# The Efficient Application of Automatic Differentiation for Computing Gradients in Financial Applications * 

Wei Xu ${ }^{1}$, Xi Chen ${ }^{2}$ and Thomas F. Coleman ${ }^{3}$<br>${ }^{1}$ Department of Mathematics, Tongji University<br>Shanghai, China, 200092<br>${ }^{2}$ Department of Statistics and Operations Research, University of North Carolina<br>Chapel Hill, NC, USA<br>${ }^{3}$ Department of Combinatorics and Optimization<br>University of Waterloo<br>Waterloo, On. Canada, N2L 3G1.<br>${ }^{1}$ wdxu@tongji.edu.cn<br>2 vernor.pipi@gmail.com<br>3 tfcoleman@uwaterloo.ca

July 31, 2014


#### Abstract

Automatic differentiation is a practical field of computational mathematics of growing interest across many industries, including finance. Use of reverse-mode AD is particularly interesting since it allows for the computation of gradients in the same time required to evaluate the objective function itself. However, it requires excessive memory. This memory requirement can make reverse-mode AD infeasible in some cases (depending on the function complexity and available RAM) and, in others, slower than expected due to use of secondary memory and non-localized memory references. On the other hand, it turns out that many complex (expensive) functions in finance exhibit a natural "substitution structure". In this paper, we illustrate this structure in computational finance arising in calibration and inverse problems, as well as determining Greeks in a Monte Carlo setting. In these cases the required memory is a small fraction of that required by reverse-mode AD but the computing time complexity is the same. In fact, our results indicate significant realized speedup over straight reverse-mode AD.


Key words: Gradient, Automatic differentiation, Reverse-mode, Greeks, Local volatility, Calibration, Inverse Problems, Algorithmic Differentiation, Monte Carlo method.

[^0]
## 1 Introduction

Many applications in finance require the efficient computation of gradients. For example, first-derivatives of pricing functions of options (i.e., "Greeks"), are required for hedging operations. As another example, calibration of complex models typically requires gradients of a regression function. More generally, minimization of functions of groups of pricing functions requires the computation of gradients at many points. Speed and accuracy are often paramount in these applications and the new discipline of automatic differentiation $(\mathrm{AD})[21]$ offers a promising solution. Indeed, since it is known that in principle the reversemode of automatic differentiation can yield accurate gradients in time proportional to the time required to evaluate the objective function itself, it appears that a complete solution is at hand. In fact, several authors have made this connection in the context of determining Greeks in a Monte Carlo setting [7, 9, 10, 19].

In 2006, Giles and Glasserman [19] applied AD to calculate Greeks of interest rate derivatives priced by the Libor market model. Subsequently, the application of AD to compute Greeks of financial derivatives has gained the attention of academia and industry. Leclerc, Liang and Schneider [24] utilized reverse-mode AD to calculate the sensitivities of receiver Bermudan swaption with Libor market model. Their pricing framework is based on Monte Carlo simulation with Longstaff-Schwartz regression to determine the timing of early exercise while Joshi et al [27, 25, 26] applied AD with the Libor Market Model in different contexts. In [28], Kaebe at al proposed a method based on adjoint methods (reverse mode) to speed up the Monte Carlo based calibration of financial market models. The adjoint method is applied to compute derivatives efficiently, rather than the finite difference method, but the massive memory storage issue is not considered in [28]. Recently, AD was also applied to analyze the sensitivity of credit derivatives, such as CODs and CDS's, and manage the credit risk $[8,5]$. Capriotti and Giles $[9,10,6]$ discussed the computation of correlation Greeks for basket default swaps, CDOs and counterparty risk of credit portfolios through reverse-mode AD. However, the credit derivative payoff functions usually are not continuous. This discontinuity may cause biased estimators produced by the direct application of AD. Some adjustment techniques on payoff functions were proposed in $[9,10,6]$ when computing sensitivities of CDOs and CDS's. Compared with the forward mode (finite difference method), the reverse-mode is efficient from the computational complexity point of view.

However, if the problem is large or the pricing function is complicated, the reverse-mode requires massive memory storage, which leads to a large realized computational time due to increased memory access time. The entire "computational tape", storing all intermediate variables, must be saved (and subsequently accessed in reverse order). For complex functions this can represent a significant memory requirement, often rendering reverse-mode AD impractical in its pure form. Xu and Coleman [16] have proposed a space-efficient variation of reverse-mode applicable to structured problems (we define "structure" below). The purpose of this paper is to illustrate that many problems of finance involving gradient computations
are structured and hence this structured approach can be usefully applied in finance, providing a practical, accurate way to compute gradients in finance, efficient in space and time.

First we recall the following AD complexity results. Assume that $F: \mathbb{R}^{n} \rightarrow \mathbb{R}^{m}$ is a differentiable map. Let $V \in \mathbb{R}^{n \times t_{V}}$. We denote the number of operations (a measure of time) to evaluate a function $F$ at an arbitrary point, given a code for $F$, as $\omega(F)$; we denote the space used by the given code to evaluate $F$ as $\sigma(F)$. Then forward-mode AD can be applied directly to function $F$ with matrix $V$ to yield the product $J \cdot V$ in time proportional to $t_{V} \cdot \omega(F)$, and space proportional to $\sigma(F)$ [21, page 36]. Similarly, if $W \in \mathbb{R}^{m \times t_{W}}$ then reverse-mode AD can be applied directly to function $F$ with matrix $W$ to yield the product $W^{T} J$ in time proportional to $t_{W} \cdot \omega(F)$ and space proportional to $\omega(F)$ [21, page 44]. (Note that reverse-mode AD has a significantly increased demand for space over forward mode AD ; typically, $\omega(F) \gg \sigma(F)$.) When $m<n$ reverse-mode AD is faster than forward-mode, at least in principle.

A special case of computing Jacobian products is when $V=I$ and then these complexity results indicate that the Jacobian can be obtained by forward mode in time $n \cdot \omega(F)$ and space $\sigma(F)$; alternatively, if $W=I$ then the Jacobian can be determined by reverse-mode AD in time $m \cdot \omega(F)$ and space $\sigma \sim \omega(F)$. We note that the sparse Jacobian case can be more efficient in time complexity: a combination of forward and reverse-modes can be used to get the Jacobian matrix in time $\chi(J) \cdot \omega(F)$ where $\chi(J)$ is a measure of sparsity. In many cases $\chi(J) \ll \min (m, n)[12,15,31]$. In this paper we are concerned with gradient determination, and so $m=1$, and these sparsity issues have little direct bearing.

Previous work addresses the problem of the extensive memory requirements of reverse-mode $\mathrm{AD}[4,21]$. Most of this work can be described under the banner of a computer science technique known as "checkpointing" [20]. Checkpointing is a general procedure that can be defined on a computational graph of a function to evaluate and differentiate an arbitrary differentiable function $z=f(x)$, where for convenience we restrict our discussion to a scalarvalued function of $n$-variables, i.e., $f: \mathbb{R}^{n} \rightarrow \mathbb{R}$. The general idea is to cut the computational graph at a finite set of places (checkpoints) in a forward evaluation of the function $f(x)$, saving some state information at each of those checkpoints and then recompute the information between checkpoints in a backward sweep in which the derivatives are computed. The advantage is potentially considerable savings in required space (compared to a reverse-mode evaluation of the gradient which requires saving the entire computational graph). There is some computing cost - but the total cost is just a constant factor times the cost to evaluate the function (i.e., the cost to perform the forward sweep). So this checkpointing procedure is the same order of work as straight reverse-mode AD but considerably less space.

There are many ways to proceed with this general checkpointing concept described above - some involve machine/environment specific aspects, some involve user interference, and all involve making choices about where to put the checkpoints and what state information
to store [29]. Examples of checkpointing manifestations are included in [23]. Typically the user of the checkpointing idea either has to explicitly, in a "hands-on" way, decide on these questions for a particular application, or allow a checkpoint/AD tool to decide. There is no "well-accepted" automatic way that we know of to choose optimal checkpoints, and choice of checkpoint location clearly matters. The authors in [11] have done some work on this matter, with reference to determining general Jacobian matrix: one conclusion is that it is very hard to choose an ideal set of checkpoints in an automatic way, given the code to evaluate $f(x)$. Some AD software apparently does decide automatically on checkpoint location (and state variable to save) though the actual rules that are used are, in our view, somewhat opaque.

