Scientific Research Publishing

Journal of Mathematical Finance, 2020, 10, 255-266
https://www.scirp.org/journal/jmf
ISSN Online: 2162-2442
ISSN Print: 2162-2434

A General Framework of Derivatives Pricing

Liangliang Zhang

101 Washington Blvd, CT 06902, Stamford, USA
Email: lianglianzhang81@qq.com

How to cite this paper: Zhang, L.L. (2020) A General Framework of Derivatives Pricing. *Journal of Mathematical Finance*, **10**, 255-266.
https://doi.org/10.4236/jmf.2020.102016

Received: March 27, 2020
Accepted: May 10, 2020
Published: May 13, 2020

Copyright © 2020 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

In this paper, we outline a general framework of derivatives pricing. The framework consists of two modules. The first is a novel simulation and machine learning based calibration module and the second one is a pricing module, which originates from [1] and [2]. Numerical examples show good applicability of the proposed framework. The methodology of calibration utilizes machine learning and simulation methods, combined, to deliver high quality parameter inference results and the pricing module is generic and can be applied to any financial derivatives. The machine learning based pricing methodologies can also generate prices on a future simulation grid, which facilitates XVA computations. Our methodologies can be applied to any pricing problem and the calibration routine is general and useful whenever a parametric model needs to be estimated.

Keywords

Clustering, Machine Learning, Calibration, Asset Pricing, Curve Fitting

1. Introduction

Despite recent advancement in model-free reinforcement learning based derivatives pricing methods and market scenario generating schemes, parametric models remain an important aspect of financial modeling for OTC and exchange traded financial derivatives, because parametric models are well-understood and can be easily interpreted. Moreover, the sensitivity measures are easy to obtain. However, in today's banking practice, the parametric calibration and asset pricing are still ad-hoc, in that, different trading desks might use different models for the same set of risk factors. Moreover, models are of low dimensions in nature, because a joint calibration is time consuming and numerical optimization routines are often unstable and return boundary solutions. In addition, products involving complex dynamics are often treated with approximations that are not accurate or convergent.

Recent literature on machine learning calibration includes [3], in which the author proposes a deep learning and simulation based approach to calibrate option pricing models. In addition, [4] proposes a similar approach.

In this paper, we propose a novel framework to alleviate the mentioned difficulties in derivatives calibration and pricing. First, we propose a simulation-based calibration method, without the need to use numerical optimization routines to minimize the sum of squares, *i.e.*, the L^2 distance, between the model and the observed prices. The intermediate simulation results can be stored and re-used. Therefore, the proposed methodology is efficient: we only need initial simulation and calibration can be done in a fast manner in an on-going basis. Second, the calibration method does not need many evaluations of derivative prices, as opposed to a standard optimization routine, which might require thousands of iterations. Third, we leverage the methods proposed in [1] and [2] for the pricing of complex financial derivatives, potentially involving optimal stopping features or other exotic properties such as a breakable swap, where both parties can terminate the contract to the best of their interest (and therefore a stochastic Nash equilibrium has to be found in order for us to obtain the price of this product). Clustering method, as an unsupervised learning method, was first applied to yield nonlinear regression computations in [5] and [2]. In this paper, we apply this approach to calibration of financial derivatives. To the best of our knowledge, our paper is the first to propose such a method. The algorithm is very easy to implement, fast and accurate. Numerical experiments show that it can give an accurate estimate to the speed of mean reversion parameter of a CIR process, which is thought to be very difficult to infer using either bond or bond option data. The calibration methodology has the potential to support joint inference using derivatives from different asset classes using data from both P and Q measures.

The organization of this paper is as follows. Section 2 introduces the main methodologies. Section 3 contains numerical experiments and Section 4 concludes. All the source code can be found in **Appendix**.

2. The Methodology

In what follows, we will assume that a pricing model (and equivalently, the model prices) is denoted by $M(X,\vartheta,\theta,C)$, where X is a set of state variables described by the model, ϑ is the model parameters related to X and θ is the parameters related to the financial product. C, defined as the set of control parameters, is related to a numerical method that solves the model. We denote M_{θ}^{mkt} the market observed prices for product θ. We use risk neutral derivatives pricing as an example to illustrate ideas, with the understanding that the method is generic and applies to all the asset pricing problems.

