Student Research Paper Competition held by HSE University

Stochastic Risk-Free Interest Rate Models: Calibration Techniques and Practical Application

Key words: mathematical finance, interest rate models, risk-free interest rate, stochastic differential equations, machine learning

Abstract

At the present paper we outline and analyse the most remarkable models of risk-free interest rates in the form of random processes using approaches and concepts from Mathematical Finance. We analyse various techniques that can be used for discrete approximation of stochastic differential equations and calibration of model parameters. We compare the predictive power of models discussed and mathematically show the convergence of estimated model parameters to certain values, which are of particular interest. The main result of the paper is a new method of parameters calibration based on statistical methods with elements from machine learning. Apart from analytical formulation the method is developed in a form of algorithm that allows to effectively predict interest rates. It is implemented as a library of Python classes which allows the user to get estimations of future risk-free interest rates, requiring only the input of a list (vector): historical data on risk-free interest rates for a certain maturity. The developed program can be accessed in the Annexes.

Contents

Chapter 1

Introduction

Proper pricing of financial instruments can be called a keystone of the world economic system. Generally, a proper mathematical model of a certain instrument price requires understanding of its volatility – the level of risk –, and interest rate – the return that the instrument yields. Many financiers developed concepts of the financial markets structure, like the CAPM (capital asset pricing model) that was introduced by *William F. Sharpe* (1964) [2], *Stephen A. Ross* (1977) [7] and other scientists. The model refers to the equilibrium condition in the market and investors' mean-variance preferences. It approaches the problem of pricing by using the market portfolio and claims that the expected return on an asset is the sum of risk-free rate of return and the market risk-premium corrected by a parameter beta, which reflects the correlation of a considered asset's expected return and the one of the market portfolio. Another respectable concept is APT (arbitrage-pricing theory), that was proposed by *Stephen A. Ross* (1976) [5]. Similarly to CAPM, it approaches the problem of pricing by assuming that the market is in equilibrium, what means that arbitrage opportunities – possibilities to gain profit immediately, bearing no risk – do not exist, and the economy is affected by a pool of certain risk factors. Understanding the sensitivities of an instrument to a list of factors (generally using linear regression) allows one to properly estimate the instrument.

Besides the models listed above gained praise and respect all across the globe in late 20^{th} century, one may mention that they are based on strict assumptions of equilibrium condition in the financial market, and the knowledge of true parameters values. In practice, proxies are used in order to apply these models to real data, e. g. a certain pool of factors is used for APT model, however if some key factor is not included, the application of such model can result in biased results due to the problem of omitted variables. Additionally, S&P 500 [1] is utilized as a proxy for market portfolio, which is not strictly correct as it does not include the financial instruments of all markets, resulting again in biased results.

Thus, understanding the expected return, or a required rate of return, on a certain instrument is a challenging task which can be approached in different ways, that can usually result in biased model estimations. In the industry of banking, several proxies are used as a risk-free rate, like RUONIA [2] for Russian financial market, SOFR [3] – for American, SARON [4] – for Swiss market. However, these rates are overnight, thus to get interest rates with longer maturities financial intermediaries either use interpolation or get equilibrium rates from interest rate swaps (IRS) market. In a nutshell, IRS is a financial instrument which allows periodically exchange floating for fixed (or visa versa) interest rate cash flows for a pre-determined period of time and in a pre-determined amount. For instance, a third-party company can buy IRS from a bank for 3 years, by which it quarterly pays the bank floating interest rate (RUONIA rate for the previous quarter) on a certain notional amount and receives from

[1] https://www.investing.com/indices/us-spx-500
[2] https://www.cbr.ru/hd_base/ruonia/
[3] https://www.newyorkfed.org/markets/reference-rates/sofr
[4] https://www.six-group.com/exchanges/indices/data_centre/swiss_reference_rates/reference_rates_en.html

the bank a fixed risk-free interest rate (which does not change through the whole 3 years), still on the same notional amount. The IRS payment scheme is illustrated in figure 1.1, where an by an interest rate swap one pays RUONIA on notional amount every period from t_0 to T (what is the floating leg), and receives a fixed rate r on notion in exchange (what is the floating rate leg). As banks face many such counterparties which demand IRSs, the risk-free interest rate at which both parties agree on is the equilibrium market risk-free interest rate which can be used for modelling.

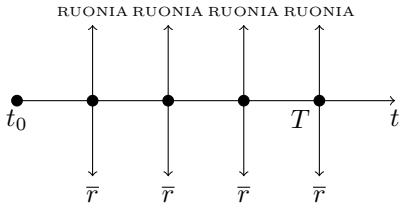


Figure 1.1: Interest rate swap payment scheme

Mathematical finance, which is also used for estimation of market-determined interest rates, has less assumptions than CAPM, APT, or other financial theories. It bases its models predominantly on stochastic calculus and approaches the problem of pricing by modelling the stochastic differential equations (SDEs) which describe the evolution of an asset, volatility, interest or any other financial parameter in time. Generally, random (stochastic) processes in financial mathematics are denoted in the form of a stochastic differential equation of the following type:

$$dX_t = a_t dt + b_t dW_t \tag{1.1}$$

where $(X_t)_{t\geq 0}$ is a random process, $a_t dt$ is a trend (drift) term with stochastic parameter a_t, and $b_t dW_t$ as a diffusion term with a stochastic parameter b_t. $(W_t)_{t\geq 0}$ is Brownian motion (Wiener's process) which is discussed in details in section 1.1 (note that (1.1) can be simplified by using constant parameters a and b instead of a_t and b_t).

The challenge of working with differential equations is that in practice time is discrete, whereas SDEs are continuous. Thus, to model SDEs we have to use discrete approximations of differentials: difference schemes. Let $L[f] = \frac{df}{dx}$ be the differential operator of a function $f(x)$ which we approximate on the grid $\overline{\omega_h} = \{x_i = ih : i = 0, ..., N, hN = 1\}$ with a step h in an arbitrary internal point $x \in \overline{\omega_h}$. Linear operator can be approximated in the following ways:

- $L_h^+[f] = \frac{f(x+h)-f(x)}{h}$ – one-sided right hand side derivative of $f(x)$
- $L_h^-[f] = \frac{f(x)-f(x-h)}{h}$ – one-sided left hand side derivative of $f(x)$
- $L_h^{0.5}[f] = \frac{f(x+h)-f(x-h)}{2h}$ – two-sided derivative of $f(x)$

By re-writing the SDE (1.1) using right-hand differential operator $L_h^+[X]$ we get:

$$X_{t+h} - X_t = a_t h + b_t(W_{t+h} - W_t)$$

In the equation above, $h = \Delta t$ and $W_{t+h} - W_t \equiv \xi_t \sim \mathcal{N}(0, h)$ (refer to section 1.1). Thus, SDE (1.1) can be re-written in the following difference scheme:

$$\begin{cases} X_{t+h} - X_t = a_t \Delta t + b_t \xi_t \\ X_0 = x \in const \end{cases}$$

what can be used for programming SDE X_t in discrete time and increments of X.

This paper tackles the problem of prediction: the main objective is to properly estimate future risk-free interest rates. For this, I consider three models of risk-free interest rate evolution. The first one is ordinary least squares linear regression (OLS):

$$r(t) = \alpha + \beta t + \varepsilon_t$$

Where α, β are estimated parameters, $r(t)$ is interest rate, t is time, and ε_t is the error term, such that for any t $\mathbb{E}[\varepsilon_t] = 0$ and ε_t are independent identically distributed (i.i.d.) random variables. It is a rather simple model which will be used as a baseline to compare the predictive power of other models.

The next two models – Merton and Vasiček models – are more advanced: they come from the stochastic nature of interest rate. Merton model has the following form:

$$dr_t = \alpha dt + \beta dW_t$$

Here dr_t is a stochastic differential of interest rate; αdt is the drift (trend) term responsible for the sign and size of interest rate growth with respect to time, βdW_t is the diffusion term responsible for volatility of interest rate.

Vasiček model is different to Merton as it has the built-in concept of mean-reversion (shown in (1.10)). It has the following form:

$$dr_t = (\alpha - \beta r_t)dt + \nu dW_t$$

Programming OLS is a basic task, however modelling stochastic differential equations requires proper parameters calibration and estimation of these continuous differentials with discrete differences. In this paper I use methods of splitting the sample into train and test sub-samples (so that the parameters are calibrated only on historical data) and machine learning techniques, such as gradient descend, to minimize the loss function of interest rates estimations in order to find most efficient parameters estimators (methods are described in details in chapter 2). The main tools that I use in my work are Python classes, which can be reached in the Annexes section. The final results of my work are the developed from scratch program based on the construction of future interest rate confidence intervals, and the algorithm of parameters calibration (α and β in Merton model, α, β and ν in Vasicek) that I invented specifically for the purpose of the problem tackled in this paper. The algorithm minimizes the estimators' loss function by the method of gradient descend, and stops the calibration procedure by the effective stopping rule so that overlearning is prevented, what constitutes the scientific novelty of this paper. As a result, the developed program is rather robust. It is proved theoretically (in chapter 2) that it outputs consistent risk-free interest rate estimates, and I showed empirically in chapter 3 the correctness of future risk-free interest rates forecasts, and in sections 2.2 and 2.3 I showed the convergence of model's parameters to non-trivial values, what is of particular interest.

