Efficient Risk Management in Monte Carlo

Module 3: Risk Management by Adjoint Algorithmic Differentiation I

Luca Capriotti

The 7th Summer School in Mathematical Finance
African Institute for Mathematical Sciences, Muizenberg, Cape Town, South Africa,
February 20-22, 2014

Section 1

Outline

Section 2

Pathwise Derivative Method

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

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

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.

Pathwise Derivative Method

- 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 simulation.
- This can be achieved by noticing that, 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]), one can write the sensitivity $\langle \bar{\theta}_k \rangle \equiv dV/d\theta_k$ as

$$\langle \bar{\theta}_k \rangle = \mathbb{E}_{\mathbb{Q}}\Big[\frac{dP_\theta(X)}{d\theta_k} \Big].$$

- In the context of MC simulations, this equation can be easily understood by thinking the random sampling of the state vector X as performed in terms of a mapping of the form, $X = X(\theta; Z)$, where Z is a random vector *independent* of θ. In fact, after this mapping, the expectation value $\mathbb{E}_{\mathbb{Q}}[\ldots]$ can be expressed as an average over the probability distribution of Z, $\mathbb{Q}(Z)$, which is independent of θ.

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

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

- 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*.

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 an N-dimensional vector and $N \times N$ matrix, respectively, 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

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

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 3

Algebraic Adjoint Approaches

‘Algebraic’ Adjoint Methods

- In 2006 Mike Giles and Paul Glasserman published a ground breaking ‘Smoking Adjoints’ in Risk Magazine [2].
- 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.* [3] and extended by Joshi *et al.* [4]).
- 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, $i = 1, \dots, N+1$, 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

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

  $0 \le t \le T_i$, and $i = 1, \dots, N$, 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

- The drift term in the spot measure, as imposed by the no arbitrage conditions, reads

$$\mu_i(L(t)) = \sum_{j=\eta(t)}^{i} \frac{\sigma_i^T \sigma_j \delta L_j(t)}{1 + \delta L_j(t)},$$

  where $\eta(t)$ denotes the index of the bond maturity immediately following time t, with $T_{\eta(t)-1}$.
- It is common in the literature, to keep this example as simple as possible, we take each vector σ_i to be a function of time to maturity

$$\sigma_i(t) = \sigma_{i-\eta(t)+1}(0) = \lambda(i - \eta(t) + 1).$$

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.
- The Euler step is best implemented by first computing

$$S_i(t_n) = \sum_{j=\eta(nh)}^{i} \frac{\sigma_j \delta L_j(t_n)}{1 + \delta L_j(t_n)}, \qquad i = \eta(nh), \ldots, N$$

  so that $\mu_i(n) = \sigma_i^T S_i$ giving a cost of $O(N)$ per time step.

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_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 [3]).

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')

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

or equivalently as

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

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

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

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

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

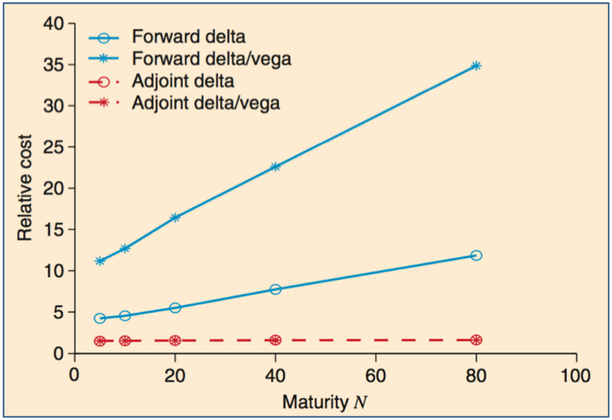
  which consists of $M + 1$ matrix-vector products with an overall computational cost of $O(MN^2)$.

