WROCLAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ELECTRONICS

FIELD: AUTOMATICS AND ROBOTICS
SPECIALIZATION: INDUSTRY 4.0

ENGINEERING THESIS

Use of functional programming in modeling selected financial instruments

Wykorzystanie programowania funkcyjnego do wyceny wybranych instrumentów finansowych

AUTHOR:
Jakub Unold

SUPERVISOR:
Prof. Wojciech Bożejko, Ph.D., D.Sc., M.A.

WROCŁAW, 2020

Contents

Abstract

This paper is an overview of some methods used for pricing selected financial instruments (or their derivatives). The main part of the project was creating a tool - a computer program written mostly in F# for that purpose. The application is named *MARS App* – it is an acronym of the English words *Market And Risk Simulation*. Two models have been presented: first being Black-Scholes model from 1973 which gives an estimate of the price of a European-style option. As the underlying asset a stock price was used. In order to generate stock prices over time Geometric Brownion Motion has been implemented as this stochastic process is usually applied in the Black-Scholes model. The other model is Black's model from 1976 which is a slightly altered Black-Scholes model while it is adjusted for valuing options on futures contracts.

This paper can be as well a kind of a guide how to create an F# application using MVVM architecture with XAML markup language for creating user interface as such tutorials are scarce in literature.

Keywords: functional programming, financial instruments, pricing, MVVM, Black-Scholes, Black, Myron, Scholes, Geometric Brownian Motion, Random Walk, Standard Brownian Motion

Streszczenie

W prezentowanej pracy inżynierskiej przedstawiono proces tworzenia narzędzia do wyceny wybranych instrumentów finansowych. Zaprezentowano wycenę europejskiej opcji przy użyciu znanego modelu Blacka-Scholes'a z 1973 roku, gdzie instrumentem podstawowym jest akcja wygenerowana za pomocą Geometrycznego Ruchu Browna. Pokazano również zastosowanie modelu Black'a z 1976 roku, gdzie w przeciwieństwie do poprzedniego modelu cena instrumentu bazowego zostaje zastąpiona zdyskontowaną wartością kontraktu futures/forward na ten instrument.

W tym celu została stworzona aplikacja desktopowa w języku F#, w paradygmacie programowania funkcyjnego, przy użyciu wzorca MVVM oraz technologii XAML do stworzenia interfejsu użytkownika. Aplikacja nosi nazwę MARS App – jest to akronim od angielskiego *Market And Risk Simulation Application*, co oznacza aplikację przeznaczoną do symulacji rynku i ryzyka. Praca zawiera również swoisty poradnik, jak stworzyć aplikację w języku F#, opartą o architekturę MVVM w zintegrowanym środowisku programistycznym Visual Studio 2019 będącym produktem firmy Microsoft. Powyższy stos technologiczny nie jest często spotykany, co przejawia się zredukowaną ilością poradników pomagających stworzyć podobną aplikację nowym użytkownikom.

Słowa kluczowe: programowanie funkcyjne, instrumenty finansowe, wycena, MVVM, Black-Scholes, Black, Myron, Scholes, Geometryczny Ruch Browna, Błądzenie Losowe, Standardowy Ruch Browna

Chapter 1

Introduction

1.1. Preface

"We all have tremendous potential, and we all are blessed with gifts. Yet, the one thing that holds all of us back is some degree of self-doubt. It is not so much the lack of technical information that holds us back, but more the lack of self-confidence."

– Robert T. Kiyosaki, "Rich Dad, Poor Dad"

It is a quote from a brilliant book, written by Robert Kiyosaki – an American businessman who's net worth is estimated at $100 million, titled "Rich Dad, Poor Dad". Kiyosaki puts stress on how illiterate and uneducated the society is when it comes to, what he calls *financial intelligence* – skills and knowledge gained from understanding financial principles and ability to use them in everyday life.

Inspired by the message from the book and the originality of functional programming, as well as its applicability in modeling financial models, this paper is an attempt to bring some issues from the world of finance and mathematics closer. Since the term *Industry 4.0* is so broad and one of its components is widely understood automation – I hope the *MARS App* will equal to the task.

1.2. Thesis assumptions

The main goal of this work is to create a tool in the functional programming paradigm which will be used to price selected financial instruments. To achieve this goal F# programming language will be used as it was created mostly to support functional programming and a European option will be priced as an example of a derivative financial instrument. The app will consist of several different technologies such as MVVM design pattern, XAML language to create View part of the project and minor C# background to connect XAML's View with F#'s ViewModel. Everything will be written in Visual Studio 2019 IDE.

The basic assumptions within the application include:

- *Geometric Brownian Motion* implementation as a model used for generating underlying asset prices.
- Opportunity to specialize the product by changing:
  - maturity (expiration time),
  - interest rate,
  - stock's initial price,
  - drift,

  - volatility.
- Implementing Black-Scholes model for option pricing.
- Preparation of the application's graphic design in XAML and connecting the View with logic.

Further chapters consecutively cover above issues.

1.3. Scope of the thesis

The next sections in this chapter will bring closer the technologies used for the sake of the project: quick overview of the most common programming paradigms, introduction to *F#* and *XAML* languages and *MVVM* design pattern. The following chapter is divided into 3 sections – explanation of the *Geometric Brownian Motion*, introduction to financial options and the *Black-Scholes Model*. Last chapter contains summary of the project assumptions and several *future works* possibilities.

1.4. Technologies Overview

This section describes the most important topics of the technological stack of the project.

1.4.1. Programming paradigms

A programming paradigm is nothing more but an overall concept which describes *how* the programming is done and what is the methodology behind the language that adheres to a specific type of a paradigm. Throughout the years programming languages evolved, new ones have been created, so that today there are from 150, according to TIOBE Programming Community, up to 700, listed by Wikipedia, different programming languages (more information can be found here [6]). Although some sources, e.g. [9], state that there are almost as many as 9000 (sic) programming languages the exact number of those is not that important – rather the variety and the sheer number of them is crucial as it indicates that they must differ from each other and it turns out these differences can somehow be grouped, what is presented below.

The most basic and intuitive approach into dividing the paradigms into 2 basic groups seems to be this one:

- Imperative Programming,
- Declarative Programming.

The following sections present above paradigms.

Imperative Programming

This way of programming is not by accident described as the first one – historically languages that present this type of programming have emerged primarily. Imperative programming enables the programmer to manage processor's behavior on much more precise level. The commands show step-by-step how the computation is executed. These commands affect a program's state. This paradigm focuses on describing *how* the goal should be achieved.

