Pricing European Cryptocurrency Options using Numerical Replication

Stan Uryasev, Jack Peters, and Taras Vorobets

Abstract

In this case study, using the methodical approach of Ryabchenko, Sarykalin, and Uryasev[12], we price European style options for Bitcoin and Ethereum. This approach involves approximating the value of an option by using a portfolio consisting of the underlying and a risk-free bond. Using quadratic programming, we produce tables of call and put prices based on the price of the underlying. We model the price of an asset using a set of sample-paths generated from historical prices which we massage to reflect the current volatility of the market.

Keywords: Bitcoin, Ethereum, Options Pricing, Implied Volatility, European Options, Quadratic Programming.

1 Introduction

In 2008, Satoshi Nakamoto authored a paper detailing a decentralized peer-to-peer electronic cash system they called Bitcoin[11]. Since the paper's publication, Bitcoin has gained an immense following of supporters and has become the face of the decentralized currency movement with a current market cap of roughly $410 billion, as of writing. Many people have been drawn to Bitcoin due to it being beyond the reach of central bank control and its fixed supply of 21 million Bitcoins. Supporters see these characteristics as being answers to instabilities that exist within current financial systems. Shortly after the inception of Bitcoin, Vitalik Buterin published a white paper in 2014 proposing a new cryptocurrency called Ethereum[5]. Ethereum, unlike Bitcoin, is a programmable blockchain that allows for the development of numerous decentralized applications in addition to offering a cryptocurrency. Many supporters of cryptocurrency advocate for a moving away from fiat currencies like the U.S. dollar and the Euro, and transitioning to the use of cryptocurrency as an alternative currency for transactions.

Contrary to what many supporters advocate for, cryptocurrency is overwhelmingly being used as a speculative asset rather than as an alternative currency[4][9][13]. In 2019, various cryptocurrency exchanges such as Binance and LedgerX started offering trading of Bitcoin options. There has also been the creation of exchanges like Deribit whose sole focus is to cryptocurrency derivatives trading. Given the incredibly volatile nature of Bitcoin[2][3], these options can be highly profitable. These trading platforms are largely using the Black-Scholes model to calculate the implied volatility of the options. Given that cryptocurrenices have not been shown to follow a random walk[1] and that it is replete with speculative bubbles[7][8], the use of the Black-Scholes model in pricing cryptocurrency options is questionable. Cao and Celik[6] as well as Jalan et al.[10] showed that the Black-Scholes model inaccurately values cryptocurrency options.

In this case study, we wish to use the robust options pricing method developed by Ryabchenko, Sarykalin, and Uryasev[12] to price options on incomplete markets. Following their method, we construct a hedging portfolio consisting of the underlying crypto (either Bitcoin or Ethereum) and a risk-free bond. We model the price of Bitcoin and Ethereum using a set of sample-paths generated from historical prices which we massage to reflect the current volatility of the market. We then construct a grid modelled on discrete time and crypto prices whose nodes reflect the position in the crypto and the amount of money invested in the bond. This can be done using two matrices which completely determine the hedging portfolio for any price path of the underlying crypto. Through minimization of the averaged quadratic error over all sample-paths; the free variables of the two matrices are optimized such that crytpo and bond positions define every node of the grid. Using the optimized free variables we can then price Bitcoin and Ethereum options given the option's strike and the given cryptocurrency's current price. We compare our results with those of the

largest Bitcoin and Ethereum options exchange by volume, Deribit, who prices its options using the Black-Scholes model.

2 Framework

Consider a European option with time to maturity T and strike price X. We assume that trading occurs at discrete times t_j for $j = 0, 1, \ldots, N$, such that $0 = t_0 < t_1 < ... < t_N = T$ where $t_{j+1} - t_j = const$ and $j = 0, 1, \ldots, N-1$. Denote the position in cryptocurrency at time t_j by u_j, the amount of money invested in the risk-free bond by v_j, the risk-free rate by r_0, and the cryptocurrency price at time t_j by B_j.

The value of the portfolio right before time t_j is $u_{j-1}B_j + (1 + r_0)\, v_{j-1}$ and the value of the portfolio at time t_j is $u_j B_j + v_j$. We define the excess/shortfall of the money in the hedging portfolio during the interval $[t_{j-1}, t_j]$ to be

$$a_j = u_j B_j + v_j - \left[u_{j-1}B_j + (1 + r_0)\, v_{j-1}\right].$$

2.1 Dynamics

We require that the value of the hedging portfolio at expiration be equal to the option payoff:

$$u_N B_N + v_N = \begin{cases} \max\left[0, B_N - X\right] & \text{for calls} \\ \max\left[0, X - B_N\right] & \text{for puts} \end{cases}$$

where X is the strike of the option. The non-self-financing portfolio dynamics are given by

$$u_{j+1}B_{j+1} + v_{j+1} = u_j B_j + (1 + r_0)\, v_j + a_j$$

for $j = 0, 1, \ldots, N-1$ where the portfolio value at time t_j is $c_j = u_j B_j + v_j$, for $j = 0, 1, \ldots, N$.

2.2 Hedging Strategy

Assuming the hedging portfolio depends only on time and cryptocurrency price, we can model it using a discrete grid with approximation rules. Consider a grid consisting of nodes $\{(k, j)\,|k = 1, 2, \ldots, K; j = 0, 1, \ldots, N\}$ where j denotes the time t_j and k denotes the cryptocurrency price $\tilde{B}_k$ (here we use the tilde sign to distinguish cryptocurrency prices on the grid from cryptocurrency prices on the sample-paths). The cryptocurrency prices on the grid, $\tilde{B}_k$, are chosen such that they are equally distanced in the logarithmic scale, i.e. for $\tilde{B}_1 < \tilde{B}_2 < \cdots < \tilde{B}_K$ we

require

$$\ln\left(\tilde{B}_{k+1}\right) - \ln\left(\tilde{B}_k\right) = const.$$

Every node (k, j) is assigned variables $U_{k,j}$ and $V_{k,j}$ which indicate the composition of the position in the hedging portfolio at time t_j with cryptocurrency price $\tilde{B}_k$. This can be represented by the following matrices which define our hedging strategy:

$$[U_{k,j}] = \begin{bmatrix} U_{1,0} & U_{1,1} & \cdots & U_{1,N} \\ U_{2,0} & U_{2,1} & \cdots & U_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ U_{K,0} & U_{K,1} & \cdots & U_{K,N} \end{bmatrix}, \quad [V_{k,j}] = \begin{bmatrix} V_{1,0} & V_{1,1} & \cdots & V_{1,N} \\ V_{2,0} & V_{2,1} & \cdots & V_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ V_{K,0} & V_{K,1} & \cdots & V_{K,N} \end{bmatrix}.$$

The matrices define the portfolio management decisions on the grid nodes.

The cryptocurrency price dynamics are modeled using a set of sample-paths $\{(B_0, B_1^p, \ldots, B_N^p) \,|p = 1, 2, .$ where B_0 is the initial price which is the same for each sample path (data normalization is used such that $B_0^p = B_0$ for all p). Here, B_j^p denotes the cryptocurrency price on sample-path p at time t_j. To determine the composition of the hedging portfolio at any time t_j and on any cryptocurrency price path p, that may or may not pass through a grid node, we must use the following approximation:

$$\begin{gathered} u_j^p = \alpha_j^p U_{k^*+1,j} + \left(1 - \alpha_j^p\right) U_{k^*,j}, \\ v_j^p = \alpha_j^p V_{k^*+1,j} + \left(1 - \alpha_j^p\right) V_{k^*,j} \\ \text{where } \alpha_j^p = \frac{\ln\left(B_j^p\right) - \ln\left(\tilde{B}_{k^*}\right)}{\ln\left(\tilde{B}_{k^*+1}\right) - \ln\left(\tilde{B}_{k^*}\right)} \end{gathered} \tag{1}$$

and k^* is such that $\tilde{B}_{k^*} \leq B_j^p < \tilde{B}_{k^*+1}$. Thus, the excess/shortfall of the money in the hedging portfolio on path p at time t_j is

$$a_j^p = u_j^p B_j^p + v_j^p - \left[u_{j-1}^p B_j^p + (1 + r_0)\, v_{j-1}^p\right].$$

3 Pricing Options

The value of an option is found from the first column values of the matrices $[U_{k,j}]$ and $[V_{k,j}]$, namely variables $U_{k,0}$ and $V_{k,0}$ for $k = 1, 2, \ldots, K$. If the cryptocurrency price B_0 corresponds with an initial grid node $\left(\tilde{k}, 0\right)$, then the price of the option is given by $C_{\tilde{k},0} = U_{\tilde{k},0} B_0 + V_{\tilde{k},0}$. If the cryptocurrency price B_0 falls between two of the initial grid nodes $(k, 0)$, $k = 1, 2, \ldots, K$, then the price of the option is given by $C = u_0 B_0 + v_0$, where u_0 and v_0 are found from equation (1) with $B_j^p = B_0$.