Our proposed technique in this paper can be categorized as a particular (and practical) manifestation of checkpointing. Specifically, it is illustrated that many practical applications (in finance) have a particular structure (see (2.1)) and if this structure is explicitly exposed in the function presentation then a natural set of checkpoints is evident and the set of state variables that are needed to allow for the backward sweep is also clear (and of relatively modest size).This observation is of considerable practical value. The contribution of this paper therefore has these two major dimensions: observing and illustrating that many applications in finance have this general structure (2.1), for relatively small ' $p$ ', and given that a function is presented in this structured form detailing an explicit well-defined way to deploy the checkpointing idea to compute the gradient.

The rest of this paper is organized as follows. In Section 2, we discuss the basics of automatic differentiation and this practical notion of structure and how it can yield significant efficiencies. In the subsequent Section 3 and 4 we consider two common but distinct settings in computational finance: gradient computation to help in the determination of volatility surfaces, and the Monte Carlo evaluation of options (and subsequent computation of Greeks). These are broad example settings to illustrate how structure can be used, in computational finance, to enable efficient gradient computation. Results of some numerical experiments are presented in Section 5 that show the memory and realized computational time saving through the structured gradient computation. Finally, we conclude with some observations in Section 6.

## 2 Calculating Gradients of Structured Functions

We consider the following structured form. To evaluate a scalar-valued function $f(x)$ suppose that the procedure is:

$$
\left.\begin{array}{l}
\text { Solve for } y_{1}: T_{1}(x)-y_{1}=0  \tag{2.1}\\
\text { Solve for } y_{2}: T_{2}\left(x, y_{1}\right)-y_{2}=0 \\
\vdots \\
\text { Solve for } y_{p}: T_{p}\left(x, y_{1}, y_{2}, \cdots, y_{p-1}\right)-y_{p}=0 \\
\text { Solve for } z: \bar{f}\left(x, y_{1}, y_{2}, \cdots, y_{p}\right)-z=0 .
\end{array}\right\}
$$

The intermediate values, $y_{i}, i=1, \cdots, p$ are, in general, vectors of varying length; functions $T_{i}, \bar{f}$ are differentiable functions of their arguments.

The corresponding extended Jacobian, i.e., Jacobian of (2.1) with respective to $x, y_{1}, \cdots, y_{p}$, can be written in a partitioned form

$$
J^{E}=\left(\begin{array}{c|cccc}
J_{x}^{1} & -I & & &  \tag{2.2}\\
J_{x}^{2} & J_{y_{1}}^{2} & -I & & \\
\vdots & \vdots & \vdots & \ddots & \\
J_{x}^{p} & J_{y_{1}}^{p} & \vdots & J_{y_{p-1}}^{p} & -I \\
\hline \nabla \overline{f_{x}^{T}} & \nabla \bar{f}_{y_{1}}^{T} & \cdots & \cdots & \nabla \overline{f_{y_{p}}^{T}}
\end{array}\right)=\left(\begin{array}{c|c}
\mathrm{A} & \mathrm{~B} \\
\hline \nabla \bar{f}_{x}^{T} & \nabla \bar{f}_{y}^{T}
\end{array}\right) .
$$

Then, the gradient of $f$, with respect to $x$, satisfies $[12,15]$

$$
\begin{equation*}
\nabla f^{T}=\nabla \bar{f}_{x}^{T}-\nabla \bar{f}_{y}^{T} B^{-1} A \tag{2.3}
\end{equation*}
$$

The key to this approach is this: It is not necessary to explicitly compute matrices $(A, B)$ in order to compute $\nabla f^{T}$ given by (2.3). Specifically, the off-diagonal submatrices in $(A, B)$ involved in the calculation of (2.3) occur only in a product form consistent with effective "just in time" application of reverse-mode $A D$.

To understand the algorithm given below, consider the following approach to the computation of $v^{T}=\nabla \bar{f}_{y}^{T} B^{-1} A$. We can define $w^{T}=\left(w_{1}^{T}, \cdots, w_{p}^{T}\right)$ to satisfy $w^{T}=\nabla \bar{f}_{y}^{T} B^{-1}$ in a transposed form

$$
\left(\begin{array}{cccccc}
-I & \left(J_{y_{1}}^{2}\right)^{T} & \left(J_{y_{1}}^{3}\right)^{T} & \cdots & \left(J_{y_{1}}^{p-1}\right)^{T} & \left(J_{y_{1}}^{p}\right)^{T}  \tag{2.4}\\
& -I & \left(J_{y_{2}}^{3}\right)^{T} & \cdots & \vdots & \left(J_{y_{2}}^{p}\right)^{T} \\
& & -I & \cdots & \vdots & \vdots \\
& & & \ddots & \left(J_{y_{p-2}}^{p-1}\right)^{T} & J_{y_{p_{-2}}^{p}}^{p} \\
& & & & -I & J_{y_{p}-1} \\
& & & & & -I
\end{array}\right)\left(\begin{array}{c}
w_{1} \\
w_{2} \\
w_{3} \\
\vdots \\
w_{p-1} \\
w_{p}
\end{array}\right)=\left(\begin{array}{c}
\nabla \bar{f}_{y_{1}} \\
\nabla \bar{f}_{y_{2}} \\
\nabla \bar{f}_{y_{3}} \\
\vdots \\
\nabla \bar{f}_{y_{p-1}} \\
\nabla f_{y_{p}}
\end{array}\right)
$$

and

$$
\begin{equation*}
v^{T}=\nabla \bar{f}_{y}^{T} B^{-1} A=w^{T} A=w_{1}^{T} J_{x}^{1}+w_{2}^{T} J_{x}^{2}+\cdots+w_{p-1}^{T} J_{x}^{p-1}+w_{p}^{T} J_{x}^{p} . \tag{2.5}
\end{equation*}
$$

Thus, we have the following algorithm to compute the structured gradient in (2.3).
Algorithm 1 Algorithm for Structured Gradient

1. Following (2.1) evaluate the values of $y_{1}, y_{2}, \cdots, y_{p}$ only.
2. Evaluate $z=\bar{f}\left(x, y_{1}, \cdots, y_{p}\right)$ and apply reverse-mode $A D$ to obtain $\nabla \bar{f}^{T}=\left(\nabla \bar{f}_{x}^{T}, \nabla \bar{f}_{y_{1}}^{T}, \cdots, \nabla \bar{f}_{y_{p}}^{T}\right)$.
3. Compute the gradient using (2.4).
(a) Initialize $v_{i}=0, i=1: p, \nabla f=\nabla \bar{f}_{x}^{T}$,
(b) For $j=p, p-1, \cdots, 1$
$w_{j}=\nabla \bar{f}_{y_{j}}-v_{j}$;

- Evaluate $T_{j}\left(x, y_{1}, \cdots, y_{j-1}\right)$ and apply reverse-mode $A D$ with vector $w_{j}^{T}$ to get
$w_{j}^{T} \cdot\left(J_{x}^{j}, J_{y_{1}}^{j}, \cdots J_{y_{j-1}}^{j}\right) . \operatorname{Set} v_{i}^{T}=v_{i}^{T}+w_{j}^{T} \cdot J_{y_{i}}^{j}$ for $i=1, \cdots, j-1$;
-Update $\nabla f^{T} \leftarrow \nabla f^{T}+w_{j}^{T} J_{x}^{j}$;

Note that Step 1 takes time proportional to $\sum_{i=1}^{p} \omega\left(T_{i}\right) \leq \omega(F)$ and space proportional to $\sigma(F)$ while Step 2 requires time proportional to $\omega(\bar{f})$ and space proportional to $\omega(\bar{f})$. When $\nabla f^{T}$ is updated in Step 3, its work is proportional to $\sum_{i=1}^{p} \omega\left(T_{i}\right) \leq \omega(F)$ and the required space satisfies $\sigma \leq \max \left\{\omega\left(T_{i}\right), i=1, \cdots, p\right\}$. Thus, the theoretical time required to evaluate the gradient by this structured approach is proportional to $\omega(f)$, similar to reverse-mode AD , but the space required satisfies $\sigma \leq \max \left\{\omega\left(T_{i}\right), i=1, \cdots, p, \omega(\bar{f})\right\}$, which compares very favorably to $\sum_{i=1}^{p} \omega\left(T_{i}\right)+\omega(\bar{f})$, the space required by reverse-mode AD.

This greater space efficiency not only implies that our proposed method will apply to a wider set of problems (i.e., before exhausting all available RAM) compared to direct application of reverse-mode AD, but also results in better realized running times, due to localized memory reference. The results in Section 5 support this claim.

