Adjoint Algorithmic Differentiation

Calibration and implicit function theorem

Marc Henrard

Quantitative Research - OpenGamma

Interest Rate Conference, London, March 30th, 2012

Based on:
Adjoint Algorithmic Differentiation: Calibration and Implicit Function Theorem
To appear in *The Journal of Computational Finance*
Available at *SSRN Working Paper series*, 1896329, September 2011.

OpenGamma

Algorithmic Differentiation

OpenGamma

Introduction

- Quantitative finance time: price.
- CPU time: greeks (derivatives with respect to the input).
- Computing derivatives is known in computer science under the name of Algorithmic Differentiation.
- Algorithmic Differentiation comes in two modes:
  1. Forward/standard
  2. Reverse/adjoint
- The Adjoint mode is often the most efficient in finance.
- It can also be used efficiently when an equation solving problem (calibration) is part of the algorithm.

Introduction

- Quantitative finance time: price.
- CPU time: greeks (derivatives with respect to the input).
- Computing derivatives is known in computer science under the name of Algorithmic Differentiation.
- Algorithmic Differentiation comes in two modes:
  1. Forward/standard
  2. Reverse/adjoint
- The Adjoint mode is often the most efficient in finance.
- It can also be used efficiently when an equation solving problem (calibration) is part of the algorithm.

OpenGamma

Introduction

- Quantitative finance time: price.
- CPU time: greeks (derivatives with respect to the input).
- Computing derivatives is known in computer science under the name of Algorithmic Differentiation.
- Algorithmic Differentiation comes in two modes:
  1. Forward/standard
  2. Reverse/adjoint
- The Adjoint mode is often the most efficient in finance.
- It can also be used efficiently when an equation solving problem (calibration) is part of the algorithm.

OpenGamma

Introduction

- Quantitative finance time: price.
- CPU time: greeks (derivatives with respect to the input).
- Computing derivatives is known in computer science under the name of Algorithmic Differentiation.
- Algorithmic Differentiation comes in two modes:
  1. Forward/standard
  2. Reverse/adjoint
- The Adjoint mode is often the most efficient in finance.
- It can also be used efficiently when an equation solving problem (calibration) is part of the algorithm.

OpenGamma

Introduction

- Quantitative finance time: price.
- CPU time: greeks (derivatives with respect to the input).
- Computing derivatives is known in computer science under the name of Algorithmic Differentiation.
- Algorithmic Differentiation comes in two modes:
  1. Forward/standard
  2. Reverse/adjoint
- The Adjoint mode is often the most efficient in finance.
- It can also be used efficiently when an equation solving problem (calibration) is part of the algorithm.

Algorithmic Differentiation

Mathematics (1): Differentiation ratio

The most often used approximation for derivative computation is

Approximation (Differentiation ratio)

One side:

$$D_{x_i} f(x) \simeq \frac{f(x+\epsilon_i) - f(x)}{|\epsilon_i|}.$$

Two sides (or symmetrical):

$$D_{x_i} f(x) \simeq \frac{f(x+\epsilon_i) - f(x-\epsilon_i)}{2|\epsilon_i|}.$$

OpenGamma

Mathematics (2): Chain rule

The main piece of mathematics for Algorithmic Differentiation is

Theorem (Chain rule)

For two differentiable functions f and g, one has

$$D(f \circ g)(x) = Df(g(x)) \cdot Dg(x).$$

Computer: function

The starting point is the algorithm for

$$z = f(a).$$

The function input are: $a = a[0 : p_a]$ (dimension $p_a + 1$).
The function output is z (dimension 1).

The program is
Initialisation $[j = -p_a : 0]$ $b[j] = a[j + p_a]$

Computer: function

The starting point is the algorithm for

$$z = f(a).$$

The function input are: $a = a[0 : p_a]$ (dimension $p_a + 1$).
The function output is z (dimension 1).