Imperative programming can be further broken down into 3 groups:

1. **Procedural Programming** – the underlying model is based on a procedure (set of coded instructions, more in [10]). It has the ability to use once written code again. The examples of procedural programming examples include:
   - C,

- C++,
- Java,
- ColdFusion,
- Pascal.

2. **Object Oriented Programming** – a program is a set of classes and objects (instances of a class) that are meant to interact with each other. The examples of languages supporting OOP include:
   - Objective-C,
   - C++,
   - Java,
   - Visual Basic .NET,
   - Python.
3. **Parallel Processing Approach** – this style of programming is designed to divide computing the code into multiple processors in order to minimize the time required for computation. The examples of languages supporting parallel programming approach include:
   - NESL,
   - C,
   - C++.

Declarative Programming

On the other hand a programmer may want to focus not on *how* the code should be ran on a computer but rather what *what the result should look like* and the way of obtaining it is not the most important thing. One would rather give it up to the compiler to decide what is the best way of achieving the result. Declarative programming is also consisted of smaller groups, such as:

1. **Logic Programming** – in logic programming the programmer has knowledge, expressed in a logical form, about facts and rules concerning a specific problem, more details here [7]. This type of programming resembles a mathematical proof. **Prolog** language is an example of this programming paradigm.
2. **Database-Driven Programming Approach** – it is based on storing the data and moving it, as well as about keeping information about the relations among entities. **SQL** is one of the most popular examples of this programming group.
3. **Functional Programming** – like object in object-related programming in functional programming the most basic unit is a function. Shared state is avoided, as well as mutable data (the programmer is unable to crate a variable). The examples of functional programming paradigm:
   - F#,
   - Haskell,
   - Common Lisp,
   - OCaml,
   - Racket.

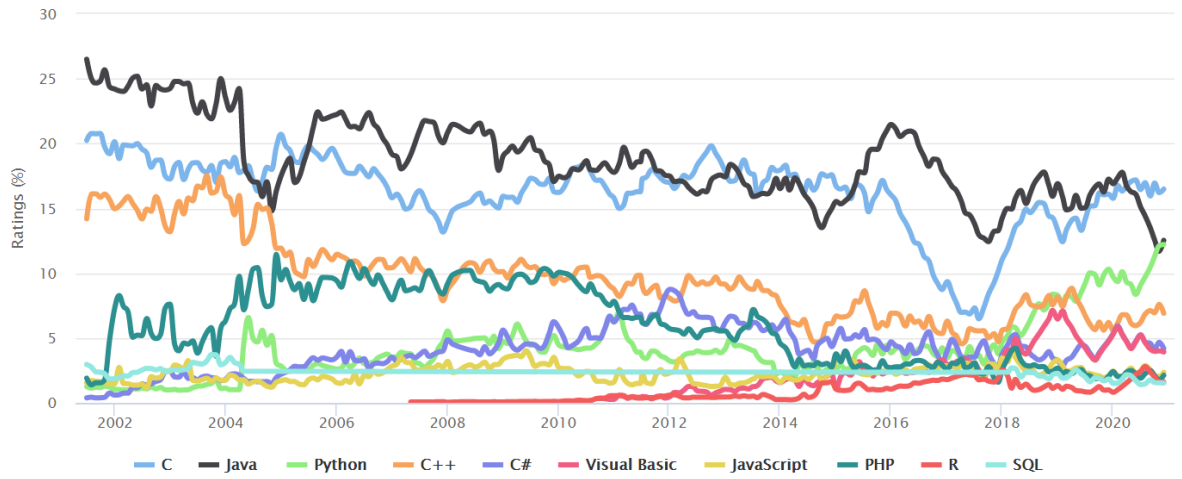

Figure 1.1: The most popular languages and their trends.

Source: https://www.tiobe.com/tiobe-index/

Figure 1.1 presents graphical overview on popularity of the most used programming languages in recent years.

1.4.2. F# Language

F# is a programming language that was created back in 2005 by Don Syme in cooperation with Microsoft Research. Although it is a multi-paradigm programming language it best fits the functional, declarative programming group. It is open-source and belongs to .NET Framework (more information here [13], [4]).

F# is a strongly typed language that uses type interference. It has lightweight syntax and is immutable by default. Lightweight syntax can be observed in low signal-to-noise ratio in comparison to other languages (for example C#).

Listing 1.1: C# code example

```
1  public static class SumOfSquaresHelper
2  {
3      public static int Square(int i)
4      {
5          return i * i;
6      }
7
8  public static int SumOfSquares(int n)
9      {
10         int sum = 0;
11         for (int i = 1; i <= n; i++)
12         {
13             sum += Square(i);
14         }
15         return sum;
16     }
17 }
18
19 int r = SumOfSquaresHelper.SumOfSquares(100);
```

Listing 1.2: F# code example

```
1 let sumOfSquares n =
2     [1..10] |> List.map (fun x -> x*x) |> List.sum
3
4 let r = sumOfSquares 100
```

Both above snippets are taken from the site `https://fsharpforfunandprofit.com/` with minor changes applied to further improve F# code by using lambda function (snippets source [2]). Both of these perform the same task – count the sum of squares of all integers preceding a certain number. The goal is to show how succinct F# can be in comparison to the same code but written in C#.

Taking all F# features into account it becomes visible that it is a very interesting, high level language that has a potential to be used in various situations. One of these is mathematical modeling vastly used for financial purposes such as pricing financial instruments – what is presented in this thesis. In order to estimate the price, a certain model has to be used and since there are so many variables nonlinearly affecting the final price – the models are frequently quite complex. For the sake of simplifying the process of implementing the model, having a syntax that very well resembles the mathematical language is beneficial.

Immutability options introduced in F# significantly reduce the possibility of making a mistake by a programmer which could be hard to find while debugging the code. Furthermore syntax such concise as F#'s is far more susceptible for code maintenance. These are the major reasons why F# was selected to be the core language of the project.

1.4.3. MVVM design pattern

Model-View-ViewModel is one of design patterns that are used in programming. It's primary role is to separate the user interface from the back-end logic. For this purpose there are 3 main components of the MVVM pattern:

- View,
- ViewModel,
- Model.

