Assignment 1: IRS Engine, Mean Reversion

Aytek Mutlu - 2648232

16 November 2019

1 Introduction

This assignment will cover two separate parts. Firstly, construction of an IRS Engine is discussed and present values and par values of some interest rate swaps will be provided as well as their sensitivities to curve shifts.

In the second part, binomial trees is builtand Monte Carlo simulations is run for Mean Reversion process. Then variety of option prices are calculated and compared.

2 IRS Engine

2.1 Development

IRS Engine is generated using programming language Python by using classes and their properties. The code consists of two classes, namely Curve class and IRS class:

2.1.1 Curve Class

Curve class is initiated with yield curve data (Section 2.2) and its corresponding compounding frequency. At its initiation, curve object prepares yield curve, zero curve, forward curve and discount factors with methodologies explained in Section 2.3. Moreover, class has methods that makes it possible to convert any given curve from discrete compounding to continuous (vice versa), interpolate any given date or dates and apply a curve shift with given basis points. Code is provided in Appendix 6.1 for further observation.

2.1.2 IRS Class

IRS class is initiated with main descriptives of an interest rate swap. Those are forward curve to be used for the reset rates, discount curve to be used, initial notional amount, valuation date, swap start and end dates, amortisation type and schedule, if applicable. Then, its initiation triggers generation of swap schedule by finding accrual start and end dates for each period, its corresponding outstanding notional amount (takes into account amortisation type of Constant, Linear or custom), corresponding reset rates, zero rates and discount factors (with the help of Interpolation method of Curve class (Section 2.1.1)). All of the calculation methodology is discussed in Section 2.3. Yet, it is important to note that day count convention is assumed to be 30/360 for the sake of simplicity. It is possible to use actual date schedules with additional Python packages which takes holidays into account, however this is out of the scope of this assignment.

Class also offers two methods. One calculates present value of the interest rate swap given initial fixed rate and the other calculates the par rate (current fixed rate that makes present value zero). Again, the code is provided in Appendix 6.1 for further observation.

2.2 Input Curves

USD and EUR swap and OIS curves are selected for this assignment due to their liquidity and availability. Yield curves are extracted from Bloomberg terminal on 04/11/2019. Swap curves are constructed by using most liquid products for each tenor. For both USD and EUR swap yield curves, cash rates are used up to 3 months tenor, then LIBOR rates are used for 3 months and 6 months (they are most commonly used on those tenors). Then, rates are derived from futures for tenors between 6 months and 12 months where they are more liquid then LIBOR rates. Finally, swap rates are used for the longer maturities. OIS curves are constructed with a same approach where main underlying is OIS swaps in the market and LIBOR-OIS spreads. Here is the yield curves for USD and EUR that are used in further calculations in this assignment (Figure 1):

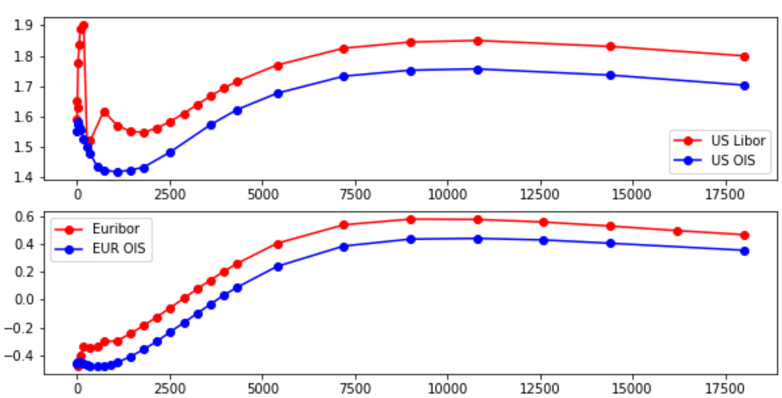

Figure 1: Swap and OIS Curves of US and EUR

2.3 Calculation Methodology

2.3.1 Yield Curve

Curves provided are all quarterly compounded curves with some given tenors. However, in order to have a complete tenor set, all swap curves are initially extended with cubic spline interpolation in order to fill in each quarterly tenor up to the maximum tenor available. This operation is handled by **Interpolate3M()** method of Curve class. Then, all curve is converted to continuously compounded rates with **Discrete2Continuous()** method of same class with the following formula:

$$r_{cont} = log((1 + r_{discrete} * \frac{compoundfreq}{360})^{\frac{360}{compoundfreq}})$$

where $r_{discrete}$ is the discretely compounded yield curve rate and $compoundfreq$ is the compounding frequency.

2.3.2 Zero Curve

Zero curve is constructed with bootstrapping methodology. For all tenors up to the compounding frequency, zero curve is assumed to be same as yield curve since there are no interim coupon payments on those tenors but only the terminal payment. For the rest of the tenors, zero rate is calculated for any tenor by calculating discounted quarterly payments that would occur with the yield curve rate of that given tenor. Discounting are performed with zero rates calculated up until that point. Then, the present value of the terminal payment is then found with this method. Indeed, this payment would occur at the tenor for which zero rate is aimed to be found. Therefore, correct zero rate that would discount the payment to the mentioned present value is solved and assigned to be the zero rate of that tenor.

2.3.3 Forward Curve

Forward curve is constructed from zero curve by using each quarterly consecutive tenor. Simply, forward rates are the rates that corresponds to the period t to period $t + 90$. Here is the formula:

$$r_{t,t+90} = log(\frac{exp(r_{t+90} * \frac{t+90}{360})}{exp(r_t * \frac{t}{360})}) * \frac{360}{90}$$

2.3.4 Discount Factor

Discount factors are simply calculated with the following formula from the zero rates:

$$discountfactor = exp(-r_{zero} * \frac{tenor}{360})$$

2.3.5 IRS Schedule

For generation of IRS schedule, accrual start and end dates are initially obtained. The swaps are assumed to make and receive payments quarterly and therefore there occurs a payment at each quarter starting from the start date up to the end date. If the dates are broken, it is assumed that swap has short coupon last, meaning that the latest period will be shorter than 90 days.

Upon having date schedules, IRS class finds outstanding notional for each period. if the IRS class is initiated with amortisation type=**'Constant'**, then the initial notional is used as the outstanding notional at each period. if amortisation type=**'Linear'** is observed, then initial notional is amortised linearly so that it gradually reduces to 0 at the end of the swap. Finally, it is also possible to initiate the swap with a custom payment schedule (amortisation type=**'Custom'**). In this case, IRS class tries to match the provided schedule with the date schedule. If the length of those match, then this payment plan is used. If not, class automatically uses a linear amortisation schedule.

2.3.6 IRS Valuation

IRS valuation method uses the date and notional schedule prepared (Section 2.3.5). Firstly, it calculates number of days left for the payment for each payment period and then obtains reset rate (forward rate), zero rate and discount factor of each payment period with the help of **Interpolate()** method of Curve class. This interpolation is again a cubic spline interpolation. With the reset rates in hand, present value of floating payment of each payment period is calculated by:

$$PV_{floatpayment} = \\ OutstandingNotional_t * (exp(r_{reset} * t_{period}/360) - 1) * DiscountFactor_t$$

where t_{period} is the number of days within that number of days.

