MEMORY-REDUCTION METHOD FOR PRICING AMERICAN-STYLE OPTIONS UNDER EXPONENTIAL LÉVY PROCESSES

RAYMOND H. CHAN* AND TAO WU†

Abstract. This paper concerns the Monte Carlo method in pricing American-style options under the general class of exponential Lévy models. Traditionally, one must store all the intermediate asset prices so that they can be used for the backward pricing in the least squares algorithm. Therefore the storage requirement grows like $O(mn)$, where m is the number of time steps and n is the number of simulated paths. In this paper, we propose a simulation method where the storage requirement is only of order $O(m+n)$. The total computational cost is less than twice of that of the traditional method. For machines with limited memory, one can now enlarge m and n to improve the accuracy in pricing the options. In numerical experiments, we illustrate the efficiency and accuracy of our method by pricing American options where the log-prices of the underlying assets follow typical Lévy processes such as Brownian motion, lognormal jump-diffusion process, and variance gamma process.

1. Introduction. During the past decade, the exponential Lévy models have been popularized in financial modeling among researchers as well as practitioners, see e.g. [11]. The classical Black-Scholes model [3] presumes that the price of the underlying asset follows a geometric Brownian motion with constant volatility. However, the empirical observation in real financial trading reveals that the implied volatility surface often displays a so-callel volatility smile [18]. Moreover, the distribution of the asset return, assumed to be Gaussian in the Black-Scholes model, exhibits a heavy tail [10], i.e. large moves of the market have decent probabilities to occur. As remedies for Black-Scholes, the exponential Lévy models contain Lévy jumps in addition to the classical diffusion, so that the phenomena of the volatility smiles and the heavy tails can be generically accounted for, see [11]. We remark that the exponential Lévy model is a very general class of models. It includes well-known examples such as the Black-Scholes model [3], lognormal jump-diffusion model [19], double-exponential jump-diffusion model [15], variance gamma model [17], normal inverse Gaussian model [2], CGMY model [5], etc. We refer to the classical reference [11] for further background in financial modeling by exponential Lévy processes. The present paper concerns the use of Monte Carlo simulation in pricing American-style options under the general framework of exponential Lévy models.

It is well known, see e.g. [13], that with the no-arbitrage principle the option price is given by the discounted expected payoff under certain risk-neutral measure. This leads to the option pricing by Monte Carlo method, for which the first application was made by Boyle [4] in 1977. Since then, Monte Carlo method has been a popular tool in pricing financial derivatives, see [13]. Yet, Monte Carlo method had been known for their difficulties in handling American-style options with early exercise feature. In 2001 Longstaff and Schwartz [16] proposed a practical algorithm, named least squares method (LSM), to price American options. Their method is based on a backward-in-time induction, where at each time step the continuation value of the option is estimated by a least square approximation.

However, one drawback of LSM is that in order to compute the intermediate exercise prices at all time steps, it requires the storage of all asset prices at all time steps for all simulated paths. Thus, the total storage requirement grows like $O(mn)$ where m is the number of time steps and n is the number of simulated paths. Therefore, the plain Monte Carlo method, referred as the *full-storage method* in this paper, is computationally inefficient since the accuracy of the simulation is severely limited by the storage requirement.

One alternative for alleviating this storage problem is those "bridge methods" such as the Brownian bridge method [9], the inverse Gaussian bridge method, [22] and the gamma bridge method [23]. By the use of the bridges, the memory requirement can be reduced to $O(n \log m)$. Nevertheless, one drawback is that a specific bridge method can only work on the corresponding model that the price of the underlying asset follows. For examples, Brownian bridge is for the Brownian motion, gamma bridge is for the variance gamma process and so on. That is to say, all bridge methods are model-dependent, which limits its use in applications.

*Department of Mathematics, The Chinese University of Hong Kong, Shatin, NT, Hong Kong SAR, PR China. Email: rchan@math.cuhk.edu.hk. Research was supported in part by HKRGC Grant CUHK 400408 and CUHK DAG 2060257.

†Department of Mathematics, The Chinese University of Hong Kong, Shatin, NT, Hong Kong SAR, PR China. Email: twu@math.cuhk.edu.hk

In this paper, we develop a memory-reduction method, which does not require storing of all intermediate asset prices. The storage is significantly reduced to $O(m+n)$. Coupled with the least squares method proposed in [16], our memory-reduction method is applicable to the general class of exponential Lévy processes. The main idea of our method is to first generate the price process forward till the expiration time and to store only the *seeds* of the random number sequences at each time step. When computing the option prices backwardly, we recompute the just-in-time asset prices using the corresponding seeds. Since the prices are recomputed exactly, the memory-reduction method gives the same result as the full-memory method. The additional computational cost is the cost of regenerating the random numbers corresponding to the asset prices. Hence the total computational cost is always less than twice of that of the full-storage method.

