Uncertainty Quantification of the Black-Scholes Model

Lihui Ji*
**Civil and Environmental Engineering, lihui.ji@duke.edu
Duke University, Durham, NC, 27708

Abstract

In this project, various methods are implemented to quantify uncertainty of the Black-Scholes Model. The parameter uncertainty comes from volatility measure with certain distribution. The methods include Maximum Entropy Analysis, Monte-Carlo approach, Polynomial Chaos Expansion, and Stochastic Collocation Methods.

1 Introduction

1.1 Background

Option is a financial instrument which gives investors the right to long or short assets at a fixed price in the future. European option can be only exercised at maturity while American option can be exercised anytime no later than maturity. A call option is the right to buy the assets, speculating that the price will go up or hedging risk of short positions. In our case, we focus on European options, noting that American options are slightly more complicated. In general, they are similar because there is a rare case to exercise the option prior to maturity (traders usually get more by selling options to others rather than by exercising them considering the time value for options). The call option is considered in this problem (put option has a different format but is fundamentally the same).

The Black-Scholes Model is essential to determine option price and guide option trading strategies. However, in most cases, simply representing current parameters with measures omits the future's change. Even though, the measure with historic data can fail to represent current value exactly. Therefore, epistemic uncertainty is introduced.

1.2 Governing Equation

The Black-Scholes PDE reads:

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

The solution to Black-Scholes PDE with call options reads:

$$\begin{aligned} C(S,t) &= S_t\phi(d_1) - e^{-r(T-t)}K\phi(d_2) \\ where\ \ d_1 &= \frac{log(\frac{S_t}{K}) + (r+\sigma^2/2)(T-t)}{\sigma\sqrt{T-t}} \\ and\ \ d_2 &= d_1 - \sigma\sqrt{T-t} \end{aligned} \tag{2}$$

In the above equations, C is call option price, S is stock price, K is strike price for options, r is risk free rate, which is approximated by the interest rate on a three-month U.S. Treasury bill for U.S.-based investors, σ is volatility, the standard deviation of stock price, t is time elapsed, and T is time to maturity.

$$dS = \mu S dt + \sigma S dW_t \tag{3}$$

2 Methods and Results

2.1 Data Preparation

The QuantConnect, an online financial research portal to get the data of option chain. One option chain comprises all call and put options with different strike price underlying one asset. In this project, S&P 500 index option is used, which is more general because it tracks the 500 largest capitalization stocks thus represents the total stock market. The data resolution is minute, the best resolution data available in QuantConnect. In the project scope, only call option is investigated. The data processing is completed in MATH551 Algorithm Trading class project.

For each option, all parameters are known in equation(2) except the implied volatility σ. By using iteration methods, the implied volatility is obtained. Then we have a dataset of implied volatility with 15209 data points total. The smoothed probability distribution probability is plotted below:

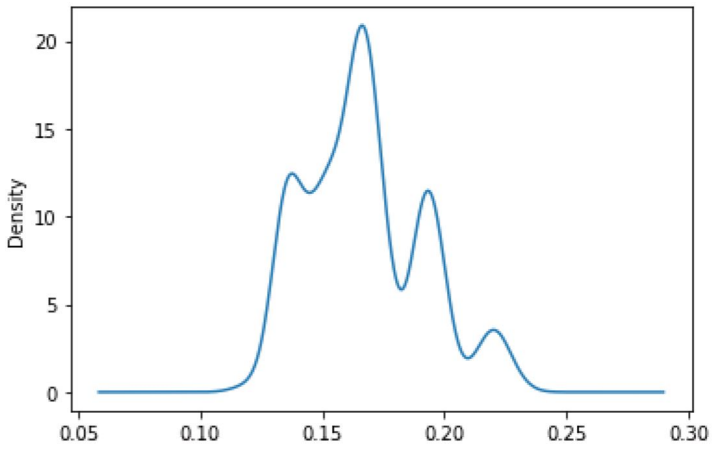


Figure 1: Probability Distribution of Implied Volatility

The figure is analyzed, and some information can be extracted: First, the extreme volatility is rare, as the curve goes flat zero at the left and right side. Second, the average implied volatility is approximately the same with most frequent implied volatility. We can roughly observe the same area of < 0.16 part and > 0.16 part even though they have different spikes. Third, the implied volatility is discontinuous. The implied volatility concentrates at several spikes and change with a large jump.

2.2 Maximum Entropy Method

input variable is normalized to have a range [0,1]. Next step is to choose one distribution to fit the data for further uncertainly analysis. Because the volatility (standard deviation mathematically) is always positive thus un-symmetric, Gaussian distribution is inappropriate in this case. Here the author tried Beta distribution and Gamma distribution, which is always positive. The entropy between distribution candidate and data is calculated separately with formula below:

$$\mathcal{E}(f_X Y) = -E\{log(f_X Y(X, Y))\} \tag{4}$$

The entropy with Beta distribution is -3.52 and the entropy with Gamma distribution is -3.98. Neither of the result shows a good fit. However, based on the project propose to do exercise on what we have learned, Beta distribution is chosen for further analysis.

Therefore, we created the simplified problem. The X follows Beta Distribution, Y is determined by X with equation (2), how to quantify the uncertainty of Y led by X.

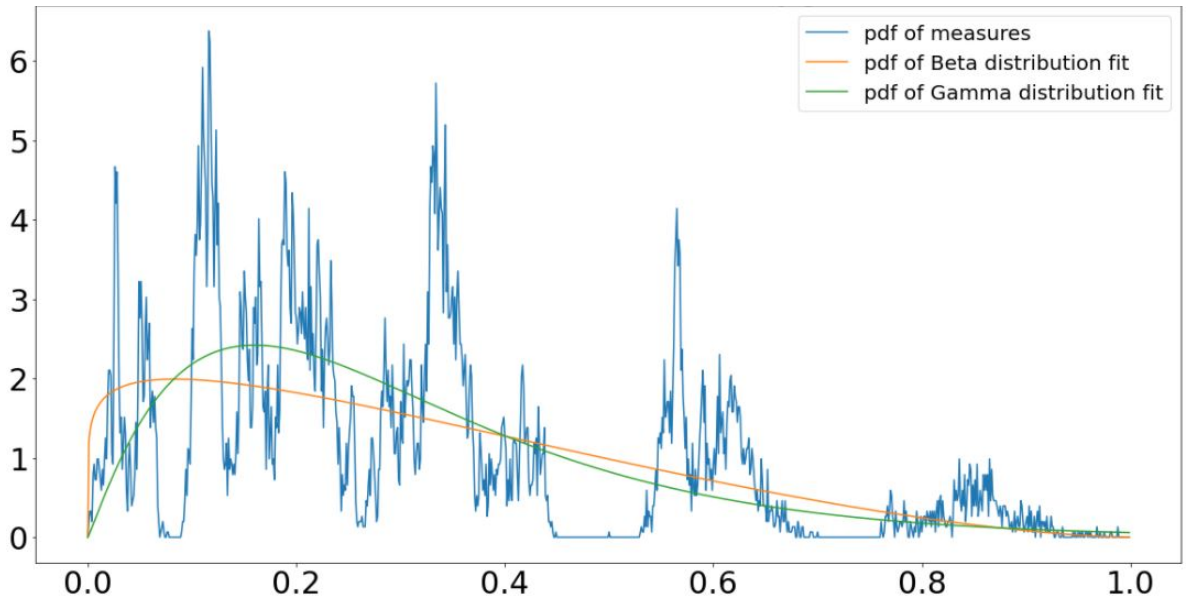


Figure 2: Probability Distribution of Implied Volatility

The Beta distribution probability density function reads:

$$P_X(dx) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}, \qquad where \;\; B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \tag{5}$$

The Gamma function Γ for positive z reads:

$$\Gamma(z) = \int_0^\infty x^{z-1}e^{-x}dx \tag{6}$$

For the Beta distribution, $\alpha = 1.1444307615618827$ and $\beta = 2.590929757651513$, which are determined by the measures. For other parameters, both stock price and strike price are set to 1, risk free rate is 0.0063 and time to maturity is 2 weeks ($2/365 = 0.00548$ years).

2.3 Monte Carlo Solver

First, we want to find out how many samples are enough to represent random field for Monte Carlo approach. by testing different number of samples(maximum 80000), estimates of the mean and second-order mean are plotted below. From the graph, 20000 is fairly enough for the convergence, i.e. representing the random field.