3.1 Optimization

The values of the matrices can be found by solving the following objective:

$$\min \bar{\varepsilon} = \frac{1}{P}\sum_{p=1}^{P}\sum_{j=1}^{N}\left(a_j^p e^{-r_0 j}\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\sum_{j=1}^{N}\left\{\left[u_j^p B_j^p + v_j^p - u_{j-1}^p B_j^p - (1+r_0)\, v_{j-1}^p\right] e^{-r_0 j}\right\}^2$$

subject to

$$\begin{gathered}\tfrac{1}{P}\textstyle\sum_{p=1}^{P}\sum_{j=1}^{N}\left[u_j^p B_j^p + v_j^p - u_{j-1}^p B_j^p - (1+r_0)\, v_{j-1}^p\right] e^{-r_0 j} = 0 \\ U_{k,N}\tilde{B}_k + V_{k,N} = \max\left[0, \tilde{B}_k - X\right] \text{ for strike } X \text{ and } k = 1, 2, \ldots, K.\end{gathered}$$

The objective function above sets the average squared error on the sample-paths we receive when we model sample cryptocurrency price dynamics. The first constraint requires that the average value of total external financing over all paths equals to zero. The second constraint equates the value of the portfolio and the option payoff at expiration. Free variables in this problem are the grid variables $U_{k,j}$ and $V_{k,j}$; the path variables u_j^p and v_j^p in the objective are expressed in terms of the grid variables using approximation (1). The total number of free variables is independent of the number of sample-paths and is equal to $2(N+1)K$.

3.2 European Call Constraints

The notation $C_{k,j}$ stands for the call option value in the node (k, j) of the grid:

$$C_{k,j} = U_{k,j}\tilde{B}_k + V_{k,j}.$$

In the following constrains for call options, the strike price of the option is denoted by X, the time to expiration by T, and the one period risk-free rate by r_0. We make five assumptions.

Assumption 1. *Non-arbitraity holds.*

Assumption 2. *Current and future interest rates are positive.*

Assumption 3. *Time homogeneity.*

Assumption 4. *The distributions of the returns per dollar invested in a cryptocurrency for any period of time is independent of the level of the cryptocurrency price.*

Assumption 5. *If the returns per dollar on cryptocurrency investments x and y are identically distributed, then: if $B_x = B_y$, $T_x = T_y$, and $X_x = X_y$ then $Call_x = Call_y$.*

3.2.1 Call Price Sensitivity Constraints

Allows for the bound sensitivity of an options price to be uninfluenced by changes in the cryptocurrency price.

$$C_{k+1,j} \leq \gamma_{k,j} C_{k,j} + X\left(\gamma_{k,j} - 1\right) e^{-r_0(T-t_j)}$$
$$\text{where } \gamma_{k,j} = \tilde{B}_{k+1}/\tilde{B}_k$$
$$\text{for } j = 0, 1, \ldots, N-1; k = 1, 2, \ldots, K-1.$$

3.2.2 Call Price Vertical Monotonicity

For any given fixed time, the price of the option is an increasing function of the cryptocurrency price.

$$\frac{\tilde{B}_k}{\tilde{B}_{k+1}} C_{k+1,j} \geq C_{k,j}$$
$$\text{for } j = 0, 1, \ldots, N; k = 1, 2, \ldots, K-1.$$

3.2.3 Call Price Horizontal Monotonicity

For any given fixed time, the price of the option is a decreasing function of time.

$$C_{k,j+1} \leq C_{k,j}$$
$$\text{for } j = 0, 1, \ldots, N-1; k = 1, 2, \ldots, K.$$

3.2.4 Call Price Convexity Constraint

Option value holds a convex function to the cryptocurrency price

$$C_{k+1,j} \leq \beta_{k+1,j} C_{k,j} \left(1 - \beta_{k+1,j}\right) C_{k+2,j}$$
$$\text{where } \beta_{k+1,j} = \frac{\tilde{B}_{k+1} - \tilde{B}_{k+2}}{\tilde{B}_k - \tilde{B}_{k+2}}$$
$$\text{for } j = 0, 1, \ldots, N; k = 1, 2, \ldots, K-2.$$

3.3 European Put Constraints

The notation $P_{k,j}$ stands for the put option value in the node (k, j) of the grid:

$$P_{k,j} = U_{k,j} \tilde{B}_k + V_{k,j}.$$

In the following constrains for put options, the strike price of the option is denoted by X, the time to expiration by T, and the one period risk-free rate by r_0. Similar to the case for calls, we make five assumptions.

Assumption 6. *Non-arbitraity holds.*

Assumption 7. *Current and future interest rates are positive.*

Assumption 8. *Time homogeneity.*

Assumption 9. *The distributions of the returns per dollar invested in a cryptocurrency for any period of time is independent of the level of the cryptocurrency price.*

Assumption 10. *If the returns per dollar on cryptocurrency investments x and y are identically distributed, then: if $B_x = B_y$, $T_x = T_y$, and $X_x = X_y$ then $Put_x = Put_y$.*

3.3.1 Put Price Sensitivity Constraints

Allows for the bound sensitivity of an options price to be uninfluenced by changes in the cryptocurrency price.

$$P_{k,j} \leq \gamma_{k,j} P_{k+1,j} + X\left(1 - \gamma_{k,j}\right) e^{-r_0(T-t_j)}$$
$$\text{where } \gamma_{k,j} = \tilde{B}_k / \tilde{B}_{k+1}$$
$$\text{for } j = 0, 1, \ldots, N-1; k = 1, 2, \ldots, K.$$

3.3.2 Put Price Vertical Monotonicity

For any given fixed time, the price of the option is an increasing function of the cryptocurrency price.

$$P_{k,j} \geq P_{k+1,j}$$
$$\text{for } j = 0, 1, \ldots, N; k = 1, 2, \ldots, K-1.$$

3.3.3 Put Price Horizontal Monotonicity

For any given fixed time, the price of the option is a decreasing function of time.

$$P_{k,j+1} \leq P_{k,j} + X(e^{-r(T-t_{j+1})} - e^{-r(T-t_j)})$$
$$\text{for } j = 0, 1, \ldots, N-1; k = 1, 2, \ldots, K.$$

3.3.4 Put Price Convexity Constraint

Option value holds a convex function to the cryptocurrency price.

$$P_{k+1,j} \leq \beta_{k+1,j} P_{k,j} \left(1 - \beta_{k+1,j}\right) P_{k+2,j}$$
$$\text{where } \tilde{B}_{k+1,j} = \beta_{k+1,j} \tilde{B}_{k,j} \left(1 - \beta_{k+1,j}\right) \tilde{B}_{k+2,j}$$
$$\text{for } j = 0, 1, \ldots, N; k = 1, 2, \ldots, K-2.$$

3.4 Cryptocurrency Position Constraints

3.4.1 Cryptocurrency Position Bounds

Cryptocurrency position values are between 0 and 1.

$$0 \leq U_{k,j} \leq 1$$
$$\text{for } j = 0, 1, \ldots, N; k = 1, 2, \ldots, K.$$

3.4.2 Cryptocurrency Position Vertical Monotonicity

The position in cryptocurrency is an increasing function of its price.

$$U_{k+1,j} \geq U_{k,j}$$
$$\text{for } j = 0, 1, \ldots, N; k = 1, 2, \ldots, K-1.$$

3.4.3 Cryptocurrency Position Horizontal Monotonicity

Above the strike price the position in Bitcoin is an increasing function of time; below the strike price it is a decreasing function of time.

$$\begin{cases} U_{k,j} \leq U_{k,j+1} & \text{if } k > \hat{k} \\ U_{k,j} \geq U_{k,j+1} & \text{if } k \leq \hat{k} \end{cases}$$
$$\text{where } \hat{k} \text{ is such that } \tilde{B}_{\hat{k}} \leq X < \tilde{B}_{\hat{k}+1}$$
$$\text{for } j = 0, 1, \ldots, N-1; k = 1, 2, \ldots, K.$$

