

# Libraries

Nodari Alessandro & Proserpio Lorenzo

## Utils

### ImpliedDrift.py

```
1 import numpy as np
2 import pandas as pd
3 from scipy.interpolate import CubicSpline
4
5 dates = np.array(["23-01-23", "24-01-23", "25-01-23", "26-01-23",
6 "27-01-23", "30-01-23", "06-02-23", "13-02-23", "21-02-23"])
7 data = pd.read_csv("ratesOIS.csv")
8 tenor = np.array(data.TENOR)
9 Forw = pd.read_csv("forw.csv")
10 spot = np.array(pd.read_csv("spot.csv").Spot)
11 TENOR = pd.read_csv("tenor.csv")
12
13
14 def r(x, index = 0):
15     rates = np.array(data[dates[index]])/100
16     cs = CubicSpline(tenor, rates)
17     return cs(x)
18
19 def drift(x, index = 0):
20     S0 = spot[index]
21     F = np.array(Forw[dates[index]]).flatten()
22     Tenor = np.array(TENOR[dates[index]]).flatten()
23     d = -np.log(S0/F).flatten()/Tenor
24     cs = CubicSpline(Tenor, d)
25     return cs(x)
26
27 def q(x, index = 0):
28     return r(x, index) - drift(x, index)
```

### BlackScholes.py

```
1 import numpy as np
2 from scipy.stats import norm
3
4 def BSCall(S0, K, T, r, q, sigma):
5
6     # Price of a call under Black&Scholes
7
8     # S0: spot price
9     # K: strike
```

```

10 # T: years to expiration
11 # r: risk free rate (1 = 100%)
12 # q: annual yield
13 # sigma: volatility (1 = 100%)
14
15
16 sig = sigma*np.sqrt(T)
17 d1 = (np.log(S0/K) + (r-q)*T)/sig + sig/2.
18 d2 = d1 - sig
19
20 return S0*norm.cdf(d1) - K*np.exp(-(r-q)*T)*norm.cdf(d2)
21
22 def BSPut(S0, K, T, r, q, sigma):
23
24     # Price of a put under Black&Scholes
25
26     # S0: spot price
27     # K: strike
28     # T: years to expiration
29     # r: risk free rate (1 = 100%)
30     # q: annual yield
31     # sigma: volatility (1 = 100%)
32
33     return BSCall(S0, K, T, r, q, sigma) + K*np.exp(-(r-q)*T) - S0
34
35 def BSImpliedVol(S0, K, T, r, q, P, Option_type = 1, toll = 1e-10):
36
37     # Calculate implied volatility from prices using bisection
38
39     # NOTE: All the parameters can be np.array(), except for P that
40     # MUST be a np.array().
41
42     # S0: spot price
43     # K: strike
44     # T: years to expiration
45     # r: risk free rate (1 = 100%)
46     # q: annual yield
47     # P: prices
48     # Option_type: 1 for calls, 0 for puts
49     # toll: error in norm 1
50
51     if Option_type:
52         BSFormula = np.vectorize(BSCall)
53     else:
54         BSFormula = np.vectorize(BSPut)
55
56     N = P.shape[0]
57     sigma_low = 1e-10*np.ones(N)
58     sigma_high = 10*np.ones(N)
59
60     P_low = BSFormula(S0, K, T, r, q, sigma_low)
61     P_high = BSFormula(S0, K, T, r, q, sigma_high)
62     sigma = (sigma_low + sigma_high)/2.
63
64     while np.sum(P_high - P_low) > toll:
65         sigma = (sigma_low + sigma_high)/2.
66         P_mean = BSFormula(S0, K, T, r, q, sigma)

```

```

66     P_low += (P_mean < P)*(P_mean - P_low)
67     sigma_low += (P_mean < P)*(sigma - sigma_low)
68     P_high += (P_mean >= P)*(P_mean - P_high)
69     sigma_high += (P_mean >= P)*(sigma - sigma_high)
70
71     return sigma

```

## variance\_curve.py

```

1  import numpy as np
2
3  Z1 = np.array([0.23934445564954748, 0.2370172145384514,
4                0.23319383855246545, 0.22779527372712297, 0.22410506177795986,
5                0.22796530521676028, 0.22992033119402192, 0.23360387896928214,
6                0.23923251959598327])
7
8  Z2 = np.array([0.2355916740288041, 0.23547872334450043,
9                0.2278527332290963, 0.2493545607795239, 0.2684664268847795,
10               0.16758709131144714, 0.23526774429356268, 0.16354991037070024,
11               0.09001896821824877])
12
13 Z3 = np.array([2.3126258447474375, 2.077549483492911,
14               2.1200577204482105, 2.637109433927137, 2.9677374658205307,
15               2.385086643216494, 3.064763505035709, 2.3367542064926443,
16               2.114562825304444])
17
18 # Compute the initial forward variance curve at a given time t
19 # using the Gompertz function with precomputed parameters
20
21 def Gompertz(t, index = 0):
22     z1 = Z1[index]; z2 = Z2[index]; z3 = Z3[index];
23     return z1 * np.exp(-z2 * np.exp(-z3 * t))
24
25 def variance_curve(t, index = 0):
26     z1 = Z1[index]; z2 = Z2[index]; z3 = Z3[index];
27     return (z1 * np.exp(-z2 * np.exp(-z3 * t)))**2 + 2*t*z1**2*z2*
28     z3*np.exp(-2*z2*np.exp(-z3*t)-z3*t)

```

## Heston

### Heston.py

```

1  import numpy as np
2  import QuantLib as ql
3  import scipy.integrate
4  from scipy.special import ive
5
6  ##### PUT-CALL PARITY #####
7
8  def put_call_parity(put, S0, strike, r, q, tau):
9      # Standard put_call_parity
10     return put + S0*np.exp(-q*tau) - strike*np.exp(-r*tau)
11
12  def call_put_parity(call, S0, strike, r, q, tau):
13     return call - S0*np.exp(-q*tau) + strike*np.exp(-r*tau)
14
15
16

```

```

17 ##### Analytic Heston #####
18
19 def phi_hest(u, tau, sigma_0, kappa, eta, theta, rho):
20
21     # Compute the characteristic function for Heston Model
22
23     # u: argument of the function (where you want to evaluate)
24     # tau: time to expiration
25     # sigma_0, kappa, eta, theta, rho: Heston parameters
26
27     alpha_hat = -0.5 * u * (u + 1j)
28     beta = kappa - 1j * u * theta * rho
29     gamma = 0.5 * theta ** 2
30     d = np.sqrt(beta**2 - 4 * alpha_hat * gamma)
31     g = (beta - d) / (beta + d)
32     h = np.exp(-d*tau)
33     A_ = (beta - d)*tau - 2*np.log((g*h-1) / (g-1))
34     A = kappa * eta / (theta**2) * A_
35     B = (beta - d) / (theta**2) * (1 - h) / (1 - g*h)
36     return np.exp(A + B * sigma_0)
37
38 def integral(x, tau, sigma_0, kappa, eta, theta, rho):
39
40     # Pseudo-probabilities
41
42     # x: log-prices discounted
43
44     integrand = (lambda u: np.real(np.exp((1j*u + 0.5)*x) * \
45                                     phi_hest(u - 0.5j, tau, sigma_0,
46                                     kappa, eta, theta, rho)) / \
47                                     (u**2 + 0.25))
48
49     i, err = scipy.integrate.quad_vec(integrand, 0, np.inf)
50
51     return i
52
53 def analytic_hest(S0, strikes, tau, r, q, kappa, theta, rho, eta,
54                   sigma_0, options_type):
55
56     # Pricing of vanilla options under analytic Heston
57
58     a = np.log(S0/strikes) + (r-q)*tau
59     i = integral(a, tau, sigma_0, kappa, eta, theta, rho)
60
61     out = S0 * np.exp(-q*tau) - strikes * np.exp(-r*tau)/np.pi * i
62     out = np.array([out]).flatten()
63
64     for k in range(len(out)):
65         if options_type[k] == 0:
66             out[k] = call_put_parity(out[k], S0, strikes[k], r, q,
67                                     tau)
68
69     return out
70
71 ##### COS METHOD Le Floch #####

