Pricing and calibration of stochastic models via neural networks

by

Mehdi Tomas (CID: 01390785)

Department of Mathematics
Imperial College London
London SW7 2AZ
United Kingdom

Thesis submitted as part of the requirements for the award of the MSc in Mathematics and Finance, Imperial College London, 2017-2018

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Signature and date:

Mehdi Tomas September 11, 2018

Acknowledgements

To my mother, for her love and unwavering support.

I am grateful to Dr. Blanka Horvath for her faith, patience, support and for encouraging me to pursue a PhD and Aitor Muguruza for his help and advice throughout the year.

Contents

1 Introduction **6**

1.1 Background 6

1.2 Litterature review 6

1.3 Outline 8

2 Stochastic models **8**

2.1 Implied volatility and the implied volatility surface 8

2.2 The Heston, Bergomi and rough Bergomi models 9

3 Numerical computation of implied volatilities **11**

3.1 Computing option prices using the Monte-Carlo algorithm 11

3.2 Reducing computation time with variance reduction 12

3.3 Computing implied volatilities from option prices 13

4 Neural networks for function approximation **13**

4.1 Feedforward neural networks 13

4.2 Approximation capabilities of neural networks 13

4.3 Application to option pricing 15

5 Optimization procedure **17**

5.1 Problem setting 17

5.2 Line search methods 17

5.3 Line search algorithms for neural networks 19

5.3.1 Gradient Descent 20

5.3.2 Stochastic Gradient Descent 20

5.3.3 Adam 21

6 Results **21**

6.1 Implementation and experimental setting 21

6.1.1 Generating surfaces using Monte Carlo 21

6.1.2 Training neural networks using generated surfaces 22

6.1.3 Using neural networks to calibrate 23

6.2 Using the code 23

6.2.1 Computing Implied volatility surfaces using Monte Carlo 24

6.2.2 Creating neural network to map parameters to implied volatility surfaces . 24

6.2.3 Computing implied volatility surfaces using generated models 24

6.2.4 Calibrating model to given implied volatility surface using neural networks 24

6.2.5 Training procedure for generator 24
6.3 Training procedure for calibrator 25
6.4 Approximation 26
6.5 Calibration 26

Conclusion **83**

1 Introduction

> Financial engineering is at the stage of Ptolemea's epicycles before Kepler's ellipses. After so much twisting and tweaking (calibration is the politically correct word for it), epicycles were more precise than ellipses... But of course, this was no theory.
>
> *Jean-Philippe Bouchaud*

1.1 Background

From the original Black-Scholes model ([1]) to the rough Bergomi model ([2]), stochastic models are powerful analytical tools to compute option prices and Greeks. However stochastic models are parametric. To use them, we need to find appropriate parameters. Assuming the market follows the model, we can use market prices to choose our parameters.

Therefore we need to find a parameter combination such that option prices induced by the model with those parameters are consistent. This is time-consuming since stochastic models may not have closed-form solutions and we then need to use numerical methods to estimate option prices.

Consequently, we are confronted with two problems. Firstly, we need a fast algorithm to compute option prices, given a model and its parameters. Secondly, we require an algorithm that outputs model parameters consistent with market prices.

1.2 Litterature review

Surprisingly, the oldest example of using neural networks for pricing and hedging derivatives dates back to 1994 by Hutchinson, Lo & Poggio ([3]). Assuming that the underlying follows a Black-Scholes model, the authors simulate asset paths and option prices along that path. At each step, the network takes the value of the underlying, strike price and time to maturity as inputs to predict the price of a Call option ([3], p.861-862). Since the authors seek a non-parametric estimation of option prices, their goal is not to map model parameters to surfaces however their approach is applicable to our problem.

Unfortunately, as the authors point out, this approach requires many paths for the network to learn the representation ([3], p.852). For our purposes, this approach could be extended to *some* stochastic volatility models. We would simply need to add volatility as an input. But as is, it could not be extended to fractional volatility models, such as the rough Bergomi model. Indeed in this instance the volatility process is no longer Markovian. To adapt this approach to non-Markovian

processes, previous values should be fed as an input as well. Since a fractional process has a power law kernel, the contribution from previous terms progressively vanishes. We may obtain a good enough approximation by taking a length proportional to the decay of the kernel. Nevertheless, the amount of data necessary for the network to learn the representation would again increase, and so would the model complexity.

Additionally, the model is restricted to learning the price of a single contract. Therefore it cannot be used to price combinations of options or exotics. On the other hand, we can use parametric models consistent with vanilla options to compute prices for other options.

More recently, De Spiegeleer et al ([4]) use Gaussian processes to approximate prices and hedges of options under the Heston model ([4], p.2-3). Their approach allows for the computation of exotic option prices, such as American and Barrier options. The methodology is fast, precise and transferable, with speed-ups up to 10,000 times ([4], p.13) and error rates of around 100 basis points for European options ([4], p.10).

In comparison, our approach yields a lower average error (40 and basis points compared to 436 ([4], p.)) and faster computation times for the Heston model under all configurations, assuming the Fast Fourier Transform methods used for the Heston model are comparable (speed-ups found to be about 650, using 1000 samples, compared to at most 40).

Finally, Hernandez ([5]) focuses on the calibration problem in the case of foreign exchange swaptions under the Hull-White model ([6]). However the author's approach is different and he seeks to directly map implied volatility surfaces to parameters. As such, a neural network is trained with surfaces as inputs and parameters as outputs.

Interestingly, this is the first approach we tried out. Like the author, we tried feed-forward networks and more convolutional neural networks to capture relationships between close points in the surface. However those approaches failed. We think this happened because the Heston, Bergomi and rough Bergomi stochastic models are over-parametrised. Indeed, different combinations of parameters may lead to the very similar surfaces. This prevented the model from learning the map, since a given surface might very well have come from two different parameter sets. But since network parameters are updated according to the error gradient, this meant that estimated parameters which gave surfaces very close to the input surface but were far from the original parameters were penalised. We believe this problem was not encountered by Hernandez ([5]) because the Hull-White model has only two parameters, the third being determined by the yield curve. Furthermore, those parameters have two very different effects on swaption prices. This reduces the chance of finding two distinct parameter combinations which yield similar prices.

This is what led us to consider first approximating option prices. Given a fast approximation of the surface we could try out many different parameters until we found a combination with a small error of the input surface. Such a combination might still be incorrect but at least we would

be confident in the similarity between the two surfaces.

1.3 Outline

We do so by approximating the mapping of option prices and using this fast map to calibration directly on option prices.

Neural networks can approximate a large class of functions using fast operations. For a given stochastic volatility model, we will find the corresponding transformations to map model parameters to option prices. Since option prices are dependent on the current underlying level and forward, price, we will instead model the map from parameters to implied volatilities. We detail in section 2 the different stochastic models used and in section 3 how we compute implied volatilities from those models.

To train the neural network, we generate many parameters and corresponding implied volatility surfaces. We can find a suitable transformation with back-propagation and line search algorithms to minimise the error. Back-propagation and line search algorithms are presented in section 5. We can then use this fast map to search for a combination of parameters that minimises the error between a surface given by the market and a surface given by a stochastic model. Finally,we present results in section 6.

2 Stochastic models

2.1 Implied volatility and the implied volatility surface

Definition 2.1 (Black Scholes model)**.** Let $(\Omega, \mathcal{F}, \mathbb{P})$ a probability space, then the Black-Scholes model with constant drift and volatility specifies the dynamics of the asset price S with the following stochastic differential equation under the physical measure $\mathbb{P}$:

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

.

Proposition 2.2 (Price of a Call option under Black Scholes ([7], p.11))**.** *Let $(\Omega, \mathcal{F}, \mathbb{P})$ a probability space and S a Black-Scholes stochastic process. Note F_T the price of the forward of maturity T on S. Then the price of a Call option with strike K, time to maturity T and current asset price S_t is:*

$$C_{BS}(S_0, K, \sigma, T) = F_T(N(d_1) - e^k N(d_2))$$

. *Where we have used the following notations:*

$$\begin{aligned} k &:= \log(K/F_T), \\ d_1 &:= \frac{1}{\sigma\sqrt{T}}(k + \frac{\sigma^2 T}{2}), \\ d_2 &:= d_1 - \sigma\sqrt{T} \end{aligned}$$

.

Proof. Using the underlying distribution and Girsanov's Theorem, one can compute the stochastic differential equation satisfied by S under the risk-neutral measure and solve it. Then, taking the discounted conditional expectation of the payoff under the risk-neutral measure we obtain the Call option price. By the same argument, the forward price is equal to $S_0 e^{rT}$, and substituting back we obtain the result. □

Remark 2.3. Even if the interest rate is not deterministic, the above formula holds. Since forward prices are given by the market, the option price depends solely on the volatility σ.

Definition 2.4. (Implied volatility) The implied volatility $\sigma_{BS}(K, T; S_0)$ is such that, given the price of a Call option $C(S_0, K, T)$ and of a forward F_T of maturity T, the following equality holds:

$$C(S_0, K, T) = C_{BS}(S_0, K, \sigma_{BS}(K, T; S_0), T)$$

.

Remark 2.5. While implied volatilities are constant under the Black-Scholes model, using market forward prices at a given tenor T and option prices for strike K and tenor T, we may recover the implied volatility $\sigma_{BS}(K, T; S_0)$ for different K and T. They would then notice that they are different, leading us to talk about a volatility surface.

Definition 2.6 (Implied volatility surface)**.** Given strikes $(K_i)_{1\leq i\leq I}$ and maturities $(T_j)_{1\leq j\leq J}$ and market prices of Call options $(C(S_0, K_i, T_j))_{i,j}$, the implied volatility surface is $(\sigma_{BS}(K_i, T_j; S_0))_{i,j}$.

2.2 The Heston, Bergomi and rough Bergomi models

Definition 2.7 (The Heston ([8]) model)**.** Let $(\Omega, \mathcal{F}, \mathbb{P})$ a probability space, where the dynamics for underlying and volatility under the physical measure $\mathbb{P}$ follow the Heston model:

$$dS_t = \mu_t S_t dt + \sqrt{v_t} S_t dZ_t^1, \tag{2.1}$$

$$dv_t = -\kappa(v_t - \theta)dt + \nu\sqrt{v_t} dZ_t^2 \tag{2.2}$$

. Where we note Z^1 and Z^2 two correlated Brownian motions, with correlation ρ.

Proposition 2.8 (Price for a European Call option under Heston ([8]) dynamics)**.** *Noting K the strike, T the tenor, we note for convenience $x := \log F_t^T/K$, $\tau = T - t$ the time to maturity, then the option price is:*

$$C(x, v, \tau) = K(e^x P_1(x, v, \tau) - P_2(x, v, \tau))$$

.

Where P_j for $j = 1, 2$ is equal to:

$$P_j(x, v, \tau) = \frac{1}{2} + \frac{1}{\pi}\int_0^\pi Re\frac{e^{-iu\log K} f_j(x, v, T, u)}{iu}du$$

.

Where f_j is the characteristic function of the probabilities that the option expires in-the-money and out-of-the money. Their expression is given in full in [8] (p.330-331).

Proof. By looking for a solution similar in form to the Black Scholes solution, we can find the PDE solved by the Call option under Heston dynamics. Once this PDE is found we can substitute the Call option price process by 2.8 to obtain a partial differential equation solved by P_1 and P_2 ([8], p.330).
Using the linearity in equation 2.8, we can search for specific solutions where the P_1 and P_2 solve the partial differential equation independently ([8], p.330). To solve those partial differential equations we can guess a specific form of the characteristic function and search for the proper coefficients ([8], p.440-442). □

Remark 2.9. Out of the three models studied, only the Heston model presents an analytical formula for European option prices. This allows us to compute option prices numerically.

Definition 2.10 (Stochastic exponential)**.** Given a stochastic process X in L^2 (check exactly definition), the stochastic exponential $\mathcal{E}_X$ is the unique process starting at 1 such that:

$$d\mathcal{E}_X = \mathcal{E}_X dX$$

. And we have the explicit formula: $\mathcal{E}_X(t) := \exp(X_t - \frac{[X]_t}{2})$.

Proof. To show that the stochastic process defined satisfies the stochastic differential equation we can apply Ito's lemma using the continuity of the exponential. The covariation terms cancel out, leaving only the first order term. To show uniqueness of the stochastic exponential one proceeds as with the deterministic exponential and considers the product of one process with the exponential of the $-X$. Using Ito's formula we can show the product is constant in time, starting at 1. Therefore the two processes are equal. □

Definition 2.11 (The Bergomi ([9]) model)**.** The Bergomi ([9]) model, follows the dynamics:

$$S_t = S_0\mathcal{E}(\int_0^t \sqrt{v_u}dZ_u), \tag{2.3}$$

$$v_t = \xi_0(t)\mathcal{E}(\nu W_t) \tag{2.4}$$

. Where $dZ_t = \rho dW_t + (1-\rho^2)dW_t^T$.

Definition 2.12 (The rough Bergomi (Bayer, Friz & Gatheral [2] p. 890) model)**.** Let $(\Omega, \mathcal{F}, \mathbb{P})$ a probability space, then the dynamics for the asset price and volatility under the physical measure under the rough Bergomi model, as introduced in Bayer, Friz & Gatheral ([2] p. 890), are as follows:

$$S_t = S_0 \mathcal{E}(\int_0^t \sqrt{v_u} dZ_u), \tag{2.5}$$

$$v_t = \xi_0(t)\mathcal{E}(\nu\sqrt{2H}\int_0^t \frac{dW_u}{(t-u)^{1/2-H}}) \tag{2.6}$$

Where Z has the decomposition : $dZ_t = \rho dW_t + \sqrt{1-\rho^2}dW_t^T$.

Remark 2.13. When the Hurst parameter H is equal to 0.5, then the rough Bergomi model reduces to the Bergomi model. The Hurst parameter is introduced to reproduce the observed relationship in realised volatility:

$$\log \sigma_{t+\Delta} - \log \sigma_t = \nu(W_{t+\Delta}^H - W_t^H)$$

.

The relationship across assets was studied by Gatheral, Jaisson & Rosenbaum in [10]. It holds over a set of futures and indices ([10] p. 936-938). The Hurst parameter found varied between 0.05 and 0.2, with some variability, as the authors note by estimating it using data from before and after the financial crisis of 2008 ([10] p.949).

3 Numerical computation of implied volatilities

3.1 Computing option prices using the Monte-Carlo algorithm

Theorem 3.1 (Strong law of large numbers)**.** *Let $(X_n)_{n\in\mathbb{N}}$ be a sequence of independent and identically distributed real-valued random variables defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Assume that $X_1 \in L^1$, then let S_N be the mean of $(X_n)_{n\in\mathbb{N}}$ up to N:*

$$S_N = \frac{1}{N}\sum_{n=1}^{N} X_n$$

.Then, $\mathbb{P}$ - almost surely,

$$S_N \to \mathbb{E}[X_1]$$

.

Proof. One is able to show the pointwise convergence of the characteristic funtion of S_N to the constant characteristic function of $\mathbb{E}[X_1]$, thereby, since all functions in the sequence are continuous, and according to Levy's Theorem, showing that S_N converges in probability to $\mathbb{E}[X_1]$. However convergence in probability or convergence almost surely are equivalent if the limit is a constant, hence the result. □

3.2 Reducing computation time with variance reduction

Two techniques are used to reduce computation time. The first is variance reduction using antithetic variates and the second is Romano-Touzi trick which allows us to simulate one random variable instead of two.

Definition 3.2 (Antithetic variate)**.** Let $(\Omega, \mathcal{F}, \mathbb{P})$ a probability space and $f : \mathbb{R} \mapsto \mathbb{R}$ a measurable function, X a real-valued random variable. Let $(X_n)_{1 \leq n \leq N}$ be identically, independently drawn samples from the probability measure induced by X, $\mathbb{P}_X$. Assume that $\mathbb{P}_X$ is symmetric, ie that for all $A \in \mathcal{B}(\mathbb{R})$, $\mathbb{P}_X(A) = \mathbb{P}_X(-A)$. Then the antithetic variate estimate for $\mathbb{E}[f(X)]$ is:

$$c = \frac{c_1 + c_2}{2}$$

.Where we used the notations

$$c_1 := \frac{1}{N} \sum_{n=1}^{N} f(X_n),$$
$$c_2 := \frac{1}{N} \sum_{n=1}^{N} f(-X_n)$$

Remark 3.3. Since c_1 and c_2 are both unbiased estimates of $\mathbb{E}[f(X)]$, then so is $c = \frac{c_1 + c_2}{2}$. However this technique allows variance to be reduced if c_1 and c_2 are negatively correlated.

Definition 3.4 (Romano-Touzi trick as used in McCrickerd & Pakkanen ([11] p.5-6))**.** In the Bergomi and rough Bergomi models, the price process can be decomposed in the product of two processes driven by orthogonal Brownian motions:

$$S_t = S_t^1 S_t^2$$
$$S_t^1 = S_0 \mathcal{E}(\int_0^t \sqrt{v_u} \rho dW_u)$$
$$S_t^2 = S_0 \mathcal{E}(\int_0^t \sqrt{v_u} \sqrt{1-\rho^2} dW_u^T)$$

It then follows that the option price can be estimated using only one of those processes. In fact, one then has that the option price is the price given by a Black Scholes model with underlying S_1 and implied volatility the integrated variance $(1-\rho^2) \int_0^{\cdot} v_u$. During the simulations, we estimate numerically S_1 and $(1-\rho^2) \int_0^{\cdot} v_u$ and hence compute option prices according to the Bergomi or rough Bergomi model. This removes the need to simulate the second Brownian motion.

3.3 Computing implied volatilities from option prices

For each given model, we compute implied volatilities from option prices using the methodology and code laid out in [12]. Using functional analysis, the author reduces the search for possible values of the implied volatility.

4 Neural networks for function approximation

4.1 Feedforward neural networks

For consistency, we follow the notations taken in Buehler et al ([13]):

Definition 4.1 (Neural network)**.** Let $L \in \mathbb{N}$, $(N_1, N_2 \cdots, N_L) \in \mathbb{N}^l$, affine functions $(W^1, \cdots, W^L)$ such that for $1 \leq l \leq L-1$, $W^l : \mathbb{R}^{N_l} \mapsto \mathbb{R}^{N_{l+1}}$. We note $F_l := \sigma_l \circ W^l$ where σ_l is an *activation function* and applied component wise on the outputs of the affine function W^l. Then a neural network $F : \mathbb{R}^{N_1} \mapsto \mathbb{R}^{N_L}$ with layers $F_1 \cdots F_L$ is :

$$F := F_L \circ F_{L-1} \circ \cdots F_1$$

The first and last layers, F_1 and F_L, are the *input* and *output* layers. Layers in between, $F_2 \cdots F_{L-1}$, are called *hidden layers*.

Remark 4.2 (Examples of activation functions)**.** Common activation functions include:
the *rectified linear unit*, or ReLU: $\sigma_{ReLU} : x \mapsto \max(0, x)$
the *exponential linear unit*, or ELU, where $\alpha > 0$: $\sigma_{ELU} : x \mapsto \max(0, x) * \min(\alpha(e^x - 1), 0)$

Remark 4.3 (Parametrisation of affine functions)**.** Given a neural network F with layers $F_1 \cdots F_L$ and respective affine functions $W^1, \cdots, W^L$, we parameterise affine functions as :

$$W^l := A^l \cdot + b^l$$

Where $A^l \in \mathbb{R}^{N_{l-1}, N_l}$ is the *weight matrix* of layer l, $b^l \in \mathbb{R}^{N_l}$ is the *bias* of layer l.

Remark 4.4 (Parametrisation of neural networks)**.** A neural network F with layers $F_1, \cdots, F_L$ is fully parametrised by its activation functions $\sigma_1, \cdots, \sigma_L$, weight matrices and biases $(A^1, b^1), \cdots, (A^l, b^l)$ such that:

$$\forall l \in [\![1, L]\!], F_l = \sigma_l \circ (A^l \cdot + b^l)$$

4.2 Approximation capabilities of neural networks

Now that we have characterised and parametrised neural networks, we show that, for properly chosen activation functions, for any smooth or L^p function there exists a neural network approximating such a function.

Definition 4.5 (Set of neural network mappings)**.** Let $\sigma : \mathbb{R} \mapsto \mathbb{R}$. We define the set of single-layer neural networks with activation function σ, input dimension d_0 and output dimension d_1 as $\mathcal{NN}^{\sigma}_{d_0,d_1}$.

Theorem 4.6 (Universal Approximation for smooth, bounded non-constant activation functions (Hornik, [14], Theorem 2, p. 252))**.** *If ψ is continuous, bounded and non-constant, $d_0 \in \mathbb{N}^*$, then $\mathcal{NN}^{\psi}_{d_0,1}$ is dense in the space of continuous functions on compact supports of $\mathbb{R}^{d_0}$.*

Proof. The proof is motivated by measure theoretic arguments. We reproduce here the outline of the proof given by Hornik ([14] p.254-256) and Cybenko ([15], p.306).
Since the space of neural network functions is a linear subspace of the space of continuous functions, we can apply the Hahn-Banach Theorem. If the closure of the first set is not the second, then there exists a continuous linear functional $L : C^c(\mathbb{R}^k) \to \mathbb{R}$ such that $L(f) = 0$ for all f single-layer neural network functions with activation function ψ while $L \neq 0$.
Since we are on the subspace of continuous functions on compact supports, the inner product $< \cdot, \cdot >: (f, g) \mapsto \int f g d\mu$ is well-defined. But by the Riesz representation Theorem, for $\phi \in C(\mathbb{R}^k)$ there exists $g \in C^c(\mathbb{R}^k)$ such that $\phi(f) = \int f g d\mu$. Applying this here, there exists $g \in C^c(\mathbb{R}^k)$ such that $\int f g d\mu = 0$ for all $f \in \mathcal{NN}^{\psi}_{k,1}$ but $f \mapsto \int f g d\mu$ is not null.
We may define the signed measure associated to g, σ, on the Borel space by:

$$\sigma(A) = \int_A g d\mu, A \in \mathcal{B}(\mathbb{R}^k)$$

. Since all functions are continuous and defined on a compact support this is indeed a signed measure and it is finite.
Combining and using the definition of single-layer neural network functions with activation function ψ, we have shown that there exists a non-zero finite signed measure σ such that for all linear maps $k : x \mapsto ax + b$,

$$\int_{\mathbb{R}^k} (\psi \circ k) d\sigma = 0$$

.
Using arguments from Fourier analysis, Hornik ([14] p.225-256) shows that no such function can exist if ψ is bounded and non-constant, hence concluding the proof. □

Theorem 4.7 (Universal Approximation for unbounded non-constant activation functions (Hornik, [14], Theorem 1, p. 252))**.** *If ψ is unbounded and non-constant, $d_0 \in \mathbb{N}^*$, then $\mathcal{NN}^{\psi}_{d_0,1}$ is dense in $L^p(\mu)$ for all finite measures μ.*

Proof. To adapt the proof from Theorem 4.6 for Theorem 4.7, the Riesz representation Theorem gives the existence of a function in the dual space of L^p, which is L^q, where $q = \dfrac{p}{p-1}$ is the conjugate of p. The inner product is still well-defined and the measure finite by Holder's inequality applied to $\sigma(A), A \in \mathcal{B}(\mathbb{R}^k)$. □

Corollary 4.8 (Extension to multiple outputs (Hornik, Stinchcombe & White, [16], corollary 2.6-2.7, p.363))**.** *If σ is unbounded and non-constant, $d_0, d_1 \in \mathbb{N}^*$, then $\mathcal{NN}^{\sigma}_{d_0,d_1}$ is dense in $L^p(\mu)$ for all finite measures μ.*

Proof. The proofs used in Theorems 4.6 and 4.7 remain valid when functions take values in $\mathbb{R}^k$. In particular the Hahn-Banach Theorem and Riesz representation Theorem can still be applied, and one may define a finite signed measure on $\mathcal{B}(\mathbb{R}^k)$. The result then follows from the same arguments as in the unidimensional case (Hornik, [14] p.225-256). □

4.3 Application to option pricing

In our framework, we wish to approximate a map from model parameters to option prices. Given strikes $(K_i)_{1\leq i\leq I}$, maturities $(T_j)_{1\leq j\leq J}$ and a stochastic model with parameters in $\Theta \subset \mathbb{R}^k$ for some $k \in \mathbb{N}$, we are concerned with approximating the map:

$$f : \Theta \to \mathbb{R}^{I\cdot J}$$

,

where, for $\theta \in \Theta$,

$$f(\theta) = (\sigma_{BS}(K_1, T_1; S_0), \cdots, \sigma_{BS}(K_I, T_1; S_0), \sigma_{BS}(K_1, T_2; S_0), \cdots \sigma_{BS}(K_I, T_J; S_0))$$

Notation 4.9. Following the parametrisation given in 4.4 and for convenience, a neural network will now be represented by $f : \mathbb{R}^d \times \mathbb{R}^D \mapsto \mathbb{R}$, where d, D are respectively the input and parameter dimension. Therefore $f(x, w)$ is the output of the neural network with parameters w given the input x.

Notation 4.10. We define a *per-sample loss*, $\mathcal{L}(x, y, w)$, quantifying the error made on a specific sample by a neural network with parameters θ. If $L : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}$ is a loss function, then $\mathcal{L}(x, y, w) = L(f(x, w), y)$.

Definition 4.11 (Generalisation error, (Goodfellow et al, [17], p.110))**.** Given a per-sample loss function $\mathcal{L}(x, y, \theta)$ such that samples (x, y) are draw independently from $\hat{p}$, the *generalisation error* is the average error made on samples drawn from this distribution:

$$J(w) = \mathbb{E}_{\hat{p}} L(x, y, w)$$

In the training procedure, we will seek to minimise $J(w)$ through updates of our parameter set. One approach is to minimise the *empirical error*, since the central limit Theorem guarantees that almost surely it converges to the generalisation error because we assume that samples are independent and drawn from the same distribution.

Definition 4.12 (Empirical error)**.** Given a per-sample loss function $\mathcal{L}(x, y, \theta)$ such that samples (x, y) are draw independently from $\hat{p}$, and a set of samples (x, y) where $x = (x^{(1)}, \cdots, x^{(K)})$, $y = (y^{(1)}, \cdots, y^{(K)})$, the *empirical error* is:

$$\mathcal{L}(x, y, w) = \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(x^{(i)}, y^{(i)}, w)$$

Definition 4.13 (Training and testing error, (Goodfellow et al, [17], p.110))**.** Given a per-sample loss function $\mathcal{L}(x, y, w)$ such that samples (x, y) are draw independently from $\hat{p}$, and a set of samples (x, y) where $x = (x^{(1)}, \cdots, x^{(K)})$, $y = (y^{(1)}, \cdots, y^{(K)})$, we may split (x, y) into two subsets, one used during the optimisation procedure and the other for test purposes. Those subsets are usually called *training* and *testing* sets. The *training error* and *testing error* are the empirical errors computed respectively on the training set and on the training set.

Because we seek to minimise a function over all possible samples from a given distribution but have access to only some, two problems may occur.

Definition 4.14 (Overfitting, underfitting, (Goodfellow et al, [17], p.111))**.** *Overfitting* occurs when the training error is much smaller than the testing error. *Underfitting* occurs when a model returns too high an error on the training set.

Remark 4.15. Overfitting and underfitting are qualitative properties. To understand if underfitting is an issue we may use other models as benchmarks for the error level we expect the current model to reach. Likewise there are different levels of overfitting but we may choose to say overfitting occurs when the difference between training and testing error exceeds a relative threshold.

To minimise the training error, we use an estimate of the gradient over the training set, $\nabla_\theta L(x, y, \theta)$. However such an estimate may be difficult to compute in practice, with the empirical gradient estimate being:

$$\nabla_w \mathcal{L}(x, y, w) = \frac{1}{K} \sum_{i=1}^{K} \nabla_w \mathcal{L}(x^{(i)}, y^{(i)}, w)$$

As such, we restrict the estimation of the gradient on a small subset of samples, a *batch*.

Definition 4.16 (mini-batch)**.** A *mini-batch* of size m of (x, y) is a subset of (x, y) of m samples.

Remark 4.17. Repeating samples are not excluded in a batch. Typical batch sizes are usually between 10-100 samples and the sampling procedure is usually uniform without replacement.

Given a mini-batch $(\hat{x}, \hat{y})$ of size m of (x, y), the gradient estimate over the mini-batch is:

$$g = \frac{1}{m} \sum_{i=1}^{m} \nabla_w \mathcal{L}(\hat{x}^{(i)}, \hat{y}^{(i)}, w)$$

This formula requires two operations: computation of gradients and sums. To further reduce computation time we can rewrite Equation 4.3 into (Goodfellow et al, [17], p.278):

$$g = \frac{1}{m}\nabla_w \sum_{i=1}^{m} \mathcal{L}(\hat{x}^{(i)}, \hat{y}^{(i)}, w)$$

Now that we have an estimate of the error gradient, we can minimise it through an optimisation procedure.

5 Optimization procedure

5.1 Problem setting

To understand how neural networks minimise the training error, we look at the different optimisation algorithms used and how they are used in the case of neural networks. First, we recall important notions in optimization.

Definition 5.1 (Local minimum, [18] p.13)**.** Given $f : \mathbb{R}^d \mapsto \mathbb{R}$, x is a *local minimum* if there exists a neighbourhood of x, $\mathcal{V}$, such that

$$\forall y \in \mathcal{V}, f(x) \leq f(y)$$

Optimisation algorithms considered here iteratively compute a sequence $(x_n)_{n\in\mathbb{N}}$ starting from an initial guess x_0. Algorithms differ in how they build the next element of the sequence ([18] p.19). In the context of neural networks, optimisation methods used derive from *line search methods.*

5.2 Line search methods

Definition 5.2 (Line search methods (Wright & Nocedal, [18], p.19))**.** Given a real-valued function f and current iterate x_k, line search methods compute a *step direction* p_k and *step size* α_k and set the next iterate as:

$$x_{k+1} = x_k + \alpha_k p_k$$

Remark 5.3 (Optimal step sizes in line search (Wright & Nocedal, [18], p.19))**.** Given an iterate x_k and a step direction p_k, the optimal step size is the one that minimises the function along the given direction:

$$\alpha_k = \text{argmin}_{\alpha>0} f(x_k + \alpha p_k)$$

Where we have excluded instances where $\alpha < 0$ because we follow the given direction.

To choose a step direction, we study the case of smooth functions.

Proposition 5.4 (Wright & Nocedal, [18], p.22)**.** *Let $f \in C^2(\mathbb{R}^n)$ a real-valued function, $x \in \mathbb{R}^n$. The normed step direction p which leads to the largest decrease is*

$$p = -\frac{\nabla f(x)}{|| \nabla f(x) ||}$$

Proof. The result stems from the Taylor-Lagrange expansion applied to $f(x + \alpha p)$ (Wright & Nocedal, [18], p.22). □

Line search algorithms are simple and, for well-behaved functions, we have results concerning the convergence of the line search algorithms. First, we have the following results guaranteeing termination.