The remainder of the paper is organized as follows. Section 2 reviews the exponential Lévy processes as well as the full-storage method. Section 3 gives the background of random number generators and the concept of seeds. Section 4 introduces our memory-reduction method. In Section 5, we show how the memory-reduction method is applied to specific models, namely the Black-Scholes model, Merton's jump-diffusion model and the variance gamma model. Numerical results are provided there to show the efficiency and accuracy of our method by comparing it with methods from other well-known approaches. Concluding remarks are drawn in Section 6.

2. Exponential Lévy processes and the full-storage method. Let the risk-neutral price dynamics be modeled by the exponential Lévy process

$$S_t = S_0 \exp(rt + L_t), \tag{2.1}$$

with the risk-free rate r and a Lévy process L_t. A Lévy process L_t is a stochastic process with stationary independent increments, continuous in probability, having sample paths that are right-continuous with left limits ("cadlag"), and satisfying $L_0 = 0$. We note that the increments, $L_s - L_t$ for any $s > t$, are independent if the increments $L_s - L_t$ and $L_u - L_v$ are independent random variables whenever the two time intervals $[t, s]$ and $[v, u]$ do not overlap. The increments are stationary if the distribution of any increment $L_s - L_t$ only depends on $s - t$; and therefore increments with equally long time intervals are identically distributed.

We first review the Monte Carlo simulation for computing American-style options. First the time horizon is discretized into m time steps with equal length $\Delta t := \frac{T-t_0}{m}$ as $t_0 < t_1 < ... < t_m = T$, or $t_j = t_0 + j\Delta t$, where t_0 is the current time and T is the expiration date of the option. Let $L_{i,j}$ denote the realization of L_t on the i-th path at time t_j. They are computed by adding the increment $\Delta L_{i,j} := L_{i,j} - L_{i,j-1}$ to $L_{i,j-1}$ recursively at each time step. Thus the whole path simulation process is to simulate the random numbers that give $\Delta L_{i,j}$. We will denote by $\Sigma_{i,j} = \{\varepsilon_{i,j}^k\}_{k=1}^{\eta_{i,j}}$ the ordered set of [0,1] uniform random numbers used in generating $\Delta L_{i,j}$. Here $\eta_{i,j}$ is the number of random numbers required to generate $\Delta L_{i,j}$. It is different for different process. The outline for a general procedure of path simulation is given below:

ALGORITHM 2.1. *(Path simulation)*
For-loop: $i = 1, 2, ..., n$
  Set $L_{i,0} \leftarrow 0$
  For-loop: $j = 1, 2, ..., m$
    1. *Get the increment* $\Delta L_{i,j}$ *by generating* $\Sigma_{i,j}$
    2. $L_{i,j} \leftarrow L_{i,j-1} + \Delta L_{i,j}$
    3. $S_{i,j} \leftarrow S_0 \exp(rj\Delta t + L_{i,j})$
  End for-loop
End for-loop

Algorithm 2.1 simulates the paths and then store all intermediate asset prices $S_{i,j}$ for later computation of the option prices. Hence the storage requirement grows like $O(mn)$. We will refer this method as the full-storage method. Once we have all the intermediate asset prices $S_{i,j}$, we can price American-style options using the least square method (LSM) suggested by [16]. Let us recall it here. At the final exercise date T, the optimal exercise strategy for an American option is to exercise it if it is in the money. This can be done as the terminal asset prices $S_{i,m}$ are available for each path i. Prior to T, however, the optimal

strategy is to compare the immediate exercise value with the expected cash flows from continuing, and then exercise if immediate exercise is more valuable. In the full-storage method, the intermediate asset prices $S_{i,j}$ are available for each path i and at each time step j. Thus, the key to optimally exercising an American option is to identify the conditional expected value of continuation. In [16], the cross-sectional information in the simulated paths is used to identify the conditional expectation function. This is done by regressing the cash flows from continuation on a set of basis functions depending on the current asset prices $S_{i,j}$. The fitted function from this regression is an efficient unbiased estimate of the conditional expectation functions, and by which, one can estimate an optimal stopping rule for the option.

Numerical illustration of LSM for pricing American put options under the Black-Scholes framework can be found for instance in [16]. The computational complexity of the full-storage method is of order $O(mn)$.

3. Random number generators. In Step 1 of Algorithm 2.1, in order to get $\Delta L_{i,j}$, we need to generate a set of [0,1] uniform random numbers $\{\Sigma_{i,j}\}$ for each time step j on each path i. Most programming softwares already have built-in functions to generate $[0,1]$ uniform random numbers. In MATLAB, we can initialize the pseudorandom number generator with seed `d` by the command `rand('seed',d)`, and then generate a pseudorandom sequence $\{\varepsilon_k\}$ by repeatedly using the command `rand`. In MATLAB, $\{\varepsilon_k\}$ is generated by a simple multiplicative congruential generator [20, Chapter 9]

$$d_0 = \mathtt{d}, \quad d_k = ad_{k-1} + c \bmod M, \text{ for } k \geq 1; \quad \varepsilon_k \equiv d_k/M. \tag{3.1}$$

The parameters in (3.1) are chosen as $a = 16807,\ c = 0,\ M = 2^{31} - 1$, due to Park and Miller [21].

