WORLDQUANT UNIVERSITY

Video Transcripts Module 6

MScFE 630

Computational Finance

Revised: 09/07/2020

Table of Contents

Unit 1: Short Rate Models

In this video we will go over what a short rate is as well as some models that have been proposed for modelling this short rate.

Up until this point, we have made the assumption that interest rates are constant. Whilst this makes the mathematics underlying our models significantly easier, this assumption is directly violated by what is observed in markets. For example, with a continuously compounded risk-free interest rate of r, a zero coupon bond with maturity of time T has a price of e^{-rT}, and all bonds will have the same yield to maturity. This is not observed in practice, and so we want to introduce some models for interest rates. The focus in this module will be on simulating interest rates and discount factors, which can be used with Monte Carlo techniques for pricing derivatives.

In order to overcome this, we are going to model interest rates using stochastic differential equations (SDEs). The first class of models that we will look at in trying to model interest rates is short rate models. These models attempt to model the instantaneous continuously-compounded interest rate at time t, r_t. This means that an investment of $\$X$ which is invested at this risk-free rate will grow to $\$\mathrm{X}e^{r_t\delta_t}$, over a very small time interval, δ_t. If we wanted to find the value of our investment over a period, we would have to integrate the short rate over that period. In other words, our investment would grow to $Xe^{\int_0^t r_s ds}$ at time t.

The first short rate model that we will look at is the Vasicek model. This models specifies a SDE for the short rate of:

$$dr_t = \alpha(\beta - r_t)dt + \sigma dW_t, r_0 = r(0)$$

where W_t is a standard Brownian motion, and α, β, and σ are positive constants. Let's spend a little time going over what the parameters in the above model mean. β is known as the mean-reversion level. This is the average rate that the short rate tends towards over time. α is the rate at which the short rate tends towards β. This means that, the larger α is, the faster the short rate tends towards its mean level and vice versa. Finally, σ is just the volatility of the short rate.

Using these dynamics, we can derive an expression for the short rate of the following form:

$$r_t = r(0)e^{-\alpha t} + b(1 - e^{-\alpha t}) + \sigma e^{-\alpha t} \int_0^t e^{\alpha s}\, dW_s$$

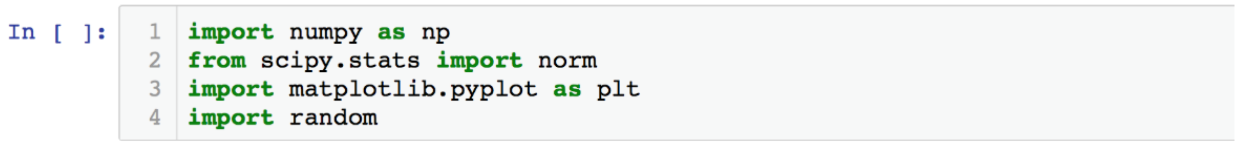
This means that the short rate under the Vasicek model is normally distributed.

Now we can move on to simulating paths for the short rate. Let's go through how to simulate 5 sample paths over a 1 year period. The exact formula that we are using for our simulations is as follows (and can now be seen on screen):

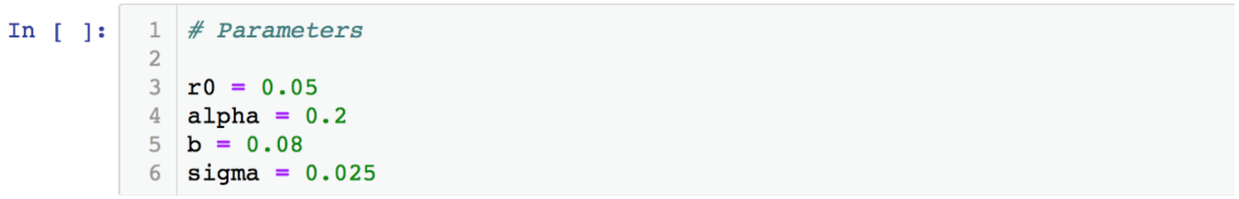
$$r_{i+1} = r_i e^{-\alpha(t_{i+1}-t_i)} + b\left(1 - e^{-\alpha(t_{i+1}-t_i)}\right) + \sqrt{\frac{\sigma^2}{2\alpha}(1 - e^{-2\alpha(t_i+1-t_i)})}Z_i$$

where Z_i is a standard normal random variable.

Now, let's go through the code to apply this formula. The first thing to do will be to import the relevant libraries, and to set our parameter values

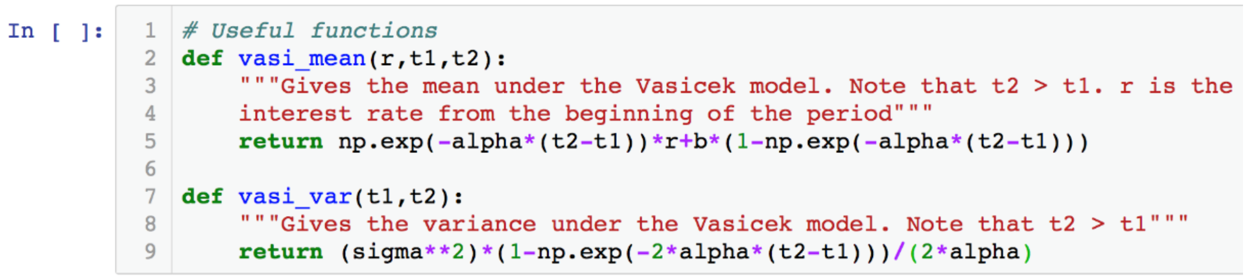


```
In [ ]:  import numpy as np
         from scipy.stats import norm
         import matplotlib.pyplot as plt
         import random
```



