Simulation (MAA313)

Project, 4.5 ECTS credits

Date:
Spring 2022

Project Variant No. 12

***Author*:**
Richard Johnsson

***Instructor(s)*:**
Hossein Nohrouzian
Anatoliy Malyarenko

DIVISION OF MATHEMATICS AND PHYSICS
MÄLARDALEN UNIVERSITY
SE-721 23, VÄSTERÅS, SWEDEN

Abstract

This report will first cover the methodology of solving a discrete monitored down-and-in put option in *the Black-Scholes* model. Next, the methods behind pricing a discretely monitored up-and-in put option in *Black-Scholes* using Sobol sequences. This report covers this in the coding language, *Vlang*.
Keywords: Black-Scholes, Barrier Options, Discrete Barrier, Monte Carlo Simulations, Quasi-Monte Carlo method, Estimate Pricing.

Contents

1 Introduction

According to [5] a *down-and-in* barrier option is a plain option if the asset price *has* been *below* the barrier level *h*. For the *up-and-in* option, it's like a plain option if the asset price *has* been *above* the barrier level *h*. It is also said to be the same payoff as regular options, meaning that a *down-and-in* put option is a vanilla European put option with payoff $max(K - S_T, 0)$ with the additional h barrier. When the option reaches the barrier level, the option expires worthless or gets activated depending on the barrier type. Therefore, the payoff behaves as a vanilla option with the condition $I = \{1, 0\}$.

2 Formulation of the problem 1

The first problem is to price a discretely down-and-in put option in *the Black-Scholes* model.

2.1 The theoretical and mathematical explanation

It is known that in the *Black Scholes* world, the stock price changes accordingly:

$$\begin{cases} dS = rSdt + \sigma SdW \\ S(0) = s \end{cases}$$

The solution to this equation is given by

$$S(T) = S(t)e^{(r-\frac{\sigma^2}{2})(T-t)+\sigma dW_{T-t}}, \tag{1}$$

where W is a brownian motion under the risk-neutral measure [5]. To implement Equation (1) in a programming software, one can use

$$S(t_{i+1}) = S(t_i)\exp\left\{\left(r - \frac{1}{2}\sigma^2\right)(t_{i+1} - t_i) + \sigma\sqrt{t_{i+1} - t_i}\, Z_{i+1}\right\}, \tag{2}$$

where Z_{i+1} is a random variable under the standard normal distribution. During the process of simulating the stock price as in (2), it is also checking for the condition whether $S(t_{i+1}) < h$, where h is the barrier level. If the stock price $S(t_{i+1}) < h$, the option is activated, and one can now price the down-and-in put option as a regular put option,

$$Price_{put} = max\{(K - S_T), 0\}, \tag{3}$$

where K is the strike price, and S_T is the stock at maturity. If the simulated stock process haven't met the condition under all iterations, i.e. $S(t_{i+1}) \not< $

h for $\forall i \in \{0, 1, ..., n\}$, where n represents the number of iterations done in the simulation process as in (2), the option is priced at 0. To generate a standard normal distributed number needed for the simulation, one must consider different approximations. In this paper, minimax approximations are used. The algorithm produced by *Peter John Acklam* is INVGAUSS. The algorithm minimises the maximum error from an estimator to the real values where the input is in $[0, 1]$. How is a uniform distributed number between $[0, 1]$ constructed? For this, a *Linear Congruential Generator (LCG)* is implemented. The form of *LCG* looks like the below,

$$X_{n+1} = (a * X_n + b) \mod m \text{ with } X_0 \text{ being the seed.}$$

$$\text{Where the uniform distributed number } u_{n+1} = \frac{X_{n+1}}{m}$$

Here, a, b and m are constants, and there have been plenty of studies determining which value gives the best result regarding randomness. In this report, it is decided to go with the following constants:

$$\begin{cases} m = 31104 \\ b = 6571 \\ a = 625 \end{cases}$$

These are some values often represented in literature [3]. The seed, X_0, is chosen by the author of this report and here, $X_0 = 42$. The origin is, therefore, 42.

2.2 Implementation in V and results

A module is created for problem one and problem two, called *Random*. This module contains every "random" function like generating a standard normal distributed normal, a uniformed number or *Sobol's* sequences. It includes all essential functions to solve problems one and two.

In the figure 1 the flow of the program is illustrated. It starts in *Main*, where the setup is run, asking the user what strike price, barrier type, barrier level etc. It is also here that the number of simulations is defined. In this case, 10,000 were simulated, and the estimate is the mean of the ten-thousand simulated options. In the process of simulating options, stock paths are forged, and this is where the module *Random* gets involved contributing with the standard normal distributed numbers needed for (2). A condition is checked during the simulation process to check whether the stock has been