The program is

Initialisation	$[j = -p_a : 0]$	$b[j] = a[j + p_a]$
Algorithm	$[j = 1 : p_b]$	$b[j] = g_j(b[-p_a : j - 1])$

Computer: function

The starting point is the algorithm for

$$z = f(a).$$

The function input are: $a = a[0 : p_a]$ (dimension $p_a + 1$).
The function output is z (dimension 1).

The program is

Initialisation	$[j = -p_a : 0]$	$b[j] = a[j + p_a]$
Algorithm	$[j = 1 : p_b]$	$b[j] = g_j(b[-p_a : j-1])$
Value		$z = b[p_b]$

This algorithm is supposed to be implemented.

OpenGamma

Computer: Algorithmic Differentiation

Goal: compute the derivatives of z (dimension 1) with respect to a_i (dimension $p_a + 1$):

$$\frac{\partial}{\partial a_i} f(a) = \frac{\partial}{\partial a_i} z.$$

The emphasis can be put on *with respect to* a_i (standard) or on *of* z (adjoint).

Computer: AD forward/standard

Goal: derivatives with respect to a_i: $\dfrac{\partial}{\partial a_i} b[j] = \dot{b}[j][i]$.

Computer: AD forward/standard

Goal: derivatives with respect to a_i: $\frac{\partial}{\partial a_i} b[j] = \dot{b}[j][i]$.

The program is

	Function	Derivatives: $[i = 0 : p_a]$
$[j = -p_a : 0]$	$b[j] = a[j + p_a]$	
$[j = 1 : p_b]$	$b[j] = g_j(b[-p_a : j - 1])$	
	$z = b[p_b]$	

OpenGamma

Computer: AD forward/standard

Goal: derivatives with respect to a_i: $\dfrac{\partial}{\partial a_i} b[j] = \dot{b}[j][i]$.

The program is

Function		Derivatives: $[i = 0 : p_a]$
$[j = -p_a : 0]$	$b[j] = a_{j+p_a}$	$\dot{b}[j][i] = \delta_{j+p_a,i}$
$[j = 1 : p_b]$	$b[j] = g_j(b[-p_a : j-1])$	$\dot{b}[j][i] = \displaystyle\sum_{k=-p_a}^{j-1} \frac{\partial}{\partial b_k} g_j \cdot \frac{\partial}{\partial a_i} b[k]$
		$= \displaystyle\sum_{k=-p_a}^{j-1} \frac{\partial}{\partial b_k} g_j \cdot \dot{b}[k][i]$
	$z = b[p_b]$	$\frac{\partial}{\partial a_i} z = \dot{b}[p_b][i]$

OpenGamma

Computer: AD forward/standard

Goal: derivatives with respect to a_i: $\frac{\partial}{\partial a_i} b[j] = \dot{b}[j][i]$.

The program is

Function		Derivatives: $[i = 0 : p_a]$
$[j = -p_a : 0]$	$b[j] = a_{j+p_a}$	$\dot{b}[j][i] = \delta_{j+p_a,i}$
$[j = 1 : p_b]$	$b[j] = g_j(b[-p_a : j-1])$	$\dot{b}[j][i] = \sum_{k=-p_a}^{j-1} \frac{\partial}{\partial b_k} g_j \cdot \frac{\partial}{\partial a_i} b[k]$
		$= \sum_{k=-p_a}^{j-1} \frac{\partial}{\partial b_k} g_j \cdot \dot{b}[k][i]$
	$z = b[p_b]$	$\frac{\partial}{\partial a_i} z = \dot{b}[p_b][i]$

OpenGamma

Computer: AD forward/standard

Goal: derivatives with respect to a_i: $\frac{\partial}{\partial a_i} b[j] = \dot{b}[j][i]$.