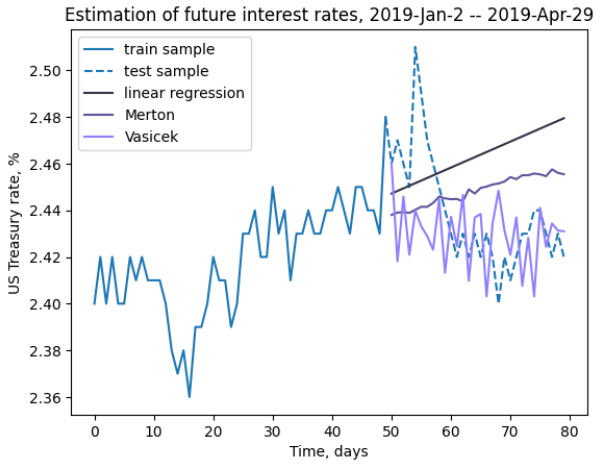


Figure 1.2: Results of different risk-free interest rate models estimations and simulations

By applying the developed library of Python Classes to real data, I conclude that stochastic models of Merton and Vasiček significantly outperform OLS: residual mean sum of squared errors (RMSE) metric is nearly two times less for both stochastic models (described in the Conclusion). The intuition is that stochastic models are more relative to the process of interest rate evolution since they include the random nature of rate behaviour. Additionally, those models do not assume linearity and same distribution for error terms, what is the case for OLS. Besides, comparing Merton and Vasiček with each other, Vasiček turns out to perform better than Merton, since Merton model is basic, whereas Vasiček has a more complex representation of the drift term.

In this paper, the first chapter gives an introduction to basic mathematical finance models, risk-free interest rate stochastic differential equations models, and the overview of machine learning techniques which are used in calibration procedures. Next chapter provides the methodology of a developed algorithm with mathematical background, as well as theoretical expectations on the results of the program. The third chapter gives results of applying the code to real data, and the last chapter concludes.

1.1 Overview of risk-free mathematical finance models

The field of study that is covered in this paper is mathematical finance. The models that are discussed in this work were developed with the aid of stochastic calculus, also known as Itô (1915-2008) calculus, which focuses on the study of random (or stochastic) processes. Generally speaking, in ordinary calculus we work with constants and variables, for example with x. In probability theory and statistics x would become a random variable which has its population distribution, say standard normal distribution $x \sim \mathcal{N}(0,1)$. In stochastic calculus $(x_t)_{t\geq 0}$ or $x(t), t \geq 0$ are random processes of time t which evolve in time and are dependent on the previous value. For instance, the sum of $x_0, x_1, ..., x_t$ where $\forall t : x_t \sim \mathcal{N}(0,1)$ would be a random process which might have different properties depending on the probability measure $\mathbb{P}$. For a more detailed description of stochastic calculus theoretical background, one may refer to the books of S.E. Shreve 'Stochastic Calculus of Finance' [10] and A.V. Bulinskiy, A.V. Shiryaev 'Theory of Random Processes' [11].

The concept of stochastic processes is deeply used in mathematical finance. Random processes can be used to describe instruments' price evolution throughout time. In stochastic calculus random process is most commonly denoted by a stochastic differential equation (SDE), not by an explicit process formula. In financial mathematics the most general approach is to split the instruments differential in two parts: one that is responsible for trend (or drift) which evolves with respect to time

t, and one that stands for the spread or volatility which evolves with respect to some random process W_t, where W_t is denoted as Wiener's (1894-1964) process or a Brownian motion.

Brownian motion or Wiener's process W_t has two mathematical definitions. By the first definition, W_t is a Brownian motion if

- $W_0 = 0$, $W_t - W_s \sim \mathcal{N}(0, t-s)\ \forall t > s \geq 0$
- Process with independent increments, i.e. $\forall n \in \mathbb{N}$; $\forall t_1, ..., t_n$: $0 < t_1 < ... < t_n$: $(W_{t_1}, W_{t_2} - W_{t_1}, ..., W_{t_n} - W_{t_{n-1}})$ are independent identically distributed random variables

By the second definition, W_t is Brownian motion if

- $(W_t)_{t\geq 0}$ is a Gaussian process, i.e. $\forall t_1, ..., t_n \geq 0$ the vector $(W_{t_1}, ..., W_{t_n})$ is normally distributed, i.e. it is jointly normal: $\forall c_1, ..., c_n \in \mathbb{R}$: $c_1 W_{t_1} + ... + c_n W_{t_n} \sim \mathcal{N}(\cdot, \cdot)$
- $\mathbb{E}[W_t] = 0$
- Covariance function is denoted as $\gamma(t, s) = Cov(W_t, W_s) = min\{t, s\}$

Combination of the SDE properties resulted in the most common formula of stock price random process $(S_t)_{t\geq 0}$, which was widely used in the article of F. Black and M. Schöles 'The Pricing of Options and Corporate Liabilities' [3] and R. Merton 'Theory of Rational Option Pricing' [4]:

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{1.2}$$

where μ is the trend (or drift) parameter and σ is the spread parameter. Note that both μ and σ are constants in this model. The parameter σ can be naively estimated by the variance of past random variable values $Var(S_t|\mathcal{F}_t)$ where $\mathcal{F}_t$ is the observable information that we have at current moment t (called as Sigma Algebra). As for the μ parameter, the estimation of it is more tricky. Naively it can be estimated by the stock's S required rate of return; however, this rate of return is not observable and one may use various proxies, getting hence the biased result. Otherwise, one may make a transition from market probability measure $\mathbb{P}$ to risk-neutral measure $\mathbb{Q}$ using Girsanov's theorem [1]. By this theorem the following is true:

$$dS_t^{\mathbb{P}} = \mu S_t dt + \sigma S_t dW_t^{\mathbb{P}} \iff dS_t^{\mathbb{Q}} = r S_t dt + \sigma S_t dW_t^{\mathbb{Q}} \tag{1.3}$$

Where r is the risk-free interest rate. For more detailed transition to risk-neutral measure one may refer to [10] and [9]. Note that in this model r is also a constant. Mathematically, it is correct, however in real life financial markets constantly adjust and risk-free measures alter with respect to time as well. Thus, considering risk-free interest rate as a random process will result in more precise stock price estimation. Hence, the model (1.2) was extended to the following form:

$$dS_t = r_t S_t dt + \sigma_t S_t dW_t \tag{1.4}$$

Where $(r_t)_{t\geq 0}$ and $(\sigma_t)_{t\geq 0}$ are not constants but random processes. Both studies of volatility and risk-free rate are vast scientific fields. This paper will focus on the latter topic, addressing to the works of Merton R.C. [4], Vasiček O. [6], and propose a statistics-based method of parameters calibration in these models using observed data and machine learning techniques.

1.1.1 Merton model (1973) of risk-free interest rate [4]

In his model, R.C. Merton considers risk-free interest rate as a random process $(r_t)_{t\geq 0}$ with two components: trend parameter α and volatility parameter β (similar to (1.2)). The proposed by him SDE is:

$$dr_t = \alpha dt + \beta dW_t \tag{1.5}$$

To derive an explicit formula, one may take Itô integral and receive:

$$\begin{aligned} r_t &= r_0 + \int_0^t \alpha ds + \rho W_t \\ &= r_0 + \alpha t + \rho W_t \end{aligned} \tag{1.6}$$

1.1.2 Vasiček model (1977) of risk-free interest rate [6]

The model proposed by Oldřich Alfons Vasiček in his paper [6] also consideres risk-free interest rate as a random process $(r_t)_{t\geq 0}$. This model, oppositely to the one of Merton (1.5), has a stochastic term r_t in dt part, and it is known as the model with *mean-reversion*. The SDE is the following:

$$dr_t = (\alpha - \beta r_t)dt + \nu W_t \tag{1.7}$$

To derive an explicit formula of r_t, suppose that (as it is proposed in [10]) r_t follows the following process:

$$r_t = e^{-\beta t} r_0 + \frac{\alpha}{\beta}(1 - e^{-\beta t}) + \sigma e^{-\beta t} \int_0^t e^{\beta s} dW_s \tag{1.8}$$

To verify that (1.8) is consistent with (1.7), I will apply Itô differential to this formula. Denote

$$f(t, X_t) = e^{-\beta t} r_0 + \frac{\alpha}{\beta}(1 - e^{-\beta t}) + \sigma e^{-\beta t} X_t, X_t = \int_0^t e^{\beta s} dW_s$$

Note that $dX_t = e^{\beta t} dW_t - e^{\beta \times 0} dW_0 = e^{\beta t} dW_t$. Next, it is necessary to take partial derivatives of $f(t, X_t)$ and plug it into formula.