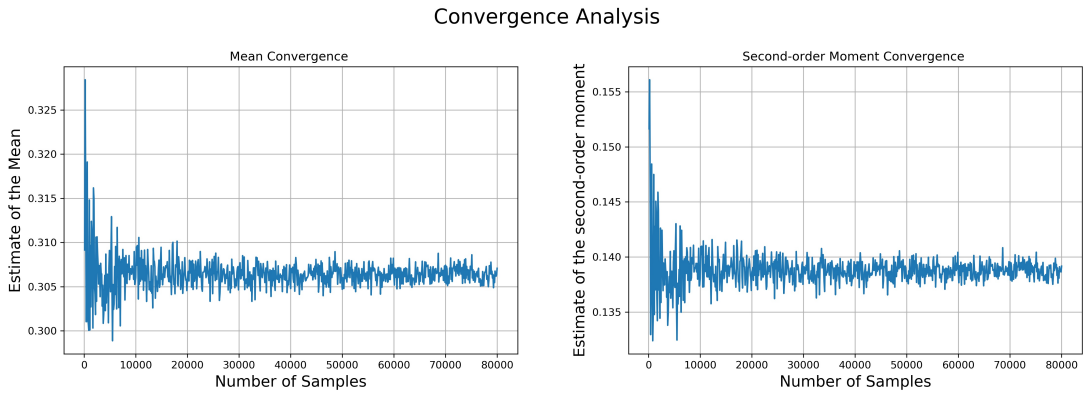
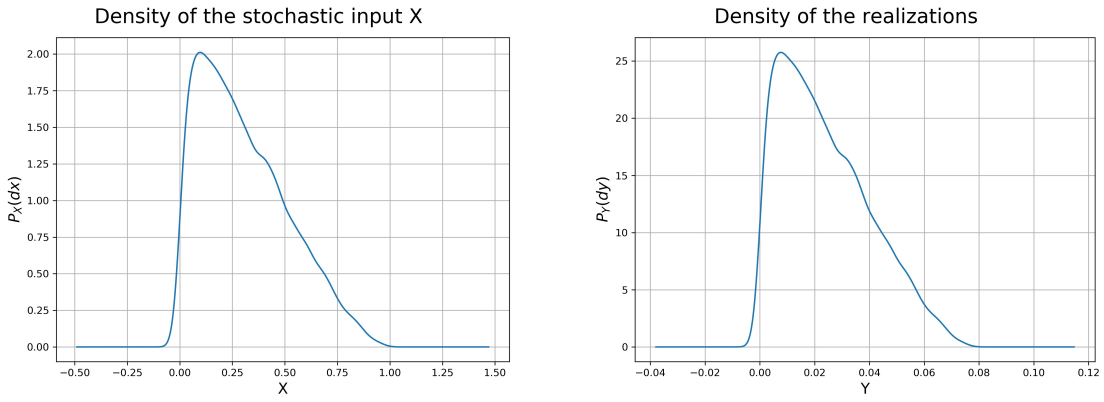


Figure 3: Estimates of First and Second Order of Mean

The result of Monte Carlo Solver is plotted below. The stochastic input X follows beta distribution, consistent with Figure 2. Figure 4(b) plots the pdf of realizations, which has similar distribution with input X.



(a) (b)

Figure 4: Monte Carlo Solver Result Visualization

2.4 Stochastic Modeling Through Series Expansions

The polynomial chaos expansion can be applied to construct mapping between variable (implied volatility) and result (option price). Therefore, the complex model, i.e. equation (2) is simplified with polynomials. This method can also handle black box models by construct mapping between X and Y.

As mentioned above, the Beta distribution is chosen to quantify the uncertainty X (implied volatility) measures. To match with distribution, Jacobi polynomials are implemented as basis

functions.

The Jacobi polynomials ($i = 0, 1, 2, ..., n$) are:

$$\psi_i^{(\alpha,\beta)}(z) = \frac{\Gamma(\alpha+i+1)}{i!\Gamma(\alpha+\beta+i+1)} \sum_{m=0}^{i} \binom{i}{m} \frac{\Gamma(\alpha+\beta+i+m+1)}{\Gamma(\alpha+m+1)} \left(\frac{z-1}{2}\right)^m \tag{7}$$

We want to decompose Y with the Jacobian polynomials:

$$Y = \sum_{i=0}^{+\infty} y_i \psi_i(X) \tag{8}$$

the coefficient y_i can be computed with the eigenfunction expansion:

$$y_i = < Y, \psi_i >_{\mathbb{H}} = E\{h(X)\psi_i(X)\} = \int_R h(x)\psi(x)P_X(dx) \tag{9}$$

Besides integrate directly, Monte Carlo approach can be also used to find coefficients:

$$y_i \approx \frac{1}{N_{MC}} \sum_{j=1}^{N_{MC}} h(x(\theta_j))\psi_i(x(\theta_j)) \tag{10}$$

However, this problem is much more complicated than what we learned in the course.

First, the beta distribution domain [0,1] is different the integration domain of Jacobi polynomial, which is [-1,1]. A map is constructed to make Jacobi polynomials match with beta distribution: $X = 1 - 2X$.

After mapping, the probability density function of X becomes:

$$P_X(dx) = \frac{1}{2} \frac{((1-x)/2)^{\alpha-1}(1-(1-x)/2)^{\beta-1}}{B(\alpha,\beta)}, \quad where \;\; B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \tag{11}$$

A test of mapped probability density distribution is shown below. First X is generated random variables between [0,1], and $X = 1 - 2X$ mapping is performed and density histogram is plotted. On the other hand, the adjusted pdf is plotted. The two figures match well with each other.

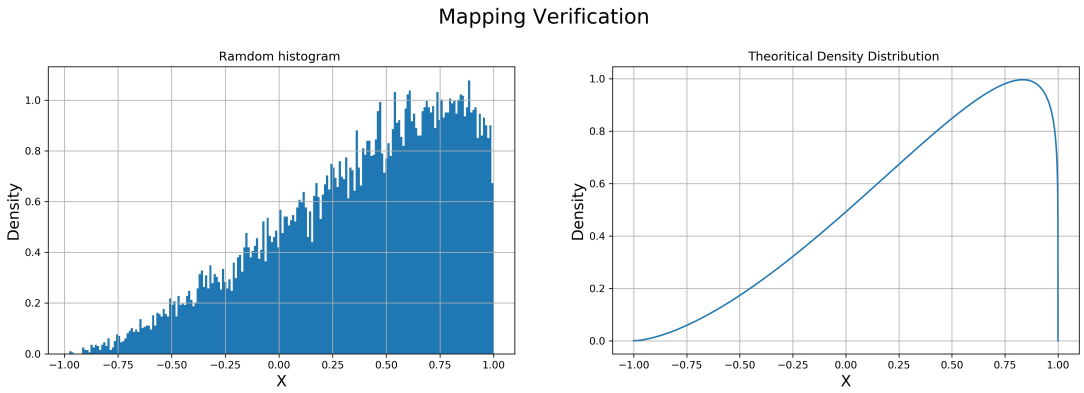


Figure 5: Mapping Verification

Second, the Jacobi polynomials are orthogonal only with weight function $\sigma(x) = (1-x)^{\alpha}(1+x)^{\beta}$, which is already embedded in probability density function. Also the self inner product is not 1, but with a expression depending on the order of polynomial:

$$\frac{2^{\alpha+\beta+1}}{2i+\alpha+\beta+1}\frac{\Gamma(i+\alpha+1)\Gamma(i+\beta++1)}{\Gamma(i+\alpha+\beta+1)n!}.$$

The orthogonal condition of Jacobi polynomials satisfies:

$$\int_{-1}^{1}(1-x)^{\alpha}(1+x)^{\beta}\psi_j^{(\alpha,\beta)}(x)\psi_i^{(\alpha,\beta)}(x)dx = \frac{2^{\alpha+\beta+1}}{2i+\alpha+\beta+1}\frac{\Gamma(i+\alpha+1)\Gamma(i+\beta++1)}{\Gamma(i+\alpha+\beta+1)n!}\delta_{ij} \quad (12)$$

According to this specific problem, equation (9) becomes:

$$y_i =< Y, \psi_i >_{\mathbb{H}} = E\{h(X)\psi_i(X)\} = \int_{-1}^{1} h((1-x)/2)\widetilde{\psi}(x)P_X(dx)$$
$$where \quad \widetilde{\psi}(x) = \psi(x)/\sqrt{\frac{\Gamma(i+\alpha)\Gamma(i+\beta)\Gamma(\alpha+\beta)}{(2i+\alpha+\beta-1)\Gamma(i+\alpha+\beta-1)i!\Gamma(\alpha)\Gamma(\beta)}} \quad (13)$$

Before applying equation (13) to get results, equation (12), the fundamental orthogonal property is tested first to verify the model setup. Due to the complexity, rather than iterate with software, the domain [-1,1] is divided to bins with width 0.001 for each to estimate integration. As the diagram shows below, the discretization reaches accuracy of around 0.001, which is fairly enough for research purpose. Higher accuracy can be achieved by increasing resolution.