```

```

70 def phi_hest_0(u, tau, r, q, sigma_0, kappa, eta, theta, rho):
71
72     # Compute the characteristic function for Heston Model with
    log_asset = 0
73
74     # u: argument of the function (where you want to evaluate)
75     # r: risk-free-rate
76     # q: annual percentage yield
77     # tau: time to expiration
78     # sigma_0, kappa, eta, theta, rho: Heston parameters
79
80     beta = (kappa - 1j*rho*u*theta)
81     d = np.sqrt(beta**2 + (theta**2)*(1j*u+u**2))
82     r_minus = (beta - d)
83     g = r_minus/(beta + d)
84     aux = np.exp(-d*tau)
85
86     term_1 = sigma_0/(theta**2) * ((1-aux)/(1-g*aux)) * r_minus
87     term_2 = kappa*eta/(theta**2) * (tau*r_minus - 2*np.log((1-g*
    aux) / (1-g)))
88     term_3 = 1j*(r-q)*u*tau
89
90     return np.exp(term_1)*np.exp(term_2)*np.exp(term_3)
91
92 def chi_k(k, c, d, a, b):
93     # Auxiliary function for U_k
94
95     aux_1 = k*np.pi/(b-a)
96     aux_2 = np.exp(d)
97     aux_3 = np.exp(c)
98
99     return (np.cos(aux_1*(d-a))*aux_2 - \
100            aux_3 + \
101            aux_1*np.sin(aux_1*(d-a))*aux_2) / (1+aux_1**2)
102
103 def psi_k(k, c, d, a, b):
104     # Auxiliary function for U_k
105
106     if k == 0:
107         return d - c
108
109     aux = k*np.pi/(b-a)
110     return np.sin(aux*(d-a)) / aux
111
112 def U_k_put(k, a, b):
113     # Auxiliary for cos_method
114
115     return 2./(b-a) * (psi_k(k, a, 0, a, b) - chi_k(k, a, 0, a, b))
116
117 def optimal_ab(r, q, tau, sigma_0, kappa, eta, theta, rho, L = 12):
118     # Compute the optimal interval for the truncation
119     aux = np.exp(-kappa*tau)
120     c1 = (r-q)*tau - sigma_0 * tau / 2
121
122     c2 = (sigma_0) / (4*kappa**3) * (4*kappa**2*(1+(rho*theta*tau
    -1)*aux) \
123
    + kappa*(4*rho*theta*(aux-1)\

```

```

124         -2*theta**2*tau*aux) \
125         +theta**2*(1-aux*aux)) \
126         + eta/(8*kappa**3)*(8*kappa**3*tau - 8*kappa**2*(1+ rho*
theta*tau +(rho*theta*tau-1)*aux)\
127         + 2*kappa*((1+2*aux)*theta**2*tau+8*(1-
aux)*rho*theta)\
128         + theta**2*(aux*aux+4*aux-5))
129
130
131     return c1 - 12*np.sqrt(np.abs(c2)), c1 + 12*np.sqrt(np.abs(c2))
132
133
134 def precomputed_terms(r, q, tau, sigma_0, kappa, eta, theta, rho, L
, N):
135     # Auxiliary term precomputed
136
137     a,b = optimal_ab(r, q, tau, sigma_0, kappa, eta, theta, rho, L)
138     aux = np.pi/(b-a)
139     out = np.zeros(N-1)
140
141     for k in range(1,N):
142         out[k-1] = np.real(np.exp(-1j*k*a*aux)*\
143             phi_hest_0(k*aux, tau, r, q, sigma_0,
kappa, eta, theta, rho))
144
145     return out, a, b
146
147 def V_k_put(k, a, b, S0, K, z):
148     # V_k coefficients for puts
149
150     return 2./(b-a)*(K*psi_k(k, a, z, a, b) - S0*chi_k(k, a, z, a,
b))
151
152 def cos_method_Heston_LF(precomp_term, a, b, tau, r, q, sigma_0,
kappa, eta, theta, rho, S0,\
153     strikes, N, options_type, L=12):
154     # Cosine Fourier Expansion for evaluating vanilla options under
Heston using LeFloch correction
155     # Should be better for deep otm options.
156
157     # precomp_term: precomputed terms from the function
precomputed_terms
158     # a,b: extremes of the interval to approximate
159     # tau: time to expiration (annualized) (must be a number)
160     # r: risk-free-rate
161     # q: yield
162     # sigma_0, kappa, eta, theta, rho: Heston parameters
163     # S0: initial spot price
164     # strikes: np.array of strikes
165     # N: number of terms of the truncated expansion
166     # options_type: binary np.array (1 for calls, 0 for puts)
167     # L: truncation level
168
169     z = np.log(strikes/S0)
170
171     out = 0.5 * np.real(phi_hest_0(0, tau, r, q, sigma_0, kappa,
eta, theta, rho))*\

```

```

172         V_k_put(0, a, b, S0, strikes, z)
173
174     for k in range(1,N):
175         out = out + precomp_term[k-1]*V_k_put(k, a, b, S0, strikes,
176             z)
177
178     D = np.exp(-r*tau)
179     out = out*D
180
181     for k in range(len(strikes)):
182         if options_type[k] == 1:
183             out[k] = put_call_parity(out[k], S0, strikes[k], r, q,
184                 tau)
185
186     return out
187
188 #####Calibration#####
189
190 def setup_model(_yield_ts, _dividend_ts, _spot,
191     init_condition):
192     # Setup Heston model object
193
194     # _yield_ts: Term Structure for yield (QuantLib object)
195     # _dividend_ts: Term Structure for dividend_ts (QuantLib object)
196     )
197     # init_condition: eta, kappa, theta, rho, sigma_0
198
199     eta, kappa, theta, rho, sigma_0 = init_condition
200     process = ql.HestonProcess(_yield_ts, _dividend_ts,
201         ql.QuoteHandle(ql.SimpleQuote(_spot)),
202         sigma_0, kappa, eta, theta, rho)
203     model = ql.HestonModel(process)
204     engine = ql.AnalyticHestonEngine(model)
205     return model, engine
206
207 def setup_helpers(engine, expiration_dates, strikes,
208     data, ref_date, spot, yield_ts,
209     dividend_ts, calendar):
210     # Helpers for Heston Calibration
211
212     # engine: Heston.setup_model output
213     # expiration_dates: maturities
214     # data: IV market data
215     # ref_date: date for the calculation
216     # yield_ts: Term Structure for yield (QuantLib object)
217     # dividend_ts: Term Structure for dividend_ts (QuantLib object)
218     # calendar: type of calendar for calculations
219
220     heston_helpers = []
221     grid_data = []
222     for i, date in enumerate(expiration_dates):
223         for j, s in enumerate(strikes):
224             t = (date - ref_date)
225             p = ql.Period(t, ql.Days)
226             vols = data[i][j]
227             helper = ql.HestonModelHelper(