3.4.4 Cryptocurrency Position Convexity Constrain

The position in cryptocurrency is a concave function in price above the strike and convex below the strike.

$$\begin{cases} (1 - \beta_{k+1,j})\, U_{k+2,j} + \beta_{k+1,j} U_{k,j} \leq U_{k+1,j} & \text{if } k > \hat{k} \\ (1 - \beta_{k-1,j})\, U_{k-2,j} + \beta_{k-1,j} U_{k,j} \geq U_{k-1,j} & \text{if } k \leq \hat{k} \end{cases}$$
$$\text{where } \hat{k} \text{ is such that } \tilde{B}_{\hat{k}} \leq X < \tilde{B}_{\hat{k}+1}$$
$$\text{and } \beta_{k+1,j} = \frac{\tilde{B}_{k+1} - \tilde{B}_{k+2}}{\tilde{B}_k - \tilde{B}_{k+2}} \text{ and } \beta_{k-1,j} = \frac{\tilde{B}_{k-1} - \tilde{B}_k}{\tilde{B}_{k-2} - \tilde{B}_k},$$
$$\text{for } j = 0, 1, \ldots, N; k = \begin{cases} \hat{k}+1, \hat{k}+2, \ldots . K-2 & \text{if } \hat{k} < 3 \\ 3, 4, \ldots, K-2 & \text{if } 3 \leq \hat{k} \leq K-2 \\ 3, 4, \ldots, \hat{k} & \text{if } \hat{k} > K-2. \end{cases}$$

4 Replication of Ryabchenko, Sarykalin, and Uryasev's[12] Results

To verify our implementation of the pricing algorithm is working correctly, we replicate the results of Ryabchenko, Sarykalin, and Uryasev[12].

4.1 Geometric Brownian Motion

Similar to Ryabchenko, Sarykalin, and Uryasev[12], we simulate 200 sample paths of a stock process which follows geometric brownian motion with drift 10%, volatility 20%, intial stcok price \$62, and time to maturity 69 days. We use 10 strike prices ranging from \$54 to \$71. Tab. 1 shows the results of pricing geometric brownian motion call options using 200 sample-paths. The price error and volatility error are comparable to the results found in [12]. Fig. 4.1 shows the volatility smiles for geometric brownian motion of the calculated volatility using the pricing algorithm and the expected volatility found using the Black-Scholes method. Notice the smiles are similar except for the ends.

Pricing geometric brownian motion calls on 200 paths

Strike(\$)	Calc.Price(\$)	B-S.Price(\$)	Price Err.(%)	Calc.Vol.(%)	B-S.Vol.(%)	Vol.Err.(%)
54.00	9.0585	9.0813	-0.25	18.69	20.00	-6.55
55.99	7.2316	7.2486	-0.23	19.55	20.00	-2.25
57.97	5.5355	5.5536	-0.32	19.71	20.00	-1.45
60.02	3.9880	4.0034	-0.38	19.82	20.00	-0.90
62.00	2.7377	2.7607	-0.83	19.78	20.00	-1.10
62.99	2.2274	2.2430	-0.69	19.85	20.00	-0.75
64.98	1.3966	1.4115	-1.05	19.85	20.00	-0.75
67.02	0.8076	0.8230	-1.87	19.82	20.00	-0.90
69.01	0.4325	0.4564	-5.24	19.64	20.00	-1.80
71.00	0.2123	0.2380	-10.80	19.43	20.00	-2.85

Tab. 1: Initial price = \$62, time to expiration = 69 days, risk-free rate = 10%, modeled using 200 sample paths following geometric brownian motion, grid dimensions: $K = 25$, $N+1 = 70$. Strike(\$) = the strike price for the European call option. Calc.Price(\$) = the calculated price of a European call option using the pricing algorithm. B-S.Price(\$) = the Black-Scholes call price. Price Err.(%) = (Calc.Price(\$) - B-S.Price(\$))/B-S.Price(\$). Calc.Vol.(%) = the calculated implied volatility using the Black-Scholes method with a call price equal to the Calc.Price(\$). B-S.Vol(%) = Black-Scholes volatility. Vol.Err.(%) = (Calc.Vol.(\$) - B-S.Vol.(\$))/B-S.Vol.(\$).

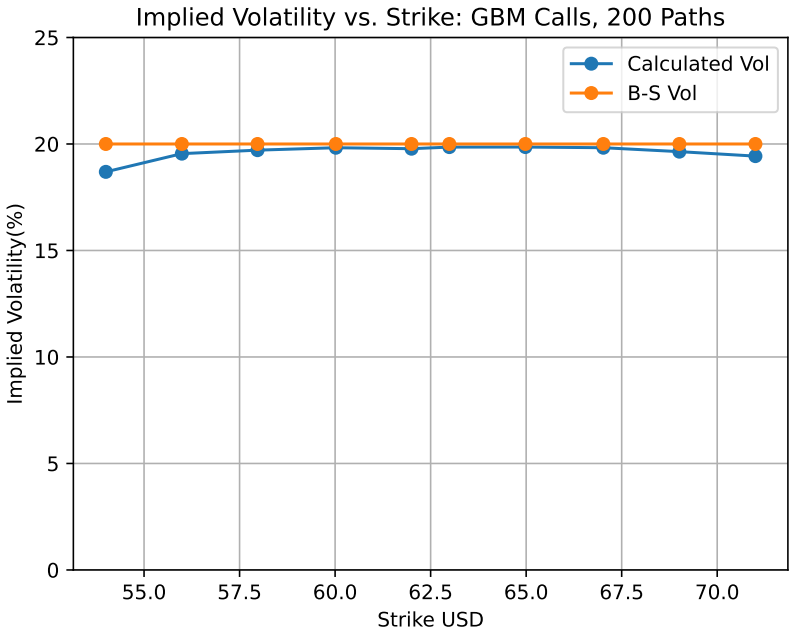

Fig. 4.1: The volatility smile of the geometric brownian motion call options. Initial price = \$62, time to expiration = 69 days, risk-free rate = 10%, modeled using 200 sample paths, grid dimensions: $K = 25$, $N + 1 = 70$. The curve of the pricing algorithm is presented in blue. The Black-Scholes data is presented in orange.

4.2 S&P 500

Next we replicate the results of the case study done in [12]. We took historical SPX daily price data ranging from 1980-01-01 to 2022-01-01. Using 100 historical sample-paths of the S&P 500 index, we price call options in Tab. 2 using the pricing algorithm. The calculated call values are converted into implied volatility using a linear interpolation method on the actual call price and actual volatility. Notice the low volatility error in Tab. 2. These results are also consistent with [12]. Fig. 4.2 shows the actual volatility smile for the SPX calls and the volatility smile found using the pricing algorithm.

We now have established that our implementation of the pricing algorithm is working effectively.

Pricing SPX calls on 100 paths

Strike($)	Calc.Price($)	Act.Price($)	Price Err.(%)	Calc.Vol.(%)	Act.Vol.(%)	Vol.Err.(%)
1100	88.5385	91.2000	-2.92	17.20	17.35	-0.86
1125	66.6392	68.9000	-3.28	16.05	16.13	-0.50
1150	46.6192	49.0000	-4.86	15.29	15.40	-0.71
1175	30.1227	31.8000	-5.27	14.50	14.60	-0.68
1190	21.8119	23.1000	-5.58	13.98	14.06	-0.57
1200	17.5147	18.2000	-3.77	13.72	13.77	-0.36
1210	13.3741	14.0000	-4.47	13.45	13.49	-0.30
1225	9.2004	9.1000	1.10	13.19	13.18	0.08
1250	4.1916	3.9500	6.12	12.82	12.80	0.16
1275	1.5820	1.4000	13.00	12.43	12.40	0.24
1300	0.6861	0.6250	9.78	12.88	12.92	-0.31
1325	0.1198	0.4000	-70.05	14.74	14.14	4.24

Tab. 2: Initial price = \$1183.77, time to expiration = 49 days, risk-free rate = 2.3%, modeled using 100 sample paths, grid dimensions: $K = 15$, $N + 1 = 50$. Strike($) = the strike price for the European call option. Calc.Price($) = the calculated price of a European call option using the pricing algorithm. Act.Price($) = the actual European call option price. Price Err.(%) = (Calc.Price($) - Act.Price($))/Act.Price($). Calc.Vol.(%) = the calculated implied volatility using an interpolation method on the SPX call data where the call price is equal to the Calc.Price($). Act.Vol(%) = the actual implied volatility. Vol.Err.(%) = (Calc.Vol.($) - Act.Vol.($))/Act.Vol.($).

