Libraries

Nodari Alessandro & Proserpio Lorenzo

Utils

ImpliedDrift.py

```python
import numpy as np
import pandas as pd
from scipy.interpolate import CubicSpline

dates = np.array(["23-01-23","24-01-23","25-01-23","26-01-23",
"27-01-23","30-01-23","06-02-23","13-02-23","21-02-23"])
data = pd.read_csv("ratesOIS.csv")
tenor = np.array(data.TENOR)
Forw = pd.read_csv("forw.csv")
spot = np.array(pd.read_csv("spot.csv").Spot)
TENOR = pd.read_csv("tenor.csv")


def r(x, index = 0):
    rates = np.array(data[dates[index]])/100
    cs = CubicSpline(tenor, rates)
    return cs(x)

def drift(x, index = 0):
    S0 = spot[index]
    F = np.array(Forw[dates[index]]).flatten()
    Tenor = np.array(TENOR[dates[index]]).flatten()
    d = -np.log(S0/F).flatten()/Tenor
    cs = CubicSpline(Tenor, d)
    return cs(x)

def q(x, index = 0):
    return r(x, index) - drift(x, index)
```

BlackScholes.py

```python
import numpy as np
from scipy.stats import norm

def BSCall(S0, K, T, r, q, sigma):

    # Price of a call under Black&Scholes

    # S0: spot price
    # K: strike
```

```
    # T: years to expiration
    # r: risk free rate (1 = 100%)
    # q: annual yield
    # sigma: volatility (1 = 100%)


    sig = sigma*np.sqrt(T)
    d1 = (np.log(S0/K) + (r-q)*T)/sig + sig/2.
    d2 = d1 - sig

    return S0*norm.cdf(d1) - K*np.exp(-(r-q)*T)*norm.cdf(d2)

def BSPut(S0, K, T, r, q, sigma):

    # Price of a put under Black&Scholes

    # S0: spot price
    # K: strike
    # T: years to expiration
    # r: risk free rate (1 = 100%)
    # q: annual yield
    # sigma: volatility (1 = 100%)

    return BSCall(S0, K, T, r, q, sigma) + K*np.exp(-(r-q)*T) - S0

def BSImpliedVol(S0, K, T, r, q, P, Option_type = 1, toll = 1e-10):

    # Calculate implied volatility from prices using bisection

    # NOTE: All the parameters can be np.array(), except for P that MUST be a np.array().

    # S0: spot price
    # K: strike
    # T: years to expiration
    # r: risk free rate (1 = 100%)
    # q: annual yield
    # P: prices
    # Option_type: 1 for calls, 0 for puts
    # toll: error in norm 1

    if Option_type:
        BSFormula = np.vectorize(BSCall)
    else:
        BSFormula = np.vectorize(BSPut)

    N = P.shape[0]
    sigma_low = 1e-10*np.ones(N)
    sigma_high = 10*np.ones(N)

    P_low = BSFormula(S0, K, T, r, q, sigma_low)
    P_high = BSFormula(S0, K, T, r, q, sigma_high)
    sigma = (sigma_low + sigma_high)/2.

    while np.sum(P_high - P_low) > toll:
        sigma = (sigma_low + sigma_high)/2.
        P_mean = BSFormula(S0, K, T, r, q, sigma)
```

```
        P_low += (P_mean < P)*(P_mean - P_low)
        sigma_low += (P_mean < P)*(sigma - sigma_low)
        P_high += (P_mean >= P)*(P_mean - P_high)
        sigma_high += (P_mean >= P)*(sigma - sigma_high)

    return sigma
```

variance_curve.py

```
import numpy as np

Z1 = np.array([0.23934445564954748, 0.2370172145384514,
    0.23319383855246545, 0.22779527372712297, 0.22410506177795986,
    0.22796530521676028, 0.22992033119402192, 0.23360387896928214,
    0.23923251959598327])
Z2 = np.array([0.235591674028804 1, 0.23547872334450043,
    0.2278527332290963, 0.2493545607795239, 0.2684664268847795,
    0.16758709131144714, 0.23526774429356268, 0.16354991037070024,
    0.09001896821824877])
Z3 = np.array([2.3126258447474375, 2.077549483492911,
    2.1200577204482105, 2.637109433927137, 2.9677374658205307,
    2.385086643216494, 3.064763505035709, 2.3367542064926443,
    2.114562825304444])

# Compute the initial forward variance curve at a given time t
# using the Gompertz function with precomputed parameters

def Gompertz(t, index = 0):
    z1 = Z1[index]; z2 = Z2[index]; z3 = Z3[index];
    return z1 * np.exp(-z2 * np.exp(-z3 * t))

def variance_curve(t, index = 0):
    z1 = Z1[index]; z2 = Z2[index]; z3 = Z3[index];
    return (z1 * np.exp(-z2 * np.exp(-z3 * t)))**2 + 2*t*z1**2*z2*
    z3*np.exp(-2*z2*np.exp(-z3*t)-z3*t)
```

Heston

Heston.py

```
import numpy as np
import QuantLib as ql
import scipy.integrate
from scipy.special import ive

################### PUT-CALL PARITY ############################

def put_call_parity(put, S0, strike, r, q, tau):
    # Standard put_call_parity
    return put + S0*np.exp(-q*tau) - strike*np.exp(-r*tau)

def call_put_parity(call, S0, strike, r, q, tau):
    return call - S0*np.exp(-q*tau) + strike*np.exp(-r*tau)



```

```
#################### Analytic Heston ###########################

def phi_hest(u, tau, sigma_0, kappa, eta, theta, rho):

    # Compute the characteristic function for Heston Model

    # u: argument of the function (where you want to evaluate)
    # tau: time to expiration
    # sigma_0, kappa, eta, theta, rho: Heston parameters

    alpha_hat = -0.5 * u * (u + 1j)
    beta = kappa - 1j * u * theta * rho
    gamma = 0.5 * theta ** 2
    d = np.sqrt(beta**2 - 4 * alpha_hat * gamma)
    g = (beta - d) / (beta + d)
    h = np.exp(-d*tau)
    A_ = (beta - d)*tau - 2*np.log((g*h-1) / (g-1))
    A = kappa * eta / (theta**2) * A_
    B = (beta - d) / (theta**2) * (1 - h) / (1 - g*h)
    return np.exp(A + B * sigma_0)

def integral(x, tau, sigma_0, kappa, eta, theta, rho):

    # Pseudo-probabilities

    # x: log-prices discounted

    integrand = (lambda u: np.real(np.exp((1j*u + 0.5)*x) * \
                                   phi_hest(u - 0.5j, tau, sigma_0, kappa, eta, theta, rho)) / \
                (u**2 + 0.25))

    i, err = scipy.integrate.quad_vec(integrand, 0, np.inf)

    return i

def analytic_hest(S0, strikes, tau, r, q,  kappa, theta, rho, eta, sigma_0, options_type):

    # Pricing of vanilla options under analytic Heston

    a = np.log(S0/strikes) + (r-q)*tau
    i = integral(a, tau, sigma_0, kappa, eta, theta, rho)

    out = S0 * np.exp(-q*tau) - strikes * np.exp(-r*tau)/np.pi * i
    out = np.array([out]).flatten()

    for k in range(len(out)):
        if options_type[k] == 0:
            out[k] = call_put_parity(out[k], S0, strikes[k], r, q, tau)

    return out

#################### COS METHOD Le Floch ########################