## 3 Nonlinear inverse problems, calibration, volatility surfaces

The problem of estimating a volatility surface, given a set of option prices on the same underlying, is an important and well-studied problem [13]. It is, in style, an engineering design problem formulated as a nonlinear inverse problem. Such problems are usually expressed as minimization of a regression function; most numerical approaches to such problems involve the determination of the gradient of the regression function. The AD technique for gradients of structured functions come into play here and can yield very efficient gradient determination. We note that these problems are often large and expensive because of the required approximation of a PDE model. Calculation of the gradient can be a non-trivial, expensive task.

The general discretized setting can be described as follows. Let $x \in \mathbb{R}^{n}$ be the vector of problem parameters, sometimes called control variables, and now (temporarily) assume that we have fixed the parameters $x=\bar{x}$ for some specified values $\bar{x}$. Suppose $y \in \mathbb{R}^{p}$ is the vector representing the state of the system given the parameters $x=\bar{x}$. Specifically, $y$ is the solution to the forward problem,

$$
\begin{equation*}
\text { solve for } y: \quad \tilde{F}(\bar{x}, y)=0 \tag{3.6}
\end{equation*}
$$

Typically function $\tilde{F}$ is a square continuously differentiable mapping, sending $p$-vectors $y$ to $p$-vectors $\tilde{F}$; function $\tilde{F}$ usually represents a discrete approximation to a continuous PDE formulation.

The inverse problem flips (3.6) around. Specifically, given a target state $y_{T}$, where $y_{T}$ is the compressed vector of a designated subset of the components of state vector $y$, the inverse problem addresses the question: what value for the parameters $x$ yield $y_{T}$ through forward process (3.6)? Generally (3.6) cannot be satisfied exactly for the target vector $y_{T}$ and so some "measure of nearness" is usually minimized in order to determine the control variables $x$. The usual "measure of nearness" is the Euclidean norm, in which case the inverse problem becomes:

$$
\begin{equation*}
\min _{x} f(x) \equiv \frac{1}{2}\|F(x)\|_{2}^{2}, \tag{3.7}
\end{equation*}
$$

where $F(x)=y_{S}(x)-y_{T}, y_{S}(x)$ is the compressed vector of the components of $y(x)$ corresponding to target components $y_{T}$, and $y(x)$ is implicitly defined through (3.6). Hence to evaluate the function $f(x)$ the following steps can be used:

$$
\left.\begin{array}{l}
\text { 1. Solve for } y: \tilde{F}(x, y)=0  \tag{3.8}\\
\text { 2. Solve for } v: y_{S}-y_{T}-v=0 \\
\text { 3. Solve for output } z: \frac{1}{2}\|v\|_{2}^{2}-z=0 \text {. }
\end{array}\right\} \text {. }
$$

A key computational observation is that for many inverse problems function $\tilde{F}$ is a discrete approximation to an underlying PDE and often yields solutions $y$ in a piece-by-piece structured manner. For example, some special time-stepping PDE methods would generate a sequence of vectors: $y_{i}=T_{i}\left(x, y_{i-1}\right), i=1, \cdots, p$, where vector $y_{S}$ is often the final vector in the sequence, i.e., $y_{S}=y_{p}$. It is clear then that form (2.1) is obtained. Occasionally (3.7) is augmented by a smoothing term $\operatorname{sm}(x)$, and then (3.7) takes the form

$$
\begin{equation*}
\min _{x} f(x) \equiv \frac{1}{2}\|F(x)\|_{2}^{2}+\lambda \cdot \operatorname{sm}(x) \tag{3.9}
\end{equation*}
$$

for a parameter $\lambda$ chosen to balance smoothness and data satisfaction ${ }^{1}$. The method above (3.8) is applicable in this case also, with the last step modified:

$$
3^{\prime} \quad \text { Solve for output } z: \frac{1}{2}\|v\|_{2}^{2}+\lambda \cdot \operatorname{sm}(x)-z=0
$$

So, for example, if the inverse problem involved a time-stepping PDE approximation as alluded to above, and included a smoothing term, then the structured form for the objective

[^1]function $z=f(x)$ is:


Note that (3.10) is a special case of the general structure given in (2.1) and so the Algorithm for Structured Gradient can easily be adapted to efficiently compute the gradient (efficient in space and time).

### 3.1 A Cluster of Inverse Problems

Inverse problems can arise from a cluster of forward problems. That is, suppose we have a cluster of forward problems, replacing (3.6),

$$
\text { Solve for } y_{i} \quad \tilde{F}_{i}\left(\bar{x}, y_{i}\right)=0, \quad i=1, \cdots, r,
$$

where each $\tilde{F}_{i}$ is a square mapping sending $p_{i}$-vectors $y_{i}$ to $p_{i}$-vectors $\tilde{F}_{i}\left(\bar{x}, y_{i}\right)$. Usually the functions $\tilde{F}_{i}$ are closely related to each other (e.g. approximations to the same PDE with different initial parameters). Note that the control variables $\bar{x} \in \mathbb{R}^{n}$ are the same for each problem in the cluster.

The inverse problem for the cluster case is very similar to the single function case. Denote $y(x)=\left(y_{1}(x), \cdots, y_{r}(x)\right)^{T}$ and let $y_{T}$ be a compressed vector of target values for a designated subset of the components ${ }^{2}$ of $y(x)$. We denote the designated components of $y$ by $y(x)$. The cluster inverse problem can be stated as (3.7) where $F(x)=y_{S}(x)-y_{T}$. The function to evaluate $z=f(x)$ is

$$
\left.\begin{array}{l}
\text { 1. Determine } y_{S}(x) \\
\quad \text { for } i=r:-1: 1 \text {, Solve for } y_{i}: \tilde{F}_{i}\left(x, y_{i}\right)=0  \tag{3.11}\\
\text { 2. Solve for } v: y_{S}(x)-y_{T}-v=0
\end{array}\right\}
$$

Clearly if each inverse problem is solved by, for example, a time-stepping PDE numerical technique, then step 1 involves a repetition of the structure given in (3.9). Hence the cluster inverse problem expression is within the family defined by (2.1) and the Algorithm Structured Gradient can be applied to compute the gradient in an efficient manner.

[^2]
### 3.2 The implied volatility problem

The implied volatility surface problem follows this cluster inverse pattern. The volatility surface problem is to determine a volatility surface, over price and (future) time, that reflects the expected volatility behavior of an underlying asset. The given data are the prices of some currently traded options on the underlying asset. Under various standard assumptions, such as the underlying $S$ follows a 1 -factor continuous diffusion equation, the value of the European option satisfies the generalized Black-Scholes equation:

$$
\begin{equation*}
\frac{\partial V}{\partial t}+\frac{1}{2} \sigma^{2}(S, t) S^{2} \frac{\partial^{2} V}{\partial S^{2}}+(r-q) S \frac{\partial V}{\partial S}-r V=0,0 \leq S<\infty, 0 \leq t \leq T, \tag{3.12}
\end{equation*}
$$

where $r$ is the risk-free interest rate, $q$ is the dividend rate and $\sigma(S, t)$ is the volatility at the underlying price $S$ and time $t$. The final condition for the European call option is:

$$
V(S, T)=\max (S-K, 0)
$$

When $S=0$, the value of the option is set to zero, that is $V(0, t)=0, t \in[0, T]$. Our inverse challenge is to infer the smooth surface $\sigma(s, t)$ - which after discretization using an $N$-by- $M$ grid, becomes $n=N \cdot M$ values, i.e., our control vector $x \in \mathbb{R}^{n}$. Given (3.9), known the final condition, and a set of known option values (i.e., known solutions to (3.12)) standard numerical PDE techniques, such as the Crank-Nicholson finite-difference scheme yield a time stepping structure as indicated above. The result is a structure consistent with the cluster inverse problem structure (3.11). Therefore, Algorithm Structured Gradient can be applied to efficiently compute the gradient.

