UROPS Project Presentation 5

Wang Zexin

Chapter 16 Simulation of Financial models
of Python for Finance

Quantitative Finance
National University of Singapore

February 15, 2017

Today's Agenda

1. Simulation of Financial models
   - Random number generation
   - Generic financial model simulation class
   - Geometric Brownian Motion
   - Jump Diffusion
   - Square-root Diffusion
   - Capture common features
   - 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

Simulation of Financial models

- Random number generation
- Generic simulation class
- Geometric Brownian motion
- Jump diffusion
- Square-root diffusion
- Wrapper class

Generate standard normally distributed random numbers

```python
import numpy as np
def sn_random_numbers(shape, antithetic=True, moment_matching=True,
                      fixed_seed=False):
    ''' Returns an array of shape shape with (pseudo)random numbers
    that are standard normally distributed.
    shape : tuple (o, n, m)
        generation of array with shape (o, n, m)
    antithetic : Boolean - generation of antithetic variates
    moment_matching : Boolean - matching of first and second moments
    fixed_seed : Boolean - flag to fix the seed
    ran : (o, n, m) array of (pseudo)random numbers
    '''
    if fixed_seed:
        np.random.seed(1000)
    if antithetic:
        ran = np.random.standard_normal((shape[0], shape[1], shape[2] // 2))
        ran = np.concatenate((ran, -ran), axis=2)
    else:
        ran = np.random.standard_normal(shape)
    if moment_matching:
        ran = ran - np.mean(ran)
        ran = ran / np.std(ran)
    if shape[0] == 1:
        return ran[0]
    else:
        return ran
```

Generic financial model simulation class

Below are all the parameters initiated for this class.

```
def __init__(self, name, mar_env, corr):
    try:
        self.name = name
        self.pricing_date = mar_env.pricing_date
        self.initial_value = mar_env.get_constant('initial_value')
        self.volatility = mar_env.get_constant('volatility')
        self.final_date = mar_env.get_constant('final_date')
        self.currency = mar_env.get_constant('currency')
        self.frequency = mar_env.get_constant('frequency')
        self.paths = mar_env.get_constant('paths')
        self.discount_curve = mar_env.get_curve('discount_curve')
        try:
            # if time_grid in mar_env take this
            # (for portfolio valuation)
            self.time_grid = mar_env.get_list('time_grid')
        except:
            self.time_grid = None
        try:
            # if there are special dates, then add these
            self.special_dates = mar_env.get_list('special_dates')
        except:
            self.special_dates = []
            self.instrument_values = None
            self.correlated = corr
            if corr is True:
                # only needed in a portfolio context when
                # risk factors are correlated
                self.cholesky_matrix = mar_env.get_list('cholesky_matrix')
                self.rn_set = mar_env.get_list('rn_set')[self.name]
                self.random_numbers = mar_env.get_list('random_numbers')
    except:
        print("Error parsing market environment.")
```

Generic simulation class - generate time grid

```
def generate_time_grid(self):
    start = self.pricing_date
    end = self.final_date
    # pandas date_range function
    # freq = e.g. 'B' for Business Day,
    # 'W' for Weekly, 'M' for Monthly
    time_grid = pd.date_range(start=start, end=end,
                              freq=self.frequency).to_pydatetime()
    time_grid = list(time_grid)
    # enhance time_grid by start, end, and special_dates
    if start not in time_grid:
        time_grid.insert(0, start)
        # insert start date if not in list
    if end not in time_grid:
        time_grid.append(end)
        # insert end date if not in list
    if len(self.special_dates) > 0:
        # add all special dates
        time_grid.extend(self.special_dates)
        # delete duplicates
        time_grid = list(set(time_grid))
        # sort list
        time_grid.sort()
    self.time_grid = np.array(time_grid)
```

Generic financial model simulation class

This method returns the values of instruments built-in.

```python
def get_instrument_values(self, fixed_seed=True):
    if self.instrument_values is None:
        # only initiate simulation if there are no instrument values
        self.generate_paths(fixed_seed=fixed_seed, day_count=365.)
    elif fixed_seed is False:
        # also initiate resimulation when fixed_seed is False
        self.generate_paths(fixed_seed=fixed_seed, day_count=365.)
    return self.instrument_values
```

Geometric Brownian Motion

$$S_T = S_0 \exp\{(r - \frac{1}{2}\sigma^2)T + \sigma\sqrt{T}z\}$$

$$S_t = S_{t-\Delta t} \exp\{(r - \frac{1}{2}\sigma^2)^{\Delta} t + \sigma\sqrt{\Delta t}z_t\}$$

Geometric Brownian Motion - implementation

```
def __init__(self, name, mar_env, corr=False):
    super(geometric_brownian_motion, self).__init__(name, mar_env, corr)

def update(self, initial_value=None, volatility=None, final_date=None):
    if initial_value is not None:
        self.initial_value = initial_value
    if volatility is not None:
        self.volatility = volatility
    if final_date is not None:
        self.final_date = final_date
    self.instrument_values = None
```