$$\begin{aligned} \frac{\partial}{\partial t} f(t, X) &= -\beta e^{-\beta t} r_0 + \frac{\alpha}{\beta}(-e^{-\beta t})(-\beta) - \beta \sigma e^{-\beta t} X \\ &= \alpha e^{-\beta t} - \beta (f(t, X) - \frac{\alpha}{\beta}(1 - e^{-\beta t})) \\ &= \alpha e^{-\beta t} - \beta f(t, X) + \beta \cdot \frac{\alpha}{\beta}(1 - e^{-\beta t}) = \alpha e^{-\beta t} - \beta f(t, X) + \alpha - \alpha e^{-\beta t} \\ &= \alpha - \beta f(t, X) \\ \frac{\partial}{\partial X} f(t, X) &= \sigma e^{-\beta t} \\ \frac{\partial^2}{\partial X^2} f(t, X) &= 0 \end{aligned}$$

By plugging in the partial derivatives of $f(t, X)$ in Itô formula, we get the following result:

$$\begin{aligned} df(t, X_t) &= \frac{\partial}{\partial t} f(t, X_t) dt + \frac{\partial}{\partial X} f(t, X_t) dX_t + \frac{1}{2} \cdot \frac{\partial^2}{\partial X^2} f(t, X_t)(dX_t)^2 \\ &= (\alpha - \beta f(t, X_t))dt + \sigma e^{-\beta t} dX_t + 0 \\ &= (\alpha - \beta f(t, X_t))dt + \sigma e^{-\beta t} \cdot e^{\beta t} dW_t \\ &= (\alpha - \beta f(t, X_t))dt + \sigma dW_t \end{aligned}$$

According to the result received above it is clear that $f(t, X_t) \sim r_t$, as $f(t, X_t)$ satisfies the equation (1.7) for dr_t and $f(t = 0, X_t = X_0) = e^0 r_0 + \frac{\alpha}{\beta}(1 - 1) + 0 = r_0$. Thus, is is possible to conclude

that $f(t, X_t) = r_t, \forall t \geq 0$.

Next, let us derive the expectation function $m(t) = \mathbb{E}[r_t](t)$ using the explicit r_t process (1.8):

$$m(t) = \mathbb{E}[r_t](t) = \mathbb{E}\left[e^{-\beta t} r_0 + \frac{\alpha}{\beta}(1 - e^{-\beta t}) + \sigma e^{-\beta t} \int_0^t e^{\beta s} dW_s\right]$$
$$= e^{-\beta t} r_0 + \frac{\alpha}{\beta}(1 - e^{-\beta t}) + \sigma e^{-\beta t} \mathbb{E}\left[\int_0^t e^{\beta s} dW_s\right]$$

Denote $g(t) = \int_0^t e^{\beta s} dW_s$ and it is a martingale. Then, by martingale property, $\mathbb{E}[g(t)] = g(0) = \int_0^0 e^{\beta s} dW_s = 0$. Hence, we get the following $m(t)$ function equation:

$$m(t) = e^{-\beta t} r_0 + \frac{\alpha}{\beta}(1 - e^{-\beta t}) \tag{1.9}$$

Having an explicit expectation function $m(t)$ in (1.9), let us find what it converges to, as $t \to \infty$:

$$\lim_{t \to \infty} m(t) = \lim_{t \to \infty} e^{-\beta t} r_0 + \lim_{t \to \infty} \frac{\alpha}{\beta}(1 - e^{-\beta t}) = 0 + \frac{\alpha}{\beta}(1 - 0) = \frac{\alpha}{\beta} \tag{1.10}$$

The fact, illustrated in (1.10) is known as mean-reversion. In Vasiĉek model (1.7) the ratio of parameters $\frac{\alpha}{\beta}$ shows the long-term risk-free interest rate, around which short-term rates fluctuate, and to which r_t converges as $t \to \infty$. This result is quite intuitive, since when $r_t > \frac{\alpha}{\beta}$, current interest rate is higher than the long-run average one, and an adverse trend change in the part of dt pulls the rate towards its mean; oppositely, when $r_t < \frac{\alpha}{\beta}$, there is a positive pressure on r_t, stimulating it to increase; and only when $r_t = \frac{\alpha}{\beta}$ corresponds to the steady state.

A more complicated version of model (1.7) is proposed by Hull J., and White A. in their paper [8]. What they propose is to consider the same dependence of interest rate with itself, trend and noise, but taking all parameters as stochastic. Eventually, the model is

$$dr_t = \alpha_t(\gamma(t) - r_t)dt + \rho_t dW_t \tag{1.11}$$

Where $\gamma(t)$ is an unknown, but non-random function of t. It is calibrated in such a way that the real yield curve and the curve that is the output of this model coincide.

1.2 Overview of machine learning techniques, further used for parameters calibration and financial modelling

Most of machine learning techniques that are used in this paper are based on the problem of optimization. This section will be based on the book [13] where the discussed algorithms are covered more precisely.

1.2.1 Gradient descend

Gradient descend is a method that uses first-order optimization algorithm to determine the minimum of a function, and it is used if such value cannot be computed analytically. Intuitively, for a given explicit function one takes steps proportional to the negative of the gradient of the function at the initial point. If one first order derivative is negative, then to approach the local minimum this variable should be decreased, and the larger is the derivative value, the further it is from the point of local minimum. Hence, by substracting a gradient multiplied by some learning rate η from the initial arguments vector one would in a limit approach the point of minimum.

Mathematically speaking, firstly it is necessary to determine the function to be minimized. Let us take the loss function $\mathcal{L}(w)$ which shows the size of error resulted by an estimation of some values $y_i, i \in \{1, ..., N\}$ by estimators $\hat{y}_i(w_1, ..., w_k), i \in 1, ..., N, k \geq 1$. The most common loss function is a mean squared error (MSE) that shows the average quadratic bias of estimation. Finally, the problem can be written mathematically as:

$$\mathcal{L}(w_1, ..., w_k) = \frac{1}{N} \sum_{1}^{N} \left(y_i - \hat{y}_i(w_1, ..., w_k)\right)^2 \rightarrow \min_{w_1, ..., w_k} \tag{1.12}$$

The gradient of the loss funcion $\mathcal{L}(w)$ is

$$\nabla \mathcal{L} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial w_k} \end{pmatrix} \tag{1.13}$$

Where

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{1}{N} \sum_{1}^{N} \left(y_i - \hat{y}_i(w_1, ..., w_k)\right) \cdot 2 \cdot \hat{y}_i, \forall i \in \{1, ..., k\} \tag{1.14}$$

To update the parameters $\left(w_1^j, ..., w_k^j\right)$ one should use the rule:

$$\begin{pmatrix} w_1^j \\ \vdots \\ w_k^j \end{pmatrix} = \begin{pmatrix} w_1^{j-1} \\ \vdots \\ w_k^{j-1} \end{pmatrix} - \eta \cdot \nabla \mathcal{L}(w_1^{j-1}, ..., w_l^{j-1}) \tag{1.15}$$

Finally, the sequence $\mathcal{L}(w_1^0, ..., w_l^0) \geq \mathcal{L}(w_1^1, ..., w_l^1) \geq ...$ in a limit would approach the point of local minimum.

1.2.2 Monte Carlo Method

This paper also uses Monte Carlo method as a means of values estimation. The theoretical background of Monte Carlo method given in this section is based on the book [12]. The intuition behind this algorithm is that to estimate some random variable value knowing its distribution but being unable to do it analytically due to the complexity of the problem, one may use random simulations of a random path (which can be a stochastic process, also called as Markov Chain in machine learning books), evaluate the mean value of the needed variable or construct a confidence interval, using it as a predicting interval of the estimated variable.

Mathematically speaking, let X be a random variable for a finite probability space $\Omega = \{x_1, ..., x_n\}$ with initial distribution $\mathbb{P}^\lambda : P(X_1 = x_i) = \lambda_i, i = \{1, ..., n\}$. In terms of this paper I will use only Ordinary Monte Carlo (OMC), considering $X_1, X_2, ...$ as independent identically distributed (i.i.d.) random variables. Assume that we want to compute the expectation

$$\mu = \mathbb{E}[g(X)]$$

where $g(\cdot)$ is a real-valued function on Ω, but it cannot be done explicitly. Instead, suppose that we can simulate $X_1, X_2, ...$ as i.i.d. by using the same distribution as X has. Then denote

$$\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} g(X_i)$$

By denoting $Y_i = g(X_i)$ we get i.i.d. Y_i with mean μ and variance

$$\sigma^2 = Var(g(x))$$

and $\hat{\mu}_n$ now is the sample mean of Y_i. By the Law of Large Numbers (LLN) $\hat{\mu}_n \to \mu$ as $n \to \infty$. Finally, by the Central Limit Theorem (CLT):

$$\hat{\mu}_n \sim \mathcal{N}\left(\mu, \frac{\sigma^2}{n}\right)$$

where the variance $\frac{\sigma^2}{n}$ can be estimated by

$$\hat{\sigma}_n = \frac{1}{n}\sum_{i=1}^{n}(g(X_i) - \hat{\mu}_n)^2$$

Note that OMC uses basic statistics concepts, for instance one can also use $\hat{\mu}_n \pm 1.96 \times \frac{\hat{\sigma}_n}{\sqrt{n}}$ as an asymptotic confidence interval for μ. Also note that the same logic may be applied for other parameters, the only requirement is the knowledge on variable's distribution.