Definition 5.5 (Global convergence)**.** A line search algorithm is said to be globally convergent if, given the sequence of iterates $(x_n)_n \in \mathbb{N}$,

$$|| \nabla f(x_n) || \to_\infty 0$$

Definition 5.6 (Woolfe conditions ([18], p.39))**.** Let $f : \mathbb{R}^n \mapsto \mathbb{R}$ a continuously differentiable function, $x_k \in \mathbb{R}^n$ $\alpha_k \in \mathbb{R}$, $p_k \in \mathbb{R}^n$, we say (x_k, p_k, α_k) satisfy the Woolfe conditions if there exists c_1, c_2 such that $0 < c_1 < c_2 < 1$ and:

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f(x_k) p_k^T$$
$$\nabla f(x_k + \alpha_k p_k) \geq c_2 \nabla f(x_k) p_k^T$$

Lemma 5.7 (Existence of step sizes satisfying the Woolfe conditions, (Lemma 3.1, Wright & Nocedal, [18], p.40))**.** *Let $f : \mathbb{R}^n \mapsto \mathbb{R}$ a continuously differentiable function, $p, x \in \mathbb{R}^n$. Assume that f is bounded below on the half-line $\{x + \alpha p : \alpha > 0\}$. Then, given any $0 < c_1 < c_2 < 1$, there exists α_l, α_r such that for all $\alpha \in [\alpha_l, \alpha_r]$, (x, p, α) satisfy the Woolfe conditions.*

Proof. A detailed proof is given in Wright & Nocedal, ([18] p.35). □

Theorem 5.8 (Theorem 3.2, Wright & Nocedal, [18], p.38)**.** *Let f be a real-valued function and $(x_n)_{n \in \mathbb{N}}$. Assume that:*

- *iterates are given by 5.2 where for all $k \in \mathbb{N}$, (x_k, p_k, α_k) satisfy the Wolfe conditions,*
- *there exists an open set O included in $\{x : f(x) \leq f(x_0)\}$ where f is continuously differentiable,*
- *∇f is Lipschitz continuous on O*

Then, noting

$$\cos(\theta_k) := -\frac{\nabla f(x_k)^T p_k}{|| \nabla f(x_k) || \cdot || p_k ||}$$

We have the convergence of the series:

$$\sum \cos^2(\theta_k) \mid\mid \nabla f(x_k) \mid\mid^2 < \infty$$

Proof. A detailed proof is given in Wright & Nocedal, ([18] p.38-39). □

Corollary 5.9 (Global convergence of line search algorithms, Wright & Nocedal, ([18], p.39)**.** *Assume the hypothesis of Theorem 5.8 are satisfied and there exists $\delta > 0$ such that for all $n \in \mathbb{N}, \cos(\theta_n) \geq \delta > 0$. Then the line search is globally convergent.*

We now move on from the study of general line search algorithms to specific examples used in optimisation of neural networks.

5.3 Line search algorithms for neural networks

We now seek to minimise the generalisation error, given by, using the notations of 4.11:

$$J(w) = \mathbb{E}_{\hat{p}} \mathcal{L}(x, y, w)$$

The smoothness properties of J depend on the underlying distribution and on the per-sample error function L. In our case, we first note that when all layers of a neural network are smooth, then so is the network.

Lemma 5.10. *Assume that $\mathcal{L}(x, y, w) = C(f(x, w), y)$, where $C : \mathbb{R}^2 \to \mathbb{R}$ is a twice continuously differentiable function and $f : \mathbb{R}^n \times \mathbb{R}^d \mapsto \mathbb{R}$ is the neural network function, where d is the total number of parameters of the network. Assume that the activation functions of all layers are twice continuously differentiable. Then, $L(x, y, \cdot)$ is twice continuously differentiable.*

Proof. Since the composition of two $\mathcal{C}^2$ functions is again $\mathcal{C}^2$, the result holds. □

Lemma 5.11. *Assume further that $\mathcal{L}(x, y, \cdot)$ and $\nabla_w \mathcal{L}(x, y, \cdot)$, $\nabla_w^2 \mathcal{L}(x, y, \cdot)$ are dominated by L^1 functions with respect to $\hat{p}$ that do not depend on w. Then $J(w) = \mathbb{E}_{\hat{p}} \mathcal{L}(x, y, w)$ is twice continuously differentiable.*

Proof. The result follows from the dominated convergence Theorem. □

Proposition 5.12. *Assuming the hypotheses of Proposition 5.11 and Theorem 5.8 and Corollary 5.9 are verified, the line search algorithm applied to the neural network converges.*

Proof. The result is an application of Theorem 5.8 given f satisfies the hypotheses. □

5.3.1 Gradient Descent

Definition 5.13 (Gradient descent (Wright & Nocedal, [18], p.22))**.** Gradient descent, also called steepest descent, is a line search method where the step direction is chosen as the gradient, yielding the following iterates:

$$w_{n+1} := w_n - \alpha_n \nabla_\theta \mathcal{L}(x, y, w)$$

Where $\nabla_w \mathcal{L}(x, y, w)$ is the empirical loss:

$$\nabla_w \mathcal{L}(x, y, w) = \frac{1}{K} \nabla_w \sum_{i=1}^{K} \mathcal{L}(x^{(i)}, y^{(i)}, w)$$

5.3.2 Stochastic Gradient Descent

Definition 5.14 (Stochastic gradient descent (Goodfellow et al, [17], p.277-279))**.** Stochastic gradient descent is a line search method where the step direction is chosen as the gradient, yielding the following iterates:

$$w_{n+1} := w_n - \alpha_n g$$

Where g is the gradient estimated on a randomly sampled mini-batch:

$$g = \frac{1}{m} \nabla_w \sum_{i=1}^{m} \mathcal{L}(\hat{x}^{(i)}, \hat{y}^{(i)}, w^{(j)})$$

Remark 5.15. Gradient descent methods are computationally expensive but should be more precise than stochastic gradient descent methods. However, assuming that the losses are independent and identically distributed, the strong law of large numbers implies the convergence rate towards the true mean is in $\sqrt{N}$, where N is the number of samples. Therefore, an x-fold improvement in precision requires x^2 more samples (Goodfellow et al [17] p.278-279). Instead of computing a single, more accurate gradient estimate it is therefore preferred to compute multiple, noisier ones instead.

Remark 5.16. A choice of step size α_n has a strong influence on the convergence of gradient descent and stochastic gradient descent and a robust choice of step size is the subject of much work. We do not go into the details of different step size policies. It has been understood that step sizes must be chosen in accordance to batch sizes. While usual schemes recommend decreasing step sizes, for example with a rate of $\alpha_n = \frac{1}{n}$, annealed step sizes cycle.

5.3.3 Adam

Definition 5.17 (Adam scheme (Kingma & Ba, [19], p.2))**.** Given parameters $0 \leq \beta_1, \beta_2 < 1, \epsilon$, α, initial iterates $m_0 := 0, v_0 := 0, \theta_0$, the adam scheme proposes the following iterates:

$$\begin{aligned} g_t &:= \nabla_\theta L(x, y, w_t) \\ m_{t+1} &:= \beta_1 m_t + (1 - \beta_1) g_t \\ v_{t+1} &:= \beta_2 v_t + (1 - \beta_2) g_t^2 \\ w_{t+1} &:= w_t - \alpha \frac{m_{t+1}}{1 - \beta_1^{t+1}} \frac{1}{\sqrt{v_t / (1 - \beta_2^{t+1})} + \epsilon} \end{aligned}$$

Remark 5.18. The above scheme can be adapted into stochastic form by replacing the gradient estimate over the full data set to its estimate on randomly sampled batches.

Remark 5.19. Default parameter values proposed in [19], p.2, are:

$$\begin{aligned} \alpha &= 0.001 \\ \beta_1 &= 0.9 \\ \beta_2 &= 0.999 \\ \epsilon &= 10^{-8} \end{aligned}$$

While those methods tell us how to update parameters given an estimate of the gradient, we have yet to compute the gradient. To do so, we will use the chain rule along the layers to obtain the gradient of the loss function with respect to every parameter. This is obtained via backpropagation (Goodfellow et al, [17], p.205-218). Backpropagation relies on the chain rule, which we use the partial derivative of the loss function with respect to parameters at a given layer as a function of the partial derivative of the loss function with respect to parameters at the next layer. Since the gradient at the output layer is known, we can iteratively compute the gradients within the network (Goodfellow et al, [17], p.205-218).

6 Results

6.1 Implementation and experimental setting

6.1.1 Generating surfaces using Monte Carlo

Each model contains code to generate implied volatility surfaces from a set of parameters. For faster simulation, algorithms are written in cython (Behnel et al, [20]) which allows us to write C code for python. This reduces computation time for the generation of surfaces through Monte Carlo. The code was provided by Aitor Muguruza and only slightly modified to be included in the

library. To compute implied volatilities we first compute option prices and then use C++ from Jäckel ([12]). We generate 20,000 surfaces for each model, using a fixed grid of strikes and tenors (define the grid here).

6.1.2 Training neural networks using generated surfaces

Once surfaces have been generated.

Neural networks are built using keras (Chollet, [21]) with tensorflow backend (Abadi et al, [22]). For clarity purposes, we provide a detailed explanation of how the graph is built and trained.

The architecture of the neural network is defined in file *SurrogateGenerator* and can be changed to test out different architectures. Using keras, the following lines define the feedforward model, the loss function and gradient update method:

```
def build(self):
        # Build the NN architecture for generator
        layer_1 = Dense(1000,input_dim = self.y.shape[1],activation = 'elu',name =
             'layer_'+str(self.model_key)+'generation_1')
        layer_2 = Dense(800,activation = 'elu',name = 'layer_'+str(self.model_key)
            +'generation_2')
        layer_3 = Dense(600,activation = 'elu',name = 'layer_'+str(self.model_key)
            +'generation_3')
        layer_4 = Dense(self.X_train.shape[1],activation = 'elu',name = 'layer_'+
            str(self.model_key)+'generation_4')

        self.model = Sequential()

        self.model.add(layer_1)
        self.model.add(layer_2)
        self.model.add(layer_3)
        self.model.add(layer_4)

        self.model.compile(loss = self.loss, optimizer = "adam")
```

After sampling parameters and surfaces from the training set, the current network is used to compute surfaces. Using the mini-batch method, the loss function is averaged over sampled parameters and used to update network parameters. This continues until we have used all examples in the training set, which constitutes an epoch. The process is repeated a certain number of times or until the training loss stops decreasing.

To monitor training and ensure the network is not overfitting, we leave a few examples out of the training set to use as validation. The average loss on the validation set is shown at the end of each training epoch.

6.1.3 Using neural networks to calibrate

Once network weights have been saved, we may load them into a variable using the *Reader* module. Since the function has already been trained and we do not wish to change weights further, the layers are excluded from the gradient updates done in the training procedure.

We define layers that map the initial guess to a new one using the following lines:

```
def __add_layers_exploration(self):
        self.__modelCalibration = Sequential()

        layer_exploration_1 = Dense(50,input_dim = self.__N_parameters,activation
            = 'elu',name = 'layer_exploration_1')
        layer_exploration_2 = Dense(10,input_dim = self.__N_parameters,activation
            = 'elu',name = 'layer_exploration_2')
        layer_exploration_3 = Dense(self.__N_parameters,activation = 'elu',name =
            'layer_exploration_3')

        self.__modelCalibration.add(layer_exploration_1)
        self.__modelCalibration.add(layer_exploration_2)
        self.__modelCalibration.add(layer_exploration_3)

        self.__layers_exploration = []
        self.__layers_exploration += [layer_exploration_1]
        self.__layers_exploration += [layer_exploration_2]
        self.__layers_exploration += [layer_exploration_3]
```

The output of this network is then fed into the generator network which will map it to surfaces.

Once the training procedure is over, we output the output of the calibrator network as the calibrated parameters. Those are the parameters that, once fed into the generator, give a close approximation of the surface passed we wish to calibrate against.

6.2 Using the code

To make the code easier to use, we have included an easily importable api. We describe below the core functionalities and code examples:

6.2.1 Computing Implied volatility surfaces using Monte Carlo

To compute implied volatilities using Monte Carlo, call the function. Strikes and maturities are specified in the file.

```
from Library.api.api import *
api = API('bergomi')
api.surface_from_parameters_MC(np.array([-0.5,2.0,0.02]))
```

6.2.2 Creating neural network to map parameters to implied volatility surfaces

Once implied volatility surfaces have been generated and saved under the appropriate format, they can be loaded to train the neural network. To do so, manually run the *SurrogateGenerator* file, specifying which model to run. After training, the neural network and parameter scaler will automatically be saved.

Example of function call and output:

6.2.3 Computing implied volatility surfaces using generated models

Once neural network weights and scalers have been saved we can load them to compute implied volatility surfaces using those models. To do so, we can use the api:

```
from Library.api.api import *
api = API('bergomi')
api.surface_from_parameters(np.array([-0.5,2.0,0.02]))
```

6.2.4 Calibrating model to given implied volatility surface using neural networks

After neural networks from parameters to surfaces have been saved, we can call the models to calibrate implied volatility surfaces and output parameters. To do so, we can use the api:

```
from Library.api.api import *
api = API('bergomi')
surface = api.surface_from_parameters_MC(np.array([-0.5,2.0,0.02]))
parameters = api.parameters_from_surface(surface)
```

6.2.5 Training procedure for generator

To train the generator, we first sample 20,000 parameters. Then, we compute the implied volatility surfaces for each of those parameters and save them. While this might seem like a small number of samples to successfully fit a neural network. In comparison using Monte Carlo we require.

The network contains 4 layers, each with the elu activation function. The first layer contains 1000 nodes, the second 800, the third 600 and the last the number of points of our implied volatility surface. The training scheme used is Adam with default parameters.
A callback is used to stop the training procedure when the training loss has not decreased in the last five epochs. There is a compromise between the number of epochs before training is stopped. If we chose too low a number we would stop training immediately and we would be underfitting However if we chose too large a number we risk training for longer than necessary and causing overfitting. We chose 5 epochs as performance only marginally improved without after this, but other choices did not significantly impact performance, indicating the training is robust with respect to this parameter.
The chosen loss is the mean squared error, which we chose because it is smooth and gives larger gradient updates for large errors rather than small ones.
We choose batch sizes of 20. Small batch sizes make for noisier gradient estimates and may reduce the accuracy of the parameter update. Very large batch sizes increase computation time because they require holding in memory and computing a larger number of gradients at a time, though they should reduce the variance of our gradient estimate. Therefore we need to find a compromise between the two. Typical batch size values range from around 10 to 100 depending on the application. In our case we started with small batch sizes for faster computations and increased them until training procedures consistently converged, which happened immediately at 16 but we found better performance at around 20, though tests have been made on only one model.
The default number of epochs is 20. Too few epochs increase the chance of underfitting while too many increase the chance of overfitting. The number of epochs is largely dependent on the application but we tested different number of epochs and found that around 20 is enough to avoid significant underfitting while preventing overfitting.
We have not fine tuned the architecture of the network aside from choosing an appropriately large number of nodes and the Adam scheme for consistent results and elu activation functions to avoid the vanishing gradient problem. Since the network achieves consistently good performance for each model studied we do not concern ourselves with further optimisation.
This operation is executed for each model and yields a neural network per model to approximate implied volatility surfaces from parameters.

6.3 Training procedure for calibrator

Once the generator network has been trained, we take as input an implied volatility surface and return parameters. We freeze the weights of the generator network to avoid updating them during the training procedure. Before passing data to the generator network we add 3 layers, each with elu activation functions and of respective node size 50, 10, and the number of output parameters.

We pass as input of the overall model an initial guess of 0, since parameters have been centered. After the training procedure, we return the output of the first three layers. Since it is scaled, we use the standard scaler created in the training procedure of the generator network to return it in original units.
Loss chosen is mean squared error and there are 200 epochs. As before, this architecture has not been fine tuned but still achieves consistent results.

Depending on the model considered, we uniformly sample each parameters. While this sampling could be improved, in particular since not all combinations of those parameters will be arbitrage free. To choose sampling bounds, we report calibration results from multiple sources to consider plausible bounds for our parameters.

6.4 Approximation

First, we compare the neural network approximation to the Monte Carlo function. Examples of generated surfaces compared to Monte Carlo surfaces are shown in Figures 8, 7, 6. Using generated surfaces we compute the average error per tenor and strike, depending on the chosen error metric. Results are shown in Figures 14, 15, 13. Those results show the neural network is successfully approximating the surface. Furthermore, we examine the generalisation capabilities of the network for parameters outside of the sampling bounds previously defined. If the network has successfully captured the map from parameters to option prices on a subset of the parameter space, then its extension should also approximate the map. And indeed we can see that the map generalises outside of the sampling bounds by looking at examples shown in Figures **??**, **??** and **??**. Finally, to compare precision with market prices, we use statistical tests to check if the average error is below the threshold given by spread. To do so, we consider the at-the-money curve for each model and test for the null hypothesis that the error is above a given spread threshold, by using the likelihood ratio test, using the chi square distribution. The null hypothesis is that the mean error is above a certain threshold. We look at spread thresholds such that null hypothesis is rejected at a 5% and 1% level. The results are shown in **??**. The methodology considerably speeds up the computation of prices, as shown in table **??**. As suspected speed up is least significant for the Heston model for which we have an analytical formula, and most important for the rough Bergomi model.

6.5 Calibration

We first take in an implied volatility surface and calibrate it. We compare the original parameters to their estimation and show the adjusted error in Figures 16. We then compare the surfaces generated using the calibrated parameters, in Figures 21, 22, 23.We test the robustness of the calibration procedure by removing points of the implied volatility surface. To do so, we randomly

sample points to be removed from the surface. Those are replaced by the average implied volatility at the given tenor and strike, according to the 20,000 surfaces generated for training the generator network. We plot the calibration error depending on the proportion of masked values in the implied volatility in Figures 26, 25, **??**. We also examine the interpolation of the volatility surface using the neural network. To do so, we choose new strikes and maturities with the same respective numbers and close to the previous ones, and examine the error between the surface generated from estimated parameters and from model parameters. We find parameters on the input surface and then estimate the volatility surface.

data and source	κ	ρ	σ	θ	V_0
EURUSD options, march 18th 2015, Saporito et al, [23]	1.025	-0.626	0.161	0.013	0.013
long-dated FX options, Cui et al, [24]	0.5	-0.9	1.0	0.04	0.04
long-dated interest rate options, Cui et al, [24]	0.3	-0.7	0.9	0.04	0.04
long-dated equity options, Cui et al, [24]	1.0	-0.3	1.0	0.09	0.09
DAX index options, march 18th 2013, Mrázek, Pospíšil, Sobotka, [25]	1.27	-0.66	0.59	0.09	0.027

Figure 1: Calibrated Heston parameters found in the literature. There is a wide variation between asset classes, though the values are consistent with expectations. Correlations are negative, the long-term mean θ is within reasonable bounds compared to implied volatilities. Depending on the asset class it seems the speed to mean-reversion can take different values, though it stays within bounds of a couple of months to little more than a year. The volatility of volatility also appears reasonably bounded; very small values would make the volatility process close to deterministic while very high values would overshadow the mean-reversion component of the process. While we expect the initial value of the volatility process,V_0, to depend on the date of calibration it stays within reasonable volatility bounds.

data and source	ρ	ν	σ_0	H
SPX options, February 4th 2010, Bayer, Friz & Gatheral, [2], p.18	-0.9	1.9	estimated empirically	0.07
SPX options, August 14th 2013, Bayer, Friz & Gatheral, [2], p.18	-0.9	2.3	estimated empirically	0.05

Figure 2: Calibrated rough Bergomi parameters found in the literature. As in **??**, correlations are negative. The values of the Hurst parameter found are consistent with other studies done in Gatheral, Jaisson, Rosenbaum ([10]). In particular they are found much lower to 0.5, suggesting the rough Bergomi model should fit the implied volatility curve better than the Bergomi model.

	lower bound	upper bound
ρ	-0.9	-0.3
κ	0.2	2.0
σ	0.1	1.5
θ	0.01	0.1
V_0	0.05	0.15

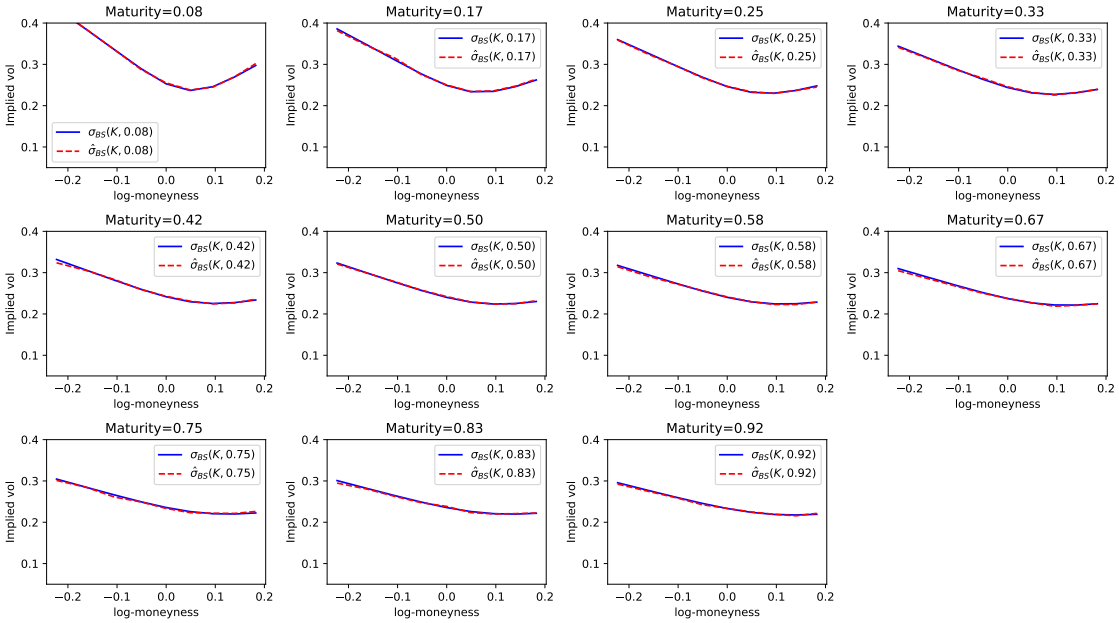
Figure 3: Heston sampling bounds chosen in our analysis. We have used parameters from figure **??** for the upper and lower bounds, allowing for much larger values of the speed to mean-reversion since we have observed that the neural network generalises poorly for values of κ close to the upper bound. Therefore we allowed for some additional margin compared to parameters found in the literature so approximation and calibration could remain precise.

	lower bound	upper bound
ρ	-0.9	-0.3
ν	0.1	2.5
σ_0	0.01	0.15

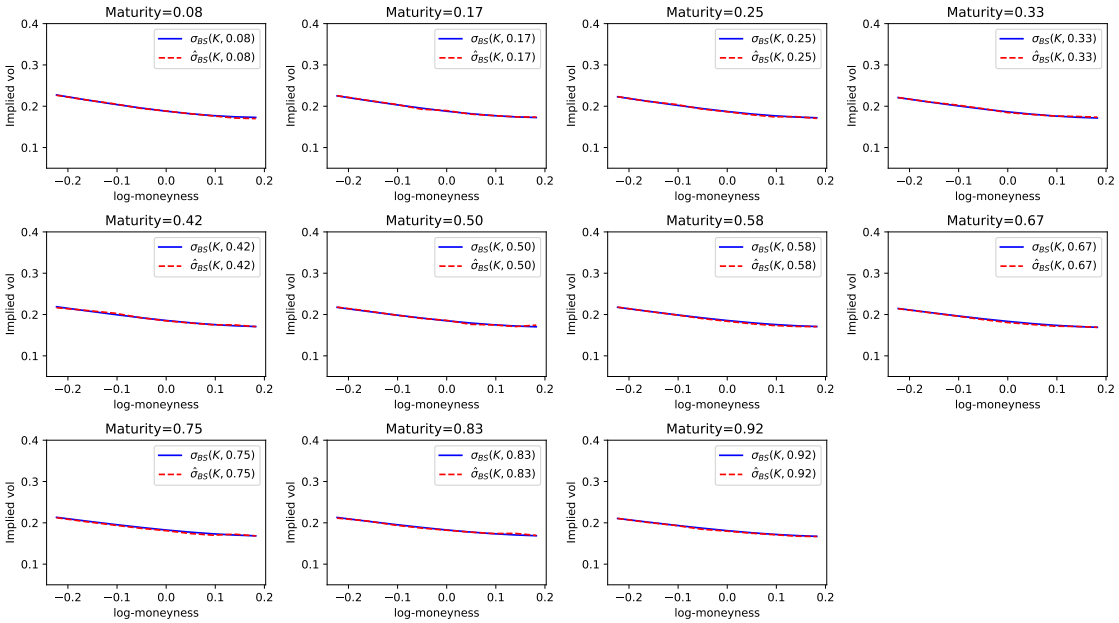
Figure 4: Bergomi sampling bounds chosen in our analysis. We have used parameters from figure **??**, where we assumed we would have found similar parameters if we manually set the Hurst parameter to $H = 0.5$. Since the initial forward volatility σ_0 is estimated from market prices, choose to sample from within reasonable bounds for forward volatilities, similar to those chosen in **??**.

	lower bound	upper bound
ρ	-0.9	-0.3
ν	0.1	2.5
σ_0	0.01	0.15
H	0.05	0.5

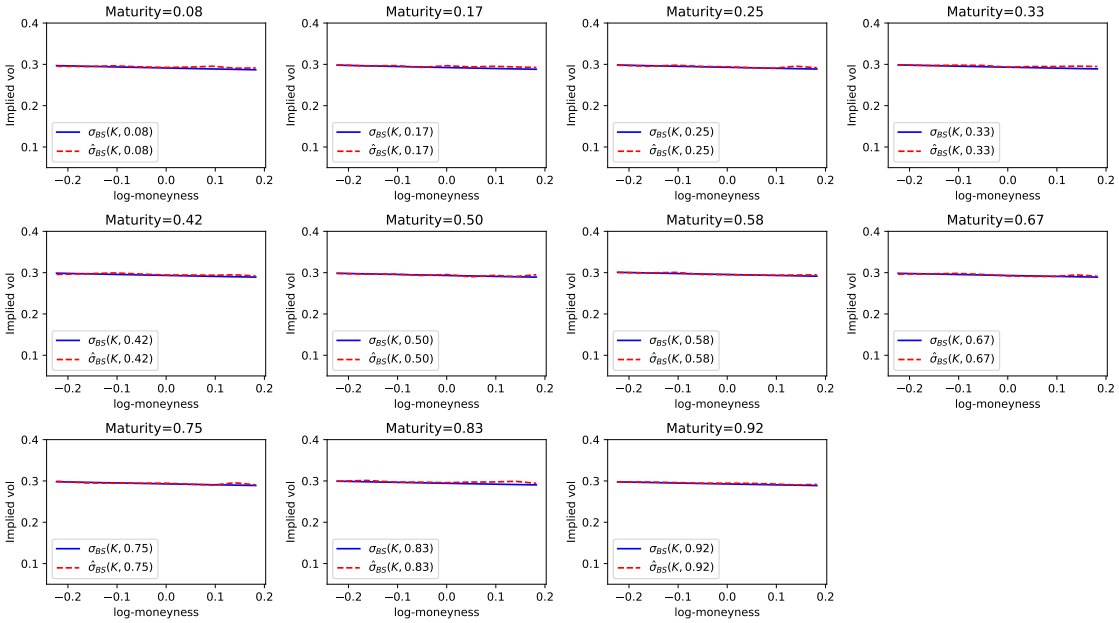
Figure 5: rough Bergomi sampling bounds chosen in our analysis. We have used parameters from figure **??**. Since the initial forward volatility σ_0 is estimated from market prices, choose to sample from within reasonable bounds for forward volatilities, similar to those chosen in **??**.



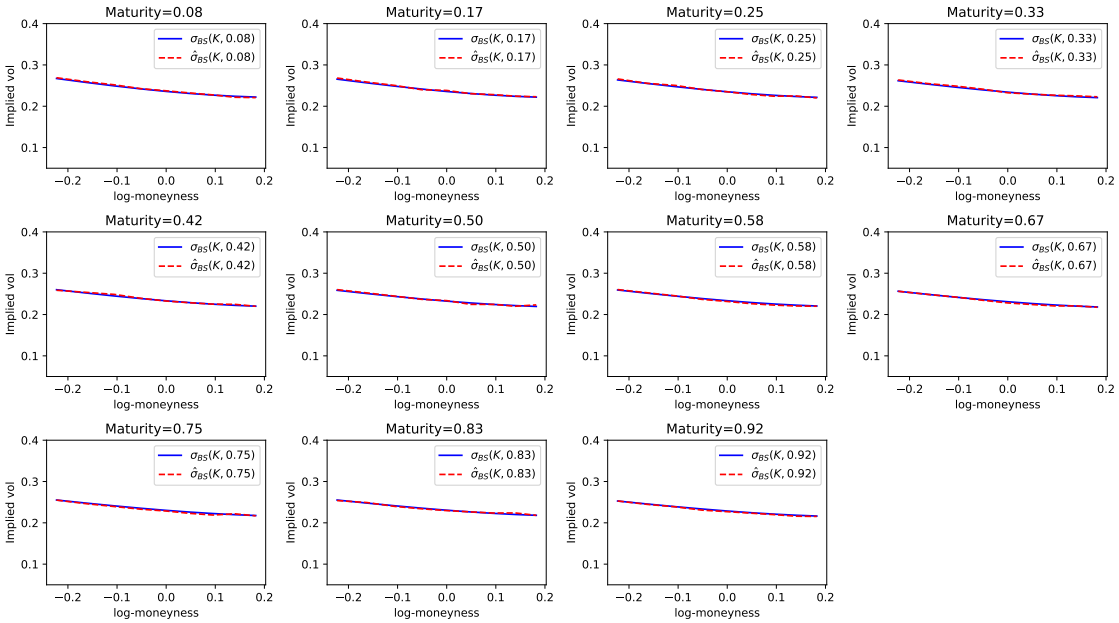
Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.52, \nu = 2.33, \sigma_0 = 0.0906, H = 0.095$.



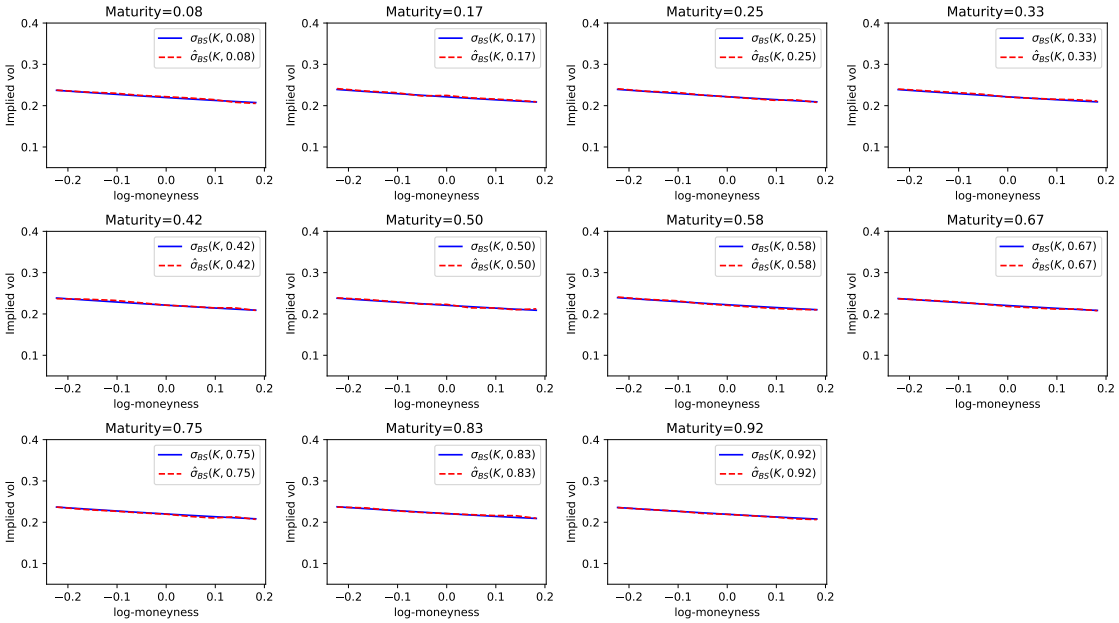
Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.45, \nu = 0.97, \sigma_0 = 0.0367, H = 0.379$.