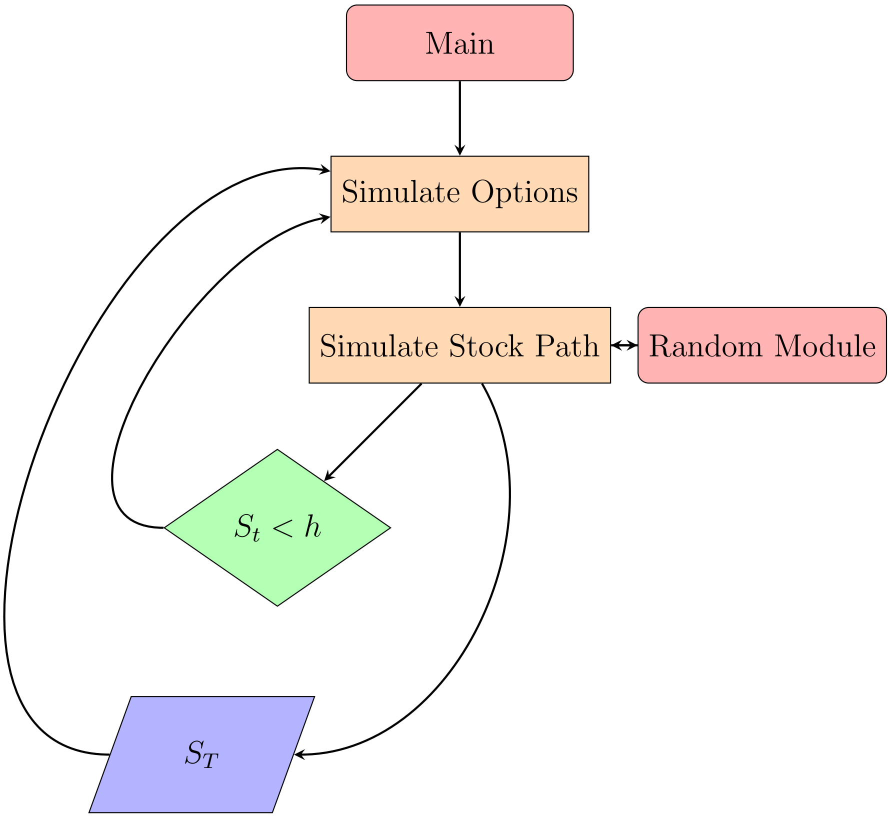

Figure 1: The Flow of Program - Problem 1

above or below the barrier level. As mentioned in the introduction, the condition lets the program know whether the option shall be activated or not1. The option is calculated in two ways. If the condition is met, i.e. the option is activated, the option price is then as a regular European put option, see (3). If the condition is not met, the option price is 0. These values are then appended to an array meant to hold all the options prices of 10,000 options. See appendix A.1 and A.3 for the code applied on problem one and the code for the random module.
Running the programA.1 at stock price at 100, strike price: 100, barrier level: 95, time to maturity: 1 year, steps: 252, the annual interest rate at: 0.05, sigma at: 0.2 and with 10,000 simulations, yields,

$$\text{Estimate: } 5.1769 \text{ with } 95\% \text{ confidence interval: } [5.0059, 5.3481].$$

where the confidence interval is calculated as

$$\hat{\mu} \pm 1.96 \frac{s}{\sqrt{n}}, \tag{4}$$

where n is the sample size which is the 10,000 simulated options. Here, $\hat{\mu}$ is the estimate of 5.1769 and s is the sample deviation.

3 Formulation of the problem 2

The second problem is to price a discretely up-and-in put option in *the Black-Scholes* model but using *Sobol'*s sequences.

3.1 The theoretical and mathematical explanation

An up-and-in put option is like an ordinary vanilla European put option. However, the difference from the option in problem one is that this option gets activated when the stock price reaches above a barrier level. Using *Sobol's* method requires one to select a *primitive polynomial* over binary arithmetic, and two properties have to be satisfied to be considered a *primitive polynomial.*

1. The polynomial can not be factored.

2. It must divide $x^p + 1$ with the smallest power $p = 2^q - 1$.

Where q is the degree of the polynomial and where a polynomial has the following appearance:

$$x^q + c_1 x^{q-1} + ... + c_{q-1} x + 1, \tag{5}$$

where $(c_1, ...c_{q-1}) \in \{0, 1\}$. A primitive polynomial can be seen as a list of bits [2]. This means that a bit $= 1$ implies that we have a x in this position. A bit $= 0$ implies there is no x here. For example, in a list of the following bits, 10011 represents the following polynomial:

$$x^4 + x + 1.$$

Where the last bit or the bit to the right will always equal 1 since it represents the constant term. The fact that the polynomial cannot be factored means our list of bits cannot be factored, which also implies that the representing number in the decimal base cannot be factored. This means prime numbers construct the primitive polynomials. For example,

$$10011_2 = 19_{10}.$$

The second properties which has to be satisfied requires one to filter through the prime numbers and is not done in this report.
The primitive polynomial has the following recurrence relation,

$$m_j = 2c_1 m_{j-1} \oplus 2^2 c_2 m_{j-2} \oplus ...2^{q-1} c_{q-1} m_{j-q+1} \oplus 2^q m_{j-q} \oplus m_{j-q}, \quad (6)$$

where m_j are integers and $\oplus$ is a XOR operation where a XOR is an operation like OR operation but where a $\{1, 1\}$ would lead to 0 instead of 1.

$$0 \oplus 0 = 0, 0 \oplus 1 = 1, 1 \oplus 0 = 1, 1 \oplus 1 = 0.$$

The values of m_1 up to m_q must be defined first before one can calculate the integer m_j. The direction numbers are defined as

$$v_j = \frac{m_j}{2j},$$

and these generates a *Sobol* sequence in the $[0, 1]^d$, where d is the dimension of a unit hypercube.

3.1.1 Choosing Initial Numbers

Assuming a d-dimensional sequence $x_0, x_1, ...x_q$ satisfies following condition: $j2^d \le k < (j + 1)2^d$ where exactly one of the points x_k falls in each of the 2^d cubes for every $j = 0, 1, ...$. The 2^d cubes has following form [2],

$$\Pi_{i=1}^d [\frac{a_i}{2}, \frac{a_i + 1}{2}], \quad a_i \in \{0, 1\}.$$

If the condition is satisfied the sequences is said to have *Property A*. The sequence is also said to have *Property A'* if for every $j = 0, 1, ..$ exactly one of the points x_k, $j2^{2d} \le k < (j + 1)2^{2d}$ falls in each of the 2^{2d} cubes of following form:

$$\Pi_{i=1}^d [\frac{a_i}{4}, \frac{a_i + 1}{4}], \quad a_i \in \{0, 1, 2, 3\}.$$

There[4] have been numbers generated so that *Property A* holds for 1111 dimensions. This will be used in this report.