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

Forward vs Adjoint: Computational Complexity

- In the specific example the forward propagation by using the same optimization employed in the propagation of the forward Libor rates can be implemented in $O(N^2)$ per time step (rather than $O(N^3)$).
- However, using the same propagation, the result still holds that the Adjoint propagation is $O(N)$ cheaper, for an overall cost $O(N)$ per time step.
- 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 holds also for Vega.

Algebric Adjoint Methods



From Ref. [2]

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 a programming technique known as Adjoint Algorithmic Differentiation (AAD) [5].
- In general AAD allows the calculation of the gradient of an algorithm at a cost that is a small constant (~ 4) times the cost of evaluating the function itself, independent of the number of input variables.
- 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 4

Adjoint Algorithmic Differentiation (AAD)

Algorithmic Differentiation

- Algorithmic 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 a computer program.
- 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 [5].

Algorithmic Differentiation

- What makes AD particularly attractive when compared to standard (e.g., finite difference) methods for the calculation of the derivatives, 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 properties in different (complementary) computational complexity.

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

$$\bar{X} = \texttt{FUNCTION\_d}(X, \dot{X})$$

(with suffix _d for "dot") calculating the linear combination of the columns of the Jacobian of the function:

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

with $j = 1, \ldots, m$, is bounded by

$$\frac{\mathrm{Cost}[\texttt{FUNCTION\_d}]}{\mathrm{Cost}[\texttt{FUNCTION}]} \leq \omega_T$$

with $\omega_T \in [2, 5/2]$.

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.

Tangent mode: Propagating Forwards

- Imagine that the function $Y = \texttt{FUNCTION}(X)$ is implemented by means of a sequence of steps

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

  where 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.
- Define the Tangent of any intermediate variable U_k as

$$\dot{U}_k = \sum_{i=1}^{n} \dot{X}_i \frac{\partial U_k}{\partial X_i}.$$

Tangent mode: Propagating Forwards

- Using the chain rule we get,

$$\dot{V}_j = \sum_{i=1}^{n} \dot{X}_i \frac{\partial V_j}{\partial X_i} = \sum_{i=1}^{n} \dot{X}_i \sum_k \frac{\partial V_j}{\partial U_k} \frac{\partial U_k}{\partial X_i} = \sum_k \frac{\partial V_j}{\partial U_k} \dot{U}_k ,$$

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

$$\dot{V}_j = \sum_k \dot{U}_k \frac{\partial V_j}{\partial U_k} ,$$

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

Tangent mode: Propagating Forwards

- Hence the computation of the Tangents can be executed in the same direction of the original function

$$\dot{X} \rightarrow \ldots \rightarrow \dot{U} \rightarrow \dot{V} \rightarrow \ldots \rightarrow \dot{Y}.$$

  This can be executed simultaneously with the original function, since at each intermediate step $U \rightarrow V$ one can compute the derivatives $\partial V_j(U)/\partial U_k$ and execute the Tangent forward propagation

$$\dot{U} \rightarrow \dot{V} \qquad \dot{V}_j = \sum_k \dot{U}_k \frac{\partial V_j}{\partial U_k}.$$

- The Tangent of the output obtained with this forward recursion is by definition:

$$\dot{Y}_k = \sum_{i=1}^{n} \dot{X}_i \frac{\partial Y_k}{\partial X_i},$$

  i.e., in a single forward sweep one can produce a linear combination of the columns of the Jacobian $\partial Y/\partial X$.

Tangent mode: Propagating Forwards

- The Tangent mode produces the linear combination of columns of the Jacobian
$$\dot{Y}_k = \sum_{i=1}^{n} \dot{X}_i \frac{\partial Y_k}{\partial X_i},$$
where $\dot{X}$ is an arbitrary vector in $\mathbb{R}^n$.
- By initializing in turn $\dot{X}$ with each vector of the canonical basis in $\mathbb{R}^n$, $(e_1, \ldots, e_n)$ with
$$e_j = (\underbrace{0, \ldots, 1}_{j}, 0, \ldots, 0)$$
one can obtain the partial derivatives of all the outputs with respect to each of the inputs $\dot{Y}_k = \partial Y_k / \partial X_i$, thus resulting in a cost that is n times the cost of a single forward Tangent sweep.