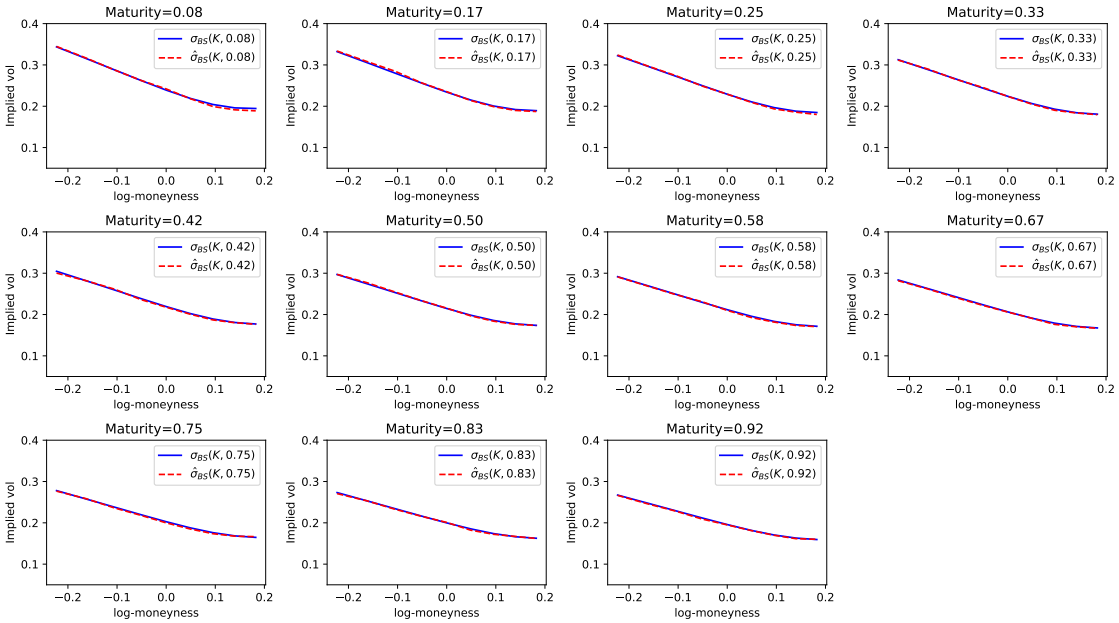
Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.42, \nu = 0.20, \sigma_0 = 0.0865, H = 0.401$.



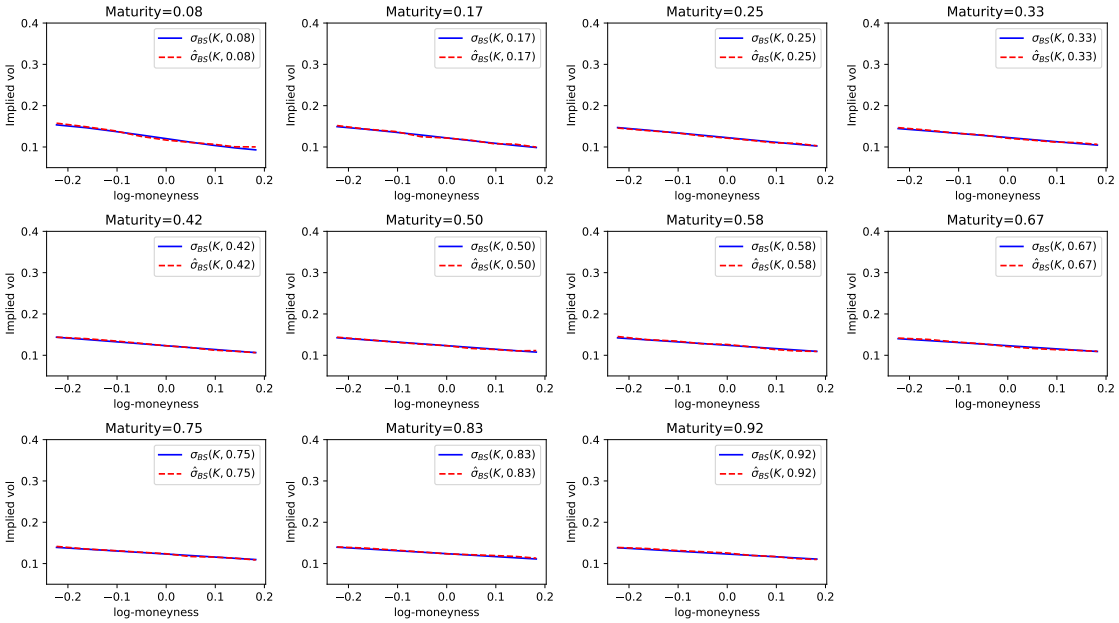
Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.40, \nu = 0.88, \sigma_0 = 0.0573, H = 0.382$.



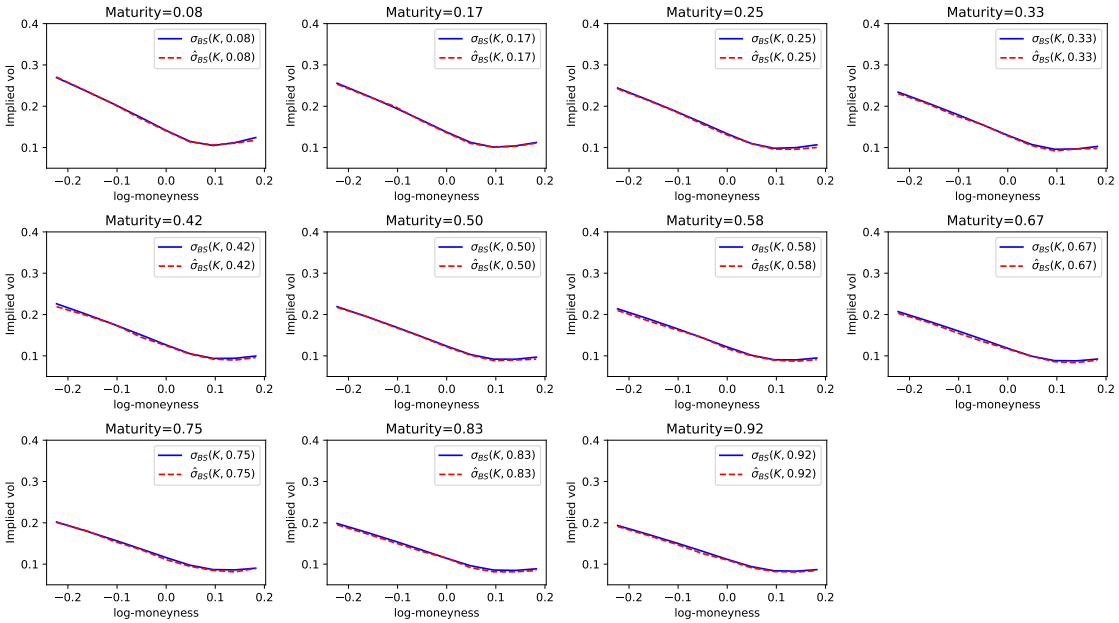
Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.55, \nu = 0.50, \sigma_0 = 0.0502, H = 0.43$.



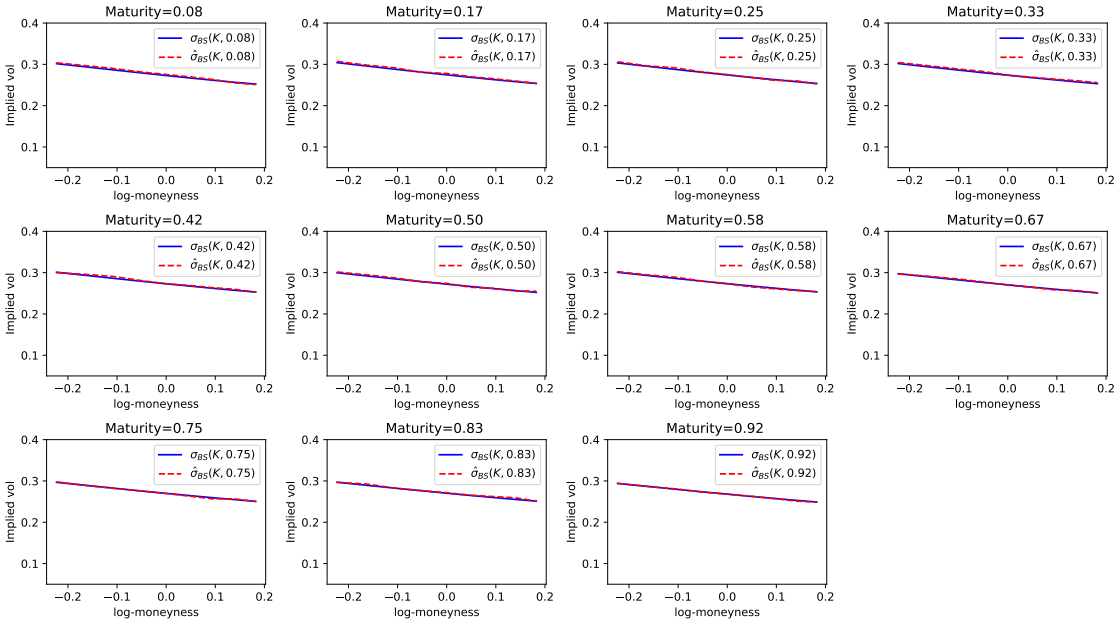
Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.67, \nu = 2.10, \sigma_0 = 0.0652, H = 0.36$.



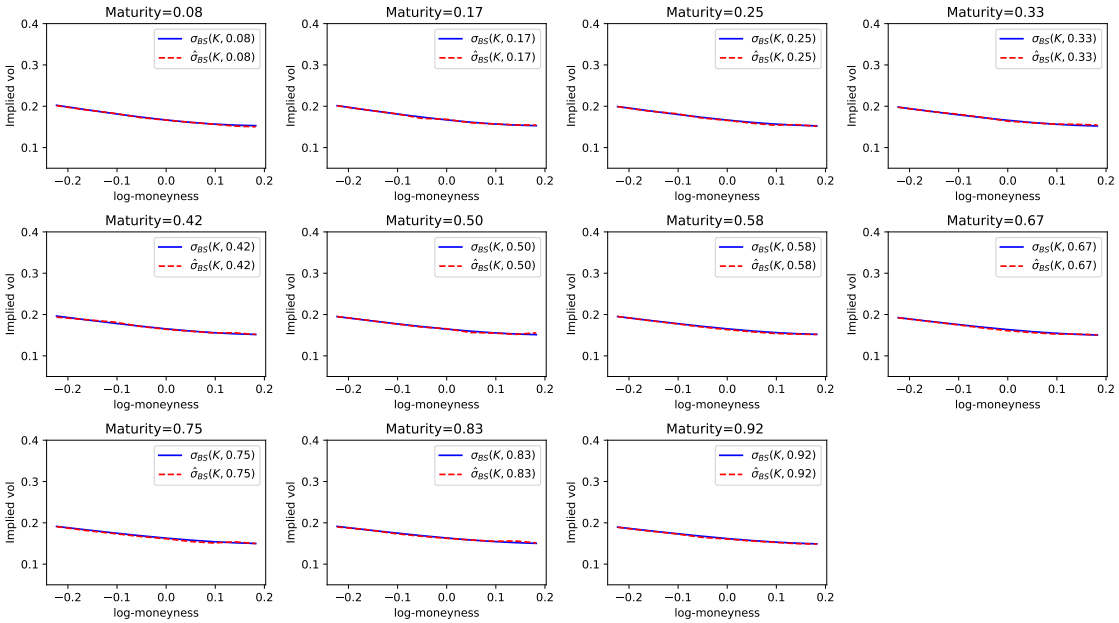
Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.76, \nu = 0.36, \sigma_0 = 0.0156, H = 0.11$.



Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.77, \nu = 2.32, \sigma_0 = 0.0239, H = 0.33$.

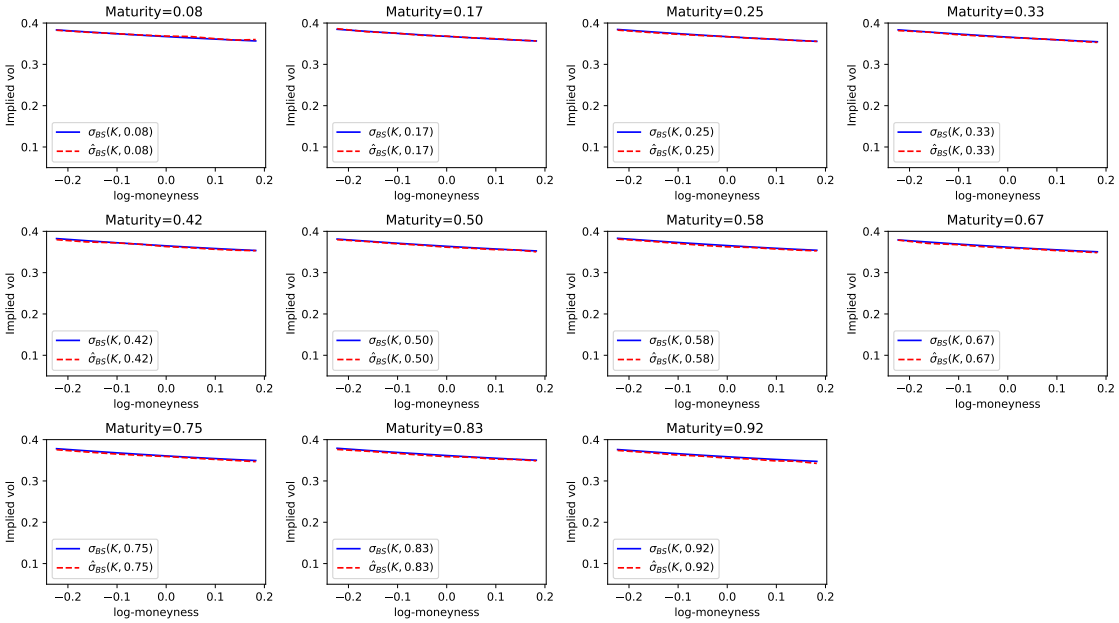


Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.62, \nu = 0.73, \sigma_0 = 0.0782, H = 0.42$.

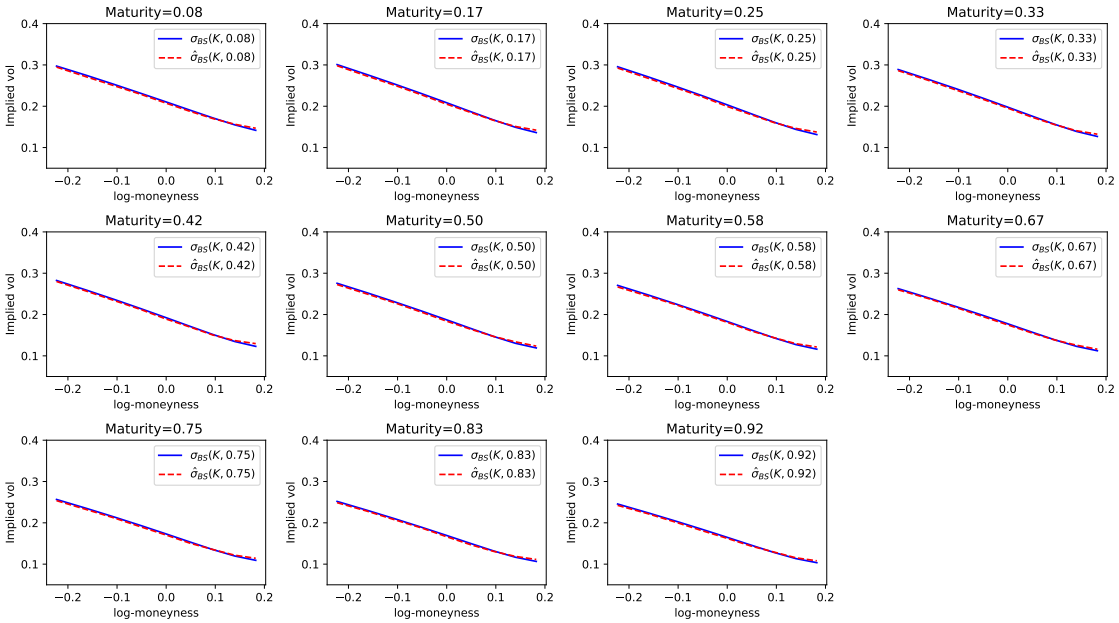


Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.46, \nu = 0.89, \sigma_0 = 0.0287, H = 0.39$.

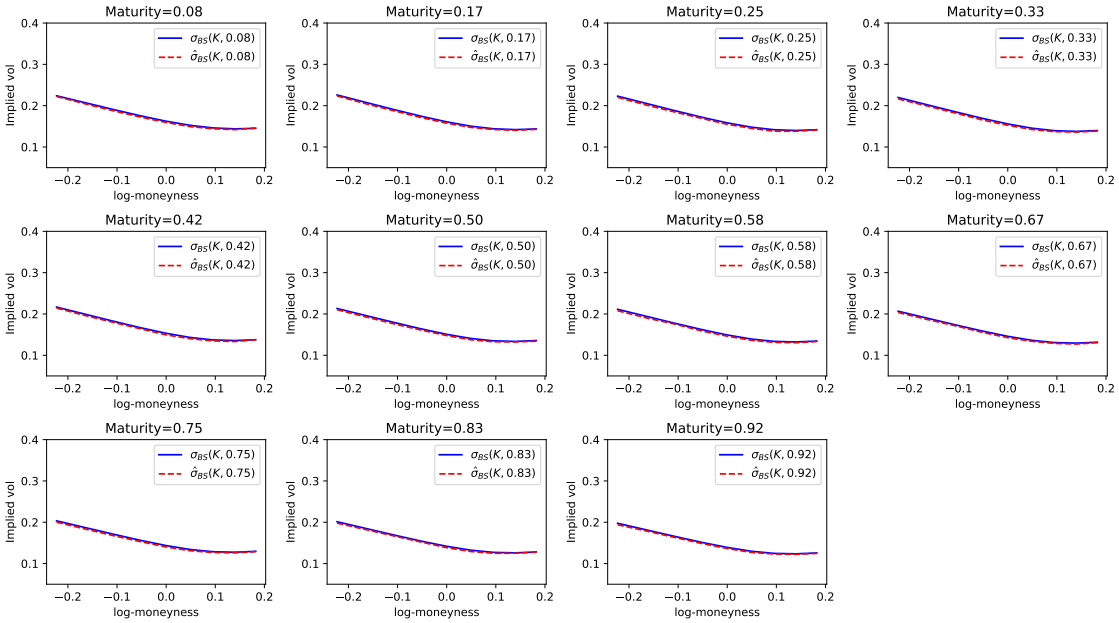
Figure 6: Approximation of implied volatility surfaces created by the rough Bergomi model. Parameters are uniformly sampled according to the bounds defined in **??**. Maturities are shown in



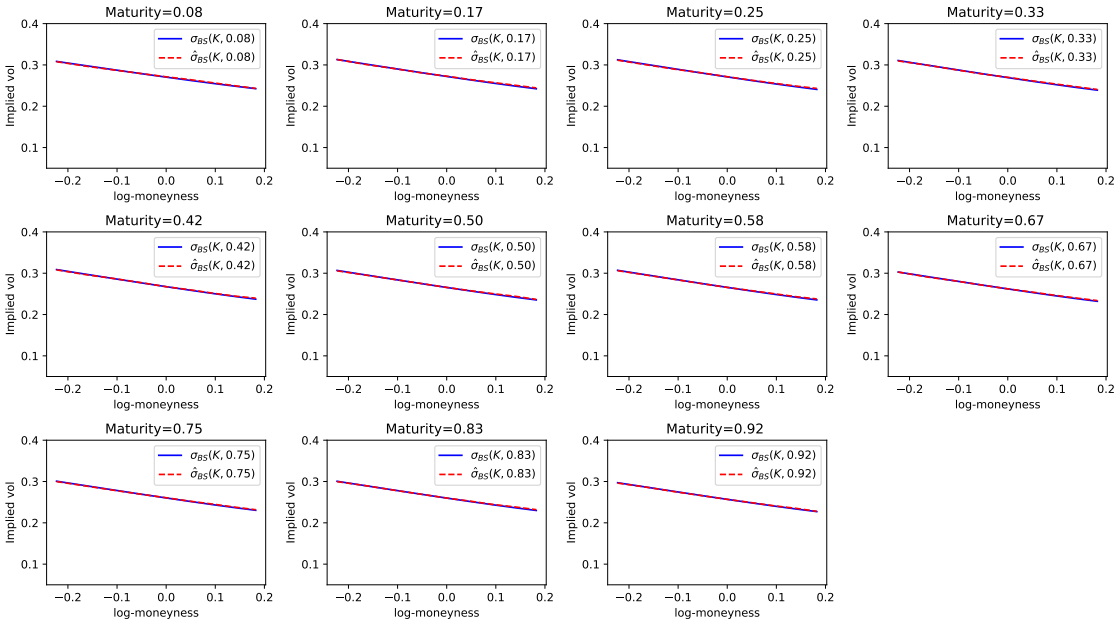
Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.41, \nu = 0.72, \sigma_0 = 0.138$.



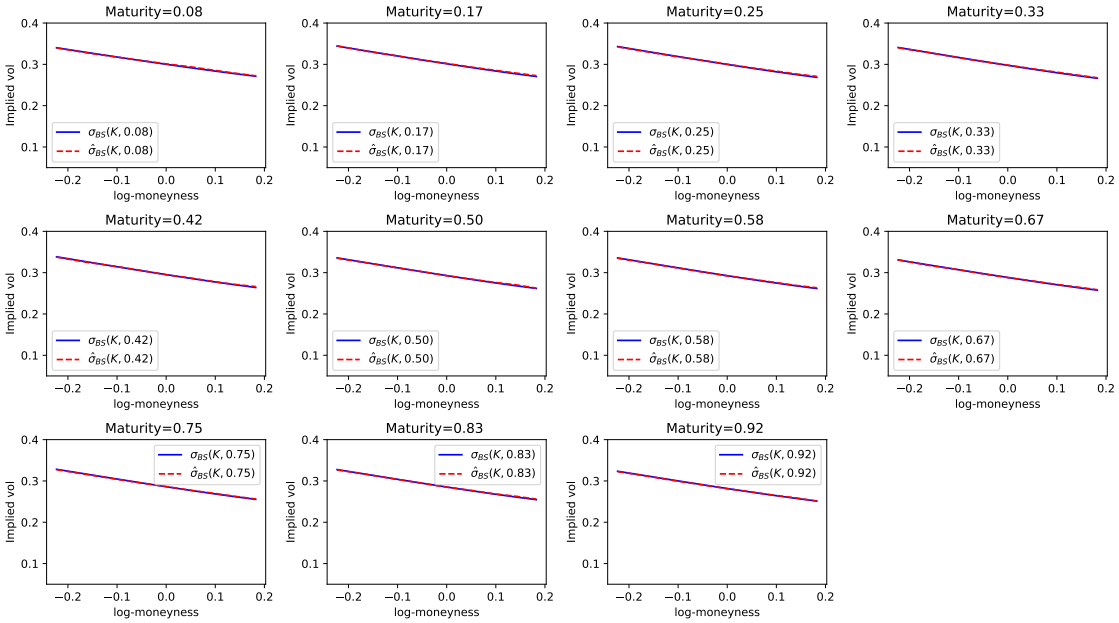
Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.88, \nu = 2.26, \sigma_0 = 0.050$.



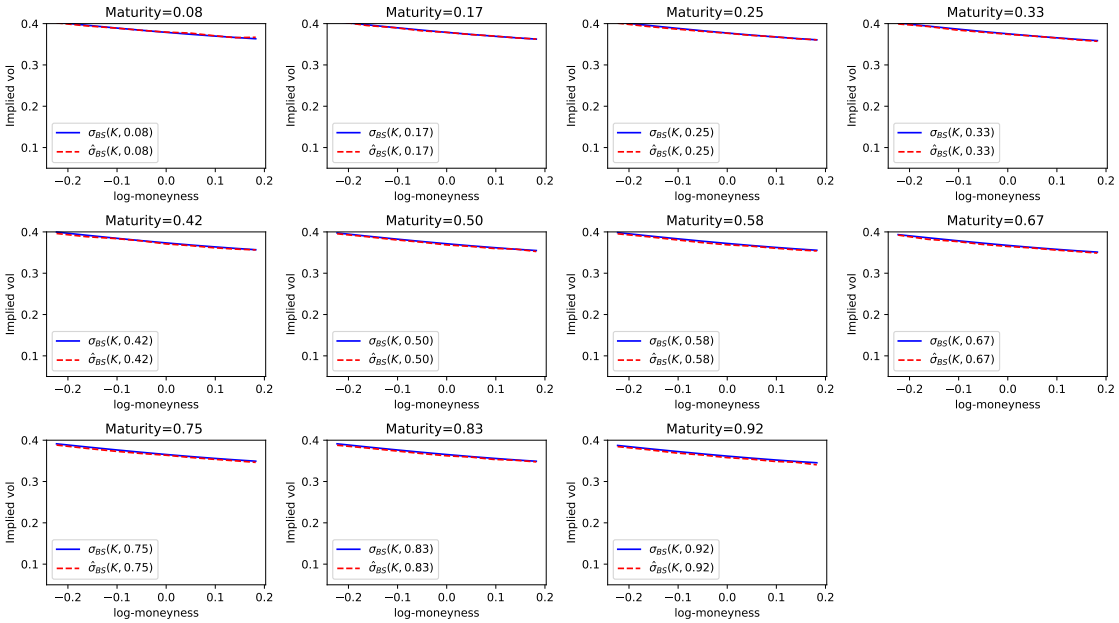
Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.55, \nu = 1.97, \sigma_0 = 0.028$.



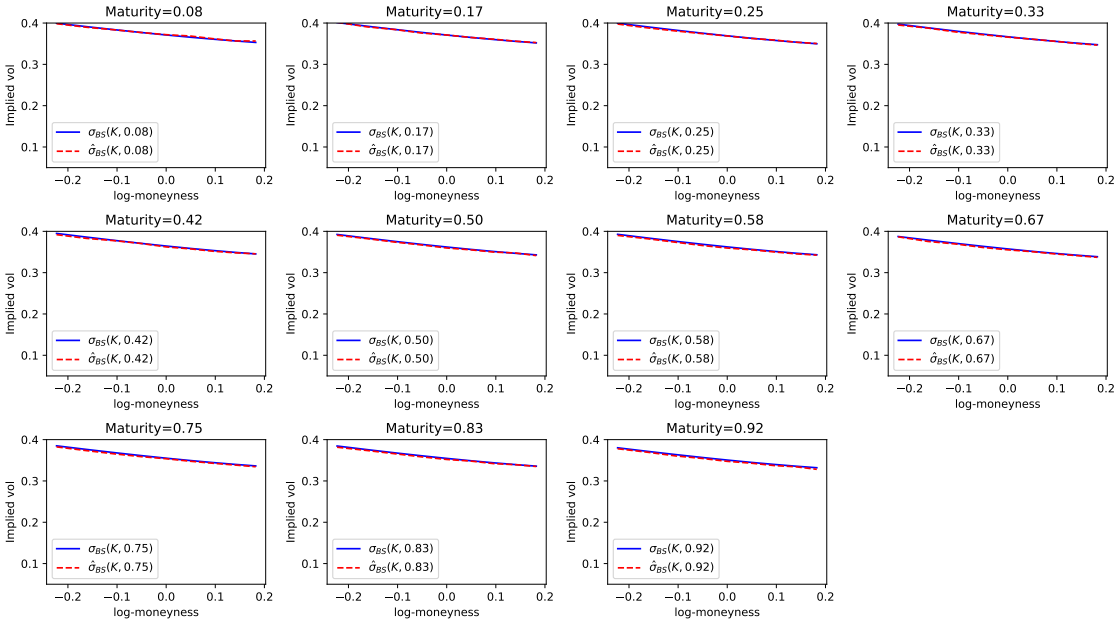
Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.70, \nu = 1.09, \sigma_0 = 0.078$.



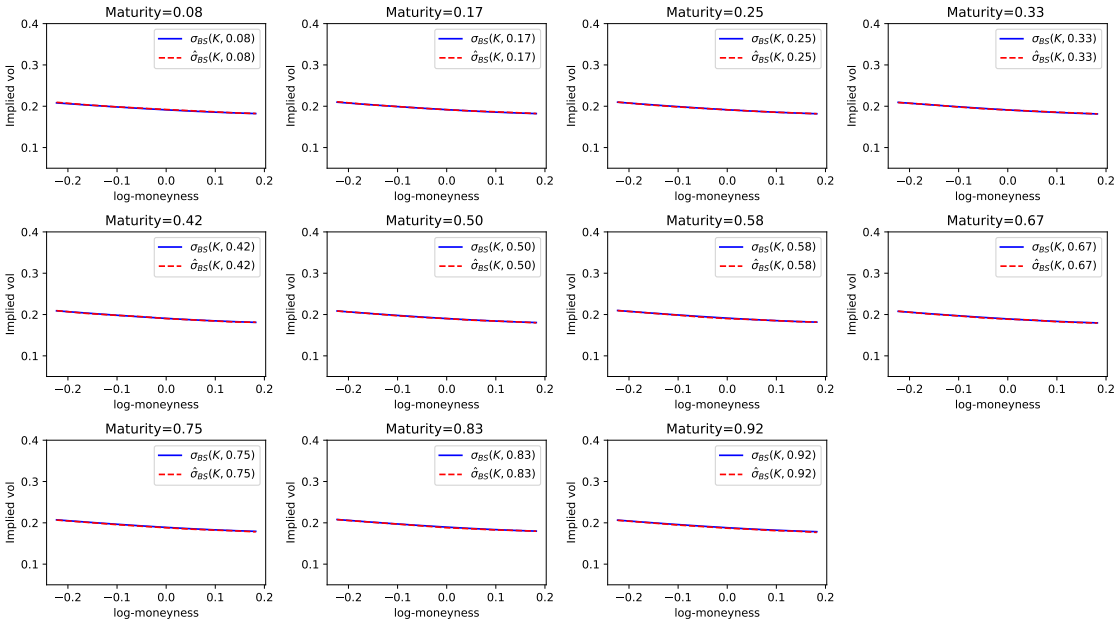
Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.68, \nu = 1.17, \sigma_0 = 0.096$.



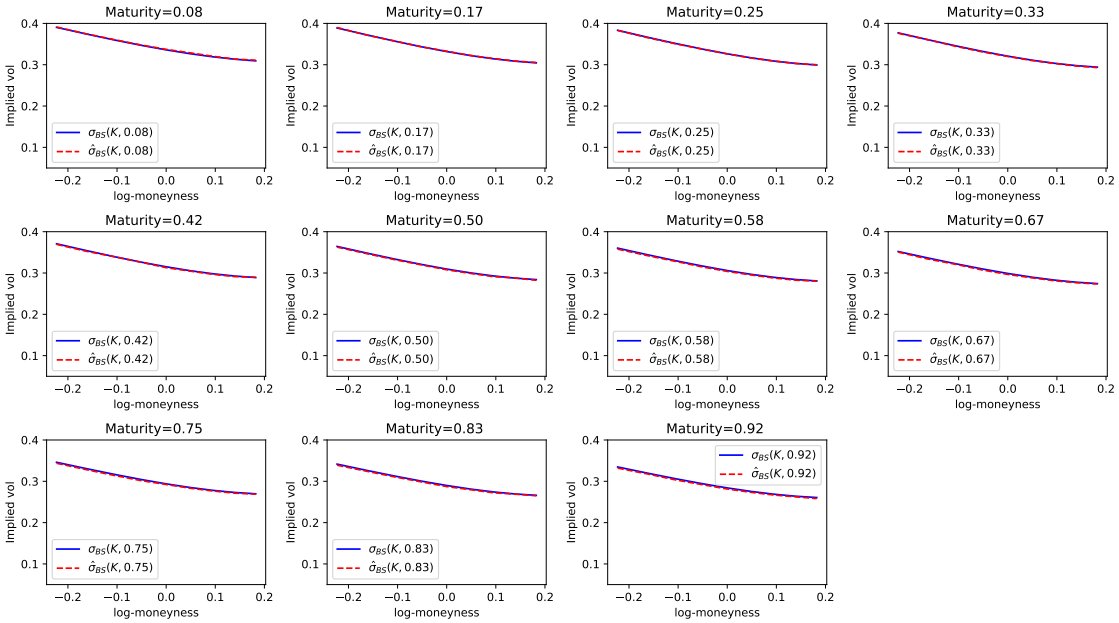
Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.45, \nu = 0.99, \sigma_0 = 0.148$.



