Adjoint Algorithmic Differentiation and the derivative of the Cholesky decomposition

Paul Koerber

Belfius Bank and Insurance, Rogierplein 11, B-1210 Brussel, Belgium

Email: paul.koerber at gmail.com

15 December 2015

Abstract

Adjoint Algorithmic Differentiation is an efficient method to calculate the sensitivities of financial instruments to the input parameters. It does require an implementation of the derivative of every elementary step in the pricing algorithm. In particular, for calculating correlation sensitivities in a Monte Carlo simulation it has been proposed to use a step-by-step adjoint derivative of the Cholesky factorization. In this paper we introduce a one-line matrix-level algorithm which uses only matrix inversion and multiplication and can thus be performed by a standard library for linear algebra. Moreover, it allows for a more time-efficient calculation of the standard errors on these sensitivities.

1 Introduction

An important tool to assess the risks of a financial instrument is the calculation of the sensitivities of its price to changes in the input parameters, the so-called Greeks. A technique to obtain these sensitivities that is easy to implement is the finite difference method, which consists in bumping each parameter in turn and revalue. The computational cost grows proportionally to the number of input parameters, and can become very high if there are a lot of input parameters as, for instance, is the case for interest rate derivatives where one considers the sensitivities to bumping all the pillars of the interest rate curve. For the case at hand in this paper, namely correlation sensitivities, the number of pairwise correlations increases quadratically in the number of random variables. Furthermore, the finite difference method introduces numerical noise which depends on the bump size.

The technique of algorithmic differentiation (AD) on the other hand provides machine-precision accuracy. For a calculation with only one output — the price — and many inputs, it is particularly efficient in its adjoint mode (AAD). Indeed, [1, Section 4.6] shows that (under reasonable assumptions of the time-cost of different operations[1]) the cost is linear in the number of outputs Q, but independent of the number of inputs P:

$$\frac{\mathrm{TIME}(\mathrm{grad}(F))}{\mathrm{TIME}(F)} \in [1+2Q, 1.5+2.5Q] \overset{Q=1}{=} [3,4]\,, \tag{1}$$

where F indicates the original calculation, and $\mathrm{grad}(F)$ the calculation of the price plus sensitivities to all the inputs. The price to pay is that the derivative of every elementary step must be explicitly implemented.

For an early application of AAD in finance see [2] and for the calculation of correlation sensitivities using the adjoint differentiation of the Cholesky decomposition see [3]. We refer to [1] for a detailed explication and to [4] for an outline with applications to finance.

To make this paper self-contained, we briefly introduce AAD for a Monte Carlo simulation in section 2, while in section 3 we apply it to the calculation of correlation sensitivities. Section 4, which introduces the derivative of the Cholesky decomposition, is the core of this paper. Section 5 provides an application to best-of options. In the appendix we provide a complete implementation in Python 2.7.

[1]Some mild assumptions are made on the relative time-costs of respectively memory operations, additions, multiplications and non-linear operations. The most restrictive assumption in our opinion is that all elementary non-linear operations utilized, like exp and sin, are unary, i.e. they have only one input and one output.

Electronic copy available at: http://ssrn.com/abstract=2703893

2 Adjoint algorithmic differentiation

In short, AAD works as follows. We are interested in the partial derivatives of the price p with respect to all of the the input parameters u_i, $i = 1, \ldots, P$, which we denote with a bar

$$\bar{u}_i := \frac{\partial p}{\partial u_i} \,. \tag{2}$$

Suppose the calculation of the price consists of L steps with intermediate results $v_i^{(n)}$, $i = 1, \ldots, P_n$ (where we take $P_0 = P$, $P_L = Q = 1$ and $v_i^{(0)} = u_i$, $v_1^{(L)} = p$). The flow of calculations looks like

$$\begin{pmatrix} u_1 \\ \vdots \\ u_P \end{pmatrix} \to \begin{pmatrix} v_1^{(1)} \\ \vdots \\ v_{P_1}^{(1)} \end{pmatrix} \to \ldots \to \begin{pmatrix} v_1^{(L-1)} \\ \vdots \\ v_{P_{L-1}}^{(L-1)} \end{pmatrix} \to p \,. \tag{3}$$

Using the chain rule of differentiation we can find the sensitivities $\bar{v}^{(n-1)}$ of the price to the intermediate results of step $n-1$ once we know the sensitivities $\bar{v}^{(n)}$ to the results of step n:

$$\bar{v}_i^{(n-1)} := \frac{\partial p}{\partial v_i^{(n-1)}} = \sum_{j=1}^{P_n} \frac{\partial p}{\partial v_j^{(n)}} \frac{\partial v_j^{(n)}}{\partial v_i^{(n-1)}} = \sum_{j=1}^{P_n} \bar{v}_j^{(n)} \frac{\partial v_j^{(n)}}{\partial v_i^{(n-1)}} \,. \tag{4}$$

We start by putting $\bar{v}_1^{(L)} = \bar{p} = 1$ and we work our way *backwards* to find the sensitivities to the input parameters $\bar{u}_i = \bar{v}_i^{(0)}$. We remark that all partial derivatives are evaluated at the point given by the values $v_i^{(n)}$ that result from the original calculation of the price. AAD therefore requires first performing the calculation of the price (the *forward sweep*) and then iterating through eq. (4) in the opposite direction to obtain the sensitivities (the *backward sweep*). In deciding how much data to store during the forward sweep, there is a trade-off between memory usage and recalculation time.