```

```
def phi_hest_0(u, tau, r, q, sigma_0, kappa, eta, theta, rho):

    # Compute the characteristic function for Heston Model with
    log_asset = 0

    # u: argument of the function (where you want to evaluate)
    # r: risk-free-rate
    # q: annual percentage yield
    # tau: time to expiration
    # sigma_0, kappa, eta, theta, rho: Heston parameters

    beta = (kappa - 1j*rho*u*theta)
    d = np.sqrt(beta**2 + (theta**2)*(1j*u+u**2))
    r_minus = (beta - d)
    g = r_minus/(beta + d)
    aux = np.exp(-d*tau)

    term_1 = sigma_0/(theta**2) *  ((1-aux)/(1-g*aux)) * r_minus
    term_2 = kappa*eta/(theta**2) * (tau*r_minus - 2*np.log((1-g*
    aux) / (1-g)))
    term_3 = 1j*(r-q)*u*tau

    return np.exp(term_1)*np.exp(term_2)*np.exp(term_3)

def chi_k(k, c, d, a, b):
    # Auxiliary function for U_k

    aux_1 = k*np.pi/(b-a)
    aux_2 = np.exp(d)
    aux_3 = np.exp(c)

    return  (np.cos(aux_1*(d-a))*aux_2 - \
            aux_3 + \
            aux_1*np.sin(aux_1*(d-a))*aux_2) / (1+aux_1**2)

def psi_k(k, c, d, a, b):
    # Auxiliary function for U_k

    if k == 0:
        return d - c

    aux = k*np.pi/(b-a)
    return np.sin(aux*(d-a)) / aux

def U_k_put(k, a, b):
    # Auxiliary for cos_method

    return 2./(b-a) * (psi_k(k, a, 0, a, b) - chi_k(k, a, 0, a, b))

def optimal_ab(r, q, tau, sigma_0, kappa, eta, theta, rho, L = 12):
    # Compute the optimal interval for the truncation
    aux = np.exp(-kappa* tau)
    c1 =  (r-q)*tau - sigma_0 * tau / 2

    c2 = (sigma_0) / (4*kappa**3) * (4*kappa**2*(1+(rho*theta*tau
    -1)*aux) \
                                    + kappa*(4*rho*theta*(aux-1)\
```

```python
                                            -2*theta**2*tau*aux) \
                                   +theta**2*(1-aux*aux)) \
        + eta/(8*kappa**3)*(8*kappa**3*tau - 8*kappa**2*(1+ rho*theta*tau +(rho*theta*tau-1)*aux)\
                           + 2*kappa*((1+2*aux)*theta**2*tau+8*(1-aux)*rho*theta)\
                           + theta**2*(aux*aux+4*aux-5))


    return c1 - 12*np.sqrt(np.abs(c2)), c1 + 12*np.sqrt(np.abs(c2))


def precomputed_terms(r, q, tau, sigma_0, kappa, eta, theta, rho, L, N):
    # Auxiliary term precomputed

    a,b = optimal_ab(r, q, tau, sigma_0, kappa, eta, theta, rho, L)
    aux = np.pi/(b-a)
    out = np.zeros(N-1)

    for k in range(1,N):
        out[k-1] = np.real(np.exp(-1j*k*a*aux)*\
                     phi_hest_0(k*aux, tau, r, q, sigma_0, kappa, eta, theta, rho))

    return out, a, b

def V_k_put(k, a, b, S0, K, z):
    # V_k coefficients for puts

    return 2./(b-a)*(K*psi_k(k, a, z, a, b) - S0*chi_k(k, a, z, a, b))

def cos_method_Heston_LF(precomp_term, a, b, tau, r, q, sigma_0, kappa, eta, theta, rho, S0,\
                     strikes, N, options_type, L=12):
    # Cosine Fourier Expansion for evaluating vanilla options under Heston using LeFloch correction
    # Should be better for deep otm options.

    # precomp_term: precomputed terms from the function precomputed_terms
    # a,b: extremes of the interval to approximate
    # tau: time to expiration (annualized) (must be a number)
    # r: risk-free-rate
    # q: yield
    # sigma_0, kappa, eta, theta, rho: Heston parameters
    # S0: initial spot price
    # strikes: np.array of strikes
    # N: number of terms of the truncated expansion
    # options_type: binary np.array (1 for calls, 0 for puts)
    # L: truncation level

    z = np.log(strikes/S0)

    out = 0.5 * np.real(phi_hest_0(0, tau, r, q, sigma_0, kappa, eta, theta, rho))*\
```

```
            V_k_put(0, a, b, S0, strikes, z)

    for k in range(1,N):
        out = out + precomp_term[k-1]*V_k_put(k, a, b, S0, strikes,
     z)

    D = np.exp(-r*tau)
    out = out*D

    for k in range(len(strikes)):
        if options_type[k] == 1:
            out[k] = put_call_parity(out[k], S0, strikes[k], r, q,
    tau)

    return out

#######################Calibration
    #####################################

def setup_model(_yield_ts, _dividend_ts, _spot,
                init_condition):
    # Setup Heston model object

    # _yield_ts: Term Structure for yield (QuantLib object)
    # _dividend_ts: Term Structure for dividend_ts (QuantLib object
    )
    # init_condition: eta, kappa, theta, rho, sigma_0

    eta, kappa, theta, rho, sigma_0 = init_condition
    process = ql.HestonProcess(_yield_ts, _dividend_ts,
                           ql.QuoteHandle(ql.SimpleQuote(_spot)),
                           sigma_0, kappa, eta, theta, rho)
    model = ql.HestonModel(process)
    engine = ql.AnalyticHestonEngine(model)
    return model, engine

def setup_helpers(engine, expiration_dates, strikes,
                  data, ref_date, spot, yield_ts,
                  dividend_ts, calendar):
    # Helpers for Heston Calibration

    # engine: Heston.setup_model output
    # expiration_dates: maturities
    # data: IV market data
    # ref_date: date for the calculation
    # yield_ts: Term Structure for yield (QuantLib object)
    # dividend_ts: Term Structure for dividend_ts (QuantLib object)
    # calendar: type of calendar for calculations

    heston_helpers = []
    grid_data = []
    for i, date in enumerate(expiration_dates):
        for j, s in enumerate(strikes):
            t = (date - ref_date )
            p = ql.Period(t, ql.Days)
            vols = data[i][j]
            helper = ql.HestonModelHelper(
```

```
                p, calendar, spot, s,
                ql.QuoteHandle(ql.SimpleQuote(vols)),
                yield_ts, dividend_ts)
            helper.setPricingEngine(engine)
            heston_helpers.append(helper)
            grid_data.append((date, s))
    return heston_helpers, grid_data

def cost_function_generator(model, helpers, norm=False):
    # Define cost function for the calibration (usually Mean Square
     Error)

    def cost_function(params):
        params_ = ql.Array(list(params))
        model.setParams(params_)
        error = [h.calibrationError() for h in helpers]
        if norm:
            return np.sqrt(np.sum(np.abs(error)))
        else:
            return error
    return cost_function

######################Simulation Heston
    ################################

def create_totems(base, start, end):
    # create the grid

    totems = np.ones(end-start+2)
    index = 1
    for j in range(start, end+1):
        totems[index] = base**j
        index += 1

    totems[0] = 0
    return totems

def calc_nu_bar(kappa, eta, theta):
    # compute v bar
    return 4*kappa*eta/theta**2

def x2_exp_var(nu_bar, kappa, theta, dt):
    # compute E[X_2] and Var[X_2]

    aux = kappa*dt/2.
    c1 = np.cosh(aux)/np.sinh(aux)
    c2 = (1./np.sinh(aux))**2
    exp_x2 = nu_bar*theta**2*((-2.+kappa*dt*c1)/(4*kappa**2))
    var_x2 = nu_bar*theta**4*((-8.+2*kappa*dt*c1+\
                              kappa**2*dt**2*c2)/(8*kappa**4))
    return exp_x2, var_x2

def Z_exp_var(nu_bar, exp_x2, var_x2):
    # compute E[Z] and Var[Z]

    return 4*exp_x2/nu_bar, 4*var_x2/nu_bar