3.2 Implementation in V and results

The program flow is similar to the flow in solution one, see figure 1. The difference here is that a *Sobol* sequence is generated to simulate the stock paths instead of pseudo-random numbers. The difference is that different functions are called within the random module in the figure 1, see A.1 and

A.2. For generating the *Sobol* sequence, a list of primitive polynomials was downloaded from DATA. The dataset also contains the initialised $m_1, ..., m_q$ values needed for the sequence. The data covers up to dimension 21201, but this report only covers up to dimension 1111. Now everything is collected to build the sequence and can now calculate (6) and from this calculate the direction numbers. Depending on how many steps one wants to simulate the stock path, a different number of direction numbers must be calculated. This report considers $252 * T$ steps, which means each step is a day forward. To convert the direction numbers to a normal distribution, one can put these in the inverse cumulative distribution function. The algorithm used in this report is taken from [1]. Doing the same procedure as in solution one, with different values for the strike price (k) and barrier level (h), will see the result below with $10,000$ simulations.

$$\text{Estimate: } 3.6285 \text{ with } 95\% \text{ confidence interval: } [3.4794, 3.7766].$$

The spot price is the same as in solution one, the strike price is changed to 105, and the barrier level is set at 105. The remaining parameters such as sigma, interest rate, time to maturity, and steps are all the same as solution one. The confidence interval is calculated like (4)

4 Conclusion and discussion

One could quickly solve these problems with a random module that is already defined. For example, Matlab has the function randn, Python has the module random, and there are plenty of more modules that could have been used. Instead, in this report, my own random module was created both to better understand the theory and, at the same time, challenge myself with a new programming language, Vlang. The results on solution one yield almost the same result as one would get by running the function barrierbybls in *Matlab*, which calculates the barrier option with the *Black-Scholes* model. Running the code listed in A.4 one would yield 5.5635, which is almost the same result in solution one. Running the code listed in A.5 one would get 4.4973. Both solutions are off by a margin, which is higher for Sobol's sequences. Perhaps running more simulations would lead to a better result when comparing. According to the *Central Limit Theorem*, the price should converge to the *Black-Scholes* price as the randomness should converge to the standard distribution. However, it is worth mentioning that the random module is not perfect and can be better programmed. For example, the Sobol's sequence generator crashes at high simulations $\approx 50,000$ due to memory issues.

A Appendix

A.1 Appendix Problem 1

```
module main

import os
import random_module
import math
import strconv
import progressbar
import time

struct Stock {
                price f64
                volatility f64
}

struct BarrierOption {
                strike f64
                opt_type string
                barrier_type string
                barrier f64
                simulations u64
}

struct RiskFreeAsset {
                rate f64
                ytd f64
}

fn main() {
        z := chan f64{cap: 1000} // A thread keeping a buffert containing 10000 random numbers all the time
        go random_module.randn(z)

        redo_inputs: // Pointers jumps here if inputs are invalid.
        stock,option,riskfree := setup()
        drift := riskfree.rate - (math.pow(stock.volatility,2))/2
        steps := math.round(252*riskfree.ytd)
        dt := riskfree.ytd/steps

        mut options := []f64{}
        if option.barrier_type == 'DO' && option.opt_type == 'Call' {
                // Price Down and Out Call option
                options = do_call(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'DO' && option.opt_type == 'Put' {
                // Price Down and Out Put option
                options = do_put(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'DI' && option.opt_type == 'Call' {
                // Price Down and In Call option
                options = di_call(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'DI' && option.opt_type == 'Put' {
                // Price Down and In Put option
                options = di_put(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'UI' && option.opt_type == 'Call' {
                // Price Up and In Call option
                options = ui_call(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'UI' && option.opt_type == 'Put' {
                // Price Up and In Put option
                options = ui_put(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'UO' && option.opt_type == 'Call' {
                // Price Up and Out Call option
                options = uo_call(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'UO' && option.opt_type == 'Put' {
                // Price Up and Out Put option
                options = uo_put(stock, option, riskfree, drift, steps, dt, z)
        }

        if mean(options) == 0 || mean(options).str() == 'nan' {
                os.system('clear')
                println('Invalid inputs! Please put a number when asked and corresponding strings!')
                ans := os.input('\n1 - Continue\n0 - Exit\n\n')
                if ans.int() == 1 {
```

```
                    unsafe {
                            goto redo_inputs
                    }
            }
            else {
                    exit(42)
            }
    }

    z.close()

    option_estimate := math.exp(-riskfree.rate*riskfree.ytd)*mean(options)
    upper_estimate := option_estimate+1.96*std(options)/math.sqrt(options.len)
    lower_estimate := option_estimate-1.96*std(options)/math.sqrt(options.len)
    println('Estimated option price is $option_estimate with confidence interval,
    95% CI:\t [$lower_estimate, $upper_estimate]')

}

fn setup() (Stock,BarrierOption,RiskFreeAsset){
    os.system('clear')
    println('Setting up Monte Carlo Simulation ... \n')