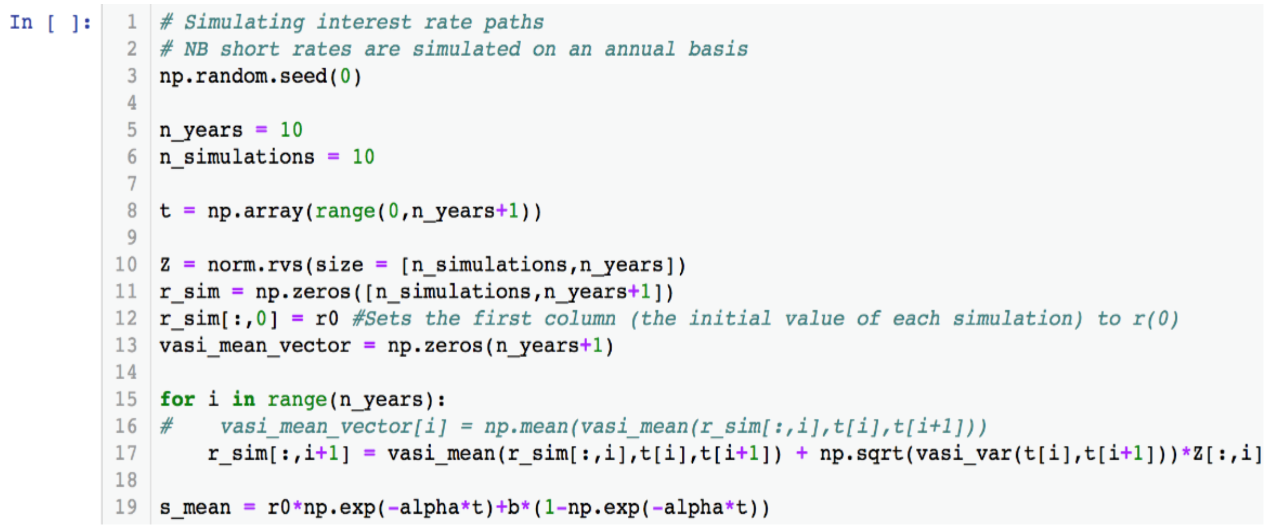
```
In [ ]:  # Parameters

         r0 = 0.05
         alpha = 0.2
         b = 0.08
         sigma = 0.025
```

Then we can define some functions which will make our code a bit easier to read. The vasi_mean and vasi_var functions return the mean and variance of the short rate under the Vasicek Model respectively.



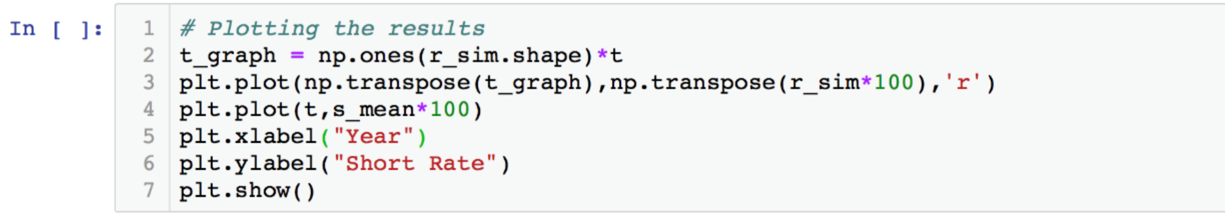
```
In [ ]:  # Useful functions
         def vasi_mean(r,t1,t2):
             """Gives the mean under the Vasicek model. Note that t2 > t1. r is the
             interest rate from the beginning of the period"""
             return np.exp(-alpha*(t2-t1))*r+b*(1-np.exp(-alpha*(t2-t1)))

         def vasi_var(t1,t2):
             """Gives the variance under the Vasicek model. Note that t2 > t1"""
             return (sigma**2)*(1-np.exp(-2*alpha*(t2-t1)))/(2*alpha)
```

Moving onto the actual simulation, we first set our seed. Then we pre-allocate space for our simulations. We will treat each row as an individual simulation, so we will need to set the first column of our pre-allocated matrix to be equal to our r0 variable, which is the current interest rate. Thereafter, we loop over the number of time intervals, and update our interest rates using the previous formula as we go.



