Modelling Options Chains Using Physics-Informed Neural Networks

CID: 02150969

Project Specification

Our client is Optidel, a large multinational options market maker. As a market maker, Optidel is interested in devising a novel method for determining the fair value of an option—with this fair value, they can set a spread around the midpoint price and show a bid–offer on the open market.

Optidel is aware of existing pricing techniques such as the Black–Scholes closed-form solution, finite differencing methods and Monte Carlo simulations. However, interested in leveraging increasingly readily-available computing power, Optidel wishes to train a neural network to discover fair midpoint values of option prices with greater speed and computational-efficiency.

We suggest a physics-informed neural network (PINN), which leverages the known relationship between option pricing inputs to the fair market value of the option. Specifically, we train the model using the Black–Scholes partial differential equation as an additional loss term; in theory, the resulting network should be better able to predict out-of-sample option prices.

To this end, we construct a PINN and test its predictive performance using out-of-sample cross validation against a standard multi-layer perceptron neural network. Through this methodology, we quantify the benefit of using the PINN for Optidel's options market making strategy.

Contents

Chapter 1

Introduction

The inherent leverage and convex payoff profile of options make them an incredibly attractive asset class for investors and speculators alike. The classical approach to pricing such products heavily revolves around parametric models such as the Black–Scholes model (Black & Scholes 1973).

With the advent of more powerful computers, there has been a push towards adopting non-parametric models such as machine learning models for options pricing—for example, Ivaşcu (2021) compares the use of several machine learning methods in pricing real-world options; Liu et al. (2019) compare different machine learning methods in predicting the implied volatility surface; Ke & Yang (2019) apply deep learning techniques on option pricing inputs and attempt to generate fair prices as observed on the market. These literature mainly leverage the non-linear optimisation capabilities of machine learning to model the non-linear options pricing surface.

Raissi et al. (2019) suggest a third approach, combining the parametric (Black–Scholes model) and model-agnostic machine learning techniques to form a physics-informed neural network (PINN). This paper applies the methodologies of Raissi et al. (2019) within the options pricing space, and demonstrates the advantages of using PINNs as a novel approach to options pricing, and especially in the pricing of more complex exotics where the pricing surface exhibits strong curvature.

Chapter 2

Theory

2.1 The Black–Scholes Equation

The value of an option V on an underlying S can be modelled with the Black–Scholes equation (Black & Scholes 1973), a linear homogeneous second-order partial derivative equation (PDE) with the following form (the full derivation is given in Appendix A):

$$\frac{1}{2}\sigma^2 S^2 V_{SS} + rSV_S - V_\tau - rV = 0 \tag{2.1}$$

where S is the spot price of the underlying, V is the value of the option, τ is the time to expiry,[1] r is the risk-free rate of return,[2] and σ is the implied volatility. The subscript terms V_S, V_{SS} and V_τ refer to the respective partial derivatives of V. Equation 2.1 must hold[3] for every strike and expiry along the entire options chain. Depending on the type of option to be priced, the boundary and initial conditions can be set appropriately and the PDE can be solved to find the fair value of the option. Within this framework, a closed-form solution of the Black–Scholes model exists for the pricing of European-style options.

However, for the pricing of more exotic options, no such analytical approach exists. The industry standard approach is thus to use a parametric model that continues to incorporate the Black–Scholes PDE. These models are then solved either using the method of finite differencing (Schwartz 1977) or with Monte Carlo simulation (Boyle 1977). From a computational standpoint, these parametric models are slow to converge but offer relatively high accuracy. Another possibility is to use a non-parametric method such as a neural network; however, the traditional neural network is black-box in nature and is not able to leverage the *a priori* known relational mappings between the option pricing inputs and the outputs. Raissi et al. (2019) suggest constructing a hybrid model that is able to incorporate the benefits of both approaches.

2.2 The Standard Approach to Neural Networks

A traditional feedforward neural network (NN) can be seen as a universal function approximator f mapping a set of predictor variables X (the feature set) to an outcome Y. In supervised

[1] Note, here that we are considering the PDE from the point of view of time to expiry τ rather than time t which is the usual approach; conveniently, this allows us to define an option using one less variable, as $\tau = T - t$.

[2] For simplicity, we are assuming the case where there are no dividends, hence we take $\delta = 0$.

[3] Given that the Black–Scholes–Model assumptions, given in section A.1, hold.

learning, the NN is trained by the varying of weight and bias terms on the hidden layers, such that a given loss term is minimised. An example of a typical loss term used is the root-mean-squared error (RMSE):

$$Y_{pred} = f(X),\ f : \arg\min_{f} \sqrt{\frac{1}{N}\sum \left(f(X) - Y_{true}\right)^2}.$$

The classical approach is excellent in modelling relationships for which the underlying process is not *a priori* known. However, the black-box nature of classical NN structures means that it is difficult to develop any understanding of the relationship between input and output variables even after successful training (Psichogios & Ungar 1992).

2.3 Physics-Informed Neural Networks

Psichogios & Ungar (1992) first introduced the concept of 'hybrid NNs', which leverage known relational mappings between inputs and outputs to train quantitatively more powerful NNs compared to a completely naïve approach. They achieve this by creating an NN structure whereby the NN is first used to generate a set of parameters; these parameters are subsequently used to predict the final outcome after parsing them through closed-form equations governing the relationship between the parameters and the output (Psichogios & Ungar 1992). Intuitively, this reduces the solution space of the NN during the optimisation phase, and results in quantitatively better fit in both interpolation and extrapolation (Psichogios & Ungar 1992).

However, one flaw of this approach is that a closed-form solution for the output must either exist and be known *a priori* or be reasonably approximated (Psichogios & Ungar 1992, Raissi et al. 2019). Raissi et al. (2019) hence improved on this methodology, introducing physics-informed neural networks (PINNs) which only require *a priori* knowledge of the relational PDE governing the inputs and outputs; this is particularly useful when the PDE is known but no analytical solution to the PDE exists. The PINN is enabled through the development of auto-differentiation methods in NN libraries, which allow one to easily obtain the N-order partial derivative terms of the output on each of the inputs.

Within the context of option pricing, it is possible to encode the Black–Scholes PDE (Equation 2.1) into the loss function of the NN during the training process—in so doing, we create a PINN which is trained with knowledge of the underlying Black–Scholes equation; this PINN should in theory be able to achieve a significant improvement in out-of-sample convergence compared to a naïve NN model.

