Inicialization of inputs

In [ ]:
```
import math
from scipy.stats import norm
from scipy.optimize import differential_evolution
from scipy.optimize import curve_fit
from scipy.optimize import fsolve
import pandas as pd
import numpy as np
from scipy.interpolate import CubicSpline
from datetime import datetime
import matplotlib.pyplot as plt
from sympy import *
import QuantLib as ql
from scipy.stats.distributions import t
import scipy.stats as st
import re
from scipy.stats import norm
import statsmodels.api as sm
from statsmodels.distributions.empirical_distribution import ECDF
from sympy.solvers import solve
from sympy import Symbol
from scipy.stats import pearsonr
import seaborn as sns
from statsmodels.nonparametric.smoothers_lowess import lowess
```

In [ ]:
```
data_sigm_p = pd.read_excel(r'C:\Users\uzivatel\Downloads\CZK Curves I.xlsx', sheet_name='CZK CAPLET')
data_ir_p = pd.read_excel(r'C:\Users\uzivatel\Downloads\CZK Curves I.xlsx', sheet_name='CZK 6M IRS')
data_swap_p = pd.read_excel(r'C:\Users\uzivatel\Diplomka\SwaptionVolatility.xlsx')
data_hist_p = pd.read_excel(r'C:\Users\uzivatel\Diplomka\CZK CurveHistory.xlsx')
data_sigm_cap_p = pd.read_excel(r'C:\Users\uzivatel\Diplomka\CZK Curves cap.xlsx', sheet_name='CZK CAP')
data_hist_swap_p = pd.read_excel(r'C:\Users\uzivatel\Diplomka\SwaptionVolHistory.xlsx', sheet_name='CZKSwaption'

data_sigm = pd.DataFrame(data_sigm_p) # volatilities for caplets and strikes
data_ir = pd.DataFrame(data_ir_p) #yields from zero curve
data_swap = pd.DataFrame(data_swap_p) #swaption volatilities, expiries and tenors
data_hist = pd.DataFrame(data_hist_p) # historical yield curves for different tenors
data_sigm_cap = pd.DataFrame(data_sigm_cap_p) # volatilities for cap and strikes

data_sigm['Strike'] = data_sigm['Strike'] - 3 # Input data are shifted
data_sigm_cap['Strike'] = data_sigm_cap['Strike'] - 3
```

In [ ]:
```
res2 = []
for i in data_hist['Tenor']:
    if re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[1] == "M":
        res2 = np.append(res2, int(re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[0])/12)
    elif re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[1] == "D":
        res2 = np.append(res2, int(re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[0])/365)
    elif re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[1] == "W":
        res2 = np.append(res2, int(re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[0])/52)
    else:
        res2 = np.append(res2, int(re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[0]))
data_hist['Tenor_num'] = res2
# Translating tenors like '1M' into number in years
```

In [ ]:
```
data_hist = data_hist.sort_values(by=['Tenor_num']).reset_index(drop=True)
```

In [ ]:
```
data_hist['Rate'] = data_hist['Rate']/100
```

In [ ]:
```
data_hist['Spot_diff'] = 0
for j in range(len(data_hist['Rate'])):
    if j % 1257 == 0:
        data_hist['Spot_diff'][j] = 0
    else:
        data_hist['Spot_diff'][j] = data_hist['Rate'][j] - data_hist['Rate'][j - 1]
# Calculating daily changes in historical spot rates
```

In [ ]:
```
sigm_str = data_sigm['MaturityDate'].astype(str).tolist()
index = 0
h_str = np.array([])
while index < len(sigm_str):
    h_str = np.append(h_str,(datetime.strptime(sigm_str[index], '%Y-%m-%d')
                             - datetime.strptime(sigm_str[0], '%Y-%m-%d')).days)
    index += 1
data_sigm['diff_date'] = (h_str/365) + 0.5

ir_str = data_ir['MaturityDate'].astype(str).tolist()
ind = 0
```

```
h_ir = np.array([])
while ind < len(ir_str):
    h_ir = np.append(h_ir,(datetime.strptime(ir_str[ind], '%Y-%m-%d')
                           - datetime.strptime(ir_str[0], '%Y-%m-%d')).days)
    ind += 1
data_ir['diff_date'] = (h_ir + 1)/365

sigm_cap_str = data_sigm_cap['MaturityDate'].astype(str).tolist()
index2 = 0
h_str2 = np.array([])
while index2 < len(sigm_cap_str):
    h_str2 = np.append(h_str2,(datetime.strptime(sigm_cap_str[index2], '%Y-%m-%d')
                               - datetime.strptime(sigm_str[0], '%Y-%m-%d')).days)
    index2 += 1
data_sigm_cap['diff_date'] = (h_str2/365) + 0.5

# Calculating time from the beginning on 31 January 2023
```

In [ ]:
```
res = []
for i in data_swap['Tenor']:
    res = np.append(res, int(re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[0]))
data_swap['Tenor'] = res
# Translating tenors like '1Y' into number in years
```

In [ ]:
```
res1 = []
for i in data_swap['Expiry']:
    if re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[1] == "M":
        res1 = np.append(res1, int(re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[0])/12)
    else:
        res1 = np.append(res1, int(re.compile("([0-9]+)([a-zA-Z]+)").match(i).groups()[0]))
data_swap['Expiry'] = res1
# Translating tenors like '1M' into number in years
```

In [ ]:
```
data_swap=data_swap.sort_values(by=['Expiry', 'Tenor']).reset_index(drop=True)
data_swap['Rate'] = data_swap['Rate']/100
```

In [ ]:
```
cs = CubicSpline(data_ir['diff_date'],data_ir['ZeroRate']/100, bc_type='natural')
cs1 = cs.derivative(1)
plt.plot(np.linspace(0, 30, 100), cs(np.linspace(0, 30, 100)))
plt.savefig("plot1.png")
plt.show()
# Cubic spline interpolation of the data points from the given zero curve and its first derivative
# and its plot
```

