IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Digital Object Identifier 10.1109/ACCESS.2024.0429000

Quantum and Classical Methods for Feynman-Kac PDEs: A Comparative Study

PHUONG-NAM NGUYEN[1]
[1]Faculty of Information Technology, School of Technology, National Economics University, Hanoi, 100000, Vietnam (e-mail: namnp@neu.edu.vn)
Corresponding author: Phuong-Nam Nguyen (e-mail: namnp@neu.edu.vn)

This work was supported in part by G.A.I.A QTech LLC and The Laboratory of Quantum Algorithms and Advanced Analytics (QAAA Lab, School of Technology, National Economics University), Vietnam

ABSTRACT This paper explores quantum computing for expectation computations in the Feynman-Kac formula and PDEs, with a focus on option pricing under Black-Scholes dynamics. It evaluates whether quantum algorithms, particularly quantum amplitude estimation, can outperform classical Monte Carlo in solving stochastic differential equations. We extend quantum approaches to multi-dimensional problems, comparing their accuracy and efficiency against classical and machine learning-based solvers. While quantum simulations offer speed advantages, challenges like computational complexity, state preparation, and scalability remain. Quantum amplitude estimation offers theoretical speedup but is constrained by hardware and error mitigation. Multi-dimensional extensions raise computational costs, while machine learning-based PDE solvers need optimization due to high overhead. Quantum computing shows promise for financial PDEs but requires hybrid frameworks, optimized circuits, and better hardware. Future work will refine quantum algorithms for high-dimensional models and improve machine learning-based solvers.

INDEX TERMS Feynman-Kac formula, quantum computing, Black-Scholes dynamics, quantum amplitude estimation, Monte Carlo simulation, machine learning, PDE solvers, financial derivatives.

I. INTRODUCTION

Partial differential equations (PDEs) play a crucial role in numerous scientific and engineering disciplines, serving as fundamental tools in physics, finance, and applied mathematics. Among these, the Feynman-Kac formula [8] provides a critical bridge between stochastic differential equations (SDEs) and PDEs, allowing solutions to be expressed as expectation values of stochastic processes. This connection has been extensively utilized in fields such as quantitative finance, where pricing derivatives often relies on solving the Black-Scholes PDE [2] where it aids in modeling diffusion and transport phenomena.

Traditional methods for solving PDEs associated with the Feynman-Kac representation rely on numerical approaches such as finite difference schemes [3], [9], finite element methods [5], [7], and Monte Carlo simulations [12]. While these techniques are well-established, they often suffer from computational inefficiencies, particularly when dealing with high-dimensional problems. In recent years, the emergence of quantum computing has opened new possibilities for simulation techniques and machine learning [13]–[20]. Quantum amplitude estimation [4], [6], [21], [22], for instance, offers a quadratic speedup over classical Monte Carlo methods, suggesting potential advantages in solving PDEs via the Feynman-Kac framework.

Despite the theoretical promise of quantum computing in expectation estimation, several open questions remain. First, the efficiency and accuracy of quantum algorithms in computing expectation values compared to classical Monte Carlo methods are not yet fully explored to extend of our knowledge. Second, while quantum techniques have been applied to one-dimensional problems, their extension to multi-dimensional diffusion processes remains an area of ongoing investigation. Finally, simulating the underlying SDE on quantum hardware and integrating them into the Feynman-Kac representation is a largely unexplored frontier [1].

This paper investigates the application of quantum computing techniques to solving PDEs via the Feynman-Kac formula. Specifically, we:

1) Evaluate the performance of quantum amplitude estimation for computing expectation values in comparison to classical Monte Carlo methods.
2) Extend quantum computing methods to multi-dimensional diffusion processes and assess their efficiency relative to traditional approaches.
3) Explore the feasibility of simulating SDEs on quantum hardware and its implications for PDE solutions.
4) Benchmark the quantum approach against classical nu-

merical methods, including finite difference, spectral, and neural operator-based techniques, in the context of the Black-Scholes equation.

By addressing these questions, we aim to contribute to the growing intersection of quantum computing, stochastic processes, and numerical PDE methods, offering insights into the potential of quantum-enhanced PDE solvers.

PROBLEM STATEMENT

Given the PDE problem:

$$\frac{\partial V}{\partial t}+\frac{1}{2}\sigma(S_t,t)^2\frac{\partial^2 V}{\partial S_t^2}+\mu(S_t,t)\frac{\partial V}{\partial S_t}-r(t)V(S_t,t)=0, \quad (1)$$

with boundary condition $V(S_T,T)=\Psi(S_T)$, where μ, σ are known functions of S_t and t, and r and Ψ are functions of t and S_T, respectively, with $t<T$. By applying Itô's formula on the process

$$Z_u = e^{-\int_u^T r(v)dv}V(S_u,u), \quad (2)$$

where S_t satisfies the generalized SDE

$$dS_t=\mu(S_t,t)\,dt+\sigma(S_t,t)\,dW_t, \quad (3)$$

such that $\{W_t : t\geq 0\}$ is a standard Wiener process on the probability space $(\Omega,\mathcal{F},\mathbb{P})$, show that under the filtration $\mathcal{F}_t$, the solution of the PDE is given by

$$V(S_t,t)=\mathbb{E}\left[e^{-\int_t^T r(v)dv}\Psi(S_T)\mid\mathcal{F}_t\right]. \quad (4)$$

Following are a few contemporary gaps in research that connect the traditional study of the Feynman-Kac formula for one-dimensional diffusion processes to modern-day quantum computing technologies. In this article, we aim to answer the research questions below

- RQ1 Can quantum algorithms to compute the expectation in Equation 4 more efficiently than classical Monte Carlo methods?
- RQ2 Can the quantum computing techniques developed for the one-dimensional Feynman–Kac formula be extended to multi-dimensional diffusion processes? How good is quantum approaches compared to existing methods?
- RQ3 How can one efficiently simulate the SDE in Equation 3 on a quantum computer, and how does this simulation feed into solving the corresponding PDE via the Feynman–Kac representation?

These gaps are intended to broaden the scope of applied Feynman-Kac representation for concerned disciplines like finance or physics, and at the same time consider the possibility of quantum algorithms providing some benefit in the solution of the associated PDEs and in the evaluation of the expectation values.

TABLE 1. Quantum amplitude estimation vs. classical Monte Carlo with $\mathbb{E}_{\text{true}} = 0.42$

ϵ_{target}	Est. Amplitude	Abs. Error	Runtime [s]
5.000×10^{-2}	4.246×10^{-1}	4.634×10^{-3}	1.398×10^{-1}
1.000×10^{-2}	4.201×10^{-1}	1.390×10^{-4}	1.093×10^{-1}
5.000×10^{-3}	4.200×10^{-1}	4.800×10^{-5}	$\mathbf{9.589\times10^{-2}}$
1.000×10^{-3}	4.201×10^{-1}	1.217×10^{-4}	9.741×10^{-2}
$\mathbf{1.000\times10^{-4}}$	$\mathbf{4.200\times10^{-1}}$	$\mathbf{8.000\times10^{-7}}$	2.495×10^{0}
#Runs	**Est. Amplitude**	**Abs. Error**	**Runtime [s]**
1.000×10^{3}	3.990×10^{-1}	2.100×10^{-2}	$\mathbf{9.341\times10^{-3}}$
1.000×10^{4}	4.288×10^{-1}	8.800×10^{-3}	2.168×10^{-2}
1.000×10^{5}	4.221×10^{-1}	2.060×10^{-3}	1.420×10^{-1}
1.000×10^{6}	4.203×10^{-1}	3.080×10^{-4}	1.329×10^{0}
$\mathbf{1.000\times10^{7}}$	$\mathbf{4.200\times10^{-1}}$	$\mathbf{8.200\times10^{-6}}$	1.330×10^{1}

II. EXPECTATION COMPUTATION WITH QUANTUM CIRCUITS

A. EXPERIMENTAL SETTING

We start with a toy example the amplitude we estimate plays the role of the expected value in Equation 4 which in a realistic Feynman–Kac application would be obtained by encoding the stochastic process S_T, the discount factor $e^{-\int_t^T r(v)dv}$, and the payoff function $\Psi(S_T)$ into a quantum state. In this demonstration we assume that the expectation (or "target amplitude") is known classically and it is equal to $\mathbb{E}_{\text{true}}=0.42$ so that we can verify that our quantum algorithm reproduces it[1] The key point is that amplitude estimation is known to yield a quadratic speed-up over classical Monte Carlo when estimating such expectation values.