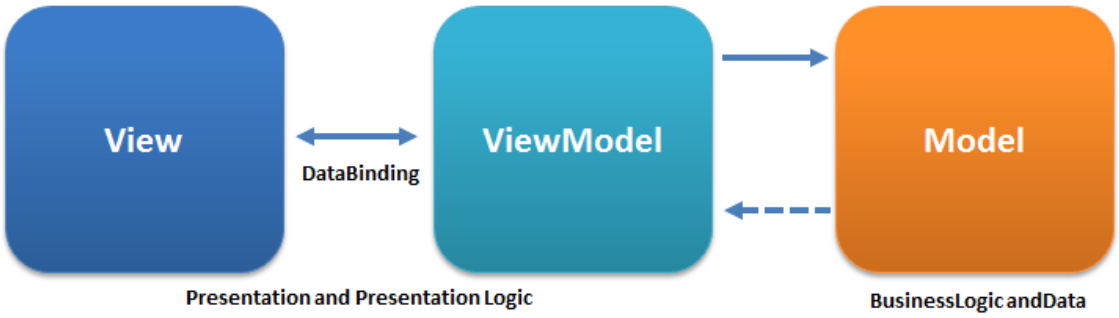

Figure 1.2: Three basic components of the MVVM architectural pattern.

Source: https://en.wikipedia.org/wiki/Model-view-viewmodel#/media/File:MVVMPattern.png

The Fig. 1.2 presents a graph with relations between the components. The important point is the fact that the View "does not see" the Model. Components of the View are *binded* with corresponding ones in the ViewModel. In the ViewModel they are specially equipped with mechanisms responsible for keeping the changes up to date (change in the logic immediately

triggers change in the View part). ViewModel works as a bridge between the Model and the View.

View

View in this case is a synonym of UI (in most cases GUI). View's purpose is to define the appearance of an app: where do buttons appear and how they look, how the data is presented to the user and which elements are static or have dynamic binding. As the name suggests, it handles the *view* of an app - what can actually be seen on the screen. In the presented application the role of the *View* part is taken by a project named *View* 1.3.

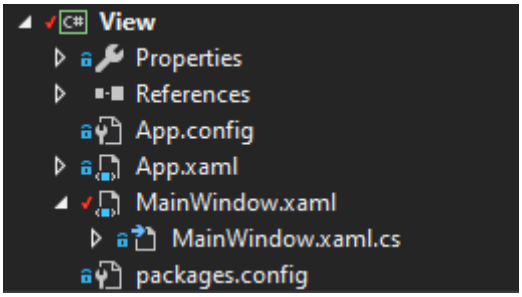

Figure 1.3: *View* project seen from Solution Explorer.

Source: own study

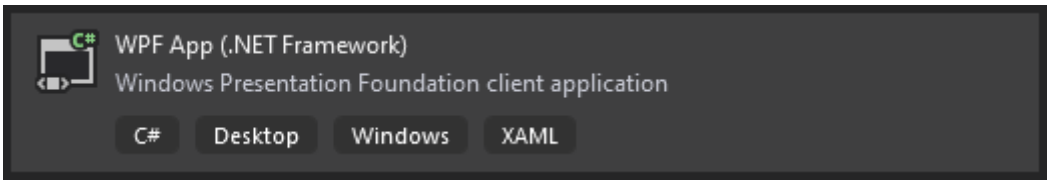

Figure 1.4: Default Visual Studio 2019 project used for writing the *View* part of an app.

Source: own study

This project, presented in the Figure 1.3, based on Visual Studio 2019 default project *WPF App (.NET Framework)* (see Fig. 1.4) is the only non-F# part of the MARS App as, for the time being (December 2020), there is no dedicated project in the F# language that would, by default, be compatible with MVVM architecture and XAML markup language for presenting the view. Therefore a few lines of C# code needed to be added in order to bind successfully View project with ViewModel one.

Model

The *Model* section handles business logic and data of the app. It is an inherent entity that could be reused in other applications. It is a significant advantage of the MVVM architectural pattern – reusability of the code is supported at the conceptual level of the design pattern.

This time *Model* project, show in the Figure 1.5, is solely an F# project that targets .Net Framework 4.6.1.

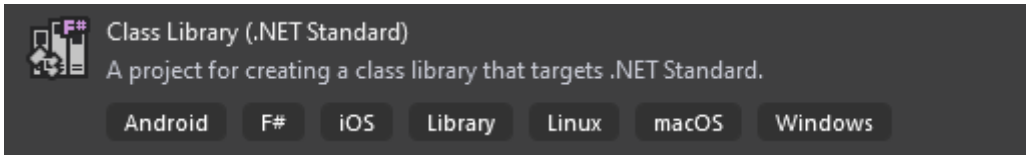

Figure 1.5: Default Visual Studio 2019 project used for writing the *Model* part of an app.

Source: own study

It is useful to remind here that whenever there is no valid reason to use *.NET Core* over *.NET Framework*, one should always opt for the latter. *.NET Core* is used when multi-platform approach is expected, but if the app is destined for Microsoft Windows system then the amount of libraries available on *.NET Standard* is vastly higher.

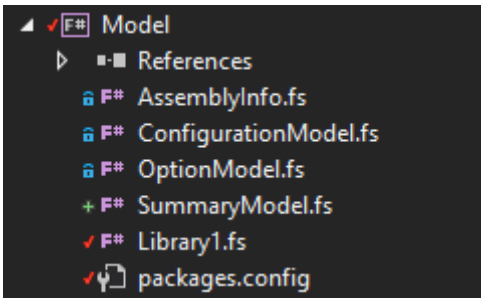

Figure 1.6: *Model* project seen from Solution Explorer.

Source: own study

ViewModel

Last but surely not least the *ViewModel* serves a role of a so-called *bridge* between *View* and the *Model*. It binds the graphical presentation of the components with hidden back-end logic. It must hold a reference to the *Model* project of the solution. It is extremely important for the programmer to design proper updates of data presented in corresponding graphic fields. *INotifyPropertyChanged* interface is perfect for this purpose. It makes sure no change in the data filed will go unnoticed if it can be seen anywhere in the app by the user.

It consists of multiple files responsible for specific purposes, all starting with the line:

Listing 1.3: F# all *ViewModel* project components beginning.

```
1 namespace ViewModel
2 // lines of code
```

An alternative notation would be to use F# *module* declaration, as in an example below. Such practice in more common nowadays than OOP-resembling *namespace*.