In the cases when an existing IRS is valued, **CalculateValue()** method of IRS class takes the fixed rate of the swap as input and calculates the present values of the fixed payments, with the same methodology with floating payments. Finally, net present value of the swap is calculated as following:

$$NPV = \sum_t PV_{FloatPayment_t} - \sum_t FixedPayment_t$$

as this assignment assumes that each and every swap executed as fixed payer and float receiver.

Moreover, in the cases when fixed rate (par rate) for a new IRS needs to be found, then **CalculatePar()** method of IRS class solves the fixed rate that makes output of **CalculateValue()** zero, meaning that a fixed rate that provides an IRS with zero NPV. That fixed rate is returned by converting it back to discrete compounding (quarterly, for this assignment).

2.4 Results

2.4.1 Already Issued Swaps with Remaining Maturity of 1.5 years

Four interest rate swaps are valued (two USD and two EUR) with similar details. In order to observe the effect of amortisation, same swaps are valued with and without an amortisation schedule. Here are the details of the swaps (Table 1):

	Forward Curve	Discount Curve	Initial Notional	Valuation Date	Start Date	End Date	Amortisation	Fixed Rate	Value
USD IRS 1	Libor	USD OIS	10,000,000	04/11/2019	01/06/2019	01/06/2021	Constant	3%	$ -225,072
USD IRS 2	Libor	USD OIS	10,000,000	04/11/2019	01/06/2019	01/06/2021	Custom[1]	3%	$ -165,709
EUR IRS 1	Euribor	EUR OIS	10,000,000	04/11/2019	01/06/2019	01/06/2021	Constant	0.5%	€-128,310
EUR IRS 2	Euribor	EUR OIS	10,000,000	04/11/2019	01/06/2019	01/06/2021	Custom[2]	0.5%	€-94,169

Table 1: Swap Details for already issued swaps with remaining maturity of 1.5 years

2.4.2 Newly Issued Swaps with 5 years maturity

Again, four swaps are priced. This case, fixed rates are solved for one constant notional and one linear amortisation swap for both USD and EUR. Results are represented in Table 2:

	Forward Curve	Discount Curve	Initial Notional	Valuation Date	Start Date	End Date	Amortisation	Fixed Rate
USD IRS 3	Libor	USD OIS	10,000,000	04/11/2019	06/11/2019	06/11/2024	Constant	1.5335%
USD IRS 4	Libor	USD OIS	10,000,000	04/11/2019	06/11/2019	06/11/2024	Linear	1.5358%
EUR IRS 3	Euribor	EUR OIS	10,000,000	04/11/2019	06/11/2019	06/11/2024	Constant	-0.1541%
EUR IRS 4	Euribor	EUR OIS	10,000,000	04/11/2019	06/11/2019	06/11/2024	Linear	-0.2354%

Table 2: Swap Details for newly issued swaps with remaining maturity of 5 years

2.4.3 Already Issued Swaps with Remaining Maturity of 20 years

In this setup, two interest rate swaps are priced (one USD and one EUR). Then, the zero curves (Libor and Euribor) are shifted 100 basis points up and down and same swaps are re-valued. Results are in Table 3:

	Forward Curve	Discount Curve	Initial Notional	Valuation Date	Start Date	End Date	Amortisation	Fixed Rate	Value
USD IRS 5	Libor	USD OIS	10,000,000	04/11/2019	01/12/2018	01/12/2039	Constant	2.1%	$ -478,404
USD IRS 6	DownShiftedLibor	USD OIS	10,000,000	04/11/2019	01/12/2018	01/12/2039	Constant	2.1%	$ -495,884
USD IRS 7	UpShiftedLibor	USD OIS	10,000,000	04/11/2019	01/12/2018	01/12/2039	Constant	2.1%	$ -460,924
EUR IRS 5	Euribor	EUR OIS	10,000,000	04/11/2019	01/12/2018	01/12/2039	Constant	1%	€-878,294
EUR IRS 6	DownShiftedEuribor	EUR OIS	10,000,000	04/11/2019	01/12/2018	01/12/2039	Constant	1%	€-898,924
EUR IRS 7	UpShiftedEuribor	EUR OIS	10,000,000	04/11/2019	01/12/2018	01/12/2039	Constant	1%	€-857,643

Table 3: Swap Details for already issued swaps with remaining maturity of 20 years

3 Comments and Conclusion

[2] Custom schedule: [10,000,000; 9,375,000; 8,750,000; 8,125,000; 7,500,000; 6,875,000; 6,250,000; 5,625,000]

4 Mean Reversion

4.1 Development

4.1.1 Binomial Tree

For construction of a Binomial Tree with mean reversion process, paper of Bastian-Pinto, Brandão and Hahn (1) is used as guideline. Following formulation is used for the price of the underlying after i up movements and j down movements and for upward probability at each node:

$$x_{(i,j)} = \bar{x} * (1 - exp(-\eta * (i + j) * \Delta t)) + x^*$$

$$p_{x_{(i,j)}} = \tfrac{1}{2} * (1 + \tfrac{\eta * (-x^*) * \sqrt{\Delta t}}{\sqrt{\eta^2 * ((-x)^*)^2 * \Delta t + \sigma^2}})$$

where;

$$x^* = (i - j) * \sigma * \sqrt{\Delta t}$$

Note that $\bar{x}$ =long-term mean and η =mean reversion speed.

Given the formulas, a binomial tree is constructed for S&P-500 Index with the following parameters (4):

Spot Price	2978.4
σ **(annual)**	0.1424
Continuous Interest Rate	0.09875%
Option Maturity	63 days
Number of Steps	63
Reversion Speed	0.05
Long-term Mean	2978.4

Table 4: Parameters of Binomial Tree with mean reversion for S&P-500 Index

This part is implemented in Matlab and for the tree implementation, function illustrated in Annex 6.2.2 is used. Note that not S&P-500 Index itself, but is natural logarithm is assumed to be following the mean reversion process. As required in the assignment, binomial tree with same parameters in Table 4 is constructed with Geometric Brownian Motion process assumption. This implementation can be investigated in the function placed in Annex 6.2.6.

Upon achieving the binomial trees, European option pricer functions are also implemented so that variety of options can be easily priced. Annex 6.2.3 shows European call option pricer with mean reversion process where Annex 6.2.7 shows pricing of same option with Geometric Brownian Motion process. Moreover, Annex 6.2.4 indicates the function for European put option pricing with mean reversion and finally Annex 6.2.5 is a function that prices a European exotic option where the payoff is the square of the difference between the terminal stock value and the strike for a stock that is assumed to follow mean reversion process.