While it is always possible to represent a calculation by a strictly linear sequence of steps as in eq. (3), most calculations naturally take a more complicated form. See figure 1 for the example of a Monte-Carlo calculation. In that diagram the same intermediate result is used in several subsequent steps. For instance the output of the nth time step of the process (the asset prices $S(t_n)$) feeds into both the next time step and the payoff accumulator. Suppose in general that some intermediate result $v_i^{(n)}$ is used in several subsequent steps. We will then have to add contributions of the form (4) for every such step. It is convenient to implement this as follows. Initially we set the sensitivity to all intermediate results to zero: $\bar{v}_i^{(n)} \leftarrow 0$. When processing a step s we update the sensitivities to all its inputs $\bar{v}_i^{(n)}$,

$$\bar{v}_i^{(n)} \leftarrow \bar{v}_i^{(n)} + \sum_{j=1}^{P_s} \bar{v}_j^{(s)} \frac{\partial v_j^{(s)}}{\partial v_i^{(n)}} \,. \tag{5}$$

To apply AAD to a Monte Carlo simulation we use the *pathwise derivative method* [5, 6]. Suppose the price of an option is given by the expectation value

$$p = E^{\mathbb{Q}}[P_u(X_u)] \,, \tag{6}$$

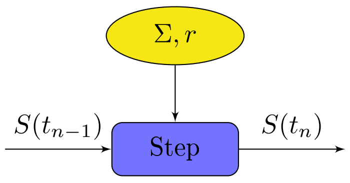
where, generically, both the payoff P_u and the probability distribution of the random variables X_u can depend on the parameters u. We rewrite the payoff in terms of random variables Z that do not depend on u: $P_u(X_u(Z)) = \tilde{P}_u(Z)$. If the payoff function is regular enough (see e.g. [5]), it is possible to move partial derivatives into the expectation value so that the sensitivities are given by

$$\bar{u}_i = \partial_{u_i} p = E^{\mathbb{Q}}[\partial_{u_i} \tilde{P}_u(Z)] \,. \tag{7}$$

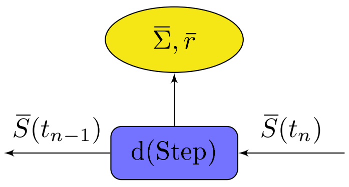
In a Monte Carlo simulation eq. (6) is approximated by

$$p \approx \frac{1}{N_{\mathrm{MC}}} \sum_{K=1}^{N_{\mathrm{MC}}} \tilde{P}_u(Z^{(K)}) \,, \tag{8}$$

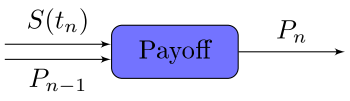
Electronic copy available at: http://ssrn.com/abstract=2703893



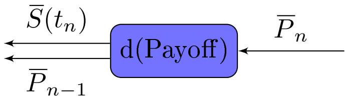
(a) Time step (forward sweep)



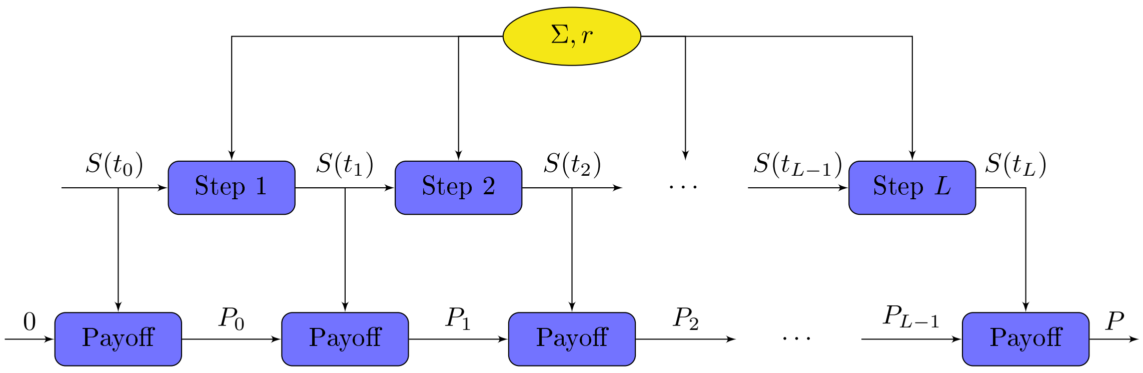
(b) Derivative of the time step (backward sweep)



(c) Payoff accumulator (forward sweep)



(d) Derivative of payoff accumulator (backward sweep)



(e) Diagram of the forward sweep of a Monte Carlo simulation

Figure 1: Diagram of one path in a generic Monte Carlo simulation showing two building blocks: the time step of the process and the payoff accumulator. We allow for Asianing: the payoff accumulator can probe the asset prices at different times t_n. This is used in the example of section 5. In the backward sweep the order of calculation and therefore all arrows in the diagram are reversed.

where $Z^{(K)}$ is the Kth sample of the random variables Z. We can likewise obtain the sensitivities from approximating eq. (7) as follows

$$\bar{u}_i \approx \frac{1}{N_{\mathrm{MC}}} \sum_{K=1}^{N_{\mathrm{MC}}} \partial_{u_i} \tilde{P}_u(Z^{(K)}) \,, \tag{9}$$

and use the same samples $Z^{(K)}$ for the calculation of the price and the sensitivities.