Valuation functions

Black-Scholes pricing of caplets and caps

In [ ]:
```
def caplet_val_BS(fixing, tenor,spot, sigma, strike, shift):
    fixing0 = fixing[fixing == 0]
    fixing = fixing[fixing != 0]

    P_0T1=np.exp(-spot(fixing)*fixing)
    P_0T2=np.exp(-spot(fixing + tenor)*(fixing + tenor))
    F_0T1T2=((P_0T1/P_0T2)-1)/tenor
    v=sigma*np.sqrt(fixing)

    caplet = ()
    if len(fixing0) > 0:
        caplet0 = 1*np.exp(-spot(fixing0 + tenor)*(fixing0 + tenor))*tenor*np.maximum(spot(fixing0) - strike, 0
        caplet = np.append(caplet, caplet0)

    d1=(np.log((F_0T1T2+shift)/(strike+shift))+(v**2/2))/v
    d2=(np.log((F_0T1T2+shift)/(strike+shift))-(v**2/2))/v
    Bl=(F_0T1T2+shift)*norm.cdf(d1) - (strike+shift)*norm.cdf(d2)
    Caplet_Bl=1*P_0T2*tenor*Bl
    caplet = np.append(caplet, Caplet_Bl)
    return caplet
```

In [ ]:
```
def cap_val_BS(maturity, tenor, spot, sigma, strike, shift):
    cap_BS = pd.DataFrame()
    cap_BS['Strike'] = strike
    cap_BS['Tenor'] = tenor
    cap_BS['Maturity'] = maturity
    cap_BS['Price'] = 0

    i = 0
    for x in maturity:
        if x > 0.5:
```

```
            mats = np.arange(0, x - tenor, tenor)
            cap_BS['Price'][i] = np.sum(
                caplet_val_BS(mats, tenor, spot, np.unique(sigma[i])[0], np.unique(strike[i])[0], shift))
        i += 1
    return cap_BS
```

Hull-White pricing of caplets and caps

In [ ]:
```
def caplet_val_HW(fixing, tenor, spot, sigma, strike, a):
    pom = 1+strike*tenor
    fixing0 = fixing[fixing == 0]
    fixing = fixing[fixing != 0]
    P_0T1=np.exp(-spot(fixing)*fixing)
    P_0T2=np.exp(-spot(fixing + tenor)*(fixing + tenor))
    cap = ()
    if len(fixing0) > 0:
        cap0 = 1*np.exp(-spot(fixing0 + tenor)*(fixing0 + tenor))*tenor*np.maximum(spot(fixing0) - strike, 0) #
        cap = np.append(cap, cap0)
    if a == 0:
        sigma_p=sigma*tenor*np.sqrt(fixing)
    else:
        sigma_p=((sigma*(1-np.exp(-a*tenor)))/a)*np.sqrt((1-np.exp(-2*a*fixing))/(2*a))
    h = (1/sigma_p)*np.log((P_0T2*pom)/P_0T1)+(sigma_p/2)
    cap1 = P_0T1*norm.cdf(-h + sigma_p) - pom*P_0T2*norm.cdf(-h)
    cap = np.append(cap, cap1)
    return cap
```

In [ ]:
```
def cap_val_HW(maturity, tenor, spot, sigma, strike, a):
    cap_HW = pd.DataFrame()
    cap_HW['Strike'] = strike
    cap_HW['Tenor'] = tenor
    cap_HW['Maturity'] = maturity
    cap_HW['Price'] = 0
    i = 0
    for x in maturity:
        if x > 0.5:
            mats = np.arange(0, x - tenor, tenor)
            cap_HW['Price'][i] = np.sum(caplet_val_HW(mats, tenor, spot, sigma, np.unique(strike[i])[0], a))
        i += 1
    return cap_HW
```

Swaption pricing

In [ ]:
```
def bond_sum(Ta, tenor, spot):
    ar=np.array([])
    for j in range(len(Ta)):
        ar2 = np.array([])
        i = 0
        while i < tenor[j]:
            i = i + 0.5
            ar2 = np.append(ar2, np.exp(-spot(Ta[j] + i)*(Ta[j] + i)))
        ar = np.append(ar, np.sum(ar2))
    return ar

def bond_sum_all(Ta, tenor, spot):
    ar=np.array([])
    for j in range(len(Ta)):
        ar2 = np.array([])
        i = 0
        while i < tenor[j]:
            i = i + 0.5
            ar2 = np.append(ar2, np.exp(-spot(Ta[j] + i)*(i)))
        ar = np.append(ar, np.sum(ar2))
    return ar

def swaption_val_BS(expiry, tenor, spot, sigma):
    df=pd.DataFrame()
    df['Expiry']=expiry
    df['Tenor']=tenor
    P_0Ta=np.exp(-spot(expiry)*expiry)
    P_0Tb=np.exp(-spot(expiry + tenor)*(expiry + tenor))
    S_ab0=(P_0Ta - P_0Tb)/(0.5*bond_sum(expiry, tenor, spot))
    df['Strike']=S_ab0
    d1=(np.log(S_ab0/S_ab0)+((sigma*np.sqrt(expiry))**2/2))/(sigma*np.sqrt(expiry))
    d2=(np.log(S_ab0/S_ab0)-((sigma*np.sqrt(expiry))**2/2))/(sigma*np.sqrt(expiry))
    bl=(S_ab0)*norm.cdf(d1) - (S_ab0)*norm.cdf(d2)
    BS_val=1*bl*0.5*bond_sum(expiry, tenor, spot)
    df['Price']=BS_val
    return df
```

In [ ]:
```
def r_solver(c, A_vec, B_vec):
    def f(r):
        return np.dot(c, (A_vec*np.exp(-B_vec*r))[0]) - 1
    sol=fsolve(f, 0)
    return sol

def swaption_val_HW(expiry, tenor, spot, sigma, alpha): #One swaption pricing in the Hull-White model
    iterable=(0.5+(x/2) for x in range(2*tenor.astype(np.int64)))
    times=np.fromiter(iterable, float)
    expiry_vec=np.full((1,len(times)), expiry)

    P_0Ta=np.exp(-spot(expiry)*expiry)
    P_0Tb=np.exp(-spot(expiry + tenor)*(expiry + tenor))
    P_0Ti=np.exp(-spot(expiry_vec+times)*(expiry_vec+times))
    S_ab0=(P_0Ta - P_0Tb)/(0.5*bond_sum(np.array([expiry]), np.array([tenor]), spot))
    c=np.full((1,len(times)-1), S_ab0*0.5)
    c=np.append(c, 1 + S_ab0*0.5)

    B_vec=(1 - np.exp(-alpha*(expiry_vec+times-expiry_vec)))/alpha
    A_vec=(P_0Ta/P_0Ti)*np.exp(B_vec*(expiry * cs1(expiry) + cs(expiry))
                               - (sigma**2 *(1 - np.exp(-2*alpha*times))* B_vec**2)/(4*alpha))
    sigma_p=sigma*B_vec*np.sqrt((1 - np.exp(-2*alpha*expiry_vec))/(2*alpha))
    strike_i=A_vec*np.exp(-B_vec*r_solver(c, A_vec, B_vec))
    h=(np.log(P_0Ti/(strike_i*P_0Ta))/sigma_p) + (sigma_p/2)
    ZBP=strike_i*P_0Ta*norm.cdf(-h+sigma_p) - P_0Ti*norm.cdf(-h)
    HW_val=1*np.dot(c, ZBP[0])
    return HW_val
```