2.1. Calibration

The calibration method first simulates N uniform samples of parameter ϑ.

Usually, N increases with the dimension of ϑ. For each simulated parameter ϑ_n, we can evaluate the model price $M\left(X,\vartheta,\vartheta_n,C\right)$ and define $\vartheta^* = \arg\min_n \left\| M\left(X,\vartheta,\vartheta_n,C\right) - M_\theta^{mkt} \right\|^2$, *i.e.*, the specific simulated parameter ϑ_n that minimizes the distance between model produced prices and market observed prices. As we expect, when $N \to \infty$, the estimated parameter $\vartheta^* \to \vartheta$, the true parameter value.

The above methodology works theoretically. However, it is difficult, or often time consuming to implement in practice. The reason is that, even for a European type product, with long time to maturity and no closed-form solution, it might take a long time for the pricer to produce even one sufficiently accurate price, let alone the American products. Often, ϑ is in high dimension, given a portfolio of financial derivatives, and N is large. This often implies an unreasonably large amount of time needed for the estimation.

An improvement utilizes clustering method on the simulated parameter space $\Theta = \left\{\vartheta_n\right\}_{n=1}^N$. For example, we can divide the Θ into K clusters $\left\{\Theta_k\right\}_{k=1}^K$, such that for each $1 \le k \le K$, we have $\left\|\Theta_k\right\| \le \epsilon$, where $\left\|\cdot\right\|$ is the radius operator of a finite set and $\epsilon > 0$ is a small positive number. In each of the cluster Θ_k, denote its centroid by $\overline{\Theta_k}$, valuate the model at each $\overline{\Theta_k}$: $M\left(X,\vartheta,\overline{\Theta_k},C\right)$ and obtain K prices $\left\{M\left(X,\vartheta,\overline{\Theta_k},C\right)\right\}_{k=1}^K$. Find $k^* = \arg\min_k \left\{M\left(X,\vartheta,\overline{\Theta_k},C\right)\right\}_{k=1}^K$. Next, let us focus on cluster Θ_{k^*}. As long as $\left|\Theta_{k^*}\right| \ge K$, *i.e.*, the number of elements in Θ_{k^*} is no less than K, we can repeat the above operations, until we find a $\hat{k}$ such that $\left|\Theta_{\hat{k}}\right| < K$. Then, use the centroid $\overline{\Theta_{\hat{k}}}$ as the estimator of ϑ.

The complexity of the algorithm grows in a logarithm manner with respect to N. Assume that $\alpha = \left\lfloor log_K^N \right\rfloor$, the integer part of log_K^N, then, the total number of evaluation times is $K \times \alpha$. The method ensures that we can find the optimal parameter values quickly without the need to evaluate the pricer at each simulated value of the model parameter.

The choice of the numerical method to implement the pricer is open to the preference of each user of our framework. It can be brute-force Monte Carlo simulation, analytical expansion, asymptotic expansion or other approximation methodologies.

2.2. Pricing

We use the pricing of financial derivatives as an example to illustrate ideas. Under no arbitrage framework and some sufficient condition, the present value of all marketed cash flows is martingales under a so-called risk-neutral measure, or Q measure. In general, derivatives pricing follows a reduce-form approach that assumes an underlying price distribution and compute the conditional expected value of the discounted payoff function. The underlying can be modeled by a discrete time-series or a system of stochastic differential equations. The simulated-based numerical methods for the latter case are discussed in [1] and [2].

We refer the readers to those references for more details.

2.3. XVA

The proposed methodologies in [1] and [2] enable evaluation in a future simulation grid and this is the foundation for XVA evaluations.

3. Numerical Experiments

In this paper, we will mainly test the calibration method, with the pricing component already validated in the reference of [1] and [2]. Due to the constraint on the computational budget, we focus on simple products to illustrate ideas. The method we adopt for the pricer is Monte Carlo simulation, which is relatively more time consuming than semi closed-form solutions.