3 Sensitivities of a multivariate lognormal process

In this section we apply AAD and the pathwise derivative method for calculating the sensitivities — and in particular the correlation sensitivities — of an option with a payoff depending on a basket of assets S_i, $i = 1, \dots, I$. Suppose the assets follow a multivariate lognormal process:

$$\begin{aligned} \mathrm{d}S_i(t) &= r_i S_i(t)\,\mathrm{d}t + \sigma_i S_i(t)\,\mathrm{d}W_i(t)\,, \\ \mathrm{cov}(\mathrm{d}W_i(t), \mathrm{d}W_j(t)) &= \rho_{ij}\mathrm{d}t\,, \end{aligned} \tag{10}$$

where r_i is the risk-neutral drift of asset i, σ_i its volatility, and ρ_{ij} the correlation matrix. Pricing the option through Monte Carlo simulation requires evolving of the multivariate lognormal process over a time step $\Delta t_n = t_n - t_{n-1}$. In fact, the exponential of the Euler discretization for $\log S_i$,

$$S_i(t_n) = S_i(t_{n-1}) \exp\left(\left(r_i - \frac{1}{2}(\sigma_i)^2 \right)(t_n - t_{n-1}) + X_i(t_{n-1})\sqrt{t_n - t_{n-1}} \right) \tag{11}$$

is exact for this process. Here $X(t) = (X_1(t), \dots, X_I(t))^T$ is a random vector drawn from a multivariate Gaussian distribution with covariance matrix Σ given by

$$\Sigma_{ij} = \sigma_i \sigma_j \rho_{ij} \,. \tag{12}$$

The probability distribution of X depends therefore on the input parameters σ_i, ρ_{ij} through Σ_{ij}. As explained above eq. (7) we would like to introduce random variables Z that do not depend on the parameters. For every matrix C such that

$$\Sigma = CC^T \,, \tag{13}$$

which we can construct for instance using the Cholesky algorithm, we have that

$$X = CZ \tag{14}$$

obeys a multivariate Gaussian distribution with covariance matrix Σ if Z is a vector of independent normalized Gaussian random variables. We will briefly review the proof of this well-known statement in section 4.1. After the transformation from X to Z we can apply the pathwise derivative formula, eq. (9), to calculate the sensitivities through Monte Carlo simulation.

To proceed we need an implementation of the derivative of each step in the pricing algorithm, the bulk of which is given by iteration of the Euler step of eq. (11). See figure 1 for a diagram of the different calculation steps. The derivative of the Euler step with respect to the input states $S_i(t_{n-1})$ and the drifts r_i is given by

$$\begin{aligned} \bar{S}_i(t_{n-1}) &\leftarrow \bar{S}_i(t_{n-1}) + \frac{S_i(t_n)}{S_i(t_{n-1})}\, \bar{S}_i(t_n) \,, \\ \bar{r}_i &\leftarrow \bar{r}_i + (t_n - t_{n-1}) S_i(t_n) \bar{S}_i(t_n) \end{aligned} \tag{15}$$

Iterating the first line above until we reach t_0 allows for obtaining the Deltas, which are the sensitivities to the initial asset prices $S_i(t_0)$.

To calculate the sensitivity to the covariance matrix $\bar{\Sigma}$ we first obtain $\bar{C}$ from the derivatives of eqs. (11) and (14)

$$\bar{X}_i(t_{n-1}) \leftarrow \bar{X}_i(t_{n-1}) + \sqrt{t_n - t_{n-1}}\, S_i(t_n) \bar{S}_i(t_n) \,, \tag{16a}$$
$$\bar{C}_{ij} \leftarrow \bar{C}_{ij} + \bar{X}_i(t_{n-1}) Z_j(t_{n-1}) \,. \tag{16b}$$

Next we need an implementation of the derivative of the Cholesky decomposition $C(\Sigma)$,

$$\bar{\Sigma}_{C;kl} = \sum_{i,j} \bar{C}_{ij} \frac{\partial C_{ij}}{\partial \Sigma_{kl}} \,, \tag{17}$$

which is the subject of the next section. The total sensitivity to the covariance matrix is

$$\bar{\Sigma}_{kl} \leftarrow \bar{\Sigma}_{kl} + \bar{\Sigma}_{C;kl} - \frac{1}{2} \delta_{kl} (t_n - t_{n-1})\, S_k(t_n) \bar{S}_k(t_n) \,, \tag{18}$$

where the additional term comes from the Itô correction to the drift in eq. (11).

The sensitivity to the covariance matrix can be re-expressed in terms of the more familiar sensitivities to the volatilities (Vegas) and correlations (Cegas). Taking the derivative of eq. (12) we find

$$\bar{\sigma}_i = 2 \sum_j \bar{\Sigma}_{ij} \rho_{ij} \sigma_j \,, \qquad \bar{\rho}_{ij} = \bar{\Sigma}_{ij} \sigma_i \sigma_j \,. \tag{19}$$

For the adjoint differentiation of the Cholesky factorization, reference [3] proposes to use the algorithm of [7], which boils down to a step-by-step differentiation of the Cholesky algorithm.

4 Derivative of the Cholesky decomposition

Next we argue that we do not need the exact derivative of the Cholesky decomposition in order to obtain the sensitivities $\bar{\Sigma}_{kl}$. As we will make precise it suffices to calculate the derivative modulo an orthogonal transformation. For completeness we will in subsection 4.4 nevertheless obtain an exact expression for the derivative of the Cholesky decomposition that reproduces the results of the algorithm of [7].