In [ ]:
```
def swaption_val_HW_all(expiry, tenor, spot, sigma, alpha): # All swaption pricing in the Hull-White model
    df=pd.DataFrame()
    df['Expiry']=expiry
    df['Tenor']=tenor
    P_0Ta=np.exp(-spot(expiry)*expiry)
    P_0Tb=np.exp(-spot(expiry + tenor)*(expiry + tenor))
    S_ab0=(P_0Ta - P_0Tb)/(0.5*bond_sum(expiry, tenor, spot))
    df['Strike']=S_ab0
    df['Price']=0

    for j in range(len(expiry)):
        df['Price'][j]=swaption_val_HW(expiry[j], tenor[j], spot, sigma, alpha)
    return df
```

Caplets calibration

Differential_evolution

In [ ]:
```
bounds = [(0.01, 5),(0.0001, 0.5)] #calibration bounds for alpha and sigma
```

In [ ]:
```
def HW_obj_fun_caplet(x): # first objective function from my thesis for differential_evolution function
    a, sigma = x
    return np.sum(np.square(caplet_val_HW(data_sigm['diff_date'], 0.5, cs, sigma, data_sigm['Strike']/100, a)
            -caplet_val_BS(data_sigm['diff_date'], 0.5, cs, data_sigm['Volatility']/100, data_sigm['Strike']/10

def HW_obj_fun_caplet_2(x): # second objective function from my thesis for differential_evolution function
    a, sigma = x
    return np.sum(np.square((caplet_val_HW(data_sigm['diff_date'], 0.5, cs, sigma, data_sigm['Strike']/100, a)
            -caplet_val_BS(data_sigm['diff_date'], 0.5, cs, data_sigm['Volatility']/100, data_sigm['Strike']/10

def HW_obj_fun_caplet_3(x): # third objective function from my thesis for differential_evolution function
    a, sigma = x
    return np.sqrt(np.sum(np.square((caplet_val_HW(data_sigm['diff_date'], 0.5, cs, sigma, data_sigm['Strike']/
            -caplet_val_BS(data_sigm['diff_date'], 0.5, cs, data_sigm['Volatility']/100, data_sigm['Strike']/10

def opti_choose_OF(func, bounds): #defined function for the application of differential_evolution
    result = differential_evolution(func = func, bounds=bounds, seed=1515)
    a, s = result.x #calibrated parameters
    f = result.fun # minimized value of the objective function
    return (a, s, f)
```

In [ ]:
```
alpha_opt, sigma_opt, fits = opti_choose_OF(HW_obj_fun_caplet, bounds)
alpha_opt2, sigma_opt2, fits2 = opti_choose_OF(HW_obj_fun_caplet_2, bounds)
alpha_opt3, sigma_opt3, fits3 = opti_choose_OF(HW_obj_fun_caplet_3, bounds)
```

In [ ]:
```
est_caplet = pd.DataFrame(np.array([[alpha_opt, sigma_opt, fits], [alpha_opt2, sigma_opt2, fits2],
                            [alpha_opt3, sigma_opt3, fits3]]), columns=['alpha', 'sigma', 'fitness'])
#table of the results
```

Curve_fit and attempts of verification of assumption on residuals

In [ ]:
```
def HW_caplet_price(data, a, sigma): #setting the price for the application of curve_fit (in meaning of the firs
    HW_NPV = caplet_val_HW(data['diff_date'], 0.5, cs, sigma, data['Strike']/100, a)
    return HW_NPV
```

In [ ]:
```
data_sigm['NPV_caplet'] = caplet_val_BS(data_sigm['diff_date'], 0.5, cs,
                            data_sigm['Volatility']/100, data_sigm['Strike']/100, 0.03)
def HW_caplet_price2(data, a, sigma):
    HW_NPV = caplet_val_HW(data['diff_date'], 0.5, cs, sigma, data['Strike']/100, a)/data['NPV_caplet']
    return HW_NPV
```

In [ ]:
```
def est_cf(init_pars, param_bounds):
    popt, pcov, info, msg, ier = curve_fit(HW_caplet_price, data_sigm, data_sigm['NPV_caplet'],
                                p0=[init_pars[0], init_pars[1]], bounds=param_bounds, full_output=True)
    return (popt, pcov, info)
alpha_opt_cf, sigma_opt_cf = est_cf(est_caplet.iloc[0,:], param_bounds)[0]
cov1_caplet = est_cf(est_caplet.iloc[0,:], param_bounds)[1]
resd = est_cf(est_caplet.iloc[0,:], param_bounds)[2]['fvec']
sse = np.sum(resd**2)
# curve_fit with the initial guesses from differential_evolution with the first objective function
```

Diagnostical plots

In [ ]:
```
plt.hist(5000*resd, bins=np.linspace(-5, 5, 50), density=True, alpha=0.5)
plt.plot(np.linspace(-5, 5, 1000), norm.pdf(np.linspace(-5, 5, 1000)), lw=4, alpha=0.7)
plt.show()
```

In [ ]:
```
points = resd + data_sigm['NPV_caplet'].to_numpy()
df_res2 = pd.DataFrame()
df_res2['fitted values'] = points
df_res2['abs.value of std.residuals'] = np.sqrt(np.abs(resd))
sns.residplot(data=df_res2, x="fitted values", y="abs.value of std.residuals", lowess=True, line_kws=dict(color=
```

In [ ]:
```
plt.plot(points, np.sqrt(np.abs(resd)), 'o')
smo = lowess(np.sqrt(np.abs(resd)), points, frac=0.5)
plt.plot(smo[:, 0], smo[:, 1], color='red')
plt.show()
```

