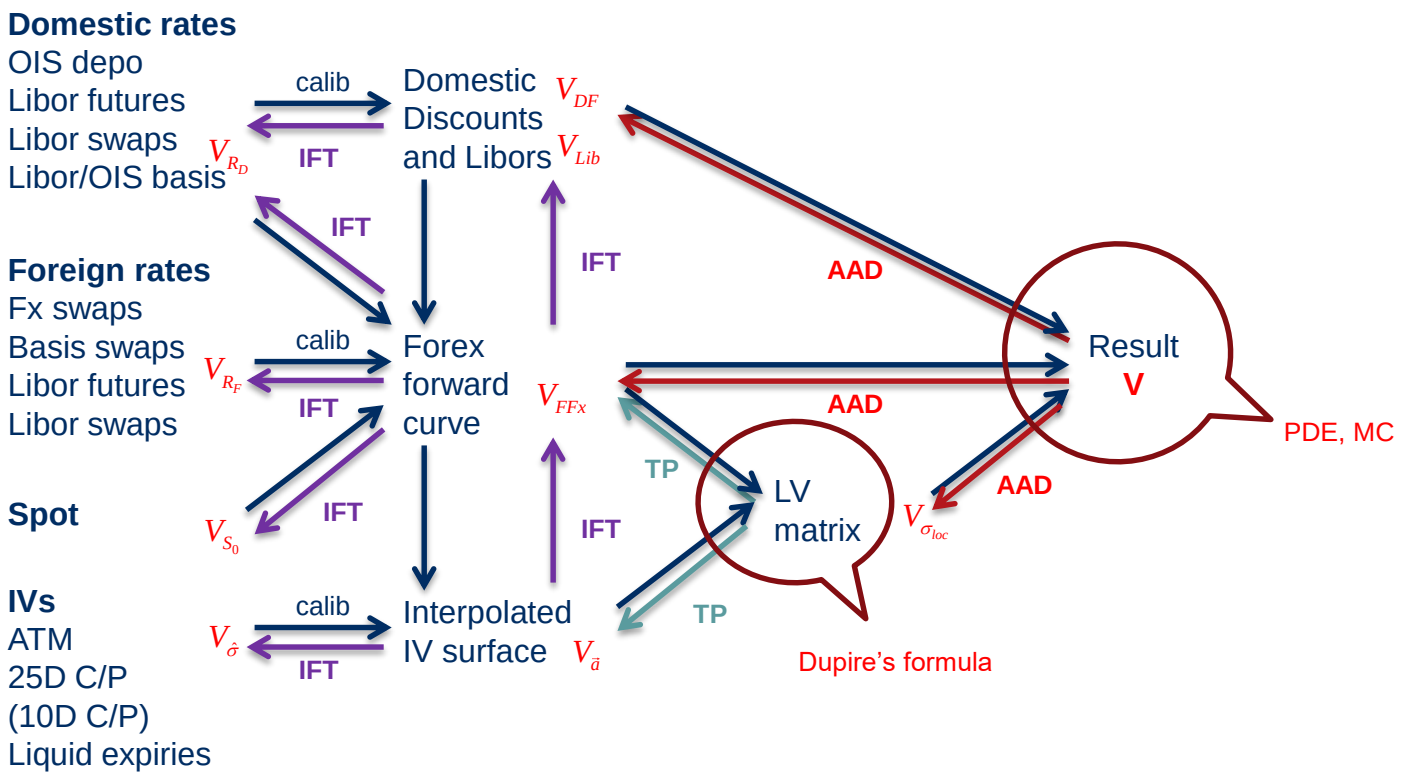
So, $\begin{cases} e_{\vec{a}}' e_{\vec{a}} h_{\hat{\sigma}} = -e_{\vec{a}}' e_{\hat{\sigma}} \\ e_{\vec{a}}' e_{\vec{a}} h_{S_0} = -e_{\vec{a}}' e_{S_0} \end{cases}$ in other terms $\begin{cases} h_{\hat{\sigma}} = -\left(e_{\vec{a}}' e_{\vec{a}} \right)^{-1} e_{\vec{a}}' e_{\hat{\sigma}} \\ h_{S_0} = -\left(e_{\vec{a}}' e_{\vec{a}} \right)^{-1} e_{\vec{a}}' e_{S_0} \end{cases}$