State preparation

The Code Exhibit A-A builds a one-qubit circuit that prepares the state

$$\cos\left(\frac{\theta}{2}\right)|0\rangle+\sin\left(\frac{\theta}{2}\right)|1\rangle, \quad (5)$$

where the probability $a=\sin^2(\theta/2)$ is set to the target amplitude (here $\mathbb{E}_{\text{true}}=0.42$). We set up an estimation function with our state-preparation circuit and specify that registered qubit 0 is the objective qubit (whose $|1\rangle$-amplitude is the value to estimate). The `Qiskit` library is used with backend `qasm_simulator` using $10,000$ shots. We choose a target precision ϵ_{target} (here 0.01) and a significance level $\alpha=0.05$. We compare the quantum approach with classical Monte Carlo with number of runs $\{1k;10k;100k;1M;10M\}$, $[k]$ is thousands and $[M]$ is millions.

B. NUMERICAL RESULTS

Table 1 compares the performance of two methods for estimating a probability amplitude with $\mathbb{E}_{\text{true}}=0.42$. The code to reproduce this evaluation is reported in SuppMat-1.

As the target precision ϵ_{target} decreases, the estimated amplitude converges toward the true value. For instance, with $\epsilon_{\text{target}}=0.01$ the absolute error is very low (approximately 1.4×10^{-4}), and it further decreases to nearly zero (8×10^{-7})

[1] In practice the amplitude would be unknown and obtained by the quantum algorithm.

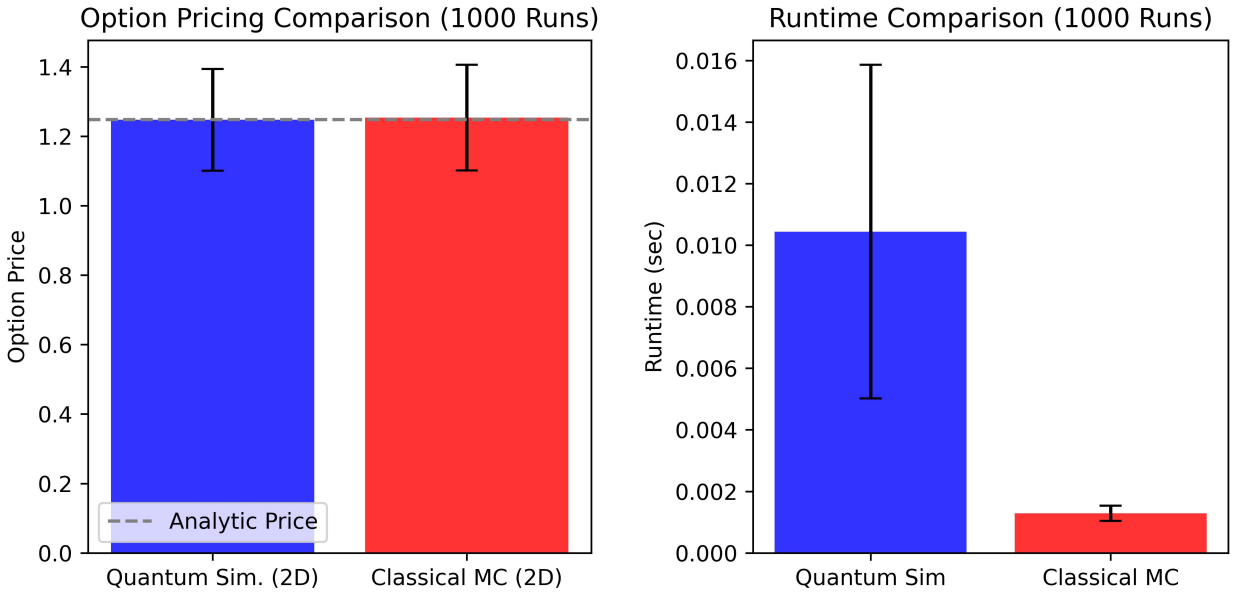

FIGURE 1. **Simulation results for option pricing in Section III**

for $\epsilon_{\text{target}} = 0.0001$. For moderate target precisions (0.01, 0.005, 0.001), the runtime is around 0.1 seconds. However, pushing for very high precision (0.0001) significantly increases the runtime (to about 2.5 seconds). This behavior is consistent with the theoretical expectation that quantum amplitude estimation requires $O(1/\epsilon)$ queries [23].

As the number of runs increases, the classical Monte Carlo estimate converges to the true amplitude. Specifically, with $1k$ samples the error is 0.021, which decreases to approximately 3.08×10^{-4} with $1M$ samples, and further down to 8.2×10^{-6} with $10M$ samples. This is consistent with the classical Monte Carlo error scaling of $O(1/\sqrt{N})$[2].

Remark 1. The quantum circuit outperforms classical Monte Carlo sampling in this task, in terms of both precision (8×10^{-7} v.s. 8×10^{-6}) and runtime ($2.49[s]$ v.s. $13.3[s]$)

Remark 2. By targeting a smaller ϵ_{target}, the quantum circuit can achieve very high precision. However, as ϵ_{target} decreases, the number of required quantum operations—and thus the runtime—increases linearly.

Conclusion 1. For RQ1: Quantum amplitude estimation offers speedup over classical Monte Carlo methods. However, we emphasize three main trade-off for using quantum approaches

Complexity vs. simplicity: Quantum algorithms use fewer samples but require complex state preparation, intricate circuits, and advanced error mitigation, posing implementation challenges on current hardware.

Overhead vs. asymptotic speedup: Theoretical speedup assumes ideal conditions, but circuit depth, noise, and adaptive techniques can reduce practical gains. High precision increases runtime due to longer circuits and error correction.

Robustness vs. efficiency: Classical Monte Carlo is robust but scales poorly with precision, while quantum methods offer better scaling but need calibration and adaptive techniques for hardware imperfections.

III. EXTENSION TO MULTI-DIMENSIONAL CASE

A. EXPERIMENTAL SETTING

In this section, we consider a digital (binary) option pricing problem with the following parameters: $S_0 = 100$ is the initial asset price; $K = 100$ is the strike price; $r = 0.05$ is the risk-free interest rate; $T = 1.0$ is the normalized time to maturity; and $\Pi = 21$ is payoff if the option expires in-the-money. The option pays a fixed amount Π if the underlying asset price meets a certain condition at expiration. Here, the probability of this condition being met independently in two dimensions is given as 0.25 in each dimension. The present value of the expected payoff is given by the discounted expectation (analytic): $V_0 = e^{-rT}\cdot\Pi\cdot p^2$. We aim to predict

$$V_{\text{true}} = \exp(-0.05)\cdot 21\cdot 0.25^2 = \boxed{1.2485} \tag{6}$$

[2]A classical Monte Carlo estimate for an integral is computed as the sample mean of independent evaluations, so by the central limit theorem the error (standard deviation) of the estimate decreases like

$$\text{RMSE} = \frac{\sigma}{\sqrt{N}},$$

which is commonly expressed as having error scaling $O(1/\sqrt{N})$. In other words, to reduce the error by a factor of 10 one must increase the number of samples by a factor of 100.