Thus a pseudorandom sequence is actually not random but deterministic, in the sense that it is generated according to some formula and hence can be regenerated exactly if the seed d_0 is known. For example, the MATLAB commands

```
rand('seed',d);
e=rand;
```

will output different `e` if the seed `d` is changing every time, but output the same `e` if `d` is fixed. By extracting and remembering a proper seed, we can regenerate part of a pseudorandom sequence as we desire. More specifically, suppose we have already generated a sequence $\{\varepsilon_k\}_{k=1}^p$, and then we want to regenerate only $\{\varepsilon_k\}_{k=q}^p$, i.e. the part of the sequence beginning at ε_q. All we need is to extract the seed after generating ε_{q-1}. The seed-extracting command in MATLAB is `rand('seed')`. Thus, given the sequence $\{\varepsilon_k\}_{k=1}^p$ generated by

$$\xrightarrow{\texttt{randn}} \varepsilon_1 \ldots \xrightarrow{\texttt{randn}} \varepsilon_{q-1} \xrightarrow{\texttt{c=randn('seed')}} \text{extract seed } \texttt{c} \xrightarrow{\texttt{randn}} \varepsilon_q \ldots \xrightarrow{\texttt{randn}} \varepsilon_p$$

we can regenerate $\{\varepsilon_k\}_{k=p}^q$ by

$$\xrightarrow{\texttt{randn('seed',c)}} \text{set seed } \texttt{c} \xrightarrow{\texttt{randn}} \varepsilon_q \xrightarrow{\texttt{randn}} \varepsilon_{q+1} \xrightarrow{\texttt{randn}} \ldots\ldots \xrightarrow{\texttt{randn}} \varepsilon_p$$

Some computer languages only provide [0,1] uniform random numbers. When we simulate Lévy processes, we will also need to generate non-uniform random variables such as the standard normal random variables, Poisson random variables, and the gamma random variables. Various kinds of methods, say the inverse transform method and the acceptance-rejection method, can be used to obtain non-uniform random variables based on [0,1] uniform random numbers. For standard normal random numbers, the most commonly used method is the Box-Muller transformation, see [12, pp. 235]. For Poisson random variables, the inverse transform method is a standard method, see [13, pp. 128]. For completeness, we provide the Best's generator for the gamma random variables in the Appendix, see also [12, pp. 410 and pp. 420]. We will be using these methods to generate the needed random variables. In the following, we will use $Z \sim \mathcal{N}(0,1)$ and $\varepsilon \sim \mathcal{U}[0,1]$ to denote random numbers Z and ε distributed as standard normal and [0,1] uniform respectively.

4. The memory-reduction method. In this section, we present our memory-reduction method which does not require one to store the intermediate asset prices $\{S_{i,j}\}_{i,j=1}^{n,m}$ when computing the op-

tion prices. In this method, each increment $\Delta L_{i,j}$ is generated twice without being stored while the corresponding intermediate asset price $S_{i,j}$ is generated only once in the backward pricing of the option.

As in the full-storage method, we compute $L_{i,j} = L_{i,j-1} + \Delta L_{i,j}$ by using the increments $\Delta L_{i,j}$. But in our memory-reduction method, we use a different way to generate the set of random numbers $\Sigma_{i,j}$ to obtain $\Delta L_{i,j}$—we generate them time-wise. More precisely, we obtain the increments $\Delta L_{i,1}$ by generating the random numbers in $\Sigma_{i,1}$ on each path i, $i = 1, ..., n$, for the time step $j = 1$ first. Then we obtain $\Delta L_{i,2}$ by generating $\Sigma_{i,2}$ on all paths for $j = 2$, etc. For each time step j, at the last path, i.e. path n, we extract and save the current seed d_j for later use. Given an arbitrary seed d_1, the procedures can be illustrated as follows:

$$
\begin{aligned}
&\text{set seed } d_1 \to \Delta L_{1,1}(\Sigma_{1,1}) \to \Delta L_{2,1}(\Sigma_{2,1}) \to ..... \to \Delta L_{n,1}(\Sigma_{n,1}) \to \\
&\text{extract seed } d_2 \to \Delta L_{1,2}(\Sigma_{1,2}) \to \Delta L_{2,2}(\Sigma_{2,2}) \to ..... \to \Delta L_{n,2}(\Sigma_{n,2}) \to \\
&\text{extract seed } d_3 \to ..... \\
&\vdots \\
&\text{extract seed } d_m \to \Delta L_{1,m}(\Sigma_{1,m}) \to \Delta L_{2,m}(\Sigma_{2,m}) \to ..... \to \Delta L_{n,m}(\Sigma_{n,m})
\end{aligned}
$$

See Phase 1 in the following Algorithm 4.1. Note that we need an m-vector to hold $\{d_j\}_{j=1}^m$ and an n-vector to hold $\{L_{i,j}\}_{i=1}^n$. That n-vector can be re-used for every time step j.

