WORLDQUANT UNIVERSITY

Compiled Content Module 7

MScFE 630 Computational Finance

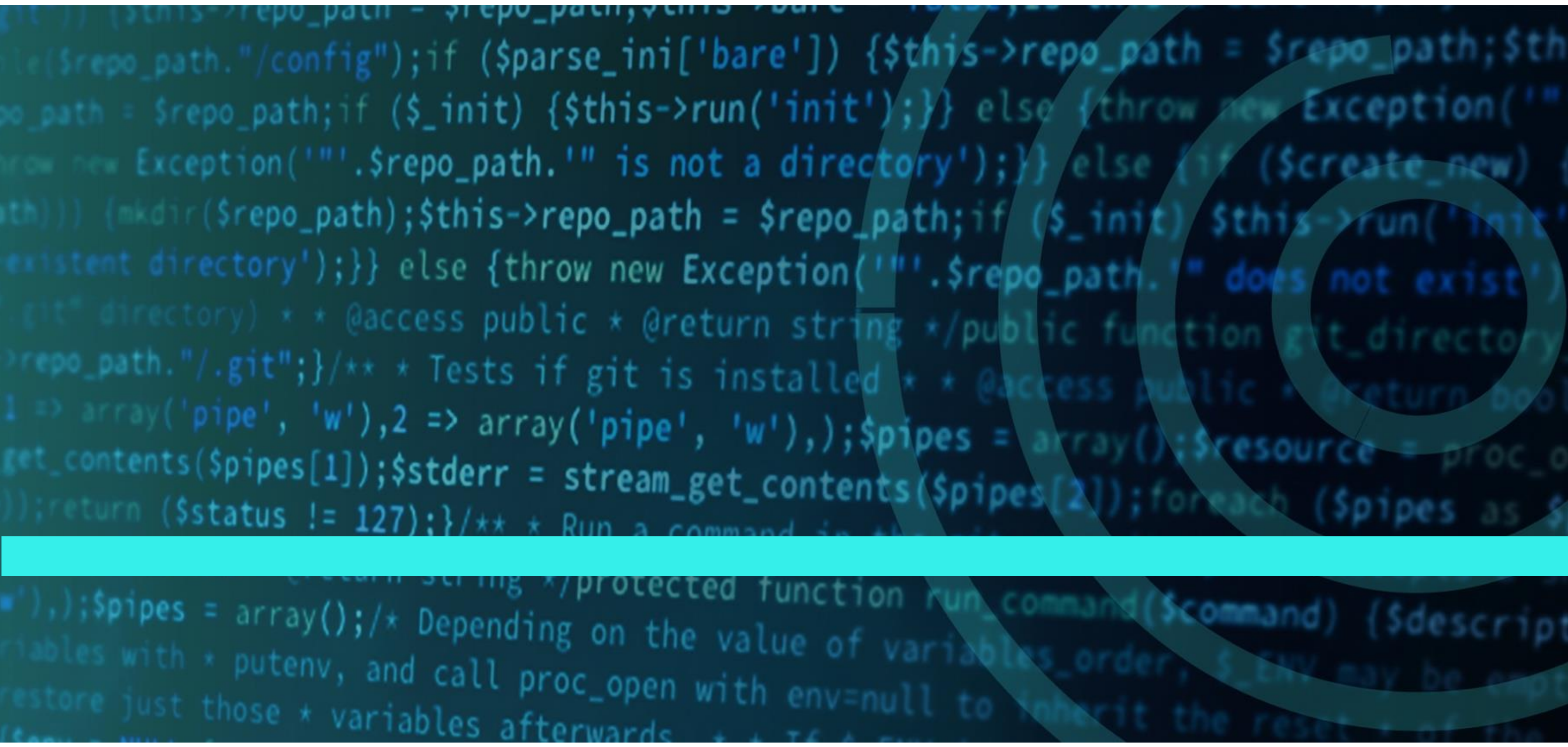


Table of Contents

Revised: 08/19/2019

Module 7: Calibration

The final module of the course provides several examples of interest rate model calibration. The module begins by exploring calibration via the closed-form solutions and characteristic functions. The module then introduces model error and describes various methods to perform interest rate model calibration. The module concludes by demonstrating the importance of accurate calibration.

Unit 1: Model Calibration

Unit 1: Video Transcript

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 1 : Notes

Suppose we wanted to price some exotic option which is not currently traded in the market. In order to do so, we would need to assume some model, and then either solve a closed-form solution for the price or implement some numerical method such as Monte Carlo. Let's say the option was on a stock, and we decide to use the Black-Scholes model. The two parameters we need to simulate a stock in Black-Scholes (in the risk-neutral measure) is a risk-free continuously compounded rate and a stock volatility. To find the risk-free rate, we could look to government bond prices and their yields, but how would we find the stock volatility? We must note at this point that option prices are usually quoted in the market as the Black-Scholes implied volatility. This means that, instead of a dollar amount, prices are given as the volatility level for the stock in the Black-Scholes model. You would then put this volatility in the Black-Scholes model and calculate the closed-form price, which is the amount you would need to pay to purchase the option.

In order to illustrate our techniques, we are going to assume that you can see the price itself, rather than the volatility. One way of estimating the volatility is to set it so that our model prices are as close as possible to market prices which we can observe. This is known as calibration – the setting of parameter values in a model to match target values (which will often be market prices in our case). This differs from estimation, which is the process of using a time series of historical data to find parameter values – such as, for example, a long-run mean. Calibration requires only a single time point's information and fitting your model best to that information.

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 consider an example of how one might calibrate in practice. Let's suppose you wanted to price a put on a stock with the following parameters:

- Risk-free continuously compounded interest rate, r, of 10%
- Initial stock price, S_0, of $\$100$
- Strike price, K, of $\$110$
- Time of maturity, T, of 2 years
- Unknown stock volatility, σ

Furthermore, you make all of the assumptions of the Black-Scholes model. This means that the put price is given by:

$$P \ = \ \Phi(-d_2)Ke^{-rT} - \Phi(-d_1)S_0, \tag{1}$$

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

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

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

We thus cannot find the closed-form put price without knowing the stock's volatility. Suppose with the following parameters:

, however, that there is a call option for this stock available in the market

- Strike, K_C, of \$95
- Time of maturity, T_C, of 1 year
- Price of \$15

We can use this call option to calibrate our model by finding an estimate for σ which would result in the same call price. Note that the call price is given by:

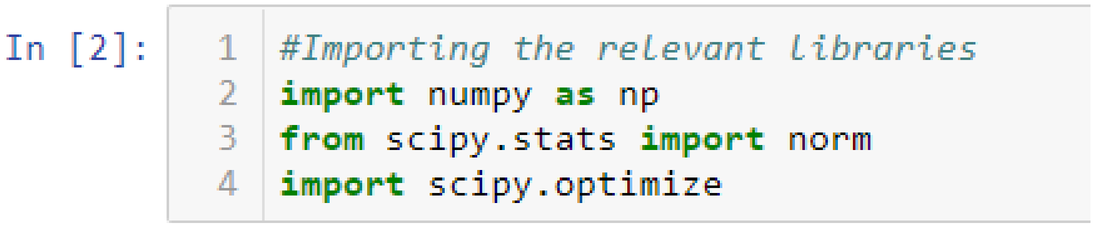
$$C = \Phi(d_{1_C})S_0 - \Phi(d_{2_C})K_C e^{-rT_C}, \tag{2}$$

where

$$d_{1_C} = \frac{1}{\sigma\sqrt{T_C}}\left(\ln\left(\frac{S_0}{K_C}\right) + \left(r + \frac{\sigma^2}{2}\right)T_C\right)$$

$$d_2 = d_1 - \sigma\sqrt{T_C}.$$

We will essentially be solving $C\ =\ \$5$ by adjusting σ, and then putting σ back into our formula for the put price. Let's see how we can do this in Python:



```
In [2]:
#Importing the relevant libraries
import numpy as np
from scipy.stats import norm
import scipy.optimize
```

As usual, we import the `numpy` and `scipy.stats.norm` libraries. In addition, we need the `scipy.optimize` library – this library contains a number of techniques which solve equations of the form $F(x) = 0$, where both F and x can be multidimensional. Since we are only solving for σ, we will not be concerned with multidimensionality.

```
In [3]:
#Setting variables
r = 0.1
S0 = 100
K = 110
T = 2
KC = 95
TC = 1
price = 15
```

We create the relevant variables. Note that we do not create a σ variable since this is what we are going to find.

```
In [4]:
#Functions for call
def d_1c(x):
    return 1/(x*np.sqrt(TC))*(np.log(S0/KC)+(r+x**2/2)*TC)
def d_2c(x):
    return d_1c(x)-x*np.sqrt(TC)
def C(x):
    return norm.cdf(d_1c(x))*S0 - norm.cdf(d_2c(x))*KC*np.exp(-r*TC)
def F(x):
    return C(x)-price
```

We create functions which find the call price for a given volatility, after having set the strike, risk-free rate, and initial stock price in advance. `d_1c` calculates d_{1_C}, while `d_2c` calculates d_{2_C} (both as a function of volatility). C uses these values to calculate the call price on the stock as a function of volatility. Finally, F finds the difference between the call price for a given volatility and the price observed in the market. Note that if we calibrate our model correctly, this difference should be as small as possible.

```
In [ ]:
#Solving for sigma
sigma = sigma.optimize.broyden1(F, 0.2)
```

We use the broyden1 method to find the value of σ for which $F(\sigma) = 0$. Note how the first argument is the function for which you want to find a root. The second argument is an initial guess. If F was a multidimensional function, the guess and output would be multidimensional arrays. You can check that you get a σ value of approximately 14.1726% – this is our calibrated σ. If you were to input this σ in the call price function, C, you would get a model price close to \$15, as required. Finally, we can use this σ to price our put according to equation (1):

```
In [ ]:
#Finding the put price
def d_1(x):
    return 1/(x*np.sqrt(T))*(np.log(S0/K)+(r+x**2/2)*T)
def d_2(x):
    return d_1(x)-x*np.sqrt(T)
def P(x):
    return -norm.cdf(-d_1(x))*S0 + norm.cdf(-d_2(x))*K*np.exp(-r*T)
put_price = P(sigma)
```

We create functions for our put price in the same way as we did for our call price. The variable `put_price` captures the price of the put option for the calibrated σ value – approximately \$3.62.

This is a very simple example of calibration. In reality, it is unlikely that you would observe vanilla call prices but not vanilla put prices, and vice versa. You could also use put-call parity to find one from the other. However, it illustrates the idea of calibration and a technique for applying it. It is worth noting that the `scipy.optimize` library contains numerous solvers (of which broyden1 is only one) – which one is best will depend on the task at hand.

In the remainder of this module, we will be performing calibrations on a few of the models introduced in earlier modules, such as the Vasicek and Heston models. It is thus important that you ensure you are up-to-date with everything that we have done up until now.

Unit 2: Error Analysis

Unit 2 : Video Transcript

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 $\hat{\sigma}\big(t, r(t)\big) =$ simply 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 2: Notes

It is important to recognize that every model we implement is unable to perfectly characterize the financial instrument that we are trying to model since models are always a simplification of reality. This means that there is an error inherent in every model we use. This is well captured by George Box, who said that "All models are wrong, but some are useful". Even though models don't give us perfect answers, they can give us answers which are close to being correct. This is better than having no idea whatsoever when tempered with an understanding of the limitations of the model. In general, we aim to strike a balance between a model being tractable and accurate – goals which are often in opposition to each other.

The main aim of this unit will be to go through where some of the sources of error in models come from. The actual analysis of this error requires extensive partial differential equation analysis, which is beyond the scope of this Module. As a result, the focus here will be to create a respect for the limitations of the models that we use.

Sources of error in models

Model error

The first source of error comes from the model itself being unable to fully capture a financial instrument's characteristics. For example, suppose 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. Now suppose
that we try to model the short rate using a Vasicek model, which assumes that the
short rate has the following dynamics:

$$dr_t = \hat{\mu}\big(t, r(t)\big)dt + \hat{\sigma}\big(t, r(t)\big)dW_t, r(0) = r,$$

where $\hat{\mu}(t, r(t)) = \alpha(\beta - r(t))$ (with their usual interpretations), $\hat{\sigma}(t, r(t)) =$ (a constant), and W_t is a one-dimensional Brownian motion.

It should be relatively easy to see that we are essentially trying to model $\mu(t, r(t))$ using $\hat{\mu}(t, r(t))$ and $\sigma(t, r(t))$ using $\hat{\sigma}(t, r(t))$. In other words, we are trying to model two potentially stochastic functions using a deterministic function and a constant. This is going to result in significant error over time. We could use a model which assumes more complex dynamics for the short rate, but this would come at the expense of the extent to which we can interpret the parameters in the model, and whether the model gives closed-form solutions to the prices of various market instruments.

Data error

The next source of error comes from the data itself. The quality of market data will have a direct impact on the accuracy of the model that you are using. This isn't necessarily a problem in a sufficiently sophisticated financial market. However, in the case of emerging markets, data quality will likely be a major consideration that you will have to account for when trying to implement a model in the market.

Another possible issue with the data that you use to calibrate a model is the number of sources of noise in the data. In the model error section, we looked at the model error inherent in the Vasicek model. Now, the Vasicek model doesn't allow for any drastic market movements whatsoever. Suppose that there is a sudden change in the short rate (which is entirely possible). The Vasicek model would not be able to account for this. This means that, if we use the data with these sudden changes in it to calibrate the model, we may get an inappropriate calibration (and subsequent error).

Strengths and weaknesses of models

You should now be convinced of the fact that all of the models which are implemented in practice are limited in some way. This includes all of the various models which we have dealt with through the modules in this course. With this in mind, we will now go through most of the models which have been dealt with and address their strengths and weaknesses, with a particular focus on calibration.

