Kaneko et al. *EPJ Quantum Technology* (2022) 9:7
https://doi.org/10.1140/epjqt/s40507-022-00125-2

EPJ Quantum Technology
a SpringerOpen Journal

RESEARCH Open Access

Quantum pricing with a smile: implementation of local volatility model on quantum computer

Kazuya Kaneko[1*], Koichi Miyamoto[1,2], Naoyuki Takeda[1] and Kazuyoshi Yoshino[1]

*Correspondence:
kazuya-kaneko@fintec.co.jp
[1] Mizuho-DL Financial Technology Co., Ltd., Tokyo, Japan
Full list of author information is available at the end of the article

Abstract

Quantum algorithms for the pricing of financial derivatives have been discussed in recent papers. However, the pricing model discussed in those papers is too simple for practical purposes. It motivates us to consider how to implement more complex models used in financial institutions. In this paper, we consider the local volatility (LV) model, in which the volatility of the underlying asset price depends on the price and time. As in previous studies, we use the quantum amplitude estimation (QAE) as the main source of quantum speedup and discuss the state preparation step of the QAE, or equivalently, the implementation of the asset price evolution. We compare two types of state preparation: One is the *amplitude encoding* (AE) type, where the probability distribution of the derivative's payoff is encoded to the probabilistic amplitude. The other is the *pseudo-random number* (PRN) type, where sequences of PRNs are used to simulate the asset price evolution as in classical Monte Carlo simulation. We present detailed circuit diagrams for implementing these preparation methods in fault-tolerant quantum computation and roughly estimate required resources such as the number of qubits and T-count.

Keywords: Finance; Pricing; Quantum computing

1 Introduction

With recent advances in quantum computing technologies, researchers are beginning to consider how to utilize them in industries. Finance is one of the major target [1]. Because financial institutions perform enormous tasks of numerical calculation in their daily works, the speedup of such calculation will bring significant benefits to them. One of such tasks is the pricing of financial derivatives [2–4]. Financial derivatives, or simply derivatives, are contracts in which payoffs are determined in reference to the prices of underlying assets at some fixed times.

In derivative pricing, movements of underlying asset prices are represented by stochastic processes, and a derivative price is written as an expected value of the sum of payoffs discounted by the risk-free interest rate. Monte Carlo simulation is often used to compute the derivative price, but it takes a computation long time. Quantum algorithms for Monte Carlo integration [5, 6] bring quadratic speedup compared with classical Monte Carlo algorithms, and several previous studies discuss their application to derivative pricing [7–10]. Although previous studies consider the Black-Scholes (BS) model [11, 12], which is the pioneering model for derivative pricing, it is inappropriate as an application target of Monte Carlo for practical business for the following reasons. First, the actual market prices of derivatives are inconsistent with the BS model. This phenomenon is called *volatility smile*, which we will explain in Sect. 2. To price derivatives precisely, financial firms often use more complicated models than the BS models. Second, the BS model is so simple that analytic formulae are available, and thus Monte Carlo simulation is not necessary. In fact, banks use Monte Carlo simulation mainly for complex models which can take into account volatility smiles. The above points motivate us to consider the advanced models in quantum algorithms.

This paper focuses on one of the advanced models, the local volatility (LV) model [13]. In the LV model, the volatility of an asset price depends on the price itself and time. The BS model is also a special case of the LV model. Because the LV model can make derivative prices consistent with volatility smiles, it is widely used for pricing derivatives, especially exotic derivatives, which have complex transaction terms such as early redemption. In order to price a derivative by Monte Carlo simulation, we generate random trajectories (paths) of the time evolution of asset prices, then calculate the expectation value of the sum of discounted payoffs in each path. In this paper, we focus on the implementation of such a time evolution in the LV model on fault-tolerant quantum computers to apply quantum algorithms for Monte Carlo simulation.

We consider two quantum integration algorithms based on the quantum amplitude estimation (QAE): the amplitude encoding (AE) type method [5] and the pseudo-random number (PRN) type method [6]. These algorithms are the same in that we prepare a quantum state encoding the integrand and estimate the integral from the state by the QAE. The difference between these algorithms is whether the probability distribution is encoded to the amplitude of a quantum state. In the AE-type method, which is adopted in previous studies [7–9], the probability distribution of the payoff is fully encoded to the probability amplitude [14]. In other words, this method takes account of all possible paths in calculating the expectation value. A problematic point of the AE-type method is that the number of qubits grows with the dimension of the integrand. In the pricing task, the number of qubits is proportional to the total number of random variables, which equals the length of the path times the number of underlying assets.[1] Because the length of the path, i.e., the number of time steps, can be large for derivatives with a long maturity, and the number of underlying assets can be multiple, the AE-type method will require many qubits. Let us see a common situation: the number of assets is $\mathcal{O}(10)$, that of time steps is $\mathcal{O}(10^2)$, and each register for random variables consists of $\mathcal{O}(10)$ qubits. Then, the total number of qubits for the derivative pricing becomes $\mathcal{O}(10^4)$. Since the large qubit overhead might incur to make a logical qubit (see Ref. [15] and references therein), calculations with a large number of logical qubits might be prohibitive.

The PRN-type method is originally proposed in Ref. [6] to reduce the number of qubits for integrating multivariate functions. In the PRN-type method, we do not encode the probability distribution to the probability amplitude, while we use PRNs whose empirical distribution reproduces the desired probability distribution as in the classical Monte Carlo

---

[1] In this paper, we assume arbitrage-free and complete markets, so the number of stochastic factors equals that of assets.

simulation. Although this method introduces an additional error in the estimation, we can reduce the error by increasing the number of sampled paths. As shown in Ref. [6], we can achieve the quadratic speedup by appropriately changing the number of sampled paths. Moreover, this approach allows us to sequentially update PRNs at each time step. In other words, we do not need to have multiple random variables simultaneously. In the PRN-type method, each quantum register is not assigned to each of the random variables, but a single register is used to generate a sequence of PRNs. Thus, the number of qubits is independent of the number of random variables, which is the advantage of the PRN-type method. On the other hand, its drawback is the increase of the circuit depth. More concretely, the circuit depth is proportional to the number of random variables. When it comes to the LV model, the circuit depth is proportional to the number of time steps in both methods, and thus the drawback of the PRN-type method will be alleviated. This is different from the situation in credit portfolio risk management [16], where the AE-type method reduces the circuit depth.

Furthermore, we design the quantum circuits implementing the above state preparation methods in the fault-tolerant quantum computer by using several quantum circuits for elementary arithmetic. We then estimate the number of logical qubits[2] and T-count [17, 18] in the proposed quantum circuits. Because the qubit number in the PRN-type method is independent of the number of time steps, it is much less than that in the AE-type method. On the other hand, the T-count is proportional to the number of time steps in both methods. However, the T-count of the PRN-type method is larger than that of the AE-type method by a factor of $\mathcal{O}(1)$.

The rest of this paper is organized as follows. Section 2 and 3 are preliminary sections, the former briefly explains the LV model, and the latter reviews the quantum algorithm for Monte Carlo simulation. In Sect. 4, we present quantum circuits for the state preparation in two methods. In Sect. 5, we estimate the qubit number and T-count of the proposed circuits. Section 6 gives a summary.

2 Local volatility model

This section is devoted to defining the LV model.

2.1 Pricing of derivatives

We consider the single-asset case, but it is straightforward to extend the discussion in this paper to the multi-asset case. Consider a party A involved in a derivative contract written on some asset. Let S_t be a stochastic process representing the asset price at time t. We assume that the payoffs arise multiple times t_i^{pay}, $i = 1, 2, \ldots$, and the i-th payoff is given by $f_i^{\text{pay}}(S_{t_i^{\text{pay}}}) \in \mathbb{R}$. Here, the positive payoff means that A receives a money from the counterparty, and the negative one means vice versa. For example, when A buys an European call option with the strike K, the payoff is given by

$$f_1^{\text{pay}}(S_{t_1^{\text{pay}}}) = \max\{S_{t_1^{\text{pay}}} - K, 0\} \tag{1}$$

with a single payment date t_1^{pay}. Note that this type of derivative contract is too simple to cover all trades in financial markets. For example, *callable* contracts, in which either

---

[2] Hereafter, we use the word 'qubit' to mean a logical qubit.

of the parties has a right to terminate the contract at some time, are widely dealt with in markets. In this paper, we consider only derivatives expressed as Eq. (1) and leave studies for exotic derivatives for future works.

Following the theory of arbitrage-free pricing [3, 4], the price V of the contract for A is given by

$$V = \mathbb{E}\left[\sum_i f_i^{\mathrm{pay}}(S_{t_i^{\mathrm{pay}}})\right], \tag{2}$$

where $\mathbb{E}[\cdots]$ represents the expectation value under a risk-neutral measure. We assume that the risk-free interest rate is 0 for simplicity.

2.2 LV model and volatility smile

In the LV model, the evolution of the asset price is modeled by the following stochastic differential equation:

$$dS_t = \sigma(t, S_t)\, dW_t \tag{3}$$

in the risk-neutral measure,[3] where W_t is the Wiener process which drives S_t. dX_t is the increment of a stochastic process X_t over an infinitesimal time interval dt, and $\sigma(t, S_t)$ ($\geq$0) represents the local volatility. The BS model corresponds to the case where

$$\sigma(t, S) = \sigma_{\mathrm{BS}} S \tag{4}$$

with a positive constant σ_{BS}, which we call a BS volatility. In the BS model, a price of a European call option with strike K and maturity T at $t = 0$ is given by the following formula:

$$\begin{aligned} &V_{\mathrm{call,BS}}(T, K, S_0, \sigma_{\mathrm{BS}}) = \Phi_{\mathrm{SN}}(d_1)S_0 - \Phi_{\mathrm{SN}}(d_2)K, \\ &d_1 := \frac{1}{\sigma_{\mathrm{BS}}\sqrt{T}}\left[\ln\left(\frac{S_0}{K}\right) + \frac{1}{2}\sigma_{\mathrm{BS}}^2 T\right], \\ &d_2 := d_1 - \sigma_{\mathrm{BS}}\sqrt{T}, \end{aligned} \tag{5}$$

where Φ_{SN} is the cumulative distribution function (CDF) of the standard normal distribution. If the BS volatility is given, we can price the option by the above equations. Conversely, we can calculate the BS volatility from the market price of the option $V_{\mathrm{call,mkt}}(T, K)$. The BS volatility determined from the market price is called implied volatility. That is, the implied volatility $\sigma_{\mathrm{IV}}(T, K)$ is defined through

$$V_{\mathrm{call,BS}}\bigl(T, K, S_0, \sigma_{\mathrm{IV}}(T, K)\bigr) = V_{\mathrm{call,mkt}}(T, K). \tag{6}$$

If the market is described well by the BS model, $\sigma_{\mathrm{IV}}(T, K)$ depends on neither K nor T. However, $\sigma_{\mathrm{IV}}(T, K)$ varies with K and T in many markets. If $\sigma_{\mathrm{IV}}(T, K)$ obtained from the market depends on K, it is said that we observe *volatility smile* for the market. Volatility

---

[3] Note that the drift term does not exist because we set the risk-free to be 0.

smile implies that possible scenarios of asset price evolution in the BS model do not match those which market participants consider. The volatility smile arises when, for example, market participants think that extreme scenarios, such as big crashes or sharp rises, occur more frequently than the BS model prediction.

The LV model allows pricing of a European option to be consistent with a market price as long as there is no arbitrage in the market. This is because, in the LV model, the local volatility $\sigma(t,S)$ has enough degrees of freedom to reproduce the two-dimensional function $V_{\mathrm{call,mkt}}(T,K)$. In fact, if $V_{\mathrm{call,mkt}}(T,K)$ is given for any T and K, we can determine the local volatility as described in Ref. [13]. However, in reality, the market option prices are available only for limited strikes and maturities. Therefore, in practical situations, we assume the functional form of $\sigma(t,S)$ as follows. We set $n_{\mathrm{t}}+1$ grid points in the time axis, $t_0 := 0 < t_1 < \cdots < t_{n_{\mathrm{t}}}$, and set n_S grid points in the asset price axis for each time grid point, $-\infty < s_{i,1} < \cdots < s_{i,n_S} < \infty$. Then, $\sigma(t,S)$ is set as a piecewise-linear function on S:

$$\sigma(t,S) = a_{i,j}S + b_{i,j} \quad \text{for } t_{i-1} \le t < t_i, s_{i,j-1} \le S < s_{i,j}, \tag{7}$$

where $a_{i,j}$ and $b_{i,j}$ are constants. In this paper, we assume that $a_{i,j}$ and $b_{i,j}$ are predetermined constants.

2.3 Monte Carlo simulation

We here describe how to estimate the derivative price (2) by Monte Carlo simulation. First, we discretize the time into sufficiently small meshes because we can deal with a continuous variable on neither classical nor quantum computers. For simplicity, we set the time grid points to $\{t_i\}_{i=0}^{n_{\mathrm{t}}}$. Then, the time evolution (3) is approximated as

$$\Delta S_{t_i} := S_{t_{i+1}} - S_{t_i} \approx \sigma(t_i, S_{t_i})\sqrt{\Delta t_i} w_i, \tag{8}$$

where $\Delta t_i := t_{i+1} - t_i$, and $w_1, \ldots, w_{n_{\mathrm{t}}}$ are mutually independent standard normal random numbers (SNRNs). Among various ways to discretize the stochastic differential equation, we here adopt the Euler-Maruyama method [19].

Second, we discretize SNRNs. Since discretized SNRN takes on a countable number of values, we denote the m-th value of the discretized SNRN by $w^{(m)}$. The associated probability mass function p_m is defined as the cumulative distribution of the standard normal distribution over a small interval of two grid points. Then, we can approximate Eq. (2) as

$$V \approx \sum_{\boldsymbol{m}} p_{\boldsymbol{m}} \sum_i f_i^{\mathrm{pay}}\left(S^{(\boldsymbol{m})}_{t_i^{\mathrm{pay}}}\right), \tag{9}$$

where $\boldsymbol{m} := (m_1, \ldots, m_{n_{\mathrm{t}}})$ and $S_t^{(\boldsymbol{m})}$ is the asset price at time t when SNRNs take values $w_1^{(m_1)}, \ldots, w_{n_{\mathrm{t}}}^{(m_{n_{\mathrm{t}}})}$.

