Crash Course on Adjoint Algorithmic Differentiation (AAD)

Luca Capriotti

MTH 9871: Advanced Comp Methods Finance

Fall 2024

Section 1

Differentiating Programs

What is Algorithmic Differentiation?

- Algorithmic or (Automatic) Differentiation (AD) is a set of programming techniques first introduced in the early 60's aimed at computing accurately and efficiently the derivatives of a function given in the form of computer programs.
- The main idea underlying AD is that any such program can be interpreted as the composition of functions each of which is in turn a composition of basic arithmetic (addition, multiplication etc.), and intrinsic operations (logarithm, exponential, etc.).
- Hence, it is possible to calculate the derivatives of the outputs of the program with respect to its inputs by applying mechanically the rules of differentiation.
- This makes it possible to generate *automatically* a computer program that evaluates efficiently and with machine precision accuracy the derivatives of the function.

What is Algorithmic Differentiation? (cont'd)

- AD is distinct from
  - *Symbolic differentiation*, which works by symbolic manipulation of closed-form expressions using rules of differential calculus.
  - Numerical differentiation (the method of finite differences).
- What makes AD particularly attractive when compared to the methods above is its computational efficiency.
- In fact, AD aims at exploiting the information on the structure of the computer function, and on the dependencies between its various parts, in order to optimize the calculation of the sensitivities.
- AD comes in two main flavors, Tangent and Adjoint mode, which are characterized by different (complementary) computational complexities.

Computations of Jacobians

- Let us consider a computer program with n inputs, $x = (x_1, \ldots, x_n)$ and and m outputs $y = (y_1, \ldots, y_m)$, that is defined by a composition of arithmetic and non-linear (intrinsic) operations. Such a program can be seen as a function of the form $F : \mathbb{R}^n \to \mathbb{R}^m$,

$$(y_1, \ldots, y_m)^t = F(x_1, \ldots, x_n) \ . \tag{1}$$

- In its simplest form, AD aims at producing a code evaluating the sensitivities of the outputs of the original program with respect to its inputs, i.e., at calculating the Jacobian of the function F

$$J_{ij} = \frac{\partial F_i(x)}{\partial x_j} \ , \tag{2}$$

with $F_i(x) = y_i$.

Computations of Jacobians: Tangent mode

- The tangent mode of AD allows the calculation of the function F and of its Jacobian with a cost – relative to the one for F – which can be shown, under a standard computational complexity model, to be bounded by a small constant, ω_T, times the number of *independent* variables, namely

$$\frac{\mathrm{Cost}[F\&J]}{\mathrm{Cost}[F]} \leq \omega_T \, n \, . \tag{3}$$

- The value of the constant ω_T can be also bounded using a model of the relative cost of algebric operations, non linear unary functions, and memory access. This analysis gives $\omega_T \in [2, 5/2]$.
- The form of the result (3) appears quite natural as it is the same computational complexity of evaluating the Jacobian by perturbing one input variable at a time, repeating the calculation of the function, and forming the appropriate finite difference estimators.

Computations of Jacobians: Tangent mode

- Consistently with Eq. (3), the tangent mode of AD provides the derivatives of all the m components of the output vector y with respect to a single input x_j, i.e., a single column of the Jacobian (2), at a cost which is independent of the number of dependent variables, and bounded by a small constant, ω_T.
- In fact, the same holds true for *any linear combination* of the columns of the Jacobian, $\mathcal{L}_c(J)$, namely

$$\frac{\mathrm{Cost}[F \,\&\, \mathcal{L}_c(J)]}{\mathrm{Cost}[F]} \leq \omega_T \ . \tag{4}$$

- We can therefore state the result on the computational complexity of the tangent mode as it follows.

Algorithmic Differentiation: Tangent mode

- Consider a function
$$Y = \texttt{FUNCTION}(X)$$
mapping a vector X in $\mathbb{R}^n$ in a vector Y in $\mathbb{R}^m$.
- The execution time of its Tangent counterpart
$$\dot{Y} = \texttt{FUNCTION\_d}(X, \dot{X})$$
(with suffix _d for "dot") calculating the linear combination of the columns of the Jacobian of the function:
$$\dot{Y}_j = \sum_{i=1}^{n} \dot{X}_i \frac{\partial Y_j}{\partial X_i},$$
with $j = 1, \ldots, m$, is bounded by
$$\frac{\text{Cost}[\texttt{FUNCTION\_d}]}{\text{Cost}[\texttt{FUNCTION}]} \leq \omega_T \tag{5}$$
with $\omega_T \in [2, 5/2]$.

Computations of Jacobians: Adjoint mode

- The adjoint mode of AD, or AAD, is characterized by a computational cost of the form

$$\frac{\text{Cost}[F \,\&\, J]}{\text{Cost}[F]} \leq \omega_A \, m \ , \tag{6}$$

with $\omega_A \in [3, 4]$, i.e., AAD allows the calculation of the function F and of its Jacobian with a cost – relative to the one for F – which is bounded by a small constant times the number of *dependent* variables.

- AAD provides the full gradient of a scalar ($m = 1$) function at a cost which is just a small constant times the cost of evaluating the function itself.
- Remarkably such relative cost is *independent* of the number of components of the gradient. Evaluating the same gradient by finite differences or by means of the tangent mode (3), scaling linearly with the number n of sensitivities!

Computations of Jacobians: Adjoint mode

- For vector valued functions, AAD provides the gradient of arbitrary linear combinations of the rows of the Jacobian, $\mathcal{L}_r(J)$, at the same computational cost of a single row, namely

$$\frac{\text{Cost}[F \,\&\, \mathcal{L}_r(J)]}{\text{Cost}[F]} \leq \omega_A. \tag{7}$$

  This clearly makes the adjoint mode particularly well-suited for the calculation of (linear combinations of) the rows of the Jacobian matrix (2).
- We can therefore state the result on the computational complexity of the adjoint mode as it follows.

Algorithmic Differentiation: Adjoint mode

- The execution time of the Adjoint counterpart of

$$Y = \texttt{FUNCTION}(X),$$

namely,

$$\bar{X} = \texttt{FUNCTION\_b}(X, \bar{Y})$$

(with suffix _b for "backward" or "bar") calculating the linear combination of the rows of the Jacobian of the function:

$$\bar{X}_i = \sum_{j=1}^{m} \bar{Y}_j \frac{\partial Y_j}{\partial X_i},$$

with $i = 1, \ldots, n$, is bounded by

$$\frac{\mathrm{Cost}[\texttt{FUNCTION\_b}]}{\mathrm{Cost}[\texttt{FUNCTION}]} \leq \omega_A$$

with $\omega_A \in [3, 4]$.

Algorithmic Differentiation: Tangent vs Adjoint mode

Given the results above:

- The Tangent mode is particularly well suited for the calculation of (linear combinations of) the columns of the Jacobian matrix.
- Instead, the Adjoint mode is particularly well-suited for the calculation of (linear combinations of) the rows of the Jacobian matrix .
- In particular, the Adjoint mode provides the full gradient of a scalar ($m = 1$) function at a cost which is just a small constant times the cost of evaluating the function itself. Remarkably such relative cost is *independent* of the number of components of the gradient.
- When the full Jacobian is required, the Adjoint mode is likely to be more efficient than the Tangent mode when the number of independent variables is significantly larger than the number of the dependent ones ($m \ll n$). Or viceversa.

Computational Graphs

Let us consider the function $F : \mathbb{R}^2 \to \mathbb{R}^3$,

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 2 \log x_1 x_2 + 2 \sin x_1 x_2 \\ 4 \log^2 x_1 x_2 + \cos x_1 x_3 - 2 x_3 - x_2 \end{pmatrix} .$$

- Given a value of the input x, the output y is calculated by means of a sequence of computer instructions and can be represented in terms of *internal variables*, $w_1, \ldots, w_N$,

$$\begin{aligned} w_i &= x_i , \quad i = 1, \ldots, n \\ w_i &= \Phi_i(\{w_j\}_{j \prec i}) , \quad i = n+1, \ldots, N . \end{aligned}$$

- Here the first n variables are copies of the input ones, and the others are given by a sequence of consecutive assignments; the symbol $\{w_j\}_{j \prec i}$ indicates the set of internal variables w_j, with $j < i$, such that w_i depends *explicitly* on w_j; the functions Φ_i represent a composition of elementary or intrinsic operations. The last m internal variables are the output.

Computational Graphs

- This representation is by no means unique, and can be constructed in a variety of ways. However, it is a useful abstraction in order to introduce the mechanism of AD.
- For instance, for the function in our example one can represent the internal calculations as follows:

$$
\begin{gathered}
w_1 = x_1 \ , \ w_2 = x_2 \ , \ w_3 = x_3 \ , \\
\downarrow \\
w_4 = 2 \log w_1 w_2 \ , \\
w_5 = 2 \sin w_1 w_2 \ , \\
w_6 = \cos w_1 w_3 \ , \\
w_7 = 2 w_3 + w_2 \ , \\
\downarrow \\
y_1 = w_8 = w_4 + w_5 \ , \\
y_2 = w_9 = w_4^2 + w_6 - w_7 \ .
\end{gathered}
$$

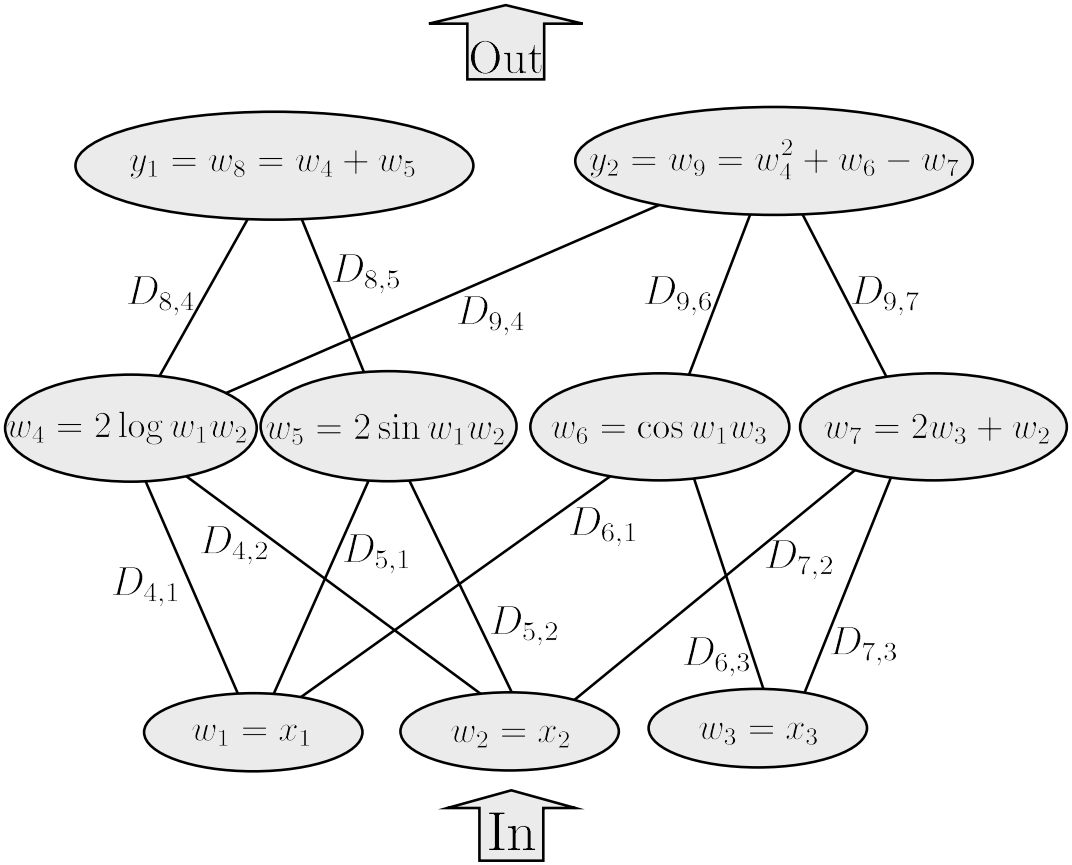
Computational Graphs

- In general, a computer program contains internal controls that alter the calculations performed according to different criteria. Nevertheless, the previous relations are an accurate representation on how the program is executed for a given value of the input vector x, i.e., for a given instance of the internal controls.
- The sequence of instructions can be effectively represented by means of a *computational graph* with nodes given by the internal variables w_i, and connecting arcs between explicitly dependent variables.
- To each arc of the computational graph, say connecting node w_i and w_j with $j < i$, it is possible to associate the *arc derivative*

$$D_{i,j} = \frac{\partial \Phi_i(\{w_k\}_{k \prec i})}{\partial w_j} \ .$$

Crucially, these derivatives can be calculated in an automatic fashion by applying mechanically the rules of differentiation instruction by instruction.

Computational Graphs



Tangent Mode

- Once the program implementing $F(x)$ is represented in terms of elementary instructions the calculation of the gradient of each of its m components,

$$\nabla F_i(x) = (\partial_{x_1} F_i(x), \partial_{x_2} F_i(x), \ldots, \partial_{x_n} F_i(x))^t \ ,$$

simply involves the application of the chain rule of differentiation.

- Starting from the independent variables, one obtains the *tangent mode* of AD

$$\nabla w_i = e_i \ , \quad i = 1, \ldots, n$$
$$\nabla w_i = \sum_{j \prec i} D_{i,j} \nabla w_j \ , \quad i = n+1, \ldots, N$$

where $e_1, e_2, \ldots, e_n$ are the vectors of the canonical basis in $\mathbb{R}^n$, and $D_{i,j}$ are the local derivatives.

Tangent Mode

$$\nabla w_1 = (1,0,0)^t \ , \ \nabla w_2 = (0,1,0)^t \ , \ \nabla w_3 = (0,0,1)^t \ ,$$

$$\downarrow$$

$$D_{4,1} = 2w_2/(w_1 w_2), \quad D_{4,2} = 2w_1/(w_1 w_2),$$
$$\nabla w_4 = D_{4,1}\nabla w_1 + D_{4,2}\nabla w_2 \ ,$$

$$D_{5,1} = 2w_2 \cos w_1 w_2, \quad D_{5,2} = 2w_1 \cos w_1 w_2,$$
$$\nabla w_5 = D_{5,1}\nabla w_1 + D_{5,2}\nabla w_2 \ ,$$

$$D_{6,1} = -w_3 \sin w_1 w_3, \quad D_{6,3} = -w_1 \sin w_1 w_3,$$
$$\nabla w_6 = D_{6,1}\nabla w_1 + D_{6,3}\nabla w_3 \ ,$$

$$D_{7,2} = 1, \quad D_{7,3} = 2,$$
$$\nabla w_7 = D_{7,2}\nabla w_2 + D_{7,3}\nabla w_3 \ ,$$

$$\downarrow$$

Tangent Mode

$$\downarrow$$

$$D_{8,4} = 1, \quad D_{8,5} = 1,$$
$$\nabla y_1 = \nabla w_8 = D_{8,4} \nabla w_4 + D_{8,5} \nabla w_5 \ ,$$

$$D_{9,4} = 2w_4, \quad D_{9,6} = 1, \quad D_{9,7} = -1,$$
$$\nabla y_2 = \nabla w_9 = D_{9,4} \nabla w_4 + D_{9,6} \nabla w_6 + D_{9,7} \nabla w_7 \ .$$

This leads to

$$\begin{aligned} \nabla y_1 &= (D_{8,4} D_{4,1} + D_{8,5} D_{5,1}, D_{8,4} D_{4,2} + D_{8,5} D_{5,2}, 0)^t \\ \nabla y_2 &= (D_{9,4} D_{4,1} + D_{9,6} D_{6,1}, D_{9,4} D_{4,2} + D_{9,7} D_{7,2}, D_{9,6} D_{6,3} + D_{9,7} D_{7,3})^t \end{aligned}$$

which gives the correct result, as it can be immediately verified.

Tangent Mode: Observations

- In the relations above each component of the gradient is computed independently. As a result, the computational cost of evaluating the Jacobian of the function F is approximately n times the cost of evaluating one of its columns, or any linear combination of them.
- For this reason, the computation in the tangent mode is more conveniently expressed by replacing the vectors ∇w_i with the scalars

$$\dot{w}_i = \sum_{j=1}^{n} \lambda_j \frac{\partial w_i}{\partial x_j} \ ,$$

  also known as *tangents*. Here λ is a vector in $\mathbb{R}^n$ specifying the chosen linear combination of columns of the Jacobian.
- With this notation we can write

$$\dot{w}_i = \lambda_i \ , \quad i = 1, \ldots, n$$
$$\dot{w}_i = \sum_{j \prec i} D_{i,j} \dot{w}_j \ , \quad i = n+1, \ldots, N \ .$$

Tangent Mode: Observations

- At the end of the computation one finds therefore $\dot{w}_i$, $i = N - m + 1, \ldots, N$,

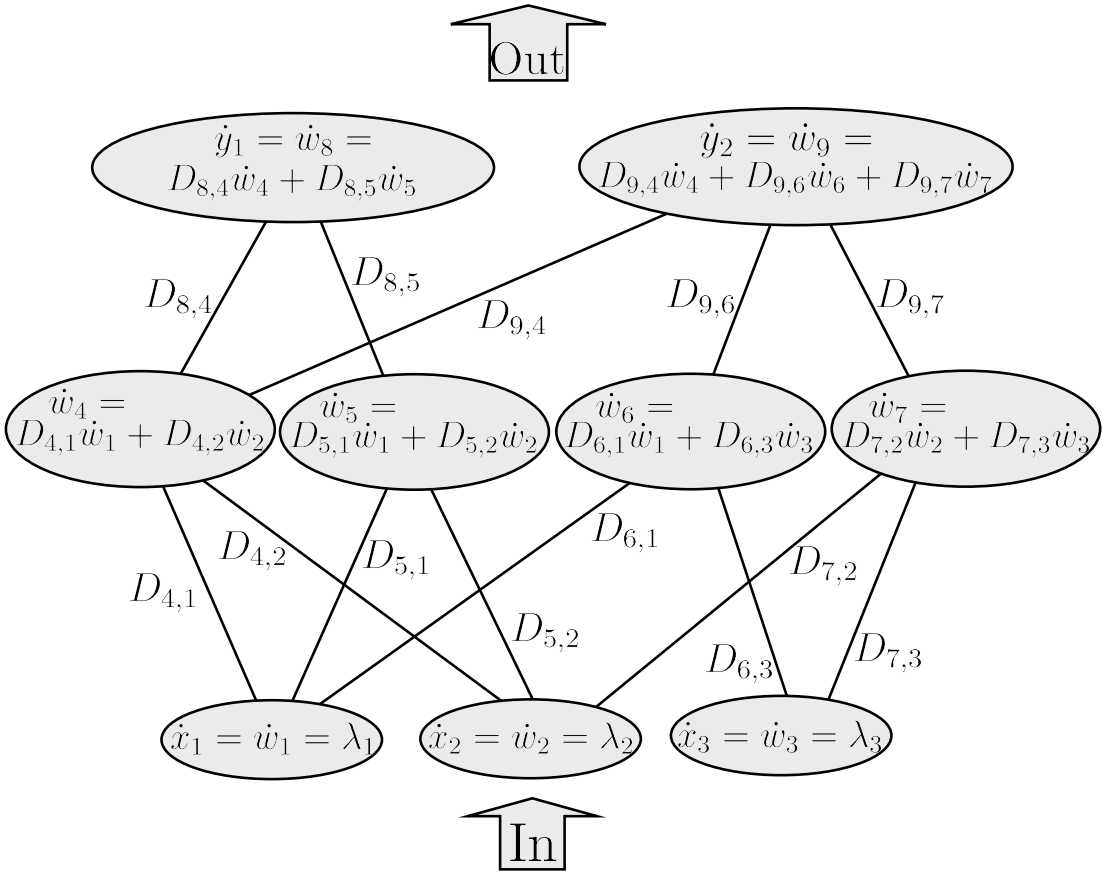
$$\dot{w}_i = \dot{y}_{i-N+m} = \sum_{j=1}^{n} \lambda_j \frac{\partial w_i}{\partial x_j} = \sum_{j=1}^{n} \lambda_j \frac{\partial y_{i-N+m}}{\partial x_j}$$

  i.e., a linear combination of the *columns* of the Jacobian.

Tangent Mode: Computational Graph

- It is useful to represent the computation of the tangent mode by means of a computational graph.
- To each node of the computational graph, w_i, we can associate the tangent of the corresponding internal variable, say $\dot{w}_i$.
- This can be calculated as a weighted average of the tangents of the variables preceding it on the graph (i.e., all the $\dot{w}_j$ such that $i \succ j$), with weights given by the arc derivatives associated with the connecting arcs.
- As a result, the tangents 'propagate' through the computational graph from the independent variables to the dependent ones, i.e., in the same direction followed in the evaluation of the original function, or *forward*. The computation of the tangents can in fact proceed instruction by instruction, at the same time when the function is evaluated.

Tangent Mode: Computational Graph



Tangent Mode: Computational Graph

- It is easy to realize that the cost for the execution of the computational graph, for a given linear combination of the columns of the Jacobian is of the same order of the cost of evaluating the function F itself.
- Hence, for the simple example considered here, Eq. (5) represents an appropriate estimate of the computational cost of any linear combination of columns of the Jacobian.
- On the other hand, in order to get each column of the Jacobian one has to repeat $n = 3$ times the calculation of the computational graph, e.g., by setting λ equal to each vector of the canonical basis in $\mathbb{R}^3$.
- As a result, the computational cost of evaluating the Jacobian relative to the cost of evaluating the function F is proportional to the number of independent variables as predicted by Eq. (3).

Adjoint Mode

- The adjoint mode provides the Jacobian of a function in a mathematically equivalent way by means of a different sequence of operations. More precisely, the adjoint mode results from computing the derivatives of the final result with respect to all the intermediate variables – the so called *adjoints* – until the derivatives with respect to the independent variables are formed.
- Formally, the adjoint of any intermediate variable w_i is defined as

$$\bar{w}_i = \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial w_i} \ , \tag{9}$$

where λ is a vector in $\mathbb{R}^m$.

Adjoint Mode

- For each of the dependent variables one has $\bar{y}_i = \lambda_i$, $i = 1, \ldots, m$, while, for the intermediate variables one has instead

$$\bar{w}_i = \sum_{k=1}^{m} \lambda_k \frac{\partial y_k}{\partial w_i} = \sum_{k=1}^{m} \lambda_k \sum_{j \succ i} \frac{\partial y_k}{\partial w_j} \frac{\partial w_j}{\partial w_i} = \sum_{j \succ i} D_{j,i} \bar{w}_j \ , \tag{10}$$

where the sum runs on the indices $j > i$ such that w_j depends explicitly on w_i. At the end of the computation one finds therefore $\bar{w}_i$, $i = 1, \ldots, n$,

$$\bar{w}_i = \bar{x}_i = \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial w_i} = \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial x_i} \ , \tag{11}$$

i.e., a given linear combination of the *rows* of the Jacobian (2).

Adjoint Mode

For our simple example this gives in particular:

$$\bar{w}_8 = \bar{y}_1 = \lambda_1 \ , \qquad \bar{w}_9 = \bar{y}_2 = \lambda_2$$
$$\downarrow$$
$$\bar{w}_4 = D_{8,4}\bar{w}_8 + D_{9,4}\bar{w}_9 \ ,$$
$$\bar{w}_5 = D_{8,5}\bar{w}_8 \ , \bar{w}_6 = D_{9,6}\bar{w}_9 \ , \bar{w}_7 = D_{9,7}\bar{w}_9 \ ,$$
$$\downarrow$$
$$\bar{w}_1 = \bar{x}_1 = D_{4,1}\bar{w}_4 + D_{5,1}\bar{w}_5 + D_{6,1}\bar{w}_6$$
$$\bar{w}_2 = \bar{x}_2 = D_{4,2}\bar{w}_4 + D_{5,2}\bar{w}_5 + D_{7,2}\bar{w}_7$$
$$\bar{w}_3 = \bar{x}_3 = D_{6,3}\bar{w}_6 + D_{7,3}\bar{w}_7 \ .$$

It is immediate to verify that by setting $\lambda = e_1$ and $\lambda = e_2$ (with e_1 and e_2 canonical vectors in $\mathbb{R}^2$), the adjoints $(\bar{w}_1, \bar{w}_2, \bar{w}_3)$ above give the components of the gradients of ∇y_1 and ∇y_2, respectively.

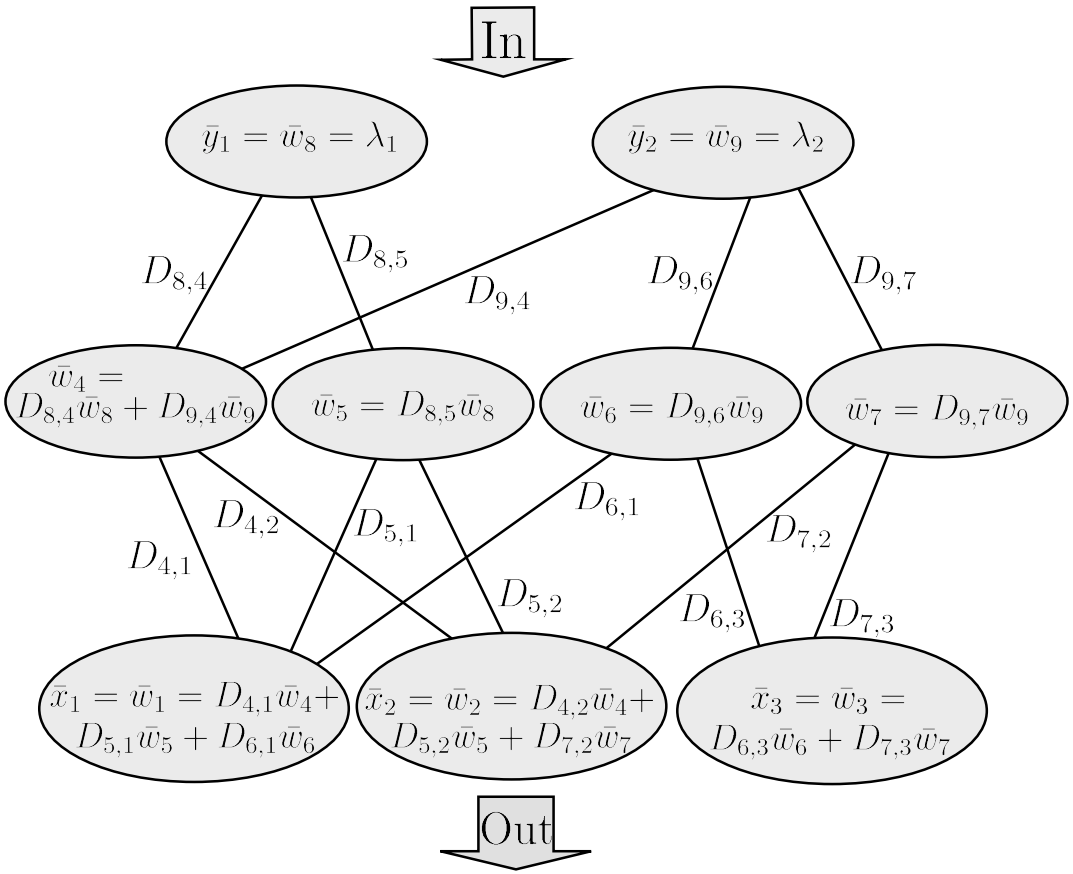
Computational Graph: Adjoint Mode

The relation:

$$\bar{w}_i = \sum_{k=1}^{m} \lambda_k \frac{\partial y_k}{\partial w_i} = \sum_{k=1}^{m} \lambda_k \sum_{j \succ i} \frac{\partial y_k}{\partial w_j} \frac{\partial w_j}{\partial w_i} = \sum_{j \succ i} D_{j,i} \bar{w}_j \ ,$$

has a clear interpretation in terms of the computational graph: the adjoint of a quantity on a given node, $\bar{w}_i$, can be calculated as a weighted sum of the adjoints of the quantities that depend on it (i.e., all the $\bar{w}_j$ such that $j \succ i$), with weights given by the local derivatives associated with the respective arcs.