Listing 1.4: F# alternative example *ViewModel* project component beginning.

```
1 module MARSApp.ViewModel.ConfigurationViewModel
2 open MARSApp.ViewModel.ViewModelBase
3 open MARSApp.Model.ConfigurationModel
4 // lines of code
```

Although the difference and these code fragments 1.3 1.4 may seem unimportant, there is a solid reason for choosing the first one – since the project contains GUI written in XAML there is a need to specify in the view where certain properties can be found. It turns out there is a major

problem if the properties lay hidden inside a module. When they are presented in a namespace it is easier to bind them with XAML as it is adapted to receive a *namespace* declaration in the *Window* heading:

```
<Window
    x:Name="MyWindow"
    x:Class="View.MainWindow"
    xmlns:vm="clr-namespace:ViewModel;assembly=ViewModel"
```

Figure 1.7: Fragment of file *MainWindow.xaml* showing *namespace* reference.

Source: own study

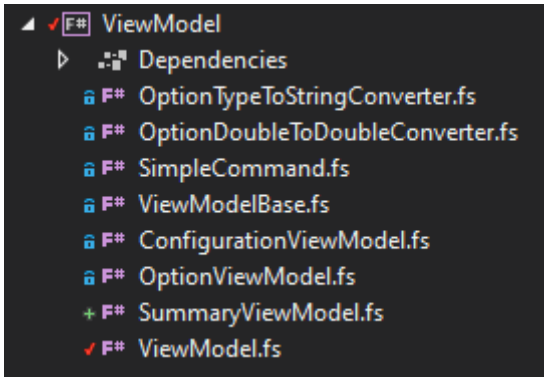

Figure 1.8: *ViewModel* project seen from Solution Explorer.

Source: own study

The reference would be much harder to achieve if *module* approach was used. For the sake of implementing *ViewModel* in this example the same default VS19 project was used as in *Model* project.

1.4.4. XAML Language

XAML stands for Extensible Application Markup Language and belongs to the declarative markup language group. It is used in *WPF* (*Windows Presentation Forms* – User Interface framework for creating desktop client applications, more details here [5]) for separating between how an application looks like and how it behaves (more how *XAML* works in [15]) (both the GUI and its behavior were created in the same language). *XAML* is an subset of *XML*, because every *XAML* file is as well a *XML* file, but not every *XML* file is a *XAML* file.

XAML may resemble *HTML*, although the idea behind those two is utterly different – by comparing *XML* to *HTML* one can observe that:

- *XML* is a markup language much like *HTML* but *HTML* was designed to display data while *XML* to carry it.
- *HTML* tags (such as: <p>, <h1>, <table>, etc.) are predefined while *XML* ones are created at the moment by the programmer.
- *HTML* allows small coding errors and is case insensitive in opposition to *XML*.

XAML is frequently used for building GUI part of *.NET* applications.

Chapter 2

Application – Theoretical Background and Practical Implementation

2.1. Geometric Brownian Motion

This section focuses on explaining what *Geometrical Brownian Motion* is and how it can be implemented in the F# language.

2.1.1. Geometric Brownian Motion – Introduction

Geometric Brownian Motion is often used as a model for simulating diverse variables. Therefore, financial processes, such as stock prices, are frequently modeled using *GBM*. In order to define correctly what a *GBM* is, it is essential to first describe several terms that will help further understanding.

2.1.2. Random Walk

Random Walk is a stochastic process used in mathematics that illustrates how objects might travel if they were to move randomly, more details here [11]. If 1D space is taken into account, then it is best to present the action on the number line.

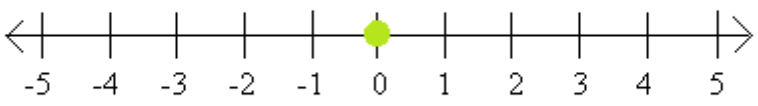

Figure 2.1: Number line with starting point $P_0 = 0$

P stands for *Position*.

Random walk starts at the point 0 (fig. 2.1). Then there occur N steps, $N \in \mathbb{N}_+$, and each of them is likely to move to the right $(+1)$ or to the left (-1) by 50%. The exemplary outcome after 4 steps could be as follows:

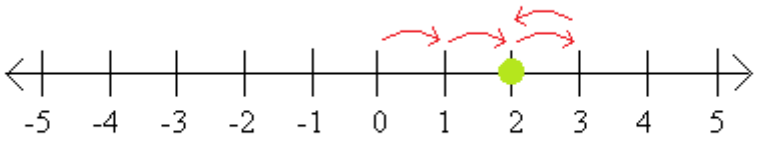

Figure 2.2: Sample outcome for $N = 4$.

In this example the steps could be described as $a_1 = a_2 = a_3 = 1$ and $a_4 = -1$, meaning the first 3 steps were to the right, while the last one was to the left. The outcome (position of the green dot after all the steps) is $P_4 = 2$ (fig. 2.2). Therefore, the position of the green dot can be specified by the following formula:

$$P_N = a_1 + a_2 + a_3 + \ldots + a_N.$$

Random Walk RW can then defined as the following series:

$$RW = \{P_n, n \in N\}.$$

Since each move has the same probability (can be either -1 or $+1$) then the expected value of such series (the final position of the dot) can be easily calculated as follows:

$$\mathbb{E}(RW) = 0.$$

An interesting observation can be made when calculating an estimated value of a series that is the same as *Random Walk* but its values are squares of the original values. Then:

$$\mathbb{E}(RW^2) = \sum_{i=1}^{N} a_i + 2 * \sum_{1 \le i \le j \le N}^{N} a_i * a_j = N.$$

Therefore, the average distance that the dot will appear on at the end of the walk will be at $\sqrt{N}$ blocks from the starting position.

2.1.3. Wiener Process

The *Wiener Process* is also usually called the *Standard Brownian Motion*. It is a continuous-time stochastic process named after Norbert Wiener for his work on 1D *Brownian Motion*. *Wiener Process*, as well as *Random Walk*, starts at 0 ($W(0) = 0$) and its increment follows Gaussian distribution with mean 0 and variance $t - s$ for any $0 \le s < t$ (more details here [14]). This way a *Brownian Motion* is basically a *Random Walk* with steps which sizes are random.

2.1.4. Geometric Brownian Motion – Explanation

Finally, having *Standard Brownian Motion* covered, it is the right time to explain what the Geometric version is. One of the biggest disadvantages of the Wiener Process is the fact that it

can reach negative values. When treated as a tool for simulating stock prices, such a flaw is disqualifying. Otherwise, it might lead to a situation, where certain stock's price goes below zero which would mean that one might actually get paid in order to obtain shares in the ownership of some corporation. Such a scenario never happens in reality, in contrast to interest rates (more information about negative interest rates can be found here: [8]).