There are several ways to calculate the right-hand side of Eq. (9). The simplest way is brute force calculation, but it takes an exponentially long calculation time. In fact, if we take M grids to discretize each SNRN, the total number of grid points is $M^{n_{\mathrm{t}}}$. To overcome this problem, usually, Monte Carlo method is used. In Monte Carlo simulation, we generate finite but sufficiently many discretized samples of SNRNs $(w_1^{(n)}, \ldots, w_{n_{\mathrm{t}}}^{(n)})$ and use them to generate sample paths of the asset price evolving according to Eq. (8). Then, Eq. (2) is approximated by the average of sums of payoffs in sample paths:

$$V \approx \frac{1}{N_{\mathrm{samp}}} \sum_{n=1}^{N_{\mathrm{samp}}} \sum_{i} f_i^{\mathrm{pay}} \bigl(S^{(n)}_{t_i^{\mathrm{pay}}} \bigr). \tag{10}$$

Here, $S_t^{(n)}$ is the value of the asset price at time t on the n-th sample path, and N_{samp} denotes the number of sample paths.

3 Quantum algorithm for Monte Carlo simulation

In this section, we review two quantum methods for Monte Carlo simulation. We consider a problem of numerically estimating a weighted average of a given function $f(s)$, that is, $V := \sum_m p_m f(s_m)$. Here, s_m represents an m-th value of a discretized random variable, and p_m is the probability that it takes a realization s_m. Equation (9) is a special case of this problem, where the integrand is $f(\cdot) = \sum_i f_i^{\mathrm{pay}}(\cdot)$.

3.1 AE-type method

We first review the AE-type method discussed in Ref. [5], which directly encodes p_m to the amplitude. It consists of the following three steps: First, we create a superposition of the inputs and the integrand values with amplitudes $\sqrt{p_m}$, that is, $\sum_m \sqrt{p_m} |s_m\rangle |f(s_m)\rangle$. This step is called the state preparation step, and we need an oracle calculating $f(s)$. Second, the integrand values are encoded to amplitudes of an ancillary qubit by a controlled rotation. The quantum state is transformed as follows:

$$\begin{aligned} |0\rangle|0\rangle|0\rangle &\rightarrow \Bigl(\sum_m \sqrt{p_m} |s_m\rangle \bigl| f(s_m) \bigr\rangle \Bigr) |0\rangle \\ &\rightarrow \sum_m \sqrt{p_m} |s_m\rangle \bigl| f(s_m) \bigr\rangle \bigl(\sqrt{1 - f(s_m)} |0\rangle + \sqrt{f(s_m)} |1\rangle \bigr). \end{aligned} \tag{11}$$

Here, the first, second, and third ket refer to the random number register, the integrand register, and the ancilla, respectively. Finally, quantum amplitude estimation [20–24] on the ancilla gives an approximation of the desired value V. We note that the AE-type method does not directly use classical Monte Carlo approximation like Eq. (10), but the estimation error induced by the QAE.

In this method, the number of calls to an oracle calculating $f(s)$ is $\mathcal{O}(\epsilon^{-1})$ with an estimation error of $\epsilon > 0$. Thus, the quantum algorithm is quadratically faster than the classical Monte Carlo algorithm, which requires $\mathcal{O}(\epsilon^{-2})$ calls. In the case of a multivariate integrand, the AE-type method requires as many random number registers as input variables. Thus, the number of qubits grows with the dimension of the integrand.

3.2 PRN-type method

We here review the PRN-type quantum method for Monte Carlo integration [6], where we prepare a state different from Eq. (11) by using the PRN generator. We first consider the case of the univariate integrand. Let $\{x_j\}_{j=0}^{\infty}$ be a PRN sequence where relative frequency of $x_j = s_m$ equals p_m. Since a PRN sequence usually corresponds to the uniform distribution, we use some transformation techniques such as inverse transform sampling if necessary.

Then, V can be approximated as $V \approx \tilde{V} := N_{\mathrm{samp}}^{-1} \sum_{j=1}^{N_{\mathrm{samp}}} f(x_j)$ by Monte Carlo sampling. The error of the approximation scales as $N_{\mathrm{samp}}^{-1/2}$. This approximation is the core of the PRN-type method, which estimates $\tilde{V}$ by the QAE instead of directly estimating V. In the PRN-type method, we prepare a quantum state encoding $f(x_j)$:

$$|0\rangle|0\rangle|0\rangle \to \left(\frac{1}{\sqrt{N_{\mathrm{samp}}}} \sum_{j=1}^{N_{\mathrm{samp}}} |x_j\rangle|f(x_j)\rangle \right)|0\rangle \tag{12}$$

and apply a controlled rotation and the QAE as in the AE-type method. Although there are two error sources in the PRN-type method, by setting the number of samples $N_{\mathrm{samp}} = \mathcal{O}(\epsilon^{-2})$, we obtain quadratic speedup over classical Monte-Carlo integration.

In contrast to the AE-type method, the number of qubits does not depend on the dimension of the integrand in the PRN-type. Let us consider a multivariate function $f(s_1,\dots,s_n)$. We assume that we can calculate $f(s_1,\dots,s_n)$ sequentially, that is, $y_n = f(s_1,\dots,s_n)$ is calcultaed as

$$y_j = f_j(y_{j-1}, s_j) \quad \text{for } j = 1,2,\dots,n \tag{13}$$

with $y_0 = 0$ and two dimentional functions $\{f_j\}_{j=1}^{n}$. In calculating a sequential function, we do not need to simultaneously keep the input values $s_1,\dots,s_n$. The PRN-type method utilizes this property to reduce the number of qubits in integrating the multivariate function. To calculate integral of $f(s_1,\dots,s_n)$ with PRNs, we divide a PRN sequence $\{x_j\}_{j=0}^{\infty}$ into N_{samp} subsequences of length n, i.e., $\{x_j\}_{j=1}^{n},\dots,\{x_j\}_{j=n(N_{\mathrm{samp}}-1)+1}^{nN_{\mathrm{samp}}}$. Then, the integral is calculated as

$$\tilde{V} = \frac{1}{N_{\mathrm{samp}}} \sum_{i=1}^{N_{\mathrm{samp}}} y_n^{(i)}, \tag{14}$$

where $y_j^{(i)} = f_j(y_{j-1}^{(i)}, x_{(i-1)n+j})$ and i is the label of the subsequence. To realize sequential calculation in the PRN-type state preparation, we replace the random number register with two registers R_{samp} and R_{PRN}, where R_{samp} stores the label of the subsequence and R_{PRN} stores an element of the subsequence, i.e., an PRN. We note that the number of qubits in R_{samp} and R_{PRN} is independent of n, i.e., the dimension of integrand f. Then, the PRN-type state preparation with sequential calculation is as follows:

1. create an equiprobable superposition of labels of subsequences on R_{PRN}: $|0\rangle \to N_{\mathrm{samp}}^{-1/2} \sum_i |i\rangle$.
2. generate a PRN on R_{PRN}: $|i\rangle|0\rangle \to |i\rangle|x_{(i-1)n+1}\rangle$.
3. calculate f_1 and write its value to the integrand register: $|x_{(i-1)n+1}\rangle|0\rangle \to |x_{(i-1)n+1}\rangle|y_1^{(i)}\rangle$.
4. update a PRN: $|x_{(i-1)n+1}\rangle \to |x_{(i-1)n+2}\rangle$.
5. iterate operations 3 and 4 for $j = 1,\dots,n$.

Finally, we obtain the desired state:

$$|0\rangle|0\rangle|0\rangle \to \frac{1}{\sqrt{N_{\mathrm{samp}}}} \sum_i |i\rangle|x_{in}\rangle|y_n^{(i)}\rangle. \tag{15}$$

Since no additional error to the aforementioned PRN-type method arises in this sequential calculation, this method provides a quadratic speedup compared to the classical calculation and uses a smaller number of qubits than the AE-type algorithm, which needs $\mathcal{O}(n)$ qubits. The drawback of the PRN-type method is the $\mathcal{O}(n)$ increase in circuit depth.

3.3 Remarks

To conclude this section, we would like to give some remarks. Even in classical computation, we can achieve quadratic speedup over the classical Monte Carlo integration by using low-discrepancy sequences instead of (pseudo) random numbers. This algorithm is known as the quasi-Monte Carlo method and is used in some pricing tasks. Thus, we cannot say that quantum algorithms for the integration are better than the best classical algorithm on the asymptotic behavior of the estimation error with respect to the calculation time. However, the complexity dependence on the function dimensions is known to be worse in the quasi-Monte Carlo method than the ordinal Monte Carlo method. For this reason, we expect that quantum algorithm is beneficial to the integration of high dimensional functions such as Eq. (2).

After the first version of this paper appeared as a preprint, Refs. [10, 25] have pointed out that the Grover-Rudolph method [14] for preparing distributions as amplitudes can eliminate the quadratic speedup. The AE-type method for the LV model with the Grover-Rudolph state preparation might be faced with a similar obstacle.[4] On the other hand, the PRN-type method is free from such a problem because it does not encode the probability distribution.

4 Quantum circuits for the LV model

This section presents quantum circuits for the state preparation in two methods: the PRN-type and the AE-type methods.

4.1 Elementary gate

Before presenting our proposals, we list up elementary gates used in following discussion:

- Adder: $|x\rangle|y\rangle \rightarrow |x+y\rangle|y\rangle$
- Controlled Adder: $|c\rangle|x\rangle|y\rangle \rightarrow \begin{cases} |c\rangle|x+y\rangle|y\rangle; & \text{for } c=1, \\ |c\rangle|x\rangle|y\rangle; & \text{for } c=0 \end{cases}$
- Multiplier: $|x\rangle|y\rangle|z\rangle \rightarrow |x\rangle|y\rangle|z+xy\rangle$
- Divider: $|x\rangle|y\rangle|0\rangle \rightarrow |x\rangle|y\rangle|x/y\rangle$

Implementation of those elementary arithmetic are studied in many works [26–46]. With these gates, we can construct the other arithmetic we use. For example, subtraction $|x\rangle|y\rangle \rightarrow |x-y\rangle|y\rangle$ can be done as addition by the 2's-complement of y. The 2's-complement of n-bit number y is defined as $2^n - y$, which is equivalent to $-y$ modulo 2^n. Moreover, comparison $|x\rangle|y\rangle|z\rangle \rightarrow |x\rangle|y\rangle|z \oplus (x > y)\rangle$ can be done as subtraction in 2's-complement method, since the most significant bit represents whether the result of subtraction is positive or negative. Thus, a comparator is constructed as two adders including uncomputation.

We also note that the above multiplier uses two registers as operands and outputs the product into another register. However, we need the self-update type of multiplier, which

---

[4] The state preparation method for the standard normal distribution introduced in Sect. 4.3.2 does not suffer from this problem since it does not use Monte Carlo integration to calculate the cumulative distribution over each interval in discrete approximation, whereas Ref. [25] assumed the use of it. Although Ref. [10] said that our method uses Monte Carlo integration in some parts, it actually does not use any. For the detail, see Sect. 4.3.2.

updates either of input registers with the product. Such a operation is realized by the following trick:

$$|x\rangle|y\rangle|0\rangle \to |x\rangle|y\rangle|xy\rangle \to |xy\rangle|y\rangle|x\rangle \to |xy\rangle|y\rangle|0\rangle. \tag{16}$$

Here, the first step is original multiplication. The second step is swap between the first and third registers. The third step is the inverse operation of division.

4.2 PRN-type method

4.2.1 Calculation flow

We present the calculation flow of the PRN-type state preparation for pricing in the LV model. Our purpose is estimating Eq. (10) by the PRN-type Monte Carlo method presented in Sect. 3.2. We show the detailed calculation flow to realize operation (15) in the case where the desired value is given by Eq. (10).

Before presenting the calculation flow, we explain our setup. We generate $N_{\mathrm{samp}} = 2^{n_{\mathrm{samp}}}$ sample paths of length n_{t} by using the PSNRNs. In this algorithm, we prepare the following registers:

- R_{samp} is a register for an index of the sample path and consists of n_{samp} qubits.
- R_W is a register for a PSNRN used to calculate the asset price evolution.
- R_S is a register for the value of the asset price.
- R_{payoff} is a register for the payoffs.

We note that R_W corresponds to the PRN register, and R_{payoff} corresponds to the integrand register in Sect. 3.2. On the other hand, R_S is a tailored ancillary register for calculating the specific integrand and has no counterpart. Although some ancillary registers are needed in addition to the above registers, we abbreviate them in the main calculation flow.

We assume that the following gates are available to generate a sequence of PSNRNs.

- J_W acts on $R_{\mathrm{samp}} \otimes R_W$ and sets the initial value of the PSNRN subsequence: $J_W|i\rangle|0\rangle = |i\rangle|x_{in_{\mathrm{t}}+1}\rangle$, where n_{t} is the number of time steps.
- P_W advances a PSNRN sequence by one step: $P_W|x_j\rangle = |x_{j+1}\rangle$, where x_j is the j-th element of the PSNRN sequence.

Applying these gates to a superposition of N_{samp} states, we obtain N_{samp} PSNRN subsequences:

$$\begin{aligned}\frac{1}{\sqrt{N_{\mathrm{samp}}}}\sum_{i=0}^{N_{\mathrm{samp}}-1}|i\rangle|0\rangle &\xrightarrow{J_W} \frac{1}{\sqrt{N_{\mathrm{samp}}}}\sum_{i=0}^{N_{\mathrm{samp}}-1}|i\rangle|x_{in_{\mathrm{t}}+1}\rangle\\ &\xrightarrow{P_W} \frac{1}{\sqrt{N_{\mathrm{samp}}}}\sum_{i=0}^{N_{\mathrm{samp}}-1}|i\rangle|x_{in_{\mathrm{t}}+2}\rangle\\ &\xrightarrow{P_W}\cdots\xrightarrow{P_W} \frac{1}{\sqrt{N_{\mathrm{samp}}}}\sum_{i=0}^{N_{\mathrm{samp}}-1}|i\rangle|x_{in_{\mathrm{t}}+n_{\mathrm{t}}}\rangle\end{aligned} \tag{17}$$

We also use gate U_j acting on $R_W \otimes R_S \otimes R_{\mathrm{payoff}}$, which calculates the j-th time step of asset price evolution and the payoff as follows:

$$U_j|x_j\rangle|S_{t_{j-1}}\rangle|V_{j-1}\rangle = |x_j\rangle|S_{t_j}\rangle\left|V_{j-1} + f_j^{\mathrm{pay}}(S_{t_j})\right\rangle. \tag{18}$$

In other words, U_j performs the time evolution (8) by using the value on R_W as w_j. After that, U_j calculates the payoff at time t_j and adds its value into R_{payoff}. The concrete implementation of these gates is presented in the next subsection.

