WorldQuant University

Compiled Content Module 6

MScFE 630 Computational Finance

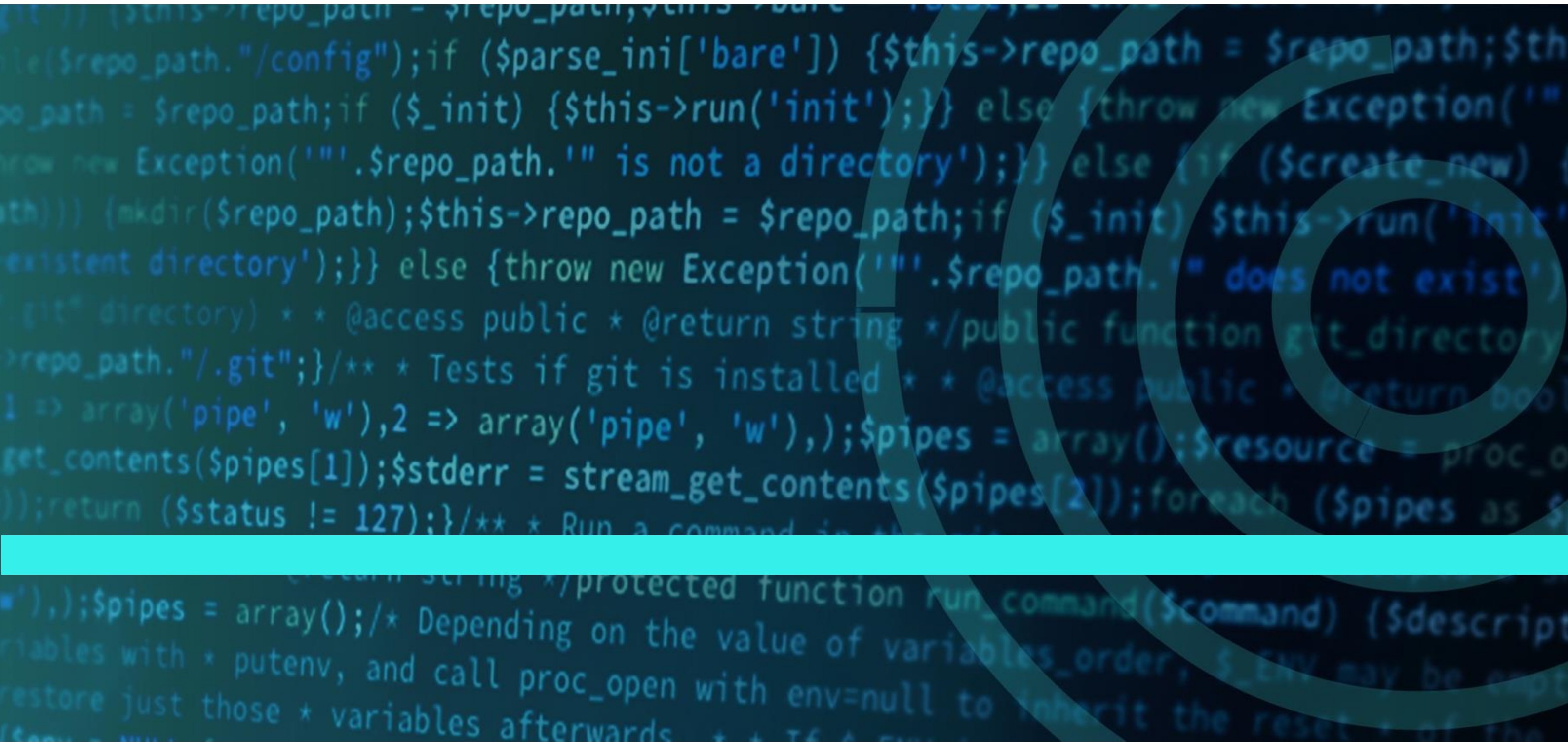


Table of Contents

Revised: 08/19/2019

Module 6: Pricing Interest Rate Options

This module is dedicated to interest rates modelling. The module begins by introducing various short rate models and continues by implementing these models in Python for interest option pricing. Finally, the module introduces the LIBOR Forward Market Model (LFMM) and demonstrates how to implement it in Python.

Unit 1: Short Rate Models

Unit 1: Notes

Introduction

In this module, we will be relaxing our assumption of a constant risk-free continuously-compounded rate, and instead model our interest rates using stochastic differential equations. We will be limiting the amount of mathematics, and instead focus on implementation in Python. This is because we are going to introduce a number of different models.

The first type of model we will be using is a short rate model. These are models which explain the instantaneous continuously-compounded interest rate at time t, r_t. This means that an investment, X, will grow to roughly $Xe^{r_t \Delta t}$over a very short period t. Over a period from 0to t, an investment X will grow $Xe^{\int_0^t r_s ds}$. Alternatively, an asset which has a value of Y at time t will have a present value of $Ye^{-\int_0^t r_s ds}$ at time 0. We will only be showing implementation for the first model, and briefly introducing two more.

Vasicek Model

Detailed derivations for the results in this section can be found in Mamon (2004).

The Vasicek model was proposed by Vasicek (1977), and specifies a stochastic differential equation for the short rate r_t as:

$$dr_t = \alpha(b - r_t)dt + \sigma dW_t, r_0 = r(0) \tag{1.1}$$

where W_t is a standard Brownian motion, $r(0)$ is the short rate at time 0, and α , b, and σ are positive constants. The parameters can be interpreted as follows:

- b is the level to which the short rate will tend in the long-run (mean level).
- α is the rate at which the short rate will tend towards b – the lager
- σ is the volatility of the short rate.