In [ ]:
```
df_res = pd.DataFrame()
df_res['fitted values'] = points
df_res['residuals'] = resd
sns.residplot(data=df_res, x="fitted values", y="residuals",
              lowess=True, line_kws=dict(color="r"))
```

In [ ]:
```
sm.qqplot(resd, line='45', fit=True)
plt.show
```

In [ ]:
```
fig, pl = plt.subplots(2,2)
sns.residplot(data=df_res, x="fitted values", y="residuals", lowess=True, line_kws=dict(color="r"), ax=pl[0,0])
pl[0, 1].plot(points, np.sqrt(np.abs(resd)), 'o')
pl[0, 1].plot(smo[:, 0], smo[:, 1], color='red')
pl[0, 1].set(xlabel='fitted values', ylabel='abs.value of std.residuals')
sm.qqplot(resd, line='45', fit=True, ax=pl[1,0])
fig.delaxes(pl[1][1])
plt.tight_layout()
plt.savefig("caplet.png")
plt.show()
#combination of the previous plots
```

Curve_fit calibration – continue

In [ ]:
```
def est_cf2(init_pars, param_bounds):
    popt, pcov, info, msg, ier = curve_fit(HW_caplet_price2, data_sigm, 1,
                                p0=[init_pars[0], init_pars[1]], bounds=param_bounds, full_output=True)
    return (popt, pcov, info)
alpha_opt2_cf, sigma_opt2_cf = est_cf2(est_caplet.iloc[1,:], param_bounds)[0]
cov2_caplet2 = est_cf2(est_caplet.iloc[1,:], param_bounds)[1]
resd2_caplet2 = est_cf2(est_caplet.iloc[1,:], param_bounds)[2]['fvec']
sse2_caplet2 = np.sum(resd2_caplet2**2)
print(alpha_opt2_cf, sigma_opt2_cf, sse2_caplet2)
# curve_fit with the initial guesses from differential_evolution with the second objective function
```

Calibration of sigma on caplets in case of Ho-Lee (alpha = 0)

In [ ]:
```
def HL_obj_fun_caplet(x): # first objective function from my thesis for differential_evolution function
    return np.sum(np.square(caplet_val_HW(data_sigm['diff_date'], 0.5, cs, x, data_sigm['Strike']/100, 0)
                  -caplet_val_BS(data_sigm['diff_date'], 0.5, cs, data_sigm['Volatility']/100, data_sigm['Str
```

```
def opti_choose_OF_sigma(func, bounds): #defined function for the application of differential_evolution
    result = differential_evolution(func = func, bounds=bounds, seed=1515)
    s = result.x #calibrated parameter
    f = result.fun # minimized value of the objective function
    return (s, f)

sigma_opt_HL, fits_HL = opti_choose_OF_sigma(HL_obj_fun_caplet, [(0.0001, 0.5)])
```

In [ ]:
```
def HW_caplet_price_HL(data, sigma):
    HW_NPV = caplet_val_HW(data['diff_date'], 0.5, cs, sigma, data['Strike']/100, 0)
    return HW_NPV

# curve_fit with the initial guesses from differential_evolution with the first objective function
popt, pcov, info, msg, ier = curve_fit(HW_caplet_price_HL, data_sigm, data_sigm['NPV_caplet'],
                                       p0=sigma_opt_HL, bounds=(0.0001, 0.5), full_output=True)
s_HL = popt[0]
cov_HL_caplet=pcov
resd_HL = info['fvec']
sse_HL = np.sum(resd_HL**2)
print(s_HL, sse_HL)
```

Caps calibration

Differential_evolution

In [ ]:
```
def HW_obj_fun_cap(x): # first objective function from my thesis for differential_evolution function
    a, sigma = x
    return np.sum(np.square(cap_val_HW(data_sigm_cap['diff_date'], 0.5, cs, sigma,
                        data_sigm_cap['Strike']/100, a)['Price']
                        -cap_val_BS(data_sigm_cap['diff_date'], 0.5, cs,
                        data_sigm_cap['Volatility']/100, data_sigm_cap['Strike']/100, 0.03)['Price']))

def HW_obj_fun_cap_2(x):
    a, sigma = x
    return np.sum(np.square((cap_val_HW(data_sigm_cap['diff_date'], 0.5, cs, sigma,
                        data_sigm_cap['Strike']/100, a)['Price']
                        -cap_val_BS(data_sigm_cap['diff_date'], 0.5, cs,
                        data_sigm_cap['Volatility']/100, data_sigm_cap['Strike']/100, 0.03)['Price'])/cap_val_BS

def HW_obj_fun_cap_3(x):
    a, sigma = x
    return np.sqrt(np.sum(np.square((cap_val_HW(data_sigm_cap['diff_date'], 0.5, cs, sigma,
                        data_sigm_cap['Strike']/100, a)['Price']
                        -cap_val_BS(data_sigm_cap['diff_date'], 0.5, cs,
                        data_sigm_cap['Volatility']/100, data_sigm_cap['Strike']/100, 0.03)['Price'])/cap_val_BS
```

In [ ]:
```
alpha_opt_cap, sigma_opt_cap, fits_cap = opti_choose_OF(HW_obj_fun_cap, bounds)
alpha_opt_cap2, sigma_opt_cap2, fits_cap2 = opti_choose_OF(HW_obj_fun_cap_2, bounds)
alpha_opt_cap3, sigma_opt_cap3, fits_cap3 = opti_choose_OF(HW_obj_fun_cap_3, bounds)
```

Curve_fit

In [ ]:
```
data_sigm_cap['NPV_cap'] = cap_val_BS(data_sigm_cap['diff_date'], 0.5, cs, data_sigm_cap['Volatility']/100, data
def HW_cap_price(data, a, sigma): #setting the price for the application of curve_fit (in meaning of the first
    HW_NPV = cap_val_HW(data['diff_date'], 0.5, cs, sigma, data['Strike']/100, a)['Price']
    return HW_NPV

def HW_cap_price2(data, a, sigma): #setting the price for the application of curve_fit (in meaning of the secon
    HW_NPV = cap_val_HW(data['diff_date'], 0.5, cs, sigma, data['Strike']/100, a)['Price']/data['NPV_cap']
    return HW_NPV
```