Chapter 2

Methodology: algorithms discussion

2.1 Technique used to model stochastic differential equations

Modelling stochastic differential equations is a tricky task. Stochastic calculus works with continuous variables, whereas all data that can be collected in real life (even if the period of measurement is extremely low) is discrete. From calculus we know that for an arbitrary function $f(x_1, ..., x_n)$ its differential df can approximate the change of a function with respect to a small change in variables Δf: $df = \frac{\partial f}{\partial x_1}dx_1 + ... + \frac{\partial f}{\partial x_n}dx_n \approx \frac{\partial f}{\partial x_1}\Delta x_1 + ... + \frac{\partial f}{\partial x_n}\Delta x_n \approx \Delta f$. And it is true that $\Delta x_n \to dx_n$ as $\forall i : x_i \to 0$. The same logic is applied to approximation of a stochastic differential equation.

Consider a standard random process $(X_t)_{t\geq 0}$ which satisfies the following stochastic differential equation:

$$dX_t = \mu dt + \sigma dW_t \tag{2.1}$$

In (2.1) $(W_t)_{t\geq 0}$ is a Wiener process: $\forall (t_1, ..., t_n) : W_{t_n} - W_{t_{n-1}}, W_{t_{n-1}} - W_{t_{n-2}}, ..., W_{t_2} - W_{t_1}, W_{t_1}$ are mutually independent and $(W_t - W_s) \sim \mathcal{N}(0, t-s)$ for $t > s \geq 0$.

Approximation of the equation (2.1) would result in $\Delta X_t \approx \mu\Delta t + \sigma\Delta W_t$. Hence, to simulate the SDE we firstly have to divide the interval $[0, T]$ into n intervals by boundary points $t_i = i\Delta t$, where denote $\Delta t = \frac{T}{n}$. Secondly, we generate n independent identically distributed random variables $\xi_1, ..., \xi_t \sim \mathcal{N}(0, \Delta t)$ which correspond to the increments of W_t: by definition $\Delta W_t = W_t - W_{t-\Delta t} \sim \mathcal{N}(0, \Delta t)$. Finally, X_{t_i} can be computed by:

$$\begin{cases} X_{t_0} = x_0 \\ X_{t_{i+1}} - X_{t_i} = \mu\Delta t + \sigma\xi_{t_{i+1}}. \end{cases} \tag{2.2}$$

what is the difference scheme, logic of which is covered in chapter 1. This is the key concept that is used in further programming of stochastic differential equations.

2.2 Modelling Merton risk-free rate SDE (1.5) using Python Classes

This section presents an initial setup and characteristics of the procedures that are used in my algorithm. The only data one needs to collect is time series data of a risk-free interest rate for a certain maturity: $(r_1, ..., r_T)$. Note that this section includes the description of main parts of the code, for the full version of the code please refer to the appendix. To start work with the algorithm, one has to set up the class using the following command:

```
rfRateMerton(train_sample, n_paths, train_random_vars)
```

Where `train_sample` is the time series data on risk-free interest rate $(r_1, ..., r_T)$, `n_paths` (denote as n) is the number of paths to be generated in Monte-Carlo procedure, `train_random_vars` is a matrix $[T \times n]$ of independent identically distributed random variables ξ_{ij}. These random variables are fixed because in several iterations the results must be comparable: SDE is run on the same dataset and has identical inputs.

As it is stated in the beginning of this chapter, programming of stochastic differential equations would be based on finite difference method, involving the application of difference schemes. In a nutshell, this method proposes to use finite differences for solving differential equations. The domain and time interval are discretized and partial differential equations (PDEs) can be solved by computing values at the discrete nearby points.

Denote $(r_t)_{T \leq t \geq 0}$ as Merton risk free interest rate random process. Denote $L[r]$ as a differential operator such that $h \cdot L[r]$ is the differential to be approximated on the grid $\overline{\omega_h} = \{t_i = ih : i = 0, ..., n; nh = T\}$ at a point $t_0 \in \overline{\omega_h}$. Denote $h \cdot L_h^+[r] = r_{t_0+h} - r_{t_0}$ as a right hand differential, which we will use as a discrete approximation of an SDE in our case. From there we get a finite difference

$$r_{t_0+h} - r_{t_0} = h \cdot L_h^+[r]$$

Where $h = \Delta t$, $\Delta r_{t_0} = r_{t_0+h} - r_{t_0}$. Therefore, we get the following difference scheme for r_t:

$$\begin{cases} r_{t+h} - r_t = \alpha h + \beta(W_{t+h} - W_t) \\ r_0 = r \in const \end{cases} \tag{2.3}$$

Note that $W_{t+h} - W_t = \xi_{t+h} \sim \mathcal{N}(0, h)$ by definition stated in section 1.1, thus the difference scheme can finally be written as:

$$\begin{cases} r_{t+h} - r_t = \alpha h + \beta \xi_{t+h} \\ r_0 = r \in const \end{cases} \tag{2.4}$$

This discrete approximation of a theoretically continuous SDE can be performed by calling the Python class:

```
rfRateMerton.simulate(params, first_value, size)
```

Where `params` are parameters (α, β) in model (1.5), `first_value` is the initial value r_0 from which simulation procedure starts, `size` is the length of a simulated path (previously denoted as T).

For proper forecasting of future interest rates, we also need to estimate parameters α and β. Theoretical (which I denote as naive) estimates of these parameters are:

$$\hat{\alpha}_{naive} = \frac{\sum_{i=1}^{T}(t_i - \bar{t})(r_i - \bar{r})}{\sum_{i=1}^{T}(t_i - \bar{t})^2} \text{ – ordinary least squares (OLS) regression of rate on time,} \tag{2.5}$$

$$\hat{\beta}_{naive} = \sqrt{Var(r)} \text{– rate standatd deviation.} \tag{2.6}$$

This procedure is performed by the function

```
rfRateMerton.naiveEstimates()
```

Having naive estimates as the baseline of the parameters, we can proceed to calibration. Firstly, we need to understand what will be the metric that would measure the performance of the model, what is called as the loss function. In this problem I will use MSE (mean squared error), similar to the one stated in (1.12), which I will minimize in order to improve the algorithm performance. Denote MSE as:

$$MSE(\alpha, \beta) = \frac{1}{n}\sum_{i=1}^{n}\left[\frac{1}{T}\sum_{t=2}^{T}(r_{t-1} + \alpha\Delta t + \beta\xi_{ti} - r_t)^2\right] \tag{2.7}$$

Naive approach would be to define ranges $[\hat{\alpha}_{naive} - s.e.(\hat{\alpha}), \hat{\alpha}_{naive} + s.e.(\hat{\alpha})]$, $[\hat{\beta}_{naive} - s.e.(\hat{\alpha}), \hat{\beta}_{naive} + s.e.(\hat{\alpha})]$ [1] of a certain length, and in a `for` loop find such values of $(\hat{\alpha}, \hat{\beta})$ that minimize MSE function (2.7). This algorithm has quadratic complexity $O(n^2)$ and for large n and T it would require much time. A more elegant method is to use gradient descend algorithm described in section 1.2.1.

Denote MSE gradient function as:

$$\nabla MSE(\alpha, \beta) = \begin{pmatrix} \frac{\partial MSE}{\partial \alpha} \\ \frac{\partial MSE}{\partial \beta} \end{pmatrix}, \tag{2.8}$$

$$\frac{\partial MSE}{\partial \alpha} = \frac{1}{Tn} \sum_{i=1}^{n} \sum_{t=1}^{T} (r_{t-1} + \alpha \Delta t + \beta \xi_{ti} - r_t) \cdot 2\Delta t,$$

$$\frac{\partial MSE}{\partial \beta} = \frac{1}{Tn} \sum_{i=1}^{n} \sum_{t=1}^{T} (r_{t-1} + \alpha \Delta t + \beta \xi_{ti} - r_t) \cdot 2\xi_{ti}.$$

Thus, by taking $(\hat{\alpha}_{naive}, \hat{\beta}_{naive})$ as an initial guess and by then updating each generation of parameters estimators (similar to the rule (1.15)):

$$\begin{pmatrix} \hat{\alpha}_i \\ \hat{\beta}_i \end{pmatrix} = \begin{pmatrix} \hat{\alpha}_{i-1} \\ \hat{\beta}_{i-1} \end{pmatrix} - \eta \times \nabla MSE(\hat{\alpha}_{i-1}, \hat{\beta}_{i-1}) \tag{2.9}$$

Thus, by iterative simulations of K parameters estimators, the final results would converge to the actual minima, and this algorithm has constant complexity $O(K)$, what is a comparative advantage over a naive algorithm. This calibration procedure is presented in the function

```
rfRateMerton.calibrate(learning_rate, n_calibrations)
```

Where `learning_rate` is the learning rate variable η from (2.9), `n_calibrations` is the number of parameters' generations to be computed, which was previously denoted as K.

