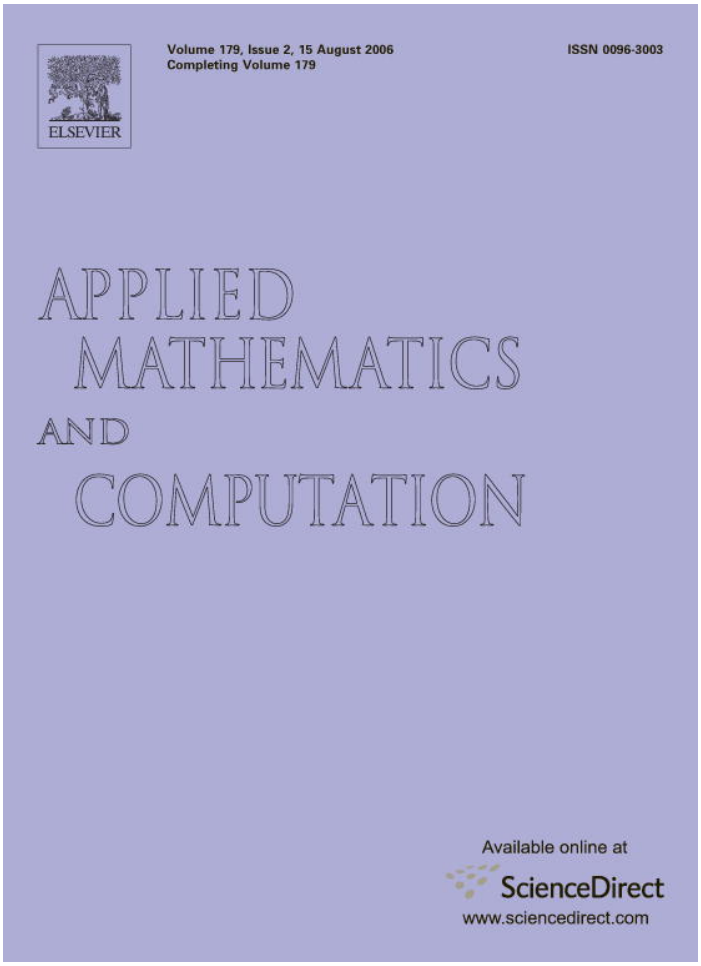
Provided for non-commercial research and educational use only.
Not for reproduction or distribution or commercial use.



This article was originally published in a journal published by Elsevier, and the attached copy is provided by Elsevier for the author's benefit and for the benefit of the author's institution, for non-commercial research and educational use including without limitation use in instruction at your institution, sending it to specific colleagues that you know, and providing a copy to your institution's administrator.

All other uses, reproduction and distribution, including without limitation commercial reprints, selling or licensing copies or access, or posting on open internet sites, your personal or institution's website or repository, are prohibited. For exceptions, permission may be sought for such use through Elsevier's permissions site at:

http://www.elsevier.com/locate/permissionusematerial

Available online at www.sciencedirect.com
ScienceDirect
Applied Mathematics and Computation 179 (2006) 535–544
APPLIED MATHEMATICS AND COMPUTATION
www.elsevier.com/locate/amc

Pricing multi-asset American-style options by memory reduction Monte Carlo methods

Raymond H. Chan, Chi-Yan Wong *, Kit-Ming Yeung

Department of Mathematics, The Chinese University of Hong Kong, Shatin, Hong Kong

Abstract

When pricing American-style options on d assets by Monte Carlo methods, one usually stores the simulated asset prices at all time steps on all paths in order to determine when to exercise the options. If N time steps and M paths are used, then the storage requirement is $d \cdot M \cdot N$. In this paper, we give a simulation method to price multi-asset American-style options, where the storage requirement only grows like $(d+1)M+N$. The only additional computational cost is that we have to generate each random number twice instead of once. For machines with limited memory, we can now use larger values of M and N to improve the accuracy in pricing the options.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Memory reduction method; Monte Carlo method; Multi-asset; American-style options; Random number

1. Introduction

Monte Carlo method is one of the main methods for computing American-style options, see for instance [12,2,3,9]. These algorithms are computationally inefficient because they require the storage of all asset prices at all simulation times for all simulated paths. Thus the total storage requirement grows like $O(dMN)$ where d is the number of underlying assets, M is the number of simulated paths and N is the number of time steps. The accuracy of the simulation is hence severely limited by the storage requirement.

The apparent difficulties in using Monte Carlo methods to price American-style options come from the backward nature of the early exercise feature. There is no way of knowing whether early exercise is optimal when a particular asset price is reached at a given time. One can look at this problem from another point of view. In Monte Carlo method, the simulated paths are all generated in the time-increasing direction, i.e. they start from the initial asset prices $\mathbf{x}_0$ and march to the expiry date according to a given geometric Brownian

Author's personal copy

* Corresponding author.
E-mail addresses: rchan@math.cuhk.edu.hk (R.H. Chan), cywong@math.cuhk.edu.hk (C.-Y. Wong), kmyeung@math.cuhk.edu.hk (K.-M. Yeung).

