UROPS Project Presentation 1

Wang Zexin

Chapter 8 Performance Python
Chpter 10 Stochastics
of Python for Finance

Quantitative Finance
National University of Singapore

January 19, 2017

Today's Agenda

1. Performance Python
   - Improvements in execution speed
   - Preserving memory efficiency
   - Efficiency in writing code
2. Stochastics
   - Random number generation
   - Simulation
   - Valuation
   - Risk measures

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

We have three aims in this chapter

- Efficiency in writing code
  - numpy vectorization in compact matrix forms
  - numba : nb.jit(f_py)
- Improvement in execution speed
  - numexpr - fast numerical operations (multithread)
  - numba - dynamic compiling for nested loops
  - IPython.parallel (ipyparallel)
  - multiprocessing - local parallel calculations
  - Cython - static compiling (almost as fast as numba)
- Preserving memory efficiency
  - numba
  - numpy : C-like (default)

We shall go through these useful methods

- Implementation paradigms
- Libraries
- Compiling
- Parallelization

Convenience function - systematic performance comparison

```python
def perf_comp_data(func_list, data_list, rep=3, number=1):
    ''' the convenience function for comparing performance systematically '''
    from timeit import repeat
    res_list = {}
    for name in enumerate(func_list):
        # enumerate basically create an array of tuples of (index, element)
        stmt = name[1] + '(' + data_list[name[0]] + ')' # function_name(data_name)
        setup = "from __main__ import " + name[1] + ', ' + data_list[name[0]]
        #from __main__ import function_name, data_name
        results = repeat(stmt=stmt, setup=setup, repeat=rep, number=number)
        res_list[name[1]] = sum(results) / rep #take average running time
    res_sort = sorted(res_list.items(), key = lambda x : (x[1],x[0]))
    for item in res_sort:
        rel = item[1] / res_sort[0][1]
        print('function: ' + item[0] + ', av. time sec: %9.5f, ' % item[1]\
              + 'relative: %6.1f' % rel) #C-like print formatting
```

Numerical operations

Multithreaded numexpr implementation is fastest compared to:

- using *eval* function
- using built-in library math
- using iterators (lists)
- using numpy's mathematical methods
- using single-threaded numexpr

```
def multithreaded_numexpr(a):
    import numexpr as ne
    ex = 'abs(cos(a)) ** 0.5 + sin(2 + 3 * a)'
    ne.set_num_threads(16)
    return ne.evaluate(ex)
```

Highly computational burden problems

Parallel calculations are superior in heavy workloads
There are two different ways to conduct parallel calculations

- IPython.parallel (or ipyparallel) using cluster
- using the standard library multiprocessing
- massive parallel operations using GPGPUs

Multiprocessing is a standard built-in library, hence recommended

```
#running on server with 8 cores/16 threads
from time import time
times = []
for w in range(1, 17):
    t0 = time()
    pool = mp.Pool(processes = w)
    result = pool.map(simulate_gbm, t * [(M, I), ])
    times.append(time() - t0)
```

Compiling to improve memory efficiency

NumPy is significantly faster than normal python
but it is using 8 times of memory compared to normal python!
There are two other ways to resolve this with the same computation speed

- Dynamic compiling using numba
- Static compiling using Cython

Cython preserves the same speed with numba in dealing with nested loops
numba has another advantage introduced in the next page!

Little additional effort to improve performance

- numpy requries us to think of the matrix form
- numba only requires to call the **jit** function
- numba also preserve the memory efficiency
- numba seems to be the best choice!

```
import numba as nb
binomial_nb = nb.jit(binomial_py)
```

Code sample for binomial option pricing

```
def binomial_py(strike):
    S0 = 100; r = 0.05 # constant short rate
    T = 1. # call option maturity
    vola = 0.20 # constant volatility factor
    M = 1000; dt = T / M;  df = exp(-r * dt) # time parameters
    u = exp(vola * sqrt(dt));  d = 1 / u # binomial parameters
    q = (exp(r * dt) - d) / (u - d) # risk neutral probability
    S = np.zeros((M+1, M+1), dtype = np.float64)
    S[0,0] = S0;  z1 = 0
    for j in range(1, M+1, 1): #future stock pricings
      z1 += 1
      for i in range(z1 + 1):
        S[i,j] = S[0,0] * (u ** j) * (d ** (i*2))
    iv = np.zeros((M+1, M+1), dtype = np.float64)
    z2 = 0
    for j in range(0, M+1, 1): #future call option pricings
      for i in range(z2 + 1):
        iv[i, j] = max(S[i,j]-strike, 0)
      z2 += 1
    pv = np.zeros((M+1, M+1), dtype = np.float64)
    pv[:, M] = iv[:, M];  z3 = M+1
    for j in range(M-1, -1, -1):
      z3 -= 1
      for i in range(z3):
        pv[i,j] = (q * pv[i, j+1] + (1-q) * pv[i+1, j+1]) * df
    return pv[0,0]
```