The stochastic differential equation can be solved to give:

$$r_t = r(0)e^{-\alpha t} + b(1 - e^{-\alpha t}) + \sigma e^{-\alpha t}\int_0^t e^{\alpha s}\, dW_s \tag{1.2}$$

Which means that r_t is normally distributed with mean $r(0)e^{-\alpha t} + b(1 - e^{-\alpha t})$ and variance $\frac{\sigma^2}{2\alpha}(1 - e^{-2\alpha t})$. Note that, as $t \to \infty$, the mean of $r_t \to b$ (as expected) and the variance of $r_t \to \frac{\sigma^2}{2\alpha}$.

With these mean and variance estimates, we can go about modelling a path r_t. Note that, if you know the value of r_{t_1} and $t_2 > t_1, r_{t_2}$ is normally distributed with mean $r_{t_1}e^{-\alpha(t_2-t_1)} + b(1 - e^{-\alpha(t_2-t_1)})$ and variance $\frac{\sigma^2}{2\alpha}(1 - e^{-2\alpha(t_2-t_1)})$. We have essentially treated t_1 as a new starting point for our modelling – this is possible due to the independent increments of Brownian motion.

Let's consider how this could be implemented in Python:

```
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
```

We first import the relevant libraries (note we import norm since we are modelling our short term rates using Brownian Motion). We then set the parameters, with $r_0 = 0.05, \alpha = 0.2,\ b = 0.08$, and $\sigma = 0.025$.

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)
```

These two functions, vasi_mean and vasi_var, calculate the mean and variance, respectively, of the short term rate at the time t_2 given the rate at time t_1.

In [ ]:
```
1  # Simulating interest rate paths
2  # NB short rates are simulated on an annual basis
3  np.random.seed(0)
4
5  n_years = 10
6  n_simulations = 10
7
8  t = np.array(range(0,n_years+1))
9
10 Z = norm.rvs(size = [n_simulations,n_years])
11 r_sim = np.zeros([n_simulations,n_years+1])
12 r_sim[:,0] = r0 #Sets the first column (the initial value of each simulation) to r(0)
13 vasi_mean_vector = np.zeros(n_years+1)
14
15 for i in range(n_years):
16 #    vasi_mean_vector[i] = np.mean(vasi_mean(r_sim[:,i],t[i],t[i+1]))
17     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]
18
19 s_mean = r0*np.exp(-alpha*t)+b*(1-np.exp(-alpha*t))
```

We create 10 simulations of possible short term interest rate paths over a period of 10 years. Since the time points we simulate at are all integers, the rates are simulated on an annual basis – you could change this by making the array t include fractional values. In line 11, we create a standard normal random variable for each year and each simulation. The r sim array stores the simulated short rates. The for loop runs through each year, defining rates as:

$$r_{i+1} = r_i e^{-\alpha(t_{i+1}-t_i)} + b(1 - e^{\alpha(t_{i+1}-t_i)} + \sqrt{\frac{\sigma^2}{2\alpha}\left(1 - e^{-2\alpha(t_{i+1}-t_i)}\right)}\, Z_i\,, \tag{1.3}$$

Where $Z_i \sim N(0,1)$. The s_mean array contains the expected value of the short rate at each time point.

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()
```

Finally, we are able to plot our short term interest rate paths. Figure (1.1) shows the 10 simulated short term rate paths. The blue line is the mean short term rate. Note that the rate can become negative – an unfavourable property in many interest rate modeling problems.

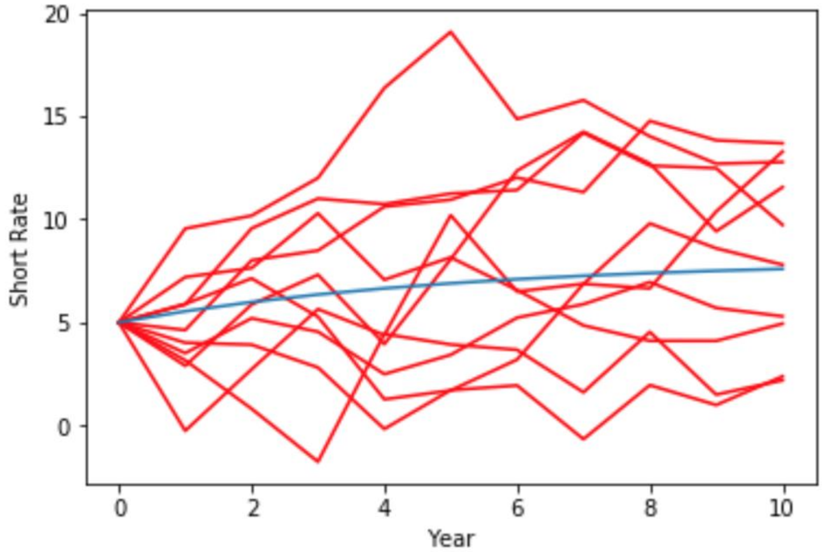


Figure 1.1: Short term rate simulated paths

Hull-White Model

The Hull-White model (Hull and White, 2001) specifies a stochastic differential equation for the short term rate r_t:

$$dr_t = (\theta(t) - \alpha(t)r_t)dt + \sigma(t)dW_t, r_0 = r(0), \tag{1.4}$$

where W_t is a standard Brownian motion , and $\theta(t)$, $\alpha(t)$, and $\sigma(t)$ can be time dependent.

The Vasicek model is thus a special case of the Hull-White model, where $\theta(t) = \alpha b$, $\alpha(t) = \alpha$, and $\sigma(t) = \sigma$. In practice, the Hull-White model is often implemented with only $\theta(t)$ being non-constant. By having a non-constant term, the mean reversion level is allowed to vary over time, and the model can be better calibrated to prices or rates seen in the market.