The calculation flow of the PRN-type method is as follows:

1. Initialize R_S to $|S_{t_0}\rangle$ and the other registers to $|0\rangle$.
2. Generate $\frac{1}{\sqrt{N_{\mathrm{samp}}}}\sum_{i=0}^{N_{\mathrm{samp}}-1}|i\rangle$ on R_{samp}. This is done by applying a Hadamard gate to each qubit of R_{samp}.
3. Apply J_W to $R_{\mathrm{samp}} \otimes R_W$. This step sets the initial value of the PSNRN subsequence.
4. Apply U_j to $R_W \otimes R_S \otimes R_{\mathrm{payoff}}$ to simulate asset price evolution.
5. Apply P_W to R_W, which updates R_W from $x_{in_{\mathrm{t}}+1}$ to $x_{in_{\mathrm{t}}+2}$.
6. Iterate operations 4-5 n_{t}-times.

The flow of the corresponding state transformations is as follows:

$$\begin{aligned}
&|0\rangle|0\rangle|S_{t_0}\rangle|0\rangle \\
&\xrightarrow{2} \frac{1}{\sqrt{N_{\mathrm{samp}}}}\sum_{i=0}^{N_{\mathrm{samp}}-1}|i\rangle|0\rangle|S_{t_0}\rangle|0\rangle \\
&\xrightarrow{3} \frac{1}{\sqrt{N_{\mathrm{samp}}}}\sum_{i=0}^{N_{\mathrm{samp}}-1}|i\rangle|x_{in_{\mathrm{t}}+1}\rangle|S_{t_0}\rangle|0\rangle \\
&\xrightarrow{4} \frac{1}{\sqrt{N_{\mathrm{samp}}}}\sum_{i=0}^{N_{\mathrm{samp}}-1}|i\rangle|x_{in_{\mathrm{t}}+1}\rangle|S^{(i)}_{t_1}\rangle|f^{\mathrm{pay}}_1(S^{(i)}_{t_1})\rangle \\
&\xrightarrow{5} \frac{1}{\sqrt{N_{\mathrm{samp}}}}\sum_{i=0}^{N_{\mathrm{samp}}-1}|i\rangle|x_{in_{\mathrm{t}}+2}\rangle|S^{(i)}_{t_1}\rangle|f^{\mathrm{pay}}_1(S^{(i)}_{t_1})\rangle \\
&\xrightarrow{6} \cdots \\
&\xrightarrow{6} \frac{1}{\sqrt{N_{\mathrm{samp}}}}\sum_{i=0}^{N_{\mathrm{samp}}-1}|i\rangle|x_{in_{\mathrm{t}}+n_{\mathrm{t}}}\rangle|S^{(i)}_{t_{n_{\mathrm{t}}}}\rangle\left|\sum_{j=1}^{n_{\mathrm{t}}}f^{\mathrm{pay}}_j(S^{(i)}_{t_j})\right\rangle,
\end{aligned} \tag{19}$$

where the first, second, third and fourth kets correspond to R_{samp}, R_W, R_S and R_{payoff}, respectively. The quantum circuit realizing the flow (19) is schematically shown in Fig. 1.

The implementation of U_j is also shown in Fig. 2, where the subroutine gates $V^{(j)}_1, \ldots, V^{(j)}_{n_S}$ are used to update the asset price according to Eqs. (7) and (8), and the gate payoff$_j$ cal-

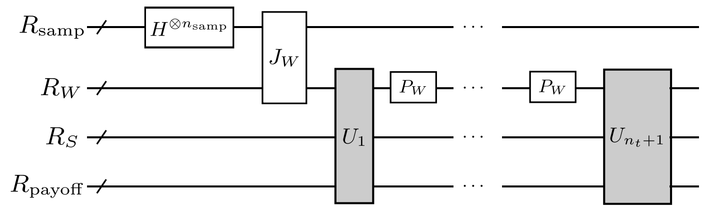


Figure 1 The quantum circuit for asset price evolution of the LV model in the PRN-type method. Ancillary qubits are abbreviated for simple display. The subroutine U_j simulates the time evolution at the j-th time step, and the details of U_j is described in Fig. 2. The implementation of J_W and P_W is shown in Fig. 5

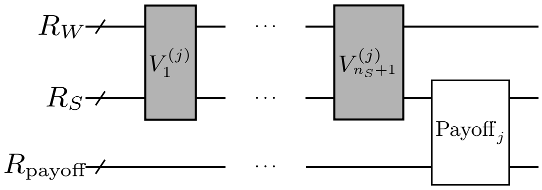


Figure 2 The implementation of the U_j, which performs the j-th step of asset price evolution, in the PRN-type method. $V_k^{(j)}$ is a subroutine for updating the asset price by Eq. (8), and **payoff**$_j$ is a subroutine for payoff calculation. The implementation of $V_k^{(j)}$ is described in Fig. 3

culates $f_j^{\mathrm{pay}}(S_{t_j}^{(i)})$ and adds its value into R_{payoff}. The subroutine $V_k^{(j)}$ realizes the following three operations:

- Checks whether the asset price R_S is in the k-th interval $[s_{j,k-1}, s_{j,k})$.
- If that is the case, updates the asset price by Eq. (8) with $\sigma(t_j, S_{t_j}^{(i)}) = a_{j,k}S_{t_j}^{(i)} + b_{j,k}$.
- Clears all the intermediate output.

This procedure requires three ancillary registers, R_{count}, $R_{S'}$ and R_g. R_{count} stores an indicator of whether the j-th step of evolution has already been done. If the j-th update has already been done, the asset price is not updated, which is necessary to avoid double updating in a single step. R_g stores the check result.

We note that there is a restriction on implementing the LV model in the PRN-type method. Through operations $V_1^{(j)}, \dots, V_{n_S+1}^{(j)}$, the state is transformed from $|j\rangle|S_{t_j}^{(i)}\rangle$ to $|j+1\rangle|S_{t_{j+1}}^{(i)}\rangle$, where the first and second kets represents states of R_{count} and R_S, respectively, and unchanged registers are abbreviated. This map must be one-to-one correspondence from the unitarity, which restricts parameters. As shown in Appendix, the unitarity is certified if we set parameters $a_{i,j}$ and $b_{i,j}$ so that $\sigma(t,S)$ is continuous with respect to S and set Δt_j sufficiently small.

4.2.2 *Implementation of subroutines*

We now consider how to implement subroutines used in the PRN-type method by arithmetic operations in Sect. 4.1.

Implementation of $V_k^{(j)}$ At the start of $V_k^{(j)}$, R_{count} takes $|j\rangle$ or $|j+1\rangle$, and the other registers take $|0\rangle$. Then, the detailed calculation flow of $V_k^{(j)}$ is as follows:

1. Check whether R_{count} equals j and R_S is in $[s_{j,k-1}, s_{j,k})$. If the check is passed, flip R_g.
2. If R_g is 1, update R_S as

$$S_{t_j} \to S_{t_{j+1}} = S_{t_j} + (a_{j,k}S_{t_j} + b_{j,k})\sqrt{\Delta t_j}x_{in_t+j}, \tag{20}$$

   where x_{in_t+j} is the value on R_W, and add 1 to R_{count}.
3. Calculate

$$\frac{S - b_{j,k}\sqrt{\Delta t_j}x_{in_t+j}}{1 + a_{j,k}\sqrt{\Delta t_j}x_{in_t+j}} \tag{21}$$

   into $R_{S'}$, where S is the value on R_S.
4. If R_{count} is $j+1$ and $R_{S'}$ is in $[s_{j,k-1}, s_{j,k})$, flip R_g. This uncomputes R_g.

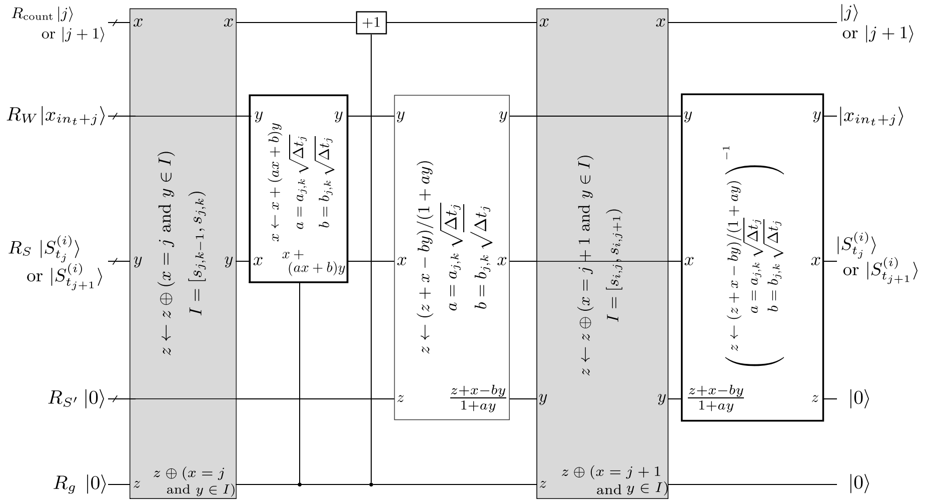

$R_{\mathrm{count}}\,|j\rangle$ or $|j+1\rangle$
$R_W\,|x_{in_t+j}\rangle$
$R_S\ |S^{(i)}_{t_j}\rangle$ or $|S^{(i)}_{t_{j+1}}\rangle$
$R_{S'}\ |0\rangle$
$R_g\ |0\rangle$
$z \leftarrow z \oplus (x = j \text{ and } y \in I)$
$I = [s_{j,k-1}, s_{j,k})$
$z \oplus (x = j$ and $y \in I)$
$+1$
$x \leftarrow x + (ax+b)y$
$a = a_{j,k}\sqrt{\Delta t_j}$
$b = b_{j,k}\sqrt{\Delta t_j}$
$x + (ax+b)y$
$z \leftarrow (z + x - by)/(1+ay)$
$a = a_{j,k}\sqrt{\Delta t_j}$
$b = b_{j,k}\sqrt{\Delta t_j}$
$\frac{z+x-by}{1+ay}$
$z \leftarrow z \oplus (x = j+1 \text{ and } y \in I)$
$I = [s_{i,j}, s_{i,j+1})$
$z \oplus (x = j+1$ and $y \in I)$
$\left(z \leftarrow (z + x - by)/(1+ay)\ \ a = a_{j,k}\sqrt{\Delta t_j}\ \ b = b_{j,k}\sqrt{\Delta t_j}\right)^{-1}$
$\frac{z+x-by}{1+ay}$
$|j\rangle$ or $|j+1\rangle$
$|x_{in_t+j}\rangle$
$|S^{(i)}_{t_j}\rangle$ or $|S^{(i)}_{t_{j+1}}\rangle$
$|0\rangle$
$|0\rangle$
x
y
z

Figure 3 The implementation of $V^{(j)}_k$, which updates R_S if the asset price is in the k-th grid of the LV function. Here and hereafter, the wire going over a gate means that the corresponding register is not used in the operation of the gate. A formula at the center of a gate represents the operation the gate performs, and superscript '-1' means the inverse operation. A formula beside a wire and in a gate represents the input or the output value of the gate

5. Do the inverse operation of 3.

If and only if the j-th step has not been done and the asset price is in $[s_{j,k-1}, s_{j,k})$, the asset price is upadated with the LV function $a_{j,k}S + b_{j,k}$. To realize this conditional update, the check result is outputted to R_g, and the gate doing update (20) is controlled by R_g. The increment of R_{count} is also controlled by R_g so that R_{count} indicates the completion of the j-th update. Steps 3-5 are necessary to clear R_g. From the result of Step 3, we can determine whether the update has been done in Step 2. In step 4, R_g is flipped if and only if it is $|1\rangle$, so it goes back to the initial state $|0\rangle$. In summary, through the sequential operation of $V^{(j)}_1, \ldots, V^{(j)}_{n_S+1}$, R_S is updated only once at the appropriate $V^{(j)}_k$, R_{count} is updated from $|j\rangle$ to $|j+1\rangle$, and all intermediate outputs on ancillary registers are cleared. See also Fig. 3.

Most of sub-parts of $V^{(j)}_k$ can be constructed from arithmetic operations, addition, subtraction, multiplication, division, and comparison. For example, Let us consider the gate $z \leftarrow z \oplus (x = j \text{ and } y \in I)$, which is divided to the following two parts. The first part is checking whether the value on R_{count} equals j. This check can be done by the multiple control Toffoli gate, which is studied in Refs. [18, 47, 48]. The second part is checking whether the asset price is in a given interval, which can be constructed from two comparisons. Combining these, the gate $z \leftarrow z \oplus (x = j \text{ and } y \in I)$ is constructed as shown in Fig. 4. Note that the bitwise flips $X^{1-j_0} \otimes \cdots \otimes X^{1-j_{n_x-1}}$ are operated before the multi control Toffoli. Here, j_a is the a-th digit of the binary representation of j, so the a-th qubit is flipped if and only if $j_a = 0$. This convert $|x\rangle$ to $|1\rangle \ldots |1\rangle$ if and only if $x = j$.