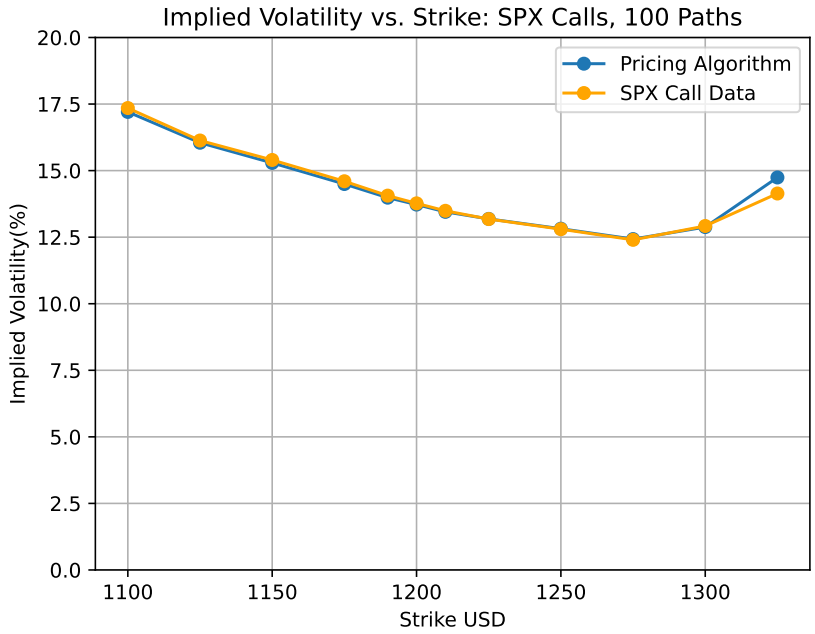

Fig. 4.2: The volatility smile of SPX call options. Initial price = \$1183.77, time to expiration = 49 days, risk-free rate = 2.3%, modeled using 100 sample paths, grid dimensions: $K = 15$, $N + 1 = 50$. The curve of the pricing algorithm is presented in blue. The SPX call data is presented in orange.

5 Cryptocurrency Case Study

We now conduct our own case study on Bitcoin and Ethereum options. Historical data for Bitcoin and Ethereum were taken from 2014-09-17 to 2022-11-05 and 2017-11-09 to 2022-11-05, respectively. We divided the historical data into non-overlapping sample-paths of equal length $N + 1 = 20$ and normalized the paths to have the same initial price B_0. Next, the volatility of the set of sample-paths was adjusted such that each sample-path had a volatility equal to that of the implied volatility $\hat{\sigma}$ of the at-the-money option. Here, $\hat{\sigma}$ was taken from the prominent exchange Deribit. Volatility is adjusted by calculating the daily returns $R_j^p = B_{j+1}^p / B_j^p - 1$ for each sample-path p, where B_j^p is the cryptocurrency price at time step t_j on path p, and modifying the returns as follows:

$$\tilde{R}_j^p = \frac{\hat{\sigma}}{\text{std.dev}\left(R^p\right)} \left(R_j^p - E\left[R^{All}\right]\right)$$

where R^p is the set of all returns on path p and R^{All} is the set of all returns across all paths. Using the modified returns, the sample-paths were reconstructed as follows:

$$\text{path } p = \left\{B_0, \tilde{I}_1^p, \tilde{I}_2^p, \ldots, \tilde{I}_N^p\right\}$$

where

$$\tilde{I}_j^p = B_0 \prod_{i=0}^{j-1} \left(\tilde{R}_i^p + 1 \right)$$

for $j = 1, 2, \ldots, N$. This set of sample-paths with adjusted volatility was used to price the options. The pricing algorithm was optimized using GUROBI. Similar to what was done in section 4.2, the calculated option prices were converted into volatilities using an interpolation method on the actual Deribit price and actual Deribit volatility. The coded implementation of the case study can be found in the appendix.

5.1 Bitcoin Results

Historical Bitcoin price data ranging from 2014-09-17 to 2022-11-05, was taken as the basis for our sample-path construction. The initial price is set to $B_0 = 21282.6914$. At the time of writing, the implied volatility of an at-the-money month long Bitcoin option on Deribit is approximately 47.50%. We take 120 sample paths, set the time to expiration $T = 19$ days, use a risk free-rate of $r_0 = 0.02$, and define our grid dimensions with $K = 25$ Bitcoin price rows and $N + 1 = 20$ time steps.

Tab. 3 and Tab. 4 show the results of pricing European call and put options on Bitcoin, respectively. The calculated price (Calc.Price) and the calculated volatility (Calc.Vol.) are the results of the pricing algorithm presented in this paper. The actual price (Act.Price) and the actual volatility (Act.Vol.) come from the exchange Deribit and reflect actual trade numbers for 20 day European calls and puts. Fig. 5.1 and Fig. 5.2 show the volatility smile for European call and put options on Bitcoin, respectively. In these figures, the result of the pricing algorithm presented here is shown in blue, while the Deribit exchange data is presented in orange.

Pricing Bitcoin calls on 120 paths

Strike($)	Calc.Price($)	Act.Price($)	Price Err.(%)	Calc.Vol.(%)	Act.Vol.(%)	Vol.Err.(%)
17000	0.202434	0.232000	-12.7440	117.57	126.32	-6.9269
18000	0.157496	0.190500	-17.3249	105.41	114.07	-7.5918
18500	0.136388	0.170500	-20.0070	100.60	108.43	-7.2212
19000	0.115972	0.152000	-23.7026	86.10	104.26	-17.4180
19500	0.096723	0.133500	-27.5483	64.27	99.35	-35.3095
20000	0.079391	0.082500	-3.7685	52.25	53.75	-2.7907
21000	0.051003	0.049500	3.0364	48.18	48.07	0.2288
21500	0.039632	0.037500	5.6853	47.35	47.32	0.0634
22000	0.030561	0.028500	7.2316	47.58	47.64	-0.1259
22500	0.023654	0.021000	12.6381	47.60	47.61	-0.0210
23000	0.017484	0.015500	12.8000	47.76	48.05	-0.6035
23500	0.012534	0.011500	8.9913	48.78	48.83	-0.1024
24000	0.008515	0.008000	6.4375	48.44	48.67	-0.4726
25000	0.004070	0.004500	-9.5556	50.34	50.91	-1.1196
26000	0.001653	0.002500	-33.8800	65.37	52.80	23.8068

Tab. 3: Initial price = \$21282.6914, time to expiration = 19 days, risk-free rate = 2%, modeled using 120 sample paths, grid dimensions: $K = 25$, $N + 1 = 20$. Strike($) = the strike price for the European call option. Calc.Price($) = the calculated price of a European call option as a fraction of the initial price using the pricing algorithm. Act.Price($) = the actual European call option price as a fraction of the initial price from Deribit. Price Err.(%) = (Calc.Price($) - Act.Price($))/Act.Price($). Calc.Vol.(%) = the calculated implied volatility using an interpolation method on the Deribit data where the put price is equal to the Calc.Price($). Act.Vol(%) = the actual implied volatility from Deribit which they calculate using the Black-Scholes equation with a call price equal to the Act.Price($). Vol.Err.(%) = (Calc.Vol.($) - Act.Vol.($))/Act.Vol.($).

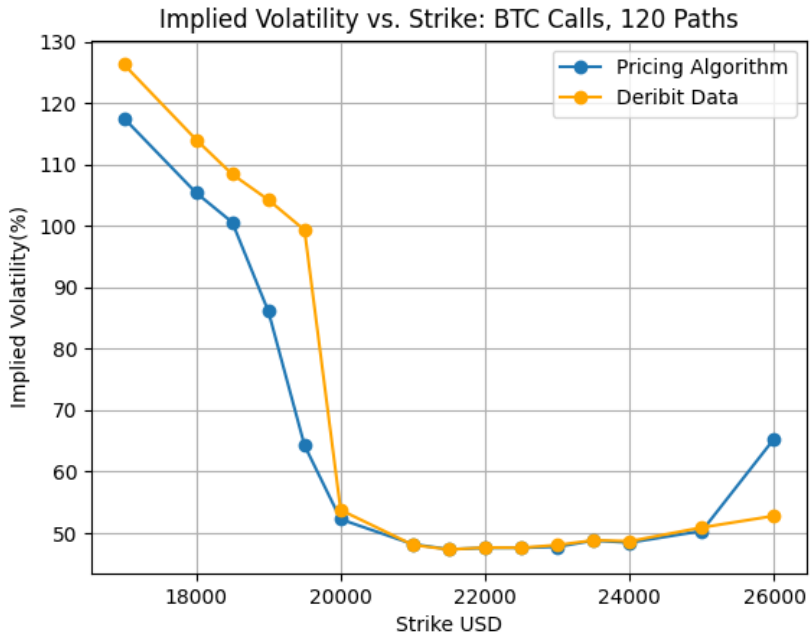

Fig. 5.1: The volatility smile of the Bitcoin call options. Initial price = \$21282.6914, time to expiration = 19 days, risk-free rate = 2%, modeled using 120 sample paths, grid dimensions: $K = 25$, $N + 1 = 20$. The curve of the pricing algorithm is presented in blue. Deribit's data (actual pricing) is presented in orange.