```

```
def xi_exp(nu_bar, kappa, theta, dt, totem):
    # compute E[\Xi] and E[\Xi^2]

    z = 2*kappa*np.sqrt(totem) / (theta**2*np.sinh(kappa*dt/2.))
    iv_pre = ive(nu_bar/2.-1., z)
    exp_xi = (z*ive(nu_bar/2.,z))/(2*iv_pre)
    exp_xi2 = exp_xi + (z**2*ive(nu_bar/2.+1,z))/(4.*iv_pre)

    return exp_xi, exp_xi2

def create_caches(base, start, end, kappa, eta, theta, dt):
    # precompute the caches for IV*

    totems = create_totems(base, start, end)
    caches_exp = np.zeros(end-start+2)
    caches_var = np.zeros(end-start+2)
    nu_bar = calc_nu_bar(kappa, eta, theta)
    exp_x2, var_x2 = x2_exp_var(nu_bar, kappa, theta, dt)
    exp_Z, var_Z = Z_exp_var(nu_bar, exp_x2, var_x2)

    for j in range(1,end-start+2):
        exp_xi, exp_xi2 = xi_exp(nu_bar, kappa, theta, dt, totems[j])
        caches_exp[j] = exp_x2 + exp_xi*exp_Z
        caches_var[j] = var_x2 + exp_xi*var_Z + \
                        (exp_xi2-exp_xi**2)*exp_Z**2

    caches_exp[0] = exp_x2
    caches_var[0] = var_x2
    return totems, caches_exp, caches_var

def x1_exp_var(kappa, theta, dt, vt, vT):
    # compute E[X_1] and Var[X_1]

    aux = kappa*dt/2.
    c1 = np.cosh(aux)/np.sinh(aux)
    c2 = (1./np.sinh(aux))**2

    exp_x1 = (vt + vT)*(c1/kappa - dt*c2/2)
    var_x1 = (vt + vT)*theta**2*(c1/kappa**3 + dt*c2/(2*kappa**2) \
                            - dt**2*c1*c2/(2*kappa))

    return exp_x1, var_x1

def lin_interp(vtvT, totems, caches_exp, caches_var):
    # compute linear interpolation for value not in caches

    exp_int = np.interp(vtvT, totems, caches_exp)
    var_int = np.interp(vtvT, totems, caches_var)
    return exp_int, var_int

def sample_vT(vt, dt, kappa, theta, nu_bar):
    # sample vT from a noncentral chisquare (given vt)

    aux = (theta**2*(1-np.exp(-kappa*dt)))/(4*kappa)
    n = np.exp(-kappa*dt)/aux *vt
    return np.random.noncentral_chisquare(nu_bar, n)*aux
```

```python

def generate_path(S0, T, dt, kappa, eta, theta, rho, r, q, sigma_0,
     totems, caches_exp, caches_var):
    # This function generate one path for the Heston model using
    the Gamma Approx algorithm

    # T: final time
    # r: risk-free-rate
    # q: yield
    # sigma_0, kappa, eta, theta, rho: Heston parameters
    # S0: initial spot price
    # dt: temporal step
    # totems, caches_exp, caches_var: precomputed grid, caches for
    expectation and caches for var

    t = dt
    index = 0
    vt = sigma_0
    xt = np.log(S0)

    path = np.zeros(int(np.ceil(T/dt))+1)
    path[0] = S0

    variance = np.zeros(int(np.ceil(T/dt))+1)
    variance[0] = vt

    nu_bar = calc_nu_bar(kappa, eta, theta)

    while t < T:
        vT = sample_vT(vt, dt, kappa, theta, nu_bar)

        exp_int, var_int = lin_interp(vt*vT, totems, caches_exp,
    caches_var)

        exp_x1, var_x1 = x1_exp_var(kappa, theta, dt, vt, vT)
        exp_int += exp_x1
        var_int += var_x1

        gamma_t = var_int/exp_int
        gamma_k = exp_int**2/var_int

        iv_t = np.random.gamma(gamma_k, gamma_t)
        z = np.random.normal()

        xt += (r-q)*dt + (- 0.5 + kappa*rho/theta)*iv_t + \
              rho/theta*(vT-vt-kappa*eta*dt) + \
              z*np.sqrt(1-rho**2)*np.sqrt(iv_t)

        index += 1
        path[index] = np.exp(xt)
        vt = vT
        variance[index] = vt
        t += dt

    return path[:-1], variance[:-1]
```

