Computational Methods In Finance Project

André Lorenzo Bittencourt

1 Stochastic Differential Equation

Consider below Local Volatility Model:

$$dS(t) = \mu S(t)dt + \sigma(S(t),t)S(t)dW(t) \tag{1.1}$$

$$\sigma(S(t),t) = \sigma_0 + \sigma_1 cos\left(\frac{2\pi S}{K}\right)sin\left(\frac{2\pi t}{K}\right) \tag{1.2}$$

The volatility model has no real economic meaning, it's just for learning purposes.

1.1 Theory

1.1.1 Milstein Method

The Milstein Method is a numerical method to solve Stochastic Differential Equation by its discretization. It is a refinement of the Euler-Maruyama Method with a additional second order term. The method has a Strong and Weak Orders of Convergence of 1, while Euler-Maruyama has $\frac{1}{2}$ and 1, respectively.

The iteration is given by:

$$X_{t+1} = X_t + \alpha(S(t),t)\Delta t + \beta(S(t),t)\Delta W_t + \frac{1}{2}\beta(S(t),t)\frac{\partial\beta(S(t),t)}{\partial S(t)}(\Delta W_t^2 - h) \tag{1.3}$$

For computational simulation purposes:

$$\begin{aligned}\Delta t &= h\\ \Delta W_t &= \xi_t\sqrt{h}\\ \xi_t &\sim N(0,1)\end{aligned} \tag{1.4}$$

For the model described in 1.1 and 1.2:

$$\begin{aligned}\alpha(S(t),t) &= \mu S(t)\\ \beta(S(t),t) = \sigma(S(t),t) &= \left(\sigma_0 + \sigma_1 cos\left(\frac{2\pi S}{K}\right)sin\left(\frac{2\pi t}{K}\right)\right)S(t)\\ \beta'(S(t),t) &= -\frac{2\pi\sigma_1}{K}sin\left(\frac{2\pi S(t)}{K}\right)sin\left(\frac{2\pi t}{K}\right) + \sigma(S(t),t)\end{aligned} \tag{1.5}$$

1.1.2 Talay-Tubaro

The Talay-Tubaro technique is used to approximate Expectations of functions of Random Variables. Consider $X(T)$ the true value of random variable in T, X_N^h the approximation of $X(T)$ with steps of size h, at N-th point ($T = N * h$ and ϕ any function. It can be shown:

$$\mathbb{E}[\phi(X(T))] - \mathbb{E}[\phi(X_N^h)] = c_1h + c_2h^2 + c_3h^3 + ... = O(h) \tag{1.6}$$

The approximation can be enhanced using a thinner grid:

$$\mathbb{E}[\phi(X(T))] - \mathbb{E}[\phi(X_{2N}^{h/2})] = c_1\frac{h}{2} + c_2\frac{h^2}{2} + c_3\frac{h^2}{2} + ... = O(h) \tag{1.7}$$

The Talay-Tubaro insight is:

$$\mathbb{E}[\phi(X(T))] - (2\mathbb{E}[\phi(X_{2N}^{h/2})] - \mathbb{E}[\phi(X_N^h)]) = -c^2\frac{h^2}{2} + ... = O(h^2) \tag{1.8}$$

Reducing the error to $O(h^2)$ using two process with error $O(h)$.

1.1.3 Numerical Integration

To Numeral Integration can be achived with simple quadrature:

$$\int_a^b f(x)dx = \sum_{i=1}^{N} f(x_i)a_i \tag{1.9}$$

In the Model:

$$\frac{1}{T}\int_0^T S(u)du = \frac{1}{T}\sum_{i=1}^{N} X_{u_i}h = \frac{h}{T}\sum_{i=1}^{N} X_{u_i} = \frac{1}{N}\sum_{i=1}^{N} X_{u_i} \tag{1.10}$$

Which is simple the X_t average.

1.2 Implementation

The solution was implemented in the *OptionMilstein* Class. In the class initialization, each input variable is saved with respective model name. The variable M for the number of simulations and *precision* to control the precision of discretization.

```
def __init__(self, mu, sigma0, sigma1, k, s0, T, r, M, precision = 1):
    self.mu = mu
    self.sigma0 = sigma0
    self.sigma1 = sigma1
    self.k = k
    self.s0 = s0
    self.T = T
    self.r = r
    self.M = M # Nº of simulations
    self.h = 1/(precision*365) #Timesteps size
```

1.2.1 Milstein Method

The parameterization of 1.5 were defined in:

```
def a(self, s):
    return s * self.mu

def b(self, s, t):
    return s*(self.sigma0 + self.sigma1 * math.cos((2*math.pi*s)/self.k) * \
            math.sin((2*math.pi*t)/self.k))

def db(self, s, t):
    return -((2*math.pi)/self.k) * s * math.sin((2*math.pi*s)/self.k) * \
            math.sin((2*math.pi*t)/self.k) * self.sigma1 + b(s,t)/s
```