Cox-Ingersoll-Ross Model

The Cox-Ingersoll-Ross (CIR) model (Cox et al., 1985) also suggests a stochastic differential equation for short term rates:

$$dr_t \ = \ \alpha(b - r_t)dt + \sigma\sqrt{r_t}dW_t, r_0 \ = \ r(0) \tag{1.5}$$

where, as usual, W_t is a standard Brownian motion. Like in the Vasicek model, b is the mean level for the short term rate, and α is the rate of mean reversion. Unlike the Vasicek model, if $2\alpha b \geq \sigma^2$ and $r(0)$ is positive, the short term rate r_t will never become negative (known as the Feller condition). This can be understood intuitively, since, as r_t becomes close to 0, the volatility of the short term rate becomes close to 0. Then, the upward drift to the mean will exceed any downward movement brought about by the volatility. This is an advantage for the CIR over the Vasicek model. However, the CIR model does not have a closed-form solution, but we can still simulate it directly, since its conditional distribution is known.

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 0 is given by:

$$P = \mathbb{E}^{\mathbb{Q}}\left[e^{-\int_0^T r_t dt}\Phi(X_T)\right] \tag{1.6}$$

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 interest rate swaps.

Unit 1 : Video Transcript

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 \$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):

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

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)
```

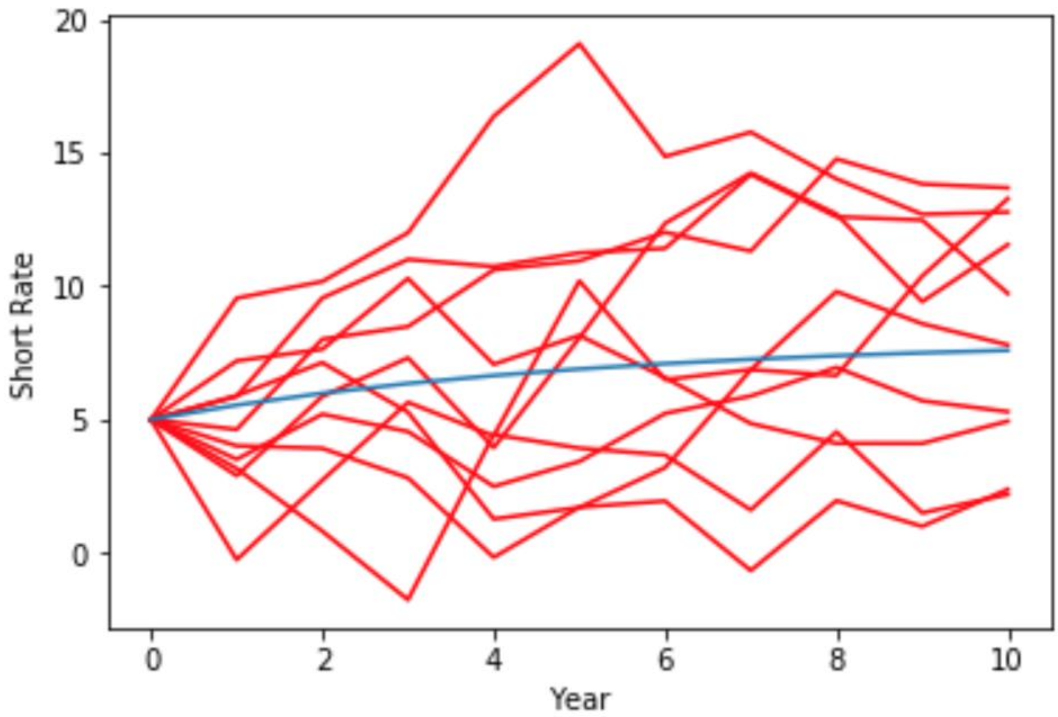
Moving onto the actual simulation, we first set our seed. Then we pre-allocate space for our simulations. We will treat each row as an individual simulation, so we will need to set the first column of our pre-allocated matrix to be equal to our r0 variable, which is the current interest rate. Thereafter, 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

Unit 2 : Notes

In the previous section, we introduced the notion of short rates and showed how one can simulate them. However, we usually don't simply want to simulate short term paths, but rather to use these paths to estimate values. For example, we may want to find zero-coupon bond (ZCB) prices – this is equivalent to finding the average discount factor. The price of a ZCB at time t with maturity T is given by:

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

(3.1)

where the expectation is taken under the risk-neutral measure. Recall that the Vasicek model defines short term rates according to the stochastic differential equation:

$$d\, r_t = \alpha(b - r_t)dt + \sigma d\, W_t, r_0 = r\,(0)$$

(3.2)

where W_t is a standard Brownian motion, α and , b, and σ are positive constants. Under this model, the expectation in equation (3.1) is given by:

$$B(t,T) = e^{-A(t,T)r_t + D(t,T)}$$

(3.3)

where $A(t,T) = \frac{1 - e^{-\alpha(T-t)}}{\alpha}$ and $D(t,T) = \left(b - \frac{\sigma^2}{2\alpha^2}\right)[A(t,T) - (T-t)] - \frac{\sigma^2 A(t,T)^2}{4\alpha}$. We thus have a closed-form solution for our bond prices in the Vasicek model.

We will be showing a few techniques to simulate short term rates and use these to estimate bond prices. We will compare the results of these techniques to the closed-form solution. One would be able to extend these techniques for pricing bonds when using a more complex model for modelling short term rates, such as the Hull-White or Cox-Ingersoll-Ross models.

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
```

We first import the relevant libraries and initialise the parameters. As in the previous section, we have set α= 0.2, b = 0.08, σ = 0.025, and r_0= 0.05.

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)
```

As before, we create functions which find the mean and variance of our short rate at time t_2 given the short rate at time $t_1 < t_2$.

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))
```

These functions find the closed-form solution for the bond price according to equation (3.3). The functions A and D calculate $A(t,T)$ and $D(t,T)$respectively, while bond price calculates $B(t,T)$ for a given initial short term rate.

In order to create estimates for our bond prices, we are going to be jointly simulating the short term rate, r_t, and a value for $\int_0^t r_s ds$, which we will call Yt. Since rt is Gaussian, and Yt is an integral of Gaussians, Y_t and r_t have a joint Gaussian distribution (Moffatt, 1997). We can thus simulate them together once we have a value for their means, variances, and the correlation between them.