State preparation

We prepare

$$|\psi\rangle = a_0|00\rangle + a_1|01\rangle + a_2|10\rangle + a_3|11\rangle \tag{7}$$

with the branch $|10\rangle$ (a_2) gives payoff p_{pred}. The initial state is set to $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}})$. We use `SLSQP` optimizer in `scipy.optimize` package and `Aer.qasm_simulator` with 1000 shots. For classical MC, we also use 1000 runs. For 2 dimensions, we desire the joint outcome $|01\rangle \otimes |01\rangle = [1010]^\intercal$. The the predicted payoff is computed by

$$p_{\text{pred}} = \frac{\text{Count}([1,0,1,0])}{\#\text{shots}} \tag{8}$$

Then, the predicted option price is compute by

$$V_{\text{pred}} = \exp(-0.05) \cdot 21 \cdot p_{\text{pred}}^2 \tag{9}$$

The simulated errors is computed by

$$\text{error} = |V_{\text{pred}} - V_{\text{true}}| \tag{10}$$

For evaluation, we perform $1,000$ independent runs for each model.

B. NUMERICAL RESULTS

The numerical results is given in Figure 1. The proposed quantum simulation approach yields a mean price of 1.2473 ± 0.1468 (mean $\pm$ std.) and the classical MC results in a slightly higher mean price of 1.2539 ± 0.1521. The small difference in mean values suggests both methods converge to similar price estimates (Equation 6). However, the slightly lower standard deviation in quantum simulation indicates marginally reduced variability in pricing results. The absolute error mean for quantum simulation is 0.1151 ± 0.0911, which is slightly lower than the 0.1225 ± 0.0904 observed in classical MC.A lower mean absolute error suggests the quantum simulation may have a slight accuracy advantage in estimating the price. Quantum simulation has an average runtime of 0.0104 ± 0.0054 seconds, while classical MC, in contrast, is significantly faster, with an average runtime of 0.0013 ± 0.0002 seconds. This highlights a major computational trade-off: quantum simulation takes approximately 8 times longer than classical MC, which may be a limiting factor for real-time applications. The code to reproduce this evaluation is given in SuppMat-2.

Conclusion 2. For RQ2: We have extend the quantum approach into 2-dimensional problem and the quantum approach still show advantage in term of accuracy over the classical counterpart. However, the trade-off of runtime and computational resources is significant.

C. GENERALIZATION FOR MULTI-DIMENSIONAL CASE

Consider a d-dimensional stochastic process $\mathbf{S}_t \in \mathbb{R}^d$ governed by the stochastic differential equation (SDE)

$$d\mathbf{S}_t = \boldsymbol{\mu}(\mathbf{S}_t, t)\, dt + \Sigma(\mathbf{S}_t, t)\, d\mathbf{W}_t, \tag{11}$$

where

- $\boldsymbol{\mu} : \mathbb{R}^d \times [0,T] \to \mathbb{R}^d$ is the drift,
- $\Sigma(\mathbf{S}_t, t)$ is a $d \times d$ volatility matrix,
- $d\mathbf{W}_t$ is a d-dimensional Wiener process.

If we considers a linear parabolic PDE of the form

$$\frac{\partial V}{\partial t} + \mathcal{L}V - r(t)V = 0, \quad V(\mathbf{S}, T) = \Psi(\mathbf{S}), \tag{12}$$

with the differential operator (generator)

$$\begin{aligned}\mathcal{L}V(\mathbf{S},t) = &\sum_{i=1}^{d} \mu_i(\mathbf{S},t)\frac{\partial V}{\partial S_i}(\mathbf{S},t) \\ &+ \frac{1}{2}\sum_{i,j=1}^{d} \left(\Sigma\Sigma^\top\right)_{ij}(\mathbf{S},t)\frac{\partial^2 V}{\partial S_i \partial S_j}(\mathbf{S},t),\end{aligned} \tag{13}$$

then by the multi-dimensional Feynman–Kac theorem the solution is given by

$$V(\mathbf{S}_t, t) = \mathbb{E}\left[e^{-\int_t^T r(v)\,dv}\Psi(\mathbf{S}_T) \,\middle|\, \mathbf{S}_t = \mathbf{S}\right]. \tag{14}$$

To simulate this on a quantum computer, we partition $\mathbb{R}^d$ (or a bounded domain thereof) into a grid $\{\mathbf{x}_j\}_{j=1}^N$. Then, we encode the probability distribution over outcomes by preparing a quantum state

$$|\psi\rangle = \sum_{j=1}^{N} \sqrt{p_j}\,|\mathbf{j}\rangle, \tag{15}$$

where p_j is the probability that $\mathbf{S}_T \approx \mathbf{x}_j$. Quantum amplitude estimation (or any related subroutines) can then be used to efficiently compute the expectation

$$V(\mathbf{S}_t, t) = e^{-\int_t^T r(v)\,dv} \sum_{j=1}^{N} \Psi(\mathbf{x}_j)\, p_j. \tag{16}$$

The implementation of multi-dimensional case is left for future research.

IV. BENCHMARKING SIMULATION METHODS FOR BLACK-SCHOLES DYNAMICS

A. EXPERIMENTAL SETTING

The Feynman–Kac theorem provides a powerful link between certain partial differential equations (PDEs) and stochastic processes. In the context of option pricing, the Black–Scholes price $C(S,t)$ satisfies the following PDE:

$$\frac{\partial C}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 C}{\partial S^2} + rS\frac{\partial C}{\partial S} - rC = 0, \tag{17}$$

with the terminal (payoff) condition:

$$C(S,T) = \max\{S - K,\, 0\}. \tag{18}$$

According to the Feynman–Kac theorem, the solution to this PDE can be represented as the expected value of the discounted payoff under the risk-neutral measure:

$$C(S_0, 0) = \mathbb{E}\left[e^{-rT}\max\{S(T) - K,\, 0\} \,\middle|\, S(0) = S_0\right] \tag{19}$$

TABLE 2. Benchmark comparison for modeling of Black-Scholes dynamics

Method	Mean Price ± Std	Mean Error ± Std	Mean Runtime ± Std
Monte Carlo [A-C1]	10.4481 ± 0.0128	0.0099 ± 0.0085	0.0868 ± 0.0087
Finite Difference [A-C2]	10.4544 ± 0.0000	0.0038 ± 0.0000	0.1368 ± 0.0099
Method of Lines [A-C3]	10.4544 ± 0.0000	0.0038 ± 0.0000	6.7122 ± 0.5914
Spectral Method [A-C4]	200.0000 ± 0.0000	189.5494 ± 0.0000	0.1406 ± 0.0271
Finite Element Method [A-C5]	7.8280 ± 0.0000	2.6226 ± 0.0000	0.2672 ± 0.0068
Deep Ritz [24]	66.4443 ± 0.5092	55.9937 ± 0.5092	42.3329 ± 1.7106
DeepONet [11]	11.0444 ± 2.8957	2.3156 ± 1.8373	22.9071 ± 0.1651
Fourier Neural Operator [10]	10.8387 ± 1.4548	1.3957 ± 0.5649	150.0197 ± 0.6010
Quantum simulation	10.2676 ± 0.7849	0.6392 ± 0.4909	0.0332 ± 0.0808

Thus, the Black–Scholes formula, which expresses the option price as:

$$C(S_0, T) = S_0\, N(d_1) - K\, e^{-rT}\, N(d_2), \tag{20}$$

is derived by applying the Feynman–Kac theorem to solve the Black–Scholes PDE.