Black-Scholes model

The biggest advantage of the Black-Scholes model is its ease of use, and ubiquity – vanilla options quoted in terms of Black-Scholes implied volatility, giving an indication of the extent to which it permeates the financial sector. However, this tractability is as a result of the numerous assumptions made in the model, most of which are empirically wrong. For example, risk-free yields curves are almost never flat; trading costs are always incurred in reality, and fractional holdings of assets are usually impossible. As a result of these incorrect assumptions, the Black-Scholes model cannot be calibrated well to the market. This is most apparent in the volatility smile, where vanilla options of different strikes have a different Black-Scholes implied volatility. Since we assume a constant volatility for any given stock, it is impossible to fit our Black-Scholes model to all of the market-observed prices concurrently. Since the Black-Scholes model is ultimately a particularly inaccurate model, other, more extensive models were developed. That being said, the Black-Scholes model can still work relatively well empirically for hedging, and its importance in the market cannot be overstated.

Constant elasticity of variance model

The constant elasticity of variance (CEV) model offers a simple extension to the Black-Scholes by expressing the diffusion coefficient in the stochastic differential equation (SDE) of a stock price process as $\sigma\, S_t^{\gamma}$ as opposed to the σS_t of the Black-Scholes model. However, according to Ballestra (2011), this model cannot be calibrated in a way which correctly prices equity options.

Heston model

Similarly to the CEV model, the Heston model differs from Black-Scholes in its diffusion coefficient for the stock price process SDE. In the Heston model, this diffusion coefficient is stochastic. According to Mikhailov (2004), this offers an improvement over Black-Scholes by capturing important observed characteristics like non-lognormal asset returns and mean-reverting volatility. In terms of calibration, they state "the Black-Scholes volatility surfaces generated by Heston's model look like empirical implied volatility surfaces". In other words, this model offers a better calibration option than Black-Scholes. The drawback is that one cannot construct a portfolio which is perfectly risk-free since the stochastic volatility cannot be traded directly. More detail can be found in the article by Mikhailov and Nögel.

Short-term rate models

As will be shown in the next unit, the Vasicek model is unable to calibrate perfectly to certain yield curves. This can be improved using the Hull-White model, which allows the mean-reversion level of the short-term rate to vary over time. Thus, one can calibrate one's mean-reversion function so as to capture market observed bond prices.

LIBOR forward market model

As mentioned in the section on the LIBOR model, one of its biggest advantages is that it is automatically calibrated. This is because the initial forward rates are set at the market forward rates – a directly observable quantity.

It is important to keep in mind the relative strengths and weaknesses of various models when working with them. Often, the strength of one will be the weakness of another, and you'll need to make a decision as to which favorable attributes you value more.

Unit 3: Interest Rate Model Calibration

Unit 3: Video Transcript

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 [1]:
# 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')
```

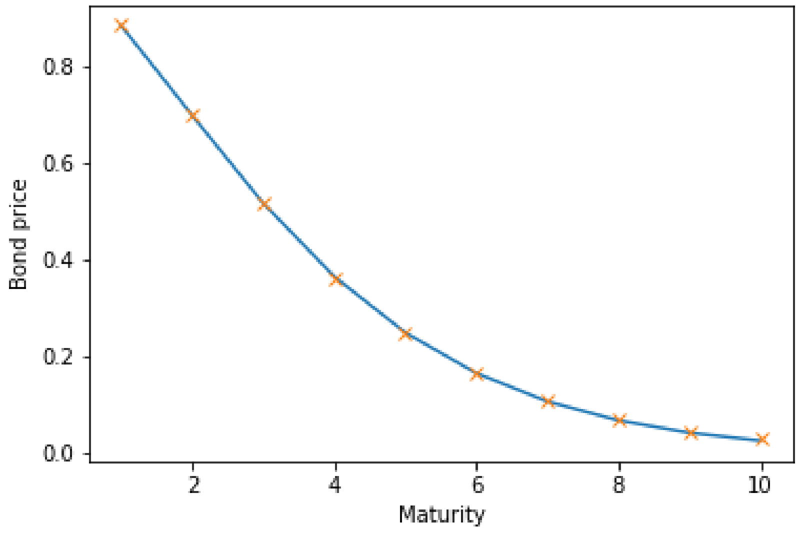


Figure 1: Market ZCB prices vs Calibrated MCB prices

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

Unit 3 : Notes

In this unit, we will be going through calibrating the Vasicek interest rate model. Since we've already introduced this model with some depth in the previous module, we will not be spending much time on the basics of it – instead, we will be focusing on conducting our calibration in Python.

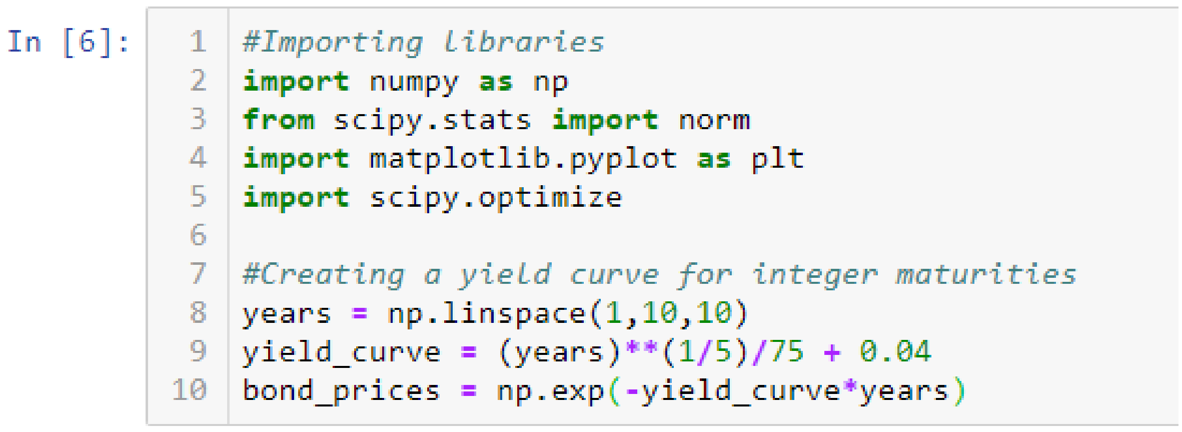
When it comes to calibrating interest rate models, the yield curve is usually used. One would use Zero-Coupon Bond (ZCB) prices and other instruments with various maturities to bootstrap a yield curve. By bootstrapping, we mean utilizing the prices for maturities which we can observe and implying prices for the maturities which we do not know. This is a complex field of study, and we will not go into the details here. Instead, we will create a believable yield curve using some function of our own devising, and calibrate our interest rate model to this.

We suppose that our yield curve for maturities from 1 to 10 years takes the form:

$$y(t) = \frac{1}{75}\sqrt[5]{t} + 0.04, \tag{3}$$

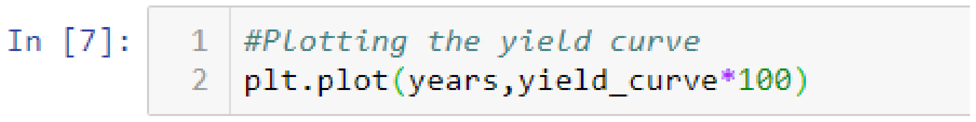
where $y(t)$ is the yield for a bond with maturity time t. We will be finding the bond prices with maturities $1, 2, 3, \ldots, 10$ years (i.e. no fractional years) and calibrating our model using these prices. We can then compare our calibrated model prices to the non-integer year maturity prices. Note that usually, you would have, for example, ZCB prices for different maturities, to which you would calibrate.

To begin, we implement this yield curve in Python, and find the associated bond prices:



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

#Creating a yield curve for integer maturities
years = np.linspace(1,10,10)
yield_curve = (years)**(1/5)/75 + 0.04
bond_prices = np.exp(-yield_curve*years)
```