    stock_price := os.input('[Number] Stock price: ').f64()
    stock_vol := os.input('[Number] Stock volatility: ').f64()

    option_strike:= os.input('[Number] Option strike: ').f64()
    option_type := os.input('[String | Call, Put] Option type: ')
    option_barrier_type := os.input('[String | DO, DI, UI, UO] Barrier type: ')
    option_barrier := os.input('[Number] Barrier level: ').f64()
    option_simulations := os.input('[Number] Simulations: ').u64()

    risk_rate := os.input('[Number] Annual Rate: ').f64()
    risk_ytd := os.input('[Number] Years to Maturity: ').f64()

    stock := Stock{
            stock_price,
            stock_vol
    }

    option := BarrierOption{
            option_strike,
            option_type,
            option_barrier_type,
            option_barrier
            option_simulations
    }

    riskfree := RiskFreeAsset{
            risk_rate,
            risk_ytd
    }
    return stock,option,riskfree
}

fn mean(array []f64) f64{
    // Calculate the mean on an array
    mut sum := 0.0
    for val in array {
            sum += val
    }
    return sum/(array.len)
}

fn std(array []f64) f64{
    // Calculate sample deviation of an array
    mut sum := 0.0
    mu := mean(array)
    for val in array {
            sum += math.pow((val-mu),2)
    }
    return math.sqrt(sum/(array.len - 1))
}

fn monte_carlo_simulation(stock Stock, option BarrierOption, rf RiskFreeAsset,
                                        drift f64, steps f64, dt f64, z chan f64) []f64{
    mut stock_prices := [stock.price]
    for i  in 0 .. int(steps) {
            z_i := <- z
            stock_prices << stock_prices[i]*math.exp(drift*dt+
                                        stock.volatility*math.sqrt(dt)*z_i)
    }
    return stock_prices
```

```
}

fn max(array []f64) f64{
	// Function to get maximum of an array.
	mut sorted_array := array.clone()
	sorted_array.sort(a > b)
	return sorted_array[0]
}

fn min(array []f64) f64{
	// Function to get minimum of an array.
	mut sorted_array := array.clone()
	sorted_array.sort(a < b)
	return sorted_array[0]
}

fn do_call(stock Stock, option BarrierOption, rf RiskFreeAsset,
		drift f64, steps f64, dt f64, z chan f64) []f64{
	strconv.v_printf('\e[?25l')
	mut p := &progressbar.Progressbar{}
	p.new('Process', option.simulations)
	mut do_call := []f64{}
	mut stock_prices := []f64{}
	for sim in 0 .. option.simulations {
		stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
		if min(stock_prices) < option.barrier{
			do_call << 0.0
		}
		else {
			do_call << math.max((stock_prices.last() - option.strike),0)
		}
		p.increment()
	}
	p.finish()
	strconv.v_printf('\e[?25h')
	return do_call
}

fn do_put(stock Stock, option BarrierOption, rf RiskFreeAsset,
		drift f64, steps f64, dt f64, z chan f64) []f64{
	strconv.v_printf('\e[?25l')
	mut p := &progressbar.Progressbar{}
	p.new('Process', option.simulations)
	mut do_put := []f64{}
	mut stock_prices := []f64{}
	for sim in 0 .. option.simulations {
		stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
		if min(stock_prices) < option.barrier{
			do_put << 0.0
		}
		else {
			do_put << math.max((option.strike - stock_prices.last()),0)
		}
		p.increment()
	}
	p.finish()
	strconv.v_printf('\e[?25h')
	return do_put
}

fn di_call(stock Stock, option BarrierOption, rf RiskFreeAsset,
		drift f64, steps f64, dt f64, z chan f64) []f64{
	strconv.v_printf('\e[?25l')
	mut p := &progressbar.Progressbar{}
	p.new('Process', option.simulations)
	mut di_call := []f64{}
	mut stock_prices := []f64{}
	for sim in 0 .. option.simulations {
		stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
		if min(stock_prices) < option.barrier{
			di_call << math.max((stock_prices.last() - option.strike),0)
		}
		else {
			di_call << 0.0
		}
		p.increment()
	}
	p.finish()
	strconv.v_printf('\e[?25h')
	return di_call
}

fn di_put(stock Stock, option BarrierOption, rf RiskFreeAsset,
```

```
                drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut di_put := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if min(stock_prices) < option.barrier{
                        di_put << math.max((option.strike - stock_prices.last()),0)
                }
                else {
                        di_put << 0.0
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return di_put
}

fn uo_call(stock Stock, option BarrierOption, rf RiskFreeAsset,
                drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut uo_call := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if max(stock_prices) > option.barrier{
                        uo_call << 0.0
                }
                else {
                        uo_call << math.max((stock_prices.last() - option.strike),0)
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return uo_call
}

fn uo_put(stock Stock, option BarrierOption, rf RiskFreeAsset,
                drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut uo_put := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if max(stock_prices) > option.barrier{
                        uo_put << 0.0
                }
                else {
                        uo_put << math.max((option.strike - stock_prices.last()),0)
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return uo_put
}

fn ui_call(stock Stock, option BarrierOption, rf RiskFreeAsset,
                drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut ui_call := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if max(stock_prices) > option.barrier{
                        ui_call << math.max((stock_prices.last() - option.strike),0)
                }
                else {
                        ui_call << 0.0
                }
                p.increment()
        }
        p.finish()
```

```
        strconv.v_printf('\e[?25h')
        return ui_call
}

fn ui_put(stock Stock, option BarrierOption, rf RiskFreeAsset,
                   drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut ui_put := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if max(stock_prices) > option.barrier{
                        ui_put << math.max((option.strike - stock_prices.last()),0)
                }
                else {
                        ui_put << 0.0
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return ui_put
}
```