0096-3003/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.amc.2005.11.108

motion. But since the pricing of American options is a backward process starting from the expiry date back to $\mathbf{x}_0$, the usual approach is to save all the intermediate asset prices along all the paths.

In this paper, we use our simulation method in [4] for computing multi-asset American-style options that does not require storing of all the intermediate asset prices. The storage is reduced from $O(dMN)$ to $(d+1)M+N$ only. Our main idea is to generate the paths twice: one in a forward sweep to establish the asset prices at the expiry date, and one in a backward sweep that computes the intermediate asset prices only when they are needed. The only additional cost in our method is that we have to generate each random number twice instead of once. The resulting computational cost is less than twice of that of the methods where all the intermediate asset prices are stored.

The remainder of this paper is organized as follows. Section 2 recalls the usual full-storage approach for computing multi-asset American-style options. Section 3 gives some background about random number generators in computers. In Section 4, we introduce our memory reduction method. In Section 5, we show how to use our method to compute multi-asset American options by adopting it to the least-squares method proposed by Longstaff and Schwartz [9]. Section 6 gives some numerical results to illustrate the effectiveness of our method.

We will use the MATLAB language [11] to explain how the codes are to be written as the language is easier to comprehend. The corresponding commands in FORTRAN 90 [5] and MATHEMATICA [13] are given in Appendix A.

2. The full-storage method

As usual, we let the prices of d non-dividend paying assets $\mathbf{x}=(x^1,x^2,\ldots,x^d)^{\mathrm{T}}$ follow the geometric Brownian motion

$$\frac{dx^k}{x^k}=r\,\mathrm{d}t+\sigma_k\,\mathrm{d}W_k,\quad 1\leqslant k\leqslant d,$$

where r is the risk-free interest rate, σ_k is the volatility of asset k, and $\mathrm{d}W_k$ is the Wiener process for asset k. By Ito's Lemma, we have

$$\begin{aligned}x^k(t+\mathrm{d}t)&=x^k(t)\exp\left(\left(r-\frac{1}{2}\sigma_k^2\right)\mathrm{d}t+\sum_{j=1}^{d}v_{kj}\,\mathrm{d}W_k\right)\\&=x^k(t)\exp\left(\left(r-\frac{1}{2}\sigma_k^2\right)\mathrm{d}t+\sqrt{\mathrm{d}t}\sum_{l=1}^{d}\mathscr{V}(k,l)\mathbf{z}(l)\right),\end{aligned}\tag{1}$$

where $x^k(t)$ is the price of asset k at time t, $\mathbf{z}$ is a d-vector of standard normal random variables, and $\mathscr{V}=[v_{ij}]$ is the volatility matrix. The volatility matrix $\mathscr{V}$ is given by

$$\mathscr{C}=\mathscr{V}\mathscr{V}^{\mathrm{T}},$$

and $\mathscr{C}=[c_{ij}]$ is the d-by-d covariance matrix with $c_{ij}=\rho_{ij}\sigma_i\sigma_j$, where ρ_{ij} is the correlation coefficient between $\mathrm{d}W_i$ and $\mathrm{d}W_j$, see for instance [8].

In the Monte Carlo simulation, we divide the time horizon into N time steps with each having the length

$$\Delta t=\frac{T-t_0}{N},$$

where t_0 is the current time and T is the expiry date of the option. Thus the time horizon is discretized as $t_0<t_1<\cdots<t_N=T$ where $t_j=t_0+\mathrm{j}\,\Delta t$.

Let the asset prices at time t_0 be $\mathbf{x}_0=(x_0^1,x_0^2,\ldots,x_0^d)^{\mathrm{T}}$. Given $\mathbf{x}_0$, we can simulate the price paths using (1). More precisely, for asset k, $1\leqslant k\leqslant d$, if we are to simulate M paths, then the ith path can be defined by the recurrence:

$$\mathbf{S}_j^i(k)=\mathbf{S}_{j-1}^i(k)\exp\left(\left(r-\frac{\sigma_k^2}{2}\right)\Delta t+\sqrt{\Delta t}\sum_{l=1}^{d}\mathscr{V}(k,l)\mathbf{z}_j^i(l)\right),\quad 1\leqslant i\leqslant M,\ \ 1\leqslant j\leqslant N,\tag{2}$$

Author's Personal Copy

where $\mathbf{S}_j^i$ is the asset price vector, and its kth entry $\mathbf{S}_j^i(k), 1 \leqslant k \leqslant d$, is the price of asset k on the ith path at time t_j with $\mathbf{S}_0^i(k) = x_0^k$ for all i, and $\mathbf{z}_j^i$ is a vector of d independent, identically distributed, standard normal random numbers. For simplicity, we denote $\mathbf{S}_0 = \mathbf{x}_0$.