We already know that $r_t \sim N\left(r(0)e^{-\alpha t} + b(1-e^{-\alpha t}), \frac{\sigma^2}{2\alpha}(1-e^{-2\alpha t})\right)$.

It can be shown that $Y_{t_2} \sim N\left(Y_{t_1} + (t_2-t_1)b + (r_{t_1}-b)A(t_1,t_2), \frac{\sigma^2}{\alpha^2}\left(t_2 - t_1 - A(t_1,t_2) - \alpha\frac{A(t_1,t_2)^2}{2}\right)\right)$,

where $t_2 > t_1$. Additionally, the covariance between Y_{t_2} and r_{t_2} given the filtration at t_1 is $\frac{\sigma^2 A(t_1,t_2)^2}{2}$, and the correlation would simply be the covariance divided by the standard deviations of r_{t_2} and Y_{t_2} . This will become clearer once we implement functions for this in Python:

```
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. We can thus simulate joint paths for r_t and Y_t by simulating standard normal random variables with this correlation, and then transforming them into normal variables with the appropriate means and variances.

In[]:
```
1  # Initial Y value
2  Y0 = 0
3
4  np.random.seed(0)
5
6  # Number of years simulated and number of simulations
7  n_years = 10
8  n_simulations = 100000
9
10 t = np.array(range(0,n_years+1))
11
12 Z_mont1 = norm.rvs(size = [n_simulations,n_years])
13 Z_mont2 = norm.rvs(size = [n_simulations,n_years])
14 r_simtemp = np.zeros([n_simulations, n_years+1])
15 Y_simtemp = np.zeros([n_simulations, n_years+1])
16
17
18 r_simtemp[:,0] = r0 #Sets the first column (the initial value of each simulation) to r(0)
19 Y_simtemp[:,0] = Y0
20
21 correlations = rY_rho(t[0:-1],t[1:])
22 Z_mont2 = correlations*Z_mont1 + np.sqrt(1-correlations**2)*Z_mont2
            #Creating correlated standard normals
23
24 for i in range(n_years):
25     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]
26     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]
27
28 ZCB_prices = np.mean(np.exp(-Y_simtemp),axis = 0)
```

We first note that $Y_0 = 0$ (since it is an integral from 0 to 0). We are going to simulate our paths for a period of 10 years, and create 100 000 simulations in total. Line 10 creates an array of the time points for which we are simulating values – note that these are integers and we are thus simulating values on an annual basis. We create standard normal random variable matrices Z_mont1 and Z_mont2, and matrices r_simtemp and Y_simtemp to store our simulated paths. We set the initial values for the paths in lines 18 and 19.

Note the following: if $Z_i \sim N(0,1)$ for $i = 1, 2$, then Z_1 and $\rho Z_1 + \sqrt{1-\rho^2}\, Z_2$ have a correlation of ρ. We thus transform Z_mont2 according to this in line 22, so that Z_mont1 and Z_mont2 have the correlations necessary to simulate r_t using Z_mont1 and Y_t using Z_mont2.

We are then able to simulate our paths. We do so recursively, by iterating through each year. Line 25 generates the r_t paths and line 26 generates the Y_t paths. We use the functions we created for the means and variances to transform the standard normal variables into normal variables with the appropriate means and variances.

Finally, we are able to estimate our bond prices using Monte Carlo, by taking the mean bond prices for each year over all the simulations. Recall that $\mathrm{B}(0,\mathrm{t}) = \mathbb{E}\left[e^{-\int_0^t r_s\,ds}\right] = \mathbb{E}[e^{-Y_t}]$, which is done in line 28.

An alternative to simulating the Y_t values is to estimate them using only the rt. This can be done as follows:

$$y_t = \int_0^t r_s\,ds \approx \sum_{i=0}^{n-1} r_{t_i}\delta_i \approx \sum_{i=0}^{n=1} \frac{r_{t_i} + r_{t_{i+1}}}{2}\delta_i \tag{3.4}$$

where $0 = t_0 < t_1 < t_2 < \cdots < t_n = t$, and $\delta_i = t_{i+1} - t_i$. This is an example of a quadrature, and we would expect the second approximation to be more accurate than the first, since the second has an $O(\delta^2)$ error versus an $O(\delta)$ error for the first. Simulating only the short term rates uses less computation time than simulating the integrals and short term rates.

In [ ]:
```
1  #Yt estimates
2  r_mat = np.cumsum(r_simtemp[:,0:-1],axis = 1)*(t [1:]-t[0:-1] )
3  r_mat2 = np.cumsum(r_simtemp[:,0:-1] + r_simtemp[:,1:], axis = 1)/2*(t[1:]-t[0:-1])
4  
5  #Bond price estimates
6  squad_prices = np.ones(n_years+1)#At time 0, bonds have a price of 1
7  trap_prices = np.ones(n_years+1)
8  
9  squad_prices[1:] = np.mean(np.exp(-r_mat), axis = 0)
10 trap_prices [1:] = np.mean(np.exp(-r_mat2), axis = 0)
```

Lines 2 and 3 find the Y_t approximations as per equation (3.4). Lines 6 and 7 find the means of the simulated bond prices for each Y_t estimate.

Finally, we use our earlier closed-form solution to find the true bond prices, and compare these to the three different estimates we have found for them.