A.2 Appendix Problem 2

```
module main

import os
import random_module
import math
import strconv
import progressbar
import time

struct Stock {
                price f64
                volatility f64
}

struct BarrierOption {
                strike f64
                opt_type string
                barrier_type string
                barrier f64
                simulations u64
}

struct RiskFreeAsset {
                rate f64
                ytd f64
}

fn main() {
        redo_inputs: // Pointers jumps here if inputs are invalid.
        stock,option,riskfree := setup()
        drift := riskfree.rate - (math.pow(stock.volatility,2))/2
        steps := math.round(252*riskfree.ytd)
        dt := riskfree.ytd/steps

        z := chan f64{cap: 1000} // A thread keeping a buffert containing 10000 random numbers all the time
        go random_module.randn_sobol(z, steps)

        mut options := []f64{}
        if option.barrier_type == 'DO' && option.opt_type == 'Call' {
                // Price Down and Out Call option
                options = do_call(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'DO' && option.opt_type == 'Put' {
                // Price Down and Out Put option
                options = do_put(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'DI' && option.opt_type == 'Call' {
                // Price Down and In Call option
                options = di_call(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'DI' && option.opt_type == 'Put' {
```

```
                // Price Down and In Put option
                options = di_put(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'UI' && option.opt_type == 'Call' {
                // Price Up and In Call option
                options = ui_call(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'UI' && option.opt_type == 'Put' {
                // Price Up and In Put option
                options = ui_put(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'UO' && option.opt_type == 'Call' {
                // Price Up and Out Call option
                options = uo_call(stock, option, riskfree, drift, steps, dt, z)
        }
        else if option.barrier_type == 'UO' && option.opt_type == 'Put' {
                // Price Up and Out Put option
                options = uo_put(stock, option, riskfree, drift, steps, dt, z)
        }

        if mean(options) == 0 || mean(options).str() == 'nan' {
                // os.system('clear')
                println('Invalid inputs! Please put a number when asked and corresponding strings!')
                ans := os.input('\n1 - Continue\n0 - Exit\n\n')
                if ans.int() == 1 {
                        unsafe {
                                goto redo_inputs
                        }
                }
                else {
                        exit(42)
                }
        }

        z.close()

        option_estimate := math.exp(-riskfree.rate*riskfree.ytd)*mean(options)
        upper_estimate := option_estimate+1.96*std(options)/math.sqrt(options.len)
        lower_estimate := option_estimate-1.96*std(options)/math.sqrt(options.len)
        println('Estimated option price is $option_estimate with confidence interval,
        95% CI:\t [$lower_estimate, $upper_estimate]')

}

fn setup() (Stock,BarrierOption,RiskFreeAsset){
        os.system('clear')
        println('Setting up Monte Carlo Simulation ... \n')

        stock_price := os.input('[Number] Stock price: ').f64()
        stock_vol := os.input('[Number] Stock volatility: ').f64()

        option_strike:= os.input('[Number] Option strike: ').f64()
        option_type := os.input('[String | Call, Put] Option type: ')
        option_barrier_type := os.input('[String | DO, DI, UI, UO] Barrier type: ')
        option_barrier := os.input('[Number] Barrier level: ').f64()
        option_simulations := os.input('[Number] Simulations: ').u64()

        risk_rate := os.input('[Number] Annual Rate: ').f64()
        risk_ytd := os.input('[Number] Years to Maturity: ').f64()

        stock := Stock{
                stock_price,
                stock_vol
        }

        option := BarrierOption{
                option_strike,
                option_type,
                option_barrier_type,
                option_barrier
                option_simulations
        }

        riskfree := RiskFreeAsset{
                risk_rate,
                risk_ytd
        }
        return stock,option,riskfree
}

fn mean(array []f64) f64{
        // Calculate the mean on an array
        mut sum := 0.0
```

```
        for val in array {
                sum += val
        }
        return sum/(array.len)
}

fn std(array []f64) f64{
        // Calculate sample deviation of an array
        mut sum := 0.0
        mu := mean(array)
        for val in array {
                sum += math.pow((val-mu),2)
        }
        return math.sqrt(sum/(array.len - 1))
}

fn monte_carlo_simulation(stock Stock, option BarrierOption, rf RiskFreeAsset,
                                                drift f64, steps f64, dt f64, z chan f64) []f64{
        mut stock_prices := [stock.price]
        for i  in 0 .. int(steps) {
                z_i := <- z
                stock_prices << stock_prices[i]*math.exp(drift*dt+
                                                stock.volatility*math.sqrt(dt)*z_i)
        }
        return stock_prices
}

fn max(array []f64) f64{
        // Function to get maximum of an array.
        mut sorted_array := array.clone()
        sorted_array.sort(a > b)
        return sorted_array[0]
}

fn min(array []f64) f64{
        // Function to get minimum of an array.
        mut sorted_array := array.clone()
        sorted_array.sort(a < b)
        return sorted_array[0]
}

fn do_call(stock Stock, option BarrierOption, rf RiskFreeAsset,
                drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut do_call := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if min(stock_prices) < option.barrier{
                        do_call << 0.0
                }
                else {
                        do_call << math.max((stock_prices.last() - option.strike),0)
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return do_call
}

fn do_put(stock Stock, option BarrierOption, rf RiskFreeAsset,
                drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut do_put := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if min(stock_prices) < option.barrier{
                        do_put << 0.0
                }
                else {
                        do_put << math.max((option.strike - stock_prices.last()),0)
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return do_put
```

```
}

fn di_call(stock Stock, option BarrierOption, rf RiskFreeAsset,
                drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut di_call := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if min(stock_prices) < option.barrier{
                        di_call << math.max((stock_prices.last() - option.strike),0)
                }
                else {
                        di_call << 0.0
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return di_call
}

fn di_put(stock Stock, option BarrierOption, rf RiskFreeAsset,
                drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut di_put := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if min(stock_prices) < option.barrier{
                        di_put << math.max((option.strike - stock_prices.last()),0)
                }
                else {
                        di_put << 0.0
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return di_put
}

fn uo_call(stock Stock, option BarrierOption, rf RiskFreeAsset,
                drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut uo_call := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if max(stock_prices) > option.barrier{
                        uo_call << 0.0
                }
                else {
                        uo_call << math.max((stock_prices.last() - option.strike),0)
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return uo_call
}

fn uo_put(stock Stock, option BarrierOption, rf RiskFreeAsset,
                drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut uo_put := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if max(stock_prices) > option.barrier{
                        uo_put << 0.0
                }
                else {
                        uo_put << math.max((option.strike - stock_prices.last()),0)
                }
```

```
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return uo_put
}

fn ui_call(stock Stock, option BarrierOption, rf RiskFreeAsset,
                   drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut ui_call := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if max(stock_prices) > option.barrier{
                        ui_call << math.max((stock_prices.last() - option.strike),0)
                }
                else {
                        ui_call << 0.0
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return ui_call
}

fn ui_put(stock Stock, option BarrierOption, rf RiskFreeAsset,
                   drift f64, steps f64, dt f64, z chan f64) []f64{
        strconv.v_printf('\e[?25l')
        mut p := &progressbar.Progressbar{}
        p.new('Process', option.simulations)
        mut ui_put := []f64{}
        mut stock_prices := []f64{}
        for sim in 0 .. option.simulations {
                stock_prices = monte_carlo_simulation(stock, option, rf, drift, steps, dt, z)
                if max(stock_prices) > option.barrier{
                        ui_put << math.max((option.strike - stock_prices.last()),0)
                }
                else {
                        ui_put << 0.0
                }
                p.increment()
        }
        p.finish()
        strconv.v_printf('\e[?25h')
        return ui_put
}
```