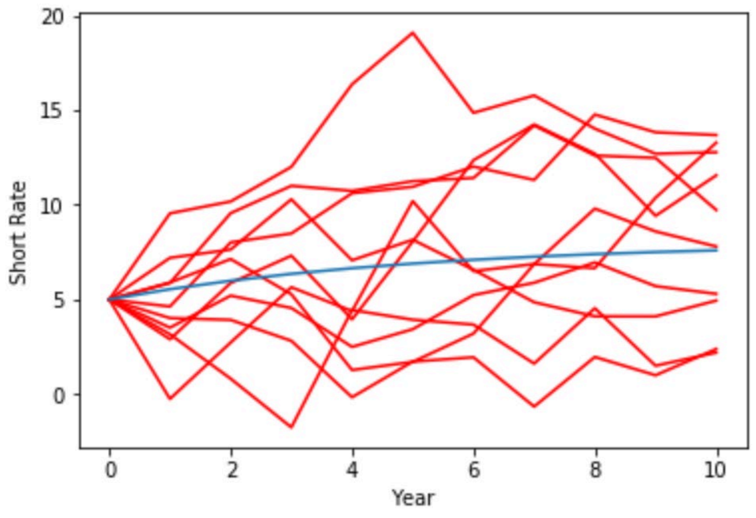
```
In [ ]:
# Simulating interest rate paths
# NB short rates are simulated on an annual basis
np.random.seed(0)

n_years = 10
n_simulations = 10

t = np.array(range(0,n_years+1))

Z = norm.rvs(size = [n_simulations,n_years])
r_sim = np.zeros([n_simulations,n_years+1])
r_sim[:,0] = r0 #Sets the first column (the initial value of each simulation) to r(0)
vasi_mean_vector = np.zeros(n_years+1)

for i in range(n_years):
#    vasi_mean_vector[i] = np.mean(vasi_mean(r_sim[:,i],t[i],t[i+1]))
    r_sim[:,i+1] = vasi_mean(r_sim[:,i],t[i],t[i+1]) + np.sqrt(vasi_var(t[i],t[i+1]))*Z[:,i]

s_mean = r0*np.exp(-alpha*t)+b*(1-np.exp(-alpha*t))
```

Finally, we can plot our results.



```
In [ ]:
# Plotting the results
t_graph = np.ones(r_sim.shape)*t
plt.plot(np.transpose(t_graph),np.transpose(r_sim*100),'r')
plt.plot(t,s_mean*100)
plt.xlabel("Year")
plt.ylabel("Short Rate")
plt.show()
```

Which should result in the following graph:



Now we can go over the Hull-White model. The Hull-White model specifies an SDE for the short rate of:

$$dr_t = (\theta(t) - \alpha(t)r_t)dt + \sigma(t)dW_t, r_0 = r(0)$$

where W_t is a standard Brownian motion, and $\theta(t), \alpha(t)$, and $\sigma(t)$ can be time dependent. Note that the Hull-White model is a more general version of the Vasicek model, which allows it to be better calibrated to market observed prices or rates by allowing parameters to vary over time.

Finally, let's look at the Cox-Ingersoll-Ross, or CIR, model. This model specifies an SDE for the short rate of:

$$d\, r_t \; = \; \alpha(b - r_t)dt \; + \; \sigma\sqrt{r_t}dW_t, r_0 \; = \; r(0)$$

where, as usual, W_t is a standard Brownian motion. You can find more information on how to interpret the parameters in the notes. What makes the CIR model so attractive is that, if $2\alpha b \geq \sigma^2$ and $r(0)$ is positive, then the short rate will also always be positive. This is generally a desirable quality for a short rate model because negative interest rates don't generally make sense. However, in recent times, negative rates have been observed in some developed countries.

Non-constant Interest Rate Pricing

Suppose the short-term rate at time t is given by r_t, and we wanted to price a derivative on some underlying process given by X_t, where X_t, could be, for example, a stock price. If the derivative has a maturity of time T, and a payoff function $\Phi(\cdot)$, the price of the option at time o is given by:

$$P = \mathbb{E}^{\mathbb{Q}}\left[e^{-\int_0^T r_t dt}\Phi(X_T)\right] \tag{2.1}$$

Where $\mathbb{Q}$ is the risk-neutral measure. Note that the underlying process, X_t, can be something related to interest rates. This formula is how one would price interest rate derivatives such as caps.

In the next video we will look at how we can calculate these integrals.

Unit 2: Using Short Rates

In this video, we will go through how we can go about using some of the short rate models presented in the previous video to price financial instruments. An example of a financial instrument that we may want to simulate prices for is a Zero Coupon Bond or ZCB. This is just a bond that has a payoff of 1 at maturity and, as the name suggests, doesn't pay any coupons. Because a ZCB has a payoff of 1 at maturity, under no arbitrage, its current value must be discounted. Thus, estimating the price of a ZCB is equivalent to finding average discount factors.

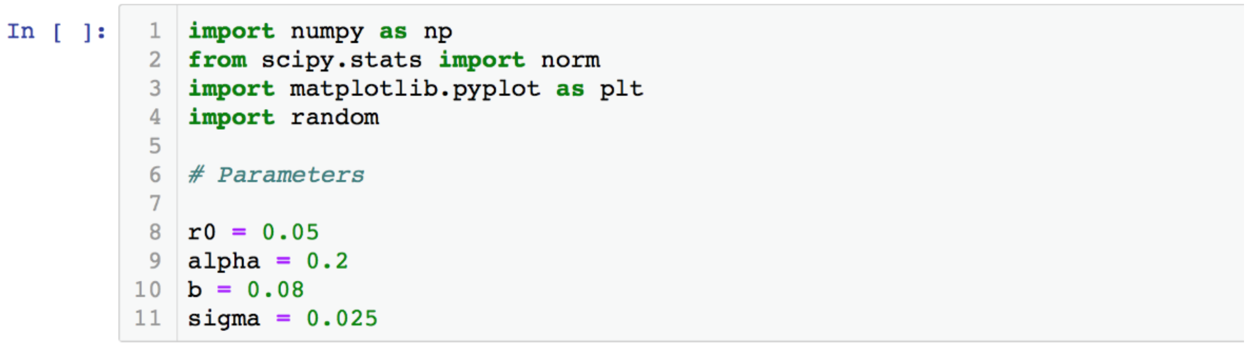
We can write the time t price of a ZCB, with maturity T as:

$$B(t,T) = \mathbb{E}\left[e^{-\int_t^T r_s\,ds}\right]$$

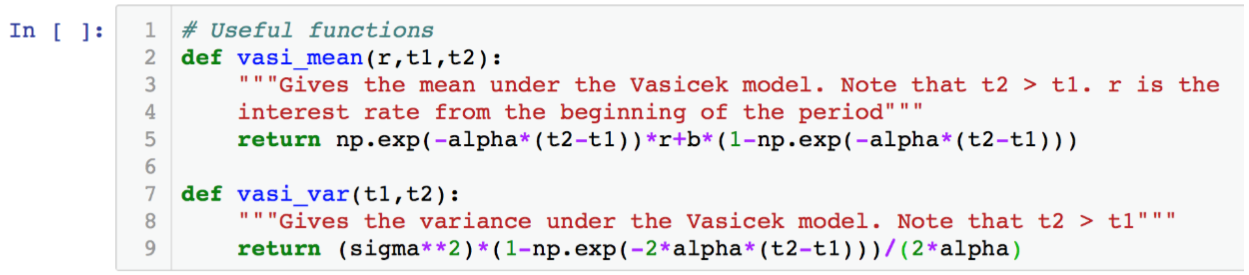
where the expectation is taken under the risk-neutral measure. Under the Vasicek model, there is a closed-form solution for a bond price. This solution can be found in the notes on this section.

Now, let's go through how to simulate bond prices using Vasicek dynamics in Python. Remember that this is important because it forms the basis for Monte Carlo estimation.

We first import the relevant libraries, set up our parameter values, and define some useful functions. Note that these functions are the same as the ones we used in the previous video.

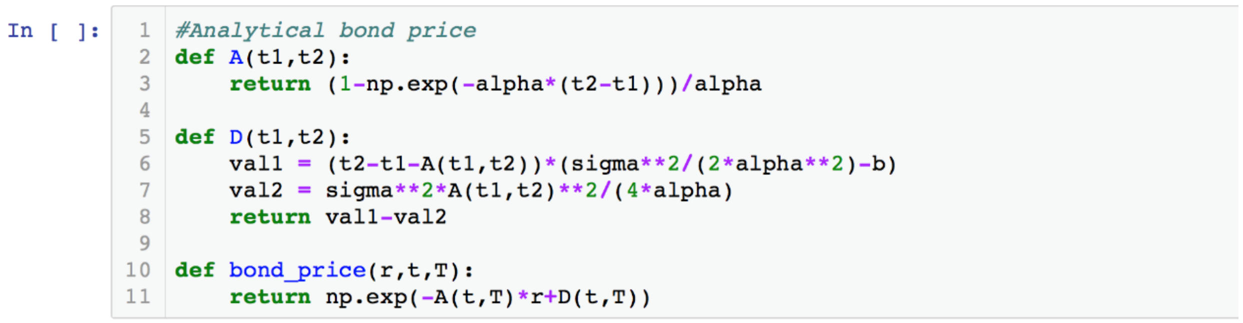


```
In [ ]:
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
import random

# Parameters

r0 = 0.05
alpha = 0.2
b = 0.08
sigma = 0.025
```



```
In [ ]:
# Useful functions
def vasi_mean(r,t1,t2):
    """Gives the mean under the Vasicek model. Note that t2 > t1. r is the
    interest rate from the beginning of the period"""
    return np.exp(-alpha*(t2-t1))*r+b*(1-np.exp(-alpha*(t2-t1)))

def vasi_var(t1,t2):
    """Gives the variance under the Vasicek model. Note that t2 > t1"""
    return (sigma**2)*(1-np.exp(-2*alpha*(t2-t1)))/(2*alpha)
```

Let’s also define some additional functions which will be useful a bit later on. These functions are relevant to pricing the closed-form solution for the bond price. The bond price function uses the A and D functions to return the bond price under the Vasicek model.



```
In [ ]:
#Analytical bond price
def A(t1,t2):
    return (1-np.exp(-alpha*(t2-t1)))/alpha

def D(t1,t2):
    val1 = (t2-t1-A(t1,t2))*(sigma**2/(2*alpha**2)-b)
    val2 = sigma**2*A(t1,t2)**2/(4*alpha)
    return val1-val2

def bond_price(r,t,T):
    return np.exp(-A(t,T)*r+D(t,T))
```

In order to improve the accuracy of our simulations, it would make sense to estimate both r_t and $\int_0^t r_s ds$ simultaneously. If we don’t do this, we would have to estimate the integral later which would result in our final value being less accurate. This would directly affect the accuracy of our final bond price estimates.

First, let $Y_t = \int_0^t r_s ds$. We can show that Y_t and r_t have a joint Gaussian distribution. This means that they should be relatively easy to simulate if we know their means, variances and correlations.