Geometric Brownian Motion is a stochastic process, also called *Exponential Brownian Motion* due to the exponent occurring in its formula:

$$S_t = S_0 e^{(\mu - \frac{\sigma^2}{2})t + \sigma W_t},$$

where:

- S_t - underlying asset price at the moment t,
- S_0 - underlying asset price at the beginning ($t = 0$),
- μ - drift (7% represented as 0.07),
- σ - volatility (20% represented as 0.2),
- W_t - Wiener process' (Standard Brownian Motion) value at the moment t.

The key to understanding constant parameters μ – drift and σ – volatility is to pair the former with modeling deterministic trends while the latter with the amount of unpredictable events occurrences during the motion (more information about *GBM* can be found here [12]).

To create *Geometric Brownian Motion*, *Wiener Process* based on random normal variables is required. Since, in F# language there is no default library containing such variables generator, one can use *Box-Muller Transform*. For two independently uniformly distributed random variables U_1 and U_2 that come from a distribution on the interval $[0, 1]$, *Box-Muller Transform* creates 2 independent, **normal** random variables N_1 and N_2:

$$N_1 = \sqrt{-2 \ln U_1} \sin(2\pi U_2), N_2 = \sqrt{-2 \ln U_1} \cos(2\pi U_2).$$

Undermentioned listing shows possible implementation of *GBM*.

Listing 2.1: F# implementation of *Geometric Brownian Motion*.

```
1  //generates list of n Uniform RVs from interval [0,1]
2  let genRandomNumbersNominalInterval (count:int) (seed:int) :
       ↪ float list =
3      let rnd = System.Random(seed)
4      List.init count (fun _ -> rnd.NextDouble())
5
6  //input: UniformRM need to be from interval (0,1]
7  //input: steps must be even
8  //output: NormalRV have mean=0 and standard_deviation=1
9  let normalizeRec (uniformList:float list) (n:int) : float list
       ↪ =
10     let rec buildNormalList (normalList:float list) =
11         if normalList.Length = n then normalList
12         else
13             let currentNIdOne = normalList.Length
14             let currentNIdTwo = currentNIdOne + 1
15             let oneU = uniformList.[currentNIdOne]
16             let twoU = uniformList.[currentNIdTwo]
17             let oneN = sqrt(-2.*Math.Log(oneU, Math.E))*sin(2.*
                   ↪ Math.PI*twoU)
```

```
18              let twoN = sqrt(-2.*Math.Log(oneU, Math.E))*cos(2.*
                    ↪ Math.PI*twoU)
19              let newUniforms = [oneN; twoN]
20              buildNormalList (normalList@newUniforms)
21          buildNormalList []
22  let simulateGBM (count:int) (steps:int) (price:float) (drift:
        ↪ float) (vol:float) (years:float) (seed:int) =
23      let normalRV = normalizeRec (
            ↪ genRandomNumbersNominalInterval steps seed) steps
24      //build stock prices list
25      let rec buildStockPricesList (currentStockPricesList:float
            ↪ list) (steps:int) (normalId:int) : float list =
26          if normalId = steps-1 then currentStockPricesList
27          else
28              let firstExpTerm =  (drift - (vol**2.)/2.) * (float
                    ↪ (years)/float(steps))
29              let secondExpTerm =  vol * sqrt(float(years)/float(
                    ↪ steps)) * normalRV.[normalId]
30              let newStockPrice = currentStockPricesList.[
                    ↪ normalId] * Math.E ** (firstExpTerm +
                    ↪ secondExpTerm)
31              buildStockPricesList (currentStockPricesList@[
                    ↪ newStockPrice]) steps (normalId+1)
32      let stockPricesList = buildStockPricesList [price] steps 0
33      stockPricesList
```

Figures 2.3 and 2.4 show how parameters such as the drift and the volatility affect trends generated by *GBM*. While drift (μ) is responsible for overall trend (whether stock prices rise overtime ($\mu > 0$) or drop very rapidly ($\mu \ll 0$)), volatility is a parameter describing dispersion of values. If it is high, then the dispersion is high as well (see red trend in the Fig. 2.4).

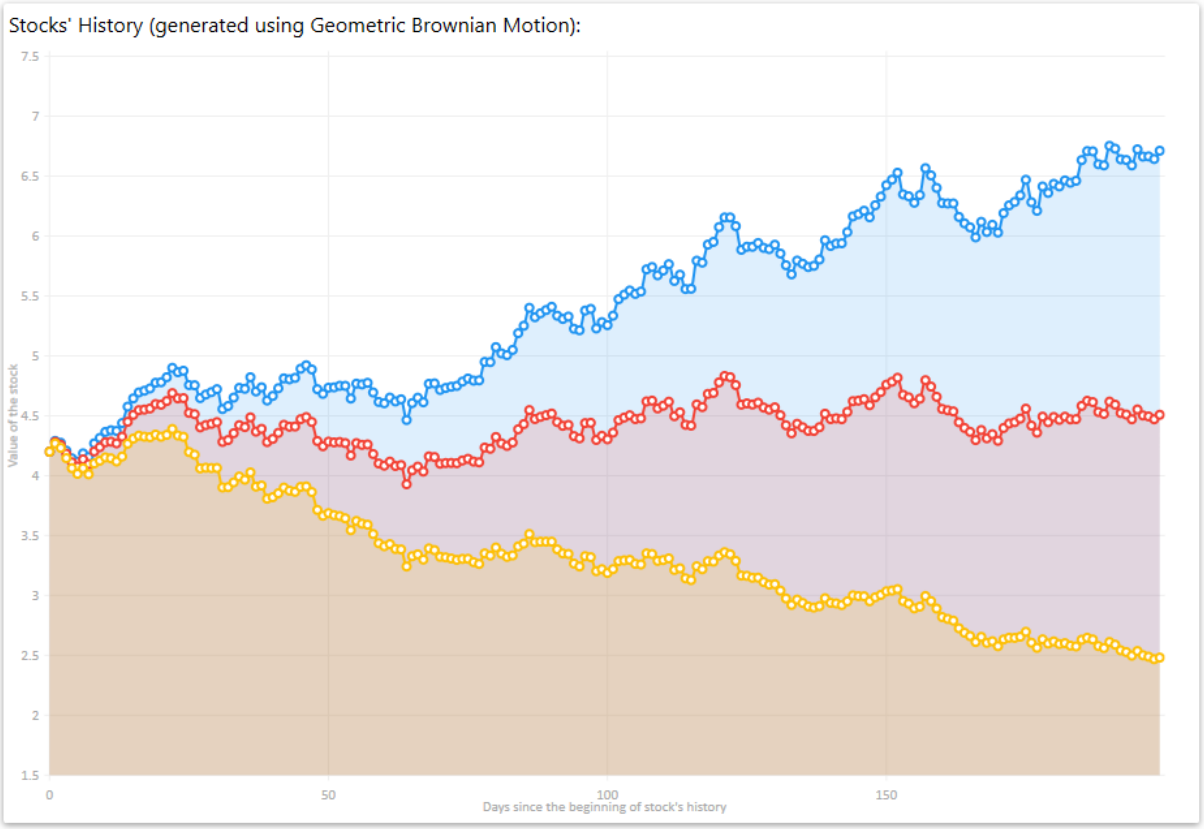

Figure 2.3: *GBM* trends when drift (μ parameter) is changing:

Source: *MARS App*

blue: $\mu = 0.6$,

red: $\mu = 0.2$,

yellow: $\mu = -0.8$.

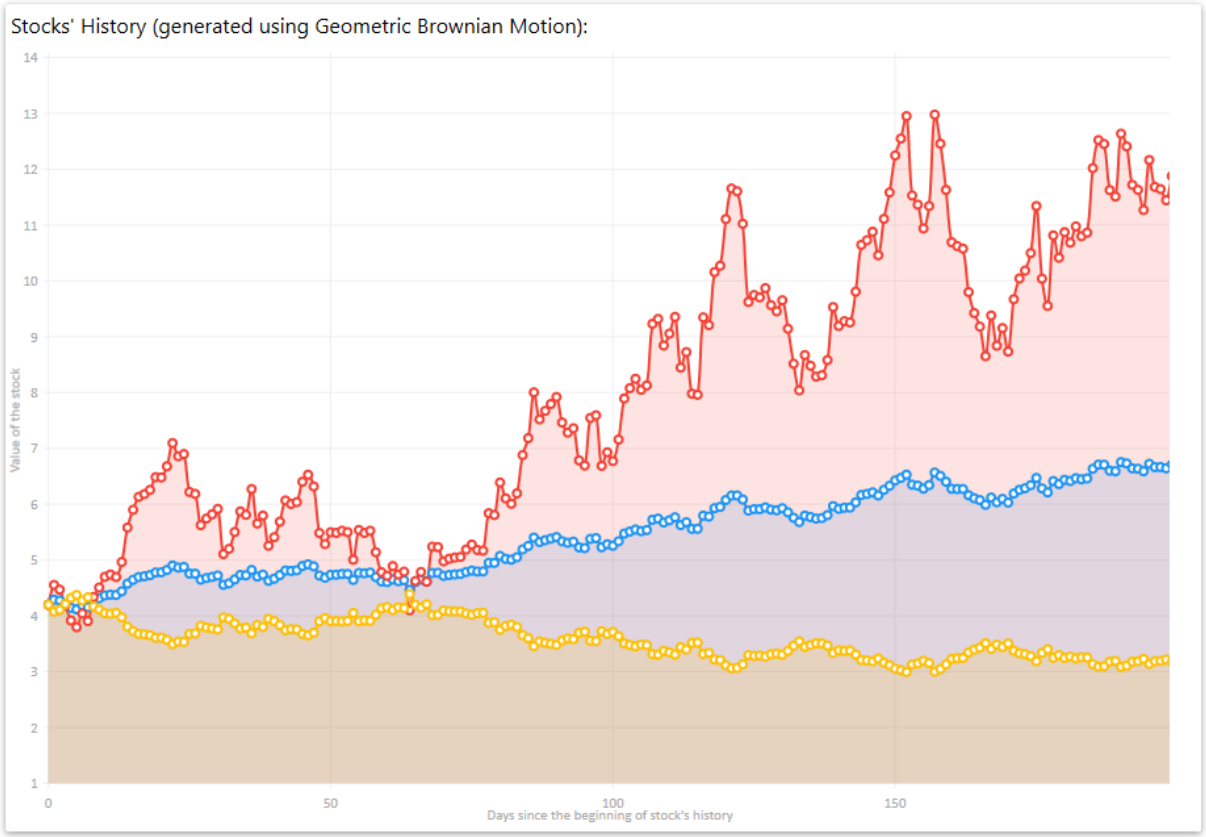

Figure 2.4: *GBM* trends when volatility (σ parameter) is changing:

Source: *MARS App*

red: $\sigma = 0.8$,

blue: $\sigma = 0.2$,

yellow: $\sigma = -0.3$.

2.2. Options

Options are some of the most popular derivative financial instruments. Derivative means in this example that it's value is reliant upon an underlying asset. The most important thing about options, one that distinguishes them from other derivative instruments it the **optionality** – contract can, but is not obligatory to be made.

There are 2 main types of options:

- **Call option** – the buyer of such option buys himself a right (it is not an obligation) to exercise the option (buy an underlying asset) from the option seller after the option has reached its maturity.
- **Put option** – the buyer of such option buys himself a right to sell an underlying asset to the option seller after the option has reached its maturity.

More about the options definitions can be found here [1].

There are several parameters that describe an option, most important ones are:

- **Expiration Date** – Specific moment in time by which the holder of the option has to decide whether he wants to exercise (use) the option.

- **Strike Price** – agreed price at which the derivative underlying asset can be bought or sold (depends on a type of option) when it is exercised.

Further division into option types depends on when the option can be exercised. The most basic one is a **European**-style option – the decision whether option will or will not be exercised is made at the time of the option's expiry. A continuous option type is an **American**–style option – option can be exercised at any time before the expiry. Combination of those two can be found in the **Burmudan**–style option – the contract can be exercised at specific days before the expiration. In this paper European options will be presented (more about option types in [3]).