Backward Risk Propagation: general algorithm

- Successive vectors of parameters $\vec{P}_i$ are derived or calibrated from one another
  - From initial market observed or trader input sets $\vec{P}_0 \ldots \vec{P}_{n-1}$ to which we want sensitivities
  - Through successive transformations $\vec{P}_i = f_i\left[\left(\vec{P}_j\right)_{j<i}\right]$ from some previously computed sets
  - Eventually leading to the set of final *model parameters* $\vec{P}_{N-1}$
- Use AAD to compute derivatives $V_{\vec{P}_{N-1}}$ of the result to model parameters $\vec{P}_{N-1}$
- Propagate risks in the reverse order to the construction of the parameters
  - For each f_i, from output to inputs: $\vec{P}_i = f_i\left[\left(\vec{P}_j\right)_{j<i}\right] \Rightarrow V_{\vec{P}_i} \xrightarrow{propagation} \left(V_{\vec{P}_j}\right)$
  - When f_i is explicit, we use tape pasting
  - When f_i is implicit, we use the Implicit Function Theorem for each input set $V_{P_j} += -V_{P_i}\left(e_{P_i}{}' e_{P_i}\right)^{-1} e_{P_i}{}' e_{P_j}$

    where $e_i = \varphi\left[\left(\vec{P}_j\right)_{j<i}, \vec{P}_i\right]$ are the (explicit) errors functions implicit to the calibration that defines f_i
  - When a $\vec{P}_j$ is an input to multiple f_i s, then the final $V_{\vec{P}_j}$ is the sum of the propogations from all f_i s
- We can compute "custom risks" with this algorithm
  - Use different parameter sets for tape pasting and IFT, than the ones used for calibration

Big Picture: real life Forex Dupire



Demo: CVA

CVA without collateral

- As presented by Andreasen at Global Derivatives 2014, we calculate CVA by

$$
\begin{aligned}
\text{CVA} &= \mathrm{E}\left[\int_0^T V(t)^+ \delta(\tau - t)dt\right] \\
&= \mathrm{E}\left[\int_0^T V(t)\mathbf{1}_{V(t)>0}\delta(\tau - t)dt\right] \\
&\approx \mathrm{E}\left[\int_0^T V(t)\mathbf{1}_{\tilde{V}(t)>0}\delta(\tau - t)dt\right] \\
&= \mathrm{E}\left[\int_0^T \mathrm{E}_t\left[\int_t^T c(u)du\right]\mathbf{1}_{\tilde{V}(t)>0}\delta(\tau - t)dt\right] \\
&= \mathrm{E}\left[\int_0^T c(u)\int_0^u \mathbf{1}_{\tilde{V}(t)>0}\delta(\tau - t)dtdu\right]
\end{aligned}
$$

- where
  - τ is the default time of the counterparty
  - *c(t)* is the cashflow of the netting set at time *t*
  - $\tilde{V}(t)$ is a LSM regression proxy of the value of the netting set at time *t*
- I.e. instead of simulating the netting set value directly
- … we simulate cashflows and calculate CVA by accumulating future cashflows if
  1. the counterparty has defaulted and
  2. the netting set value is positive to us at default

Demo: CVA

CVA without collateral

- The two methods are economically equivalent, but with this approach we
  - Avoid nested MC simulations
  - Use LSM regression proxies as an approximation, … but only to check positivity!
  - Get a very efficient and accurate CVA implementation

- Live CVA demo …

Demo: CVA

Risk for CVA without collateral

CVA Risk

- Calculating CVA as just described, we have

$$\frac{\partial \text{CVA}}{\partial \tilde{V}} = \text{E}\left[\int_0^T V(t)\frac{\partial \mathbf{1}_{\tilde{V}(t)>0}}{\partial \tilde{V}}\delta(\tau - t)dt\right] = \text{E}\left[\int_0^T V(t)\delta(\tilde{V}(t))\delta(\tau - t)dt\right]$$

- If the proxy is good, $\tilde{V} \approx V$ around zero, the above is approximately zero
  - If the proxy is perfect, $\tilde{V} = V$ around zero, the above is exactly zero
- If the proxy is bad, the above is numerical noise that we want to ignore
- Hence, we want to keep the proxy frozen during risk calculations
- This is done, by not “AD’ing” LSM
  - i.e. to use native doubles – and not kDoubleAds – in the LSM regression implementation
- Live CVA risk demo …

Demo: Capital Requirement for Counterparty Credit Risk

Regulatory-given formula for RWA CCR

- The minimum requirements for regulatory capital is given as a fixed proportion of the Risk-Weighted exposure Amount (RWA aka REA)
- For Counterparty Credit Risk (CRR), RWA is calculated for a netting set, by

$$\mathrm{RWA} = \mathrm{RW} \cdot \mathrm{EAD}$$

$$\mathrm{RW} = \mathrm{LGD} \cdot \left(N\left(\frac{N^{-1}(\mathrm{PD}) + \sqrt{\rho(\mathrm{PD})}\, N^{-1}(0.999)}{\sqrt{1-\rho(\mathrm{PD})}} \right) - \mathrm{PD} \right) \cdot k(\mathrm{PD}, M) \cdot 12.5 \cdot 1.06$$