4.1 Relaxing the definition of the Cholesky decomposition

The Cholesky decomposition C of Σ is defined as the unique matrix that satisfies

$$\Sigma = CC^T \,, \tag{20a}$$
$$C = C|_{\mathrm{LT}} \,, \tag{20b}$$

where $C|_{\mathrm{LT}}$ denotes the projection of C on its lower triangular part. Similarly we denote with $\cdot|_{\mathrm{UT}}, \cdot|_{\mathrm{SLT}}$ and $\cdot|_{\mathrm{SUT}}$ the projection of a matrix on its upper triangular part, its strictly lower triangular part and strictly upper triangular part respectively, where by "strict" we mean that the diagonal is zero.

For the purpose of sampling correlated normal random variables the key property of the Cholesky decomposition is the following: if $Z = (Z_1, \dots, Z_I)^T$ is a vector of independent standard normal random variables, it follows that

$$X = CZ \tag{21}$$

is a vector of normal random variables with covariance Σ. Indeed, we know that linear combinations of independent normal random variables are still normal random variables, which are now dependent. The covariance matrix of the resulting X is

$$\mathrm{cov}(X, X^T) = C\,\mathrm{cov}(Z, Z^T)\,C^T = CIC^T = CC^T = \Sigma\,. \tag{22}$$

It is important to realize that the second condition in the definition above, eq. (20b), is not essential. We did not need it to show that X has the required covariance matrix. It follows that any matrix that satisfies $CC^T = \Sigma$ will do the job. Taking two such matrices C and $\tilde{C}$, we find

$$CC^T = \tilde{C}\tilde{C}^T \Rightarrow (C^{-1}\tilde{C})(C^{-1}\tilde{C})^T = \mathbb{1}\,, \tag{23}$$

where we used that C is non-degenerate because Σ is non-degenerate. It follows that

$$\tilde{C} = CO\,, \quad \text{with } OO^T = \mathbb{1}\,. \tag{24}$$

In other words, all matrices $\tilde{C}$ satisfying $\tilde{C}\tilde{C}^T = \Sigma$ are related to the lower triangular matrix of the Cholesky decomposition and to each other by an orthogonal transformation ($O \in \mathrm{O}(\mathbb{R})$). Condition (20b) makes a convenient choice among these matrices. It leads to a unique solution and, importantly, allows for an efficient algorithm to find this solution. Borrowing from physics terminology, we will call (20b) a *gauge condition*.

In the context of a Monte Carlo simulation, choosing $\tilde{C}$ instead of C will change each individual path. Indeed, given the sample vector $Z^{(K)}$ in path K we replace $X^{(K)} = CZ^{(K)}$ by $\tilde{X}^{(K)} = COZ^{(K)}$. Nevertheless, the $\tilde{X}^{(K)}$ are still unbiased samples from the same multivariate Gaussian distribution with covariance matrix Σ. This implies in particular that the resulting option price, eq. (6), is unchanged, which holds approximately also for its simulation as the Monte Carlo mean. Let us now apply the same argument to the calculation of the sensitivities to the covariance matrix. We consider an infinitesimal variation $\Sigma \to \Sigma + \delta\Sigma$ and we want to calculate the corresponding variation $C \to C + \delta C$. The crucial point is that to generate samples $X^{(K)} + \delta X^{(K)} = (C + \delta C)Z^{(K)}$ it is not essential to insist that $C + \delta C$ should preserve the gauge condition (20b). Even without this condition we will obtain an unbiased sample from a multivariate Gaussian distribution with covariance matrix $\Sigma + \delta\Sigma$.

4.2 Gauge-violating derivative of the Cholesky decomposition

Let us now construct a derivative of the Cholesky decomposition that does not (necessarily) obey the gauge condition (20b). We take the differential of the first condition (20a):

$$\mathrm{d}\Sigma = \mathrm{d}CC^T + C(\mathrm{d}C)^T\,. \tag{25}$$

It turns out that this equation is solved by

$$\mathrm{d}C = \frac{1}{2}\,\mathrm{d}\Sigma\, C^{-T}\,, \tag{26}$$

which can easily be checked by plugging it back into the equation. As we will discuss in more detail in section 4.4 it is not the only solution to eq. (25). Generically it is not lower triangular, so it does *not* correspond to the derivative of the Cholesky decomposition that preserves the gauge condition. But it does suit our purpose of sampling $X + \delta X$ from the perturbed multivariate normal distribution. To proceed it is convenient to reintroduce indices. From

$$\mathrm{d}C_{ij} = \frac{1}{2} \sum_l \mathrm{d}\Sigma_{il} (C^{-1})_{jl} \tag{27}$$

we find

$$\frac{\partial C_{ij}}{\partial \Sigma_{kl}} = \frac{1}{2}\, \delta_{i(k|} (C^{-1})_{j|l)} \,, \tag{28}$$

where (kl) indicates appropriately weighted symmetrization of the indices, and indices between vertical bars do not partake, e.g.

$$A_{i(k|mn|l)j} := \frac{1}{2} \left(A_{ikmnlj} + A_{ilmnkj} \right) \,. \tag{29}$$