As I have derived theoretically and proved practically, for a large number of iterations parameters $(\hat{\alpha}_{naive}, \hat{\beta}_{naive})$ converge to such values that the predicted random paths coincide with an ordinary least squares (OLS) regression line, where interest rate is regressed on time. The main reason is that MSE minimization problem results in OLS regression. Let us verify this mathematically. The problem we solve is

$$MSE(\alpha, \beta) \to \min_{\alpha, \beta}. \tag{2.10}$$

The first order conditions (FOCs) are:

$$\begin{cases} \frac{1}{TN} \sum_{i=1}^{N} \sum_{t=1}^{T} (r_{t-1} + \alpha \Delta t + \beta \xi_{ti} - r_t) \cdot 2\xi_{ti} = 0 \\ \frac{1}{TN} \sum_{i=1}^{N} \sum_{t=1}^{T} (r_{t-1} + \alpha \Delta t + \beta \xi_{ti} - r_t) \cdot 2\Delta t = 0 \end{cases}.$$

From the second FOC we get [2]

[1] $s.e.(\hat{\theta})$ is the standard error of $\hat{\theta}$ which estimates θ, $s.e.(\hat{\theta}) = \sqrt{\frac{\sum_{i=1}^{n} (\hat{\theta}_i - \theta)}{n-1}}$

[2] $\overline{X}$ denotes the sample mean of $(X_1, ..., X_n)$

$$\frac{1}{TN}\sum_{i=0}^{N}\sum_{t=1}^{T}(r_{t-1}-r_t)\Delta t+\alpha(\Delta t)^2+\beta\times\Delta t\times\overline{\xi}=0$$
$$-\frac{(r_T-r_0)\times\Delta t}{T}\Delta t+\alpha(\Delta t)^2+\beta\times\Delta t\times\overline{\xi}=0$$
$$-\frac{r_T-r_0}{T}+\alpha+\frac{\beta\overline{\xi}}{\Delta t}=0$$
$$\hat{\alpha}=-\overline{\xi}\frac{\beta}{\Delta t}+\frac{r_T-r_0}{T}$$

In a probability limit[3], $\overline{\xi}\xrightarrow{\mathbb{P}}0$ as $n\to\infty$ since $\mathbb{E}[\xi]=0$, thus, in a probability limit estimator of α converges to the following:

$$\hat{\alpha}\xrightarrow[n\to\infty]{\mathbb{P}}\frac{r_T-r_0}{T-0} \tag{2.11}$$

The result is quite logical, because to minimize MSE the deviation from the trend must be as small as possible. Thus, it is most optimal to follow the overall interest rate trend which is the increment of rate with respect to the increment of time.

From the first FOC we get

$$\overline{r\xi}+\alpha\times\Delta t\times\overline{\xi}+\beta\overline{\xi^2}-\overline{r\xi}=0$$
$$\hat{\beta}=-\frac{\overline{\xi}\hat{\alpha}\Delta t}{\overline{\xi^2}}$$

By Slutsky Theorem [4], note that $\overline{\xi}\xrightarrow{\mathbb{P}}\mathbb{E}[\xi]=0$; $\hat{\alpha}\xrightarrow{\mathbb{P}}\alpha\in const\neq 0$; $\Delta t\to 0$; $\overline{\xi^2}\xrightarrow{\mathbb{P}}\mathbb{E}[\xi^2]\in const\neq 0$, thus

$$\hat{\beta}=-\frac{\overline{\xi}\hat{\alpha}\Delta t}{\overline{\xi^2}}\xrightarrow{\mathbb{P}}\frac{0\cdot\alpha\cdot 0}{\mathbb{E}[\xi^2]}=0 \tag{2.12}$$

Hence, the parameter $\hat{\beta}$ is zero almost surely (a.s.), what is, again, rather logical, since to minimize the mean squared error we have to minimize the spread of estimated values, i.e. make the diffusion term as least significant as possible. This can be done by setting it close to zero.

Mathematically, $\hat{\beta}$ converges to zero in probability limit, resulting degeneration of 'spread' part of the SDE. In order to fix this overeducation problem, I propose the following algorithm. Using estimated parameters I run Monte-Carlo simulations and generate random paths of interest rate for a train sample, receiving

$$\begin{bmatrix}\hat{r}_{t_1,1} & \hat{r}_{t_2,1} & \dots & \hat{r}_{t_T,1}\\ \hat{r}_{t_1,2} & \hat{r}_{t_2,2} & \dots & \hat{r}_{t_T,2}\\ \vdots & \vdots & \ddots & \vdots\\ \hat{r}_{t_1,N} & \hat{r}_{t_2,N} & \dots & \hat{r}_{t_T,N}\end{bmatrix}.$$

Then, by splitting this matrix into vectors such that each vector corresponds to a cross section for a certain time moment and sort interest rates, we will get:

[3] By definition, a sequence X_n converges towards the random variable X if for $\forall\varepsilon>0$ $\lim\limits_{n\to\infty}\mathbf{P}(|X_n-X|>\varepsilon)=0$.

[4] Slutsky's Theorem says that if $X_n\xrightarrow{\mathbb{P}}X$ (converges in probability) and $Y_n\xrightarrow{\mathbb{P}}Y$, then $X_n\cdot Y_n\xrightarrow{\mathbb{P}}X\cdot Y$ and $\frac{X_n}{Y_n}\xrightarrow{\mathbb{P}}\frac{X}{Y}$

$$\begin{pmatrix} \hat{r}_{t_1,(1)} \\ \hat{r}_{t_1,(2)} \\ \dots \\ \hat{r}_{t_1,(N)} \end{pmatrix}, \begin{pmatrix} \hat{r}_{t_2,(1)} \\ \hat{r}_{t_2,(2)} \\ \dots \\ \hat{r}_{t_2,(N)} \end{pmatrix}, ..., \begin{pmatrix} \hat{r}_{t_T,(1)} \\ \hat{r}_{t_T,(2)} \\ \dots \\ \hat{r}_{t_T,(N)} \end{pmatrix}$$

Thus, for any given time moment $t \in [0, T]$ a 95% confidence interval for interest rate r_t is

$$[\hat{r}_{t,([0.025N])}, \hat{r}_{t,([0.975N])}]. \tag{2.13}$$

As long as parameters evolve, they converge to the point of minimum and this interval converges to zero. Thus, for each generation of parameters I check whether there exists at least one time moment, when actual value is not covered by the confidence interval. If it is true, then calibration stops and the last values of parameters which correspond to 100% of CI accuracy are outputted as calibrated ones. Mathematically speaking, the loop stops if

$$inf\left\{t \geq 0 : r_t \notin (\hat{r}_{t,([0.025N])}, \hat{r}_{t,([0.975N])})\right\} \leq T. \tag{2.14}$$

Also, the calibration algorithm can be illustrated by figure 2.1:

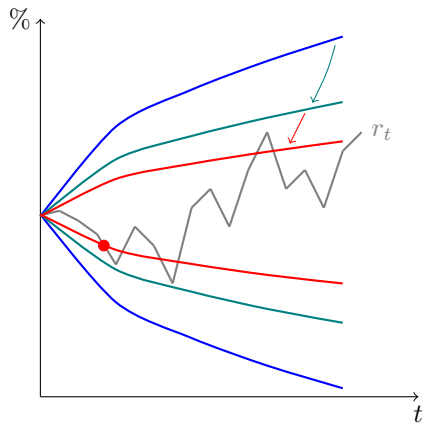


Figure 2.1: Contraction of estimating interval as a result of calibration procedure.

Note that in general, naive parameters perform worse than the ones received as a result of calibration. So, the intervals constructed with naive parameters would be wide, with enough space for contraction. However, if the parameters are such that the calibration loop does not even start, the user has an opportunity to choose another naive estimates and normallum calibrate the parameters.

This rule is built in the function `rfRateMerton.calibrate(learning_rate, n_calibrations)` and does not require the user do any additional manipulations in order to receive proper estimations.

Eventually, the key function that is implemented in the class `rfRateMerton` is the function that estimates future interest rates and outputs Monte-Carlo paths and 95% confidence intervals (2.13) for each of the time moments t. The function can be called by

```
rfRateMerton.estimateFutureRates(learning_rate, n_calibrations, size)
```

It inputs `learning_rate` whcih is the learning rate previously denoted as η; `n_calibrations` which is the number of (α, β) gradient descend generations to be computed, which was previously denoted as K; `size` which is the length of each generated path, previously denoted as T.

Application of class `rfRateMerton` as well as all its methods listed above to real data is characterized in chapter 3. It will also discuss the shortcomings of this algorithm and its comparative advantages over other, less mathematically complicated estimation algorithms (i.e. baseline models).

2.3 Modelling Vasiček risk-free rate SDE (1.7) using Python Classes

This section provides an overview on algorithm used to calibrate parameters of Vasiček risk-free interest rate model and use the results to forecast future interest rates. The general setup is quite similar to the one described in section 2.2 for Merton model. The main difference of Vasiček model from Merton model is that it has mean-reversion (see (1.10)), i.e. its drift term is not fixed. To calibrate parameters I will also use the method of objective loss function minimization, however, I will approach it in a different way.

Similarly to Merton, Vasiček class is a part of a more general `rfRate` class which inputs `train_sample` denoted as $(r_{t_1}, ..., r_{t_T})$, `n_paths` denoted as n, `train_random_vars` denoted as Ξ (3.1). The logic behind this setup is that a user does not have to make any extra actions to switch from one model to another one.