When computing the option price, we move backward in time, and compute on each path i the corresponding asset prices $S_{i,j} = S_0 \exp(rj\Delta t + L_{i,j})$ at each time step j. This requires $L_{i,j}$. Given $L_{i,j+1}$, to obtain $L_{i,j}$, we only need to regenerate $\Delta L_{i,j+1}$. This can be done by reproducing the random number sequence in $\Sigma_{i,j+1}$ using the seed d_{j+1}, i.e.

$$
\text{set seed } d_j \to \Delta L_{1,j+1}(\Sigma_{1,j+1}) \to ..... \to \Delta L_{n,j+1}(\Sigma_{n,j+1})
$$

Once we get all the $S_{i,j}$ for the time step j, we can compute the option prices on all paths at time step j by using the LSM method in [16]. We summarize our memory-reduction method in Algorithm 4.1 below:

ALGORITHM 4.1.
Phase 1 (path simulation):
Set $L_0^i \leftarrow 0$ *for* $i = 1, 2, ..., n$
For-loop: $j = 1, 2, ..., m$
  1. Extract the current seed d_j
  For-loop: $i = 1, 2, ..., n$
    2. Get the increment $\Delta L_{i,j}$ *by generating* $\Sigma_{i,j}$
    3. $L_{i,j} \leftarrow L_{i,j-1} + \Delta L_{i,j}$
  End for-loop
End for-loop

Phase 2 (price computation):
For-loop: $j = m, ..., 1$
  If $j < m$,
    4. Recall the seed d_{j+1}
    For-loop: $i = 1, 2, ..., n$
      5. Get the increment $\Delta L_{i,j+1}$ *by regenerating* $\Sigma_{i,j+1}$
      6. $L_{i,j} \leftarrow L_{i,j+1} - \Delta L_{i,j+1}$
      7. $S_{i,j} \leftarrow S_0 \exp(rj\Delta t + L_{i,j})$
    End for-loop
  End if
  Compute the current option price on all paths using the LSM method
End for-loop

We note that our memory-reduction approach totally requires only three vectors: an m-vector for storing the seeds $\{d_j\}_{j=1}^m$ in Steps 1 and 4, an n-vector to hold $\{L_{i,j}\}_{i=1}^n$ for the current time-step j in

Steps 3 and 6 and an n-vector to hold $\{S_{i,j}\}_{i=1}^n$ for the current time-step j in Step 7. The additional computational burden is Steps 1–4 in Phase 1, where we generate the paths and remember the seeds. Since in Phase 2, we are regenerating the exact paths as those in the full-storage method, it is clear that the results obtained by the full-storage method and the memory-reduction method are exactly the same. Moreover, since path generation is only one part of all the computations required in the algorithm (the other part—the major part— being the least-squares methods of [16]), we see that the total cost of our method is less than twice of that of the full-storage method. We will illustrate these facts numerically in Section 5. We note that in order to use our Algorithm 4.1 for different kinds of options, we only need to specify how $\Delta L_{i,j}$ in Step 2 are generated.

5. Numerical examples. In this section, we apply our method to different models in the class of exponential Lévy processes. In Subsection 5.1, we consider the Black-Scholes model and compare our memory-reduction method with the Brownian-bridge method and also the Crank-Nicolson method. In Subsections 5.2 and 5.3, numerical results are reported for both finite-activity and infinite-activity jump processes, respectively. We compare our results with a binomial tree method and an integro-differential equation method. Regarding the LSM we used, we estimate the continuing values of an option on those "in-the-money" samples and choose the first three Laguerre polynomials plus a constant term as our basis functions throughout the section.

5.1. Black-Scholes model. As an illustration for how to use the memory-reduction method, we begin with the Black-Scholes model:

$$\frac{dS_t}{S_t} = rdt + \sigma dW_t, \tag{5.1}$$

where r is the risk-free rate, σ is the volatility, and W_t is the standard Wiener Process. The memory-reduction method for this simple case was considered in [6], but we repeat here as an introduction to our method. By Itô's lemma, the L_t in (2.1) becomes $L_t = -\frac{1}{2}\sigma^2 t + \sigma W_t$ and hence

$$\Delta L_{i,j} = -\frac{1}{2}\sigma^2 \Delta t + \sigma\sqrt{\Delta t} Z_{i,j} \tag{5.2}$$

where $Z_{i,j} \sim \mathcal{N}[0,1]$. By the Box-Muller transformation, see [12, pp. 235], a pair of $Z_{i,j}$ can be generated by a pair of $\varepsilon_{i,j} \sim \mathcal{N}[0,1]$. Hence here the set $\Sigma_{i,j}$ in Algorithm 4.1 has only one element $\varepsilon_{i,j}$. Now we can apply Algorithm 4.1 by specifying the procedures in Step 2 as follows:

ALGORITHM 5.1 (Black-Scholes).
1. *Generate* $Z_{i,j} \sim \mathcal{N}(0,1)$ *using* $\varepsilon_{i,j} \sim \mathcal{U}[0,1]$
2. $\Delta L_{i,j} \leftarrow -\frac{1}{2}\sigma^2\Delta t + \sigma\sqrt{\Delta t} Z_{i,j}$

Next we compare our memory-reduction method with the Brownian-bridge method in [9] and the Crank-Nicolson method in [24] on pricing American put options under model (5.1). Note that the results obtained by the full-storage method and the memory-reduction method are exactly the same since the same paths are used to price the option. In our test, we choose the risk-free rate $r = 0.1$, the volatility $\sigma = 0.4$, and the expiration date $T = 0.5$ year. In Table 5.1, "CNM" stands for the results computed by the Crank-Nicolson method given in [24]. The means and the standard deviations after 25 trials are shown under "Mean" and "STD" for both the memory-reduction method and the Brownian-bridge method. The two "Error" columns represent the difference between the corresponding "Mean" and "CNM". We observe that the accuracy is almost the same for all methods. Table 5.2 presents the average CPU times for five consecutive trials of each method. We see that our method brings about slight additional cost but significantly reduces the storage requirement when compared with the other two methods. We also observe from Table 5.2 that for all three methods there, the CPU time increases linearly with respect to m and n, if either one is fixed. This is as expected since the CPU times should be increasing like $O(mn)$.

5.2. Merton's jump-diffusion model. Merton's jump-diffusion process [19] can be described by the following stochastic differential equation under risk-neutral measure $\mathbb{Q}$ (generally not unique)

$$\frac{dS_t}{S_{t-}} = rdt + \sigma dW_t + dJ_t - \varpi dt. \tag{5.3}$$

TABLE 5.1
Black-Scholes model with $n = 10^5$ (50,000 plus 50,000 antithetic) and $m = 64$

		Memory-reduction			Brownian- bridge		
S_0	CNM	Mean	STD	Error	Mean	STD	Error
6	4.0000	3.99220	0.00002	−0.00780	3.99220	0.00005	−0.00780
8	2.0951	2.09459	0.00192	−0.00051	2.09311	0.00226	−0.00199
10	0.9211	0.92117	0.00167	0.00007	0.92059	0.00232	−0.00051
12	0.3622	0.36190	0.00208	−0.00030	0.36181	0.00231	−0.00039
14	0.1320	0.13225	0.00125	0.00025	0.13184	0.00127	−0.00016

TABLE 5.2
CPU time in seconds and memory requirement when $S_0 = 10$

m	32			32	64	128	Memory
n	20,000	40,000	80,000	20,000			requirement
Full-storage	4.25	8.59	17.19	4.25	8.50	16.98	$n(m+1)$
Memory-reduction	4.37	8.87	17.74	4.37	8.78	17.53	$m+2n$
Brownian-bridge	4.58	9.22	18.53	4.58	9.21	18.43	$n(\log_2 m+1)$

Here $t-$ denotes the instant immediately before time t, $J_t = \sum_{k=1}^{N_t}(Y_k - 1)$ represents sudden jumps in price evolution, N_t is a Poisson counting process with intensity λ and $\{\log Y_k\}_{k=1}^{N_t}$ are independent and identically distributed $\mathcal{N}(\alpha, \beta^2)$ numbers. Also in (5.3),

$$\varpi = \lambda \mathbf{E}^{\mathbb{Q}}[Y_k - 1] = \lambda[\exp(\alpha + \frac{1}{2}\beta^2) - 1] \tag{5.4}$$

is the compensator such that $\mathbf{E}^{\mathbb{Q}}[\exp(-rt)S_t] = S_0$. Rewriting (5.3) as (2.1), we have

$$L_t = -\frac{1}{2}\sigma^2 t + \sigma W_t + \sum_{k=1}^{N_t} \log(Y_k) - \varpi t. \tag{5.5}$$

Thus for Merton's jump-diffusion model, Step 2 in Algorithm 4.1 is

ALGORITHM 5.2 (Merton).
1. *Generate $N_{i,j} \sim$Poisson$(\lambda\Delta t)$ using the inverse method [13, pp. 128]*
2. *Generate $Z^1_{i,j} \sim \mathcal{N}(0,1)$*
3. *If $N_{i,j} > 0$, generate $Z^2_{i,j} \sim \mathcal{N}(0,1)$*
4. $\Delta L_{i,j} \leftarrow (-\frac{1}{2}\sigma^2 - \varpi)\Delta t + \sigma\sqrt{\Delta t} Z^1_{i,j} + \alpha N_{i,j} + \beta\sqrt{N_{i,j}} Z^2_{i,j}$