In [ ]:
```
est_cap = pd.DataFrame(np.array([[alpha_opt_cap, sigma_opt_cap, fits_cap], [alpha_opt_cap2, sigma_opt_cap2, fits
def est_cf_cap(init_pars, param_bounds):
    popt, pcov, info, msg, ier = curve_fit(HW_cap_price, data_sigm_cap, data_sigm_cap['NPV_cap'], p0=[init_pars
    return (popt, pcov, info)
# curve_fit with the initial guesses from differential_evolution with the first objective function
alpha_opt_cap_cf, sigma_opt_cap_cf = est_cf_cap(est_cap.iloc[0,:], param_bounds)[0]
cov1_cap = est_cf_cap(est_cap.iloc[0,:], param_bounds)[1]
resd_cap = est_cf_cap(est_cap.iloc[0,:], param_bounds)[2]['fvec']
sse_cap = np.sum(resd_cap**2)
```

Diagnostical plots

In [ ]:
```
plt.hist(5000*resd_cap, bins=np.linspace(-5, 5, 50), density=True, alpha=0.5)
plt.plot(np.linspace(-5, 5, 1000), norm.pdf(np.linspace(-5, 5, 1000)), lw=4, alpha=0.7)
plt.show()
```

In [ ]:
```
points_cap = resd_cap + data_sigm_cap['NPV_cap'].to_numpy()
df_res2_cap = pd.DataFrame()
df_res2_cap['fitted values'] = points_cap
df_res2_cap['abs.value of std.residuals'] = np.sqrt(np.abs(resd_cap))
sns.residplot(data=df_res2_cap, x="fitted values", y="abs.value of std.residuals", lowess=True)
```

In [ ]:
```
df_res_cap = pd.DataFrame()
df_res_cap['fitted values'] = points_cap
df_res_cap['residuals'] = resd_cap
sns.residplot(data=df_res_cap, x="fitted values", y="residuals", lowess=True)
```

In [ ]:
```
sm.qqplot(resd_cap, line='45', fit=True)
plt.show
```

In [ ]:
```
smo_cap = lowess(np.sqrt(np.abs(resd_cap)), points_cap, frac=0.6)
fig, pl = plt.subplots(1,2)
sns.residplot(data=df_res_cap, x="fitted values", y="residuals",
              lowess=True, line_kws=dict(color="r"), ax=pl[0])
pl[1].plot(points_cap, np.sqrt(np.abs(resd_cap)), 'o')
pl[1].plot(smo_cap[:, 0], smo_cap[:, 1], color='red')
pl[1].set(xlabel='fitted values', ylabel='abs.value of std.residuals')
plt.tight_layout()
plt.savefig("cap.png")
plt.show()
#combination of the previous plots
```

Curve_fit calibration - continue

In [ ]:
```
def est_cap_cf2(init_pars, param_bounds):
    popt, pcov, info, msg, ier = curve_fit(HW_cap_price2, data_sigm_cap, 1,
                                p0=[init_pars[0], init_pars[1]], bounds=param_bounds, full_output=True)
    return (popt, pcov, info)
alpha_opt2_cap_cf, sigma_opt2_cap_cf = est_cap_cf2(est_cap.iloc[1,:], param_bounds)[0]
cov1_cap2 = est_cap_cf2(est_cap.iloc[1,:], param_bounds)[1]
resd2_cap2 = est_cap_cf2(est_cap.iloc[1,:], param_bounds)[2]['fvec']
sse2_cap2 = np.sum(resd2_cap2**2)
# curve_fit with the initial guesses from differential_evolution with the second objective function
```

Calibration of sigma on caps in Ho-Lee (alpha = 0)

In [ ]:
```
def HL_obj_fun_cap(x): # first objective function from my thesis for differential_evolution function
    return np.sum(np.square(cap_val_HW(data_sigm['diff_date'], 0.5, cs, x, data_sigm['Strike']/100, 0)['Price']

sigma_opt_cap_HL, fits_cap_HL = opti_choose_OF_sigma(HL_obj_fun_cap, [(0.0001, 0.5)])
```

In [ ]:
```
def HW_cap_price_HL(data, sigma):
    HW_NPV = cap_val_HW(data['diff_date'], 0.5, cs, sigma, data['Strike']/100, 0)
    return HW_NPV['Price']

# curve_fit with the initial guesses from differential_evolution with the first objective function
popt_cap, pcov_cap, info_cap, msg_cap, ier_cap = curve_fit(HW_cap_price_HL, data_sigm_cap,
                                                cap_val_BS(data_sigm_cap['diff_date'], 0.5, cs,
                                                data_sigm_cap['Volatility']/100,
                                                data_sigm_cap['Strike']/100, 0.03)['Price'],
                                                p0=sigma_opt_cap_HL, bounds=(0.0001, 0.5), full_output=True)
s_HL_cap = popt_cap[0]
cov_HL_cap=pcov_cap
resd_cap_HL = info_cap['fvec']
sse_cap_HL = np.sum(resd_cap_HL**2)
```

Swaptions calibration

Differential_evolution for alpha

In [ ]:
```
def HW_obj_fun_swaption_impl_vol(x): #objective funtion with the double sum for alpha calibration
    ar=np.array([])
    for i in np.unique(data_swap['Expiry']):
        tenors=data_swap[data_swap['Expiry']==i]['Tenor'].sort_values().values
        expiry_vec=np.full((1,len(tenors)-1), i)

        P_0Ta=np.exp(-cs(expiry_vec)*expiry_vec)
        P_0Tj=np.exp(-cs(expiry_vec+tenors[:-1])*(expiry_vec+tenors[:-1]))
        P_0Tjj=np.exp(-cs(expiry_vec+tenors[1:])*(expiry_vec+tenors[1:]))
        B_j=(1 - np.exp(-x*(expiry_vec+tenors[:-1]-expiry_vec)))/x
        B_jj=(1 - np.exp(-x*(expiry_vec+tenors[1:]-expiry_vec)))/x

        vol_ratio=((P_0Ta - P_0Tj)*B_jj)/((P_0Ta - P_0Tjj)*B_j)
```

```
        impl_vol_ratio=data_swap[data_swap['Expiry']==i]['Rate'].sort_values().values[1:]
                    /data_swap[data_swap['Expiry']==i]['Rate'].sort_values().values[:-1]
        sum_j=(vol_ratio - impl_vol_ratio)**2
        ar=np.append(ar, np.sum(sum_j))

    sum_i=np.sum(ar)
    return sum_i
```