4.1.2 Monte-Carlo Simulation

Monte-Carlo simulation is applied to natural logarithm of S&P-500 Index for the discretized version of mean-reversion process as following:

$$X_{t+1} = X_t + \eta * (\bar{X} - X_t) * \Delta t + \sigma * (W_{t+1} - W_t)$$

where $X_t = log(S_t)$

Moreover, another Monte-Carlo simulation is run for Geometric Brownian Motion. All parameters are used same as Table 4 and number of simulations is set to be 100,000.

This implementation can be observed in Annex 6.2.1, the main code.

4.2 Results

4.2.1 ATM European Call Price Comparison

In this section, price of an ATM European call with 3000 strike is compared with 4 different methods. The results are in Table 5:

Model	**Call Price**
Mean Reversion Process with Binomial Tree	76.3367
Mean Reversion Process with Monte-Carlo Simulation	76.1722
GBM with Binomial Tree	77.8093
GBM with Monte-Carlo Simulation	78.0534

Table 5: European Call Price with 3000 Strike with 4 different approaches

4.2.2 ATM European Put and European Exotic Option Price Comparison

In this section, an ATM European Put with 3000 strike and a European Exotic option where the payoff is the square of the difference between terminal stock value and strike is priced with mean-reversion process both using binomial trees and Monte-Carlo simulation. The results for put option is depicted in Table 6 and for exotic option is shown in Figure 2 with variety of strikes from OTM to ITM:

Model	**Put Price**
Mean Reversion Process with Binomial Tree	90.2598
Mean Reversion Process with Monte-Carlo Simulation	91.2555

Table 6: European Put Price with 3000 Strike with 2 different approaches

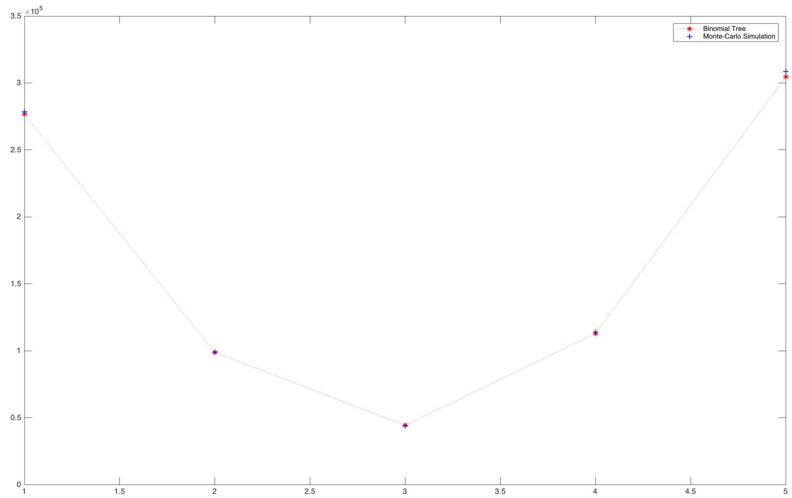

Figure 2: European Exotic Option Price Comparison for Binomial-Tree and Monte-Carlo simulation for mean-reversion process with variety of strikes

4.2.3 European Call Price Comparison with Different Moneyness Levels

This section compares how European call prices change with moneyness of the option. For this purpose, call option with 63 days of maturity is priced for 11 different strikes between 2500 and 3500 with mean reversion process and Geometric Brownian Motion. Although the differences are not obvious, prices are plotted in Figure 3 in order to see the change in the option price. Then, Table 7 includes the call prices for all 4 approach for each strike and the percentage differences.

Models / Strikes	**2500**	**2600**	**2700**	**2800**	**2900**	**3000**	**3100**	**3200**	**3300**	**3400**	**3500**
MR with Binomial Tree	478.9836	381.9004	288.3637	202.8651	130.6934	76.3367	40.1427	18.8196	7.7232	2.8156	0.9451
MR with Monte-Carlo Simulation	483.7674	385.5017	290.8453	204.2912	131.2944	76.1450	39.3841	18.1097	7.4207	2.6889	0.8558
GBM with Binomial Tree	485.0083	386.7547	292.1510	205.7343	132.7973	77.8093	41.1019	19.3863	8.0192	2.9514	1.0010
GBM with Monte-Carlo Simulation	484.6549	386.4614	291.9875	205.6043	132.8089	77.7259	40.8605	19.1863	8.0494	2.9926	0.9935
% Difference in Binomial Tree	1.2%	1.3%	1.3%	1.4%	1.6%	1.9%	2.3%	2.9%	3.7%	4.6%	5.6%
% Difference in Monte-Carlo Simulation	0.2%	0.2%	0.4%	0.6%	1.1%	2.0%	3.6%	5.6%	7.8%	10.1%	13.9%

Table 7: Comparison of European Call Prices with different models and moneyness levels

Finally, this difference is depicted in Figure 4:

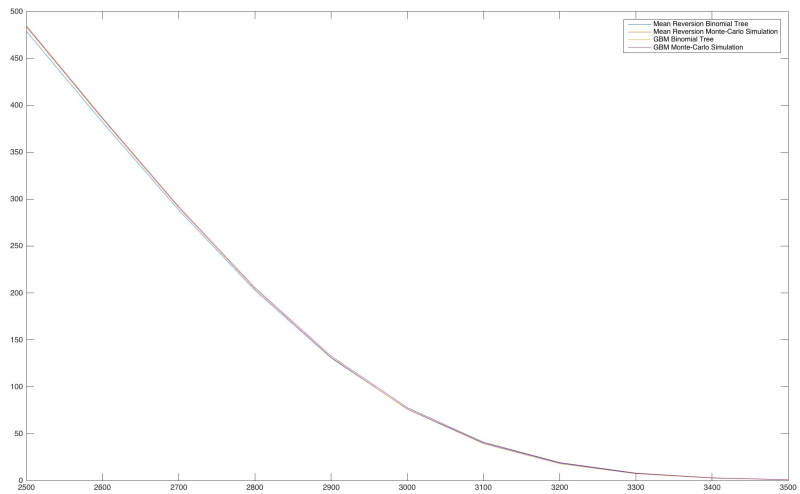

Figure 3: European Call Option Price Comparison for Binomial-Tree and Monte-Carlo simulation for mean-reversion process and Geometric Brownian Motion with variety of moneyness levels (strikes)

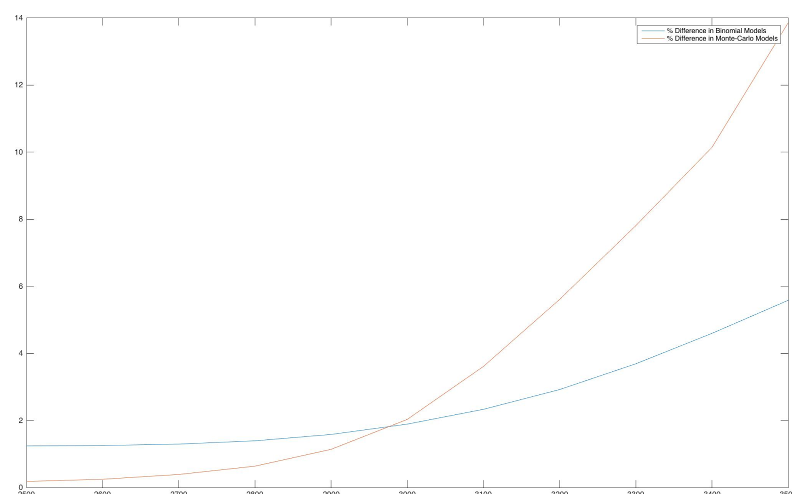

Figure 4: Percentage Differences in Binomial and Monte-Carlo models of Mean Reversion and Geometric Brownian Motion by changing Moneyness

4.2.4 ATM European Call Price Comparison with Different Volatilities

This section now compares how European call prices change with volatility of the option. For this purpose, same call option (3000 Strike) is priced for 10 different volatilities ranges between 1% and 100% with mean reversion process and Geometric Brownian Motion. Prices are plotted in Figure 5 in order to see the change in the option price. Then, Table 8 includes the call prices for all 4 approach for each volatility and the percentage differences.