The operation $x \leftarrow x + (ax+b)y$ in Fig. 3 can be realized as follows:

$$
\begin{aligned}
|x\rangle|y\rangle|0\rangle &\rightarrow |x\rangle|y\rangle|1\rangle \\
&\rightarrow |x\rangle|y\rangle|1+ay\rangle \\
&\rightarrow |(1+ay)x\rangle|y\rangle|1+ay\rangle
\end{aligned}
$$

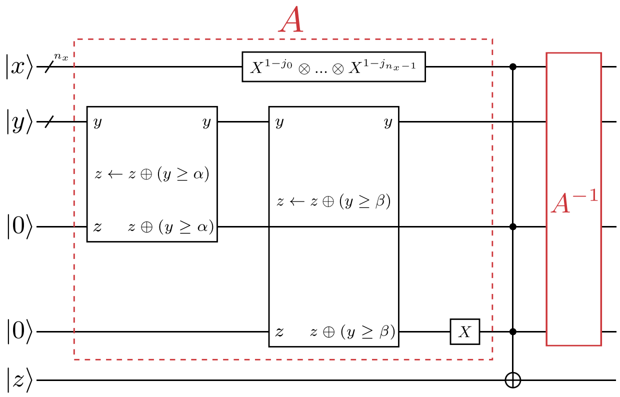

A
$|x\rangle$
n_x
$X^{1-j_0} \otimes \ldots \otimes X^{1-j_{n_x-1}}$
$|y\rangle$
y
y
$z \leftarrow z \oplus (y \geq \alpha)$
$|0\rangle$
z
$z \oplus (y \geq \alpha)$
y
y
$z \leftarrow z \oplus (y \geq \beta)$
$|0\rangle$
z
$z \oplus (y \geq \beta)$
X
A^{-1}
$|z\rangle$

Figure 4 The gate which outputs whether $x = j$ and $y \in [\alpha, \beta)$ or not. The control by a register means the multiple control by qubits therein

$$
\begin{aligned}
&\to |(1+ay)x+by\rangle|y\rangle|1+ay\rangle \\
&\to |(1+ay)x+by\rangle|y\rangle|0\rangle,
\end{aligned}
\tag{22}
$$

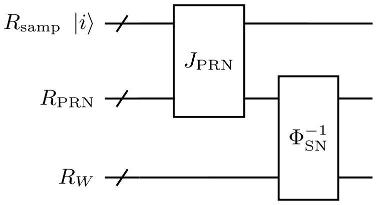
where the third ket corresponds to an ancillary register. The first step is just setting a constant on the ancillary register. The second step is the multiplication by a. The third step is self-update multiplication. The fourth step is multiplication by b, and the final step is uncomputation of the first and second steps. Note that this is done under control by R_g. In order for this to be controlled, it is sufficient to control only the second, fourth and final arrows because the third arrow becomes multiplication by 1 without the second. Also note that multiplication by an n-bit constant can be done by n-adders, that is, n shift-and-add's: $ax = \sum_{i=0}^{n-1} a_i 2^i x$, where a_i is the i-th bit of a. This method saves the number of qubits compared with the case of using a multiplier, where we need to hold a on an ancillary register.

The operation $x \leftarrow (x-by)/(1+ay)$ in Fig. 3 is done as follows:

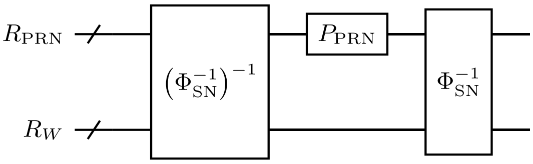
$$
\begin{aligned}
|x\rangle|y\rangle|0\rangle|0\rangle &\to |x\rangle|y\rangle|1\rangle|0\rangle \\
&\to |x\rangle|y\rangle|1+ay\rangle|0\rangle \\
&\to |x-by\rangle|y\rangle|1+ay\rangle|0\rangle \\
&\to |x-by\rangle|y\rangle|1+ay\rangle|(x-by)/(1+ay)\rangle,
\end{aligned}
\tag{23}
$$

where the first, second, third and fourth states correspond to R_S, R_W, an ancillary register and $R_{S'}$, respectively. The first and second steps are the same as Eq. (22), the third step is the multiplication by $-b$, and the final step is division. Here, we do not have to uncompute R_S and the ancillary register because the whole of this operation is uncomputed soon after in $V_{j,k}$.

Implementation of J_W and P_W In Ref. [6], implementation of PRN on quantum circuits is based on permuted congruential generator (PCG) [49], which is a PRN generation algorithm with small memory requirements. We use the following two gates to run PCG: (i) J_{PRN} lets the PRN sequence jump to the $in_{\mathrm{t}} + 1$. (ii) P_{PRN} progresses the PRN sequence by



(a) J_W, the gate to let the PSNRN sequence jump to the $in_{\mathrm{t}}+1$ element.


R_{PRN}
R_W
$\left(\Phi_{\mathrm{SN}}^{-1}\right)^{-1}$
P_{PRN}
Φ_{SN}^{-1}

(b) P_W, the gate to progress the PSNRN sequence by a step.

Figure 5 The circuits used to generate a PSNRN sequence. R_{PRN} is a register for a PRN sequence generated by PCG, and R_W is a register for a PSNRN sequence. J_{PRN} and P_{PRN} are the jump gate and the progress gate of PCG, respectively. $\Phi_{\mathrm{SN}}^{-1}(x)$ is the inverse CDF of standard normal distribution

a step. Since PCG basically generates uniform PRNs, we transform them to PSNRNs by adopting the inverse transform sampling. The implementation of J_W and P_W are schematically shown in Fig. 5.

Although we refer to Ref. [6] for the detail of the implementation of the PRN generator, we here briefly explain it. PCG is a combination of linear congruential generator (LCG) and permutation of bit string. For LCG, update of the PRN sequence is done by

$$x_{n+1} = ax_n + c \bmod N, \tag{24}$$

where a and N are positive integers, c is a nonnegative integer. From the above equation, the n-th element of the sequence is computed from the initial value x_0 by

$$x_n = a^n x_0 + \frac{c(a^n - 1)}{a - 1} \bmod N. \tag{25}$$

We can implement Eqs. (24) and (25) using only controlled adders. According to Ref. [26], the modular adder can be constructed by 5 plain adders. Modular multiplication by a n-bit constant can be done as n modular shift-and-add's. Modular division by a constant $a-1$ can be done as modular multiplication by an integer β such that $\beta(a-1) = 1 \bmod N$. Modular exponentiation $a^x \bmod N$ is computed as a sequence controlled modular multiplication [26]. We do not explain permutation: see Ref. [6] for the detail. We make a comment that it is implemented by a simple circuit; for example, Xorshift is implemented as a sequence of CNOT.

The step by step transformation of the implementation of Eq. (24) is as follows:

$$\begin{aligned}|x_n\rangle|0\rangle|0\rangle &\rightarrow |x_n\rangle|ax_n \bmod N\rangle|0\rangle \\ &\rightarrow |0\rangle|ax_n \bmod N\rangle|0\rangle \\ &\rightarrow |0\rangle|ax_n \bmod N\rangle|c\rangle \\ &\rightarrow |0\rangle|ax_n + c \bmod N\rangle|c\rangle \\ &\rightarrow |0\rangle|ax_n + c \bmod N\rangle|0\rangle \\ &= |0\rangle|x_{n+1}\rangle|0\rangle. \end{aligned} \tag{26}$$

Here, the first register is R_{PRN}, and the other registers are ancillary registers. The first step is modular multiplication. The second step is the inverse modular multiplication by an integer α such that $a\alpha = 1 \bmod N$, which is necessary to avoid the increase of ancillae. The third step is the load of c into an ancillary register, the fourth step is modular addition, and the last step is to unload. Equation (25) progresses as follows:

$$\begin{aligned}&|n\rangle|0\rangle|0\rangle|0\rangle \\ &\quad\rightarrow |n\rangle\left|a^n \bmod N\right\rangle|0\rangle|0\rangle \\ &\quad\rightarrow |n\rangle\left|a^n \bmod N\right\rangle\left|\left(x_0 + \frac{c}{a-1}\right)a^n \bmod N\right\rangle|0\rangle \\ &\quad\rightarrow |n\rangle\left|a^n \bmod N\right\rangle\left|\left(x_0 + \frac{c}{a-1}\right)a^n \bmod N\right\rangle\left|\frac{c}{a-1}\right\rangle \\ &\quad\rightarrow |n\rangle\left|a^n \bmod N\right\rangle\left|\left(x_0 + \frac{c}{a-1}\right)a^n - \frac{c}{a-1} \bmod N\right\rangle\left|\frac{c}{a-1}\right\rangle \\ &\quad\rightarrow |n\rangle|0\rangle\left|\left(x_0 + \frac{c}{a-1}\right)a^n - \frac{c}{a-1} \bmod N\right\rangle|0\rangle \\ &\quad= |n\rangle|0\rangle|x_n\rangle|0\rangle, \end{aligned} \tag{27}$$

where the first and third registers are R_{samp} and R_{PRN}, and the other registers are ancillary registers. The first step is modular exponentiation, the second step is modular multiplication, the third step is loading, the fourth step is modular addition, and the last step is uncomputation of the first and third steps.

Implementation of Φ^{-1}_{SN} We also need the gate to calculate Φ^{-1}_{SN}, the inverse function of the CDF of standard normal distribution. We adopt the method in Ref. [50], where Φ^{-1}_{SN} is approximated by a piecewise polynomial function. Let us set the number n_{ICDF} of intervals to be 109 and polynomials to be cubic, that is, Φ^{-1}_{SN} is approximated as

$$\Phi^{-1}_{\mathrm{SN}}(x) \approx c_{m,3}x^3 + c_{m,2}x^2 + c_{m,1}x + c_{m,0} \tag{28}$$

in $x^{\mathrm{ICDF}}_{m-1} \leq x < x^{\mathrm{ICDF}}_m$, where $\{x^{\mathrm{ICDF}}_m\}^{n_{\mathrm{ICDF}}}_{m=0}$ are the end points of the intervals. This approximation realizes the error smaller than 10^{-6}. Such a piecewise cubic function can be implemented as in Fig. 6. The sequence of comparators and "Load $c_{m,i}$'s" gates loads appropriate values of $c_{m,0},\dots,c_{m,3}$ into the register $R_{c,0},\dots,R_{c,3}$, respectively, as explained later. After the load of coefficients, the cubic function is calculated in the Horner's method, which is based on the following representation

$$\big((c_{m,3}x + c_{m,2})x + c_{m,1}\big)x + c_{m,0}. \tag{29}$$

Horner's method is realized only by adders and multipliers as presented in the latter half of the circuit in Fig. 6.

Let us explain the way to load the appropriate coefficients. Comparing the input value x of R_{PRN} and the grid point x^{ICDF}_m, the m-th comparator flips a qubit on R_g if $x < x^{\mathrm{ICDF}}_m$. The register R_g rules the activation of "Load $c_{m,i}$'s" gate, that is, the "Load $c_{m,i}$'s" gate is activated if R_g is 1 at m-th step. If $x \geq x^{\mathrm{ICDF}}_{n_{\mathrm{ICDF}}}$, only "Load $c_{n_{\mathrm{ICDF}}+1,i}$'s" gate is activated, and $c_{n_{\mathrm{ICDF}}+1,0},\dots,c_{n_{\mathrm{ICDF}}+1,3}$ are loaded to the registers. However, if $x^{\mathrm{ICDF}}_{n_{\mathrm{ICDF}}-1} \leq x < x^{\mathrm{ICDF}}_{n_{\mathrm{ICDF}}}$, "Load $c_{n_{\mathrm{ICDF}},i}$'s" and "Load $c_{n_{\mathrm{ICDF}}+1,i}$'s" gates are performed. Hence, we have to set "Load $c_{n_{\mathrm{ICDF}},i}$'s" to compensate the effect of "Load $c_{n_{\mathrm{ICDF}}+1,i}$'s". More generally, the activated gates are "Load $c_{m,i}$'s" of $m = M, M+2,\dots,n_{\mathrm{ICDF}}, n_{\mathrm{ICDF}}+1$ if $n_{\mathrm{ICDF}} - M$ is even and that of $m = M, M+2,\dots,n_{\mathrm{ICDF}}-1, n_{\mathrm{ICDF}}+1$ if $n_{\mathrm{ICDF}} - M$ is odd. This is because R_g is flipped by all comparators after the M-th step and alternates between 0 and 1. Considering those, we set the X gates in "Load $c_{m,i}$'s" as in Fig. 7, so that $c_{m,0},\dots,c_{m,3}$ for appropriate m are loaded after the sequence of all activated gates.

Implementation of payoff In this paper, we do not consider gates to calculate payoffs in detail because the resource the gates require is the same in both the PRN-type method and the AE-type method. We here make just a short comment. In many cases, a payoff is expressed as

$$f^{\mathrm{pay}}_i = \min\big\{\max\{a_i S_{t_i} + b_i, f_i\}, c_i\big\}, \tag{30}$$

where a_i, b_i, c_i, f_i are real constants. Thst is, a payoff is a linear function of the asset price with the upper bound (*cap*) c_i and the lower bound (*floor*) f_i. For example, a payoff in an European call option (1) corresponds to the case of $a_i = 1$, $b_i = -K$, $c_i = +\infty$, $f_i = 0$. The right-hand side of Eq. (30) can be calculated by a combination of comparators, adders, and multipliers.

4.3 The AE-type method

4.3.1 Calculation flow

The AE-type method is simpler than the PRN-type method, but it requires more registers. In the AE-type method, we use the following registers:

- R_{W_i} is a register for the i-th SNRN ($i = 1,\dots,n_{\mathrm{t}}$).
- R_{S_i} is a register for the asset price at time t_i ($i = 0,\dots,n_{\mathrm{t}}$).
- $R_{\mathrm{payoff},i}$ is a register for the sum of payoffs by t_i ($i = 1,\dots,n_{\mathrm{t}}$).

