UROPS Project Presentation 8

Wang Zexin

Chapter 18 Portfolio Valuation
of Python for Finance

Quantitative Finance
National University of Singapore

March 21, 2017

Today's Agenda

1. Portfolio Valuation
   - General Modularization
   - Numerical Evaluation of Greeks
   - European Exercise
   - American Exercise
   - Wrapper class

Changes due to different Python version

We are using Python 3.6 while the version in the book is Python 2.7
So here is a list of items to change

- print x now becomes print(x)
- dict.iteritems() now becomes dict.items()
- xrange now becomes range
- lambda (k, v) : (v, k) is no longer available
- instead we can only use: lambda x : (x[1], x[0])
- x / 2 is float division, while x // 2 is integer division

Derivatives Valuation

- Generic Valuation Class
- Numerical Evaluation of Greeks
- European Exercise
- American Exercise
- Wrapper class

General Modularization

The almost complete modularization of the analytics library:
(Based on Monte Carlo simulation being the only numerical method)

- Discounting - *constant_short_rate*
- Relevant data - *market_environment*
- Simulation objects
  - geometric_brownian_motion
  - jump_diffusion
  - square_root_diffusion
- Valuation objects
  - valuation_mcs_european
  - valuation_mcs_american
- Nonredundancy
- Correlations
- Positions

Generate Valuation Class

The valuation class will also require these attributes:
S_0(Initial value), σ(Volatility), K(Strike), T(Maturity)

```
def update(self, initial_value=None, volatility=None,
               strike=None, maturity=None):
    if initial_value is not None:
        self.underlying.update(initial_value=initial_value)
    if volatility is not None:
        self.underlying.update(volatility=volatility)
    if strike is not None:
        self.strike = strike
    if maturity is not None:
        self.maturity = maturity
        # add new maturity date if not in time_grid
        if not maturity in self.underlying.time_grid:
            self.underlying.special_dates.append(maturity)
            self.underlying.instrument_values = None
```

Numerical Evaluation of Greeks

Delta:

$$\Delta = \lim_{\Delta S \to 0} \frac{V(S_0 + \Delta S, \sigma_0) - V(S_0, \sigma_0)}{\Delta S} = \frac{\partial V}{\partial S}$$

Vega:

$$\nu = \lim_{\Delta \sigma \to 0} \frac{V(S_0, \sigma_0 + \Delta \sigma) - V(S_0, \sigma_0)}{\Delta \sigma} = \frac{\partial V}{\partial \sigma}$$

The sensitivity of the derivatives price to underlying price as well as to volatility can be numerically computed.

Numerical Evaluation of Δ

This implementation calculates the change in derivatives value with respect to 2% change in underlying asset price. (Can 2% be too much?)

```
def delta(self, interval=None, accuracy=4):
    if interval is None:
        interval = self.underlying.initial_value / 50.
    # forward-difference approximation
    # calculate left value for numerical Delta
    value_left = self.present_value(fixed_seed=True)
    # numerical underlying value for right value
    initial_del = self.underlying.initial_value + interval
    self.underlying.update(initial_value=initial_del)
    # calculate right value for numerical delta
    value_right = self.present_value(fixed_seed=True)
    # reset the initial_value of the simulation object
    self.underlying.update(initial_value=initial_del - interval)
    delta = (value_right - value_left) / interval
    # correct for potential numerical errors
    if delta < -1.0:
        return -1.0
    elif delta > 1.0:
        return 1.0
    else:
        return round(delta, accuracy)
```

Numerical Evaluation of ν

This implementation calculates the change in derivatives value with respect to 2% change in volatility. (How can we correct for potential numerical errors?)

```python
def vega(self, interval=0.01, accuracy=4):
    if interval < self.underlying.volatility / 50.:
        interval = self.underlying.volatility / 50.
    # forward-difference approximation
    # calculate the left value for numerical Vega
    value_left = self.present_value(fixed_seed=True)
    # numerical volatility value for right value
    vola_del = self.underlying.volatility + interval
    # update the simulation object
    self.underlying.update(volatility=vola_del)
    # calculate the right value for numerical Vega
    value_right = self.present_value(fixed_seed=True)
    # reset volatility value of simulation object
    self.underlying.update(volatility=vola_del - interval)
    vega = (value_right - value_left) / interval
    return round(vega, accuracy)
```

European Exercise Generate Payoff

If self.strike is not initiated, there will still be errors.
Also, computation using the *eval* function can be slow.

```
try:
    # strike defined?
    strike = self.strike
except:
    pass
paths = self.underlying.get_instrument_values(fixed_seed=fixed_seed)
time_grid = self.underlying.time_grid
try:
    time_index = np.where(time_grid == self.maturity)[0]
    time_index = int(time_index)
except:
    print("Maturity date not in time grid of underlying.")
maturity_value = paths[time_index]
# average value over whole path
mean_value = np.mean(paths[:time_index], axis=1)
# maximum value over whole path
max_value = np.amax(paths[:time_index], axis=1)[-1]
# minimum value over whole path
min_value = np.amin(paths[:time_index], axis=1)[-1]
try:
    payoff = eval(self.payoff_func)
    return payoff
except:
    print("Error evaluating payoff function.")
```