Adjoint Mode: Computational Graph



Computational Graph: Adjoint Mode

- In contrast to the tangent mode, the computation of the adjoints cannot be in general simultaneous with the execution of the function. Indeed, the adjoint of each node depends on variables that are yet to be determined on the computational graph.
- As a result, the computation of the adjoints can in general begin only after the construction of the computational graph has been completed, and the information on the value and dependences of the nodes on the graph, e.g., the arc derivatives, has been appropriately stored.

Computational Graph: Adjoint Mode

- It is easy to realize that the cost of executing the computational graph for a given linear combination of the rows of the Jacobian is of the same order of the cost of evaluating the function F itself, in agreement with Eq. (7).
- On the other hand, in order to get each row of the Jacobian, one has to repeat $m = 2$ times the calculation of the computational graph, e.g., by setting λ equal to each vector of the canonical basis in $\mathbb{R}^2$. As a result, the computational cost of evaluating the Jacobian relative to the cost of evaluating the function F itself is proportional to the number of dependent variables, as predicted by Eq. (6).

First Financial Examples: Derivatives of Payoff Functions

- As a first example let's consider the Payoff of a Basket Option

$$P(X(T)) = e^{-rT}\left(\sum_{i=1}^{N} w_i X_i(T) - K\right)^+ ,$$

where $X(T) = (X_1(T), \ldots, X_N(T))$ represents the value of a set of N underlying assets, say a set of equity prices, at time T, w_i, $i = 1, \ldots, N$, are the weights defining the composition of the basket, K is the strike price, and r is the risk free yield for the considered maturity.

- For this example, we are interested in the calculation of the sensitivities with respect to r and the N components of the state vector X so that the other parameters, i.e., strike and maturity, are seen here as dummy constants.

Pseudocode of the Basket Option

```
(P)= payout (r, X[N]){

  B = 0.0;
  for (i = 1 to N)
   B += w[i]* X[i];

  x = B - K;
  D = exp(-r * T);
  P = D * max(x, 0.0);
};
```

From Ref. [8]

Pseudocode of the Tangent Payoff for the Basket Option

```
(P, P_d)= payout_d(r, X[N], r_d, X_d[N]){

  B = 0.0;
  for (i = 1 to N) {
   B += w[i]*X[i];
   B_d += w[i]*X_d[i];
  }

  x = B - K;
  x_d = B_d;

  D = exp(-r * T);
  D_d = -T * D * r_d;

  P = D * max(x, 0.0);
  P_d = 0;
  if(x > 0)
   P_d = D_d*x + D*x_d;

};
```

From Ref. [8]

- The computational cost of the Tangent payoff is of the same order of the original Payoff.
- To get all the components of the gradient of the payoff, the Tangent payoff code must be run $N + 1$ times, setting in turn one component of the Tangent input vector $I = (\dot{r}, \dot{X})^t$ to one and the remaining ones to zero.

Pseudocode of the MultimodeTangent Payoff for the Basket Option

```
(P, P_d[Nd]) = payout_dv(r, X[N], r_d[Nd], X_d[N,Nd]){

  B = 0.0;
  for (id = 1 to Nd)
     B_d[id] = 0.0;

  for (i = 1 to N) {
    B += w[i]*X[i];
    for (j = 1 to Nd)
      B_d[j] += w[i]*X_d[i,j];
  }
  x = B - K;
  for (j = 1 to Nd)
    x_d[j] = B_d[j];

  D = exp(-r * T);
  for (j = 1 to Nd)
    D_d[j] = -T * D * r_d[j];

  P = D * max(x, 0.0);
  P_d = 0;
  if(x > 0){
     for (j = 1 to Nd)
        P_d[j] = D_d[j]*x + D*x_d[j];
  }
};
```

From Ref. [8]

- To get all the components of the gradient of the payoff, the Tangent payoff code must be run only once.
- The computational cost of the Multimode Tangent payoff still scales as N times the cost of the original Payoff.

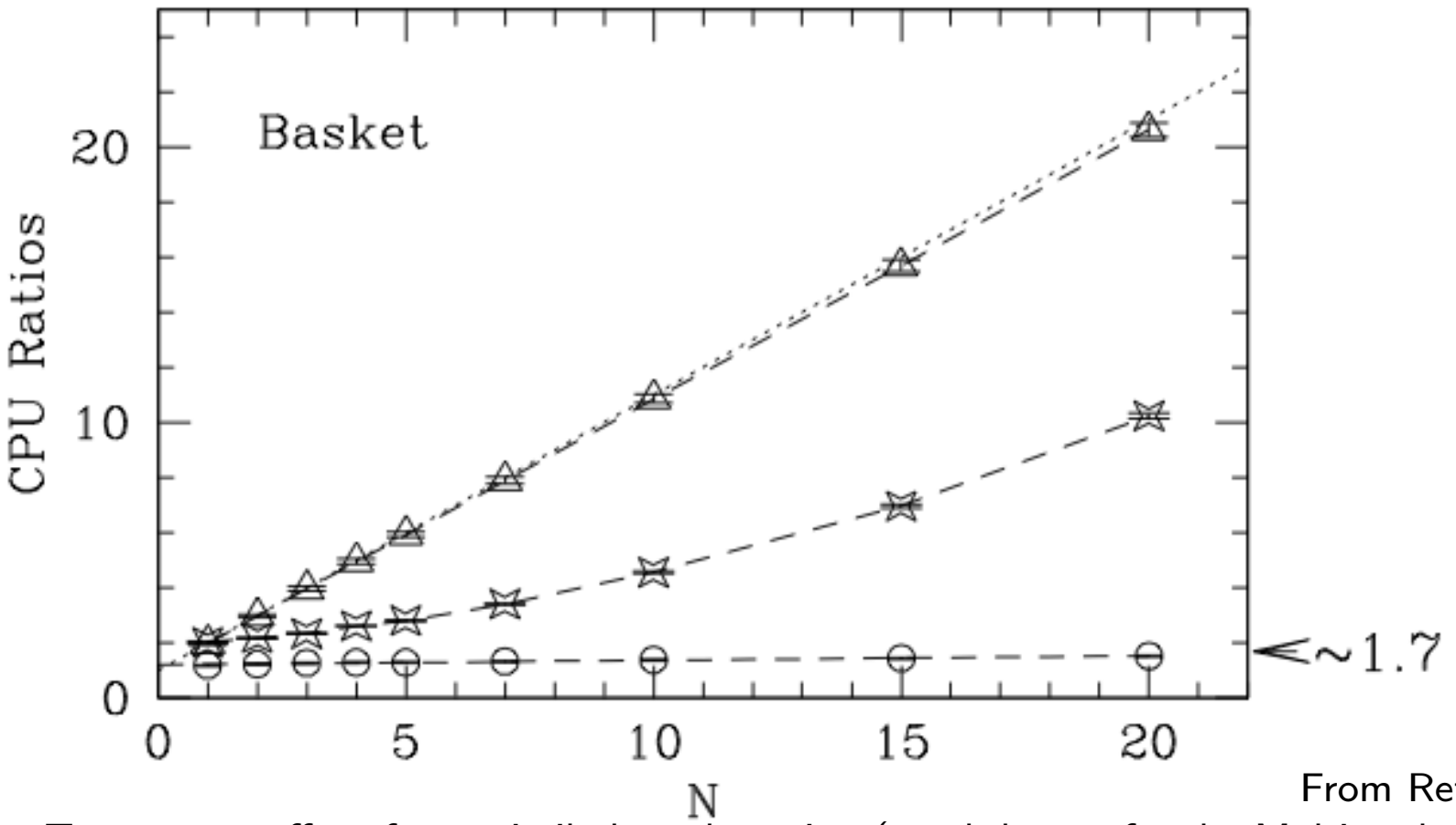
Pseudocode of the Adjoint Payoff for the Basket Option

```
(P, r_b, X_b[N]) = payout_b(r, X[N], P_b){

                                // Forward sweep
  B = 0.0;
  for (i = 0 to N)
   B += w[i] * X[i];

  x = B - K;
  D = exp(-r * T);
  P = D * max(x, 0.0);
                                // Backward sweep
  D_b = max(x, 0.0) * P_b;

  x_b = 0.0;
  if (x > 0)
   x_b = D * P_b;

  r_b = - D * T * D_b;
  B_b = x_b;

  for (i = 0 to N)
   X_b[i] = w[i] * B_b;
};
```

From Ref. [8]

- The Adjoint payoff contains a forward sweep.
- The computational cost of the Adjoint payoff is of the same order of the original Payoff.
- All the components of the gradient of the payoff, are obtained by running the Adjoint payoff only once setting $\bar{P} = 1$.

Tangent vs Adjoint



From Ref. [8]

- The Tangent payoff performs similarly to bumping (much better for the Multimode version) and has a computational complexity that scales with the number of inputs.
- In the Adjoint mode the calculation of all the derivatives of the payoff requires an extra overhead of just 70% with respect to the calculation of the payoff itself for *any* number of inputs.

Tangent vs Adjoint

- In general we are interested in computing the sensitivities of a derivative or of a portfolio of derivatives with respect to a large number of risk factors.
- The Adjoint mode of Algorithmic Differentiation is therefore the one best suited for the task.
- In some applications, however, one is also interested in computing the sensitivities of a multiplicity of derivatives individually. In those cases one can effectively combine the Adjoint and Tangent mode. See e.g. [8].
- The Adjoint mode of Algorithmic Differentiation (AAD) is the one of wider applicability.

Section 2

Automatic Algorithmic Differentiation Tools

Automatic Algorithmic Differentiation Tools

- As illustrated in the previous examples, AD gives a clear set of prescriptions by which, given any computer function, one can develop the code implementing the tangent or adjoint mode for the calculation of its derivatives. This involves representing the computer function in terms of its computational graph, calculating the derivatives associated with each of the elementary arcs, and computing either the tangents or the adjoints in the appropriate direction. This procedure, being mechanical in nature, can be automated [6, 7].
- Several AD tools have been developed that allow the automatic implementation of the calculation of derivatives either in the tangent or in the adjoint mode. These tools falls in two main categories, namely *source code transformation* and *operator overloading*. An excellent source of information in the field can be found at `www.autodiff.org`.

Source Code Transformation Tools

- Source code transformation tools are computer programs that take as an input the source code of a function, and return the source code implementing its derivatives. These tools rely on parsing the instructions of the input code and constructing a representation of the associated computational graph. In particular, an AD tool typically splits each instruction into the constituent unary or binary elementary operations for which the corresponding derivatives functions are known.
- In the tangent mode, for each elementary instruction, the AD tool generates the code calculating the tangent of the output variable given the tangents of the input ones. This involves the mechanical application of a finite set of rules, and encoding the derivatives of the intrinsic unary and binary operations. For instance, for an elementary instruction of the form

$$w_1 = \sin w_2$$

the source code transformation tool will typically generate a code of the form

$$\begin{aligned} w_1 &= \sin w_2 \\ \dot{w}_1 &= \cos w_2 \ \dot{w}_2 \end{aligned}$$

evaluating simultaneously the original function, and computing the associated tangents.

Source Code Transformation Tools: Adjoint

- In the adjoint mode, AD tools typically produce first a forward sweep that replicates the original function decorated with a record of the arc derivatives (or of the required information to calculate them on the fly in the backward sweep) in a data structure called the *tape*. Then, the tool produces the backward sweep by inverting the order of the instructions of the forward sweep. For each elementary instruction, the AD tool generates code for the calculation of the adjoint of the inputs given the adjoint of the output and the saved value of the arc derivatives. In the example above, the forward sweep could be for instance of the form

$$w_1 = \sin w_2$$
$$\text{save a record for} \quad \cos w_2 \ .$$

  The corresponding instruction in the backward sweep would instead read

$$\text{retrieve the record for} \quad \cos w_2$$
$$\bar{w}_2 = \cos w_2 \ \bar{w}_1 \ .$$

Operator Overloading

- The operator overloading approach exploits the flexibility of object oriented languages in order to introduce new abstract data types suitable to represent tangents and adjoints. Standard operations and intrinsic functions are then defined for the new types in order to allow the calculation of the tangents and the adjoints associated with any elementary instruction in a code. These tools operate by linking a suitable set of libraries to the source code of the function to be differentiated, and by redefining the type of the internal variables. Utility functions are generally provided to retrieve the value of the desired derivatives.

Operator Overloading: Tangent

- How this is possible is in fact very easy to understand for the tangent mode, as no information needs to be stored during the function evaluation for the calculation of the tangents. An operator overloading implementation is based on the definition of a new data type holding information on a certain variable w, and on its tangent $\dot{w} = \sum_{j=1}^{n} \lambda_j \partial w / \partial x_j$ for a given vector x on which w may depend on, and for a given set of weights λ, specifying a linear combination of derivatives. Given such a type, it is then easy to extend the algebra of real number in order to compute consistently both components of the pair $\widetilde{w} = (w, \dot{w})$. This can be formally done by considering the algebra of the dual real numbers $\widetilde{w} = (w, \dot{w}) \equiv w + d\dot{w}$ defined by $d^2 = 0$. This way, for instance, the ordinary multiplication between real numbers is extended to the new types as

$$\begin{aligned} \widetilde{w}_1 \cdot \widetilde{w}_2 &= (w_1 + d\dot{w}_1) \cdot (w_2 + d\dot{w}_2) = \\ &= w_1 w_2 + d(w_1 \dot{w}_2 + \dot{w}_1 w_2) + d^2(\dot{w}_1 \dot{w}_2) \\ &= (w_1 w_2, w_1 \dot{w}_2 + \dot{w}_1 w_2) \ , \end{aligned}$$

while, for a generic function assignment $w_2 = f(w_1)$ the relation

$$\widetilde{w}_2 = f(\widetilde{w}_1) = f(w_1) + df'(w_1)\dot{w}_1 = (f(w_1), f'(w_1)\dot{w}_1) \ , \tag{12}$$

that can be formally derived from the Taylor expansion of $f(w_1 + d\dot{w}_1)$ in w_1, defines the corresponding assignment on the new types.

Automatic Algorithmic Differentiation Tools: Comments

- The application of such automatic AD tools on large inhomogeneous computer codes, like the ones used in financial practice, is challenging. Indeed, pricing applications are rarely available as self contained packages, e.g., that can be easily parsed by an automatic AD tool. On the contrary, pricing applications generally consist of several independent components that are typically reusable in different contexts. Furthermore, they are possibly written in more than one programming language in an often heterogeneous programming environment, involving GPU, FPGA and cloud computing. In some cases the source code may not be even available as in the case when a third party library is used.
- Fortunately, as we will discuss in the next Section, the principles of AD can be used as a programming paradigm for any algorithm and hand coding adjoints is a perfectly viable software development strategy when the analytics library is in a stable state, as it is usually the case in practice for mature, production standard, libraries. AD tools remain very useful nonetheless for the parts of the code that are self contained, especially if they are subject to frequent changes, like the implementation of the payoff functions, which are often tweaked to meet the demands of investors.

Section 3

An introduction to the AAD Design Principles

Computational graphs (again)

- An easy way to illustrate the adjoint design paradigm is to consider again the arbitrary function

$$Y = \texttt{FUNCTION}(X)$$

mapping a vector X in $\mathbb{R}^n$ to a vector Y in $\mathbb{R}^m$ through a sequence of steps

$$X \ \rightarrow \ \ldots \ \rightarrow \ U \ \rightarrow \ V \ \rightarrow \ \ldots \ \rightarrow \ Y,$$

and to imagine that this represents a certain high level algorithm that we want to differentiate.

- Here, the real vectors U and V represent intermediate variables used in the calculation and each step can be a distinct high-level function or even an individual instruction.

Computational graphs (again)

- Given the sequence of instructions

$$X \rightarrow \ldots \rightarrow U \rightarrow V \rightarrow \ldots \rightarrow Y.$$

- Define the Adjoint of any intermediate variable V_k as

$$\bar{V}_k = \sum_{j=1}^{m} \bar{Y}_j \frac{\partial Y_j}{\partial V_k},$$

where $\bar{Y}$ is vector in $\mathbb{R}^m$.

Computational graphs (again)

- Using the chain rule we get,

$$\bar{U}_i = \sum_{j=1}^{m} \bar{Y}_j \frac{\partial Y_j}{\partial U_i} = \sum_{j=1}^{m} \bar{Y}_j \sum_k \frac{\partial Y_j}{\partial V_k} \frac{\partial V_k}{\partial U_i},$$

  which corresponds to the Adjoint mode equation for the intermediate step represented by the function $V = V(U)$

$$\bar{U}_i = \sum_k \bar{V}_k \frac{\partial V_k}{\partial U_i},$$

  namely a function of the form $\bar{U} = \bar{V}(U, \bar{V})$.

Computational graphs (again)

- Starting from the Adjoint of the outputs, $\bar{Y}$, we can apply this rule to each step in the calculation, working from right to left,

$$\bar{X} \leftarrow \ldots \leftarrow \bar{U} \leftarrow \bar{V} \leftarrow \ldots \leftarrow \bar{Y}$$

until we obtain $\bar{X}$, i.e., the following linear combination of the rows of the Jacobian $\partial Y / \partial X$

$$\bar{X}_i = \sum_{j=1}^{m} \bar{Y}_j \frac{\partial Y_j}{\partial X_i},$$

with $i = 1, \ldots, n$.

AAD as a Software Design Principle

- However, the actual calculation graph might have a more complex structure. For instance the step $U \to V$ might be implemented in terms of two computer functions of the form

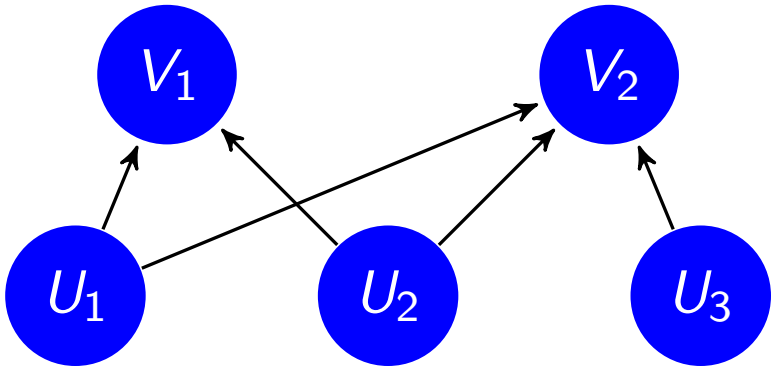
$$\begin{aligned} V^1 &:= \mathtt{V1}(U^1, U^2) \,, \\ V^2 &:= \mathtt{V2}(U^1, U^2, U^3) \,, \end{aligned}$$

  with $U = (U^1, U^2, U^3)^t$ and $V = (V^1, V^2)^t$. Here the notation $W = (W^1, W^2)^t$ simply indicates a specific partition of the components of the vector W in two sub-vectors etc.
- A natural way to represent the step $\bar{U} \leftarrow \bar{V}$ in

$$\bar{X} \;\leftarrow\; \ldots \;\leftarrow\; \bar{U} \;\leftarrow\; \bar{V} \;\leftarrow\; \ldots \;\leftarrow\; \bar{Y}$$

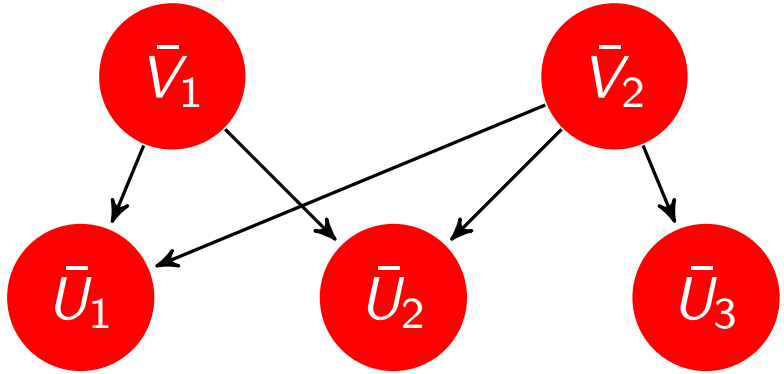
  i.e., the function $\bar{U} = \bar{V}(U, \bar{V})$, can be given in terms of an Adjoint calculation graph.

AAD as a Software Design Principle



$$V_1 = \mathtt{v1}(U_1, U_2)$$

$$V_2 = \mathtt{v2}(U_1, U_2, U_3)$$



$$(\bar{U}_1, \bar{U}_2, \bar{U}_3)^t = \mathtt{v2\_b}(U_1, U_2, U_3, \bar{V}_2)$$

$$(\bar{U}_1, \bar{U}_2)^t \mathrel{+}= \mathtt{v1\_b}(U_1, U_2, \bar{V}_1)$$

$$\bar{U}_1 = \frac{\partial V_1}{\partial U_1}\bar{V}_1 + \frac{\partial V_2}{\partial U_1}\bar{V}_2$$
$$\bar{U}_2 = \frac{\partial V_1}{\partial U_2}\bar{V}_1 + \frac{\partial V_2}{\partial U_2}\bar{V}_2$$
$$\bar{U}_3 = \frac{\partial V_2}{\partial U_3}\bar{V}_2$$

AAD as a Software Design Principle

- The relation between the Adjoint nodes is defined by the correspondence between

$$Y = \texttt{FUNCTION}(X)$$

and

$$\bar{X} = \texttt{FUNCTION\_b}(X, \bar{Y}),$$

e.g., in the specific example we associate to the following two function calls:

$$V_1 := \texttt{V1}(U_1, U_2) \ ,$$
$$V_2 := \texttt{V2}(U_1, U_2, U_3)$$

their adjoint counterpart:

$$\begin{aligned} (\bar{U}^1, \bar{U}^2, \bar{U}^3)^t &:= \texttt{V2\_b}(U^1, U^2, U^3, \bar{V}^2) \ , \\ (\bar{U}_1, \bar{U}_2)^t &:= \bar{U}^1 + \texttt{V1\_b}(U^1, U^2, \bar{V}^1) \ . \end{aligned}$$

AAD as a Software Design Principle

- This can be understood as it follow: the variable U^1 is an input of two distinct functions so that, by applying the definition of Adjoint for the variable U^1 as an input of the function $V = V(U^1, U^2) = (V^1(U^1), V^2(U^1, U^2))^t$, we get

$$\bar{U}^1 = \sum_j \bar{V}_j \frac{\partial V_j}{\partial U^1} = \sum_k \bar{V}_k^1 \frac{\partial V_k^1}{\partial U^1} + \sum_k \bar{V}_k^2 \frac{\partial V_k^2}{\partial U^1}$$

where we have simply partitioned the components of the vector V as $(V^1, V^2)^t$ for the second equality.

AAD as a Software Design Principle

- Similarly, one has for $\bar{U}^2$

$$\bar{U}^2 = \sum_j \bar{V}_j \frac{\partial V_j}{\partial U^2} = \sum_k \bar{V}_k^2 \frac{\partial V_k^1}{\partial U^2} + \sum_k \bar{V}_k^2 \frac{\partial V_k^2}{\partial U^2}$$

and for $\bar{U}^3$

$$\bar{U}^3 = \sum_j \bar{V}_j \frac{\partial V_j}{\partial U^3} = \sum_k \bar{V}_k^2 \frac{\partial V_k^2}{\partial U^3},$$

where we have used the fact that V^1 has no dependence on U^3.

- Therefore, one can realize that the Adjoint calculation graph implementing the instructions in

$$\begin{aligned} (\bar{U}^1, \bar{U}^2)^t &:= \mathtt{V2\_b}(U^1, U^2, \bar{V}^2) \,, \\ \bar{U}^1 &:= \bar{U}^1 + \mathtt{V1\_b}(U^1, \bar{V}^1) \,. \end{aligned}$$

indeed produces the Adjoint $\bar{U} = (\bar{U}^1, \bar{U}^2)^t$.

Forward and Backward Sweeps

- The Adjoint instructions

$$
\begin{aligned}
(\bar{U}^1, \bar{U}^2, \bar{U}^3)^t &:= \mathtt{V2\_b}(U^1, U^2, U^3, \bar{V}^2) \ , \\
(\bar{U}_1, \bar{U}_2)^t &:= \bar{U}^1 + \mathtt{V1\_b}(U^1, U^2, \bar{V}^1) \ .
\end{aligned}
$$

  depend on the variables U^1, U^2 and U^3.
- As a result, the Adjoint algorithm can be executed only after the original instructions

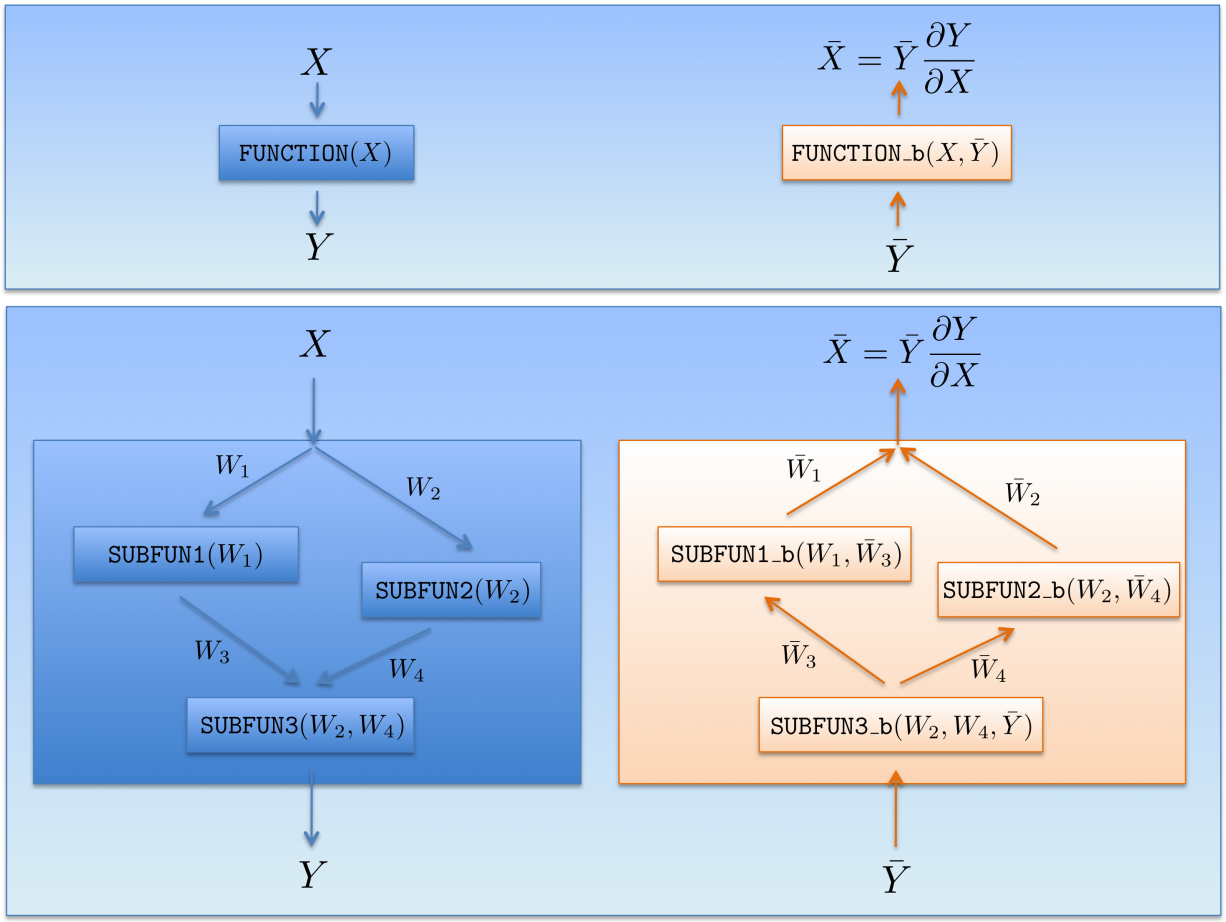
$$
X \ \rightarrow \ \ldots \ \rightarrow \ U \ \rightarrow \ V \ \rightarrow \ \ldots \ \rightarrow \ Y.
$$

  have been executed and the necessary intermediate results have been computed and stored.
- This is the reason why, as noted before, the Adjoint of a given algorithm generally contains a *forward sweep*, which reproduces the steps of the original algorithm, plus a *backward sweep*, which propagates the Adjoints.