3.1. Heston European Equity Option Pricing Model

Assume that under the risk neutral measure, the stock price follows a Heston- type stochastic volatility model, with parameter values described in Table 1 below.

In order to estimate the true values, we generate 75,000 uniform samples of $(\kappa,\theta,\sigma,\rho)$ in interval $[0.0000,1.5000]\times[0.0000,0.0900]\times[0.0000,0.5000]\times[-1.0000,1.0000]$. Using the methodology outlined in Section 2.1, we have the following estimates. Risk free rate is 1.00%, time to maturity is 0.50 years and the option prices are evaluated via Monte Carlo simulation method with 75,000 sample paths and 50 time discretization points. Figure 1 shows the price fit for case 1. Blue curve is the true price[1] and the orange curve represents the calibrated prices at maturity date across different strikes. We choose 250 clusters for the estimation. Table 2 contains the results.

3.2. CIR Bond Pricing Model

In this section, we study a zero-coupon bond pricing problem, where the short rate process follows a Cox-Ingersoll-Ross model. The parametrization of the problem is listed in Table 3 and inference result is in Table 4. We use a whole term structure of bond prices to calibrate the model. For more details, we refer the interested readers to the sample code in the Appendix. The pricing fit is in Figure 2.

Table 1. Parameter Values. $(\kappa,\theta,\sigma,\rho)$ are speed of mean reversion, long term mean, volatility of volatility and correlation parameters.

Index	κ	θ	σ	ρ
1	1.1000	0.0400	0.2500	−0.2500
2	0.5000	0.0225	0.1500	−0.5000
3	1.2500	0.0750	0.3000	−0.7500

[1]The true prices are generated by Monte Carlo approach with the true parameter values.

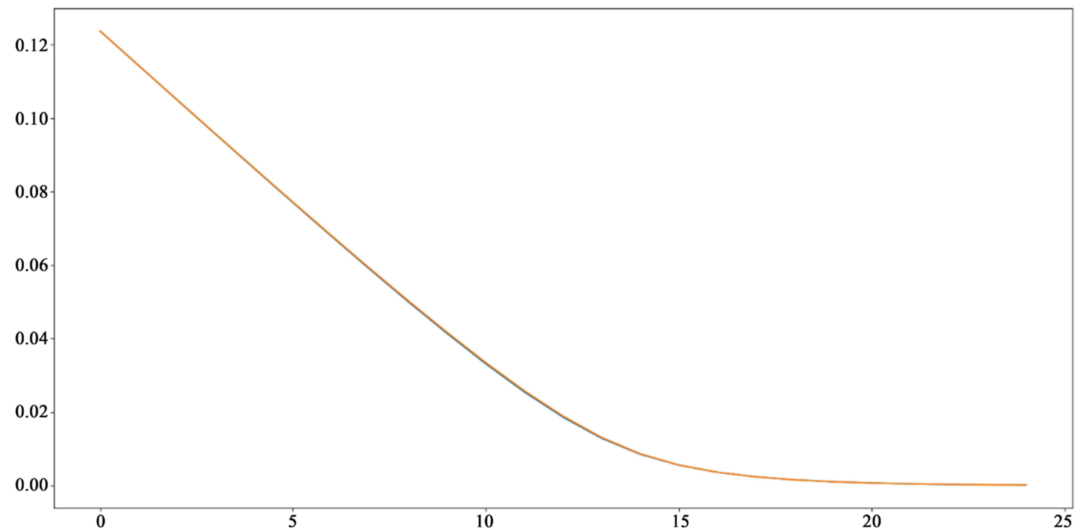

Figure 1. Price fitting curve.

Table 2. Estimated parameter values.

Index	κ	θ	σ	ρ
1	1.0078	0.0400	0.2324	−0.1919
2	0.1179	0.0234	0.1340	−0.4600
3	1.2763	0.0695	0.3245	−0.6665

Table 3. CIR short rate parameter table.

Index	κ	θ	σ	r_0
1	0.6000	0.0150	0.1500	0.0100
2	0.4000	0.0225	0.1500	0.0100
3	1.2500	0.0350	0.3000	0.0100

Table 4. Inference table.