Now we test our method on an American put option under Merton's jump-diffusion model. The underlying stock price S_0 at the current time is \$40. The parameter values are $r = 8\%$, $\sigma = \sqrt{0.05}$, $\lambda = 5$, and $\beta = \sqrt{0.05}$. We let $\alpha = -\frac{1}{2}\beta^2$ such that $\mathbf{E}^{\mathbb{Q}}[Y_i] = 1$. The numerical results are reported in Table 5.3 where the columns "Mean" and "STD" are the means and the standard deviations obtained after 25 trials. We use the 200-time-step discrete time binomial tree model in [1] as a benchmark and it is listed under the heading "Amin's". We observe that the two methods agree up to 2 decimals. Table 5.4 gives the average CPU times for five consecutive runs of the methods. Again the CPU time by our method is always less than twice of that by the full-storage method.

5.3. Variance gamma model. A variance gamma (VG) process [17] with parameters $\mu \in \mathbb{R}$, $\sigma > 0$, and $\nu > 0$ can be represented as a time-changed Brownian motion. Let $B_t = \mu t + \sigma W_t$ be a Brownian motion with drift μ and volatility σ. Define a gamma process G_t with independent gamma increments of mean h and variance νh over any non-overlapping time intervals of length h, or $G_t \sim \gamma(\frac{t}{\nu}, \nu) \sim \nu\gamma(\frac{t}{\nu})$. Then the three-parameter VG process X_t is defined by $X_t = B_{G_t}$ and its characteristic function is

$$\Phi_{X_t}(u) = \mathbf{E}[\exp(iuX_t)] = \left(\frac{1}{1 - i\mu\nu u + \frac{1}{2}\sigma^2\nu u^2}\right)^{\frac{t}{\nu}}. \tag{5.6}$$

TABLE 5.3
Merton's model with $n = 10^5$ and $m = T/0.01$

Strike K	Amin's	Mean	STD	Error
Expiring time $T = 0.25$ year				
30	0.674	0.6741	0.0064	0.0001
35	1.688	1.6872	0.0121	−0.0008
40	3.630	3.6248	0.0174	−0.0052
45	6.734	6.7288	0.0256	−0.0052
50	10.696	10.6867	0.0203	−0.0093
Expiring time $T = 1$ year				
30	2.720	2.7191	0.0132	−0.0009
35	4.603	4.6064	0.0204	0.0034
40	7.030	7.0242	0.0199	−0.0058
45	9.954	9.9461	0.0326	−0.0079
50	13.318	13.3050	0.0326	−0.0130

TABLE 5.4
CPU time in seconds and memory requirement when $T = 1, K = 40$

m	50			50	100	200	Memory
n	20,000	40,000	80,000	20,000			requirement
Full-storage	22.05	43.86	87.77	22.05	43.52	86.88	$n(m+1)$
Memory-reduction	36.93	73.04	146.35	36.93	73.14	146.06	$m+2n$

Accordingly, the asset price process S_t is modeled as

$$S_t = S_0 \exp((r-q)t + X_t - \varpi t) \tag{5.7}$$

under the risk-neutral measure $\mathbb{Q}$ (generally not unique) with a continuous dividend yield of q and a constant continuously compounded interest rate of r. In model (5.7), the risk-neutral drift rate is $r-q$ and the compensator ϖ satisfies $\exp(\varpi) = \mathbf{E}^{\mathbb{Q}}[\exp(X_t)]$ such that $\mathbf{E}^{\mathbb{Q}}[\exp(-(r-q)t)S_t] = S_0$. By evaluating $\Phi_{X_t}(u)$ at $-i$, we have

$$\varpi = -\frac{1}{\nu}\ln(1 - \mu\nu - \frac{1}{2}\sigma^2\nu). \tag{5.8}$$

Thus Step 2 in Algorithm 4.1 becomes:

ALGORITHM 5.3 (variance gamma).
1. *Generate $Z_{i,j} \sim \mathcal{N}(0,1)$*
2. *Generate $\Delta G_{i,j} \sim \gamma(\frac{\Delta t}{\nu})$ using Best's generator given in Algorithm 7.1*
3. *$\Delta L_{i,j} \leftarrow \mu\Delta G_{i,j} + \sigma\sqrt{\Delta G_{i,j}}Z_{i,j} - \varpi\Delta t$*

Now consider an American put option with maturity $T = 0.56164$ written on a stock with current price $S_0 = 1369.41$. The VG parameters after model calibration are given by $r = 0.0541$, $q = 0.012$, $\sigma = 0.20722$, $\nu = 0.50215$, and $\theta = -0.22898$. We test our method on various strike prices K and with $m = 56 \approx T/0.01$. The results are presented in Table 5.5. For comparison, results obtained by the partial integro-differential equation approach in [14] are given under "PIDE". As usual, the "Mean" and "STD" are the means and the standard deviations, respectively, obtained after 25 trials. The difference between "Mean" and "PIDE" are computed in the column "Error". Again, the numerical results confirm the accuracy of our method. The average CPU times of five consecutive trials are given in Table 5.6, and the CPU time by our method is again bounded above by twice of that by the full-storage method.