Rough Heston

rHeston.py

```
import numpy as np
import ImpliedDrift
import Heston
import BlackScholes
import scipy.integrate

from variance_curve import variance_curve, Gompertz
from scipy.special import gamma
from scipy.interpolate import CubicSpline
from scipy.stats import norm

du = 1e-4
GRID = np.linspace(0,10,int(1./du))


###################### Pade rHeston ################################

def Pade33(u, t, H, rho, theta):
    alpha = H + 0.5

    aa = np.sqrt(u * (u + (0+1j)) - rho**2 * u**2)
    rm = -(0+1j) * rho * u - aa
    rp = -(0+1j) * rho * u + aa

    gamma1 = gamma(1+alpha)
    gamma2 = gamma(1+2*alpha)
    gammam1 = gamma(1-alpha)
    gammam2 = gamma(1-2*alpha)

    b1 = -u*(u+1j)/(2 * gamma1)
    b2 = (1-u*1j) * u**2 * rho/(2* gamma2)
    b3 = gamma2/gamma(1+3*alpha) * \
        (u**2*(1j+u)**2/(8*gamma1**2)+(u+1j)*u**3*rho**2/(2*gamma2))

    g0 = rm
    g1 = -rm/(aa*gammam1)
    g2 = rm/aa**2/gammam2 * \
         (1 + rm/(2*aa)*gammam2/gammam1**2)

    den = g0**3 +2*b1*g0*g1-b2*g1**2+b1**2*g2+b2*g0*g2

    p1 = b1
    p2 = (b1**2*g0**2 + b2*g0**3 + b1**3*g1 + b1*b2*g0*g1 - \
          b2**2*g1**2 +b1*b3*g1**2 +b2**2*g0*g2 - b1*b3*g0*g2)/den
    q1 = (b1*g0**2 + b1**2*g1 - b2*g0*g1 + b3*g1**2 - b1*b2*g2 -b3*g0*g2)/den
    q2 = (b1**2*g0 + b2*g0**2 - b1*b2*g1 - b3*g0*g1 + b2**2*g2 - b1*b3*g2)/den
    q3 = (b1**3 + 2*b1*b2*g0 + b3*g0**2 -b2**2*g1 +b1*b3*g1 )/den
    p3 = g0*q3