Chapter 3

Methodology

3.1 Implementation of the Physics-Informed Neural Network

The construction of the PINN follows the approach introduced in Raissi et al. (2019), and is implemented using the TensorFlow library (Abadi et al. 2015). Ultimately, the PINN is designed to minimise a custom loss function consisting two terms: the 'standard' RMSE, as well as the deviation of the model from the Black–Scholes model:

$$\text{loss} = \text{RMSE} + \lambda \cdot \text{RMSE}_{BSM}, \tag{3.1}$$

where:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum \left(f(X) - Y_{pred}\right)^2}, \tag{3.2}$$

and:

$$\text{RMSE}_{BSM} = \sqrt{\frac{1}{N}\sum \left(\frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} + rS\frac{\partial f}{\partial S} - \frac{\partial f}{\partial \tau} - rf\right)^2}. \tag{3.3}$$

RMSE measures the raw difference between the model predicted value of the option against the true value, given a set of inputs. RMSE_{BSM} measures the deviation of the model from the Black–Scholes PDE, which we know should hold in theory. Ultimately, the existence of the RMSE_{BSM} term in the loss function turns the PINN into a constrained optimisation which is forced to be fit to the Black–Scholes PDE.

λ is a scalar multiple on the RMSE_{BSM} loss term, which adjusts the relative weights of the Black–Scholes PDE RMSE_{BSM}. For this project, we have set $\lambda = 1$, but this hyperparameter can be changed depending on one's expectation of the relative importance between both terms.

The NN that underlies the PINN is built as a 3-layer deep NN with 400 neurons on each layer, with dropout layers in between each dense layer; the purpose of these dropout layers are to act as a computationally-cheap way of reducing overfit (Srivastava et al. 2014). The dense layers are activated using hyperbolic tangent activation functions to model the smooth curvature of the option price surface; these dense layers are initialised using the Glorot normal initialiser proposed by Glorot & Bengio (2010) as an attempt to avoid the vanishing gradient problem. For training, the PINN uses the Adam algorithm (Kingma & Ba 2014) for the stochastic gradient descent, and

has callbacks to reduce the Adam learning rate when loss reduction plateaus, as well as to stop training when the loss in the validation set stops decreasing in order to avoid overfit.

As a control, we construct a naïve NN with an identical underlying architecture (to distinguish this from the PINN, we subsequently refer to this model as the multi-layer perceptron, or MLP model), but trained to minimise only the RMSE.[4] The code implementing both the PINN and the naïve MLP model is given in Appendix B.

To compare the performance of both models, we first generate a set of true options prices using existing classical approaches. We divide this into an in-sample training set, with which we train both the PINN and the MLP. We then cross-validate the converged models by comparing their accuracy when pricing options from unseen out-of-sample inputs. The scope of this project covers both the vanilla European call and the European up-and-out call.

3.2 Generating 'True' Option Prices

As we are interested in comparing the performance of the PINN against the naïve MLP, we need to generate sample options prices which obey the Black–Scholes PDE. This is done through two approaches, for two different option types.

3.2.1 Pricing the Vanilla European Call

In the first approach, we are interested in pricing a vanilla European call. The vanilla European call has payoff function:

$$\text{payoff} = (K - S_T,\, 0)^+,$$

where K is the strike price and S_T is the spot price of the underlying at expiry. Then, following the assumptions and methodology outlined in Black & Scholes (1973), we can find the 'true' prices of European-style calls on an example underlying using the closed-form solution. We start by setting the boundary conditions at $S = 0$:

$$C(0,\, \tau) = 0 \quad \forall \tau;$$

the value of C as $S \to \infty$:

$$C(S,\, \tau) \sim S - Ke^{-r\tau} \text{ for } S \to \infty;$$

and the terminal condition of the call as $\tau \to 0$:

$$C(S,\, 0) = (S - K,\, 0)^+.$$

Given these boundary conditions, Black & Scholes (1973) demonstrate that the closed-form solution to pricing a European-style call option is:

$$C(S,\, \tau) = \Phi(d_1)S - \Phi(d_2)Ke^{-r\tau},$$

[4] Technically, this is implemented as a minimisation of the built in mean-squared-error (MSE) loss term in TensorFlow; however the square-root function is monotonic so the minimisation is equivalent.

where:

$$d_1 = \frac{1}{\sigma\sqrt{\tau}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)\tau\right],$$

and:

$$d_2 = d_1 - \sigma\sqrt{\tau},$$

with $\Phi(\cdot)$ representing the standard normal cumulative distribution function.

We generate option prices over the range of strikes, expiries, underlying spot prices and implied volatility measures outlined in Table 3.1; for convenience, we assume the annual risk-free rate of interest is kept constant as $r = 0.02$. This set of data forms our first set of 'true' data with which we can train the PINN. The sample code used to generate the prices of the vanilla European call are given in section C.1.

Table 3.1: *Parameters used to generate 'true' prices of vanilla European call*

Variable	**Range of variables**
Strike price K	55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105, 110, 115, 120, 125, 130, 135, 140, 145, 150
Spot price S	95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105
Implied (periodic) volatility σ	1.0%, 1.2%, 1.3%, 1.4%, 1.5%, 1.6%, 1.8%, 2.0%
Expiry τ, in periods	3, 5, 10, 15, 21, 42, 63, 126, 252

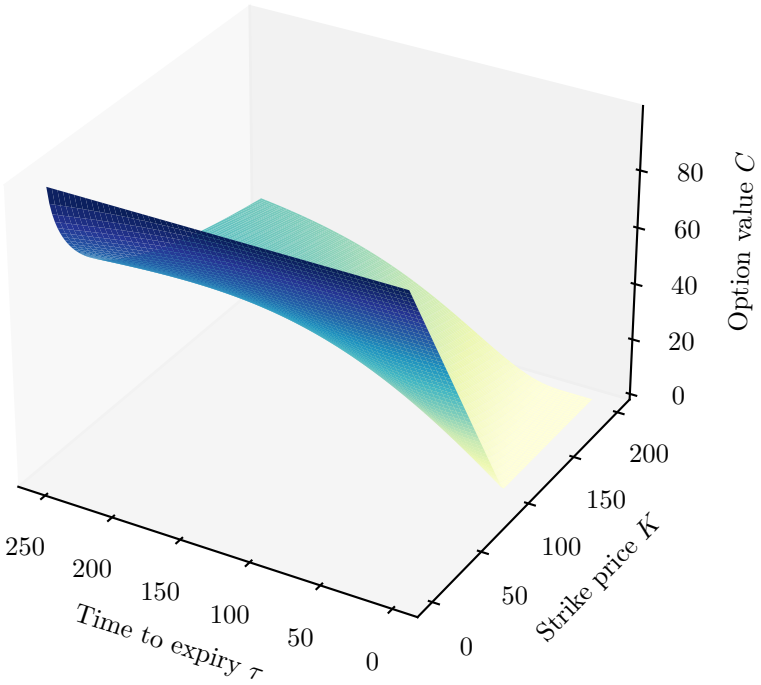

Figure 3.1: *Pricing surface of an up-and-out European call varying time to expiry τ and strike price K; we see that the price surface matches prior work by Wilmott (2006).*