And the Milstein discretization in:

```
def milstein(self, dw, h):

    lenght = len(dw)
    path = np.zeros(lenght)
    path[0] = self.s0

    for i in range(lenght-1):

        t = (i+1) * h
        path[i+1] = path[i] + self.a(path[i]) * h + self.b(path[i], t) * \
            np.sqrt(h) * dw[i] + (0.5 * self.b(path[i], t) * \
                self.b(path[i], t) * (h * dw[i]*dw[i] - h))

    return path
```

1.2.2 Talay-Tubaro

The Talay-Tubaro was implemented having the ϕ function as input, allowing for different functions with little new code. It also save the path simulations for reuse in other methods of the Class, saving time to price different types of options, for the same parameters. Lastly, the same brownian vector is used for both discretization levels, reducing variance and computational cost.

```python
def talayTubaro(self,function):

    v1 =  np.zeros(self.M) #Expectation with finer disctretization
    v2 =  np.zeros(self.M)

    if not hasattr(self,'p1'): #Check if the paths were already created.

        size1 = 2*int(self.T/self.h)
        size2 = int(self.T/self.h)

        self.p1 = np.empty([self.M, size1])
        self.p2 = np.empty([self.M, size2])

        for i in range(self.M):

            dw1 = np.random.standard_normal(size1) #Brownian Motion
            #Aggregated dW. Remembering to normalize by sqrt(2)
            # for variance = 1
            dw2 = [(dw1[2*j]+dw1[2*j+1])/math.sqrt(2) for j in range(size2)]

            self.p1[i] = self.milstein(dw1, self.h*0.5)
            self.p2[i] = self.milstein(dw2, self.h)

    v1 = np.array([function(self.p1[i]) for i in range(self.M)])
    v2 = np.array([function(self.p2[i]) for i in range(self.M)])

    return 2*v1.mean() - v2.mean()
```

1.2.3 Additional Functions

The payoff functions of European Call and Asian Call (the ϕ functions) are defined. Other payoffs can be easily implemented.

```python
def callPayoff(self, s):

    if hasattr(s, "__len__"):
        return np.maximum(s[-1]-self.k,0)
    else:
        return np.maximum(s-self.k,0)

def asianPayoff(self, s):

    return np.maximum(s.mean()-self.k,0)

def callPrice(self):

    return self.talayTubaro(self.callPayoff)

def asianPrice(self):

    return self.talayTubaro(self.asianPayoff)
```

1.2.4 Results

Below the result of some simulations. For $\sigma_1 = 0$ and $\mu = r$ the problem reduce to the classical Black-Scholes and the result is compared with the analytical solution.

Parameters	1	2	3	4
mu	5%	5%	10%	5%
σ_0	20%	20%	15%	20%
σ_1	0%	30%	40%	0
Strike	10	10	10	12
S_0	10	10	15	10
T	1	2	1	1
r	5%	5%	5%	5%
M	2000	2000	2000	2000
Call Price	1.10316	2.17194	6.62976	0.33502
Asian Price	0.60258	1.16806	5.76397	0.05350

Table 1: Example Values. Precision = 1

Parameters	1	2	3	4
mu	5%	5%	5%	5%
σ_0	20%	20%	20%	20%
σ_1	0%	0%	0%	0
Strike	10	10	10	10
S_0	10	10	10	10
T	1	1	1	1
r	5%	5%	5%	5%
M	1000	5000	1000	5000
Precision	1	1	2	2
Call Price	1.01594	1.07780	1.10998	1.06875
Analítico	1.04506	1.04506	1.04506	1.04506

Table 2: Convergence Analysis.

Pode-se notar que a convergência da técnica é um pouco lenta, por isso é interessante implementar em linguagens mais eficientes como C++ ou ainda incluir outras técnicas de redução de variância como Variedades Antitéticas, por exemplo.

2 Partial Diffenrential Equation

Consider the Local Volatility Model described by the PDE:

$$\frac{\partial C}{\partial t} - \frac{1}{2}\sigma(S(t),t)S^2\frac{\partial^2 C}{\partial S^2} - rS\frac{\partial C}{\partial S} + rC = 0 \tag{2.1}$$

With boundary conditions:

$$\begin{aligned} C(S,0) &= (S(0)-K)^+ \\ C(0,t) &= 0 \\ C(S_{max},t) &= L_t = S_{max} - Ke^{-rt} \end{aligned} \tag{2.2}$$

And variance:

$$\sigma(S(t),t) = \sigma_0 + \sigma_1 cos\left(\frac{2\pi t}{T}\right) exp\left(-\left(\frac{S}{K}-1\right)^2\right) \tag{2.3}$$

Note the model is stated in backward time, with maturity in t=0. It is also stated in variance, not standard deviation as usual.