	0	1	2	3	4
0	0.999789419002016	-0.000304472239597669	-0.000383338867566917	-0.000453670196368348	-0.000518151339530541
1	-0.000304472239597669	0.999559769606514	-0.000554252913307230	-0.000655958381653051	-0.000749164531746407
2	-0.000383338867566917	-0.000554252913307230	0.999302170885729	-0.000825841774924408	-0.000943248442348874
3	-0.000453670196368348	-0.000655958381653051	-0.000825841774924408	0.999022591974159	-0.00111624878145781
4	-0.000518151339530541	-0.000749164531746407	-0.000943248442348874	-0.00111624878145781	0.998725008919918

Figure 6: The result of checking Jacobi polynomials

The normalized polynomials are also tested and the result is shown below:

	0	1	2	3	4
0	0.999789419002016	-0.000304472239597765	-0.000383338867566863	-0.000453670196368365	-0.000518151339530545
1	-0.000304472239597765	0.999559769606514	-0.000554252913307099	-0.000655958381653016	-0.000749164531746435
2	-0.000383338867566863	-0.000554252913307099	0.999302170885729	-0.000825841774924314	-0.000943248442348878
3	-0.000453670196368365	-0.000655958381653016	-0.000825841774924314	0.999022591974159	-0.00111624878145791
4	-0.000518151339530545	-0.000749164531746435	-0.000943248442348878	-0.00111624878145791	0.998725008919918

Figure 7: The result of checking normalized polynomials

The y_i coefficients are solved by integration with bins approximation. The array of Y_i result is below:

[0.05424, -0.11564, 0.14227, -0.145694, 0.143758]

Plugging these coefficients, Y approximations of different polynomial numbers are obtained and compared with Monte Carlo results.

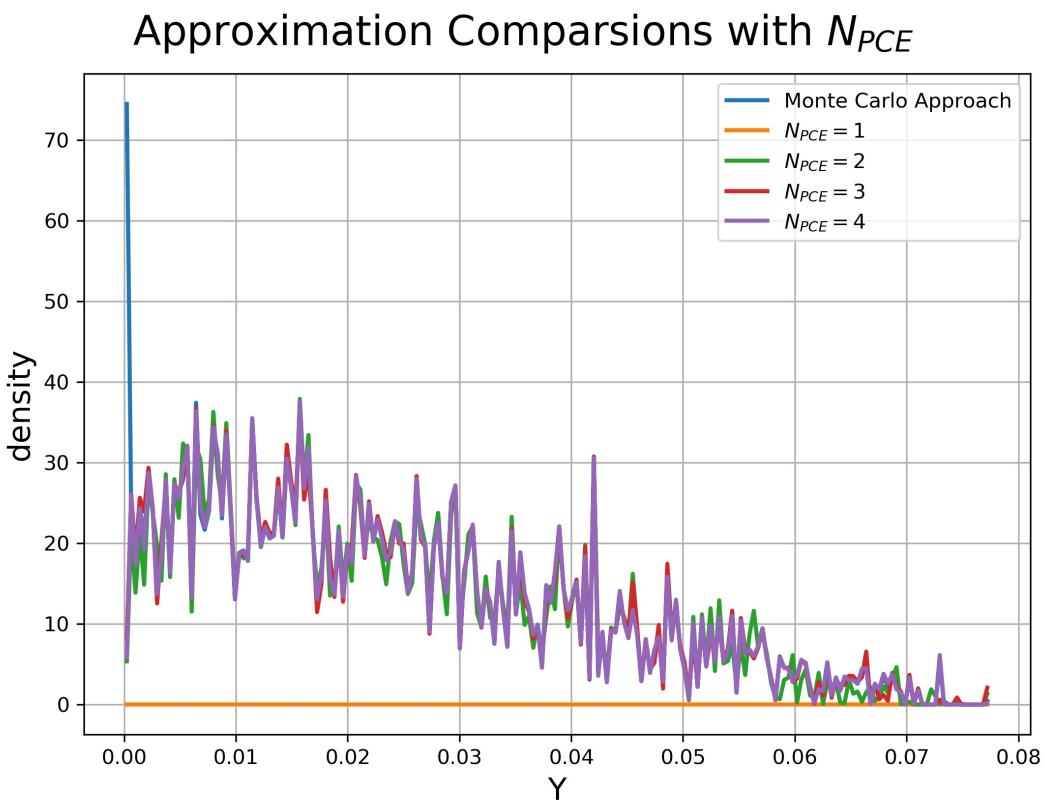


Figure 8: Mapping Verification

Intuitively, despite $N_{PCE} = 1$, higher order expansions show relatively good approximation. However, all of them failed to catch the spike at $Y \approx 0$. The mean square errors for different polynomial numbers comparing to Monte Carlo result are calculated, and plotted below. The convergence is observed, as error approaches to zero (10^{-11}) as order increases.

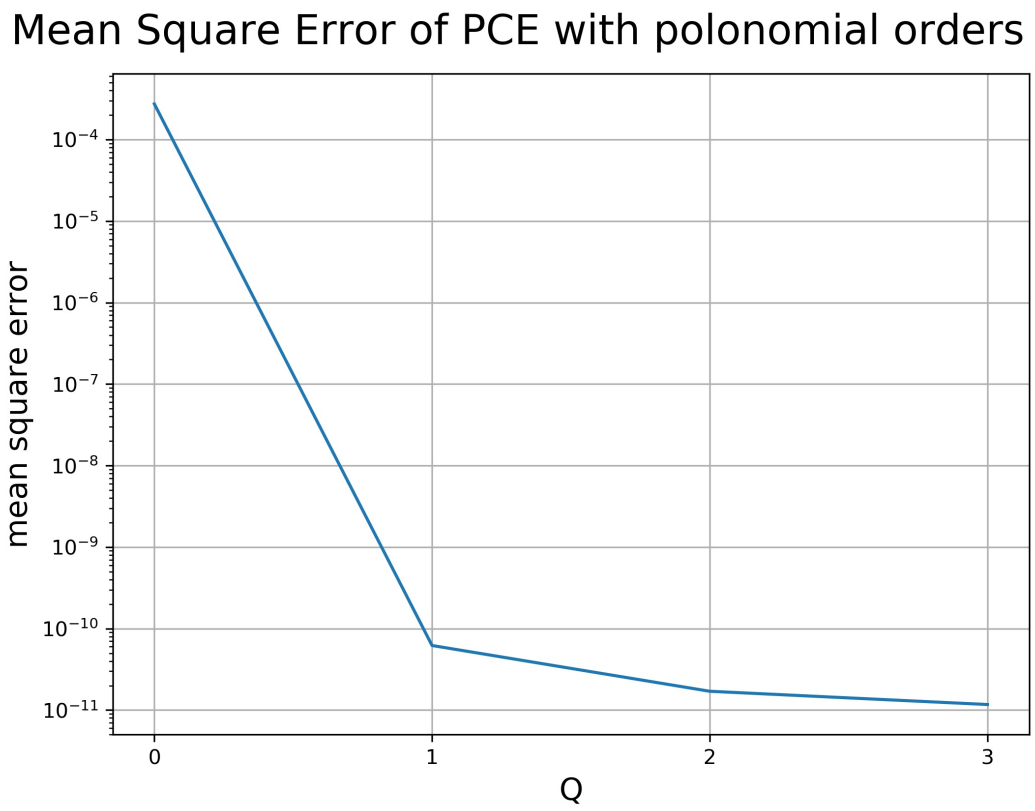


Figure 9: Mapping Verification

2.5 Stochastic collation method

Stochastic collation method is to first calculate mapped values with certain sampled variable points (X), then calculate approximate functions with the mapped value (Y). The input points are called collocation points.

In 1D sense, stochastic collation method with pseudo projection is equivalent to the Monte Carlo approach for PCE analysis, where both Y realizations from X samples are utilized to get the distribution.

