UROPS Project Presentation 8

Wang Zexin

Chapter 18 Portfolio Valuation
of Python for Finance

Quantitative Finance
National University of Singapore

March 23, 2017

Today's Agenda

1. General Modularization
2. Derivatives Positions
   - Initialization
   - *get_info* method
   - Use Case
3. Derivatives Portfolio
   - Initialization
   - Attributes
   - Time Grid
   - Cholesky Matrix
   - Positions
   - Getter methods
   - 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

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

Initializiation

The derivatives positions class will include these attributes:

- Quantity
- Underlying
- Market Environment (A *mar_env* object)
- Otype (which valuation class to use)
- Payoff function (A string with the formula for payoff)

Derivatives Position

```python
class derivatives_position(object):
    ''' Class to model a derivatives position.
    Attributes
    ==========
    name : string
        name of the object
    quantity : float
        number of assets/derivatives making up the position
    underlying : string
        name of asset/risk factor for the derivative
    mar_env : instance of market_environment
        constants, lists, and curves relevant for valuation_class
    otype : string
        valuation class to use
    payoff_func : string
        payoff string for the derivative
    Methods
    =======
    get_info :
        prints information about the derivative position
    '''
    def __init__(self, name, quantity, underlying, mar_env, otype, payoff_func):
        self.name = name
        self.quantity = quantity
        self.underlying = underlying
        self.mar_env = mar_env
        self.otype = otype
        self.payoff_func = payoff_func
```

Derivatives Position

This class also comes with a *get_info* method.
In which *payoff_function* is only a string for symbolic computations.

```
def get_info(self):
    print("NAME")
    print(self.name, "\n")
    print("QUANTITY")
    print(self.quantity, "\n")
    print("UNDERLYING")
    print(self.underlying, "\n")
    print("MARKET ENVIRONMENT")
    print("\n**Constants**")
    for key, value in self.mar_env.constants.items():
        print(key, value)
    print("\n**Lists**")
    for key, value in self.mar_env.lists.items():
        print(key, value)
    print("\n**Curves**")
    for key in self.mar_env.curves.items():
        print(key, value)
    print("\nOPTION TYPE")
    print(self.otype, "\n")
    print("PAYOFF FUNCTION")
    print(self.payoff_func)
```

Derivatives Position

Here is a simple use case of the *derivatives_position* class.

```
from dx import *

me_gbm = market_environment("me_gbm", dt.datetime(2015, 1, 1))
me_gbm.add_constant("initial_value", 36.)
me_gbm.add_constant("volatility", 0.2)
me_gbm.add_constant("currency", "EUR")

me_gbm.add_constant("model", "gbm")

from derivatives_position import derivatives_position
me_am_put = market_environment("me_am_put", dt.datetime(2015, 1, 1))
me_am_put.add_constant("maturity", dt.datetime(2015, 12, 31))
me_am_put.add_constant("strike", 40.)
me_am_put.add_constant("currency", "EUR")
payoff_func = "np.maximum(strike - instrument_values, 0)"
am_put_pos = derivatives_position(
    name="am_put_pos",
    quantity=3,
    underlying="gbm",
    mar_env=me_am_put,
    otype="American",
    payoff_func=payoff_func)
am_put_pos.get_info()
```

Derivative Portfolio

Initialization for the *derivatives_portfolio* class.

```
import pandas as pd
from dx_valuation import *
# models available for risk factor modeling
models = {'gbm' : geometric_brownian_motion,
    'jd' : jump_diffusion,
    'srd' : square_root_diffusion}
# allowed exercise types
otypes = {'European' : valuation_mcs_european,
    'American' : valuation_mcs_american}
```

Derivatives Portfolio

The attributes of the *derivatives_portfolio* class are initialized as follows:

```
def __init__(self, name, positions, val_env, assets,
    correlations=None, fixed_seed=False):
    self.name = name
    self.positions = positions
    self.val_env = val_env
    self.assets = assets
    self.underlyings = set()
    self.correlations = correlations
    self.time_grid = None
    self.underlying_objects = {}
    self.valuation_objects = {}
    self.fixed_seed = fixed_seed
```