Tangent mode: Propagating Forwards

- It is not difficult to realize that the cost of computing each single step $\dot{U} \to \dot{V}$ is just a small multiple of the cost of executing $U \to V$.
- Consider for instance the example:

$$V_1 = \cos(U_1) U_1 + U_2 \exp(U_2)$$

the corresponding Tangents read

$$\dot{V}_1 = \dot{U}_1(-\sin(U_1) U_1 + cos(U_1)) + \dot{U}_2 (U_2 + 1) \exp(U_2),$$

- Computing $\dot{V}$ (3 intrinsic operations, 4 multiplication and 3 additions) has the same computational complexity of computing the original function (2 intrinsic operations, 2 multiplications and 1 additions). Assuming that all the operations have the same cost it would be twice as expensive.

Tangent mode: Propagating Forwards

- Extending to the whole computation one can see how keeping into account of the relative cost of different types of operation one can arrive to the result [5]:

$$\frac{\mathrm{Cost}[\texttt{FUNCTION\_d}]}{\mathrm{Cost}[\texttt{FUNCTION}]} \leq \omega_T$$

  with $\omega_T \in [2, 5/2]$.
- By performing simultaneously the calculation of all the components of the gradient one can optimize the calculation by reusing a certain amount of computations (for instance the arc derivatives). This leads to a more efficient implementation also known as *Tangent Multimode*. The constant ω_T for these implementations is generally smaller than in the standard Tangent mode.

Adjoint mode: Propagating Backwards

- Let's consider again the function $Y = \mathtt{FUNCTION}(X)$ implemented by means of a sequence of steps

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

- 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 $\mathbb{R}^m$.

Adjoint mode: Propagating Backwards

- 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 $\bar{U} = \bar{V}(U, \bar{V})$.

Adjoint mode: Propagating Backwards

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

- Contrary to the Tangent mode, the backward propagation can start only after the calculation of the function an the intermediate variables have been computed and stored.

Adjoint mode: Propagating Backwards

- Consider as before the example:

$$V_1 = \cos(U_1)U_1 + U_2 \exp(U_2)$$

  the corresponding Adjoints read

$$\begin{aligned}\bar{U}_1 &= \bar{V}_1(-\sin(U_1)U_1 + \cos(U_1)),\\ \bar{U}_2 &= \bar{V}_1(U_2 + 1)\exp(U_2))\end{aligned}$$

- Computing $\bar{U}$ (3 intrinsic operations, 4 multiplications and 2 additions) has the same computational complexity of computing the original function (2 intrinsic operations, 2 multiplications and 1 addition). Assuming that all the operations have the same cost it would be about twice as expensive.

Adjoint mode: Propagating Backwards

- Extending to the whole computation one can see how keeping into account of the relative cost of different types of operation one can arrive to the result [5]:

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

  with $\omega_A \in [3, 4]$.
- This result is based on the number of arithmetic operations which must be performed. It also includes the cost of memory operations, but assumes a uniform cost for these, irrespective of the total amount of memory used. This assumption is violated in practice due to the cache hierarchy in modern computers.
- Nevertheless, it remains true in practice that one can obtain the sensitivity of a single output, or a linear combination of outputs, to an unlimited number of inputs for only a little more work than the original calculation.