In [ ]:
```
def opti_choose_OF_swaption(func, bounds): #defined function for the application of differential_evolution
    result = differential_evolution(func = func, bounds=bounds, seed=1515)
    a = result.x #calibrated parameter
    f = result.fun # minimized value of the objective function
    return (a, f)
```

In [ ]:
```
alpha_opt_sw_impl_vol, fits_alpha_sw = opti_choose_OF_swaption(HW_obj_fun_swaption_impl_vol, [(0.01, 5)])
```

Differential_evolution for sigma

In [ ]:
```
def HW_obj_fun_swaption(x):
    sigma = x
    return np.sum(np.square(swaption_val_HW_all(data_swap['Expiry'], data_swap['Tenor'], cs, sigma,
                        alpha_opt_sw_impl_vol)['Price']
                        -swaption_val_BS(data_swap['Expiry'], data_swap['Tenor'], cs,
                        data_swap['Rate'])['Price']))
#sigma calibration with the first objective function from my thesis for differential_evolution function
def HW_obj_fun_swaption_2(x):
    sigma = x
    return np.sum(np.square((swaption_val_HW_all(data_swap['Expiry'], data_swap['Tenor'], cs, sigma,
                        alpha_opt_sw_impl_vol)['Price']
                        -swaption_val_BS(data_swap['Expiry'], data_swap['Tenor'], cs,
                        data_swap['Rate'])['Price'])/swaption_val_BS(data_swap['Expiry'], data_swap['Tenor'], cs

def HW_obj_fun_swaption_3(x):
    sigma = x
    return np.sqrt(np.sum(np.square((swaption_val_HW_all(data_swap['Expiry'], data_swap['Tenor'], cs, sigma,
                        alpha_opt_sw_impl_vol)['Price']
                        -swaption_val_BS(data_swap['Expiry'], data_swap['Tenor'], cs,
                        data_swap['Rate'])['Price'])/swaption_val_BS(data_swap['Expiry'], data_swap['Tenor'], cs
```

In [ ]:
```
sigma_opt_sw, fits_sigma_sw = opti_choose_OF_swaption(HW_obj_fun_swaption, [(0.0001, 0.5)])
sigma_opt_sw2, fits_sigma_sw2 = opti_choose_OF_swaption(HW_obj_fun_swaption_2, [(0.0001, 0.5)])
sigma_opt_sw3, fits_sigma_sw3 = opti_choose_OF_swaption(HW_obj_fun_swaption_3, [(0.0001, 0.5)])
```

In [ ]:
```
est_swaption = pd.DataFrame(np.array([[alpha_opt_sw_impl_vol.item(), sigma_opt_sw.item(), fits_sigma_sw],
                            [alpha_opt_sw_impl_vol.item(), sigma_opt_sw2.item(), fits_sigma_sw2],
                            [alpha_opt_sw_impl_vol.item(), sigma_opt_sw3.item(), fits_sigma_sw3]]),
                            columns=['alpha', 'sigma', 'fitness'])
```

Curve_fit for sigma

In [ ]:
```
def HW_swaption_price(data, sigma):
    HW_NPV = swaption_val_HW_all(data['Expiry'], data['Tenor'], cs, sigma, alpha_opt_sw_impl_vol)
    return HW_NPV['Price']
#setting the price for the application of curve_fit (in meaning of the first objective funtion)
def est_cf_swaption(init_pars, param_bounds):
    popt, pcov, info, msg, ier = curve_fit(HW_swaption_price, data_swap, swaption_val_BS(data_swap['Expiry'],
                                        data_swap['Tenor'], cs, data_swap['Rate'])['Price'], p0=init_pars,
                                        bounds=param_bounds, full_output=True)
    return (popt, pcov, info)
```

In [ ]:
```
param_bounds_swaption = ([0.0001], [0.5])

result_cf_swaption = est_cf_swaption(est_swaption['sigma'][0], param_bounds_swaption)
sigma_opt_swaption_cf = result_cf_swaption[0]
cov1_swaption = result_cf_swaption[1]
resd_swaption = result_cf_swaption[2]['fvec']
sse_swaption = np.sum(resd_swaption**2)
# curve_fit with the initial guesses from differential_evolution with the first objective function
```

Diagnostical plot

In [ ]:
```
plt.hist(500*resd_swaption, bins=np.linspace(-5, 5, 50), density=True, alpha=0.5)
plt.plot(np.linspace(-5, 5, 1000), norm.pdf(np.linspace(-5, 5, 1000)), lw=4, alpha=0.7)
plt.show()
```

In [ ]:
```
points_swaption = resd_swaption + swaption_val_BS(data_swap['Expiry'], data_swap['Tenor'], cs,
                                        data_swap['Rate'])['Price'].to_numpy()
df_res_swaption = pd.DataFrame()
```

```
df_res_swaption['fitted'] = points_swaption
df_res_swaption['resd'] = resd_swaption
sns.residplot(data=df_res_swaption, x="fitted", y="resd", lowess=True)
```

In [ ]:
```
df_res2_swaption = pd.DataFrame()
df_res2_swaption['fitted'] = points_swaption
df_res2_swaption['resd'] = np.sqrt(np.abs(resd_swaption))
sns.residplot(data=df_res2_swaption, x="fitted", y="resd", lowess=True)
```

In [ ]:
```
sm.qqplot(resd_swaption, line='45', fit=True)
plt.show
```

Curve_fit calibration – continue

In [ ]:
```
def HW_swaption_price2(data, sigma):
    HW_NPV = swaption_val_HW_all(data['Expiry'], data['Tenor'], cs, sigma, alpha_opt_sw_impl_vol)['Price']
                    /swaption_val_BS(data_swap['Expiry'], data_swap['Tenor'], cs, data_swap['Rate'])['Price']
    return HW_NPV
```

In [ ]:
```
def est_cf_swaption2(init_pars, param_bounds):
    popt, pcov, info, msg, ier = curve_fit(HW_swaption_price2, data_swap, 1, p0=init_pars, bounds=param_bounds,
    return (popt, pcov, info)
# curve_fit with the initial guesses from differential_evolution with the second objective function
result_cf2_swaption2 = est_cf_swaption2(est_swaption['sigma'][1], param_bounds_swaption)
sigma_opt_swaption2_cf2 = result_cf2_swaption2[0]
cov2_swaption2 = result_cf2_swaption2[1]
resd2_swaption2 = result_cf2_swaption2[2]['fvec']
sse2_swaption2 = np.sum(resd_swaption2**2)
```