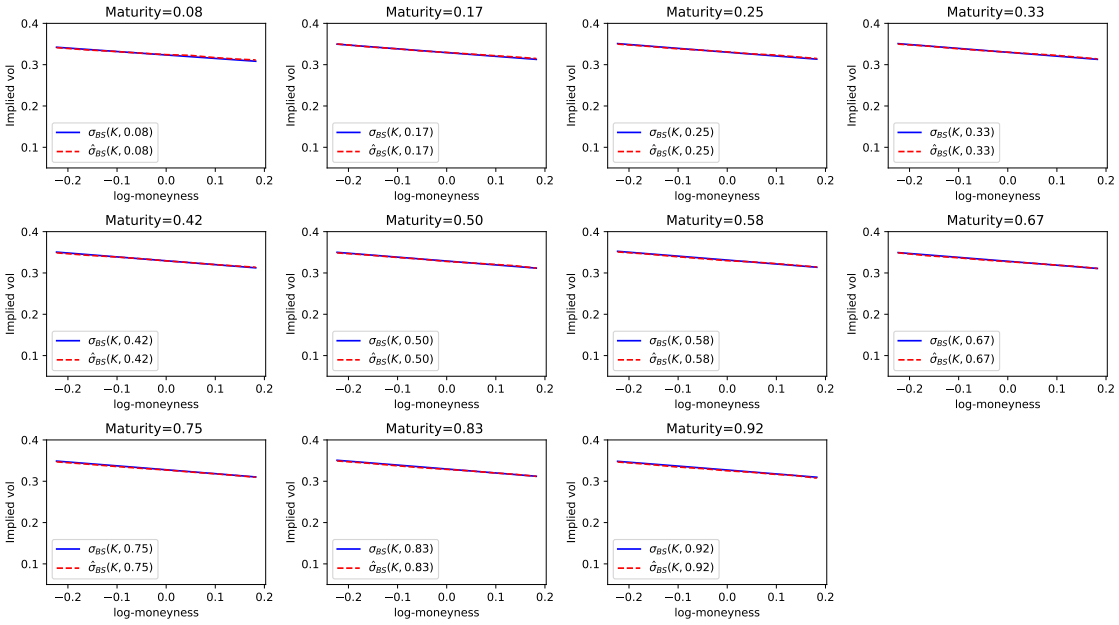
Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.48, \nu = 1.09, \sigma_0 = 0.142$.



Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.41, \nu = 0.69, \sigma_0 = 0.037$.

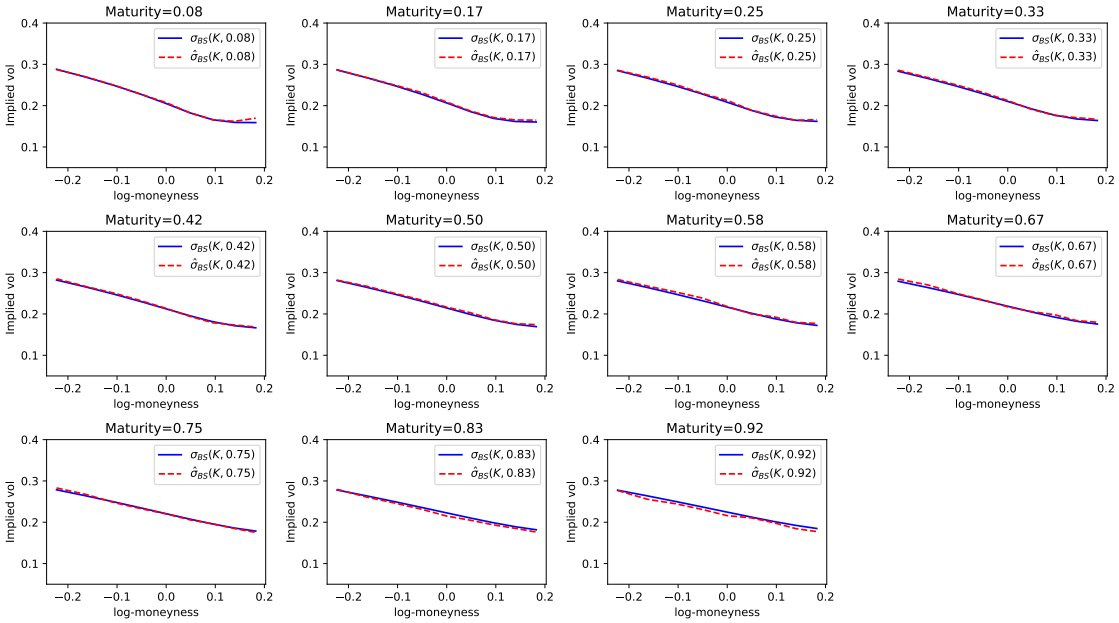


Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = 0.48, \nu = 1.95, \sigma_0 = 0.119$.

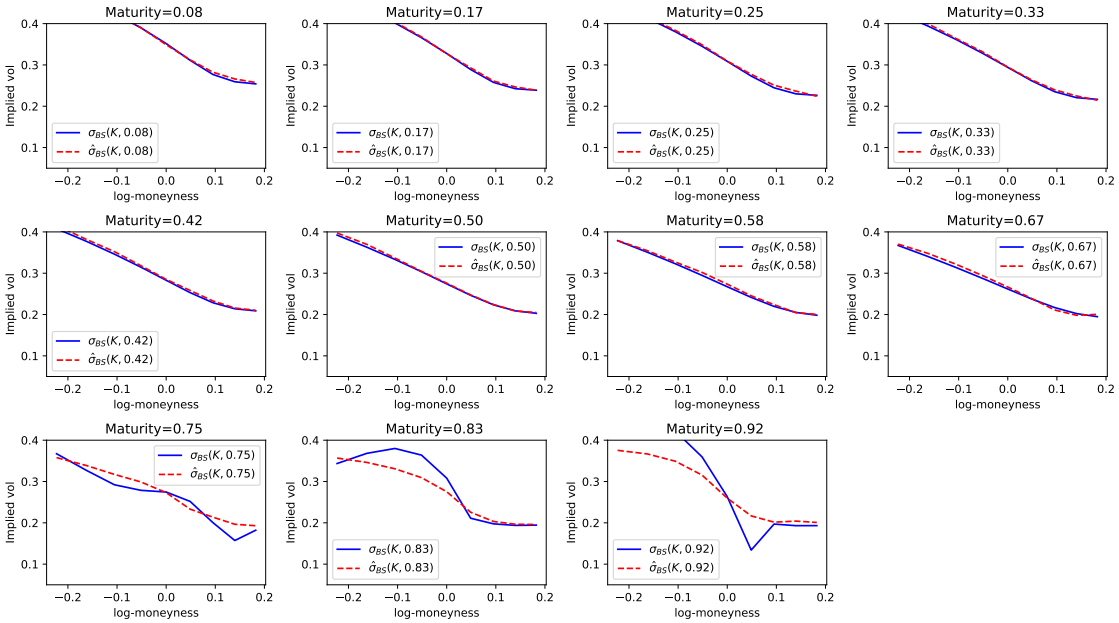


Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = 0.84, \nu = 0.46, \sigma_0 = 0.0114$.

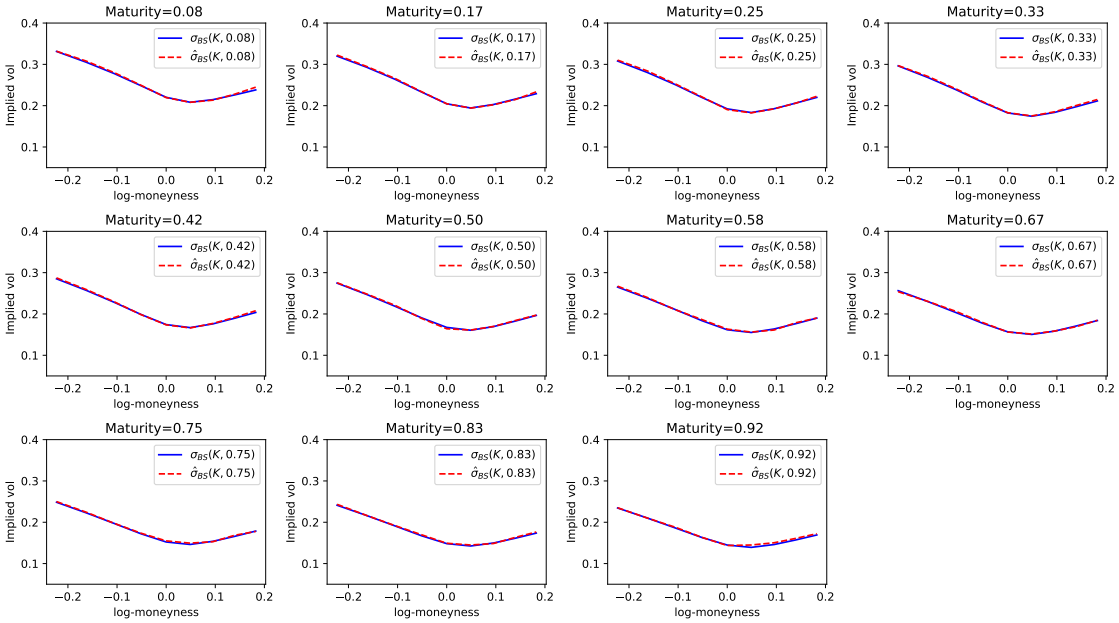
Figure 7: Approximation of implied volatility surfaces created by the Bergomi model. Parameters are uniformly sampled according to the bounds defined in **??**. Maturities are shown in yearly terms and the initial value of the spot process is 1.0.



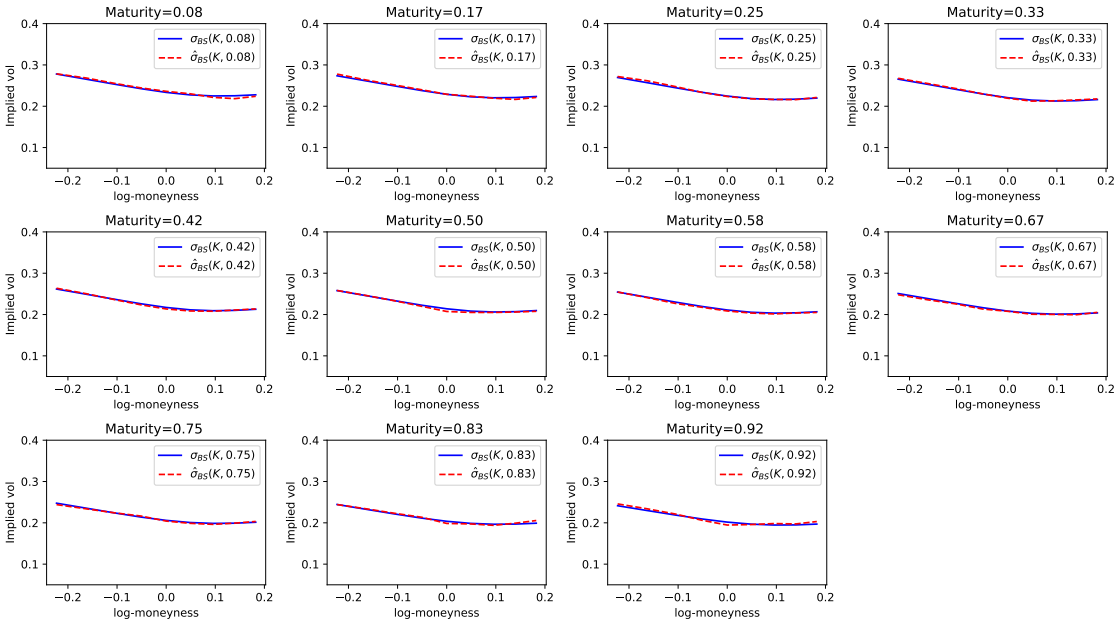
Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 1.25, \rho = -0.71, \sigma = 0.55, \theta = 0.092, V_0 = 0.0415$.



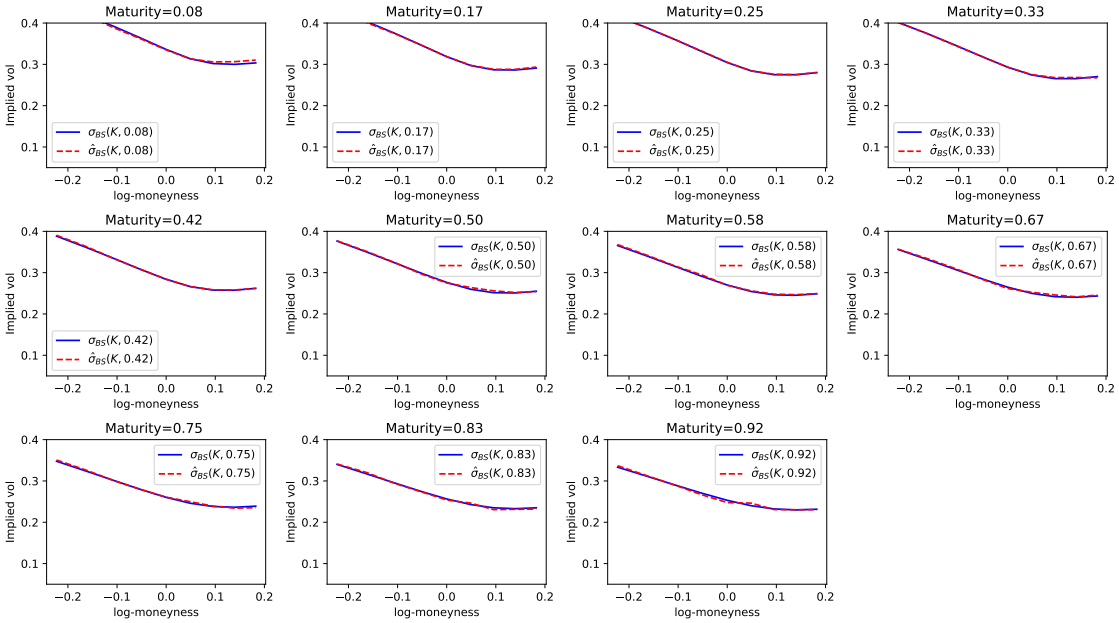
Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 1.75, \rho = -0.76, \sigma = 1.47, \theta = 0.088, V_0 = 0.1428$



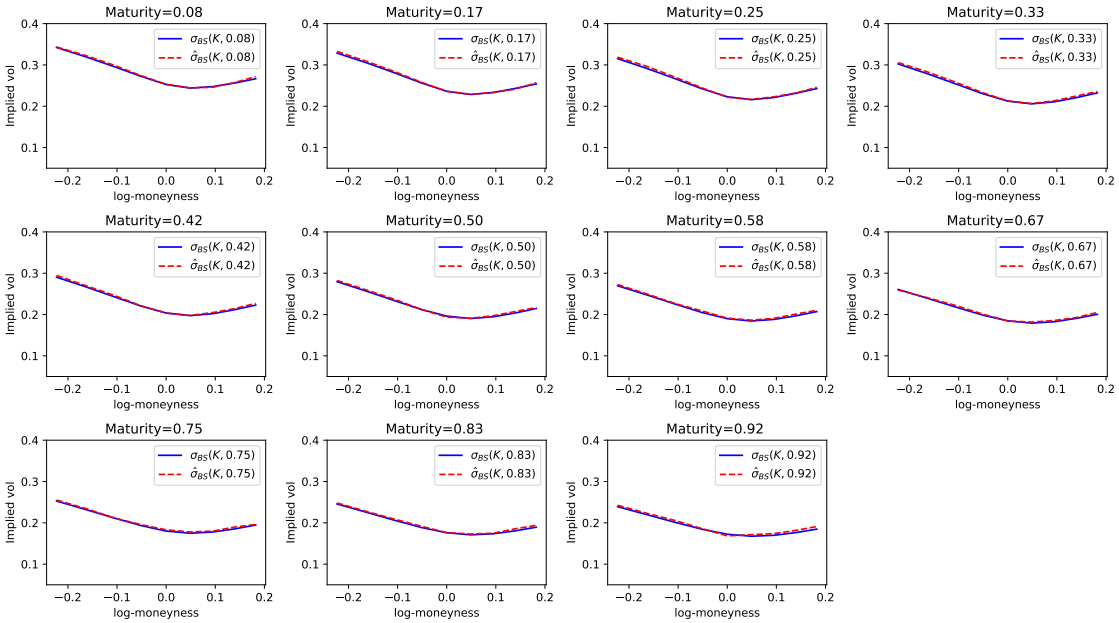
Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 0.74, \rho = -0.38, \sigma = 1.05, \theta = 0.017, V_0 = 0.0570$.



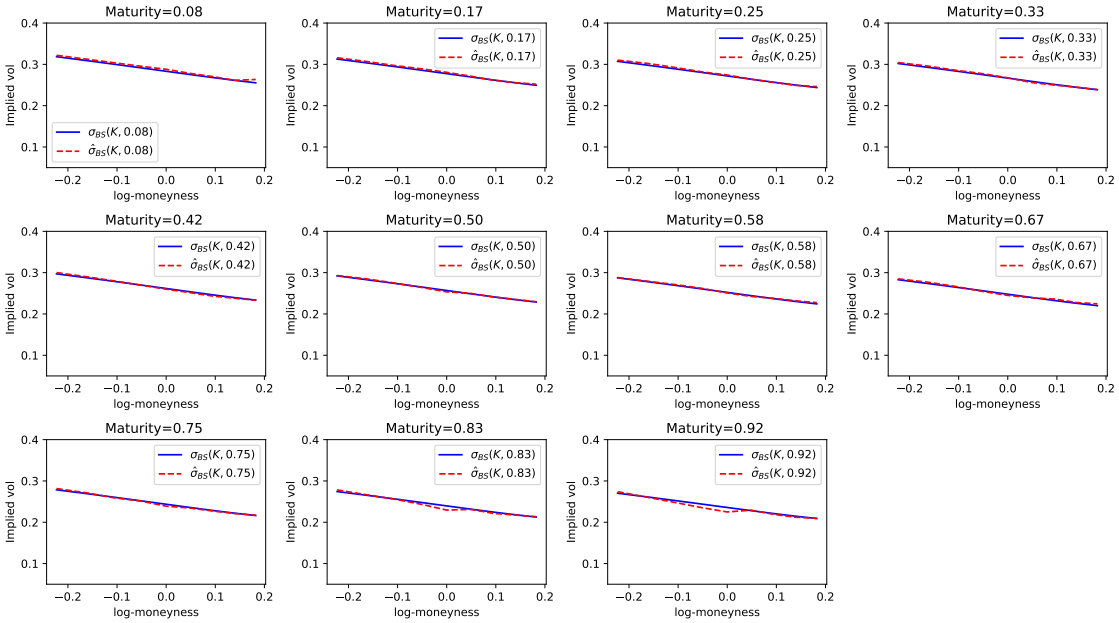
Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 0.91, \rho = -0.31, \sigma = 0.46, \theta = 0.035, V_0 = 0.0568$.



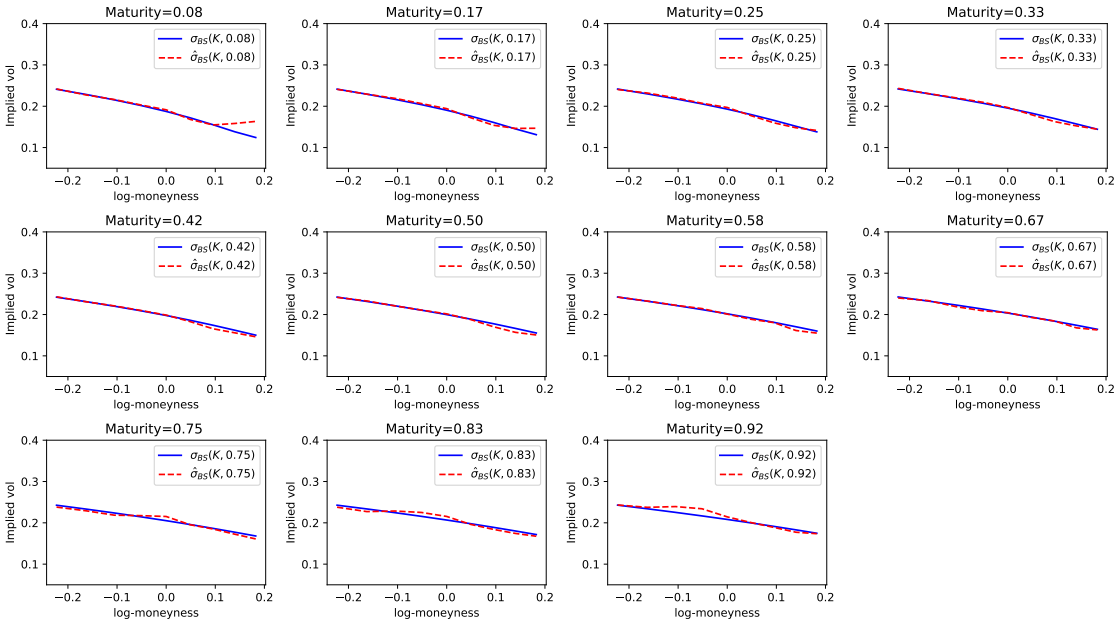
Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 0.93, \rho = -0.52, \sigma = 1.31, \theta = 0.094, V_0 = 0.1265$.



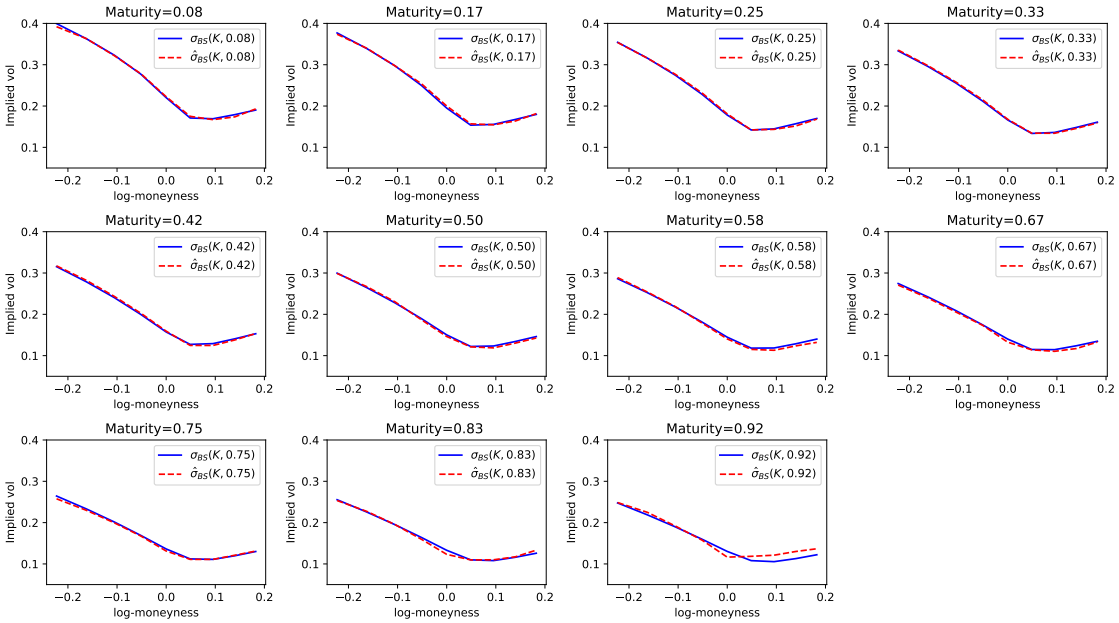
Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 1.62, \rho = -0.30, \sigma = 1.02, \theta = 0.022, V_0 = 0.0740$.



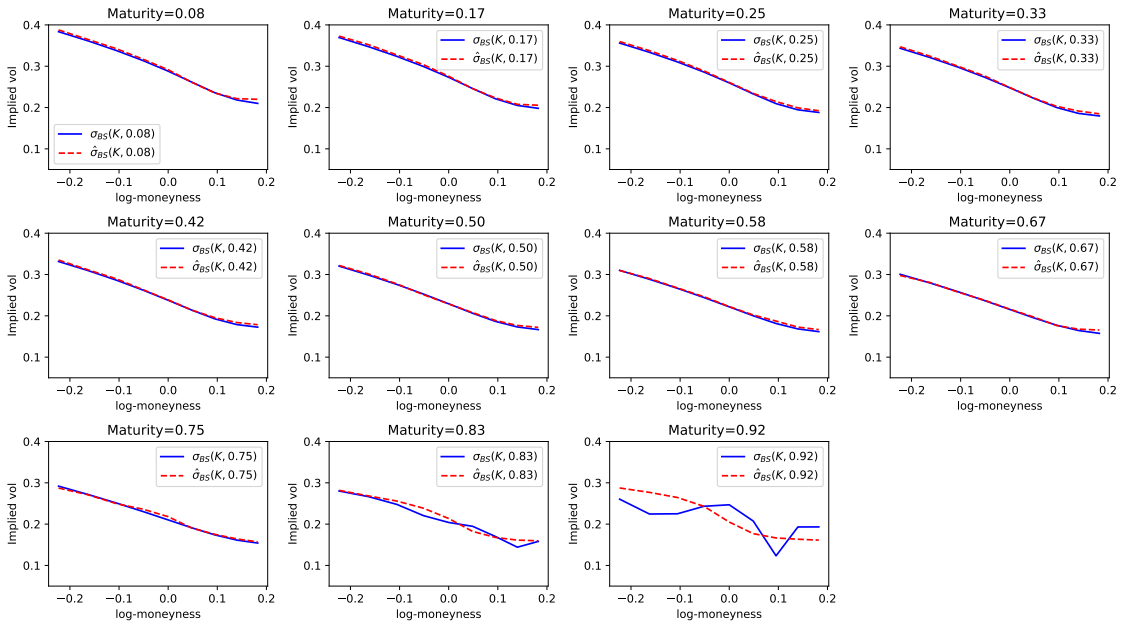
Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 1.02, \rho = -0.62, \sigma = 0.30, \theta = 0.020, V_0 = 0.0836$.



Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 1.50, \rho = -0.88, \sigma = 0.27, \theta = 0.062, V_0 = 0.0339$.



Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 0.99, \rho = -0.75, \sigma = 1.48, \theta = 0.033, V_0 = 0.0641$.

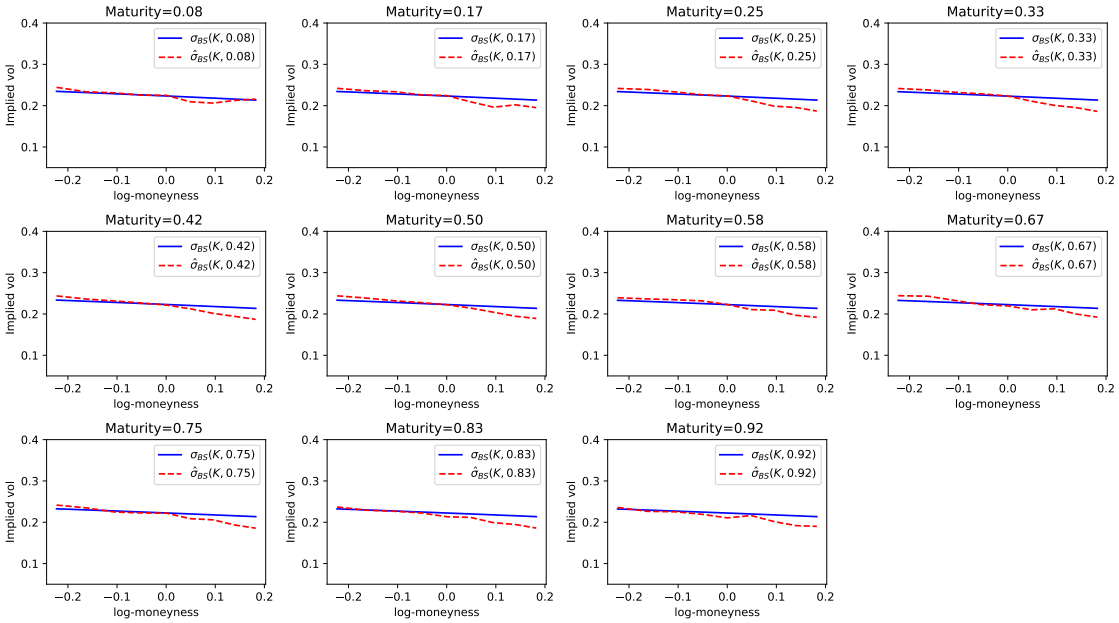


Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 1.97, \rho = -0.77, \sigma = 0.81, \theta = 0.032, V_0 = 0.0924$.

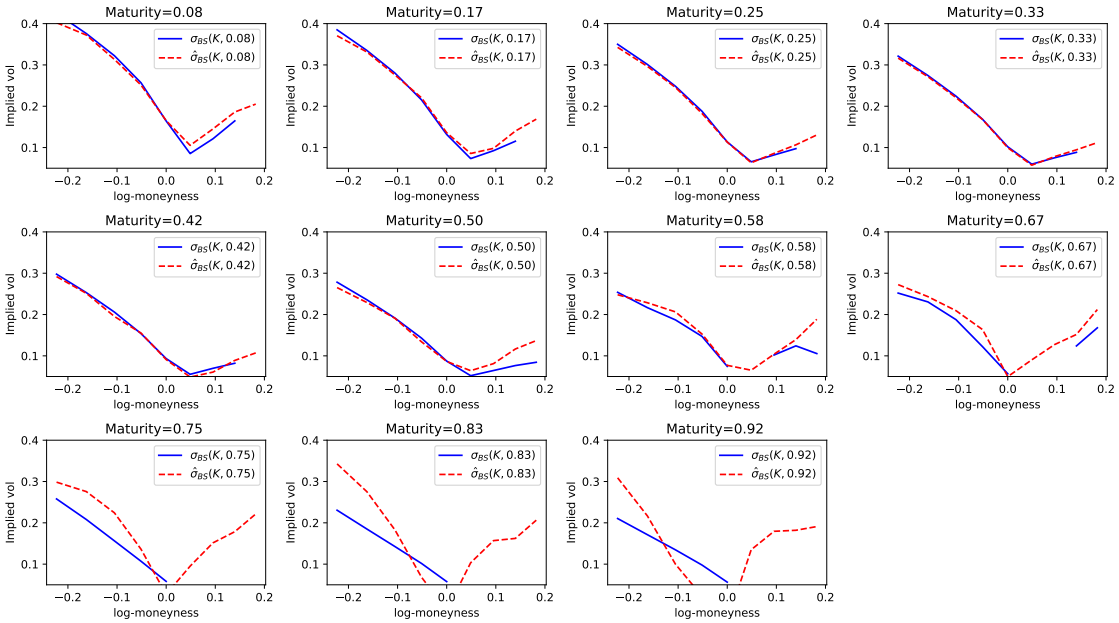
Figure 8: Approximation of implied volatility surfaces created by the Heston model. Parameters are uniformly sampled according to the bounds defined in Figure **??**. Maturities are shown in yearly terms and the initial value of the spot process is 1.0. Performance is poorest when the period to mean reversion is close to the upper sampling bound of 2. Given parameter values found in the literature and reported in Figure **??**, it is unlikely market prices would come from such a process. The Figures show that the fit worsens for values of κ close to the upper sampling bound of 2.0. We believe the fit is worse than in Figures 7 and 6 because of the curse of dimensionality. Since we use the same number of samples for all models and the Heston model has 5 parameters, the network trained to map Heston parameters to implied volatility surfaces has seen a much smaller volume of the parameter space than the previous models.

model	speedup
Heston	650
Bergomi	1950
rough Bergomi	2500

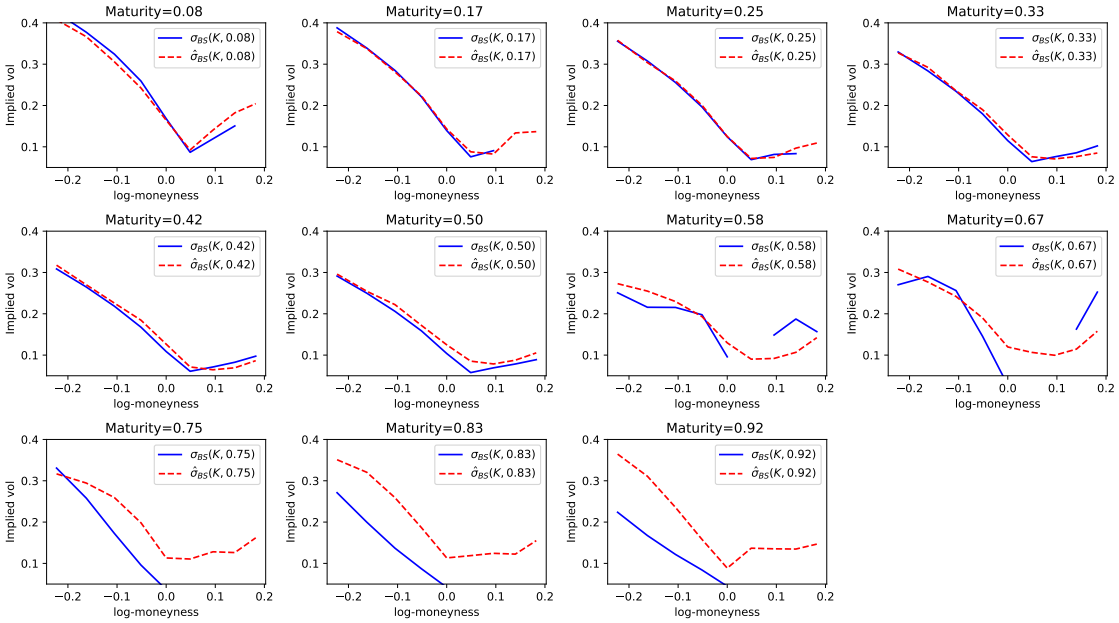
Figure 9: Speed up of computation of option prices compared to alternative methods. The speed up is computed by dividing the average computation time of the option price using the techniques presented in Section 3 by the average computation time of option prices via neural networks. There is a significant difference between the speed up for which model, which is natural given the Heston model's option prices can be computed using an analytical formula, while we must generate paths for the Bergomi and rough Bergomi models. The speed-ups are larger than those obtained in [4] (p.10) by a more than a hundred fold. This is surprising given there should be less operations in [4]. Our computation of option prices may have been smaller than the one used in [4], which would explain the difference in speed-up.