3 Discussion and Conclusion

First the historic data is obtained and analyzed. For financial market, the mechanism is complicated and fitting data with an existing distribution is hard, especially for high-level data which is driven by unknown forces. Direct mapping can be a choice for such chaos

data. However, for practicing, maximum entropy method is implemented to compare Beta distribution with Gamma distribution. For propagating uncertainty from input to realizations, various approaches are used. Monte Carlo approach shows a similar distribution between X and Y. Then polynomial chaos expansion (PCE) method is implemented with Jacobi polynomials. Although any orthogonal basis can be projected, Jacobi polynomials are optimal because it corresponds to the Beta distribution, i.e., the weight function of inner product is equivalent to the probability density distribution despite constant multipliers. There is rationality hidden, such as representation and completeness, accuracy, convergence mode and convergence rate and optimality in a finite dimensional setting. Stochastic collation method is discussed as well, which shares similar fundamentals with polynomial chaos expansion in 1D sense.

Acknowledgements

Thanks to everyone who helped us so much. Special thanks to Johann Guilleminot, Ph.D., the instructor of Uncertainty Quantification, who carefully reviewed my proposal and provided valuable suggestions. Thanks Hao Zhang and Rixi Peng, who helped me solve encountered problems with insightful ideas. Thanks all friends for making Duke life fantastic.

References

[1] *The Black-Scholes Model IEOR E4706: Foundations of Financial Engineering c 2016 by Martin Haugh*

[2] Wikipedia, *Beta distribution,* https://en.wikipedia.org/wiki/Beta_distribution.

[3] Wikipedia, *Gamma distribution,* https://en.wikipedia.org/wiki/Gamma_distribution.

[4] Wikipedia, *Jacobi polynomials*
https://en.wikipedia.org/wiki/Jacobi_polynomials.

[5] Wikipedia, *Gamma function*
https://en.wikipedia.org/wiki/Gamma_function.

Attachment

The Python codes are attached below. The code includes two files, the first pdf includes data retrieval and maximum entropy analysis. The second pdf includes polynomial chaos expansion and stochastic collocation methods.

In [21]:
```
from scipy.stats import beta
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
from sympy import *
from sympy.stats import Normal, cdf, density
import numpy as np
import pandas as pd
import math
import statsmodels
import statsmodels.api as sm
from statsmodels.tsa.stattools import coint, adfuller
from statsmodels import regression,stats
import matplotlib.pyplot as plt
pd.set_option('display.expand_frame_repr', False)
pd.set_option("display.precision", 4)
```

In [2]:
```
def getoption(symbol, start_time, end_time, option_type, resolution):
    qb = QuantBook()
    contract = qb.AddOption(symbol)
    contract.SetFilter(-1, +1, timedelta(days=0), timedelta(days=60))
    option = qb.GetOptionHistory(contract.Symbol, start_time, end_time)
    option_history = option.GetAllData()
    if len(option_history) == 0:
        return [None, None]
    stock_history = option_history
    option_history.reset_index(inplace=True)
    stock_data = stock_history[stock_history["expiry"].astype(str)==
"NaT"][::resolution]
    expiry = option.GetExpiryDates()[-1]
    option_history = option_history[option_history['expiry'] == expiry]
    option_data = option_history[option_history['type']==option_type]
[::resolution]
    option_data =
option_data[option_data["strike"]==option_data["strike"].median().round()]

    return [option_data, stock_data]
```

In [3]:
```
def call_iv (r,C,K,S,dT):
    iv = Symbol('\sigma')
```

```
    d1 = (log (S/K)+(r+iv*iv/2)*dT)/iv/sqrt(dT)
    d2 = d1 - iv * sqrt(dT)
    x = Symbol ('x')
    X = Normal ('x', 0, 1)
    expr = S * simplify(cdf(X))(d1) - exp(-r * dT) * K *
simplify(cdf(X))(d2) - C
    func_np = lambdify(iv, expr, modules=['numpy'])
    max_iter=10
    curr_iter=0
    start_value=10
    check = False
    while check==False and curr_iter<max_iter:
        ans=fsolve(func_np,start_value)[0]
        check = np.isclose(func_np(ans),0)
        start_value=start_value/2
        curr_iter=curr_iter+1
    if check == False:
        print ("warning, not converge")
    return [ans,check]
```

In [4]:

```
def call_greeks (r,iv,K,S,dT):
    r_sym, iv_sym, K_sym, S_sym, dT_sym = symbols ('r \sigma K S dT')
    x_sym = Symbol ('x')
    X = Normal ('x', 0, 1)
    d1 = (log (S_sym/K_sym)+
(r_sym+iv_sym*iv_sym/2)*dT_sym)/iv_sym/sqrt(dT_sym)
    d2 = d1 - iv_sym * sqrt(dT_sym)
    C = S_sym * simplify(cdf(X))(d1) - exp(-r_sym * dT_sym) * K_sym *
simplify(cdf(X))(d2)
    delta = diff(C,S_sym).evalf(subs={r_sym:r, iv_sym:iv, K_sym:K,
S_sym:S, dT_sym:dT})
    gamma = diff(diff(C,S_sym),S_sym).evalf(subs={r_sym:r, iv_sym:iv,
K_sym:K, S_sym:S, dT_sym:dT})
    vega = iv/100*diff(C,iv_sym).evalf(subs={r_sym:r, iv_sym:iv,
K_sym:K, S_sym:S, dT_sym:dT})
    theta = -1/365*diff(C,dT_sym).evalf(subs={r_sym:r, iv_sym:iv,
K_sym:K, S_sym:S, dT_sym:dT})
    rho = r/100*diff(C,r_sym).evalf(subs={r_sym:r, iv_sym:iv, K_sym:K,
S_sym:S, dT_sym:dT})
    return np.array([delta, gamma, vega, theta, rho]).astype('float64')
```

In [5]:

```
start_time =  datetime(2021, 3, 1, 0, 0)
call_history = None
```

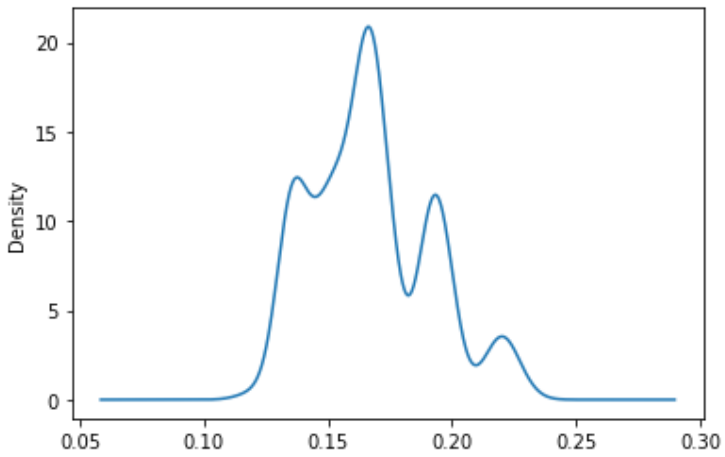
```
stock_history = None
resolution = 10
option_type = "Call"
symbol = "SPY"
for i in range(30):
    start_time = start_time + timedelta(days = i)
    end_time = start_time + timedelta(days = i+1)
    data_current = getoption(symbol, start_time, end_time, option_type,
resolution)
    call_history = pd.concat([call_history, data_current[0]])
    stock_history = pd.concat([stock_history, data_current[1]])
call_history.reset_index(drop=True, inplace=True)
stock_history.reset_index(drop=True, inplace=True)
```

In [6]:

```
option_history = call_history
option_greeks = option_history[{'expiry','strike','type','time'}][:-1]
for newcolumn_name in ['imp_vol','delta','gamma','vega','theta','rho']:
    option_greeks[newcolumn_name]=np.nan
r = 0.0063
for i in range (len(option_history)-1):
#for i in range (10):
    Price=0.5*(option_history['askclose'][i]+option_history['bidclose']
[i])
    K=option_history['strike'][i]
    S=0.5*
(stock_history['askclose'].values[i]+stock_history['bidclose'].values[i])

    time_delta=option_history['expiry'][i]-option_history['time'][i]
    dT=(time_delta.seconds/3600/24+time_delta.days)/365 ##time measured
in years
    opt_type=option_history['type'][i]
    if opt_type=='Call':
        iv=call_iv (r,Price,K,S,dT)[0]
        #print(call_iv (r,Price,K,S,dT)[1])
        greeks = call_greeks(r,iv,K,S,dT)
    if opt_type=='Put':
        iv=put_iv (r,Price,K,S,dT)[0]
        greeks = put_greeks(r,iv,K,S,dT)
    option_greeks.loc[i,'imp_vol']=iv
    option_greeks.loc[i,'delta':'rho']=greeks
option_greeks =
option_greeks.set_index(option_greeks["time"]).drop(columns = "time")
```