We already have the mean and variance for r_t from the previous section. We can also show that, for some time $t_2 > t_1$ the mean of Y_t is:

$$Y_{t_1} + (t_2 - t_1)b + \left(r_{t_1} - b\right)A(t_1, t_2)$$

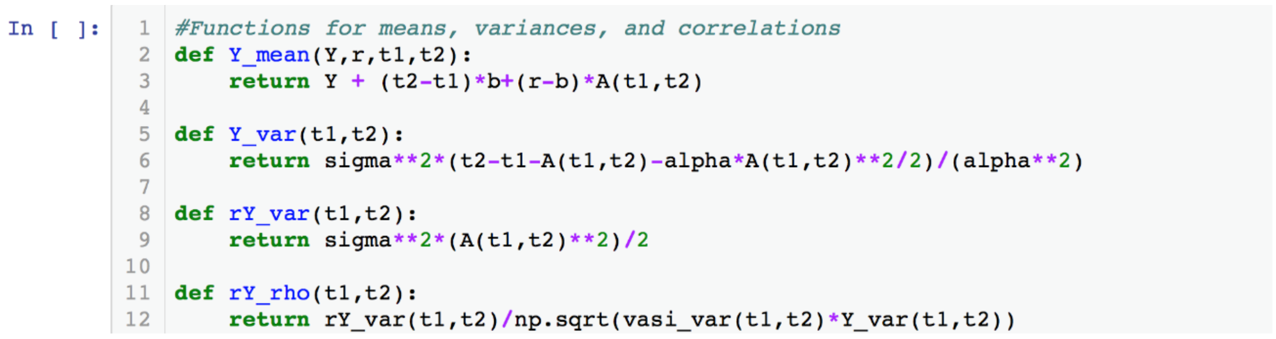
and that the variance of Y_t is:

$$\frac{\sigma^2}{\alpha^2}(t_2 - t_1 - A(t_1, t_2) - \alpha \frac{A(t_1, t_2)^2}{2}$$

Finally, we can show that the covariance given the filtration up until time s between Y_t and r_t, is:

$$\frac{\sigma^2 A(t_1, t_2)^2}{2}$$

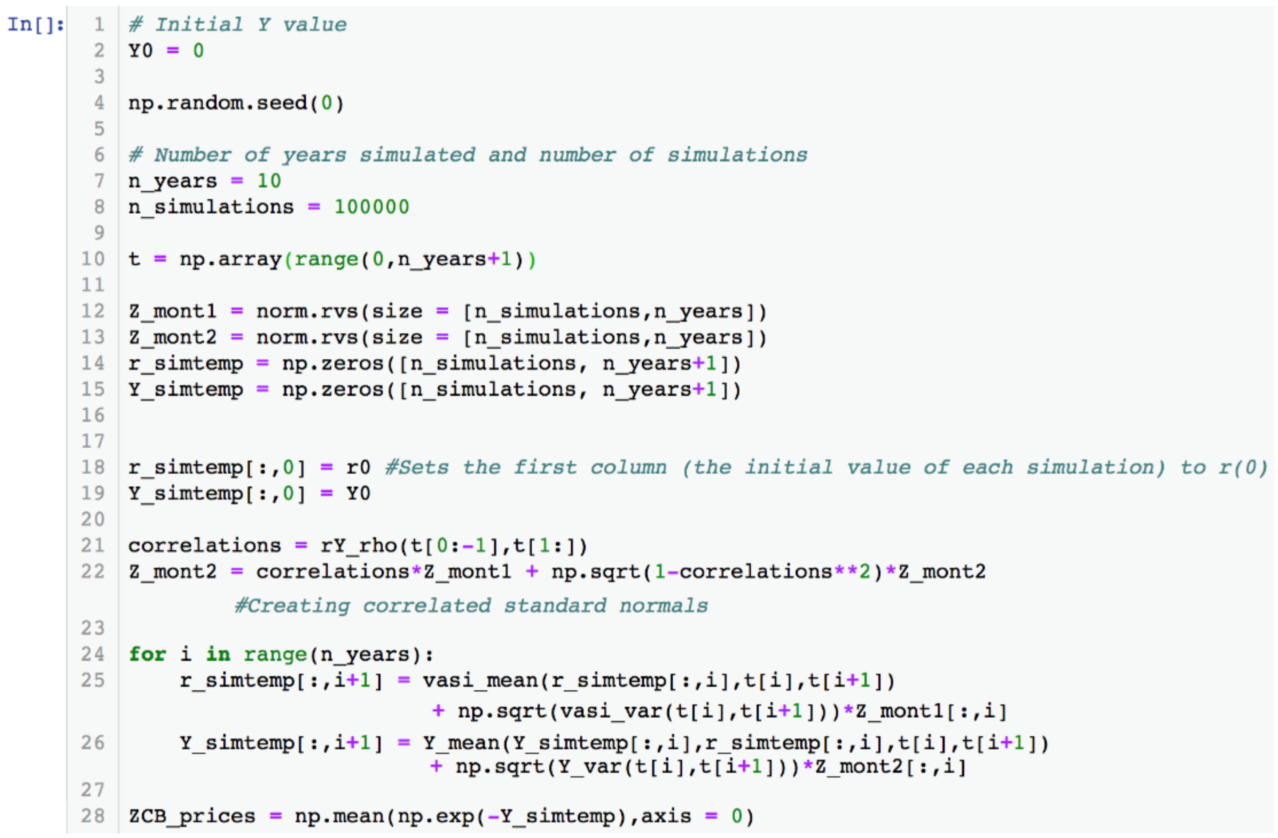
Given these new expressions, it would make sense to define functions for them. This is done with:



```
In [ ]:
#Functions for means, variances, and correlations
def Y_mean(Y,r,t1,t2):
    return Y + (t2-t1)*b+(r-b)*A(t1,t2)

def Y_var(t1,t2):
    return sigma**2*(t2-t1-A(t1,t2)-alpha*A(t1,t2)**2/2)/(alpha**2)

def rY_var(t1,t2):
    return sigma**2*(A(t1,t2)**2)/2

def rY_rho(t1,t2):
    return rY_var(t1,t2)/np.sqrt(vasi_var(t1,t2)*Y_var(t1,t2))
```

The function Y_mean calculates the mean for Y_{t_2} given Y_{t_1} and r_{t_1} , where $t_1 < t_2$. Y_var calculates the variance for Y_{t_2} from time $t_1 < t_2$. rY_var calculates the covariance between Y_{t_2} and r_{t_2} from time $t_1 < t_2$, and rY_rho finds the correlation from this covariance.