Let $S(t)$ denote the price of the underlying asset at time t. Under the risk-neutral measure, the asset price is assumed to follow a geometric Brownian motion:

$$dS(t) = r\, S(t)\, dt + \sigma\, S(t)\, dW(t), \quad S(0) = S_0, \tag{21}$$

where S_0 is the initial asset price; r is the risk-free interest rate; σ is the volatility of the asset; $W(t)$ is a standard Brownian motion. For a European call option with strike price K and maturity T, the payoff at expiration is

$$\text{Payoff} = \max\{S(T) - K,\, 0\}. \tag{22}$$

The risk-neutral pricing principle states that the price $C(S_0, T)$ of the call option at time $t = 0$ is the discounted expected payoff under the risk-neutral measure

$$C(S_0, T) = e^{-rT}\, \mathbb{E}\left[\max\{S(T) - K,\, 0\}\right]. \tag{23}$$

The Black–Scholes formula provides a closed-form solution for this expectation

$$C(S_0, T) = S_0\, N(d_1) - K\, e^{-rT}\, N(d_2), \tag{24}$$

where

$$\begin{aligned} d_1 &= \frac{\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{1}{2}\sigma^2\right) T}{\sigma\sqrt{T}}, \\ d_2 &= d_1 - \sigma\sqrt{T}, \end{aligned} \tag{25}$$

and $N(\cdot)$ denotes the cumulative distribution function (CDF) of the standard normal distribution. The parameters used for our evaluation are

$$\begin{aligned} S_0 = 100, \quad K = 100, \\ r = 0.05, \sigma = 0.2, \quad T = 1.0. \end{aligned} \tag{26}$$

Thus, the true value must be predicted is

$$\boxed{S_{\text{True}} = 10.4506} \tag{27}$$

B. NUMERICAL RESULTS

The numerical and machine learning approaches capture the tradeoffs between accuracy, consistency, and the speed of computations as shown in the Table 2.

The mean price of 10.4544 for the Finite Difference (Appendix A-C2) and Method of Lines (Appendix A-C3) is accompanied with an error of only 0.0038 and a standard deviation of zero indicating extreme robustness. Monte Carlo (Appendix A-C1) has slightly lower accuracy with a mean price of 10.4481 and error of 0.0099 but greatly improved speed of computations as shown in the runtime of 0.0868. The spectral method (Appendix A-C4) with a mean price of 200 and an error of 189.5494 seems to perform with such extreme deviations that it is safe to say is unstable.

Mean price for Deep Ritz [24] of 66.4443 and the error of 55.9937 along with mean price of 11.0444 for DeepONet [11] with a considerably lower error makes these two methods rather useful. But their long runtimes suggest the need for further calibration to dramatically reduce them. The Fourier Neural Operator seems to be able to to extract some features underneath with an error of 1.3957 against mean price of 10.8387, but does so at a ridiculously high computational cost of runtime 150.02[s].

With quantum simulation, speed is fast as seen in its remarkable runtime of 0.0332 while achieving hight accuracy with a mean price of 10.2676, error of 0.6392.

Remark 3. The machine learning–based approaches (Deep Ritz, DeepONet, Fourier Neural Operator) show promise but currently suffer from higher runtimes, larger errors, and increased variability. Further work in network architecture optimization, training stability, and hyperparameter tuning is needed to make these methods competitive with traditional approaches.

Conclusion 3. For RQ3: Although the quantum simulation method does not achieve the lowest error, its unparalleled runtime performance suggests it may be a strong candidate for applications where speed is the highest priority. Future research should aim to reduce the variability in its outputs and improve its accuracy.

V. CONCLUSION

The presented work has explored the application of quantum computing techniques to solve expectation computations related to the Feynman-Kac formula and PDEs, with a focus on option pricing under Black-Scholes dynamics. Our findings indicate that quantum algorithms, particularly quantum amplitude estimation, provide a potential speedup over classical Monte Carlo methods (Section II). The extension to multi-dimensional cases demonstrated that quantum approaches could maintain accuracy advantages, albeit with higher computational costs (Section III). Additionally, benchmarking against traditional numerical methods and machine learning-based approaches showed that while quantum simulations offer notable speed benefits, their accuracy still falls short compared to established techniques like finite difference methods (Section IV). Overall, our study highlights the potential of quantum computing for PDE-based financial applications, while also underscoring the challenges that must be addressed for practical deployment. Future research should focus on refining quantum techniques and integrating them into hybrid frameworks for more efficient market simulations and risk assessments.

Despite the promising results, several challenges remain. Current quantum devices are still in early development, and real-world applications require fault-tolerant quantum systems. Quantum approaches require complex encoding and error mitigation techniques, which can offset theoretical speed advantages. While quantum simulations reduce runtime in some cases, classical Monte Carlo remains more robust and scales better for high-precision tasks. Extending quantum methods to multi-dimensional PDEs remains computationally expensive and requires further refinement.

Several avenues for future research emerge from our study. First, optimizing quantum circuits for near-term quantum hardware can help bridge the gap between theoretical and practical performance. Second, leveraging quantum techniques in conjunction with classical solvers could balance efficiency and accuracy. Third, extending quantum PDE solvers to more complex financial models, such as regime-switching or mean-reverting processes, would be valuable. Finally, deep learning approaches such as DeepONet and Fourier Neural Operators require further tuning to enhance accuracy while reducing computational costs.

REFERENCES

[1] H. Alghassi, A. Deshmukh, N. Ibrahim, N. Robles, S. Woerner, and C. Zoufal. A variational quantum algorithm for the feynman-kac formula. *Quantum*, 6:730, 2022.
[2] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637–654, 1973.
[3] G. Boole and J. F. Moulton. *A treatise on the calculus of finite differences*. Macmillan and company, 1872.
[4] G. Brassard, P. Hoyer, M. Mosca, and A. Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.
[5] R. Courant et al. Variational methods for the solution of problems of equilibrium and vibrations. *Lecture notes in pure and applied mathematics*, pages 1–1, 1994.
[6] D. Grinko, J. Gacon, C. Zoufal, and S. Woerner. Iterative quantum amplitude estimation. *npj Quantum Information*, 7(1):52, 2021.
[7] A. Hrennikoff. Solution of problems of elasticity by the framework method. 1941.
[8] M. Kac. On distributions of certain wiener functionals. *Transactions of the American Mathematical Society*, 65(1):1–13, 1949.
[9] R. J. LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.
[10] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.
[11] L. Lu, P. Jin, and G. E. Karniadakis. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *arXiv preprint arXiv:1910.03193*, 2019.
[12] N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949.
[13] N. Nguyen and K.-C. Chen. Bayesian quantum neural networks. *IEEE Access*, 10:54110–54122, 2022.
[14] N. Nguyen and K.-C. Chen. Quantum embedding search for quantum machine learning. *IEEE Access*, 10:41444–41456, 2022.
[15] P.-N. Nguyen. Biomarker discovery with quantum neural networks: a case-study in ctla4-activation pathways. *BMC bioinformatics*, 25(1):149, 2024.
[16] P.-N. Nguyen. The duality game: a quantum algorithm for body dynamics modeling. *Quantum Information Processing*, 23(1):21, 2024.
[17] P.-N. Nguyen. Quantum dna encoder: A case-study in grna analysis. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 232–239. IEEE, 2024.
[18] P.-N. Nguyen. A quantum neural network for sequential data analysis in machine learning. *Quantum Machine Intelligence*, 6(2):88, 2024.
[19] P.-N. Nguyen. Quantum tunneling with linear potential: Case studies in biological processes. *IEEE Transactions on Molecular, Biological, and Multi-Scale Communications*, 2024.
[20] P.-N. Nguyen. Quantum word embedding for machine learning. *Physica Scripta*, 99(8):086004, 2024.
[21] K. Plekhanov, M. Rosenkranz, M. Fiorentini, and M. Lubasch. Variational quantum amplitude estimation. *Quantum*, 6:670, 2022.
[22] P. Rall and B. Fuller. Amplitude estimation from quantum signal processing. *Quantum*, 7:937, 2023.
[23] A. Roux and T. Zastawniak. Accelerated quantum amplitude estimation without qft. *arXiv preprint arXiv:2407.16795*, 2024.
[24] B. Yu et al. The deep ritz method: a deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 6(1):1–12, 2018.

NAM NGUYEN received a B.S. in Mathematics and Education from Hanoi University of Education (2018) and an M.A. in Statistics from the University of South Florida (2019). He is currently a lecturer at the School of Technology, National Economics University, Vietnam. His research focuses on quantum computing, quantum machine intelligence, and artificial intelligence.