Index	κ	θ	σ	r_0
1	0.6514	0.0151	0.2172	0.0100
2	0.3230	0.0264	0.2845	0.0100
3	1.3929	0.0336	0.2845	0.0100

3.3. Vasicek Bond Option Pricing Model

The results are listed in the Table 5 and Table 6, and Figure 3. Details of this exercise can be found in the source code.

4. Conclusion

The main contribution of this paper is a general framework of financial asset pricing and calibration, where the calibration module consists of a novel simulation and clustering-based methodology. The simulated numbers and intermediate

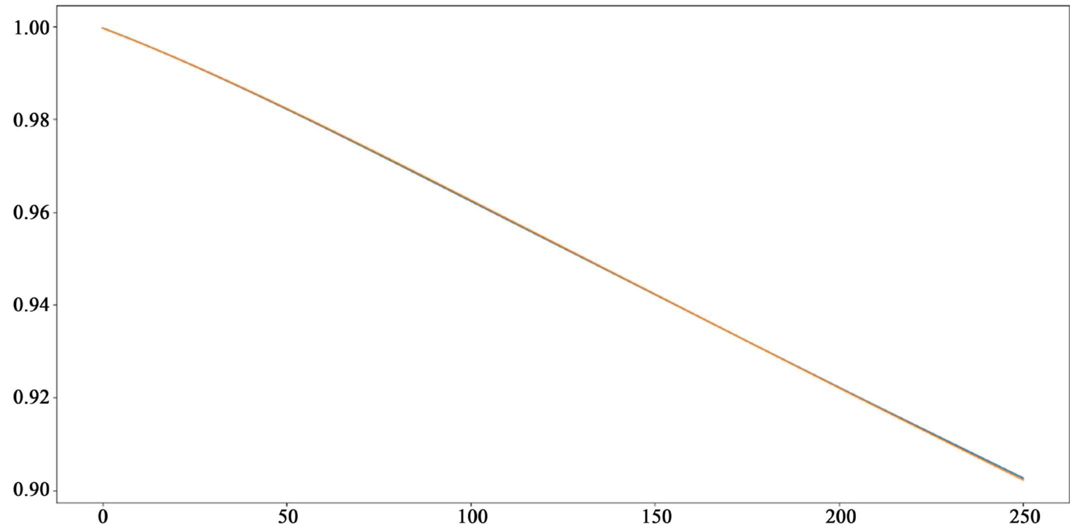

Figure 2. Bond pricing fit for case 1.

Table 5. True parameters.

Index	κ	θ	σ	r_0
1	0.6000	0.0150	0.1500	0.0100
2	0.4000	0.0225	0.1500	0.0100
3	1.2500	0.0350	0.3000	0.0100

Table 6. Inference results.

Index	κ	θ	σ	r_0
1	0.5461	0.0133	0.1469	0.0100
2	0.3490	0.0226	0.1450	0.0100
3	1.0868	0.0385	0.2844	0.0100