Geometric Brownian Motion - implementation

```
def generate_paths(self, fixed_seed=False, day_count=365.):
    if self.time_grid is None:
        self.generate_time_grid() # method from generic simulation class
    M = len(self.time_grid) # number of dates for time grid
    I = self.paths # number of paths
    paths = np.zeros((M, I)) # array initialization for path simulation
    paths[0] = self.initial_value # initialize first date with initial_value
    if not self.correlated: # if not correlated, generate random numbers
        rand = sn_random_numbers((1, M, I), fixed_seed=fixed_seed)
    else:
        # if correlated, use random number object as provided in market environment
        rand = self.random_numbers
    short_rate = self.discount_curve.short_rate
    # get short rate for drift of process
    for t in range(1, len(self.time_grid)):
        # select the right time slice from the relevant random number set
        if not self.correlated:
            ran = rand[t]
        else:
            ran = np.dot(self.cholesky_matrix, rand[:, t, :])
            ran = ran[self.rn_set]
        dt = (self.time_grid[t] - self.time_grid[t - 1]).days / day_count
        # difference between two dates as year fraction
        paths[t] = paths[t - 1] * np.exp((short_rate - 0.5
                                         * self.volatility ** 2) * dt
                                         + self.volatility * np.sqrt(dt) * ran)
        # generate simulated values for the respective date
    self.instrument_values = paths
```

Geometric Brownian Motion - Use Case

```
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('final_date', dt.datetime(2015, 12, 31))
me_gbm.add_constant('currency', 'EUR')
me_gbm.add_constant('frequency', 'M')
# monthly frequency (respective month end)
me_gbm.add_constant('paths', 10000)
csr = constant_short_rate('csr', 0.05)
me_gbm.add_curve('discount_curve', csr)

from dx_simulation import *
gbm = geometric_brownian_motion('gbm', me_gbm)
gbm.generate_time_grid()
paths_1 = gbm.get_instrument_values()
gbm.update(volatility=0.5)
paths_2 = gbm.get_instrument_values()
```

The GBM simulation class is able to:

- Import parameters from market environment class
- Generate datetime object time grid
- Get instrument values based on parameters

Jump Diffusion Model

$$dS_t = (r - r_{\mathrm{J}})S_t dt + S_t dZ_t + \mathrm{J}_t S_t dN_t$$

Euler discretization:

$$S_t = S_{t-\Delta t}[e^{(r-r_{\mathrm{J}}-\sigma^2/2)\Delta t+\sigma\sqrt{\Delta t}z_t^1} + (e^{\mu_{\mathrm{J}}+\delta z_t^2} - 1)y_t]$$

Jump Diffusion - implementation

```python
def __init__(self, name, mar_env, corr=False):
    super(jump_diffusion, self).__init__(name, mar_env, corr)
    try:
        # additional parameters needed
        self.lamb = mar_env.get_constant('lambda')
        self.mu = mar_env.get_constant('mu')
        self.delt = mar_env.get_constant('delta')
    except:
        print("Error parsing market environment.")

def update(self, initial_value=None, volatility=None, lamb=None,
           mu=None, delta=None, final_date=None):
    if initial_value is not None:
        self.initial_value = initial_value
    if volatility is not None:
        self.volatility = volatility
    if lamb is not None:
        self.lamb = lamb
    if mu is not None:
        self.mu = mu
    if delta is not None:
        self.delt = delta
    if final_date is not None:
        self.final_date = final_date
    self.instrument_values = None
```

Jump Diffusion - implementation

This part of generate_path is repeated, maybe we can capture this feature

```
def generate_paths(self, fixed_seed=False, day_count=365.):
    if self.time_grid is None:
        self.generate_time_grid()
            # method from generic simulation class
    M = len(self.time_grid) # number of dates for time grid
    I = self.paths # number of paths
    paths = np.zeros((M, I)) # array initialization for path simulation
    paths[0] = self.initial_value # initialize first date with initial_value
    if self.correlated is False: # if not correlated, generate random numbers
        sn1 = sn_random_numbers((1, M, I),
                                fixed_seed=fixed_seed)
    else:
        # if correlated, use random number object as provided in market environment
        sn1 = self.random_numbers

    # standard normally distributed pseudorandom numbers for the jump component
    sn2 = sn_random_numbers((1, M, I),
                            fixed_seed=fixed_seed)
```

Jump Diffusion - implementation

```
rj = self.lamb * (np.exp(self.mu + 0.5 * self.delt ** 2) - 1)

short_rate = self.discount_curve.short_rate
for t in range(1, len(self.time_grid)):
    # select the right time slice from the relevant random number set
    if self.correlated is False:
        ran = sn1[t]
    else:
        # only with correlation in portfolio context
        ran = np.dot(self.cholesky_matrix, sn1[:, t, :])
        ran = ran[self.rn_set]
    dt = (self.time_grid[t] - self.time_grid[t - 1]).days / day_count
    # difference between two dates as year fraction
    poi = np.random.poisson(self.lamb * dt, I)
    # Poisson-distributed pseudorandom numbers for jump component
    paths[t] = paths[t - 1] * (np.exp((short_rate - rj
                               - 0.5 * self.volatility ** 2) * dt
                               + self.volatility * np.sqrt(dt) * ran)
                               + (np.exp(self.mu + self.delt *
                               sn2[t]) - 1) * poi)
self.instrument_values = paths
```