(a) Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 0.5, \rho = -0.95, \sigma = 0.05, \theta = 0.05, V_0 = 0.05$

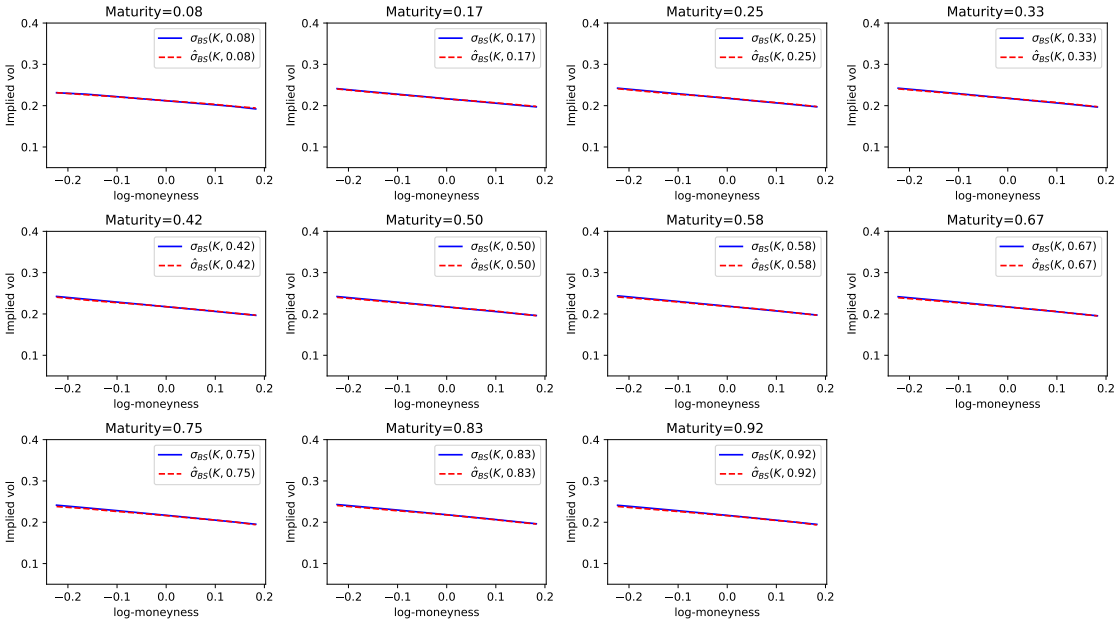


(b) Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 0.2, \rho = -0.95, \sigma = 2.0, \theta = 0.05, V_0 = 0.05$ - 0.95,0.2,2.0,0.05,0.05

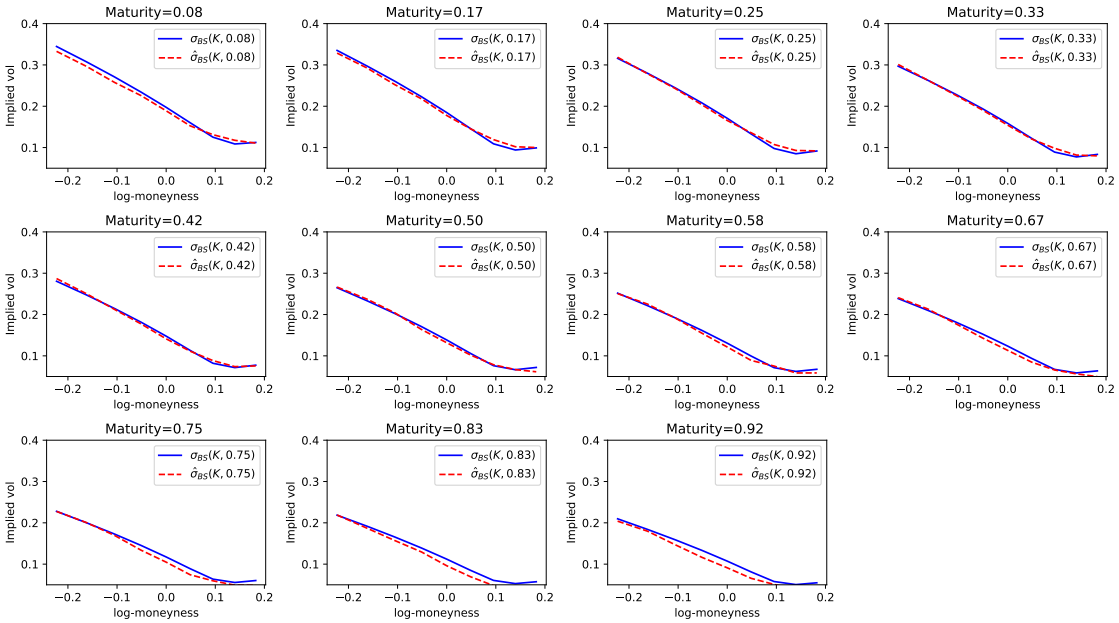


(c) Comparison of volatility surface from the Heston model computed using a neural network and the Monte-Carlo algorithm with parameters $\kappa = 0.2, \rho = -0.95, \sigma = 2.0, \theta = 0.2, V_0 = 0.05$

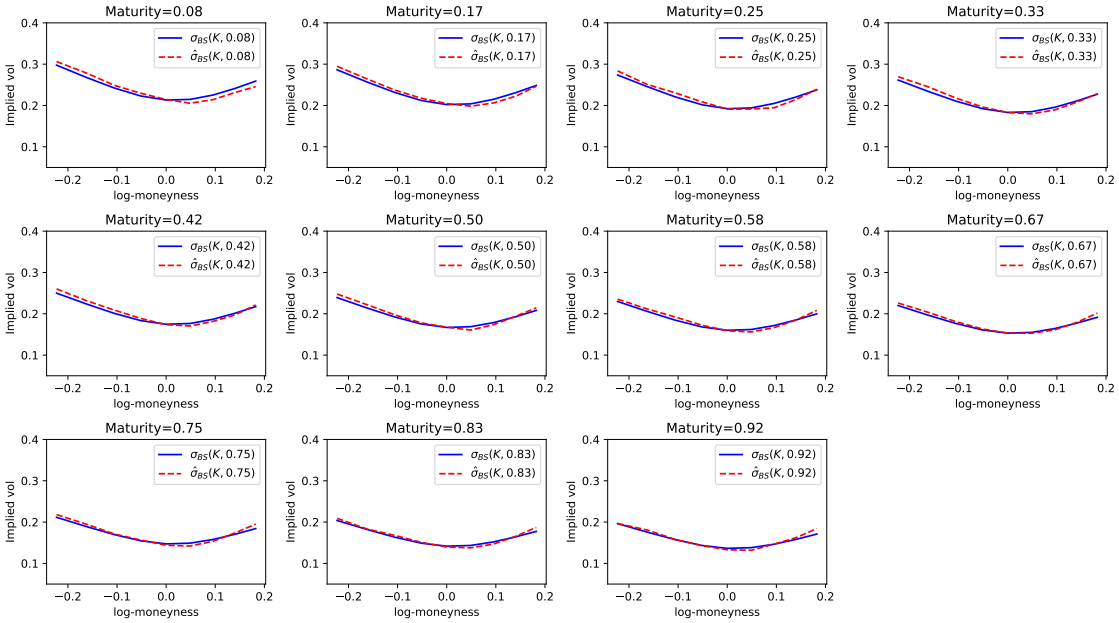
Figure 10: Heston approximation with parameters outside of sampling range.



(a) Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.95, \nu = 0.5, \sigma_0 = 0.05$

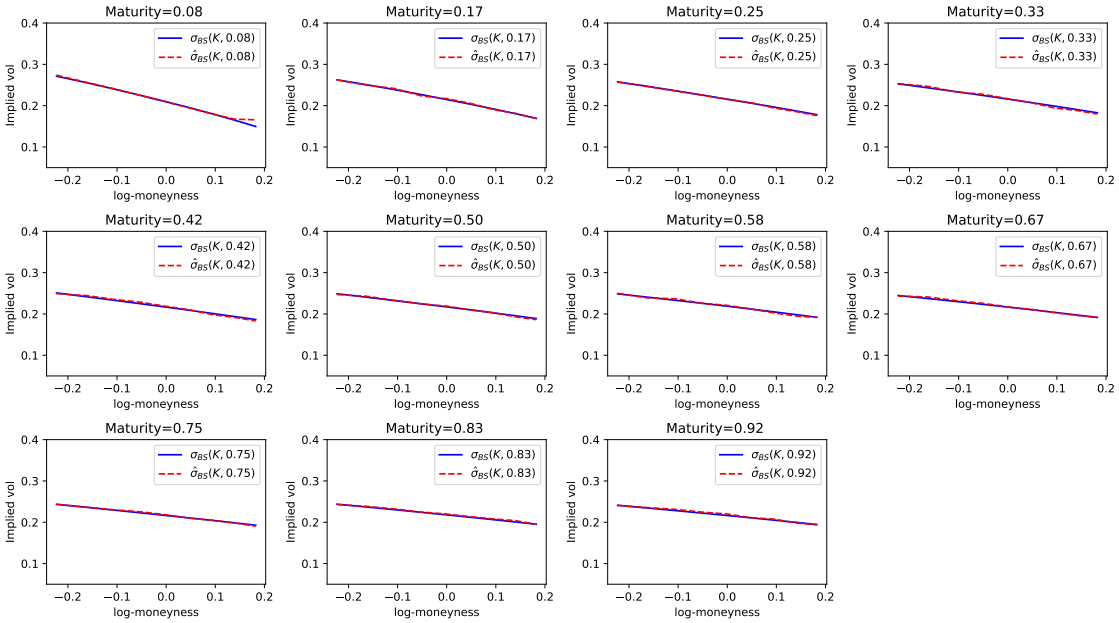


(b) Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.95, \nu = 4.0, \sigma_0 = 0.05$

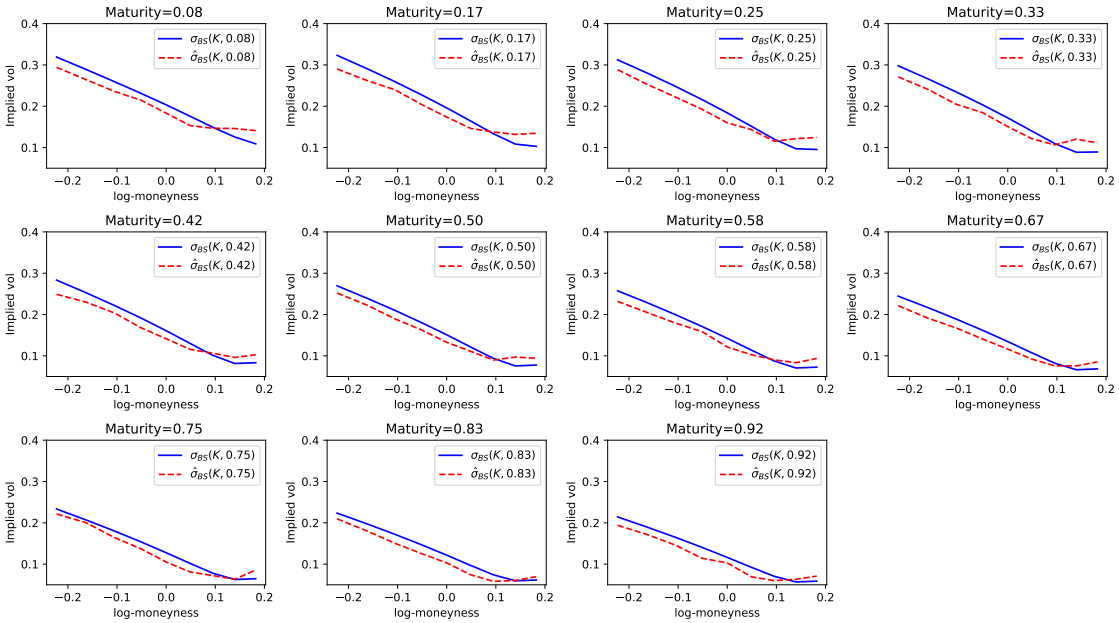


(c) Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.1, \nu = 4.0, \sigma_0 = 0.05$

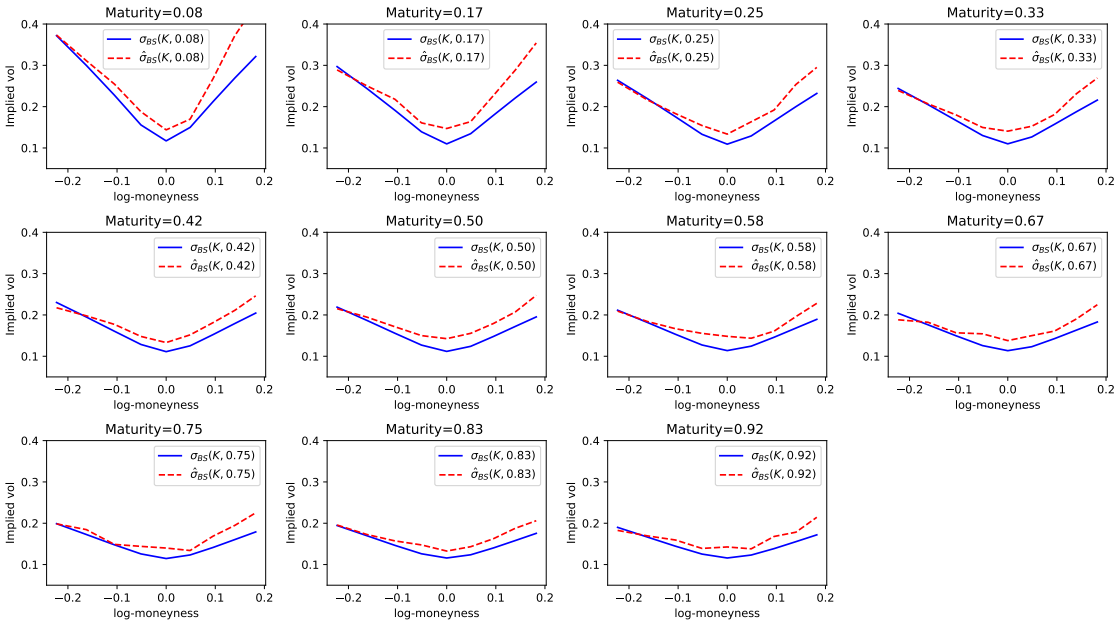
Figure 11: Bergomi approximation with parameters outside of sampling range.



(a) Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.95, \nu = 0.5, \sigma_0 = 0.05, H = 0.10$.



(b) Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.95, \nu = 4.0, \sigma_0 = 0.05, H = 0.60$.



(c) Comparison of volatility surface from the rough Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.1, \nu = 4.0, \sigma_0 = 0.05, H = 0.02$.

Figure 12: rough Bergomi approximation with parameters outside of sampling range. Even for parameters outside of the sampling range, the neural network provides a good fit to the implied volatility surface.

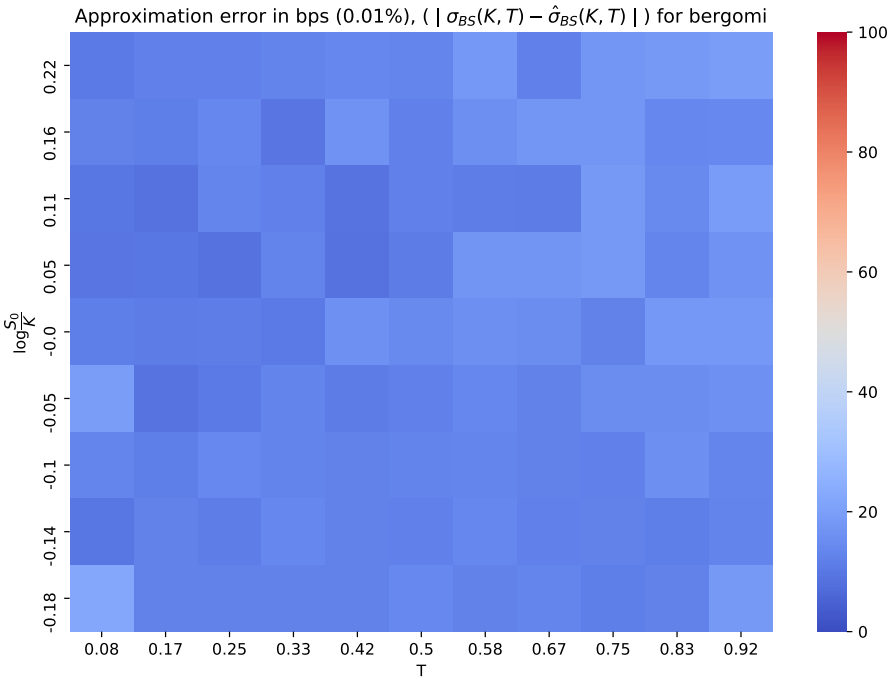


Figure 13: Approximation error between approximation and truth, averaged across samples generated with randomly sampled parameters for the Bergomi model. 1000 parameters were sampled according to bounds in figure **??**. Implied volatility surfaces were computed using the Monte Carlo algorithm and compared to surfaces produced by the neural network. Overall performance is uniform, which indicates the network is approximating the whole surface correctly.

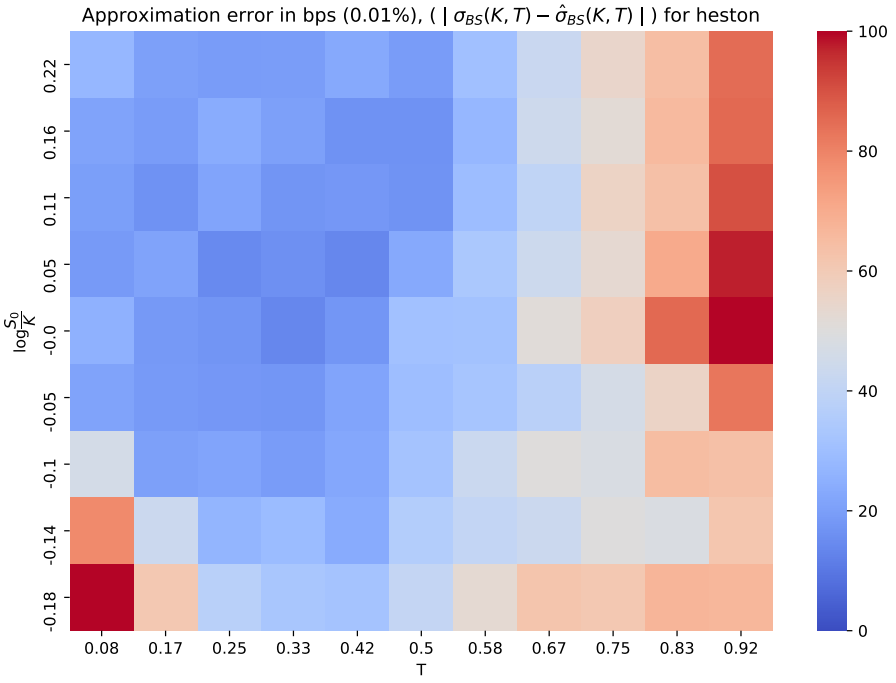


Figure 14: Approximation error between approximation and truth, averaged across samples generated with randomly sampled parameters for the Heston model. 1000 parameters were sampled according to bounds in figure **??**. It seems performance is worse for out-of-the-money options and longer tenors. It may be because long term options are determined by κ, θ, V_0 instead of a single parameter. It is surprising that for short term, in-the-money options the network exhibits good performance whereas it has poor performance on short term out-of-the-money options.

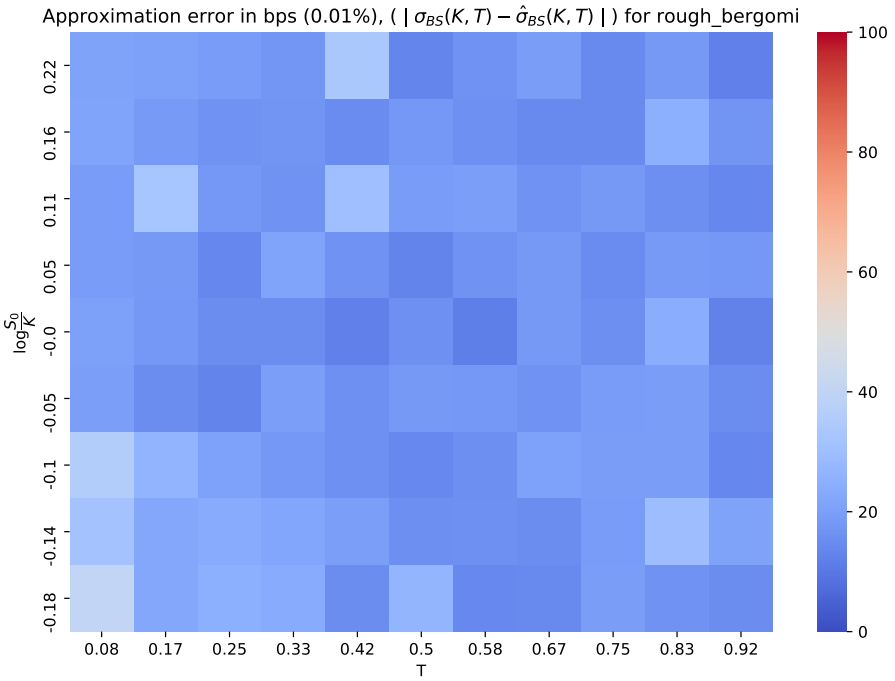


Figure 15: Approximation error between approximation and truth, averaged across samples generated with randomly sampled parameters for the rough Bergomi model. 1000 parameters were sampled according to bounds in figure **??**. Implied volatility surfaces were computed using the Monte Carlo algorithm and compared to surfaces produced by the neural network. Overall performance is uniform, which indicates the network is approximating the whole surface correctly.

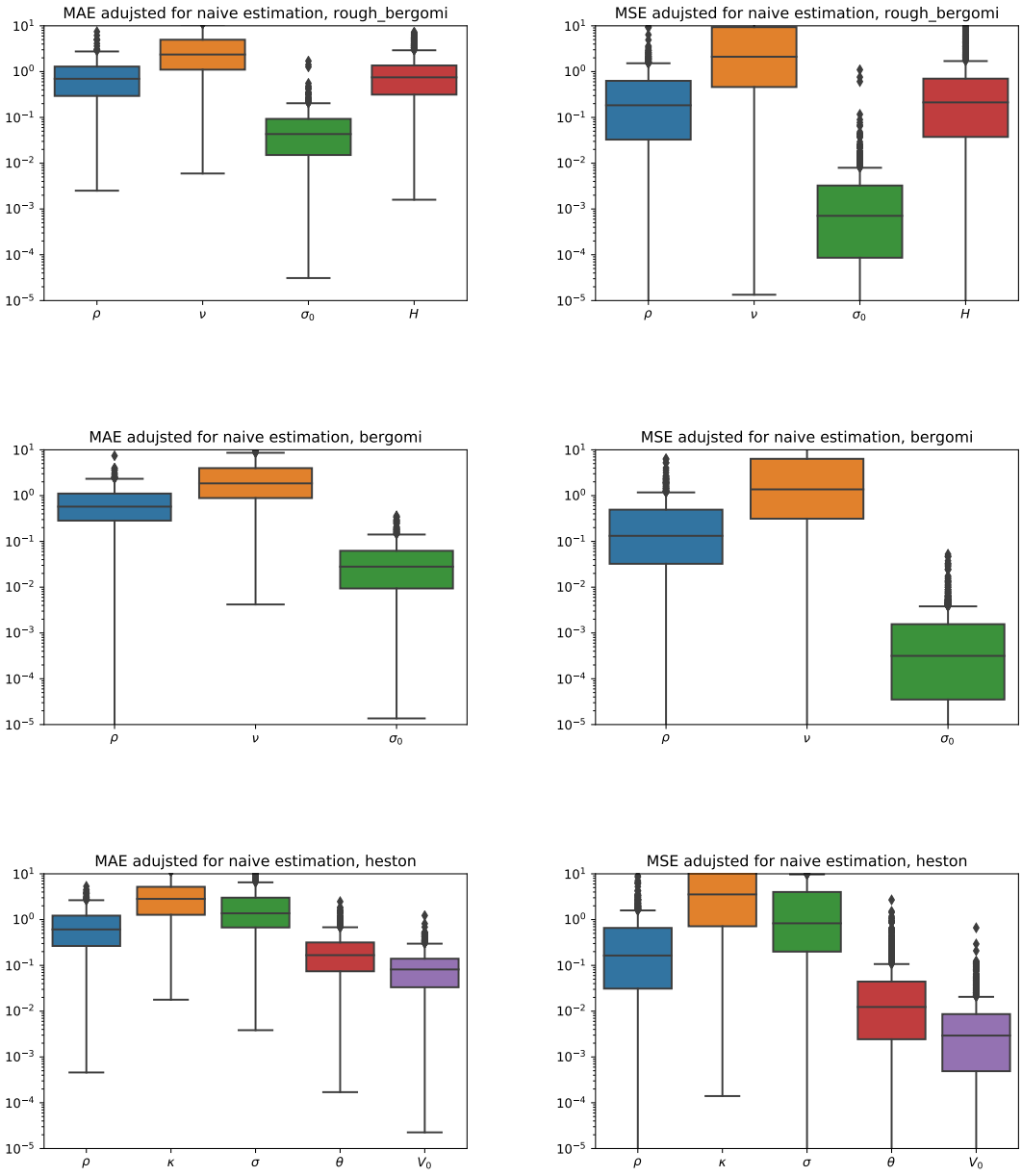
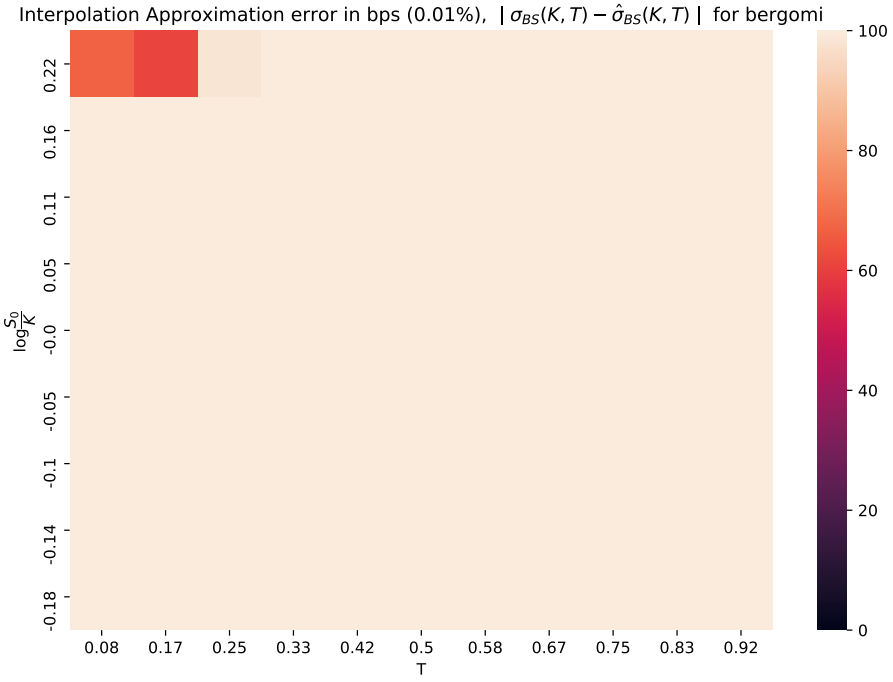
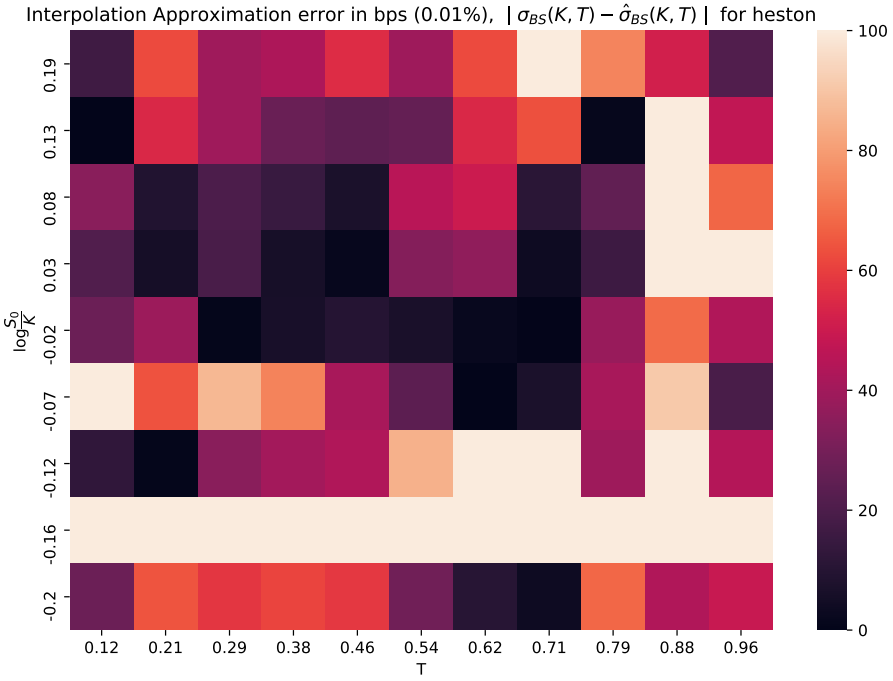


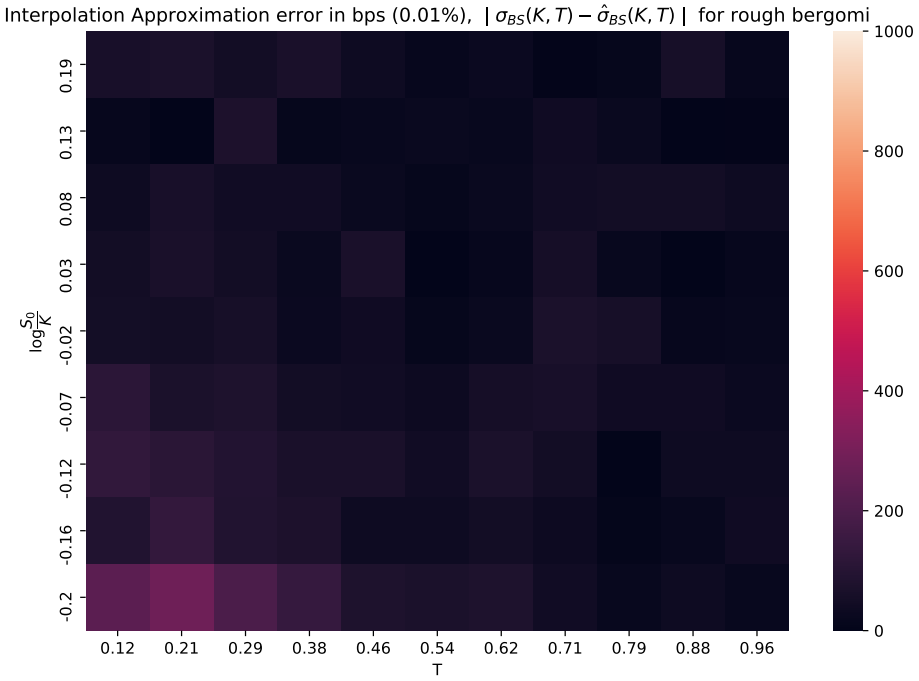
Figure 16: Approximation error between estimated parameters and truth, averaged across samples generated with randomly sampled parameters. Since we use a uniform distribution to sample parameters, the guess that minimises the mean absolute error and mean squared error is simply the middle point of the upper and lower bounds. The resulting absolute and squared errors are used as reference points. If the mean error is below this threshold then the calibration works better than the random guess. The presented error was normalised by this quantity, therefore if it is smaller than 1 then the calibration improves on the random guess.