The symmetrization comes about because Σ and thus also $\mathrm{d}\Sigma$ is a symmetric matrix. Plugging this result into eq. (17) leads to

$$\bar{\Sigma}_{C;kl} = \frac{1}{2} \sum_j \bar{C}_{(k|j} (C^{-1})_{j|l)} \,, \tag{30}$$

or in matrix notation

$$\bar{\Sigma}_C = \frac{1}{2} \mathrm{Sym} \left(\bar{C} C^{-1} \right) \,, \tag{31}$$

where we introduced the symmetrization of a matrix

$$\mathrm{Sym}(A) := \frac{1}{2} \left(A + A^T \right) \,. \tag{32}$$

Finally from eq. (16b)

$$\bar{\Sigma}_{kl} \leftarrow \bar{\Sigma}_{kl} + \frac{1}{2} \sum_j \bar{X}_{(k|}(t_{n-1}) Z_j(t_{n-1}) (C^{-1})_{j|l)} \,, \tag{33}$$

for all time steps n, or in matrix notation

$$\bar{\Sigma} \leftarrow \bar{\Sigma} + \frac{1}{2} \mathrm{Sym} \left(\bar{X}(t_{n-1}) Z^T(t_{n-1}) C^{-1} \right) \,. \tag{34}$$

4.3 Time complexity

Since C is lower triangular the calculation of its inverse can be efficiently implemented, although the asymptotic time complexity is still $O(I^3)$, where I is the number of factors. To solely obtain the Monte Carlo estimate of $\bar{\Sigma}$ it suffices to take the average of $\bar{C}$ over Monte Carlo paths and do the calculation in eq. (17) only once at the end.

In order to obtain an error estimate for $\bar{\Sigma}$ as well, reference [3] proposes to work out $\bar{\Sigma}$ for every Monte Carlo path and then to take the standard deviation. In this case, since C does not depend on the specific Monte Carlo path, the inversion of C still has to be performed only once. Remaining are matrix multiplications for every path. It is important to perform the matrix-vector multiplication $Z^T(t_{n-1})C^{-1}$ first, with a cost of order $O(I^2)$ per path, and then the multiplication with $\bar{X}(t_{n-1})$ with a cost of the same order. Performing the matrix multiplications in the other order, i.e. $\bar{X}(t_{n-1})Z^T(t_{n-1})$ first and then a full matrix-matrix multiplication, would incur a cost of order $O(I^3)$ instead (with the naive algorithm for matrix multiplication). The total cost is thus of order $O(I^3) + O(I^2 N_{\mathrm{MC}}) = O(I^2 N_{\mathrm{MC}})$ (since $N_{\mathrm{MC}} \gg I$), which is the same order as the other steps in the Monte Carlo implementation. In contrast, using the step-by-step derivative of the Cholesky algorithm by Smith [7] for every Monte Carlo path leads to a complexity of $O(I^3 N_{\mathrm{MC}})$ as was observed in [3]. It would then become the bottleneck of the calculation since all other steps cost only $O(N_{\mathrm{MC}} I^2)$. Reference [3] proposes to remedy this by dividing the N_{MC} paths into bins, and taking the standard deviation of $\bar{\Sigma}$ per bin, adding bookkeeping complexity to the implementation.

Another and practically more interesting advantage of the result (34) is that we do not have to implement a custom algorithm, like the Smith algorithm. Instead we can use a standard optimized library for linear algebra to perform the matrix inversion and multiplications.

4.4 Gauge-preserving derivative of the Cholesky decomposition

For completeness let us obtain an alternative expression for the gauge-preserving derivative of the Cholesky decomposition, which exactly reproduces the result of [7]. The differential $\mathrm{d}C$ should then satisfy eq. (25) *and* be lower triangular. We know already that $\mathrm{d}C^p = \frac{1}{2}\,\mathrm{d}\Sigma\, C^{-T}$ is a particular solution of eq. (25). The most general solution of this linear equation is obtained by putting $\mathrm{d}C = \mathrm{d}C^p + L$. After some manipulation we find that L should satisfy

$$C^{-1}L = -(C^{-1}L)^T\,. \tag{35}$$

In other words $A = C^{-1}L$ should be an antisymmetric matrix, $A^T = -A^T$. So the general solution is of the form

$$\mathrm{d}C = \frac{1}{2}\,\mathrm{d}\Sigma\, C^{-T} + CA\,, \tag{36}$$

where A is a generic antisymmetric matrix. In fact, the extra term corresponds to the infinitesimal version of the orthogonal transformation introduced in eq. (24).[2]

The task at hand is to determine A such that $\mathrm{d}C$ is lower triangular. We impose

$$\mathrm{d}C|_{\mathrm{SUT}} = \left(\frac{1}{2}\,\mathrm{d}\Sigma\, C^{-T} + CA\right)\bigg|_{\mathrm{SUT}} = 0\,. \tag{37}$$

We rewrite $A = U - U^T$ where U is strictly upper triangular. We can then solve for U using the following facts: C is lower triangular, the product of a lower triangular matrix with a lower triangular matrix is lower triangular and the inverse of a lower triangular matrix is lower triangular as well. We find that

$$U = -\frac{1}{2}\left(C^{-1}\mathrm{d}\Sigma\, C^{-T}\right)\Big|_{\mathrm{SUT}} \Longrightarrow A = \frac{1}{2}\left(\left(C^{-1}\mathrm{d}\Sigma\, C^{-T}\right)\Big|_{\mathrm{SLT}} - \left(C^{-1}\mathrm{d}\Sigma\, C^{-T}\right)\Big|_{\mathrm{SUT}}\right)\,. \tag{38}$$