- Random number geneartion
- Simulation
- Valuation
- Risk measures

Random number generation

using functions provided in *numpy.random*
(import numpy.random as npr: for convenience)

- Standard normal: npr.standard_normal(sample_size)
- Normal: npr.normal(mu, sigma, sample_size)
- Chi Square: npr.chisquare(df = 0.5, size = sample_size)
- Poisson: npr.poisson(lam = 1.0, size = sample_size)

The return value of the above methods are numpy arrays

Simulation

- Geometric Brownian Motion
- Square-root Diffusion Model
- Jump Diffusion Model
- Stochastic Volatility Model

Simulation for Geometric Brownian Motion

We have two different ways to generate simulations for this equation:

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

```
import numpy as np
import numpy.random as npr
S0 = 100 # initial value
r = 0.05 # constant short rate
sigma = 0.25 # constant volatility
T = 2.0 # in years
I = 10000
ST1 = S0 * np.exp((r-0.5*sigma**2)*T+sigma*np.sqrt(T)*npr.standard_normal(I))
ST2 = S0 * npr.lognormal((r-0.5*sigma**2)*T, sigma*np.sqrt(T), size = I)
```

Simulation for Geometric Brownian Motion

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

```
def bsm_mcs_valuation(strike):
    ''' Dynamic Black-Scholes-Merton Monte Carlo estimator
        for European calls.
        Parameters
        ==========
        strike : float
        strike price of the option
        Results
        =======
        value : float
        estimate for present value of call option
    '''
    import numpy as np
    S0 = 100.; T = 1.0; r = 0.05; vola = 0.2
    M = 50; I = 20000
    dt = T / M
    rand = np.random.standard_normal((M + 1, I))
    S = np.zeros((M + 1, I)); S[0] = S0
    for t in range(1, M + 1):
        S[t]=S[t-1]*np.exp((r-0.5*vola**2)*dt+vola*np.sqrt(dt)*rand[t])
    value = (np.exp(-r * T) * np.sum(np.maximum(S[-1] - strike, 0)) / I)
    return value
```

Simulation for Stochastic volatility model

$$dS_t = rS_t dt + \sqrt{v_t} S_t dZ_t^1$$
$$dv_t = \kappa_v(\theta_v - v_t)dt + \sigma_v \sqrt{v_t} dZ_t^2$$
$$dZ_t^1 dZ_t^2 = \rho$$

Euler discretization:

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

Still we need to ensure that:

$$dZ_t^1 dZ_t^2 = \rho$$

Simulation for Stochastic volatility model

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

```
# Parameters
S0 = 100;     r = 0.05;     v0 = 0.1;
kappa = 3.0;     theta = 0.25;     sigma = 0.1;     rho = 0.6
T = 1.0;     M = 50;     I = 10000

#Cholesky decomposition of correlation matrix
corr_mat = np.zeros((2,2))
corr_mat[0, :] = [1.0, rho]
corr_mat[1, :] = [rho, 1.0]
cho_mat = np.linalg.cholesky(corr_mat)

ran_num = npr.standard_normal((2, M + 1, I))

v = np.zeros_like(ran_num[0]);     vh = np.zeros_like(v)
v[0] = v0;     vh[0] = v0; dt = T / M
for t in range(1, M + 1): # simulation of volatility process
    ran = np.dot(cho_mat, ran_num[:, t, :]) # ensure certain correlation
    vh[t]=(vh[t-1]+kappa*(theta-np.maximum(vh[t-1],0))*dt\
           +sigma*np.sqrt(np.maximum(vh[t-1],0))*np.sqrt(dt)*ran[1])
v = np.maximum(vh, 0)

S = np.zeros_like(ran_num[0]);     S[0] = S0
for t in range(1, M + 1): # simulation of stock prices
    ran = np.dot(cho_mat, ran_num[:, t, :]) # ensure certain correlation
    S[t] = S[t-1] * np.exp((r-0.5 * v[t]) * dt + np.sqrt(v[t]) * ran[0] * np.sqrt(dt))
```