We again abbreviated ancillary registers. In R_{W_i}, an SNRN is encoded into a superposition state $|\mathrm{SN}\rangle$, which is defined as

$$|\mathrm{SN}\rangle := \sum_{i=0}^{N_{\mathrm{SN}}-1} \sqrt{p_{\mathrm{SN},i}}\,|i\rangle, \tag{31}$$

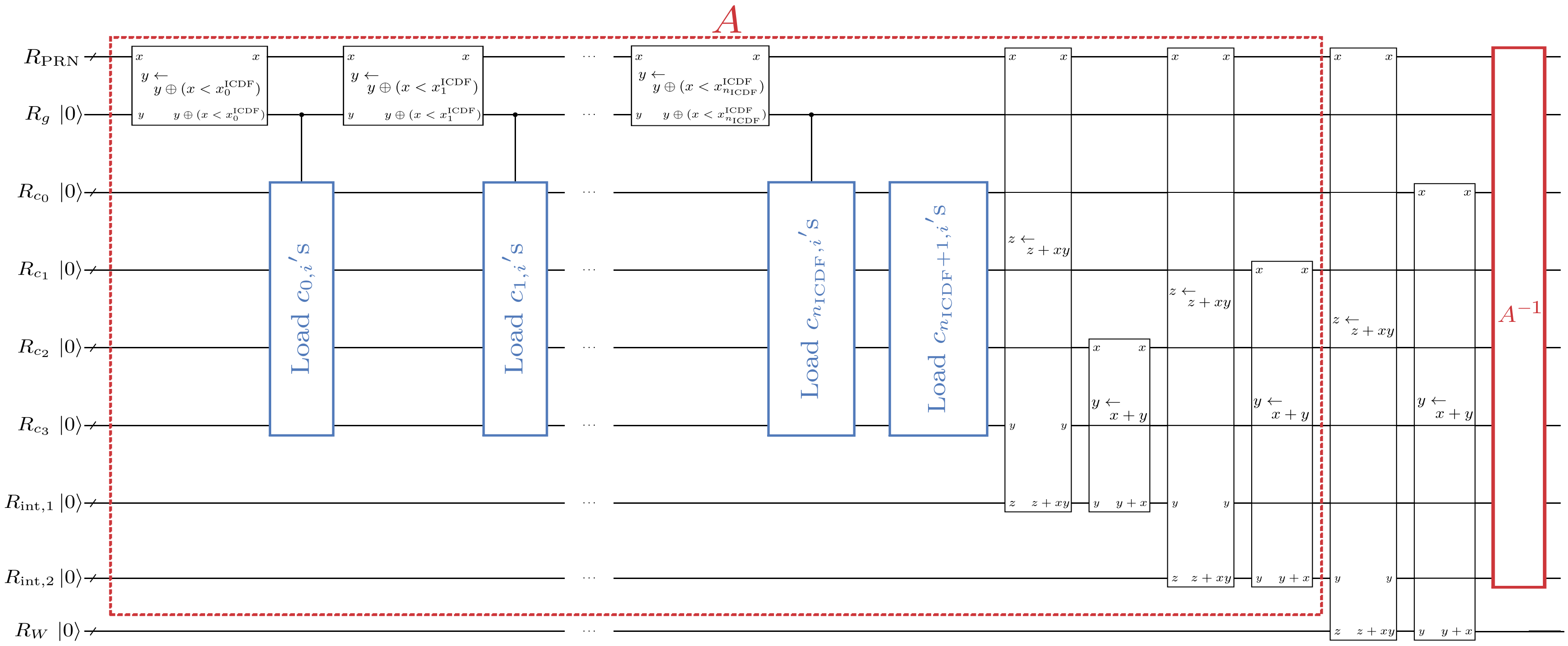
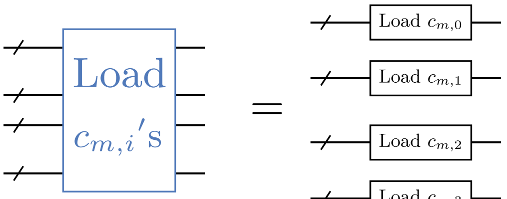
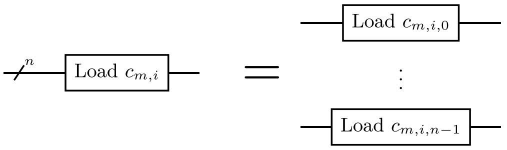

R_{PRN}
R_g $|0\rangle$
R_{c0} $|0\rangle$
R_{c_1} $|0\rangle$
R_{c_2} $|0\rangle$
R_{c_3} $|0\rangle$
$R_{\mathrm{int},1}$ $|0\rangle$
$R_{\mathrm{int},2}$ $|0\rangle$
R_W $|0\rangle$
A
$y \leftarrow y \oplus (x < x_0^{\mathrm{ICDF}})$
$y \leftarrow y \oplus (x < x_1^{\mathrm{ICDF}})$
$y \leftarrow y \oplus (x < x_{n_{\mathrm{ICDF}}}^{\mathrm{ICDF}})$
Load $c_{0,i}$'s
Load $c_{1,i}$'s
Load $c_{n_{\mathrm{ICDF}},i}$'s
Load $c_{n_{\mathrm{ICDF}}+1,i}$'s
$z \leftarrow z + xy$
$y \leftarrow x + y$
A^{-1}

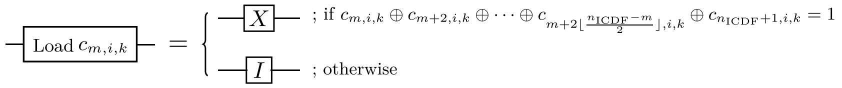
Figure 6 The gate to calculate the inverse CDF of standard normal distribution by piecewise polynomial approximation (28)



(a) The detail of the "Load $c_{m,i}$'s" gate.



(b) The detail of the "Load $c_{m,i}$" gate.


Load $c_{m,i,k}$
=
X
; if $c_{m,i,k} \oplus c_{m+2,i,k} \oplus \cdots \oplus c_{m+2\lfloor \frac{n_{\mathrm{ICDF}}-m}{2} \rfloor,i,k} \oplus c_{n_{\mathrm{ICDF}}+1,i,k} = 1$
I
; otherwise

(c) The detail of the "Load $c_{m,i,k}$" gate.

Figure 7 details of the "Load $c_{m,i}$'s" gate in Fig. 6. Here, $c_{m,i,k}$ denotes the k-th digit of $c_{m,i}$

where $p_{\mathrm{SN},i} := \Phi_{\mathrm{SN}}(x_{\mathrm{SN},i+1}) - \Phi_{\mathrm{SN}}(x_{\mathrm{SN},i})$. Here, $x_{\mathrm{SN},0} < x_{\mathrm{SN},1} < \cdots < x_{\mathrm{SN},N_{\mathrm{SN}}}$ are the equally spaced $N_{\mathrm{SN}} + 1$ points for discretizing the distribution. We also assume $N_{\mathrm{SN}} = 2^{n_{\mathrm{dig}}}$ with the bit size n_{dig} of floating point number for simplicity. We discuss a gate creating such a state in the next subsection.

The calculation flow of the AE-type method is as follows:

1 Initialize R_{S_0} to $|S_{t_0}\rangle$ and the others to $|0\rangle$.
2 Generate $|\mathrm{SN}\rangle$ on each of $R_{W_1},\dots,R_{W_{n_t}}$.
3 Calculate S_{t_1} by the time evolution (8) and output the result to R_{S_1}.
4 Calculate the payoff at time t_1 and add its value to $R_{\mathrm{payoff},i}$.
5 Iterate operations 3-4 n_t-times. Then, we obtain a superposition of states in which the value on R_{payoff,n_t} is the sum of payoffs for each path.

The flow of the corresponding state transformation is as follows. Writing only $R_{W_1},\dots,R_{W_{n_t}}, R_{S_0}, R_{S_1},\dots,R_{S_{n_t}}$ and $R_{\mathrm{payoff},1},\dots,R_{\mathrm{payoff},n_t}$,

$$
\begin{aligned}
&|0\rangle^{\otimes n_t}|S_{t_0}\rangle|0\rangle^{\otimes n_t}|0\rangle^{\otimes n_t} \\
&\quad \xrightarrow{2} |\mathrm{SN}\rangle^{\otimes n_t}|S_{t_0}\rangle|0\rangle^{\otimes n_t}|0\rangle^{\otimes n_t} \\
&\quad \xrightarrow{3} \sum_{i_1=0}^{N_{\mathrm{SN}}-1} \sqrt{p_{\mathrm{SN},i_1}}|i_1\rangle|\mathrm{SN}\rangle^{\otimes n_t-1}|S_{t_0}\rangle\big|S_{t_1}^{(i_1)}\big\rangle|0\rangle^{\otimes n_t-1}|0\rangle^{\otimes n_t} \\
&\quad \xrightarrow{4} \sum_{i_1=0}^{N_{\mathrm{SN}}-1} \sqrt{p_{\mathrm{SN},i_1}}|i_1\rangle|\mathrm{SN}\rangle^{\otimes n_t-1}|S_{t_0}\rangle\big|S_{t_1}^{(i_1)}\big\rangle|0\rangle^{\otimes n_t-1}\big|f_{i_1}^{\mathrm{pay}}\big(S_{t_1}^{(i_1)}\big)\big\rangle|0\rangle^{\otimes n_t-1} \\
&\quad \xrightarrow{5} \dots
\end{aligned}
$$

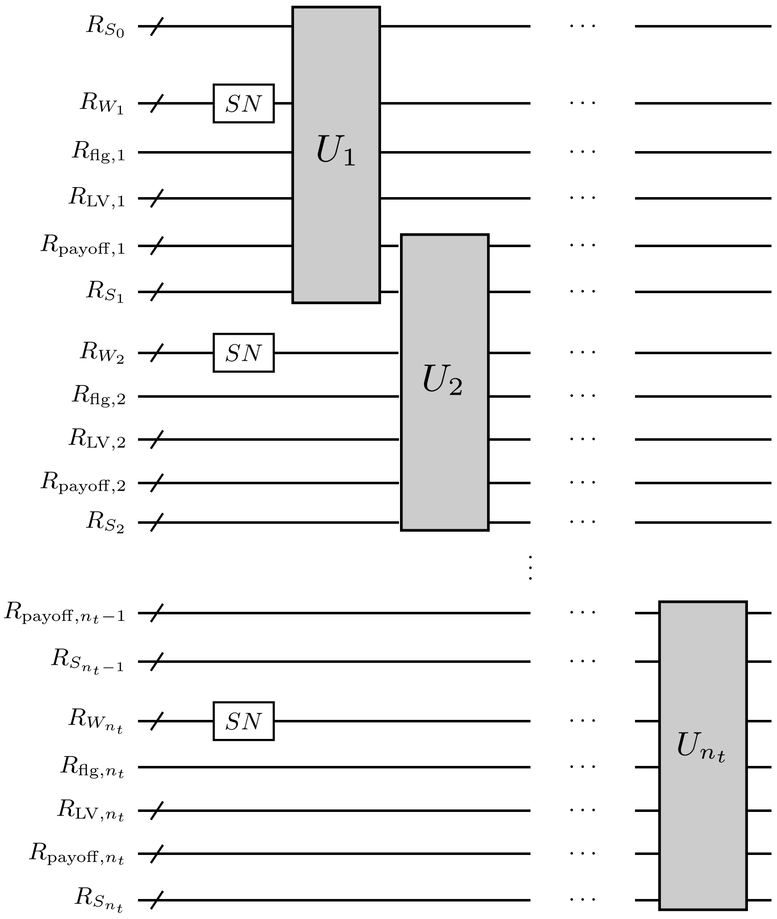


Figure 8 The overview of the circuit for asset price evolution in the LV model in the AE-type method

$$\xrightarrow{5} \sum_{i_1,\dots,i_{n_\mathrm{t}}=0}^{N_\mathrm{SN}-1} \sqrt{p_{\mathrm{SN},i_1}\dots p_{\mathrm{SN},i_{n_\mathrm{t}}}}\,|i_1\rangle\dots|i_{n_\mathrm{t}}\rangle|S_{t_0}\rangle\left|S_{t_1}^{(i_1)}\right\rangle\dots$$

$$\left|S_{t_{n_\mathrm{t}}}^{(i_1\cdots i_{n_\mathrm{t}})}\right\rangle\left|f_{i_1}^{\mathrm{pay}}\left(S_{t_1}^{(i_1)}\right)\right\rangle\dots\left|\sum_{j=1}^{n_\mathrm{t}} f_j^{\mathrm{pay}}\left(S_{t_j}^{(i_1\dots i_j)}\right)\right\rangle, \tag{32}$$

where $S_{t_j}^{(i_1\dots i_j)}$ is the value of the asset price at time t_j evolved by $w_1 = x_{\mathrm{SN},i_1},\dots,w_j = x_{\mathrm{SN},i_j}$.

The quantum circuit of the AE-type state preparation is shown in Fig. 8. First, $|\mathrm{SN}\rangle$ is created on each R_{W_j} by SN gate. After that, the gate U_j performs the j-th step of asset price evolution and payoff calculation. For each evolution step, we additionally use ancillary registers $R_{\mathrm{flg},j}$ and $R_{\mathrm{LV},j}$, which have 1 and $2n_\mathrm{dig}$ qubits, respectively. The implementation of U_j is shown in Fig. 9. In this gate, the sequence of comparators and "Load" gates set $a_{j,k}$, $b_{j,k}$ in Eq. (7) into $R_{\mathrm{LV},j}$ by the trick similar to that in the circuit presented in Fig. 6. Then, operation $x \leftarrow x + (ax + b)y$ updates the asset price according to Eq. (8). Operation $x \leftarrow x + (ax + b)y$ can be done as follows:

$$\begin{aligned}|x\rangle|y\rangle|a\rangle|b\rangle|0\rangle|0\rangle &\to |x\rangle|y\rangle|a\rangle|b\rangle|0\rangle|x\rangle\\ &\to |x\rangle|y\rangle|a\rangle|b\rangle|xy\rangle|x\rangle\end{aligned}$$

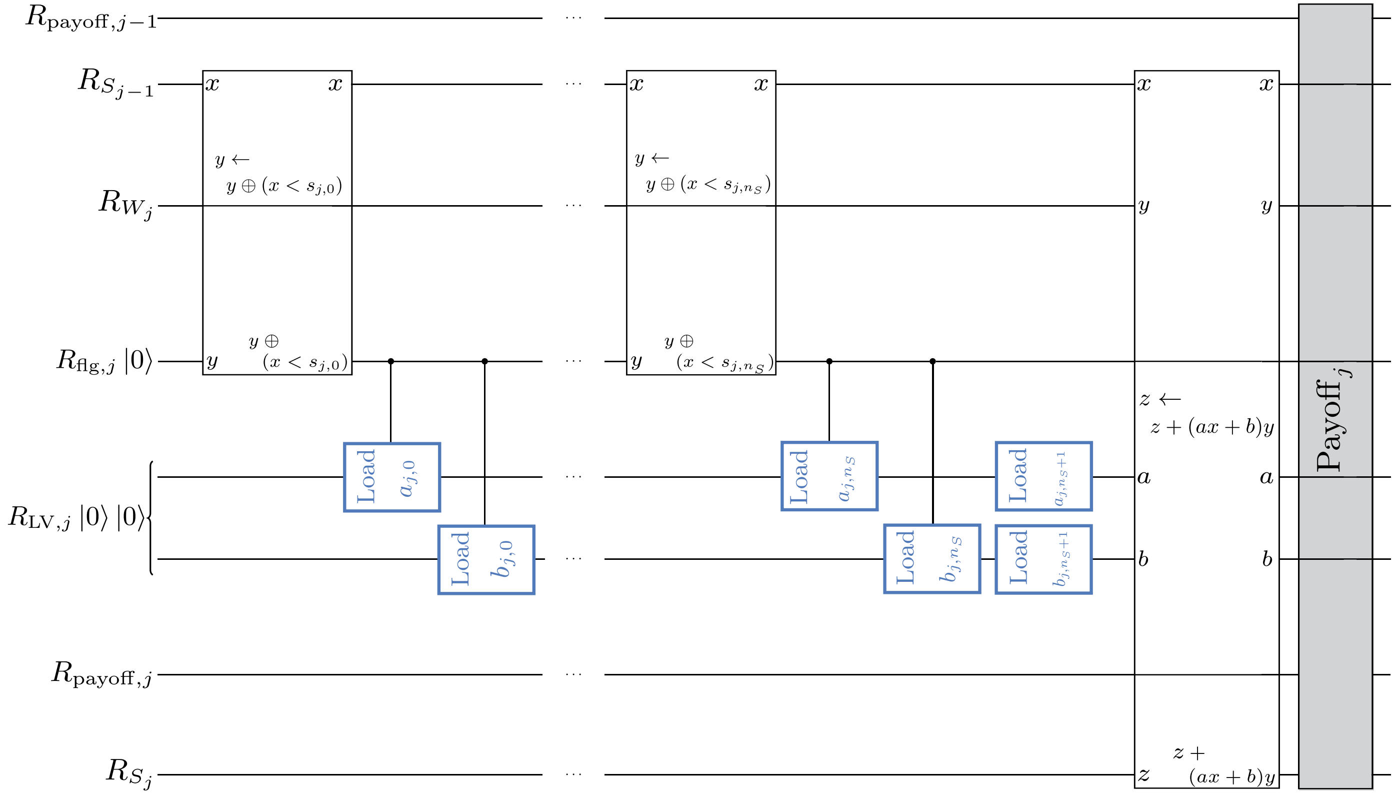


Figure 9 The implementation of U_j, which performs the j-th step of asset price evolution, in the AE-type method