We can check the validity of the closed-form approach by plotting a sample of option prices given one-period implied volatility $\sigma = 0.01$, annualised risk-free interest rate $r = 0.02$, spot price $S = 100$, over a range of strikes K and expiries τ. The resultant price surface is shown in Figure 3.1. We can see that the value of the option converges to the terminal payoff at expiry as $\tau \to 0$; then, the prices at every other point on the chain are a diffusion of this terminal condition with a diffusivity rate determined by implied volatility σ.

3.2.2 Pricing the Up-and-Out European Call

The closed-form solution is insufficient for pricing exotic options with complex path dependent payoffs, for example the European up-and-out barrier call. The up-and-out call has the following payoff function:

$$\text{payout} = \begin{cases} (K - S_T,\, 0)^+ & \text{if } S_t < B \,\forall\, t \in T \\ 0 & \text{otherwise} \end{cases}$$

where B is the barrier price, S_T is the terminal value of the underlying, and S_t is the underlying price achieved at any point in time from when the call is written until the point of expiry T. In the up-and-out call, if the underlying price S_t hits or exceeds the barrier B at any point during the life of the call, the call becomes worthless. The benefit to buying an up-and-out call as opposed to a vanilla call is that it provides similar upside exposure to a speculator but is relatively cheaper, as the upside is *ex ante* limited so from the point of view of the market maker, hedging is cheaper.

The up-and-out call has no closed-form solution, hence we adopt a finite differencing approach to price this instrument. Specifically, we use the explicit forward differencing approach (Schwartz 1977, Hull 2018), using a forward differencing approach in time and a central differencing in space. With this set up, we can find the price of the up-and-out European call, while varying the strike K, spot S, implied volatility σ and expiry τ; the range of variables used to generate these prices are given in Table 3.2. We fix the level of the barrier $B = 140$.

Table 3.2: *Parameters used to generate 'true' values of up-and-out European call*

Variable	**Range of variables**
Strike price K	90, 95, 100, 105, 110, 115, 120
Spot price S	$(0, 140)$
Implied (periodic) volatility σ	1.00%, 1.25%, 1.50%, 1.75%, 1.85%, 2.00%
Expiry τ, in periods	3, 5, 10, 15, 21, 42, 63, 126, 252

The sample code used to generate the true option prices for the up-and-out European barrier call is given in section C.2. As a sanity check, we then plot the option price surface for an example call with the following parameters: $B = 140$, $K = 110$, $\sigma = 0.02$, over a range of times to expiry τ and underlying spot prices S. The resultant price surface is shown in Figure 3.2.

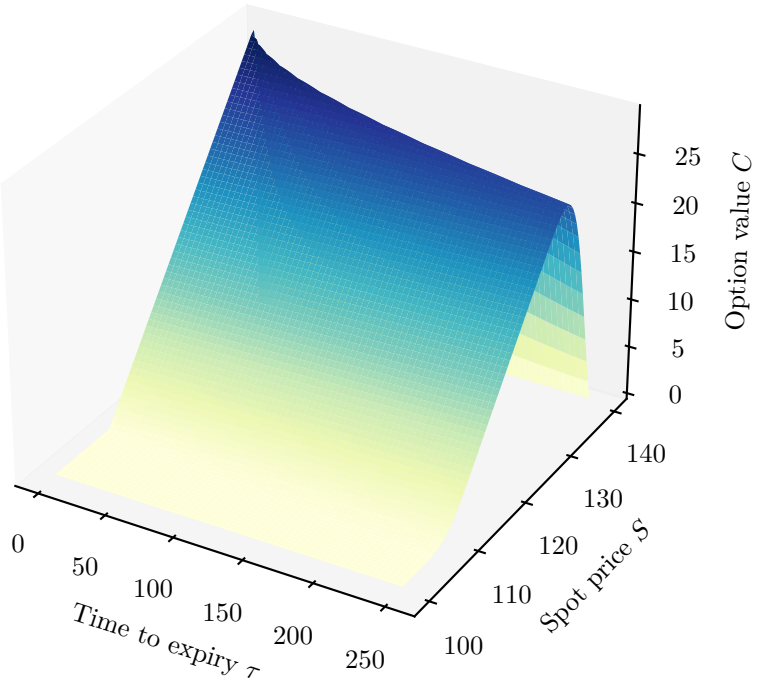

Figure 3.2: *Pricing surface of an up-and-out European call varying time to expiry τ and spot price S; we see that this surface matches the solution given in Wilmott (2006).*

Chapter 4

Results and Discussion

4.1 Training on Vanilla European Call Prices

According to Raissi et al. (2019), one benefit of the PINN is its ability to accurately extrapolate the relationship in unseen data. To test this hypothesis, we first train both the PINN and the naïve MLP on a subset of the true option prices generated in section 3.2; specifically, we train using all options at the implied volatility levels $\sigma = 1.2\%$ and $\sigma = 1.4\%$. We then compare the relative performance of both methods by comparing the predictive performance of both models in out-of-sample testing. This is done by using both the PINN and the MLP to predict the options chain (for all strikes K and expiries τ) given $S = 100$ and $\sigma = 1.3\%$.

Table 4.1: *Out-of-sample median and standard deviations of RMSE and MAE when pricing vanilla European call using PINN versus naïve MLP*

Model	**Root-mean-squared error**	**Mean absolute error**
PINN	0.510 ± 0.338	0.392 ± 0.246
MLP	0.679 ± 0.820	0.505 ± 0.607

Then, we use both models to predict option prices given out-of-sample testing inputs. We compare these predictions against the 'true' prices computed using the closed-form approach. We quantify the difference in the error terms by looking at the both the RMSE—given in Equation 3.2—and the mean absolute error (MAE) between the true and predicted option prices for the given inputs:

$$\text{MAE} = \frac{1}{N}\left|Y_{pred} - Y_{true}\right|$$

As the training of the NN underlying the PINN and the MLP is stochastic in nature, we repeat this training and testing process 100 times to find the median[5] and standard deviations of RMSE and MAE using either model. The result of this approach is shown in Table 4.1. It is clear to see from this comparison that in the case of pricing the vanilla European call, the PINN performance is superior to that of the naïve MLP, according to both metrics of RMSE and MAE.