The straightforward approach, like those in [12,3,9], is to simulate the paths and then store all intermediate asset price vectors $\mathbf{S}_j^i$ up for later computation of the option prices. We will refer this approach as the *full-storage method*. It requires $d \cdot M \cdot N$ storage. Notice that to generate the paths, we need $d \cdot M \cdot N$ standard normal random numbers.

3. The random number generator

According to (2), we need to generate one standard normal random number for each asset at each time step on each path. Most computer languages already have built-in functions to generate these random numbers. In MATLAB, it is "`randn`". By using the concept of a *seed*, one has the flexibility to change or fix the sequence of random numbers each time they are generated. For example, the MATLAB commands

```
randn('seed',s);
e = randn;
```

give a different random number `e` each time the seed `s` is changed, but give the same random number if `s` is fixed.

In MATLAB, one can even extract the seed that generates the next random number in a sequence. The command is

```
s = randn('seed');
```

More precisely, if the sequence corresponding to the seed `s` is

$$\text{set seed } \mathtt{s} \overset{\mathtt{randn}}{\rightarrow} \varepsilon_1 \overset{\mathtt{randn}}{\rightarrow} \varepsilon_2 \overset{\mathtt{randn}}{\rightarrow} \cdots \overset{\mathtt{randn}}{\rightarrow} \varepsilon_{l-1} \overset{\mathtt{randn}}{\rightarrow} \varepsilon_l \overset{\mathtt{randn}}{\rightarrow} \cdots \tag{3}$$

then we can obtain a partial sequence $\{\varepsilon_l\}_{l=k}^{\infty}$ starting from ε_k, provided that we have extracted the seed `c` after generating ε_{k-1}, i.e.

$$\begin{aligned} &\text{set seed } \mathtt{s} \overset{\mathtt{randn}}{\rightarrow} \varepsilon_1 \overset{\mathtt{randn}}{\rightarrow} \varepsilon_2 \overset{\mathtt{randn}}{\rightarrow} \cdots \overset{\mathtt{randn}}{\rightarrow} \varepsilon_{k-2} \overset{\mathtt{randn}}{\rightarrow} \varepsilon_{k-1} \overset{\mathtt{randn('seed')}}{\rightarrow} \mathtt{c} \\ &\text{set seed } \mathtt{c} \overset{\mathtt{randn}}{\rightarrow} \varepsilon_k \overset{\mathtt{randn}}{\rightarrow} \varepsilon_{k+1} \overset{\mathtt{randn}}{\rightarrow} \cdots \overset{\mathtt{randn}}{\rightarrow} \varepsilon_{l-1} \overset{\mathtt{randn}}{\rightarrow} \varepsilon_l \overset{\mathtt{randn}}{\rightarrow} \cdots \end{aligned} \tag{4}$$

In view of this, we will call an integer `c` the *current seed* of a random number ε_k, if setting the seed to `c`, the next random number so generated is ε_k.

4. The forward-path method

In this section, we apply our method which does not need to store the intermediate asset price vectors $\{\mathbf{S}_j^i\}$ when computing the multi-asset option prices. In this method, each vector of random numbers $\{\mathbf{z}_j^i\}$ is generated twice, but the intermediate asset price vectors $\mathbf{S}_j^i$ are generated only once as in the full-storage method.

From (2), the intermediate asset price vectors are given by

$$\mathbf{S}_j^i = \mathbf{S}_0 \cdot \exp\left(\mathrm{j}\left(r - \frac{\boldsymbol{\sigma} \cdot \boldsymbol{\sigma}}{2}\right)\Delta t + \sqrt{\Delta t}\,\mathscr{V}(\mathbf{z}_1^i + \mathbf{z}_2^i + \cdots + \mathbf{z}_j^i)\right), \quad 1 \leqslant i \leqslant M, \ \ 1 \leqslant j \leqslant N, \tag{5}$$

where $\cdot$ is the *Hadamard product*, and $\sigma \cdot \sigma = ((\sigma_1)^2, (\sigma_2)^2, \ldots, (\sigma_d)^2)^{\mathrm{T}}$.

In the forward-path method, we generate the vectors of random numbers $\mathbf{z}_j^i$ in the following manner. Given an arbitrary seed $\mathbf{s}(1)$:

Author's personal copy

$$\begin{aligned}
\text{set seed } \mathbf{s}(1) &\to \mathbf{z}_1^1 \to \mathbf{z}_1^2 \to \cdots \to \mathbf{z}_1^M \quad (\to \text{ get seed } \mathbf{s}(2))\\
&\to \mathbf{z}_2^1 \to \mathbf{z}_2^2 \to \cdots \to \mathbf{z}_2^M \quad (\to \text{ get seed } \mathbf{s}(3))\\
&\to \cdots\\
&\to \mathbf{z}_{N-1}^1 \to \mathbf{z}_{N-1}^2 \to \cdots \to \mathbf{z}_{N-1}^M \quad (\to \text{ get seed } \mathbf{s}(N))\\
&\to \mathbf{z}_N^1 \to \mathbf{z}_N^2 \to \cdots \to \mathbf{z}_N^M.
\end{aligned} \tag{6}$$