  - where N is the cumulative normal distribution function
  - PD is the default probability
  - LGD is Loss Given Default
  - ρ is a regulatory-given correlation function of PD: $\rho(x) = 0.12\frac{1-\exp(-50x)}{1-\exp(-50)} + 0.24\left(1 - \frac{1-\exp(-50x)}{1-\exp(-50)}\right)$
  - M is the effective maturity
  - EAD is the Exposure At Default
  - k is a regulatory-given maturity adjustment factor function of PD and M: $k(x,y) = \frac{1+(y-2.5)\,b(x)}{1-1.5b(x)}$

$$b(x) = (0.11852 - 0.05478\ln(x))^2$$

Demo: Capital Requirement for Counterparty Credit Risk

RWA CCR as a function of the expected exposure profile

- RWA CCR is (when using IMM) a function of the expected exposure profile

$$EE(t) = \mathrm{E}\left[(V(t) - K(t))^{+}\right]$$

  - where *V(t)* is the (stochastic) value of the netting set at time *t*
  - and *K(t)* is the (stochastic) value of the collateral (in case of a CSA agreement) at time *t*

- The expected exposure profile enters via EAD and *M*

$$\mathrm{EAD} = \frac{\alpha}{T}\int_0^T \max_{0\le u\le t}(EE(u))\,dt,\ \alpha = 1.4,\ T = \min(T_{\mathrm{mat}}, 1Y)$$

$$M = 1 + \min\left(4, \frac{\int_1^{T_{\mathrm{mat}}} EE(t)\cdot P(0,t)\,dt}{\int_0^1 \max_{0\le u\le t}(EE(u))\cdot P(0,t)\,dt}\right)$$

- Calculating the ***expected exposure*** is *the challenge* in the RWA CCR calculation!

Demo: Capital Requirement for Counterparty Credit Risk

RWA CCR as 2-script setup

- Our approach for RWA CCR can be considered a corollary to our CVA approach
  - We are using the same models as for CVA in the calculation of the EE: beast, MFC, SLV, …
  - We are using/extending a lot of the functionality (trade compression, trade decoration, netting set setup, realtime dynamic model building etc)
  - … and we are using LSM proxies to check for positivity
- When a new trading opportunity occur, we can do a full RWA CCR calculation for the netting set
  - … at the trading desk – based on realtime models
  - For a typical netting set, the calculation takes 5-10 sec and full risk takes about a minute (1.5 – 2x longer with CSA)
- The calculation is done in 2 steps
  - First the expected exposure profile is generated for the netting set
  - Then RWA is calculated with the expected exposure profile (and PD, LGD etc) as input
    - The ***RWA formula is scripted*** in the same way as we script trades and netting sets

Demo: Capital Requirement for Counterparty Credit Risk

RWA CCR without collateral

- For a netting set without CSA, the expected exposure is

$$
\begin{aligned}
EE(t) &= \mathrm{E}\left[(V(t))^{+}\right] \\
&= \mathrm{E}\left[V(t)\mathbf{1}_{V(t)>0}\right] \\
&= \mathrm{E}\left[\mathrm{E}_t\left[\int_t^T c(u)du\right]\mathbf{1}_{V(t)>0}\right] \\
&\approx \mathrm{E}\left[\left(\int_0^T c(u)du - \int_0^t c(u)du\right)\underbrace{\mathbf{1}_{\tilde{V}(t)>0}}_{\text{LSM regression proxy}}\right]
\end{aligned}
$$

- Live RWA CCR demo ...

Demo: Capital Requirement for Counterparty Credit Risk

RWA CCR risk as 2-script risk with macro-level checkpointing

- RWA CCR can be written as

$$\mathrm{RWA} = \mathrm{RWA}(a, EE(a))$$

  - where *a* is a vector of parameters and *EE* is the expected exposure profile

- Then

Risk directly from the RWA formula script

Risk from netting set script enters only via *EE*

$$\frac{d\mathrm{RWA}}{da_i} = \frac{\partial \mathrm{RWA}}{\partial a_i} + \sum_{j=1}^{N_{EE}} \frac{\partial \mathrm{RWA}}{\partial EE_j} \frac{\partial EE_j}{\partial a_i}$$

- This can be calculated using AD with checkpointing in **a single AD calculation with one checkpoint!**