$$\begin{aligned}&\rightarrow |x\rangle|y\rangle|a\rangle|b\rangle|xy\rangle|x+axy\rangle\\ &\rightarrow |x\rangle|y\rangle|a\rangle|b\rangle|xy\rangle|x+axy+by\rangle,\end{aligned}\tag{33}$$

where the first ket is the state of $R_{S_{j-1}}$, the second is the state of R_{W_j}, the third and fourth are the state of $R_{\mathrm{LV},j}$, the fifth is the state of an ancillary register, and the last is the state of R_{S_j}. So, this operation consists of copying a state and three multiplications. At the end of U_j, the payoff is calculated by the "payoff$_j$" gate, which performs the following operation

$$\left|S_{t_j}^{(i_1\dots i_j)}\right\rangle\left|\sum_{k=1}^{j-1} f_k^{\mathrm{pay}}\left(S_{t_k}^{(i_1\dots i_k)}\right)\right\rangle|0\rangle \rightarrow \left|S_{t_j}^{(i_1\dots i_j)}\right\rangle\left|\sum_{k=1}^{j-1} f_k^{\mathrm{pay}}\left(S_{t_k}^{(i_1\dots i_k)}\right)\right\rangle\left|\sum_{k=1}^{j} f_k^{\mathrm{pay}}\left(S_{t_k}^{(i_1\dots i_k)}\right)\right\rangle,\tag{34}$$

where the first, second and third kets correspond to R_{S_j}, $R_{\mathrm{payoff},j-1}$ and $R_{\mathrm{payoff},j}$. This operation is done by copying $R_{\mathrm{payoff},j-1}$ to $R_{\mathrm{payoff},j}$ and adding $f_j^{\mathrm{pay}}(S_{t_j}^{(i_1\dots i_j)})$ into $R_{\mathrm{payoff},j}$.

4.3.2 Implementation of the SN gate

Let us consider the implementation of the SN gate, which creates a superposition state $|\mathrm{SN}\rangle$. Although our implementation is mainly based on Ref. [14], we use an approximate by the Taylor expansion.

We construct $|\mathrm{SN}\rangle$ in an inductive way. An intermediate state at m-step is given by

$$|\mathrm{SN}_m\rangle := \sum_{i=0}^{2^m-1}\sqrt{p_{\mathrm{SN},i}^{(m)}}|i\rangle,\tag{35}$$

where $p_{\mathrm{SN},i}^{(m)} = \int_{x_{\mathrm{SN},i}^{(m)}}^{x_{\mathrm{SN},i+1}^{(m)}}\phi_{\mathrm{SN}}(x)\,dx$, and $\phi_{\mathrm{SN}}(x)$ is the probability density function of the standard normal distribution. Here, $\{x_{\mathrm{SN},i}\}_{i=0}^{2^m}$ is the set of equally-spaced 2^m+1 points dividing the range $[x_{\mathrm{SN},0}, x_{\mathrm{SN},N_{\mathrm{SN}}}]$. We assume the existence of a gate efficiently computing $\theta_i^{(m)} := \arccos\sqrt{f_i^{(m)}}$ with the input i, where $f_i^{(m)}$ is

$$f_i^{(m)} := \frac{\int_{x_{\mathrm{SN},i}^{(m)}}^{(x_{\mathrm{SN},i}^{(m)}+x_{\mathrm{SN},i+1}^{(m)})/2}\phi_{\mathrm{SN}}(x)\,dx}{\int_{x_{\mathrm{SN},i}^{(m)}}^{x_{\mathrm{SN},i+1}^{(m)}}\phi_{\mathrm{SN}}(x)\,dx}.\tag{36}$$

Then, the following state transformation is possible:

$$\begin{aligned}|\mathrm{SN}_m\rangle|0\rangle|0\rangle &= \sum_{i=0}^{2^m-1}\sqrt{p_{\mathrm{SN},i}^{(m)}}|i\rangle|0\rangle|0\rangle\\ &\rightarrow \sum_{i=0}^{2^m-1}\sqrt{p_{\mathrm{SN},i}^{(m)}}|i\rangle|0\rangle\left|\theta_i^{(m)}\right\rangle\\ &\rightarrow \sum_{i=0}^{2^m-1}\sqrt{p_{\mathrm{SN},i}^{(m)}}|i\rangle\left(\cos\theta_i^{(m)}|0\rangle+\sin\theta_i^{(m)}|1\rangle\right)\left|\theta_i^{(m)}\right\rangle\end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=0}^{2^{m+1}-1} \sqrt{p^{(m+1)}_{\mathrm{SN},i}}\,|i\rangle\big|\theta^{(m)}_i\big\rangle \\
&= |\mathrm{SN}_{m+1}\rangle|0\rangle,
\end{aligned}
\tag{37}
$$

where we use the gate computing $\theta^{(m)}_i$ at the first step and perform the controlled rotation at the second step. Repeating this operation until $m = n_{\mathrm{dig}} - 1$, we finally obtain the desired state $|\mathrm{SN}\rangle$.

The remaining part is constructing the gate to compute $f^{(m)}_i$. Here, we propose a way based on simple Taylor expansion. Let us consider function

$$
g(x, \delta) := \frac{\int_x^{x+\delta/2} \phi_{\mathrm{SN}}(x)\,dx}{\int_x^{x+\delta} \phi_{\mathrm{SN}}(x)\,dx}. \tag{38}
$$

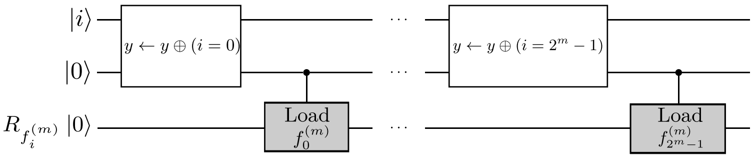
By simple calculation, it is approximated as

$$
g(x, \delta) \approx \frac{1}{2} + \frac{1}{8}\delta x + \frac{1}{16}\delta^2 + \mathcal{O}\big(\delta^3\big). \tag{39}
$$

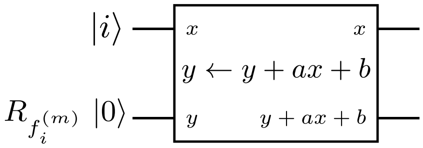
This result means that, for small δ, $g(x, \delta)$ is well-approximated by a linear function of x. We use the above approximation to compute $f^{(m)}_i$, which is represented as

$$
f^{(m)}_i = g\left(x^{(m)}_{\mathrm{SN},i}, \frac{\Delta}{2^m}\right), \qquad \Delta := x_{\mathrm{SN},N_{\mathrm{SN}}} - x_{\mathrm{SN},0}. \tag{40}
$$

If $\Delta/2^m$ is sufficiently small, $f^{(m)}_i$ can be approximately written as a linear function of i, which is derived from the approximation of g and $x^{(m)}_{\mathrm{SN},i} = x_{\mathrm{SN},0} + \frac{\Delta}{2^m} i$. We then reach the circuit in Fig. 10 for calculation of $f^{(m)}_i$. For $m \leq 6$, the above approximation yields a large error, and thus we use another method. Here, we apply the most straightforward way, loading pre-computed values. The quantum circuit of this method is shown in Fig. 10(a), and it uses a similar technique to the circuit in Fig. 6. In this method, each comparator checks whether the input value i equals its inherent value, and the check result is used for activation of the Load gate. If the input value is I, "Load $f^{(m)}_i$" gates are activated for all $i \geq I$. Therefore, each "Load" gate is set to compensate the effect of the following load gates. For

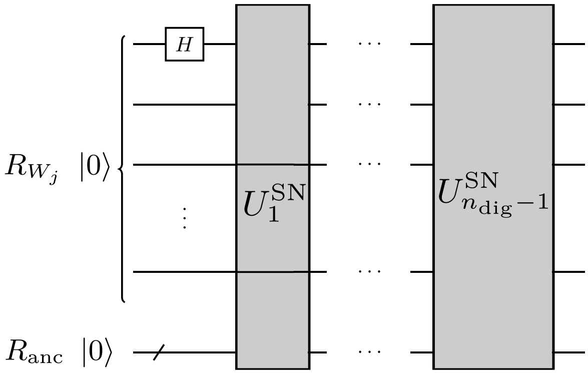

$|i\rangle$
$|0\rangle$
$R_{f^{(m)}_i}\ |0\rangle$
$y \leftarrow y \oplus (i = 0)$
Load $f^{(m)}_0$
$\cdots$
$y \leftarrow y \oplus (i = 2^m - 1)$
Load $f^{(m)}_{2^m-1}$

(a) For $m \leq 6$.

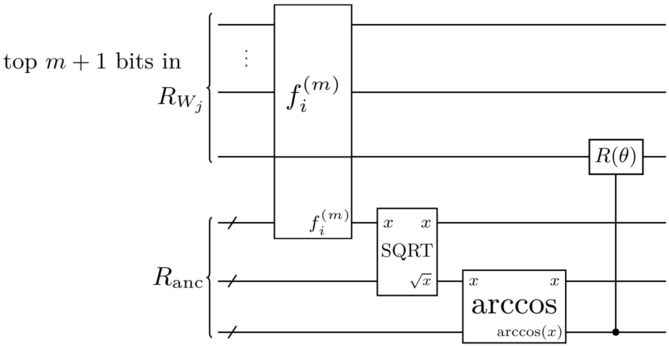


(b) For $m \geq 7$.

Figure 10 Circuit to compute $f^{(m)}_i$



(a) The overview of SN gate.



(b) U_m^{SN}, the m-th step in the SN gate.

Figure 11 Implementation of the SN gate

$m \geq 7$, $f_i^{(m)}$ is well approximated by a linear transformation. This transformation can be implemented as bitwise flips followed by a constant multiplier. We note that, depending on the required accuracy, we should adjust the threshold value of m switching calculation method of $f_i^{(m)}$ and also increase the degree of the Taylor expansion.

Then, SN gate is constructed as shown in Fig. 11. First, we operate a Hadamard gate to the most significant bit in R_{W_j} to assign probability 1/2 to positive and negative halves of $[x_{\mathrm{SN},0}, x_{\mathrm{SN},N_{\mathrm{SN}}}]$. We next operate a sequence of gates $U_1^{\mathrm{SN}}, \dots, U_{n_{\mathrm{dig}}-1}^{\mathrm{SN}}$. U_m^{SN} corresponds to the m-th step of the above recursive calculation and is constructed as a combination of $f_i^{(m)}$ gate, gates for square root and arc cosine, and controlled rotation gate $R(\theta)$.

Finally, we comment on the implementation of arccos and square root. Reference [51] discusses the implementation of the inverse trigonometric function by the piecewise polynomial approximation. Although they consider not arccos but arcsin, we can easily apply their result by $\arccos(x) = \frac{\pi}{2} - \arcsin(x)$. We adopt a setting with the polynomial degree 3 and 2 intervals, which leads to accuracy 10^{-5} [51]. The circuit to calculate square root is given in Ref. [52].

5 Estimation of required resources

We roughly estimate the machine resources for the fault-tolerant implementation in the PRN-type method and the AE-type method. We consider the two metrics, the number of logical qubits and T-count. Our resource estimation focuses only on the leading contribution from the state preparation step, and we must take the implementation of the QAE

Table 1 Resources for the elementary gates. We here assume that operands are n-bit. We omit subleading terms with respect to n

Gate	Logical qubits				T-count	Reference
	Total	Operand	Output	Ancilla		
Adder	$2n$	$2n$	0 (self-update)	0	$14n$	[37, 38, 46]
Ctrl Adder	$2n$	$2n$	0 (self-update)	0	$21n$	[42, 46]
Modular Adder*	$2n$	$2n$	0 (self-update)	0	$70n$	[26, 37, 38, 46]
Multiplier	$3n$	$2n$	n	0	$21n^2$	[42]
Divider	$5n$	$2n$	n	$2n$	$35n^2$	[46]
Multi Ctrl Toffoli	$2n$	n	1	n	$8n$	[18, 48]
Square Root†	$4n$	n	n	$2n$	$14n^2$	[52]
arccos	105				3.4×10^4	[51]
controlled rotation (with accuracy of 2^{-n})	2				$3n$	[16, 47, 53]

*Since the modular adder is constructed by 5 plain adders [26], its T-count is 5 times the values of the adder.

†The circuit given in [52] takes an n-bit input and returns an $n/2$-bit output of square root and an $n/2$-bit remainder. To keep n-bit accuracy, we add n qubits with 0's to the input of the circuit and calculate the n-bit square root with $2n$-bit input. The added n bits are treated as the input, and the n bits remainder is treated as ancillae.

into consideration for evaluating the total resource of the derivative pricing. We also neglect the resource of calculating payoffs because it can be implemented by a combination of a few arithmetic circuits, as discussed in Ref. 4.2.2.

5.1 Elementary gates

We first summarize the resources of elementary gates necessary to construct the LV circuit. We here consider fixed-point arithmetic. Resources of the elementary gates in the case of n-bit operands are summarized in Table 1. Because we aim to estimate the orders of the metrics, we take only the leading term with respect to n. For example, we approximate $an + b$ as an.