To proceed we plug this into eq. (36) and then use eq. (17) to obtain $\overline{\Sigma}$. In index notation we find

$$\overline{\Sigma}_{kl} = \frac{1}{2}\sum_j \overline{C}_{(k|j}(C^{-1})_{j|l)} + \frac{1}{2}\sum_{i,j,r,j<r} \overline{C}_{ij}C_{ir}(C^{-1})_{r(k|}(C^{-1})_{j|l)} - \frac{1}{2}\sum_{i,j,r,j>r} \overline{C}_{ij}C_{ir}(C^{-1})_{r(k|}(C^{-1})_{j|l)}\,, \tag{39}$$

which can finally be rewritten without indices as

$$\overline{\Sigma} = \frac{1}{2}\,\mathrm{Sym}\left(\overline{C}C^{-1} + C^{-T}\left(\left(\overline{C}^T C\right)_{\mathrm{SUT}} - \left(\overline{C}^T C\right)_{\mathrm{SLT}}\right)C^{-1}\right)\,. \tag{40}$$

The first term reproduces the gauge-violating result, eq. (31), and the following terms correspond to an infinitesimal compensating orthogonal transformation restoring the gauge condition. Furthermore if $\overline{C}$ is strictly upper triangular we have $\overline{\Sigma} = 0$. This makes sense as sensitivities of the price to changes of the upper triangular part of C are spurious since we keep this part constant at zero upon changing Σ while respecting the gauge constraint.

We tested our formula by comparing to the results of the Smith algorithm and got exact agreement. A similar expression was also found in [8, Section 3.6.2].

5 Example: Asian best-of option

As an example we implement an Asian best-of option. For a basket of stocks we take the average of the price of each stock over a series of Asianing dates. At maturity the payoff is a vanilla call/put payoff having as underlying U the maximum of these averages

$$U = \max_i\left(\frac{1}{N}\sum_{n=1}^{N} S_i(t_n)\right)\,, \qquad P = \max\left(\eta\left(U - K\right), 0\right)\,, \tag{41}$$

[2]The Lie-algebra of orthogonal transformations consists of antisymmetric matrices.

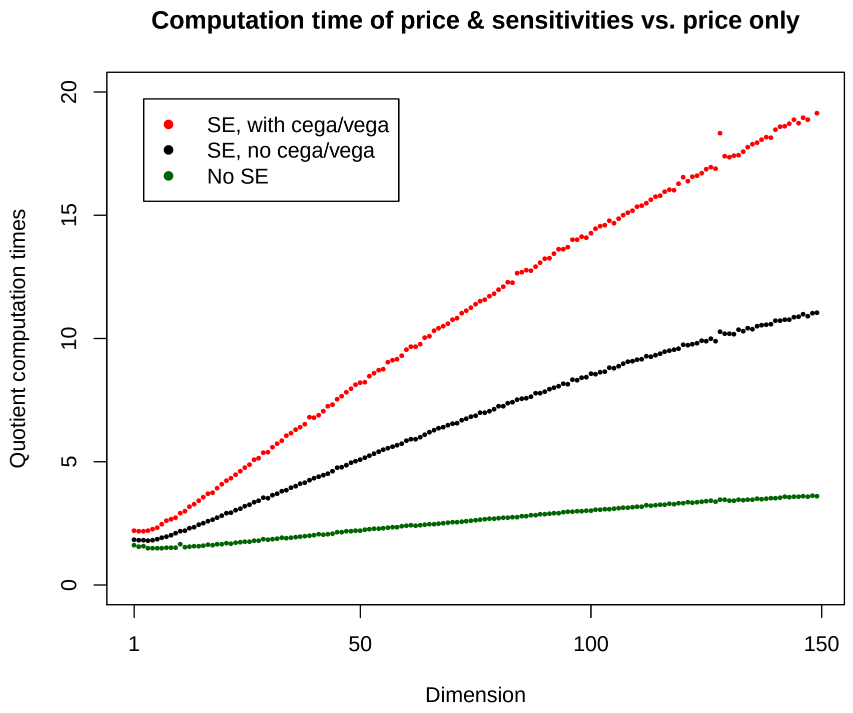


Figure 2: Calculation time of price and sensitivities as compared to the calculation time for the price only for an implementation in C++. We plot the quotient of computation times with and without calculation of the standard errors of the sensitivities (see the discussion of section 4.3), and the latter with and without the calculation of Vegas and Cegas in addition to the sensitivities to the covariance matrix. The option under consideration has a best-of payoff with 5 Asianing dates and maturity equal to the last Asianing date. The initial stock prices, the strike as well as the covariance matrix are randomly chosen. The number of simulations is taken to be 25000 and the number of stocks is increasing from 1 to 150. Each point corresponds to the minimum time of a series of identical runs.

where $\eta = \pm 1$ for a call/put option respectively. To implement AAD we need the derivative of this payoff:

$$\overline{U} = \eta\,\theta(\eta(U-K))\overline{P}\,, \qquad \overline{S}_i(t_n) = \begin{cases} \overline{U}/N & \text{for } i = k \\ 0 & \text{otherwise} \end{cases}\,, \qquad \text{with } k = \text{argmax}_i \left(\sum_{n=1}^{N} S_i(t_n) \right)\,, \quad (42)$$

where θ is the step function. The sensitivities $\overline{S}_i(t_n)$ then feed into the derivative of the Euler step, eq. (15).

We model the evolution of the stocks as the multivariate lognormal process of eq. (10). We take the calculation dates (t_n) to include the Asianing dates and the maturity date. Listing 1 shows an implementation of the multivariate lognormal step function in Python.[3] The derivative of the step function for each step is constructed and returned within the step function itself, and likewise for the payoff function. In a programming language with functional capabilities such nested functions have access to the environment of their parent function. This provides a lightweight way for the derivative functions to have access to the intermediate and final variables of the forward calculation. For an implementation of the payoff function and its derivative see listing 2.