APPENDIX A CODE EXHIBITS

A. EXPECTATION COMPUTATION WITH QUANTUM CIRCUITS

```
def create_state_preparation(probability: float)
    -> QuantumCircuit:
    qc = QuantumCircuit(1)
    theta = 2 * np.arcsin(np.sqrt(probability))
    qc.ry(theta, 0)
    return qc
```

This function creates a 1-qubit circuit that rotates $|0\rangle$ to quantum state in Equation 5. Then, we compute the rotation angle θ by

$$\theta = 2\arcsin(\sqrt{p}) \tag{28}$$

with p is the encoded probability. Of note, we do not add measurements here because the amplitude estimation algorithm uses the state-vector (or equivalent) to extract the amplitude.

B. EXTENSION TO MULTI-DIMENSIONAL CASE

1) 2D quantum simulator

```
initial_state
    = np.array([0.7071, 0, 0.7071, 0.7071])
constraints
    = [{'type': 'eq', 'fun': norm_constraint}]
bounds
    = [(-2*np.pi, 2*np.pi)] * 4
result_opt
    = minimize(objective,
        initial_state,
        args=(r, T,
            target_price_1d,
            payoff),
        method='SLSQP', bounds=bounds, constraints
    =constraints,
        options={'ftol': 1e-12})
joint_state
    = np.kron(optimized_state,
        optimized_state)
qc_quantum
    = QuantumCircuit(4)
qc_quantum.initialize(joint_state,
    [0, 1, 2, 3])
qc_quantum.measure_all()

backend_qasm
    = Aer.get_backend("qasm_simulator")
shots = 1000
start_qc = time.time()
job = execute(qc_quantum, backend_qasm, shots=
    shots)
result_qc = job.result()
counts = result_qc.get_counts(qc_quantum)
end_qc = time.time()
runtime_quantum = end_qc - start_qc

p_joint = counts.get("1010", 0) / shots
price_quantum_measured = exp(-r*T) * payoff *
    p_joint
```

2) Monte Carlo simulator

```
def monte_carlo_2d(n_samples, r, T, payoff=21):
    outcomes = []
    for _ in range(n_samples):
        outcome1 = payoff
            if np.random.rand() < 0.25
            else 0
        outcome2 = payoff
            if np.random.rand() < 0.25
            else 0
        outcomes.append(payoff if
            (outcome1 == payoff)
            and outcome2 == payoff)
            else 0)
    return exp(-r*T) * np.mean(outcomes)
```

C. BENCHMARKING SIMULATION METHODS FOR BLACK-SCHOLES DYNAMICS

1) Monte Carlo implementation

```
def monte_carlo_price(S0, K, r, sigma, T, N
    =1000000):
    Z = np.random.randn(N)
    ST = S0 * np.exp((r - 0.5*sigma2)*T + sigma*np
    .sqrt(T)*Z)
    payoff = np.maximum(ST - K, 0)
    price = np.exp(-r*T) * np.mean(payoff)
    return price
```

2) Finite Difference implementation

```
def finite_difference_price(S0, K, r, sigma, T,
    S_max=300, M=200, N=200):
    # Set up spatial grid
    ds = S_max / M
    dt = T / N
    S_grid = np.linspace(0, S_max, M+1)
    # Terminal condition: payoff at T
    V = np.maximum(S_grid - K, 0)
    # Precompute coefficients for interior nodes
    A = np.zeros((M-1, M-1))
    B = np.zeros((M-1, M-1))
    for i in range(1, M):
        S = i * ds
        a = 0.25 * dt * (sigma2 * (i2) - r * i)
        b = -0.5 * dt * (sigma2 * (i2) + r)
        c = 0.25 * dt * (sigma2 * (i2) + r * i)
        idx = i - 1
        if idx > 0:
            A[idx, idx-1] = -a
            B[idx, idx-1] = a
        A[idx, idx] = 1 - b
        B[idx, idx] = 1 + b
        if idx < M-2:
            A[idx, idx+1] = -c
            B[idx, idx+1] = c
    # Time-stepping (backward in time)
    for j in range(N):
        V_interior = V[1:M]
        RHS = B.dot(V_interior)
        t = T - j * dt
        # Incorporate boundary conditions:
        # V(0)=0 and V(S_max)=S_max - K*exp(-r*(T-
    t))
        RHS[0] += 0  # since V(0)=0
        RHS[-1] += (0.25*dt*(sigma2 * (M2) + r*M))
     * (S_max - K*np.exp(-r*t))
        V[1:M] = np.linalg.solve(A, RHS)
    price = np.interp(S0, S_grid, V)
    return price
```

3) Method of Lines implementation

```
def method_of_lines_price(S0, K, r, sigma, T,
    S_max=300, M=200):
    ds = S_max / M
    S_grid = np.linspace(0, S_max, M+1)
```

```
    # Terminal condition
    V_T = np.maximum(S_grid - K, 0)
    def ode_system(t, V):
        dVdt = np.zeros_like(V)
        # Use central finite differences for
    interior points
        for i in range(1, M):
            S = S_grid[i]
            dVdx = (V[i+1] - V[i-1]) / (2*ds)
            d2Vdx2 = (V[i+1] - 2*V[i] + V[i-1]) /
    (ds2)
            dVdt[i] = -0.5 * sigma2 * S2 * d2Vdx2
    - r * S * dVdx + r * V[i]
        dVdt[0] = 0   # Boundary at S=0
        dVdt[M] = -r * V[M]  # Approximate
    boundary at S_max
        return dVdt
    sol = solve_ivp(ode_system, [T, 0], V_T,
    method='RK45', t_eval=[0])
    V0 = sol.y[:, -1]
    price = np.interp(S0, S_grid, V0)
    return price
```

4) Spectral method implementation

We compute $N+1$ Chebyshev nodes $x_j = \cos(\pi j/N)$ and map them to the asset price domain by

$$S_j = \frac{x_j + 1}{2} S_{\max}.$$

The Chebyshev differentiation matrix D (and its square D^2) are computed using standard formulas. Because $S = \frac{x+1}{2} S_{\max}$, the derivatives with respect to S are obtained by scaling the x–derivatives:

$$\frac{d}{dS} = \frac{2}{S_{\max}} \frac{d}{dx}, \quad \frac{d^2}{dS^2} = \left(\frac{2}{S_{\max}}\right)^2 \frac{d^2}{dx^2}.$$

The Black–Scholes PDE is transformed into an ODE system in the "time" variable $\tau = T - t$ (so that the payoff becomes the initial condition). The PDE becomes