5.4. Remarks on the efficiency of the memory-reduction method. In the above three subsections, we have illustrated how to apply our memory-reduction method to specific exponential Lévy models. For both the full-storage method and the memory-reduction method, the computational cost is

TABLE 5.5
Variance gamma model with $n = 10^5$ and $m = 56$

Strike K	PIDE	Mean	STD	Error
1200	35.530	35.363	0.288	−0.167
1260	48.798	48.642	0.306	−0.156
1320	65.991	65.850	0.404	−0.141
1380	87.991	87.777	0.345	−0.214

TABLE 5.6
CPU time in seconds and memory requirement when $K = 1320$

m	50			50	100	200	Memory
n	20,000	40,000	80,000	20,000			requirement
Full-storage	58.61	117.41	234.93	58.61	118.58	240.12	$n(m+1)$
Memory-reduction	112.53	225.34	450.73	112.53	229.05	462.36	$m+2n$

composed of two parts: the cost in path simulation and the cost in price computation. Compared with the full-storage method, the cost in path simulation is almost doubled in the memory-reduction method while the cost in price computation of both methods are the same. Hence our method always uses less than twice the time required by the full-storage method. In the following, we mention two factors affecting this overhead cost.

In Table 5.7, we give the ratio of the timing between the two methods in the "Ratio" rows for $m = 50$ and $n = 20,000$. In the table, the number in the square bracket $[\cdot]$ for each model is the average CPU time in seconds for generating 1,000 sample paths with 50 time steps. We observe from the table that the cost in path simulation in the Black-Scholes model is much less than that in the variance gamma model. As a consequence, our memory-reduction method almost produces no additional computational cost in the Black- Scholes model while in the variance gamma model, the CPU time of our method nearly doubles that of the full-storage method.

Another factor is the number of $S_{i,j}$ that are in-the-money. The rows "In-the-money (%)" in Table 5.7 count the average percentages of those "in-the-money" $S_{i,j}$ in the $m \cdot n$ samples in 5 trials. As the difference $K - S_0$ goes up, the number of "in-the-money" samples goes up, which leads to an increase in the cost of price computation. Consequently, the ratio goes down.

TABLE 5.7
CPU time in seconds with $m = 50, n = 20,000$

Black-Scholes model [0.0331]					
S_0	6	8	10	12	14
"In-the-money" (%)	98.9	87.3	49.0	15.5	4.7
Full-storage	13.8	11.52	6.68	2.74	1.48
Memory-reduction	14.11	11.78	6.89	2.87	1.61
Ratio	1.022	1.023	1.031	1.047	1.088
Merton's model ($T = 1$) [1.62]					
Strike K	30	35	40	45	50
"In-the-money" (%)	21.6	33.6	52.4	69.7	79.1
Full-storage	17.94	19.37	21.65	23.71	24.87
Memory-reduction	32.31	33.86	36.02	38.11	39.29
Ratio	1.801	1.748	1.664	1.607	1.580
Variance gamma model [3.85]					
Strike K	1200	1260	1320	1380	
"In-the-money" (%)	11.8	16.3	23.1	37.2	
Full-storage	57.90	58.51	59.37	61.04	
Memory-reduction	112.47	113.18	113.91	115.61	
Ratio	1.942	1.934	1.919	1.894	

6. Conclusion. In this paper, we propose a new simulation technique for pricing American options under exponential Lévy processes. It reduces the storage requirement to $O(m+n)$. For machines with limited memory, we can now enlarge m and n to improve the accuracy of the pricing. Furthermore, our memory-reduction method can easily be extended to pricing other path-dependent options with early-exercise features, say the Asian Bermudan options, as well as the multi-asset American options. Hence, our method can be valuable in investigating option prices, especially those written on single or multiple assets with complex American triggers, long-term options, or any combination of these properties. We also remark that our memory reduction method adopts a natural extension to other relevant models such as stochastic volatility models, as long as the forward-path method (with no memory reduction) uses pseudorandom numbers in Monte Carlo simulation. Yet, the implementation becomes somehow more subtle, as differential levels of randomness arise. We plan to consider such extensions in our future work.

7. Appendix. For completeness, here we give the algorithm for generating the gamma random variables. We also give the commands in FORTRAN and MATHEMATICA for finding the seeds of a sequence of random numbers.

Algorithm 7.1 below generates Gamma random variables $\gamma(a)$ with density

$$p(x) = \frac{x^{a-1}}{\Gamma(a)} e^{-x}$$

when $a \geq 1$. For $a < 1$, one uses the transformation $\gamma(a) = \gamma(1+a)U^{1/a}$ with $U \sim \mathcal{U}[0,1]$. See [12, pp. 410 and pp. 420] for a comprehensive discussion.