More precisely, we generate the d-vector of random numbers $\mathbf{z}_j^i$ on every path i, $1 \leqslant i \leqslant M$, for the time step $j = 1$ first. Then we generate them on all paths for $j = 2$ etc. At the last path (path M) for each time step (i.e. after generating dM random numbers), we get and save the current seed $\mathbf{s}(j)$ for later use. One important point to note here is that we do not need to store any of the random number $\mathbf{z}_j^i(k)$ so generated. In fact, according to (5), in order to compute $\mathbf{S}_N^i$, we only need the sum $\sum_{j=1}^N \mathbf{z}_j^i$. We assign a d-by-M matrix $\mathscr{U}$ to hold this. Thus once $\mathbf{z}_j^i$ is generated, we add it to the ith column $\mathscr{U}(:,i)$ of $\mathscr{U}$:

$$\mathscr{U}(:,i) = \mathscr{U}(:,i) + \mathbf{z}_j^i(:), \quad 1 \leqslant i \leqslant M, \tag{7}$$

and $\mathbf{z}_j^i$ can then be discarded. When all the random numbers are generated, we compute $\mathbf{S}_N^i$ on all paths by (5):

$$\mathbf{S}_N^i = \mathbf{S}_0 \cdot \exp\left(N\left(r - \frac{\boldsymbol{\sigma}\cdot\boldsymbol{\sigma}}{2}\right)\Delta t + \sqrt{\Delta t}\mathscr{V}\mathscr{U}(:,i)\right), \quad 1 \leqslant i \leqslant M. \tag{8}$$

When computing the option price, we have to move backward in time. This can be done by rewriting (2) as

$$\mathbf{S}_{j-1}^i(k) = \mathbf{S}_j^i(k)\exp\left(-\left(r - \frac{\sigma_k^2}{2}\right)\Delta t - \sqrt{\Delta t}\sum_{l=1}^d \mathscr{V}(k,l)\mathbf{z}_j^i(l)\right), \tag{9}$$

$1 \leqslant i \leqslant M$, $1 \leqslant j \leqslant N$, $1 \leqslant k \leqslant d$. Thus given $\{\mathbf{S}_j^i\}_{i=1}^M$, to obtain $\{\mathbf{S}_{j-1}^i\}_{i=1}^M$, we only need the vectors of random numbers $\{\mathbf{z}_j^i\}_{i=1}^M$. In view of (6), we can generate them by using the seed $\mathbf{s}(j)$, i.e.

$$\text{set seed } \mathbf{s}(j) \to \mathbf{z}_j^1 \to \mathbf{z}_j^2 \to \cdots \to \mathbf{z}_j^M. \tag{10}$$

Repeating the idea, we can generate the asset price vectors $\{\mathbf{S}_j^i\}_{i=1}^M$ for all time steps j in the backward manner, and at each time step, we only need the storage for storing $\{\mathbf{S}_j^i\}_{i=1}^M$ for that time step.

We note that the whole forward-path approach requires only an N-vector $\mathbf{s}$ and a d-by-M matrix $\mathscr{U}$. The current asset price vectors $\{\mathbf{S}_j^i\}_{i=1}^M$ at any time step j can be stored using the storage for the matrix $\mathscr{U}$ (which is not needed once $\mathbf{S}_N^i$ is computed). Regarding the computational cost, we note that the cost of (9) is the same as that of (2). Hence in the forward-path method, the only additional costs as compared to the full-storage method are

(i) the generation of the current seeds $\mathbf{s}(j)$ in the forward sweep (6),
(ii) the generation of the random numbers in the backward sweep (10), and
(iii) the summation of these vectors of random numbers in (7).

More precisely, the additional cost will be N calls of `randn('seed')` and `randn('seed',s)`, dMN calls of `randn`, and dMN additions. We will see in the numerical example in Section 6 that the total cost is less than twice of that of the full-storage method. We emphasize that all results obtained by the forward-path method and full-storage method are exactly the same, since we are using the same paths to price the option.

We end this section by pointing out that our forward-path method can be applied to more general pseudo-random number generators. More precisely, if the generator has the properties (3) (i.e. a unique sequence is generated for a fixed seed), and (4) to extract the current seeds in any random number sequence it generates, then we can apply the forward-path method.

Author's personal copy

5. The least-squares method

Our path generating technique can reduce the memory requirement of Monte Carlo methods for pricing multi-asset American-style options. In this section, we illustrate this by pricing an American put option on the maximum of multi-assets using the least-squares approach developed by Longstaff and Schwartz [9].