Jump Diffusion - Use Case

```
me_jd = market_environment('me_jd', dt.datetime(2015, 1, 1))
# add jump diffusion specific parameters
me_jd.add_constant('lambda', 0.3)
me_jd.add_constant('mu', -0.75)
me_jd.add_constant('delta', 0.1)
me_jd.add_environment(me_gbm)
from jump_diffusion import jump_diffusion
jd = jump_diffusion('jd', me_jd)
paths_3 = jd.get_instrument_values()
jd.update(lamb=0.9)
paths_4 = jd.get_instrument_values()
```

The Jump Diffusion simulation class is able to:

- Import parameters from market environment class
- Generate datetime object time grid (inherited)
- Get instrument values based on parameters

$$dx_t = \kappa(\theta - x_t)dt + \sigma\sqrt{x_t}dZ_t$$

Euler discretization:

$$\text{letting } s = t -^{\Delta} t, \text{ and } x^{+} = max(x, 0)$$

$$\tilde{x}_t = \tilde{x}_s + \kappa(\theta - \tilde{x}_s^{+})^{\Delta} t + \sigma\sqrt{\tilde{x}_s^{+\Delta} t} z_t$$

$$x_t = \tilde{x}_t^{+}$$

Square-root Diffusion - implementation

```
def __init__(self, name, mar_env, corr=False):
    super(square_root_diffusion, self).__init__(name, mar_env, corr)
    try:
        self.kappa = mar_env.get_constant('kappa')
        self.theta = mar_env.get_constant('theta')
    except:
        print("Error parsing market environment.")

def update(self, initial_value=None, volatility=None, kappa=None,
    theta=None, final_date=None):
    if initial_value is not None:
        self.initial_value = initial_value
    if volatility is not None:
        self.volatility = volatility
    if kappa is not None:
        self.kappa = kappa
    if theta is not None:
        self.theta = theta
    if final_date is not None:
        self.final_date = final_date
    self.instrument_values = None
```

Square-root Diffusion - implementation

This part of generate_path is repeated, maybe we can capture this feature

```
def generate_paths(self, fixed_seed=True, day_count=365.):
    if self.time_grid is None:
        self.generate_time_grid()
    M = len(self.time_grid)
    I = self.paths
    paths = np.zeros((M, I))
    paths_ = np.zeros_like(paths)
    paths[0] = self.initial_value
```

Square-root Diffusion - implementation

```
paths_[0] = self.initial_value
if self.correlated is False:
    rand = sn_random_numbers((1, M, I),
                             fixed_seed=fixed_seed)
else:
    rand = self.random_numbers

for t in range(1, len(self.time_grid)):
    dt = (self.time_grid[t] - self.time_grid[t - 1]).days / day_count
    if self.correlated is False:
        ran = rand[t]
    else:
        ran = np.dot(self.cholesky_matrix, rand[:, t, :])
        ran = ran[self.rn_set]

    # full truncation Euler discretization
    paths_[t] = (paths_[t - 1] + self.kappa
                 * (self.theta - np.maximum(0, paths_[t - 1, :])) * dt
                 + np.sqrt(np.maximum(0, paths_[t - 1, :]))
                 * self.volatility * np.sqrt(dt) * ran)
    paths[t] = np.maximum(0, paths_[t])
self.instrument_values = paths
```

Square-root Diffusion - Use Case

```
me_srd = market_environment('me_srd', dt.datetime(2015, 1, 1))
me_srd.add_constant('initial_value', .25)
me_srd.add_constant('volatility', 0.05)
me_srd.add_constant('final_date', dt.datetime(2015, 12, 31))
me_srd.add_constant('currency', 'EUR')
me_srd.add_constant('frequency', 'W')
me_srd.add_constant('paths', 10000)

# specific to simualation class
me_srd.add_constant('kappa', 4.0)
me_srd.add_constant('theta', 0.2)

# required but not needed for the class
me_srd.add_curve('discount_curve', constant_short_rate('r', 0.0))
from square_root_diffusion import square_root_diffusion
srd = square_root_diffusion('srd', me_srd)

srd_paths = srd.get_instrument_values()[:, :10]
```

The Square-root Diffusion simulation class is able to:

- Import parameters from market environment class
- Generate datetime object time grid (inherited)
- Get instrument values based on parameters

Capture common features from generate_path() method

We now define a new method in the simulation class to be inherited.

```
def generate_path_common(self):
    if self.time_grid is None:
        self.generate_time_grid() # method from generic simulation class
    M = len(self.time_grid) # number of dates for time grid
    I = self.paths # number of paths
    paths = np.zeros((M, I)) # array initialization for path simulation
    paths[0] = self.initial_value # initialize first date with initial_value
    return (M, I, paths)
```

And it will then be called by the subclasses using this statement:

```
M, I, paths = super().generate_path_common()
```

Wrapper class - implementation

```
import numpy as np
import pandas as pd

from dx_frame import *
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
```

With this *dx_frame.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* 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
# 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
```

Thank You

E0007424@u.nus.edu