Simulation for Square-root diffusion model

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

Euler discretization:

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

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

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

Simulation for Square-root diffusion model

With the Euler discretization:

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

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

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

```
x0 = 0.05; kappa = 3.0; theta = 0.02; sigma = 0.1 # model parameters
I = 10000; M = 50; T = 1; dt = T / M # time parameters
xh = np.zeros((M+1, I)); x1 = np.zeros_like(xh)
xh[0] = x0; x1[0] = x0
for t in range(1, M+1):
    xh[t] = (xh[t-1]+kappa*(theta-np.maximum(xh[t-1],0))*dt\
      +sigma*np.sqrt(np.maximum(xh[t-1],0))*np.sqrt(dt)*npr.standard_normal(I))
x1 = np.maximum(xh, 0)
```

Simulation for Square-root diffusion model

With the exact Euler discretization:

$$\text{letting } s = t - {}^{\Delta}t$$

$$x_t = \frac{\sigma^2(1-e^{-\kappa^{\Delta}t})}{4\kappa}\chi_d^2\left[\frac{4\kappa e^{-\kappa^{\Delta}t}}{\sigma^2(1-e^{-\kappa^{\Delta}t})}x_s\right]$$

```
def srd_exact():
    x0 = 0.05; kappa = 3.0; theta = 0.02; sigma = 0.1 # model parameters
    I = 10000; M = 50; T = 1; dt = T / M # time parameters
    x2 = np.zeros((M + 1, I));    x2[0] = x0
    c = (sigma ** 2 * (1 - np.exp(-kappa * dt))) / (4 * kappa)
    df = 4 * theta * kappa / sigma ** 2 # degree of freedom
    for t in range(1, M + 1):
        nc = np.exp(-kappa * dt) / c * x2[t - 1] #noncentrality parameter
        x2[t] = c * npr.noncentral_chisquare(df, nc, size=I)
    return x2
x2 = srd_exact()
```

Simulation for 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:

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

Simulation for jump diffusion model

Euler discretization:

$$S_t = S_{t-\Delta t}[e^{(r-r_J-\sigma^2)^\Delta t+\sigma\sqrt{\Delta t}z_t^1} + (e^{\mu_J+\delta z_t^2} - 1)y_t]$$

```
def simulate_jump_diffusion(r=.05, sigma=.2, lamb=.75, mu=-.6, delta=.25):
    M = 50;    I = 10000;    T = 1.0;     dt = T / M
    rj = lamb * (np.exp(mu + 0.5 * delta ** 2) - 1)
    S = np.zeros((M + 1, I))
    S0=100;    S[0] = S0
    sn1 = npr.standard_normal((M + 1, I))
    sn2 = npr.standard_normal((M + 1, I))
    poi = npr.poisson(lamb * dt, (M + 1, I))
    for t in range(1, M + 1, 1):
        S[t] = S[t - 1]*(np.exp((r-rj-0.5*sigma**2)*dt
            + sigma * np.sqrt(dt) * sn1[t])
            + (np.exp(mu + delta * sn2[t]) - 1) * poi[t])
        S[t] = np.maximum(S[t], 0)
    return S
```

Variance Reduction

There are two main techniques

- antithetic variates - first moment
- moment matching - first and second moments

```
def gen_sn(M, I, anti_paths=True, mo_match=True):
    ''' Function to generate random numbers for simulation.
    Parameters
    M : int  number of time intervals for discretization
    I : int  number of paths to be simulated
    anti_paths: Boolean  use of antithetic variates
    mo_math : Boolean  use of moment matching'''
    if anti_paths is True: # "is True" is different from == True
        sn = npr.standard_normal((M + 1, I / 2)) # generate half
        sn = np.concatenate((sn, -sn), axis=1) # add negative half
    else:
        sn = npr.standard_normal((M + 1, I))
    if mo_match is True:
        sn = (sn - sn.mean()) / sn.std()
    return sn
```