Error heat map for the Bergomi model.



Error heat map for the Heston model.

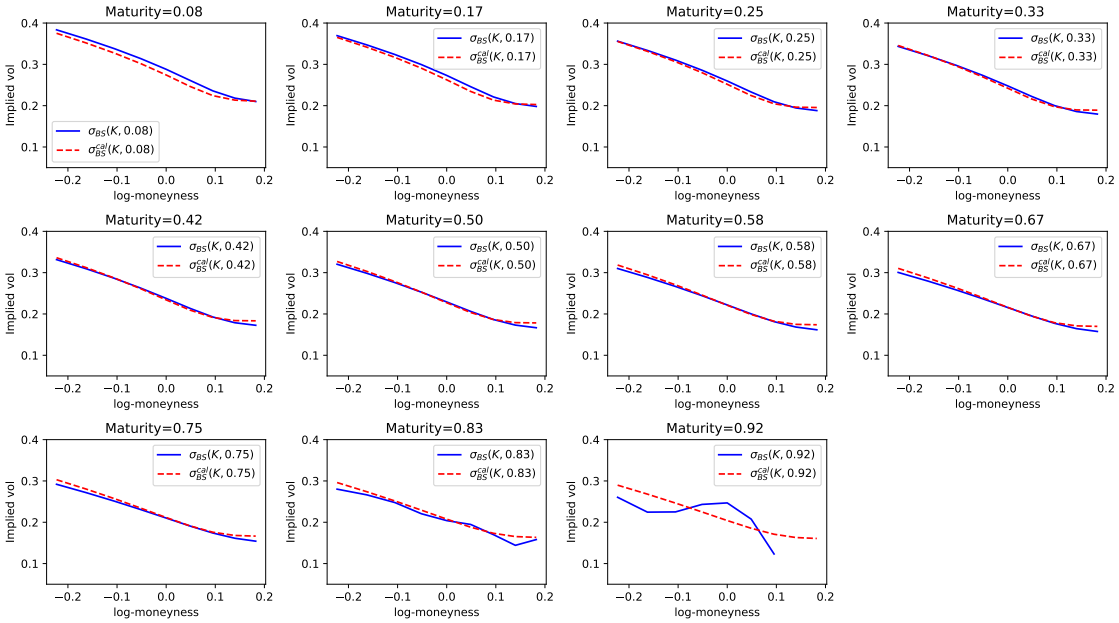


Error heat map for the rough Bergomi model.

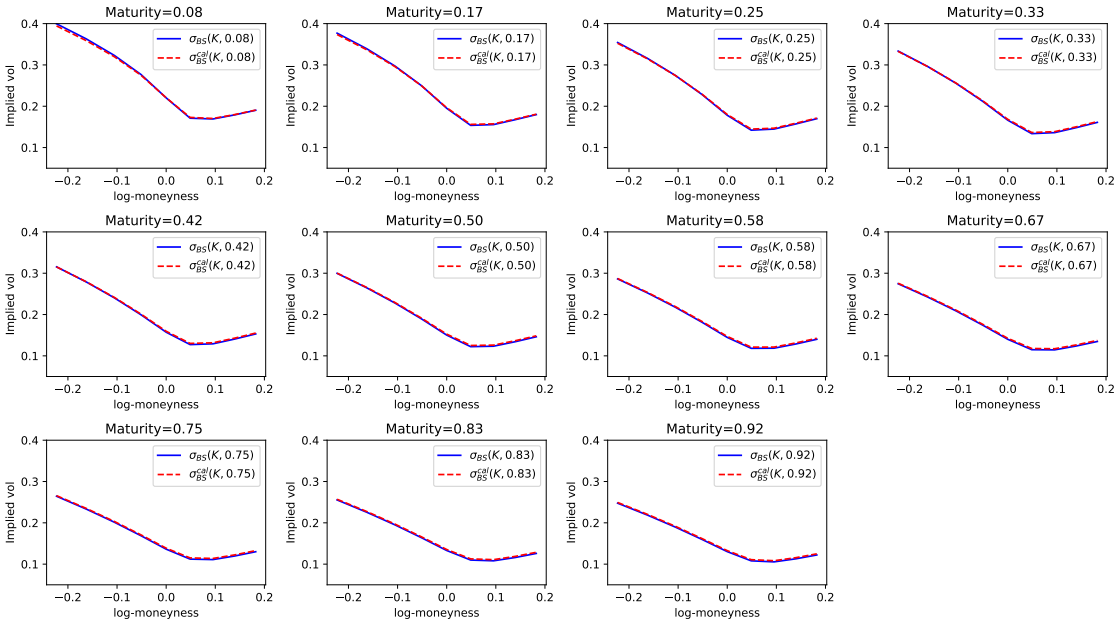
Figure 19: Heat maps of absolute error, per tenor and strike, between linear interpolation of a network generated surface with a Monte-Carlo generated surface defined on new strikes and maturities. The strikes and maturities used were obtained by adding 0.025 to each strike in the previous grid and 15/360 to each maturity in the previous grid. This shows that the neural network combined with a linear interpolation does not accurately capture the volatility curve for all models. Depending on model parameters, a linear interpolation may cause a large error, especially given that we have shifted both tenors and strikes. This illustrates other possibilities of the model, though we would need to study in detail at which distance the interpolation breaks down.

model	spread threshold at 5%	spread threshold at 1%
Heston	0.58	0.59
Bergomi	0.27	0.28
rough Bergomi	0.34	0.35

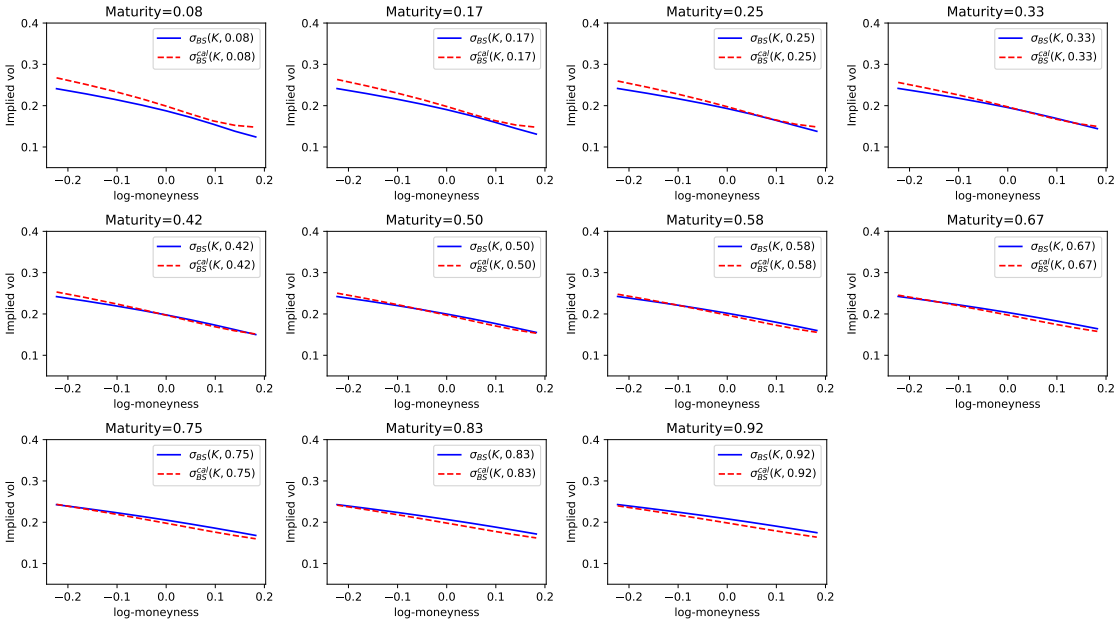
Figure 20: Spread threshold such that the resulting likelihood ratio test has a 5% and 1% p-value. The test was computed using at-the-money implied volatilities, for which the spread, depending on the liquidity of the underlying and tenor of the contract, may vary from 0.1 to 0.5. Therefore for applications to options pricing we would still need to reduce the average error. We would also need to tackle extreme errors. Those pose an additional challenge that we set aside.



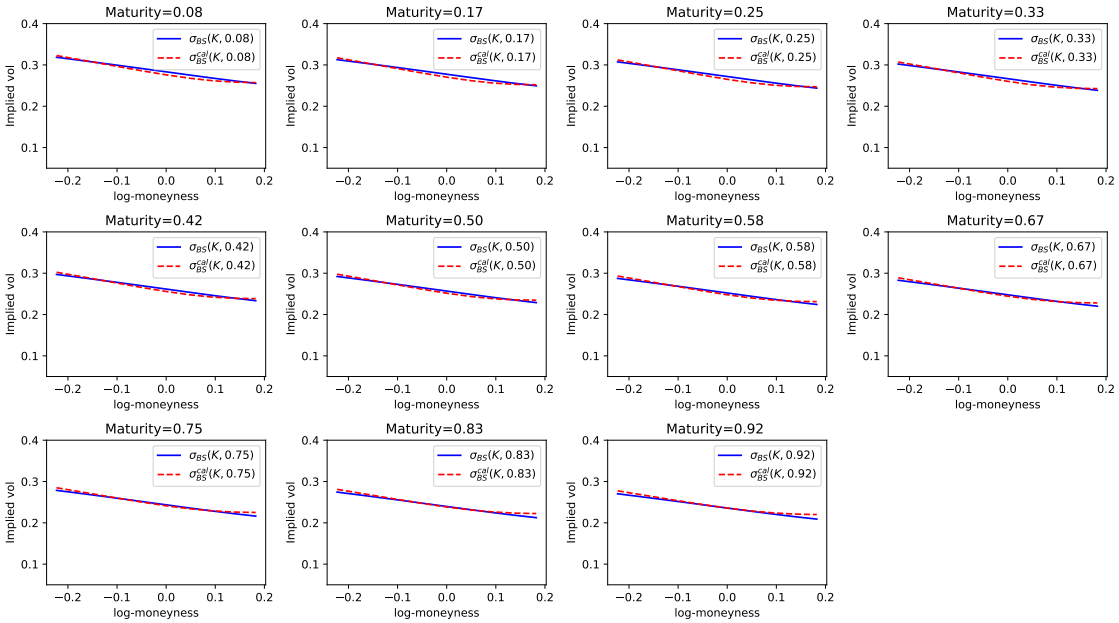
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\kappa = 1.97, \rho = -0.77, \sigma = 0.81, \theta = 0.032, V_0 = 0.0924$.



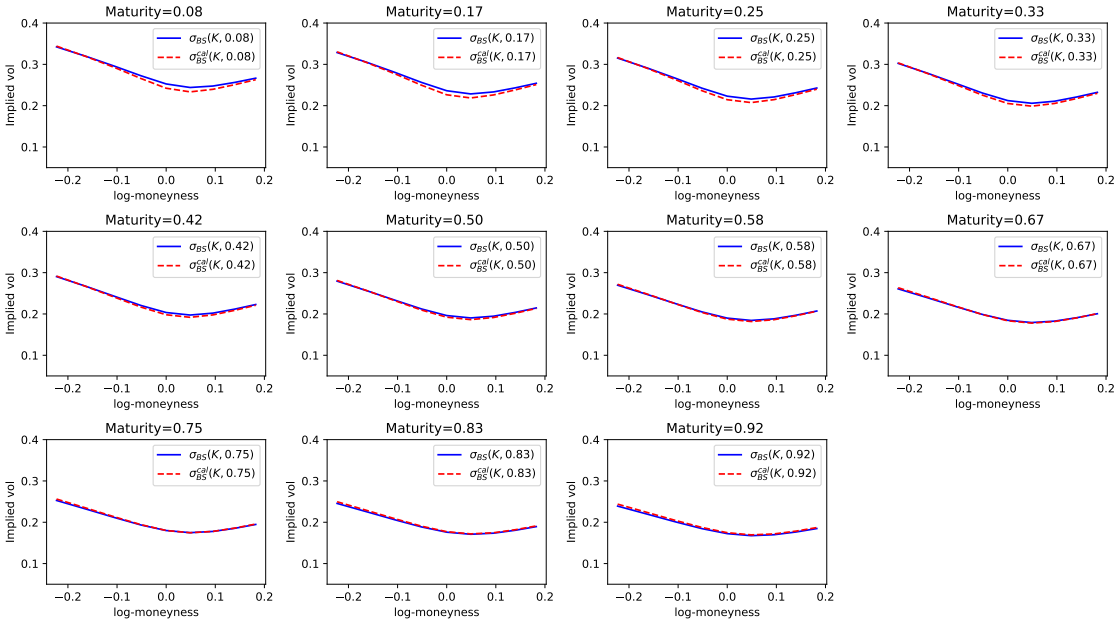
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\kappa = 0.99, \rho = -0.75, \sigma = 1.48, \theta = 0.033, V_0 = 0.0641$.



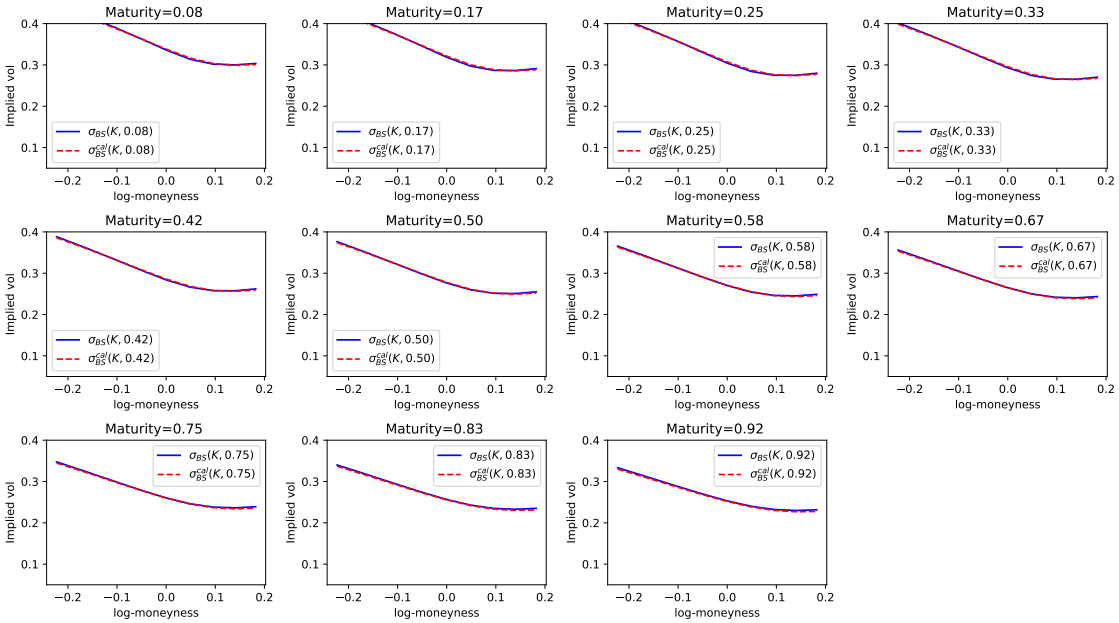
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\kappa = 1.50, \rho = -0.88, \sigma = 0.27, \theta = 0.062, V_0 = 0.0339$.



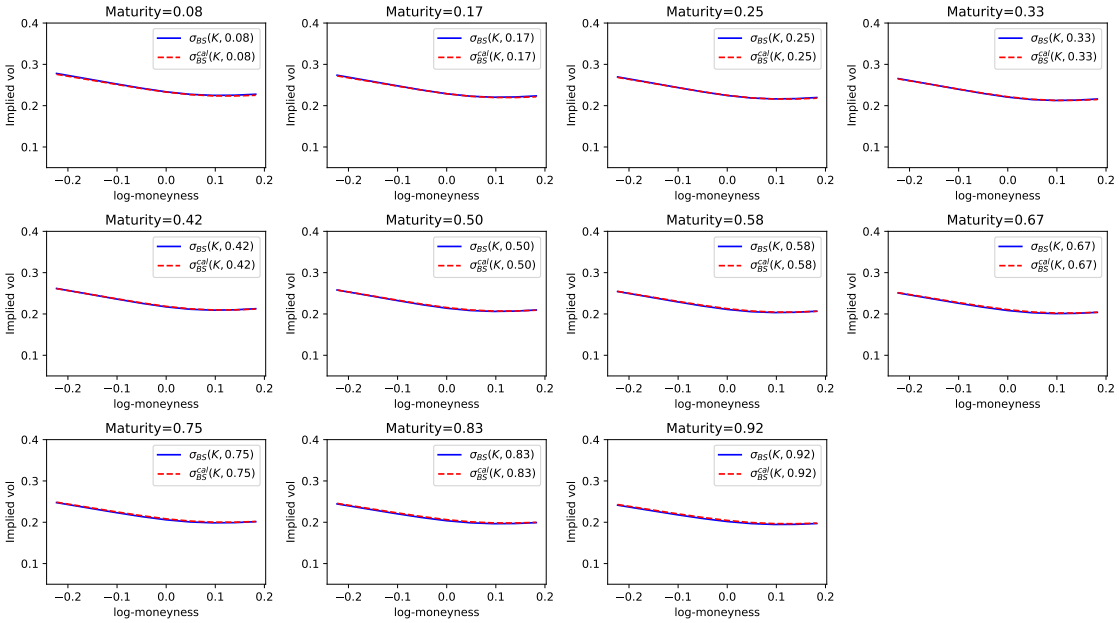
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\kappa = 1.02, \rho = -0.62, \sigma = 0.30, \theta = 0.020, V_0 = 0.0836$.



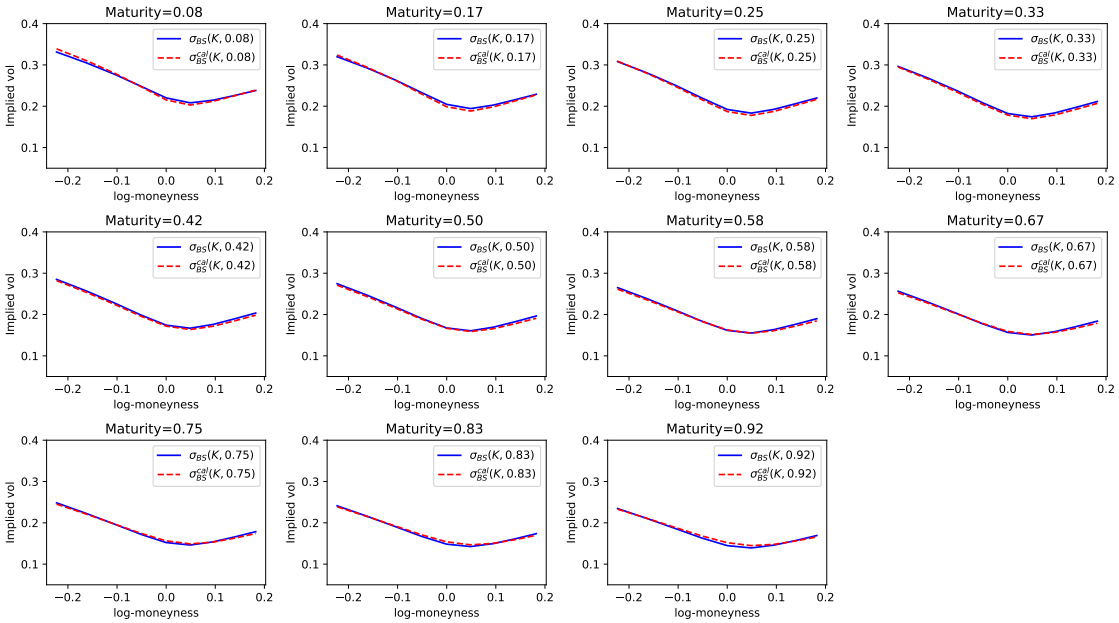
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\kappa = 1.62, \rho = -0.30, \sigma = 1.02, \theta = 0.022, V_0 = 0.0740$.



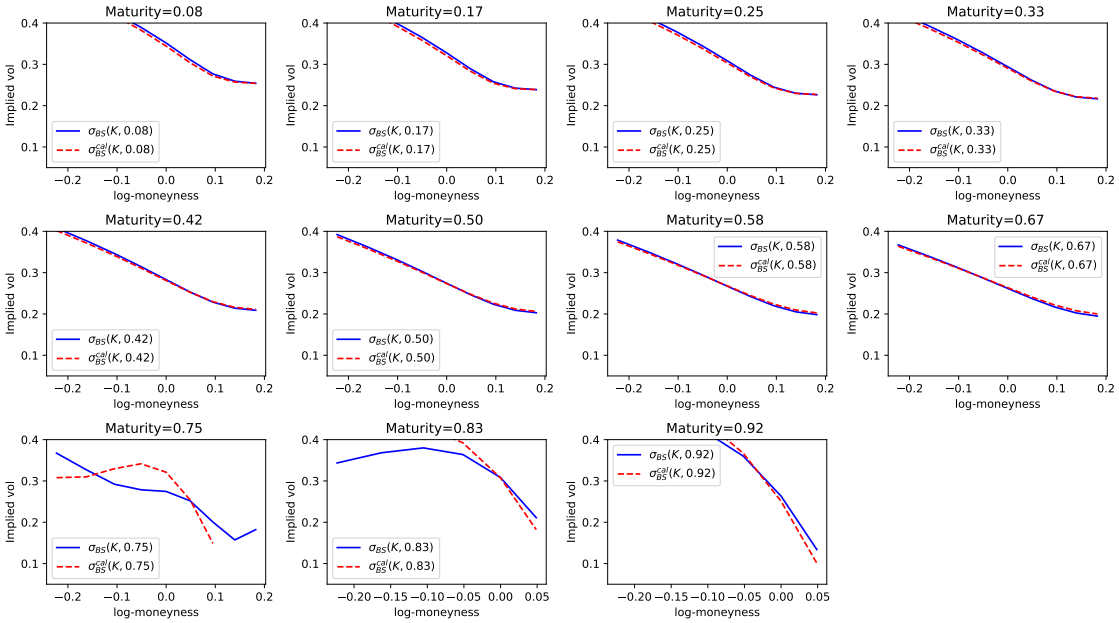
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\kappa = 0.93, \rho = -0.52, \sigma = 1.31, \theta = 0.094, V_0 = 0.1265$.



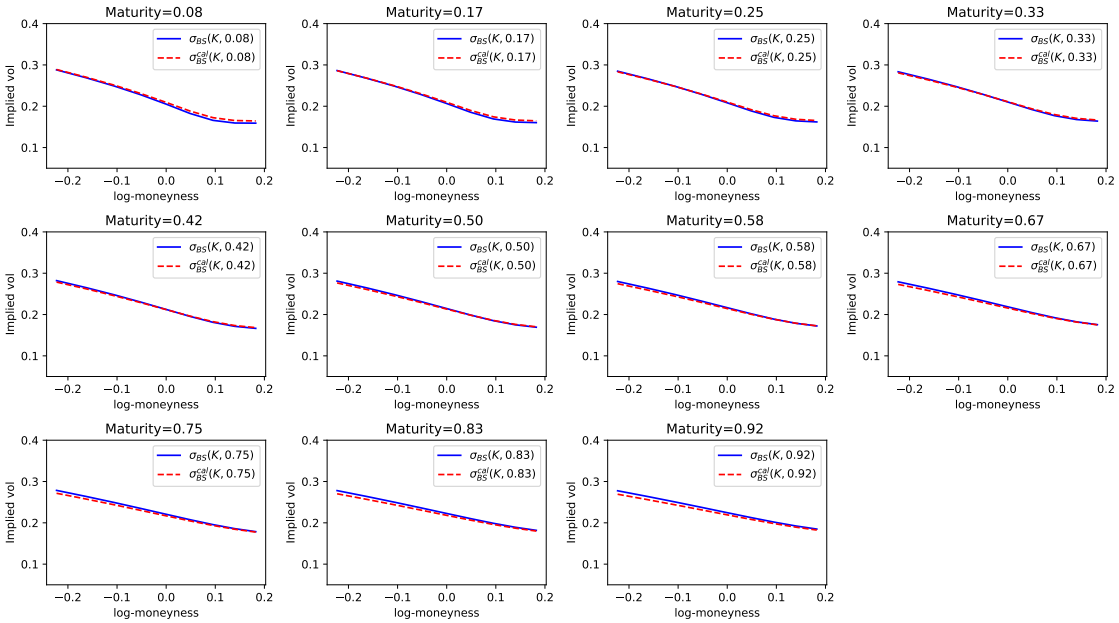
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\kappa = 0.91, \rho = -0.31, \sigma = 0.46, \theta = 0.035, V_0 = 0.0568$.



Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\kappa = 0.74, \rho = -0.38, \sigma = 1.05, \theta = 0.017, V_0 = 0.0570$.

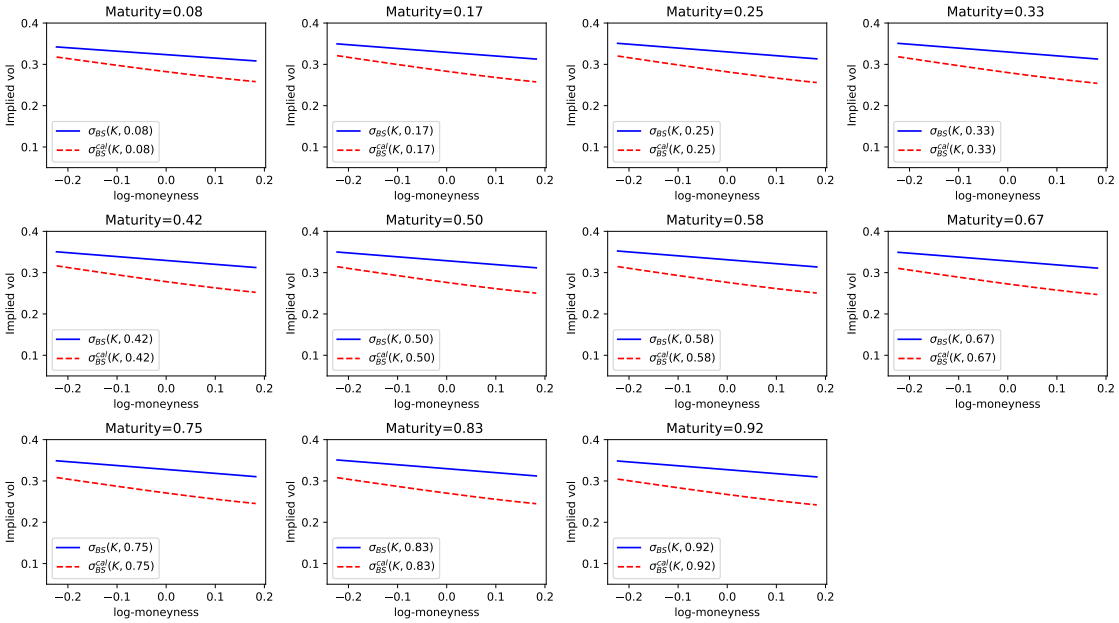


Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\kappa = 1.75, \rho = -0.76, \sigma = 1.47, \theta = 0.088, V_0 = 0.1428$.

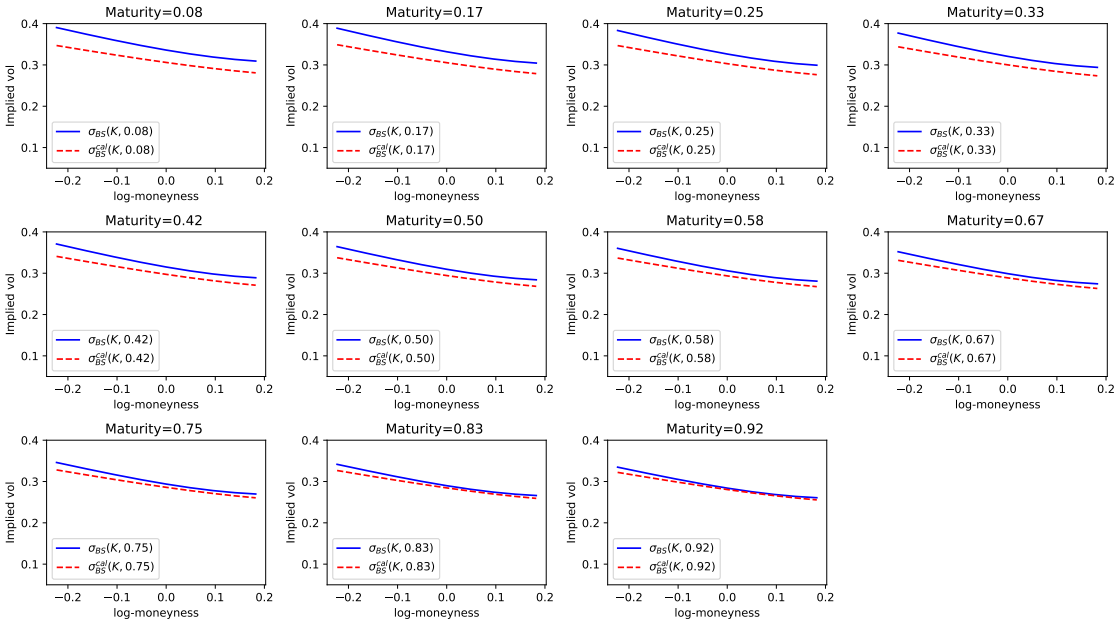


Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\kappa = 1.25, \rho = -0.71, \sigma = 0.55, \theta = 0.092, V_0 = 0.0415$.

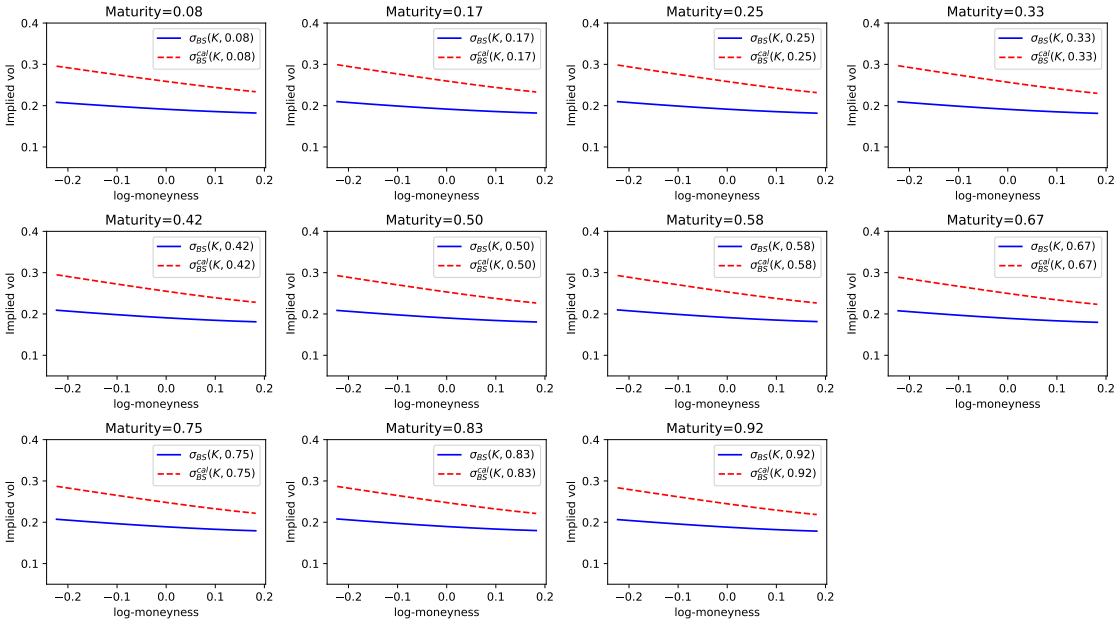
Figure 21: Examples of implied volatility surfaces generated by Monte Carlo using calibrated parameters compared to implied volatility surfaces passed as input, for the Heston model.