A.3 Appendix Random Module

```
module random_module

import math
import os
import time
const (
        // Used for problem 1, generating pseudorandom numbers
        a           = 625
        c           = 6571
        m           = 31104
        // Used for problem 2, generating low-discrepancy sequences and converting to inverse normal distribution
        a_1 = -3.969683028665376e+01
        a_2 = 2.209460984245205e+02
        a_3 = -2.759285104469687e+02
        a_4 = 1.383577518672690e+02
        a_5 = -3.066479806614716e+01
        a_6 = 2.506628277459239e+00

        b_1 = -5.447609879822406e+01
        b_2 = 1.615858368580409e+02
        b_3 = -1.556989798598866e+02
        b_4 = 6.680131188771972e+01
        b_5 = -1.328068155288572e+01

        c_1 = -7.784894002430293e-03
        c_2 = -3.223964580411365e-01
```

```
c_3 = -2.400758277161838e+00
c_4 = -2.549732539343734e+00
c_5 = 4.374664141464968e+00
c_6 = 2.938163982698783e+00

d_1 = 7.784695709041462e-03
d_2 = 3.224671290700398e-01
d_3 = 2.445134137142996e+00
d_4 = 3.754408661907416e+00

p_low = 0.02425
p_high = 0.97575
// Polynomials up to 1111 dimensions
polynomials = [1, 3, 7, 11, 13, 19, 25, 37, 59, 47, 61, 55, 41, 67, 97, 91, 109, 103, 115,
        131, 193, 137, 145, 143, 241, 157, 185, 167, 229, 171, 213, 191, 253, 203, 211, 239, 247,
        285, 369, 299, 301, 333, 351, 355, 357, 361, 391, 397, 425, 451, 463, 487, 501, 529, 539,
        545, 557, 563, 601, 607, 617, 623, 631, 637, 647, 661, 675, 677, 687, 695, 701, 719, 721,
        731, 757, 761, 787, 789, 799, 803, 817, 827, 847, 859, 865, 875, 877, 883, 895, 901, 911,
        949, 953, 967, 971, 973, 981, 985, 995, 1001, 1019, 1033, 1051, 1063, 1069, 1125, 1135,
        1153, 1163, 1221, 1239, 1255, 1267, 1279, 1293, 1305, 1315, 1329, 1341, 1347, 1367, 1387,
        1413, 1423, 1431, 1441, 1479, 1509, 1527, 1531, 1555, 1557, 1573, 1591, 1603, 1615, 1627,
        1657, 1663, 1673, 1717, 1729, 1747, 1759, 1789, 1815, 1821, 1825, 1849, 1863, 1869, 1877,
        1881, 1891, 1917, 1933, 1939, 1969, 2011, 2035, 2041, 2053, 2071, 2091, 2093, 2119, 2147,
        2149, 2161, 2171, 2189, 2197, 2207, 2217, 2225, 2255, 2257, 2273, 2279, 2283, 2293, 2317,
        2323, 2341, 2345, 2363, 2365, 2373, 2377, 2385, 2395, 2419, 2421, 2431, 2435, 2447, 2475,
        2477, 2489, 2503, 2521, 2533, 2551, 2561, 2567, 2579, 2581, 2601, 2633, 2657, 2669, 2681,
        2687, 2693, 2705, 2717, 2727, 2731, 2739, 2741, 2773, 2783, 2793, 2799, 2801, 2811, 2819,
        2825, 2833, 2867, 2879, 2881, 2891, 2905, 2911, 2917, 2927, 2941, 2951, 2955, 2963, 2965,
        2991, 2999, 3005, 3017, 3035, 3037, 3047, 3053, 3083, 3085, 3097, 3103, 3159, 3169, 3179,
        3187, 3205, 3209, 3223, 3227, 3229, 3251, 3263, 3271, 3277, 3283, 3285, 3299, 3305, 3319,
        3331, 3343, 3357, 3367, 3373, 3393, 3399, 3413, 3417, 3427, 3439, 3441, 3475, 3487, 3497,
        3515, 3517, 3529, 3543, 3547, 3553, 3559, 3573, 3589, 3613, 3617, 3623, 3627, 3635, 3641,
        3655, 3659, 3669, 3679, 3697, 3707, 3709, 3713, 3731, 3743, 3747, 3771, 3791, 3805, 3827,
        3833, 3851, 3865, 3889, 3895, 3933, 3947, 3949, 3957, 3971, 3985, 3991, 3995, 4007, 4013,
        4021, 4045, 4051, 4069, 4073, 4179, 4201, 4219, 4221, 4249, 4305, 4331, 4359, 4383, 4387,
        4411, 4431, 4439, 4449, 4459, 4485, 4531, 4569, 4575, 4621, 4663, 4669, 4711, 4723, 4735,
        4793, 4801, 4811, 4879, 4893, 4897, 4921, 4927, 4941, 4977, 5017, 5027, 5033, 5127, 5169,
        5175, 5199, 5213, 5223, 5237, 5287, 5293, 5331, 5391, 5405, 5453, 5523, 5573, 5591, 5597,
        5611, 5641, 5703, 5717, 5721, 5797, 5821, 5909, 5913, 5955, 5957, 6005, 6025, 6061, 6067,
        6079, 6081, 6231, 6237, 6289, 6295, 6329, 6383, 6427, 6453, 6465, 6501, 6523, 6539, 6577,
        6589, 6601, 6607, 6631, 6683, 6699, 6707, 6761, 6795, 6865, 6881, 6901, 6923, 6931, 6943,
        6999, 7057, 7079, 7103, 7105, 7123, 7173, 7185, 7191, 7207, 7245, 7303, 7327, 7333, 7355,
        7365, 7369, 7375, 7411, 7431, 7459, 7491, 7505, 7515, 7541, 7557, 7561, 7701, 7705, 7727,
        7749, 7761, 7783, 7795, 7823, 7907, 7953, 7963, 7975, 8049, 8089, 8123, 8125, 8137, 8219,
        8231, 8245, 8275, 8293, 8303, 8331, 8333, 8351, 8357, 8367, 8379, 8381, 8387, 8393, 8417,
        8435, 8461, 8469, 8489, 8495, 8507, 8515, 8551, 8555, 8569, 8585, 8599, 8605, 8639, 8641,
        8647, 8653, 8671, 8675, 8689, 8699, 8729, 8741, 8759, 8765, 8771, 8795, 8797, 8825, 8831,
        8841, 8855, 8859, 8883, 8895, 8909, 8943, 8951, 8955, 8965, 8999, 9003, 9031, 9045, 9049,
        9071, 9073, 9085, 9095, 9101, 9109, 9123, 9129, 9137, 9143, 9147, 9185, 9197, 9209, 9227,
        9235, 9247, 9253, 9257, 9277, 9297, 9303, 9313, 9325, 9343, 9347, 9371, 9373, 9397, 9407,
        9409, 9415, 9419, 9443, 9481, 9495, 9501, 9505, 9517, 9529, 9555, 9557, 9571, 9585, 9591,
        9607, 9611, 9621, 9625, 9631, 9647, 9661, 9669, 9679, 9687, 9707, 9731, 9733, 9745, 9773,
        9791, 9803, 9811, 9817, 9833, 9847, 9851, 9863, 9875, 9881, 9905, 9911, 9917, 9923, 9963,
        9973, 10003, 10025, 10043, 10063, 10071, 10077, 10091, 10099, 10105, 10115, 10129, 10145,
        10169, 10183, 10187, 10207, 10223, 10225, 10247, 10265, 10271, 10275, 10289, 10299, 10301,
        10309, 10343, 10357, 10373, 10411, 10413, 10431, 10445, 10453, 10463, 10467, 10473, 10491,
        10505, 10511, 10513, 10523, 10539, 10549, 10559, 10561, 10571, 10581, 10615, 10621, 10625,
        10643, 10655, 10671, 10679, 10685, 10691, 10711, 10739, 10741, 10755, 10767, 10781, 10785,
        10803, 10805, 10829, 10857, 10863, 10865, 10875, 10877, 10917, 10921, 10929, 10949, 10967,
        10971, 10987, 10995, 11009, 11029, 11043, 11045, 11055, 11063, 11075, 11081, 11117, 11135,
        11141, 11159, 11163, 11181, 11187, 11225, 11237, 11261, 11279, 11297, 11307, 11309, 11327,
        11329, 11341, 11377, 11403, 11405, 11413, 11427, 11439, 11453, 11461, 11473, 11479, 11489,
        11495, 11499, 11533, 11545, 11561, 11567, 11575, 11579, 11589, 11611, 11623, 11637, 11657,
        11663, 11687, 11691, 11701, 11747, 11761, 11773, 11783, 11795, 11797, 11817, 11849, 11855,