```

```
    y = t**alpha

    return (p1*y + p2*y**2 + p3*y**3)/(1 + q1*y + q2*y**2 + q3*y**3)

def DH_Pade33(u, x, H, rho, theta):
    alpha = H + 0.5

    aa = np.sqrt(u * (u + 1j) - rho**2 * u**2)
    rm = -1j * rho * u - aa
    rp = -1j * rho * u + aa

    b1 = -u*(u+1j)/(2 * gamma(1+alpha))
    b2 = (1-u*1j) * u**2 * rho/(2* gamma(1+2*alpha))
    b3 = gamma(1+2*alpha)/gamma(1+3*alpha) * \
              (u**2*(1j+u)**2/(8*gamma(1+alpha)**2)+(u+1j)*u**3*rho**2/(2*gamma(1+2*alpha)))

    g0 = rm
    g1 = -rm/(aa*gamma(1-alpha))
    g2 = rm/aa**2/gamma(1-2*alpha) * (1 + rm/(2*aa)*gamma(1-2*alpha)/gamma(1-alpha)**2)

    den = g0**3 +2*b1*g0*g1-b2*g1**2+b1**2*g2+b2*g0*g2

    p1 = b1
    p2 = (b1**2*g0**2 + b2*g0**3 + b1**3*g1 + b1*b2*g0*g1 - b2**2*g1**2 +b1*b3*g1**2 +b2**2*g0*g2 - b1*b3*g0*g2)/den
    q1 = (b1*g0**2 + b1**2*g1 - b2*g0*g1 + b3*g1**2 - b1*b2*g2 -b3*g0*g2)/den
    q2 = (b1**2*g0 + b2*g0**2 - b1*b2*g1 - b3*g0*g1 + b2**2*g2 - b1*b3*g2)/den
    q3 = (b1**3 + 2*b1*b2*g0 + b3*g0**2 -b2**2*g1 +b1*b3*g1 )/den
    p3 = g0*q3

    y = x**alpha

    hpade = (p1*y + p2*y**2 + p3*y**3)/(1 + q1*y + q2*y**2 + q3*y**3)

    res = 0.5*(hpade-rm)*(hpade-rp)

    return res

def phi_rhest(u, t, H, rho, theta):
    if u == 0:
        return 1.

    N = int(t*365)
    alpha = H + 0.5
    dt = t/N
    tj = np.linspace(0,N,N+1,endpoint = True)*dt

    x = theta**(1./alpha)*tj
    xi = np.flip(variance_curve(tj))

    aux = DH_Pade33(u, x, H, rho, theta)
```

```python

    return np.exp(np.matmul(aux,xi)*dt)

def DH_Pade33_vec(u, x, H, rho, theta):
    alpha = H + 0.5

    aa = np.sqrt(u * (u + 1j) - rho**2 * u**2)
    rm = -1j * rho * u - aa
    rp = -1j * rho * u + aa

    b1 = -u*(u+1j)/(2 * gamma(1+alpha))
    b2 = (1-u*1j) * u**2 * rho/(2* gamma(1+2*alpha))
    b3 = gamma(1+2*alpha)/gamma(1+3*alpha) * \
             (u**2*(1j+u)**2/(8*gamma(1+alpha)**2)+(u+1j)*u**3*rho**2/(2*gamma(1+2*alpha)))

    g0 = rm
    g1 = -rm/(aa*gamma(1-alpha))
    g2 = rm/aa**2/gamma(1-2*alpha) * (1 + rm/(2*aa)*gamma(1-2*alpha)/gamma(1-alpha)**2)

    den = g0**3 +2*b1*g0*g1-b2*g1**2+b1**2*g2+b2*g0*g2

    p1 = b1
    p2 = (b1**2*g0**2 + b2*g0**3 + b1**3*g1 + b1*b2*g0*g1 - b2**2*g1**2 +b1*b3*g1**2 +b2**2*g0*g2 - b1*b3*g0*g2)/den
    q1 = (b1*g0**2 + b1**2*g1 - b2*g0*g1 + b3*g1**2 - b1*b2*g2 -b3*g0*g2)/den
    q2 = (b1**2*g0 + b2*g0**2 - b1*b2*g1 - b3*g0*g1 + b2**2*g2 - b1*b3*g2)/den
    q3 = (b1**3 + 2*b1*b2*g0 + b3*g0**2 -b2**2*g1 +b1*b3*g1 )/den
    p3 = g0*q3

    y = x**alpha
    y2 = y**2
    y3 = y**3

    size_ = len(u)
    Y = np.tile(y, (size_,1)).transpose()
    Y2 = np.tile(y2, (size_,1)).transpose()
    Y3 = np.tile(y3, (size_,1)).transpose()

    hpade = (Y*p1 + Y2*p2 + Y3*p3)/(1 + Y*q1 + Y2*q2 + Y3*q3)

    res = 0.5*(hpade-rm)*(hpade-rp)

    return res

def phi_rhest_vec(u, t, H, rho, theta, N = 1000):

    mask = (u == 0)

    alpha = H + 0.5
    dt = t/N
    tj = np.linspace(0,N,N+1,endpoint = True)*dt

    x = theta**(1./alpha)*tj
```

```python
    xi = np.flip(variance_curve(tj))

    res = np.zeros(len(u), dtype = complex)

    if mask.any():
        aux = DH_Pade33_vec(u[~mask], x, H, rho, theta)
        res[~mask] = np.exp(np.matmul(xi,aux)*dt)
        res[mask] = 1.
    else:
        aux = DH_Pade33_vec(u, x, H, rho, theta)
        res = np.exp(np.matmul(xi,aux)*dt)

    return res

# ###################### Analytic rHeston ################################

def integral(x, t, H, rho, theta):

    integrand = (lambda u: np.real(np.exp((1j*u)*x) * \
                               phi_rhest(u - 0.5j, t, H, rho, theta)) / \
                 (u**2 + 0.25))

    i, err = scipy.integrate.quad_vec(integrand, 0, np.inf)

    return i

# def integral_vec(x, t, H, rho, theta, grid = GRID):
#     aux = (np.tile(grid, (len(x),1)).transpose()*x).transpose()

#     i = np.real(np.exp(1j*aux)*phi_rhest_vec(grid - 0.5j, t, H, rho, theta)) / \
#         (grid**2 + 0.25)

#     i = i.sum(axis = 1)*(grid[1]-grid[0])
#     return i

def analytic_rhest(S0, strikes, t, H, rho, theta, options_type):

    # Pricing of vanilla options under "analytic" rHeston using Lewis Formula

    a = np.log(S0/strikes) + ImpliedDrift.drift(t)*t
    i = integral(a, t, H, rho, theta)
    r = ImpliedDrift.r(t)
    q = ImpliedDrift.q(t)
    out = S0 * np.exp(-q*t) - np.sqrt(S0*strikes) * np.exp(-(r+q)*t*0.5)/np.pi * i
    out = np.array([out]).flatten()

    for k in range(len(options_type)):
        if options_type[k] == 0:
            out[k] = Heston.call_put_parity(out[k], S0, strikes[k], r, q, t)

    if (out < 0).any():
```

```
        out[out < 0] = 0.

    return out

# def analytic_rhest_vec(S0, strikes, t, H, rho, theta,
    options_type):

#     # Pricing of vanilla options under "analytic" rHeston using
    Lewis Formula

#     a = np.log(S0/strikes) + ImpliedDrift.drift(t)*t
#     i = integral_vec(a, t, H, rho, theta)
#     r = ImpliedDrift.r(t)
#     q = ImpliedDrift.q(t)
#     out = S0 * np.exp(-q*t) - np.sqrt(S0*strikes) * np.exp(-(r+q)
    *t*0.5)/np.pi * i
#     out = np.array([out]).flatten()

#     for k in range(len(options_type)):
#         if options_type[k] == 0:
#             out[k] = Heston.call_put_parity(out[k], S0, strikes[k
    ], r, q, t)

#     if (out < 0).any():
#         out[out < 0] = 0.

#     return out

######################Simulation rHeston
    ###############################

# Psi for the QE Scheme of Lemma 7.
def psi_m(psi, ev, w):
    #psi minus

    beta2 = psi
    mask = psi > 0
    mask1 = psi <= 0
    if np.any(mask):
        beta2[mask] = 2./psi[mask]-1+np.sqrt(2./psi[mask]* \
                                             np.abs(2./psi[mask]-1)
    )
    if np.any(mask1):
        beta2[mask1] = 0.
    return ev/(1+beta2)*(np.sqrt(np.abs(beta2))+w)**2

def psi_p(psi, ev, u):
    #psi plus

    p = 2/(1+psi)
    res = (u<p)*(-ev)/2*(1+psi)
    mask = u > 0
    if np.any(mask):
        res[mask] = np.log(u[mask]/p[mask])
    return res

# functions for K_i, K_ii and K_01
```

```
def Gi(eta, alpha, dt, i):
    return np.sqrt(2*alpha-1)*eta/alpha * dt**alpha * ((i+1)**alpha
     - i**alpha)

def Gii(eta, H, dt, i):
    aux = 2*H
    return eta**2 * dt**aux * ((i+1)**aux - i**aux)

def G01(eta, alpha, dt):
    return Gi(eta,alpha,dt,0)*Gi(eta,alpha,dt,1)/dt

def HQE_sim(theta, H, rho, T, S0, paths, steps, eps0 = 1e-10):
    # HQE scheme

    # theta, H, rho: parameters of the rHeston model
    # T: final time of the simulations, in years
    # S0: spot price at time 0
    # paths: number of paths to simulate
    # steps: number of timesteps between 0 and T
    # eps0: lower bound for xihat

    dt = T/steps
    dt_sqrt = np.sqrt(dt)
    alpha = H + 0.5
    eta = theta/(gamma(alpha)*np.sqrt(2*H))
    rho2m1 = np.sqrt(1-rho*rho)

    W = np.random.normal(0.,1.,size = (steps,paths))
    Wperp = np.random.normal(0.,1.,size = (steps,paths))
    Z = np.random.normal(0.,1.,size = (steps,paths))
    U = np.random.uniform(0.,1.,size = (steps,paths))
    Uperp = np.random.uniform(0.,1.,size = (steps,paths))

    tj = np.arange(0,steps,1)*dt
    tj += dt

    xij = variance_curve(tj)
    G0del = Gi(eta,alpha,dt,0)
    G00del = Gii(eta,alpha,dt,0)
    G11del = Gii(eta,alpha,dt,1)
    G01del = G01(eta,alpha,dt)
    G00j = np.zeros(steps)

    for j in range(steps):
        G00j[j] = Gii(eta,H,dt,j)
    bstar = np.sqrt((G00j)/dt)

    rho_vchi = G0del/np.sqrt(G00del*dt)
    beta_vchi = G0del/dt

    u = np.zeros((steps,paths))
    chi = np.zeros((steps,paths))
    v = np.ones(paths)*variance_curve(0)
    hist_v = np.zeros((steps,paths))
    hist_v[0,:] = v
    xihat = np.ones(paths)*xij[0]
    x = np.zeros((steps,paths))
```

```python
    y = np.zeros(paths)
    w = np.zeros(paths)

    for j in range(steps):
        xibar = (xihat + 2*H*v)/(1+2*H)

        psi_chi = 2*beta_vchi*xibar*dt/(xihat**2)
        psi_eps =  2/(xihat**2)*xibar*(G00del - G0del**2/dt)
        aux_ = xihat/2

        z_chi = np.zeros(paths)
        z_eps = np.zeros(paths)

        mask1 = psi_chi < 1.5
        mask2 = psi_chi >= 1.5
        mask3 = psi_eps < 1.5
        mask4 = psi_eps >= 1.5

        if np.any(mask1):
            z_chi[mask1] = psi_m(psi_chi[mask1],aux_[mask1],W[j,mask1])
        if np.any(mask2):
            z_chi[mask2] = psi_m(psi_chi[mask2],aux_[mask2],U[j,mask2])
        if np.any(mask3):
            z_eps[mask3] = psi_m(psi_eps[mask3],aux_[mask3],Wperp[j,mask3])
        if np.any(mask4):
            z_eps[mask4] = psi_m(psi_eps[mask4],aux_[mask4],Uperp[j,mask4])

        chi[j,:] = (z_chi-aux_)/beta_vchi
        eps = z_eps - aux_
        u[j,:] = beta_vchi*chi[j,:]+eps
        vf = xihat + u[j,:]
        vf[vf < eps0] = eps0

        dw = (v+vf)/2*dt
        w += dw
        y += chi[j,:]
        x[j,:] = x[j-1,:] + ImpliedDrift.drift(T)*dt - dw/2 + np.sqrt(dw) \
                 * (rho2m1*Z[j,:]) + rho*chi[j,:]

        btilde = np.flip(bstar[1:j+1])
        if j < steps-1:
            xihat = xij[j+1] + (np.matmul(btilde,chi[:j,:]))
        v = vf
        hist_v[j,:] = v
    return np.vstack([np.ones(paths)*S0,(np.exp(x)*S0)]),hist_v
```