We import the necessary libraries, including `scipy.optimize`, and create our yield curve and bond prices for the integer year maturities. We can plot the yield curve, to check that it has a realistic shape:



In [7]:
```
#Plotting the yield curve
plt.plot(years,yield_curve*100)
```

This gives a yield curve as shown in figure (2). Note that it has an increasing yield for maturity – a property often seen in practice.

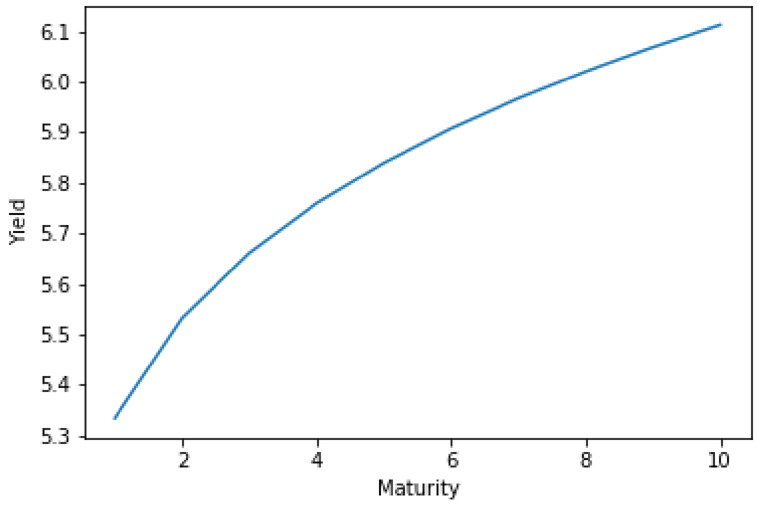


Figure 2: Yield curve

Now that we have a yield curve and associated bond prices, we can go about calibrating our interest rate model. Recall that, under the Vasicek model, the short rate is defined by the stochastic differential equation:

$$dr_t = \alpha(b - r_t)dt + \sigma dW_t,\ r_0 = r(0), \tag{4}$$

where W_t is a standard Brownian motion, and α, b, and σ are positive constants. We assume that we can observe the current short rate, r_0, leaving us with α, b, and σ to calibrate. In the previous module, we noted that the bond price at time t with maturity T is given by:

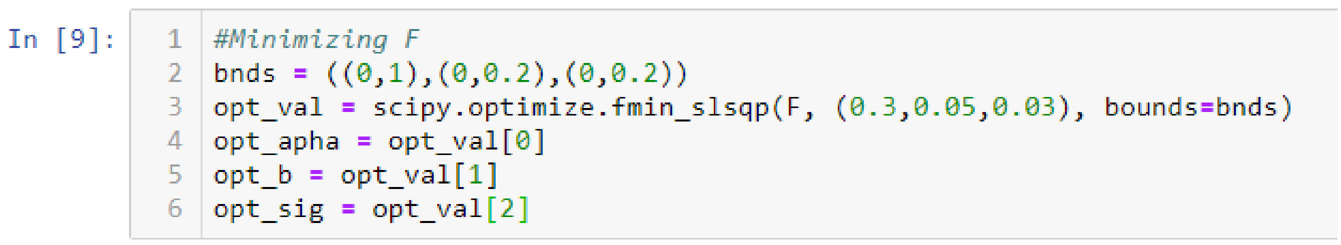
$$B(t,T) = e^{-A(t,T)r_t + D(t,T)}, \tag{5}$$

where $A(t,T) = \frac{1-e^{-\alpha(T-t)}}{\alpha}$ and $D(t,T) = \left(b - \frac{\sigma^2}{2\alpha^2}\right)[A(t,T) - (T-t)] - \frac{\sigma^2 A(t,T)^2}{4\alpha}$. In order to calibrate our model, we are going to create functions for A, D, and B that take t, T, α, b, and σ as parameters, and then minimize the difference between the bond prices implied by equation (4) and the bond prices given by our yield curve.

```
In [8]:
#Analytical bond price
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

#Difference between model bond prices and yield curve bond prices
def F(x):
    alpha = x[0]
    b = x[1]
    sigma = x[2]
    return sum(np.abs(bond_price_fun(r0,0,years,alpha,b,sigma)-bond_prices))
```

The functions A, D, and `bond_price_fun` calculate model implied prices for given α, b, and σ parameters (as well as time parameters, but we are not optimizing over these). We have assumed that the current short term rate, which we can observe, is 5%. F takes in a vector x, where the first element of x is α, the second is b, and the third is σ. It returns the sum of the absolute differences between the model implied bond prices, and the yield implied bond prices. Our goal in calibration is to make this value as small as possible. Note that we are optimizing over three variables, but it is essentially 10 equations which we are trying to set to 0. This means that it is unlikely that we will be able to calibrate our model perfectly.

In order to minimize F, we are going to use the `scipy.optimize.fmin_slsqp` routine. This routine minimizes a function which returns a scalar, over some vector input. We use this routine as it allows one to supply lower and upper bounds for each variable which you are optimizing over. This ensures that we don't have a case where volatility is negative, for example.



```
In [9]:  1 #Minimizing F
          2 bnds = ((0,1),(0,0.2),(0,0.2))
          3 opt_val = scipy.optimize.fmin_slsqp(F, (0.3,0.05,0.03), bounds=bnds)
          4 opt_apha = opt_val[0]
          5 opt_b = opt_val[1]
          6 opt_sig = opt_val[2]
```

The variable bounds in line 2 define our lower and upper bounds. The first tuple in bounds is the lower and upper bounds for `x[0]` (α); the next tuple lower and upper bounds for `x[1]` (b); and the last for `x[2]` (σ). This means our bounds are given by:

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

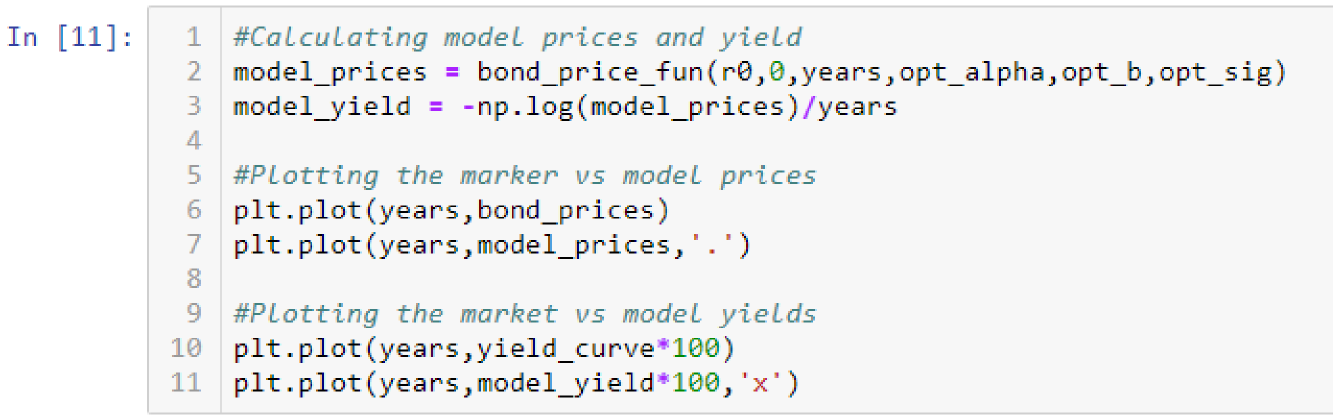
$$0 \leq b \leq 0.2$$

$$0 \leq \sigma \leq 0.2. \tag{6}$$

The lower bounds are all 0, since negative values would not make sense for these variables. The upper bounds are based on realistic expectations of how large these variables could be. For instance, it is unlikely that the long-term mean short-term rate would exceed 20%, and so that is our upper bound for b.

Line 3 performs the minimization of F, and outputs a vector of values for α, b, and σ.
As an inputs to the function `scipy.optimize.fmin_slsqp`, we provided the function to be minimized (F), an initial guess ($\alpha = 0.3$, $b = 0.05$, $\sigma = 0.03$), and the bounds for these variables being optimized over. Note that the initial guess should be within the magnitude that you expect values to be. In some cases, it may be necessary to rerun the algorithm with a number of different initial guesses and bounds, to find a good fit. But this needs to be balanced with realism in the calibrated parameters that the algorithm kicks out e.g. a b value of 200% may provide a good fit, but it is wholly unrealistic.

Finally, in lines 4, 5, and 6, we extract the α, b, and σ values from the optimized vector. Note that it is important to remain consistent with the order in which variables are placed in the vector.



```
In [11]:
 1  #Calculating model prices and yield
 2  model_prices = bond_price_fun(r0,0,years,opt_alpha,opt_b,opt_sig)
 3  model_yield = -np.log(model_prices)/years
 4  
 5  #Plotting the marker vs model prices
 6  plt.plot(years,bond_prices)
 7  plt.plot(years,model_prices,'.')
 8  
 9  #Plotting the market vs model yields
10  plt.plot(years,yield_curve*100)
11  plt.plot(years,model_yield*100,'x')
```

In lines 2 and 3, we find the model implied bond prices and yields with the optimized parameters. We then plot these model implied values versus the market values. Figure (3) shows the model prices as the yellow points, with the blue line showing the market prices. Figure (4) shows the model yields as yellow stars, with the blue line showing the market yields. As you can see, the model does not perfectly match the market. This is to be expected, since our model represents a simplified view of reality, and so cannot capture all intricacies present in the market.

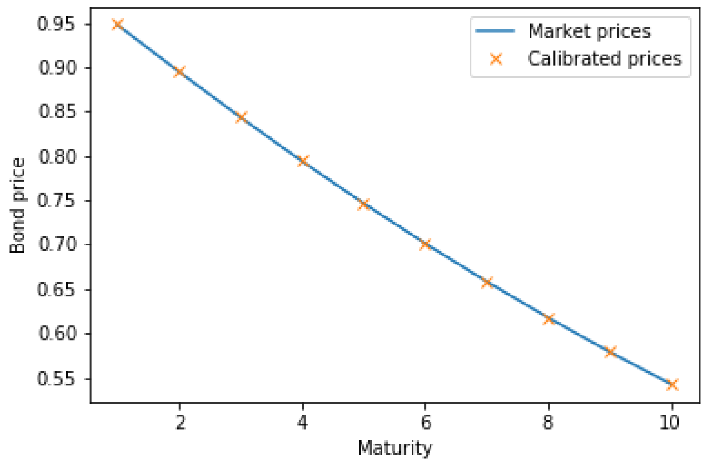


Figure 3: Model versus market bond prices

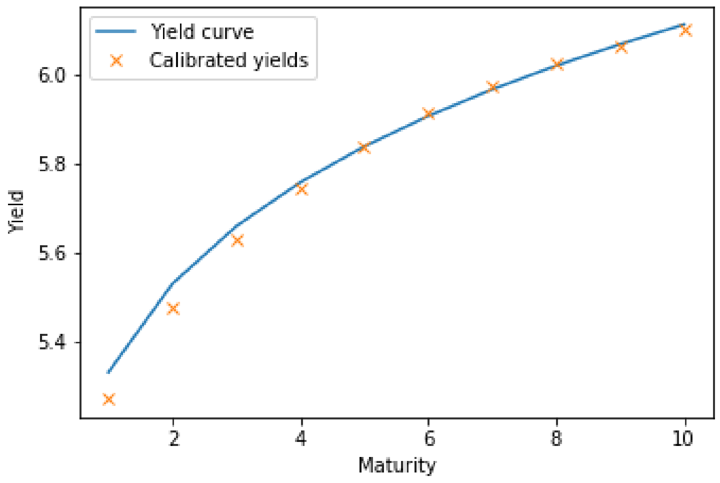


Figure 4: Model versus market yield

You might notice with the yield curve, that the model yield appears to fit the market yields closer for the longer maturities. This leads on to an important consideration in calibration. If you are calibrating a model to price a certain portfolio or derivative, and you know the maturity time of that portfolio, you might only use market values with similar maturity values as your derivative to calibrate your model. However, there are drawbacks to this.

Calibrating your model for certain maturities only might make it seem like your model replicates the market well, when in reality it only matches well with those maturities. This is not a problem if you are assuming that the model holds for those maturities only, but this would usually change any closed-form solutions which you are using. Calibrating a model to all maturities for which you have prices can potentially identify issues with the model you are using. With this in mind, discretion needs to be exercised when calibrating models, but you must always keep in mind that instruments with which you are hedging and pricing need to be calibrated.

In this unit, we have focused on calibrating the Vasicek model. The procedure we have followed will be similar for any model calibration for which you have closed-form solutions, namely:

1. Identify variables for which you need to calibrate
2. Collect market prices of instruments which rely on these variables
3. Minimize difference between market values and model values

In our case, we minimized the absolute differences, and weighted all maturities equally. You could change this to weight, for example, later maturities more heavily – this carries the same risks as using only certain maturities to calibrate.