$$U_\tau = a(S) U_{SS} + b(S) U_S - rU,$$

where $a(S) = 0.5\sigma^2 S^2$ and $b(S) = rS$. We enforce the Dirichlet boundary conditions at $S = 0$ and $S = S_{\max}$ at each time step via a wrapper function. After integrating from $\tau = 0$ to $\tau = T$ (which corresponds to $t = 0$), the solution $U(S, T)$ is interpolated at $S = S_0$ to yield the option price. The code below is our implementation of spectral method to model the given Black-Scholes dynamics.

```
def spectral_method_price(S0, K, r, sigma, T,
    S_max=300, N=50):
    # Number of collocation points
    N_points = N + 1
    j = np.arange(0, N_points)
    # Chebyshev nodes in [-1, 1]
    x = np.cos(np.pi * j / N)
    # Map nodes to S: S = (x+1)/2 * S_max
    S_nodes = (x + 1) / 2 * S_max

    # Build the Chebyshev differentiation matrix
    D = np.zeros((N_points, N_points))
    # Set up the scaling factors c (with c[0] and
    c[-1] = 2, others = 1) and sign factors.
    c = np.ones(N_points)
    c[0] = 2
    c[-1] = 2
    c = c * ((-1)j)
    for i in range(N_points):
        for k in range(N_points):
            if i != k:
                D[i, k] = (c[i]/c[k]) / (x[i] - x[
    k])
    # Diagonal entries (Trefethen's formula)
    for i in range(N_points):
        if i == 0:
            D[i, i] = (2 * N2 + 1) / 6.0
        elif i == N:
            D[i, i] = -(2 * N2 + 1) / 6.0
        else:
            D[i, i] = -x[i] / (2*(1 - x[i]2))

    # Second derivative matrix in x
    D2 = np.dot(D, D)

    # Transform derivatives from x to S.
    # Since S = (x+1)/2 * S_max, we have d/dS =
    (2/S_max) d/dx and d^2/dS^2 = (2/S_max)^2 d^2/
    dx^2.
    D_S = (2 / S_max) * D
    D2_S = (2 / S_max)2 * D2

    # Set up the initial condition (terminal
    condition for Black-Scholes)
    # In variable tau = T - t, initial condition
    at tau=0 is U(S, 0) = max(S-K,0)
    U0 = np.maximum(S_nodes - K, 0)

    # Define the ODE system in tau
    # The transformed PDE is:
    #   U_tau = a(S) * U_SS + b(S) * U_S - r * U,
    # with a(S)=0.5*sigma^2*S^2, b(S)=r*S.
    def ode_system(tau, U):
        U_S_val = D_S.dot(U)
        U_SS_val = D2_S.dot(U)
        a = 0.5 * sigma2 * S_nodes2
        b = r * S_nodes
        F = a * U_SS_val + b * U_S_val - r * U
        # (Interior nodes: i = 1,...,N-1;
    boundaries are handled below.)
        return F

    # To enforce the Dirichlet boundary conditions
     at every time step,
    # we define a wrapper that resets the boundary
     values.
    def ode_system_bc(tau, U):
        F = ode_system(tau, U)
        # Enforce boundary conditions:
        U[0] = 0  # U(0,tau)=0
        U[-1] = S_max - K * np.exp(-r * tau)  # U(
    S_max,tau)=S_max - K*exp(-r*tau)
        F[0] = 0
        F[-1] = 0
        return F

    # Solve the ODE system forward in tau from 0
    to T
    sol = solve_ivp(ode_system_bc, [0, T], U0,
    method='BDF', t_eval=[T])
    U_final = sol.y[:, -1]  # solution at tau=T,
    which corresponds to t=0

    # Interpolate to get the price at S0
    price = np.interp(S0, S_nodes, U_final)
    return price
```

5) Finite element method

```
def fem_price(S0, K, r, sigma, T, kwargs):
```

```
    # Set parameters
    S_max = kwargs.get("S_max", 300)
    M = kwargs.get("M", 200)  # number of elements; there will be M+1 nodes
    N = kwargs.get("N", 200)  # number of time steps
    h = S_max / M
    dt = T / N

    # Spatial grid: nodes S_0, S_1, ..., S_M
    S_nodes = np.linspace(0, S_max, M+1)

    # Initialize global matrices for mass M_mat and operator A_mat (both (M+1)x(M+1))
    M_mat = np.zeros((M+1, M+1))
    A_mat = np.zeros((M+1, M+1))

    # Two-point Gauss-Legendre quadrature on reference interval [-1, 1]
    quad_pts = np.array([-1/np.sqrt(3), 1/np.sqrt(3)])
    quad_wts = np.array([1.0, 1.0])

    # Loop over elements (each element e spans nodes i and i+1)
    for i in range(M):
        a = S_nodes[i]
        b = S_nodes[i+1]
        he = b - a  # element length (should be equal to h)
        # Map quadrature points from [-1,1] to [a, b]: S = (he/2)*xi + (a+b)/2
        S_q = (he/2)*quad_pts + (a+b)/2
        # Jacobian for the transformation
        J = he / 2

        # Local basis functions and derivatives on element e:
        # phi0(S) = (b - S) / he, phi1(S) = (S - a) / he.
        # Their derivatives are constant: dphi0/dS = -1/he, dphi1/dS = 1/he.
        phi0 = (b - S_q) / he
        phi1 = (S_q - a) / he
        dphi0 = -np.ones_like(S_q) / he
        dphi1 = np.ones_like(S_q) / he

        # Initialize local matrices (2x2)
        M_local = np.zeros((2,2))
        A_local = np.zeros((2,2))

        # Loop over quadrature points to compute local integrals
        for q in range(len(quad_pts)):
            wq = quad_wts[q]
            S_val = S_q[q]
            # Mass term: phi_i * phi_j
            M_local[0,0] += phi0[q] * phi0[q] * wq * J
            M_local[0,1] += phi0[q] * phi1[q] * wq * J
            M_local[1,0] += phi1[q] * phi0[q] * wq * J
            M_local[1,1] += phi1[q] * phi1[q] * wq * J

            # Diffusion term: 0.5*sigma^2 * S^2 * phi'_i * phi'_j
            diff = 0.5 * sigma2 * S_val2
            A_local[0,0] += diff * dphi0[q] * dphi0[q] * wq * J
            A_local[0,1] += diff * dphi0[q] * dphi1[q] * wq * J
            A_local[1,0] += diff * dphi1[q] * dphi0[q] * wq * J
            A_local[1,1] += diff * dphi1[q] * dphi1[q] * wq * J

            # Convection term: r * S * (phi'_j) * (phi_i)
            # Note: we sum for each pair (i,j); here we use the convention: test function index i,
            # trial function index j.
            A_local[0,0] += r * S_val * dphi0[q] * phi0[q] * wq * J
            A_local[0,1] += r * S_val * dphi1[q] * phi0[q] * wq * J
            A_local[1,0] += r * S_val * dphi0[q] * phi1[q] * wq * J
            A_local[1,1] += r * S_val * dphi1[q] * phi1[q] * wq * J

            # Reaction term: -r * phi_i * phi_j (added to operator)
            A_local[0,0] += -r * phi0[q] * phi0[q] * wq * J
            A_local[0,1] += -r * phi0[q] * phi1[q] * wq * J
            A_local[1,0] += -r * phi1[q] * phi0[q] * wq * J
            A_local[1,1] += -r * phi1[q] * phi1[q] * wq * J

        # Assemble into global matrices (add contributions to nodes i and i+1)
        indices = [i, i+1]
        for ii in range(2):
            for jj in range(2):
                M_mat[indices[ii], indices[jj]] += M_local[ii, jj]
                A_mat[indices[ii], indices[jj]] += A_local[ii, jj]

    A1 = M_mat + (dt/2)*A_mat
    B1 = M_mat - (dt/2)*A_mat

    # Terminal condition at t = T: V(S,T) = max(S-K,0)
    V = np.maximum(S_nodes - K, 0)

    # Time stepping: n = N-1 down to 0, where t_n = n*dt.
    for n in range(N-1, -1, -1):
        t = n * dt
        # Right-hand side for current time step:
        RHS = B1.dot(V)
        # Impose Dirichlet BC in the solution V^n:
        # At S=0: V=0.
        RHS[0] = 0
        # At S=S_max: V = S_max - K*exp(-r*t)
        RHS[-1] = S_max - K * np.exp(-r * t)

        # Modify the system matrix A1 for boundary nodes.
        A_mod = A1.copy()
        # For node 0:
        A_mod[0, :] = 0
        A_mod[0, 0] = 1
        # For node M:
        A_mod[-1, :] = 0
        A_mod[-1, -1] = 1

        # Solve for V at current time step.
        V = solve(A_mod, RHS)

    # Interpolate to get the price at S = S0.
    price = np.interp(S0, S_nodes, V)
```

```
    return price
```

6) DeepRitz implementation

Here we adopt a closely related residual minimization strategy (akin to a PINN) for the stationary ODE that the European call price approximately satisfies. For a European call, one may consider the following boundary value problem:

$$\frac{1}{2}\sigma^2 S^2 V''(S) + r\,S\,V'(S) - r\,V(S) = 0, \qquad S \in (0, S_{\max}), \tag{29}$$

with boundary conditions

$$V(0) = 0, \quad V(S_{\max}) = S_{\max} - K.$$

A deep Ritz (or residual minimization) approach constructs a trial solution that automatically satisfies the boundary conditions and minimizes the mean squared residual of the differential operator at collocation points. (This is very similar in spirit to physics-informed neural networks but with a "Ritz" trial function that enforces boundary conditions exactly.) In this implementation we use `PyTorch` to define a feedforward network $N(S;\theta)$ and define a trial solution

$$\tilde{V}(S) = A(S) + B(S)N(S;\theta),$$

where $A(S) = \frac{S}{S_{\max}}(S_{\max} - K)$ satisfies $A(0) = 0$ and $A(S_{\max}) = S_{\max} - K$, and $B(S) = S(S_{\max} - S)$ vanishes at the boundaries.