```
In [ ]:
#Closed-form bond prices
bond_vec = bond_price(r0, 0, t)

#Plotting bond prices
plt.plot(t, bond_vec)#Analytical solution
plt.plot(t, ZCB_prices, '.') #Simulated Yt and rt
plt.plot(t, squad_prices, 'x') #Simulated rt and estimated Yt
plt.plot(t, trap_prices, '^') #Simulated rt and estimated Yt
plt.show ()
```

If you have coded this up correctly, you should replicate figure (3.1). As you can see, all three estimations give reasonably good approximations for the true bond prices.

The final extension we will add to this is finding the implied yield for the given bond prices. Note that the yield, y_t, is defined as:

$$B(0,t) = e^{(-y_t)t} \tag{3.5}$$

i.e. it is the 'constant interest rate' which gives the correct bond price. Solving this gives you $y_t = -\frac{\ln(B(0,t))}{t}$. We can find the yields for our bond prices using this formula:

```
In [ ]:
#Determining yields
bond_yield = -np.log(bond_vec[1:])/t[1:]
mont_yield = -np.log(ZCB_prices[1:])/t[1:]
squad_yield = -np.log(squad_prices[1:])/t[1:]
trap_yield = -np.log(trap_prices[1:])/t[1:]
```

Note that we do not include the first bond price, since the price is always 1, and a unique yield thus cannot be defined. We plot this yields over time:

```
In [ ]:
#Plotting the yields
plt.plot(t[1:], bond_yield*100)
plt.plot(t[1:], mont_yield*100,'.')
plt.plot (t[1:], squad_yield*100, 'x')
plt.plot(t[1:], trap_yield*100, '^')
plt.show ()
```

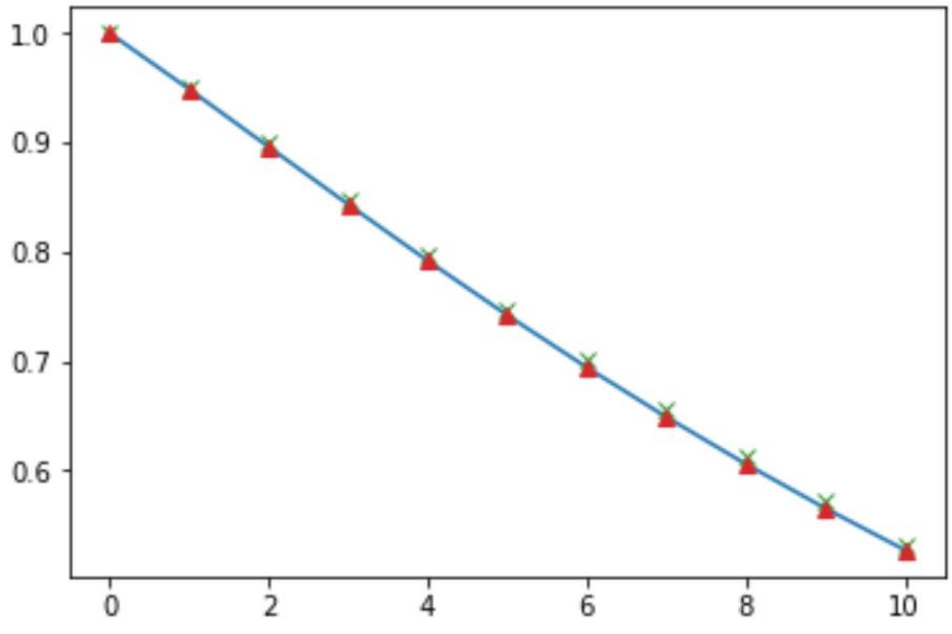


Figure 3.1: Bond prices

This gives figure (3.2). Note that we have scaled the yields by a factor of 100. As you can see, most of the estimates are still fairly good, with the exception of squad_yield. This is the estimate where Y_t was estimated using $\sum_{i=0}^{n=1} r_{t_i}\delta_i$. As a result, it would be more reliable to use the other estimation techniques which were implemented. In general, discretization error arises when using quadrature, which is why we observe greater errors than the joint simulation, which only has error as a result of finite simulations.

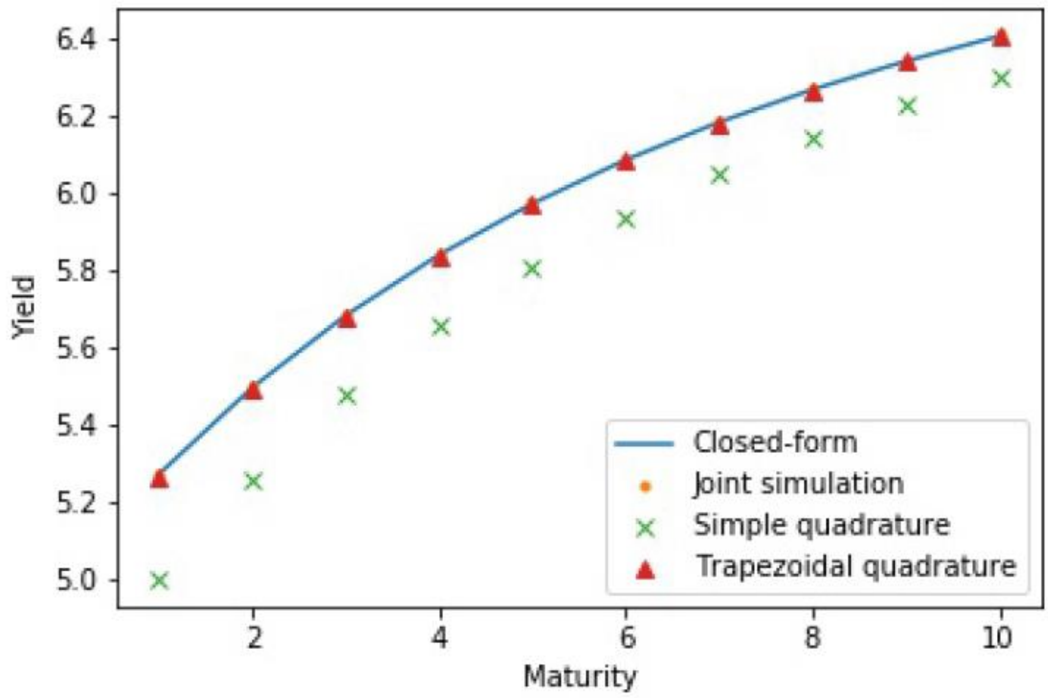


Figure 3.2: Bond yield

Unit 2: Video Transcript

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 initialise 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

Unit 3: Notes

Introduction

The LIBOR Forward Market Model (LFMM) is seen as the benchmark interest rate model, in a similar way the Black-Scholes model for equity option pricing. A market model differs from the models that we have generally presented in that it attempts to model financial instruments which are actually traded in the market, instead of focusing on an idealisations of a financial variable (such as the short rate) which drives the market. The LFMM models simple forward rates in a market. 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.

Because of the number of forward rates that exist in financial markets, and because the LFMM tries to model each and every one of these, the LFMM becomes very complicated very quickly. As a result, we will be focusing on the most simple case, in order to illustrate how you would go about using this model. Just keep in mind that there are many ways to make the model more complex (including, but not limited to, introducing stochastic volatility).

What you need to know before applying the LFMM

In the previous sections of this model, we focused on how to model con-tinuous rates. However, continuous rates aren't directly observable in the market. Thus, because the LFMM is a market model, it consistently models simple rates (which are directly observable in markets).

A simple rate is just an interest rate where there is no compounding on the interest earned; the interest generated is only based on the initial investment. This means that, if you invest $1 today, you will receive (1+iT) at the end of T years, where i is the interest rate.

As the name suggests, the LFMM models the simple forward rates in the market. A forward rate is an interest rate relevant to a future period that you can agree on today. For example, if a bank agrees to let me save money with them at 10% in one year's time (if I am willing to sign a contract which says that I will save my money with them at that time), then the 1 year forward rate is 10%.

Some mathematics underlying the LFMM

As has been noted, we will not be focusing on the rigorous mathematics underlying the LFMM. There are several very good textbooks which cover the LFMM in a large amount of depth. If you would like to read further, you can try looking at Chapter 6 in Brigo and Mercurio (2007) which goes into considerable depth when exploring the LFMM.

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, $\{T_0, T_0, \ldots, T_N\}$. Let $P_j(t)$ be the price of a bond at time t which expires at time T_j, $F(t, T_j, T_{j+1}1) = F_j(t)$ 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 associ-ated with Fj(t), and Wt is a Brownian motion. Now, we could assume that each forward rate has its own Brownian motion. This assumption would imply that there is a source of noise, or

randomness, for every forward rate in the market. However, for simplicity's sake, we are going to assume that there is only one source of noise in the market (i.e. there is only one Brownian motion).

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 $\tau(t) = \min\{i : t < T_i\}$. This $\tau(t)$ function is important because of how the forward rates evolve over time. Note that forward rates aren't always necessarily useful. For example, if we are at year 2, then the forward rates that only applied between time 0 and year 2 no longer need to be modelled because they are known. This means that the only forward rates that continue to add to the overall uncertainty are the ones that apply after year 2. This is the reason why we include the $\tau(t)$ function.

Simulating the LFMM

UNDERLYING THE MATHEMATICAL APPROXIMATIONS.

A BASIC APPROXIMATION

It can be shown that:

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

where $\hat{\mathrm{F}}_j(t_i)$ is the approximation for the j^{th} forward rate at time t_i, $Z_i \sim N(0,1)$, and:

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

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:

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

IMPROVING THE ACCURACY OF APPROXIMATIONS

A method for improving the overall accuracy of the forward rate approximations was proposed by Hunter *et al.* (2001). This method, known as the Predictor-Corrector method, revolves around estimating an initial and a terminal drift, then using an average of these to project future forward rates.

Once again, we initialize $\hat{F}_j(t_0) = F_j\,(t_0)$ in the same way above. Now, we first estimate the forward rate for the next price period using:

$$\tilde{F}_j(t_i) = \bar{F}_j(t_{i-1})\,\exp\left[\left(\mu_j^1(t_i - 1) - \frac{1}{2}\sigma^2\right)\delta_{i-1} + \sigma_j\sqrt{\partial_i - 1}\,Z_i\right] \tag{5.4}$$

where:

$$\mu_j^1(t) = \sum_{k=i}^{j} \frac{\delta_k \bar{F}_j(t_{i-1})\sigma_k\sigma_j}{1 + \delta_k \bar{F}_j(t_{i-1})} \tag{5.5}$$

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

$$\mu_j^2(t_{i-1}) = \sum_{k=i}^{j} \frac{\delta_k \tilde{F}_j(t_i)\sigma_k\sigma_j}{1 + \delta_k \tilde{F}_j(t_i)} \tag{5.6}$$

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

$$\bar{F}_j(t_i) = \bar{F}_j(t_{i-1})\exp\left[\frac{1}{2}\left(\mu_j^1(t_{i-1}) + \mu_j^1(t_i) - \sigma^2\right)\delta_{i-1} + \sigma_j\sqrt{\delta_{i-1}}Z_i\right] \tag{5.7}$$

Note that the Z_i value used for the preliminary and the final forward estimate is the same.

Simulating the LFMM in Python

We will be applying both methods presented in the previous sub-section simultaneously. The aim here will be to simulate bond prices using the LFMM. We can do this by simulating forward rates, using these to imply capitalisation factors and then inverting these to imply discount factors (which should be equal to the bond price under no arbitrage). Note that when taking averages to get to our Monte Carlo estimate, we can only take averages when we have all our simulations for the bond prices and note before.

The last note that we would like to make before getting into the simulation is that this is a computationally expensive simulation. As a result, we will be vectorising as much as possible to ensure that the final code is reasonably efficient.

The code below gives the parameter values and the libraries necessary to do Monte Carlo simulations using the LFMM. The first set of parameters are the same as in the Vasicek section. Thereafter, the t variable is the matu-rity times for the bonds (40 years is the longest term, and we dealing with bond maturities that are 2 years apart). The sigmaj variable is the volatility for the LFMM model. We are going to assume that the volatilities for each forward rate are constant and equal to 20%.

```
In [ ]:
#Libraries
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

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

Usually you would use market prices of bonds for different tenors. However, for our sake, we will generate our own bond prices using Vasicek dynamics. The next portion of code generates synthetic bond prices that we can use for our simulations.