Backward-looking approach calibration

Differential_evolution

In [ ]:
```
sample_sigma = np.array([])
for i in np.unique(data_hist['Tenor']):
    df = pd.DataFrame()
    df = data_hist[data_hist['Tenor']==i]
    sample_sigma = np.append(sample_sigma, np.var(df['Spot_diff'][1:]))
```

In [ ]:
```
def backward_calibration_obj_fun(x): # objective function for the backward-looking approach
    alpha, sigma = x
    res = np.sum(np.square(((sigma*(1 - np.exp(-alpha * np.unique(data_hist['Tenor_num']))))
                /(alpha * np.unique(data_hist['Tenor_num'])))*(1/np.sqrt(251)) - sample_sigma))
    return res
```

In [ ]:
```
alpha_opt_back, sigma_opt_back, fitness_back = opti_choose_OF(backward_calibration_obj_fun, bounds)
```

Curve_fit

In [ ]:
```
def th_vol(data_hist, alpha, sigma): # theoretical volatilities set for the curve_fit calibration
    vol = ((sigma*(1 - np.exp(-alpha * np.unique(data_hist['Tenor_num']))))
           /(alpha * np.unique(data_hist['Tenor_num'])))*(1/np.sqrt(12))
    return vol

def est_cf_back(init_pars, param_bounds):
    popt, pcov, info, msg, ier = curve_fit(th_vol, data_hist, sample_sigma, p0=init_pars, bounds=param_bounds,
    return (popt, pcov, info)
# curve_fit with the initial guesses from differential_evolution
```

In [ ]:
```
alpha_opt_back_cf, sigma_opt_back_cf = est_cf_back([alpha_opt_back, sigma_opt_back], param_bounds)[0]
cov_back = est_cf_back([alpha_opt_back, sigma_opt_back], param_bounds)[1]
resd_hist = est_cf_back([alpha_opt_back, sigma_opt_back], param_bounds)[2]['fvec']
sse_hist = np.sum(resd_hist**2)
```

Diefferential_evolution

In [ ]:
```
plt.hist(5000*resd_hist, bins=np.linspace(-5, 5, 50), density=True, alpha=0.5)
plt.plot(np.linspace(-5, 5, 1000), norm.pdf(np.linspace(-5, 5, 1000)), lw=4, alpha=0.7)
plt.show()
```

In [ ]:
```
points_hist = resd_hist + sample_sigma
df_res_hist = pd.DataFrame()
df_res_hist['fitted'] = points_hist
df_res_hist['resd'] = resd_hist
sns.residplot(data=df_res_hist, x="fitted", y="resd", lowess=True)
```

In [ ]:
```
df_res2_hist = pd.DataFrame()
df_res2_hist['fitted'] = points_hist
df_res2_hist['resd'] = np.sqrt(np.abs(resd_hist))
sns.residplot(data=df_res2_hist, x="fitted", y="resd", lowess=True)
```

Simulations

In [ ]:
```
cs2 = cs.derivative(2) # the second derivative of the natural cubic spline interpolation of the zero curve
def theta (t, alpha, sigma): # defined function for creating theta(t) function
    if alpha == 0:
        return cs2(t) * t + 2 * cs1(t) + t * sigma**2
    else:
        return cs2(t) * t + 2 * cs1(t) + alpha * t * cs1(t) + alpha * cs(t)
        + (sigma**2/(2 * alpha)) * (1 - np.exp(-2 * alpha * t))
```

In [ ]:
```
empt = np.array([])
for x in data_ir['MaturityDate'].astype(str).tolist():
    empt = np.append(empt,
            ql.Date(int(x.split('-')[2]),
                    int(x.split('-')[1]),
                    int(x.split('-')[0])))
data_ir['NewMaturity'] = empt
# setting the format of time for simulation functions
```

Caplet simulations

In [ ]:
```
curve = ql.NaturalCubicZeroCurve(data_ir['NewMaturity'].tolist(),(data_ir['ZeroRate']/100).tolist(),
                        ql.ActualActual(ql.ActualActual.ISDA), ql.CzechRepublic())
# natural cubic spline of the zero curve for simulation functions
curve2 = ql.YieldTermStructureHandle(curve)
# handling the term structure of the zero curve for the Hull-White process simulation
process_caplet = ql.HullWhiteProcess(curve2, est_caplet['alpha'][0], est_caplet['sigma'][0])
```

In [ ]:
```
rng = ql.GaussianRandomSequenceGenerator(ql.UniformRandomSequenceGenerator(360, ql.UniformRandomGenerator()))
seq_caplet = ql.GaussianPathGenerator(process_caplet, 30, 360, rng, False)
# generating random sequence of points for path generating
```

In [ ]:
```
def generate_paths(num_paths, timestep, seq):
    arr = np.zeros((num_paths, timestep+1))
    for i in range(num_paths):
        sample_path = seq.next()
        path = sample_path.value()
        time = [path.time(j) for j in range(len(path))]
        value = [path[j] for j in range(len(path))]
        arr[i, :] = np.array(value)
    return np.array(time), arr
# defined function for generating paths of the Hull-White process
```

In [ ]:
```
num_paths = 1000 # number of simulations
time_caplet, paths_caplet = generate_paths(num_paths, 360, seq_caplet)
for i in range(num_paths-1):
    plt.plot(time_caplet, paths_caplet[i, :], lw=0.8, alpha=0.6) # plotting each simulated path
plt.plot(time_caplet, paths_caplet[num_paths-1, :], "g", lw=3) # one simulated path is highlighted
plt.plot(np.linspace(0, 30, 100),
    [process_caplet.drift(t, 0)/est_caplet['alpha'][0] for t in np.linspace(0, 30, 100)], 'b--', lw = 1.5, alpha
# plotting the long-term mean theta(t)/alpha
avg_caplet = [np.mean(paths_caplet[:, i]) for i in range(360+1)]
plt.plot(time_caplet, avg_caplet, "r", lw=3) # plotting the sample mean line
plt.title("Hull-White Short Rate Simulation")
plt.ylim((-0.15, 0.35))
plt.savefig("caplet sim.png")
plt.show()
```