```

```

225         p, calendar, spot, s,
226         ql.QuoteHandle(ql.SimpleQuote(vols)),
227         yield_ts, dividend_ts)
228         helper.setPricingEngine(engine)
229         heston_helpers.append(helper)
230         grid_data.append((date, s))
231     return heston_helpers, grid_data
232
233 def cost_function_generator(model, helpers, norm=False):
234     # Define cost function for the calibration (usually Mean Square
235     # Error)
236
237     def cost_function(params):
238         params_ = ql.Array(list(params))
239         model.setParams(params_)
240         error = [h.calibrationError() for h in helpers]
241         if norm:
242             return np.sqrt(np.sum(np.abs(error)))
243         else:
244             return error
245     return cost_function
246
247 #####Simulation Heston
248 #####
249
250 def create_totems(base, start, end):
251     # create the grid
252
253     totems = np.ones(end-start+2)
254     index = 1
255     for j in range(start, end+1):
256         totems[index] = base**j
257         index += 1
258
259     totems[0] = 0
260     return totems
261
262 def calc_nu_bar(kappa, eta, theta):
263     # compute v bar
264     return 4*kappa*eta/theta**2
265
266 def x2_exp_var(nu_bar, kappa, theta, dt):
267     # compute E[X_2] and Var[X_2]
268
269     aux = kappa*dt/2.
270     c1 = np.cosh(aux)/np.sinh(aux)
271     c2 = (1./np.sinh(aux))**2
272     exp_x2 = nu_bar*theta**2*((-2.+kappa*dt*c1)/(4*kappa**2))
273     var_x2 = nu_bar*theta**4*((-8.+2*kappa*dt*c1+\
274         kappa**2*dt**2*c2)/(8*kappa**4))
275     return exp_x2, var_x2
276
277 def Z_exp_var(nu_bar, exp_x2, var_x2):
278     # compute E[Z] and Var[Z]
279
280     return 4*exp_x2/nu_bar, 4*var_x2/nu_bar

```



```

280 def xi_exp(nu_bar, kappa, theta, dt, totem):
281     # compute  $E[X_i]$  and  $E[X_i^2]$ 
282
283     z = 2*kappa*np.sqrt(totem) / (theta**2*np.sinh(kappa*dt/2.))
284     iv_pre = ive(nu_bar/2.-1., z)
285     exp_xi = (z*ive(nu_bar/2.,z))/(2*iv_pre)
286     exp_xi2 = exp_xi + (z**2*ive(nu_bar/2.+1,z))/(4.*iv_pre)
287
288     return exp_xi, exp_xi2
289
290 def create_caches(base, start, end, kappa, eta, theta, dt):
291     # precompute the caches for IV*
292
293     totems = create_totems(base, start, end)
294     caches_exp = np.zeros(end-start+2)
295     caches_var = np.zeros(end-start+2)
296     nu_bar = calc_nu_bar(kappa, eta, theta)
297     exp_x2, var_x2 = x2_exp_var(nu_bar, kappa, theta, dt)
298     exp_Z, var_Z = Z_exp_var(nu_bar, exp_x2, var_x2)
299
300     for j in range(1,end-start+2):
301         exp_xi, exp_xi2 = xi_exp(nu_bar, kappa, theta, dt, totems[j])
302
303         caches_exp[j] = exp_x2 + exp_xi*exp_Z
304         caches_var[j] = var_x2 + exp_xi*var_Z + \
305             (exp_xi2-exp_xi**2)*exp_Z**2
306
307     caches_exp[0] = exp_x2
308     caches_var[0] = var_x2
309     return totems, caches_exp, caches_var
310
311 def x1_exp_var(kappa, theta, dt, vt, vT):
312     # compute  $E[X_1]$  and  $Var[X_1]$ 
313
314     aux = kappa*dt/2.
315     c1 = np.cosh(aux)/np.sinh(aux)
316     c2 = (1./np.sinh(aux))**2
317
318     exp_x1 = (vt + vT)*(c1/kappa - dt*c2/2)
319     var_x1 = (vt + vT)*theta**2*(c1/kappa**3 + dt*c2/(2*kappa**2) \
320         - dt**2*c1*c2/(2*kappa))
321
322     return exp_x1, var_x1
323
324 def lin_interp(vtvT, totems, caches_exp, caches_var):
325     # compute linear interpolation for value not in caches
326
327     exp_int = np.interp(vtvT, totems, caches_exp)
328     var_int = np.interp(vtvT, totems, caches_var)
329     return exp_int, var_int
330
331 def sample_vT(vt, dt, kappa, theta, nu_bar):
332     # sample vT from a noncentral chisquare (given vt)
333
334     aux = (theta**2*(1-np.exp(-kappa*dt)))/(4*kappa)
335     n = np.exp(-kappa*dt)/aux *vt
336     return np.random.noncentral_chisquare(nu_bar, n)*aux

```

```

336
337 def generate_path(S0, T, dt, kappa, eta, theta, rho, r, q, sigma_0,
338                  totems, caches_exp, caches_var):
339     # This function generate one path for the Heston model using
340     # the Gamma Approx algorithm
341
342     # T: final time
343     # r: risk-free-rate
344     # q: yield
345     # sigma_0, kappa, eta, theta, rho: Heston parameters
346     # S0: initial spot price
347     # dt: temporal step
348     # totems, caches_exp, caches_var: precomputed grid, caches for
349     # expectation and caches for var
350
351     t = dt
352     index = 0
353     vt = sigma_0
354     xt = np.log(S0)
355
356     path = np.zeros(int(np.ceil(T/dt))+1)
357     path[0] = S0
358
359     variance = np.zeros(int(np.ceil(T/dt))+1)
360     variance[0] = vt
361
362     nu_bar = calc_nu_bar(kappa, eta, theta)
363
364     while t < T:
365         vT = sample_vT(vt, dt, kappa, theta, nu_bar)
366
367         exp_int, var_int = lin_interp(vt*vT, totems, caches_exp,
368                                     caches_var)
369
370         exp_x1, var_x1 = x1_exp_var(kappa, theta, dt, vt, vT)
371         exp_int += exp_x1
372         var_int += var_x1
373
374         gamma_t = var_int/exp_int
375         gamma_k = exp_int**2/var_int
376
377         iv_t = np.random.gamma(gamma_k, gamma_t)
378         z = np.random.normal()
379
380         xt += (r-q)*dt + (- 0.5 + kappa*rho/theta)*iv_t + \
381              rho/theta*(vT-vt-kappa*eta*dt) + \
382              z*np.sqrt(1-rho**2)*np.sqrt(iv_t)
383
384         index += 1
385         path[index] = np.exp(xt)
386         vt = vT
387         variance[index] = vt
388         t += dt
389
390     return path[:-1], variance[:-1]