In [7]:
```
option_greeks["imp_vol"].plot.kde()
```

Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x7faa444e2e80>



In [8]:
```
len(option_greeks)
```

Out[8]: 1208

In [118…
```
data = option_greeks["imp_vol"].values
data_trans = (data - data.min() )/ (data.max() - data.min())
x = np.arange (0, 1, 0.02)
```

In [119…
```
data_P = [sum (data_trans < i) / len(data_trans) for i in x]
```

In [120…
```
data_pdf = np.zeros (len(x))
data_pdf [0] = 0
for i in range (len(data_P)-1):
    data_pdf[i+1] = (data_P [i+1] - data_P [i] )/ (x[i+1] - x[i])
    np.outer (data_pdf)
```

Entropy Calculation

In [193…
```
def cal_entropy (pdf1, pdf2, gridsize):
    pdf_avg1 = (pdf1 [1:] + pdf1[:-1]) / 2
    pdf_avg2 = (pdf2 [1:] + pdf2[:-1]) /2
    pdf12 = np.outer (pdf_avg1, pdf_avg2)
    log_pdf12 = np.outer (pdf_avg1, pdf_avg2)
    return -(pdf12 * log_pdf12).sum() * gridsize**2
```

Beta Distribution

In [194…
```
beta_a = (exp * (1 - exp) / var - 1) * exp
```

```
beta_b = (exp * (1 - exp) /var -1) * (1 - exp)
B = math.gamma(beta_a) * math.gamma(beta_b) / math.gamma(beta_a+beta_b)
beta_pdf = (x**(beta_a-1) * (1-x)**(beta_b-1)) / B
```

In [195…

```
beta_entropy = cal_entropy (data_pdf, beta_pdf, x[1]-x[0])
print ("Beta Distribution Entropy is " + str(beta_entropy))
```

```
Beta Distribution Entropy is -2.575559882761846
```

Gamma Distribution

In [196…

```
gamma_a = exp**2 / var
gamma_b = exp / var
gamma_pdf = gamma_b ** gamma_a / math.gamma (gamma_a) * x ** (gamma_a -1
) * np.exp(-gamma_b * x)
```

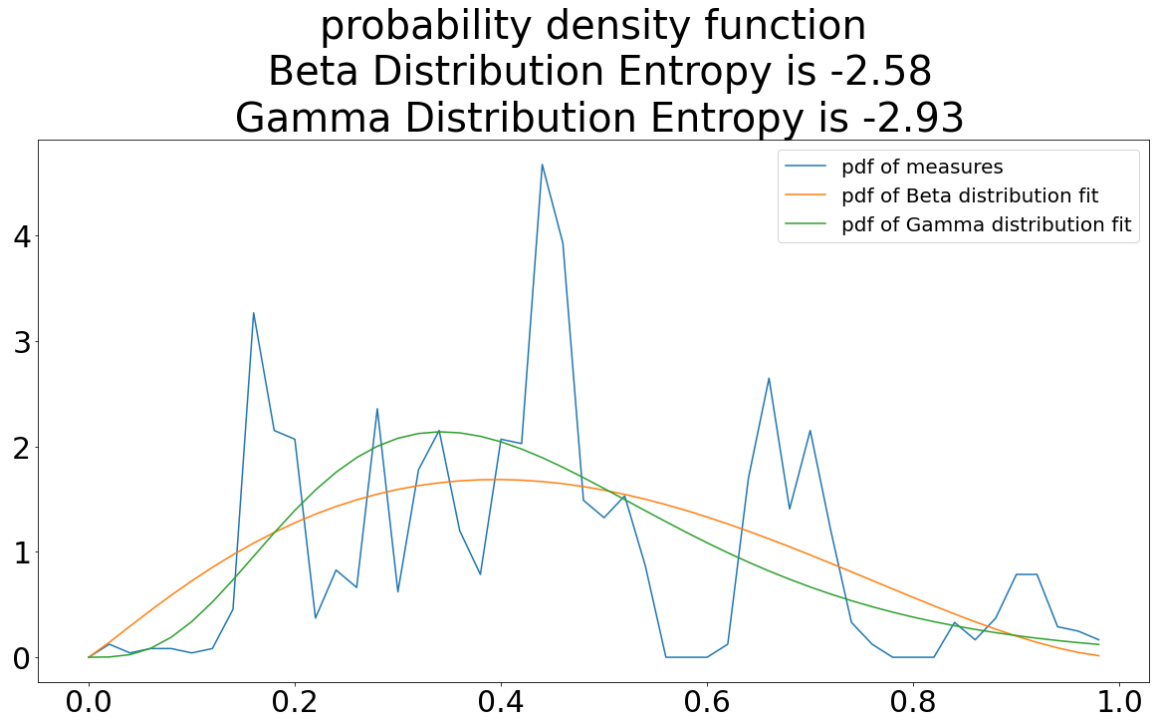
In [197…

```
gamma_entropy = cal_entropy (data_pdf, gamma_pdf, x[1]-x[0])
print ("Gamma Distribution Entropy is " + str(gamma_entropy))
```

```
Gamma Distribution Entropy is -2.930284111030331
```

In [198…

```
plt.figure(figsize=(20,10))
plt.plot(x, data_pdf, label = "pdf of measures")
plt.plot(x, beta_pdf, label = "pdf of Beta distribution fit")
plt.plot(x, gamma_pdf, label = "pdf of Gamma distribution fit")
plt.title("probability density function" + "\n Beta Distribution Entropy
is "+ str(round(beta_entropy,2))
          + "\n Gamma Distribution Entropy is "+
str(round(gamma_entropy,2)), fontsize=40)
plt.xticks(fontsize=30)
plt.yticks(fontsize=30)
plt.legend(fontsize=20)
plt.show()
```



In [199…

```
print ("Gamma Distribution alpha = " + str (gamma_a))
print ("Gamma Distribution beta = " + str (gamma_b))
print ("Beta Distribution alpha = " + str (beta_a))
print ("Beta Distribution beta = " + str (beta_b))
```

```
Gamma Distribution alpha = 4.53005940022824
Gamma Distribution beta = 10.319263276474011
```

In [ ]:

Uncertainty Quantification Final Project

April 28, 2021

```
In [577]: import random
          import numpy as np
          import pandas as pd
          import matplotlib.pyplot as plt
          import seaborn as sns
          from sympy import *
          from sympy.stats import Normal, cdf
          from matplotlib.ticker import MaxNLocator
          import scipy.stats
          pd.set_option('display.expand_frame_repr', False)
```

0.1 Monte Carlo Solver

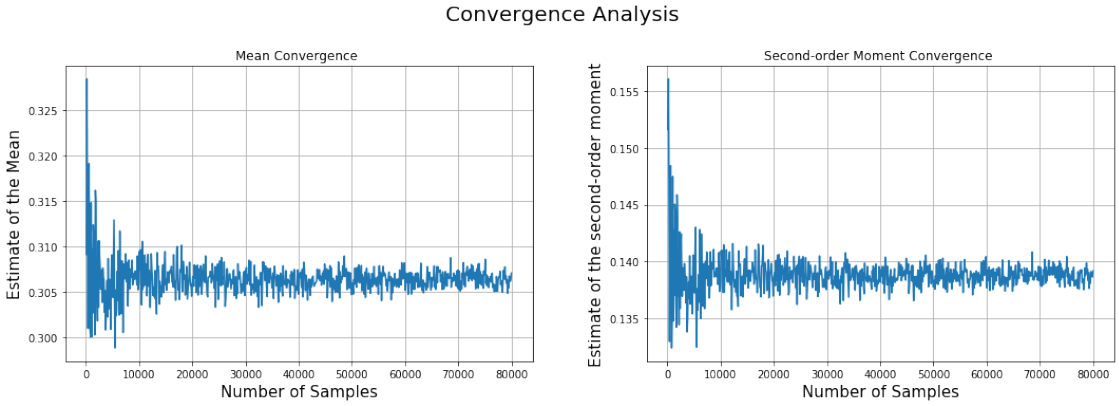
```
In [433]: alpha = 1.1444307615618827
          beta = 2.590929757651513
```

```
In [19]: def getbetarandomvars (alpha, beta, number):
             random.seed()
             ans = []
             for i in range(number):
                 ans.append(random.betavariate(alpha, beta))
             return ans
```

```
In [55]: CA = pd.DataFrame(columns={"number", "mean", "second order moment"})
         nums = np.linspace (100, 80000, 800).astype(int)
         for ind in range (len(nums)) :
             num = nums[ind]
             rvs = np.array( getbetarandomvars (alpha, beta, num))
             CA.loc [ind] = [num, rvs.mean(), (rvs*rvs).mean()]
```

