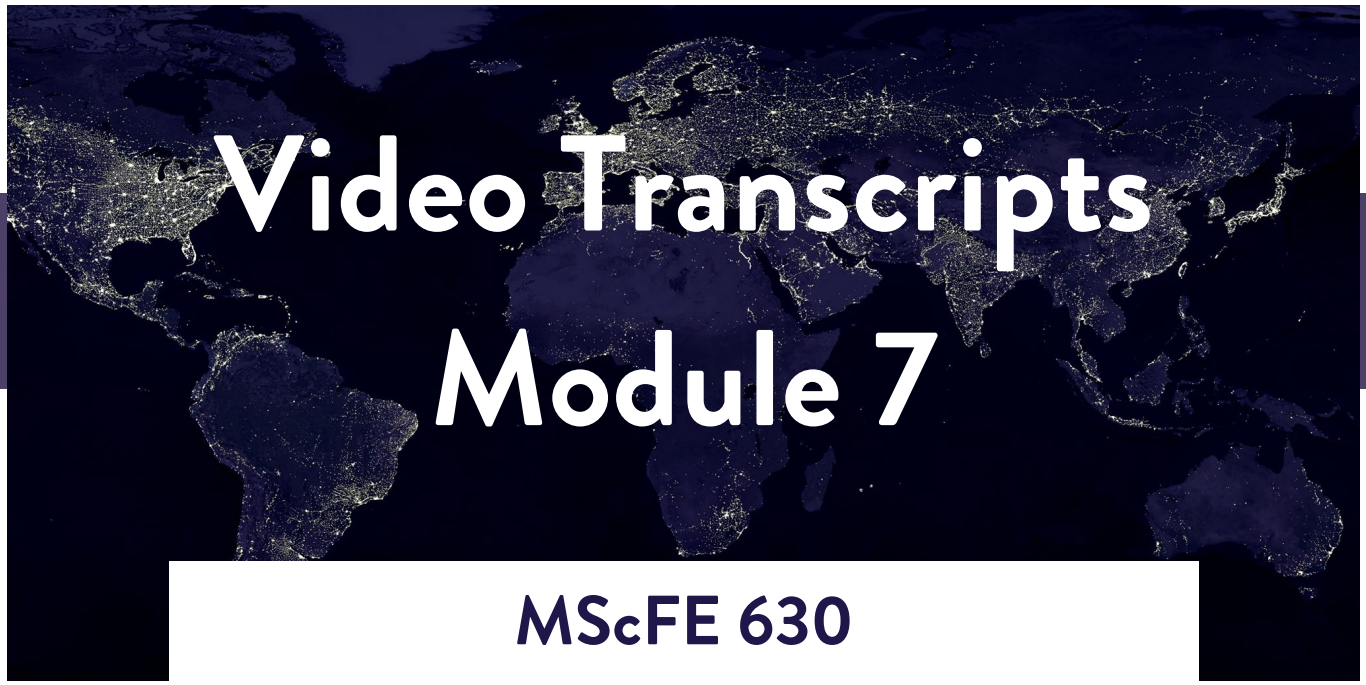
WORLDQUANT UNIVERSITY



Video Transcripts Module 7

MScFE 630

Computational Finance

Revised: 09/07/2020

Table of Contents

Unit 1: Model Calibration

Hello and welcome to the first video of Module 7. In this video, we will define what calibration is, and spend time going through a very simple, non-numerical example of calibration.

We have spent a considerable amount of time going through different models that we can use to model different asset classes. When presenting these models, we have always taken the values of each model's parameters as being given. We will now go through how to use market data to find these parameter values. The process of finding a set of parameter values based on some dataset is known as **calibration**. This is different from **estimation**, which uses a time series of data, instead of the time slice used by calibration.

Before we look at how you can calibrate a model, though, let's set the scene. Suppose that we want to price some exotic option whose price can't be directly observed in the market. We can price this option using some financial model. However, before we can use this model to price the option, we must first calibrate this model. As a result, calibration is an incredibly important part of the modeling process as it directly affects the quality of the results produced by any of our models.

Calibration is ultimately about finding agreement between the various theoretical aspects of quantitative work, such as modeling, and the practical aspects, like prices observed in the market. Consider the process of delta-hedging, whereby a holding equal to the negative of the delta of a derivative in the underlying is held in conjunction with the derivative. If done correctly, this leads to the portfolio as a whole being immunized to small changes in the underlying's price. In order to implement this in practice, one needs to have a measurement of the underlying's delta. This delta will usually be found using the model used for the underlying. As a result, an uncalibrated or poorly calibrated model will give an incorrect delta value, and so the portfolio will not be correctly delta-hedged. If the portfolio is under-hedged, you would not be protected fully against changes in the underlying's value. If the portfolio is over-hedged, you will tie up unnecessary capital in your hedging portfolio. With this in mind, any instruments which you are using for hedging needs to be calibrated correctly to the market, as well as the instruments which you are attempting to hedge.