Pricing Bitcoin puts on 120 paths

Strike($)	Calc.Price($)	Act.Price($)	Price Err.(%)	Calc.Vol.(%)	Act.Vol.(%)	Vol.Err.(%)
17000	0.000727	0.004000	-81.8250	197.02	66.32	197.0748
18000	0.002912	0.006000	-51.4667	74.62	58.39	27.7959
19000	0.011154	0.011000	1.4000	53.66	53.73	-0.1303
20000	0.025728	0.020000	28.6400	48.18	49.56	-2.7845
21000	0.046182	0.037000	24.8162	47.78	47.82	-0.0836
22000	0.073145	0.064500	13.4031	50.86	49.02	3.7536
23000	0.106235	0.185000	-42.5757	65.65	150.22	-56.2974

Tab. 4: Initial price = $21282.6914, time to expiration = 19 days, risk-free rate = 2%, modeled using 120 sample paths, grid dimensions: $K = 25$, $N + 1 = 20$. Strike($) = the strike price for the European put option. Calc.Price($) = the calculated price of a European put option as a fraction of the initial price using the pricing algorithm. Act.Price($) = the actual European put option price as a fraction of the initial price from Deribit. Price Err.(%) = (Calc.Price($) - Act.Price($))/Act.Price($). Calc.Vol.(%) = the calculated implied volatility using an interpolation method on the Deribit data where the put price is equal to the Calc.Price($). Act.Vol(%) = the actual implied volatility from Deribit which they calculate using the Black-Scholes equation with a call price equal to the Act.Price($). Vol.Err.(%) = (Calc.Vol.($) - Act.Vol.($))/Act.Vol.($).

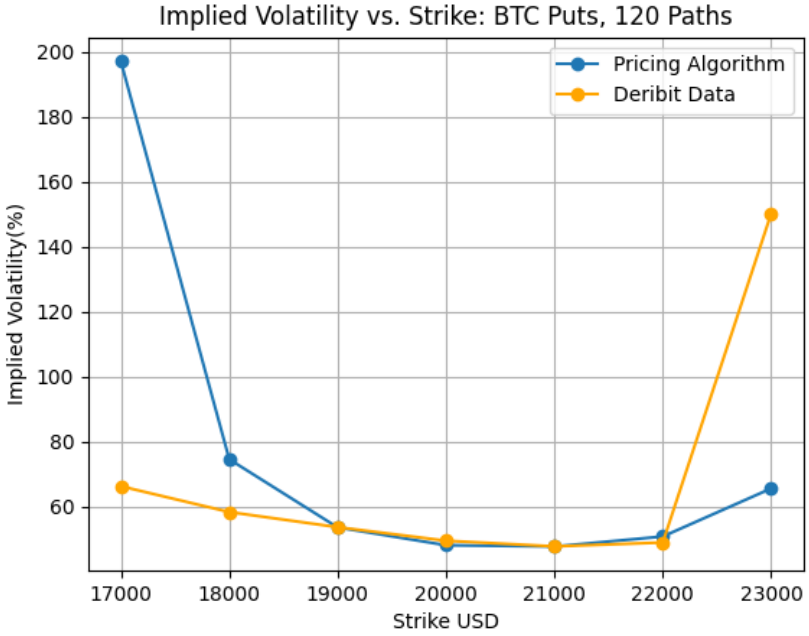

Fig. 5.2: The volatility smile of the Bitcoin put options. Initial price = $21282.6914, time to expiration = 19 days, risk-free rate = 2%, modeled using 120 sample paths, grid dimensions: $K = 25$, $N + 1 = 20$. The curve of the pricing algorithm is presented in blue. Deribit's data (actual pricing) is presented in orange.

Pricing Ethereum calls on 90 paths

Strike($)	Calc.Price($)	Act.Price($)	Price Err.(%)	Calc.Vol.(%)	Act.Vol.(%)	Vol.Err.(%)
1400	0.155646	0.250000	-37.7416	127.52	207.58	-38.5683
1450	0.132590	0.228000	-41.8465	105.50	200.05	-47.2632
1500	0.111634	0.139000	-19.6878	89.25	111.19	-19.732
1600	0.074812	0.072000	3.9056	72.54	72.08	0.6382
1700	0.049522	0.045500	8.8396	71.90	72.00	-0.1389
1800	0.030086	0.027500	9.4036	72.14	72.23	-0.1246
1900	0.016913	0.016500	2.5030	73.32	72.42	-0.1362
2000	0.008142	0.009500	-14.2947	73.07	74.11	-1.4033
2100	0.003787	0.005000	-24.2600	77.40	73.67	5.0631
2200	0.000997	0.003500	-71.5143	90.02	77.82	15.6772
2300	0.000111	0.002000	-94.4500	111.53	78.63	41.8415
2400	0.000096	0.001500	-93.6000	111.99	82.64	35.5155

Tab. 5: Initial price = \$1627.9680, time to expiration = 19 days, risk-free rate = 2%, modeled using 90 sample paths, grid dimensions: $K = 25$, $N + 1 = 20$. Strike($) = the strike price for the European call option. Calc.Price($) = the calculated price of a European call option as a fraction of the initial price using the pricing algorithm. Act.Price($) = the actual European call option price as a fraction of the initial price from Deribit. Price Err.(%) = (Calc.Price($) - Act.Price($))/Act.Price($). Calc.Vol.(%) = the calculated implied volatility using an interpolation method on the Deribit data where the put price is equal to the Calc.Price($). Act.Vol(%) = the actual implied volatility from Deribit which they calculate using the Black-Scholes equation with a call price equal to the Act.Price($). Vol.Err.(%) = (Calc.Vol.($) - Act.Vol.($))/Act.Vol.($).

5.2 Ethereum Results

Historical Ethereum price data ranging from 2017-11-09 to 2022-11-05, was taken as the basis for our sample-path construction. The initial price is set to $B_0 = 1627.9680$. At the time of writing, the implied volatility of an at-the-money month long Ethereum option on Deribit is approximately 72.00%. We take 90 sample paths, set the time to expiration $T = 19$ days, use a risk free-rate of $r_0 = 0.02$, and define our grid dimensions with $K = 25$ Ethereum price rows and $N + 1 = 20$ time steps.

Tab. 5 and Tab. 6 show the results of pricing European call and put options on Ethereum, respectively. The calculated price (Calc.Price) and the calculated volatility (Calc.Vol.) are the results of the pricing algorithm presented in this paper. The actual price (Act.Price) and the actual volatility (Act.Vol.) come from the exchange Deribit and reflect actual trade numbers for 20 day European calls and puts. Fig. 5.3 and Fig. 5.4 show the volatility smile for European call and put options on Ethereum, respectively. In these figures, the result of the pricing algorithm presented here is shown in blue, while the Deribit exchange data is presented in orange.