The program is

Function		Derivatives: $[i = 0 : p_a]$
$[j = -p_a : 0]$	$b[j] = a_{j+p_a}$	$\dot{b}[j][i] = \delta_{j+p_a,i}$
$[j = 1 : p_b]$	$b[j] = g_j(b[-p_a : j-1])$	$\dot{b}[j][i] = \sum_{k=-p_a}^{j-1} \frac{\partial}{\partial b_k} g_j \cdot \frac{\partial}{\partial a_i} b[k]$
		$= \sum_{k=-p_a}^{j-1} \frac{\partial}{\partial b_k} g_j \cdot \dot{b}[k][i]$
	$z = b[p_b]$	$\frac{\partial}{\partial a_i} z = \dot{b}[p_b][i]$

Computer: AD forward/standard

Goal: derivatives with respect to a_i: $\frac{\partial}{\partial a_i} b[j] = \dot{b}[j][i]$.

The program is

Function		Derivatives: $[i = 0 : p_a]$
$[j = -p_a : 0]$	$b[j] = a_{j+p_a}$	$\dot{b}[j][i] = \delta_{j+p_a,i}$
$[j = 1 : p_b]$	$b[j] = g_j(b[-p_a : j-1])$	$\dot{b}[j][i] = \sum_{k=-p_a}^{j-1} \frac{\partial}{\partial b_k} g_j \cdot \frac{\partial}{\partial a_i} b[k] = \sum_{k=-p_a}^{j-1} \frac{\partial}{\partial b_k} g_j \cdot \dot{b}[k][i]$
	$z = b[p_b]$	$\frac{\partial}{\partial a_i} z = \dot{b}[p_b][i]$

OpenGamma

Computer: AD forward/standard

Goal: derivatives with respect to a_i: $\frac{\partial}{\partial a_i} b[j] = \dot{b}[j][i]$.

The program is

Function		Derivatives: $[i = 0 : p_a]$
$[j = -p_a : 0]$	$b[j] = a_{j+p_a}$	$\dot{b}[j][i] = \delta_{j+p_a,i}$
$[j = 1 : p_b]$	$b[j] = g_j(b[-p_a : j-1])$	$\dot{b}[j][i] = \sum_{k=-p_a}^{j-1} \frac{\partial}{\partial b_k} g_j \cdot \frac{\partial}{\partial a_i} b[k] = \sum_{k=-p_a}^{j-1} \frac{\partial}{\partial b_k} g_j \cdot \dot{b}[k][i]$
	$z = b[p_b]$	$\frac{\partial}{\partial a_i} z = \dot{b}[p_b][i]$

There are p_a *line of code* for each line in f.

$$\text{Cost}(P + D) \leq (1 + 1.5 p_a)\,\text{Cost}(P)$$

Computer: AD reverse/adjoint

Goal: derivatives of z: $\frac{\partial}{\partial b[j]} z = \bar{b}[j]$.

Init	$[j = -p_a : 0]$	$b[j] = a[j + p_a]$
Algorithm	$[j = 1 : p_b]$	$b[j] = g_j(b[-p_a : j - 1])$
Value		$z = b[p_b]$

Computer: AD reverse/adjoint

Goal: derivatives of z: $\frac{\partial}{\partial b[j]} z = \bar{b}[j]$.

Init	$[j = -p_a : 0]$	$b[j]$	$= a[j + p_a]$
Algorithm	$[j = 1 : p_b]$	$b[j]$	$= g_j(b[-p_a : j-1])$
Value		z	$= b[p_b]$
Value		$\bar{z}$	$= 1.0$
Value		$\bar{b}[p_b]$	$= 1.0$
Algorithm	$[j = p_b - 1 : -1 : -p_a]$	$\bar{b}[j]$	$= \sum_{k=j+1}^{p_b} \frac{\partial}{\partial b_k} z \frac{\partial}{\partial b_j} b_k$
			$= \sum_{k=j+1}^{p_b} \bar{b}[k] \frac{\partial}{\partial b_j} g_k$
Init	$[i = 0 : p_a]$	$\frac{\partial}{\partial a_i} z$	$= \bar{b}[i - p_a + 1]$

OpenGamma

Computer: AD reverse/adjoint

Goal: derivatives of z: $\frac{\partial}{\partial b[j]} z = \bar{b}[j]$.