In [ ]:

```
# 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)
```

Now we can get to the actual simulation. The next portion of code sets the seed and creates a few new parameters. The n simulations variable is the number of Monte Carlo simulations we are going to do at each time point. The n steps variable stores the number of time steps we are simu-lating over. Note that this is just the number of time increments we have, which is stored in the variable t. Finally, we pre-allocate space for our for-ward rates under both methods, as well as the final capitalisation factors. Note that, because we are only taking averages once we have simulations for the bond prices, we need to pre-allocate entire matrices for our forward rates, and not vectors, so that we keep all of the simulated forward rates as we go.

We also initialise our forward rates in this step (which is why we are multiplying a matrix of ones by the forward rates at time t0 implied by the bond prices. Note that each row of the matrix constitutes one sample path, so the code in lines 6 and 7 initialises each row to the implied forward rates.

The delta variable stores the time increments between the bond maturities.

We create this as a matrix so that we can use this to vectorise our code.

In [ ]:
```
# Applying the algorithms
np.random.seed(0)
n_simulations = 100000
n_steps = len(t)

mc_forward = np.ones([n_simulations,n_steps-1])*(vasi_bond[:-1]-vasi_bond[1:])/(2*vasi_bond[1:])
predcorr_forward = np.ones([n_simulations,n_steps-1])*(vasi_bond[:-1]-vasi_bond[1:])/(2*vasi_bond[1:])
predcorr_capfac = np.ones([n_simulations,n_steps])
mc_capfac = np.ones([n_simulations,n_steps])

delta = np.ones([n_simulations,n_steps-1])*(t[1:]-t[:-1])
```

Now we simulate our forward rates. Because the next period's forward rate is dependent on the previous period's forward rate, we will need to make use of a for loop (i.e. we can't vectorise everything). We apply the basic Monte Carlo simulation in lines 5 and 6. Line 5 refers to equation (6.2), and line 6 refers to equation (6.1). We apply the predictor-corrector method in lines 9 to 12. Line 9 refers to equation (6.5), line 10 refers to equation (6.3),

line 11 refers to equation (6.6), and line 12 refers to equation 6.7). In line 5 (and several times thereafter), we use the np.cumsum function. The np.cumsum function returns the cumulative sum of the elements in a matrix. For example, if we have a vector, then the second element of the returned vector is the sum of the first and second element in the original vector. The axis = 1 command results in the np.cumprod function returning the cumulative product of each row in the matrix.

```
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)
```

Now that we have simulated forward rates, the next step is to use this to imply capitalisation factors. This can be done using the following equation:

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

Where C(t, T) is the capitalisation factor for the period between t0 and tn. The np.cumprod function returns the cumulative product of the elements in a matrix. For example, if we have a vector, then the second element of the returned vector is the product of the first and second element in the original vector. The axis = 1 command results in the np.cumprod function returning the cumulative product of each row in the matrix. We then take the inverse of the capitalisation factors to get the bond prices (which are the same as the discount factors). Finally, we take the mean, and we are done.

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)
```

We can now plot our results using the following code:

```
In [ ]:
plt.xlabel("Maturity")
plt.ylabel("Bond Price")
plt.plot(t,vasi_bond, label = "Vasicek Bond Prices")
plt.plot(t,mc_final,'o', label = "Simple Monte Carlo Bond Prices")
plt.plot(t,predcorr_final,'x', label = "Predictor-Corrector Bond Prices")
plt.legend()
plt.show()
```

If you applied everything correctly, you should get the following two graphs. Figure (6.1) is the original bond prices, and the two simulated bond prices. Figure (5.2) is the forward rates at time t_0 implied by the bond prices.

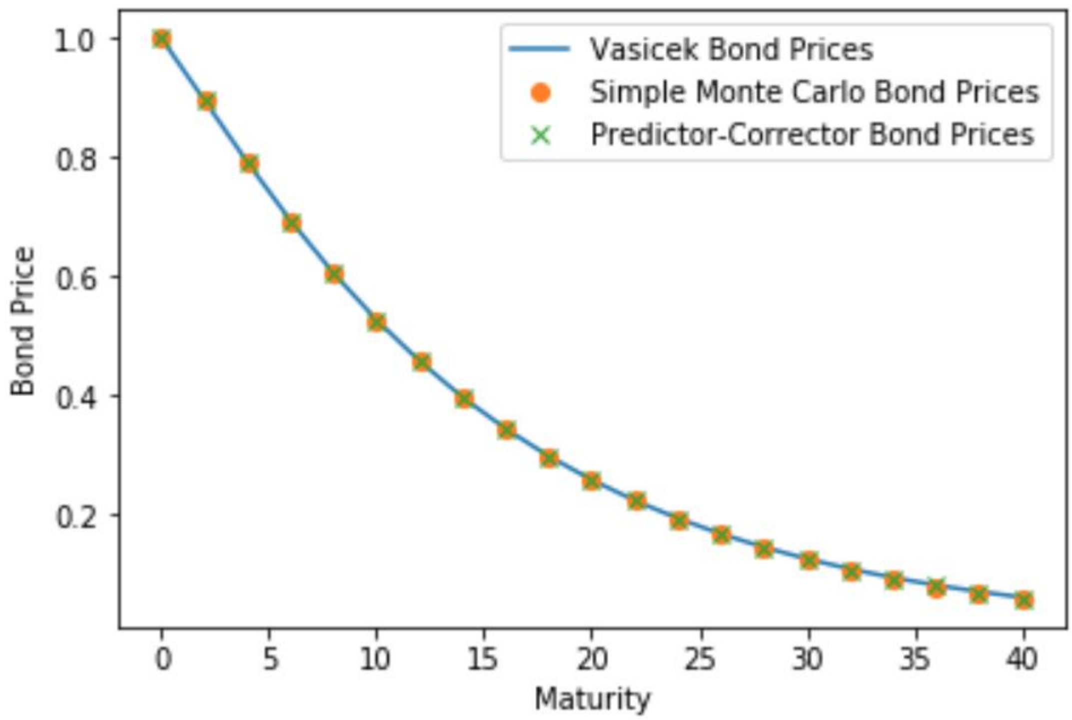


Figure 5.1: Bond Prices

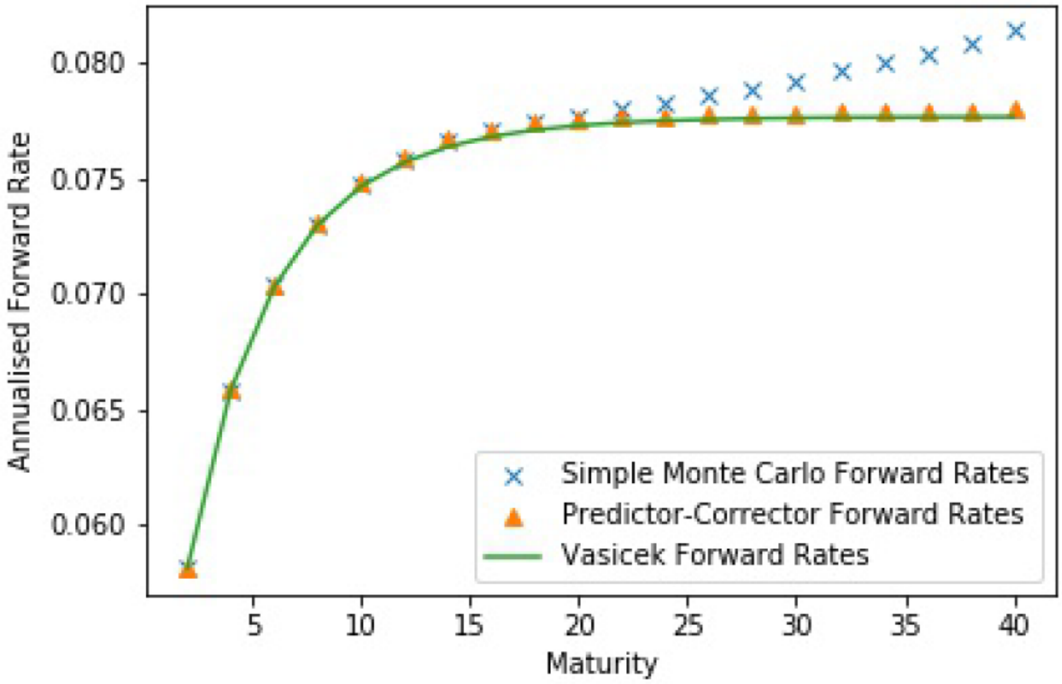


Figure 5.2: Forward rates

Unit 3 : Video Transcript

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 idealised 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 for-ward 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 initialising 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:

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

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:

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

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 set-ting our initial parameters. The first group of parameters is as per the Va-sicek model. The second group is for our LFMM. Note that we are assum-ing 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 [ ]:
```
#Libraries
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