[5]Here, we chose to use the median of the observed errors (as opposed to the mean) as it is more indicative of the performance of either model. This is because have an *a priori* expectation that the distribution of errors will not follow a normal distribution, as the minimum possible error is 0, so the tail of the distribution of errors is heavier on the upside.

4.2 Training on Up-and-Out Call Price

The initial results from testing on the vanilla European call are promising for the PINN; we can see a small but significant improvement in model performance when incorporating the Black–Scholes PDE as a custom loss term. However, we know that in reality, the PINN adds relatively less value to the pricing of the vanilla European call as this pricing can be obtained simply as a closed-form function of the inputs. Conversely, the pricing of a path dependent option such as the up-and-out call has to be done using either a finite differencing approach (Schwartz 1977) or a Monte Carlo simulation (Boyle 1977). However, these approaches are relatively inefficient in that they require a much longer time to converge. Fortunately, we know that the up-and-out call must still obey the Black–Scholes PDE, and this lends credence to using the PINN approach.

It is particularly more interesting to observe the effect of using the PINN in pricing the up-and-out call due to the strong curvature of the option pricing surface relative to the vanilla European call—in particular this can be seen by comparing Figure 3.1 against Figure 3.2.

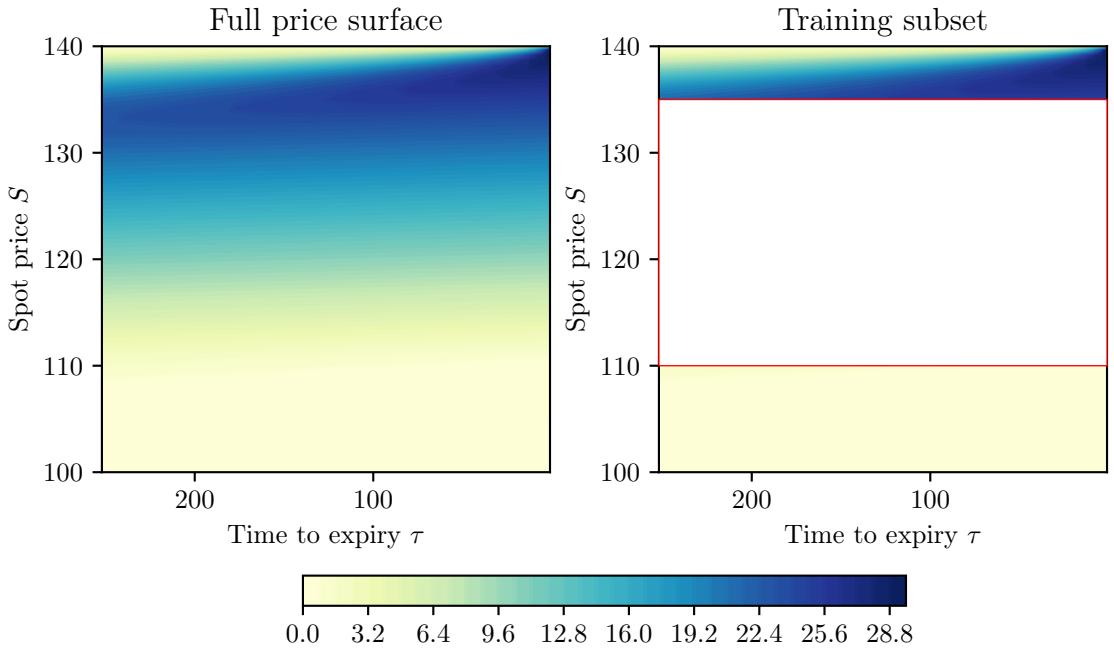

Figure 4.1: *Comparison between full training set (left) versus restricted subset (right) seen by the PINN and MLP models in training; the price surface shown is with* $\sigma = 2.00\%$ *and* $K = 110$. *Option value at each point in time represented by colour.*

In a similar manner to pricing the vanilla European call, the PINN and MLP are both trained on a subset of the true prices obtained earlier using the finite differencing method. Specifically, the training data is restricted to implied volatility levels of $\sigma \in (1.00\%, 1.25\%, 1.50\%, 1.75\%, 2.00\%)$ and the range of spot prices $S \leq 110$ and $S \geq 135$. A visualisation of the masking is shown in Figure 4.1—it can be seen that the selection of training parameters precludes a region of high curvature. Then, we attempt to generate the options chain at the out-of-sample implied volatility level $\sigma = 1.85\%$ and given a spot price of $S = 115$, using both the PINN and the MLP. The RMSE and MAE between the true and predicted values for both models are calculated. This is repeated over 100 iterations, and the median and standard deviation of the error terms are shown in Table 4.2.

Table 4.2: *Out-of-sample median and standard deviations of RMSE and MAE when pricing up-and-out European call using PINN versus naïve MLP*

Model	Root-mean-squared error	Mean absolute error
PINN	0.597 ± 0.087	0.498 ± 0.077
MLP	1.068 ± 0.234	0.608 ± 0.129

As with the vanilla European call, this methodology highlights the benefit of using the PINN to price the up-and-out call versus using the MLP. Again, we observe that the pricing error from using the PINN is greatly reduced compared to using the naïve MLP.

4.3 Analysis of Findings

Our out-of-sample testing methodology demonstrates the ability of the PINN to price both the vanilla call and the exotic up-and-out call. The realised benefit from the point of view of the market maker is they can quote tighter spreads as the uncertainty of the model-implied prices is much lower when using the PINN; this in turn allows the market maker to capture more flow and reduces the market maker's exposure to model risk.

In both the pricing of the vanilla European call and the up-and-out call, we observe that the PINN has a better out-of-sample predictive accuracy compared to the naïve MLP model. The extent of the improvement in using the PINN is more significant in the case of the up-and-out call. This result is in line with our expectations. The result of using the Black–Scholes PDE as an additional loss term in the convergence of the PINN effectively informs the underlying network of the slope of the true pricing surface given the inputs. In the case of pricing the up-and-out call, we further increase the difficulty of the prediction by masking a big part of the pricing surface. Both these constraints cause the ultimate out-of-sample error terms to be much larger for both the MLP and the PINN; however, as the PINN is relatively more aware of the curvature of the surface, it is better able to arrive at the correct value for the up-and-out call.