```

# Rough Heston

## rHeston.py

```
1 import numpy as np
2 import ImpliedDrift
3 import Heston
4 import BlackScholes
5 import scipy.integrate
6
7 from variance_curve import variance_curve, Gompertz
8 from scipy.special import gamma
9 from scipy.interpolate import CubicSpline
10 from scipy.stats import norm
11
12 du = 1e-4
13 GRID = np.linspace(0,10,int(1./du))
14
15
16 ##### Pade rHeston
17 #####
18
19 def Pade33(u, t, H, rho, theta):
20     alpha = H + 0.5
21
22     aa = np.sqrt(u * (u + (0+1j)) - rho**2 * u**2)
23     rm = -(0+1j) * rho * u - aa
24     rp = -(0+1j) * rho * u + aa
25
26     gamma1 = gamma(1+alpha)
27     gamma2 = gamma(1+2*alpha)
28     gammam1 = gamma(1-alpha)
29     gammam2 = gamma(1-2*alpha)
30
31     b1 = -u*(u+1j)/(2 * gamma1)
32     b2 = (1-u*1j) * u**2 * rho/(2* gamma2)
33     b3 = gamma2/gamma(1+3*alpha) * \
34         (u**2*(1j+u)**2/(8*gamma1**2)+(u+1j)*u**3*rho**2/(2*gamma2)
35         )
36
37     g0 = rm
38     g1 = -rm/(aa*gammam1)
39     g2 = rm/aa**2/gammam2 * \
40         (1 + rm/(2*aa)*gammam2/gammam1**2)
41
42     den = g0**3 +2*b1*g0*g1-b2*g1**2+b1**2*g2+b2*g0*g2
43
44     p1 = b1
45     p2 = (b1**2*g0**2 + b2*g0**3 + b1**3*g1 + b1*b2*g0*g1 - \
46         b2**2*g1**2 +b1*b3*g1**2 +b2**2*g0*g2 - b1*b3*g0*g2)/den
47     q1 = (b1*g0**2 + b1**2*g1 - b2*g0*g1 + b3*g1**2 - b1*b2*g2 -b3*
48         g0*g2)/den
49     q2 = (b1**2*g0 + b2*g0**2 - b1*b2*g1 - b3*g0*g1 + b2**2*g2 - b1
50         *b3*g2)/den
51     q3 = (b1**3 + 2*b1*b2*g0 + b3*g0**2 -b2**2*g1 +b1*b3*g1 )/den
52     p3 = g0*q3
```

```

50     y = t**alpha
51
52     return (p1*y + p2*y**2 + p3*y**3)/(1 + q1*y + q2*y**2 + q3*y
53         **3)
54
55 def DH_Pade33(u, x, H, rho, theta):
56     alpha = H + 0.5
57
58     aa = np.sqrt(u * (u + 1j) - rho**2 * u**2)
59     rm = -1j * rho * u - aa
60     rp = -1j * rho * u + aa
61
62     b1 = -u*(u+1j)/(2 * gamma(1+alpha))
63     b2 = (1-u*1j) * u**2 * rho/(2* gamma(1+2*alpha))
64     b3 = gamma(1+2*alpha)/gamma(1+3*alpha) * \
65         (u**2*(1j+u)**2/(8*gamma(1+alpha)**2)+(u+1j)*u**3*rho
66         **2/(2*gamma(1+2*alpha)))
67
68     g0 = rm
69     g1 = -rm/(aa*gamma(1-alpha))
70     g2 = rm/aa**2/gamma(1-2*alpha) * (1 + rm/(2*aa)*gamma(1-2*alpha)
71         )/gamma(1-alpha)**2)
72
73     den = g0**3 + 2*b1*g0*g1 - b2*g1**2 + b1**2*g2 + b2*g0*g2
74
75     p1 = b1
76     p2 = (b1**2*g0**2 + b2*g0**3 + b1**3*g1 + b1*b2*g0*g1 - b2**2*
77         g1**2 + b1*b3*g1**2 + b2**2*g0*g2 - b1*b3*g0*g2)/den
78     q1 = (b1*g0**2 + b1**2*g1 - b2*g0*g1 + b3*g1**2 - b1*b2*g2 - b3*
79         g0*g2)/den
80     q2 = (b1**2*g0 + b2*g0**2 - b1*b2*g1 - b3*g0*g1 + b2**2*g2 - b1
81         *b3*g2)/den
82     q3 = (b1**3 + 2*b1*b2*g0 + b3*g0**2 - b2**2*g1 + b1*b3*g1 )/den
83     p3 = g0*q3
84
85     y = x**alpha
86
87     hpade = (p1*y + p2*y**2 + p3*y**3)/(1 + q1*y + q2*y**2 + q3*y
88         **3)
89
90     res = 0.5*(hpade-rm)*(hpade-rp)
91
92     return res
93
94 def phi_rhest(u, t, H, rho, theta):
95     if u == 0:
96         return 1.
97
98     N = int(t*365)
99     alpha = H + 0.5
100    dt = t/N
101    tj = np.linspace(0,N,N+1,endpoint = True)*dt
102
103    x = theta**(1./alpha)*tj
104    xi = np.flip(variance_curve(tj))
105
106    aux = DH_Pade33(u, x, H, rho, theta)

```

```

100     return np.exp(np.matmul(aux,xi)*dt)
101
102
103 def DH_Pade33_vec(u, x, H, rho, theta):
104     alpha = H + 0.5
105
106     aa = np.sqrt(u * (u + 1j) - rho**2 * u**2)
107     rm = -1j * rho * u - aa
108     rp = -1j * rho * u + aa
109
110     b1 = -u*(u+1j)/(2 * gamma(1+alpha))
111     b2 = (1-u*1j) * u**2 * rho/(2* gamma(1+2*alpha))
112     b3 = gamma(1+2*alpha)/gamma(1+3*alpha) * \
113         (u**2*(1j+u)**2/(8*gamma(1+alpha)**2)+(u+1j)*u**3*rho
114         **2/(2*gamma(1+2*alpha)))
115
116     g0 = rm
117     g1 = -rm/(aa*gamma(1-alpha))
118     g2 = rm/aa**2/gamma(1-2*alpha) * (1 + rm/(2*aa)*gamma(1-2*alpha)
119     )/gamma(1-alpha)**2)
120
121     den = g0**3 +2*b1*g0*g1-b2*g1**2+b1**2*g2+b2*g0*g2
122
123     p1 = b1
124     p2 = (b1**2*g0**2 + b2*g0**3 + b1**3*g1 + b1*b2*g0*g1 - b2**2*
125     g1**2 +b1*b3*g1**2 +b2**2*g0*g2 - b1*b3*g0*g2)/den
126     q1 = (b1*g0**2 + b1**2*g1 - b2*g0*g1 + b3*g1**2 - b1*b2*g2 -b3*
127     g0*g2)/den
128     q2 = (b1**2*g0 + b2*g0**2 - b1*b2*g1 - b3*g0*g1 + b2**2*g2 - b1
129     *b3*g2)/den
130     q3 = (b1**3 + 2*b1*b2*g0 + b3*g0**2 -b2**2*g1 +b1*b3*g1 )/den
131     p3 = g0*q3
132
133     y = x**alpha
134     y2 = y**2
135     y3 = y**3
136
137     size_ = len(u)
138     Y = np.tile(y, (size_,1)).transpose()
139     Y2 = np.tile(y2, (size_,1)).transpose()
140     Y3 = np.tile(y3, (size_,1)).transpose()
141
142     hpade = (Y*p1 + Y2*p2 + Y3*p3)/(1 + Y*q1 + Y2*q2 + Y3*q3)
143
144     res = 0.5*(hpade-rm)*(hpade-rp)
145
146     return res
147
148 def phi_rhest_vec(u, t, H, rho, theta, N = 1000):
149
150     mask = (u == 0)
151
152     alpha = H + 0.5
153     dt = t/N
154     tj = np.linspace(0,N,N+1,endpoint = True)*dt
155
156     x = theta**(1./alpha)*tj