AAD as a Software Design Principle

- The construction described above can be applied recursively for each of the functions involved in the calculation.

AAD as a Software Design Principle



- The principles of AD can be used as a programming paradigm for any algorithm.
- The Adjoint graph has the same structure of the original graph with each node/variable representing the Adjoint of the original node/variable, and it is executed in opposite direction with respect to the original one.

AAD as a Software Design Principle

- Here it is clear that one needs to define an adjoint counterpart to each function of the original program with the dependencies among their inputs and outputs defined by the rules described above.
- Following this design paradigm it is therefore easy to write the blueprint of any numerical algorithm.

AAD as a Software Design Principle

- Each adjoint function, taken in isolation, contains in turn a forward sweep recovering the information that is necessary for the computation of the adjoints.
- However, this is clearly suboptimal since all the information necessary to perform the adjoint of the algorithm is computed when performing the forward sweep of the algorithm as a whole.
- Hence, this information could be saved during this stage. This way, when the adjoint functions are invoked during the backward sweep there is no need to perform the functions' forward sweeps again.
- Strictly speaking, this is necessary to ensure that the computational cost of the overall algorithm remains within the expected bounds. However, there is a tradeoff between the time and space necessary to store and retrieve this information and the time to recalculate it from scratch. Thus, in practice it is useful to store in the forward sweep only the results of relatively expensive calculations (for more details about this technique, known as "checkpointing" please see [8] and the reference text books [6, 7]).

Section 4

Adjoint programming in a nutshell

Adjoint programming in a nutshell

a) Each intermediate differentiable variable U can be used not only by the subsequent instruction but also by several others occurring later in the program. As a result, the adjoint of U has several contributions, one for each instruction of the original function in which U was on the right hand side of the assignment operator. Hence, by exploiting the linearity of differential operators, it is generally easier to program according to a syntactic paradigm in which adjoints are always updated so that the adjoint of an instruction of the form

$$V = V(U)$$

reads

$$\bar{U}_i = \bar{U}_i + \sum_k \frac{\partial V_k(U)}{\partial U_i} \bar{V}_k \ .$$

This implies that the adjoints have to be appropriately initialized. In particular, to cope with input variables that are changed by the algorithm (see next point), it is generally best to initialize the adjoint of a given variable to zero on the instruction in which it picks up its first contribution (i.e., immediately before the adjoint counterpart of the last instruction of the original code in which the variable was to the right of the assignment operator).

Adjoint programming in a nutshell

For instance, the adjoint of the following sequence of instructions where x is the input, u and v are local variables, and y is the output

$$\begin{aligned} u &= F(x) \\ v &= G(x,u) \\ y &= H(v) \end{aligned}$$

can be written as:

$$\begin{aligned} \bar{v} &= 0 & \bar{v} &= \bar{v} + \frac{\partial H(v)}{\partial v}\bar{y} \\ \bar{u} &= 0 & \bar{u} &= \bar{u} + \frac{\partial G(x,u)}{\partial u}\bar{v} \\ \bar{x} &= 0 & \bar{x} &= \bar{x} + \frac{\partial G(x,u)}{\partial x}\bar{v} \\ & & \bar{x} &= \bar{x} + \frac{\partial F(x)}{\partial x}\bar{u} \ , \end{aligned}$$

where $\bar{y}$ is the input, $\bar{u}$ and $\bar{v}$ are local variables, and $\bar{x}$ is the output. Note that the life-cycle of an adjoint variable terminates after the adjoint of the instruction that initializes the corresponding forward variable. For instance, in the example above $\bar{y}$ can be reset to zero after the second adjoint instruction, $\bar{v}$ after the sixth, and $\bar{u}$ after the seventh. Doing so explicitly, although somewhat redundant, is often a helpful programming idiom.

Adjoint programming in a nutshell

b) In some situations the input U of a function $V = V(U)$ is modified by the function. This situation is easily analyzed by introducing an auxiliary variable U' representing the value of the input after the functions evaluation. Therefore, the original function can be thought of the form $(V, U') = (V(U), U'(U))$, where $V(U)$ and $U'(U)$ do not mutate their inputs, in combination with the assignment $U = U'$, overwriting the original input U. The adjoint of this pair of instructions clearly reads

$$\begin{aligned} \bar{U}'_i &= 0 \\ \bar{U}'_i &= \bar{U}'_i + \bar{U}_i \ , \end{aligned}$$

where we have used the fact that the auxiliary variable U' is not used elsewhere (so $\bar{U}'_i$ does not have any previous contribution), and

$$\begin{aligned} \bar{U}_i &= 0 \\ \bar{U}_i &= \bar{U}_i + \sum_k \frac{\partial V_k(U)}{\partial U_i} \bar{V}_k + \sum_l \frac{\partial U'_l(U)}{\partial U_i} \bar{U}'_l \ , \end{aligned}$$

where, again, we have also used the fact that the original input U is not used after the instruction $V = V(U)$ as it gets overwritten. One can therefore eliminate altogether the adjoint of the auxiliary variable $\bar{U}'$ and simply write

$$\bar{U}_i \mathrel{+}= \sum_k \frac{\partial V_k(U)}{\partial U_i} \bar{V}_k + \sum_l \frac{\partial U'_l(U)}{\partial U_i} \bar{U}_l \ .$$

Adjoint programming in a nutshell

Very common examples of this situation are given by increments of the form

$$U_i = a\,U_i + b$$

with a and b constant with respect to U. According to the above recipe, the adjoint counterpart of this instruction simply reads

$$\bar{U}_i = a\,\bar{U}_i \ .$$

These situations are common in iterative loops where a number of variables are typically updated at each iteration.

Adjoint programming in a nutshell

c) Each function, subroutine or method can be abstracted as a function with some inputs and some outputs even if some of these variables are implicit. For instance, in an object oriented language, a class constructor can be seen as a function whose (implicit) outputs are the member variables of the class. These member variables, say θ, can be also seen as implicit inputs of all the other methods of the class, e.g.,

$$Y = \texttt{METHOD}(X, \theta) \ .$$

Hence, the corresponding adjoint methods – in addition to the sensitivities to its explicit inputs – generally produce the sensitivities with respect to the member variables, $\bar{\theta}$, e.g.,

$$(\bar{X}, \bar{\theta}) \mathrel{+}= \texttt{METHOD\_b}(X, \theta, \bar{Y}) \ .$$

Section 5

Computing Risk in Monte Carlo

Option Pricing Problems

- Option pricing problems can be typically formulated in terms of the calculation of expectation values of the form

$$V = \mathbb{E}_{\mathbb{Q}}\Big[P(X(T_1), \ldots, X(T_M))\Big].$$

- Here $X(t)$ is a N-dimensional vector and represents the value of a set of underlying market factors (e.g., stock prices, interest rates, foreign exchange pairs, etc.) at time t.
- $P(X(T_1), \ldots, X(T_M))$ is the discounted payout function of the priced security, and depends in general on M observations of those factors.
- In the following, we will indicate the collection of such observations with a $d = N \times M$ dimensional state vector

$$X = (X(T_1), \ldots, X(T_M)).$$

Monte Carlo Sampling of the Payoff Estimator

- The expectation value above can be estimated by means of Monte Carlo (MC) by sampling a number N_{MC} of random replicas of the underlying state vector $X[1], \ldots, X[N_{\mathrm{MC}}]$, sampled according to the distribution $\mathbb{Q}(X)$, and evaluating the payout $P(X)$ for each of them.
- This leads to the estimate of the option value V as

$$V \simeq \frac{1}{N_{\mathrm{MC}}} \sum_{i_{\mathrm{MC}}=1}^{N_{\mathrm{MC}}} P\left(X[i_{\mathrm{MC}}]\right),$$

with standard error $\Sigma/\sqrt{N_{\mathrm{MC}}}$, where

$$\Sigma^2 = \mathbb{E}_{\mathbb{Q}}[P\left(X\right)^2] - \mathbb{E}_{\mathbb{Q}}[P\left(X\right)]^2$$

is the variance of the sampled payout.

Price Sensitivities/Greeks

We indicate with $\theta = (\theta_1, \ldots, \theta_{N_\theta})$, the set of N_θ parameters (*e.g.*, spot price, interest-rates, volatilities etc.) the option price, V_θ, is dependent on. For example, for an equity problem,

$$dX(t) = (r + \delta)\, X(t)dt + \sigma X(t)dW_t$$

where X_0 is the $t = 0$ spot price, r is the 'risk-free' interest rate and δ is the dividend yield, the vector θ is $\theta = (X_0, \sigma, r, \delta)$ and

$$\frac{\partial V_\theta}{\partial \theta_1} = \frac{\partial V_\theta}{\partial X_0} \quad \text{Delta} \qquad \frac{\partial V_\theta}{\partial \theta_2} = \frac{\partial V_\theta}{\partial \sigma} \quad \text{Vega}$$

$$\frac{\partial V_\theta}{\partial \theta_3} = \frac{\partial V_\theta}{\partial r} \quad \text{IR Risk} \qquad \frac{\partial V_\theta}{\partial \theta_4} = \frac{\partial V_\theta}{\partial \delta} \quad \text{Dividend Risk}$$

Subsection 1

Finite Differences ('Bumping')

Finite-Differences (a.k.a. Bumping)

Exploiting the definition of derivatives, one can write a one-sided (forward) estimator:

$$\frac{\partial V(\theta)}{\partial \theta} = \frac{V(\theta + h) - V(\theta)}{h} ,$$

a one-sided (backward) estimator:

$$\frac{\partial V(\theta)}{\partial \theta} = \frac{V(\theta) - V(\theta - h)}{h} ,$$

a two-sided (central) estimator:

$$\begin{aligned} \frac{\partial V(\theta)}{\partial \theta} &= \frac{1}{2}\left[\frac{V(\theta + h) - V(\theta)}{h} + \frac{V(\theta) - V(\theta - h)}{h}\right] \\ &= \frac{V(\theta + h) - V(\theta - h)}{2h} . \end{aligned}$$

Finite-Differences: Pros and Cons

Pros:

- The main advantage of the approach is its simplicity as it requires hardly any implementation: just repeating the calculation of the value function.

Cons:

- the computational cost: $\sim N_\theta$ for one-sided, $2 \times N_\theta$ for two-sided estimators;
- the accuracy of the calculation; the sources of error are *bias* (associated to the finite-difference approximation) and *variance* (associated with the MC sampling);
- (in its simplest incarnation) the lack of an estimator of the statistical uncertainty of the sensitivity.

Finite-Differences: Bias

Starting from the Taylor's expansion

$$V(\theta \pm h) = V(\theta) \pm \frac{\partial V(\theta)}{\partial \theta} h + \frac{1}{2} \frac{\partial^2 V(\theta)}{\partial \theta^2} h^2 \pm \frac{1}{3!} \frac{\partial^3 V(\theta)}{\partial \theta^3} h^3 + \mathcal{O}(h^4) \ ,$$

- One-sided estimators:

$$\frac{V(\theta + h) - V(\theta)}{h} = \frac{\partial V(\theta)}{\partial \theta} + \mathcal{O}(h)$$

- Two-sided estimators:

$$\frac{V(\theta + h) + V(\theta - h)}{h^2} = \frac{\partial V(\theta)}{\partial \theta} + \mathcal{O}(h^2)$$

Finite-Differences: Machine-Precision Issues

The bias cannot be reduced to an arbitrarily small number because of finite machine precision. Indeed, taking this factor into account, the one-sided forward estimator for the first derivative can be expressed as

$$\frac{f(x+h)+\epsilon_1 - f(x)+\epsilon_2}{h} = f'(x) + \frac{2\epsilon}{h} + \frac{h}{2}f''(x) + \mathcal{O}(h^2) \ ,$$

where $\epsilon > \max(|\epsilon_1|, |\epsilon_2|)$. For h small enough (namely $h \ll \epsilon$), the right hand side diverges, corresponding to an unreliable estimate of the derivative. A bound on the total error can be written as

$$\left| f'(x) - \frac{f(x+h)+\epsilon_1 - f(x)+\epsilon_2}{h} \right| \leq \frac{h}{2}M + \frac{2\epsilon}{h} \ ,$$

where M is a bound of $|f''(x)|$ in $x \in [x, x+h]$. This can be minimized, to give the optimal value of h as

$$h^\star = 2\sqrt{\frac{\epsilon}{M}} \ .$$

Finite-Differences: Variance

Let's consider again the one-sided estimator $(\bar{V}(\theta+h)-\bar{V}(\theta))/h$, with

$$\bar{V}(\theta)=\frac{1}{N_{MC}}\sum_{i=1}^{N_{MC}}P_i(\theta) \qquad \bar{V}(\theta+h)=\frac{1}{N_{MC}}\sum_{i=1}^{N_{MC}}P_i(\theta+h)$$

(with $P(\theta)=P(X(\theta))$ and assuming no dependency on θ in $\mathbb{Q}$) and compute the variance of the corresponding MC estimator

$$\begin{aligned}
&\mathrm{Var}\left[\frac{\bar{V}(\theta+h)-\bar{V}(\theta)}{h}\right]=\frac{1}{h^2}\,\mathrm{Var}\left[\bar{V}(\theta+h)-\bar{V}(\theta)\right]\\
&=\frac{1}{h^2N_{MC}}\left(\mathrm{Var}\left[P(\theta+h)\right]+\mathrm{Var}\left[P(\theta)\right]-2\mathrm{CoVar}\left[P(\theta+h)P(\theta)\right]\right)\\
&\simeq\frac{2}{h^2N_{MC}}\left(\mathrm{Var}\left[P(\theta)\right]-\mathrm{CoVar}\left[P(\theta+h)P(\theta)\right]\right)
\end{aligned}$$

The variance of the finite-difference estimators depends on the covariance between the estimators $P(\theta+h)$ and $P(\theta)$.

Estimating the Variance

Estimating the variance from the result of a MC simulation is a problem in itself. In fact,

$$\mathrm{Var}\left[\frac{\bar{V}(\theta+h)-\bar{V}(\theta)}{h}\right]=$$
$$\frac{1}{h^2 N_{MC}}\left(\mathrm{Var}\left[P(\theta+h)\right]+\mathrm{Var}\left[P(\theta)\right]-2\mathrm{CoVar}\left[P(\theta+h)P(\theta)\right]\right)$$

cannot be computed if only estimates of $\mathrm{Var}\left[P(\theta)\right]$ and $\mathrm{Var}\left[P(\theta+h)\right]$ are available. We need

$$\mathrm{CoVar}\left[\bar{P}(\theta)\bar{P}(\theta+h)\right]\simeq\frac{1}{N_{MC}}\sum_{i=1}^{N_{MC}}P_i(\theta)P_i(\theta+h)-\left(\frac{1}{N_{MC}}\sum_{i=1}^{N_{MC}}P_i(\theta)\right)^2$$

which can be computed only if we store the intermediate samples of the estimator or if we compute simultaneously $P(\theta)$ and $P(\theta+h)$.

Finite-Differences: Variance (cont'd)

$$\mathrm{Var}\left[\frac{\bar{V}(\theta+h)-\bar{V}(\theta)}{h}\right] = \frac{1}{h^2 N_{\mathrm{MC}}}\mathrm{Var}\left[P(\theta+h)-P(\theta)\right]$$

Three cases of primary importance arise in practice (see, e.g. [1]) for $\mathrm{Var}\left[P(\theta+h)-P(\theta)\right]$:

1. $\mathcal{O}(1)$: independent sampling of the base and bumped estimator:

$$\mathrm{Var}\left[P(\theta+h)-P(\theta)\right] \to 2\mathrm{Var}\left[P(\theta)\right] \sim \mathcal{O}(1)\ ;$$

2. $\mathcal{O}(h)$: achieved at worst if we use the same random seed;
3. $\mathcal{O}(h^2)$: requires regularity (Lipschitz continuity) on the payoff and the process and other technical conditions (see, e.g. [1]).

In the first two cases reducing the increment h increases the variance which diverges for $h \to 0$.

Section 6

Greeks Without Bumping (Classical Approaches)

Greeks Without Bumping (Classical Approaches)

- Likelihood Ratio Method
  - Differentiation of the pdf associated with the stochastic process.
  - Greeks obtained by multiplying the payoff by a suitable weight.
- Pathwise Derivative Method
  - Differentiate both the process and the payout through the chain rule.
  - Equivalent to standard correlated bumping as the bump goes to zero.

Subsection 1

Likelihood Ratio Method

Likelihood Ratio Method

This method hinges on isolating the dependence on the parameters in the density function:

$$V(\theta) = \mathbb{E}_{\mathbb{Q}_\theta}\Big[P(X(T_1), \ldots, X(T_M))\Big] = \int P(x) q_\theta(x)\, dx \ .$$

If the probability density function, $q_\theta(x) = dQ_\theta/dx$ is regular enough, then

$$\partial_\theta V(\theta) = \int P(x)\, \partial_\theta q_\theta(x)\, dx \ .$$

How can we estimate this quantity by MC?

Likelihood Ratio Method (cont'd)

By multiplying and dividing by $q_\theta(x)$, the sensitivity

$$\partial_\theta V(\theta) = \int P(x)\, \partial_\theta q_\theta(x)\, dx = \int P(x)\, \frac{\partial_\theta q_\theta(x)}{q_\theta(x)} q_\theta(x)\, dx \ .$$

can be expressed as an expectation value over the original probability density of a re-weighted version of the original payoff.

$$\partial_\theta V(\theta) = \mathbb{E}_{\mathbb{Q}_\theta}\Big[P(X)\Omega_\theta(X)\Big] \ ,$$

where the LRM wright reads:

$$\Omega_\theta(X) = \frac{\partial_\theta q_\theta(x)}{q_\theta(x)} = \partial_\theta \log q_\theta(X) \ ,$$

which can be sampled as usual

$$\bar{\theta} = \frac{1}{N_{MC}} \sum_{i=1}^{N_{MC}} P(X[i])\Omega_\theta(X[i]) \ .$$

Example: Black Scholes Setup

Let's consider the BS setup:

$$dX(t) = rX(t)dt + \sigma X(t)dW_t \quad X(0) = X_0$$
$$X(T) = X_0 \exp\left[(r - \sigma^2/2)T + \sigma\sqrt{T}Z\right] .$$

$X(T)$ is distributed according to a lognormal distribution

$$q_\theta(X) = \frac{1}{X\sigma\sqrt{T}}\phi(Z(X|X_0)) \qquad \phi(Z) = e^{-Z^2/2}/\sqrt{2\pi}$$
$$Z(X|X_0) = \frac{\log X/X_0 - (r - \sigma^2/2)T}{\sigma\sqrt{T}}$$

The LRM weights for Delta and Vega read:

$$\Omega_{X_0}(X) = \partial_{X_0} \log q_\theta(X) = \frac{Z(X|X_0)}{\sigma\sqrt{T}X_0} ,$$
$$\Omega_\sigma(X) = \partial_\sigma \log q_\theta(X) = \frac{Z(X|X_0)^2 - 1}{\sigma} - Z(X|X_0)\sqrt{T} .$$

Example: Black Scholes Setup (cont'd)

Given the form of the LRM weights for Delta and Vega:

$$\Omega_{X_0}(X) = \partial_{X_0} \log q_\theta(X) = \frac{Z(X|X_0)}{\sigma\sqrt{T}X_0} \ ,$$

$$\Omega_\sigma(X) = \partial_\sigma \log q_\theta(X) = \frac{Z(X|X_0)^2 - 1}{\sigma} - Z(X|X_0)\sqrt{T} \ ,$$

it is clear that the both diverge as $\sigma\sqrt{T}$ and $\sigma \to 0$ (respectively). As a result, the variance of the LRM estimators for a payoff $P(X)$

$$\mathrm{Var}[P(X)\Omega_\theta(X)] = \mathbb{E}[P(X)^2\Omega_\theta(X)^2] - \bar{\theta}^2 \to \infty$$

also generally diverges.

Example: Asian Options for Lognormal Model

Let's consider again the BS setup and a path-dependent option, like, e.g.,

$$P(X_1, \dots, X_M) = e^{-rT}(\bar{X} - K)^+ \quad \bar{X} = \frac{1}{M}\sum_{i=1}^{M} X_i \ .$$

In order to compute the LRM wrights we need the joint distribution of $X = (X_1, \dots, X_M)$. Thanks to the Markov property of the GBM we have

$$q_\theta(X) = q_\theta^{(1)}(X_1|X_0) \dots q_\theta^{(M)}(X_M|X_{M-1})$$

with

$$q_\theta^{(i)}(X_i|X_{i-1}) = \frac{1}{X_i \sigma \sqrt{T_i - T_{i-1}}} \phi(Z(X_i|X_{i-1})) \quad \phi(Z) = e^{-Z^2/2}/\sqrt{2\pi}$$

$$Z(X_i|X_{i-1}) = \frac{\log X_i/X_{i-1} - (r - \sigma^2/2)(T_i - T_{i-1})}{\sigma\sqrt{T_i - T_{i-1}}}$$

Example: Asian Options for Lognormal Model (cont'd)

Given that

$$\log q_\theta^{(1)}(X_1|X_0)\dots q_\theta^{(M)}(X_M|X_{M-1}) = \sum_{i=1}^{M} \log q_\theta^{(i)}(X_i|X_{i-1})$$

the LRM weights for Delta and Vega read:

$$\Omega_{X_0}(X) = \partial_{X_0} \log q_\theta(X) = \partial_{X_0} \log q_\theta^{(1)}(X_1|X_0) = \frac{Z(X_1|X_0)}{\sigma\sqrt{T_1 - T_0}X_0} \ ,$$

$$\Omega_\sigma(X) = \partial_\sigma \log q_\theta(X) = \sum_{i=1}^{M} \frac{Z(X_i|X_{i-1})^2 - 1}{\sigma} - Z(X_i|X_{i-1})\sqrt{T_i - T_{i-1}} \ .$$

Observations:

- The Delta LRM weight depends only on the distribution across the first time step.
- The LRM weights for Delta and Vega both diverge as $\sigma\sqrt{T}$ and $\sigma \to 0$ (respectively).

Gaussian Vectors

Let us consider a random vector $Y \sim \phi_d(\mu(\theta), \Sigma(\theta))$ with

$$\phi_d(\mu(\theta), \Sigma(\theta)) = \frac{1}{(2\pi)^{d/2}\sqrt{\det \Sigma(\theta)}} \exp\left[\frac{1}{2}(Y-\mu(\theta))^T \Sigma^{-1}(\theta)(Y-\mu(\theta))\right]$$

The generic LRM weight reads

$$\begin{aligned}\Omega_\theta(Y) &= (Y-\mu(\theta))^T \Sigma^{-1}(\theta)\partial_\theta \mu(\theta) - \frac{1}{2}\mathrm{Tr}(\Sigma^{-1}(\theta)\partial_\theta \Sigma(\theta)) \\ &\quad + \frac{1}{2}(Y-\mu(\theta))^T \Sigma^{-1}(\theta)\partial_\theta \Sigma(\theta)\Sigma^{-1}(\theta)(Y-\mu(\theta)) \ .\end{aligned}$$

If we simulate Y as $\mu(\theta) + AZ$ with $Z \sim \phi_d(0, I_d)$ for some matrix such that $AA^T = \Sigma$ then the LRM weight simplifies as:

$$\begin{aligned}\Omega_\theta(Y) &= Z^T A^{-1}(\theta)\partial_\theta \mu(\theta) - \frac{1}{2}\mathrm{Tr}(A^{-1}(\theta)\partial_\theta \Sigma(\theta)A^{-1}(\theta)^T) \\ &\quad + \frac{1}{2}Z^T A^{-1}(\theta)\partial_\theta \Sigma(\theta)A^{-1}(\theta)^T Z \ .\end{aligned}$$

d-dimensional GBM

$$dX_i(t) = rX_i(t)dt + \sigma_i X_i(t)dW_t^i \quad X_i(0) = X_0^i$$

with $\mathbb{E}[dW_i dW_j] = \rho_{ij}dt$. This can be recast as $X_i(T) = \exp Y_i$

$$Y_i = \mu_i + \sqrt{T}\,(AZ)_i \quad \mu_i = \log X_0^i + (r - \sigma_i^2/2)T, \quad Z \sim \phi_d(0, I_d)$$

with $AA^T = \Sigma$, $\Sigma_{ij} = \sigma_i\sigma_j\rho_{ij}$, and any discounted payoff $P(X(T))$ can be seen as a function of the Gaussian random vector $Y(T)$. The LRM weights for Delta and Vega read therefore:

$$\Omega_{X_0^i} = \frac{(Z^T A^{-1})_i}{X_0^i\sqrt{T}} \qquad \Omega_{\sigma_i} = \left(AZ - \sigma_i\sqrt{T}\right)(Z^T A^{-1})_i - \frac{1}{\sigma_i}.$$

Euler Discretization

Let's consider a generic d-dimensional diffusion process

$$dX_i(t) = \mu_i(X_i(t), t, \theta)dt + \sigma_i(X_i(t), t, \theta)dW_t^i \quad X_i(0) = X_0^i \ .$$

If the SDE cannot be integrated exactly, one typically relies on a discretization scheme, *e.g.* Euler's,

$$X_i(t + \Delta t) = X_i(t) + \mu_i(X_i(t), t, \theta)\Delta t + \sigma_i(X_i(t), t, \theta)\sqrt{\Delta t}\ Z_i'(t) \ ,$$

with $\mathbb{E}[Z_i' Z_j'] = \rho_{i,j}$ and can be simulated as

$$\begin{aligned} X_i(t + \Delta t) &= (X_i(t) + \mu_i(X_i(t), t, \theta)\Delta t) + \sqrt{\Delta t}\,(AZ(t))_i \ , \\ Z(t) &\sim \phi_d(0, I_d) \end{aligned}$$

with $AA^T = \Sigma$, and $\Sigma_{ij} = \rho_{ij}\sigma_i(X_i(t), t, \theta)\sigma_j(X_j(t), t, \theta)$.