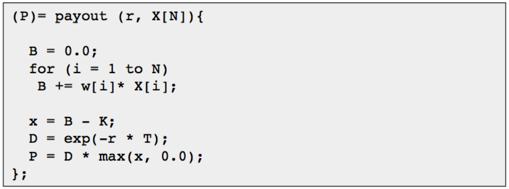
First Examples: Derivatives of Payoff Functions

- 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 $X(T) = (X_1(T), \ldots, X_N(T))$ represent 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. [6]

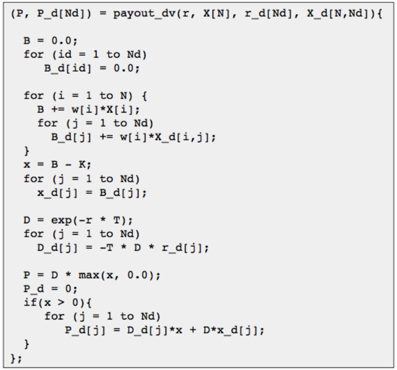
Pseudocode of the Tangent Payoff for the Basket Option

```
(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. [6]

- 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

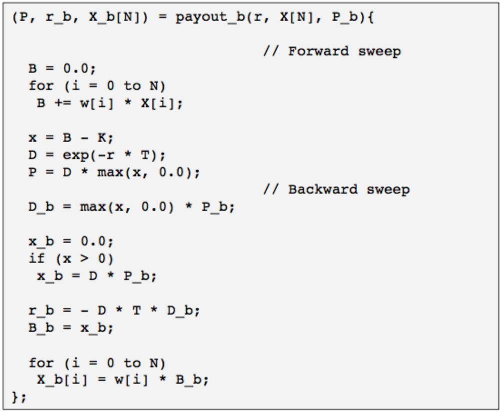


```
(P, P_d[Nd]) = payout_dv(r, X[N], r_d[Nd], X_d[N,Nd]){

  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. [6]