        11867, 11869, 11873, 11883, 11919, 11921, 11927, 11933, 11947, 11955, 11961, 11999, 12027,
        12029, 12037, 12041, 12049, 12055, 12095, 12097, 12107, 12109, 12121, 12127, 12133, 12137,
        12181, 12197, 12207, 12209, 12239, 12253, 12263, 12269, 12277, 12287, 12295, 12309, 12313,
        12335, 12361, 12367, 12391, 12409, 12415, 12433, 12449, 12469, 12479, 12481, 12499, 12505,
        12517, 12527, 12549, 12559, 12597, 12615, 12621, 12639, 12643, 12657, 12667, 12707, 12713,
        12727, 12741, 12745, 12763, 12769, 12779, 12781, 12787, 12799, 12809, 12815, 12829, 12839,
        12857, 12875, 12883, 12889, 12901, 12929, 12947, 12953, 12959, 12969, 12983, 12987, 12995,
        13015, 13019, 13031, 13063, 13077, 13103, 13137, 13149, 13173, 13207, 13211, 13227, 13241,
        13249, 13255, 13269, 13283, 13285, 13303, 13307, 13321, 13339, 13351, 13377, 13389, 13407,
        13417, 13431, 13435, 13447, 13459, 13465, 13477, 13501, 13513, 13531, 13543, 13561, 13581,
        13599, 13605, 13617, 13623, 13637, 13647, 13661, 13677, 13683, 13695, 13725, 13729, 13753,
        13773, 13781, 13785, 13795, 13801, 13807, 13825, 13835, 13855, 13861, 13871, 13883, 13897,
        13905, 13915, 13939, 13941, 13969, 13979, 13981, 13997, 14027, 14035, 14037, 14051, 14063,
        14085, 14095, 14107, 14113, 14125, 14137, 14145, 14151, 14163, 14193, 14199, 14219, 14229,
        14233, 14243, 14277, 14287, 14289, 14295, 14301, 14305, 14323, 14339, 14341, 14359, 14365,
        14375, 14387, 14411, 14425, 14441, 14449, 14499, 14513, 14523, 14537, 14543, 14561, 14579,
        14585, 14593, 14599, 14603, 14611, 14641, 14671, 14695, 14701, 14723, 14725, 14743, 14753,
        14759, 14765, 14795, 14797, 14803, 14831, 14839, 14845, 14855, 14889, 14895, 14909, 14929,
        14941, 14945, 14951, 14963, 14965, 14985, 15033, 15039, 15053, 15059, 15061, 15071, 15077,
```

```
		15081, 15099, 15121, 15147, 15149, 15157, 15167, 15187, 15193, 15203, 15205, 15215, 15217,
		15223, 15243, 15257, 15269, 15273, 15287, 15291, 15313, 15335, 15347, 15359, 15373, 15379,
		15381, 15391, 15395, 15397, 15419, 15439, 15453, 15469, 15491, 15503, 15517, 15527, 15531,
		15545, 15559, 15593, 15611, 15613, 15619, 15639, 15643, 15649, 15661, 15667, 15669, 15681,
		15693, 15717, 15721, 15741, 15745, 15765, 15793, 15799, 15811, 15825, 15835, 15847, 15851,
		15865, 15877, 15881, 15887, 15899, 15915, 15935, 15937, 15955, 15973, 15977, 16011, 16035,
		16061, 16069, 16087, 16093, 16097, 16121, 16141, 16153, 16159, 16165, 16183, 16189, 16195,
		16197, 16201, 16209, 16215, 16225, 16259, 16265, 16273, 16299, 16309, 16355, 16375, 16381]
)

pub struct Inits {
pub mut:
		seed u32
		count u16
}

fn rand(seed u32) (f64, f64) {
	// Generates a uniform distributed number in [0,1] by using lcg (Linear Congruential Generator)
	next_seed := math.fmod(a*seed+c, m)
	return next_seed/m, next_seed
}


pub fn randn(ch chan f64) chan f64{
	// Generates random standard distributed numbers
	mut rand_seed := Inits{
		seed: 42
	}
	for !ch.closed {
	u, seed := rand(rand_seed.seed)
	ch <- normal_inverse(u)
	rand_seed.seed = u32(seed)
	}
	return ch
}

fn normal_inverse(p f64) f64{
	// Called from function randn. Takes a input p in [0,1] and returns inverse cumulative distribution
	mut q := 0.0
	mut x := 0.0
	if p > 0 && p < p_low {
		q = math.sqrt(-2.0*math.log(p))
		x =  (((((c_1* q+c_2)*q+c_3)*q+c_4)*q+c_5)*q+c_6) /
			((((d_1*q+d_2)*q+d_3)*q+d_4)*q+1)
	}
	if p >= p_low && p <= p_high {
		q = p - 0.5
		r := q*q
		x = (((((a_1*r+a_2)*r+a_3)*r+a_4)*r+a_5)*r+a_6)*q /
			(((((b_1*r+b_2)*r+b_3)*r+b_4)*r+b_5)*r+1)
	}
	if p > p_high && p < 1 {
		q = math.sqrt(-2.0*math.log(1-p))
		x = -(((((c_1*q+c_2)*q+c_3)*q+c_4)*q+c_5)*q+c_6) /
			((((d_1*q+d_2)*q+d_3)*q+d_4)*q+1)
	}
	return x
}

fn sobol(ch chan f64, count u16, steps f64) {
	// Generates a new directional number in corresponding count polynomial.
	mut bit := int_to_bit(polynomials[count])
	// Polynomial coefficients is stored in bit
	// Do not want first and last coefficient in bit
	bit = bit[1..bit.len-1]
	init_m := generate_init_m(count)
	// Loads initilizaed values

	mut m_j := []u64{cap:50}
	// Store the initialized values in our m_j array
	for i in 0 ..init_m.len {
		m_j << init_m[i].u64()
		ch <- normal_inverse(init_m[i].u64()/(math.pow(2,i+1)))
	}
	for m_j.len <= 50{
		m_j = m_j.reverse()
		mut val := u64(0)
		mut j := 0
		for j < bit.len {
			val ^= u64(math.pow(2,j+1)) * u64(bit[j])*(m_j[j])
			j ++
		}
		val ^= u64(math.pow(2, j+1)) * m_j[j]^m_j[j]
```

```
                ch <- normal_inverse(val/(math.pow(2,m_j.len+1)))
                m_j = m_j.reverse()
                m_j << val
        }
        return
}

pub fn randn_sobol(ch chan f64, steps f64) {
        // Generates random standard distributed numbers from sobol generated [0,1].
        mut sobol_seed := Inits{
                count: 4 //
        }
        for !ch.closed {
                sobol(ch, sobol_seed.count, steps)
                sobol_seed.count ++
                if sobol_seed.count == 1111 {
                        // Property A holds for all 1111
                        sobol_seed.count = 1
                }
        }
        return
}

fn int_to_bit(integer f64) []int {
        // Converts integer to binary
        mut work_int := integer
        mut bit := []int{}
        mut i := 0
        for math.pow(2, i) <= integer {
                i++
        }
        i = i - 1
        bit << 1
        work_int = work_int - math.pow(2, i)
        i = i - 1
        for i >= 0 {
                if math.pow(2, i) <= work_int {
                        bit << 1
                        work_int = work_int - math.pow(2, i)
                } else {
                        bit << 0
                }
                i--
        }
        return bit
}

fn generate_init_m(count u16) []string{
        // generate initialized values from file
        path := os.resource_abs_path('assets/data/initmvalues.csv')
        file := os.read_file(path) or { panic(err) }
        // Propert A only holds for 1111 dimensions.
        return file.split_into_lines()[count].split(';')[3].split(' ')
}
```