Euler Discretization (cont'd)

As a result, the approximate distribution of $X_i(t + \Delta t)$, conditional on $X_i(t)$, is normal, so that using the Markov property

$$\log q_\theta(X) = \sum_{m=1}^{M} \log q_\theta^{(m)}(X_m|X_{m-1}) \ ,$$

where

$$q_\theta^{(m)}(X_m|X_{m-1}) = \phi_d(X_i(t) + \mu_i(X_i(t), t)\Delta t, \Sigma(X_i(t), t)\Delta t) \ ,$$

with $\Sigma_{ij} = \rho_{ij}\sigma_i(X_i(t), t, \theta)\sigma_j(X_j(t), t, \theta)$.

Euler Discretization (cont'd)

Using a derivation similar to the one employed for generic Gaussian vectors, the LRM weights for Delta and Vega read therefore:

$$\Omega_{\theta_{\mu_i}} = \frac{(Z(T_1)^T A^{-1})_i}{\sqrt{\Delta T}} \partial_{\theta_{\mu_i}} \mu_i(X_0^i(0), 0, \theta) \ ,$$

$$\Omega_{\theta_{\sigma_i}} = \sum_{m=1}^{M} \left(AZ(T_m) - \partial_{\theta_{\sigma_i}} \frac{\sigma_i^2(\theta)}{2} \sqrt{\Delta t} \right) (Z(T_m)^T A^{-1})_i - \frac{1}{\sigma_i^2} \partial_{\theta_{\sigma_i}} \frac{\sigma_i^2(\theta)}{2} \ ,$$

where θ_{μ_i} and θ_{σ_i} are, respectively, any of the parameters defining the drift and instantaneous volatility for the process i.

Bias and Variance

Let us consider the expectation value of the LRM weighs:

$$\mathbb{E}_{\mathbb{Q}_\theta}\Big[\Omega_\theta(X)\Big] = \int \frac{\partial_\theta q_\theta(x)}{q_\theta(x)} q_\theta(x)\, dx = \partial_\theta \int q_\theta(x)\, dx = \partial_\theta 1 = 0 \ .$$

This indicates that the LRM weights have no definite sign. This can give rise to poor variance properties whenever the configurations with opposite sign have similar weight in the MC average

$$\bar{\theta} = \frac{1}{N_{MC}} \sum_{i=1}^{N_{MC}} P(X[i])\Omega_\theta(X[i]) \ ,$$

so that the final outcome is the result of the cancellation of two comparable and not necessarily highly correlated quantities.

Reducing the Variance of Likelihood Ratio Greeks in Monte Carlo (cont'd)

As a special case, let's consider again the d-dimensional GBM case and European payoff, $P(X(T))$ with $X_i(T) = \exp Y_i$

$$Y_i = \mu_i + \sqrt{T}(AZ)_i \quad \mu_i = \log X_0^i + (r - \sigma_i^2/2)T, \quad Z \sim \phi_d(0, I_d)$$

and $AA^T = \Sigma$, $\Sigma_{ij} = \sigma_i \sigma_j \rho_{ij}$. Recall that we have found that the LRM weights for Delta and Vega read:

$$\Omega_{X_0^i} = \frac{(Z^T A^{-1})_i}{X_0^i \sqrt{T}} \qquad \Omega_{\sigma_i} = \left(AZ - \sigma_i \sqrt{T}\right)(Z^T A^{-1})_i - \frac{1}{\sigma_i}.$$

Reducing the Variance of Likelihood Ratio Greeks in Monte Carlo (cont'd)

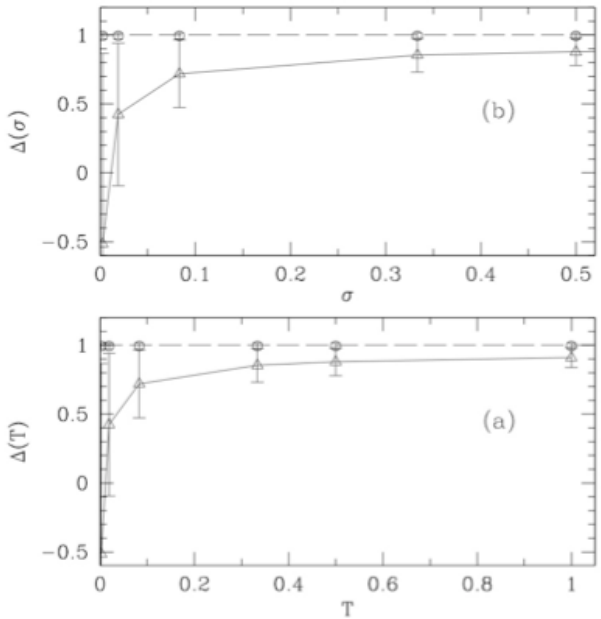
The variance of the naïve estimator

$$\mathrm{Var}\left[\frac{(Z^T A^{-1})_i}{X_0^i \sqrt{T}}\right] \to \infty$$

as $\sigma_i \sqrt{T} \to 0$ (as $(A^{-1}Z)_i \sim 1/\sigma_i$).
However, the variance of the antithetic estimator

$$\mathrm{Var}\left[\frac{((Z^T - Z^T)A^{-1})_i}{X_0^i \sqrt{T}}\right] = 0$$

Reducing the Variance of Likelihood Ratio Greeks in Monte Carlo (cont'd)



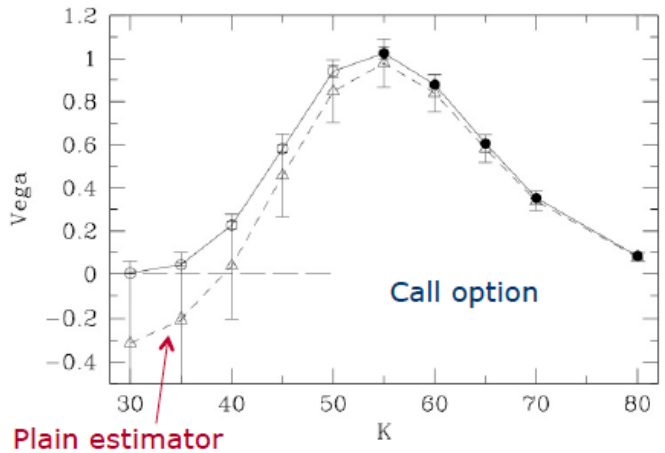
LRM Delta for a deep in the money option as a function of Time to Maturity (a) and Volatility (b). From Ref. [2].

Reducing the Variance of Likelihood Ratio Greeks in Monte Carlo (cont'd)

Unfortunately, Antithetic Variables do not help much for Vega because the estimator is not odd:

$$\Omega_{\sigma_i} = \left(AZ - \sigma_i \sqrt{T}\right)(Z^T A^{-1})_i - \frac{1}{\sigma_i} \ .$$

Control variates (and LSIS) can provide significant improvements however.



Natural controls:

$$\partial_{\sigma_i} \mathbb{E}[S_i] = 0 \quad \mathbb{E}[\Omega_{\sigma_i}] = 0$$

Variance Reductions

K	AV	AV+CV	AV+LSIS
30	1.3(1)	$4(1)10^4$	11(1)
40	1.7(1)	150(10)	7.0(6)
50	2.1(1)	22(2)	8.0(8)
60	2.1(2)	5.5(6)	80(10)
70	1.9(2)	5.2(5)	340(40)
80	1.7(3)	1.7(3)	1100(100)

LRM Vega for a deep in the money option. From Ref. [2].

Likelihood Ratio Method: Pros and Cons

$$\bar{\theta} = \frac{1}{N_{MC}} \sum_{i=1}^{N_{MC}} P(X[i]) \Omega_\theta(X[i]) \qquad \Omega_\theta(X) = \partial_\theta \log q_\theta(X).$$

Pros:

- Simultaneous valuation of value and sensitivities: Generally more efficient (for a given number of replications) than repeating the simulation from scratch.
- Does not require regularity conditions on the payoff.
- No finite-difference bias.

Cons:

- (To be bias free), requires explicit knowledge of the density function.
- The variance of the estimator is difficult to predict a priori. A large variance can destroy the computational benefit of resampling with LRM weights instead of repeating the simulation.

Subsection 2

Pathwise Derivative Method

Pathwise Derivative Method

Let's consider again the value of an option computed by Monte Carlo sampling

$$V(\theta) = \mathbb{E}_{\mathbb{Q}}\Big[P(X(T_1), \ldots, X(T_M))\Big] \ .$$

and the problem of computing sensitivities with respect to a parameter θ_k:

$$\bar{\theta}_k = \partial_{\theta_k} V(\theta) \ .$$

It is tempting to bring the derivative operator under the expectation value (which is allowed under certain regularity conditions discussed later) and write

$$\bar{\theta}_k = \partial_{\theta_k} V(\theta) = \mathbb{E}_{\mathbb{Q}}\Big[\partial_{\theta_k} P(X(T_1), \ldots, X(T_M))\Big] \ .$$

Pathwise Derivative Method (cont'd)

But what does the symbol

$$\bar{\theta}_k = \partial_{\theta_k} V(\theta) = \mathbb{E}_{\mathbb{Q}}\Big[\partial_{\theta_k} P(X(T_1), \ldots, X(T_M))\Big]$$

actually mean? It is useful to think of the expectation as an integral

$$\mathbb{E}_{\mathbb{Q}}\Big[P(X)\Big] = \int P(X) dQ(\omega) \ .$$

so that

$$\mathbb{E}_{\mathbb{Q}}\Big[\partial_{\theta_k} P(X)\Big] = \int \left[\partial_{\theta_k} P(X(\theta, \omega))\right] dQ(\omega) \ .$$

From here it is clear that taking the derivative inside the expectation implies choosing as a sampling distribution a θ-independent probability density $dQ(\omega)$ and moving the θ dependence in the mapping between the sampled random variates and the vector $X = X(\theta, \omega)$.

Pathwise Derivative Method (cont'd)

Under the Pathwise Derivative Method, the θ dependence is all in the payoff function and none in the probability density. This is the complementary view of the LRM where instead we see the expectation value as an integral over a θ dependent probability density of a (θ-independent) function of dummy variables:

$$\partial_{\theta_k} \mathbb{E}_{\mathbb{Q}}\Big[P(X)\Big] = \int P(X)\partial_{\theta_k} dQ_\theta(X) = \int P(X)\left[\partial_{\theta_k} \log q_\theta(X)\right] dQ_\theta(X)$$
$$= \mathbb{E}_{\mathbb{Q}}\Big[P(X)\Omega_\theta(X)\Big] \ .$$

Pathwise Derivative Method (cont'd)

- The Pathwise Derivative Method allows the calculation of the sensitivities of the option price V with respect to a set of N_θ parameters $\theta = (\theta_1, \ldots, \theta_{N_\theta})$, with a single set of N_{MC} simulations.
- In order for the Pathwise Derivative Method to be applicable

$$\langle \bar{\theta}_k \rangle = \mathbb{E}_{\mathbb{Q}} \left[\frac{\partial P(X)}{\partial \theta_k} \right] ,$$

must hold. This holds whenever the payout function is regular enough, e.g., Lipschitz-continuous, and under additional conditions that are often satisfied in financial pricing (see, e.g., [1]). These are the same conditions under which the variance of the finite-difference estimator remains finite for $h \to 0$.

- As a result, when it is applicable, the Pathwise Derivative Method provides the same numerical result as the finite-difference method (using the same random seed in the base and perturbed paths).

Pathwise Derivative Method: Interpretation

- The calculation of $\langle \bar{\theta}_k \rangle$ can be performed by applying the chain rule, and averaging on each MC sample the so-called Pathwise Derivative Estimator

$$\bar{\theta}_k \equiv \frac{dP(X)}{d\theta_k} = \sum_{j=1}^{d} \frac{\partial P(X)}{\partial X_j} \times \frac{\partial X_j}{\partial \theta_k} \ .$$

- The matrix of derivatives of each state variable, or *Tangent state vector*, is by definition given by

$$\frac{\partial X_j}{\partial \theta_k} = \lim_{\Delta\theta \to 0} \frac{X_j(\theta_1, \ldots, \theta_k + \Delta\theta, \ldots, \theta_{N_\theta}) - X_j(\theta)}{\Delta\theta}.$$

- This gives the intuitive interpretation of $\partial X_j / \partial \theta_k$ in terms of the difference between the sample of the j-th component of the state vector obtained after an infinitesimal 'bump' of the k-th parameter, $X_j(\theta_1, \ldots, \theta_k + \Delta\theta, \ldots, \theta_{N_\theta})$, and the base sample $X_j(\theta)$, both calculated on *the same random realization*.

Example: Black Scholes Setup

Let us consider again the BS setup:

$$dX(t) = rX(t)dt + \sigma X(t)dW_t \quad X(0) = X_0 \ ,$$
$$X(T) = X_0 \exp\left[(r - \sigma^2/2)T + \sigma\sqrt{T}Z\right] \ ,$$
$$P(X(T)) = (X(T) - K)e^{-rT} \ .$$

The Pathwise Derivative estimator for Delta reads:

$$\bar{\theta}_{X_0} = \partial_{X_0} P(X(T)) = \frac{\partial_\theta P(X(T))}{\partial X(T)} \frac{\partial X(T)}{\partial X_0}$$
$$= \mathbb{I}(X(T) - K)\frac{X(T)}{X_0} \ ,$$

while the one for Vega is

$$\bar{\theta}_\sigma = \partial_\sigma P(X(T)) = \frac{\partial_\theta P(X(T))}{\partial X(T)} \frac{\partial X(T)}{\partial \sigma}$$
$$= \mathbb{I}(X(T) - K)X(T)(\sqrt{T}Z - \sigma T) \ .$$

Example: Digital Options

Let us consider again the BS setup, but this time, for a digital option

$$
\begin{aligned}
dX(t) &= rX(t)dt + \sigma X(t)dW_t \quad X(0) = X_0 \ , \\
X(T) &= X_0 \exp\left[(r - \sigma^2/2)T + \sigma\sqrt{T}Z\right] \ , \\
P(X(T)) &= \mathbb{I}(X(T) - K)e^{-rT} \ .
\end{aligned}
$$

In this case the Payout is not Lipschitz continuous and the method is not applicable. Indeed, the payoff is not differentiable. In the sense of distribution one has

$$
\frac{\partial P(X(T))}{\partial X(T)} = \delta(X(T) - K)e^{-rT} \ ,
$$

which is zero everywhere apart from when $X(T) = K$ when it's infinity and which cannot be sampled by Monte Carlo as it will give 0, *a.s.* .

Pathwise Derivative Method: Diffusions

- Consider the case for instance in which the state vector X is a path of a N-dimensional diffusive process,

$$dX(t) = \mu(X(t), t, \theta)\, dt + \sigma(X(t), t, \theta)\, dW_t,$$

with $X(t_0) = X_0$. Here the drift $\mu(X, t, \theta)$ and volatility $\sigma(X, t, \theta)$ are N-dimensional vectors and W_t is a N-dimensional Brownian motion with instantaneous correlation matrix $\rho(t)$ defined by $\rho(t)\, dt = \mathbb{E}_{\mathbb{Q}}\left[dW_t dW_t^T\right]$.

- The Pathwise Derivative Estimator may be rewritten as

$$\bar{\theta}_k = \sum_{l=1}^{M} \sum_{j=1}^{N} \frac{\partial\, P(X(T_1), \ldots, X(T_M))}{\partial X_j(T_l)} \frac{\partial X_j(T_l)}{\partial \theta_k} + \frac{\partial P_\theta(X)}{\partial \theta_k}$$

where we have relabeled the d components of the state vector X grouping together different observations $X_j(T_1), \ldots, X_j(T_M)$ of the same (j-th) asset.

Pathwise Derivative Method: Diffusions

- In particular, the components of the Tangent vector for the k-th sensitivity corresponding to observations at times $(T_1, \ldots, T_M)$ along the path of the j-th asset, say,

$$\Delta_{jk}(T_l) = \frac{\partial X_j(T_l)}{\partial \theta_k}$$

with $l = 1, \ldots, M$, can be obtained by solving a stochastic differential equation

$$\begin{aligned} d\Delta_{jk}(t) = \sum_{i=1}^{N} \left[\frac{\partial \mu_j(X(t), t; \theta)}{\partial X_i(t)}\, dt + \frac{\partial \sigma_j(X(t), t; \theta)}{\partial X_i(t)}\, dW_t^j \right] \Delta_{ik}(t) \\ + \left[\frac{\partial \mu_j(X(t), t; \theta)}{\partial \theta_k}\, dt + \frac{\partial \sigma_j(X(t), t; \theta)}{\partial \theta_k}\, dW_t^j \right], \end{aligned}$$

with the initial condition $\Delta_{jk}(0) = \partial X_j(0)/\partial \theta_k$.

Pathwise Derivative Method: Is it worth the trouble?

- The Pathwise Derivative Estimators of the sensitivities are mathematically equivalent to the estimates obtained by standard finite-differences approaches when using the same random numbers in both simulations and for a vanishing small perturbation. In this limit, the Pathwise Derivative Method and finite-differences estimators provide exactly the same estimators for the sensitivities, i.e., estimators with the same expectation value, *and* the same MC variance.
- As a result, the implementation effort associated with the Pathwise Derivative Method is generally justified if the computational cost of the Pathwise Estimator is significantly less than the corresponding finite-differences one.
- This is the case for instance in very simple models but difficult to achieve for those used in the financial practice.

Section 7

Adjoint Algorithmic Differentiation (AAD)

'Algebraic' Adjoint Methods

- In 2006 Mike Giles and Paul Glasserman published a ground breaking 'Smoking Adjoints' in Risk Magazine [3].
- They proposed a very efficient implementation of the Pathwise Derivative Method in in the context of the Libor Market Model for European payouts (generalized to Bermudan options by Leclerc *et al.* [4] and extended by Joshi *et al.* [5]).
- In a nutshell:
  1. Concentrate on the Tangent process and formulate it propagation in terms of Linear Algebra operations.
  2. Optimize the computation time by rearranging the order of the computations.
  3. Implement the rearranged sequence of operations.
- In the following we denote these Adjoint approaches as *algebraic*.

Libor Market Model

- Let's indicate with T_i, $i = 1, \ldots, N+1$, a set of $N+1$ bond maturities, with spacings $\delta = T_{i+1} - T_i$ (constant for simplicity).
- In a Lognormal setup the dynamics of the forward Libor rates as seen at time t for the interval $[T_i, T_{i+1})$, $L_i(t)$, takes the form

$$\frac{dL_i(t)}{L_i(t)} = \mu_i(L(t))dt + \sigma_i(t)^T dW_t,$$

$0 \leq t \leq T_i$, and $i = 1, \ldots, N$, where W_t is a d_W-dimensional standard Brownian motion, $L(t)$ is the N-dimensional vector of Libor rates, and $\sigma_i(t)$ the d_W-dimensional vector of volatilities, at time t.

Libor Market Model: Euler Discretization

- The dynamics of the forward Libor rates can be simulated by applying a Euler discretization to the logarithms of the forward rates.
- By dividing each interval $[T_i, T_{i+1})$ into N_s steps of equal width, $h = \delta/N_s$. This gives

$$\frac{L_i(t_{n+1})}{L_i(t_n)} = \exp\left[\left(\mu_i(L(t_n)) - ||\sigma_i(t_n)||^2/2\right) h + \sigma_i^T(n) Z(t_n)\sqrt{h}\right],$$

for $i = \eta(nh), \ldots, N$, and $L_i(t_{n+1}) = L_i(t_n)$ if $i < \eta(nh)$. Here Z is a d_W-dimensional vector of independent standard normal variables and t_0 is today.

Swaption Payout

- The standard test case are contracts with expiry T_m to enter in a swap with payments dates $T_{m+1}, \ldots, T_{N+1}$, at a fixed rate K

$$V(T_m) = \sum_{i=m+1}^{N+1} B(T_m, T_i)\delta(S_m(T_m) - K)^+,$$

where $B(T_m, T_i)$ is the price at time T_n of a bond maturing at time T_i

$$B(T_m, T_i) = \prod_{l=m}^{i-1} \frac{1}{1 + \delta L_l(T_m)},$$

and the swap rate reads

$$S_m(T_m) = \frac{1 - B(T_m, T_{N+1})}{\delta \sum_{l=m+1}^{N+1} B(T_m, T_l)}.$$

- Here we consider European style payouts. It is simple to generalize to Bermudan options (see [4]).

Pathwise Derivative Estimator for Delta

- The Pathwise Estimator for the Delta,

$$\bar{L}_k(t_0) = \frac{\partial V(T_m)}{\partial L_k(t_0)},$$

reads:

$$\bar{L}_k(t_0) = \sum_{j=1}^{N} \frac{\partial V(T_m)}{\partial L_j(T_m)} \frac{\partial L_j(T_m)}{\partial L_k(t_0)} = \frac{\partial V(T_m)}{\partial L(T_m)}^T \Delta(T_m),$$

where the Tangent process is

$$\Delta_{jk}(t) = \frac{\partial L_j(t)}{\partial L_k(t_0)}.$$

Euler Evolution of the Tangent Process

- By differentiating the Euler discretization for the Libor dynamics one obtains the Euler discretization of the Tangent process dynamics:

$$\Delta_{ik}(t_{n+1}) = \Delta_{ik}(t_n)\frac{L_i(t_{n+1})}{L_i(t_n)} + L_i(t_{n+1})\sum_{j=1}^{N}\frac{\partial \mu_i(t_n)}{\partial L_j(t_n)}\Delta_{jk}(t_n),$$

  where $\Delta_{ik}(t_0) = \partial L_i(t_0)/\partial L_k(t_0) = \delta_{jk}$.
- The evolution of the Tangent process can be expressed as the matrix recursion:

$$\Delta(t_{n+1}) = B(t_n)\Delta(t_{n+1})$$

  where $B(t_n)$ is an $N \times N$ matrix.

Standard (Forward) Implementation of the Pathwise Derivative Estimator

- A standard implementation for the calculation of the Pathwise Estimator

$$\bar{L}_k(T_0) = \frac{\partial V(T_m)}{\partial L(T_m)}^T \Delta(T_m),$$

where $T_m = t_M$, with $M = N_s \times m$, involves:

1. Apply the matrix recursion

$$\Delta(t_{n+1}) = B(t_n)\Delta(t_n),$$

M times starting from $\Delta_{ik}(t_0) = \delta_{jk}$ in order to compute $\Delta(T_m)$. The total cost in the general case is $O(MN^3)$.

2. Compute analytically the derivatives of the payoff

$$\frac{\partial V(T_m)}{\partial L(T_m)},$$

and multiply it by $\Delta(T_m)$, at a cost $O(N^2)$.

Standard (Forward) Implementation of the Pathwise Derivative Estimator

- This involves proceeding from right to left (i.e., forward in time):

$$\bar{L}_k(T_0) = \frac{\partial V(T_m)}{\partial L(T_m)}^T B(t_{M-1}) \dots B(t_0)\Delta(t_0)$$

  at a total computational cost $O(MN^3)$ in the general case.

- However, a simple observation allows a much more efficient implementation...

Adjoint (Backward) Implementation

- After completing the evolution of the Libor path up to T_m the right hand side of

$$\bar{L}_k(T_0) = \frac{\partial V(T_m)}{\partial L(T_m)}^T B(t_{M-1}) \dots B(t_0)\Delta(t_0)$$

can be computed from left to right (i.e., backward in time) by taking the transpose (i.e., the 'Adjoint')

$$\bar{L}_k(T_0)^T = \Delta(t_0)B(t_0)^T \dots B(t_{M-1})^T \frac{\partial V(T_m)}{\partial L(T_m)},$$

or equivalently as

$$\bar{L}_k(T_0)^T = \Delta(t_0)A(t_0)^T$$

where the $A(t_0)$ is the N dimensional vector given by the matrix-vector recursion

$$A(t_n) = B(t_n)^T A(t_{n+1}) \quad A_k(t_M) = \frac{\partial V(T_m)}{\partial L_k(T_m)}.$$

Forward vs Adjoint: Computational Complexity

Compare:

- The forward computation of the Pathwise Estimator

$$\bar{L}_k(T_0) = \frac{\partial V(T_m)}{\partial L(T_m)}^T B(t_{M-1}) \dots B(t_0)\Delta(t_0)$$

  which consists of M matrix-matrix products and a final matrix-vector product for an overall cost of $O(MN^3)$ in the general case.

- The Adjoint computation of the Pathwise Estimator

$$\bar{L}_k(T_0)^T = \Delta(t_0)B(t_0)^T \dots B(t_{M-1})^T \frac{\partial V(T_m)}{\partial L(T_m)},$$

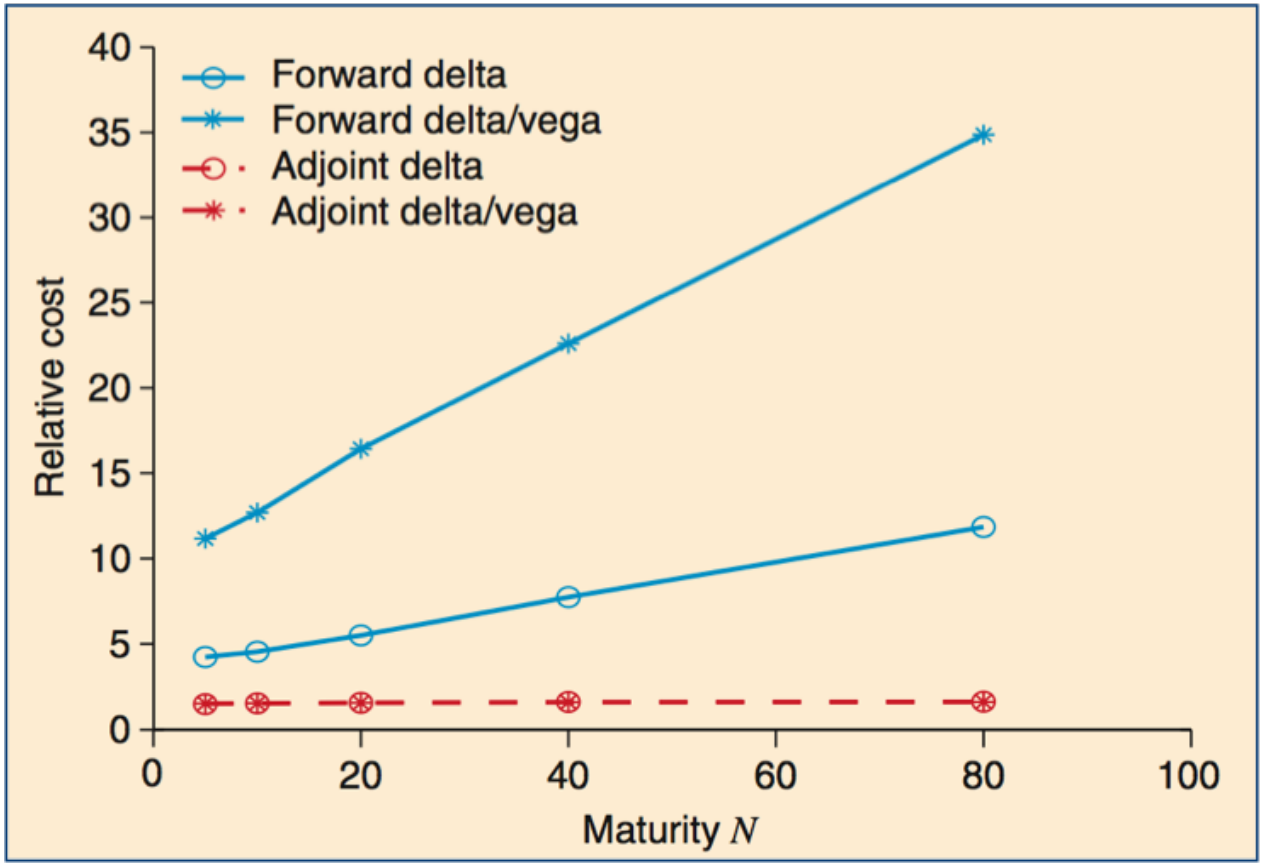
  which consists of $M+1$ matrix-vector products with an overall computational cost of $O(MN^2)$.