```

```

152     xi = np.flip(variance_curve(tj))
153
154     res = np.zeros(len(u), dtype = complex)
155
156     if mask.any():
157         aux = DH_Pade33_vec(u[~mask], x, H, rho, theta)
158         res[~mask] = np.exp(np.matmul(xi,aux)*dt)
159         res[mask] = 1.
160     else:
161         aux = DH_Pade33_vec(u, x, H, rho, theta)
162         res = np.exp(np.matmul(xi,aux)*dt)
163
164     return res
165
166 # ##### Analytic rHeston
167 # #####
168 def integral(x, t, H, rho, theta):
169
170     integrand = (lambda u: np.real(np.exp((1j*u)*x) * \
171                                     phi_rhest(u - 0.5j, t, H, rho,
172                                     theta)) / \
173                 (u**2 + 0.25))
174
175     i, err = scipy.integrate.quad_vec(integrand, 0, np.inf)
176
177     return i
178
179 # def integral_vec(x, t, H, rho, theta, grid = GRID):
180 #     aux = (np.tile(grid, (len(x),1)).transpose()*x).transpose()
181 #     i = np.real(np.exp(1j*aux)*phi_rhest_vec(grid - 0.5j, t, H,
182 #     rho, theta)) / \
183 #         (grid**2 + 0.25)
184 #     i = i.sum(axis = 1)*(grid[1]-grid[0])
185 #     return i
186
187 def analytic_rhest(S0, strikes, t, H, rho, theta, options_type):
188
189     # Pricing of vanilla options under "analytic" rHeston using
190     # Lewis Formula
191
192     a = np.log(S0/strikes) + ImpliedDrift.drift(t)*t
193     i = integral(a, t, H, rho, theta)
194     r = ImpliedDrift.r(t)
195     q = ImpliedDrift.q(t)
196     out = S0 * np.exp(-q*t) - np.sqrt(S0*strikes) * np.exp(-(r+q)*t
197     *0.5)/np.pi * i
198     out = np.array([out]).flatten()
199
200     for k in range(len(options_type)):
201         if options_type[k] == 0:
202             out[k] = Heston.call_put_parity(out[k], S0, strikes[k],
203             r, q, t)
204
205     if (out < 0).any():

```

```

203         out[out < 0] = 0.
204
205     return out
206
207 # def analytic_rhest_vec(S0, strikes, t, H, rho, theta,
208     options_type):
209 #     # Pricing of vanilla options under "analytic" rHeston using
210     # Lewis Formula
211 #     a = np.log(S0/strikes) + ImpliedDrift.drift(t)*t
212 #     i = integral_vec(a, t, H, rho, theta)
213 #     r = ImpliedDrift.r(t)
214 #     q = ImpliedDrift.q(t)
215 #     out = S0 * np.exp(-q*t) - np.sqrt(S0*strikes) * np.exp(-(r+q)
216 #         *t*0.5)/np.pi * i
217 #     out = np.array([out]).flatten()
218 #     for k in range(len(options_type)):
219 #         if options_type[k] == 0:
220 #             out[k] = Heston.call_put_parity(out[k], S0, strikes[k]
221 #                 ], r, q, t)
222 #     if (out < 0).any():
223 #         out[out < 0] = 0.
224
225 #     return out
226
227 #####Simulation rHeston
228 #####
229 # Psi for the QE Scheme of Lemma 7.
230 def psi_m(psi, ev, w):
231     #psi minus
232
233     beta2 = psi
234     mask = psi > 0
235     mask1 = psi <= 0
236     if np.any(mask):
237         beta2[mask] = 2./psi[mask]-1+np.sqrt(2./psi[mask]* \
238             np.abs(2./psi[mask]-1)
239         )
240     if np.any(mask1):
241         beta2[mask1] = 0.
242     return ev/(1+beta2)*(np.sqrt(np.abs(beta2))+w)**2
243
244 def psi_p(psi, ev, u):
245     #psi plus
246
247     p = 2/(1+psi)
248     res = (u<p)*(-ev)/2*(1+psi)
249     mask = u > 0
250     if np.any(mask):
251         res[mask] = np.log(u[mask]/p[mask])
252     return res
253
254 # functions for K_i, K_ii and K_01

```

```

254 def Gi(eta, alpha, dt, i):
255     return np.sqrt(2*alpha-1)*eta/alpha * dt**alpha * ((i+1)**alpha
      - i**alpha)
256
257 def Gii(eta, H, dt, i):
258     aux = 2*H
259     return eta**2 * dt**aux * ((i+1)**aux - i**aux)
260
261 def G01(eta, alpha, dt):
262     return Gi(eta,alpha,dt,0)*Gi(eta,alpha,dt,1)/dt
263
264 def HQE_sim(theta, H, rho, T, S0, paths, steps, eps0 = 1e-10):
265     # HQE scheme
266
267     # theta, H, rho: parameters of the rHeston model
268     # T: final time of the simulations, in years
269     # S0: spot price at time 0
270     # paths: number of paths to simulate
271     # steps: number of timesteps between 0 and T
272     # eps0: lower bound for xihat
273
274     dt = T/steps
275     dt_sqrt = np.sqrt(dt)
276     alpha = H + 0.5
277     eta = theta/(gamma(alpha)*np.sqrt(2*H))
278     rho2m1 = np.sqrt(1-rho*rho)
279
280     W = np.random.normal(0.,1.,size = (steps,paths))
281     Wperp = np.random.normal(0.,1.,size = (steps,paths))
282     Z = np.random.normal(0.,1.,size = (steps,paths))
283     U = np.random.uniform(0.,1.,size = (steps,paths))
284     Uperp = np.random.uniform(0.,1.,size = (steps,paths))
285
286     tj = np.arange(0,steps,1)*dt
287     tj += dt
288
289     xij = variance_curve(tj)
290     G0del = Gi(eta,alpha,dt,0)
291     G00del = Gii(eta,alpha,dt,0)
292     G11del = Gii(eta,alpha,dt,1)
293     G01del = G01(eta,alpha,dt)
294     G00j = np.zeros(steps)
295
296     for j in range(steps):
297         G00j[j] = Gii(eta,H,dt,j)
298     bstar = np.sqrt((G00j)/dt)
299
300     rho_vchi = G0del/np.sqrt(G00del*dt)
301     beta_vchi = G0del/dt
302
303     u = np.zeros((steps,paths))
304     chi = np.zeros((steps,paths))
305     v = np.ones(paths)*variance_curve(0)
306     hist_v = np.zeros((steps,paths))
307     hist_v[0,:] = v
308     xihat = np.ones(paths)*xij[0]
309     x = np.zeros((steps,paths))