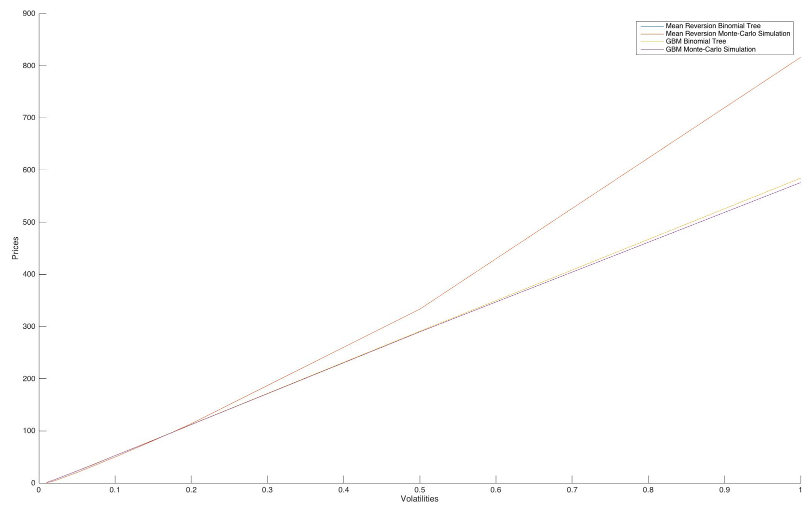

Figure 5: European Call Option Price Comparison for Binomial-Tree and Monte-Carlo simulation for mean-reversion process and Geometric Brownian Motion with variety of volatility levels

Models \Volatilities	**1.00%**	**2.00%**	**5.00%**	**7.50%**	**10.00%**	**14.24%**	**17.50%**	**20.00%**	**50.00%**	**100.00%**
MR with Binomial Tree	0.4756	4.0509	20.2210	34.9376	49.8938	76.3367	97.3731	113.8047	333.3754	816.1165
MR with Monte-Carlo Simulation	0.4671	4.0550	20.0604	34.9602	50.2636	76.4191	96.6940	114.5275	329.8715	818.6150
GBM with Binomial Tree	1.3615	6.1790	23.3371	38.0484	52.6407	77.8093	97.3315	112.2874	291.0210	584.5203
GBM with Monte-Carlo Simulation	1.3703	6.1713	23.1835	37.4871	52.7297	77.3421	96.9256	112.0025	289.8189	576.1182
% Difference in Binomial Tree	65.1%	34.4%	13.4%	8.2%	5.2%	1.9%	0.0%	1.4%	14.6%	39.6%
% Difference in Monte-Carlo Simulation	65.9%	34.3%	13.5%	6.7%	4.7%	1.2%	0.2%	2.3%	13.8%	42.1%

Table 8: Comparison of European Call Prices with different models and Volatility levels

The change in differences are depicted in Figure 6:

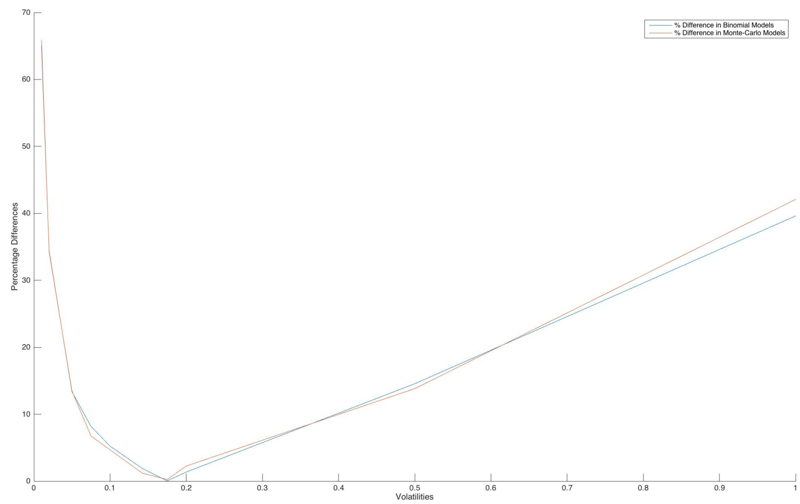

Figure 6: Percentage Differences in Binomial and Monte-Carlo models of Mean Reversion and Geometric Brownian Motion by changing Volatility

4.2.5 ATM European Call Price Comparison with Different Maturities

This section is reserved for comparison of European call prices with changing maturities of the option. For this purpose, same call option (3000 Strike) is priced for 7 different maturities ranges between 5 days to 2 years with mean reversion process and Geometric Brownian Motion. Prices are shown in Figure 7 in order to see the change in the option price. Then, Table 9 depicts the call prices for all 4 approach for each maturity and the percentage differences.

Models \ Maturities	**5**	**10**	**21**	**63**	**186**	**252**	**504**
MR with Binomial Tree	14.5226	25.2495	39.6132	76.3367	137.6143	160.2465	219.7209
MR with Monte-Carlo Simulation	12.4788	22.6279	38.5938	76.2959	141.9543	166.8206	240.9241
GBM with Binomial Tree	14.5440	25.3233	39.8649	77.8093	145.6594	173.1165	256.8153
GBM with Monte-Carlo Simulation	14.9448	24.3830	39.7586	77.8515	146.3361	173.0816	258.1023
% Difference in Binomial Tree	0.1%	0.3%	0.6%	1.9%	5.5%	7.4%	14.4%
% Difference in Monte-Carlo Simulation	16.5%	7.2%	2.9%	2.0%	3.0%	3.6%	6.7%

Table 9: Comparison of European Call Prices with different models and Maturities

Lastly, the change in differences are depicted in Figure 8:

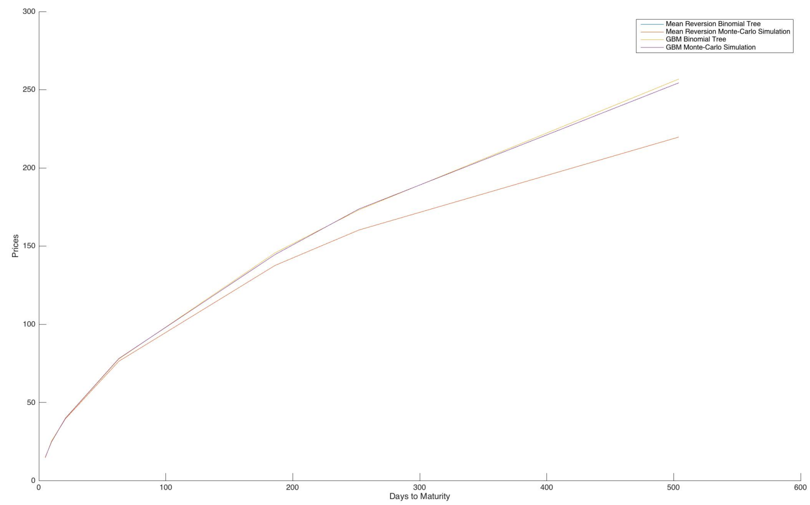

Figure 7: European Call Option Price Comparison for Binomial-Tree and Monte-Carlo simulation for mean-reversion process and Geometric Brownian Motion with variety of maturities

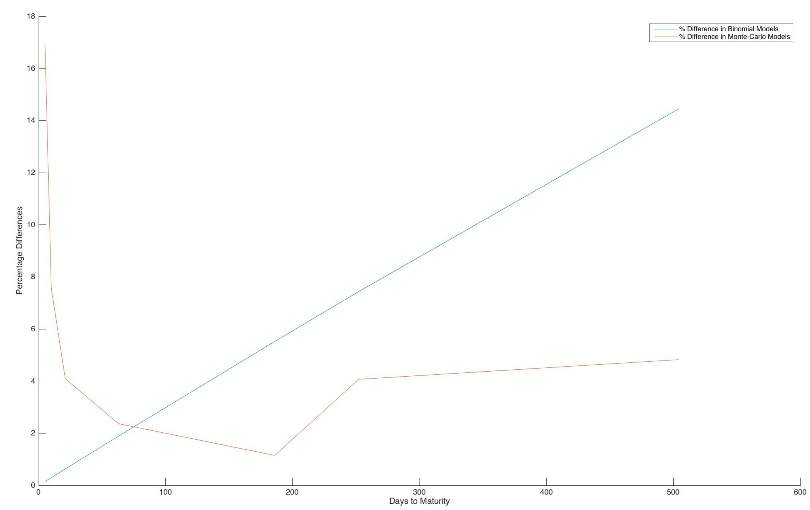

Figure 8: Percentage Differences in Binomial and Monte-Carlo models of Mean Reversion and Geometric Brownian Motion by changing Maturities

4.2.6 ATM European Call Price with Mean Reversion Process Comparison with Changing Mean Reversion Speed Levels

This section presents the results of observation of sensitivity of option prices to mean reversion speed. Same ATM call option is priced with binomial trees and Monte-Carlo simulation with mean reversion process using mean reversion speed levels from 0.0001 to 0.5

Figure 9 shows how sensitive option prices are to mean reversion speed. For simplicity, logarithmic scale is used for changing mean reversion speeds.

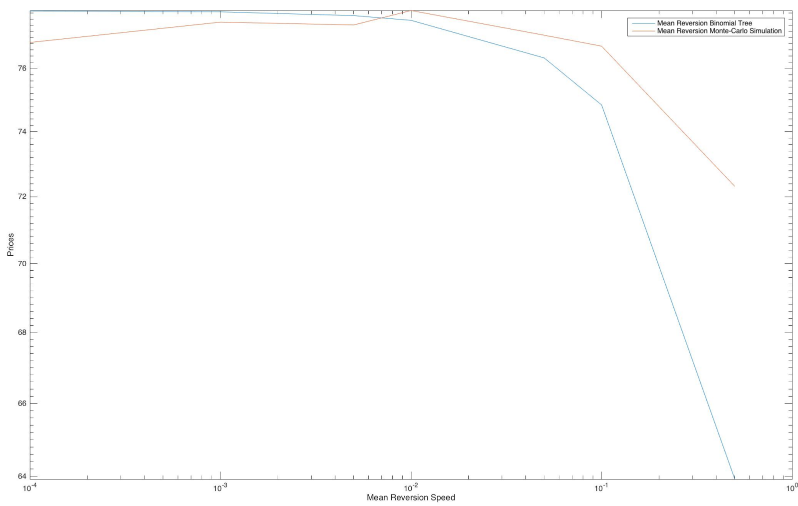

Figure 9: Sensitivity of European Call Option prices to Mean Reversion Speed Levels