The code which implements options is as follows:

Listing 2.2: F# implementation of an option.

```
type CallOrPutFlag =
    | Call
    | Put
    override this.ToString() =
                match this with
                | Call -> "Call"
                | Put -> "Put"

type OptionRecord =
    (* Model for Option Record. *)
    {
        OptionName:      string
        Expiry:          DateTime
        Strike:          float
        CallOrPutFlag:   CallOrPutFlag
        StockPrice:      float
        UnderlyingStock:Stock
    }

    (* Simple utility method for creating a random option. *)
    static member sysRandom = System.Random()
    static member Random (calc : CalculationParameters) (market
      ↪ : MarketData) =
        let rnd  = System.Random()
        (* Below OptionRecord type  will be returned *)
        {
            OptionName       = sprintf "Option%03d" (
              ↪ OptionRecord.sysRandom.Next(999))
            Expiry           = (DateTime.Now.AddMonths(
              ↪ OptionRecord.sysRandom.Next(2, 12))).Date
            Strike           =
                let s =
                    match market.TryFind "stock::price" with
                    | Some price -> float price
                    | None -> 6.70  // default 6.70$ for stock
                      ↪ price
                s*1.1    //Strike price is 110% of stock price

            CallOrPutFlag    =
```

```
36                  if rnd.Next()%2 |> System.Convert.ToBoolean
                        ↪ then
37                      Call
38                  else
39                      Put
40
41              StockPrice      =
42                  let s =
43                      match market.TryFind "stock::price" with
44                      | Some price -> float price
45                      | None -> 6.70  // default 6.70$ for stock
                            ↪ price
46                  s
47              UnderlyingStock = Stock.Random (calc:
                    ↪ CalculationParameters) (market : MarketData)
48          }
```

Generating options is presented in Figure 2.5:

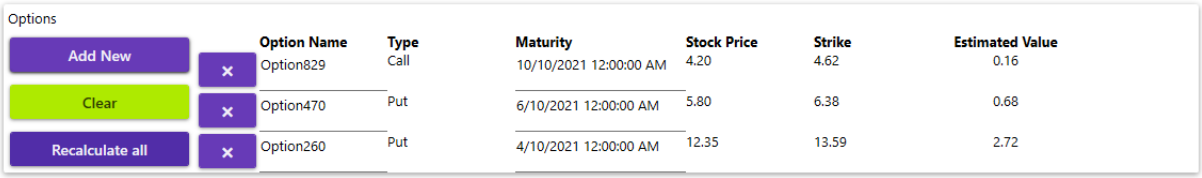

Figure 2.5: Options view in the application.

Source: *MARS App*

Before an option is generated, the user is able to specify parameters affecting the calculation, as shown in Fig. 2.6.

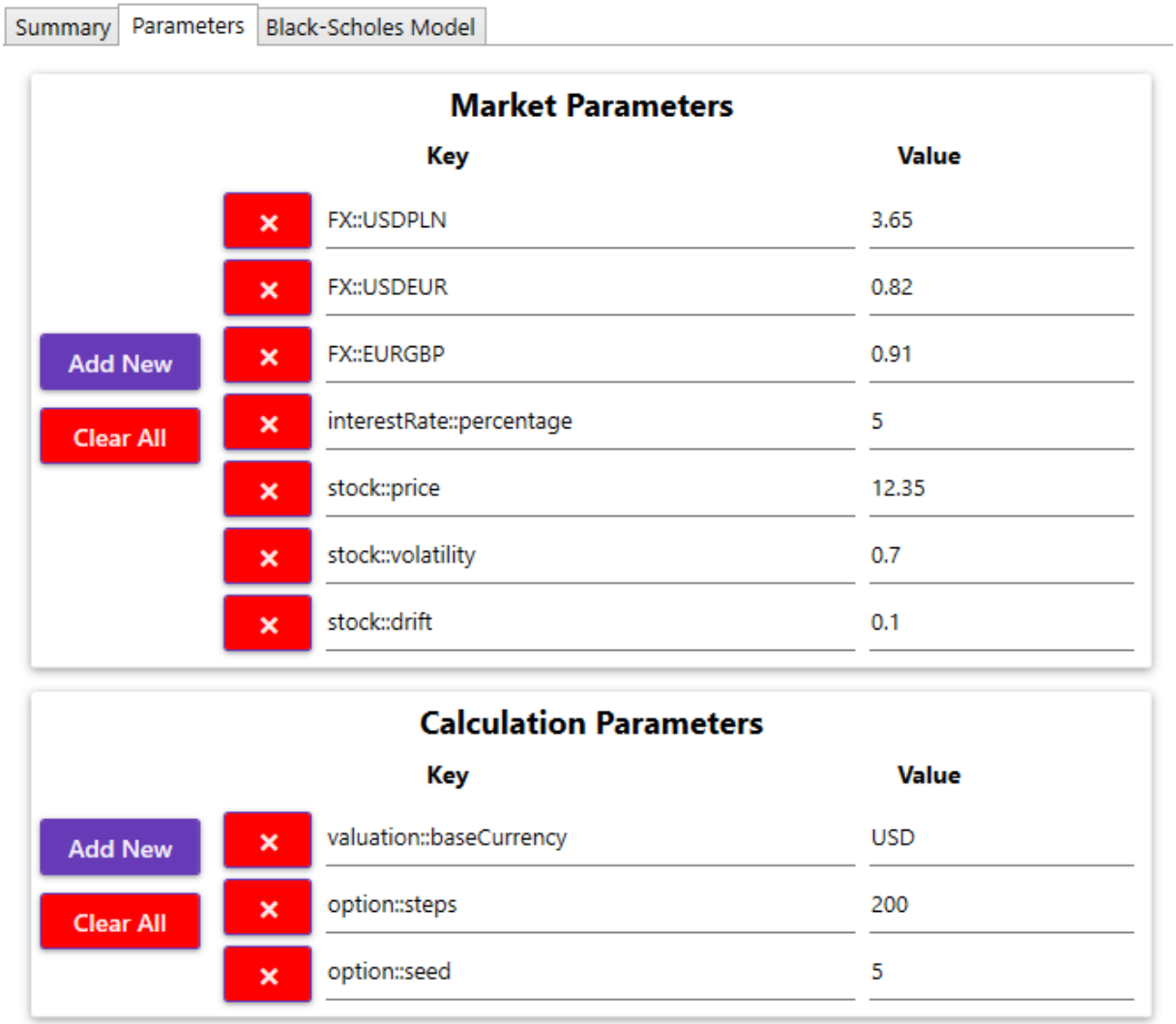

Figure 2.6: Parameters specification for options generation and valuation.

Source: *MARS App*

2.3. Black-Scholes Model

The Black–Scholes formula was created back in 1970s by 3 great economists: Fischer Black, Myron Scholes and Robert Merton (Scholes and Merton were later awarded Nobel Prize in Economics in 1997. We may assume same would happen for Black if sadly it was not for his death in 1975). That why this model is sometimes also called Black–Scholes–Merton. Their model was a significant breakthrough in the world of mathematical models used for pricing derivative instruments. It provides a framework for European-style option valuations, such as calls and puts.