Let's finally look at the code which simulates the bond prices:



```
In[]:
# Initial Y value
Y0 = 0

np.random.seed(0)

# Number of years simulated and number of simulations
n_years = 10
n_simulations = 100000

t = np.array(range(0,n_years+1))

Z_mont1 = norm.rvs(size = [n_simulations,n_years])
Z_mont2 = norm.rvs(size = [n_simulations,n_years])
r_simtemp = np.zeros([n_simulations, n_years+1])
Y_simtemp = np.zeros([n_simulations, n_years+1])


r_simtemp[:,0] = r0 #Sets the first column (the initial value of each simulation) to r(0)
Y_simtemp[:,0] = Y0

correlations = rY_rho(t[0:-1],t[1:])
Z_mont2 = correlations*Z_mont1 + np.sqrt(1-correlations**2)*Z_mont2
        #Creating correlated standard normals

for i in range(n_years):
    r_simtemp[:,i+1] = vasi_mean(r_simtemp[:,i],t[i],t[i+1])
                       + np.sqrt(vasi_var(t[i],t[i+1]))*Z_mont1[:,i]
    Y_simtemp[:,i+1] = Y_mean(Y_simtemp[:,i],r_simtemp[:,i],t[i],t[i+1])
                       + np.sqrt(Y_var(t[i],t[i+1]))*Z_mont2[:,i]

ZCB_prices = np.mean(np.exp(-Y_simtemp),axis = 0)
```

In this code, we first initialize our Y0 variable as being 0. Now, we are going to be simulating over a 10 year period and for each period, we are going to perform 100,000 simulations. The t variable creates a vector going from 0 to 10. This represents our time periods, which means that we will be doing a simulation each year.

We then sample from the normal distribution for our simulations. We can do this all at once because we will be doing the same number of simulations at each time point. Doing it all at once can help to make our code a little bit more efficient.

Last, we calculate the correlations between our r and Y variables using one of our functions, and then use this to generate a correlated normal sample. The last step is to perform the actual simulations.

In the for loop, we first simulate the interest rate at the next time point. We then use this interest rate to simulate a value for Y at that time point as well. Once we have run through all time points, we can then generate values for our zero coupon bond prices.

If we wanted to price a derivative, we would simulate values for the underlying jointly with our Y variables, and discount the payoff for the simulated underlying using the discount factor implied by our simulated Y variables. The mean of these discounted payoffs would give us a value estimate for our derivative.

We have only gone through one method for simulating ZCB prices. However, the notes also go through a few other methods for simulation.

In the next video, we will go through the LIBOR Forward Market Model, and how we can simulate this model in Python.

Unit 3: The LIBOR Model

The LIBOR Forward Market Model (LFMM) is what is known as a market model. This means it aims to model instruments which are directly traded on the market, rather than idealized quantities like the short rate. Its popularity came largely as a result of the fact that it results in the analytical Black price for market caps. This is generally a desirable characteristic of the model as it gives results which can be directly checked against the market. It also helps when it comes to calibrating the model, as the model can then be calibrated using easily observable market instruments. These properties have led to the LFMM being seen as the benchmark interest rate model.

The LFMM gets complicated very quickly when using a lot of market data, and so we will be limiting ourselves to the simplest case, in order to illustrate the workings of the model.

We first introduce simple and forward rates. A simple rate is a rate which is not compounded, meaning an investment of A with grow to $A(1+\ it)$, where $t$is the period of the investment and i is the simple rate. A forward rate is a rate in the future which you can agree on/lock in today. The LFMM models forward simple rates i.e. simple rates in the future.

Before we can go through how to run simulations using the LFMM, we first need to define some notation. Suppose we have a set of dates, $\{T_0, T_1, \dots, T_N\}$. Let $P_j(t)$ be the price of a bond at time t which expires at time T_j, $F\left(t, T_j, T_{j+1}1\right) = F_j(t)$ be the forward rate between times T_j and T_{j+1} at time t, and $\delta_j = T_{j+1} - T_j$ be the time between two dates. The LFMM says that each market forward rate has the following SDE:

$$dF_j(t) = F_j(t)\mu_j(t)dt + F_j(t)\sigma_j(t)dW_t$$

where $\mu_j(t)$ and $\sigma_j(t)$ are the (time-dependent) drift and volatilities associated with $F_j(t)$, and W_t is a Brownian motion. There can be more than one Brownian motion (e.g. one for each forward rate), but we assume there is just one. It can be shown that:

$$\mu_j(t) = \sum_{k=\tau(t)}^{j} \frac{\delta_k F_k(t)\sigma_k(t)\sigma_j(t)}{1+\delta_k F_k(t)}$$

where $\tau(t) = \min\{i : t < T_i\}$.