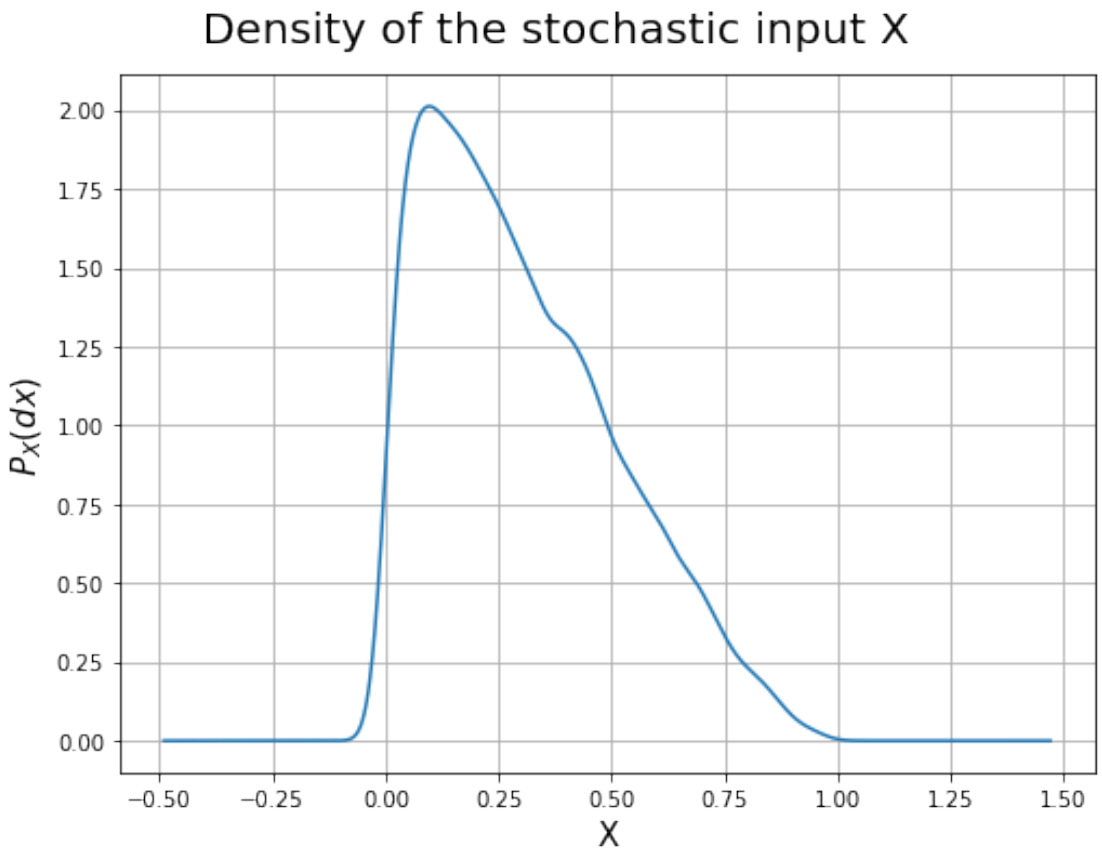
```
In [117]: fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(16, 6));

          ax1.plot(CA['number'].values, CA['mean'].values);
          ax1.set_title('Mean Convergence ')
          ax1.set_xlabel('Number of Samples',fontsize=15)
          ax1.set_ylabel('Estimate of the Mean',fontsize=15)
          ax1.grid(True)
```

```
ax2.plot(CA['number'].values, CA['second order moment'].values);
ax2.set_title('Second-order Moment Convergence ')
ax2.set_xlabel('Number of Samples',fontsize=15)
ax2.set_ylabel('Estimate of the second-order moment',fontsize=15)
ax2.grid(True)

fig.suptitle('Convergence Analysis',fontsize=20)
fig.tight_layout(pad=5.0)
fig.savefig(r"C:\DUKE\courses\Uncertainty Quantification\final project\convergence a
```

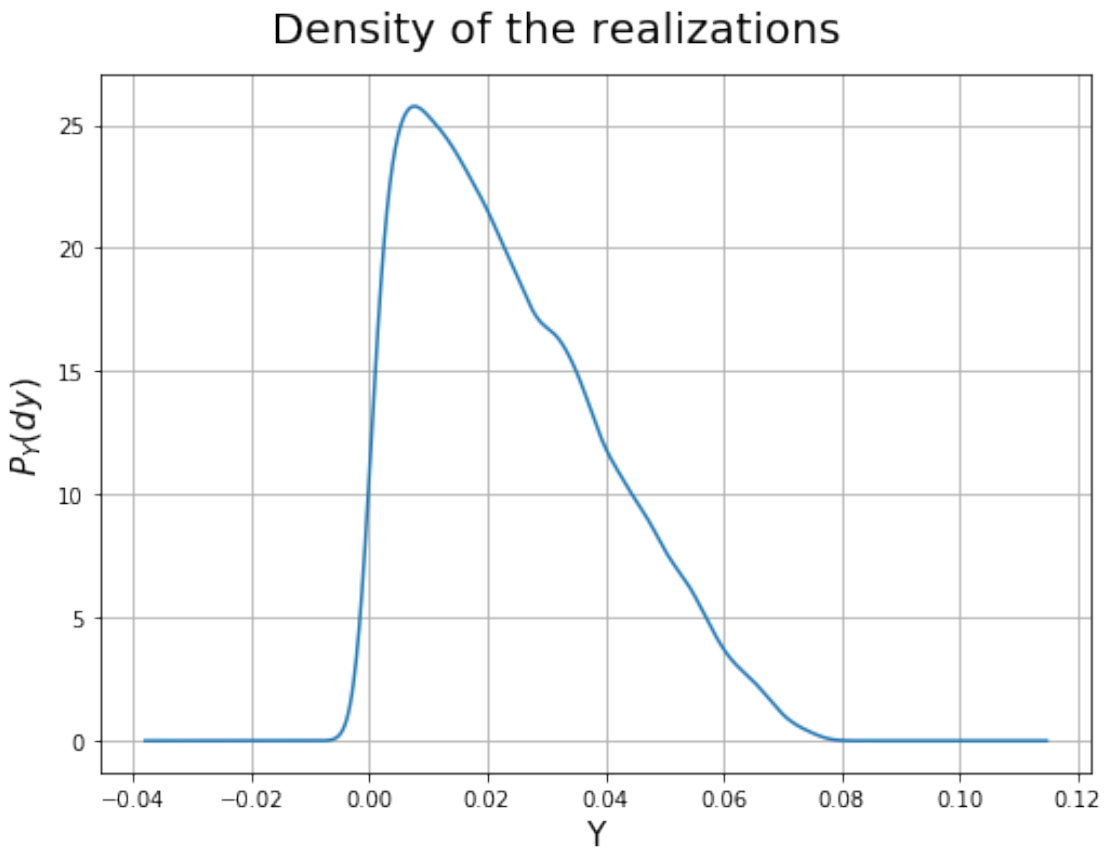



```
In [221]: S = 1
          K = 1
          r = 0.0063
          dT = 14/365
          iv = Symbol('\sigma')
          d1 = (log (S/K)+(r+iv*iv/2)*dT)/iv/sqrt(dT)
          d2 = d1 - iv * sqrt(dT)
          N = Symbol ('n')
          N = Normal ('n', 0, 1)
          C = S * simplify(cdf(N))(d1) - exp(-r * dT) * K * simplify(cdf(N))(d2)

In [92]: number = 20000
         X = np.array(getbetarandomvars (alpha, beta, number))

In [122]: number = 20000
          X = np.array(getbetarandomvars (alpha, beta, number))
          MC = pd.DataFrame(columns={"ramdom variable", "realization"})
          for ind in range (number) :
              rv = X[ind]
              realization = float(C.subs(iv,rv).evalf())
              MC.loc [ind] = [rv, realization]

In [325]: fig, ax = plt.subplots(nrows=1, ncols=1,figsize=(8, 6));
          ax = MC['ramdom variable'].plot.kde()
```

```
ax.set_xlabel('X',fontsize=15)
ax.set_ylabel('$P_X(dx)$',fontsize=15)
ax.grid(True)
fig.suptitle('Density of the stochastic input X',fontsize=20)
fig.tight_layout(pad=3.5)
fig.savefig(r"C:\DUKE\courses\Uncertainty Quantification\final project\MC solver X p
```