A.4 Appendix Down-and-In Put Option [Matlab]

```
Rates = 0.05;
Settle = '01-Jan-2015';
Maturity = '01-Jan-2016';
Compounding = -1;
Basis = 1;

RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates', Maturity, ...
'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis)

AssetPrice = 100;
Volatility = 0.2;
StockSpec = stockspec(Volatility, AssetPrice)

Strike = 100;
OptSpec = 'put';
Barrier = 95;
BarrierSpec = 'DI';

Price = barrierbybls(RateSpec, StockSpec, OptSpec, Strike, Settle,...
Maturity,  BarrierSpec, Barrier)
```

A.5 Appendix Up-and-In Put Option [Matlab]

```
Rates = 0.05;
Settle = '01-Jan-2015';
Maturity = '01-Jan-2016';
Compounding = -1;
Basis = 1;

RateSpec = intenvset('ValuationDate', Settle, 'StartDates', Settle, 'EndDates', Maturity, ...
'Rates', Rates, 'Compounding', Compounding, 'Basis', Basis)

AssetPrice = 100;
Volatility = 0.2;
StockSpec = stockspec(Volatility, AssetPrice)

Strike = 105;
OptSpec = 'put';
Barrier = 105;
BarrierSpec = 'UI';

Price = barrierbybls(RateSpec, StockSpec, OptSpec, Strike, Settle,...
Maturity,  BarrierSpec, Barrier)
```

References

[1] Acklam, P. J., July 2013. Availabe at: https://github.com/liorzoue/js-normal-inverse/blob/master/normal.inverse.js.

[2] Glasserman, P. *Monte Carlo methods in financial engineering*, vol. 53 of *Applications of Mathematics (New York)*. Springer-Verlag, New York, 2004.

[3] Hernndez J.C. & Ribagorda, A. . I. P. . S. J. Finding near optimal parameters for linear congruential pseudorandom number generators by means of evolutionary computation.

[4] Joe, S., and Kuo, F. Y. Constructing sobol sequences with better two-dimensional projections. *SIAM Journal on Scientific Computing 30*, 5 (2008), 2635–2654.

[5] Röman, J. *The mathematics of Interest Rate Derivatives, Markets, Risk and Valuation.* Springer International Publishing AG, 2017.