Computationally, this problem can be solved using a discretization of spatial and time domains and employing a Crank-Nicholson finite difference scheme. For example, we can use following discretization

$$
\begin{array}{ll}
S_{i}=S_{\text {min }}+\Delta S \cdot i, & i=0, \cdots, N-1 ; \\
t_{j}=j \cdot \frac{T}{M-1}, & j=0, \cdots, M-1,
\end{array}
$$

where we assume $S_{i} \in\left[S_{\min }, S_{\max }\right], \Delta S=\left(S_{\max }-S_{\min }\right) /(N-1)$ and $t \in[0, T]$. Theoretically, $S_{\text {min }}=0$ and $S_{\max }=+\infty$. However, given a period of optional data, the stock price never goes from zero to infinity. Thus, in practice, $\left[S_{\min }, S_{\max }\right]$ represents the prices variation range during the given period. For example, we set $S_{\text {min }}=0.8 S_{0}$ and $S_{\max }=1.2 S_{0}$ in our numerical experiments in Section 5 where $S_{0}$ is the initial stock price. Then, the Crank-Nicholson finite difference method has the following form

$$
\begin{align*}
& \frac{\partial V}{\partial t}=\frac{V_{i, j+1}-V_{i, j}}{\Delta t},  \tag{3.13}\\
& \frac{\partial V}{\partial S}=\frac{1}{4 \Delta S}\left(V_{i+1, j}-V_{i-1, j}+V_{i+1, j+1}-V_{i-1, j+1}\right) \\
& \frac{\partial^{2} V}{\partial S^{2}}=\frac{1}{2 \Delta S^{2}}\left(V_{i+1, j}-2 V_{i, j}+V_{i-1, j}+V_{i+1, j+1}-2 V_{i, j+1}+V_{i-1, j+1}\right),
\end{align*}
$$

where $V_{i, j} \equiv V\left(S_{i}, t_{j}\right)$. Thus, substituting (3.13) into (3.12), we obtain

$$
\begin{aligned}
r V_{i, j}= & \frac{V_{i, j+1}-V_{i, j}}{\Delta \Delta}+(r-q) S_{i} \frac{1}{4 \Delta S}\left(V_{i+1, j}-V_{i-1, j}+V_{i+1, j+1}-V_{i-1, j+1}\right) \\
& +0.5 \sigma^{2}\left(S_{i}, t_{j}\right) S_{i}^{2} \cdot \frac{1}{2 \Delta S^{2}}\left(V_{i+1, j}-2 V_{i, j}+V_{i-1, j}+V_{i+1, j+1}-2 V_{i, j+1}+V_{i-1, j+1}\right),
\end{aligned}
$$

which can be rewritten as

$$
A_{i-1, j+1} V_{i-1, j+1}+B_{i, j+1} V_{i, j+1}+C_{i+1, j+1} V_{i, j+1}=\tilde{A}_{i-1, j} V_{i-1, j}+\tilde{B}_{i, j} V_{i, j}+\tilde{C}_{i+1, j} V_{i+1, j}
$$

where

$$
\begin{cases}A_{i-1, j+1} & =-\frac{\Delta t}{4 \Delta S}(r-q) S_{i}+\frac{\Delta t}{4 \Delta S^{2}} \sigma^{2}\left(S_{i}, t_{j}\right) S_{i}^{2} \\ B_{i, j+1} & =1-\frac{\Delta t}{\Delta S^{2}} \sigma^{2}\left(S_{i}, t_{j}\right) S_{i}^{2} \\ C_{i+1, j+1} & =\frac{\Delta t}{4 \Delta S}(r-q) S_{i}+\frac{\Delta t}{4 \Delta S^{2}} \sigma^{2}\left(S_{i}, t_{j}\right) S_{i}^{2} \\ \tilde{A}_{i-1, j} & =\frac{\Delta t}{4 \Delta S}(r-q) S_{i}-\frac{\Delta t}{4 \Delta S_{i}^{2}} \sigma^{2}\left(S_{i}, t_{j}\right) S_{i}^{2} \\ \tilde{B}_{i, j} & =1+\Delta t r+\frac{\Delta t}{2 \Delta S^{2}} \sigma^{2}\left(S_{i}, t_{j}\right) S_{i}^{2} \\ \tilde{C}_{i+1, j} & =-\frac{\Delta t}{4 \Delta S}(r-q) S_{i}-\frac{\Delta t}{4 \Delta S^{2}} \sigma^{2}\left(S_{i}, t_{j}\right) S_{i}^{2}\end{cases}
$$

With the boundary and final conditions, we have

$$
\begin{aligned}
& V_{0, j}=0, j=0, \cdots, M-1 \\
& V_{i, M-1}=\max \left\{S_{i}-K, 0\right\}, i=0, \cdots, N-1
\end{aligned}
$$

Hence, the value $V_{i, j}$ at time $t_{j}$ can be derived from values at $t_{j+1}$ as,

$$
\tilde{F}_{i}\left(\sigma\left(S_{i}, t_{j}\right), \mathbf{y}_{j+1}, \mathbf{y}_{j}\right) \equiv \tilde{\mathbf{A}}_{j}^{-1} \mathbf{A}_{j+1} \mathbf{y}_{j+1}-\mathbf{y}_{j}=0
$$

where $\mathbf{y}_{j}=\left[V_{1, j}, \cdots, V_{N-1, j}\right]^{T}$,

$$
\mathbf{A}_{j+1}=\left(\begin{array}{ccccc}
B_{1, j+1} & C_{2, j+1} & & & \\
A_{1, j+1} & B_{2, j+1} & C_{3, j+1} & & \\
& \ddots & \ddots & \ddots & \\
& & \ddots & \ddots & C_{N, j+1} \\
& & & A_{N-1, j+1} & B_{N-1, j+1}
\end{array}\right)
$$

and

$$
\tilde{\mathbf{A}}_{j}=\left(\begin{array}{ccccc}
\tilde{B}_{1, j} & \tilde{C}_{2, j} & & & \\
\tilde{A}_{1, j} & \tilde{B}_{2, j} & \tilde{C}_{3, j} & & \\
& \ddots & \ddots & \ddots & \\
& & \ddots & \ddots & \tilde{C}_{N, j} \\
& & & \tilde{A}_{N-1, j} & \tilde{B}_{N-1, j}
\end{array}\right)
$$

Then, the structure process (3.11) can be rewritten as

1. Solve for $\mathbf{y}_{M-1}: \max \{\mathbf{S}-K, 0\}-\mathbf{y}_{M-1}=0$
2. Determine $\mathbf{y}_{S}$
for $j=M-2:-1: 0$, Solve for $\mathbf{y}_{i}: \tilde{F}_{j}\left(\sigma\left(S_{i}, t_{j}\right), \mathbf{y}_{j+1}, \mathbf{y}_{j}\right)=0$
3. Solve for $v: \mathbf{y}_{S}(x)-\mathbf{y}_{T}-v=0$
4. Solve for output $z: \frac{1}{2}\|v\|_{2}^{2}-z=0$.

The corresponding extended Jacobian matrix $J^{E}$ for (3.11) is in the form of

$$
J^{E}=\left(\begin{array}{c|ccccc}
0 & -I & & & & \\
\frac{\partial \tilde{F}_{M-2}}{\partial \sigma} & \tilde{\mathbf{A}}_{M-2}^{-1} \mathbf{A}_{M-1} & -I & & & \\
\vdots & & \ddots & \ddots & & \\
\frac{\partial \tilde{F}_{0}}{\partial \sigma} & & & \tilde{\mathbf{A}}_{0}^{-1} \mathbf{A}_{1} & -I & \\
\frac{\partial y_{S}}{\partial \sigma} & & & & 0 & -I \\
\hline 0 & 0 & \cdots & \cdots & \cdots & v^{T}
\end{array}\right) .
$$

### 3.3 Using Splines for Smoothness and Variable Reduction

The Coleman et al method [13] assumes a spline form to model the unknown volatility surface. The set-up is similar to that defined above except that the "control vector $x$ " is a small set of knot values for the cubic spline representation of the volatility surface. Therefore once those values are assigned the entire volatility surface is determined (assuming boundary values). Therefore we can write $x x=c(x)$ where $x x$ is the vector of $N \cdot M$ values of the volatility surface at the grid points. So in this case $f=f(x)$ is a function of a handful of variables, but the computation of $f$ is similar to (3.9):

$$
\left.\begin{array}{l}
\text { Solve for } x x: c(x)-x x=0  \tag{3.14}\\
\text { Solve for } y_{1}: T_{1}(x x)-y_{1}=0 \\
\text { Solve for } y_{2}: T_{2}\left(x x, y_{1}\right)-y_{2}=0 \\
\vdots \\
\text { Solve for } y_{p}: T_{p}\left(x x, y_{p-1}\right)-y_{p}=0 \\
\text { Solve for } v: y_{S}-y_{T}-v=0 \\
\text { Solve for } z: \frac{1}{2}\|v\|_{2}^{2}+\lambda \cdot \operatorname{sm}(x)-z=0 . .
\end{array}\right\}
$$

The cluster idea presented above applies here essentially without change. Again the structure of in this case is subsumed by (2.1) and the efficient Algorithm Structured Gradient can be used to compute $\nabla f(x)$.