Both the step function and the payoff function are fed into the Monte Carlo implementation, which follows the diagram 1. The Python code is shown in listing 3.

Figure 2 plots in terms of the number of assets the calculation time of price and sensitivities as compared to the calculation time for the price only, i.e. the quotient of eq. (1). For the calculation

[3] We also have an implementation in C++, giving clearer timing results, which we show in figure 2. The C++ code is however less succinct and not suitable for displaying in its entirety.

without standard errors of the sensitivities we find that the quotient starts at 1.6, below the expected value for AAD, and converges to the expected value between 3 and 4 of eq. (1) for high values of the dimension I. This is explained by the fact that for low values of I the operation of constructing the random numbers, which has time complexity $O(I)$, dominates the price calculation, while this operation is absent in the calculation of the sensitivities. It is only at higher values of I that the $O(I^2)$ operations discussed in section 4.3 take over. Including the calculation of the standard errors on the sensitivities, the quotient converges to a higher factor. However, for a huge amount of factors, namely 150, the quotient is still only 11.0 without or 19.1 with the extra calculation of Vegas and Cegas and their standard errors.

We tested pricing and sensitivities by comparing with an implementation based on the Smith algorithm.

6 Conclusion

Our main result is a simple one-line matrix expression for calculating the sensitivities to the covariance matrix of a basket of assets in a path of a Monte Carlo simulation through Adjoint Algorithmic Differentiation, eq. (34):

$$\overline{\Sigma} = \frac{1}{2}\mathrm{Sym}\left(\overline{X} Z^T C^{-1}\right) , \tag{43}$$

where $\overline{X}$ indicate the sensitivities to the multivariate normal random variables and Z are the standard normal variables used in the Monte Carlo path. Since the matrix C, which we need for constructing samples from a multivariate normal distribution in the Monte Carlo simulation, is obtained from the covariance matrix Σ through the Cholesky decomposition, we would have expected AAD to require an implementation of the derivative of the Cholesky decomposition. We argued however that eq. (31),

$$\overline{\Sigma} = \frac{1}{2}\mathrm{Sym}\left(\overline{C} C^{-1}\right) , \tag{44}$$

on which eq. (34) is based, is not the exact expression for this derivative, but it suffices for our purpose. We also derived the exact expression, which is given in eq. (40):

$$\overline{\Sigma} = \frac{1}{2}\mathrm{Sym}\left(\overline{C} C^{-1} + C^{-T}\left(\left(\overline{C}^T C\right)_{\mathrm{SUT}} - \left(\overline{C}^T C\right)_{\mathrm{SLT}}\right) C^{-1}\right) . \tag{45}$$

We verified that our formula works correctly in a concrete implementation for the example of Asian best-of options on a basket of assets.

Acknowledgments I would like to thank the people at Belfius Validation for stimulating discussions and especially Tim Pillards and Thierry Becker for proofreading the draft. Nevertheless all mistakes are solely mine, and the opinions and views in this paper do not necessarily represent those of Belfius Bank.

A Python code

```
from math import sqrt
import numpy as np

# Create the step function for the multivariate lognormal process
# Input:  Sigma:  covariance matrix, r:  vector of risk-neutral drifts
# Output:  step function (including derivative)
def step_lognormalmultivariate(Sigma, r=0):
    C0 = np.linalg.cholesky(Sigma)
    C0inv_transpose = np.transpose(np.linalg.inv(C0))
    # $\sigma_i$:  sqrt of diagonal of covariance matrix $\Sigma$
    sigma = np.sqrt(np.diagonal(Sigma))
    sigma_sq_half = np.outer(0.5*sigma,sigma)
    def step_fun(S0, t0, t1):
        Z = np.random.randn(Sigma.shape[0])
        S1= S0*np.exp((r-0.5*np.diagonal(Sigma))*(t1-t0)+sqrt(t1-t0)*np.dot(C0,Z))
        def step_fun_derivative(S1_diff):
            # X_diff is $\overline{X}/\sqrt{t_1 - t_0}$
            X_diff=S1_diff*S1
            S0_diff=X_diff/S0
            r_diff=X_diff*(t1-t0)
            # Sensitivity $\Sigma$
            # Part 1:  diagonal part from Ito correction to the drift (eq. (18))
            Sigma_diff=np.diag((-0.5*(t1-t0))*X_diff)
            # Part 2:  from derivative Cholesky (eq. (34))
            Sigma_diff0=np.outer((0.25*sqrt(t1-t0))*X_diff, np.dot(C0inv_transpose, Z))
            Sigma_diff+=Sigma_diff0+np.transpose(Sigma_diff0)
            # Extracting Vegas and Cegas (eq. (19))
            Vega=2*np.einsum("ij,ij->i",Sigma_diff,Sigma)/sigma
            Cega0=Sigma_diff*sigma_sq_half
            Cega=Cega0+np.transpose(Cega0)
            return S0_diff, [r_diff, Sigma_diff, Vega, Cega]
        return S1, step_fun_derivative
    step_fun.which_sensitivities=["r_diff", "Sigma_diff", "Vega", "Cega"]
    return step_fun
```