Init	$[j = -p_a : 0]$	$b[j]$	$= a[j + p_a]$
Algorithm	$[j = 1 : p_b]$	$b[j]$	$= g_j(b[-p_a : j-1])$
Value		z	$= b[p_b]$
Value		$\bar{z}$	$= 1.0$
Value		$\bar{b}[p_b]$	$= 1.0$
Algorithm	$[j = p_b - 1 : -1 : -p_a]$	$\bar{b}[j]$	$= \sum_{k=j+1}^{p_b} \frac{\partial}{\partial b_k} z \frac{\partial}{\partial b_j} b_k = \sum_{k=j+1}^{p_b} \bar{b}[k] \frac{\partial}{\partial b_j} g_k$
Init	$[i = 0 : p_a]$	$\frac{\partial}{\partial a_i} z$	$= \bar{b}[i - p_a + 1]$

There is one *line of code* for each line in f.

$$\mathrm{Cost}(P + D) \leq \omega_A \, \mathrm{Cost}(P) \qquad \omega_A \in [3, 4]$$

AD adjoint: advantages/drawbacks

Required: algorithmic differentiation for (almost) all functions g_j.
Bottom-up approach: it can be implemented for an algorithm only if all the components are already implemented.
When it is there, it can be very fast!

AD adjoint: example

The function (with 4 inputs)

$$z = (a_0 + \exp(a_1))\,(\sin(a_2) + \cos(a_3)) + (a_1)^2 + a_3.$$

```
public double f(double[] a) {
  double b1 = a[0] + Math.exp(a[1]);
  double b2 = Math.sin(a[2]) + Math.cos(a[3]);
  double b3 = b1 * b2 + Math.pow(a[1], 2) + a[3];
  return b3;
}
```

OpenGamma

AD adjoint: example

```
public double f(double[] a, double[] aBar) {
  // Forward sweep
  double b1 = a[0] + Math.exp(a[1]);
  double b2 = Math.sin(a[2]) + Math.cos(a[3]);
  double b3 = b1 * b2 + Math.pow(a[1], 2) + a[3];
  // Backward sweep
  double b3Bar = 1.0;
  double b2Bar = b1 * b3Bar;
  double b1Bar = b2 * b3Bar + 0.0 * b2Bar;
  aBar[3] = 1.0 * b3Bar - Math.sin(a[3]) * b2Bar;
  aBar[2] = Math.cos(a[2]) * b2Bar;
  aBar[1] = 2 * a[1] * b3Bar + Math.exp(a[1]) * b1Bar;
  aBar[0] = 1.0 * b1Bar;
  return b3;
}
```

AD adjoint: example

Example implementation:
1,000,000 f - value: 79 ms
1,000,000 f - value and 4 derivatives (adjoint): 149 ms
1,000,000 f - value and 4 derivatives (adjoint) - exp$(a[1])$ stored: 126 ms

AD adjoint: SABR swaption

```
public double pv(Swaption swpt, SABRData sabr) {
 double maturity = swpt.getMaturityTime();
 double expiry = swpt.getTimeToExpiry();
 FixedCouponSwap swap = swpt.getUnderlyingSwap();
 // Forward sweep
 double fwd = PRC.visit(swap, sabr);
 double pvbp = SwapMethod.pvbp(swap, sabr);
 double strike = SwapMethod.couponEquivalent(swap, pvbp, sabr);
 EuropeanVanillaOption option = new EuropeanVanillaOption(strike,
 double volatility = sabr.vol(expiry, maturity, strike, fwd);
 BlackData dataBlack = new BlackData(fwd, 1.0, volatility);
 double black = black.price(dataBlack);
 double pv = pvbp * black;
 return pv
}
```