- 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

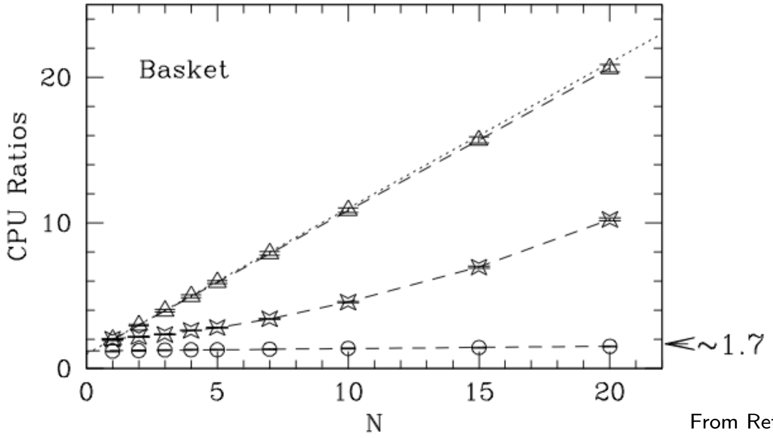


```
(P, r_b, X_b[N]) = payout_b(r, X[N], P_b){

                             // 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. [6]

- 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. [6]

- 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 model 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. [6].
- In the following we will concentrate on the Adjoint mode of Algorithmic Differentiation (AAD) as it is the one of wider applicability.

Section 5

AAD as a Design Paradigm

AAD as a Design Paradigm

- The propagation of the Adjoints according to the steps, being mechanical in nature, can be automated.
- Several AD tools are available that given a procedure of the form:

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

  generate the Adjoint function:

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

- An excellent source of information can be found at www.autodiff.org.
- Unfortunately, the application of such automatic AD tools on large inhomogeneous computer codes, like the one used in financial practices, is challenging.
- Fortunately, the principles of AD can be used as a programming paradigm for any algorithm.

AAD as a Design Paradigm

- Fortunately, the principles of AD can be used as a programming paradigm for any algorithm.
- An easy way to illustrate the Adjoint design paradigm is to consider again the arbitrary computer function

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

  and to imagine that this represents a certain high level algorithm that we want to differentiate.
- By appropriately defining the intermediate variables, any such algorithm can be abstracted in general as a composition of functions like

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

AAD as a Design Paradigm

- 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

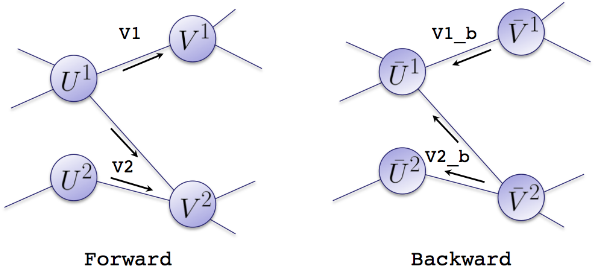
$$\begin{aligned} V^1 &:= \mathtt{V1}(U^1)\ , \\ V^2 &:= \mathtt{V2}(U^1, U^2)\ , \end{aligned}$$

  with $U = (U^1, U^2)^t$ 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.
- 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 Design Paradigm



- 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 Design Paradigm

- 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

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

- 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 Design Paradigm

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

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

  where we have used the fact that V^1 has no dependence on U^2.

- 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 $\bar{U} = (\bar{U}^1, \bar{U}^2)^t$.

AAD as a Design Paradigm

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

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

  where we have used the fact that V^1 has no dependence on U^2.
- Therefore, one can realize that the Adjoint calculation graph implementing the instructions in

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

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

Forward and Backward Sweeps

- The Adjoint instructions

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

  depend on the variables U^1 and U^2.
- 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 note 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.

Checkpointing

- The construction described above can be applied recursively for each of the functions involved in the calculation. In particular, each Adjoint function, taken in isolation, contains in turn a forward sweep recovering the information that is necessary for the propagation 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, for variables that do not occur linearly in the code, storing information is necessary to ensure that the computational cost of the overall algorithm remains within the expected bounds.

Checkpointing

- 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.
- Consider for instance a sequence of 4 steps: $A \to B \to C \to D \to E$.