Caplet simulations in the Ho-Lee model

In [ ]:
```
process_caplet_HL = ql.HullWhiteProcess(curve2, 0, sigma_opt_HL.item())
seq_caplet_HL = ql.GaussianPathGenerator(process_caplet_HL, 30, 360, rng, False)

time_caplet_HL, paths_caplet_HL = generate_paths(num_paths, 360, seq_caplet_HL)
for i in range(num_paths):
    plt.plot(time_caplet_HL, paths_caplet_HL[i, :], lw=0.8, alpha=0.6)
plt.plot(np.linspace(0, 30, 100), theta(np.linspace(0, 30, 100), 0, sigma_opt_HL.item()), 'b--',
         lw = 1.5, alpha=0.8)
# plotting theta(t) function for the Ho-Lee model
avg_caplet_HL = [np.mean(paths_caplet_HL[:, i]) for i in range(360+1)]
plt.plot(time_caplet_HL, avg_caplet_HL, "r", lw=3)
plt.title("Ho-Lee Short Rate Simulation")
```

```
plt.ylim((-0.15, 0.35))
plt.savefig("caplet HL sim.png")
plt.show()
```

Cap simulations

In [ ]:
```
process_cap = ql.HullWhiteProcess(curve2, est_cap['alpha'][0], est_cap['sigma'][0])
seq_cap = ql.GaussianPathGenerator(process_cap, 30, 360, rng, False)

time_cap, paths_cap = generate_paths(num_paths, 360, seq_cap)
for i in range(num_paths-1):
    plt.plot(time_cap, paths_cap[i, :], lw=0.8, alpha=0.6)
plt.plot(time_cap, paths_cap[num_paths-1, :], "g", lw=3)
plt.plot(np.linspace(0, 30, 100), [process_cap.drift(t, 0)
        /est_cap['alpha'][0] for t in np.linspace(0, 30, 100)], 'b--', lw = 1.5, alpha=0.8)
avg_cap = [np.mean(paths_cap[:, i]) for i in range(360+1)]
plt.plot(time_cap, avg_cap, "r", lw=3)
plt.title("Hull-White Short Rate Simulation")
plt.ylim((-0.3, 0.4))
plt.savefig("cap sim.png")
plt.show()
```

In [ ]:
```
# alternative plot with added lines
for i in range(num_paths-1):
    plt.plot(time_cap, paths_cap[i, :], lw=0.8, alpha=0.6)
plt.plot(time_cap, paths_cap[num_paths-1, :], "g", lw=3)
plt.plot(np.linspace(0, 30, 100), [process_cap.drift(t, 0)
        /est_cap['alpha'][0] for t in np.linspace(0, 30, 100)], 'b--', lw = 1.5, alpha=0.8)
high_q_cap = [np.quantile(paths_cap[:, i], 0.975) for i in range(360+1)]
low_q_cap = [np.quantile(paths_cap[:, i], 0.025) for i in range(360+1)]
plt.plot(time_cap, avg_cap, "r", lw=3)
plt.plot(time_cap, high_q_cap, "y", lw=1.5) # sample 97,5 % quantile line
plt.plot(time_cap, low_q_cap, "y", lw=1.5) # sample 2,5 % quantile line
plt.title("Hull-White Short Rate Simulation")
plt.ylim((-0.3, 0.4))
plt.savefig("cap sim.png")
plt.show()
```

Swaption simulations

In [ ]:
```
process_sw = ql.HullWhiteProcess(curve2, est_swaption['alpha'][0], est_swaption['sigma'][0])
seq_sw = ql.GaussianPathGenerator(process_sw, 30, 360, rng, False)

time_sw, paths_sw = generate_paths(num_paths, 360, seq_sw)
for i in range(num_paths-1):
    plt.plot(time_sw, paths_sw[i, :], lw=0.8, alpha=0.6)
plt.plot(time_sw, paths_sw[num_paths-1, :], "g", lw=3)
plt.plot(np.linspace(0, 30, 100), [process_sw.drift(t, 0)
        /est_swaption['alpha'][0] for t in np.linspace(0, 30, 100)], 'b--', lw = 1.5, alpha=0.8)
avg_sw = [np.mean(paths_sw[:, i]) for i in range(360+1)]
plt.plot(time_sw, avg_sw, "r", lw=3)
plt.title("Hull-White Short Rate Simulation")
plt.ylim((-0.05, 0.2))
plt.savefig("swaption sim.png")
plt.show()
```

In [ ]:
```
# alternative plot with quantiles lines
for i in range(num_paths-1):
    plt.plot(time_sw, paths_sw[i, :], lw=0.8, alpha=0.6)
plt.plot(time_sw, paths_sw[num_paths-1, :], "g", lw=3)
plt.plot(np.linspace(0, 30, 100), [process_sw.drift(t, 0)
        /est_swaption['alpha'][0] for t in np.linspace(0, 30, 100)], 'b--', lw = 1.5, alpha=0.8)
high_q_sw = [np.quantile(paths_sw[:, i], 0.975) for i in range(360+1)]
low_q_sw = [np.quantile(paths_sw[:, i], 0.025) for i in range(360+1)]
plt.plot(time_sw, avg_sw, "r", lw=3)
plt.plot(time_sw, high_q_sw, "y", lw=1.5)
plt.plot(time_sw, low_q_sw, "y", lw=1.5)
plt.axhline(y=0, color="k", linestyle="--", lw=0.8, alpha=0.8)
plt.title("Hull-White Short Rate Simulation")
plt.ylim((-0.05, 0.2))
plt.savefig("swaption sim.png")
plt.show()
```

Historical simulations

In [ ]:
```
process_hist = ql.HullWhiteProcess(curve2, alpha_opt_back, sigma_opt_back)
seq_hist = ql.GaussianPathGenerator(process_hist, 30, 360, rng, False)

time_hist, paths_hist = generate_paths(num_paths, 360, seq_hist)
```

```
for i in range(num_paths):
    plt.plot(time_hist, paths_hist[i, :], lw=0.8, alpha=0.6)
plt.plot(np.linspace(0, 30, 100), [process_hist.drift(t, 0)
        /alpha_opt_back for t in np.linspace(0, 30, 100)], 'b--',
        lw = 1.5, alpha = 0.8)
avg_hist = [np.mean(paths_hist[:, i]) for i in range(360+1)]
plt.plot(time_hist, avg_hist, "r", lw=3)
plt.title("Hull-White Short Rate Simulation")
plt.ylim((-0.15, 0.35))
plt.savefig("hist sim.png")
plt.show()
```