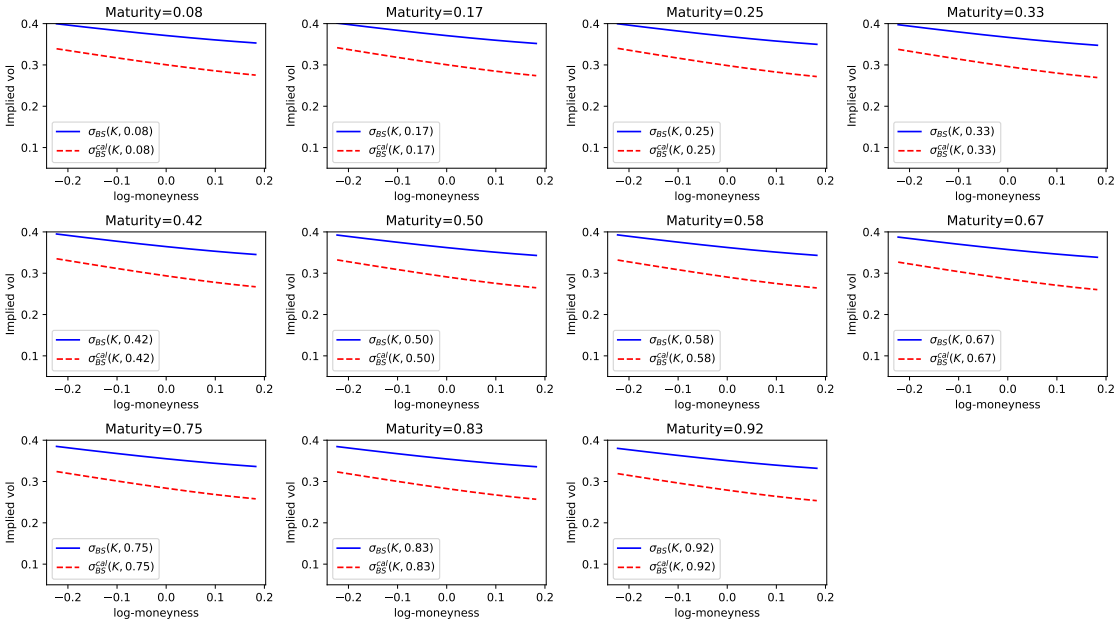
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\rho = 0.84, \nu = 0.46, \sigma_0 = 0.0114$.



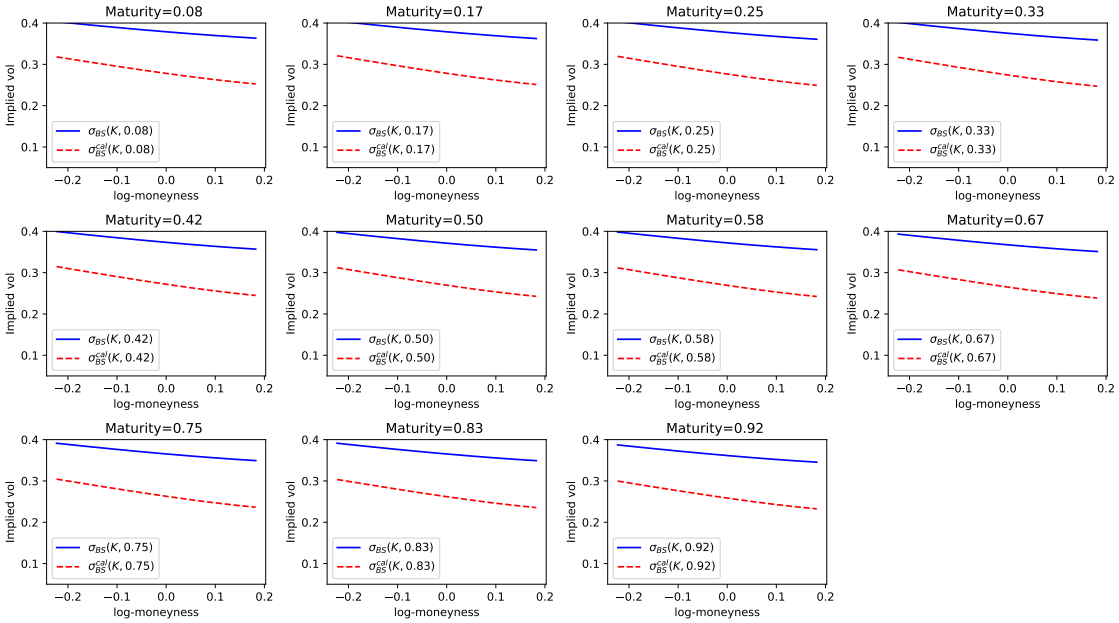
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\rho = 0.48, \nu = 1.95, \sigma_0 = 0.119$.



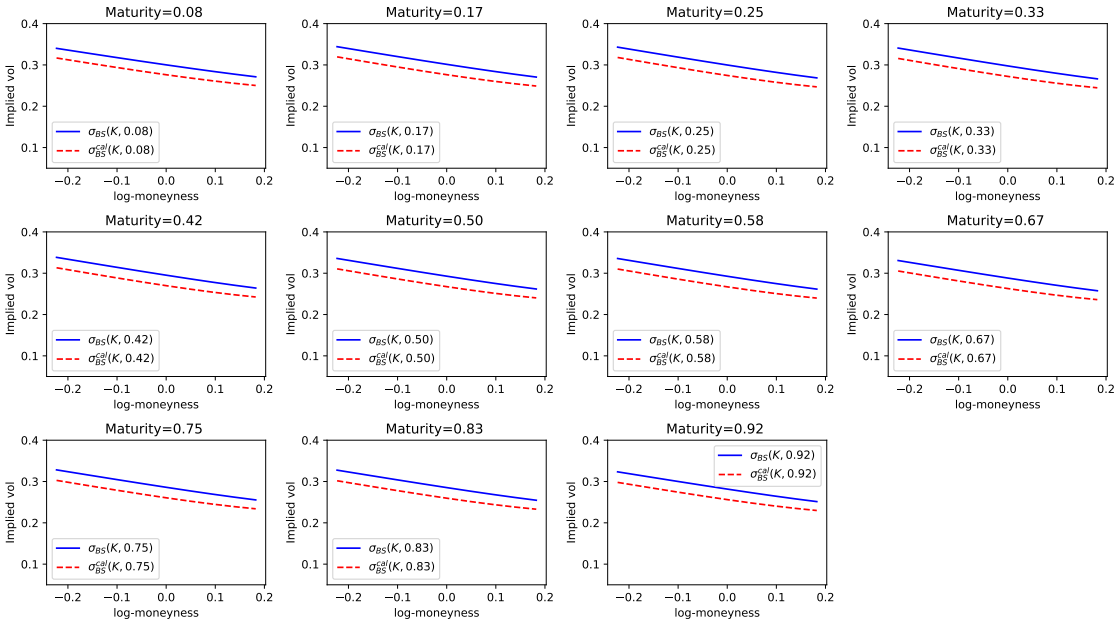
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\rho = -0.41, \nu = 0.69, \sigma_0 = 0.037$.



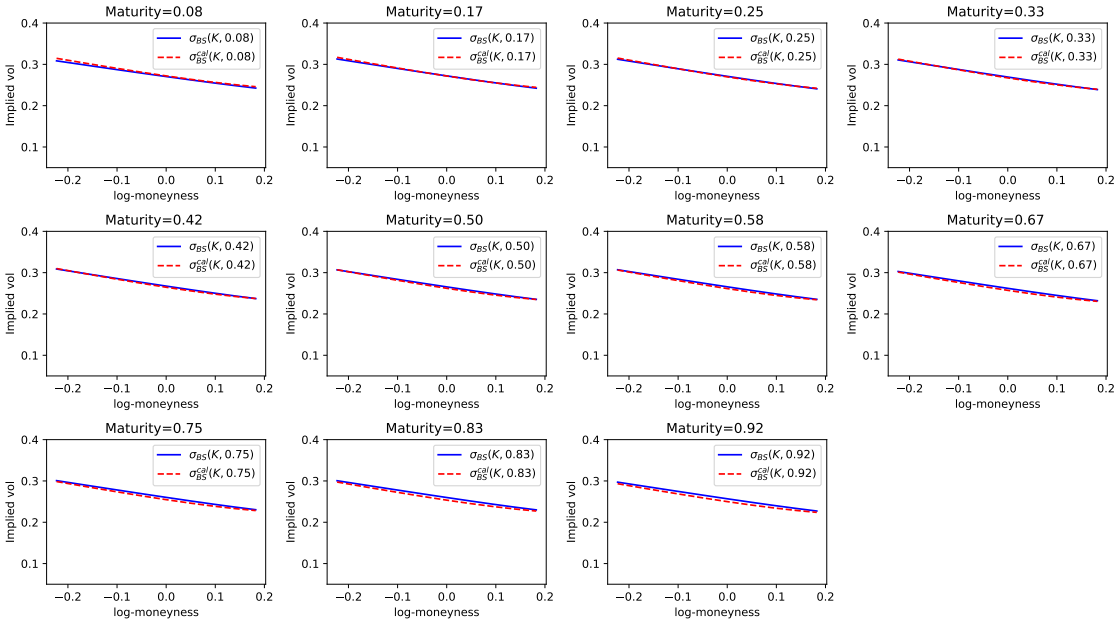
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\rho = -0.48, \nu = 1.09, \sigma_0 = 0.142$.



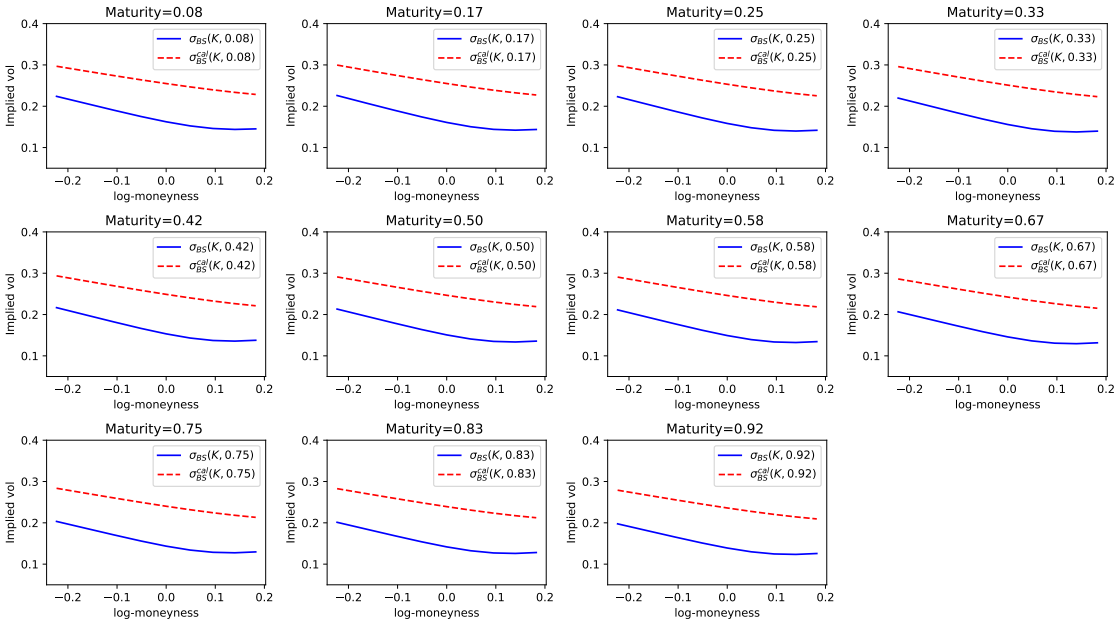
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\rho = -0.45, \nu = 0.99, \sigma_0 = 0.148$.



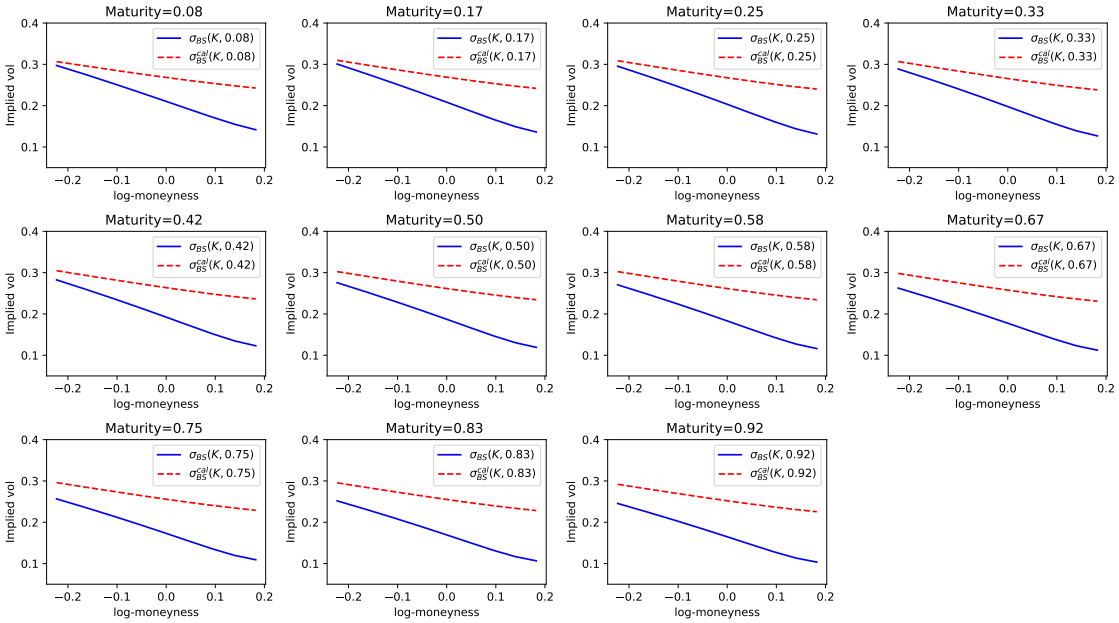
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\rho = -0.68, \nu = 1.17, \sigma_0 = 0.096$.



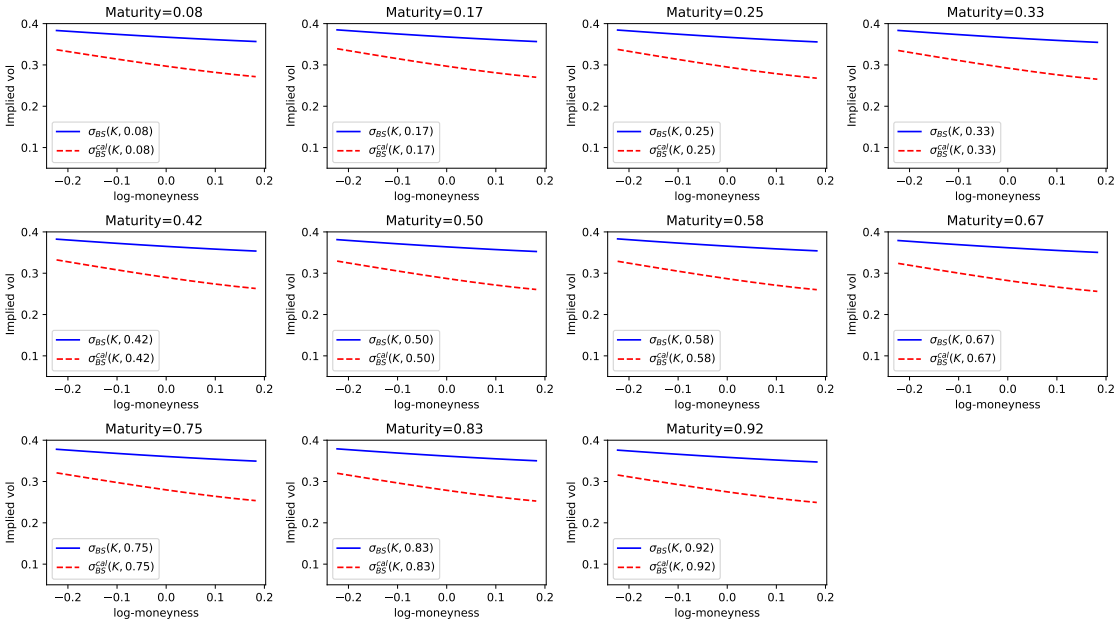
Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\rho = -0.70, \nu = 1.09, \sigma_0 = 0.078$.



Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\rho = -0.55, \nu = 1.97, \sigma_0 = 0.028$.

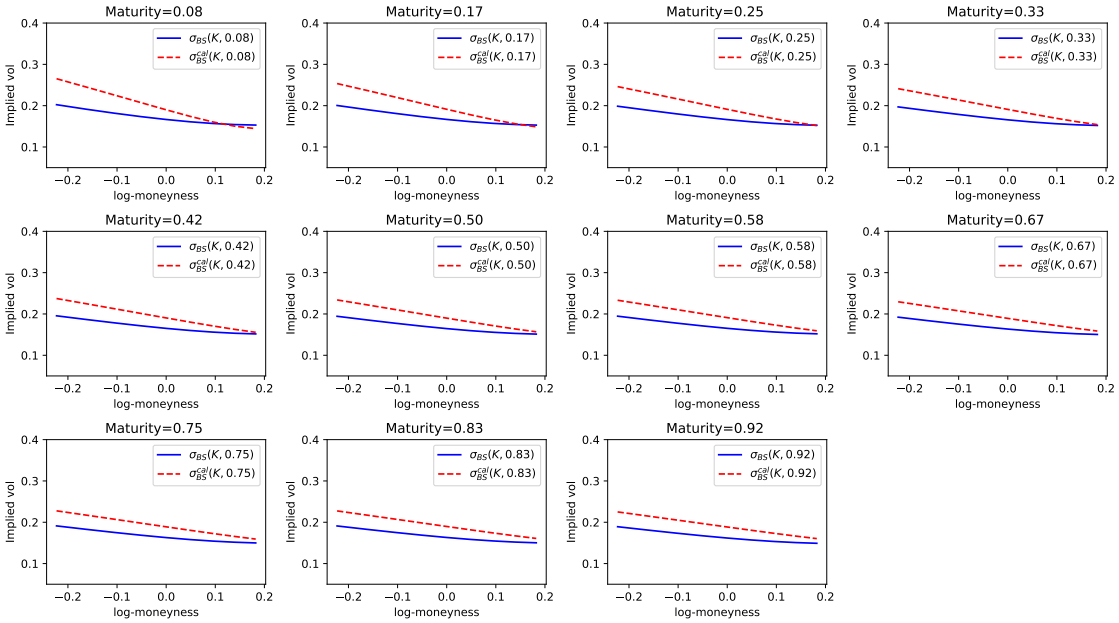


Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\rho = -0.88, \nu = 2.26, \sigma_0 = 0.050$.

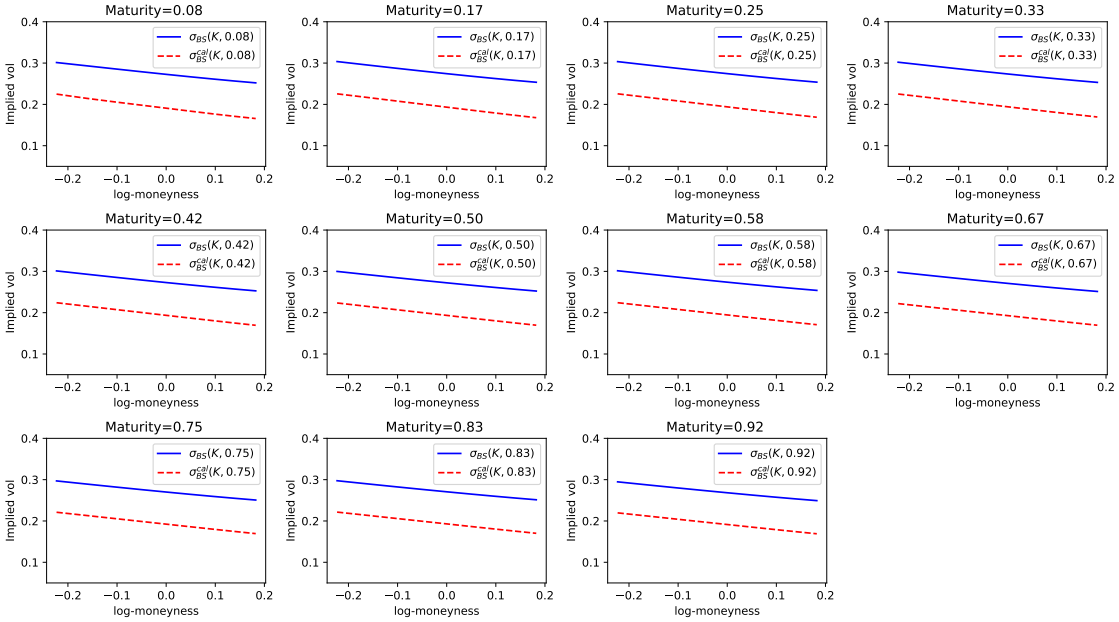


Comparison of generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces $\rho = -0.41, \nu = 0.72, \sigma_0 = 0.138$.

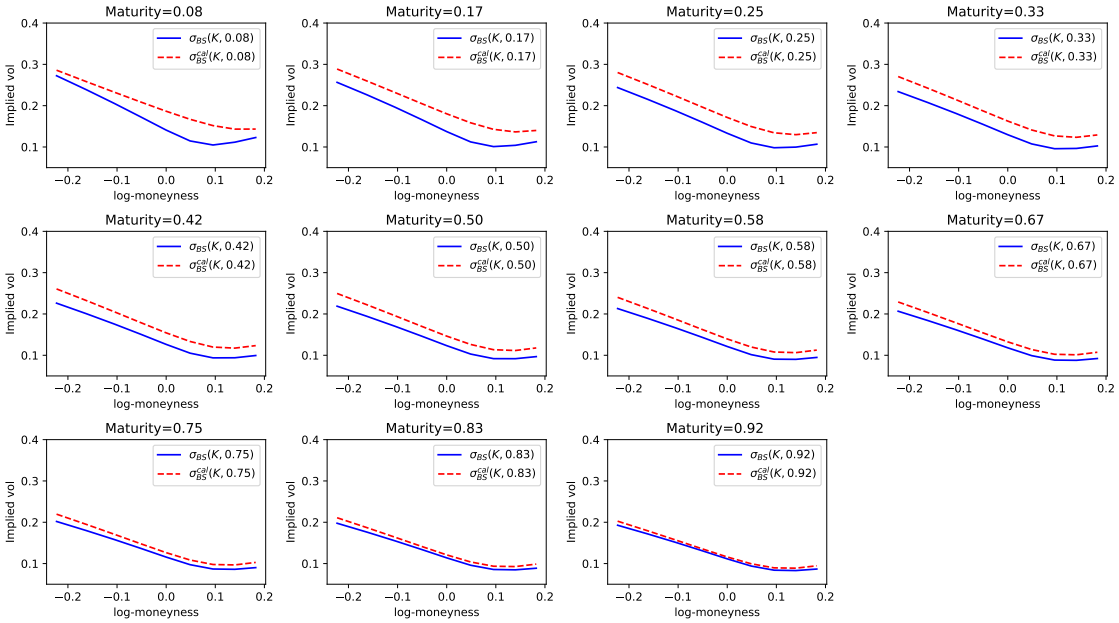
Figure 22: Examples of implied volatility surfaces generated by Monte Carlo using calibrated parameters compared to implied volatility surfaces passed as input, for the Bergomi model.


Maturity=0.08
Maturity=0.17
Maturity=0.25
Maturity=0.33
Maturity=0.42
Maturity=0.50
Maturity=0.58
Maturity=0.67
Maturity=0.75
Maturity=0.83
Maturity=0.92
Implied vol
log-moneyness

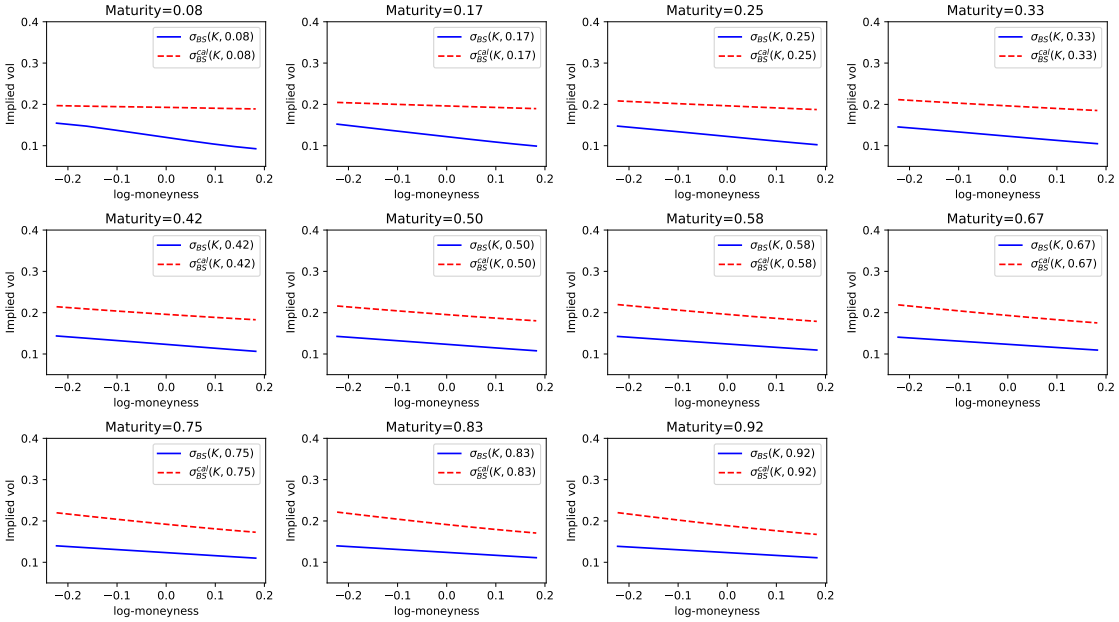
(a)


Maturity=0.08
Maturity=0.17
Maturity=0.25
Maturity=0.33
Maturity=0.42
Maturity=0.50
Maturity=0.58
Maturity=0.67
Maturity=0.75
Maturity=0.83
Maturity=0.92
Implied vol
log-moneyness

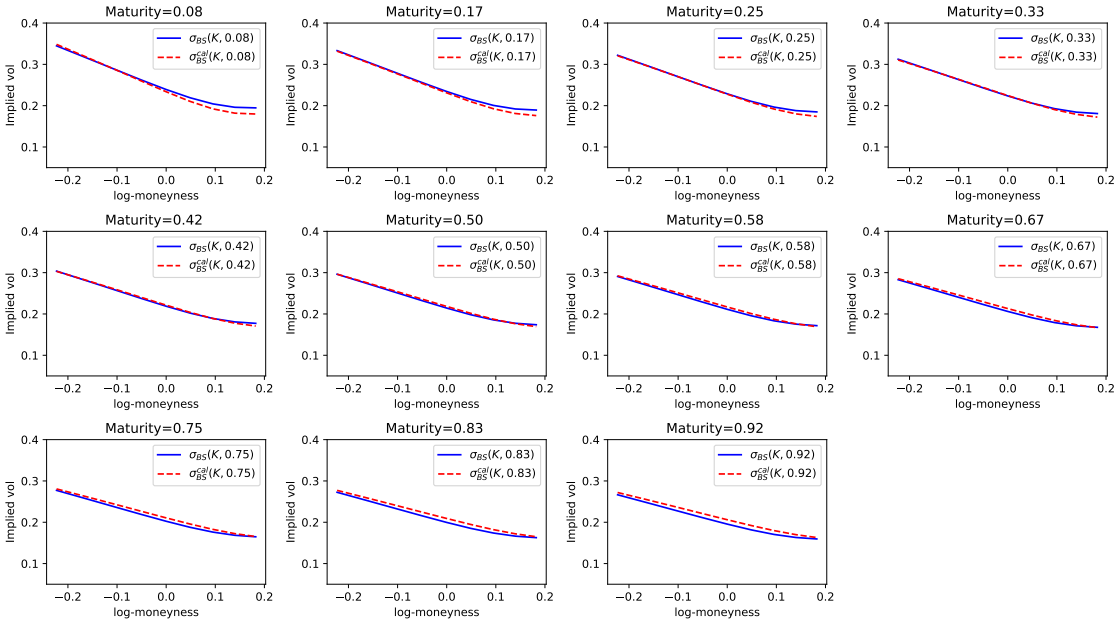
(b)


Maturity=0.08
Maturity=0.17
Maturity=0.25
Maturity=0.33
Maturity=0.42
Maturity=0.50
Maturity=0.58
Maturity=0.67
Maturity=0.75
Maturity=0.83
Maturity=0.92
Implied vol
log-moneyness

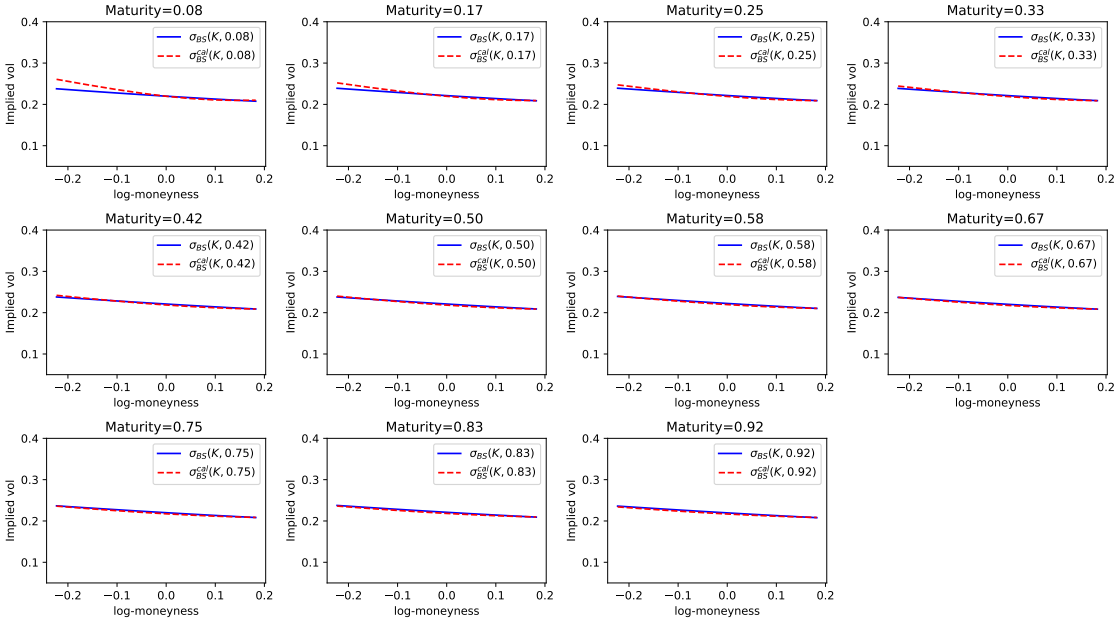
(c)


Maturity=0.08
Maturity=0.17
Maturity=0.25
Maturity=0.33
Maturity=0.42
Maturity=0.50
Maturity=0.58
Maturity=0.67
Maturity=0.75
Maturity=0.83
Maturity=0.92
Implied vol
log-moneyness