At the final exercise date, the optimal exercise strategy for an American option is to exercise the option if it is in the money. Prior to the final date, however, the optimal strategy is to compare the immediate exercise value with the expected cash flow from continuing, and then exercise if immediate exercise is more valuable. Thus the key to optimally exercising an American option is to identify the conditional expected payoff from continuation. In [9], the cross-sectional information in the simulated paths is used to identify the conditional expectation function. This is done by regressing the payoffs from continuation on a set of basis functions depending on the state variables at each time step. The fitted function from this regression is an efficient unbiased estimate of the conditional expectation function, and by which, one can estimate an optimal stopping rule for the option.

According to the stopping rule, the cash flows generated by the option at each exercise time are identified. The option price is then calculated by discounting all these cash flows back to time t_0 at the risk-free rate r. Since only the paths for which the option is in the money are included in the regression, the method significantly reduces the computational time. However, as in other Monte Carlo methods for computing American options, the method saves all the intermediate asset prices for the computation of the option price. It therefore requires a storage of size dMN, where d is the number of underlying assets, M is the number of stimulated paths, and N is the number of time steps. Here we apply our method in Section 4 to reduce the storage requirement to $(d+1)M+N$.

We illustrate our method with a numerical example. Consider an American put option on the maximum of three non-dividend paying assets (i.e. $d=3$) with strike price E equals to \$45. The current prices of the three assets $\mathbf{S}_0$ are $(\$40,\$40,\$40)^{\mathrm{T}}$, the risk-free rate r is 0.05, the volatilities of the assets $\boldsymbol{\sigma}$ are 0.2, 0.3 and 0.5, respectively, and the time to maturity T is 7 months. Their correlation coefficients are all equal to 0.5. We assume that the option is exercisable at time $j=0,1,2,3,4$ and 5 (i.e. $N=5$). We illustrate the algorithm by using 20 simulated paths, i.e. $M=20$.

Algorithm

1. Initialization:
   (a) Set the seed to $\mathbf{s}(1)$ which is chosen arbitrarily. Using (6)–(8) get and save the current seed $\{\mathbf{s}(j)\}_{j=2}^{5}$, and for each path i, compute $\mathbf{S}_5^i$ and save it in $\mathscr{U}(:,i)$.
   (b) For each path i, compute the present value of the payoff at the expiry date T, (i.e. $\mathrm{e}^{-5r\,\Delta t}\max\{E-\max_{1\leqslant k\leqslant 3}\{\mathbf{S}_5^i(k)\},0\}$), and save them in a vector $\mathbf{p}$. The value is the cash flow realized by the option holder conditional on not exercising the option before the expiry date $j=5$.

Path i	$\mathbf{S}_5^i(1)$	$\mathbf{S}_5^i(2)$	$\mathbf{S}_5^i(3)$	$\mathbf{p}(i)=\mathrm{e}^{-5r\,\Delta t}\max\{E-\max_{1\leqslant k\leqslant 3}\mathbf{S}_5^i(k),0\}$
1	44.392	35.403	36.127	0.590
2	40.421	36.054	42.817	2.121
3	42.083	54.663	33.340	0.000
4	44.327	53.906	40.571	0.000
5	48.002	36.657	36.067	0.000
6	42.348	57.317	41.173	0.000
7	44.265	51.509	61.581	0.000
8	30.598	30.325	48.162	0.000
⋮	⋮	⋮	⋮	⋮
19	43.555	36.844	55.607	0.000
20	40.086	43.058	55.567	0.000

Author's personal copy

2. Backward time-marching to $j = 4$:
   (a) With the saved seed $\mathbf{s}(5)$, generate the random numbers $\{\mathbf{z}_5^i\}_{i=1}^M$ again by (10). Then compute $\mathbf{S}_4^i$ for all paths by (9) and store them in $\mathscr{U}(:,i)$, i.e. the memory location of $\mathbf{S}_5^i$ will be overwritten by $\mathbf{S}_4^i$.
   (b) Decide if the path is in the money, i.e. if $\max_{1\leqslant k\leqslant 3}\mathbf{S}_4^i(k) < E$.

Path i	$\mathbf{p}(i)$	$\mathbf{S}_4^i(1)$	$\mathbf{S}_4^i(2)$	$\mathbf{S}_4^i(3)$	In the money?
1	0.590	46.111	44.486	43.397	
2	2.121	40.403	38.493	50.854	
3	0.000	43.223	40.234	36.557	YES
4	0.000	49.026	56.502	43.556	
5	0.000	45.488	32.801	33.771	
6	0.000	39.441	53.467	34.646	
7	0.000	38.725	45.518	48.849	
8	0.000	28.830	32.181	42.138	YES
⋮	⋮	⋮	⋮	⋮	⋮
19	0.000	41.122	38.945	56.107	
20	0.000	39.870	45.827	57.358	