We comment on multiplication and division. For these operations, we use modified versions of circuits proposed in Refs. [42, 46] for the following reason. Original circuits use $2n$-bits, but, in our setting, this causes a problem that the number of qubits doubles at every multiplication. Therefore, we have to truncate lower bits of the product and keep the digit number. This is why the number of qubits for a divider in Table 1 is different from that in Refs. [42, 46]. We explain the details of the modified multiplier and divider in Appendix.

5.2 The number of qubits in registers

We assume that the qubit numbers of the registers is as follows. Some of them have already been mentioned.

- Registers which store numerical numbers, R_W, R_S, R_{payoff}, $R_{\mathrm{LV},j}$ etc., and ancillary registers concerning them have n_{dig} qubits. n_{dig} depends on computational representation of real numbers, which is determined according to the required accuracy and range. We set $n_{\mathrm{dig}} = 16$.
- R_{PRN} has n_{PRN} qubits, and n_{PRN} is so large value that the PRN sequence has good statistical property, e.g. long period. Ancillary registers for calculating a PRN sequence have n_{PRN} qubits too. We set the bit of the PRN generator as $n_{\mathrm{PRN}} = 64$ as in Ref. [49].
- R_{samp} has n_{samp} qubits.
- Other registers, e.g. R_{count}, have small number of qubits, and thus we neglect their contributions to the total number.

Table 2 Logical qubits necessary in each step in the PRN-type circuit. We neglect registers with only several qubits

Part	Register	Logical qubit	Note
Whole	R_{samp}	n_{samp}	
	R_S	n_{dig}	
	R_{payoff}	n_{dig}	
	R_{PRN}	n_{PRN}	
J_{PRN}	ancilla	$2n_{\mathrm{PRN}}$	To hold intermediate outputs; see (25)
Φ^{-1}_{SN}	R_W	n_{dig}	
	ancilla	$6n_{\mathrm{dig}}$	To hold the coefficients of the polynomial and the intermediate outputs; see Fig. 6
$V_k^{(j)}$	R_W	n_{dig}	
	$R_{S'}$	n_{dig}	
	ancilla	$4n_{\mathrm{dig}}$	For $x \leftarrow x + (ax + b)y$ and $z \leftarrow \frac{z+x-by}{1+ay}$; see the comment in the body text.
P_{PRN}	ancilla	$2n_{\mathrm{PRN}}$	To hold intermediate outputs; see (26).

5.3 The PRN-type method

Then, let us consider the required resources in the PRN-type method.

5.3.1 Qubit number

In Table 2, we summarize qubits necessary in each step in the circuit. Registers which hold some values throughout the circuit are as follows: R_{samp}, R_S, R_{payoff} ans R_{PRN}. Except these, the following parts in the circuit can consume qubit number most heavily.

- J_{PRN} and P_{PRN}: $2n_{\mathrm{PRN}}$ qubits
- Φ^{-1}_{SN}: $7n_{\mathrm{dig}}$ qubits

Therefore, the total number of qubits required in the PRN-type method is roughly

$$n_{\mathrm{samp}} + 2n_{\mathrm{dig}} + n_{\mathrm{PRN}} + \max\{2n_{\mathrm{PRN}}, 7n_{\mathrm{dig}}\} \tag{41}$$

Let us comment on some technical points for obtaining Table 2. We first make a supplementary explanation on the ancillary qubit number in $V_k^{(j)}$. There are two parts requiring ancillae in $V_k^{(j)}$. First, $x \leftarrow x + (ax + b)y$ needs the following ancillae: a n_{dig}-bit register to which $1 + ay$ is output, a n_{dig}-bit register to which the result is temporally output in the self-update multiplication and a $2n_{\mathrm{dig}}$-bit register necessary for the inverse division to clear the input x. Second, $z \leftarrow \frac{z+x-by}{1+ay}$ needs the following: a n_{dig}-bit register to which $1 + ay$ is output and a $2n_{\mathrm{dig}}$-bit register necessary for division. In total, $4n_{\mathrm{dig}}$ bits are sufficient.[5]

We also comment on the ancilla number in Φ^{-1}_{SN}. As we can see from Fig. 6, we need four registers to which coefficients are loaded and two registers for intermediate outputs. Therefore, $6n_{\mathrm{dig}}$ ancillae are necessary[6]

5.3.2 T-count

Because we are interested in only the leading contribution, we focus on multiplications, divisions, and repeated additions. We do not consider the T-count of J_W because it is used only once. For the parts in U_j, which is used repeatedly, we specify T-counts as follows:

---

[5] Strictly speaking, comparisons between R_S or $R_{S'}$ and $s_{j,k}$'s require loading $s_{j,k}$'s into some register. This does not require another register, since at least one of ancillary registers used in $x \leftarrow x + (ax + b)y$ and $z \leftarrow \frac{z+x-by}{1+ay}$ is empty at loading.

[6] Although we also need a register to which x_m^{ICDF}'s are loaded at comparisons between them and R_{PRN}, we can use R_W or intermediate output registers, which are empty at comparisons.

1. $V_k^{(j)}$

   One $V_k^{(j)}$ includes the following parts:

   - $x \leftarrow x + (ax + b)y$

     As we can see in (22), this includes one multiplication and one division, which come from one self-update multiplication, and $3n_{\mathrm{dig}}$ controlled additions, which comes from two controlled multiplications by constant and one inverse. In total, the T-count is $119n_{\mathrm{dig}}^2$.
   - $z \leftarrow \frac{z+x-by}{1+ay}$

     As we can see in (23), this includes one division and $2n_{\mathrm{dig}}$ additions, which comes from two multiplications by constant. In total, the T-count is $63n_{\mathrm{dig}}^2$.
   - Uncomputation of $z \leftarrow \frac{z+x-by}{1+ay}$

     Similar to the above.

   Therefore, the total T-count in one $V_k^{(j)}$ is $245n_{\mathrm{dig}}^2$. Since $V_k^{(j)}$ is used $n_S + 1$ times, the total T-count in them is $245n_{\mathrm{dig}}^2 n_S$ (only the leading term).
2. P_{PRN}

   This includes two modular multiplications by constant, which comes from one self-update modular multiplication. These are decomposed into $2n_{\mathrm{PRN}}$ modular additions. So the T-count is roughly $140n_{\mathrm{PRN}}^2$.
3. Φ_{SN}^{-1} and its inverse

   Each of them includes $2(n_{\mathrm{ICDF}} + 1)$ additions ($n_{\mathrm{ICDF}} + 1$ comparisons) and five multiplications. So the T-count for each is roughly $105n_{\mathrm{dig}}^2 + 28n_{\mathrm{dig}}n_{\mathrm{ICDF}}$.

Summing up these and considering U_j is used in n_{t} times, the T-count in the whole circuit is roughly

$$\left(245n_{\mathrm{dig}}^2 n_S + 140n_{\mathrm{PRN}}^2 + 210n_{\mathrm{dig}}^2 + 56n_{\mathrm{dig}}n_{\mathrm{ICDF}}\right)n_{\mathrm{t}}. \tag{42}$$

5.4 The AE-type method

Next, we consider the required resources in the AE-type method.

5.4.1 Qubit number

In the AE-type method, registers shown in Table 3 are added per time step. Note that we do not uncompute ancillae. Summing up all registers, the qubit number necessary for one time step is roughly $3n_{\mathrm{dig}}^2 + 111n_{\mathrm{dig}}$. Therefore, for the entire circuit, it is

$$\left(3n_{\mathrm{dig}}^2 + 111n_{\mathrm{dig}}\right)n_{\mathrm{t}}. \tag{43}$$

Table 3 Logical qubits added in each time step in the AE-type circuit. We neglect registers with only several qubits. We only take the leading contributions

Register		Logical qubit	Note
R_{S_i}		n_{dig}	
R_{W_i}		n_{dig}	
$R_{\mathrm{payoff},i}$		n_{dig}	
$R_{\mathrm{LV},i}$		$2n_{\mathrm{dig}}$	
ancilla used for $x \leftarrow x + (ax + b)y$ in U_{t_i}		n_{dig}	See (33)
ancilla for $U_m^{\mathrm{SN}}, m = 1, \ldots, n_{\mathrm{dig}} - 1$	output $f_i^{(m)}$	n_{dig}^2	n_{dig} for one U_m^{SN}
	ancilla for SQRT	$2n_{\mathrm{dig}}^2$	$2n_{\mathrm{dig}}$ for one U_m^{SN}
	qubits used for arccos (input, output and intermediate output)	$105n_{\mathrm{dig}}$	105 for one U_m^{SN}

Note that the dominant part comes from the iterative calculation in the SN gates, which prepare superpositions of the values of the SNRNs.

5.4.2 T-count

Again, we focus on operations with large T-count. For each part in the circuit, we estimate the T-count as follows:

1 SN gate

The m-th iteration $U_m^{\rm SN}$ in the SN gate includes the following parts:

- square root, arccos, controlled rotation

  T-counts are $14n_{\rm dig}$, 3.4×10^4 and $3n_{\rm dig}$, respectively.

- $f_i^{(m)}$

  For $2 \leq m \leq 6$, we use 2^m m-controlled Toffoli gates to check the value on R_{W_i} and load $f_i^{(m)}$ which corresponds to the value. T-count for this is $2^m(8m-9)$.[7] Summing this for $m = 2, \ldots, 6$ leads to about 4000. Since this is much smaller than T-count for arccos in one iteration, we neglect this. For $m \geq 7$, we do multiplication between a m-bit variable and a $n_{\rm dig}$-bit constant, which is decomposed $n_{\rm dig}$ additions of m-bit. Then, T-count is $14mn_{\rm dig}$.

Summing up these counts and taking only dominant contributions, one SN gate has T-count of $(7n_{\rm dig}^2 + 3.4 \times 10^4)n_{\rm dig}$ roughly.

2 U_j

This includes $2n_S$ additions (n_S comparisons) and three multiplications. So one U_j gates has T-count of $63n_{\rm dig}^2 + 28n_S n_{\rm dig}$ roughly.

In total, we can estimate the T-count of the entire circuit in the AE-type method as

$$\left(7n_{\rm dig}^2 + 63n_{\rm dig} + 28n_S + 3.4 \times 10^4\right)n_{\rm dig}n_{\rm t}. \tag{44}$$

5.5 Comparison between two methods

Table 4 compares resources necessary in two methods. The number of qubits is independent of $n_{\rm t}$ in the PRN-type method but proportional to $n_{\rm t}$ in the AE-type method. On the other hand, T-count is proportional to $n_{\rm t}$ in both methods.

Table 4 Comparison of the number of qubits and T-count in the PRN-type method and the AE-type method

	PRN-type	AE-type
logical qubit	$n_{\rm samp} + 2n_{\rm dig} + n_{\rm PRN} + \max\{2n_{\rm PRN}, 7n_{\rm dig}\}$	$(3n_{\rm dig}^2 + 111n_{\rm dig})n_{\rm t}$
T-count	$(245n_{\rm dig}^2 n_S + 140n_{\rm PRN}^2 + 210n_{\rm dig}^2 + 56n_{\rm dig}n_{\rm ICDF})n_{\rm t}$	$(7n_{\rm dig}^2 + 63n_{\rm dig} + 28n_S + 3.4 \times 10^4)n_{\rm dig}n_{\rm t}$

---

[7] Here, we use $8m-9$, the accurate value of T-count of the m-controlled Toffoli gates [18, 48], since the approximation as $8m$ is too crude for small m.

Table 5 Resources to implement the PRN-type and AE-type methods in the practical situation (45). The following values are obtained from substituting Eq. (45) to Table 4

	PRN-type	AE-type
logical qubit	2.4×10^2	9.2×10^5
T-count	3.7×10^8	2.1×10^8

Let us consider the following setting, which is necessary for practical use in derivative pricing:

$$\begin{aligned} &n_{\mathrm{samp}} = 16, \\ &n_{\mathrm{dig}} = 16, \\ &n_{\mathrm{PRN}} = 64, \\ &n_{\mathrm{ICDF}} = 109, \\ &n_{\mathrm{t}} = 360, \\ &n_S = 5. \end{aligned} \tag{45}$$

Table 5 presents resources in this setting. The total T-count is of the same order of magnitude in both methods but larger for the PRN-type method by a factor of about 2.

We here comment on parts consuming T-count most heavily in each method. In the PRN-type method, there are two parts dominantly contributing to T-count. The first part is the update of the asset price in $V_k^{(j)}$. Additional operations for reducing the number of qubits, such as inverse division in self-update multiplication and drawing back the asset price to clear R_g, increase T-count compared with the AE-type method. The second part is modular multiplications in the update of the PRN sequence. The T-count of operations for the PRN becomes large because the PRN generator requires the large bit number, say $n_{\mathrm{PRN}} = 64$, to keep good statistical properties. On the other hand, in the AE-type method, the dominant contribution to T-count comes from the calculation of arccos in preparing SNRNs. Because an arccos is not only T-count consuming but also used in each iteration in the SN gate, it piles up T-count.

6 Summary

In this paper, we presented the implementation of the time evolution of the asset price in the LV model on quantum computers. Similar to other problems in finance, derivative pricing by Monte Carlo simulation requires a large number of random numbers, which is proportional to the number of time steps for asset price evolution. We considered two methods of implementation: the PRN-type method and the AE-type method. In the former, we sequentially generate PRNs on a register and use them to evolve the asset price. In the latter, SNRNs are created as superpositions on separate registers. For both methods, we presented the concrete quantum circuits in detail (see Fig. 1 and 8). We then gave estimations of the qubit number and T-count required in each method. In the PRN-type method, the qubit number is kept constant against the number of time steps. On the other hand, in the AE-type method, the qubit number is proportional to the number of time steps. The total T-counts for both methods are of the same order of magnitude, but the PRN-type method has the larger T-count by a factor of about 2.

Note that analyses of resources required for implementing the LV model in this paper depend on designs of elementary circuits for arithmetic. For example, in the AE-type method, the dominant contribution to T-count comes from arccos's in preparing SNRNs. If more efficient circuits are proposed, the required resources will change from our estimation.

Finally, we would like to note that this study is not enough for the application of a quantum algorithm for Monte Carlo simulation to pricing in the LV model. Although we assumed that the LV function is given, in practice, we have to calibrate the LV so that the model prices of European options fit the market prices. Besides, we have not considered how to evaluate terms in exotic derivatives, for example, early exercise. In future works, we will consider such things and aim to present how to apply quantum computers in the whole process of exotic derivative pricing.