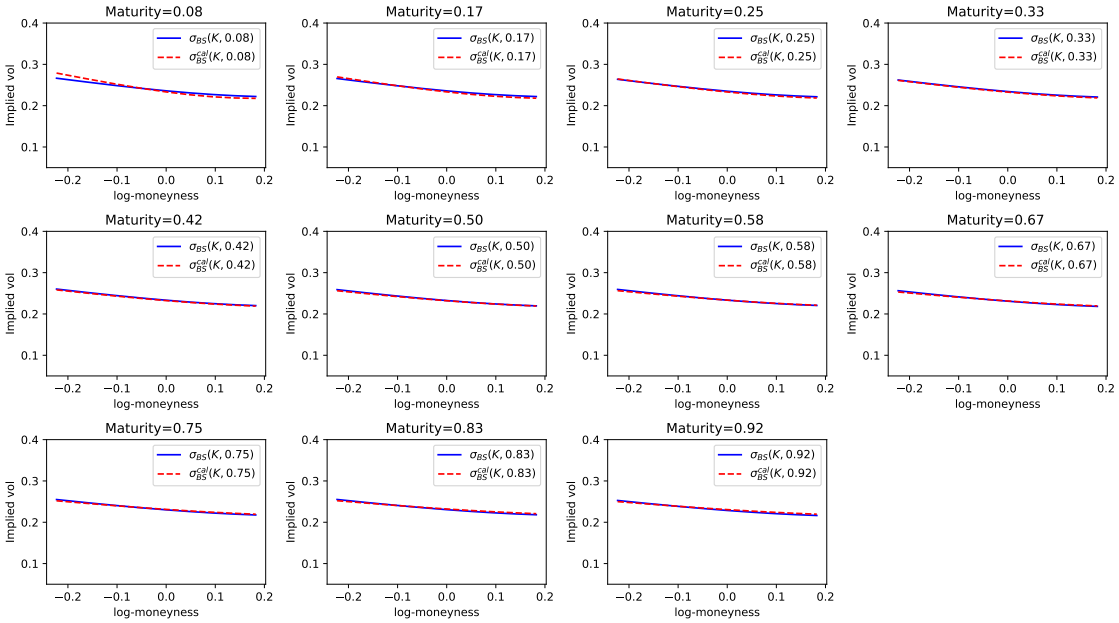
(d)


Maturity=0.08
Maturity=0.17
Maturity=0.25
Maturity=0.33
Maturity=0.42
Maturity=0.50
Maturity=0.58
Maturity=0.67
Maturity=0.75
Maturity=0.83
Maturity=0.92
Implied vol
log-moneyness

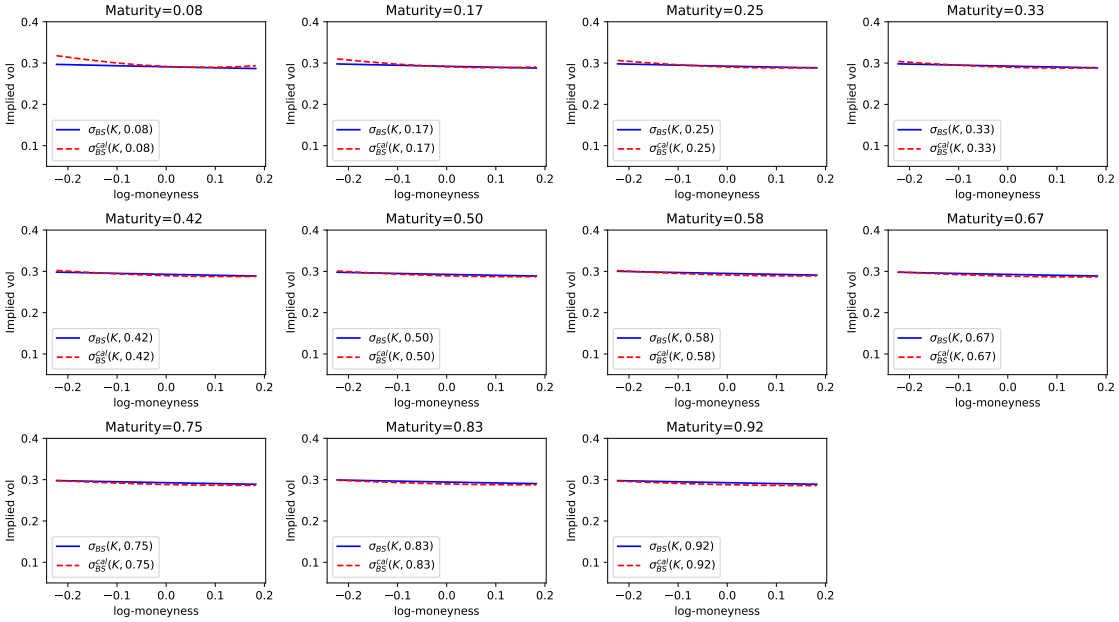
(e)


Maturity=0.08
Maturity=0.17
Maturity=0.25
Maturity=0.33
Maturity=0.42
Maturity=0.50
Maturity=0.58
Maturity=0.67
Maturity=0.75
Maturity=0.83
Maturity=0.92
Implied vol
log-moneyness

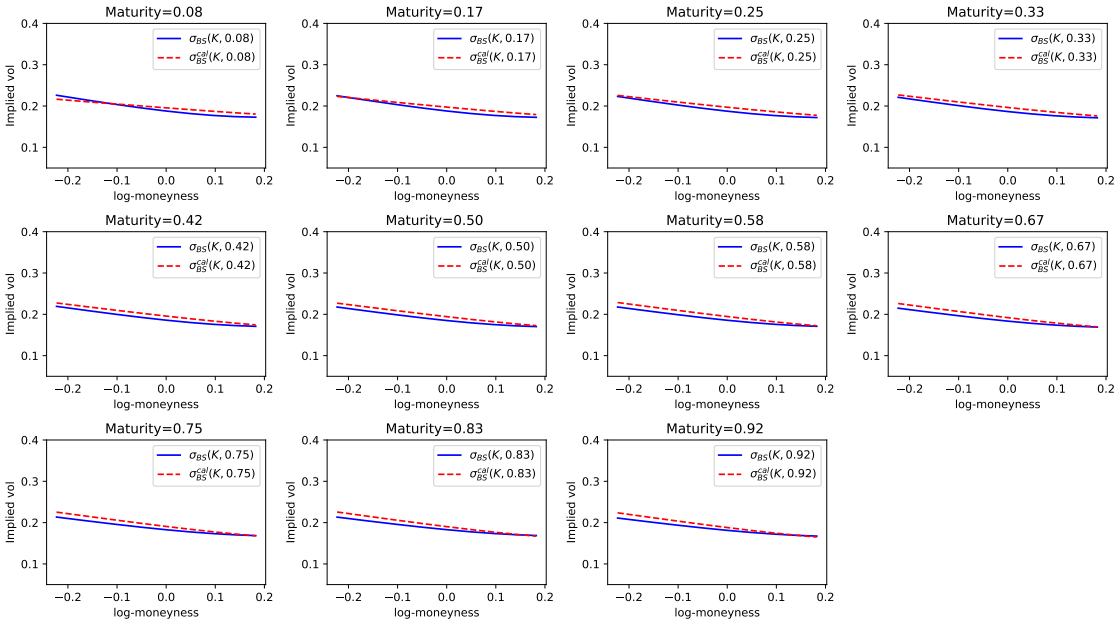
(f)


Maturity=0.08
Maturity=0.17
Maturity=0.25
Maturity=0.33
Maturity=0.42
Maturity=0.50
Maturity=0.58
Maturity=0.67
Maturity=0.75
Maturity=0.83
Maturity=0.92
Implied vol
log-moneyness
$\sigma_{BS}(K, 0.08)$
$\sigma^{cal}_{BS}(K, 0.08)$
$\sigma_{BS}(K, 0.17)$
$\sigma^{cal}_{BS}(K, 0.17)$
$\sigma_{BS}(K, 0.25)$
$\sigma^{cal}_{BS}(K, 0.25)$
$\sigma_{BS}(K, 0.33)$
$\sigma^{cal}_{BS}(K, 0.33)$
$\sigma_{BS}(K, 0.42)$
$\sigma^{cal}_{BS}(K, 0.42)$
$\sigma_{BS}(K, 0.50)$
$\sigma^{cal}_{BS}(K, 0.50)$
$\sigma_{BS}(K, 0.58)$
$\sigma^{cal}_{BS}(K, 0.58)$
$\sigma_{BS}(K, 0.67)$
$\sigma^{cal}_{BS}(K, 0.67)$
$\sigma_{BS}(K, 0.75)$
$\sigma^{cal}_{BS}(K, 0.75)$
$\sigma_{BS}(K, 0.83)$
$\sigma^{cal}_{BS}(K, 0.83)$
$\sigma_{BS}(K, 0.92)$
$\sigma^{cal}_{BS}(K, 0.92)$

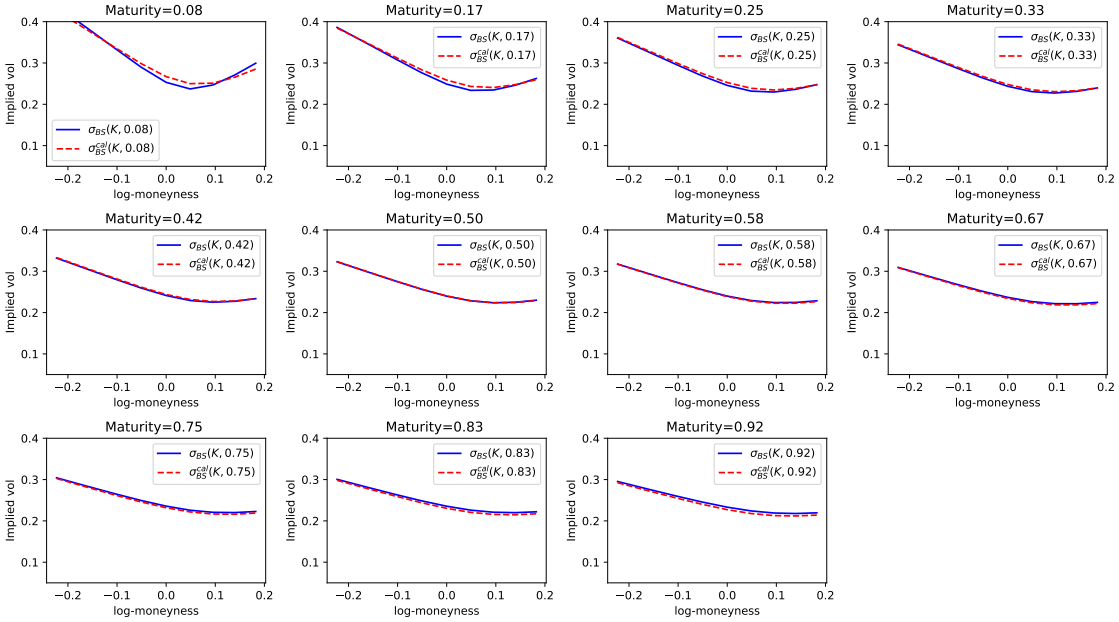
(g)


Maturity=0.08
Maturity=0.17
Maturity=0.25
Maturity=0.33
Maturity=0.42
Maturity=0.50
Maturity=0.58
Maturity=0.67
Maturity=0.75
Maturity=0.83
Maturity=0.92
Implied vol
log-moneyness
$\sigma_{BS}(K, 0.08)$
$\sigma^{cal}_{BS}(K, 0.08)$
$\sigma_{BS}(K, 0.17)$
$\sigma^{cal}_{BS}(K, 0.17)$
$\sigma_{BS}(K, 0.25)$
$\sigma^{cal}_{BS}(K, 0.25)$
$\sigma_{BS}(K, 0.33)$
$\sigma^{cal}_{BS}(K, 0.33)$
$\sigma_{BS}(K, 0.42)$
$\sigma^{cal}_{BS}(K, 0.42)$
$\sigma_{BS}(K, 0.50)$
$\sigma^{cal}_{BS}(K, 0.50)$
$\sigma_{BS}(K, 0.58)$
$\sigma^{cal}_{BS}(K, 0.58)$
$\sigma_{BS}(K, 0.67)$
$\sigma^{cal}_{BS}(K, 0.67)$
$\sigma_{BS}(K, 0.75)$
$\sigma^{cal}_{BS}(K, 0.75)$
$\sigma_{BS}(K, 0.83)$
$\sigma^{cal}_{BS}(K, 0.83)$
$\sigma_{BS}(K, 0.92)$
$\sigma^{cal}_{BS}(K, 0.92)$

(h)



Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.88, \nu = 2.26, \sigma_0 = 0.050$.

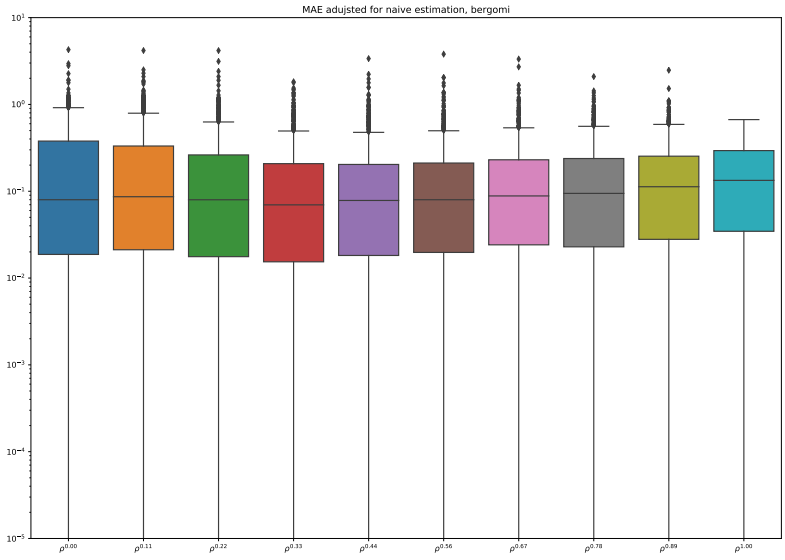


Comparison of volatility surface from the Bergomi model computed using a neural network and the Monte-Carlo algorithm with parameters $\rho = -0.41, \nu = 0.72, \sigma_0 = 0.138$.

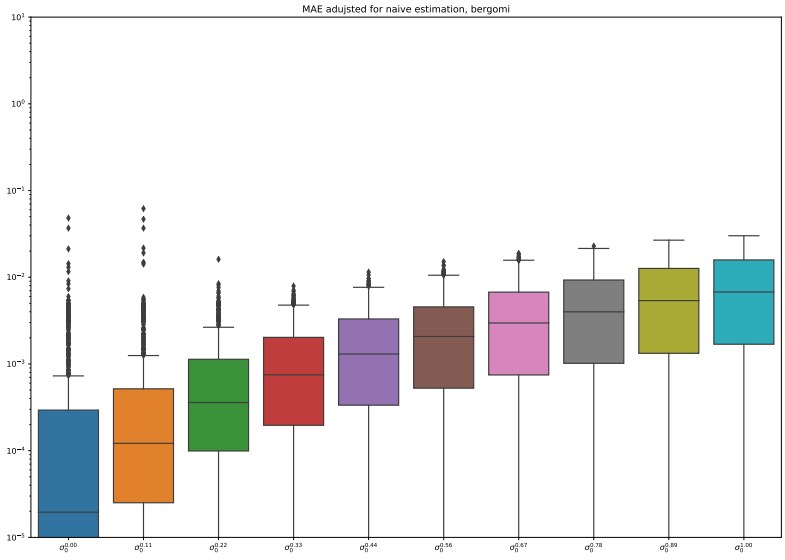
Figure 23: Examples of implied volatility surfaces for the rough Bergomi model, generated by Monte Carlo using calibrated parameters compared to input implied volatility surfaces. Though the majority is a good fit there are examples where the calibrated surface is far away from the surface generated using calibrated parameters. As a breakdown of the calibration performance per model parameter in Figure 16 shows, our estimation of ν is poor. However, for small values of ν the implied volatility surface becomes flatter. A small estimation error on ν would then yield a different surface.

model	computation time
Heston	0.81
Bergomi	0.06
rough Bergomi	0.13

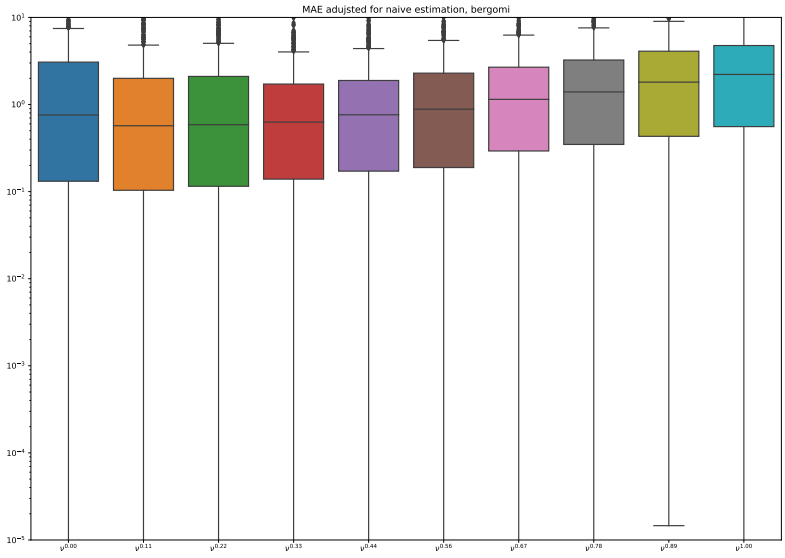
Figure 24: Computation time of calibration to option prices divided by computation time of a single surface. Therefore, the calibration is faster than any calibration method requiring a larger number of function evaluations than provided in the table. Strikingly, the speed-up is so significant that calibration is faster than an evaluation using the numerical methods from 2. As with the computation of option prices, we find that larger speed ups are obtained for the Bergomi and rough Bergomi model.



Boxplot of the average calibration error for parameter ρ, as a function of the proportion of removed points from the implied volatility surface.

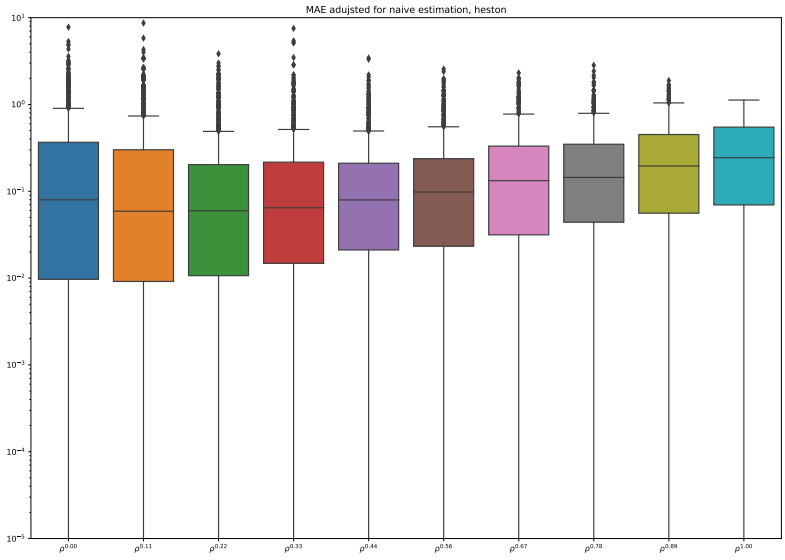


Boxplot of the average calibration error for parameter σ, as a function of the proportion of removed points from the implied volatility surface.

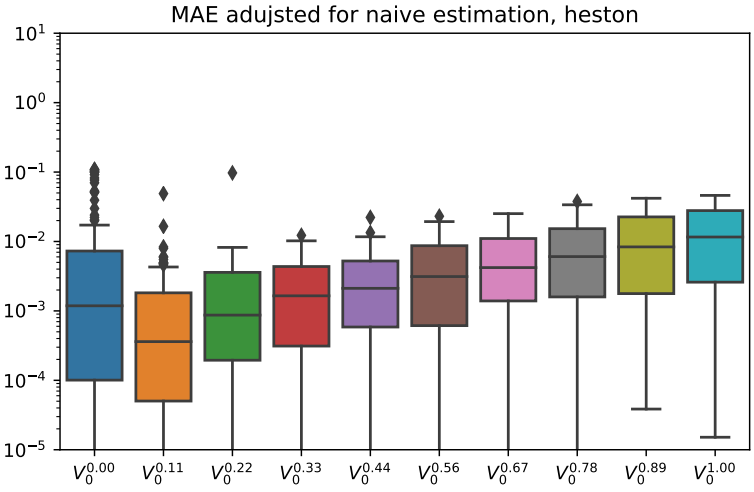


Boxplot of the average calibration error for parameter ν, as a function of the proportion of removed points from the implied volatility surface.

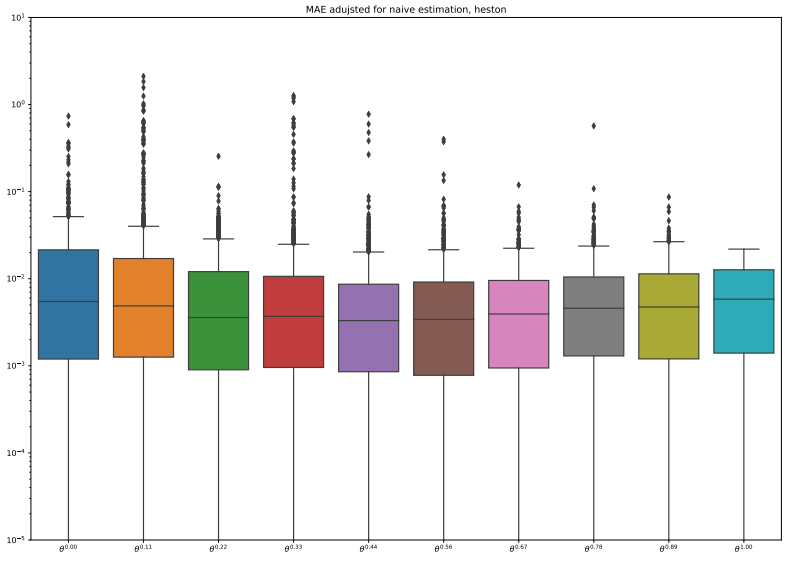
Figure 25: Performance of the calibration for the Bergomi model according to the percentage of overall points available in the implied volatility surface. Each point of the surface has an equal probability of being removed. We represent the mean absolute error (MAE) divided by the error using the mid point of the sampled distribution. Errors are averaged across the surface and we used 1,000 samples. The line of each box plot is the median, while the edges are at the first and third quartiles. Whiskers extend further by 1.5 times the interquartile range. Samples outside of this range are represented as dots. We can observe that performance deteriorates as we remove more points from the surface. Furthermore, the estimation of σ_0 is more robust to missing points than the other parameters. We remark that for both ρ and ν, the average error increases when only 20% of the implied volatility surface is remaining. This means that we would be able to calibrate the Bergomi model on surfaces with more than 20% of points available. It is at around the same percentage that the average error of σ_0 stops increasing, indicating that the methodology breaks down for all parameters at around the same number of values removed from the surface. This might be due to the way we replace values on the surface. Since the network still requires a whole surface to calibrate on, we replace missing values by the median for that tenor and strike. When the surface is made up mostly of those points, then to minimise the error the network will fit those points instead of the ones that were not removed.



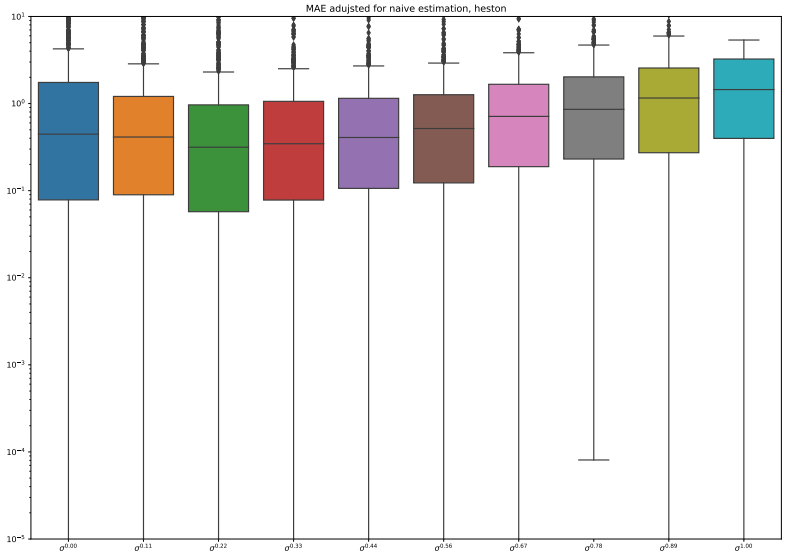
Boxplot of the average calibration error for parameter ρ, as a function of the proportion of removed points from the implied volatility surface.



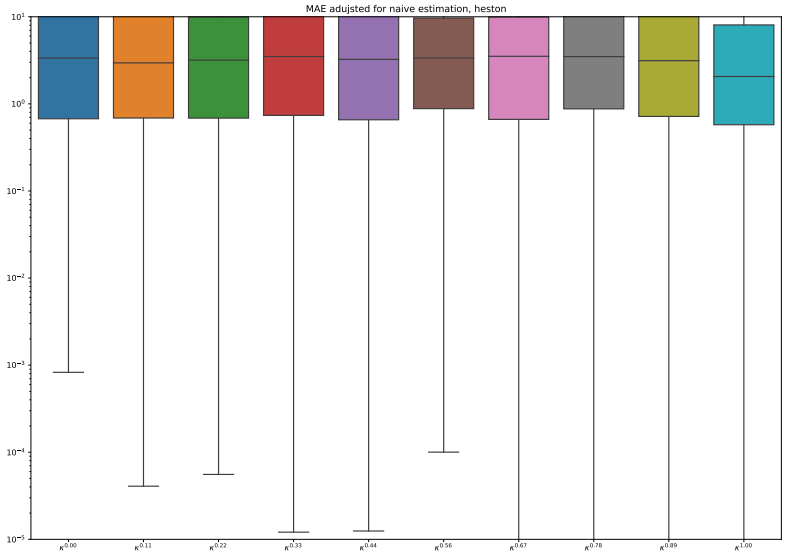
Boxplot of the average calibration error for parameter V_0, as a function of the proportion of removed points from the implied volatility surface.



Boxplot of the average calibration error for parameter θ, as a function of the proportion of removed points from the implied volatility surface.



Boxplot of the average calibration error for parameter σ, as a function of the proportion of removed points from the implied volatility surface.



Boxplot of the average calibration error for parameter κ, as a function of the proportion of removed points from the implied volatility surface.

Figure 26: Performance of the calibration for the Heston model according to the percentage of overall points available in the implied volatility surface. We use the same plotting conventions as in Figure 25. Interestingly, as with the Bergomi model, the calibration is robust until we remove more than 80% of the surfaces' points. This reinforces our belief that this is not model-dependent but instead due to the calibration procedure. In contrast with the Bergomi model however, the average error for parameter κ stays at around the same level, no matter how many points we remove. This is expected since as Figure 16 shows, we fail to estimate κ even with the whole surface.

Conclusion

Neural networks can price and calibrate stochastic models quickly and accurately. Previously impossible applications that required millions of simulations are now realistic. We look at potential applications before discussing possible improvements.

For backtesting purposes, we can now calibrate and test stochastic models across large databases. For example, assuming it takes a second to compute option prices, testing a model that recalibrates daily over a thousand assets and twenty years of data would have taken more than 2 months. Now, it would take less than a day.

Additionally, those fast computations can help monitor risk. We can break down and track in real time the exposure of a portfolio of vanilla options with model parameters. Of course, standard methods can still be run in the background. In fact, we can combine them and use estimated model parameters as a starting point for their calibration. Such speed may be crucial to estimating risk in scenarios of market stress, where monitoring multiple models at once may be necessary.

Furthermore, we can quickly adjust our implied volatility surface after a move in the market. This is especially useful around economic events, where we would usually observe a jump in implied volatility, after which we would have to remark the surface. Now we may calibrate in real time and adjust our surface to stay consistent with market prices. For example, if the 3 month at-the-money moves, we will adjust all prices including the 1 month and 6 month. If the model is correct we could track parameter values, especially since our methodology is robust to missing points in the implied volatility surface.

Some of the above applications would require lowering the error since it is now larger than spreads used on liquid, at-the-money contracts.

One possibility is to change the sampling procedure. De Spiegeleer et al ([4]) have found that training a model on smaller volatilities leads to better performance. This has led them to use a bounded exponential distribution to sample the volatility of volatility and long term variance (De Spiegeleer et al, [4], p.9). This is especially interesting given we have encountered the same issues, as shown in figure 16.

Another possibility would be to increase the number of nodes and layers. Complexity would however come at the cost of speed, as operations become more expensive. Another approach, especially relevant for models with many parameters, is to increase the number of training samples. The results from De Spiegeleer et al ([4] p.10) confirm this intuition as they show how the error decreases by almost half from a training set of size 5,000 to 20,000.

Though this model is fast and precise, it lacks the ability to price complex options, though it can price linear combination of vanilla options. Therefore, if we have a portfolio of vanilla options for which the pay-off is close to that of an exotic option, we could use this model to price the portfolio,

and therefore the exotic by proxy. We leave this topic for future research.

References

[1] Black F, Scholes M. The pricing of options and corporate liabilities. Journal of political economy. 1973;81(3):637–654.

[2] Bayer C, Friz P, Gatheral J. Pricing under rough volatility. Quantitative Finance. 2016;16(6):887–904.

[3] Hutchinson JM, Lo AW, Poggio T. A nonparametric approach to pricing and hedging derivative securities via learning networks. The Journal of Finance. 1994;49(3):851–889.

[4] De Spiegeleer J, Madan DB, Reyners S, Schoutens W. Machine learning for quantitative finance: fast derivative pricing, hedging and fitting. Quantitative Finance. 2018;p. 1–9.

[5] Hernandez A. Model calibration with neural networks. 2016;.

[6] Hull J, White A. The general Hull-White model and supercalibration. Financial Analysts Journal. 2001;p. 34–43.

[7] Gatheral J. The volatility surface: a practitioner's guide. vol. 357. John Wiley & Sons; 2011.

[8] Heston SL. A closed-form solution for options with stochastic volatility with applications to bond and currency options. The review of financial studies. 1993;6(2):327–343.

[9] Bergomi L. Smile dynamics III. 2008;.

[10] Gatheral J, Jaisson T, Rosenbaum M. Volatility is rough. Quantitative Finance. 2018;18(6):933–949.

[11] McCrickerd R, Pakkanen MS. Turbocharging Monte Carlo pricing for the rough Bergomi model. Quantitative Finance. 2018;p. 1–10.

[12] Jäckel P. Let's be rational. Wilmott. 2015;2015(75):40–53.

[13] Buehler H, Gonon L, Teichmann J, Wood B. Deep hedging. 2018;.

[14] Hornik K. Approximation capabilities of multilayer feedforward networks. Neural networks. 1991;4(2):251–257.

[15] Cybenko G. Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems. 1989;2(4):303–314.

[16] Hornik K, Stinchcombe M, White H. Multilayer feedforward networks are universal approximators. Neural networks. 1989;2(5):359–366.

[17] Goodfellow I, Bengio Y, Courville A, Bengio Y. Deep learning. vol. 1. MIT press Cambridge; 2016.

[18] Wright S, Nocedal J. Numerical optimization. Springer Science. 1999;35(67-68):7.

[19] Kingma DP, Ba J. Adam: A method for stochastic optimization. arXiv preprint arXiv:14126980. 2014;.

[20] Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn DS, Smith K. Cython: The best of both worlds. Computing in Science & Engineering. 2011;13(2):31–39.

[21] Chollet F, et al.. Keras; 2015.

[22] Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, et al. Tensorflow: a system for large-scale machine learning. In: OSDI. vol. 16; 2016. p. 265–283.

[23] Saporito YF, Yang X, Zubelli JP. The Calibration of Stochastic-Local Volatility Models-An Inverse Problem Perspective. arXiv preprint arXiv:171103023. 2017;.

[24] Cui Y, del Baño Rollin S, Germano G. Full and fast calibration of the Heston stochastic volatility model. European Journal of Operational Research. 2017;263(2):625–638.

[25] Mrázek M, Pospíšil J, Sobotka T. On calibration of stochastic and fractional stochastic volatility models. European Journal of Operational Research. 2016;254(3):1036–1046.