5 Comments and Conclusion

6 Annex

6.1 IRS Engine Code

```
# -*- coding: utf-8 -*-
"""
Created on Mon Nov  4 11:23:25 2019

@author: aytekm
"""

import pandas as pd
import numpy as np
import datetime as dt
import copy
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline
from dateutil.relativedelta import relativedelta
import scipy.optimize as opt


class Curve:
    '''
    Curve object:
        Object accepts swap yield curve with respective
            days to maturity
        and its compounding frequency as input.

        Assumptions:
            Day count: 30/360
            Ignore holidays, weekends

        Object attributes:
            yield_curve
            zero_curve
            fwd_curve
            discount_factor

        Object methods:

            Interpolate:              Cubic spline
                interpolation of broken days from a given
                curve
            Curve2Discount:           Conversion of any
                given zero curve to discount factors
            CurveShift:               Curve shift of any
```

```
                given Curve object by specified basis points
            Discrete2Continuous      Conversion of curve with discrete given compounding to continuous compounding
            Continuous2Discrete      Conversion of curve with continuous compounding to discrete given compounding
            Continuous2DiscreteDF    Conversion of dataframe with continuous compounding to discrete given compounding
    '''
    def Discrete2Continuous(self,curve,compound_freq):
        df = getattr(self,curve)
        return np.log(pow((1+(df*compound_freq/36000)),360/compound_freq)) * 100

    def Continuous2Discrete(self,curve,compound_freq):
        df = getattr(self,curve)
        return (pow(np.exp(df/100),compound_freq/360)-1)*36000/compound_freq

    def Continuous2DiscreteDF(self,df):
        compound_freq = self.compound_freq
        return (pow(np.exp(df/100),compound_freq/360)-1)*36000/compound_freq

    def __init__(self, curve_df,compound_freq):

        self.curve_df = curve_df
        self.max_dtm = curve_df.Dtm.max()
        self.compound_freq = compound_freq


        def Interpolate3M(df):
            cs = CubicSpline(df.index,list(df))
            for d in np.arange(90,self.max_dtm,90):
                #df.loc[d] = np.interp(d,df.index,list(df))
                df.loc[d] = float(cs(d))
                df.sort_index(inplace=True)
            return df


        def ZeroCurve(df):
            zero_df = pd.Series(index = df.index,name='
```

```
Rate')
            zero_df.loc[:compound_freq] = df.loc[:compound_freq]
            self.discount_factor.loc[:compound_freq] = np.exp(-zero_df.loc[:compound_freq]*zero_df.loc[:compound_freq].index /36000)

            dtm_to_bootstrap  = list(zero_df.loc[compound_freq:].index[1:])
            for d in dtm_to_bootstrap:
                d_prev = d - 90
                bootstrap_pv = 0
                int_payment = np.exp(df.loc[d] * (90/360)/100) - 1
                while d_prev>0:
                    bootstrap_pv += int_payment * self.discount_factor.loc[d_prev]
                    d_prev -= 90

                self.discount_factor.loc[d] = (1 - bootstrap_pv) /(1 + int_payment)
                zero_df.loc[d] = -np.log(self.discount_factor.loc[d])*(360/d) * 100

            return zero_df

        def FwdCurve(df):
            fwd_df = pd.Series(index=np.arange(0,self.max_dtm,90),name='Rate')
            fwd_df.loc[0] = df.loc[90]

            for d in np.arange(90,self.max_dtm,90):
                fwd_df.loc[d] = (np.log(np.exp(df.loc[d+90] * (d+90)/36000) / np.exp(df.loc[d] * (d)/36000)) * (360/90)) *100

            return fwd_df

        self.yield_curve = Interpolate3M(curve_df.set_index('Dtm')['Rate'])
        self.yield_curve = self.Discrete2Continuous('yield_curve',compound_freq)
        self.discount_factor = pd.Series(index=self.yield_curve.index,name='Rate')
        self.zero_curve = ZeroCurve(self.yield_curve)
        self.fwd_curve = FwdCurve(self.zero_curve)
```

```python

    def Interpolate(self,curve,dtm):
        df_to_interpolate = getattr(self,curve)
        cs = CubicSpline(df_to_interpolate.index,list(
            df_to_interpolate))
        return pd.Series(list(cs(dtm)),index=[dtm],name='
            Rate')

    def Curve2Discount(self,curve,dtm):
        return np.exp(-np.array(curve) * (np.array(dtm)
            /360)/100)

    def CurveShift(self,shift):
        curve_df = copy.deepcopy(self.curve_df)
        curve_df['Rate'] = curve_df['Rate'] + shift/10000
        return Curve(curve_df,self.compound_freq)

class IRS:
    '''
    Curve object:
        Object accepts forward curve, discount curve,
            initial notional, start and end dates,
        amortisation type, amortisation schedule if
            needed as input

        Assumptions:
            Short last coupon in case the swap has broken
                dates

        Object attributes:
            fwd_curve
            discount_curve
            notional
            today
            start_date
            end_date
            amortisation_type
            date_schedule
            num_of_payments
            amortisation_schedule

        Object methods:
            CalculateValue:      Calculation of present
                value of any given already issued swap
            CalculatePar:        Calculation of par value
                of any given new swap
```

```
    '''

    def __init__(self,fwd_curve,discount_curve,notional,today,start_date,end_date,amortisation_type='Constant',amortisation_schedule=None):

        self.fwd_curve= fwd_curve
        self.discount_curve = discount_curve
        self.notional  = notional
        self.today = today
        self.start_date = start_date
        self.end_date = end_date
        self.amortisation_type= amortisation_type


        def ScheduleGenerator(self):

            schedule = list()
            schedule.append(self.start_date)
            new_date = self.start_date + relativedelta(months=3)
            while new_date <= self.end_date:
                schedule.append(new_date)
                new_date += relativedelta(months=3)
            if (new_date - relativedelta(months=3)) > self.end_date:
                schedule.append(self.end_date)
            return schedule

        self.date_schedule = ScheduleGenerator(self)
        self.num_of_payments = len(self.date_schedule)-1

        if self.amortisation_type =='Constant':
            self.amortisation_schedule = [notional for i in range(self.num_of_payments)]
        elif self.amortisation_type =='Linear':
            self.amortisation_schedule = [notional - (i/self.num_of_payments)*notional for i in range(self.num_of_payments)]
        elif self.amortisation_type =='Custom':
            if len(amortisation_schedule) == self.num_of_payments:
                    self.amortisation_schedule  = amortisation_schedule
            else:
                print('No amortisation schedule is
```

```
                    provided or provided schedule is not suitable with the given swap parameters. Linear amortisation is applied instead')
                self.amortisation_schedule = [notional - (i/self.num_of_payments)*notional for i in range(self.num_of_payments)]


    def CalculateValue(self,par):

        df = pd.DataFrame(index = range(self.num_of_payments),columns=['Period_Start','Period_End','Dtm','Dtp','Notional','Reset_Rate','Zero_Rate','Discount_Factor',