Let's go through a simple example of calibration so that we can build some intuition regarding how calibration works.

In this example, we are going to be calibrating the volatility term in the Black-Scholes framework. Suppose that we have access to market data concerning the price, strike, and maturity of a put option. Note that put option prices are generally quoted in terms of the volatility that would be used in Black-Scholes pricing formula, which means that the market would be directly quoting the volatility parameter that you are trying to calibrate. For now, let's just assume that the market is directly quoting the Black-Scholes price of the option.

The first step is to recognize that, in the Black-Scholes framework, the price of a put option is given by:

$$P = \Phi(-d_2)Ke^{-rT} - \Phi(-d_1)S_0$$

where $\Phi(\bullet)$ is the cumulative distribution function of a standard normal random variable, and,

$$d_1 = \frac{1}{\sigma\sqrt{T}}\left(\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T\right)$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

What we do now is guess values for the volatility, substitute these into the Black-Scholes pricing formula, and see which result in a price closest to the market price. The reason that we have to guess values is that it is not possible to invert the put price formula above to solve for the volatility term. Thankfully, Python has access to various functions which can help us to find the volatility value, which let's us get a Black-Scholes price which is nearest to the market price. Very simply, these functions work off of the following premise: they start with an initial guess and check how close the guess gets you to the market price. Then, they change the guess slightly and try again. If the new value results in a closer approximation, then this value is kept, and the process is started again. Otherwise, the algorithm will change the initial guess and start again. This continues until the difference between the market price and the Black-Scholes price falls below some tolerance level.

We have spent some time talking about the general way to calibrate a model. The notes on this unit go through a more specific example of calibration, and present Python code that performs a simple calibration. Note that the notes use a Python minimization function. This means that you would need to set up a function which takes in the volatility term as a parameter and returns the absolute difference between the market price and the Black-Scholes price. Returning the absolute

difference is important; otherwise, the minimization function will try and find the volatility which makes the difference between the market price and the Black-Scholes price a large negative.

In the next video, we will go through how to calibrate interest rate models, and how to perform calibration over several parameters at once.

Unit 2: Error Analysis

Whenever we do any sort of modeling, we are often making a sacrifice: either our model is extremely accurate, but harder to use, or it is very tractable, but inaccurate. In financial modeling, these two ideas, ease-of-use and accuracy, are often at odds with each other. No model we implement is able to capture perfectly the various instruments which we would be modeling with it. As George Box said, "All models are wrong, but some are useful". In this video, we will address two of the primary reasons why our models are often wrong – known as **model error** and **data error**.

Model error

Model error is the name given to the discrepancy that arises as a result of the model itself not being able to capture a financial instrument's characteristic. Suppose for example that we wanted to model the short rate in a market, whose true dynamics are of the form:

$$dr(t) = \mu\big(t, r(t)\big)dt + \sigma\big(t, r(t)\big)dW_t$$

where $\mu\big(t, r(t)\big)$ and $\sigma\big(t, r(t)\big)$ are not necessarily deterministic functions, and can be stochastic as well, and W_t is a multidimensional Brownian motion. We do not know what form these coefficients μ and σ take, and so we make a simplifying assumption.

So we could assume that they take the form given by the Vasicek model – namely that $\hat{\mu}\big(t, r(t)\big) = \alpha\big(\beta - r(t)\big)$, with β being the mean-reversion level and α being the rate of mean reversion, and $\hat{\sigma}\big(t, r(t)\big) =$ sigma (a constant). In this case, our Brownian motion will also be one-dimensional, instead of multidimensional. So, we are taking potentially stochastic parameters, and turning them into deterministic functions of a very particular form.

Depending on how much of a stretch these assumptions are relative to the true underlying processes, there could be significant differences between the model and reality. Making the model more complex might reduce this error but could come at the cost of parameter interpretation and ease of calculation.

Data error

Another common source of error is the issue of the data being used to either build or calibrate the model. Suppose that the data you are using to decide on the form of the model which you are going to use is far out-of-date. You might observe trends in the data which are no longer observed in practice. This could result in you choosing a model which can capture these trends, but it may be a poor choice given that trends have changed. Another issue could be the data itself being inaccurate, due to things like inaccurate recording or reporting. These are all things you need to be cognizant of when building models. The more developed the market from which you are getting your data, the less of an issue these things should be.