Class Vasiček is denoted as `rfRateVasicek(train_sample, n_paths, train_random_vars)`. To calibrate parameters α, β, ν from (1.7), I use MSE minimization approach, which is:

$$MSE(\alpha, \beta, \nu) = \frac{1}{N}\sum_{i=1}^{n}\frac{1}{T}\sum_{t=1}^{T}(r_{t-1} + (\alpha - \beta r_{t-1})\Delta t + \nu\xi_{t,i} - r_t)^2 \to \min_{\alpha,\beta,\nu} \tag{2.15}$$

The resulting first order conditions are:

$$\begin{cases} \frac{\partial MSE}{\partial \alpha} = \frac{1}{NT}\sum_{i=1}^{n}\sum_{t=1}^{T}(r_{t-1} + (\alpha - \beta r_{t-1})\Delta t + \nu\xi_{t,i} - r_t) \times 2\Delta t = 0 \\ \frac{\partial MSE}{\partial \beta} = \frac{1}{NT}\sum_{i=1}^{n}\sum_{t=1}^{T}(r_{t-1} + (\alpha - \beta r_{t-1})\Delta t + \nu\xi_{t,i} - r_t) \times 2r_{t-1}\Delta t = 0 \\ \frac{\partial MSE}{\partial \nu} = \frac{1}{NT}\sum_{i=1}^{n}\sum_{t=1}^{T}(r_{t-1} + (\alpha - \beta r_{t-1})\Delta t + \nu\xi_{t,i} - r_t) \times 2\xi_{r,i} = 0 \end{cases} \tag{2.16}$$

From the last condition we get

$$\frac{1}{NT}\sum_{i=1}^{n}\sum_{t=1}^{T}(r_{t-1} - r_t)\xi_{t,i} + \alpha\Delta t\bar{\xi} - \beta\overline{r\xi}\Delta t + \nu\bar{\xi^2} = 0$$

From where we get the explicit formula for $\hat{\nu}$ estimator:

$$\hat{\nu} = \frac{\hat{\beta}\overline{r\xi} - \overline{\xi\Delta r} - \hat{\alpha}\bar{\xi}\Delta t}{\overline{\xi^2}} \tag{2.17}$$

Which, by Slutsky theorem (refer to section 2.2), in probability limit will be

$$\begin{aligned} \operatorname*{plim}_{n\to\infty}\hat{\nu} &= \operatorname*{plim}_{n\to\infty}\left(\frac{\hat{\beta}\overline{r\xi} - \overline{\xi\Delta r} - \hat{\alpha}\bar{\xi}\Delta t}{\overline{\xi^2}}\right) \\ &= \frac{\operatorname*{plim}_{n\to\infty}(\hat{\beta}) \times \operatorname*{plim}_{n\to\infty}(\overline{r\xi}) - \operatorname*{plim}_{n\to\infty}(\overline{\xi\Delta r}) - \operatorname*{plim}_{n\to\infty}(\hat{\alpha}) \times \Delta t \times \operatorname*{plim}_{n\to\infty}(\bar{\xi})}{\operatorname*{plim}_{n\to\infty}(\overline{\xi^2})} \\ &= \frac{\beta \times 0 - 0 - \alpha\Delta t \times 0}{Var(\xi)} = 0 \end{aligned}$$

The result is quite intuitive, because as we minimize the mean squared error, the shorter becomes the distance between the prediction and the trend line. Thus, as the number of calibrations increases, the forecasting intervals degenerate into points which coincide with the trend, i.e. the diffusion term νdW_t degenerates into zero, being consistent with Merton model. To prevent this phenomenon of

overlearning, I use the same algorithm of its detection similar to the one in Merton class described in section 2.2.

As for the parameters α and β, their fraction $\frac{\alpha}{\beta}$ converges to the non-zero expectation of interest rate $\mathbb{E}[r_t]$ as it has been proved in (1.10).

For calibration of α, β, ν one may use the function

```
rfRateVasicek.calibrate(learning_rate, n_calibrations, mode)
```

Which imputs the same parameters as the one of Merton, and one additional parameter `mode` which takes two values `['G', 'M']`, where `'G'` stands for gradient descend method calibration, and `'M'` for an explicit minimization of objective function method. On the grounds of performance tests run, the preferable calibration method is the latter one `'M'`. The main idea of this algorithm is that it generates naive estimators $(\hat{\alpha}_{naive}, \hat{\beta}_{naive}, \hat{\nu}_{naive})$ using `rfRateVasicek.naiveEstimates()` function, takes intervals $(\hat{\alpha}_{naive} \pm \delta, \hat{\beta}_{naive} \pm \delta, \hat{\nu}_{naive} \pm \delta)$ in the neighbourhood of these naive estimates with a range of 2δ, and for these intervals it finds such combination $(\hat{\alpha}_i, \hat{\beta}_j, \hat{\nu}_k)$ that minimizes the function $MSE(\alpha, \beta, \nu)$.

Function `rfRateVasicek.calibrate` is used in function

```
rfRateVasicek.estimateFutureRates(learning_rate, n_calibrations,
                                  size, mode)
```

which inputs same parameters as the same function of Merton does, again with the addition of a parameter `mode` which has the same purpose as in `rfRateVasicek.calibrate`. It outputs a list of n random paths with size T, upper and lower boundaries of 95% confidence intervals for each time moment t:

$$\left\{ \begin{bmatrix} r_{1,t_1} & r_{1,t_2} & \cdots & r_{1,t_N} \\ r_{2,t_1} & r_{2,t_2} & \cdots & r_{2,t_N} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n,t_1} & r_{n,t_2} & \cdots & r_{n,t_N} \end{bmatrix}, \begin{bmatrix} r_{([0.025\times n]),t_1} \\ r_{([0.025\times n]),t_1} \\ \vdots \\ r_{([0.025\times n]),t_1} \end{bmatrix}, \begin{bmatrix} r_{([0.975\times n]),t_1} \\ r_{([0.975\times n]),t_1} \\ \vdots \\ r_{([0.975\times n]),t_1} \end{bmatrix} \right\}. \tag{2.18}$$

Chapter 3

Empirical application of developed classes to real data

This part of the paper is dedicated to the application of the classes developed in the previous chapter to real data and evaluation of the algorithms performance. The data that I will use is time series data of U.S. Treasury rates[1]. It includes 250 observations on daily interest rates from Jan 2^{nd}, 2019 until Dec 31^{st}, 2019 and covers a range of 12 maturities from 1 Mo to 30 years. The sample of the data is presented in the table below:

date	1 Mo	2 Mo	$\cdots$	20 Yr	30 Yr
2019-01-02	2.40	2.40	$\cdots$	2.83	2.97
2019-01-03	2.42	2.42	$\cdots$	2.75	2.92
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
2019-12-31	1.48	1.51	$\cdots$	2.25	2.39

Table 3.1: Dataset structure.

3.1 Application of Merton model

In this section I test whether the model of R. C. Merton (1.5) is applicable to the dataset illustrated in table 3.1. Firstly, I select 1 Mo rates from Jun 11^{th} 2019 until Nov 1^{st}, 2019 as a train sample $(r_1, ..., r_T)$, , and rates from, Nov 1^{st}, 2019 until Dec 31^{st} as a test sample $r_{T+1}, ..., r_{T+Q}$.

To estimate future rates, it is firstly needed to simulate a matrix of iid variables:

$$\Xi = \begin{bmatrix} \xi_{1,1} & \xi_{1,2} & \cdots & \xi_{1,T} \\ \xi_{2,1} & \xi_{2,2} & \cdots & \xi_{2,T} \\ \vdots & \vdots & \ddots & \vdots \\ \xi_{N,1} & \xi_{N,2} & \cdots & \xi_{N,T} \end{bmatrix}. \tag{3.1}$$

Where N is the number of random paths to be generated in an algorithm, in code denoted as `n_paths`; T is the length of a train sample; and $\xi_{i,j} \overset{i.i.d.}{\sim} \mathcal{N}(0, \Delta t)$ (refer to (2.4)). As observations are daily, $\Delta t = 1$.

Now we are set to execute a class, using the command:

[1] https://home.treasury.gov/policy-issues/financing-the-government/interest-rate-statistics

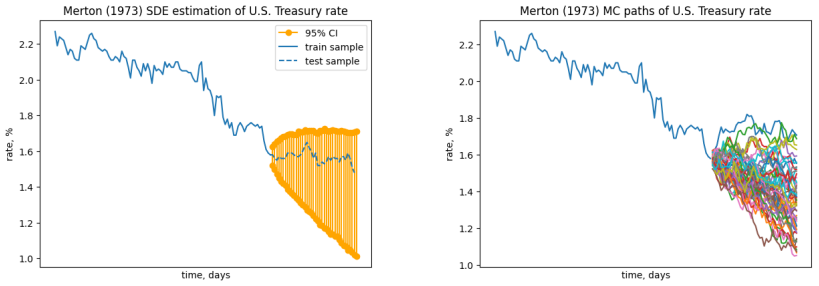
```
rfRateMerton(train_sample, n_paths=1000, random_vars)
```

Where `train_sample` is train set $(r_1, ..., r_T)$, `n_paths` is manually set as 1000, and `random_vars` is Ξ. Then, we can call the method `estimateFutureRates` to estimate values of test sample using the command:

```
rfRateMerton.estimateFutureRates(learning_rate=0.01,
                                 n_calibrations=100,
                                 size)
```

Where `learning_rate` denoted previously as η is set manually as 0.01, `n_calibrations` is the number of gradient descend adjustments iterations (1.15) to be performed that is set manually to 100, `size` is the length of test sample that in this section is denoted as Q.