   (c) For those paths that are in the money, we use the least-squares approach in [9] to estimate the expected payoff from continuing to hold the option conditional on $\mathbf{S}_4^i$. More precisely, regress $\mathbf{p}(i)$ on the set of basis functions 1, $\mathbf{S}_4^i(1)$, $\mathbf{S}_4^i(2)$, $\mathbf{S}_4^i(3)$, $(\mathbf{S}_4^i(1))^2$, $(\mathbf{S}_4^i(2))^2$ and $(\mathbf{S}_4^i(3))^2$ for $i = 3, 8, 10, 12, 13, 15, 16, 17$. The resulting conditional expected payoff function is

$$\mathscr{E}[p|\mathbf{S}_4^i] = 3.797 + 0.203\mathbf{S}_4^i(1) - 0.183\mathbf{S}_4^i(2) - 0.133\mathbf{S}_4^i(3) - 2.545(\mathbf{S}_4^i(1))^2 - 9.484(\mathbf{S}_4^i(2))^2 - 4.417(\mathbf{S}_4^i(3))^2. \tag{11}$$

   The seven coefficients in (11) are obtained by solving the least-squares problem:

$$\begin{bmatrix} 1 & \mathbf{S}_4^3(1) & \mathbf{S}_4^3(2) & \mathbf{S}_4^3(3) & \cdots & (\mathbf{S}_4^3(3))^2 \\ \vdots & & & & \vdots & \\ 1 & \mathbf{S}_4^i(1) & \mathbf{S}_4^i(2) & \mathbf{S}_4^i(3) & \cdots & (\mathbf{S}_4^i(3))^2 \\ & & & \ddots & & \\ \vdots & & & & & \vdots \\ 1 & \mathbf{S}_4^{17}(1) & \mathbf{S}_4^{17}(2) & \cdots & (\mathbf{S}_4^{17}(2))^2 & (\mathbf{S}_4^{17}(3))^2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{bmatrix} = \begin{bmatrix} \mathbf{p}(3) \\ \vdots \\ \mathbf{p}(i) \\ \vdots \\ \mathbf{p}(17) \end{bmatrix}. \tag{12}$$

   Here only the paths for which the option is in the money are included, i.e. $i = 3, 8, 10, 12, 13, 15, 16, 17$.
   (d) For the paths that are in the money, substitute their $\mathbf{S}_4^i$ into the right-hand side of (11) to obtain the conditional expected payoffs for continuing to hold the option. Then compare them with the present value of immediate exercising to see if we should exercise (see the table below). For those paths where we should exercise (i.e. $i = 3, 10, 13, 15, 17$), we update the payoff vector $\mathbf{p}(i)$ by the exercising value, i.e.

$$\mathbf{p}(i) = \begin{cases} \mathrm{e}^{-4r\,\Delta t}\max\left\{E - \max\limits_{1\leqslant k\leqslant 3}\mathbf{S}_4^i(k), 0\right\}, \\ \quad \text{if } \mathrm{e}^{-4r\,\Delta t}\max\left\{E - \max\limits_{1\leqslant k\leqslant 3}\mathbf{S}_4^i(k), 0\right\} > \mathscr{E}[p|\mathbf{S}_4^i], \\ \text{unchanged}, \\ \quad \text{if otherwise.} \end{cases}$$

Author's personal copy

Path i	Continuation $\mathscr{E}[p\|\mathbf{S}_4^i]$	Exercising $e^{-4r\Delta t}\max\{E-\max_{1\leqslant k\leqslant 3}\mathbf{S}_4^i(k),0\}$	Should we exercise?	Updated $\mathbf{p}$(i)
3	−0.162	1.736	YES	1.736
8	3.040	2.796		Unchanged
10	7.658	10.029	YES	10.029
12	9.250	3.118		Unchanged
13	4.548	5.474	YES	5.474
15	8.432	9.315	YES	9.315
16	7.110	3.809		Unchanged
17	3.041	4.887	YES	4.887

(e) Go back to Step 2(a) and backward time-marching to $j = 3$ etc.

In essence, given $\mathbf{S}_{j+1}^i$ and $\mathbf{p}$, the algorithm first computes $\mathbf{S}_j^i$ using (10) and (9). Then it finds $\mathscr{E}[p|\mathbf{S}_j^i]$ by solving the least-squares problem (12) for those paths that are in the money. Using $\mathscr{E}[p|\mathbf{S}_j^i]$, it computes the conditional expected payoffs for all paths that are in the money, and then compare them with the present value of immediate exercising. Finally, it updates the corresponding entries of the payoff vector $\mathbf{p}$ in case we should exercise.

To complete the example, we backward time-march to $j = 1$ and get

$$\mathbf{p} = [0.590, 8.207, 1.736, 4.951, 0.000, 3.323, 5.068, 10.716, 1.509, \ldots, 4.887, 0.000, 5.064, 0.000]^{\mathrm{T}}.$$

The option can now be valued by averaging all the entries of $\mathbf{p}$ at time $j = 1$. For this example, it will be \$4.161. Then compare it with the value of immediate exercising to see if we should exercise at time $j = 0$. Thus, we get the option price equals to \$5.000.