- The Adjoint implementation is $O(N)$ cheaper than the forward one.

Forward vs Adjoint: Computational Complexity

- As a result, computing the Pathwise Derivative Estimators for Delta has the same computational complexity of propagating the forward Libor rates and evaluating the payout.
- This means that we can get all the Delta sensitivities at a cost that is of the same order of magnitude than computing the payout (rather than $O(N)$ larger if we were computing the Deltas by bumping).
- The same results hold also for Vega.

Algebric Adjoint Methods



From Ref. [3]

Arbitrary number of sensitivities at a **fixed small** cost.

Limitations of Algebraic Adjoint Methods

The Libor Market Model is bit of an ad-hoc application:

- Difficult to generalize to Path Dependent Options or multi asset simulations.
- The required algebraic analysis is in general cumbersome.
- Not general enough for all the applications in Finance.
- The derivatives required are often not available in closed form.
- What about the derivatives of the payout?

Algorithmic Adjoint Approaches: AAD

- Adjoint implementations can be seen as instances of Adjoint Algorithmic Differentiation (AAD).
- Given that for each random realization the Payoff estimator can be seen as a map

$$\theta_k \to P(X(\theta_k)),$$

AAD allows the calculation of the Pathwise Derivative Estimators for **any** number of sensitivities

$$\bar{\theta}_k = \frac{\partial P(X(\theta_k))}{\partial \theta_k},$$

at a **small fixed cost**, similarly to the Algebric Adjoint applications of the Libor Market Model, but now in complete generality.

Section 8

AAD and the Pathwise Derivative Method

AAD and the Pathwise Derivative Method

- AAD provides a general design and programming paradigm for the efficient implementation of the Pathwise Derivative Method.
- This stems from the observation that the Pathwise Estimator in

$$\bar{\theta}_k \equiv \frac{dP_\theta(X)}{d\theta_k} = \sum_{j=1}^{d} \frac{\partial P_\theta(X)}{\partial X_j} \times \frac{\partial X_j}{\partial \theta_k} + \frac{\partial P_\theta(X)}{\partial \theta_k},$$

  is a l.c. of the rows of the Jacobian of the map $\theta \to X(\theta)$, with weights given by the X gradient of the payout function $P_\theta(X)$, plus the derivatives of the payout function with respect to θ.
- Both the calculation of the derivatives of the payout and of the linear combination of the rows of $\partial X/\partial\theta$ are tasks that can be performed efficiently by AAD.
- We know now that we can compute all the Pathwise sensitivities with respect to θ, $\bar{\theta}$, at a cost that is at most roughly 4 times the cost of calculating the payout estimator itself.

AAD enabled Monte Carlo Engines: Forward Sweep

- In a typical MC simulation, in order to generate each sample $X[i_{\mathrm{MC}}]$, the evolution of the process X is usually simulated, possibly by means of an approximate discretization scheme, by sampling $X(t)$ on a discrete grid of points, $0 = t_0 < t_1 < \ldots < t_n < \cdots < t_{N_s}$, a superset of the observation times $(T_1, \ldots, T_M)$.
- The state vector at time t_{n+1} is obtained by means of a function of the form

$$X(t_{n+1}) = \texttt{PROP}_n[\{X(t_m)\}_{m \leq n}, Z(t_n), \theta],$$

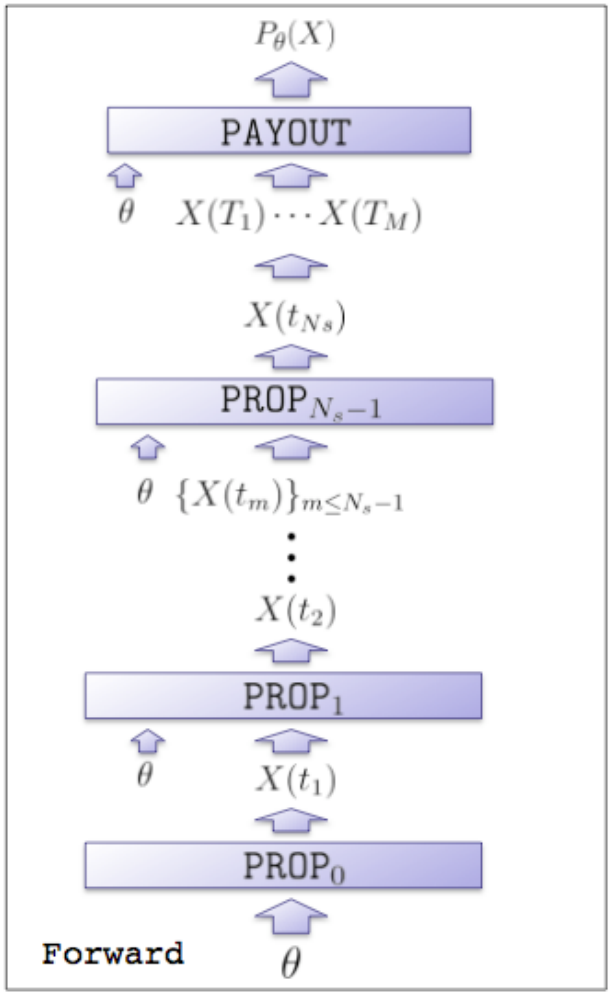
mapping the set of state vector values on the discretization grid up to t_n, $\{X(t_m)\}_{m \leq n}$, into the value of the state vector at time $t_n + 1$.

AAD enabled Monte Carlo Engines: Forward Sweep

- Note that in $X(t_{n+1}) = \texttt{PROP}_n[\{X(t_m)\}_{m\leq n}, Z(t_n), \theta]$:
  a. The propagation method is a function of the model parameters θ and of the particular time step considered.
  b. $Z(t_n)$ indicates the vector of uncorrelated random numbers which are used for the MC sampling in the step $n \to n+1$.
  c. The initial values of the state vector $X(t_0)$ are known quantities and they can be considered as components of θ so that the $n = 0$ step is of the form, $X(t_1) = \texttt{PROP}_0[Z(t_0), \theta]$.
- Once the full set of state vector values on the simulation time grid $\{X(t_m)\}_{m\leq N_s}$ is obtained, the subset of values corresponding to the observation dates is passed to the the payout function, evaluating the payout estimator $P_\theta(X)$ for the specific random sample $X[i_{\mathrm{MC}}]$

$$(X(T_1), \ldots, X(T_M)) \to P_\theta(X(T_1), \ldots, X(T_M)).$$

AAD enabled Monte Carlo Engines: Forward Sweep



Schematic illustration of the orchestration of a MC engine.

AAD enabled Monte Carlo Engines: Backward Sweep

- The evaluation of a MC sample of a Pathwise Estimator can be seen as an algorithm implementing a function of the form $\theta \to P(\theta)$.
- As a result, it is possible to design its Adjoint counterpart $(\theta, \bar{P}) \to (P, \bar{\theta})$ which gives (for $\bar{P} = 1$) the Pathwise Derivative Estimator $dP/d\theta_k$.
- The backward sweep can be simply obtained by reversing the flow of the computations, and associating to each function its Adjoint counterpart.

AAD enabled Monte Carlo Engines: Backward Sweep

- In particular, the first step of the Adjoint algorithm is the Adjoint of the payout evaluation $P = P(X, \theta)$. This is a function of the form

$$(\bar{X}, \bar{\theta}) = \bar{P}(X, \theta, \bar{P}),$$

where $\bar{X} = (\bar{X}(T_1), \ldots, \bar{X}(T_M))$ is the Adjoint of the state vector on the observation dates, and $\bar{\theta}$ is the Adjoint of the model parameters vector, respectively (for $\bar{P} = 1$)

$$\begin{aligned} \bar{X}(T_m) &= \frac{\partial P_\theta(X)}{\partial X(T_m)}, \\ \bar{\theta} &= \frac{\partial P_\theta(X)}{\partial \theta}, \end{aligned}$$

for $m = 1, \ldots, M$. The Adjoint of the state vector on the simulation dates corresponding to the observation dates are initialized at this stage. The remaining ones are initialized to zero.

AAD enabled Monte Carlo Engines: Backward Sweep

- The Adjoint state vector is then propagated backwards in time through the Adjoint of the propagation method, namely

$$(\{\bar{X}(t_m)\}_{m\leq n}, \bar{\theta}) \mathrel{+}= \texttt{PROP\_b}_n[\{X(t_m)\}_{m\leq n}, Z(t_n), \theta, \bar{X}(t_{n+1})],$$

for $n = N_s - 1, \ldots, 1$, giving

$$\bar{X}(t_m) \mathrel{+}= \sum_{j=1}^{N} \bar{X}_j(t_{n+1}) \frac{\partial X_j(t_{n+1})}{\partial X(t_m)},$$

with $m = 1, \ldots, n$,

$$\bar{\theta} \mathrel{+}= \sum_{j=1}^{N} \bar{X}_j(t_{n+1}) \frac{\partial X_j(t_{n+1})}{\partial \theta}.$$

AAD enabled Monte Carlo Engines: Backward Sweep

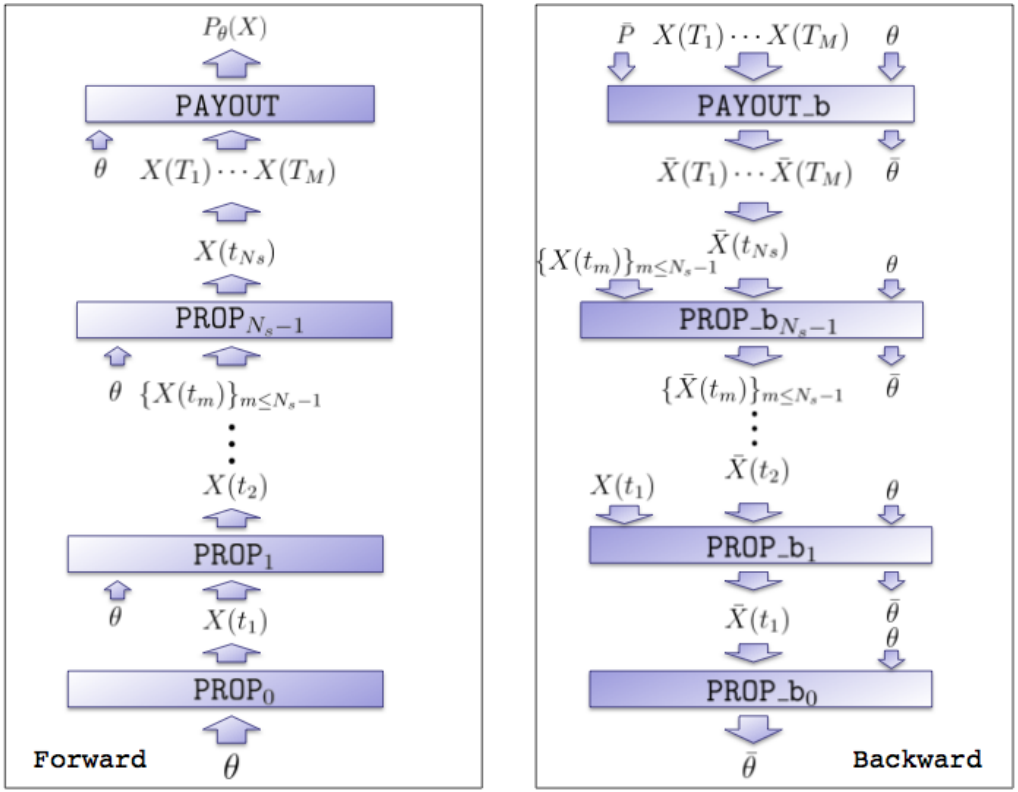
- Here, according to the principles of AAD, the Adjoint of the propagation method takes as arguments the inputs of its forward counterpart, namely the state vectors up to time t_n, $\{X(t_m)\}_{m\leq n}$, the vector of random variates $Z(t_n)$, and the θ vector. The additional input is the Adjoint of the state vector at time t_{n+1}, $\bar{X}(t_{n+1})$.
- The return values of $\texttt{PROP\_b}_n$ are the contributions associated with the step $n+1 \to n$ to the Adjoints of
  - i) the state vector $\{\bar{X}(t_m)\}_{m\leq n}$;
  - ii) the model parameters $\bar{\theta}_k$, $k = 1, \ldots, N_\theta$.
- The final step of the backward propagation corresponds to the Adjoint of $X(t_1) = \texttt{PROP}_0[Z(t_0), \theta]$, giving

$$\bar{\theta} \mathrel{+}= \texttt{PROP\_b}_0[X(t_0)\, Z(t_0), \theta, \bar{X}(t_1)],$$

  i.e., the final contribution to the Adjoints of the model parameters.
- It is easy to verify that the final result is the Pathwise Derivative Estimator $dP/d\theta_k$ for all k's on the given MC path.

AAD enabled Monte Carlo Engines: The complete blueprint

- The resulting algorithm can be illustrated as follows:



Schematic illustration of the orchestration of an AAD enabled MC engine.

Diffusion Processes and Euler Discretization

- As a first example, consider the case in which the underlying factors follow multi dimensional diffusion processes introduced in slide 107

$$dX(t) = \mu(X(t), t, \theta)\, dt + \sigma(X(t), t, \theta)\, dW_t.$$

- In this case, the evolution of the process X is usually approximated by sampling $X(t)$ on a discrete grid of points by means, for instance, of an Euler scheme, so that the propagation function

$$X(t_{n+1}) = \texttt{PROP}_n[\{X(t_m)\}_{m \leq n}, Z(t_n), \theta]$$

implements the rule

$$X(t_{n+1}) = X(t_n) + \mu(X(t_n), t_n, \theta)\, h_n + \sigma(X(t_n), t_n, \theta)\, \sqrt{h_n}\, Z(t_n),$$

where $h_n = t_{n+1} - t_n$, and $Z(t_n)$ is a N-dimensional vector of correlated unit normal random variables.

Diffusion Processes and Euler Discretization: Forward Sweep

- In particular, given the state vector at time t_n, $X(t_n)$, and the vector $Z(t_n)$, one can implement the method $\texttt{PROP}_n$ according to the following steps:

Step 1. Compute the drift vector, by evaluating the function:

$$\mu(t_n) = \mu(X(t_n), t_n, \theta) \ .$$

Step 2. Compute the volatility vector, by evaluating the function:

$$\sigma(t_n) = \sigma(X(t_n), t_n, \theta) \ .$$

Step 3. Compute the function

$$(X(t_n), \mu(t_n), \sigma(t_n), Z(t_n), \theta) \to X(t_{n+1}) \ ,$$

defined by

$$X(t_{n+1}) = X(t_n) + \mu(t_n)h_n + \sigma(t_n)\sqrt{h_n}Z(t_n) \ .$$

Diffusion Processes and Euler Discretization: Backward Sweep

- The corresponding Adjoint method PROP_b$_n$ is executed from time step t_{n+1} to t_n and consists of the Adjoint counterpart of each of the steps above executed in reverse order, namely:

Step $\bar{3}$. Compute the Adjoint of the function defined by Step 3. This is a function

$$(X(t_n), \mu(t_n), \sigma(t_n), Z(t_n), \bar{X}(t_{n+1})) \to \bar{X}(t_n)$$

defined by the instructions

$$\begin{aligned}
&\bar{X}(t_n) \mathrel{+}= \bar{X}(t_{n+1}),\\
&\bar{\mu}(t_n) = 0, \qquad && \bar{\mu}(t_n) \mathrel{+}= \bar{X}(t_{n+1}) h_n,\\
&\bar{\sigma}(t_n) = 0, \qquad && \bar{\sigma}(t_n) \mathrel{+}= \bar{X}(t_{n+1}) \sqrt{h_n}\, Z(t_n),\\
&\bar{Z}(t_n) = 0, \qquad && \bar{Z}(t_n) \mathrel{+}= \bar{X}(t_{n+1}) \sqrt{h_n}\, \sigma(t_n).
\end{aligned}$$

Diffusion Processes and Euler Discretization: Backward Sweep

- And:

Step $\bar{2}$. Compute the Adjoint of the volatility function in Step 2, namely

$$\bar{X}_i(t_n) \mathrel{+}= \sum_{j=1}^{N} \bar{\sigma}_j(t_n) \frac{\partial \sigma_j(t_n)}{\partial X_i}, \qquad \bar{\theta}_k \mathrel{+}= \sum_{j=1}^{N} \bar{\sigma}_j(t_n) \frac{\partial \sigma_j(t_n)}{\partial \theta_k},$$

for $i = 1, \ldots, N$ and $k = 1, \ldots, N_\theta$.

Step $\bar{1}$. Compute the Adjoint of the drift function in Step 1, namely

$$\bar{X}_i(t_n) \mathrel{+}= \sum_{j=1}^{N} \bar{\mu}_j(t_n) \frac{\partial \mu_j(t_n)}{\partial X_i(t_n)}, \qquad \bar{\theta}_k \mathrel{+}= \sum_{j=1}^{N} \bar{\mu}_j(t_n) \frac{\partial \mu_j(t_n)}{\partial \theta_k},$$

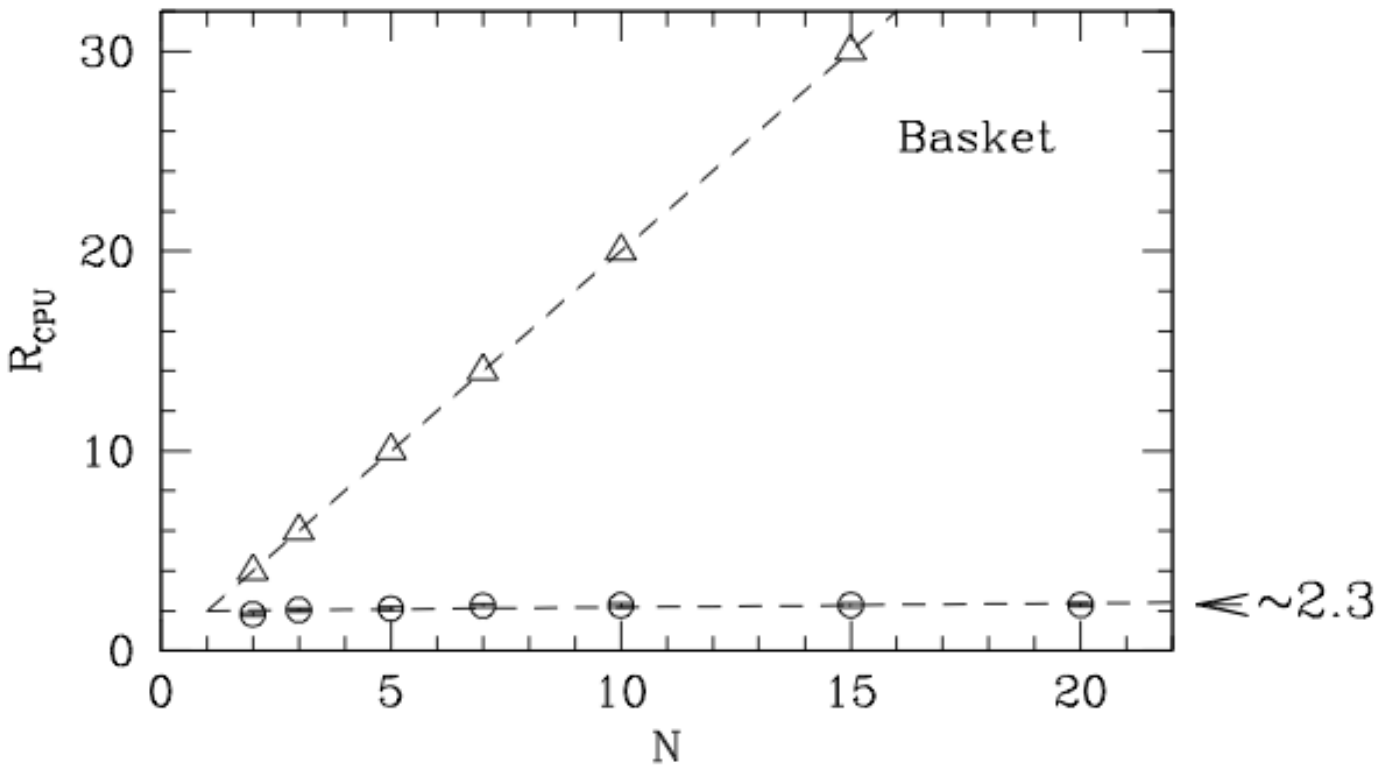
for $i = 1, \ldots, N$ and $k = 1, \ldots, N_\theta$.

Diffusion Processes and Euler Discretization: Backward Sweep

- Note that, the variables $\bar{X}(t_{n+1})$, $\bar{X}(t_n)$ and $\bar{\theta}$ typically contain on input the derivatives of the payout function. During the backward propagation $\bar{X}(t_n)$ (resp. $\bar{\theta}$) accumulate several contributions, one for each Adjoint of an instruction in which $X(t_n)$ (resp. θ) is on the right hand side of the assignment operator in the forward sweep (Steps 1-3).
- The implementation of the Adjoint of the drift and volatility functions in Step $\bar{2}$ and Step $\bar{1}$ is problem dependent. In many cases, the drift and volatility may be represented by computer routines self-contained enough to be processed by means of an automatic differentiation tool, thus facilitating the implementation.

Basket Options: Results

- Let's consider again the Basket Option example introduced earlier for the Payoff.



CPU time ratios for the calculation of Delta and Vega Risk as a function of the number of underlying assets, N: circles (AAD), triangles (Bumping).

Basket Options: Comments

- The performance of the AAD implementation of the Pathwise Derivative Method in this setup is well within the expected bounds.
- In particular, the computation of the $2 \times N$ sensitivities for the N assets requires a very small overhead (of about 130%) with respect to the calculation of the option value itself. This is true for any number of underlying assets.
- This is in stark contrast with the relative cost of evaluating the same sensitivities by means of finite-differences, scaling linearly with the number of assets.
- For typical applications this clearly results in remarkable speedups with respect to bumping.

Libor Market Model

- Let's consider again the application of the seminal paper by Giles and Glasserman [3] See slide 112 and ff. Recall that:
- T_i, $i = 1, \ldots, N+1$, is a set of $N+1$ bond maturities, with spacings $\delta = T_{i+1} - T_i$ (constant for simplicity).
- In a Lognormal setup the dynamics of the forward Libor rates as seen at time t for the interval $[T_i, T_{i+1})$, $L_i(t)$, takes the form

$$\frac{dL_i(t)}{L_i(t)} = \mu_i(L(t))dt + \sigma_i(t)^T dW_t,$$

$0 \leq t \leq T_i$, and $i = 1, \ldots, N$, where W_t is a d_W-dimensional standard Brownian motion, $L(t)$ is the N-dimensional vector of Libor rates, and $\sigma_i(t)$ the d_W-dimensional vector of volatilities, at time t.

Libor Market Model: Euler Discretization

- The dynamics of the forward Libor rates can be simulated by applying a Euler discretization to the logarithms of the forward rates.
- By dividing each interval $[T_i, T_{i+1})$ into N_s steps of equal width, $h = \delta/N_s$. This gives

$$\frac{L_i(t_{n+1})}{L_i(t_n)} = \exp\left[\left(\mu_i(L(t_n)) - ||\sigma_i(t_n)||^2/2\right)h + \sigma_i^T(n)Z(t_n)\sqrt{h}\right],$$

for $i = \eta(nh), \ldots, N$, and $L_i(t_{n+1}) = L_i(t_n)$ if $i < \eta(nh)$. Here Z is a d_W-dimensional vector of independent standard normal variables and t_0 is today.

Swaption Payout

- The standard test case are contracts with expiry T_m to enter in a swap with payments dates $T_{m+1}, \ldots, T_{N+1}$, at a fixed rate K

$$V(T_m) = \sum_{i=m+1}^{N+1} B(T_m, T_i)\delta(S_n(T_m) - K)^+,$$

where $B(T_m, T_i)$ is the price at time T_m of a bond maturing at time T_i

$$B(T_m, T_i) = \prod_{l=m}^{i-1} \frac{1}{1 + \delta L_l(T_m)},$$

and the swap rate reads

$$S_m(T_m) = \frac{1 - B(T_m, T_{N+1})}{\delta \sum_{l=m+1}^{N+1} B(T_m, T_l)}.$$

- Here we consider European style payouts. It is simple to generalize to Bermudan options (see [4]).

Libor Market Model: Forward Sweep

```
PROP(n, L[,], Z, lambda[], L0[])

  if(n=0)
    for(i= 1 .. N)
      L[i,n] = L0[i]; Lhat[i,n] = L0[i];

  for (i = 1 .. eta[n]-1)
    L[i,n+1] = L[i,n];  // settled rates

  sqez = sqrt(h)*Z;
  v = 0.; v_pc = 0.;
  for (i = eta[n] .. N)
    lam = lambda[i-eta[n]+1];
    c1 = del*lam; c2 = h*lam;
    v += (c1*L[i,n])/(1.+del*L[i,n]);
    vrat = exp(c2*(-lam/2.+v)+lam*sqez);
    // standard propagation with the Euler drifts
    Lhat[i,n+1] = L[i,n]*vrat;
    // (n + 1) drift term
    v_pc += (c1*Lhat[i,n+1])/(1.+del*Lhat[i,n+1]);
    vrat_pc = exp(c2*(-lam/2.+(v_pc+v)/2.)+lam*sqez);
    // actual propagation using the average drift
    L[i,n+1]  = L[i,n]*vrat_pc;
    // store what is needed for the reverse sweep
    hat_scra[i,n+1] = vrat*((v-lam)*h+sqez);
    scra[i,n+1] = vrat_pc*(((v_pc+v)/2.-lam)*h+sqez);
```

Pseudocode implementing the propagation method $\texttt{PROP}_n$ for the Libor Market Model for $d_W = 1$, under the predictor corrector Euler approximation.