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

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

In [ ]:
```
# 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 vectorise 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 capitalisation 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 capitalisation from forward rates formula in line 2, trans-form these factors to prices in line 5, and take the average prices over our simulations in line 8.

```
In [ ]:
1  # Implying capitalisation factors from the forward rates
2  mc_capfac[:,1:] = np.cumprod(1+delta*mc_forward, axis = 1)
3  predcorr_capfac[:,1:] = np.cumprod(1+delta*predcorr_forward, axis = 1)
4
5  # Inverting the capitalisation factors to imply bond prices (discount factors)
6  mc_price = mc_capfac**(-1)
7  predcorr_price = predcorr_capfac**(-1)
8
9  # Taking averages
10 mc_final = np.mean(mc_price,axis = 0)
11 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.

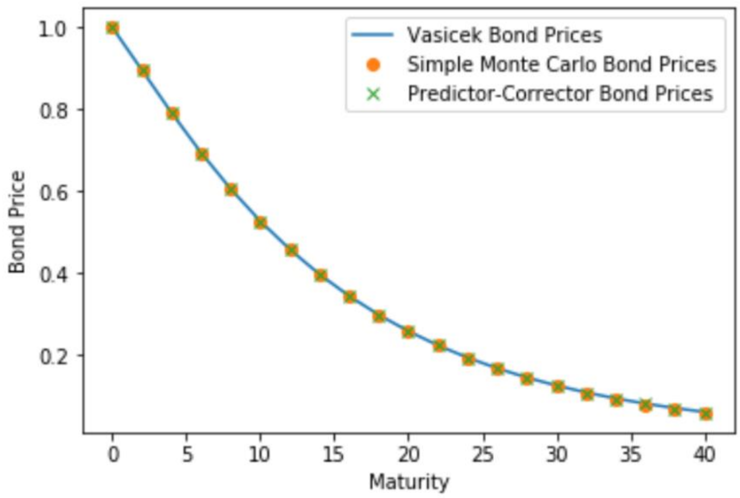


Figure 6.1: Bond Prices

Bibliography

Brigo, D. and Mercurio, F. (2007). Interest rate models-theory and practice: With smile, inflation and credit, Springer Science & Business Media.

Cox, J. C., Ingersoll Jr, J. E. and Ross, S. A. (1985). An intertemporal general equilibrium model of asset prices, Econometrica: *Journal of the Econometric Society* pp. 363–384.

Hull, J. and White, A. (2001). The general hull-white model and supercalibration*, Financial Analysts Journal* pp. 34–43.

Hunter, C. J., Jäckel, P. and Joshi, M. S. (2001). Drift approximations in a forward-rate-based libor market model, *Getting the Drift* pp. 81–84.

Mamon, R. S. (2004). Three ways to solve for bond prices in the vasicek model, *Advances in Decision Sciences* 8(1): 1–14.

Moffatt, H. (1997). *Mathematics of Derivative Securities*, Vol. 15, Cambridge University Press.

Vasicek, O. (1977). An equilibrium characterization of the term structure, *Journal of financial economics* 5(2): 177–188.