AD adjoint: SABR swaption - curve sensitivity

```
public IRSensitivity pvSensi(Swaption swpt, SABRData sabr) {
 double maturity = swpt.getMaturityTime();
 double expiry = swpt.getTimeToExpiry();
 FixedCouponSwap swap = swpt.getUnderlyingSwap();
 double fwd = PRC.visit(swap, sabr);
 double pvbp = SwapMethod.pvbp(swap, sabr);
 double strike = SwapMethod.couponEquivalent(swap, pvbp, sabr);
 EuropeanVanillaOption option = new EuropeanVanillaOption(strike,
 double[] volAdj = sabr.volAdj(expiry, maturity, strike, fwd);
 BlackData dataBlack = new BlackData(forward, 1.0, volAdj[0]);
 double[] bsAdj = black.priceAdj(option, dataBlack);
 double pv = pvbp * bsAdj[0]; double pvBar = 1.0;
 double volBar = pvbp * bsAdj[2] * pvBar;
 double pvbpBar = bsAdj[0] * pvBar;
 double fwdBar = pvbp * bsAdj[1] * pvBar + volAdj[1] * volBar;
 IRSensi pvbpDr = SwapMethod.pvbpSensi(swap, sabr);
 IRSensi fwdDr = PRSC.visit(swap, sabr);
 return pvbpDr.mult(pvbpBar).plus(fwdDr.mult(fwdBar));
}
```

AD adjoint: example (financial functions)

The standard option price in the Black framework (`BlackPriceFunction`):
1,000,000 - value: 156 ms
1,000,000 - value + 3 derivatives: 176 ms

The price of European swaptions (5Y quarterly) with physical delivery for a swap rate following a SABR model (Hagan et al. approximation) in a multi-curve framework (`SwaptionPhysicalFixedIborSABRMethod`):
1,000 - swaptions SABR (price): 20 ms
1,000 - swaptions SABR (price + 20+24 delta + 3 vega): 98 ms

OpenGamma

Calibration

- the price of an *exotic instrument* is related to a specific basket of *vanilla instruments*;
- the price of these vanilla instruments is computed in a given *base model*;
- the complex model parameters are calibrated to fit the vanilla option prices from the base model. This step is usually done through a generic numerical equation solver; and
- the exotic instrument is then priced with the calibrated complex model.
- Goal: derivative of the exotic instrument price with respect to the base model parameters.

Calibration

- the price of an *exotic instrument* is related to a specific basket of *vanilla instruments*;
- the price of these vanilla instruments is computed in a given *base model*;
- the complex model parameters are calibrated to fit the vanilla option prices from the base model. This step is usually done through a generic numerical equation solver; and
- the exotic instrument is then priced with the calibrated complex model.
- Goal: derivative of the exotic instrument price with respect to the base model parameters.

OpenGamma

Greeks through calibration

Input C: yield curves
Input Θ: parameters for the base model (SABR parameters).
Intermediary value Φ: parameters for the calibrated model.
$\text{NPV}^{\text{Vanilla}}_{\text{Base}}$: pv of vanilla instruments in base model.
$\text{NPV}^{\text{Vanilla}}_{\text{Calibrated}}$: pv of vanilla instruments in calibrated model.
$\text{NPV}^{\text{Exotic}}_{\text{Calibrated}}$: pv of exotic instrument in calibrated model.

The calibration procedure (perfect calibration) is

$$0 = f(C, \Theta, \Phi) = \text{NPV}^{\text{Vanilla}}_{\text{Base}}(C, \Theta) - \text{NPV}^{\text{Vanilla}}_{\text{Calibrated}}(C, \Phi).$$

OpenGamma

AD: equation solving

The calibration problem looks like:

$$\begin{array}{ccl} b & = & g_1(a) \\ c & \text{s. t.} & g_2(b,c) = 0 \\ z & = & g_3(c) \end{array}$$

with $g_1 : \mathbb{R}^{p_a} \to \mathbb{R}^{p_b}$, $g_2 : \mathbb{R}^{p_b} \times \mathbb{R}^{p_c} \to \mathbb{R}^{p_c}$ and $g_3 : \mathbb{R}^{p_c} \to \mathbb{R}^{p_z}$.
We know how to deal with

$$\begin{array}{ccl} b & = & g_1(a) \\ c & = & g_4(b) \\ z & = & g_3(c) \end{array}$$