```



```

310 y = np.zeros(paths)
311 w = np.zeros(paths)
312
313 for j in range(steps):
314     xibar = (xihat + 2*H*v)/(1+2*H)
315
316     psi_chi = 2*beta_vchi*xibar*dt/(xihat**2)
317     psi_eps = 2/(xihat**2)*xibar*(G0del - G0del**2/dt)
318     aux_ = xihat/2
319
320     z_chi = np.zeros(paths)
321     z_eps = np.zeros(paths)
322
323     mask1 = psi_chi < 1.5
324     mask2 = psi_chi >= 1.5
325     mask3 = psi_eps < 1.5
326     mask4 = psi_eps >= 1.5
327
328     if np.any(mask1):
329         z_chi[mask1] = psi_m(psi_chi[mask1],aux_[mask1],W[j,
mask1])
330     if np.any(mask2):
331         z_chi[mask2] = psi_m(psi_chi[mask2],aux_[mask2],U[j,
mask2])
332     if np.any(mask3):
333         z_eps[mask3] = psi_m(psi_eps[mask3],aux_[mask3],Wperp[j
,mask3])
334     if np.any(mask4):
335         z_eps[mask4] = psi_m(psi_eps[mask4],aux_[mask4],Uperp[j
,mask4])
336
337     chi[j,:] = (z_chi-aux_)/beta_vchi
338     eps = z_eps - aux_
339     u[j,:] = beta_vchi*chi[j,:]+eps
340     vf = xihat + u[j,:]
341     vf[vf < eps0] = eps0
342
343     dw = (v+vf)/2*dt
344     w += dw
345     y += chi[j,:]
346     x[j,:] = x[j-1,:] + ImpliedDrift.drift(T)*dt - dw/2 + np.
sqrt(dw) \
347         * (rho2m1*Z[j,:]) + rho*chi[j,:]
348
349     btilde = np.flip(bstar[1:j+1])
350     if j < steps-1:
351         xihat = xij[j+1] + (np.matmul(btilde,chi[:j,:]))
352     v = vf
353     hist_v[j,:] = v
354     return np.vstack((np.ones(paths)*S0,(np.exp(x)*S0)),hist_v

```

## Rough Bergomi

### utils\_rBergomi.py

```

1 import numpy as np

```

```

2
3 # TBSS kernel applicable to the rBergomi variance process.
4 def g(x, a):
5     return x**a
6
7 # Optimal discretisation of TBSS process for minimising hybrid
8   scheme error.
9 def b(k, a):
10     return ((k**(a+1)-(k-1)**(a+1))/(a+1))**(1/a)
11
12 # Covariance matrix for given alpha and n, assuming kappa = 1.
13 def cov(a, n):
14     cov = np.array([[0.,0.],[0.,0.]])
15     cov[0,0] = 1./n
16     cov[0,1] = 1./((1.*a+1) * n**(1.*a+1))
17     cov[1,1] = 1./((2.*a+1) * n**(2.*a+1))
18     cov[1,0] = cov[0,1]
19     return cov

```

## rbergomi.py

```

1 import numpy as np
2 from scipy.signal import convolve
3 from numpy.random import default_rng
4 from utils_rBergomi import *
5 import ImpliedDrift as iD
6
7 # Class for generating paths of the rBergomi model.
8 class rBergomi(object):
9
10     def __init__(self, n, N, T, a):
11
12         # Basic assignments
13         self.T = T
14         # Maturity
15         self.n = n
16         # Steps per year
17         self.dt = 1.0/self.n
18         # Step size
19         self.s = np.round(self.n * self.T).astype(int)
20         # Number of total steps
21         self.t = np.linspace(0, self.T, 1 + self.s)[np.newaxis,:]
22         # Time grid
23         self.a = a
24         # Alpha
25         self.N = N
26         # Number of paths
27
28         # Construct hybrid scheme correlation structure with kappa
29         = 1
30         self.e = np.array([0,0])
31         self.c = cov(self.a, self.n)
32
33     def dW1(self):
34         np.random.seed(0)
35         # Produces random numbers for variance process with
36         required covariance structure

```

```

28     N = int(self.N/2)
29     w = np.random.multivariate_normal(self.e, self.c, (N, self.
    s))
30     return np.concatenate((w,-w), axis = 0)
31
32 def dW2(self):
33     np.random.seed(0)
34     #Obtain orthogonal increments
35     N = int(self.N/2)
36     w = np.random.randn(N, self.s) * np.sqrt(self.dt)
37     return np.concatenate((w,-w), axis = 0)
38
39 def Y(self, dW):
40     #Constructs Volterra process from appropriately correlated
    2d Brownian increments
41
42     Y1 = np.zeros((self.N, 1 + self.s)) # Exact integrals
43     Y2 = np.zeros((self.N, 1 + self.s)) # Riemann sums
44
45     Y1[:,1 : self.s+1] = dW[:, :self.s, 1] # Assumes kappa =
    1
46
47     # Construct arrays for convolution
48     G = np.zeros(1 + self.s) # Gamma
49     for k in np.arange(2, 1 + self.s, 1):
50         G[k] = g(b(k, self.a)/self.n, self.a)
51
52     X = dW[:, :, 0] # Xi
53
54     # Compute convolution and extract relevant terms
55     for i in range(self.N):
56         Y2[i, :] = np.convolve(G, X[i, :])[:1+self.s]
57
58     # Finally construct and return full process
59     return np.sqrt(2 * self.a + 1) * (Y1 + Y2)
60
61 def dZ(self, dW1, dW2, rho):
62     # Constructs correlated price Brownian increments, dB
63
64     self.rho = rho
65     return rho * dW1[:, :, 0] + np.sqrt(1 - rho**2) * dW2
66
67 def V(self, Y, xi, eta):
68     # rBergomi variance process.
69     self.xi = xi
70     self.eta = eta
71     a = self.a
72     t = self.t
73     return xi * np.exp(eta * Y - 0.5 * eta**2 * t**(2 * a + 1))
74     #return xi * ne.evaluate('exp(eta * Y - 0.5 * eta**2 * t
    **(2 * a + 1))')
75
76 def S_all_path(self, V, dZ, r, q, S0):
77     # rBergomi price process.
78     self.S0 = S0
79     dt = self.dt
80     rho = self.rho