The problem with Black-Scholes

The reason that the Black-Scholes model is as easy to use as it is, is because it makes a number of strong assumptions – for example, frictionless trading and a constant risk-free interest rate. Most of these assumptions are not seen in practice, which is why it is actually impossible to calibrate the Black-Scholes model well to market prices. The mere existence of the volatility smile is an indication of this fact. However, it still provides us with a strong baseline for pricing and hedging exercises, and it has had continued importance in the financial sector. That being said, there are other, more complex models which are able to perform better, particularly in terms of calibration.

In the next two units, we will be going through a calibration for the Vasicek and the Heston model. These will be far more involved than the calibration covered in the previous unit, so make sure you're comfortable with the ideas discussed there before moving on.

Unit 3: Interest Rate Model Calibration

In the previous video, we went through what calibration is in a relatively general sense. In this video, we will go through a specific example of calibration. In this example, we will also be calibrating several parameters at once.

In Module 6, we went through the Vasicek model. When presenting that model, we used some pre-defined parameter values. Let's assume now that we have some market data, and that we want to find the parameter values which best allows the Vasicek model to reproduce what we observe in the market.

Before we move on, it would be useful to go over what the Vasicek model is in the context of what can be observed in the market. The Vasicek model models the short rate. The short rate is a continuous instantaneous rate. This means that, it is the interest rate you would get if you saved some money with a bank for a very small increment of time. This rate isn't directly observable in markets, however. In fact, the smallest time increment that can be observed is generally a day. As a result, we will need to calibrate the Vasicek model using the bond pricing formula that we derived in Module 6.

Remember that the dynamics of the short rate under the Vasicek model is given as:

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

where W_t is a standard Brownian motion, and α, b, and σ are positive constants.

Using these dynamics, we were able to arrive at the following formula for bond prices:

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

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 can now look at how we can calibrate the Vasicek model using Python.

As per usual, the first step is to import the relevant libraries. Note that we now import an additional library: `scipy.optimize`. This has various optimization functions which are essential for the calibration process.

In [6]:
```
#Importing libraries
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
import scipy.optimize
```

The next step is to create the data set which we will be using for our calibration. Note that you would use market data at this point. We are going to assume that zero-coupon bond (ZCB) prices have the following parametric form:

$$B(0,t) = e^{-\frac{1+(1+6t)log(1+t)}{40}}$$

where t is the maturity time of the bond. The code to create the ZCB data is:

In [2]:
```
years = np.linspace(1,10,10)
bond_prices = np.exp(-((1+6*years)*np.log(1+years))/40)
```

Note that usually, rather than a function for bond prices, you would have ZCB market prices for different maturities, to which you would calibrate (this is what you will do in the final part of the group work assignment). We then create the functions that make up the bond prices under the Vasicek model:

In [3]:
```
#Analytical bond price Vasicek
def A(t1,t2,alpha):
    return (1-np.exp(-alpha*(t2-t1)))/alpha

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

def bond_price_fun(r,t,T,alpha,b,sigma):
    return np.exp(-A(t,T,alpha)*r+D(t,T,alpha,b,sigma))

r0 = 0.05
def F(x):
    alpha = x[0]
    b = x[1]
    sigma = x[2]
    #return sum((bond_price_fun(r0,0,years,alpha,b,sigma)-bond_prices)**2)
    return sum(np.abs(bond_price_fun(r0,0,years,alpha,b,sigma)-bond_prices))
```

Note that F is the function that we are optimizing. This returns the absolute value of the sum of the differences between the model prices and the market prices. This means that we are essentially performing a least-absolute difference minimization.

Now, note that the Vasicek model assumes that the parameter values are positive.

We will assume that our parameter values satisfy the following system of inequalities:

$$0 \leq \alpha \leq 1$$

$$0 \leq b \leq 0.5$$

$$0 \leq \sigma \leq 0.2$$

Note that the lower bounds are all required by the model. We set the upper bounds using some form of rationality check. Because b is the long-term equilibrium rate, we would need to imply a yield curve from the ZCB prices, which is done in the notes in this unit. However, if you do it for the data that we have created, you would see that the largest yield is approximately 36.5%. So, to be safe, we set an upper bound for the long-term equilibrium to be 50%.