In this case, the discretization of $\operatorname{PDE}(3.12)$ is the same as the one in previous subsection, but we introduce a cubic spline to generate all knot values on $x x$ based on knots $\sigma$. The smoothness term, $\operatorname{sm}(\sigma)$ has the form

$$
s m(\sigma)=\int_{u}^{v} \int_{a}^{b}\left(\frac{d^{2} c(S, t, \sigma)}{d s d t}\right)^{2} d s d t
$$

where $c(S, t, \sigma)$ is the cubic spline surface constructed by $\sigma$ at point $(S, t)$ and $a, b, u$ and $v$ can be any specified grid to measure the smoothness of the surface. Thus, the corresponding
extended Jacobian matrix $J^{E}$ of (3.14) is in the form of

$$
J^{E}=\left(\begin{array}{c|cccccc}
\frac{\partial c}{\partial \sigma} & -I & & & & & \\
0 & \frac{\partial \tilde{F}_{M-1}}{\partial x x} & -I & & & & \\
0 & \frac{\partial F_{M-2}}{\partial x x} & \tilde{\mathbf{A}}_{M-2}^{-1} \mathbf{A}_{M-1} & -I & & & \\
\vdots & \vdots & & \ddots & \ddots & & \\
\vdots & \frac{\partial \tilde{F}_{0}}{\partial x x} & & & \tilde{\mathbf{A}}_{0}^{-1} \mathbf{A}_{1} & -I & \\
0 & \frac{\partial \mathbf{y}_{S}}{\partial x x} & & & & 0 & -I \\
\hline 0 & 0 & 0 & \cdots & \cdots & \cdots & v^{T}
\end{array}\right) .
$$

## 4 Calculation of First Derivatives ("Greeks") in a Monte Carlo Setting

The structured form (2.1) is broadly applicable and covers another important application in computational finance: the evaluation of a pricing function (and its Greeks) by Monte Carlo method. In this section we show that Algorithm Structured Gradient can be tailored to this problem to compute gradients (Greeks), efficiently in space and time.

First we briefly review the theoretical foundation of path-wise derivatives. The price of a financial derivative is expressed as the expectation of the payoff under risk neutral measure, where:

$$
P=\mathcal{E}(V(S))
$$

Here $P$ is the price, and $V(S)$ is the payoff function of the value of the underlying asset at maturity. The asset value $S$ can be further expressed as a function of initial parameters and random variables, i.e.:

$$
S=g(x, Z)
$$

where $x$ is the vector of initial parameters and $Z$ is the random innovation that drives the price. Parameters $x$ could include initial value of $S$, volatility of $S$, risk free interest rate or any other deterministic parameters that can influence the evolution of $S$. Hence, $P$ can be often expressed as the integral:

$$
P=\int V(g(x, Z)) \rho(Z) d Z
$$

where $\rho(Z)$ is the probability density function of $Z$. Here $\rho(Z)$ is not influenced by the parameters $x$. If the function $V(\cdot)$ satisfies certain regularity conditions (please refer to [1] for a detailed discussion of the regularity conditions), we can interchange the order of integration and differentiation to obtain the "path-wise derivatives":

$$
\frac{\partial P}{\partial x}=\frac{\partial}{\partial x} \int V(g(x, Z)) \rho(Z) d Z=\int \frac{\partial V(g(x, Z))}{\partial x} \rho(Z) d Z
$$



Figure 1: Computation structure of the option pricing by Monte Carlo method.

The above formula can also be expressed as:

$$
\frac{\partial P}{\partial x}=\frac{\partial \mathcal{E}(V(S))}{\partial x}=\mathcal{E}\left(\frac{\partial V(S)}{\partial x}\right) .
$$

The above integral can be estimated using Monte Carlo simulation, assuming $p$ simulations:

$$
\frac{\widehat{\partial P}}{\partial x}=\frac{1}{p} \sum_{k=1}^{p} \frac{\partial V\left(g\left(x, Z_{k}\right)\right)}{\partial x} .
$$

Note that random innovation $Z$ is now broken down to $p$ discrete vectors $Z_{k}$, each vector corresponding to a simulation path. So if the simulation consists of $q$ time steps then $Z_{k} \in \mathbb{R}^{q}$. Correspondingly, the Monte Carlo estimator for the price is:

$$
\hat{P}=\frac{1}{p} \sum_{k=1}^{p} V\left(g\left(x, Z_{k}\right)\right) \equiv \frac{1}{p} \sum_{k=1}^{p} \hat{P}_{k},
$$

We illustrate the structure of the computation below in Figure 1, where $\hat{P}_{k}$ represents simulation path corresponding to innovation $Z_{k} \in \mathbb{R}^{q}$. Due to the characteristics of the Monte Carlo method, the derivatives of the price $\hat{P}$ can be estimated pathwise. Thus, our general structure idea is simplified.

It is easy to see that structure (2.1) applies to the evaluation of $f(x) \equiv \hat{P}(x)$, where $y_{k}=T_{k}\left(x, y_{1}, \cdots, y_{k-1}\right)$ in (2.1) corresponds to path $\hat{P}_{k}$ with $\left.y_{k}=V\left(g\left(x, Z_{k}\right)\right)\right)$. Note that
vector $y_{k}$ has no dependence on vector $y_{j}, j<k$ in this case. Summarizing, (2.1) reduces to:

$$
\left.\begin{array}{l}
\text { Solve for } y_{1}: T_{1}(x)-y_{1}=0  \tag{4.15}\\
\text { Solve for } y_{2}: T_{2}(x)-y_{2}=0 \\
\vdots \\
\text { Solve for } y_{p}: T_{p}(x)-y_{p}=0 \\
\text { Solve for } z: \bar{f}\left(x, y_{1}, y_{2}, \cdots, y_{p}\right)-z=0
\end{array}\right\}
$$

where $T_{k}(x) \equiv \hat{P}_{k}, k=1, \cdots p$. This special case of the structured form (2.1), where the transition functions $T_{k}$ depend only on the control variables $x$, and not on any previous state vector $y_{j}, j<k$, is known as a generalized partially separable function. Many functions in science and engineering exhibit partial separability. When $\bar{f}\left(x, y_{1}, \cdots, y_{p}\right)$ is a simple averaging computation, i.e., $\bar{f}\left(x, y_{1}, \cdots, y_{p}\right)=\frac{1}{p} \sum_{k=1}^{p} y_{k}$ then we call this a partially separable function. Specifically, the corresponding extended Jacobian matrix $J^{E}(2.2)$ can be written as

$$
J^{E}=\left(\begin{array}{ccccc}
J_{x}^{1} & -I & & & \\
J_{x}^{2} & & -I & & \\
\vdots & & & \ddots & \\
J_{x}^{p} & & \vdots & & -I \\
0 & \nabla \bar{f}_{y_{1}}^{T} & \cdots & \cdots & \nabla \bar{f}_{y_{p}}^{T}
\end{array}\right)
$$

The gradient is given by (2.3), which in generalized partially separable case reduces to:

$$
\nabla f^{T}(x)=\sum_{i=1}^{p} \nabla \bar{f}_{y_{i}}^{T} \nabla J_{x}^{i}
$$

The Algorithm for Structured Gradient can be reduced as follows.
Algorithm 2 Algorithm for Compute-GPS-Gradient (C-GPS-G)

1. Evaluate $y_{i}=T_{i}(x), i=1, \cdots, p$.
2. Evaluate and differentiate $\bar{f}\left(y_{1}, \cdots, y_{p}\right)$ using reverse-mode $A D$ to get $w_{i}=\nabla \bar{f}_{y_{i}}, i=$ $1, \cdots, p$.
3. Compute by reverse-mode $A D: v_{i}^{T}=w_{i}^{T} J_{i}$, where $J_{i}$ is the Jacobian of $T_{i}(x), i=$ $1, \cdots, p$. Set $\nabla f(x) \leftarrow \sum_{i=1}^{p} v_{i}$.