European Exercise Valuation

Calculate present value by discounting the expectation of payoff.

```
def present_value(self, accuracy=6, fixed_seed=False, full=False):
    '''
    Parameters
    ==========
    accuracy : int
        number of decimals in returned result
    fixed_seed : Boolean
        use same/fixed seed for valuation
    full : Boolean
        return also full 1d array of present values
    '''
    cash_flow = self.generate_payoff(fixed_seed=fixed_seed)
    discount_factor = self.discount_curve.get_discount_factors(
                        (self.pricing_date, self.maturity))[0, 1]
    result = discount_factor * np.sum(cash_flow) / len(cash_flow)
    if full:
        return round(result, accuracy), discount_factor * cash_flow
    else:
        return round(result, accuracy)
```

European Exercise User Case

Assuming the attributes are correctly updated, for an European call option with $S_0 = 36$, $\sigma = 0.2$ and $K = 40$, we may obtain the following:

```
In [2]: eur_call.present_value()
Out[2]: 9.1016809999999992

In [3]: eur_call.delta()
Out[3]: 0.78769999999999996

In [4]: eur_call.vega()
Out[4]: 10.292
```

American Exercise Generate Payoff

This function will pass out the time points of the optimal exercise price.

```
def generate_payoff(self, fixed_seed=False):
    '''
    Parameters
    ==========
    fixed_seed :
        use same/fixed seed for valuation
    '''
    try:
        strike = self.strike
    except:
        pass
    paths = self.underlying.get_instrument_values(fixed_seed=fixed_seed)
    time_grid = self.underlying.time_grid
    try:
        time_index_start = int(np.where(time_grid == self.pricing_date)[0])
        time_index_end = int(np.where(time_grid == self.maturity)[0])
    except:
        print("Maturity date not in time grid of underlying.")
    instrument_values = paths[time_index_start:time_index_end + 1]
    try:
        payoff = eval(self.payoff_func)
        return instrument_values, payoff, time_index_start, time_index_end
    except:
        print("Error evaluating payoff function.")
```

American Exercise Valuation

Fixed seed: same randomized values for separation simulations.
This function will derive different discount factors for different time points.

```
instrument_values, inner_values, time_index_start, time_index_end = self.generate_pa
time_list = self.underlying.time_grid[time_index_start:time_index_end + 1]
discount_factors = self.discount_curve.get_discount_factors(time_list, dtobjects=Tru
V = inner_values[-1]
for t in range(len(time_list) - 2, 0, -1):
    # derive relevant discount factor for given time interval
    df = discount_factors[t, 1] / discount_factors[t + 1, 1]
    # regression step
    rg = np.polyfit(instrument_values[t], V * df, bf)
    # calculation of continuation values per path
    C = np.polyval(rg, instrument_values[t])
    # optimal decision step:
    # if condition is satisfied (inner value > regressed cont. value)
    # then take inner value; take actual cont. value otherwise
    V = np.where(inner_values[t] > C, inner_values[t], V * df)
df = discount_factors[0, 1] / discount_factors[1, 1]
result = df * np.sum(V) / len(V)
if full:
    return round(result, accuracy), df * V
else:
    return round(result, accuracy)
```

American Exercise User Case

Assuming the attributes are correctly updated, for an American call option with $S_0 = 36$, $\sigma = 0.2$ and $K = 40$, we may obtain the following:

```
S0 | Vola | T | Value
----------------------
36 | 0.2 | 1 | 4.444
36 | 0.2 | 2 | 4.769
36 | 0.4 | 1 | 7.000
36 | 0.4 | 2 | 8.378
38 | 0.2 | 1 | 3.210
38 | 0.2 | 2 | 3.645
38 | 0.4 | 1 | 6.066
38 | 0.4 | 2 | 7.535
40 | 0.2 | 1 | 2.267
40 | 0.2 | 2 | 2.778
40 | 0.4 | 1 | 5.203
40 | 0.4 | 2 | 6.753
42 | 0.2 | 1 | 1.554
42 | 0.2 | 2 | 2.099
42 | 0.4 | 1 | 4.459
42 | 0.4 | 2 | 6.046
44 | 0.2 | 1 | 1.056
44 | 0.2 | 2 | 1.618
44 | 0.4 | 1 | 3.846
44 | 0.4 | 2 | 5.494

In [8]: am_put.present_value(fixed_seed=True, bf=5)
Out[8]: 5.494116
```

Wrapper class - implementation

```
import numpy as np
import pandas as pd

from dx_simulation import *
from valuation_class import valuation_class
from valuation_mcs_european import valuation_mcs_european
from valuation_mcs_american import valuation_mcs_american
```

With this *dx_valuation.py*, we are now able to import the valuation framework package as well the simulation classes in one line.

Wrapper class - testing

Now we need to enhance the *__init__.py* which initially has the same content as *dx_frame.py* and *dx_simulation.py* in the same directory to include importing the simulation classes.

```
import numpy as np
import pandas as pd
import datetime as dt
# frame
from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment
from plot_option_stats import plot_option_stats
# simulation
from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class
from geometric_brownian_motion import geometric_brownian_motion
from jump_diffusion import jump_diffusion
from square_root_diffusion import square_root_diffusion
# valuation
from valuation_class import valuation_class
from valuation_mcs_european import valuation_mcs_european
from valuation_mcs_american import valuation_mcs_american
```

Thank You

E0007424@u.nus.edu