Rough Bergomi

utils_rBergomi.py

```python
import numpy as np
```

```python

# TBSS kernel applicable to the rBergomi variance process.
def g(x, a):
    return x**a

# Optimal discretisation of TBSS process for minimising hybrid scheme error.
def b(k, a):
    return ((k**(a+1)-(k-1)**(a+1))/(a+1))**(1/a)

# Covariance matrix for given alpha and n, assuming kappa = 1.
def cov(a, n):
    cov = np.array([[0.,0.],[0.,0.]])
    cov[0,0] = 1./n
    cov[0,1] = 1./((1.*a+1) * n**(1.*a+1))
    cov[1,1] = 1./((2.*a+1) * n**(2.*a+1))
    cov[1,0] = cov[0,1]
    return cov
```

rbergomi.py

```python
import numpy as np
from scipy.signal import convolve
from numpy.random import default_rng
from utils_rBergomi import *
import ImpliedDrift as iD

# Class for generating paths of the rBergomi model.
class rBergomi(object):

    def __init__(self, n, N, T, a):

        # Basic assignments
        self.T = T # Maturity
        self.n = n # Steps per year
        self.dt = 1.0/self.n # Step size
        self.s = np.round(self.n * self.T).astype(int) # Number of total steps
        self.t = np.linspace(0, self.T, 1 + self.s)[np.newaxis,:] # Time grid
        self.a = a # Alpha
        self.N = N # Number of paths

        # Construct hybrid scheme correlation structure with kappa = 1
        self.e = np.array([0,0])
        self.c = cov(self.a, self.n)

    def dW1(self):
        np.random.seed(0)
        # Produces random numbers for variance process with required covariance structure
```

```
        N = int(self.N/2)
        w = np.random.multivariate_normal(self.e, self.c, (N, self.s))
        return np.concatenate((w,-w), axis = 0)

    def dW2(self):
        np.random.seed(0)
        #Obtain orthogonal increments
        N = int(self.N/2)
        w = np.random.randn(N, self.s) * np.sqrt(self.dt)
        return np.concatenate((w,-w), axis = 0)

    def Y(self, dW):
        #Constructs Volterra process from appropriately correlated 2d Brownian increments

        Y1 = np.zeros((self.N, 1 + self.s)) # Exact integrals
        Y2 = np.zeros((self.N, 1 + self.s)) # Riemann sums

        Y1[:,1 : self.s+1] = dW[:, :self.s, 1]   # Assumes kappa = 1

        # Construct arrays for convolution
        G = np.zeros(1 + self.s) # Gamma
        for k in np.arange(2, 1 + self.s, 1):
            G[k] = g(b(k, self.a)/self.n, self.a)

        X = dW[:,:,0] # Xi

        # Compute convolution and extract relevant terms
        for i in range(self.N):
            Y2[i,:] = np.convolve(G, X[i,:])[:1+self.s]

        # Finally contruct and return full process
        return np.sqrt(2 * self.a + 1) * (Y1 + Y2)

    def dZ(self, dW1, dW2, rho):
        # Constructs correlated price Brownian increments, dB

        self.rho = rho
        return rho * dW1[:,:,0] + np.sqrt(1 - rho**2) * dW2

    def V(self, Y, xi, eta):
        # rBergomi variance process.
        self.xi = xi
        self.eta = eta
        a = self.a
        t = self.t
        return xi * np.exp(eta * Y - 0.5 * eta**2 * t**(2 * a + 1))
        #return xi * ne.evaluate('exp(eta * Y - 0.5 * eta**2 * t**(2 * a + 1))')

    def S_all_path(self, V, dZ, r, q, S0):
        # rBergomi price process.
        self.S0 = S0
        dt = self.dt
        rho = self.rho
```

```

        # Construct non-anticipative Riemann increments
        increments = np.sqrt(V[:,:-1]) * dZ - 0.5 * V[:,:-1] * dt + (r - q) * dt
        integral = np.cumsum(increments, axis = 1)

        S = np.zeros_like(V)
        S[:,0] = S0
        S[:,1:] = S0 * np.exp(integral)
        return S

    def S(self, V, dZ, r, q, S0):
        # rBergomi price process.
        self.S0 = S0
        dt = self.dt
        rho = self.rho

        # Construct non-anticipative Riemann increments
        exponent = np.zeros(self.N)
        for i in range(self.s):
            exponent += np.sqrt(V[:,i]) * dZ[:,i] - 0.5 * V[:,i] * dt + (r - q) * dt

        return S0 * np.exp(exponent)

    def global_S(self, V, dZ, S0, steps, index = 0):
        # rBergomi price process.
        self.S0 = S0
        dt = self.dt
        rho = self.rho

        r = iD.r(self.t[0], index)
        q = iD.q(self.t[0], index)

        S = list()
        logS = np.log(S0)
        for i in range(self.s):
            logS += np.sqrt(V[:,i]) * dZ[:,i] - 0.5 * V[:,i] * dt + (r[i] - q[i]) * dt
            if i in steps:
                S.append(np.exp(logS))

        S.append(np.exp(logS))

        return np.array(S)
```