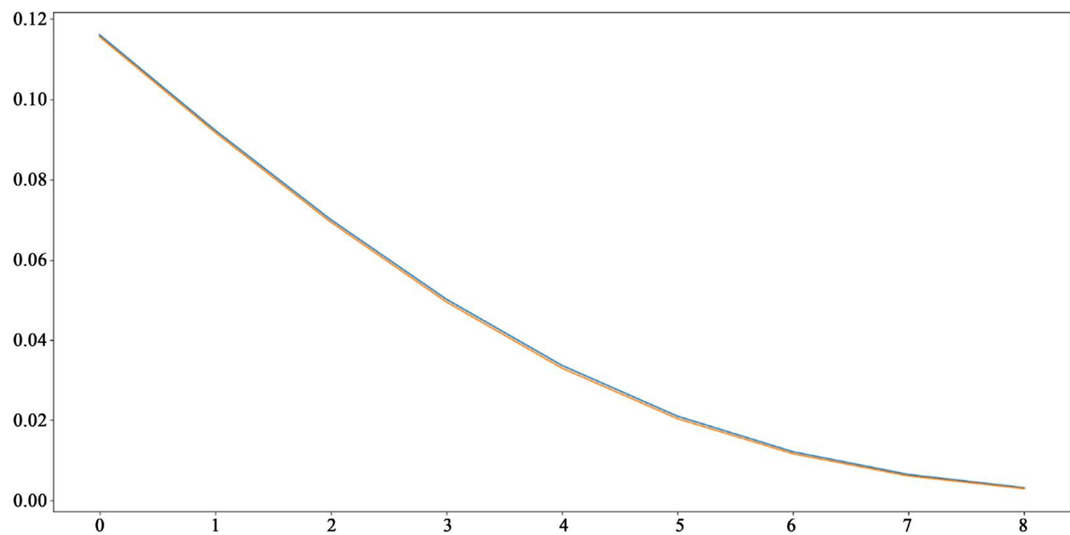

Figure 3. Price fit.

pricing results can be re-used and are therefore very efficient. The methodology potentially applies to any problem that requires curve fitting, *i.e.*, minimizing a parametric objective function and obtaining the optimal parameters.

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Ye, T. and Zhang, L. (2019) Derivatives Pricing via Machine Learning. *Journal of Mathematical Finance*, **9**, 561-589. https://doi.org/10.4236/jmf.2019.93029

[2] Zhang, L. (2020) A Clustering Method to Solve Backward Stochastic Differential Equations with Jumps. *Journal of Mathematical Finance*, **10**, 1-9. https://doi.org/10.4236/jmf.2020.101001

[3] Itkin, A. (2019) Deep Learning Calibration of Option Pricing Models: Some Pitfalls and Solutions.

[4] Li, S., Borovykh, A., Grzelak, L.A. and Oosterlee, C. (2019) A Neural Network-Based Framework for Financial Model Calibration.

[5] Zhang, L. (2019) Asset Return Prediction via Machine Learning. *Journal of Mathematical Finance*, **9**, 691-697. https://doi.org/10.4236/jmf.2019.94035

Appendix: Sample Source Code

A1. Heston Option Pricing

```
# Python Code for Calibration
import numpy                    as      np
import matplotlib.pyplot as      plt
from     sklearn.cluster    import MiniBatchKMeans
# Parameters
r              = 0.01
kappa          = 1.25
theta          = 0.075
sigma          = 0.30
rho            = -0.75
H              = 0.50
N              = 50
h              = H / N
MUnif          = 75000
MNorm          = 75000
S0             = 1.00
v0             = 0.15 ** 2
TimeNode       = np.array([5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37, 40, 42, 45, 47, 50])
Clusters       = 250
StrikeLen      = 25

# Rrandom Numbers
dWt            = np.random.normal(0, np.sqrt(h), [N, MNorm])
dBt            = np.random.normal(0, np.sqrt(h), [N, MNorm])
KappaRnd       = np.random.uniform(0.000, 1.50, MUnif)
ThetaRnd       = np.random.uniform(0.000, 0.30 ** 2, MUnif)
SigmaRnd       = np.random.uniform(0.000, 0.50, MUnif)
RhoRnd         = np.random.uniform(-1.00, 1.00, MUnif)
Y              = np.transpose(np.array([KappaRnd, ThetaRnd, SigmaRnd, RhoRnd]))

# Pricer Definition
def StrikeFunc(x, y, z):
    Upper       = S0 * (1 + 0.6 * x / y * z * np.sqrt(v0))
    Lower       = np.maximum(0, S0 * (1 - 0.6 * x / y * z * np.sqrt(v0)))
    return(np.linspace(Lower, Upper, StrikeLen))

def HestonPrice(x, y, z, w):
    S                      = S0 * np.ones([N+1, MNorm])
    V                       = v0 * np.ones([N+1, MNorm])
for i in range(N):
        S[i + 1, :]       = S[i, :] * (1 + r * h + V[i, :] * dWt[i, :])
        V[i + 1, :]       = V[i, :] + x * (y - V[i, :]) * h + \
            z * np.sqrt(np.abs(V[i, :])) * (w * dWt[i, :] + np.sqrt(1 - w ** 2) * dBt[i, :])
    Price                  = np.zeros([len(TimeNode), StrikeLen])
```

```
for j in range(len(TimeNode)):
            Strike                  = StrikeFunc(TimeNode[j] + 1, N, H)
for k in range(len(Strike)):
Price[j, k] = np.mean(np.maximum(0, S[TimeNode[j], :] - Strike[k]))
      return(np.exp(-r * H) * Price)

# True Solution
PriceTrue               = HestonPrice(kappa, theta, sigma, rho)

# Clustering
Div                         = MUnif / Clusters
while(Div >= Clusters):
kmeans              = MiniBatchKMeans(n_clusters       = Clusters,
random_state          = 0,
batch_size              = 256,
max_iter                 = 20000).fit(Y)
YCenters    = kmeans.cluster_centers_
      LSE            = np.zeros(Clusters)
      Prices       = np.zeros([Clusters, len(TimeNode), StrikeLen])
for j in range(Clusters):
            Prices[j, :, :] = HestonPrice(YCenters[j][0], YCenters[j][1], YCenters[j][2], YCenters[j][3])
            LSE[j]                  = np.sqrt(np.mean((Prices[j, :, :] - PriceTrue) ** 2 / 1 ** 2))
Idx             = np.argmin(LSE)
KmIdx        = np.where(kmeans.labels_ == Idx)[0]
PricesFit = Prices[Idx, :, :]
      Y               = Y[KmIdx, :]
Div             = Y.shape[0]
Params              = YCenters[Idx]

# Plots
plt.plot(PricesFit[-1, :])
plt.plot(PriceTrue[-1, :])
print('Mean Squared Error: ', np.sqrt(np.mean((PricesFit - PriceTrue) ** 2)))
print('Params Fitted: ', Params)
print('Params True', [kappa, theta, sigma, rho])
```