Libor Market Model: Backward Sweep

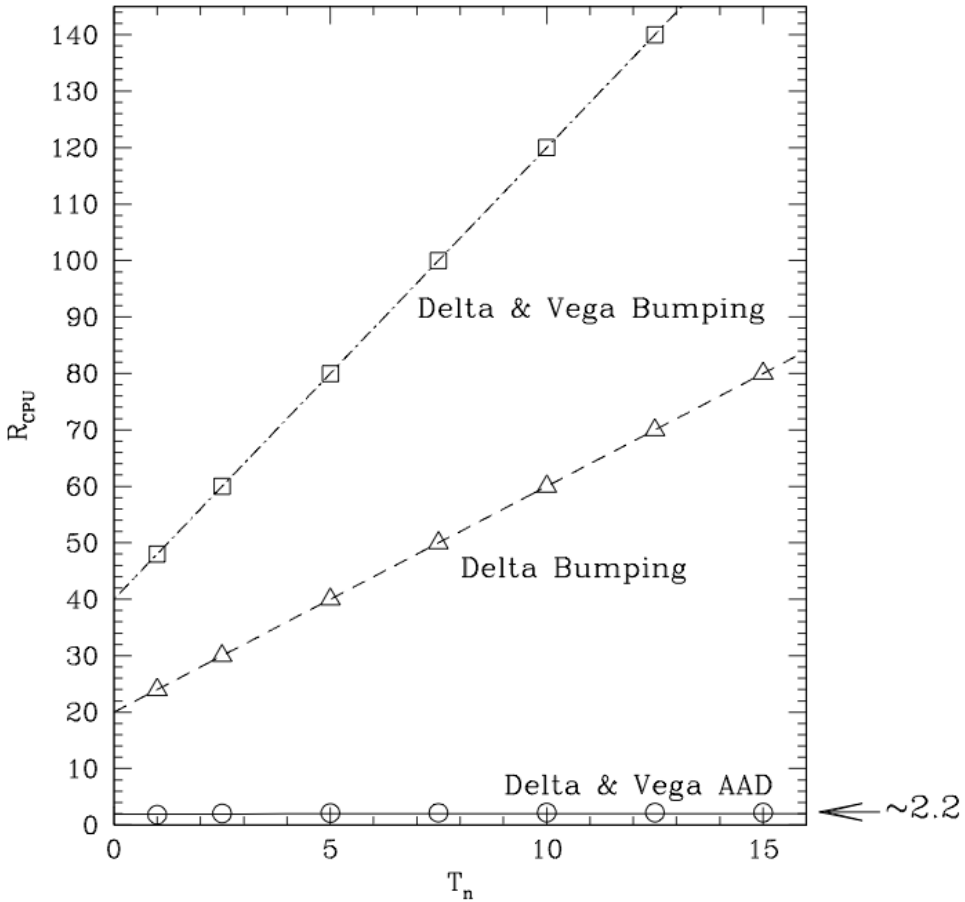
```
PROP_b(n, L[,], Z, lambda[], L0[], lambda_b[], L0_b[])

  v_b = 0.; v_pc_b = 0.;
  for (i=N .. eta[n])
    lam = lambda[i-eta[n]+1];
    c1 = del*lam; c2 = lam*h;
    //L[i,n+1] = L[i,n]*vrat_pc
    vrat_pc = L[i,n+1]/L[i,n];
    vrat_pc_b = L[i,n]*L_b[i,n+1];
    L_b[i,n] = vrat_pc*L_b[i,n+1];
    // vrat_pc = exp(c2*(-lam/2.+(v_pc+v)/2.)+lam*sqez)
    lambda_b[i-eta[n]+1] += scra[i,n+1]*vrat_pc_b;
    v_pc_b += vrat_pc*lam*h*vrat_pc_b/2.;
    v_b += vrat_pc*lam*h*vrat_pc_b/2.;
    // v_pc += (c1*Lhat[i,n+1])/(1.+del*Lhat[i,n+1])
    rpip = 1./(del*Lhat[i,n+1]+1.);
    Lhat_b[i,n+1] += (c1-c1*Lhat[i,n+1]*del*rpip)*rpip*v_pc_b;
    c1_b = Lhat[i,n+1]*rpip*v_pc_b;
    // Lhat[i,n+1] = L[i,n]*vrat
    vrat_b = L[i,n]*Lhat_b[i,n+1];
    vrat = Lhat[i,n+1]/L[i,n];
    L_b[i,n] += vrat*Lhat_b[i,n+1];
    // vrat = exp(lam*h*(-lam/2.+v)+lam*sqez)
    lambda_b[i-eta[n]+1] += hat_scra[i,n+1]*vrat_b;
    v_b += vrat*lam*h*vrat_b;
    // v += (c1*L[i,n])/(1.+del*L[i,n])
    rpip = 1./(del*L[i,n]+1.);
    L_b[i,n] += (c1-c1*L[i,n]*del*rpip)*rpip*v_b;
    c1_b += L[i+n]*rpip*v_b;
    // lam = lambda[i-eta[n]+1]; c1 = del*lam
    lambda_b[i-eta[n]+1] += del*c1_b;

  for (i=eta[n]-1 .. 1)
    // L[i,n+1] = L[i,n]
    L_b[i,n] += L_b[i,n];

  if(n=0)
    for(i=1 .. N)
      // L[i,n] = L0[i]
      L0_b[i] = L0[i,0];
```

Adjoint of the propagation method $\texttt{PROP\_b}_n$ [9].

Libor Market Model: Comments on the Code

- The algebraic formulation discussed in [5] comes with a significant analytical effort. Instead, as illustrated in the Figure above, the AAD implementation is quite straightforward.
- According to the general design of AAD, this simply consists of the Adjoints of the instructions in the forward sweep executed in reverse order.
- In this example, the information computed by `PROP` that is required by `PROP_b` is stored in the vectors `scra` and `hat_scra`.
- By inspecting the structure of the pseudocode it also appears clear that the computational cost of `PROP_b` is of the same order as evaluating the original function.

Libor Market Model: Results



Ratio of the CPU time required for the calculation of Delta and Vega and the time to calculate the option value for the Swaption as a function of the option expiry T_n.

Exercise

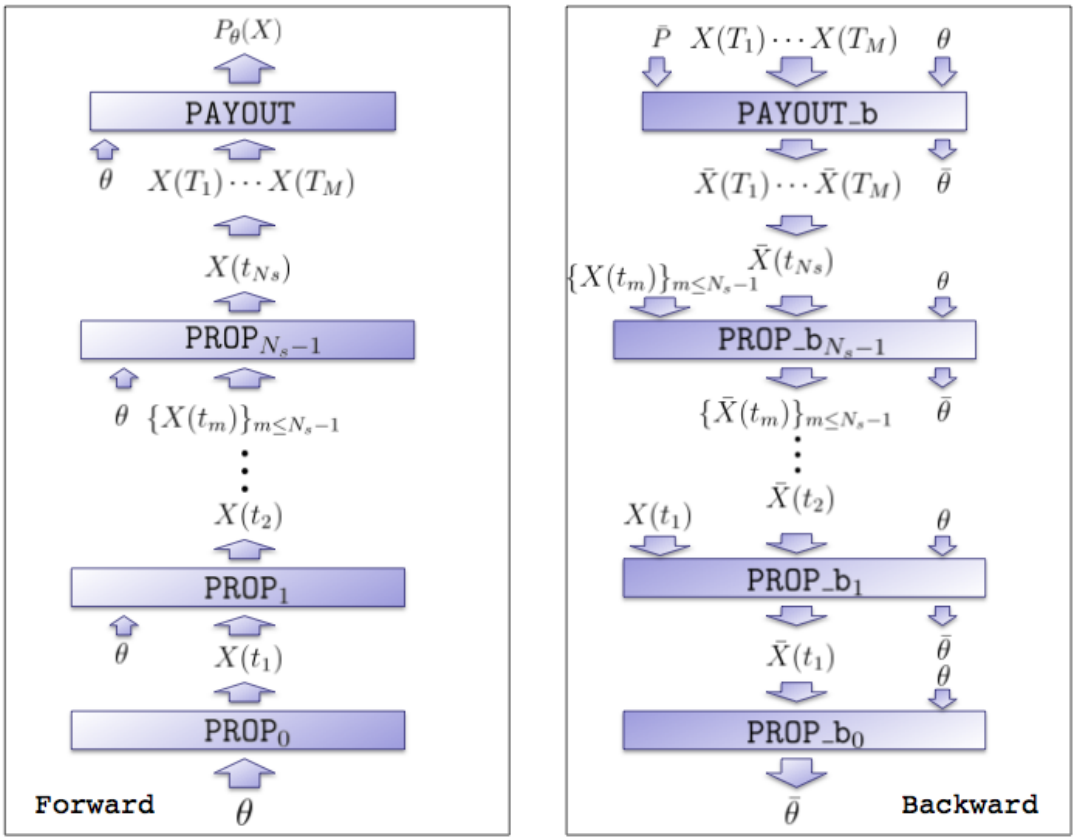
Exercise: Assuming lognormal processes implement the calculation of Delta and Vega for the Basket call option using bumping, the pathwise derivative method, the tangent mode and the adjoint model of AD. Compare the numerical results between the different methods and plot the timing as a function of the size of the basket.

Section 9

Correlation Greeks and Binning Techniques

Correlation Structure of the Random Variates

Recall the general AAD MC design for the computation of the estimators on each MC path:



Correlation Structure of the Random Variates

- In the AAD MC design we have assumed for simplicity that the random variates $Z(t_n)$ entering in the propagation method:

$$X(t_{n+1}) = \texttt{PROP}_n[\{X(t_m)\}_{m \leq n}, Z(t_n), \theta],$$

  are dummy variables carrying no interesting sensitivities.
- As a result, in the corresponding adjoint propagation methods:

$$(\{\bar{X}(t_m)\}_{m \leq n}, \bar{\theta}) \mathrel{+}= \texttt{PROP\_b}_n[\{X(t_m)\}_{m \leq n}, Z(t_n), \theta, \bar{X}(t_{n+1})],$$

  the adjoint of the random variates $\bar{Z}(t_n)$ do not appear among the outputs.
- If we want to compute the sensitivities with respect to the correlation structure of the random variates, this scheme needs to be extended.

Correlation Structure of the Random Variates

- In a typical setup, the random variates Z_i driving the random processes are correlated.
- For instance, assume that the random variates $Z(t_n)$ are jointly normal, and denote with $\rho_{ij}(t_m) = \mathbb{E}[Z_i(t_m)Z_j(t_m)]$ the correlation matrix.
- Uncorrelated random variates $Z'(t_n)$ are therefore mapped into their correlated counterparts $Z(t_n)$ and then used to implement the propagation step $X(t_n) \to X(t_{n+1})$ so that the propagation step is modified as

$$\begin{aligned} Z(t_n) &= \texttt{CORRELATE}(Z'(t_n), \theta) \\ X(t_{n+1}) &= \texttt{PROP}_n[\{X(t_m)\}_{m \leq n}, Z(t_n), \theta] \ , \end{aligned}$$

where we have included the correlation parameters defining the correlation matrix ρ in the vector θ.

Modified Adjoint of the Propagation Step

- The adjoint of the Propagation Step

$$X(t_{n+1}) = \texttt{PROP}_n[\{X(t_m)\}_{m\leq n}, Z(t_n), \theta],$$

is modified as

$$(\{\bar{X}(t_m)\}_{m\leq n}, \bar{\theta}, \bar{Z}(t_n)) \mathrel{+}= \texttt{PROP\_b}_n[\{X(t_m)\}_{m\leq n}, Z(t_n), \theta, \bar{X}(t_{n+1})],$$

where

$$\bar{X}(t_m) \mathrel{+}= \sum_{j=1}^{N} \bar{X}_j(t_{n+1}) \frac{\partial X_j(t_{n+1})}{\partial X(t_m)} \qquad \bar{\theta} \mathrel{+}= \sum_{j=1}^{N} \bar{X}_j(t_{n+1}) \frac{\partial X_j(t_{n+1})}{\partial \theta},$$

with $m = 1, \ldots, n$. Here the additional output is given by the adjoint of the correlated variates:

$$\bar{Z}(t_n) \mathrel{+}= \sum_{j=1}^{N} \bar{X}_j(t_{n+1}) \frac{\partial X_j(t_{n+1})}{\partial Z(t_n)}.$$

Adjoint of the Correlation Step

- The adjoint of the Correlation Step

$$Z(t_n) = \texttt{CORRELATE}(Z'(t_n), \theta),$$

reads

$$\bar{\theta} \mathrel{+}= \texttt{CORRELATE\_b}(Z'(t_n), \theta, \bar{Z}(t_n)),$$

corresponding to the operation

$$\bar{\theta} \mathrel{+}= \sum_{j=1}^{N} \bar{Z}'_j(t_n) \frac{\partial Z_j(t_n)}{\partial \theta}$$

updating the components of the vector θ corresponding to the adjoint of the correlation parameters.

Example: Cholesky Factorization

- In a simple setup the method `CORRELATE` generally involves the so-called Cholesky factorization of an $N \times N$ correlation matrix ρ.
- Recall that the Cholesky factorization of a Hermitian positive-definite matrix ρ produces a lower triangular $N \times N$ matrix L such that $\rho = LL^T$.
- Given the Cholesky factor L, and a vector of N uncorrelated normal Z', it is immediate to verify that $Z = LZ'$ are correlated normal such that $\mathbb{E}[Z_i Z_j] = \rho_{ij}$.

Adjoint of the Cholesky Factorization

- When implemented in terms of the Cholesky factorization, the method `CORRELATE` reads

  Step 1 Perform Cholesky factorization, say $L = \texttt{CHOLESKY}(\rho)$.
  Step 2 Compute: $Z = LZ'$.

- The corresponding method `CORRELATE_b` reads

  Step $\bar{2}$ Compute: $\bar{L} = \bar{Z}Z'^{t}$.
  Step $\bar{1}$ Compute: $\bar{\rho} = \texttt{CHOLESKY\_b}(\rho, \bar{L})$, where

$$\bar{\rho}_{ij} = \sum_{l,m=1}^{N} \frac{\partial L_{l,m}}{\partial \rho_{ij}} \bar{L}_{lm},$$

  providing the sensitivities with respect to the entries of the correlation matrix. These are copied in the appropriate components of the vector $\bar{\theta}$.

- Note that Z' are now dummy integration variables (sampled stochastically). Thefore their adjoints $\bar{Z}'$ are not computed.

Adjoint of the Cholesky Factorization (Pseudocode)

```
Cholesky_b(rho, L_b,rho_b)

   // Forward Sweep
   for (i=0 .. n-1)
      for (j=i .. n-1)
      sum[i,j] = rho[i,j];
      for (k=i-1 .. 0)
         sum[i,j] -= L[i,k] * L[j,k];
         if (i == j)
           L[i,i] = sqrt(sum[i,j]);
         else
           L[j,i] = sum[i,j] / L[i,i];

   // Backward Sweep
   for (i=n-1 .. 0)
      for (j=n-1 .. i)
         sum_b =  0.0;

         if (i == j)
           if (sum[i,j] == 0.0)
              sum_b = 0.0;
           else
              sum_b = L_b[i,j]/( 2.0 * L[i,j]);
           L_b[i,j] = 0.0;
         else
           sum_b = L_b[j,i]/L[i,i];
           L_b[i,i] -= sum[i,j] * sum_b / L[i,i];
           L_b[j,i] = 0.0;

         for (k=i-1 .. 0)
           L_b[i,k] -= L[j,k]*sum_b;
           L_b[j,k] -= L[i,k]*sum_b;

         rho_b[i,j] += sum_b;
```

The adjoint algorithm contains the original Cholesky factorization plus a backward sweep with the same complexity and a similar number of operations [10].

Hence, as expected, the computational cost is just a small multiple (of order 2, in this case) of the cost of evaluating the original factorization.

Adjoint of the Cholesky Factorization

- The Cholesky factorization $L = \texttt{CHOLESKY}(\rho)$ does not depend on the random variates Z therefore it can be performed before the first Monte Carlo path is performed. As a result, `CORRELATE` consists of the matrix multiplication $Z = LZ'$, only.

- Similarly the Adjoint of `CORRELATE_b` consists only of the step $\bar{L} = \bar{Z}'Z^t$, ($\bar{L}$ will contain the adjoint of the Cholesky factors L rather than the entries of the correlation matrix ρ) and the Adjoint of the Cholesky factorization

$$\bar{\rho} = \texttt{CHOLESKY\_b}(\rho, \bar{L})$$

can be performed after the end of the backward sweep after the last MC path.

Statistical Uncertainties

- Given the MC estimators for the Cholesky factors sensitivities $\langle \bar{L} \rangle = \langle \partial V(X)/\partial L \rangle$ and their statistical uncertainties

$$\langle \bar{L} \rangle = \frac{1}{N_{\mathrm{MC}}} \sum_{i_{\mathrm{MC}}=1}^{N_{\mathrm{MC}}} \bar{L}(X[i_{\mathrm{MC}}]) \qquad \sigma_{\bar{L}} = \sqrt{\frac{1}{N_{\mathrm{MC}}} \sum_{i_{\mathrm{MC}}=1}^{N_{\mathrm{MC}}} \left(\bar{L}(X[i_{\mathrm{MC}}])^2 - \langle \bar{L} \rangle \right)^2}$$

- One can compute the estimator for the correlation sensitivities via the Cholsesky factorization

$$\langle \bar{\rho} \rangle = \texttt{CHOLESKY\_b}(\rho, \langle \bar{L} \rangle)$$

but not their sensitivities:

$$\sigma_{\bar{\rho}} \neq \texttt{CHOLESKY\_b}(\rho, \sigma_{\bar{L}})$$

- Performing the adjoint of the Cholesky decomposition once per simulation does not allow the calculation of a confidence interval for the correlation sensitivities.

Path by Path Adjoint Cholesky Factorization

- An alternative approach would be to convert $\bar{L}$ to $\bar{\rho}$ for each individual path $i_{\mathrm{MC}} = 1, \ldots, N_{\mathrm{MC}}$

$$\bar{\rho}(X[i_{\mathrm{MC}}]) = \texttt{CHOLESKY\_b}(\rho, \bar{L}(X[i_{\mathrm{MC}}]))$$

and then compute the average and standard deviation of $\bar{\rho}[i_{\mathrm{MC}}]$ in the usual way:

$$\langle \bar{\rho} \rangle = \frac{1}{N_{\mathrm{MC}}} \sum_{i_{\mathrm{MC}}=1}^{N_{\mathrm{MC}}} \bar{\rho}(X[i_{\mathrm{MC}}]) \qquad \sigma_{\bar{\rho}} = \sqrt{\frac{1}{N_{\mathrm{MC}}} \sum_{i_{\mathrm{MC}}=1}^{N_{\mathrm{MC}}} \left(\bar{\rho}(X[i_{\mathrm{MC}}])^2 - \langle \bar{\rho} \rangle \right)^2}$$

However, this is rather costly.

Binning

- An excellent compromise between these two extremes is to divide the N_{MC} paths into N_{B} 'bins' of equal size $n = N/N_{\mathrm{B}}$.
- For each bin $j_{\mathrm{B}} = 1, \ldots, N_{\mathrm{B}}$, an average value of $\langle \bar{L} \rangle_{j_{\mathrm{B}}}$ is computed

$$\langle \bar{L} \rangle_{j_{\mathrm{B}}} = \frac{1}{n} \sum_{i_{\mathrm{MC}}=1}^{n} \bar{L}(X[i_{\mathrm{MC}}])$$

and converted into a corresponding value for

$$\langle \bar{\rho} \rangle_{j_{\mathrm{B}}} = \texttt{CHOLESKY\_b}(\rho, \langle \bar{L} \rangle_{j_{\mathrm{B}}}).$$

Binning

- These N_B estimates for $\bar{\rho}$ can then be combined in the usual way to form an overall estimate of the correlation risk:

$$\langle \bar{\rho} \rangle = \frac{1}{N_{\mathrm{B}}} \sum_{j_{\mathrm{B}}=1}^{N_{\mathrm{B}}} \langle \bar{\rho} \rangle_{j_{\mathrm{B}}} = \frac{1}{N_{\mathrm{MC}}} \sum_{i_{\mathrm{MC}}=1}^{N_{\mathrm{MC}}} \bar{\rho}(X[i_{\mathrm{MC}}]),$$

where the second equality follows from the linearity of the adjoint functions, and the associated confidence interval:

$$\sigma_{\bar{\rho}} = \sqrt{\frac{1}{N_{\mathrm{B}}} \sum_{j_{\mathrm{B}}=1}^{N_{\mathrm{B}}} \left(\langle \bar{\rho} \rangle_{j_{\mathrm{B}}}^2 - \langle \bar{\rho} \rangle \right)^2}.$$

Binning

- In the standard evaluation, the cost of the Cholesky factorization is $O(N^3)$ (where N is the number of random factors), and the cost of the MC sampling is $O(N_{\mathrm{MC}}N^2)$, so the total cost is $O(N^3 + N_{\mathrm{MC}}N^2)$. Since N_{MC} is always much greater than N, the cost of the Cholesky factorization is usually negligible.
- The cost of the adjoint steps in the MC sampling is also $O(N_{\mathrm{MC}}N^2)$, and when using N_B bins the cost of the adjoint Cholesky factorization is $O(N_{\mathrm{B}}N^3)$.
- To obtain an accurate confidence interval, but with the cost of the Cholesky factorisation being negligible, requires that N_{B} is chosen so that $1 \ll N_{\mathrm{B}} \ll N_{\mathrm{MC}}/N$.
- Without binning, i.e., using $N_{\mathrm{B}} = N_{\mathrm{MC}}$, the cost to calculate the average of the estimators for $\langle\rho\rangle$ is $O(N_{\mathrm{MC}}N^3)$, and so the relative cost compared to the evaluation of the option value is $O(N)$.

Binning and Risk Transforms

- We have presented Binning in the context of the calculation of correlation risk, but there is nothing specific to correlation. In fact these ideas can be applied everytime some computational preprocessing is performed before the MC simulation, and we need to transform the adjoint MC estimators and their confidence intervals into the corresponding quantities for the inputs of such preprocessing.
- This is the case for instance when a calibration routine performed before the MC simulation transforms some market inputs $M = (M_1, \ldots, M_{N_M})$, corresponding to the observable prices of securities which the model is calibrated to, into the set of internal model parameters that are used in the MC simulation, θ:

$$\theta = \texttt{CALIBRATION}(M).$$

Binning and Risk Transforms

- The binned MC estimators of the adjoint of the internal model parameters $\langle \bar{\theta} \rangle_{j_\mathrm{B}}$ can be transformed into binned MC estimators of the market inputs

$$\langle \bar{M} \rangle_{j_\mathrm{B}} = \texttt{CALIBRATION\_B}(M, \langle \bar{\theta} \rangle_{j_\mathrm{B}}).$$

- Then their distribution can be used to construct the overall MC estimator and the associated statistical uncertainty

$$\langle \bar{M} \rangle = \frac{1}{N_\mathrm{B}} \sum_{j_\mathrm{B}=1}^{N_\mathrm{B}} \langle \bar{M} \rangle_{j_\mathrm{B}},$$

$$\sigma_{\bar{M}} = \sqrt{\frac{1}{N_\mathrm{B}} \sum_{j_\mathrm{B}=1}^{N_\mathrm{B}} \left(\langle \bar{M} \rangle_{j_\mathrm{B}}^2 - \langle \bar{M} \rangle \right)^2}.$$

Credit Basket Contracts

- Credit basket contracts are derivatives that are contingent on credit events (defaults for short) of a pool of reference entities typically sovereign, financial or corporate. Generally the credit event is defined as failure to pay a specific liability, say a coupon on a specific bond or category of bonds referenced by the contract, but it can include other events not involving a proper default, like a restructuring of the debt, or regulatory action on a financial institution.
- n-th to default, Collateralized Debt Obligations (CDO) and their variations are examples of credit basket products.
- In the context of basket credit default products the random factors X_i are the time of default τ_i of the i-th reference entity in a basket of N names and the payoff is of the form:

$$P = P(\tau_1, \ldots, \tau_N)$$

Example: n-th to default Basket Default Swap

- In a n-th to default Basket Default Swap one party (protection buyer) makes regular payments to a counterparty (protection seller) at time $T_1, \ldots, T_M \leq T$ provided that less than n defaults events among the components of the basket are observed before time T_M.
- If n defaults occur before time T, the regular payments cease and the protection seller makes a payment to the buyer of $(1 - R_i)$ per unit notional, where R_i is the normalized recovery rate of the i-th asset.
- The value at time zero of the Basket Default Swap on a given realization of the default times $\tau_1, \ldots, \tau_N$, i.e., the Payout function, can be expressed as

$$P(\tau_1, \ldots, \tau_N) = P_{prot}(\tau_1, \ldots, \tau_N) - P_{prem}(\tau_1, \ldots, \tau_N)$$

i.e., as the difference between the so-called *protection* and *premium* legs.

Example: n-th to default Basket Default Swap

- The value leg is given by

$$P_{prot}(\tau_1, \ldots, \tau_N) = (1 - R_n) D(\tau) \mathbb{I}(\tau \leq T),$$

where R_n and τ are the recovery rate and default time of the n-th to default, respectively, $D(t)$ is the discount factor for the interval $[0, t]$ (here we assume for simplicity uncorrelated default times and interest rates), and $\mathbb{I}(\tau \leq T)$ is the indicator function of the event that the n-th default occurs before T.

- The premium leg reads instead, neglecting for simplicity any accrued payment,

$$P_{prem}(\tau_1, \ldots, \tau_N) = \sum_{k=1}^{T_M} c_k D(T_k) \mathbb{I}(\tau \geq T_k)$$

where c_k is the premium payment (per unit notional) at time T_k.

Copula Models

- Credit Basket Products are also known as *correlation products* because their value depends not only on the marginal distribution of the default times but also on their correlation structure.
- Such correlation structure is typically captured by means of a copula model. For instance, in a Gaussian copula, the cumulative joint distribution of default times is assumed of the form:

$$\mathbb{P}(\tau_1 \leq t_1, \ldots, \tau_N \leq t_N) = \Phi_N(\Phi^{-1}(F_1(t_1)), \ldots, \Phi^{-1}(F_N(t_N)); \rho)$$

where $\Phi_N(Z_1, \ldots, Z_N; \rho)$ is a N-dimensional multivariate Gaussian distribution with zero mean, and a $N \times N$ positive semidefinite correlation matrix ρ; Φ^{-1} is the inverse of the standard normal cumulative distribution, and $F_i(t) = \mathbb{P}(\tau_i \leq t)$, $i = 1, \ldots, N$, are the marginal distributions of the default times of each reference entity, depending on a set of model parameters θ.

Hazard Rate Model

- The key concept for the valuation of credit derivatives, in the context of the models generally used in practice, is the *hazard rate*, λ_u, representing the probability intensity of default of the reference entity between times u and $u + du$, conditional on survival up to time u. The hazard rate function λ_u is commonly parameterized as piece-wise constant with M knot points at time $(t_1, \ldots, t_M)$, $\lambda = (\lambda_1, \ldots, \lambda_M)$, .
- By modelling the default event of a reference entity i as the first arrival time of a Poisson process with intensity λ_u^i, the survival probability, $\mathbb{P}(\tau_i > t)$, is given by

$$\mathbb{P}(\tau_i > t) = \exp\left[-\int_0^t du\ \lambda_u^i\right],$$

so that the marginal cumulative distribution of default times reads

$$F_i(t; \lambda^i) = \mathbb{P}(\tau \leq t) = 1 - \exp\left[-\int_0^t du\ \lambda_u^i\right],$$

Forward Simulation Algorithm

The simulation of a Gaussian Copula model can be seen as a single time-step instance of the general approach, consisting of the following steps:

Step 0 Perform a Cholesky factorization of the matrix ρ, say $L = \texttt{CHOLESKY}(\rho)$.