We then compute the residual of the ODE

$$R(S) = \tfrac{1}{2}\sigma^2 S^2 \tilde{V}''(S) + rS\tilde{V}'(S) - r\tilde{V}(S), \tag{30}$$

and minimize the mean square error $L(\theta) = \frac{1}{N}\sum R(S_i)^2$ over collocation points $S_i \in (0, S_{\max})$. After training, we evaluate $\tilde{V}(S_0)$ as the approximate option price at S_0.

We define a simple feedforward neural network (with two hidden layers of 50 neurons each and Tanh activations) that maps S to a scalar output. This network is our free function $N(S;\theta)$. The trial solution

$$\tilde{V}(S) = \underbrace{\frac{S}{S_{\max}}(S_{\max} - K)}_{A(S)} + \underbrace{S(S_{\max} - S)}_{B(S)}\,N(S;\theta) \tag{31}$$

is chosen so that it automatically satisfies the boundary conditions:

- $\tilde{V}(0) = 0$ (since $A(0) = 0$ and $B(0) = 0$), and
- $\tilde{V}(S_{\max}) = S_{\max} - K$ (since $A(S_{\max}) = S_{\max} - K$ and $B(S_{\max}) = 0$).

We sample N_{colloc} collocation points in the domain $(0, S_{\max})$. For each collocation point we compute the first and second derivatives of $\tilde{V}(S)$ via automatic differentiation. The residual of the PDE

$$R(S) = 0.5\,\sigma^2\,S^2\,\tilde{V}''(S) + r\,S\,\tilde{V}'(S) - r\,\tilde{V}(S) \tag{32}$$

is squared and averaged to form the loss. We use the Adam optimizer over a fixed number of epochs (default 5000) to minimize the loss. After training, the network's trial solution is evaluated at $S = S_0$ and returned as the approximate option price.

```
def deep_ritz_price(S0, K, r, sigma, T, kwargs):
    device = torch.device("cpu")
    S_max = kwargs.get("S_max", 300.0)
    num_collocation = kwargs.get("num_collocation"
    , 1000)
    num_epochs = kwargs.get("num_epochs", 5000)
    lr = kwargs.get("lr", 1e-3)
    # Define the neural network N(S; theta)
    class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.net = nn.Sequential(
                nn.Linear(1, 50),
                nn.Tanh(),
                nn.Linear(50, 50),
                nn.Tanh(),
                nn.Linear(50, 1)
            )
        def forward(self, x):
            return self.net(x)
```

7) DeepONet implementation

DeepONet Architecture: The BranchNet processes a discretized representation of the terminal payoff function $g(S) = \max(S-K, 0)$ (sampled on a grid of m points). The TrunkNet processes the query coordinate S (a scalar). The network output is the inner product (dot product) of the branch and trunk outputs, which serves as the predicted option price at the query point.

For each training sample, a strike K is drawn uniformly from [80, 120]. The branch input is formed by evaluating the payoff function on a fixed grid $S \in [0, S_{\max}]$. The target function is the Black–Scholes solution $V(S)$ computed on the same grid. A random query point is sampled from the grid, and its corresponding target value is used as the training target. After training, the network is evaluated at the query point S_0 for the given strike K. The branch input is recomputed using K, and the trunk net is fed S_0.

```
class BranchNet(nn.Module):
    def __init__(self, input_size, output_size,
    hidden_dim=50):
        super(BranchNet, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_size)
        )
    def forward(self, x):
        return self.net(x)

class TrunkNet(nn.Module):
    def __init__(self, output_size, hidden_dim=50)
    :
        super(TrunkNet, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(1, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_size)
        )
    def forward(self, x):
```

```
        return self.net(x)

class DeepONet(nn.Module):
    def __init__(self, branch_input_size, output_size, hidden_dim=50):
        super(DeepONet, self).__init__()
        self.branch_net = BranchNet(branch_input_size, output_size, hidden_dim)
        self.trunk_net = TrunkNet(output_size, hidden_dim)
    def forward(self, branch_input, trunk_input):
        # branch_input: shape (batch_size, branch_input_size)
        # trunk_input: shape (batch_size, 1)
        branch_out = self.branch_net(branch_input)  # shape (batch_size, p)
        trunk_out = self.trunk_net(trunk_input)  # shape (batch_size, p)
        # Dot product (inner product) along the latent dimension:
        output = torch.sum(branch_out * trunk_out, dim=1, keepdim=True)
        return output

# ------------------------------
# Training Routine for DeepONet
# ------------------------------
def train_deeponet(num_samples=1000, epochs=5000, lr=1e-3, m=100, p=50, S_max=300, r=0.05, sigma=0.2, T=1.0):
    device = torch.device("cpu")
    S_grid = np.linspace(0, S_max, m)
    # Generate training data by sampling strikes K in [80, 120]
    Ks = np.random.uniform(80, 120, num_samples)
    branch_inputs = []    # each entry is g(S) = max(S-K, 0) on S_grid, shape (m,)
    target_functions = [] # corresponding target V(S) on S_grid (from Black-Scholes)
    for K_val in Ks:
        g = np.maximum(S_grid - K_val, 0)  # payoff function
        branch_inputs.append(g)
        V = black_scholes_price(S_grid, K_val, r, sigma, T)
        target_functions.append(V)
    branch_inputs = np.array(branch_inputs)  # shape (num_samples, m)
    target_functions = np.array(target_functions)  # shape (num_samples, m)

    query_indices = np.random.randint(0, m, size=num_samples)
    trunk_inputs = S_grid[query_indices].reshape(-1, 1)  # shape (num_samples, 1)
    targets = np.array([target_functions[i, query_indices[i]] for i in range(num_samples)]).reshape(-1, 1)

    branch_inputs_tensor = torch.tensor(branch_inputs, dtype=torch.float32, device=device)
    trunk_inputs_tensor = torch.tensor(trunk_inputs, dtype=torch.float32, device=device)
    targets_tensor = torch.tensor(targets, dtype=torch.float32, device=device)

    # Initialize DeepONet model
    model = DeepONet(branch_input_size=m, output_size=p, hidden_dim=50).to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()

    # Training loop
    for epoch in range(epochs):
        optimizer.zero_grad()
        outputs = model(branch_inputs_tensor, trunk_inputs_tensor)  # shape (num_samples, 1)
        loss = loss_fn(outputs, targets_tensor)
        loss.backward()
        optimizer.step()
        if epoch % 500 == 0:
            print(f"Epoch {epoch:5d}, Loss: {loss.item():.6f}")

    return model, S_grid

# ------------------------------
# DeepONet Price Prediction Function
# ------------------------------
def deeponet_price(S0, K, r, sigma, T, kwargs):
    num_samples = kwargs.get("num_samples", 1000)
    epochs = kwargs.get("epochs", 5000)
    lr = kwargs.get("lr", 1e-3)
    m = kwargs.get("m", 100)
    p = kwargs.get("p", 50)
    S_max = kwargs.get("S_max", 300)

    print("Training DeepONet...")
    model, S_grid = train_deeponet(num_samples=num_samples, epochs=epochs, lr=lr,
                                   m=m, p=p, S_max=S_max, r=r, sigma=sigma, T=T)

    # Build branch input for the given strike K: g(S)=max(S-K,0) on S_grid
    branch_input = np.maximum(S_grid - K, 0).reshape(1, -1)  # shape (1, m)
    branch_input_tensor = torch.tensor(branch_input, dtype=torch.float32)

    # Build trunk input for the query point S0
    trunk_input = np.array([[S0]], dtype=np.float32)  # shape (1, 1)
    trunk_input_tensor = torch.tensor(trunk_input, dtype=torch.float32)

    model.eval()
    with torch.no_grad():
        price_tensor = model(branch_input_tensor, trunk_input_tensor)
    price = price_tensor.item()
    return price
```

8) Fourier Neural Operator implementation

We construct `SpectralConv1d` that computes a 1D Fourier transform of the input, keeps only a fixed number of Fourier modes, multiplies them by learnable complex weights, and then returns to physical space via an inverse FFT. The FNO model first "lifts" (by `FNO1d`) the input function (augmented with its normalized spatial coordinate) to a higher-dimensional representation. It then applies several layers of spectral convolution (using Fourier transforms) combined with pointwise convolutions and residual connections. Finally, the output is projected back to a one-dimensional function.