Valuation

We will be doing valuation on two different types of derivatives

- European options
- American options

Valuation - European options

European options

Pricing by risk-neutral expectation

$$C_0 = e^{-rT}\mathbb{E}_0^Q(h(S_T)) = e^{-rT}\int_0^\infty h(s)q(s)\,ds$$

Risk-neutral Monte Carlo estimator

$$\tilde{C}_0 = e^{-rT}\frac{1}{\mathrm{I}}\sum_{i=1}^{\mathrm{I}} h(\tilde{S}_T^i)$$

where $h(S_t)$ stands for the payoff function,
S_t is the index level at maturity,
and $\tilde{S}_t^i$ stands for the ith simulated index level at maturity

Valuation - European call options

$$\tilde{C}_0 = e^{-rT}\frac{1}{\mathrm{I}}\sum_{i=1}^{\mathrm{I}} h(\tilde{S}_T^i)$$

$$\tilde{S}_T = S_0 \exp\{(r - \tfrac{1}{2}\sigma^2)T + \sigma\sqrt{T}\tilde{z}_T\}$$

```
def gbm_mcs_stat(K, S0 = 100, r = 0.05, sigma = 0.25, T = 1.0, I = 50000):
    ''' Valuation of European call option in Black-Scholes-Merton
    by Monte Carlo simulation (of index level at maturity)
    Parameters
    ==========
    K : float     (positive) strike price of the option
    C0 : float     estimated present value of European call option
    '''
    sn = gen_sn(1, I)
    # simulate index level at maturity
    ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma * np.sqrt(T) * sn[1])
    # calculate payoff at maturity
    hT = np.maximum(ST - K, 0)
    # calculate MCS estimator
    C0 = np.exp(-r * T) * 1 / I * np.sum(hT)
    return C0
```

Valuation - European options

We can make a generic pricing function!

$$\tilde{C}_0 = e^{-rT} \frac{1}{\mathrm{I}} \sum_{i=1}^{\mathrm{I}} h(\tilde{S}_T^i)$$

$$\tilde{S}_T = S_0 \exp\{(r - \tfrac{1}{2}\sigma^2)T + \sigma\sqrt{T}\tilde{z}_T\}$$

```
def gbm_mcs_stat_generic(K, payoff_f, S0=100, r=0.05, sigma=0.25, T=1.0, I=50000):
    ''' Valuation of European call option in Black-Scholes-Merton
    by Monte Carlo simulation (of index level at maturity)
    Parameters
    payoff: a payoff function for the european option
    K : float     (positive) strike price of the option
    C0 : float     estimated present value of European call option
    '''
    sn = gen_sn(1, I)
    # simulate index level at maturity
    ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma * np.sqrt(T) * sn[1])
    # calculate payoff at maturity
    hT = payoff_f(ST - K)
    # calculate MCS estimator
    C0 = np.exp(-r * T) * 1 / I * np.sum(hT)
    return C0
```

Valuation - European options

Testing the generic pricing function using payoff function of call option

```
def call_payoff_function(x):
    return np.maximum(x, 0)
```

```
>>> gbm_mcs_stat_generic(105, call_payoff_function)
10.020535172468639
```

```
>>> gbm_mcs_stat(105)
9.987946616123371 9
```

Valuation - European options

We can adopt the dynamic simulation approach

```
def gbm_mcs_dyna(K, M = 50, option='call', r = .05, sigma = .25):
    ''' Valuation of European options in Black-Scholes-Merton
    by Monte Carlo simulation (of index level paths)
    Parameters
    K : float     (positive) strike price of the option
    option : string    type of the option to be valued ('call', 'put')
    C0 : float    estimated present value of European call option
    '''
    T = 1.0;   I = 20000;   dt = T / M;   S0 = 100
    # simulation of index level paths
    S = np.zeros((M + 1, I))
    S[0] = S0
    sn = gen_sn(M, I)
    for t in range(1, M + 1):
        S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt
                         + sigma * np.sqrt(dt) * sn[t])
    # case-based calculation of payoff
    if option == 'call':
        hT = np.maximum(S[-1] - K, 0)
    else:
        hT = np.maximum(K - S[-1], 0)
    # calculation of MCS estimator
    C0 = np.exp(-r * T) * 1 / I * np.sum(hT)
    return C0
```

optimal stopping approach - theory not understood yet

$$V_0 = \sup_{\tau \in \{0, {}^{\Delta}t, 2^{\Delta}t, \ldots, T\}} e^{-rT} \mathbb{E}_0^Q (h_\tau(S_\tau))$$

Least-sqaures regression for American option valuation

$$\min_{\alpha_{1,t},\ldots,\alpha_{D,t}} \frac{1}{\mathrm{I}} \sum_{d=1}^{D} (\alpha_{d,t} \cdot b_d(S_t, i))^2$$