```

```

81
82     # Construct non-anticipative Riemann increments
83     increments = np.sqrt(V[:, :-1]) * dZ - 0.5 * V[:, :-1] * dt +
(r - q) * dt
84     integral = np.cumsum(increments, axis = 1)
85
86     S = np.zeros_like(V)
87     S[:, 0] = S0
88     S[:, 1:] = S0 * np.exp(integral)
89     return S
90
91 def S(self, V, dZ, r, q, S0):
92     # rBergomi price process.
93     self.S0 = S0
94     dt = self.dt
95     rho = self.rho
96
97     # Construct non-anticipative Riemann increments
98     exponent = np.zeros(self.N)
99     for i in range(self.s):
100         exponent += np.sqrt(V[:, i]) * dZ[:, i] - 0.5 * V[:, i] *
dt + (r - q) * dt
101
102     return S0 * np.exp(exponent)
103
104 def global_S(self, V, dZ, S0, steps, index = 0):
105     # rBergomi price process.
106     self.S0 = S0
107     dt = self.dt
108     rho = self.rho
109
110     r = iD.r(self.t[0], index)
111     q = iD.q(self.t[0], index)
112
113     S = list()
114     logS = np.log(S0)
115     for i in range(self.s):
116         logS += np.sqrt(V[:, i]) * dZ[:, i] - 0.5 * V[:, i] * dt +
(r[i] - q[i]) * dt
117         if i in steps:
118             S.append(np.exp(logS))
119
120     S.append(np.exp(logS))
121
122     return np.array(S)

```

## Quintic Ornstein-Uhlenbeck

```

1 import numpy as np
2 import variance_curve as vc
3 import ImpliedDrift as iD
4 import scipy
5 import BlackScholes as bs
6
7 from scipy.integrate import quad
8
9 def horner_vector(poly, n, x):

```

```

10 #Initialize result
11 result = poly[0].reshape(-1,1)
12 for i in range(1,n):
13     result = result*x + poly[i].reshape(-1,1)
14 return result
15
16
17
18 def gauss_dens(mu,sigma,x):
19     return 1/np.sqrt(2*np.pi*sigma**2)*np.exp(-(x-mu)**2/(2*sigma
20 **2))
21
22
23 def vix_futures(H, eps, T, a_k_part, k, r, q, n_steps, index = 0):
24
25     a2,a4 = (0,0)
26     a0,a1,a3,a5 = a_k_part
27     a_k = np.array([a0, a1, a2, a3, a4, a5])
28
29     kappa_tild = (0.5-H)/eps
30     eta_tild = eps**(H-0.5)
31
32     delt = 30/365
33     T_delta = T + delt
34
35     std_X = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*
36 kappa_tild*T)))
37     dt = delt/(n_steps)
38     tt = np.linspace(T, T_delta, n_steps+1)
39
40     FV_curve_all_vix = vc.variance_curve(tt, index)
41
42     exp_det = np.exp(-kappa_tild*(tt-T))
43     cauchy_product = np.convolve(a_k,a_k)
44
45     std_Gs_T = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*
46 kappa_tild*(tt-T))))
47     std_X_t = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*
48 kappa_tild*tt)))
49     std_X_T = std_X
50
51     n = len(a_k)
52
53     normal_var = np.sum(cauchy_product[np.arange(0,2*n,2)].reshape
54 (-1,1)*std_X_t**(np.arange(0,2*n,2).reshape(-1,1))*\
55 scipy.special.factorial2(np.arange(0,2*n,2).reshape(-1,1)-1),
56 axis=0) #g(u)
57
58     beta = []
59     for i in range(0,2*n-1):
60         k_array = np.arange(i,2*n-1)
61         beta_temp = ((std_Gs_T**((k_array-i).reshape(-1,1))*((
62 k_array-i-1)%2).reshape(-1,1))*\
63         scipy.special.factorial2(k_array-i-1).reshape(-1,1))*\
64         (scipy.special.comb(k_array,i)).reshape(-1,1))*\
65         exp_det**(i))*cauchy_product[k_array].reshape(-1,1)

```

```

60     beta.append(np.sum(beta_temp,axis=0))
61
62     beta = np.array(beta)*FV_curve_all_vix/normal_var
63     beta = (np.sum((beta[:, :-1]+beta[:, 1:])/2,axis=1))*dt
64
65     sigma = np.sqrt(eps**(2*H)/(1-2*H)*(1-np.exp((2*H-1)*T/eps)))
66
67     f = lambda x: np.sqrt(horner_vector(beta[:, :-1], len(beta), x)/
68                             deltt)*100 * gauss_dens(0, sigma, x)
69
70     Ft, err = quad(f, -np.inf, np.inf)
71
72     return Ft * np.exp((r-q)*T)
73
74
75 def vix_iv(H, eps, T, a_k_part, K, r, q, n_steps, index = 0):
76
77     a2,a4 = (0,0)
78     a0,a1,a3,a5 = a_k_part
79     a_k = np.array([a0, a1, a2, a3, a4, a5])
80
81     kappa_tild = (0.5-H)/eps
82     eta_tild = eps**(H-0.5)
83
84     deltt = 30/365
85     T_delta = T + deltt
86
87     std_X = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*
88                             kappa_tild*T)))
89     dt = deltt/(n_steps)
90     tt = np.linspace(T, T_delta, n_steps+1)
91
92     FV_curve_all_vix = vc.variance_curve(tt, index)
93
94     exp_det = np.exp(-kappa_tild*(tt-T))
95     cauchy_product = np.convolve(a_k,a_k)
96
97     std_Gs_T = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*
98                             kappa_tild*(tt-T))))
99     std_X_t = eta_tild*np.sqrt(1/(2*kappa_tild)*(1-np.exp(-2*
100                             kappa_tild*tt)))
101     std_X_T = std_X
102
103     n = len(a_k)
104
105     normal_var = np.sum(cauchy_product[np.arange(0,2*n,2)].reshape
106                             (-1,1)*std_X_t**((np.arange(0,2*n,2).reshape(-1,1))*\
107                             scipy.special.factorial2(np.arange(0,2*n,2).reshape(-1,1)-1),
108                             axis=0) #g(u)
109
110     beta = []
111     for i in range(0,2*n-1):
112         k_array = np.arange(i,2*n-1)
113         beta_temp = ((std_Gs_T**((k_array-i).reshape(-1,1))*((
114             k_array-i-1)%2).reshape(-1,1))*\
115             scipy.special.factorial2(k_array-i-1).reshape(-1,1))*\

```

```

110         (scipy.special.comb(k_array,i)).reshape(-1,1))*\
111         exp_det**(i))*cauchy_product[k_array].reshape(-1,1)
112     beta.append(np.sum(beta_temp,axis=0))
113
114     beta = np.array(beta)*FV_curve_all_vix/normal_var
115     beta = (np.sum((beta[:, :-1]+beta[:, 1:])/2,axis=1))*dt
116
117     sigma = np.sqrt(eps**(2*H)/(1-2*H)*(1-np.exp((2*H-1)*T/eps)))
118
119     N = len(K); P = np.zeros(N);
120
121     for i in range(N):
122
123         f = lambda x: np.maximum(np.sqrt(horner_vector(beta[:, :-1],
124         len(beta), x)/delt)*100 - K[i], 0) * gauss_dens(0, sigma, x)
125         P[i], err = quad(f, -np.inf, np.inf)
126
127     return P * np.exp((r-q)*T)
128
129
130 def dW(n_steps,N_sims):
131     w = np.random.normal(0, 1, (n_steps, N_sims))
132     #Antithetic variates
133     w = np.concatenate((w, -w), axis = 1)
134     return w
135
136
137
138 def local_reduction(rho,H,eps,T,a_k_part,S0,strike_array,n_steps,
139 N_sims,w1,r,q, index = 0):
140
141     eta_tild = eps**(H-0.5)
142     kappa_tild = (0.5-H)/eps
143
144     a_0,a_1,a_3,a_5 = a_k_part
145     a_k = np.array([a_0,a_1,0,a_3,0,a_5])
146
147     dt = T/n_steps
148     tt = np.linspace(0., T, n_steps + 1)
149
150     exp1 = np.exp(kappa_tild*tt)
151     exp2 = np.exp(2*kappa_tild*tt)
152
153     diff_exp2 = np.concatenate((np.array([0.]),np.diff(exp2)))
154     std_vec = np.sqrt(diff_exp2/(2*kappa_tild))[:,np.newaxis] #to
155     be broadcasted columnwise
156     exp1 = exp1[:,np.newaxis]
157     X = (1/exp1)*(eta_tild*np.cumsum(std_vec*w1, axis = 0))
158     Xt = np.array(X[:-1])
159     del X
160
161     tt = tt[:-1]
162     std_X_t = np.sqrt(eta_tild**2/(2*kappa_tild)*(1-np.exp(-2*
163     kappa_tild*tt)))
164     n = len(a_k)