So Algorithm for Compute-GPS-Gradient calculates the gradient of a (generalized) partially separable function in time proportional to $\omega(f)$ and space $\sigma$ satisfying $\sigma \sim \max \left\{\omega(\bar{f}), \omega\left(T_{i}(x)\right), i=\right.$ $1, \cdots, p)\}$. This contrasts with direct application of reverse-mode AD which has the same theoretical time complexity but requires memory $\sigma$ satisfying $\sigma \sim \omega(f(x)) \sim \sum_{i=1}^{p} \omega\left(T_{i}(x)\right)+$
$\omega(\bar{f})$. So for large $p$, Algorithm for Compute-GPS-Gradient will require significantly less memory. Specifically, for large $p$

$$
\omega(f) \gg \max \left\{\omega(\bar{f}), \omega\left(T_{i}(x)\right), i=1, \cdots, p\right\} .
$$

Note that it is not necessary to follow any particular order over the indices $i, i=1, \cdots, p$ when updating $\nabla f$. When $\bar{f}\left(x, y_{1}, \cdots, y_{p}\right) \equiv \frac{1}{p} \sum_{k=1}^{p} y_{k}$ further simplifications to the gradient algorithm can be made. Specifically, in this case it is know that $\nabla \bar{f}_{y_{i}}=1 / p$ and so steps 1,2 in Compute-GPS-Gradient are not needed and so the simplified gradient algorithm becomes:
Algorithm 3 Algorithm for Compute-PS-Gradient (C-PS-G)

1. Let $\nabla f \leftarrow 0$
2. for $k=1: p$
3. Compute by reverse-mode AD applied to function $T_{i}$ with "matrix" $1, v_{i}^{T}=1 \cdot J_{x}^{i}$
4. $\quad \nabla f \leftarrow \nabla f+v_{i} / p$.

Note that if the memory requirement for the reverse mode in Step 3 is large, our general structure idea can be used to compute the vector $v_{i}^{T}$ for $T_{i}$ as well so that the memory can be saved significantly.

There are a number of problem-dependent ways in which the basic gradient computation methods, given above in Algorithm for C-PS-G and C-GPS-G can be further improved in speed. First there is pruning. Pruning is based on the following observation: payoffs of options are usually of the following form:

$$
\left[S_{T}-K\right]^{+} \quad \text { or } \quad\left[K-S_{T}\right]^{+} .
$$

Here $S_{T}$ is the value of the underlying asset at time $T$ and $K$ is the strike price. As discussed in the previous section, the path-wise derivative estimator has the form:

$$
\frac{1}{p} \sum_{k=1}^{p} \frac{\partial V\left(g\left(x, Z_{k}\right)\right)}{\partial x} \mathbf{1}_{\left\{S_{T_{k}}>K\right\}}
$$

where $\mathbf{1}_{\left\{S_{T_{k}}>K\right\}}$ is the indicator function that flags if the value of the underlying asset is bigger than $K$ at time $T$. Due to this indicator function, $\frac{\partial V\left(g\left(x, Z_{k}\right)\right)}{\partial x} \mathbf{1}_{\left\{S_{T_{k}}>K\right\}}$ will be zero regardless of the value of $\frac{\partial V\left(g\left(x, Z_{k}\right)\right)}{\partial x}$ if $S_{T}<K$. This property implies that there is no need to generate the value of $\frac{\partial V\left(g\left(x, Z_{k}\right)\right)}{\partial x}$ in the case of $S_{T}<K$. We need only set the value of $\frac{\partial V\left(g\left(x, Z_{k}\right)\right)}{\partial x} \boldsymbol{1}_{\left\{S_{T_{k}}>K\right\}}$ to zero. For reverse-mode AD, it is natural to do this since before executing the reverse sweep we already have information about the final price. We can simply omit the reverse sweep and set the value to zero. In contrast, forward mode AD will have
to generate the value of $\frac{\partial V\left(g\left(x, Z_{k}\right)\right)}{\partial x}$.

Due to this additional structure, reverse-mode AD can save significant calculation time compared with the forward mode. This comparison will be especially sharp for options that are deep out-of-the-money. Moreover, due to the put-call parity, we can further employ this property for options that are deep in-the-money. Put-call parity is the following relationship:

$$
C_{t}+K e^{-r(T-t)}=P_{t}+S_{t}
$$

Here $C_{t}$ and $P_{t}$ are the prices of call and put options at time $t . K$ is the strike price and $r$ is the continuous interest rate.

To illustrate, consider the computation of vega. If we differentiate the above equation on both sides with respect to $\sigma$, we have

$$
\frac{\partial C_{t}}{\partial \sigma}=\frac{\partial P_{t}}{\partial \sigma}
$$

since $\frac{\partial\left(K e^{-r t}\right)}{\partial \sigma}=\frac{\partial S_{t}}{\partial \sigma}=0$. Suppose we want to evaluate vega for a call option that is deep in-the-money with reverse-mode AD. The indicator function $\mathbf{1}_{\left\{S_{T_{k}}>K\right\}}$ has a significant probability of being non-zero. This means that we would have to perform the reverse sweep most of the time. However, due to (3.9), we can evaluate the vega of the put option and then induce vega for the call option. The put option will be deep out-of-the-money and reverse sweep can be omitted most of the time.

By rearranging the order of forward sweep and reverse sweep, this structure further allows us to save space for the reverse-mode AD. For the forward sweep to determine the payoff of the option, we only save the random numbers generating the path of the price and do not store all the other intermediate variables. Then only the random numbers with options in-the-money will be kept. After going through all the random paths, we perform an additional forward sweep for the paths that are in-the-money to generate the intermediate variables again and then perform the reverse sweep to get the path-wise estimators for the Greeks. For options that are out-of-the-money, the additional backward sweep will only represent marginal additional computation but will produce considerable space savings. For options that are deep in-the-money, we can utilize the put-call parity mentioned in the previous paragraph to avoid much needless calculation, and save space at the same time.

## 5 Numerical Experiments

In this section, results of numerical experiments are presented with respect to the structured financial cases discussed in previous sections. All experiments are carried out on a machine with Intel Core i5 2.3GHz CPU, 8GB memory and 1TB hard drive running Matlab R2011b under Windows 7 Professional. The AD toolbox for the reverse-mode is ADMAT 2.0 [1].

| Maturity $\backslash$ Strike | $85 \%$ | $90 \%$ | $95 \%$ | $100 \%$ | $105 \%$ | $110 \%$ | $115 \%$ | $120 \%$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0.695 | 101.9 | 76.26 | 52.76 | 32.75 | 16.47 | 6.02 | 1.93 | 0.62 |
| 1 | 108 | 83.6 | 61.55 | 41.57 | 25.41 | 12.75 | 5.5 | 2.13 |
| 1.5 | 117.2 | 94.37 | 73.14 | 53.97 | 37.33 | 23.68 | 14.3 | 7.65 |

Table 1: S\&P index European call option prices on October 1995 with strike price in the percentage of spot price.

Three typical experiments are considered in this section, the volatility inverse problem with and without smoothness term and Greeks for an interest rate derivative priced by the Libor market model. Note that the sparsity of the function $T_{i}$ itself is not considered in our experiments. A tape is used in the reverse-mode AD to record all intermediate variables in the function evaluation and gradient computation. In our experiments, the available fast memory is set to 500 MB for the reverse-mode AD, similar to the setting in ADOLC [2]. The purpose for this limit is to avoid affecting the performance of other applications on the machine during running the reverse-mode since it may use up all available fast memory as the tape grows. In other words, if the memory usage for the reverse-mode computation is more than 500 MB , only a chunk of the tape will be kept in memory while the rest will be written onto the hard drive. In the experiments, we use Matlab command matfile to read from and write to the hard drive. The times for accessing the hard drive depends on the times of total used memory and memory limit in the reverse sweep.

## a. Local volatility surface construction

In this example, we assume that a set of European option prices of an underlying asset is given, then we try to calibrate the volatility surface $\sigma(S, t)$ based on the PDE model (3.12) via the Crank-Nicholson finite difference method. Here we adapt S\&P 500 market option price data for S\&P 500 index options on October 1995 in Table 1. This data is also used in $[3,13,14]$. On this day, the S\&P500 index value $S_{0}=\$ 590$, interest rate $r=6 \%$ and dividend rate $q=2.62 \%$. Then, the option prices are list in Table 1. Thus, we set the direct grid for (3.12) as $\left[0.8 S_{0}, 1.2 S_{0}\right] \times[0, T]$. Table 2 records the memory and realized computational times of the gradient computation (3.7) based on the direct reverse-mode and structure gradient idea with various $N \times M$. The results in Table 2 show that the structure gradient computation saves the memory requirement (i.e., the length of whole tape) significantly. When the memory requirement of the direct reverse-mode is under the fast memory limit, the realized computational times for both methods are close. However, the direct reverse-mode takes much more computational times than the structure gradient method when running out of the fast memory.