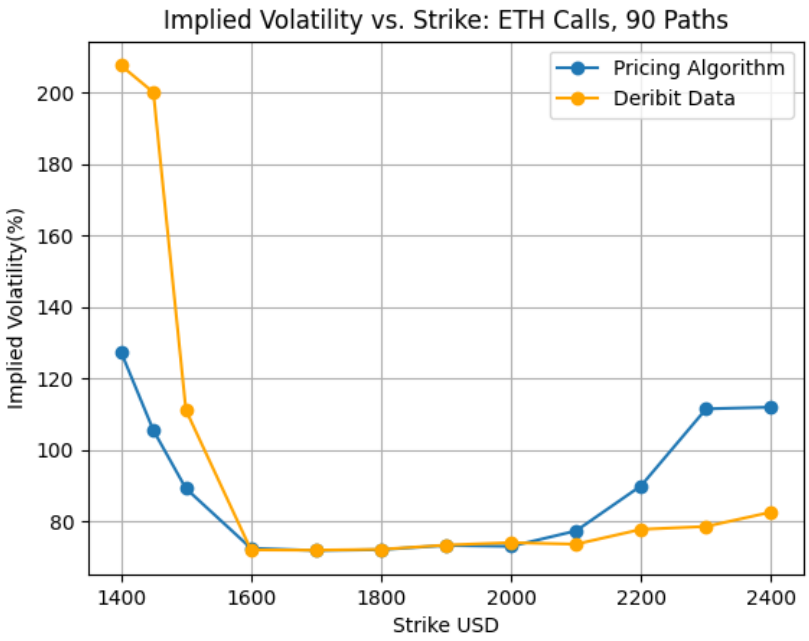

Fig. 5.3: The volatility smile of the Ethereum call options. Initial price = \$1627.9680, time to expiration = 19 days, risk-free rate = 2%, modeled using 90 sample paths, grid dimensions: $K = 25$, $N + 1 = 20$. The curve of the pricing algorithm is presented in blue. Deribit's data (actual pricing) is presented in orange.

Pricing Ethereum puts on 90 paths

Strike($)	Calc.Price($)	Act.Price($)	Price Err.(%)	Calc.Vol.(%)	Act.Vol.(%)	Vol.Err.(%)
900	0.000575	0.001000	-42.5000	126.75	119.11	6.4142
1000	0.001034	0.001500	-31.0667	118.36	106.02	11.6393
1150	0.003710	0.003000	23.6667	85.72	89.08	-3.7719
1200	0.005361	0.004000	34.0250	81.15	84.64	-4.1233
1250	0.007807	0.005500	41.9455	78.68	89.90	-2.7441
1300	0.012162	0.008000	52.0250	76.14	78.55	-3.0681
1350	0.017995	0.011500	56.4783	74.49	76.44	-2.5510
1400	0.025987	0.016500	57.4970	72.86	74.89	-2.7106
1450	0.036017	0.023000	56.5957	72.38	73.26	-1.2012
1500	0.047940	0.032000	49.8125	71.86	72.56	-0.9647
1600	0.076561	0.057000	34.3175	71.20	71.54	-0.4753
1700	0.112852	0.093000	21.3462	77.14	72.47	6.4440
1800	0.153141	0.248000	-38.2496	97.19	200.26	-51.4681

Tab. 6: Initial price = \$1627.9680, time to expiration = 19 days, risk-free rate = 2%, modeled using 90 sample paths, grid dimensions: $K = 25$, $N + 1 = 20$. Strike($) = the strike price for the European put option. Calc.Price($) = the calculated price of a European put option as a fraction of the initial price using the pricing algorithm. Act.Price($) = the actual European put option price as a fraction of the initial price from Deribit. Price Err.(%) = (Calc.Price($) - Act.Price($))/Act.Price($). Calc.Vol.(%) = the calculated implied volatility using an interpolation method on the Deribit data where the put price is equal to the Calc.Price($). Act.Vol(%) = the actual implied volatility from Deribit which they calculate using the Black-Scholes equation with a call price equal to the Act.Price($). Vol.Err.(%) = (Calc.Vol.($) - Act.Vol.($))/Act.Vol.($).

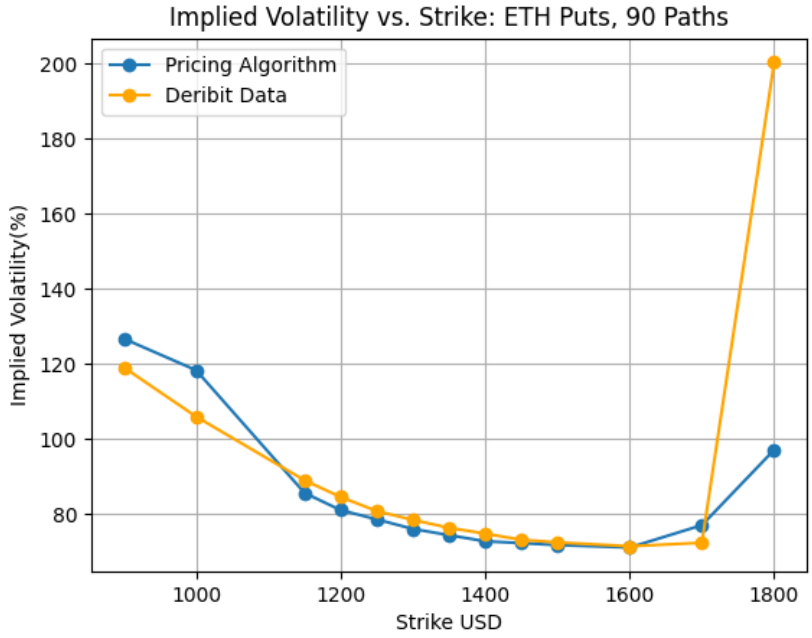

Fig. 5.4: The volatility smile of the Ethereum put options. Initial price = \$1627.9680, time to expiration = 19 days, risk-free rate = 2%, modeled using 90 sample paths, grid dimensions: $K = 25$, $N + 1 = 20$. The curve of the pricing algorithm is presented in blue. Deribit's data (actual pricing) is presented in orange.

5.3 Discussion of Results

One of the difficulties with this case study was the reliability of exchange data. Cryptocurrency options tend to have limited volume compared to stock options which are more mainstream. This can lead to outliers in the implied volatilities data. For example, in Tab. 5.4 notice the jump in volatility between a put with strike 1700 and a put with strike 1800. From Fig. 5.4 it is clear this data point is priced incorrectly on the exchange. It turns out this 1800 strike only had a trade volume of 1, making its reliability as a valid data point questionable. We decided to leave this data point in to highlight how cryptocurrency exchange data which is deep in-the-money and deep out-of-the-money can be unreliable due to a lack of trade volume. As more people trade on these cryptocurrency exchanges, we expect this unreliability to decrease.

Looking at Tab. 3, we see that the error between the calculated prices and actual prices increases as we move away from the at-the-money option. However, the error between the calculated volatility and the actual volatility is not nearly as high. The volatility error is only high at the ends of the volatility smile and for strike 19500, which upon further inspection had low trade volume at the time of data collection. Other than that, the volatilities seem reasonable. Furthermore, we find similar volatility error trends for Bitcoin puts as well as Ethereum calls and puts. One further observation is that the pricing algorithm seems to underestimate volatility for low strikes and overestimate volatility for high strikes when pricing calls. Conversely, the pricing algorithm overestimates volatility for low strikes and underestimates volatility for high strikes when pricing puts.

While the Deribit data does reflect real world trades, the volume of investors that trade these options on Deribit is at most in the thousands of people. With a lack of open interest and trades on options with strikes below the initial price, we question the reliability of the lower and upper end strike data points for Deribit. As cryptocurrency options trading expands to more investors, we expect to see more reliable data in the future which will allow us to accurately price the calls and puts of cryptocurrency options.

6 Conclusion

We applied a pricing algorithm of Ryabchenko, Sarykalin, and Uryasev et al.[12] to price European calls on Cryptocurrency. The pricing algorithm is a minimization of the expected quadratic error subject to constraints. Using the hedging strategy modeled by two matrices which represent the positions of Cryptocurrency and a bond within a portfolio, we were able to viably recreate the theory of our predecessors to price European options on Bitcoin and Ethereum.

We produced volatility smiles that are reasonable compared to the actual Deribit exchange data. Like our predecessors, we believe there is a lot of potential in cryptocurrency options pricing. We are

interested in seeing how the pricing algorithm improves as the volume of cryptocurrency options that are traded increases. Implementing new cryptocurrencies into our algorithm as well as repeating the study with higher trade volume are all subsequent papers that interest us. Currently the field of crypto options pricing is expanding at a phenomenal rate which will lead to new subsequent methodologies that might one day be implemented into everyday trading practices.

7 References

[1] Divya Aggarwal, *Do bitcoins follow a random walk model?*, Research in Economics **73** (2019), no. 1, 15–22.

[2] Dirk G Baur and Thomas Dimpfl, *Realized bitcoin volatility*, SSRN **2949754** (2017), 1–26.

[3] Dirk G Baur, Lai T Hoang, and Md Zakir Hossain, *Is bitcoin a hedge? how extreme volatility can destroy the hedge property*, Finance Research Letters (2022), 102655.