With this information, we get a basic approximation for the j^{th} forward rate at time t_i:

$$\hat{\mathrm{F}}_j(t_i) = \hat{\mathrm{F}}_j(t_{i-1})\exp\left[\left(\hat{\mu}_j(t_{i-1}) - \frac{1}{2}\sigma^2\right)\delta_{i-1} + \sigma_j\sqrt{\delta_{i-1}}Z_i\right] \tag{6.1}$$

where $\hat{\mathrm{F}}_j(t_i)$ is the approximation $Z_i \sim N(0,1)$, and:

$$\hat{\mu}_j(t_{i-1}) = \sum_{k=i}^{j} \frac{\delta_k \hat{F}_j(t_{i-1})\sigma_k\sigma_j}{1 + \delta_k \hat{F}_j(t_{i-1})} \tag{6.2}$$

Essentially, we are using the fact that the forward rates have a log-normal distribution, and simulating possible movements for the rate over time. All we need to apply this using Monte Carlo simulation are values for $\hat{F}_j(t_0)$. We can initialize these values to $\hat{F}_j(t_0) = F_j\,(t_0)$, where $F_j(t_0)$ are implied from the market zero-coupon bond prices, and the following formula:

$$F_j(t_0) = \frac{P_j(0) - P_{j+1}(0)}{\delta_0 P_{j+1}(0)} \tag{6.3}$$

Note that, by initializing our values to the current market values, our model is automatically calibrated (don't worry if you don't know what this means – it will be covered in the final module).

This approximation can be improved by considering the average of the drifts over two periods given in the standard approximation. We start in exactly the same way by initializing $\hat{F}_j(t_0) = F_j\,(t_0)$. Now, we first estimate the forward rate for the next period using:

$$\tilde{F}_j(t_i) = \bar{F}_j(t_{i-1})\exp\left[\left(\mu_j^1(t_i - 1) - \frac{1}{2}\sigma^2\right)\delta_{i-1} + \sigma_j\sqrt{\delta_{i-1}}\,Z_i\right] \tag{6.4}$$

where:

$$\mu_j^1(t) = \sum_{k=i}^{j} \frac{\delta_k \bar{F}_j(t_{i-1})\sigma_k\sigma_j}{1 + \delta_k \bar{F}_j(t_{i-1})} \tag{6.5}$$

We then use these forward estimates to estimate the drift at time t_i:

$$\mu_j^2(t_{i-1}) = \sum_{k=i}^{j} \frac{\delta_k \tilde{F}_j(t_i)\sigma_k\sigma_j}{1 + \delta_k \tilde{F}_j(t_i)} \tag{6.6}$$

The final step is to compute the final estimate for the forward rate in the next period using:

$$\bar{F}_j(t_i) = \bar{F}_j(t_{i-1})\exp\left[\frac{1}{2}\left(\mu_j^1(t_{i-1}) + \mu_j^2(t_{i-1}) - \sigma^2\right)\delta_{i-1} + \sigma_j\sqrt{\delta_{i-1}}Z_i\right] \tag{6.7}$$

Note that the Z_i value used for the preliminary and the final forward estimate is the same. This improved approximation is known as the Predictor-Correcter method.

Let's take a quick look at how we can implement the Predictor-Correcter method in Python.

We first need to import the libraries we are going to be using and setting our initial parameters. The first group of parameters is as per the Vasicek model. The second group is for our LFMM. Note that we are assuming that volatility is constant through time (this is variable sigmaj) and are finding the forward rates at the times two years apart for 40 years.

In [ ]:

```python
#Libraries
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
import random
```

In [ ]:

```python
# Parameters
r0 = 0.05
alpha = 0.2
b = 0.08
sigma = 0.025

# Problem parameters
t = np.linspace(0,40,21)
sigmaj = 0.2
```

This code is used to generate synthetic bond prices, which we will be using for setting our initial forward rates in our LFMM. Note that you would usually use real world bond prices at this point.

In [ ]:

```python
# Vasicek Bond prices
def A(t1,t2):
    return (1-np.exp(-alpha*(t2-t1)))/alpha

def C(t1,t2):
    val1 = (t2-t1-A(t1,t2))*(sigma**2/(2*alpha**2)-b)
    val2 = sigma**2*A(t1,t2)**2/(4*alpha)
    return val1-val2

def bond_price(r,t,T):
    return np.exp(-A(t,T)*r+C(t,T))

vasi_bond = bond_price(r0,0,t)
```

We use these synthetic bond prices to set initial forward rate values in line 6. The delta variable measures the difference between times, which we will need to vectorize our code.

```
1  # Applying the algorithms
2  np.random.seed(0)
3  n_simulations = 100000
4  n_steps = len(t)predcorr_forward = np.ones([n_simulations, n_steps -1])*(vasi_bon[:-1]-vasi_bond[1:])
5  /(2*vasi_bond[1:])
6  predcorr_capfac = np.ones([n_simulations, n_steps])delta = np.ones ([n_simulations, n_steps -1])*(t[:-1])
```

For each time step, we implement the Predictor-Correcter method. We set our initial forward rates to the previous times forward rates, and create temporary simulated forward rates using our first drift estimate. We then use these new rates to create a second drift estimate. Finally, we use the average of these two drift estimates to simulate the next time step for-ward rates from our previous forward rates. Make sure you read the notes carefully for this code, to make sure you understand how this is working.