Our current implementation has ultimately achieved a respectable pricing accuracy. However, it is possible to further improve this accuracy through more careful tuning of the model's hyperparameters. The current implementation is a simple 3-layer deep NN, however a deeper NN or one with more hidden neurons would better be able to take into account the steeper non-linear effects around each input term. Ultimately, this work is left as an exercise to the reader.

Chapter 5

Conclusion

5.1 Summary

We find that the PINN trained with a custom loss function accounting for the Black–Scholes PDE is better able to predict option prices in out-of-sample testing as compared to the naïve MLP which only minimises model RMSE. The implication of this is that the PINN is a strong contender for the pricing of exotics, where closed-form solutions to true prices are not known.

When using the PINN, the computationally-intensive portion of the work (the training) is front-loaded, and therefore the pricing itself can be carried out in $\mathcal{O}(1)$ time.[6] This is relevant to market-making, as one wants to be able to very quickly react to changes in the option pricing inputs—the PINN could be a first-resort approximation of option value in the event of an unexpected change in the modelling inputs, before slower models kick in and take over the pricing.

Ultimately, we have successfully implemented the PINN and shown its efficacy in the options pricing space.

5.2 Recommendations for Further Work

The present work considers the simple vanilla European call and the European up-and-out call. Further study could look into the use of a PINN to predict other exotic path dependent option types, such as the Asian, Bermudan or lookback options. As these exotics come with their own pricing challenges, it would be instructive to see if the PINN remains able to accurately price these complex derivatives.

While the PINN is excellent at pricing the idealised options we have generated from the Black–Scholes equations, it would be interesting to see if it holds up when trained and tested on real-world inputs, which do not necessarily obey this relationship (Ivaşcu 2021). In theory, it should be possible to modify the existing Black–Scholes PDE within the PINN to better reflect the real world, say, by incorporating non-constant implied volatility.

In the methodology proposed by Raissi et al. (2019), a second-stage optimisation using

[6] Technically, the speed of the prediction is a function of the size of the NN, in terms of both number of hidden layers and number of hidden neurons per layer; however, assuming the same level of complexity of the model architecture, the fitting time is constant. This is as opposed to both the numerical finite differencing approach and the Monte Carlo simulation, both of which have a convergence accuracy positively correlated to the amount of time required to converge: for the Monte Carlo simulation, more simulations will lead to more confident price; for the finite differencing approach, a finer grid will lead to greater accuracy of convergence.

the L-BFGS algorithm (Xiao et al. 2008) is used for further training after training using the Adam optimiser finishes. The L-BFGS has the advantage of global minimisation, with the disadvantage being that it is slower to converge; as a result, it makes more sense to use it as a second-stage optimiser once Adam has been minimised. We have not implemented the L-BFGS minimiser in this report as this would make the comparison against the naïve MLP unfair; however, in actual deployment, it would be beneficial to additionally train the PINN using the L-BFGS optimisation algorithm.

Finally, Raissi et al. (2019) demonstrate that the PINN can be used to generate outputs even in the absence of training data. Intuitively, the Black–Scholes equation defines the slope at all points on the option pricing surface; given well-defined boundary conditions, hence, given the correct set of inputs, the PINN should be able to be trained to output prices even in an unsupervised manner. Initial testing within the scope of this project yielded interesting results—but further work here could lead to the development of an entirely new approach to option pricing.

(3,267 words)— I have exceeded the 3,000 word limit in part because I have listed quite a few possible recommendations; this section could probably be cut short, but I believe they are really worthwhile avenues for further research. The rest of the report has already been edited to have *de minimis* excess verbiage; owing to the rather technical nature of both options pricing and neural networks, it is my opinion that further trimming of the content and explanations will result in a paper which is much more confusing and unclear.

Bibliography

Abadi, M. et al. (2015), 'TensorFlow: Large-scale machine learning on heterogeneous systems'. Software available from tensorflow.org.
URL: *https://www.tensorflow.org/* 5

Black, F. & Scholes, M. (1973), 'The pricing of options and corporate liabilities', *Journal of Political Economy* **81**(3), 637–654.
URL: *http://www.jstor.org/stable/1831029* 2, 3, 6

Boyle, P. P. (1977), 'Options: A Monte Carlo approach', *Journal of Financial Economics* **4**(3), 323–338.
URL: *https://www.sciencedirect.com/science/article/pii/0304405X77900058* 3, 11

Capiński, M. J. & Zastawniak, T. (2012), *Numerical Methods in Finance with C++*, 1 edn, Cambridge University Press. 24

Glorot, X. & Bengio, Y. (2010), 'Understanding the difficulty of training deep feedforward neural networks', *Journal of Machine Learning Research - Proceedings Track* **9**, 249–256. 5

Hovhannisyan, V. (2022), 'Finite difference methods with Python', Imperial College Business School. Accessed: 2022-08-31. 24

Hull, J. C. (2018), *Options, futures, and other derivatives*, 9 edn, Pearson. 8, 18

Ivaşcu, C.-F. (2021), 'Option pricing using machine learning', *Expert Systems with Applications* **163**, 113799.
URL: *https://www.sciencedirect.com/science/article/pii/S0957417420306187* 2, 13

Ke, A. & Yang, A. (2019), 'Options pricing with deep learning'. Accessed: 2022-08-31.
URL: *https://cs230.stanford.edu/projects_fall_2019/reports/26260984.pdf* 2

Kingma, D. P. & Ba, J. (2014), 'Adam: A method for stochastic optimization'.
URL: *https://arxiv.org/abs/1412.6980* 5

Liu, S., Oosterlee, C. & Bohte, S. (2019), 'Pricing options and computing implied volatilities using neural networks', *Risks* **7**(1), 16.
URL: *https://doi.org/10.3390%2Frisks7010016* 2

Psichogios, D. C. & Ungar, L. H. (1992), 'A hybrid neural network-first principles approach to process modeling', *AIChE Journal* **38**(10), 1499–1511.
URL: *https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/aic.690381003* 4

Raissi, M., Perdikaris, P. & Karniadakis, G. (2019), 'Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations', *Journal of Computational Physics* **378**, 686–707.
URL: *https://www.sciencedirect.com/science/article/pii/S0021999118307125* 2, 3, 4, 5, 10, 13, 14

Schwartz, E. S. (1977), 'The valuation of warrants: Implementing a new approach', *Journal of Financial Economics* **4**(1), 79–93.
URL: *https://www.sciencedirect.com/science/article/pii/0304405X7790037X* 3, 8, 11

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. (2014), 'Dropout: A simple way to prevent neural networks from overfitting', *J. Mach. Learn. Res.* **15**(1), 1929–1958. 5