ALGORITHM 7.1 (Best's generator).
1. $b \leftarrow a-1$, $c \leftarrow 3a - \frac{3}{4}$
Repeat
  2. Generate random variables $U, V \sim \mathcal{U}[0,1]$
  3. $W \leftarrow U(1-U)$, $Y \leftarrow \sqrt{\frac{c}{W}}(U - \frac{1}{2})$, $X \leftarrow b + Y$
  4. If $X < 0$, *go to Repeat*
  5. $Z \leftarrow 64W^3V^2$
Until $\log(Z) \leq 2b\log(\frac{X}{b} - Y)$
Return X

In FORTRAN 90 [8], the command to get a $\mathcal{U}[0,1]$ number is `rand()`. The commands to set the seed to `d` are:

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

where `k` is the number of 32-bit words used to hold the seed. The commands to extract the current seed `d` are:

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

In MATHEMATICA [25], the seeds are set by "`SeedRandom[d]`". To extract the current seed, use "`c=$RandomState`". MATHEMATICA provides $\mathcal{U}[0,1]$ numbers with the command "`Random[ ]`".

REFERENCES

[1] K.I. Amin, *Jump diffusion option valuation in discrete time*, Journal of Finance, 48 (1993), pp. 1833–1863.
[2] O. Barndorff-Nielsen, *Processes of normal inverse Gaussian type*, Finance and Stochastics, 2 (1997), pp. 41–68.
[3] F. Black and M. Scholes, *The pricing of options and corporate liabilities*, Journal of Political Economy, 81 (1973), pp. 637–654.
[4] P. Boyle, *Option: a Monte Carlo approach*, Journal of Financial Economics, 4 (1977), pp. 323–338.

[5] P. Carr, H. Geman, D. Madan, and M. Yor, *The fine structure of asset returns: an empirical investigation*, Journal of Business, 75 (2002), pp. 305–332.
[6] R.H. Chan, Y. Chen and K.M. Yeung, *A memory reduction method in pricing American options*, Journal of Statistical Computation and Simulation, 74 (2004), pp. 501–511.
[7] R.H. Chan, C.Y. Wong and K.M. Yeung, *Pricing multi-asset American-style options by memory reduction Monte Carlo methods*, Applied Mathematics and Computation, 179 (2006), pp. 535–544.
[8] S. Chapman (1998), *Fortran 90/95 for scientists and engineers*, McGraw-Hill.
[9] S.K. Chaudhary, *American options and the LSM algorithm: quasi-random sequences and Brownian bridges*, Journal of Computational Finance, 8 (2005), pp. 101–115.
[10] R. Cont, *Empirical properties of asset returns: stylized facts and statistical issues*, Quantitative Finance, 1 (2001), pp. 1–14.
[11] R. Cont and P. Tankov, *Financial Modelling with Jump Processes*, Chapman & Hall/CRC Press, London, 2004.
[12] L. Devroye, *Non-Uniform Random Variate Generation*, Springer Verlag, New York, 1986.
[13] P. Glasserman, *Monte Carlo Methods in Financial Engineering*, Springer Verlag, New York, 2003.
[14] A. Hirsa and D.B. Madan, *Pricing American options under variance gamma*, Journal of Computational Finance, 7 (2003), pp. 63–80.
[15] S.G. Kou, *A jump-diffusion model for option pricing*, Management Science, 48 (2002), pp. 1086–1101.
[16] F.A. Longstaff and E.S. Schwartz, *Valuing American options by simulation: a simple least-squares approach*, The Review of Financial Studies, 14 (2001), pp. 113–147.
[17] D.B. Madan, P.P. Carr, and E.C. Chang, *The variance gamma process and option pricing*, European Finance Review, 2 (1998), pp. 79–105.
[18] B.B. Mandelbrot, *The variation of certain speculative prices*, Journal of Business, XXXVI (1963), pp. 392–417.
[19] R.C. Merton, *Option pricing when underlying stock returns are discontinuous*, Journal of Financial Economics, 3 (1976), pp. 125–144.
[20] C. Moler, *Numerical Computing with MATLAB*, SIAM, Philadelphia, 2004.
[21] S.K. Park and K.W. Miller, *Random number generators: good ones are hard to find*, Communications of the ACM, 31 (1988), pp. 1192–1201.
[22] C. Ribeiro and N. Webber, *A Monte Carlo method for the normal inverse Gaussian option valuation model using an inverse Gaussian bridge*, working paper, Cass Business School, City University, 2003.
[23] C. Ribeiro and N. Webber, *Valuing path-dependent options in the variance gamma model by Monte Carlo with a gamma bridge*, Journal of Computational Finance, 7 (2004), pp. 81–100.
[24] P. Wilmott, S. Howison, and J. Dewynne, *The Mathematics of Financial Derivatives*, Cambridge University Press, Cambridge, 1998.
[25] S. Wolfram, *The Mathematica Book, 4th ed.*, Cambridge University Press, Cambridge, 1999.