                                                 'Interest_Payment_Float','Payment_PV','Fixed_Rate','Fixed_Payment',
                                                 'Fixed_Payment_PV'])
        df.Period_Start = self.date_schedule[:-1]
        df.Period_End = self.date_schedule[1:]
        df.Dtm = (df.Period_End - self.today).dt.days
        df.Notional = self.amortisation_schedule
        df = df[df.Dtm>=0]
        df.reset_index(drop=True,inplace=True)
        if df.loc[0,'Period_Start']<self.today:
             df.loc[0,'Period_Start'] = self.today

        df.Dtp = (df.Period_End - df.Period_Start).dt.days
        df.Reset_Rate  = list(self.fwd_curve.Interpolate('fwd_curve',df.Dtm))
        df.Zero_Rate = list(self.discount_curve.Interpolate('zero_curve',df.Dtm))
        df.Discount_Factor= list(self.discount_curve.Curve2Discount(df.Zero_Rate,df.Dtm))
        df.Interest_Payment_Float = df.Notional * (np.exp(df.Reset_Rate/100 * (df.Dtp/360))-1)
        df.Payment_PV = df.Interest_Payment_Float * df.Discount_Factor
        df.Fixed_Rate = np.ones(len(df))*par
        df.Fixed_Payment = df.Notional * (np.exp(df.Fixed_Rate/100 * (df.Dtp/360))-1)
        df.Fixed_Payment_PV = df.Fixed_Payment * df.
```

```
            Discount_Factor

        return df.Payment_PV.sum() - df.Fixed_Payment_PV.
            sum()

    def CalculatePar(self):

        r0=1
        pv = lambda r: self.CalculateValue(r)
        res= opt.root(pv, r0, method="hybr")
        return self.fwd_curve.Continuous2DiscreteDF(res.x
            [0])


### main
def main():

    valuation_date= pd.to_datetime('2019/11/04',dayfirst=
        False)

    #read curves
    us_libor_df = pd.read_csv(r'Curves/US_LIBOR.csv')
    us_ois_df = pd.read_csv(r'Curves/US_OIS.csv')
    euribor_df = pd.read_csv(r'Curves/EURIBOR.csv')
    eur_ois_df = pd.read_csv(r'Curves/EUR_OIS.csv')

    #plot curves
    fig,ax = plt.subplots(2,1,figsize=(10,5))
    ax[0].plot('Dtm','Rate',data=us_libor_df,marker='o',
        color='r',label='US Libor')
    ax[0].plot('Dtm','Rate',data=us_ois_df,marker='o',
        color='b',label='US OIS')
    ax[0].legend()
    ax[1].plot('Dtm','Rate',data=euribor_df,marker='o',
        color='r',label='Euribor')
    ax[1].plot('Dtm','Rate',data=eur_ois_df,marker='o',
        color='b',label='EUR OIS')
    ax[1].legend()
    fig.savefig('Curves.png')


    #curve objects
    US_Libor = Curve(us_libor_df,compound_freq = 90)
    US_OIS = Curve(us_ois_df,compound_freq = 90)
    Euribor = Curve(euribor_df,compound_freq = 90)
    EUR_OIS = Curve(eur_ois_df,compound_freq = 90)
```

```
##################################
    #US
    #already issued swap with 1.5 years of remaining maturity
    start_date = pd.to_datetime('2019/06/01',dayfirst=False)
    end_date =  pd.to_datetime('2021/06/01',dayfirst=False)
    usd_irs1 = IRS(US_Libor,US_OIS,10000000,valuation_date,start_date,end_date,amortisation_type='Constant')
    usd_irs1_value = usd_irs1.CalculateValue(par=3)

    #already issued swap with 1.5 years of remaining maturity with custom amortisation schedule
    start_date = pd.to_datetime('2019/06/01',dayfirst=False)
    end_date =  pd.to_datetime('2021/06/01',dayfirst=False)
    notional  = 10000000
    schedule = [notional,notional*15/16,notional*14/16,notional*13/16,notional*12/16,
                notional*11/16,notional*10/16,notional*9/16]
    usd_irs2 = IRS(US_Libor,US_OIS,notional,valuation_date,start_date,end_date,amortisation_type='Custom',amortisation_schedule=schedule)
    usd_irs2_value = usd_irs2.CalculateValue(par=3)

    #EUR
    #already issued swap with 1.5 years of remaining maturity
    start_date = pd.to_datetime('2019/06/01',dayfirst=False)
    end_date =  pd.to_datetime('2021/06/01',dayfirst=False)
    eur_irs1 = IRS(Euribor,EUR_OIS,10000000,valuation_date,start_date,end_date,amortisation_type='Constant')
    eur_irs1_value = eur_irs1.CalculateValue(par=0.5)


    #already issued swap with 1.5 years of remaining maturity with custom amortisation schedule
```

```
    start_date = pd.to_datetime('2019/06/01',dayfirst=False)
    end_date =  pd.to_datetime('2021/06/01',dayfirst=False)
    notional  = 10000000
    schedule = [notional,notional*15/16,notional*14/16,notional*13/16,notional*12/16,
                notional*11/16,notional*10/16,notional*9/16]
    eur_irs2 = IRS(Euribor,EUR_OIS,notional,valuation_date,start_date,end_date,amortisation_type='Custom',amortisation_schedule=schedule)
    eur_irs2_value = eur_irs2.CalculateValue(par=0.5)


###########################
    #US
    #spot starting swap with 5 years of maturity and no amortisation
    start_date = pd.to_datetime('2019/11/04',dayfirst=False)
    end_date =  pd.to_datetime('2024/11/04',dayfirst=False)
    usd_irs3 = IRS(US_Libor,US_OIS,10000000,valuation_date,start_date,end_date,amortisation_type='Constant')
    usd_irs3_price = usd_irs3.CalculatePar()

    #spot starting swap with 5 years of maturity and linear amortisation
    start_date = pd.to_datetime('2019/11/04',dayfirst=False)
    end_date =  pd.to_datetime('2024/11/04',dayfirst=False)
    usd_irs4 = IRS(US_Libor,US_OIS,10000000,valuation_date,start_date,end_date,amortisation_type='Linear')
    usd_irs4_price = usd_irs4.CalculatePar()

    #EUR
    #spot starting swap with 5 years of maturity and no amortisation
    start_date = pd.to_datetime('2019/11/04',dayfirst=False)
    end_date =  pd.to_datetime('2024/11/04',dayfirst=
```

```
False)
    eur_irs3 = IRS(Euribor,EUR_OIS,10000000,valuation_date,start_date,end_date,amortisation_type='Constant')
    eur_irs3_price = eur_irs3.CalculatePar()

    #spot starting swap with 5 years of maturity and linear amortisation
    start_date = pd.to_datetime('2019/11/04',dayfirst=False)
    end_date =  pd.to_datetime('2024/11/04',dayfirst=False)
    eur_irs4 = IRS(Euribor,EUR_OIS,10000000,valuation_date,start_date,end_date,amortisation_type='Linear')
    eur_irs4_price = eur_irs4.CalculatePar()

####################################
    #USD
    #already started 20yr IRS with no amortisation and fixed rate of 2.1%
    start_date = pd.to_datetime('2018/12/01',dayfirst=False)
    end_date =  pd.to_datetime('2039/12/01',dayfirst=False)
    usd_irs5 = IRS(US_Libor,US_OIS,10000000,valuation_date,start_date,end_date,'Constant')
    usd_irs5_value = usd_irs5.CalculateValue(2.1)

    Libor_100_up   = US_Libor.CurveShift(100)
    Libor_100_down  = US_Libor.CurveShift(-100)

    usd_irs6_value  = IRS(Libor_100_down,US_OIS,10000000,valuation_date,start_date,end_date,'Constant').CalculateValue(2.1)
    usd_irs7_value  = IRS(Libor_100_up,US_OIS,10000000,valuation_date,start_date,end_date,'Constant').CalculateValue(2.1)

    #EUR
    #already started 20yr IRS with no amortisation and fixed rate of 2.1%
    start_date = pd.to_datetime('2018/12/01',dayfirst=False)
    end_date =  pd.to_datetime('2039/12/01',dayfirst=False)
```

```
    eur_irs5 = IRS(Euribor,EUR_OIS,10000000,
        valuation_date,start_date,end_date,'Constant')
    eur_irs5_value = eur_irs5.CalculateValue(1)

    Euribor_100_up   = Euribor.CurveShift(100)
    Euribor_100_down   = Euribor.CurveShift(-100)

    eur_irs6_value   = IRS(Euribor_100_down,EUR_OIS
        ,10000000,valuation_date,start_date,end_date,'
        Constant').CalculateValue(1)
    eur_irs7_value   = IRS(Euribor_100_up,EUR_OIS
        ,10000000,valuation_date,start_date,end_date,'
        Constant').CalculateValue(1)


### start main
if __name__ == "__main__":
    main()
```