Wilmott, P. (2006), *Paul Wilmott on Quantitative Finance*, 2 edn, John Wiley & Sons. 7, 9, 18

Xiao, Y., Wei, Z. & Wang, Z. (2008), 'A limited memory BFGS-type method for large-scale unconstrained optimization', *Computers & Mathematics with Applications* **56**(4), 1001–1009.
URL: *https://www.sciencedirect.com/science/article/pii/S0898122108001028* 14

Appendix A

Derivation of the Black–Scholes Model

We are interested in pricing an option on an underlying security. We begin with the assumption that the underlying obeys a lognormal random walk with constant volatility:

$$dS = \mu S dt + \sigma S dX. \tag{A.1}$$

We know that the value of a contingent claim V on an underlying S can be modelled, by Itô's lemma, as:

$$dV = V_S dS + \frac{1}{2} V_{SS} dS^2 + V_t dt. \tag{A.2}$$

Expanding the dS^2 term:

$$dS^2 = \mu^2 S^2 dt^2 + \sigma^2 S^2 dX^2 + \mu\sigma S^2 dt dX. \tag{A.3}$$

By Itô's lemma:

$$dt^2 = 0; \qquad dX^2 = dt; \qquad dX dt = 0.$$

Equation A.3 becomes:

$$dS^2 = \sigma^2 S^2 dt. \tag{A.4}$$

Substituting the result from Equation A.4 into Equation A.2:

$$dV = V_S dS + \frac{1}{2}\sigma^2 V_{SS} S^2 dt + V_t dt.$$

Formulating a portfolio comprising the underlying and the option:

$$\Pi = V + \Delta S.$$

Then. the equation of motion of the portfolio becomes:

$$\begin{aligned} d\Pi &= dV + \Delta dS \\ &= V_S dS + \frac{1}{2}\sigma^2 S^2 V_{SS} dt + V_t dt + \Delta dS. \end{aligned}$$

Here, by setting $\Delta = -V_S$, it is possible to completely remove risk terms from the portfolio—but by no arbitrage, this risk-free portfolio must make the risk-free rate:

$$\begin{aligned} d\Pi &= \frac{1}{2}\sigma^2 S^2 V_{SS} dt + V_t dt \\ &= r\Pi dt \\ &= r(V - V_S S)dt. \end{aligned}$$

Finally, rearranging, we obtain:

$$\frac{1}{2}\sigma^2 S^2 V_{SS} + V_t + rSV_S - rV = 0.$$

It can be convenient to express the above PDE in terms of time to expiry $\tau = T - t$ as opposed to time t:

$$\frac{1}{2}\sigma^2 S^2 V_{SS} + rSV_S - V_\tau - rV = 0, \tag{A.5}$$

with an appropriate modification to the terminal conditions. Equation A.5 is the Black–Scholes PDE which we subsequently minimise in the PINN.

A.1 Black–Scholes Assumptions

The Black–Scholes model derivation relies on the following assumptions (Hull 2018, Wilmott 2006):

1. The underlying price follows a log-normal distribution (Equation A.1) with constant drift μ and implied volatility σ.
2. Short selling of securities is allowed and free of costs.
3. There are no other transaction costs or taxes, and all securities can be traded in any quantity.
4. The risk-free rate r is constant and time-invariant (or, a known function of time).
5. Delta hedging can be done in a continuous manner.

Appendix B

Structure of the Physics-Informed Neural Network

This is the sample code that can be used to generate the physics-informed neural network. The below code is used to construct a 3-layer deep NN with 400 neurons in each hidden layer, each layer activated using a hyperbolic tangent activation function. Between each hidden layer is a dropout layer which reduces the possibility of overfit.

It is possible to further optimise the NN hyperparameters by further tuning of the NN architecture. This is beyond the scope of the current project.

If using the below code directly, take note of the file structure; the `load_data` function assumes that the dataframe of true option prices are stored in a folder named `option_prices`, and that these dataframes are saved as pickle files. The Python version used is 3.10 and the TensorFlow version is 2.9.0.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


import pandas as pd

import os

import tensorflow as tf
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import MeanSquaredError


def load_data(path = "options_prices"):
    temp = []
    for i in os.listdir(path):
        temp.append(pd.read_pickle(os.path.join(path, i)))

    return pd.concat(temp)


data = load_data()

```

```
25
26 def input_output_split(df, use_64_bit=False):
27     X = df.iloc[:, 0:4]
28     Y = df.iloc[:, 4]
29
30     if use_64_bit:
31         tf.keras.backend.set_floatx("float64")
32         return X.to_numpy(dtype="float64"), Y.to_numpy(dtype="float64")
33
34     tf.keras.backend.set_floatx("float32")
35     return X.to_numpy(dtype="float32"), Y.to_numpy(dtype="float32")
36
37
38 data_train = data[(data["sigma"] != 0.0185) & (((data["spot"] <= 110)) | (data["
       spot"] >= 135))]
39 data_test = data[(data["sigma"] == 0.0185) & (data["spot"] == 115)]
40
41
42 X, Y = input_output_split(data_train, use_64_bit=True)
43 X_test, Y_test = input_output_split(data_test, use_64_bit=True)
44
45
46 noLayers = 3
47 noNeurons = 400
48 acti = "tanh"
49
50 inputNo = X.shape[1]
51 outputNo = 1
52
53
54 ub = X.max(0)
55 lb = X.min(0)
56
57
58 # Construct the PINN as a 3-layer 400 neuron network
59 PINN = tf.keras.Sequential()
60 PINN.add(tf.keras.layers.InputLayer(input_shape=(inputNo,)))
61 PINN.add(tf.keras.layers.Lambda(
62     lambda X: 2.0 * (X - lb) / (ub - lb) - 1.0))
63 init = "he_normal"
64 if acti == "tanh":
65     init = "glorot_normal"
66 for _ in range(noLayers):
67     PINN.add(tf.keras.layers.Dense(
68         noNeurons, activation=acti,
69         kernel_initializer=init))
70     PINN.add(tf.keras.layers.Dropout(
71         rate=0.1))
72 PINN.add(tf.keras.layers.Dense(
73     outputNo, activation=None,
74     kernel_initializer=init))
75
76
77 # Copy the PINN model as a control
78 MLP = tf.keras.models.clone_model(PINN)
79
```