In [324]:
```
fig, ax = plt.subplots(nrows=1, ncols=1,figsize=(8, 6));
ax = MC['realization'].plot.kde()
ax.set_xlabel('Y',fontsize=15)
ax.set_ylabel('$P_Y(dy)$',fontsize=15)
ax.grid(True)
fig.suptitle('Density of the realizations',fontsize=20)
fig.tight_layout(pad=3.5)
fig.savefig(r"C:\DUKE\courses\Uncertainty Quantification\final project\MC solver Y p
```



0.2 Stochastic Modeling Through Series Expansions

Jacobi Polynomials

```
In [687]: alpha = 1.1444307615618827
          beta = 2.590929757651513
          i = Symbol('i')
          x = Symbol('x')
          m = Symbol('m')
          J = gamma(alpha+i)/factorial(i)/gamma(alpha+beta+i-1)*Sum(binomial(i,m)*gamma(alpha+b

In [606]: # weight function
          w = (1-x)**(alpha-1) * (1+x)**(beta-1)
          # normalization factor
          n = sqrt(gamma(i+alpha)*gamma(i+beta)/(2*i+alpha+beta-1)/gamma(i+alpha+beta-1)/facto
          # normalization factor for Jacobi polynomial inner product
          nJ = sqrt(2**(alpha+beta-1)/(2*i+alpha+beta-1)*gamma(i+alpha)*gamma(i+beta)/gamma(i+a
```

Check Jacobi polynomials properties

```
In [689]: check = pd.DataFrame(np.zeros((5,5)))
          for ii in range(5):
              for jj in range(5):
                  ans=0
                  for xx in np.linspace(-1,1,2001):
                      ans+=((J/nJ).subs(i,ii)*(J/nJ).subs(i,jj)*w).subs(x,xx).evalf()*0.001
                  check.iloc[ii,jj]=ans
          print(check)
```

```
                         0                       1                       2                       3
0      0.999789419002016  -0.000304472239597669  -0.000383338867566917  -0.000453670196368348
1  -0.000304472239597669      0.999559769606514  -0.000554252913307230  -0.000655958381653051
2  -0.000383338867566917  -0.000554252913307230      0.999302170885729  -0.000825841774924408
3  -0.000453670196368348  -0.000655958381653051  -0.000825841774924408      0.999022591974159
4  -0.000518151339530541  -0.000749164531746407  -0.000943248442348874   -0.00111624878145781
```

Check the normalization

```
In [690]: check = pd.DataFrame(np.zeros((5,5)))
          for ii in range(5):
              for jj in range(5):
                  ans=0
                  for xx in np.linspace(-1,1,2001):
                      ans+=((J/n).subs(i,ii)*(J/n).subs(i,jj)*0.5*P.subs(x,(1-x)/2)).subs(x,xx
                  check.iloc[ii,jj]=ans
          print(check)
```

```
                         0                       1                       2                       3
0      0.999789419002016  -0.000304472239597765  -0.000383338867566863  -0.000453670196368365
1  -0.000304472239597765      0.999559769606514  -0.000554252913307099  -0.000655958381653016
2  -0.000383338867566863  -0.000554252913307099      0.999302170885729  -0.000825841774924314
3  -0.000453670196368365  -0.000655958381653016  -0.000825841774924314      0.999022591974159
4  -0.000518151339530545  -0.000749164531746435  -0.000943248442348878   -0.00111624878145791
```

h(x)

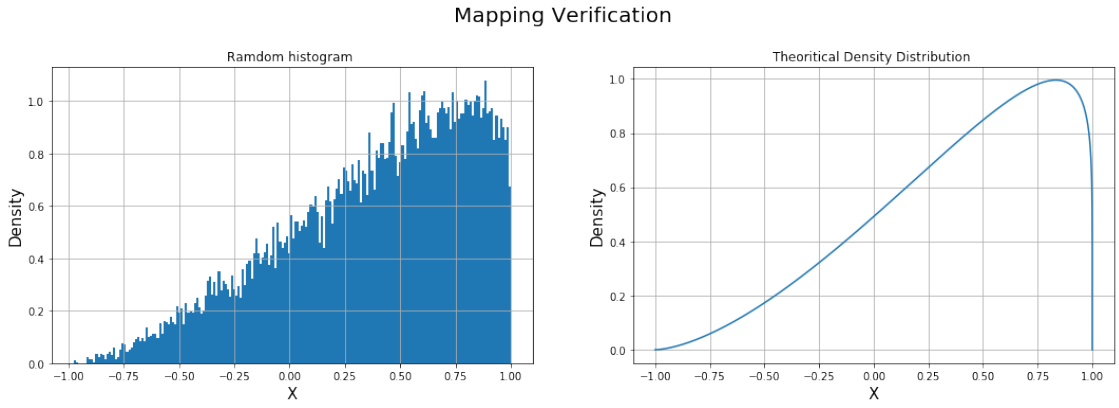
```
In [522]: S = 1
          K = 1
          r = 0.0063
          dT = 14/365
          x = Symbol('x')
          d1 = (log (S/K)+(r+x*x/2)*dT)/x/sqrt(dT)
          d2 = d1 - x * sqrt(dT)
          N = Symbol ('n')
          N = Normal ('n', 0, 1)
          h = S * simplify(cdf(N))(d1) - exp(-r * dT) * K * simplify(cdf(N))(d2)
```

P_X(dx) for Beta distribution

```
In [608]: P = x**(alpha-1) * (1-x)**(beta-1)/gamma(alpha)/gamma(beta)*gamma(alpha+beta)
```

Check mapping

```
In [646]: X = np.array(getbetarandomvars (alpha, beta, 20000))
          X = 1-X*2
          fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(16, 6));
          pdf_mapped = [0.5*P.subs(x,(1-x)/2).subs(x,xx) for xx in np.linspace(-1,1,20000)]

          ax1.hist(X, 200, density=1);
          ax1.set_title('Ramdom histogram ')
          ax1.set_xlabel('X',fontsize=15)
          ax1.set_ylabel('Density',fontsize=15)
          ax1.grid(True)

          ax2.plot(np.linspace(-1,1,20000), pdf_mapped);
          ax2.set_title('Theoritical Density Distribution ')
          ax2.set_xlabel('X',fontsize=15)
          ax2.set_ylabel('Density',fontsize=15)
          ax2.grid(True)

          fig.suptitle('Mapping Verification',fontsize=20)
          fig.tight_layout(pad=5.0)
          fig.savefig(r"C:\DUKE\courses\Uncertainty Quantification\final project\mapping verifi
```




```
In [627]: X = np.array(getbetarandomvars (alpha, beta, 2000))
          X = 1-2*X
```

Then integrate from 0 to 1 to find yi.

```
In [635]: yi = []
          for ii in range(5):
```

```
            ans=0
            for xx in np.linspace(-1,1,2001):
                ans+=(h.subs(x,(1-x)/2)*(J/n).subs(i,ii)*0.5*P.subs(x,(1-x)/2)).subs(x,xx).ev
            yi.append(ans)
```

```
In [629]: yi
```

```
Out[629]: [0.0542393444813036,
           -0.115640573811862,
           0.142271972338891,
           -0.145694613572554,
           0.143758197099776]
```