Quintic Ornstein-Uhlenbeck

```
import numpy as np
import variance_curve as vc
import ImpliedDrift as iD
import scipy
import BlackScholes as bs

from scipy.integrate import quad

def horner_vector(poly, n, x):
```

```
    #Initialize result
    result = poly[0].reshape(-1,1)
    for i in range(1,n):
        result = result*x + poly[i].reshape(-1,1)
    return result



def gauss_dens(mu,sigma,x):
    return 1/np.sqrt(2*np.pi*sigma**2)*np.exp(-(x-mu)**2/(2*sigma**2))



def vix_futures(H, eps, T, a_k_part, k, r, q, n_steps, index = 0):

    a2,a4 = (0,0)
    a0,a1,a3,a5 = a_k_part
    a_k = np.array([a0, a1, a2, a3, a4, a5])

    kappa_tild = (0.5-H)/eps
    eta_tild = eps**(H-0.5)

    delt = 30/365
    T_delta = T + delt

    std_X = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*kappa_tild*T)))
    dt = delt/(n_steps)
    tt = np.linspace(T, T_delta, n_steps+1)

    FV_curve_all_vix = vc.variance_curve(tt, index)

    exp_det = np.exp(-kappa_tild*(tt-T))
    cauchy_product = np.convolve(a_k,a_k)

    std_Gs_T = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*kappa_tild*(tt-T))))
    std_X_t = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*kappa_tild*tt)))
    std_X_T = std_X

    n = len(a_k)

    normal_var = np.sum(cauchy_product[np.arange(0,2*n,2)].reshape(-1,1)*std_X_t**(np.arange(0,2*n,2).reshape(-1,1))*\
    scipy.special.factorial2(np.arange(0,2*n,2).reshape(-1,1)-1),axis=0) #g(u)

    beta = []
    for i in range(0,2*n-1):
        k_array = np.arange(i,2*n-1)
        beta_temp = ((std_Gs_T**((k_array-i).reshape(-1,1))*((k_array-i-1)%2).reshape(-1,1)*\
            scipy.special.factorial2(k_array-i-1).reshape(-1,1)*\
            (scipy.special.comb(k_array,i)).reshape(-1,1))*\
            exp_det**(i))*cauchy_product[k_array].reshape(-1,1)
```

```
        beta.append(np.sum(beta_temp,axis=0))

    beta = np.array(beta)*FV_curve_all_vix/normal_var
    beta = (np.sum((beta[:,:-1]+beta[:,1:])/2,axis=1))*dt

    sigma = np.sqrt(eps**(2*H)/(1-2*H)*(1-np.exp((2*H-1)*T/eps)))

    f = lambda x: np.sqrt(horner_vector(beta[::-1], len(beta), x)/delt)*100 * gauss_dens(0, sigma, x)

    Ft, err = quad(f, -np.inf, np.inf)

    return Ft * np.exp((r-q)*T)



def vix_iv(H, eps, T, a_k_part, K, r, q, n_steps, index = 0):

    a2,a4 = (0,0)
    a0,a1,a3,a5 = a_k_part
    a_k = np.array([a0, a1, a2, a3, a4, a5])

    kappa_tild = (0.5-H)/eps
    eta_tild = eps**(H-0.5)

    delt = 30/365
    T_delta = T + delt

    std_X = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*kappa_tild*T)))
    dt = delt/(n_steps)
    tt = np.linspace(T, T_delta, n_steps+1)

    FV_curve_all_vix = vc.variance_curve(tt, index)

    exp_det = np.exp(-kappa_tild*(tt-T))
    cauchy_product = np.convolve(a_k,a_k)

    std_Gs_T = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*kappa_tild*(tt-T))))
    std_X_t = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*kappa_tild*tt)))
    std_X_T = std_X

    n = len(a_k)

    normal_var = np.sum(cauchy_product[np.arange(0,2*n,2)].reshape(-1,1)*std_X_t**(np.arange(0,2*n,2).reshape(-1,1))*\
    scipy.special.factorial2(np.arange(0,2*n,2).reshape(-1,1)-1), axis=0) #g(u)

    beta = []
    for i in range(0,2*n-1):
        k_array = np.arange(i,2*n-1)
        beta_temp = ((std_Gs_T**((k_array-i).reshape(-1,1))*((k_array-i-1)%2).reshape(-1,1)*\
            scipy.special.factorial2(k_array-i-1).reshape(-1,1)*\
```

```
            (scipy.special.comb(k_array,i)).reshape(-1,1))*\
            exp_det**(i))*cauchy_product[k_array].reshape(-1,1)
        beta.append(np.sum(beta_temp,axis=0))

    beta = np.array(beta)*FV_curve_all_vix/normal_var
    beta = (np.sum((beta[:,:-1]+beta[:,1:])/2,axis=1))*dt

    sigma = np.sqrt(eps**(2*H)/(1-2*H)*(1-np.exp((2*H-1)*T/eps)))

    N = len(K); P = np.zeros(N);

    for i in range(N):

        f = lambda x: np.maximum(np.sqrt(horner_vector(beta[::-1], len(beta), x)/delt)*100 - K[i], 0) * gauss_dens(0, sigma, x)
        P[i], err = quad(f, -np.inf, np.inf)

    return P * np.exp((r-q)*T)



def dW(n_steps,N_sims):
    w = np.random.normal(0, 1, (n_steps, N_sims))
    #Antithetic variates
    w = np.concatenate((w, -w), axis = 1)
    return w



def local_reduction(rho,H,eps,T,a_k_part,S0,strike_array,n_steps, N_sims,w1,r,q, index = 0):

    eta_tild = eps**(H-0.5)
    kappa_tild = (0.5-H)/eps

    a_0,a_1,a_3,a_5 = a_k_part
    a_k = np.array([a_0,a_1,0,a_3,0,a_5])

    dt = T/n_steps
    tt = np.linspace(0., T, n_steps + 1)

    exp1 = np.exp(kappa_tild*tt)
    exp2 = np.exp(2*kappa_tild*tt)

    diff_exp2 = np.concatenate((np.array([0.]),np.diff(exp2)))
    std_vec = np.sqrt(diff_exp2/(2*kappa_tild))[:,np.newaxis] #to be broadcasted columnwise
    exp1 = exp1[:,np.newaxis]
    X = (1/exp1)*(eta_tild*np.cumsum(std_vec*w1, axis = 0))
    Xt = np.array(X[:-1])
    del X

    tt = tt[:-1]
    std_X_t = np.sqrt(eta_tild**2/(2*kappa_tild)*(1-np.exp(-2*kappa_tild*tt)))
    n = len(a_k)