2.1 Theory

2.1.1 FTCS

The FTCS method (*Forward-Time Centered-Space*) is a numerical procedure to solve PDEs. The time partial differentiation are *forward*-approximated and space pd are *center*-approximated.

The 2.1 derivatives ar approximated by:

$$\begin{aligned}\frac{\partial C}{\partial t} &\approx \frac{C(S,t+\delta t)-C(S,t)}{\delta t}\\ \frac{\partial C}{\partial S} &\approx \frac{C(S+\delta S,t)-C(S-\delta S,t)}{2\delta S}\\ \frac{\partial^2 C}{\partial S^2} &\approx \frac{C(S+\delta S,t)+C(S-\delta S,t)-2C(S,t)}{\delta S^2}\end{aligned} \tag{2.4}$$

Replacing 2.6 in 2.1, we can compute the approximation for $C(S,t+\delta t)$:

$$C(S,t+\delta t) = X(S,t)C(S-\delta S,t) + (1-Y(S,t))C(S,t) + Z(S,t)C(S+\delta S,t) \tag{2.5}$$

With:

$$\begin{aligned}X(S,t) &= a(S,t)\frac{\delta t}{\delta S^2} - b(S,t)\frac{\delta t}{2\delta S}\\ Y(S,t) &= r\delta t - 2a(S,t)\frac{\delta t}{\delta S^2}\\ Z(S,t) &= a(S,t)\frac{\delta t}{\delta S^2} + b(S,t)\frac{\delta t}{2\delta S}\\ a(S,t) &= -\frac{1}{2}\sigma(S,t)S^2\\ b(S,t) &= rS\end{aligned} \tag{2.6}$$

In [Wilmott, 2007] is proved the method is convergent and stable if:

$$\delta t \leq \frac{1}{\sigma(S,t)}\left(\frac{\delta S}{S}\right)^2 \tag{2.7}$$

Defining $\delta t = \frac{1}{365p}$, being p the precision parameter and assuming the variance $\sigma(S,T)$ is ≤ 1, reasonable for most financial applications. We get:

$$\begin{aligned}\frac{1}{364p} = \left(\frac{\delta S}{S_{max}}\right)^2 &= \left(\frac{S_{max}x}{S_{max}}\right)^2\\ x &= \sqrt{\frac{1}{364p}}\\ \delta S &= S_{max}\sqrt{\frac{1}{364p}}\end{aligned} \tag{2.8}$$

2.2 Implementation

The solution was implemented in the *FTCS* Class.

```
def __init__(self, sigma0, sigma1, strike , sMax, T, r, precision = 1):
    self.sigma0 = sigma0
    self.sigma1 = sigma1
    self.strike = strike
    self.T = T
    self.r = r
    self.sMax = sMax
    self.L = self.sMax - self.strike * np.exp(-self.r * self.T)
    self.ds = sMax * np.sqrt(1/(364*precision))
    self.dt = 1/(365 * precision)
    self.v1 = self.dt/(self.ds*self.ds)
    self.v2 = self.dt/self.ds
```

2.2.1 Variance

The variance is implemented in *vol*:

```
def vol(self, s, t): #Variancia
    return self.sigma0 + self.sigma1*math.cos((2*math.pi*t)/self.T)\
                *math.exp(-((s/self.strike)-1)**2)
```

2.2.2 FTCS

Each grid is saved in *self.grid* for reuse in other class methods as before.

```
def ftcs(self, function):

    self.grid = np.zeros([len(self.rangeT()), len(self.rangeP())])
    rangePreco = self.rangeP()

    self.grid[0] = [function(s) for s in self.rangeP()]

    for i in range(len(self.rangeT())-1):

        for j in range(len(self.rangeP())-2):

            self.grid[i+1][j+1] = self.grid[i][j] * self.X(rangePreco[j],
             ↪  self.rangeT()[i]) + \
                                    self.grid[i][j+1] * (1+self.Y(rangePreco[j],
                                    ↪  self.rangeT()[i])) + \
                                    self.grid[i][j+2] * self.Z(rangePreco[j],
                                    ↪  self.rangeT()[i])

        self.grid[i+1][0] = 0
        self.grid[i+1][-1] = self.sMax - self.strike * np.exp(-self.r *
         ↪  self.rangeT()[i])

    return self.grid
```

Auxiliary functions:

```python
def a(self, s, t):
    return 0.5*self.vol(s,t) * s * s

def b(self, s):
    return self.r * s

def X(self, s, t):
    return self.a(s,t) * self.v1 - self.b(s) * 0.5 * self.v2

def Y(self, s, t):
    return -self.r * self.dt - 2 * self.a(s, t) * self.v1

def Z(self, s, t):
    return self.a(s,t) * self.v1 + self.b(s) * 0.5 * self.v2
```