The output is a list of forecasted interest rate random paths and confidence intervals for each Δt (each day in this case). Resulted Monte-Carlo paths and 95% confidence intervals are presented in the figures below:



(a) 95% confidence intervals of r_t test sample

(b) Paths of test sample r_t generated by using *Monte-Carlo* method

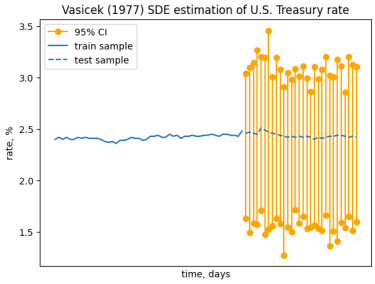
Figure 3.1: Results of Merton model algorithm execution

It can be seen in figure 3.1 that the confidence interval covers interest rate at all time moments what shows that the developed algorithm works correctly and the parameters are properly calibrated. However, the intervals become quite large as long as the time passes (approximately 70 b.p. on the 40^{th} estimated day), hence, in order to get more precise results an algorithm must be adjusted.

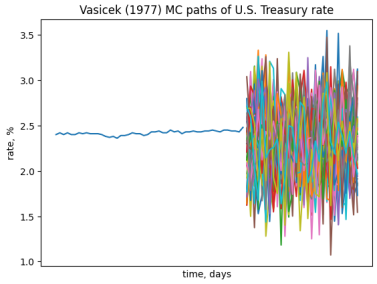
3.2 Application of Vasiček model

This section provides a practical application of the Vasiček model (1.7) algorithm described in section 2.3. Similarly to the Merton algorithm usage, I select a sample of 1 Mo U.S. Treasury rates described in table 3.1, take rates from Jan 2^{nd}, 2019 until March 15^{th}, 2019 as a train sample, ones from March 18^{th}, 2019 until April 29^{th}, 2019 as a test sample. Also, I use the same random variables $\xi_{i,j}$ described in (3.1) as `random_vars`.

The results of an algorithm are presented in the figure 3.2. Empirically, it is clear that the algorithm works correctly, and parameters α, β, ν from (1.7) are calibrated properly. However, the main drawback of this model is that it has mean-reversion (1.10), thus the estimated paths of future interest rates always return to the mean value of the train sample interest rate $\bar{r}_{train} = \frac{1}{T}(r_{t_1} + ... + r_{t_T})$. Additionally, due to the specifics of my algorithm, if train sample has a large range $(max\{r_{train}\} - min\{r_{train}\})$ then the spread of the confidence interval would be large as well (in this case, the spread of CI is nearly 1.5pp for each observation). The output can be called a corridor with mean $\frac{\hat{\alpha}}{\hat{\beta}}$ with some spread which is impacted by the diffusion term νdW_t.



(a) 95% confidence intervals of r_t test sample



(b) Paths of test sample r_t generated by using *Monte-Carlo* method

Figure 3.2: Results of Vasiček model algorithm execution

Chapter 4

Conclusion

To summarize, the proposed program that is described in this work uses difference schemes for discrete approximation of continuous stochastic differential equations. The developed models output mathematically strict and financially meaningful results, since in theory estimated parameters are consistent and converge to theoretically correct values, and financially the model correctly estimates future rates given only historical data. To be objective, let us compare the developed models of Merton (3.1) and Vasiček (1.7) with ordinary least squares (OLS) regression. The OLS has the form:

$$r(t) = \alpha + \beta t + \varepsilon_t$$

Where r is interest rate, t is time, α and β are estimated constant parameters, ε_t is an error term such that $\forall t$: ε_t - i.i.d and $\mathbb{E}[\varepsilon_t] = 0$. As the outputed values of my program are intervals, not points, I will use the means of intervals as an estimate in order to compare the predictive power of SDEs with OLS. The measure used to compute the predictive power of models is residual mean squared of errors (RMSE) which is computed as

$$RMSE = \frac{1}{n}\sum_{i=1}^{n}(X_i - \hat{X}_i)^2$$

where $\hat{X}_i$ is an estimator of X_i for $i \in \{1, ..., n\}$. The results are provided in table .

Metric	Merton SDE	Vasiček SDE	OLS Regression
RMSE	$907 \cdot 10^{-6}$	$752 \cdot 10^{-6}$	$1590 \cdot 10^{-6}$

Table 4.1: RMSE of several models for estimation of future risk-free interest rates. Train sample: Jun 11^{th}, 2019 – Nov 1^{st}, 2019; Test sample: Nov 1^{st}, 2019 – Dec 31^{st}, 2019

Thus, by RMSE measure, Vasiček and Merton models are best in predictive power with RMSE nearly two times less than the OLS. Note that Vasiček model performs better that Merton model, since it is more complex and differently approaches the modelling of drift term.

Finally, it is possible to conclude that the newly developed algorithm of parameters calibration works correct, and the programmed toolkit of Python classes properly estimates future interest rates. Thus, the program can be used in further research for building models with implied stochastic risk-free interest rates.

Bibliography

[1] Girsanov I.V. "On Transforming a Certain Class of Stochastic Processes by Absolutely Continuous Substitution of Measures". In: *Theory of Probability and Its Applications* 5.3 (1960), pp. 285–301. DOI: https://doi.org/10.1137/1105027.

[2] Sharpe W.F. "Capital Asset Prices: A Theory of Market Equilibrium under Conditions of Risk". In: *The Journal of Finance* 19.3 (1964), pp. 425–422. DOI: https://doi.org/10.2307/2977928.

[3] Scholes M. Black F. "The Pricing of Options and Corporate Liabilities". In: *Journal of Political Economy* 81.3 (1973), pp. 637–654. DOI: http://dx.doi.org/10.1086/260062.

[4] Merton R.C. "Theory of Rational Option Pricing". In: *The Bell Journal of Economics and Management Science* 4.1 (1973), pp. 141–183. DOI: http://dx.doi.org/10.2307/30031434.

[5] Ross S.A. "The arbitrage theory of capital asset pricing". In: *Journal of Economic Theory* 13.3 (1976), pp. 341–360. DOI: https://doi.org/10.1016/0022-0531(76)90046-6.

[6] Vasiček O. "An equilibrium characterization of the term structure". In: *Journal of Financial Economics* 5.2 (1977), pp. 177–188. DOI: https://doi.org/10.1016/0304-405X(77)90016-2.

[7] Ross S.A. "The Capital Asset Pricing Model (CAPM), Short-Sale Restrictions and Related Issues". In: *The Journal of Finance* 32.1 (1977), pp. 177–183. DOI: https://doi.org/10.2307/2326912.

[8] White A. Hull J. "Pricing Interest Rate Derivatives". In: *Review of Financial Studies* 3.5 (1990), pp. 573–592. DOI: https://doi.org/10.1093/rfs/3.4.573.

[9] Shiryaev A.N. *Principals of Stochastic Financial Mathematics (in Russian)*. Moscow, Russia: Fasis, 1998.

[10] Shreve S.E. *Stochastic Calculus for Finance*. Pittsburgh, PA, USA: Springer Science+ Business Media, Inc., 2000.

[11] Shiryaev A.N. Bulinskiy A.V. *Theory of Random Procceses*. Moscow, Russia: Phismatlit, 2005.

[12] Brooks S. *Handbook of Markov Chain Monte Carlo*. Boston, MA, USA: Chapman, Hall, CRC, Taylor, and Francis Group, 2011.

[13] Ong C.S. Deisenroth M.P. Faisal A.A. *Mathematics for Machine Learning*. Cambridge, UK: Cambridge Academ, 2020.