6.1 Condition on the parameters in the PRN-type method

We show that it is necessary for the PRN-type method working well that $\sigma(t,S)$ is continuous on S and Δt_j is sufficiently small. These conditions lead to one-to-one correspondence between $S^{(i)}_{t_j}$ and $S^{(i)}_{t_{j+1}}$. We define a function f by

$$f(S) = S + \sigma(t_j,S)\sqrt{\Delta t_j}w_j, \tag{46}$$

then $S^{(i)}_{t_{j+1}} = f(S^{(i)}_{t_j})$ holds. Except for the grid points $\{s_{j,0},\dots,s_{j,n_S}\}$, $f(S)$ is differentiable, and its derivative is given by

$$f'(S) = 1 + a_{j,k}\sqrt{\Delta t_j}w_j \tag{47}$$

for $s_{j,k-1} < S < s_{j,k}$ and $k = 0,\dots,n_S + 1$. If we take sufficiently small Δt_j, $f'(S)$ is positive expect the grid points. Besides, if $\sigma(t_j,S)$ is continuous on S, $f(S)$ is continuous too. Combining these facts, we find that $f(S)$ is strictly increasing, that is, one-to-one mapping if the above two conditions hold.

6.2 Truncated multiplier and divider

We here describe the modified version of multiplier and divider. We assume that we consider the fixed-point arithmetic with n_{int} bits in the integer part and n_{frac} bits in the fractional part, $n = n_{\text{int}} + n_{\text{frac}}$ bits in total. We hereafter call such numbers $(n_{\text{int}}, n_{\text{frac}})$-bit numbers.

Let us consider truncated multiplication. In order to keep this digit setting during multiplication, we adopt the following policy.

- We simply truncate the digits lower than the n_{frac}-th fractional digit in the product. This might cause numerical errors around and the n_{frac}-th fractional digit and such a tiny error might accumulate, but we simply neglect this concern.
- We assume the overflow from the n_{int}-bit integer part never occurs.

We write a number x in binary representation as $x_{n_{\text{int}}-1}x_{n_{\text{int}}-2}\dots x_0.x_{-1}\dots x_{-n_{\text{frac}}}$, where x_i is the i-th integer digit of x and x_{-j} is the j-th fractional digit of x. We then approximate the product of x and y as follows:

$$xy = \sum_{i=-n_{\text{frac}}}^{n_{\text{int}}-1} x_i 2^i y \approx \sum_{i=-n_{\text{frac}}}^{n_{\text{int}}-1} x_i 2^i \tilde{y}_i =: f^{\text{mul}}_{n_{\text{frac}},n_{\text{int}},y}(x), \tag{48}$$

where

$$\tilde{y}_i = \begin{cases} y_{n_{\mathrm{int}}-1}\dots y_0.y_{-1}\dots y_{-(n_{\mathrm{frac}}-i)}; & \text{for } i < 0, \\ y; & \text{for } i \geq 0. \end{cases} \tag{49}$$

Under our assumption that the overflow from the n-bit integer part never occurs, we have

$$\begin{aligned} f^{\mathrm{mul}}_{n_{\mathrm{frac}},n_{\mathrm{int}},y}(x) = {} & \sum_{i=0}^{n_{\mathrm{int}}-1} x_i 2^i (y_{n_{\mathrm{int}}-1-i}\dots y_0.y_{-1}\dots y_{-n_{\mathrm{frac}}}) \\ & + \sum_{j=1}^{n_{\mathrm{frac}}} x_{-j} 2^{-j} (y_{n_{\mathrm{int}}-1}\dots y_0.y_{-1}\dots y_{-(n_{\mathrm{frac}}-j)}). \end{aligned} \tag{50}$$

This calculation can be constructed by using n-bit controlled adders n-times as in Ref. [42]. Thus, the qubit number and T-count of the circuit for truncated multiplication are the same as those in Ref. [42].

We define the truncated division of z by y as the inverse of the truncated multiplication: $z/y \approx (f^{\mathrm{mul}}_{n_{\mathrm{frac}},n_{\mathrm{int}},y})^{-1}(z)$. Given two $(n_{\mathrm{int}}, n_{\mathrm{frac}})$-bit numbers y and z, we can find an $(n_{\mathrm{int}}, n_{\mathrm{frac}})$-bit number x satisfying $z = f^{\mathrm{mul}}_{n_{\mathrm{frac}},n_{\mathrm{int}},y}(x)$ by the following iterative procedure:

1 Set $i = n_{\mathrm{int}} - 1$ and $x = 0$.
2 Update z with $z - 2^i\tilde{y}_i$
3 Set $x_i = 0$ if $z < 0$, else set $x_i = 1$.
4 If $x_i = 0$, update z with $z + 2^i\tilde{y}_i$ (z returns to the value before step 2).
5 Decrement i by 1.
6 Repeat steps 2-5 until $i = -n_{\mathrm{frac}} - 1$.
7 Output x.

Note that $2^i\tilde{y}_i > \sum_{j=-n_{\mathrm{frac}}}^{i-1} 2^j\tilde{y}_j$. This ensures that sequential subtractions by $2^i\tilde{y}_i$ and checking whether the difference is positive or negative lead to determining each digit of x. In the above procedure, we need to convert z to $(2n_{\mathrm{int}} - 1, n_{\mathrm{frac}})$-bit number to calculate $z \pm 2^{n_{\mathrm{int}}-1}\tilde{y}_{n_{\mathrm{int}}-1}$. So, we introduce n_{int}-dummy qubits corresponding to the $2n_{\mathrm{int}} - 1$-th to n_{int}-th integer digits of z. Then, steps 2-4 are implemented as the circuit in Fig. 12. We also note that the dividend register is reset from $|z\rangle$ to $|0\rangle$ through the above procedure. If

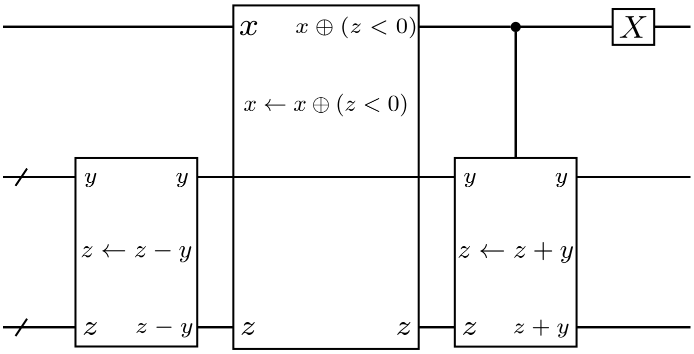


Figure 12 Circuit corrsponding to steps 2-4 in the truncated division. Since, in the 2's complement method, we can check the sign of a number by seeing the most significant bit, we do not have to use an adder as a comparator

we want to reserve $|z\rangle$, we need to copy the dividend state to another ancillary register by CNOT gates, which increases the total number of qubits by n.

Despite the trick to truncate the digits, the structure of the circuit for truncated division is similar to the restoring division circuit in Ref. [46]. Thus, the T-count of our truncated division circuit is the same as that of the circuit in Ref. [46]. On the other hand, since we have introduced dummy qubits and a register to keep the dividend, the qubit number of divider is $5n$.[8]

Funding
The research was funded by Mizuho-DL Financial Technology Co., Ltd.

Abbreviations
LV, Local Volatility; RN, Random Number; PRN, Pseudo-Random Number; BS, Black-Scholes; CDF, Cumulative Distribution Function; SNRN, Standard Normal Random Number; PSNRN, Pseudo Standard Normal Random Number; PCG, Permuted Congruential Generator; LCG, Linear Congruential Generator.

Declarations

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
The original idea to this paper came from K.M. All authors contributed to the preparation of the manuscript. All authors read and approved the final manuscript.

Author details
[1] Mizuho-DL Financial Technology Co., Ltd., Tokyo, Japan. [2] Center for Quantum Information and Quantum Biology, Osaka University, Osaka, Japan.

Publisher's Note

Received: 30 August 2021 Accepted: 28 January 2022 Published online: 12 February 2022

References

1. Orus R et al. Quantum computing for finance: overview and prospects. Rev Phys. 2019;4:100028.
2. Hull JC. Options, futures, and other derivatives. New York: Prentice Hall; 2012.
3. Shreve S. Stochastic calculus for finance I: the binomial asset pricing model. Berlin: Springer; 2004.
4. Shreve S. Stochastic calculus for finance II: continuous-time models. Berlin: Springer; 2004.
5. Montanaro A. Quantum speedup of Monte Carlo methods. Proc R Soc Ser A. 2015;471:2181.
6. Miyamoto K, Shiohara K. Reduction of qubits in quantum algorithm for Monte Carlo simulation by pseudo-random number generator. Phys Rev A. 2020;102:022424.
7. Rebentrost P et al. Quantum computational finance: Monte Carlo pricing of financial derivatives. Phys Rev A. 2018;98:022321.
8. Stamatopoulos N et al. Option pricing using quantum computers. Quantum. 2020;4:291.
9. Ramos-Calderer S et al. Quantum unary approach to option pricing. Phys Rev A. 2021;103:032414.
10. Chakrabarti S et al. A threshold for quantum advantage in derivative pricing. Quantum. 2021;5:463.
11. Black F, Scholes M. The pricing of options and corporate liabilities. J Polit Econ. 1973;81:637.
12. Merton RC. Theory of rational option pricing. Bell J Econ Manag Sci. 1973;4:141.
13. Dupire B. Pricing with a smile. Risk. 1994;7:18–20.
14. Grover L, et al. Creating superpositions that correspond to efficiently integrable probability distributions. quant-ph/0208112.
15. Campbell ET et al. Roads towards fault-tolerant universal quantum computation. Nature. 2017;549:172.
16. Egger DJ, et al. Credit Risk Analysis using Quantum Computers. 1907.03044.
17. Amy M et al. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. IEEE Trans Comput-Aided Des Integr Circuits Syst. 2013;32(6):818–30.
18. Selinger P. Phys Rev A. 2013;87:042302.
19. Maruyama G. On the transition probability functions of the Markov process. Rend Circ Mat Palermo. 1955;4:48.
20. Bassard G et al. Quantum amplitude amplification and estimation. Contemp Math. 2002;305:53.
21. Suzuki Y et al. Amplitude estimation without phase estimation. Quantum Inf Process. 2020;19:75.
22. Nakaji K. Faster Amplitude Estimation. 2003.02417.

---

[8] Actually, added qubits are not $2n$ but $n_{\mathrm{int}} + n$, but we consider that $2n$ qubits are added for simplicity and conservativeness.

23. Giurgica-Tiron T, et al. Low depth algorithms for quantum amplitude estimation. 2012.03348.
24. Plekhanov K, et al. Variational quantum amplitude estimation. 2109.03687.
25. Herbert S. The problem with grover-rudolph state preparation for quantum Monte-Carlo. Phys Rev E. 2021;103:063302.
26. Vedral V et al. Quantum networks for elementary arithmetic operations. Phys Rev A. 1996;54:147.
27. Beckman D et al. Efficient networks for quantum factoring. Phys Rev A. 1996;54:1034.
28. Draper TG. Addition on a quantum computer. quant-ph/0008033.
29. Cuccaro SA et al. A new quantum ripple-carry addition circuit. In: The eighth workshop on quantum information processing. 2004.
30. Takahashi Y et al. A linear-size quantum circuit for addition with no ancillary qubits. Quantum Inf Comput. 2005;5(6):440–8.
31. Van Meter R et al. Fast quantum modular exponentiation. Phys Rev A. 2005;71(5):052320.
32. Draper TG et al. A logarithmic-depth quantum carry-lookahead adder. Quantum Inf Comput. 2006;6(4):351.
33. Takahashi Y et al. Quantum addition circuits and unbounded fan-out. Quantum Inf Comput. 2010;10(9–10):0872.
34. Portugal R et al. Reversible Karatsubas algorithm. J Univers Comput Sci. 2006;12(5):499.
35. Alvarez-Sanchezet JJ et al. A quantum architecture for multiplying signed integers. J Phys Conf Ser. 2008;128(1):012013.
36. Takahashi Y et al. A fast quantum circuit for addition with few qubits. Quantum Inf Comput. 2008;8(6):636.
37. Thapliyal H. Mapping of subtractor and adder-subtractor circuits on reversible quantum gates. Transactions on Computational Science XXVII. 2016;**10**.
38. Thapliyal H, Ranganathan N. Design of efficient reversible logic based binary and BCD adder circuits. ACM J Emerg Technol Comput Syst. 2013;9:17.
39. Lin C-C et al. Qlib: quantum module library. ACM J Emerg Technol Comput Syst. 2014;11(1):7:1–7:20.
40. Babu HMH. Cost-efficient design of a quantum multiplier-accumulator unit. Quantum Inf Process. 2016;16(1):30.
41. Jayashree HV et al. Ancilla-input and garbage-output optimized design of a reversible quantum integer multiplier. J Supercomput. 2016;72(4):1477.
42. Muñoz-Coreas E, Thapliyal H. Quantum circuit design of a T-count optimized integer multiplier. IEEE Trans Comput. 2019;68:5.
43. Khosropour A et al. Quantum division circuit based on restoring division algorithm. In: Information technology: new generations (ITNG), 2011 eighth international conference on. Las Vegas: IEEE; 2011. p. 1037–40.
44. Jamal L, Babu HMH. Efficient approaches to design a reversible floating point divider. In: 2013 IEEE international symposium on circuits and systems (ISCAS2013). 2013. p. 3004–7.
45. Dibbo SV et al. An efficient design technique of a quantum divider circuit. In: 2016 IEEE international symposium on circuits and systems (ISCAS). 2016. p. 2102–5.
46. Thapliyal H et al. Quantum circuit designs of integer division optimizing T-count and T-depth. In: IEEE transactions on emerging topics in computing. 2019.
47. Amy M, Maslov D, Mosca M. IEEE Trans CAD. 2014;33(10):1476.
48. Maslov D. On the advantages of using relative phase Toffolis with an application to multiple control Toffoli optimization. Phys Rev A. 2016;93:022311.
49. O'Neill ME. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. Harvey Mudd College Computer Science Department Tachnical Report. 2014. http://www.pcg-random.org/.
50. Hörmann W, Leydold J. Continuous random variate generation by fast numerical inversion. ACM Trans Model Comput Simul. 2003;13(4):347.
51. Haner T, et al. Optimizing Quantum Circuits for Arithmetic. 1805.12445.
52. Muñoz-Coreas E, Thapliyal H. T-count and qubit optimized quantum circuit design of the non-restoring square root algorithm. ACM J Emerg Technol Comput Syst. 2018;14:3.
53. Kliuchnikov V et al. Practical approximation of single-qubit unitaries by single-qubit quantum Clifford and T circuits. IEEE Trans Comput. 2016;65(1):161.