```

```

163     cauchy_product = np.convolve(a_k,a_k)
164     normal_var = np.sum(cauchy_product[np.arange(0,2*n,2)].reshape
165         (-1,1)*std_X_t**(np.arange(0,2*n,2).reshape(-1,1))*\
166         scipy.special.factorial2(np.arange(0,2*n,2).reshape(-1,1)
167         -1),axis=0)
168
169     f_func = horner_vector(a_k[::-1], len(a_k), Xt)
170
171     del Xt
172
173     fv_curve = vc.variance_curve(tt, index).reshape(-1,1)
174
175     volatility = f_func/np.sqrt(normal_var.reshape(-1,1))
176     del f_func
177     volatility = np.sqrt(fv_curve)*volatility
178
179     logS1 = np.log(S0)
180     for i in range(w1.shape[0]-1):
181         logS1 = logS1 - 0.5*dt*(volatility[i]*rho)**2 + np.sqrt(dt)
182         *rho*volatility[i]*w1[i+1] + rho**2*(r-q)*dt
183     del w1
184     ST1 = np.exp(logS1)
185     del logS1
186
187     int_var = np.sum(volatility[:-1]**2*dt,axis=0)
188     Q = np.max(int_var)+1e-9
189     del volatility
190     X = (bs.BSCall(ST1, strike_array.reshape(-1,1), T, r, q, np.
191         sqrt((1-rho**2)*int_var/T))).T
192     Y = (bs.BSCall(ST1, strike_array.reshape(-1,1), T, r, q, np.
193         sqrt(rho**2*(Q-int_var)/T))).T
194     del int_var
195     eY = (bs.BSCall(S0, strike_array.reshape(-1,1), T, r, q, np.
196         sqrt(rho**2*Q/T))).T
197
198     c = []
199     for i in range(strike_array.shape[0]):
200         cova = np.cov(X[:,i]+10,Y[:,i]+10)[0,1]
201         varg = np.cov(X[:,i]+10,Y[:,i]+10)[1,1]
202         if (cova or varg)<1e-8:
203             temp = 1e-40
204         else:
205             temp = np.nan_to_num(cova/varg,1e-40)
206         temp = np.minimum(temp,2)
207         c.append(temp)
208     c = np.array(c)
209
210     call_mc_cv1 = X-c*(Y-eY)
211     del X
212     del Y
213     del eY
214
215     return np.average(call_mc_cv1,axis=0)
216
217 def global_reduction(rho,H,eps,T,a_k_part,S0,strike_array,n_steps,

```



```

N_sims,w1,steps,maturities, index = 0):
214
215     eta_tild = eps**(H-0.5)
216     kappa_tild = (0.5-H)/eps
217
218     a_0,a_1,a_3,a_5 = a_k_part
219     a_k = np.array([a_0,a_1,0,a_3,0,a_5])
220
221     dt = T/n_steps
222     tt = np.linspace(0., T, n_steps + 1)
223
224     r = iD.r(tt, index)
225     q = iD.q(tt, index)
226
227     exp1 = np.exp(kappa_tild*tt)
228     exp2 = np.exp(2*kappa_tild*tt)
229
230     diff_exp2 = np.concatenate((np.array([0.]),np.diff(exp2)))
231     std_vec = np.sqrt(diff_exp2/(2*kappa_tild))[:,np.newaxis] #to
    be broadcasted columnwise
232     exp1 = exp1[:,np.newaxis]
233     X = (1/exp1)*(eta_tild*np.cumsum(std_vec*w1, axis = 0))
234     Xt = np.array(X[:-1])
235     del X
236
237     tt = tt[:-1]
238     std_X_t = np.sqrt(eta_tild**2/(2*kappa_tild)*(1-np.exp(-2*
    kappa_tild*tt)))
239     n = len(a_k)
240
241     cauchy_product = np.convolve(a_k,a_k)
242     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))*\
243         scipy.special.factorial2(np.arange(0,2*n,2).reshape(-1,1)
    -1),axis=0)
244
245     f_func = horner_vector(a_k[:-1], len(a_k), Xt)
246
247     del Xt
248
249     fv_curve = vc.variance_curve(tt, index).reshape(-1,1)
250
251     volatility = f_func/np.sqrt(normal_var.reshape(-1,1))
252     del f_func
253     volatility = np.sqrt(fv_curve)*volatility
254
255     ST1 = list()
256     logS1 = np.log(S0)
257     for i in range(w1.shape[0]-1):
258         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
259         if i in steps:
260             ST1.append(np.exp(logS1))
261     del w1
262     ST1.append(np.exp(logS1))
263     ST1 = np.array(ST1)
264     del logS1

```

```

265     int_var = np.sum(volatility[:-1,]**2*dt,axis=0)
266     Q = np.max(int_var)+1e-9
267     del volatility
268
269     P = list()
270
271     for i in range(len(steps)):
272         T_aux = maturities[i]
273         r = iD.r(T_aux, index); q = iD.q(T_aux, index)
274
275         X = (bs.BSCall(ST1[i], strike_array.reshape(-1,1), T_aux, r
276 , q, np.sqrt((1-rho**2)*int_var/T))).T
277         Y = (bs.BSCall(ST1[i], strike_array.reshape(-1,1), T_aux, r
278 , q, np.sqrt(rho**2*(Q-int_var)/T))).T
279         eY = (bs.BSCall(S0, strike_array.reshape(-1,1), T_aux, r, q
280 , np.sqrt(rho**2*Q/T))).T
281
282         c = []
283         for i in range(strike_array.shape[0]):
284             cov = np.cov(X[:,i]+10,Y[:,i]+10)[0,1]
285             var = np.cov(X[:,i]+10,Y[:,i]+10)[1,1]
286             if (cov or var)<1e-8:
287                 temp = 1e-40
288             else:
289                 temp = np.nan_to_num(cov/var,1e-40)
290             temp = np.minimum(temp,2)
291             c.append(temp)
292         c = np.array(c)
293
294         call_mc_cv1 = X-c*(Y-eY)
295         P.append(np.average(call_mc_cv1,axis=0))
296
297     return np.array(P)

```