From the example, we see that for the forward-path method, we only need to store $\mathscr{U}$, $\mathbf{p}$, and $\mathbf{s}$. We summarize the memory requirement in Table 1. For completeness, we also list in the table the additional cost required by our method as compared to the full-storage method. From the algorithm, we notice that the additional costs are precisely the costs of generating the random numbers in the backward sweep, and these costs are already given at the end of Section 4.

6. Numerical examples

In this section, we test our method on an example given in [1, p. 400]. It is an American put option on the maximum of three assets (i.e. $d = 3$) with strike price E. The current prices of the three assets $\mathbf{S}_0$ are $(\$40, \$40, \$40)^{\mathrm{T}}$, the riskless rate r is 0.05, the volatilities of the assets $\boldsymbol{\sigma}$ are 0.2, 0.3 and 0.5 respectively, and the expiry date is T months. Their correlation coefficients are given by ρ. We emphasize that the results obtained by the full-storage method and the forward-path method are exactly the same, since we are using the same paths to price the option. In our experiment, all our computations were done by FORTRAN 90 on a SGI Origin 3200 machine with 16 Gigabyte RAM and only one processor is used. Again we use M and N to denote the number of paths and the number of time steps respectively.

Table 1
Memory and cost comparison

Method	Memory	Additional computational cost
Full-storage	$dMN + M + O(1)$	–
Forward-path	$(d+1)M + N + O(1)$	N calls to get $\mathbf{s}$ in (6) dMN additions to find $\mathscr{U}$ in (7) N calls to set $\mathbf{s}$ in (10) dMN calls to generate $\{\mathbf{z}_j^i\}$ in (10)

Authors personal copy

We remark that the least-square problem (12) is usually ill-conditioned especially when $\{\mathbf{S}_j^i(k)\}$ are large. We remedied that by choosing the basis $\left\{1, \widetilde{\mathbf{S}}_j^i(1), \widetilde{\mathbf{S}}_j^i(2), \widetilde{\mathbf{S}}_j^i(3), \frac{1}{1+\widetilde{\mathbf{S}}_j^i(1)}, \frac{1}{1+\widetilde{\mathbf{S}}_j^i(2)}, \frac{1}{1+\widetilde{\mathbf{S}}_j^i(3)}\right\}$ instead of $\left\{1, \mathbf{S}_j^i(1), \mathbf{S}_j^i(2), \mathbf{S}_j^i(3), (\mathbf{S}_j^i(1))^2, (\mathbf{S}_j^i(2))^2, (\mathbf{S}_j^i(3))^2\right\}$ when doing the regression in (11), where

$$\widetilde{\mathbf{S}}_j^i(k) = \mathbf{S}_j^i(k) - \frac{\sum_{k=1}^d \mathbf{S}_0(k) \mathrm{e}^{j\left(r-\frac{\sigma_k^2}{2}\right)\Delta t}}{d}.$$

The least-squares problem is then solved by using the Givens QR method [7, p. 226].

Tables 2 and 3 show the effect on the errors by increasing M from 1000 to 100,000 while N is fixed. In the tables, the data under the column "P_{PDE}" are results computed by the classical integration method and are given in [1, p. 400]. The results under the "Mean" and "STD" are the means and standard deviations obtained after 10 trials. The final column "Error" is the difference between "P_{PDE}" and the "Mean". From the tables, we see that the error decreases by one decimal point when M is increased 100 times. This is consistent with the error estimate $O(1/\sqrt{M})$ of the Monte Carlo method [6].

Table 4 gives the CPU time for one run of the methods when $E = 40$ and $\rho = 0.5$. The symbol "†" signifies that memory is not enough to run the problem of that size. From the table, we observe that the time increases roughly like linear with respect to M and N. Moreover, as expected, the time required by our memory reduction method is always less than twice of that of the full-storage method.

Table 2
Forward-path method with $M = 10^3$ and $N = 10$

ρ (%)	T	E	P_{PDE}	Mean	STD	Error
0	1	35	0.00	0.000	0.000	0.000
		40	0.23	0.232	0.021	0.002
		45	5.00	5.000	0.000	0.000
0	4	35	0.01	0.011	0.003	0.001
		40	0.44	0.448	0.040	0.008
		45	5.00	5.000	0.000	0.000
0	7	35	0.04	0.049	0.012	0.009
		40	0.57	0.582	0.053	0.012
		45	5.00	5.000	0.000	0.000
50	1	35	0.00	0.003	0.001	0.003
		40	0.48	0.479	0.037	−0.001
		45	5.00	5.000	0.000	0.000
50	4	35	0.09	0.101	0.020	0.011
		40	0.93	0.919	0.066	−0.011
		45	5.00	5.000	0.000	0.000
50	7	35	0.20	0.234	0.033	0.034
		40	1.19	1.173	0.088	−0.017
		45	5.00	5.000	0.000	0.000
100	1	35	0.01	0.006	0.003	−0.004
		40	0.84	0.875	0.053	0.035
		45	5.00	5.000	0.000	0.000
100	4	35	0.19	0.211	0.032	0.021
		40	1.56	1.607	0.096	0.047
		45	5.00	5.040	0.057	0.040
100	7	35	0.42	0.457	0.054	0.037
		40	1.96	2.006	0.113	0.046
		45	5.20	5.253	0.115	0.053