We can then set our bounds, and perform the calibration using the following code:

In [4]:
```
# Minimizing F
bnds = ((0,2),(0,0.5),(0,0.2))
opt_val = scipy.optimize.fmin_slsqp(F, (0.3,0.05,0.03), bounds=bnds)
opt_alpha = opt_val[0]
opt_b = opt_val[1]
opt_sig = opt_val[2]
```

We finally plot the implied ZCB curve against the market ZCB curve to show the quality of the fit:

In [5]:
```
# Calculating model prices and yield
model_prices = bond_price_fun(r0,0,years,opt_alpha,opt_b,opt_sig)

plt.xlabel("Maturity")
plt.ylabel("Bond price")
plt.plot(years,bond_prices)
plt.plot(years,model_prices,'x')
```

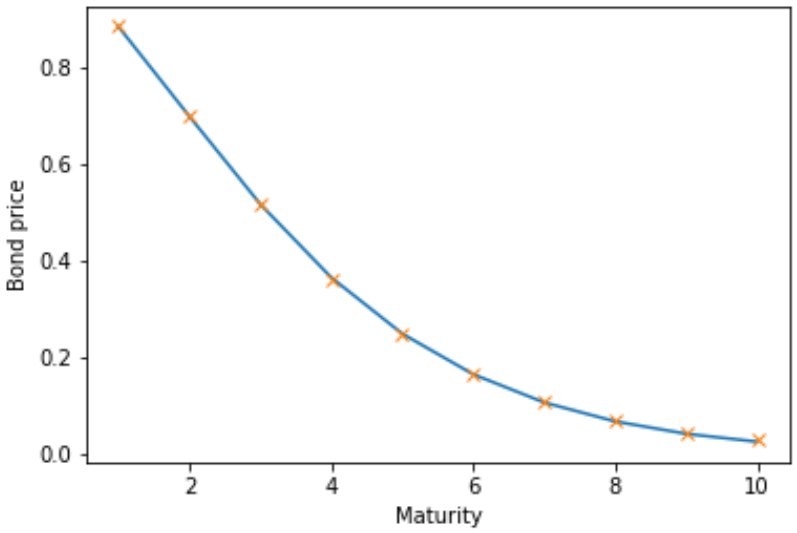


Figure 1: Market ZCB prices vs Calibrated MCB prices

That brings us to the end of video 3. In the final video of this module, we will look at characteristic function calibration.

Unit 4: Characteristic Function Calibration

In this unit, we are going to introduce the use of characteristic function based techniques, which we covered in module 4, for calibration.

But why would we use characteristic function techniques for calibration? There are many models which result in a closed-form characteristic functions, but do not have a closed-form solution for even vanilla call option prices. When we are calibrating our models using market data, what we are essentially doing is guessing values for whatever parameter values that we are trying to estimate, substituting this into the pricing formula, and seeing which guesses get our model prices closest to the market prices. Thus, by applying these characteristic function methods, we can determine vanilla option prices for a wide range of models, which allows us to calibrate a wider range of models.

Note that it is certainly possible that we may be running this process for thousands of guesses of parameter values, or even more. As a result, we want each evaluation to be as fast as possible, which is where Fourier techniques come in. Fourier pricing techniques are generally more efficient than the Gil-Pelaez approach. Thus, we could use these Fourier techniques to ensure an efficient calibration, whilst still making use of characteristic function pricing techniques.

In order to illustrate the usefulness of characteristic-function-based calibration methods, we will look at how we can go about calibrating the Heston model. Note that the Heston model does not have a closed-form solution to vanilla option prices but does have a closed-form characteristic function. As a result, the previous calibration methods would not be of use to calibrate the Heston model. Given that we have a closed-form characteristic function, we can apply the Gil-Pelaez theorem to determine the price of a vanilla call option.

Recall that the dynamics of an asset under the Heston model are given by:

$$dS_t = rS_t dt + \sqrt{v_t} S_t dW_t^1$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} dW_t^2$$

where dW_t^2 and dW_t^2 have correlation ρ.

In addition, the price of a vanilla call option is given by:

$$c = S_0 Q^S[S_T > K] - e^{-rT} K Q[S_T > K]$$

$$= S_0 \left(\frac{1}{2} + \frac{1}{\pi} \int_0^\infty \frac{Im\left[e^{-it \ln K} \varphi_{M_2}(t)\right]}{t} dt \right)$$

$$-e^{-rT} K \left(\frac{1}{2} + \frac{1}{\pi} \int_0^\infty \frac{Im\left[e^{-it \ln K} \varphi_{M_1}(t)\right]}{t} dt \right)$$

where:

$$\varphi_{M_1}(t) = \phi_{s_T}(t)$$

$$\varphi_{M_2}(t) = \frac{\phi_{s_T}(u - i)}{\phi_{s_T}(-i)}$$

and $s_t = \log(S_t)$ is the log-share price.