```
80
81  S = tf.convert_to_tensor(X[:, 0:1])
82  sigma = tf.convert_to_tensor(X[:, 1:2])
83  strike = tf.convert_to_tensor(X[:, 2:3])
84  tau = tf.convert_to_tensor(X[:, 3:4])
85
86
87  import numpy as np
88  r = np.power(1.02, 1/252) - 1
89
90
91  def bsm_net():
92      with tf.GradientTape(watch_accessed_variables=False, persistent=True) as tape:
93          tape.watch(S)
94          tape.watch(tau)
95
96          # Packing together the inputs
97          X_pinn = tf.stack([S[:, 0], sigma[:, 0], strike[:, 0], tau[:, 0]], axis=1)
98
99          # Getting the prediction
100         V = PINN(X_pinn)
101
102         # Deriving INSIDE the tape (since we"ll need the S derivative of this later
                , V_SS)
103         V_S = tape.gradient(V, S)
104
105     # Finding V_tau
106     V_tau = tape.gradient(V, tau)
107
108     # Getting the second order derivatives
109     V_SS = tape.gradient(V_S, S)
110
111     # Free memory
112     del tape
113
114     return V, V_SS, V_S, V_tau
115
116
117 def loss(Y_actual, Y_pred):
118     """
119     Neural network custom loss function.
120
121     Takes in two inputs, Y_actual and Y_pred
122     Outputs a scalar loss term, which is based on the Black-Scholes equations.
123     """
124
125     V, V_SS, V_S, V_tau = bsm_net()
126
127     rmse = tf.sqrt(tf.reduce_mean(tf.square(Y_actual - Y_pred)))
128
129     bsm_error = tf.sqrt(tf.reduce_mean(tf.square(0.5*tf.square(sigma)*tf.square(S)*
            V_SS + r*S*V_S - V_tau - r*V)))
130
131     return rmse + bsm_error
132
133
```

```
# Callbacks
lr = 0.01
reduce_lr = ReduceLROnPlateau(monitor="loss", mode="min", factor=0.2, verbose=1,
    patience=4, min_lr=0)
e_stop = EarlyStopping(monitor="val_loss", mode="min", min_delta=1.0e-8, patience
    =32, verbose=0)


batch_size = 1000


PINN.compile(loss=loss, optimizer=Adam(learning_rate=lr))
PINN.fit(
        X, Y,
        batch_size=batch_size,
        epochs=10000,
        validation_split=0.1,
        callbacks=[reduce_lr, e_stop])


MLP.compile(loss=MeanSquaredError(), optimizer=Adam(learning_rate=lr))
MLP.fit(
        X, Y,
        batch_size=batch_size,
        epochs=10000,
        validation_split=0.1,
        callbacks=[reduce_lr, e_stop])


Y_pred_PINN = PINN.predict(X_test).T
Y_pred_MLP = MLP.predict(X_test).T


data_test["PINN prediction"] = Y_pred_PINN.tolist()[0]
data_test["MLP prediction"] = Y_pred_MLP.tolist()[0]


rmse_PINN = tf.sqrt(tf.reduce_mean(tf.square(Y_pred_PINN - Y_test)))
rmse_MLP = tf.sqrt(tf.reduce_mean(tf.square(Y_pred_MLP - Y_test)))


print(f"\n\nPINN OOS RMSE: {rmse_PINN}; MLP OOS RMSE: {rmse_MLP}\n\n")
```

Appendix C

Code Used to Generate 'True' Option Prices

C.1 Closed-Form Solution for Vanilla European Call

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


import pandas as pd
import numpy as np
from scipy.stats import norm

from datetime import datetime

r = np.power(1.02, 1/252) - 1

dt = 1
expiries = [3, 5, 10, 15, 21, 42, 63, 126, 252]
strikes = [55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105, 110, 115, 120, 125, 130,
    135, 140, 145, 150]
spots = [90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106,
     107, 108, 109, 110]
sigmas = [0.01, 0.012, 0.013, 0.014, 0.015, 0.016, 0.018, 0.02]


start = datetime.now()


def Call(spot, tau, strike, sigma, r):
    d1 = (1/(sigma*tau)) * (np.log(spot/strike) + (r+sigma**2/2)*tau)
    d2 = d1 - sigma * tau
    return norm.cdf(d1) * spot - norm.cdf(d2) * strike * np.exp(-r * tau)


stored = pd.DataFrame(columns=["spot", "sigma", "strike", "expiry", "value"])

chain = []

```

```
for sigma in sigmas:
    for tau in expiries:
        for spot in spots:
            for strike in strikes:
                value = Call(spot=spot, tau=tau, strike=strike, sigma=sigma, r=r)
                chain_link = pd.DataFrame({"spot":spot, "sigma":sigma, "strike":
                    strike, "expiry":tau, "value":value}, index=[0])
                chain.append(chain_link)

stored = pd.concat(chain, ignore_index=True, copy=False).round(4)

stored.to_pickle("options_prices_closed_form/options_chain.pkl")

print(f"Runtime = {datetime.now() - start}")
```

C.2 Finite Differencing Solution for Up-and-Out European Call

This code was adapted from Hovhannisyan (2022), who in turn adapted from Capiński & Zastawniak (2012).

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


import pandas as pd
from datetime import datetime
import numpy as np



class EuCallBlackScholes:
    """
    Class containing data and methods to describe the Black-Scholes PDE for
    up-and-out European call option
    """
    def __init__(self, sigma, strike, rate, t_up, x_low, x_up, barrier):
        """
        Constructor
        :param sigma: underlying asset's standard deviation
        :param rate: interest rate
        :param strike: option's strike price
        :param t_up: option"s time to expiration
        :param x_low: lower bound for the option's spot price
        :param x_up: upper bound for the option's spot price
        """
        self.sigma = sigma
        self.rate = rate
        self.strike = strike
        self.t_up = t_up
        self.x_low = x_low
        self.x_up = x_up
        self.barrier = barrier

```

```
    def coeff_a(self, t, x):
        return -0.5*self.sigma**2*x**2

    def coeff_b(self, t, x):
        return -self.rate*x

    def coeff_c(self, t, x):
        return self.rate

    def coeff_d(self, t, x):
        return 0

    def bound_cond_tup(self, x):
        return max(x - self.strike, 0) if x < self.barrier else 0