Derivatives Portfolio - Time Grid

During initialization, the class would then go on to calculate the time grid.

```
for pos in self.positions:
    # determine earliest starting_date
    self.val_env.constants['starting_date'] = \
        min(self.val_env.constants['starting_date'],
        positions[pos].mar_env.pricing_date)
    # determine latest date of relevance
    self.val_env.constants['final_date'] = \
        max(self.val_env.constants['final_date'],
        positions[pos].mar_env.constants['maturity'])
    # collect all underlyings
    # add to set; avoids redundancy
    self.underlyings.add(positions[pos].underlying)

# generate general time grid
start = self.val_env.constants['starting_date']
end = self.val_env.constants['final_date']
time_grid = pd.date_range(start=start,end=end,
    freq=self.val_env.constants['frequency']
    ).to_pydatetime()
time_grid = list(time_grid)
for pos in self.positions:
    maturity_date = positions[pos].mar_env.constants['maturity']
    if maturity_date not in time_grid:
        time_grid.insert(0, maturity_date)
        self.special_dates.append(maturity_date)
if start not in time_grid:
    time_grid.insert(0, start)
if end not in time_grid:
    time_grid.append(end)
# delete duplicate entries
time_grid = list(set(time_grid))
# sort dates in time_grid
time_grid.sort()
self.time_grid = np.array(time_grid)
self.val_env.add_list('time_grid', self.time_grid)
```

Derivatives Portfolio - Cholesky Matrix

If the correlation matrix is input in as a parameter, the class would calculate the Cholesky matrix during the initialization.

```
if correlations is not None:
    # take care of correlations
    ul_list = sorted(self.underlyings)
    correlation_matrix = np.zeros((len(ul_list), len(ul_list)))
    np.fill_diagonal(correlation_matrix, 1.0)
    correlation_matrix = pd.DataFrame(correlation_matrix,
                                      index=ul_list, columns=ul_list)
    for i, j, corr in correlations:
        corr = min(corr, 0.999999999999)
        # fill correlation matrix
        correlation_matrix.loc[i, j] = corr
        correlation_matrix.loc[j, i] = corr

    # determine Cholesky matrix
    cholesky_matrix = np.linalg.cholesky(np.array(correlation_matrix))

    # dictionary with index positions for the
    # slice of the random number array to be used by
    # respective underlying
    rn_set = {asset: ul_list.index(asset) for asset in self.underlyings}
    # random numbers array, to be used by
    # all underlyings (if correlations exist)
    random_numbers = sn_random_numbers((len(rn_set),
        len(self.time_grid),
        self.val_env.constants['paths']),
        fixed_seed=self.fixed_seed)
    # add all to valuation environment that is
    # to be shared with every underlying
    self.val_env.add_list('cholesky_matrix', cholesky_matrix)
    self.val_env.add_list('random_numbers', random_numbers)
    self.val_env.add_list('rn_set', rn_set)
```

Derivatives Portfolio - Positions

The portfolio will contain the positions for derivatives as well.

```python
for asset in self.underlyings:
    # select market environment of asset
    mar_env = self.assets[asset]
    # add valuation environment to market environment
    mar_env.add_environment(val_env)
    # select right simulation class
    model = models[mar_env.constants['model']]
    # instantiate simulation object
    if correlations is not None:
        self.underlying_objects[asset] = model(asset, mar_env,
        corr=True)
    else:
        self.underlying_objects[asset] = model(asset, mar_env,
        corr=False)

for pos in positions:
    # select right valuation class (European, American)
    val_class = otypes[positions[pos].otype]
    # pick market environment and add valuation environment
    mar_env = positions[pos].mar_env
    mar_env.add_environment(self.val_env)
    # instantiate valuation class
    self.valuation_objects[pos] = \
        val_class(name=positions[pos].name,
            mar_env=mar_env,
            underlying=self.underlying_objects[positions[pos].underlying],
            payoff_func=positions[pos].payoff_func)
```

Derivatives Portfolio - Getter methods

The portfolio class comes with two convenient getters.

```
def get_positions(self):
    ''' Convenience method to get information about
    all derivatives positions in a portfolio. '''
    for pos in self.positions:
        bar = '\n' + 50 * '-'
        print(bar)
        self.positions[pos].get_info()
        print(bar)

def get_statistics(self, fixed_seed=False):
    ''' Provides portfolio statistics. '''
    res_list = []
    # iterate over all positions in portfolio
    for pos, value in self.valuation_objects.items():
        p = self.positions[pos]
        pv = value.present_value(fixed_seed=fixed_seed)
        res_list.append([
            p.name,
            p.quantity,
            # calculate all present values for the single instruments
            pv,
            value.currency,
            # single instrument value times quantity
            pv * p.quantity,
            # calculate Delta of position
            value.delta() * p.quantity,
            # calculate Vega of position
            value.vega() * p.quantity,
        ])
    # generate a pandas DataFrame object with all results
    res_df = pd.DataFrame(res_list,
    columns=['name', 'quant.', 'value', 'curr.',
    'pos_value', 'pos_delta', 'pos_vega'])
    return res_df
```

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 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