A2. CIR Bond Pricing Model

```
# Python Code for Calibration
import numpy                        as         np
import matplotlib                as         plt
from      sklearn.cluster      import MiniBatchKMeans
from      matplotlib.pyplot import plot
# Parameters
r0                  = 0.010
kappa             = 0.600
theta              = 0.015
sigma             = 0.150
```

```
H           = 7.50
N           = 250
h           = H / N
MUnif       = 50000
MNorm       = 100000
Clusters    = 200

# Rrandom Numbers
dWt         = np.random.normal(0, np.sqrt(h), [N, MNorm])
KappaRnd    = np.random.uniform(0.000, 1.50, MUnif)
ThetaRnd    = np.random.uniform(0.000, 0.03, MUnif)
SigmaRnd    = np.random.uniform(0.000, 0.30, MUnif)
Y           = np.transpose(np.array([KappaRnd, ThetaRnd, SigmaRnd]))

# Pricer Definition
def BondPrice(x, y, z):
    V                   = r0 * np.ones([N+1, MNorm])
    C                   = r0 * np.ones([N+1, MNorm])
for i in range(N):
        V[i + 1, :]     = V[i, :] + x * (y - V[i, :]) * h + z * np.sqrt(np.abs(V[i, :])) * dWt[i, :]
        C[i + 1, :]     = C[i, :] + V[i + 1, :]
    Price               = np.mean(np.exp(-C * h), 1)
    return(Price)

# True Solution
PriceTrue   = BondPrice(kappa, theta, sigma)

# Clustering
Div         = MUnif
while(Div >= Clusters):
kmeans      = MiniBatchKMeans(n_clusters    = Clusters,
random_state  = 0,
batch_size    = 256,
init_size     = 256,
max_iter      = 20000).fit(Y)
YCenters    = kmeans.cluster_centers_
    LSE         = np.zeros(Clusters)
    Prices      = np.zeros([Clusters, N+1])
for j in range(Clusters):
        Prices[j, :]    = BondPrice(YCenters[j][0], YCenters[j][1], YCenters[j][2])
        LSE[j]          = np.sqrt(np.mean((Prices[j, :] - PriceTrue) ** 2))
    Idx             = np.argmin(LSE)
    KmIdx           = np.where(kmeans.labels_ == Idx)[0]
    PricesFit       = Prices[Idx, :]
    Y               = Y[KmIdx, :]
    Div             = Y.shape[0]
    print('Cluster Length: ', Div)
```

```
Params          = YCenters[Idx]

# Plots
plt.pyplot.plot(PricesFit)
plt.pyplot.plot(PriceTrue)
print('Mean Squared Error: ', np.sqrt(np.mean((PricesFit - PriceTrue) ** 2)))
print('Params Fitted: ', Params)
print('Params True', [kappa, theta, sigma])
```