[4] Dirk G Baur, Adrian D Lee, and KiHoon Hong, *Bitcoin: currency or investment?*, Available at SSRN **2561183** (2015).

[5] Vitalik Buterin et al., *A next-generation smart contract and decentralized application platform*, white paper **3** (2014), no. 37, 2–1.

[6] Melanie Cao and Batur Celik, *Valuation of bitcoin options*, Journal of Futures Markets **41** (2021), no. 7, 1007–1026.

[7] Eng-Tuck Cheah and John Fry, *Speculative bubbles in bitcoin markets? an empirical investigation into the fundamental value of bitcoin*, Economics letters **130** (2015), 32–36.

[8] Shaen Corbet, Brian Lucey, and Larisa Yarovaya, *Datestamping the bitcoin and ethereum bubbles*, Finance Research Letters **26** (2018), 81–88.

[9] Florian Glaser, Kai Zimmermann, Martin Haferkorn, Moritz Christian Weber, and Michael Siering, *Bitcoin-asset or currency? revealing users' hidden intentions*, Revealing Users' Hidden Intentions (April 15, 2014). ECIS (2014).

[10] Akanksha Jalan, Roman Matkovskyy, and Saqib Aziz, *The bitcoin options market: A first look at pricing and risk*, Applied Economics **53** (2021), no. 17, 2026–2041.

[11] Satoshi Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, Decentralized Business Review (2008), 21260.

[12] Valeriy Ryabchenko, Sergey Sarykalin, and Stan Uryasev, *Pricing european options by numerical replication: quadratic programming with constraints*, Asia-Pacific Financial Markets **11** (2004), no. 3, 301–333.

[13] David Yermack, *Is bitcoin a real currency? an economic appraisal*, Handbook of digital currency, Elsevier, 2015, pp. 31–43.