Author's personal copy

Table 3
Forward-path method with $M = 10^5$ and $N = 10$

ρ (%)	T	E	P_{PDE}	Mean	STD	Error
0	1	35	0.00	0.000	0.000	0.000
		40	0.23	0.221	0.003	−0.009
		45	5.00	5.000	0.000	0.000
0	4	35	0.01	0.013	0.000	0.003
		40	0.44	0.431	0.004	−0.009
		45	5.00	5.000	0.000	0.000
0	7	35	0.04	0.041	0.000	0.001
		40	0.57	0.558	0.005	−0.012
		45	5.00	5.000	0.000	0.000
50	1	35	0.00	0.002	0.000	0.002
		40	0.48	0.478	0.003	−0.002
		45	5.00	5.000	0.000	0.000
50	4	35	0.09	0.087	0.001	−0.003
		40	0.93	0.917	0.002	−0.013
		45	5.00	5.000	0.000	0.000
50	7	35	0.20	0.205	0.003	0.005
		40	1.19	1.175	0.005	−0.015
		45	5.00	5.000	0.000	0.000
100	1	35	0.01	0.006	0.000	−0.004
		40	0.84	0.848	0.003	0.008
		45	5.00	5.000	0.000	0.000
100	4	35	0.19	0.199	0.002	0.009
		40	1.56	1.567	0.005	0.007
		45	5.00	5.002	0.004	0.002
100	7	35	0.42	0.428	0.003	0.008
		40	1.96	1.968	0.004	0.008
		45	5.20	5.191	0.006	−0.009

Table 4
CPU time in seconds for different methods

	M							
	10^5				10^4	5×10^4	10^5	5×10^5
N	50	100	150	200	50			
Full-storage	7.80	†	†	†	0.79	3.98	7.85	†
Forward-path	10.34	20.77	30.96	41.32	1.03	5.18	10.36	51.91

Acknowledgment

The research was partially supported by the Hong Kong Research Grant Council Grant CUHK4243/01P and CUHK DAG 2060220.

Appendix A

In FORTRAN 90 [5], the commands to set the seed to s are:

```
call random_seed(size=k)
seed(1:k)=s
call random_seed(put=seed(1:k))
```

Author's personal copy

where k is the number of 32-bit words used to hold the seed. In our machine, $k = 64$. The commands to extract the current seed c are:

```
call random_seed(get=current(1:k))
c=current(1:k)
```

We remark that our FORTRAN 90 only provides uniformly distributed random numbers. We have used the Box–Muller transform [10, p. 73] to produce normal distributed random numbers.

In MATHEMATICA [13], the seeds are set by "SeedRandom[s]". To extract the current seed, use "c=$RandomState". To reset the seed to c, use "$RandomState=c". MATHEMATICA provides uniformly distributed random numbers with "Random[ ]". One can again use the Box–Muller transform to produce normal distributed random numbers.

References

[1] J. Barraquand, D. Martineau, Numerical valuation of high dimensional multivariate American securities, J. Financ. Quant. Anal. 30 (1995) 383–405.
[2] P. Boyle, M. Broadie, P. Glasserman, Monte Carlo methods for security pricing, J. Econom. Dynam. Control 21 (1997) 1267–1321.
[3] M. Broadie, P. Glasserman, Pricing American-style securities using simulation, J. Econom. Dynam. Control 21 (1997) 1323–1352.
[4] R. Chan, C. Wong, K. Yeung, Pricing American-Style Opinions by Monte Carlo Methods without Storing all the Intermediate Asset Prices, CUHK Math. Dept. Research Report #2002-11.
[5] S. Chapman, Fortran 90/95 for Scientists and Engineers, McGraw-Hill, 1998.
[6] S. Galanti, A. Jung, Low-discrepancy sequences: Monte Carlo simulation of option prices, J. Dervi. 5 (1997) 63–83.
[7] G. Golub, C. Van Loan, Matrix Computations, JHU Press, 1996.
[8] Y. Kwok, Mathematical Models of Financial Derivatives, Springer-Verlag, 1998.
[9] F. Longstaff, E. Schwartz, Valuing American options by simulation: a simple least-squares approach, Rev. Financ. Stud. 14 (2001) 113–147.
[10] S. Ross, A First Course in Probability, fifth ed., Prentice-Hall, 1998.
[11] K. Sigmon, T. Davis, MATLAB Primer, sixth ed., CRC Press, 2002.
[12] J. Tilley, Valuing American options in a path-simulation model, Trans. Soc. Actuaries 45 (1993) 563–577.
[13] S. Wolfram, The Mathematica Book, fourth ed., Cambridge University Press, 1999.

Author's personal copy