## b. Local volatility surface construction with smoothness term

In this problem, we will use the same data set used in the previous experiments, but we

|  | Reverse-Mode |  | Structured Gradient |  | tape | time |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $N \times M$ | Tape $(\mathrm{MB})$ | Time $(\mathrm{s})$ | Tape $(\mathrm{MB})$ | Time $(\mathrm{s})$ | ratio | speedup |
| $20 \times 100$ | 13.02 | 1.17 | 0.17 | 1.20 | 76.58 | 0.975 |
| $20 \times 300$ | 57.54 | 3.60 | 0.32 | 3.52 | 179.8 | 1.02 |
| $50 \times 500$ | 350.92 | 7.27 | 1.26 | 7.12 | 278.50 | 1.02 |
| $50 \times 600$ | 466.93 | 8.72 | 1.43 | 8.68 | 325.87 | 1.00 |
| $100 \times 500$ | 907.71 | 35.48 | 2.95 | 8.70 | 307.70 | 4.07 |
| $100 \times 600$ | 1181.03 | 53.57 | 3.73 | 10.71 | 316.63 | 5.00 |
| $150 \times 500$ | 1692.90 | 75.59 | 5.08 | 13.09 | 333.24 | 5.77 |
| $150 \times 600$ | 2169.05 | 158.74 | 6.21 | 20.71 | 349.28 | 7.65 |

Table 2: Length of tape and running times of the gradient computation based on the structure and direct reverse-mode AD for the local volatility surface construction problem.
introduce the cubic spline and the smoothness term into the objective function. The cubic spline idea is introduced to emphasize smoothness in the solution. Rather than estimating the volatility surface $\sigma(S, t)$ at each $S_{i}$ and $t_{j}$ in the whole discritization grid, we only estimate a few spline knots value of $\sigma(S, t)$ while other grid values are estimated as a function of cubic spline of these knots. An advantage of this idea is to reduce the number of variables, $\sigma(S, t)$, from $N \times M$ to a small fixed number. Determining the number of spline knots itself is tricky. Thus, in order to further balance the error and the smoothness of the surface, we introduce a smoothness term in the objective function in (3.9). Table 3 records the memory and realized computational times with the number of spline knots equals $10 \times 10$ and various $N \times M$. Table 4 records the memory and realized computational times with various numbers of spline knots and $N \times M=100 \times 500$. In Table 3 , similar results are given as in Table 2 . The structured gradient computation saves memory and realized computing time, especially when fast memory runs out. From Table 4, it demonstrates that as the number of spline knots increases, the memory and computational time of the structured gradient computation increase very slowly while the direct reverse-mode increases dynamically. The reason for this dynamic increase is due to the number of access to the virtual memory on hard drive. For the $30 \times 30$ case, the memory requirement is about 983 MB , so it only needs to access the virtual memory once while it requires to access the virtual memory twice in the $50 \times 50$ case. Thus, it leads a dynamic increase in the computational time.

## c. Interest rate derivatives priced by Libor market model

The Libor market model (LMM) is very important for pricing interest rate derivatives. Because its joint distribution has no closed form density function, pricing with LMM must be through a multi-step simulation. Consequently, the path-wise derivative regarding LMM is also based on a recursive formula. The derived formula can be very complicated and is not practical for implementation, which leaves great space for AD. Based on terminal mea-

|  | Reverse-Mode |  | Structured Gradient |  | tape | time |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $N \times M$ | Tape (MB) | Time (s) | Tape (MB) | Time (s) | ratio | speedup |
| $20 \times 200$ | 67.24 | 2.74 | 2.97 | 2.72 | 22.63 | 1.01 |
| $20 \times 500$ | 164.80 | 6.37 | 6.51 | 6.12 | 25.31 | 1.04 |
| $50 \times 500$ | 397.84 | 8.31 | 15.69 | 8.19 | 25.36 | 1.02 |
| $50 \times 600$ | 516.62 | 31.60 | 18.34 | 9.18 | 28.17 | 3.44 |
| $100 \times 500$ | 968.71 | 44.98 | 30.00 | 10.81 | 32.26 | 4.16 |
| $100 \times 600$ | 1274.03 | 66.17 | 35.28 | 13.55 | 36.73 | 4.88 |
| $150 \times 500$ | 1760.90 | 82.02 | 38.02 | 15.61 | 46.28 | 5.25 |
| $150 \times 600$ | 2401.05 | 168.47 | 54.83 | 21.71 | 43.80 | 7.98 |

Table 3: Length of tape and running times of the gradient computation based on the structure and direct reverse-mode AD for the spline local volatility surface construction problem with smoothness term on various number of grids.

|  | Reverse-Mode |  | Structured Gradient |  | tape | time |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $n x \times n y$ | Tape $(\mathrm{MB})$ | Time $(\mathrm{s})$ | Tape $(\mathrm{MB})$ | Time $(\mathrm{s})$ | ratio | speedup |
| $10 \times 10$ | 968.32 | 32.86 | 30.86 | 6.88 | 31.95 | 4.78 |
| $20 \times 20$ | 975.27 | 31.41 | 33.41 | 6.31 | 29.19 | 4.97 |
| $30 \times 30$ | 983.25 | 32.04 | 36.91 | 6.33 | 26.92 | 5.02 |
| $50 \times 50$ | 1008.81 | 39.25 | 47.83 | 6.74 | 21.09 | 5.82 |
| $70 \times 70$ | 1045.12 | 42.72 | 55.27 | 6.79 | 18.92 | 6.28 |

Table 4: Length of tape and running times of the gradient computation based on the structure and direct reverse-mode AD for the spline local volatility surface construction problem with smoothness term on $100 \times 500$ grids with different number of spline knots.
surement, the $j$-th point on the term structure of interest rate has the following dynamic:

$$
\begin{equation*}
d L_{j}(t)=-L_{j}(t)\left(\sum_{k=j+1}^{N} \frac{\alpha_{k} L_{k}(t)}{1+\alpha_{k} L_{k}(t)} \sigma_{j}(t) \sigma_{k}(t) \rho_{j k}\right) d t+L_{j}(t) \sigma_{j}(t) d W_{j}^{N}(t) \tag{5.16}
\end{equation*}
$$

where $L_{j}(t)$ and $L_{k}(t)$ are the values of the $j$-th and $k$-th interest rates of the term structure at time $t, d L_{j}(t)$ is the instantaneous increment of the $j$-th interest rate at time $t, \alpha_{k}$ is a constant about maturity, $\sigma_{j}(t)$ and $\sigma_{k}(t)$ are the volatility of the $j$-th and $k$-th interest rates and $\rho_{j k}$ is the correlation between the $j$-th and $k$-th interest rates.

The above dynamics demonstrate that the instantaneous increment of the $j$-th point on the term structure only depends on the values and volatilities of the points beyond current point on the term structure. We denote $L(t)=\left[L_{1}(t), L_{2}(t), \cdots, L_{N}(t)\right]$ and $d L(t)=$ [dL $\left.L_{1}(t), d L_{2}(t), \cdots, d L_{N}(t)\right]$ at time $t$, then the above dynamic can be written in a function form

$$
d L(t) \equiv f\left(L(t), \sigma(t) \mid d W^{N}(t)\right)
$$

It is easy to show that the Jacobian matrices, $\frac{\partial f}{\partial L(t)}$ and $\frac{\partial f}{\partial \sigma(t)}$ are upper triangular. Therefore, based on the special structures of the Jacobian matrices and the structure idea, the sensitivity delta of a swaption under LMM can be computed efficiently. In this experiment, we consider a ten year swaption with quarterly payments and it expires in five years. The simulation step is the same as payment period so there are twenty simulation steps. Figure 2 and Figure 3 illustrate the length of tape and the realized CPU times for computing the price of the swaption and corresponding Greeks based on the direct reverse-mode and structured gradient technique. Similar to results of the other two examples in this section, the structured gradient idea can save the required space and realized CPU times dynamically, especially when the number of simulations is large.

## 6 Conclusions

Automatic differentiation is a practical field of computational mathematics of growing interest across many industries, including finance. Use of reverse-mode AD is particularly interesting since it allows for the computation of gradients in the same time required to evaluate the objective function itself - there is no analogy in finite differencing (where the usual finite-difference approach takes time proportional to the number of variables multiplied by the time to evaluate the objective function). Finite-differencing time can easily dominate the overall computing time.

While reverse-mode AD holds great promise for computing gradients of "expensive" functions, the drawback is the excessive memory requirement - memory proportional to the number of operations required to evaluate the function itself. This memory requirement can make reverse-mode AD infeasible in some cases (depending on the function complexity and


Figure 2: Length of tape of the gradient computation based on the structure and direct reverse-mode AD for Swaption based on Libor market model via Monte Carlo simulation.