This thesis' aim was not to goo too deep into the understanding of the mathematical background behind the model but rather to implement the model in a practical tool that one would be able to effectively use. Therefore more information and specifics can be found in the original publication [16] of the model from 1973 in *The Pricing of Options and Corporate Liabilities* of the Journal of Political Economy by Fischer Black and Myron Scholes.

The main formula of the model used for option pricing is as follows:

$$C = S_0\phi(d_1) - Ke^{-rt}\phi(d_2),$$

$$P = -S_0\phi(-d_1) + Ke^{-rt}\phi(-d_2),$$
$$d_1 = ln\frac{S_0}{K} + (r + \frac{\sigma^2}{2})t,$$
$$d_2 = d_1 - \sigma\sqrt{t},$$

where:

- C - call option price,
- P - put option price,
- S_0 - current price of an underlying asset,
- $\phi()$ - standardized cumulative normal distribution,
- K - strike price,
- r - risk free rate (1% represented as 0.01),
- t - time to maturity in years (18 months represented as 1.5),
- σ - volatility (20% represented as 0.2).

The formula assumes that the price history of an underlying asset (in this example - a stock price) has a lognormal distribution and follows *Geometric Brownian Motion* with constant drift and volatility. A simple analytical formula allows the model to be written in several lines of code thanks to the functional paradigm of the F# language.

Listing 2.3: F# implementation of *Black–Scholes Model*.

```
1     (* Black-Scholes for Option Valuation
2     Parameters:
3         call_or_put_flag (CallOrPutFlag):
4             Flag that determines whether this option is put or
                ↪ call
5         s0 (float): current price of an underlying asset (stock
              ↪ )
6         k (float): strike price
7         t (float): expiry date (counted in years from now on, e
              ↪ .g. 1.5)
8         r (float): risk free rate (e.g. 0.05 means 5%)
9         v (float): volatility (e.g. 0.2 means 20%)
10    *)
11    member this.BlackScholes() : float =
12        let d1 = ( log(s0/k) + (r+v*v/2.)*t) / (v*sqrt(t))
13        let d2 = d1 - v*sqrt(t)
14        match call_or_put_flag with
15        | Call -> s0*fi(d1) - k*exp(-r*t)*fi(d2)
16        | Put ->  k*exp(-r*t)*fi(-d2) - s0*fi(-d1)
```

Chapter 3

Conclusions

3.1. Have the assumptions been met?

In conclusion, the aim of this thesis has been achieved by creating a program – *MARS App* – that fulfills this work's assumptions. 83% of the code behind the app is an F# code and, wherever possible, a functional approach has been used. Therefore, the functional part seems to have been covered sufficiently.

Subsequently, a derivative financial instrument in the form of a European-style option has been implemented and priced under the *Black-Scholes Model* as presented in Fig. 2.5. Underlying asset's history has been generated using *Geometric Brownian Motion*, as presented in the previous chapter. The possibility to specialize the product has been introduced as well, what can bee seen in Fig. 2.6. The elements of the graphic design concerning product specialization, option presentation and *GBM* visualisation can be seen in corresponding sections.

This way all the assumptions have been met and several conclusions can be drawn from this work:

- Complex mathematical models can be introduced in few lines of F# code.
- For the time being (December 2020) F# language is not the best choice for creating a desktop application due to not large enough community, lack of tutorials and overall onerousness related to binding UI with back-end logic.
- *Black–Scholes Model*, although nearly 50-year-old, is still widely used despite the era of Artificial Intelligence and its benefits.

3.2. Further development possibilities

Future works may concern adding:

- connection to an online database in order to use historical data for an underlying asset instead of artificial trends generated by *GBM*,
- so-called *greeks* to further improve *MARS App* utility,
- *Monte Carlo Simulation* (written in C++ to speed up computing) as another model for comparison with *GBM*.

List of Figures

List of listings

Bibliography

[1] Call/put option definition. https://economictimes.indiatimes.com/definition/call-option. Accessed: 26 November 2020.

[2] Comparing f# with c#: A simple sum. https://fsharpforfunandprofit.com/posts/fvsc-sum-of-squares/. Accessed: 01 December 2020.

[3] Corporate Finance Institute european/american/bermudan option types. https://corporatefinanceinstitute.com/resources/knowledge/trading-investing/american-vs-european-vs-bermudan-options/. Accessed: 26 November 2020.

[4] F sharp (programming language). https://en.wikipedia.org/wiki/F_Sharp_(programming_language). Accessed: 02 December 2020.

[5] Get started with wpf. https://docs.microsoft.com/en-gb/visualstudio/designers/getting-started-with-wpf?view=vs-2019. Accessed: 05 December 2020.

[6] How many programming languages are there? https://devskiller.com/how-many-programming-languages/. Accessed: 29 November 2020.

[7] Logic programming. https://en.wikipedia.org/wiki/Logic_programming. Accessed: 01 December 2020.

[8] Negative interest rate. https://www.investopedia.com/terms/n/negative-interest-rate.asp. Accessed: 09 December 2020.

[9] Online historical encyclopaedia of programming languages. https://hopl.info/. Accessed: 29 November 2020.

[10] Procedures and functions. https://www.bbc.co.uk/bitesize/guides/zqh49j6/revision/1. Accessed: 01 December 2020.

[11] Random walks. https://www.mit.edu/~kardar/teaching/projects/chemotaxis(AndreaSchmidt)/random.htm. Accessed: 07 December 2020.

[12] Title. URL. Accessed: 01 December 2020.

[13] What is f#. https://docs.microsoft.com/en-ca/dotnet/fsharp/what-is-fsharp. Accessed: 02 December 2020.

[14] Wiener process. https://mathworld.wolfram.com/WienerProcess.html. Accessed: 08 December 2020.

[15] Xaml - overview. https://www.tutorialspoint.com/xaml/xaml_overview.htm. Accessed: 05 December 2020.

[16] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

Appendix A

CD/DVD included

Attached CD/DVD includes the following directories:

```
/
├── MARS_App.zip ...  This is a zipped file containing
│                     entire projects that create the
│                     solution as well as .sln file
│                     for easy further development in
│                     Microsoft Visual Studio 2019.
└── thesis_pdf ...  This directory contains .pdf
                    version of the Engineering Thesis.
```

Alternatively, application and this thesis can be found under these links:
https://github.com/kubaunold/MARS_App.git
https://github.com/kubaunold/EngineeringThesis.git

Index