- Live RWA CCR risk demo …

Demo: Capital Requirement for Counterparty Credit Risk 6/6

RWA CCR risk as 2-script risk with macro-level checkpointing

Calculate risk for RWA CCR

Split in 2 scripts/slices:

$$EE = EE(a)$$

1 Calc EE from the EE script (use double – no AD)

$$\mathrm{RWA} = \mathrm{RWA}(a, EE)$$

2 Generate tape for RWA script (use kDoubleAd)

3 Set initial conditions $A_{\mathrm{RWA}} = 1,\ A_{i \neq \mathrm{RWA}} = 0$

Run RWA script bwds to calculate adjoints

4 Store derivatives and wipe tape

5 Generate tape for EE script (use kDoubleAd)

6 Set initial conditions $A_{EE_j} = \dfrac{\partial \mathrm{RWA}}{\partial EE_j},\ A_{i \neq EE_j} = 0$

Run EE script bwds to calculate adjoints

7 Pick derivatives $\dfrac{\partial \mathrm{RWA}}{\partial a_i}$ from the a_i nodes and wipe tape

Demo: Capital Requirement for CCR with collateral

RWA CCR with collateral

- For a 2-way CSA agreement with thresholds H^c and H^p and independent amounts I^c and I^p, collateral is modeled as

$$K(t) = \underbrace{(V(t-\Delta t) - H^c)^+ + I^c}_{\text{to protect us}} - \underbrace{(-V(t-\Delta t) - H^p)^+ - I^p}_{\text{to protect cpty}}$$

- For simplicity we will ignore thresholds and independent amounts and let

$$K(t) = V(t-\Delta t)$$

- The Margin Period of Risk, Δt, is, in case of RWA, given by regulators
  - For a netting set with OTC derivatives having daily margin calls, it is typically 10b
- The Margin Period of Risk makes it non-trivial to extend our approach to XVA and RWA with collateral …

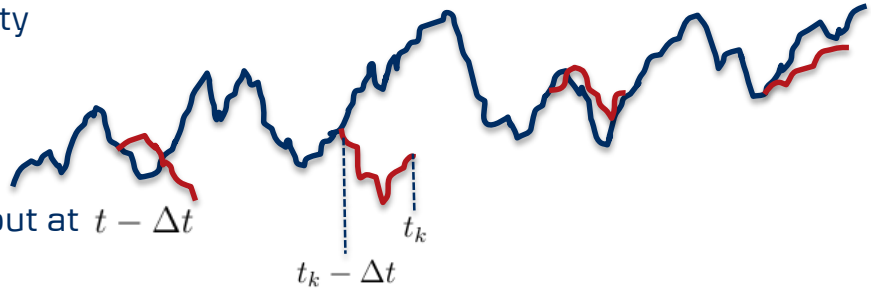
Demo: Capital Requirement for CCR with collateral

RWA CCR with collateral using branching

- However, *Andreasen (2014)* showed that *K* can be taken into account in the expected exposure by

$$\begin{aligned} EE(t) &= \mathrm{E}\left[(V(t) - V(t-\Delta t))^{+}\right] \\ &= \mathrm{E}\left[\left(\int_0^T c(u)du - \int_0^t c(u)du\right)\left(\mathbf{1}_{V(t)-V(t-\Delta t)>0} - \mathbf{1}_{V^B(t)-V(t-\Delta t)>0}\right)\right] \\ &\approx \mathrm{E}\left[\left(\int_0^T c(u)du - \int_0^t c(u)du\right)\left(\mathbf{1}_{\tilde{V}(t)-\tilde{V}(t-\Delta t)>0} - \mathbf{1}_{\tilde{V}^B(t)-\tilde{V}(t-\Delta t)>0}\right)\right] \end{aligned}$$

  - The expected exposure is still calculated by summing notional adjusted cashflows *c(u)*
  - And LSM proxies are still only used to check positivity
  - But 2 proxies are now in play
    - $\tilde{V}$ is calculated on the **main simulation path**
    - $\tilde{V}^B$ is calculated on a **separate path** branched out at $t - \Delta t$



- Live RWA CCR with collateral demo …

Demo: Capital Requirement for CCR with collateral

RWA CCR with collateral using branching

- The extra paths from branching will increase the calculation time by a factor 1.5-2
  - But still fast enough to be used realtime on the front
- By e.g. approximating the *effective maturity* with *time to maturity* in the riskweight for RWA for CCR, we can similarly calculate **KVA** for CCR (and associated risk) in realtime
- In addition, AD can also be used to calculate derivatives with respect to notionals … and allocate capital to individual desks, books, trades etc ("Euler decomposition")