6.2 Mean Reversion Code

6.2.1 Main Code

```
%%Assignment1 Q2

%%MEAN REVERSION

%%BINOMIAL TREE

%parameters
std_dev = 0.1424;
stock_price = 2978.4;
NumPeriods = 63;
int_rate = 0.01;
compound_freq = 0.25;
option_maturity = 0.25;
cont_rate = log(power((1+int_rate*compound_freq),1/
    compound_freq));
reversion_speed = 0.05;
reversion_level = log(stock_price);

[BinTree,rate,p_up,p_down] = mean_reversion_tree(
    stock_price,std_dev,NumPeriods,cont_rate,
    option_maturity,reversion_speed,reversion_level);

%%3000 strike european call
```

```
europ_call_3000 = mean_reversion_call(BinTree,3000,rate,
    p_up,p_down);

%%%european put strike 3000
europ_put_3000 = mean_reversion_put(BinTree,3000,rate,
    p_up,p_down);

%%option prices for all strikes
strikes = 2500:100:3500;
call_prices = zeros(1,length(strikes));
put_prices = zeros(1,length(strikes));
strike_count=1;

for strike=strikes
    call_prices(1,strike_count) = mean_reversion_call(
        BinTree,strike,rate,p_up,p_down);
    put_prices(1,strike_count) = mean_reversion_put(
        BinTree,strike,rate,p_up,p_down);
    strike_count = strike_count+1;
end

%%%european exotic
strikes = 2500:250:3500;
european_exotic_prices = zeros(1,length(strikes));
strike_count=1;
for strike=strikes
    european_exotic_prices(1,strike_count) =
        mean_reversion_european_exotic(BinTree,strike,rate
        ,p_up,p_down);
    strike_count = strike_count+1;
end

%%MONTE-CARLO
M = 100000;
time_step = option_maturity/NumPeriods;
discount  = exp(-cont_rate*option_maturity);

MC_matrix = zeros(M,NumPeriods);
MC_matrix(:,1) = log(stock_price);

for i=2:NumPeriods
    brownian = randn(M,1);
    MC_matrix(:,i) = MC_matrix(:,i-1) + reversion_speed *
         (reversion_level - MC_matrix(:,i-1)) * time_step
        + std_dev*brownian*sqrt(time_step);
end
```

```
MC = exp(MC_matrix(:,NumPeriods));

%%%%3000 strike european call
europ_call_3000_mc =  mean(discount * max(MC-3000,0));

%%%%3000 strike european put
europ_put_3000_mc =  mean(discount * max(3000-MC,0));

%%option prices for all strikes
strikes = 2500:100:3500;
call_prices_mc = zeros(1,length(strikes));
put_prices_mc = zeros(1,length(strikes));
strike_count=1;

for strike=strikes
    call_prices_mc(1,strike_count) = mean(discount * max(
        MC-strike,0));
    put_prices_mc(1,strike_count) = mean(discount * max(
        strike-MC,0));
    strike_count = strike_count+1;
end

%%%european exotic
strikes = 2500:250:3500;
european_exotic_prices_mc = zeros(1,length(strikes));
strike_count=1;
for strike=strikes
    european_exotic_prices_mc(1,strike_count) = mean(
        discount * power((strike-MC),2));
    strike_count = strike_count+1;
end
```

6.2.2 Mean Reversion Binomial Tree Code

```
function [BinTree,rate,p_up,p_down] = mean_reversion_tree
    (stock_price,std_dev,NumPeriods,cont_rate,
    option_maturity,reversion_speed,reversion_level)

    time_step = option_maturity/NumPeriods;
    BinTree = zeros(NumPeriods+1);
    p_up = zeros(NumPeriods+1);
    %%build tree by hand
    for i = 1:NumPeriods+1
        for j=1:i
            x_star = (i-2*j+1)*std_dev*sqrt(time_step);
```

```
            BinTree(j,i) = reversion_level * (1-exp(-
                reversion_speed*(i-1)*time_step)) ...
            + log(stock_price)*exp(-reversion_speed*(i-1)
                *time_step) ...
            + x_star;

            p_up(j,i) = 0.5 * (1+(reversion_speed* -
                x_star * sqrt(time_step))/(sqrt(power(
                reversion_speed*-x_star,2)*time_step+power
                (std_dev,2))));
        end
    end

    rate = exp(cont_rate*time_step)-1;
    p_down = 1-p_up;
end
```

6.2.3 Mean Reversion Call Pricing Code

```
function f = mean_reversion_call(BinTree,Strike,rate,p_up
    ,p_down)

    treeLength = length(BinTree);
    OptPrice(:,treeLength) = max(0,exp(BinTree(:,
        treeLength)) - Strike);
    for i = treeLength-1:-1:1
        for j=1:i
            OptPrice(j,i) = (OptPrice(j,i+1)*p_up(j,i+1)
                + OptPrice(j+1,i+1)*p_down(j+1,i+1))/(1+
                rate);
        end
    end
    f = OptPrice(1,1);
end
```

6.2.4 Mean Reversion Put Pricing Code

```
function f = mean_reversion_put(BinTree,Strike,rate,p_up,
    p_down)

    treeLength = length(BinTree);
    OptPrice(:,treeLength) = max(0,Strike - exp(BinTree
        (:,treeLength)));
    for i = treeLength-1:-1:1
        for j=1:i
            OptPrice(j,i) = (OptPrice(j,i+1)*p_up(j,i+1)
                + OptPrice(j+1,i+1)*p_down(j+1,i+1))/(1+
```

```
                rate);
        end
    end
    f = OptPrice(1,1);
end
```

6.2.5 Mean Reversion Exotic Option Pricing Code

```
function f = mean_reversion_european_exotic(BinTree,
    Strike,rate,p_up,p_down)

    treeLength = length(BinTree);
    OptPrice(:,treeLength) = power((exp(BinTree(:,
        treeLength)) - Strike),2);
    for i = treeLength-1:-1:1
        for j=1:i
            OptPrice(j,i) = (OptPrice(j,i+1)*p_up(j,i+1)
                + OptPrice(j+1,i+1)*p_down(j+1,i+1))/(1+
                rate);
        end
    end
    f = OptPrice(1,1);
end
```

6.2.6 GBM Binomial Tree Code

```
function [BinTree,rate,p_up,p_down] = gbm_tree(
    stock_price,std_dev,NumPeriods,cont_rate,
    option_maturity)
time_step = option_maturity/NumPeriods;
u = exp(std_dev*sqrt(time_step));
    d = 1/u;
    BinTree = zeros(NumPeriods+1);

    %%build tree by hand
    for i = 1:NumPeriods+1
        for j=1:i
            BinTree(j,i) = stock_price * power(u,i-j) *
                power(d,j-1);
        end
    end

    rate = exp(cont_rate*time_step)-1;
    p_up = (1+rate-d)/(u-d);
    p_down = 1-p_up;
end
```

6.2.7 GBM Call Pricing Code

```
function f = gbm_call(BinTree,Strike,rate,p_up,p_down)

    treeLength = length(BinTree);
    OptPrice(:,treeLength) = max(0,BinTree(:,treeLength)
        - Strike);
    for i = treeLength-1:-1:1
        for j=1:i
            OptPrice(j,i) = (OptPrice(j,i+1)*p_up +
                OptPrice(j+1,i+1)*p_down)/(1+rate);
        end
    end
    f = OptPrice(1,1);
end
```

References

[1] C. Bastian-Pinto, L. Brandão, and W. Hahn, "A non-censored binomial model for mean reverting stochastic processes," *Proceedings 14. Annual international conference on real options*, 01 2010.