```

```
    cauchy_product = np.convolve(a_k,a_k)
    normal_var = np.sum(cauchy_product[np.arange(0,2*n,2)].reshape(-1,1)*std_X_t**(np.arange(0,2*n,2).reshape(-1,1))*\
        scipy.special.factorial2(np.arange(0,2*n,2).reshape(-1,1)-1),axis=0)

    f_func = horner_vector(a_k[::-1], len(a_k), Xt)

    del Xt

    fv_curve = vc.variance_curve(tt, index).reshape(-1,1)

    volatility = f_func/np.sqrt(normal_var.reshape(-1,1))
    del f_func
    volatility = np.sqrt(fv_curve)*volatility

    logS1 = np.log(S0)
    for i in range(w1.shape[0]-1):
        logS1 = logS1 - 0.5*dt*(volatility[i]*rho)**2 + np.sqrt(dt)*rho*volatility[i]*w1[i+1] + rho**2*(r-q)*dt
    del w1
    ST1 = np.exp(logS1)
    del logS1

    int_var = np.sum(volatility[:-1,]**2*dt,axis=0)
    Q = np.max(int_var)+1e-9
    del volatility
    X = (bs.BSCall(ST1, strike_array.reshape(-1,1), T, r, q, np.sqrt((1-rho**2)*int_var/T))).T
    Y = (bs.BSCall(ST1, strike_array.reshape(-1,1), T, r, q, np.sqrt(rho**2*(Q-int_var)/T))).T
    del int_var
    eY = (bs.BSCall(S0, strike_array.reshape(-1,1), T, r, q, np.sqrt(rho**2*Q/T))).T

    c = []
    for i in range(strike_array.shape[0]):
        cova = np.cov(X[:,i]+10,Y[:,i]+10)[0,1]
        varg = np.cov(X[:,i]+10,Y[:,i]+10)[1,1]
        if (cova or varg)<1e-8:
            temp = 1e-40
        else:
            temp = np.nan_to_num(cova/varg,1e-40)
        temp = np.minimum(temp,2)
        c.append(temp)
    c = np.array(c)

    call_mc_cv1 = X-c*(Y-eY)
    del X
    del Y
    del eY

    return np.average(call_mc_cv1,axis=0)



def global_reduction(rho,H,eps,T,a_k_part,S0,strike_array,n_steps,
```

```
    N_sims,w1,steps,maturities, index = 0):

    eta_tild = eps**(H-0.5)
    kappa_tild = (0.5-H)/eps

    a_0,a_1,a_3,a_5 = a_k_part
    a_k = np.array([a_0,a_1,0,a_3,0,a_5])

    dt = T/n_steps
    tt = np.linspace(0., T, n_steps + 1)

    r = iD.r(tt, index)
    q = iD.q(tt, index)

    exp1 = np.exp(kappa_tild*tt)
    exp2 = np.exp(2*kappa_tild*tt)

    diff_exp2 = np.concatenate((np.array([0.]),np.diff(exp2)))
    std_vec = np.sqrt(diff_exp2/(2*kappa_tild))[:,np.newaxis] #to be broadcasted columnwise
    exp1 = exp1[:,np.newaxis]
    X = (1/exp1)*(eta_tild*np.cumsum(std_vec*w1, axis = 0))
    Xt = np.array(X[:-1])
    del X

    tt = tt[:-1]
    std_X_t = np.sqrt(eta_tild**2/(2*kappa_tild)*(1-np.exp(-2*kappa_tild*tt)))
    n = len(a_k)

    cauchy_product = np.convolve(a_k,a_k)
    normal_var = np.sum(cauchy_product[np.arange(0,2*n,2)].reshape(-1,1)*std_X_t**(np.arange(0,2*n,2).reshape(-1,1))*\
        scipy.special.factorial2(np.arange(0,2*n,2).reshape(-1,1)-1),axis=0)

    f_func = horner_vector(a_k[::-1], len(a_k), Xt)

    del Xt

    fv_curve = vc.variance_curve(tt, index).reshape(-1,1)

    volatility = f_func/np.sqrt(normal_var.reshape(-1,1))
    del f_func
    volatility = np.sqrt(fv_curve)*volatility

    ST1 = list()
    logS1 = np.log(S0)
    for i in range(w1.shape[0]-1):
        logS1 = logS1 - 0.5*dt*(volatility[i]*rho)**2 + np.sqrt(dt)*rho*volatility[i]*w1[i+1] + rho**2*(r[i]-q[i])*dt
        if i in steps:
            ST1.append(np.exp(logS1))
    del w1
    ST1.append(np.exp(logS1))
    ST1 = np.array(ST1)
    del logS1
```

```

    int_var = np.sum(volatility[:-1,]**2*dt,axis=0)
    Q = np.max(int_var)+1e-9
    del volatility

    P = list()

    for i in range(len(steps)):
        T_aux = maturities[i]
        r = iD.r(T_aux, index); q = iD.q(T_aux, index)

        X = (bs.BSCall(ST1[i], strike_array.reshape(-1,1), T_aux, r
    , q, np.sqrt((1-rho**2)*int_var/T))).T
        Y = (bs.BSCall(ST1[i], strike_array.reshape(-1,1), T_aux, r
    , q, np.sqrt(rho**2*(Q-int_var)/T))).T
        eY = (bs.BSCall(S0, strike_array.reshape(-1,1), T_aux, r, q
    , np.sqrt(rho**2*Q/T))).T

        c = []
        for i in range(strike_array.shape[0]):
            cova = np.cov(X[:,i]+10,Y[:,i]+10)[0,1]
            varg = np.cov(X[:,i]+10,Y[:,i]+10)[1,1]
            if (cova or varg)<1e-8:
                temp = 1e-40
            else:
                temp = np.nan_to_num(cova/varg,1e-40)
            temp = np.minimum(temp,2)
            c.append(temp)
        c = np.array(c)

        call_mc_cv1 = X-c*(Y-eY)
        P.append(np.average(call_mc_cv1,axis=0))

    return np.array(P)
```