- In the initial calculation, the input to each step is stored, but none of the intermediate values within the step except for the last. The Adjoint of the final step is performed, then before performing the Adjoint of the second last step it is necessary to re-evaluate that step, storing all of its intermediate values. This process is then repeated for the earlier steps, working backwards in order.
- The total cost is increased because all steps except for the last are computed twice, but this significantly reduces the memory requirements, and in practice will often reduce the cost as well because of the time taken to fetch data from memory.

Checkpointing

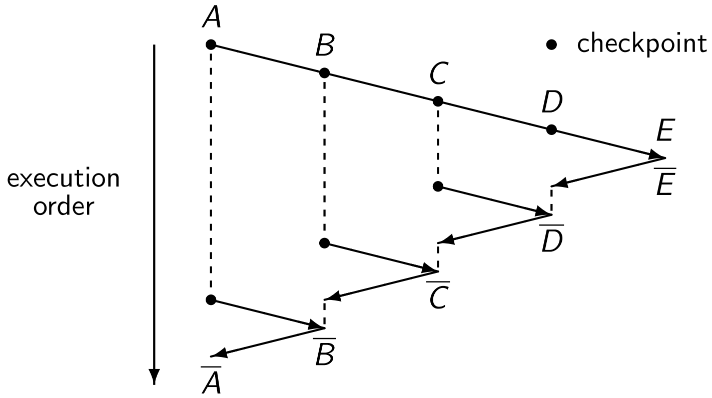


Illustration of checkpointing in which the “state” of the forward calculation is saved at critical points, so that the intermediate values required for the Adjoint calculation can be re-computed later.

Adjoint Programming in a Nutshell

- A detailed tutorial on the programming techniques that are useful for Adjoint implementations is beyond the scope of this course.
- However, when hand-coding the Adjoint counterpart of a set of instructions it is often enough to keep in mind just a few practical recipes.

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

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

Adjoint Programming in a Nutshell

b) 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).

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

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

Adjoint Programming in a Nutshell

The corresponfind Adjoint can be written as:

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

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 is often a helpful programming idiom.

Adjoint Programming in a Nutshell

c) In some situations the input U of a function $V = V(U)$ is modified by the function. As above, 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.

Adjoint Programming in a Nutshell

The Adjoint of this pair of instructions clearly reads

$$\bar{U}'_i = 0, \qquad \bar{U}'_i = \bar{U}'_i + \bar{U}_i,$$

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

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

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

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

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

d) 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.,

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

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.,

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

where we have used the standard addition assignment operator $+=$.

Section 6

AAD enabled Monte Carlo Engines

AAD enabled Monte Carlo Engines

- 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

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

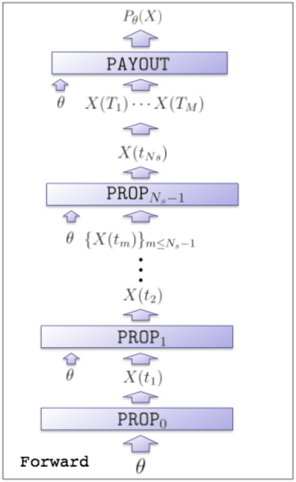
  mapping the set of state vector values on the discretization grid up to t_n, $\{X(t_m)\}_{m\leq n}$, into the value of the state vector at time $t_n + 1$.

AAD enabled Monte Carlo Engines: Forward Sweep

- Note that in $X(t_{n+1}) = \mathtt{PROP}_n[\{X(t_m)\}_{m\leq n}, Z(t_n), \theta]$:
  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, $X(t_1) = \mathtt{PROP}_0[Z(t_0), \theta]$.
- 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 $\bar{X} = (\bar{X}(T_1), \ldots, X(T_M))$ 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

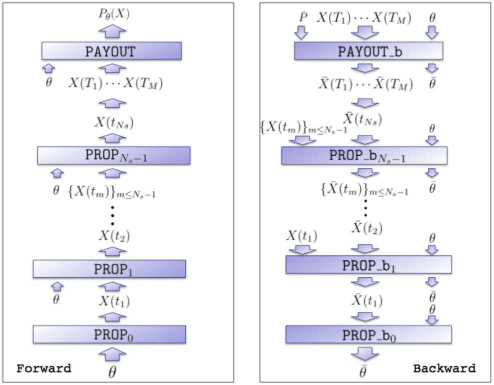
- Here, according to the principles of AAD, the Adjoint of the propagation method takes as arguments the inputs of its forward counterpart, namely the state vectors up to time t_n, $\{X(t_m)\}_{m\leq n}$, the vector of random variates $Z(t_n)$, and the θ vector. The additional input is the Adjoint of the state vector at time t_{n+1}, $\bar{X}(t_{n+1})$.
- The return values of $\mathtt{PROP\_b}_n$ 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 $X(t_1) = \mathtt{PROP}_0[Z(t_0), \theta]$, giving

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

  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.

Section 7