For each MC replication:

Step 1 Generate an N dimensional vector of uncorrelated normal Gaussian variates Z'.

Step 2 Correlate the random variates: $Z = \texttt{CORRELATE}(Z', L)$, where as previously discussed the correlation step consist of a single matrix vector multiplication $Z = LZ'$.

Step 3 Perform the 'propagation step' $\tau = \texttt{PROP}_0[Z, \theta]$.

Step 4 Evaluate the payout function: $P = P(\tau)$.

Forward Simulation Algorithm

- From the form of the cumulative joint distribution of default times

$$\mathbb{P}(\tau_1 \leq t_1, \ldots, \tau_N \leq t_N) = \Phi_N(\Phi^{-1}(F_1(t_1; \lambda^1)), \ldots, \Phi^{-1}(F_N(t_N, \lambda^N)); \rho)$$

  it follows that the random variates $\Phi^{-1}(F_1(\tau_i, \lambda^i))$ are distributed according to a multivariate normal distribution.
- Hence the propagation step $\tau = \texttt{PROP}_0[Z, \theta]$ consists in turn of the following sub-steps:

  Step 3a Set $U_i = \Phi(Z_i)$, $i = 1, \ldots, N$.
  Step 3b Set $\tau_i = F_i^{-1}(U_i; \lambda_i)$, $i = 1, \ldots, N$.

  where $F_i^{-1}(U_i; \lambda_i)$ is the root τ_i of the equation

$$\exp\left[-\int_0^{\tau_i} du\ \lambda_u^i\right] = 1 - U_i.$$

Adjoint Simulation Algorithm

- The corresponding adjoint algorithm consists of the following steps:

  Step $\bar{4}$ Evaluate the adjoint Payout $\bar{\tau}_i = \partial P / \partial \tau_i$, for $i = 1, \ldots, N$.

  Step $\bar{3}$ Evaluate the adjoint of the propagation step:

$$(\bar{\lambda}, \bar{Z}) = \texttt{PROP\_b}_0[Z, \theta, \bar{\tau}].$$

  Step $\bar{2}$ Calculate the adjoint of the correlation step:

$$\bar{L} = \texttt{CORRELATE\_b}(Z', \bar{Z}),$$

  implemented as

$$\bar{L} = \bar{Z} Z'^t.$$

Adjoint Simulation Algorithm

- In turn, the adjoint of the propagation step reads:

  Step $3\bar{b}$ Calculate:

$$\bar{U}_i = \bar{\tau}_i \frac{\partial F_i^{-1}(U_i;\lambda^i)}{\partial U_i} = \bar{\tau}_i \frac{1}{f_i(F_i^{-1}(U_i;\lambda^i);\lambda)},$$

$$\bar{\lambda}_j^i = \bar{\tau}_i \frac{\partial F_i^{-1}(U_i;\lambda^i)}{\partial \lambda_j^i},$$

  for $i = 1,\ldots,N$ and $j = 1,\ldots,M$.

  Step $3\bar{a}$ Calculate: $\bar{Z}_i = \bar{U}_i \phi(Z_i)$, $i = 1,\ldots,N$.

where $f_i(t;\lambda) = \partial F(t;\lambda)/\partial t$ is the p.d.f. of the default time of the i-th reference entity and $\phi(x)$ is the standard normal p.d.f. Note that computing the derivative $\partial F_i^{-1}(U_i;\lambda^i)/\partial \lambda_j^i$ involves differentiating the root searching algorithm used to determine the default time τ_i. However, a much better implementation is possible by means of the so-called implicit function theorem [13] (discussed later on).

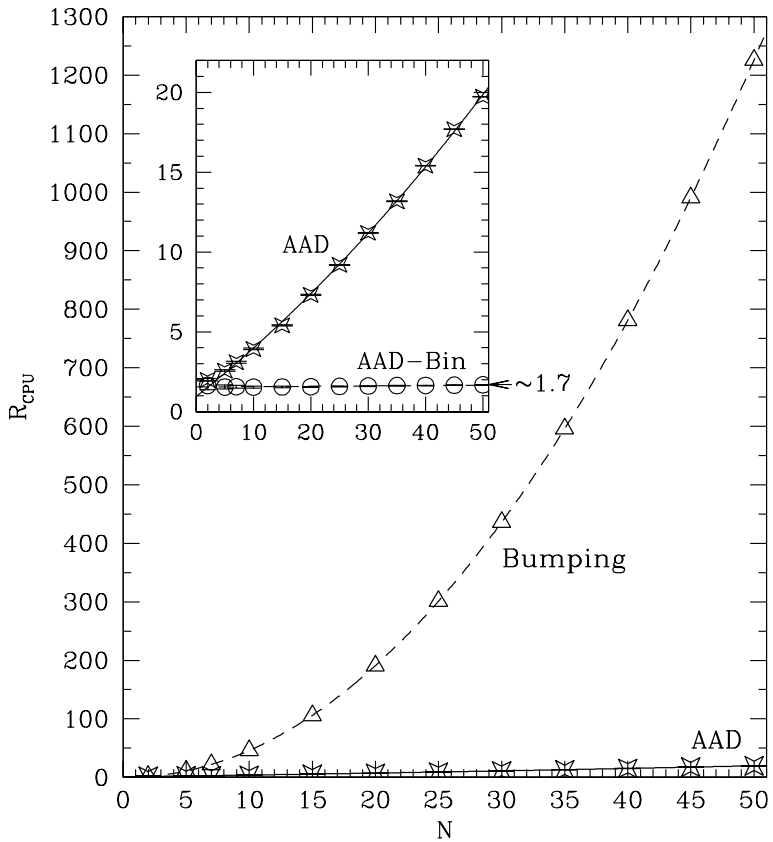
Payout Smoothing

- In order to apply the Pathwise Derivative method to the payout above, the indicator functions in the premium and protection legs

$$P_{prem}(\tau_1, \ldots, \tau_N) = \sum_{k=1}^{T_M} c_k D(T_k) \mathbb{I}(\tau \geq T_k),$$
$$P_{prot}(\tau_1, \ldots, \tau_N) = (1 - R_n) D(\tau) \mathbb{I}(\tau \leq T),$$

  need to be regularized.
- As seen before, one simple and practical way of doing that is to replace the indicator functions with their smoothed counterpart, at the price of introducing a small amount of bias in the Greek estimators.
- For the problem at hand, as it is also generally the case, such bias can be easily reduced to be smaller than the statistical errors that can be obtained for any realistic number of MC iteration N_{MC}.

Results



Ratios of the CPU time required for the calculation of the option value, and correlation Greeks, and the CPU time spent for the computation of the value alone, as functions of the number of names in the basket. Symbols: Bumping (one-sided finite differences) (triangles), AAD without binning (i.e. $N_{\mathrm{B}} = N_{\mathrm{MC}}$) (stars), AAD with binning ($N_{\mathrm{B}} = 20$) (empty circles) [11].

Results

- As expected, for standard finite-difference estimators, such ratio increases quadratically with the number of names in the basket. Already for medium sized basket ($N \simeq 20$) the cost associated with Bumping is over 100 times more expensive than the one of AAD.
- Nevertheless, at a closer look (see the inset) the relative cost of AAD without binning is $O(N)$, because of the contribution of the adjoint of the Cholsesky decomposition.
- However, when using $N_{\mathrm{B}} = 20$ bins the cost of the adjoint Cholesky computation is negligible and the numerical results show that all the Correlation Greeks can be obtained with a mere 70% overhead compared to the calculation of the value of the option.
- This results in over 2 orders of magnitude savings in computational time for a basket of over 40 Names.

Section 10

Case Study: Real Time Counterparty Credit Risk Management

Counterparty Credit Risk Problem

- Credit Valuation Adjustment (CVA):

$$V_{\mathrm{CVA}} = \mathbb{E}\Big[\mathbb{I}(\tau_c \leq T) D(\tau_c) \times L_{\mathrm{GD}}(\tau_c)\Big(NPV(\tau_c) - C(R(\tau_c^-))\Big)^+\Big],$$

  where τ_c is the default time of the counterparty, $NPV(t)$ is the net present value of the portfolio at time t, $C(R(t))$ is the collateral outstanding, typically dependent on the rating R of the counterparty, $L_{\mathrm{GD}}(t)$ is the loss given default, $D(t)$ is the discount factor, and T is the longest deal maturity in the portfolio.
- Here for simplicity we consider the unilateral CVA, the generalization to bilateral CVA and Debt Valuation Adjustment (DVA) or Funding Valuation Adjustment (FVA) is straightforward.

Counterparty Credit Risk Problem

- The expectation above is typically computed on a discrete time grid of 'horizon dates' $T_0 < T_1 < \ldots < T_{N_O}$ as, for instance,

$$V_{\text{CVA}} \simeq \sum_{i=1}^{N_O} \mathbb{E}\Big[\mathbb{I}(T_{i-1} < \tau_c \leq T_i) D(T_i) \times L_{\text{GD}}(T_i)\Big(NPV(T_i) - C\left(R(T_i^-)\right)\Big)^+\Big].$$

- Risk managing CVA/DVA is challenging because all the trades facing the same counterparty must be valued at the same time, typically with Monte Carlo.

A New Challenge: Rating Dependent Payoffs

- We are dealing with expectation values of the form

$$V = \mathbb{E}_{\mathbb{Q}}\Big[P(R, X)\Big] \,,$$

with 'payout' given by

$$P = \sum_{i=1}^{N_O} P\left(T_i, R(T_i), X(T_i)\right),$$

where

$$P\left(T_i, R(T_i), X(T_i)\right) = \sum_{r=0}^{N_R} \tilde{P}_i(X(T_i); r)\, \delta_{r,R(T_i)}.$$

- The Rating variable is discrete so the Payoff is not Lipschitz-continuous.

Rating Transition

- We consider the rating transition Markov chain model of Jarrow, Lando and Turnbull [12]:

$$R(T_i) = \sum_{r=1}^{N_R} \mathbb{I}\left(\tilde{Z}_i^R > Q(T_i, r)\right),$$

  where $\tilde{Z}_i^R$ is a standard normal variate, and $Q(T_i, r)$ is the quantile-threshold corresponding to the transition probability from today's rating to a rating r at time T_i.
- Note that the discussion below is not limited to this particular model, and it could be applied with minor modifications to other commonly used models describing the default time of the counterparty, and its rating.

Singular Pathwise Derivative Estimator

- Due to the discreteness of the state space of the rating factor, the pathwise estimator for its related sensitivities is not well defined.
- This can be easily seen by expressing the Payoff as

$$P\Big(T_i,\tilde{Z}_i^R,X(T_i)\Big) = \tilde{P}_i(X(T_i);0) + \sum_{r=1}^{N_R}\Big(\tilde{P}_i(X(T_i);r) - \tilde{P}_i(X(T_i);r-1)\Big)\mathbb{I}\Big(\tilde{Z}_i^R > Q(T_i,r;\theta)\Big),$$

so that the singular contribution reads

$$\partial_{\theta_k} P\big(T_i,\tilde{Z}_i,X(T_i)\big) = -\sum_{r=1}^{N_R}\Big(\tilde{P}_i(X(T_i);r) - \tilde{P}_i(X(T_i);r-1)\Big) \times \delta\Big(\tilde{Z}_i^R = Q(T_i,r;\theta)\Big)\,\partial_{\theta_k} Q(T_i,r;\theta).$$

- This cannot be sampled with Monte Carlo.

Singular Pathwise Derivative Estimator

- The singular contribution can be integrated out using the properties of Dirac's delta, giving after straightforward computations,

$$\bar{\theta}_k = -\sum_{r=1}^{N_R} \frac{\phi(Z^\star, Z_i^X, \rho_i)}{\sqrt{i}\,\phi(Z_i^X, \rho_i^X)} \partial_{\theta_k} Q(T_i, r; \theta) \times \left(\tilde{P}_i(X(T_i); r) - \tilde{P}_i(X(T_i); r-1)\right),$$

  where $Z^\star$ is such that $(Z^\star + \sum_{j=1}^{i-1} Z_j^R)/\sqrt{i} = Q(T_i, r; \theta)$, and $\phi(Z_i^X, \rho_i^X)$ is a N_X-dimensional standard normal probability density function with correlation matrix ρ_i^X obtained by removing the first row and column of ρ_i; here $\partial_{\theta_k} Q(T_i, r; \theta)$ is not stochastic, and can be evaluated (e.g., using AAD) once per simulation.
- The final result is rather intuitive as it is given by the probability weighted sum of the discontinuities in the payout.

Test Application: CVA of a portfolio of swaps on commodity Futures

- We consider a simple one factor lognormal model for the Futures curve of the form

$$\frac{dF_T(t)}{F_T(t)} = \sigma_T \exp(-\beta(T-t))d\,W_t,$$

where W_t is a standard Brownian motion; $F_T(t)$ is the price at time t of a Futures contract expiring at T; σ_T and β define a simple instantaneous volatility function that increases approaching the contract expiry, as empirically observed for many commodities.

- As underlying portfolio, we consider a set of commodity swaps, paying on a strip of Futures (e.g., monthly) expiries t_j, $j = 1, \ldots, N_e$ the amount $F_{t_j}(t_j) - K$. The net present value for this portfolio reads

$$NPV(t) = \sum_{j=1}^{N_e} D(t, t_j)\Big(F_{t_j}(t) - K\Big).$$

Forward and Backward Propagation

- The propagation (PROP) step for the Futures price reads:

$$F_T(T_i) = F_T(T_{i-1}) \exp\left(\sigma_i \sqrt{\Delta T_i} Z - \frac{1}{2}\sigma_i^2 \Delta T_i\right),$$

where $\Delta T_i = T_i - T_{i-1}$, and

$$\sigma_i^2 = \frac{\sigma_T^2}{2\beta \Delta T_i} e^{-2\beta T}\left(e^{2\beta T_i} - e^{2\beta T_{i-1}}\right).$$

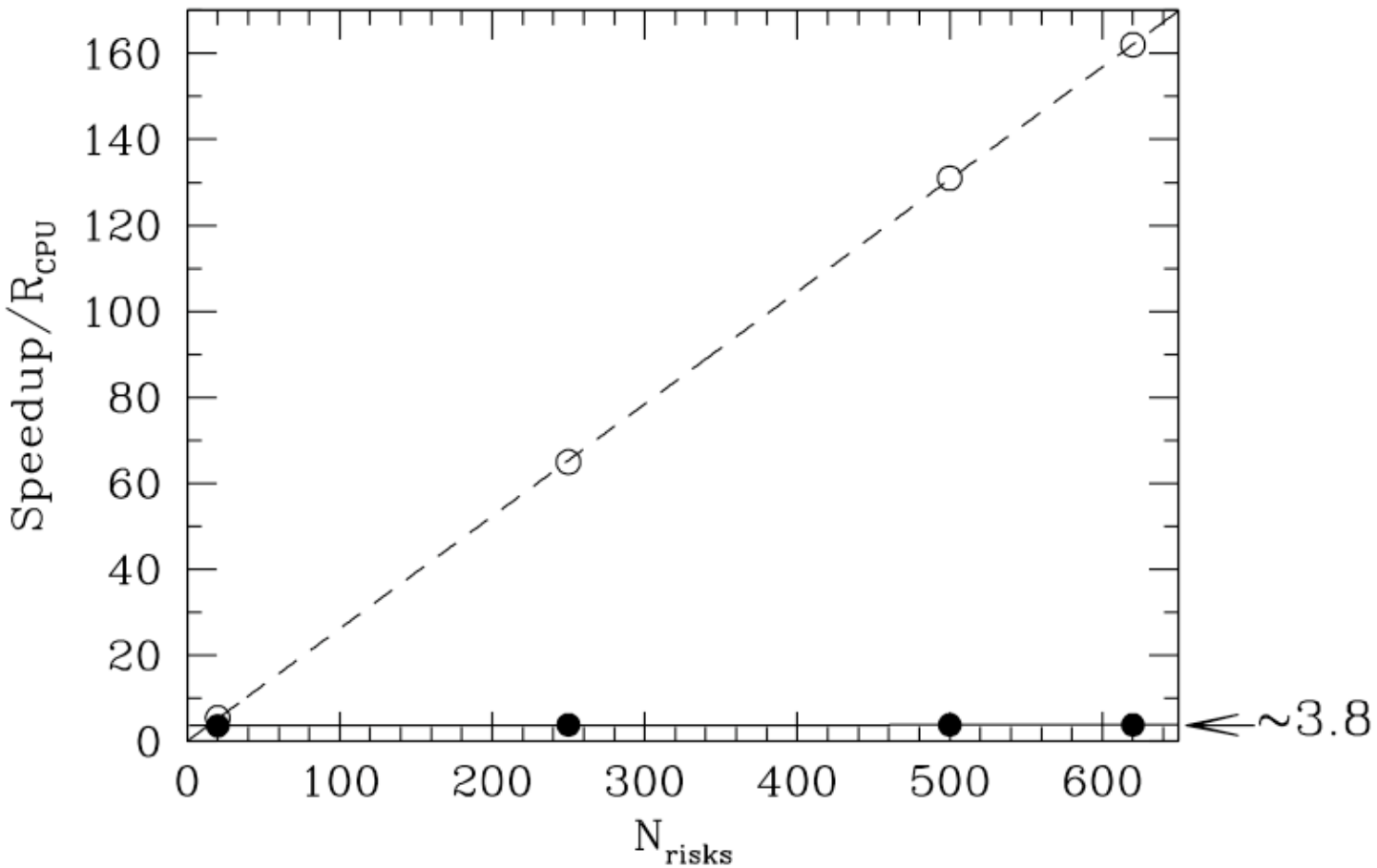
- The associated adjoint (PROP_b) reads:

$$\bar{F}_T(T_i - 1) \mathrel{+}= \bar{F}_T(T_i) \exp\left(\sigma_i \sqrt{\Delta T_i} Z - \frac{1}{2}\sigma_i^2 \Delta T_i\right),$$
$$\bar{\sigma}_i = \bar{F}_T(T_i) F(T_i)(\sqrt{\Delta T_i}\, Z - \sigma_i \Delta T_i)$$

with

$$\bar{\sigma}_T \mathrel{+}= \frac{\bar{\sigma}_i}{\sqrt{2\beta \Delta T_i}} \sqrt{e^{-2\beta T}\left(e^{2\beta T_i} - e^{2\beta T_{i-1}}\right)}.$$

Results



Portfolio of 5 commodity swaps over a 5 years horizon. Bumping (empty dots), AAD (full dots).

Total time: 1h 40 min (Bumping); 10 sec (AAD). From Ref.[12].

Variance Redution

- Because of the analytic integration of the singularities, the AAD risk is typically less noisy than the one produced by Bumping.

δ	VR[Q(1,1)]	VR[Q(1,2)]	VR[Q(1,3)]
0.1	24	16	12
0.01	245	165	125
0.001	2490	1640	1350

Table: Variance reduction (VR) on the sensitivities with respect to the thresholds $Q(1, r)$ ($N_R = 3$). δ indicates the perturbation used in the finite-differences estimators of the sensitivities.

- The variance reduction can be thought of as a further speedup factor because it corresponds to the reduction in the computation time for a given statistical uncertainty on the sensitivities. This diverges as the perturbation δ tends to zero, and may be very significant even for a fairly large value of δ.

Section 11

Application to Partial Differential Equations for Short Rate Models

Beyond Monte Carlo Applications: Partial Differential Equations

- Option pricing problems can be often formulated in terms of the solution of a parabolic (backward) PDE of the form

$$\frac{\partial V}{\partial t} + \mu(x,t;\theta)\frac{\partial V}{\partial x} + \frac{1}{2}\sigma^2(x,t;\theta)\frac{\partial^2 V}{\partial x^2} - \nu(x,t;\theta)\,V = 0,$$

where

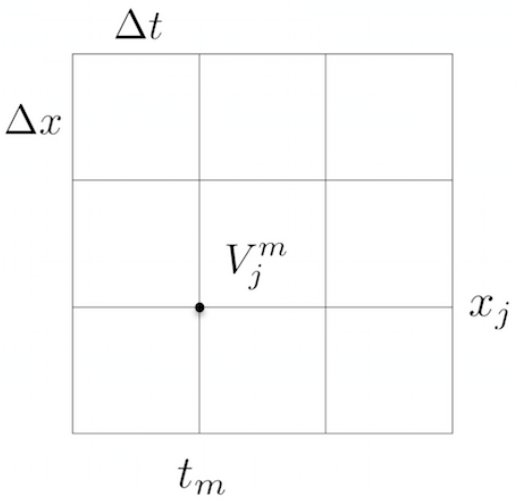
$$V = V(x_t,t;\theta) \equiv \mathbb{E}\left[e^{-\int_t^T \nu(x_u,u;\theta)du} P(x_T;\theta)\right],$$

is the value of a derivative contract at time t, with payoff at expiry $P(x_T;\theta)$. Here the risk factor x_t follows a diffusion of the form

$$dx_t = \mu(x_t,t;\theta)dt + \sigma(x_t,t;\theta)dW_t.$$

- As before, $\theta = (\theta_1,\ldots,\theta_{N_\theta})$ represents the vector of N_θ model parameters the model is dependent on.

Numerical Solution by Finite-Difference Discretization



The solution $V_0(\theta) = V(x_{t_0}, t_0; \theta)$ of the backward PDE can be found numerically by discretization on the rectangular domain $(t, x) \in [t_0, T] \times [x_{min}, x_{max}]$:

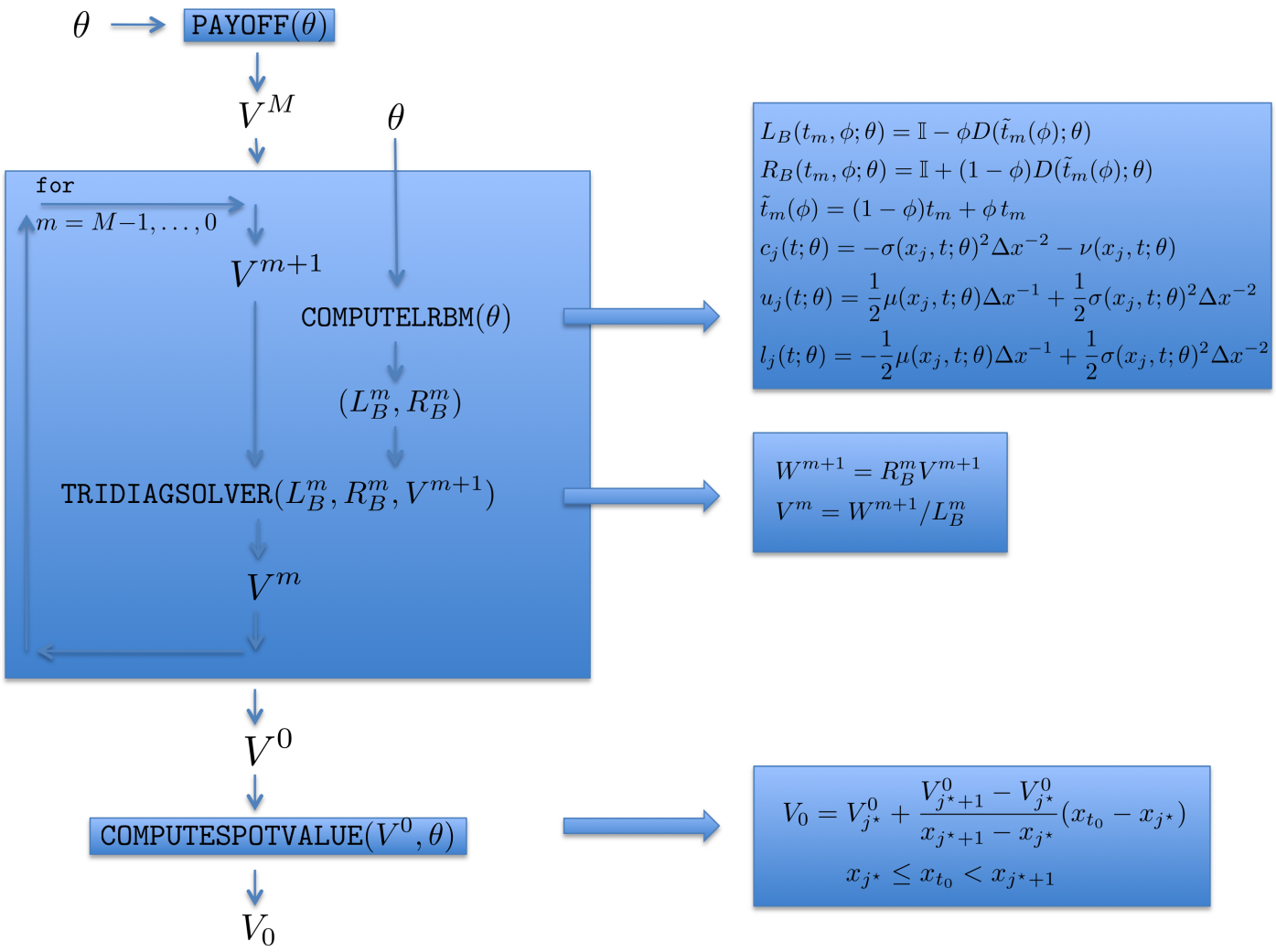
$$V^m(\theta) = (V(x_1, t_m; \theta), \ldots, V(x_N, t_m; \theta))^t.$$

- Given the value of the option at expiry, $V_j^M(\theta) = P(x_j; \theta)$, the value of the option at time t_0 can be found by iterating for $m = M - 1, \ldots, 0$ the matrix recursion

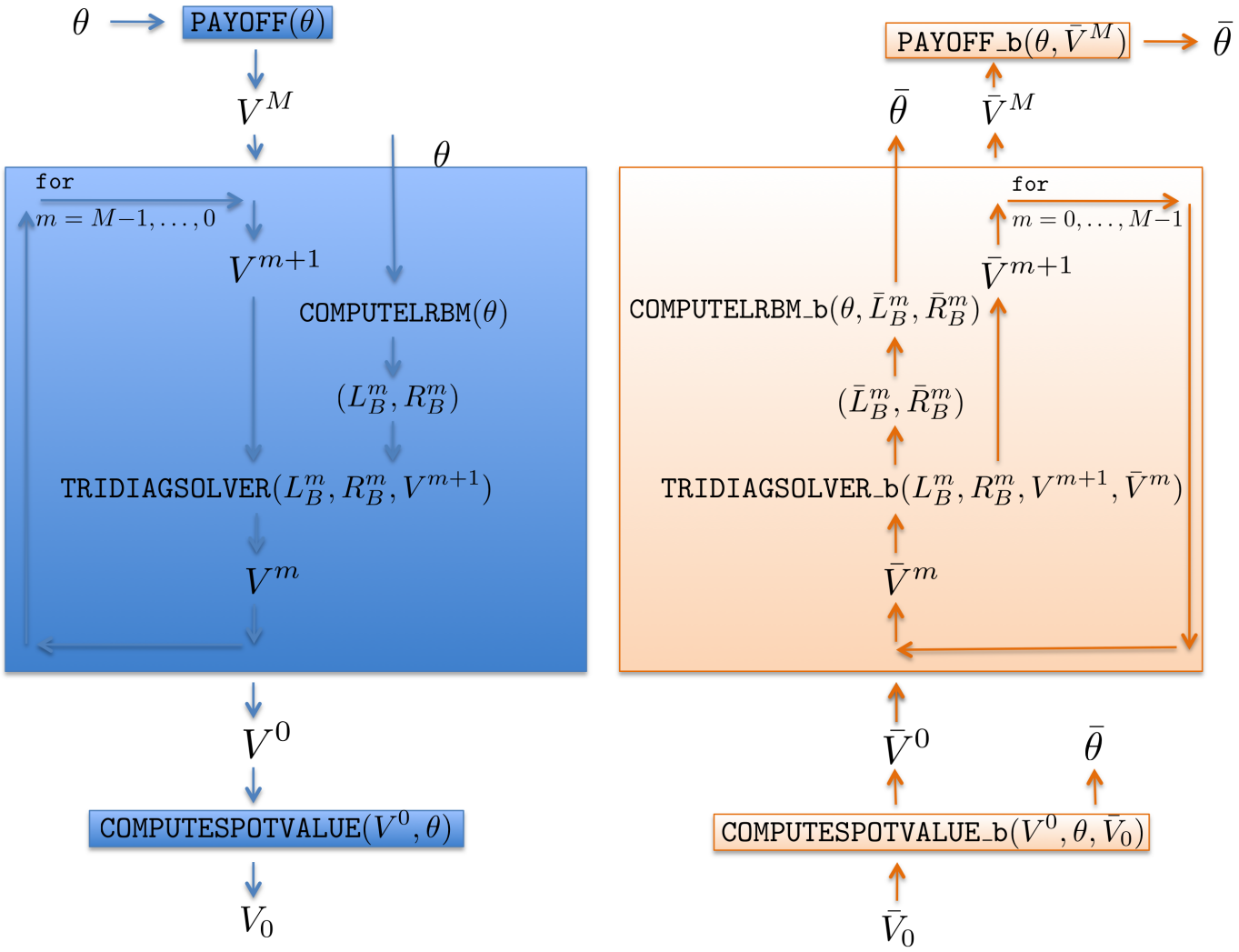
$$L_B(t_m, \phi; \theta) V^m(\theta) = R_B(t_m, \phi; \theta) V^{m+1}(\theta)$$

obtained with finite-difference approximations of the first and second derivatives in the backward PDE.