OpenGamma

Mathematics (2): Implicit function theorem

Theorem (Implicit function theorem)

Under mild regularity conditions on f, if

$$f(x_0, y_0) = 0$$

and if $D_y f(x_0, y_0)$ is invertible, then, near x_0, there exists a (implicit) function g such that $f(x, g(x)) = 0$, g is differentiable in x_0 and

$$D_x g(x_0) = -\left(D_y f(x_0, y_0)\right)^{-1} D_x f(x_0, y_0).$$

The term *exists* is in the sense of the mathematicians, not of the computer scientists!

OpenGamma

Implicit AAD

The elements of $\mathbb{R}^p$ are represented by column vectors. The derivative $Df(a) \in \mathcal{L}(\mathbb{R}^{p_a}, \mathbb{R}^{p_z})$ is represented by a $p_z \times p_a$ matrix (p_z rows, p_a columns).
The adjoint version of the algorithm is

$$
\begin{aligned}
\bar{z} &= I \quad \text{(with } I \text{ the } p_z \times p_z \text{ identity)} \\
\bar{c} &= (D_c g_3(c))^T \bar{z} \\
\bar{b} &= (D_b g_4(b))^T \bar{c} = -\left((D_c g_2(b,c))^{-1} D_b g_2(b,c) \right)^T \bar{c} \\
\bar{a} &= (D_a g_1(a))^T \bar{b}.
\end{aligned}
$$

OpenGamma

Greeks through calibration

Calibration: $\Phi = \Phi(C, \Theta)$.
Using the calibration:

$$\text{NPV}^{\text{Exotic}}_{\text{Base}}(C, \Theta) = \text{NPV}^{\text{Exotic}}_{\text{Calibrated}}(C, \Phi(C, \Theta))$$

The quantities of interest are

$$D_C \text{NPV}^{\text{Exotic}}_{\text{Base}} \text{ and } D_\Theta \text{NPV}^{\text{Exotic}}_{\text{Base}}.$$

Through composition we have

$$D_C \text{NPV}^{\text{Exotic}}_{\text{Base}} = D_C \text{NPV}^{\text{Exotic}}_{\text{Calibrated}}(C, \Phi) + D_\Phi \text{NPV}^{\text{Exotic}}_{\text{Calibrated}}(C, \Phi) D_C \Phi(C, \Theta),$$

and

$$D_\Theta \text{NPV}^{\text{Exotic}}_{\text{Base}}(C, \Theta) = D_\Phi \text{NPV}^{\text{Exotic}}_{\text{Calibrated}}(C, \Phi) D_\Theta \Phi(C, \Theta)$$

where $D_C \Phi$ and $D_\Theta \Phi$ are unknown.

Greeks through calibration

Using the implicit function theorem, the function Φ is differentiable and its derivatives can be computed from the derivative of f:

$$D_\Theta \Phi(C,\Theta) = \left(D_\Phi \mathsf{NPV}^{\mathsf{Vanilla}}_{\mathsf{Calibrated}}(C,\Phi) \right)^{-1} D_\Theta \mathsf{NPV}^{\mathsf{Vanilla}}_{\mathsf{Base}}(C,\Theta)$$

and

$$\begin{aligned} D_C \Phi(C,\Theta) &= \left(D_\Phi \mathsf{NPV}^{\mathsf{Vanilla}}_{\mathsf{Calibrated}}(C,\Phi) \right)^{-1} \\ &\quad \left(D_C \mathsf{NPV}^{\mathsf{Vanilla}}_{\mathsf{Base}}(C,\Theta) - D_C \mathsf{NPV}^{\mathsf{Vanilla}}_{\mathsf{Calibrated}}(C,\Phi) \right). \end{aligned}$$