First Applications

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 9

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

- 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

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

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 matrix, 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

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

Diffusion Processes and Euler Discretization: Backward Sweep

- And:

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

$$\bar{X}_i(t_n) \mathrel{+}= \sum_{l,m=1}^{N} \bar{\sigma}_{l,m}(t_n) \frac{\partial \sigma_{l,m}(t_n)}{\partial X_i}, \qquad \bar{\theta}_k \mathrel{+}= \sum_{l,m=1}^{N} \bar{\sigma}_{l,m}(t_n) \frac{\partial \sigma_{l,m}(t_n)}{\partial \theta_k},$$

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

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

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

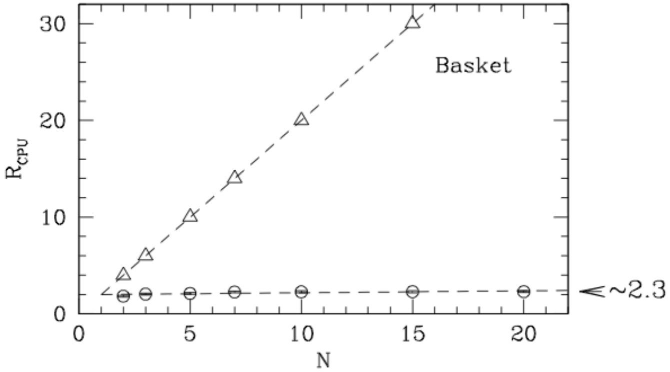
for $i = 1, \dots, N$ and $k = 1, \dots, N_\theta$.

Diffusion Processes and Euler Discretization: Backward Sweep

- Note that in the algorithm above we have followed the rules described in Slide 64 with respect to initializing and incrementing the Adjoints of intermediate variables.
- 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 (see slide 49) 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.

Section 8

Case Study: Adjoint Greeks for the Libor Market Model

Libor Market Model

- In order to make the connection with the algebraic implementations of Adjoint methods described before let's consider again the Libor Market Model (see Slide 14 and ff.).

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

- Denson and Joshi [4] extended the original implementation of Giles and Glasserman [2] to include the more accurate predictor-corrector scheme, consisting in replacing the usual Euler drift in Slide 15 with

$$\mu_i^{pc}(L(t_n)) = \frac{1}{2}\sum_{j=\eta(nh)}^{i}\left(\frac{\sigma_i^T\sigma_j\delta L_j(t_n)}{1+\delta L_j(t_n)} + \frac{\sigma_i^T\sigma_j\delta \hat{L}_j(t_{n+1})}{1+\delta \hat{L}_j(t_{n+1})}\right)$$

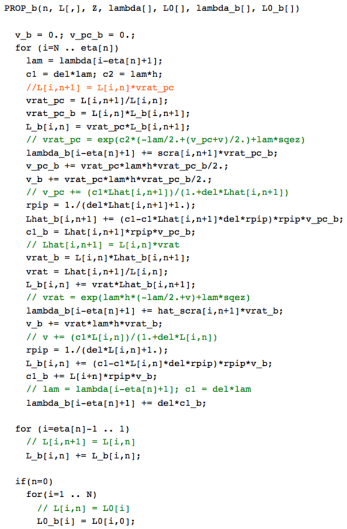
where $\hat{L}_j(t_{n+1})$ is calculated from $L_j(t_n)$ using the simple Euler drift.

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



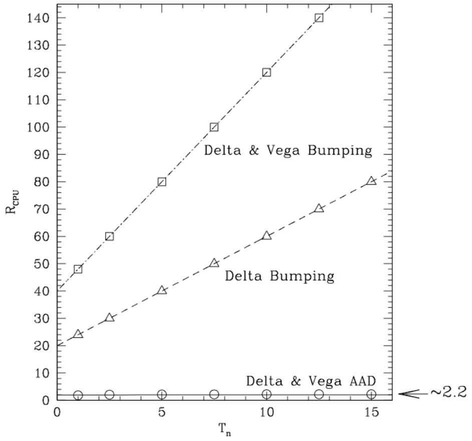
```
PROP_b(n, L[,], Z, lambda[], L0[], lambda_b[], L0_b[])

  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 $\texttt{PROP\_b}_n$.

Libor Market Model: Comments on the Code

- The algebraic formulation discussed in [4] 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.

Section 9

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.

- Like its algebraic counterpart, the proposed method allows the calculation of the complete risk at a computational cost which is at most 4 times the cost of calculating the P&L of the portfolio itself, resulting in remarkable computational savings with respect to standard finite differences approaches.

Conclusions

- In contrast to algebraic Adjoint methods, however, the algorithmic approach can be straightforwardly applied to both path dependent options and multi asset simulations. It also eliminates altogether the need for the sometimes cumbersome analytical work required by algebraic formulations.
- In this respect the Libor Market Model example we discuss is enlightening: a lenghty algebraic analysis can be substituted by the few lines of codes that can be written with the simples recipes presented in the paper.
- For these reasons, Algorithmic Differentiation is crucial to make Adjoint implementations practical in an industrial environment.

References I

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

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

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

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

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

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

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

See also:

My Publications' Page