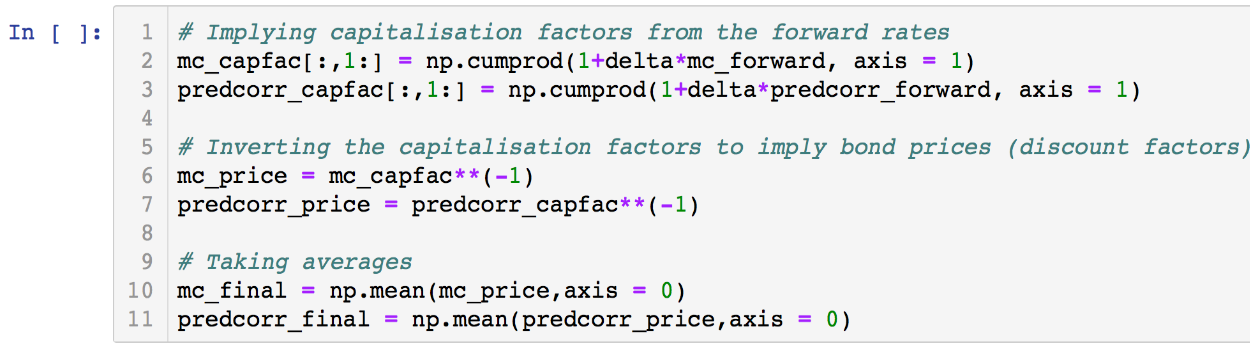
```
for i in range(1,n_steps):
    Z = norm.rvs(size = [n_simulations,1])

    # Explicit Monte Carlo simulation
    muhat = np.cumsum(delta[:,i:]*mc_forward[:,i:]*sigmaj**2/(1+delta[:,i:]*mc_forward[:,i:]),axis = 1)
    mc_forward[:,i:] = mc_forward[:,i:]*np.exp((muhat-sigmaj**2/2)*delta[:,i:]+sigmaj*np.sqrt(delta[:,i:])*Z)

    # Predictor-Corrector Monte Carlo simulation
    mu_initial = np.cumsum(delta[:,i:]*predcorr_forward[:,i:]*sigmaj**2/(1+delta[:,i:]*predcorr_forward[:,i:]),
                           axis = 1)
    for_temp = predcorr_forward[:,i:]*np.exp((mu_initial-sigmaj**2/2)*delta[:,i:]+sigmaj*np.sqrt(delta[:,i:])*Z)
    mu_term = np.cumsum(delta[:,i:]*for_temp*sigmaj**2/(1+delta[:,i:]*for_temp),axis = 1)
    predcorr_forward[:,i:] = predcorr_forward[:,i:]*np.exp((mu_initial+mu_term-sigmaj**2)*delta[:,i:]
                                                           /2+sigmaj*np.sqrt(delta[:,i:])*Z)
```

Finally, we are able to use these forward rates to create capitalization rates using the following formula:

$$C(t_0, t_n) = \prod_{k=1}^{n}\left(1 + \delta_k F_k(t_k)\right)$$

These rates can be used to give us our bond prices, since $B(t_0, t_n) = C(t_0, t_n)^{-1}$.

We use the capitalization from forward rates formula in line 2, transform these factors to prices in line 5, and take the average prices over our simulations in line 8.



```
In [ ]:
# Implying capitalisation factors from the forward rates
mc_capfac[:,1:] = np.cumprod(1+delta*mc_forward, axis = 1)
predcorr_capfac[:,1:] = np.cumprod(1+delta*predcorr_forward, axis = 1)

# Inverting the capitalisation factors to imply bond prices (discount factors)
mc_price = mc_capfac**(-1)
predcorr_price = predcorr_capfac**(-1)

# Taking averages
mc_final = np.mean(mc_price,axis = 0)
predcorr_final = np.mean(predcorr_price,axis = 0)
```

This figure shows the LFMM simulated bond prices versus the known prices from the Vasicek model. Note that the prices implied by the simple LFMM approximation are included as well, which the notes detail on how to implement.

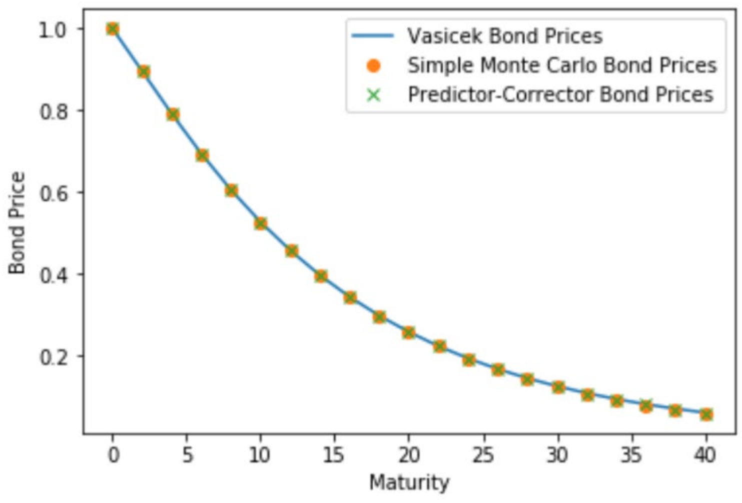


Figure 6.1: Bond Prices

That brings us to the end of Module 6. In Module 7, which is the final module of Computational Finance, we will focus on Calibration.