Unit 4 : Pricing Options with Monte Carlo

Unit 4 : Video Transcript

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:

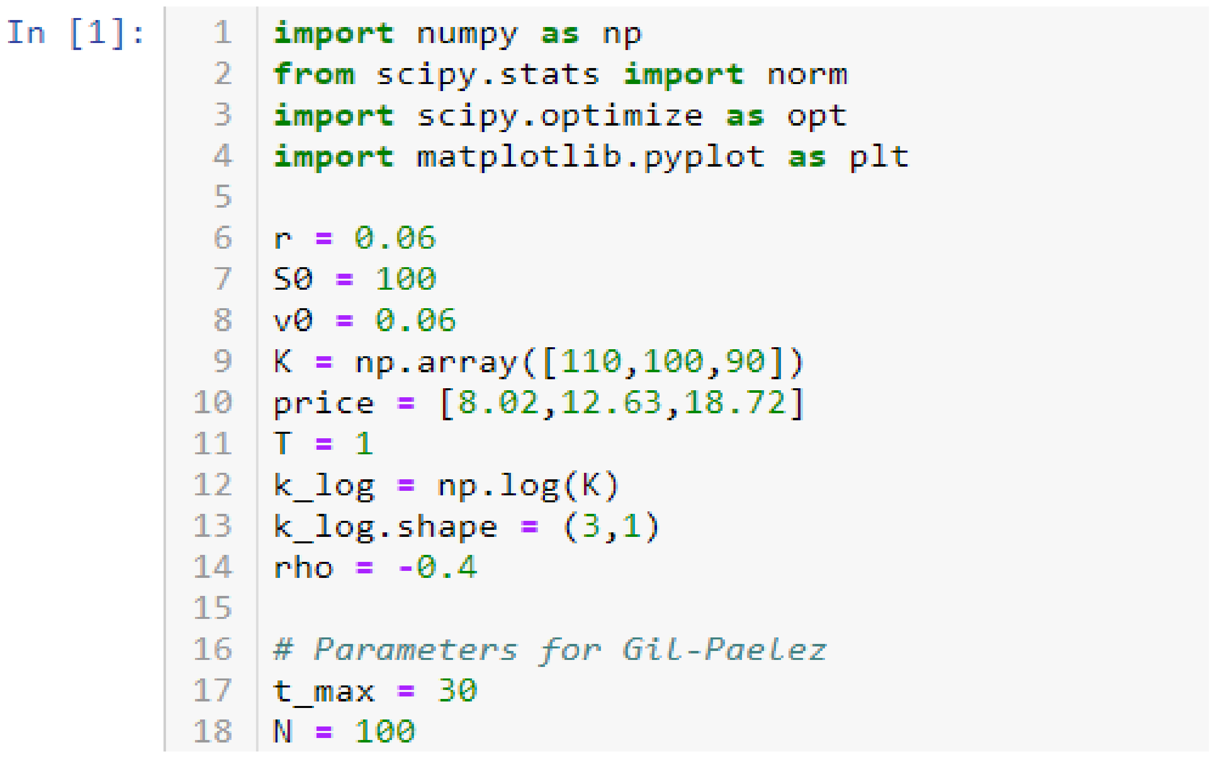
$$\varphi_{M_1}(t) = \phi_{s_T}(t)$$

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

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:



```python
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.