A3. Vasicek Bond Option Pricing Model

```
# Python Code for Calibration
import numpy                    as      np
import matplotlib               as      plt
from    sklearn.cluster         import MiniBatchKMeans
from    matplotlib.pyplot       import plot
from    sklearn.linear_model import LinearRegression
# Parameters
r0          = 0.0100
kappa       = 0.4000
theta       = 0.0225
sigma       = 0.1500
HBond       = 1.0000
HOption     = 0.5000
N           = 50
h           = HBond / N
LookBack    = int((HBond - HOption) / h)
K           = [0.90, 0.925, 0.95, 0.975, 1.00, 1.025, 1.05, 1.075, 1.10]
MUnif       = 75000
MNorm       = 75000
Clusters    = 250

# Rrandom Numbers
dWt         = np.random.normal(0, np.sqrt(h), [N, MNorm])
KappaRnd    = np.random.uniform(0.000, 1.50, MUnif)
ThetaRnd    = np.random.uniform(0.000, 0.05, MUnif)
SigmaRnd    = np.random.uniform(0.000, 0.30, MUnif)
Y           = np.transpose(np.array([KappaRnd, ThetaRnd, SigmaRnd]))

# Pricer Definition
def FutureBondPrice(x, y, z):
    V                   = r0 * np.ones([N+1, MNorm])
    C                   = r0 * np.ones([N+1, MNorm])
    for i in range(N):
        V[i + 1, :]     = V[i, :] + x * (y - V[i, :]) * h + z * dWt[i, :]
    Regression          = LinearRegression().fit(np.array(V[N-LookBack, :]).reshape(-1, 1),
                                        -np.array(np.sum(V[(N-LookBack):(N+1),    :],    0)    *
h).reshape(-1, 1))
```

```
    Price                     = np.exp(-Regression.predict(np.array(V[N-LookBack, :]).reshape(-1, 1)))
    return(Price)

def BondOptionPrice(x, y, z):
    FutureBond                = FutureBondPrice(x, y, z)
    PriceTemp                 = np.zeros(len(K))
for i in range(len(K)):
        PriceTemp[i]          = np.mean(np.maximum(FutureBond - K[i], 0))
    return(PriceTemp)

# True Solution
PriceTrue                     = BondOptionPrice(kappa, theta, sigma)

# Clustering
Div              = MUnif
while(Div >= Clusters):
kmeans       = MiniBatchKMeans(n_clusters       = Clusters,
random_state  = 0,
batch_size       = 256,
init_size          = 256,
max_iter          = 20000).fit(Y)
YCenters          = kmeans.cluster_centers_
    LSE           = np.zeros(Clusters)
    Prices        = np.zeros([Clusters, len(K)])
for j in range(Clusters):
Prices[j, :]        = BondOptionPrice(YCenters[j][0], YCenters[j][1], YCenters[j][2])
        LSE[j]                = np.sqrt(np.mean((Prices[j, :] - PriceTrue) ** 2))
    Idx            = np.argmin(LSE)
    KmIdx          = np.where(kmeans.labels_ == Idx)[0]
    PricesFit = Prices[Idx, :]
    Y              = Y[KmIdx, :]
    Div            = Y.shape[0]
    print('Cluster Length: ', Div)
Params              = YCenters[Idx]

# Plots
plt.pyplot.plot(PricesFit)
plt.pyplot.plot(PriceTrue)
print('Mean Squared Error: ', np.sqrt(np.mean((PricesFit - PriceTrue) ** 2)))
print('Params Fitted: ', Params)
print('Params True', [kappa, theta, sigma])
```