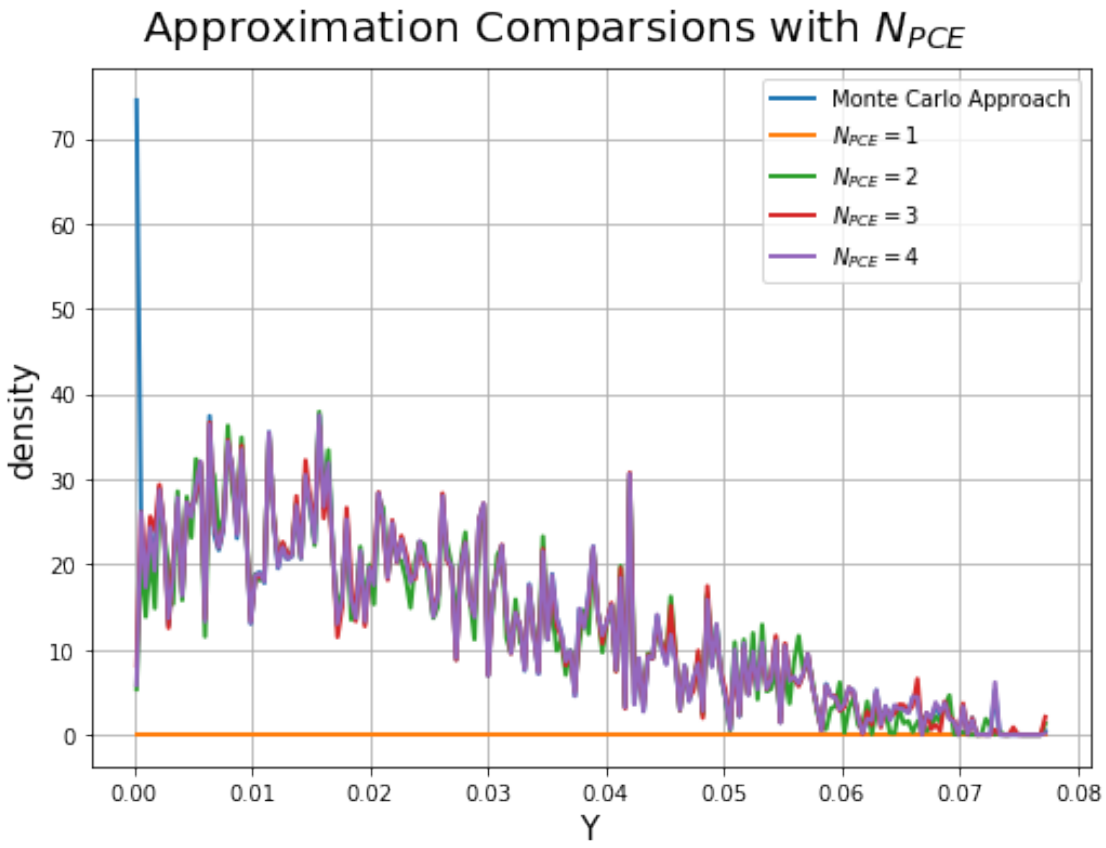
```
In [653]: Poly0 = yi[0]*(J/n).subs(i,0)
          Poly1 = yi[0]*(J/n).subs(i,0)+yi[1]*(J/n).subs(i,1)
          Poly2 = yi[0]*(J/n).subs(i,0)+yi[1]*(J/n).subs(i,1)+yi[2]*(J/n).subs(i,2)
          Poly3 = yi[0]*(J/n).subs(i,0)+yi[1]*(J/n).subs(i,1)+yi[2]*(J/n).subs(i,2)+yi[3]*(J/n)
          Poly4 = yi[0]*(J/n).subs(i,0)+yi[1]*(J/n).subs(i,1)+yi[2]*(J/n).subs(i,2)+yi[3]*(J/n)
          Poly = [Poly0, Poly1, Poly2, Poly3, Poly4]
```

```
In [654]: Y_approx = [np.array([Poly[i].subs(x,xx).evalf() for xx in X]).astype(float) for ii
```

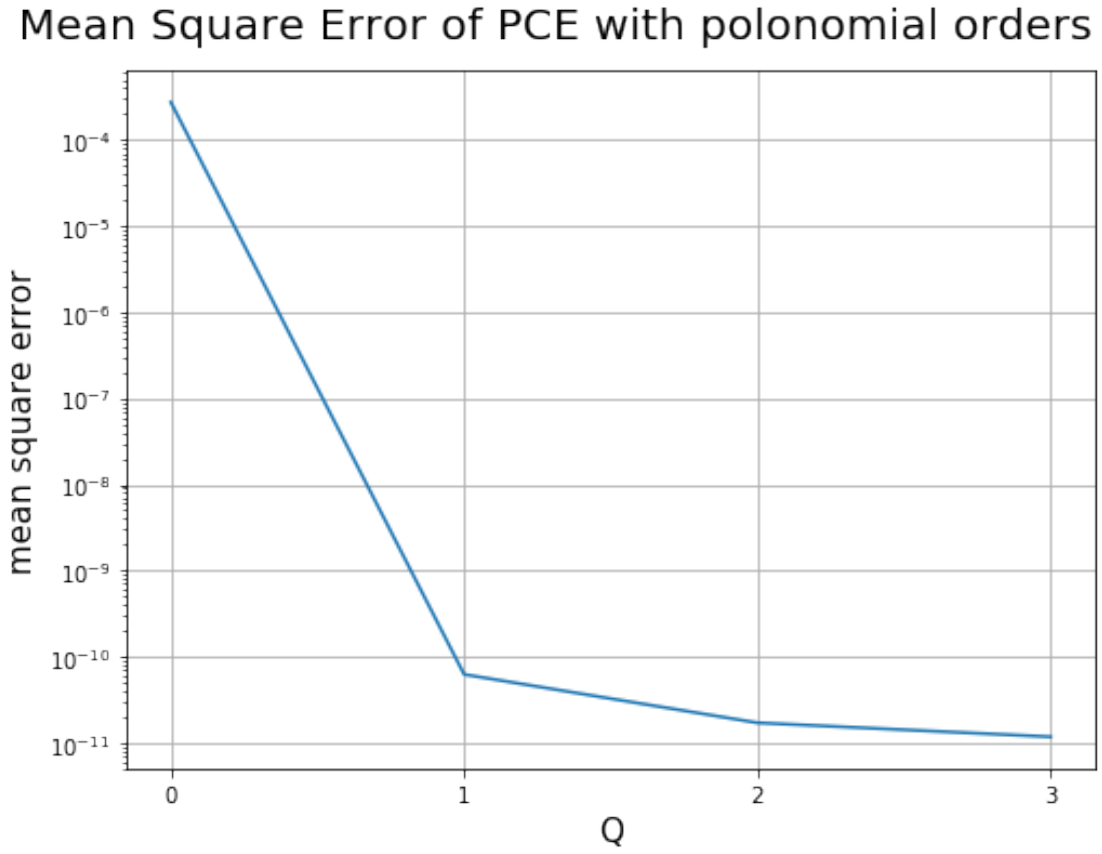
```
In [655]: Y_real = np.array([h.subs(x,(1-xx)/2) for xx in X]).astype(float)
```

```
In [667]: kde_real = scipy.stats.gaussian_kde(Y_real,bw_method=0.0005)
          kde_approx = [scipy.stats.gaussian_kde(Y_approx[i],bw_method=0.0005) for  ii in range
          Y_range = np.linspace(Y_real.min(),Y_real.max(),200)
```

```
In [670]: fig, ax = plt.subplots(nrows=1, ncols=1,figsize=(8, 6));
          ax.plot(t_range,kde_real(t_range),lw=2, label='Monte Carlo Approach')
          for ii in range(4):
              ax.plot(t_range,kde_approx[ii](t_range),lw=2, label='$N_{PCE}=$'+str(ii+1))
          ax.set_xlabel('Y',fontsize=15)
          ax.set_ylabel('density',fontsize=15)
          ax.grid(True)
          ax.legend()
          fig.suptitle('Approximation Comparsions with $N_{PCE}$',fontsize=20)
          fig.tight_layout(pad=3.5)
          fig.savefig(r"C:\DUKE\courses\Uncertainty Quantification\final project\N_PCE compars
```




```
In [671]: mse=[]
          for ii in range(4):
              mse.append(((Y_approx[i]-Y_real)**2).mean())

In [675]: fig, ax = plt.subplots(nrows=1, ncols=1,figsize=(8, 6));
          ax.plot(mse)
          ax.set_xlabel('Q',fontsize=15)
          ax.set_ylabel('mean square error',fontsize=15)
          ax.set_yscale('log')
          ax.xaxis.set_major_locator(MaxNLocator(integer=True))
          ax.grid(True)
          fig.suptitle('Mean Square Error of PCE with polonomial orders',fontsize=20)
          fig.tight_layout(pad=3.5)
          fig.savefig(r"C:\DUKE\courses\Uncertainty Quantification\final project\PCE mse.jpg",
```


Mean Square Error of PCE with polonomial orders
mean square error
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-10}
10^{-11}
0
1
2
3
Q