8 Appendix

optimization.py creates the hedging strategy and optimizes the minimization problem using GUROBI:

```
import numpy as np
import math
import statistics as stat
from scipy.stats import norm
import random as rand
import gurobipy as gp
from gurobipy import GRB
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

#vDeribitData = np.loadtxt('BTC-3JUN22-export.csv',delimiter=',',skiprows = 19, usecols = (1,7,8))
vDeribitData = np.loadtxt('BTC-25NOV22-export.csv',delimiter=',',skiprows = 14, usecols = (1,7,8))
vDeribitData = vDeribitData[vDeribitData[:, 0].argsort()]
vDeribitData = np.delete(vDeribitData, -1, 0)

amendedX = np.append(vDeribitData.T[1],0)
amendedY = np.append(vDeribitData.T[2],115)
f = interp1d(amendedX, amendedY, kind='quadratic')
x = np.linspace(min(amendedX), max(amendedX), num=10**6, endpoint=True)

def optimization(nStrike):
    # Load in BTC prices data
    vBTCPrices = np.loadtxt('BTC-USD.csv',delimiter=',',skiprows = 1,usecols = 5)

    # Set parameters
    nPaths = 120
    nSteps = 25
    r = 0.02
```

```
    nInitialPrice = vBTCPrices[-1]
    nExpiry = 19
    nTimeSteps = nExpiry + 1

    # Divide the BTC prices into non-overlapping sample paths each with (nTimeSteps+1) steps
    mnDividedPaths = vBTCPrices[:(nTimeSteps)*math.trunc(len(vBTCPrices)/(nTimeSteps))].reshape(-1,(nTimeSteps))

    # Randomly select nPaths and normalize the sample paths to the same initial BTC price
    rand.seed(10)
    vRandomPathIndicies = rand.sample([*range(len(mnDividedPaths))],k=nPaths)
    mnPricePaths = []
    for i in range(nPaths):
        nNormCoef = mnDividedPaths[vRandomPathIndicies[i],0]/nInitialPrice
        mnPricePaths.append(mnDividedPaths[vRandomPathIndicies[i]]/nNormCoef)

    print("INITIAL BTC PRICE:",mnPricePaths[0][0])

    ### Perform volatility massaging of sample-paths.
    # Calculate the daily returns across each sample-path
    mnPricePathReturns = []
    for i in range(nPaths):
        tempList = []
        for j in range(nTimeSteps-1):
            tempList.append((mnPricePaths[i][j+1]/mnPricePaths[i][j])-1)
        mnPricePathReturns.append(tempList)
    nMeanReturn = np.mean(mnPricePathReturns)

    # Calculate the implied volatility of the at-the-money option
    implied_vol = 0.4750/math.sqrt(365)
    print("IMPLIED VOL of ATM:",implied_vol*math.sqrt(365))

    # Modify the returns
    mnReturnsMas = []
    for i in range(len(mnPricePaths)):
        mnReturnsMas.append((implied_vol/np.std(mnPricePathReturns[i]))*(mnPricePathReturns[i]-nMeanReturn))

    # Convert returns back into prices
    mnPricePathMas = []
    for i in range(nPaths):
```

```
68          tempList = [mnPricePaths[i][0]]
69          prevPrice = mnPricePaths[i][0]
70          for j in range(nTimeSteps-1):
71              prevPrice = (mnReturnsMas[i][j]+1)*prevPrice
72              tempList.append(prevPrice)
73          mnPricePathMas.append(tempList)
74      mnPricePathMas = np.array(mnPricePathMas)
75
76      # Build the price vector for the grid
77      nMinGridPrice = math.trunc(min([min(sublist) for sublist in mnPricePathMas])
    )-1
78      nMaxGridPrice = math.ceil(max([max(sublist) for sublist in mnPricePathMas]))
    +1
79      print("MIN GRID VALUE:",nMinGridPrice)
80      print("MAX GRID VALUE:",nMaxGridPrice)
81      # We require the prices on the grid to be equally distanced in logarithmic
    scale
82      nLogStepCoef = math.log((nMaxGridPrice)/(nMinGridPrice))/(nSteps-1)
83      vPriceGrid = []
84      for i in range(nSteps):
85          vPriceGrid.append(math.exp(math.log((nMinGridPrice))+i*nLogStepCoef))
86      print(vPriceGrid)
87
88      # Initialize Gurobi model
89      m = gp.Model("NewModel")
90
91      # Build the grid position matricies
92      mnStockPositionGrid = m.addMVar(shape=(nSteps,nTimeSteps), lb=float('-inf'),
        name="mnStockPositionGrid")
93      mnBondPositionGrid = m.addMVar(shape=(nSteps,nTimeSteps), lb=float('-inf'),
    name="mnBondPositionGrid")
94
95      # Build the linear approximation positions for each path
96      mnStockPositionsPath = []
97      mnBondPositionsPath = []
98      for i in range(nPaths):
99          vTempStockPositions = []
100         vTempBondPositions = []
101         for j in range(nTimeSteps):
102             for k in range(nSteps-1):
103                 if mnPricePathMas[i,j] >= vPriceGrid[k] and mnPricePathMas[i,j]
    < vPriceGrid[k+1]:
104                     kTilda = k
```

```
                    break
            alpha = math.log(mnPricePathMas[i,j]/vPriceGrid[kTilda])/math.log(
    vPriceGrid[kTilda+1]/vPriceGrid[kTilda])
            u = alpha*mnStockPositionGrid[kTilda+1,j]+(1-alpha)*
    mnStockPositionGrid[kTilda,j]
            v = alpha*mnBondPositionGrid[kTilda+1,j]+(1-alpha)*
    mnBondPositionGrid[kTilda,j]
            vTempStockPositions.append(u)
            vTempBondPositions.append(v)
        mnStockPositionsPath.append(vTempStockPositions)
        mnBondPositionsPath.append(vTempBondPositions)
    mnStockPositionsPath = np.array(mnStockPositionsPath)
    mnBondPositionsPath = np.array(mnBondPositionsPath)

    # Construct the average quadratic error
    objSum = 0
    constrSum = 0
    for i in range(nPaths):
        for j in range(1,nTimeSteps):
            a = mnStockPositionsPath[i,j]*mnPricePathMas[i,j]+
    mnBondPositionsPath[i,j]-(mnStockPositionsPath[i,j-1]*mnPricePathMas[i,j]+(1+
    r/365)*mnBondPositionsPath[i,j-1])
            objSum += (a*math.exp(-r*j/365))**2
            constrSum += a*math.exp(-r*j/365)
    # Define the objective for the optimization problem
    m.setObjective(objSum/nPaths, GRB.MINIMIZE)
    # We require the external financing over all paths to equal zero.
    m.addConstr(constrSum/nPaths==0)
    # We also require the value of the portfolio to equal the option payoff at
    expiration.
    for i in range(nSteps):
        call = mnStockPositionGrid[i,-1]*vPriceGrid[i]+mnBondPositionGrid[i,-1]
        m.addConstr(call==max(0,vPriceGrid[i]-nStrike))


    ### Define CALL constraints

    # Immediate exercise constraint
    for i in range(nSteps):
        for j in range(nTimeSteps-1):
            call = mnStockPositionGrid[i,j]*vPriceGrid[i]+mnBondPositionGrid[i,j
    ]
```

```
140             m.addConstr(call>=max(0,vPriceGrid[i]-nStrike*math.exp(-r*(nExpiry-j
    )/365)))
141
142     # Option price sensitivity constraint
143     for i in range(nSteps-1):
144         for j in range(nTimeSteps-1):
145             gamma = vPriceGrid[i+1]/vPriceGrid[i]
146             callOne = mnStockPositionGrid[i,j]*vPriceGrid[i]+mnBondPositionGrid[
    i,j]
147             callTwo = mnStockPositionGrid[i+1,j]*vPriceGrid[i+1]+
    mnBondPositionGrid[i+1,j]
148             m.addConstr(callTwo<=gamma*callOne+nStrike*(gamma-1)*math.exp(-r*(
    nExpiry-j)/365))
149             # *** Maybe this should be for all j ***
150
151     # Option vertical monotonicity constraint
152     for i in range(nSteps-1):
153         for j in range(nTimeSteps):
154             gamma = vPriceGrid[i+1]/vPriceGrid[i]
155             callOne = mnStockPositionGrid[i,j]*vPriceGrid[i]+mnBondPositionGrid[
    i,j]
156             callTwo = mnStockPositionGrid[i+1,j]*vPriceGrid[i+1]+
    mnBondPositionGrid[i+1,j]
157             m.addConstr(callTwo/gamma>=callOne)
158
159     # Option horizontal monotonicity constraint
160     for i in range(nSteps):
161         for j in range(nTimeSteps-1):
162             callOne = mnStockPositionGrid[i,j]*vPriceGrid[i]+mnBondPositionGrid[
    i,j]
163             callTwo = mnStockPositionGrid[i,j+1]*vPriceGrid[i]+
    mnBondPositionGrid[i,j+1]
164             m.addConstr(callTwo<=callOne)
165
166     # Option convexity constraint
167     for i in range(nSteps-2):
168         for j in range(nTimeSteps):
169             beta = (vPriceGrid[i+1]-vPriceGrid[i+2])/(vPriceGrid[i]-vPriceGrid[i
    +2])
170             callOne = mnStockPositionGrid[i,j]*vPriceGrid[i]+mnBondPositionGrid[
    i,j]
171             callTwo = mnStockPositionGrid[i+1,j]*vPriceGrid[i+1]+
    mnBondPositionGrid[i+1,j]
```

```
172             callThree = mnStockPositionGrid[i+2,j]*vPriceGrid[i+2]+
    mnBondPositionGrid[i+2,j]
173             m.addConstr(callTwo<=beta*callOne+(1-beta)*callThree)
174
175     # Find where on the price grid the strike is located
176     for i in range(nSteps-1):
177         if vPriceGrid[i]<=nStrike and nStrike<vPriceGrid[i+1]:
178             kHat = i
179     print("kHat: ",kHat)
180
181     # Stock position bounds constrant
182     for i in range(nSteps):
183         for j in range(nTimeSteps):
184             m.addConstr(0<=mnStockPositionGrid[i,j])
185             m.addConstr(mnStockPositionGrid[i,j]<=1)
186
187     # Stock vertical monotonicity constraint
188     for i in range(nSteps-1):
189         for j in range(nTimeSteps):
190             m.addConstr(mnStockPositionGrid[i+1,j]>=mnStockPositionGrid[i,j])
191
192     # Stock horizontal monotonicity constraint
193     for i in range(nSteps):
194         for j in range(nTimeSteps-1):
195             if i>kHat:
196                 m.addConstr(mnStockPositionGrid[i,j]<=mnStockPositionGrid[i,j
    +1])
197             else:
198                 m.addConstr(mnStockPositionGrid[i,j]>=mnStockPositionGrid[i,j
    +1])
199
200     # Stock convexity constraint
201     if kHat<2:
202         for i in range(kHat+1,nSteps-2):
203             betaPlus = (vPriceGrid[i+1]-vPriceGrid[i+2])/(vPriceGrid[i]-
    vPriceGrid[i+2])
204             for j in range(nTimeSteps):
205                 # Only upper
206                 m.addConstr((1-betaPlus)*mnStockPositionGrid[i+2,j]+betaPlus*
    mnStockPositionGrid[i,j]<=mnStockPositionGrid[i+1,j])
207     elif 2<=kHat and kHat<=nSteps-3:
208         for i in range(2,nSteps-2):
```

```
209             betaPlus = (vPriceGrid[i+1]-vPriceGrid[i+2])/(vPriceGrid[i]-
    vPriceGrid[i+2])
210             betaMin = (vPriceGrid[i-1]-vPriceGrid[i])/(vPriceGrid[i-2]-
    vPriceGrid[i])
211             for j in range(nTimeSteps):
212                 # Upper and lower
213                 if i>kHat:
214                     # Upper
215                     m.addConstr((1-betaPlus)*mnStockPositionGrid[i+2,j]+betaPlus
    *mnStockPositionGrid[i,j]<=mnStockPositionGrid[i+1,j])
216                 else:
217                     # lower
218                     m.addConstr((1-betaMin)*mnStockPositionGrid[i-2,j]+betaMin*
    mnStockPositionGrid[i,j]>=mnStockPositionGrid[i-1,j])
219      else:
220          for i in range(2,kHat):
221              betaMin = (vPriceGrid[i-1]-vPriceGrid[i])/(vPriceGrid[i-2]-
    vPriceGrid[i])
222              for j in range(nTimeSteps):
223                  # lower only
224                  m.addConstr((1-betaMin)*mnStockPositionGrid[i-2,j]+betaMin*
    mnStockPositionGrid[i,j]>=mnStockPositionGrid[i-1,j])
225
226
227      # Set GUROBI solver parameters
228      m.setParam('NonConvex', 2)
229      m.setParam('FeasibilityTol', 1e-4)
230      m.setParam('OptimalityTol', 1e-4)
231      m.setParam('NumericFocus', 0)
232      m.setParam('Quad', 1)
233      m.setParam('Method',2) # Use barrier method to solve
234      m.setParam('TimeLimit', 300)
235
236      # Optimize objective subject to constraints
237      m.optimize()
238
239      for i in range(nSteps-1):
240          if nInitialPrice >= vPriceGrid[i] and nInitialPrice < vPriceGrid[i+1]:
241              alpha = math.log(nInitialPrice/vPriceGrid[i])/math.log(vPriceGrid[i
    +1]/vPriceGrid[i])
242              u = alpha*mnStockPositionGrid[i+1,0].X+(1-alpha)*mnStockPositionGrid
    [i,0].X
```

```
            v = alpha*mnBondPositionGrid[i+1,0].X+(1-alpha)*mnBondPositionGrid[i
    ,0].X
            call = u*nInitialPrice+v
            print()
            print("STRIKE:",nStrike)
            print("CALL:",call/nInitialPrice)
            implied_vol = f(call/nInitialPrice)
            return [nStrike,implied_vol,call,u,v]

strikes = []
vols = []
calls = []
u = []
v = []
for strike in vDeribitData.T[0]:
    opt = optimization(strike)
    strikes.append(opt[0])
    vols.append(opt[1])
    calls.append(opt[2])
    u.append(opt[3])
    v.append(opt[4])
print("Call: ",calls)
print()
print("u's: ",u)
print()
print("v's: ",v)
print()
print("IV's: ",vols)


vDeribitData = vDeribitData[vDeribitData[:, 0].argsort()]

plt.plot(strikes,vols,'-o',label='Pricing Algorithm')
plt.plot(vDeribitData.T[0], vDeribitData.T[2], '-o', color='orange', label='
    Deribit Data')
plt.grid()
plt.xlabel('Strike USD')
plt.ylabel('Implied Volatility(%)')
plt.title('Implied Volatility vs. Strike: BTC Calls, 120 Paths')
plt.legend()
plt.show()
```