Listing 1: Implementation of the multivariate lognormal step function in Python

```
import numpy as np

# Creates a payoff function with Asianing
# Input:  K: strike, asianing_dates:  vector of Asianing dates, mat:  maturity date
# callorput:  1:  call, -1:  put, discountfactor:  discount factor for payoff
# final_payoff_fun:  how to combine averages of stocks into payoff (default and in text:  take the max)
# final_payoff_fun_derivative:  derivative of final_payoff_fun for calculating sensitivities
# Output:  payoff function (including derivative)
def asian_payoff_fun(K, asianing_dates, mat, callorput=1, discountfactor=1.0, final_payoff_fun=np.max,
                     final_payoff_fun_diff=lambda x: (x==np.max(x))*1.0):
    def payoff_fun(previous, t1, S1):
        acc_payoff=previous+S1 if t1 in asianing_dates else previous
        if t1==mat:
            final_underlying=final_payoff_fun(acc_payoff/len(asianing_dates))
            payoff=discountfactor*max(callorput*(final_underlying-K),0)
        else: payoff=acc_payoff
        def payoff_fun_derivative(payoff_diff):
            if t1==mat:
                payoff_diff=discountfactor*(callorput*(final_underlying-K)>0)*callorput*
                                            final_payoff_fun_diff(acc_payoff)/len(asianing_dates)
            state1_diff=payoff_diff if t1 in asianing_dates else 0
            return state1_diff, payoff_diff
        return payoff, payoff_fun_derivative
    return payoff_fun
```

Listing 2: Implementation of the payoff function with Asianing

```python
from math import sqrt
from collections import OrderedDict
import numpy as np

# Implements a general Monte-Carlo calculation as in diagram 1
# Input:  start_state:  start state, step_times:  vector calculation times,
# step_fun:  step function (and derivative), payoff_fun:  payoff function (and derivative)
# num_simulations:  number of Monte-Carlo paths
# calculate_sensitivities:  whether to calculate sensitivities
# Output:  price and its standard error, sensitivities and their standard errors """
def montecarlo(start_state, step_times, step_fun, payoff_fun, num_simulations=25000,
               calculate_sensitivities=True):
    acc_price, acc_var=0, 0
    labels_sensitivities=["Delta"]+step_fun.which_sensitivities
    acc_sensitivities=[0]*len(labels_sensitivities)
    acc_sensitivities_var=[0]*len(labels_sensitivities)
    sensitivities=[0]*(len(labels_sensitivities)-1)
    step_diff_funs=[0]*len(step_times)
    payoff_diff_funs=[0]*len(step_times)
    for n in xrange(1,num_simulations+1):
        # Forward sweep
        state=start_state
        payoff=0
        n_inv=1/float(n)
        for step_idx in xrange(0, len(step_times)):
            payoff, payoff_diff_funs[step_idx]=payoff_fun(payoff, step_times[step_idx], state)
            if step_idx<len(step_times)-1:
                state, step_diff_funs[step_idx+1]=step_fun(state, step_times[step_idx], step_times[step_idx+1])
        acc_price+=payoff
        acc_var+=(payoff-acc_price*n_inv)**2
       # Backward sweep
        if calculate_sensitivities:
            payoff_diff=1
            state_diff=0
            for step_idx in xrange(len(step_times)-1,-1,-1):
                state1_diff, payoff_diff=payoff_diff_funs[step_idx](payoff_diff)
                state_diff += state1_diff
                if step_idx>0:
                    step_diff_s=step_diff_funs[step_idx](state_diff)
                    state_diff=step_diff_s[0]
                    for idx, val in enumerate(step_diff_s[1]):
                            sensitivities[idx]+=val
                else:
                    acc_sensitivities[0]+=state_diff
                    acc_sensitivities_var[0]+=(state_diff-acc_sensitivities[0]*n_inv)**2
                    for idx in xrange(1, len(labels_sensitivities)):
                        acc_sensitivities[idx]+=sensitivities[idx-1]
                        acc_sensitivities_var[idx]+=(sensitivities[idx-1]-acc_sensitivities[idx]*n_inv)**2
                        sensitivities[idx-1]=0
    result = OrderedDict({'price': acc_price/num_simulations,
                      'standard error price': sqrt(acc_var)/num_simulations})
    if calculate_sensitivities:
        result['sensitivities']= OrderedDict({labels_sensitivities[idx]: s/num_simulations
                                             for idx, s in enumerate(acc_sensitivities)})
        result['standard error sensitivities']=OrderedDict({labels_sensitivities[idx]: np.sqrt(s)/num_simulations
                                                          for idx, s in enumerate(acc_sensitivities_var)})
    return result
```

Listing 3: Implementation of the general Monte Carlo calculation

References

[1] A. Griewank and A. Walther, *Evaluating derivatives: Principles and techniques of algorithmic differentiation, 2nd ed.* SIAM, Philadelphia, 2008.

[2] M. Giles and P. Glasserman, “Smoking adjoints: fast Monte Carlo Greeks,” *Risk*, pp. 88–92, January 2006.

[3] L. Capriotti and M. Giles, “Fast correlation Greeks by adjoint algorithmic differentation,” *Risk*, pp. 79–83, April 2010.

[4] L. Capriotti and M. Giles, “Algorithmic differentiation: Adjoint Greeks made easy.” SSRN 1801522, April 2011.

[5] M. Broadie and P. Glasserman, “Estimating security price derivatives using simulation,” *Management Science*, vol. 42, pp. 269–285, February 1996.

[6] P. Glasserman, *Monte Carlo methods in Financial Engineering*. Springer, NY, 2003.

[7] S. Smith, “Differentiation of the Cholesky algorithm,” *Journal of Computational and Graphical Statistics*, vol. 4, no. 2, pp. 134–147, 1995.

[8] S. F. Walter, *Structured higher-order algorithmic differentiation in the forward and reverse mode with application in optimum experimental design*. PhD thesis, Humboldt-Universität zu Berlin, April 2011.