The function `train_fno` generates training data by sampling strikes uniformly (here in [80,120]), computing the corresponding terminal payoff function $g(S) = \max(S - K, 0)$ and the target solution $V(S)$ (using the

analytical Black–Scholes formula) on a uniform grid over $[0, S_{\max}]$. The FNO is trained to minimize the mean-squared error between its output and the target. The function `fourier_neural_operator_price` trains the FNO (or loads a pre-trained model in a more advanced implementation) and then uses it to predict the solution function for a given strike K. The predicted function is then interpolated to obtain the price at $S = S_0$.

```
# ----------------------------
# Spectral Convolution for 1D
# ----------------------------
class SpectralConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, modes):
        """
        1D Fourier layer. It does FFT, multiplies some Fourier modes by learned weights, and returns via inverse FFT.
        """
        super(SpectralConv1d, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.modes = modes  # number of Fourier modes to keep
        self.scale = 1 / (in_channels * out_channels)
        # weights is a complex tensor: shape (in_channels, out_channels, modes)
        self.weights = nn.Parameter(self.scale * torch.rand(in_channels, out_channels, modes, dtype=torch.cfloat))

    def compl_mul1d(self, input, weights):
        # input: (batch, in_channels, n_ft), weights: (in_channels, out_channels, modes)
        # returns: (batch, out_channels, modes)
        return torch.einsum("bix, iox -> box", input, weights)

    def forward(self, x):
        """
        x: shape (batch, in_channels, n)
        """
        batchsize = x.shape[0]
        n = x.shape[-1]
        # Compute Fourier transform (rfft returns half-spectrum)
        x_ft = torch.fft.rfft(x)  # shape (batch, in_channels, n//2 + 1)
        # Allocate output in Fourier space, same shape as x_ft
        out_ft = torch.zeros(batchsize, self.out_channels, x_ft.shape[-1], device=x.device, dtype=torch.cfloat)
        # Multiply only the first self.modes frequencies
        out_ft[:, :, :self.modes] = self.compl_mul1d(x_ft[:, :, :self.modes], self.weights)
        # Return inverse FFT to get back to physical space
        x = torch.fft.irfft(out_ft, n=n)
        return x

# ----------------------------
# Fourier Neural Operator (FNO) for 1D
# ----------------------------
class FNO1d(nn.Module):
    def __init__(self, modes, width, layers=4):
        """
        FNO that maps an input function defined on a 1D grid to an output function.
        The input is augmented with the spatial coordinate.
        - modes: number of Fourier modes to keep in each spectral layer.
        - width: number of channels (feature dimension) in the lifted space.
        - layers: number of Fourier layers.
        """
        super(FNO1d, self).__init__()
        self.modes = modes
        self.width = width
        self.layers = layers
        # Lift the input (which has 2 channels: function value and coordinate) to 'width' channels.
        self.fc0 = nn.Linear(2, width)
        self.spectral_layers = nn.ModuleList()
        self.w_layers = nn.ModuleList()
        for _ in range(layers):
            self.spectral_layers.append(SpectralConv1d(width, width, modes))
            self.w_layers.append(nn.Conv1d(width, width, 1))
        self.fc1 = nn.Linear(width, 128)
        self.fc2 = nn.Linear(128, 1)
        self.activation = nn.ReLU()

    def forward(self, x):
        batchsize, n = x.shape
        # Create a grid of coordinates normalized to [0,1]
        grid = torch.linspace(0, 1, n, device=x.device).unsqueeze(0).repeat(batchsize, 1)  # shape (batch, n)
        # Concatenate input function and grid as features.
        x = x.unsqueeze(-1)  # shape (batch, n, 1)
        inp = torch.cat([x, grid.unsqueeze(-1)], dim=-1)  # shape (batch, n, 2)
        # Lift to higher dimension
        x = self.fc0(inp)  # shape (batch, n, width)
        x = x.permute(0, 2, 1)  # shape (batch, width, n)
        # Apply Fourier layers with residual connections.
        for i in range(self.layers):
            x1 = self.spectral_layers[i](x)
            x2 = self.w_layers[i](x)
            x = x1 + x2
            x = self.activation(x)
        x = x.permute(0, 2, 1)  # shape (batch, n, width)
        x = self.fc1(x)         # shape (batch, n, 128)
        x = self.activation(x)
        x = self.fc2(x)         # shape (batch, n, 1)
        x = x.squeeze(-1)       # shape (batch, n)
        return x

# ----------------------------
# Training Routine for FNO
# ----------------------------
def train_fno(num_samples=1000, epochs=5000, lr=1e-3, m=100, modes=16, width=64, layers=4, S_max=300, r=0.05, sigma=0.2, T=1.0):
    device = torch.device("cpu")
    # Create a fixed spatial grid on [0, S_max]
    S_grid = np.linspace(0, S_max, m)

    # Generate training data by sampling strikes in [80, 120]
```

```
    branch_inputs = []   (num_samples, m)
    targets = []
    strikes = np.random.uniform(80, 120,
    num_samples)
    for K_val in strikes:
        g = np.maximum(S_grid - K_val, 0)  #
    payoff function
        branch_inputs.append(g)
        V = black_scholes_price(S_grid, K_val, r,
    sigma, T)
        targets.append(V)
    branch_inputs = np.array(branch_inputs)  #
    shape (num_samples, m)
    targets = np.array(targets)              #
    shape (num_samples, m)

    # Convert training data to torch tensors
    X = torch.tensor(branch_inputs, dtype=torch.
    float32, device=device)
    Y = torch.tensor(targets, dtype=torch.float32,
     device=device)

    model = FNO1d(modes, width, layers).to(device)
    optimizer = optim.Adam(model.parameters(), lr=
    lr)
    loss_fn = nn.MSELoss()

    for epoch in range(epochs):
        optimizer.zero_grad()
        pred = model(X)  # shape (num_samples, m)
        loss = loss_fn(pred, Y)
        loss.backward()
        optimizer.step()
        if epoch % 500 == 0:
            print(f"Epoch {epoch:5d}, Loss: {loss.
    item():.6f}")
    return model, S_grid

# ---------------------------
# Fourier Neural Operator Price Prediction
    Function
# ---------------------------
def fourier_neural_operator_price(S0, K, r, sigma,
     T, kwargs):
    num_samples = kwargs.get("num_samples", 1000)
    epochs = kwargs.get("epochs", 5000)
    lr = kwargs.get("lr", 1e-3)
    m = kwargs.get("m", 100)
    modes = kwargs.get("modes", 16)
    width = kwargs.get("width", 64)
    layers = kwargs.get("layers", 4)
    S_max = kwargs.get("S_max", 300)

    print("Training Fourier Neural Operator...")
    model, S_grid = train_fno(num_samples=
    num_samples, epochs=epochs, lr=lr,
                              m=m, modes=modes,
    width=width, layers=layers,
                              S_max=S_max, r=r,
    sigma=sigma, T=T)
    # For the given strike K, compute the terminal
     payoff function on S_grid
    g = np.maximum(S_grid - K, 0).reshape(1, -1)
    # shape (1, m)
    X_test = torch.tensor(g, dtype=torch.float32)
    model.eval()
    with torch.no_grad():
        pred = model(X_test)  # shape (1, m)
    pred = pred.squeeze(0).cpu().numpy()  #
    predicted solution function V(S) at t=0
    # Interpolate to get the price at S0
    f_interp = interp1d(S_grid, pred, kind='linear
    ', fill_value="extrapolate")
    price = f_interp(S0)
    return float(price)
```