Figure 3: Realized CPU time of the gradient computation based on the structure and direct reverse-mode AD for Swaption based on Libor market model via Monte Carlo simulation.
available RAM) and, in others, slower than expected due to use of secondary memory and non-localized memory references.

A general technique known as "checkpointing" has previously been proposed to reduce the storage required to implement reverse-mode $\mathrm{AD}[20,21]$ in gradient computations. In this paper we have observed and illustrated that many complex functions arising in finance exhibit a natural "substitution structure"; if this structure is explicitly exposed in the presentation of the objective function, then checkpoint location, state variables to be saved at the checkpoints, and explicit and efficient gradient computing formulae are all readily available.

The finance examples we have used here to demonstrate this structural approach are important and broad examples, not narrow in any sense. Nevertheless they are examples without question there are many other examples in computational finance amenable to this structured AD approach.

What about more general Jacobians and Hessian matrices? There has been considerable work on determining such derivative matrices for structured (and sparse) problems. Most of this work utilizes sparsity in some way - either explicit sparsity in the derivative matrices themselves $[15,12]$, or "hidden sparsity" that shows up below the surface in the structured form [30]. We note that the approach provided here for gradients does not utilize any "hidden sparsity" and this allows for a simpler implementation. The extension of this idea in the case of general Jacobians, and Hessians is currently being explored by the authors.

Fast and accurate gradient computation is useful in optimization in general, and computational finance specifically, in itself. For example determination of Greeks (say for hedging purposes) is an important gradient computation. In optimization while it can be advantageous to accurately compute Hessian matrices, there are many effective optimization approaches built strictly on gradient evaluation. For example, quasi-Newton methods (eg., [18]) and nonlinear conjugate gradient methods [22] require first derivatives (and only first derivatives). In addition, a fast gradient code can be used to approximate the Hessian matrix by finite differences (usually a much better approach, numerically, than approximating the Hessian with double finite differences using the function code). If the Hessian matrix is sparse then the fast gradient approach illustrated here can be used to approximate the Hessian matrix, by difference the gradient, in compact space (as we have illustrated here) and in time proportional to $\chi(H) \cdot \omega(f)$, where $\chi(H)$ is the chromatic number of the adjacency graph of the Hessian matrix $H$, and $\omega(f)$ is the number of operations required to evaluate $f(x)$. For sparse problems often $\chi(H) \ll n$.

Acknowledgement. We would like to thank Prof. Mike Giles for his email discussion and clarification of the concept of "checkpointing". We also would like to thank the anonymous reviewer for his valuable comments which improve the readability of our manuscript.

## References

[1] Cayuga Research, ADMAT-2.0 User's Guide, www.cayugaresearch.com, 2013.
[2] ADOLC package, https://projects.coin-or.org/ADOL-C, 2013.
[3] L.B.G., Andersen, and R. Brotherton-Ratcliffe, The equity option volatility smile: an implicit finite difference approach, J. Comput. Finance Vol.1, 1997, 5C37.
[4] M. Bartholomew-Biggs, S. Brown, B. Christianson and L. Dixon, Automatic differentiation of Algorithm, J. Comput. App. Math. Vol. 124, 171-190, 2000.
[5] L. Capriotti and S. Lee, Adjoint Credit Risk Management, to appear in Risk.
[6] L. Capriotti and M. Giles, Fast correlation Greeks by adjoint algorithmic differentiation, Risk, 79-83, 2010.
[7] L. Capriotti and M. Giles, Adjoint Greeks made easy, Risk, 96-102, September 2012.
[8] L. Capriotti, S. Lee and M. Peacock, Real time counterparty credit risk management in Monte Carlo, Risk Magazine, 2011.
[9] Z. Chen and P. Glasserman, Sensitivity estimates for portfolio credit derivatives using Monte Carlo, Finance and Stochastics, Vol.12, 2008, 507-540.
[10] Z. Chen and P. Glasserman, Fast pricing of basket default swaps, Operations Research Vol. 56, 2008, 286-303.
[11] T. F. Coleman, X. Xiong, and W. Xu, Using Directed Edge Separators to Increase Efficiency in the Determination of Jacobian Matrices via Automatic Differentiation, Recent Advances in Algorithmic Differentiation Lecture Notes in Computational Science and Engineering, Springer, 87, 2012, 209-219.
[12] T.F. Coleman and G.F. Jonsson, The efficient computation of structured gradients using automatic differentiation, SIAM J. Sci. Comput., Vol. 20, 1999, 1430-1437.
[13] T. F. Coleman, Y. Li and A. Verma, Reconstructing the Unknown Local Volatility Function, J. Comput. Finan., Vol. 2, 77-102, 1999.
[14] T. F. Coleman, Y. Li and C. Wang,Stable local volatility function calibration using spline kernel, Comput. Optim. Appl., DOI 10.1007/s10589-013-9543-x, 2013.
[15] T.F.Coleman and A.Verma, The efficient computation of sparse Jacobian matrices using automatic differentiation, SIAM J. Sci. Comput. Vol. 19, 1998, 1210-1233.
[16] T. F. Coleman and W. Xu, Fast (Structured) Newton Computations, SIAM J. Sci. Comput.Vol. 31, 2008, 1175-1191.
[17] C.H. Cristian, Adjoints and Automatic (Algorithmic) Differentiation in Computational Finance, Available at SSRN 1828503, 2011.
[18] J.E. Dennis and R.B. Schnabel, Numerical Methods for Unconstrained Optimization and Nonlinear Equations, SIAM, Philadelphia, PA, 1996.
[19] M. Giles and P. Glasserman, Computation methods: Smoking adjoints: fast Monte Carlo Greeks, Risk, Vol.19, 2006, 88-92.
[20] A.Griewank and A. Walther, Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiatio, ACM Trans. on Math. Soft.,Vol.2, 19-45, 2000.
[21] A. Griewank and A. Walther, Evaluating Derivatives : Principles, and Techniques of Algorithmic Differentiation 2nd Ed., SIAM, Philadelphia, PA, 2005.
[22] W. W. Hager and H. Zhang, A survey of nonlinear conjugate gradient methods, Pacific journal of Optimization, Vol.2, 2006, 35-58.
[23] L. Hascoet and M. Araya-polo, Enabling user-driven checkpointing strategies in reversemode automatic differentiation,European Conference on Computational Fluid Dynamics, P. Wesseling, E. Oñate and J. Périaux (Eds), 1-19, 2006.
[24] M. Leclerc, Q. Liang and I. Schneider,Interest rates-Fast Monte Carlo Bermudan Greeks, Risk, Vol.22, 2009, 84-87.
[25] M. Joshi and C. Yang, Fast Delta computations in the swap-rate market model, J. of Economic Dynamics and Control, Vol.35, 2011, 764-775.
[26] M. Joshi and C. Yang, Efficient Greek estimation in generic swap-rate market models, Algorithmic Finance, Vol.1, 2011, 17-33.
[27] M. Joshi and C. Yang, Fast and accurate pricing and hedging of long-dated CMS spread options, International Journal of Theoretical and Applied Finance, Vol.13, 2010, 839865.
[28] C. Kaebe, J. H. Maruhn, and E. W. Sachs, Adjoint based Monte Carlo calibration of financial market models, J. of Finance and Stochastics, Vol.13, 2009, 351379.
[29] U. Naumann, Reducing the Memory Requirement in Reverse Mode Automatic Differentiation by Solving TBR Flow Equations, Lecture Notes in Computer Science, Vol. 2330, 1039-1048, 2002.
[30] A. Walther, Computing Sparse Hessians with Automatic Differentiation, ACM Trans. Maths. Soft., Vol. 34, 2008, 1-15.
[31] W. Xu and T. F. Coleman, Efficient(Partial) Determination of Derivative Matrices via Automatic Differentiation, SIAM Scientific Computing, 35, A1398-A1416, 2013.


[^0]:    *This work was supported in part by the Ophelia Lazaridis University Research Chair (held by Thomas F. Coleman), the National Sciences and Engineering Research Council of Canada and the Natural Science Foundation of China (Project No: 11101310 ).

[^1]:    ${ }^{1}$ As $\lambda$ increases problem (3.9) increasingly emphasizes smoothness over driving $F(x) \rightarrow 0$.

[^2]:    ${ }^{2}$ Typically each state $y_{i}$ will have at least one "representative" in the target vector $y_{T}$