    # Set upper and lower boundary conditions
    def bound_cond_xlow(self, t):
        return 0

    def bound_cond_xup(self, t):
        return 0



class PDESolver:
    """
    Abstract class to solve Black-Scholes PDEs.
    """
    def __init__(self, pde, imax, jmax):
        """
        Constructor
        :param pde: The PDE to solve
        :param imax: last value of the first variable's discretisation
        :param jmax: last value of the second variable's discretisation
        """
        # 1. Initialise `pde`, `imax`, `jmax` members
        # 2. Pre-compute `dt` and `dx` members for the grid
        # 3. Initialise a grid matrix to contain solutions using `np.empty`
        self.pde = pde
        self.imax = imax
        self.jmax = jmax
        self.dt = pde.t_up / imax
        self.dx = (pde.x_up - pde.x_low) / jmax
        self.grid = np.empty((imax + 1, jmax + 1), dtype = float)

    def _t(self, i):
        """Return the discretised value of t at index i"""
        return self.dt * i

    def _x(self, j):
        """Return the discretised value of x at index j"""
        return self.dx * j + self.pde.x_low

    def a(self, i, j):
        """
        Helper umbrella function to get coefficient a at discretised locations
```

```
90          """
91          return self.pde.coeff_a(self._t(i), self._x(j))
92
93      def b(self, i, j):
94          """
95          Helper umbrella function to get coefficient b at discretised locations
96          """
97          return self.pde.coeff_b(self._t(i), self._x(j))
98
99      def c(self, i, j):
100         """
101         Helper umbrella function to get coefficient c at discretised locations
102         """
103         return self.pde.coeff_c(self._t(i), self._x(j))
104
105     def d(self, i, j):
106         """
107         Helper umbrella function to get coefficient d at discretised locations
108         """
109         return self.pde.coeff_d(self._t(i), self._x(j))
110
111     def tup(self, j):
112         """
113         Helper umbrella function to get upper boundary condition
114         for time at discretised j index
115         :param j: discretised x index
116         """
117         return self.pde.bound_cond_tup(self._x(j))
118
119     def xlow(self, i):
120         """
121         Helper umbrella function to get lower boundary condition
122         for x at discretised i index
123         :param i: discretised t index
124         """
125         return self.pde.bound_cond_xlow(self._t(i))
126
127     def xup(self, i):
128         """
129         Helper umbrella function to get upper boundary condition
130         for x at discretised i index
131         :param i: discretised t index
132         """
133         return self.pde.bound_cond_xlow(self._t(i))
134
135     def interpolate(self, t, x):
136         """
137         Get interpolated solution value at given time and space
138         :param t: point in time
139         :param x: point in space
140         :return: interpolated solution value
141         """
142         i = int(t // self.dt)
143         j = int(x // self.dx)
144
145         l1 = (t - self.dt*i) / self.dt
```

```
146         l0 = 1 - l1
147
148         w1 = (x - (self.pde.x_low + self.dx*j)) / self.dx
149         w0 = 1 - w1
150
151         return l1*w1*self.grid[i+1, j+1] + l1*w0*self.grid[i+1, j] + l0*w1*self.
                grid[i, j+1] + l0*w0*self.grid[i, j]
152
153
154
155 class ExplicitScheme(PDESolver):
156     """
157     Black-Scholes PDE solver using the explicit scheme
158     """
159
160     def __init__(self, pde, imax, jmax):
161         super().__init__(pde, imax, jmax)
162
163     def _A(self, i, j):
164         """
165         Coefficient A_{i,j} for the explicit scheme
166         :param i: index of x discretisation
167         :param j: index of t discretisation
168         """
169         return self.dt/self.dx*(self.b(i,j)/2 - self.a(i,j)/self.dx)
170
171     def _B(self, i, j):
172         """
173         Coefficient B_{i,j} for the explicit scheme
174         :param i: index of x discretisation
175         :param j: index of t discretisation
176         """
177         return 1 - self.dt*self.c(i,j) + (2*self.dt*self.a(i,j))/(self.dx**2)
178
179     def _C(self, i, j):
180         """
181         Coefficient C_{i,j} for the explicit scheme
182         :param i: index of x discretisation
183         :param j: index of t discretisation
184         """
185         return -self.dt/self.dx*(self.b(i,j)/2 + self.a(i,j)/self.dx)
186
187
188     def _D(self, i, j):
189         """
190         Coefficient D_{i,j} for the explicit scheme
191         :param i: index of x discretisation
192         :param j: index of t discretisation
193         """
194         return -self.dt*self.d(i,j)
195
196     def solve_grid(self):
197         """
198         Solves the PDE and saves the values in grid
199         """
200         for j in range(self.jmax + 1):
```

```
            self.grid[self.imax, j] = self.tup(j)
        self.grid[self.imax, self.jmax] = 0

        def v(i, j):
            return (self._A(i+1,j)*self.grid[i+1,j-1] +
                    self._B(i+1,j)*self.grid[i+1,j] +
                    self._C(i+1,j)*self.grid[i+1,j+1] +
                    self._D(i+1,j))

        for i in range(self.imax - 1, -1, -1):
            self.grid[i, 0] = self.xlow(i)
            self.grid[i, self.jmax] = self.xup(i)
            for j in range(1, self.jmax):
                self.grid[i,j] = v(i,j)



expiries = [3, 5, 10, 15, 21, 42, 63, 126, 252]
strikes = [90, 95, 100, 105, 110, 115, 120]
sigmas = [0.01, 0.0125, 0.015, 0.0175, 0.0185, 0.02]
barrier = 140


T = 1
rate = 0.01
strike = 100
spot_low = 0
spot_up = barrier


start = datetime.now()


for sigma in sigmas:
    chain = []
    for strike in strikes:
        # Initialise a PDE object representing the Black-Scholes equation
        pde = EuCallBlackScholes(sigma=sigma,
                                 strike=strike,
                                 rate=rate,
                                 t_up=T,
                                 x_low=spot_low,
                                 x_up=spot_up,
                                 barrier=barrier)

        # Create a solver object for the explicit scheme
        imax = 12600
        jmax = 1400
        exp_scheme = ExplicitScheme(pde, imax, jmax)
        # Solve the PDE on grid points
        exp_scheme.solve_grid()

        for spot in range(80, 141, 1):
            for expiry in expiries:
                t = 1 - expiry/252
                value = exp_scheme.interpolate(t, spot)
```

```
                # print(f"Sigma: {sigma}; strike: {strike}; spot: {spot}; expiry: {
                    expiry}; value: {value}")

                chain_link = pd.DataFrame({"spot":spot, "sigma":sigma, "strike":
                    strike, "expiry":expiry, "value":value}, index=[0])
                chain.append(chain_link)

    stored = pd.DataFrame(columns=["spot", "sigma", "strike", "expiry", "value"])
    stored = pd.concat(chain, ignore_index=True, copy=False).round(4)
    stored.to_pickle(f"options_prices/options_chain_{sigma}.pkl")

print(f"Time taken: {datetime.now() - start}")
```