OpenGamma

Amortised swaptions in LMM

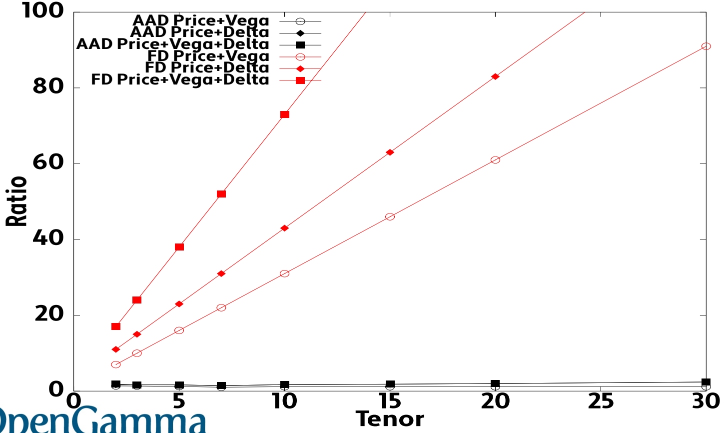
Exotic: 10Y amortised European swaption (yearly amortisation)
Vanilla: 10 vanilla swaptions with tenors between 1Y and 10Y
Base model: SABR model
Calibrated model: two-factor LMM with displaced diffusion.
Calibration: for each year the weights of the 4 parameters are fixed; weights multiplied by a common factor (Φ).

Amortised swaptions in LMM

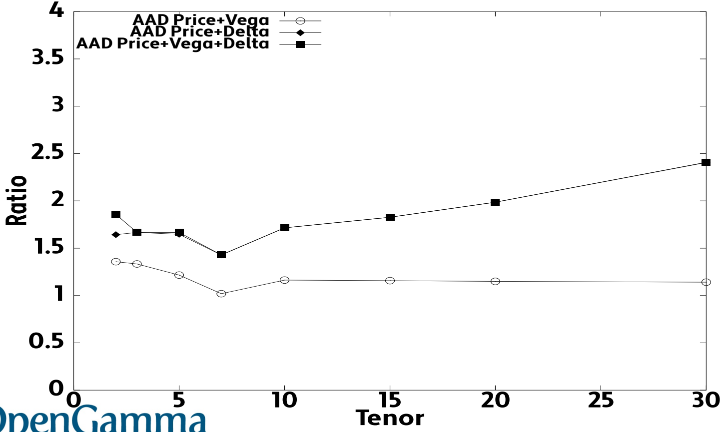
Exotic: 10Y amortised European swaption (yearly amortisation)
Vanilla: 10 vanilla swaptions with tenors between 1Y and 10Y
Base model: SABR model
Calibrated model: two-factor LMM with displaced diffusion.
Calibration: for each year the weights of the 4 parameters are fixed; weights multiplied by a common factor (Φ).

Risk type	Approach	Price time	Risks time	Total
SABR	FD	1.00	30×1.00	31.00
SABR	AAD	1.00	0.18	1.18
Curve	FD	1.00	42×1.00	43.00
Curve	AAD	1.00	0.74	1.74
Curve and SABR	FD	1.00	72×1.00	73.00
Curve and SABR	AAD	1.00	0.75	1.75

Amortised swaptions in LMM



Amortised swaptions in LMM



OpenGamma

Algorithmic Differentiation

Conclusion

- Algorithmic differentiation: price and derivatives (greeks) at the computation cost of less than 4 times the cost of one price.
- The fast execution time comes with a cost: a (slightly) longer development time (the code length is doubled, not the development time).
- Calibrations require equation solving. With the implicit function approach: only the adjoint methods for the prices, not for the equation solver, are required.
- Prices including calibration: ratio price and derivatives computation cost to the price computation cost can be below two.

OpenGamma

Conclusion

- Algorithmic differentiation: price and derivatives (greeks) at the computation cost of less than 4 times the cost of one price.
- The fast execution time comes with a cost: a (slightly) longer development time (the code length is doubled, not the development time).
- Calibrations require equation solving. With the implicit function approach: only the adjoint methods for the prices, not for the equation solver, are required.
- Prices including calibration: ratio price and derivatives computation cost to the price computation cost can be below two.