Blueprint of a Backward PDE Solver



Blueprint of an AAD Backward PDE Solver



- Using the AAD rules it is easy to pin down the structure of the Adjoint PDE solver.

The heart of the algorithm: the tridiagonal solver

$V^m = \texttt{TRIDIAGSOLVER}(L_B^m, R_B^m, V^{m+1})$

$$W^{m+1} = R_B^m V^{m+1}$$
$$V^m = W^{m+1} / L_B^m$$

$$y = Bx \quad \begin{cases} \bar{x} = A^t \bar{y} \\ \bar{A} = \bar{y} x^t \end{cases}$$
$$\bar{A} = -A^{-t} \overline{A^{-1}} A^{-t}$$

$\bar{V}^{m+1} = \texttt{TRIDIAGSOLVER\_b}(L_B^m, R_B^m, V^{m+1}, \bar{V}^m)$

$$\bar{W}^{m+1} = [L_B^m]^{-t} \bar{V}^m$$
$$\overline{[L_B^m]^{-1}} = \bar{V}^m [W^{m+1}]^t$$
$$\bar{L}_B^m = -[L_B^m]^{-t} \overline{[L_B^m]^{-1}} [L^m]_B^{-t}$$
$$\bar{R}_B^m = \bar{W}^{m+1} [V^{m+1}]^t$$
$$\bar{V}^{m+1} = [R_B^m]^t \bar{W}^{m+1}$$

- A collection of results on the Adjoint of linear algebra operations (see M. Giles' 'Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation') is very useful when dealing with code implementing linear algebra.
- The computational cost of the naïve AAD algorithm above is $O(N^3)$. In order to reduce the computational cost to $O(N)$, as in the original algorithm, we need to avoid the matrix inversion by using all the information that is available to us (including the forward sweep!).

The heart of the algorithm: the tridiagonal solver (cont'd)

$V^m = \texttt{TRIDIAGSOLVER}(L_B^m, R_B^m, V^{m+1})$

$$W^{m+1} = R_B^m V^{m+1}$$
$$V^m = W^{m+1} / L_B^m$$

$$y = Bx \quad \begin{cases} \bar{x} = A^t \bar{y} \\ \bar{A} = \bar{y} x^t \end{cases}$$
$$\bar{A} = -A^{-t} \overline{A^{-1}} A^{-t}$$

$\bar{V}^{m+1} = \texttt{TRIDIAGSOLVER\_b}(L_B^m, R_B^m, V^{m+1}, \bar{V}^m)$

$$\bar{W}^{m+1} = [L_B^m]^{-t} \bar{V}^m = \bar{V}^m / [L_B^m]^t$$
$$\cancel{\overline{[L_B^m]^{-1}} = \bar{V}^m [W^{m+1}]^t}$$
$$\bar{L}_B^m = -[L_B^m]^{-t} \overline{[L_B^m]^{-1}} [L_B^m]^{-t}$$
$$= -[L_B^m]^{-t} \bar{V}^m [W^{m+1}]^t [L_B^m]^{-t}$$
$$= -\bar{W}^{m+1} [[L_B^m]^{-1} W^{m+1}]^t$$
$$= -\bar{W}^{m+1} [V^m]^t$$
$$\bar{R}_B^m = \bar{W}^{m+1} [V^{m+1}]^t$$
$$\bar{V}^{m+1} = [R_B^m]^t \bar{W}^{m+1}$$

- Only the elements on the three main diagonals of $\bar{L}_B^m$ and $\bar{R}_B^m$ contribute to the sensitivities, so that only $3N$ multiplications are required for their computation.
- The overall computational cost of the adjoint tridiagonal solver is $O(N)$, exactly as for the forward counterpart and as expected from the general result on the computational efficiency of AAD.

Some Results: the Black-Karasinski model for default intensities

- To illustrate the efficiency of the AAD-PDE approach the Black-Karasinki (BK) model for the stochastic instantaneous hazard rate $h_t = \exp x_t$, namely

$$d \log h_t = \kappa(t)(\mu(t) - \log h_t)dt + \sigma(t)dW_t.$$

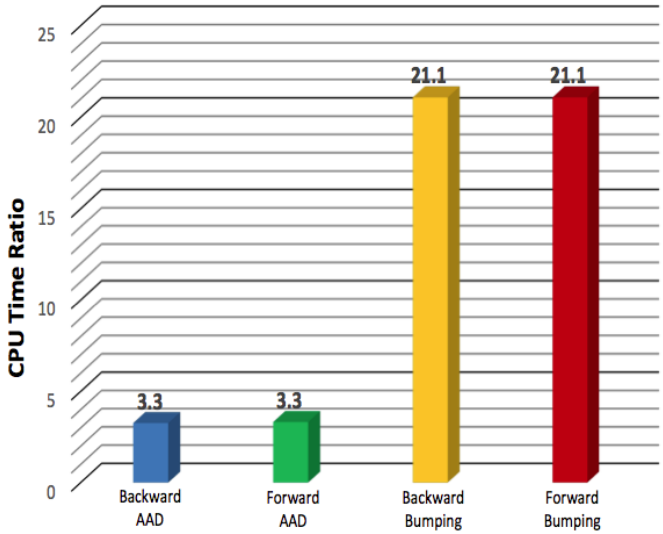
  Here, we will fix the mean reversion rate $\kappa = 0.01$ and assume $\mu(t)$ and $\sigma(t)$ to be left-continuous piecewise constant functions.
- The conditional probability of the obligor surviving up to time T is given by

$$Q(h_t, t, T) = \mathbb{E}\Big[\exp\Big[-\int_t^T du\ h_u\Big]\Big|h_t, \tau > t\Big].$$

- Any credit derivative whose payoff at time T is a function of the hazard rate h_T, such as defaultable bonds, CDS, bond options and CDS options can be valued within the PDE approach.

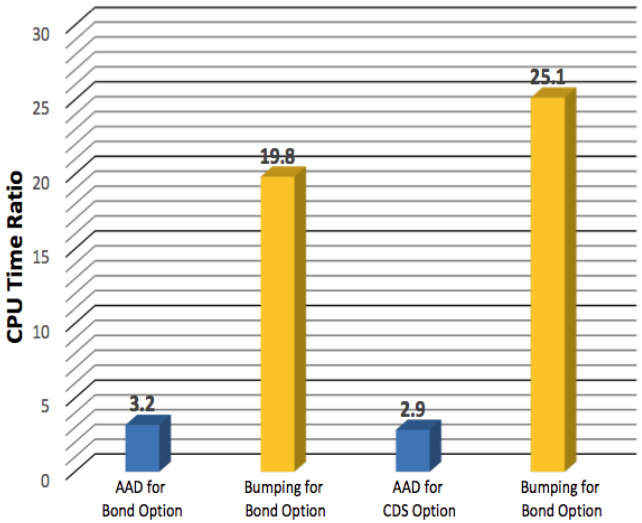
Warming up: Defaultable Zero Bond

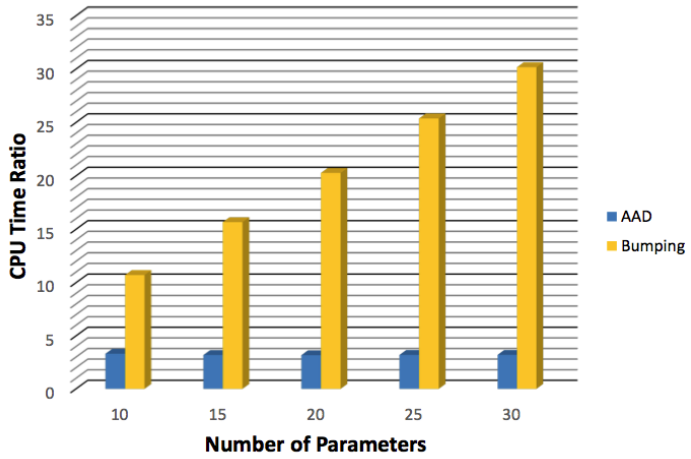
	Bwd PDE (AAD)	Fwd PDE (AAD)	Bwd PDE (FD)	Fwd PDE (FD)
μ_1	-1.8e-4	-1.8e-4	-1.8e-4	-1.8e-4
μ_2	-1.7e-4	-1.6e-4	-1.7e-4	-1.6e-4
μ_3	-1.6e-4	-1.6e-4	-1.6e-4	-1.6e-4
μ_4	-1.5e-4	-1.5e-4	-1.5e-4	-1.5e-4
μ_5	-2.7e-4	-2.7e-4	-2.7e-4	-2.7e-4
μ_6	-2.3e-4	-2.3e-4	-2.3e-4	-2.3e-4
μ_7	-2.0e-4	-2.0e-4	-2.0e-4	-2.0e-4
μ_8	-1.6e-4	-1.6e-4	-1.6e-4	-1.6e-4
μ_9	-1.3e-4	-1.3e-4	-1.3e-4	-1.3e-4
μ_{10}	-9.1e-5	-9.1e-5	-9.2e-5	-9.1e-5
μ_{11}	-5.6e-5	-5.5e-5	-5.6e-5	-5.5e-5
μ_{12}	-1.8e-5	-1.8e-5	-1.9e-5	-1.8e-5
σ_1	-7.8e-3	-7.8e-3	-7.5e-3	-7.8e-3
σ_2	-0.011	-0.01	-0.011	-0.01
σ_3	-0.016	-0.016	-0.017	-0.016
σ_4	-0.015	-0.015	-0.015	-0.015
σ_5	-0.013	-0.013	-0.013	-0.013
σ_6	-0.012	-0.011	-0.012	-0.011
σ_7	-9.9e-3	-9.9e-3	-0.01	-9.9e-3
σ_8	-0.019	-0.019	-0.02	-0.019



- As expected, the results obtained with both the AAD version of the backward and forward PDE are consistent with the ones obtained by bumping.
- For both the AAD version of the backward and the forward PDE scheme the calculation of the sensitivities can be performed in about 3.3 times the cost of computing the value of the option, i.e., well within the theoretical bound of 4.

Bond and CDS Options



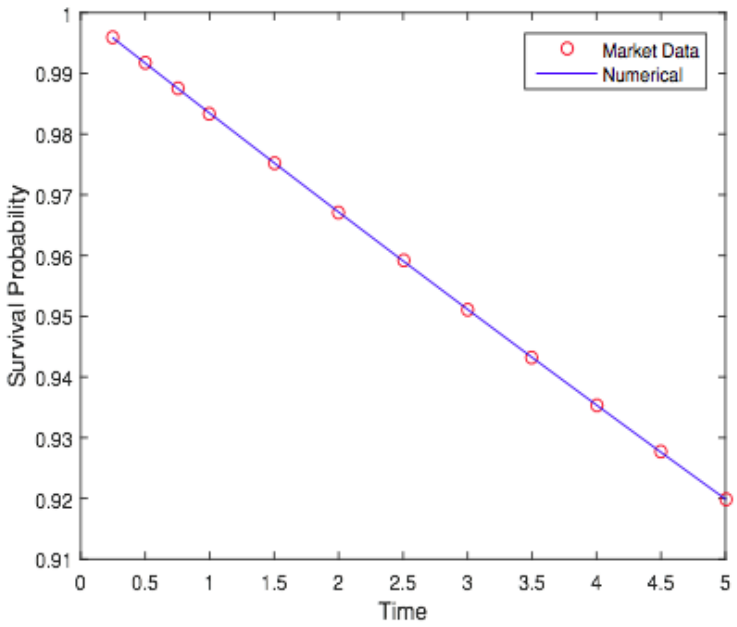


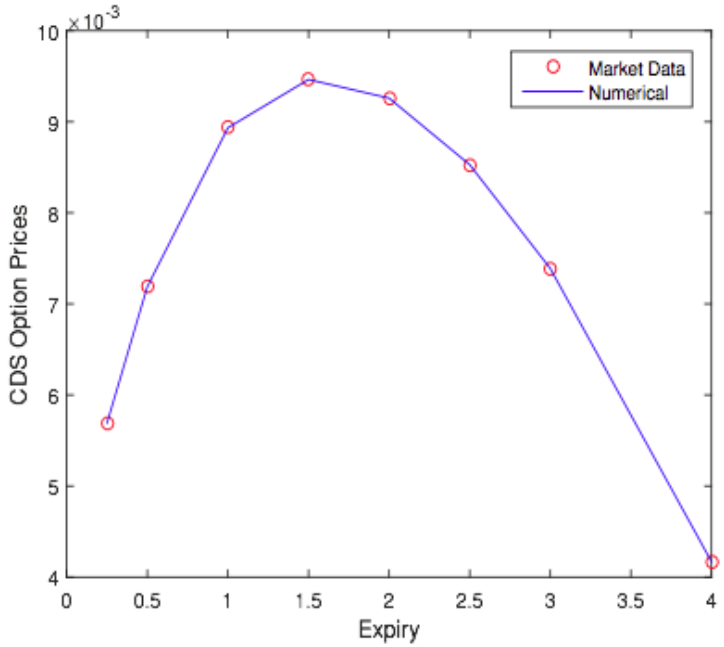
- The computational cost of the AAD algorithm is well within the theoretical bound of 4.
- The overall cost of computing all the sensitivities by means of AAD, relative to the cost of a single valuation of the option, is independent on the number of sensitivities.

Calibration: Model Parameters and Market Parameters

- The sensitivities with respect to the internal model parameters θ are generally of limited utility because they do not correspond directly to financially meaningful quantities.
- The sensitivities we need for hedging are the sensitivities with respect to the market observables M that have been used to calibrate the model.
- It is useful to think in terms of two distinct steps:

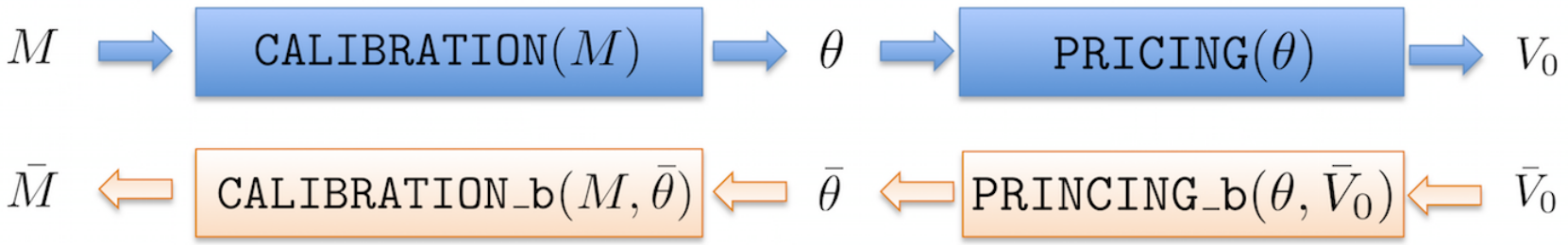
Calibration





- As it is customary, we have used a combination of the forward and backward PDE algorithm to bootstrap the survival probabilities and calibrate to the CDS option prices.

Getting the Market Parameters Sensitivities



$$\bar{M}_m = \sum_{k=1}^{N_\theta} \bar{\theta}_k \frac{\partial \theta_k}{\partial M_m} \qquad \bar{\theta}_k = \bar{V}_0 \frac{\partial V_0}{\partial \theta_k}$$

- The adjoint of the calibration step $M \to \theta(M)$ can be produced following the general rules of AAD.
- The computational cost can be expected to be of the order of the cost of performing the calibration algorithm a few (less than 4) times.
- This in itself is generally much better than bumping, involving repeating the calibration algorithm as many times as sensitivities required.
- However, we can do better thanks to the ...

Implicit Function Theorem

- The calibration algorithm consists of the numerical solution of a system of equations of the form

$$G_i(M, \theta) = 0,$$

with $M \in \mathcal{R}^{N_M}$, $\theta \in \mathcal{R}^{N_\theta}$ and $i = 1, \ldots, N_\theta$, where the function $G_i(M, \theta)$ is of the form

$$G_i(M, \theta) = T_i(M) - V_i(\theta)$$

where $V_i(\theta)$ is the price of the i-th calibration instrument as produced by the model we want to calibrate, and $T_i(M)$ are the prices of the target instruments.

- By differentiating with respect to M

$$\frac{\partial G_i}{\partial M_m} + \sum_{j=1}^{N_\theta} \frac{\partial G_i}{\partial \theta_j} \frac{\partial \theta_j}{\partial M_m} = 0$$

for $m = 1, \ldots, N_M$.

Implicit Function Theorem (cont'd)

- Or equivalently

$$\frac{\partial \theta_k}{\partial M_m} = -\left[\left(\frac{\partial G}{\partial \theta}\right)^{-1} \frac{\partial G}{\partial M}\right]_{km}$$

with $[\partial G/\partial M]_{ij} = \partial G_i/\partial M_j$.

- This relation allows the computation of the sensitivities of $\theta(M)$, locally defined in an implicit fashion by the calibration equation, in terms of the sensitivities of the function G. These can be computed by implementing the corresponding adjoint function

$$(\bar{M}, \bar{\theta}) = \bar{G}(M, \theta, \bar{G})$$

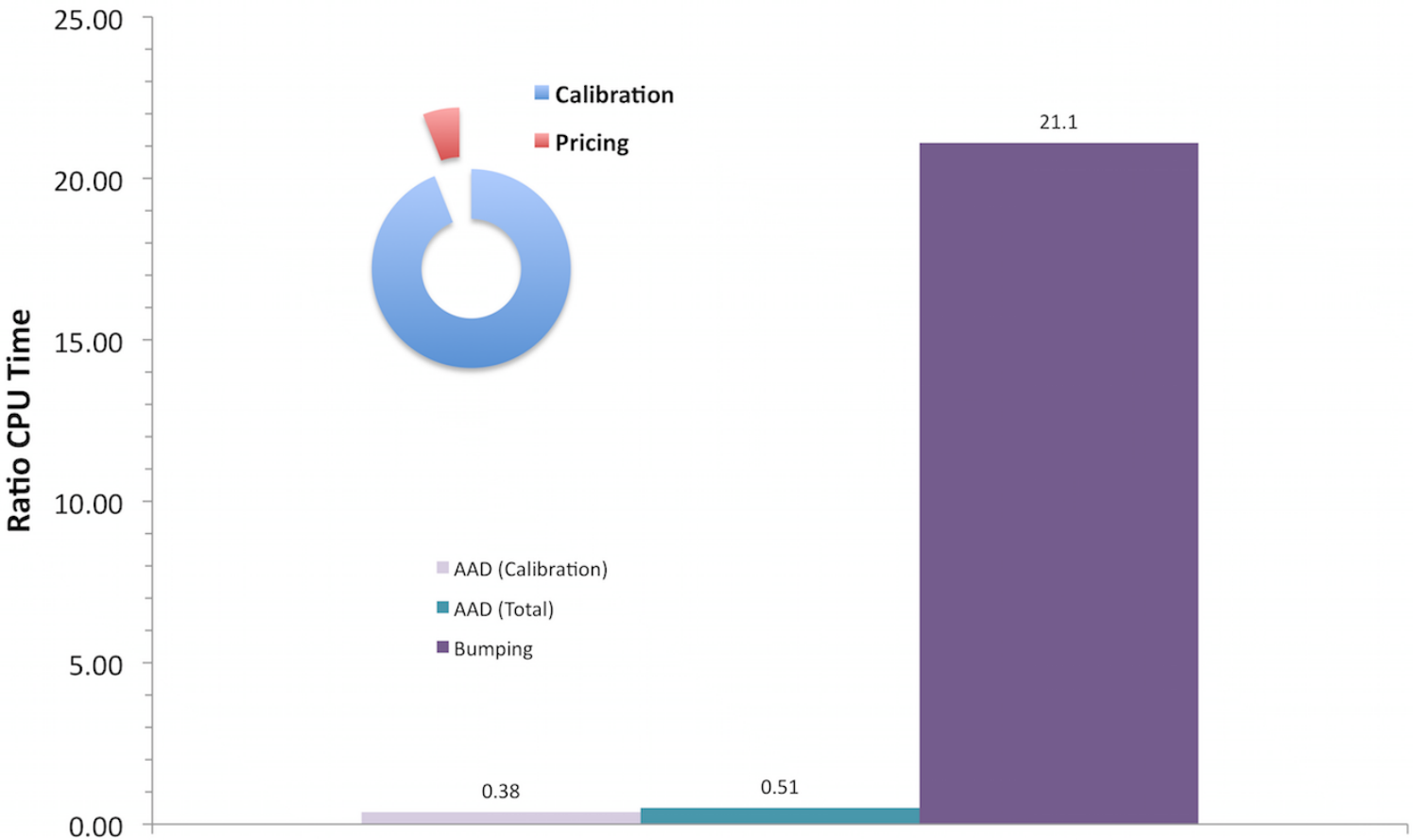
giving according to the general rule

$$\bar{M}_m = \sum_{i=1}^{N_\theta} \bar{G}_i \frac{\partial G_i}{\partial M_m} \quad \bar{\theta}_k = \sum_{i=1}^{N_\theta} \bar{G}_i \frac{\partial G_i}{\partial \theta_k}.$$

Implicit Function Theorem (cont'd)

- The Implicit Function Theorem method is significantly more stable and efficient than calculating the derivatives of the implicit functions $M \to \theta(M)$ by applying AAD to the calibration step.
- This is because $G_i(M, \theta) = T_i(M) - V_i(\theta)$ are *explicit* functions of the model and market parameters that are easy to compute and differentiate, e.g., using the AAD version of the combination of the forward and backward PDE for the calculation of $V_i(\theta)$ and the AAD version of the algorithm for the computation of $T_i(M)$.
- Combining the implicit function theorem with AAD results in extremely efficient risk computations.

AAD and the Implicit Function Theorem: Results



- Combining AAD with the Implicit Function Theorem allows the computation of risk in 50% *less* than the cost of computing the option value, resulting in remarkable savings in computational time.

Section 12

Conclusions

Conclusions

- We have shown how Adjoint Algorithmic Differentiation (AAD) can be used to implement the Adjoint calculation of price sensitivities in a straightforward manner and in complete generality.
- In contrast to algebraic Adjoint methods, the algorithmic approach can be straightforwardly applied to both path dependent options and multi asset Monte Carlo simulations. It also eliminates altogether the need for the sometimes cumbersome analytical work required by algebraic formulations.
- AAD can be applied to any numerical technique. In particular, it can be used to implement efficiently and in full generality the calculation of sensitivities of option prices computed by means of the numerical solution of Partial Differential Equations (PDE).

Conclusions

- By combining the adjoint version of the pricing algorithm, and the Implicit Function Theorem one can avoid the necessity of repeating multiple times the calibration algorithm or implementing the AAD version of the calibration routine.

- This allows the calculation of all price sensitivities for an additional computational cost that is a small multiple of the cost of computing the P&L of the portfolio, thus typically resulting in orders of magnitudes savings in computational time with respect to the standard finite-difference approach.

References I

[1] P. Glasserman, *Monte Carlo Methods in Financial Engineering*, Springer, New York (2004).

[2] Luca Capriotti, Reducing the Variance of Likelihood Ratio Greeks in Monte Carlo, Proceedings of Winter Simulation Conference (2008).

[3] M. Giles and P Glasserman, *Smoking Adjoints: Fast Monte Carlo Greeks*, Risk **19**, 88 (2006).

[4] M. Leclerc, Q.Liang and I. Schneider, *Fast Monte Carlo Bermudan Greeks*, Risk **22**, 84 (2009).

[5] N. Denson and M.S. Joshi, *Fast and Accurate Greeks for the Libor Market Mode*, J. of Computational Finance **14**, 115 (2011).

[6] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Frontiers in Applied Mathematics, Philadelphia, 2000.

[7] U. Naumann, *The Art of Differentiating Computer Programs*, SIAM (2012).

[8] L. Capriotti, *Fast Greeks by Algorithmic Differentiation*, J. of Computational Finance, **14**, 3 (2011).

[9] L. Capriotti and M. Giles, *Algorithmic Differentiation: Adjoint Greeks Made Easy*, Risk **25**, 92 (2012).

References II

[10] S. P. Smith, *Differentiation of Cholesky Algorithm*, J. of Computational and Graphic Statistics, **4**, 134 (1995).

[11] L. Capriotti and M. Giles, *Fast Correlation Greeks by Adjoint Algorithmic Differentiation*, Risk **23**, 79 (2010).

[12] L. Capriotti, Jacky Lee, and Matthew Peacock, *Real Time Counterparty Credit Risk Management in Monte Carlo*, Risk **24**, 86 (2011).

[13] L. Capriotti and J. Lee, *Adjoint Credit Risk Management*, Risk Magazine, **27**, 90 (2014).

[14] M. Henrard, *Calibration and Implicit Function Theorem*, OpenGamma Quantitative Research, 1 (2011).

[15] L. Capriotti, Y. Jiang, A. Macrina, *Real-Time Risk Management: An AAD-PDE Approach*, International Journal of Financial Engineerineg **2**, 1550039 (2015).

[16] L. Capriotti and M. Giles *15 Years of Adjoint Algorithmic Differentiation in Finance*, Quantitative Finance, (2024)

See also:

- My Publications' Page