Now suppose that we wanted to calibrate the κ, θ, and σ parameters.

Let's look at the Python code to do this:

In [1]:

```
import numpy as np
from scipy.stats import norm
import scipy.optimize as opt
import matplotlib.pyplot as plt

r = 0.06
S0 = 100
v0 = 0.06
K = np.array([110,100,90])
price = [8.02,12.63,18.72]
T = 1
k_log = np.log(K)
k_log.shape = (3,1)
rho = -0.4

# Parameters for Gil-Paelez
t_max = 30
N = 100
```

We first set up our parameters and import the relevant libraries. Note that we have three market prices for 3 strike values all with the same maturity of one year.

```
#Characteristic function code
def a(sigma):
    return sigma**2/2

def b(u,theta,kappa,sigma):
    return kappa - rho*sigma*1j*u

def c(u,theta,kappa,sigma):
    return -(u**2+1j*u)/2

def d(u,theta,kappa,sigma):
    return np.sqrt(b(u,theta,kappa,sigma)**2-4*a(sigma)*c(u,theta,kappa,sigma))

def xminus(u,theta,kappa,sigma):
    return (b(u,theta,kappa,sigma)-d(u,theta,kappa,sigma))/(2*a(sigma))

def xplus(u,theta,kappa,sigma):
    return (b(u,theta,kappa,sigma)+d(u,theta,kappa,sigma))/(2*a(sigma))

def g(u,theta,kappa,sigma):
    return xminus(u,theta,kappa,sigma)/xplus(u,theta,kappa,sigma)

def C(u,theta,kappa,sigma):
    val1 = T*xminus(u,theta,kappa,sigma)-np.log((1-g(u,theta,kappa,sigma)*np.exp(-T*d(u,theta,kappa,sigma)))/
                                                (1-g(u,theta,kappa,sigma)))/a(sigma)
    return r*T*1j*u + theta*kappa*val1

def D(u,theta,kappa,sigma):
    val1 = 1-np.exp(-T*d(u,theta,kappa,sigma))
    val2 = 1-g(u,theta,kappa,sigma)*np.exp(-T*d(u,theta,kappa,sigma))
    return (val1/val2)*xminus(u,theta,kappa,sigma)

def log_char(u,theta,kappa,sigma):
    return np.exp(C(u,theta,kappa,sigma) + D(u,theta,kappa,sigma)*v0 + 1j*u*np.log(S0))

def adj_char(u,theta,kappa,sigma):
    return log_char(u-1j,theta,kappa,sigma)/log_char(-1j,theta,kappa,sigma)
```

Here, we have defined all of our functions relevant to the Heston characteristic function. Take note that these functions take in the parameters that we are trying to calibrate as inputs, in addition to the variable, u, which they would take as an input normally.

In [4]:
```
delta_t = t_max/N
from_1_to_N = np.linspace(1,N,N)
t_n = (from_1_to_N-1/2)*delta_t
```

We then define a few extra variables which will allow us to efficiently evaluate the call price.

In [5]:

```python
# Calibration functions

def Hest_Pricer(x):
    theta = x[0]
    kappa = x[1]
    sigma = x[2]
    first_integral = np.sum((((np.exp(-1j*t_n*k_log)*adj_char(t_n,theta,kappa,sigma)).imag)/t_n)*delta_t,axis = 1)
    second_integral = np.sum((((np.exp(-1j*t_n*k_log)*log_char(t_n,theta,kappa,sigma)).imag)/t_n)*delta_t,axis = 1)
    fourier_call_val = S0*(1/2 + first_integral/np.pi)-np.exp(-r*T)*K*(1/2 + second_integral/np.pi)
    return fourier_call_val

def opt_func(x):
    return sum(np.abs(price - Hest_Pricer(x)))
```

We define a function, `Hest_Pricer`, which returns the call price for the given parameter values and an additional function, `opt_func`, which we are going to be using for calibration. This function calculates the sum of the absolute difference between our model call prices for a given set of parameter values and the observed market prices. We want to find the parameter values which gets this function as close to 0 as possible.

We have now calibrated a model which doesn't have a closed-form vanilla call price. The notes go into some more detail on using the fast Fourier transform, which is more efficient than the method we went through here.