```python
#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.

```python
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.

Unit 4: Notes

We will now focus on how we can go about using the characteristic function pricing techniques developed in Module 4 to calibrate our models. Please make sure that you are comfortable with applying these techniques as we will be referring to some of these techniques often.

You may be asking yourself why we would even want to use characteristic function techniques for the sake of calibration given that we have been able to calibrate several models without it. The first reason is that there are many models which have closed-form characteristic functions, but do not have closed-form prices for even vanilla options. One such model is the Heston Model, presented in Module 5. This means that characteristic function techniques allow us to calibrate a much wider range of models.

The second reason comes from the efficiency of Fourier techniques when it comes to evaluating 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.

In other words, we are trying to find parameter values which minimize the difference between the market price and our model price. Because we are guessing values and checking how close this is to the market value, we will be doing many evaluations of our pricing function. If we are trying to calibrate several parameters, this can result in our need to make several thousands of function evaluations (and possibly even more). As a result, being able to efficiently price a market instrument will result in an efficient calibration process.

Calibrating the Heston model using Gil-Pelaez

Heston model revision

In this unit, we will go through how to calibrate the Heston model parameters using the Gil-Pelaez technique. Before we move on, remember that the dynamics of the Heston model are as follows:

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

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

Where dW_t^2 and dW_t^2 have correlation ρ. From these dynamics, we get the following characteristic function for $\log(S_t)$:

$$\phi_{S_T} = \exp\big(C(\tau; u) + D(\tau; u)v_0 + iulog(S_0)\big), \tag{7}$$

where:

$$C(\tau; u) = ri\tau u + \theta\kappa\left[\tau x_- - \frac{1}{a}log\left(\frac{1 - ge^{d\tau}}{1 - g}\right)\right],$$

$$D(\tau; u) = \left(\frac{1 - e^{d\tau}}{1 - ge^{d\tau}}\right)x_-, \tag{8}$$

and

$$\tau = T - t,$$

$$g = \frac{x_-}{x_+},$$

$$x_\pm = \frac{b \pm d}{2a}$$

$$d = \sqrt{b^2 - 4ac}$$

$$c = -\frac{u^2 + ui}{2}$$

$$b = \kappa - \rho\sigma iu$$

$$a = \frac{\sigma^2}{2}.$$

(9)

Option pricing using Gil-Pelaez

We can write the price of a vanilla call option as:

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

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

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

(10)

where:

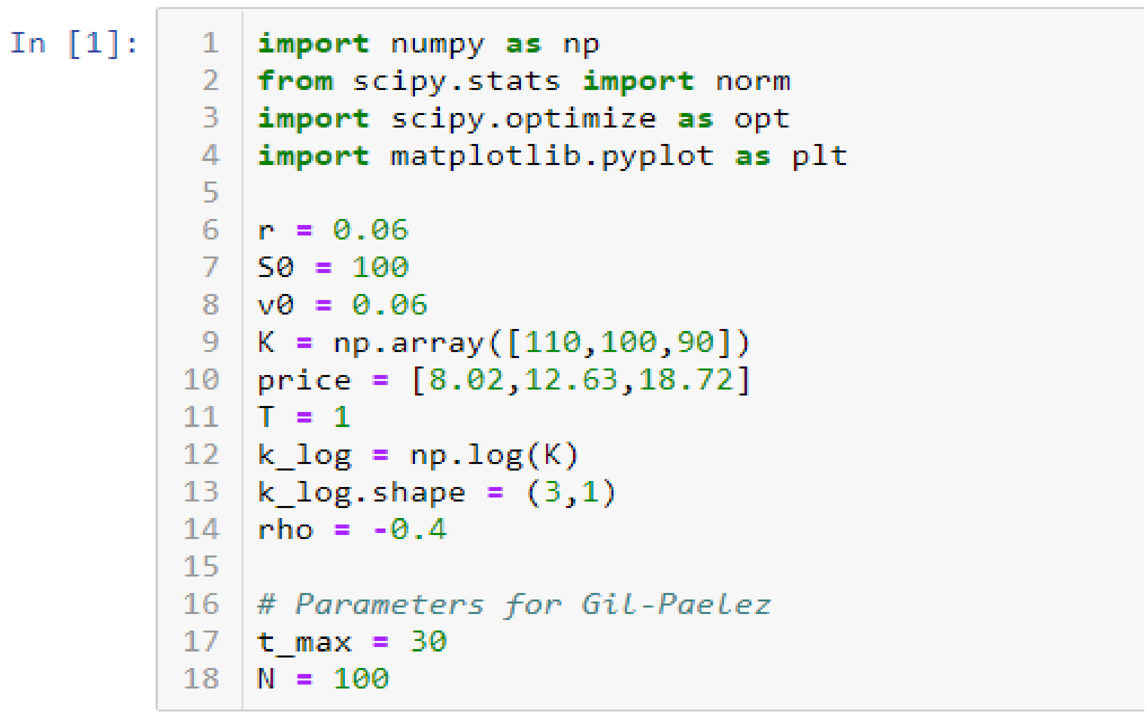
$$\varphi_{M_1}(t) = \phi_{s_T}(t)$$

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

Calibrating the Heston model in Python

Let's take ρ and r as being given, and look at how we can calibrate κ, θ, and σ. Because we are trying to calibrate 3 parameters, we need at least 3 data points.

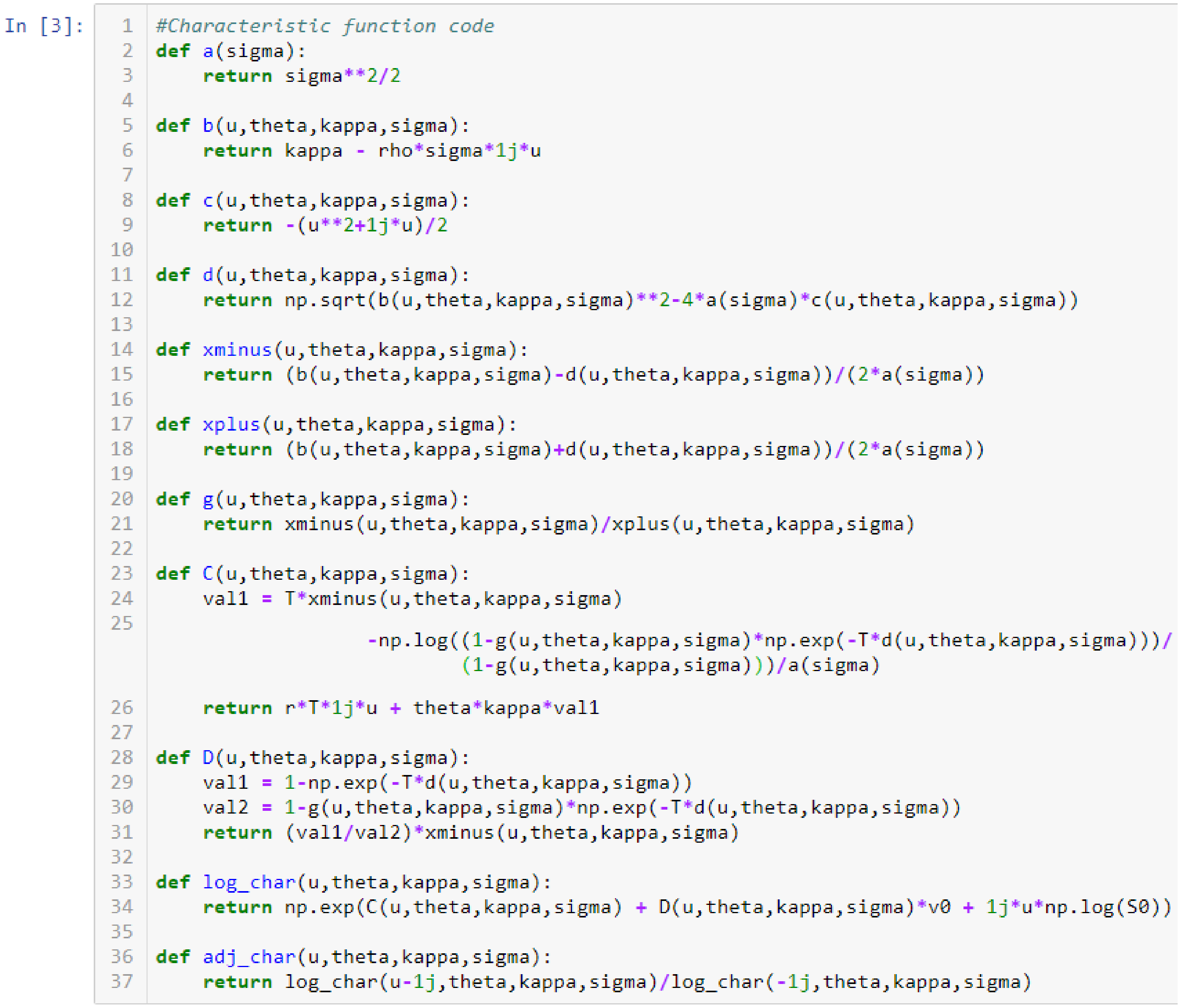
Now, let's import our libraries and set our initial parameter values:



```
In [1]:
 1  import numpy as np
 2  from scipy.stats import norm
 3  import scipy.optimize as opt
 4  import matplotlib.pyplot as plt
 5  
 6  r = 0.06
 7  S0 = 100
 8  v0 = 0.06
 9  K = np.array([110,100,90])
10  price = [8.02,12.63,18.72]
11  T = 1
12  k_log = np.log(K)
13  k_log.shape = (3,1)
14  rho = -0.4
15  
16  # Parameters for Gil-Paelez
17  t_max = 30
18  N = 100
```

Observe that lines 9-11 represent our data points. We have the strikes, and market prices of 3 call options all with a maturity of 1 year. The r variable represents the risk-free continuously compounded interest rate, $S0$ is the current share price, $v0$ is the current share variance, rho is the correlation between the share price and its variance process, and `k_log` is the log-strike. Line 13 adjusts the log-strike variable so that it is a column vector. This will help us to vectorize our code later on. Then, the `t_max` variable is the effective upper bound of integration, and N is the number of intervals for our integration.

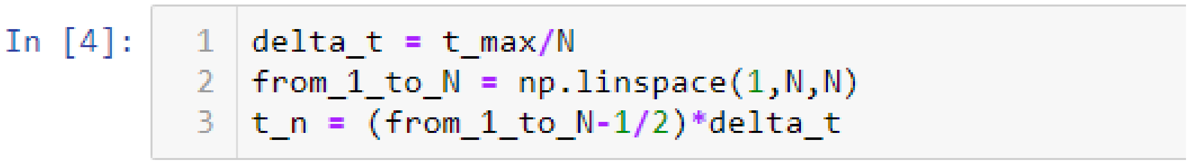
We can now define the functions required to evaluate the characteristic function of the Heston model:



```
In [3]:
#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)
```

Observe that every function takes in as inputs u, as per normal, as well as the parameters that we are trying to calibrate. The only difference now is that the function, a, takes in sigma as an input as it depends on the σ value of the variance process.

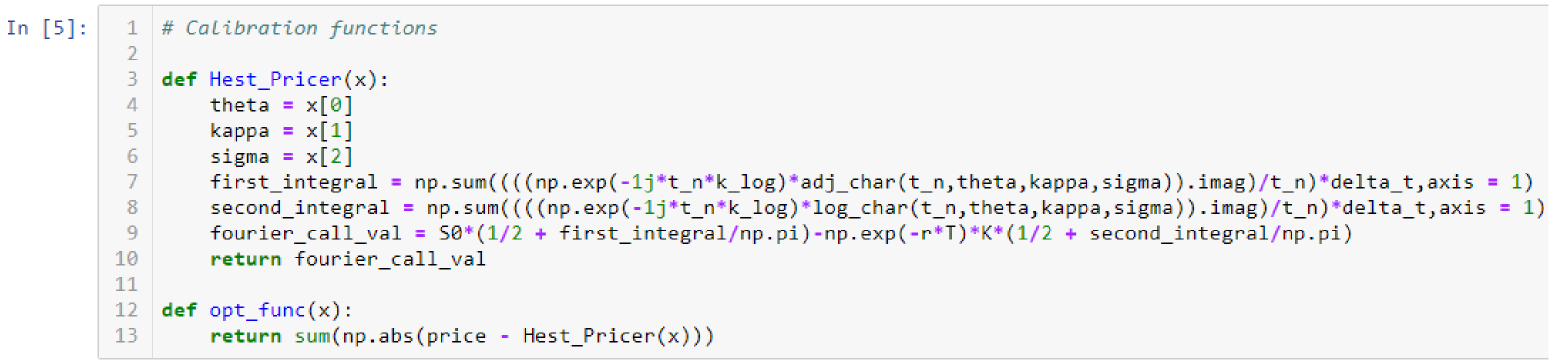
We now define additional variables required to apply Gil-Pelaez:



```
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
```

Where `delta_t` represents the step length, and `t_n` is the grid of values that we evaluate the function at.

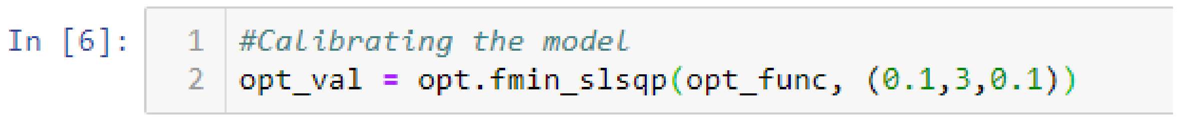
Then we define a function which evaluates the Heston call price for a given set of parameter values, as well as a function which we can use for our optimization:



```
In [5]:
 1  # Calibration functions
 2
 3  def Hest_Pricer(x):
 4      theta = x[0]
 5      kappa = x[1]
 6      sigma = x[2]
 7      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)
 8      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)
 9      fourier_call_val = S0*(1/2 + first_integral/np.pi)-np.exp(-r*T)*K*(1/2 + second_integral/np.pi)
10      return fourier_call_val
11
12  def opt_func(x):
13      return sum(np.abs(price - Hest_Pricer(x)))
```

The `Hest_Pricer` function returns the Heston call price. Given that we have three strikes, it will return a 1x3 array. The `opt_func` returns the sum of the absolute differences between our estimated price (for each parameter guess) and the actual market prices. Thus, it will return a single value. Note that the optimization function must take in only one input. The input can be an array, however, which allows us to optimize several variables at once. We could have let the `Hest_Pricer` function depend on the variables we are trying to calibrate, and move the code from lines 4-6 into the `opt_func` function. This gives the same result.

We can now apply an optimizer to our optimization function:



```python
#Calibrating the model
opt_val = opt.fmin_slsqp(opt_func, (0.1,3,0.1))
```

The first input into the optimizer is the function that we are trying to minimize, and the second input is our initial guesses for each value.

If applied correctly, this leads to the following values for our variables:

$$\hat{\theta} = 0.05988378$$

$$\hat{\kappa} = 3.07130476$$

$$\hat{\sigma} = 0.25690418,$$

where $\hat{\theta}$ denotes the calibrated value of θ. The resulting sum of absolute differences between the estimated call prices and the market call prices is 0.0024527. Note that we were able to calibrate our model to account for different implied volatilities for different strikes – something which the Black-Scholes model is not able to do. This means that we can calibrate our Heston model to take into account the volatility smile.

Calibrating using FFT pricing

Using the FFT for calibration has several pros and cons. The benefit is that the algorithm for calculating the call values is extremely efficient and allows us to calculate the prices of options for a range of strikes all at once. However, the FFT method requires the use of a fixed grid of log-strike values. As a result, it will also calculate option prices which aren't relevant to the available data. As a result, you will likely be forced to use interpolation to find the call price corresponding to specific strike values. Despite this, the FFT with interpolation is still generally more efficient than other calibration methods.

Because of this, it is sometimes inappropriate to use the FFT directly for calibration. When using this method, you will need to balance the additional efficiency against any potential loss in accuracy as a result of any interpolation used.

Bibliography

Ballestra, L.V., and Pacelli, G. (2011). “The Constant Elasticity of Variance Model: Calibration, Test and Evidence from the Italian Equity Market”, *Applied Financial Economics* 21(20): 1479-1487.

Brigo, D. and Mercurio, F. (2007). *Interest Rate Models-theory and Practice: {W}ith Smile, Inflation, and Credit*. Springer Science & Business Media.

Carr, P. and Madan, D. (1999). “Option Valuation Using the Fast Fourier Transform”, *Journal of Computational Finance* 2(4): 61–73.

Cooley, J.W. and Tukey, J.W. (1965). “An Algorithm for the Machine Calculation of Complex Fourier Series”, *Mathematics of Computation* 19(90): 297–301.

Cox, J.C. and Ingersoll Jr, J.E. and Ross, S.A. (1985). “An Intertemporal General Equilibrium Model of Asset Prices”, *Econometrica: Journal of the Econometric Society*, 363-384.

Davis, M.H. (2011). “The Dupire Formula”, *Imperial College London, Finite Difference Methods Course material.*

Heston, S.L. (1993). “A Closed-form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options”, *The Review of Financial Studies* 6(2): 327-343.

Hull, J. and White, A. (2001). “The General Hull-White Model and Supercalibration”, *Financial Analysts Journal*, pp 34-43.

Hunter, C.J., Jackel, 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.

Mikhailov, S., and Nigel, U. (2004). *Heston's Stochastic Volatility Model: Implementation, Calibration and Some Extensions*. John Wiley and Sons

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

Vasicek, Oldrich. (1977). "An equilibrium characterization of the term structure", *Journal of Financial Economics* 5(2): 177-188.