Annexes

Python class of Merton and Vasiček SDE models

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive
import scipy.special
from scipy.stats import norm
from scipy.stats import bernoulli
from scipy.stats import linregress
import statsmodels.api as sm
import sys
from statsmodels.tsa.ar_model import AutoReg


# Global class of interest rates. Then Vasicek and other models to be added there
class rfRate:
  #train_sample: sample on which model is educated, n_paths:
                                                    #number of simulations to be run,
  #train_random_vars: iid N(0,1) of size len(train_sample) x n_paths
  def __init__(self, train_sample, n_paths, train_random_vars):
    self.train_sample = train_sample
    self.n_paths = n_paths
    self.train_random_vars = train_random_vars


#Merton Class
class rfRateMerton(rfRate):
  def __init__(self, train_sample, n_paths, train_random_vars):
    super().__init__(train_sample, n_paths, train_random_vars)

  # Naive estimators of Merton Model
  def naiveEstimates(self):
    ans = linregress(x = range(0, len(self.train_sample)), y = self.train_sample)
          #alpha -- estimator is OLS slope
    alpha_naive = ans.slope

    beta_naive = np.std(self.train_sample)
          #beta -- estimator is std of sample

    return [alpha_naive, beta_naive]
```

```
# MSE of certain parameters
def __mse(self, parameters):
  mses = []
  for i in range(self.n_paths):
    path = [self.train_sample[0]]
    for j in range(1, len(self.train_sample)):
      path.append(path[j-1]+1*parameters[0]
                          +parameters[1]*self.train_random_vars[i][j])
    mses.append(((np.array(path) - np.array(self.train_sample))) ** 2)
  return np.mean(mses)

# Gradient of mse function at point (\hat{\alpha}, \hat{\beta})
def __grad(self, parameters):
  grad_alpha = 0
  grad_beta = 0

  for i in range(self.n_paths):
    path = [self.train_sample[0]]
    for j in range(1, len(self.train_sample)):
      path.append(path[j-1]+1*parameters[0]
              +parameters[1]*self.train_random_vars[i][j])
    # Partial derivative of mse wrt alpha
    grad_alpha += 2 * np.sum(np.array(path)
      - np.array(self.train_sample)) / (self.n_paths * len(self.train_sample))
    # Partial derivative of mse wrt beta
    grad_beta += 2 * np.sum((np.array(path)
      - np.array(self.train_sample)) * np.array(self.train_random_vars[i]))
          / (self.n_paths * len(self.train_sample))
  return np.array([grad_alpha, grad_beta])

# Computation of confidence interval on given paths
def __confidenceInterval(self, paths):
  CI_up = []
  CI_low = []
  for i in range(0, len(paths[0])):
    cross = []
    for j in range(0, len(paths)):
        cross.append(paths[j][i])
    cross.sort()
    CI_up.append(cross[int(0.975 * len(paths))])
    CI_low.append(cross[int(0.025 * len(paths))])

  return [CI_up, CI_low]

# Simulation of random paths
# params: alpha and beta, first_value: value at which we start modelling,
  # size: length of estimated interval
def simulate(self, params, first_value, size):
  paths = []
  for i in range(0, self.n_paths):
```

```
      np.random.seed(i)
      xis_ = np.random.normal(loc=0, scale=1, size=size)
      path = [first_value + params[0]*1 + params[1]*xis_[0]]
      [path.append(path[j-1] + params[0]*1 + params[1]*xis_[j]) for j in range(1, size)]
      paths.append(path)
    return paths

  # Calibration engine. Finds parameters using gradient descend
    #n_calibrations: number of iterations to be repeated
  def calibrate(self, learning_rate, n_calibrations):
    # firstly use naive estimators of parameters
    params_history = [rfRateMerton.naiveEstimates(self)]
    n = 0
    for i in range(1, n_calibrations):
      # update parameters by correcting them by their gradient times the learning rate
      params_updated = params_history[i-1]
        - learning_rate * rfRateMerton.__grad(self, params_history[i-1])
      cis = rfRateMerton.__confidenceInterval(self, rfRateMerton.simulate(self,
                                              params_updated, self.train_sample[0],
                                              len(self.train_sample)))
      # to prevent overlearning, detect when
            #the confidence interval no longer covers the actual path of the rate
      incorrect = 0
      for j in range(0, len(self.train_sample)):
        if self.train_sample[j] > cis[0][j] or self.train_sample[j] < cis[1][j]:
          incorrect += 1
          break
      if incorrect > 0: # outlier detected, stop calibration procedure
        n += 1
        break
      params_history.append(params_updated) #if outlier is not detected,
                    #the procedure repeats and parameters are updated
    return params_history[-1]

  # Estimation of future interest rates
  def estimateFutureRates(self, learning_rate, n_calibrations, size):
    # Find parameters using gradient descend calibration procedure
    params = rfRateMerton.calibrate(self, learning_rate=learning_rate,
                                    n_calibrations=n_calibrations)
    paths = []
    # Iterate Merton SDE to compute random paths of future interest rate
    for i in range(0, self.n_paths):
      np.random.seed(i)
      xis_ = np.random.normal(loc=0, scale=1, size=size)
      path = [self.train_sample[-1] + params[0]*1 + params[1]*xis_[0]]
      [path.append(path[j-1] + params[0]*1 + params[1]*xis_[j]) for j in range(1, size)]
      paths.append(path)

    # Return estimated random paths as well as confidence intervals
          #to estimate the performance of the model
    return [paths] + rfRateMerton.__confidenceInterval(self, paths)
```

```
# Vasicek Class
class rfRateVasicek(rfRate):
  def __init__(self, train_sample, n_paths, train_random_vars):
    super().__init__(train_sample, n_paths, train_random_vars)


  def naiveEstimates(self):
    # [A] TBD
    return [1,1,1]


  def __mse(self, paths_modeled, path_act):
    if len(paths_modeled[0]) != len(path_act):
      raise Exception('Lists must be of same length')


    mses = []
    for i in range(len(paths_modeled)):
      mses.append(np.sum((np.array(paths_modeled[i]-path_act))**2))
    return np.sum(mses) / (len(paths_modeled)*len(path_act))


  def __grad(self, parameters):
    grad_alpha, grad_beta, grad_eta = 0,0,0
    k=0
    for i in range(self.n_paths):
      k += 1
      path = [self.train_sample[0]]
      [path.append(path[j-1] + (parameters[0] - parameters[1]*path[j-1])*1
                             + parameters[2]*self.train_random_vars[i][j])
                                    for j in range(1, len(self.train_sample))]

      grad_alpha += 2 * np.sum(np.array(path) - np.array(self.train_sample)*1)
                                          / (self.n_paths * len(self.train_sample))
      grad_beta += 2 * np.sum((np.array(path) - self.train_sample)
            *np.array(self.train_sample)*1) / (self.n_paths * len(self.train_sample))
      grad_eta += 2 * np.sum((np.array(path)-self.train_sample)
                         *np.array(self.train_random_vars[i])*1)
                                / (self.n_paths * len(self.train_sample))
    return np.array([grad_alpha, grad_beta, grad_eta])


  def simulate(self, params, first_value, size):
    paths = []
    for i in range(0, self.n_paths):
      np.random.seed(i)
      xis_ = np.random.normal(loc=0, scale=1, size=size)
      path = [first_value + (params[0] - params[1]*first_value)*1 + params[2]*xis_[0]]
      [path.append(path[j-1] + (params[0] - params[1]*path[j-1])*1 + params[2]*xis_[j])
```

```
    paths.append(path)
  return paths


def __confidenceInterval(self, paths):
  CI_up = []
  CI_low = []
  for i in range(0, len(paths[0])):
    cross = []
    for j in range(0, len(paths)):
        cross.append(paths[j][i])
    cross.sort()
    CI_up.append(cross[int(0.975 * len(paths))])
    CI_low.append(cross[int(0.025 * len(paths))])
  return [CI_up, CI_low]


def calibrate(self, learning_rate, n_calibrations, mode):
      # Mode — what type of calibration to use
          #G-gradient descend, E-explicit mse function minimization
  if mode == 'G':
    params_history = [rfRateVasicek.naiveEstimates(self)]
    n = 0
    for i in range(1, n_calibrations):
      params_updated = params_history[i-1] - learning_rate
              * rfRateVasicek.__grad(self, parameters=params_history[i-1])
      cis = rfRateVasicek.__confidenceInterval(self,
                    rfRateVasicek.simulate(self, params_updated,
                                           self.train_sample[0],
                                           len(self.train_sample)))

      incorrect = 0
      for j in range(0, len(self.train_sample)):
        if self.train_sample[j] > cis[0][j] or self.train_sample[j] < cis[1][j]:
          incorrect += 1
          break
      if incorrect > 0:
        break
      params_history.append(params_updated)
    return params_history[-1]


  elif mode == 'E':
    [alpha, beta, eta] = rfRateVasicek.naiveEstimates(self)
    alphas = np.linspace(alpha-5, alpha+2, 20)
    betas = np.linspace(beta-5, beta+5, 20)
    etas = np.linspace(eta-5, eta+5, 20)

    paths_sim = rfRateVasicek.simulate(self, params=[alpha, beta, eta],
                                       first_value = self.train_sample[0],
                                       size = len(self.train_sample))
```

```
    mse_min = np.mean([(paths_sim[k] - self.train_sample) ** 2
                       for k in range(0, len(paths_sim))])

    import sys
    mse_min = sys.maxsize

    for a in alphas:
      for b in betas:
        for i in range(0, len(etas)):
          left -= 1
          paths_sim = rfRateVasicek.simulate(self, params=[a, b, etas[i]],
                                             first_value = self.train_sample[0],
                                             size = len(self.train_sample))
          mse = np.mean([(paths_sim[k] - self.train_sample) ** 2
                         for k in range(0, len(paths_sim))])

          if mse < mse_min:
            alpha, beta, eta = a, b, etas[i]
            mse_min = mse

    return [alpha, beta, eta]

def estimateFutureRates(self, learning_rate, n_calibrations, size, mode):
  params = rfRateVasicek.calibrate(self, learning_rate=learning_rate,
                                   n_calibrations=n_calibrations, mode=mode)
  paths = rfRateVasicek.simulate(self, params=params,
                                 first_value=self.train_sample[-1], size=size)
  return [paths] + rfRateVasicek.__confidenceInterval(self, paths)
```