Valuation - American options

```
def gbm_mcs_amer(K, S0=100, T=1, M=50, I= 20000, option='call', r=.05, sigma=.25):
    ''' Valuation of American option in Black-Scholes-Merton by MCS LSM
    K : float    (positive) strike price of the option
    option : string    type of the option to be valued ('call', 'put')
    C0 : float    estimated present value of European call option    '''
    dt = T / M;    df = np.exp(-r * dt)
    # simulation of index levels
    S = np.zeros((M + 1, I));    S[0] = S0;    sn = gen_sn(M, I)
    for t in range(1, M + 1):
        S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt
        + sigma * np.sqrt(dt) * sn[t])
    if option == 'call': # case-based calculation of payoff
        h = np.maximum(S - K, 0)
    else:
        h = np.maximum(K - S, 0)
    # LSM algorithm
    V = np.copy(h)
    for t in range(M - 1, 0, -1):
        reg = np.polyfit(S[t], V[t + 1] * df, 7)
        C = np.polyval(reg, S[t])
        V[t] = np.where(C > h[t], V[t + 1] * df, h[t])
    C0 = df * 1 / I * np.sum(V[1]) # MCS estimator
    return C0
```

Risk measures

- Value at Risk
- Credit Value Adjustment

Value at Risk - Black Scholes' World

Holding a stock has probability of $x\%$ to suffer a loss greater than y

scs: scipy.stats

```
import numpy as np
import numpy.random as npr
import scipy.stats as scs

def VaR_stock_price(S0=100, r=.05, sigma=.25, T=30/365, I=10000):
    ST = S0*np.exp((r-0.5*sigma**2)*T+sigma*np.sqrt(T)*npr.standard_normal(I))
    R_gbm = np.sort(ST - S0) / S0
    percs = [0.01, 0.1, 1., 2.5, 5.0, 10.0]
    var = scs.scoreatpercentile(R_gbm, percs)
    print("%16s %16s" % ('Confidence Level', 'Value-at-Risk'))
    print(33 * "-")
    for pair in zip(percs, var):
        print("%16.2f %16.3f" % (100 - pair[0], -pair[1]))
```

Holding a stock has probability of $x\%$ to suffer a loss greater than y

```
def VaR_jump_diffusion(r=.05, sigma=.2, lamb=.75, mu=-.6, delta=.25):
    M = 50;    I = 10000;    dt = 30. / 365 / M
    rj = lamb * (np.exp(mu + 0.5 * delta ** 2) - 1)
    S = np.zeros((M + 1, I));    S0 = 100;    S[0] = S0
    sn1 = npr.standard_normal((M + 1, I))
    sn2 = npr.standard_normal((M + 1, I))
    poi = npr.poisson(lamb * dt, (M + 1, I))
    for t in range(1, M + 1, 1):
        S[t] = S[t - 1]*(np.exp((r-rj-0.5*sigma**2)*dt
            + sigma * np.sqrt(dt) * sn1[t])
            + (np.exp(mu + delta * sn2[t]) - 1) * poi[t])
        S[t] = np.maximum(S[t], 0)
    R_jd = np.sort(S[-1] - S0)
    percs = [0.01, 0.1, 1., 2.5, 5.0, 10.0]
    var = scs.scoreatpercentile(R_jd, percs)
    print("%16s %16s" % ('Confidence Level', 'Value-at-Risk'))
    print(33 * "-")
    for pair in zip(percs, var):
        print("%16.2f %16.3f" % (100 - pair[0], -pair[1]))
```

Credit Value Adjustment through CVaR

Holding a stock has probability of $x\%$ to suffer a loss greater than y

```
def CVaR():
    S0 = 100.
    r = 0.05
    sigma = 0.2
    T = 1.
    I = 100000
    ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T
                     + sigma * np.sqrt(T) * npr.standard_normal(I))
    L = 0.5
    p = 0.01
    D = npr.poisson(p * T, I)
    D = np.where(D > 1, 1, D)
    CVaR = np.exp(-r * T) * 1 / I * np.sum(L * D * ST)
    S0_CVA = np.exp(-r * T) * 1 / I * np.sum((1 - L * D) * ST)
    S0_adj = S0 - CVaR
    return (CVaR, S0_CVA, S0_adj)
```

Thank You

E0007424@u.nus.edu