2.2.3 Payoff

Again, Payoff is defined externally for easily implementation of other payoffs:

```python
def callPayoff(self, s):
        return np.maximum(s-self.strike,0)

def callPrice(self):
    return self.ftcs(self.callPayoff)
```

2.2.4 Interpolation

The method itself, create a grid with discrete spacing in time and space, but we may need the option price in between the discrete spacing. To solve this problem, I implemented an interpolation to have prices in the whole continuum:

1. Find the closest price and time point in the grid.
2. Look the 8 points around the center point, getting 9 points.
3. A Linear Regression is made using those 9 points.
4. Return the regression result for the desired price and time.

```python
def interpolate(self, s, t):

    closeT = np.argmin(np.abs(self.rangeT() - t))
    closeP = np.argmin(np.abs(self.rangeP() - s))

    T = self.rangeT()
    P = self.rangeP()

    rT = [T[closeT-1], T[closeT-1], T[closeT-1], T[closeT], T[closeT],
    ↪  T[closeT], T[closeT+1], T[closeT+1], T[closeT+1]]
    rP = [P[closeP-1], P[closeP], P[closeP+1], P[closeP-1], P[closeP],
    ↪  P[closeP+1], P[closeP-1], P[closeP], P[closeP+1]]

    resul = np.ones(9)
    k = 0
    for i in range(3):
        for j in range(3):
            resul[k] = self.grid[closeT-1+i][closeP-1+j]
            k += 1
    f = interp2d(rT, rP, resul)
    return f(t,s)[0]
```

2.2.5 Results

The first thing to notice in the FTCS method is the speed of the method, orders faster than the Monte Carlo. It also have all points in the grid, which allows to immediately compute the option Greeks.

Below the results for $\sigma_0 = 4\%$, $\sigma_1 = 2\%$, Strike $= 10$, $S_{max} = 20$, $T = 1$, $r = 3\%$ and *precision* $= 3$.

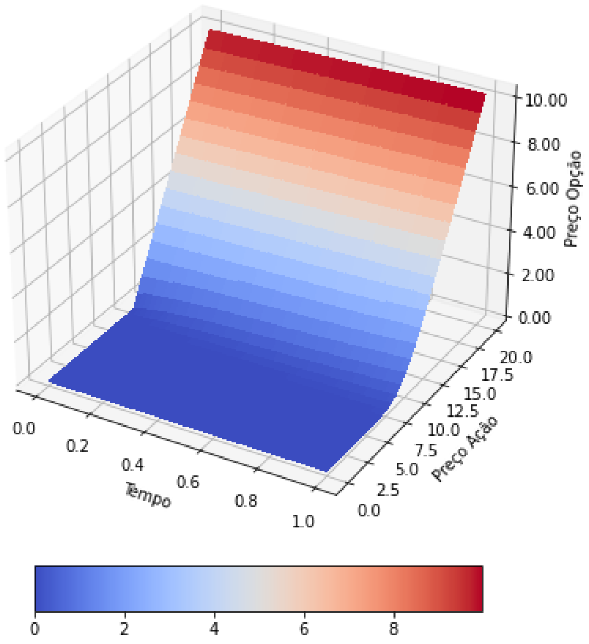

Figure 1: Option Price Surface.

The option profile:

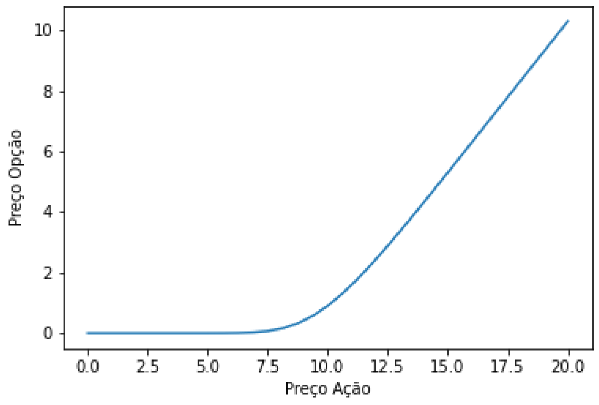

Figure 2: Option Price for t=1

.

Price (S\t)	0.2	0.5	0.8
8	0.00723	0.0361	0.0806
10	0.41274	0.60217	0.75838
12	2.07381	2.19625	2.32236

Table 3: Different interpolation results.

References

[Burden et al., 2016] Burden, R. L., Faires, D. J., and Burden, A. M. (2016). *Numerical Analysis*. CENGAGE.

[De La Cruz, 2022] De La Cruz, H. (2022). Class Notes - Computational Methods in Finance.

[Wilmott, 2007] Wilmott, P. (2007). *Paul Wilmott Introduces Quantitative Finance*. Wiley.