FE522 - C++ in Finance

Humphrey De Guzman

Contents

Assignment 1 **1**

Assignment 1

For every problem below, create a different project folder. You should then put all these folders inside a .zip file with your name before submitting everything in Canvas. Remember to delete the ***cmake-build-debug*** *folders before doing so. This .zip file should be accompanied by a .pdf file containing a report (one single report for the whole assignment). We do not provide test cases for any of the problems, and these* ***must*** *be provided by you. When providing a solution to a problem, you must be able to present test cases alongside it,* ***otherwise it will not be accepted as a valid solution.***

If a problem requires input and/or output files, please provide the whole file content (if not large) in the body of the report. If the file is too large to be pleasantly presented in a report (larger than a page), please provide a sample. You should include these files in folders named "input" and "output", respectively, in the root folder of each project. In order for your code to work on any machine (not only yours), use relative paths to these files in your source code:

- *for input files, use:* `../../input/filename.txt`
- *for output files, use:* `../../output/filename.txt`

Problem 1 (20 points). Study the documentation in http://en.cppreference.com/w/cpp/numeric/random. Choose 5 different random number distributions, generate a sample of 10,000 numbers with each of them, and create a table (output to a file) with their mean, median, and standard deviation.

Answer 1. I utilize many of the already implemented distributions in the standard library to generate my 5 different samples of 10,000. To generate a seed for each distribution I use the library `<ctime>` to give me a starting seed. I put this seed through the Mersenne Twister RNG algorithm and use the `independent_bits_engine` adapter on it. I put this through another Mersenne Twister whose output I will be using as my seed. Due to `<ctime>`, I get a new random value as long as it is called at a different time. For the five distributions I used a normal distribution, log-normal distribution, poisson distribution, binomial distribution, and negative binomial distribution. See code below.

```
std::independent_bits_engine<std::mt19937 ,64, uint_fast64_t> seed(time(0));
std::mt19937_64 rng(seed());
std::normal_distribution<double> norm_dist{0,1};
std::lognormal_distribution<double> ln_dist(0,1);
std::poisson_distribution<int> pois_dist{3};
std::binomial_distribution<int> bn_dist(100,0.5);
std::negative_binomial_distribution<int> negBn_dist(10,0.5);
```

Answer 1(cont). Afterwards I create five vectors to store my samples for each distribution. I repeat this but instead declare numerics to store the mean, sigma, and median for each of my samples. Once this is done I simply just run a for loop filling in my vectors via `exampleVector.push_back(examplDist(rng))`. Once I find the mean, I can find the variance (I used the population variance formula). I can find the median by taking the mid point between our two middle values within our vectors, post sort of course. Within the main file I also print out test values from each distribution to check if my distribtuions work. The entirety of the main file with tests can be seen **here**. Below one can see the results that I output to a textfile.

	Mean	Standard.Deviation	Median
Normal Distribution	0.01	1.01	0.01
Log-Normal Distribution	1.64	2.43	0.99
Poisson Distribution	3.00	1.74	3.00
Binomial Distribution	49.00	5.06	50.00
Negative Binomial Distribution	10.00	4.44	9.00

Problem 2 (20 points). Implement the following methods to find the root of a function and test them with a polynomial function of your choice:

(a) Bisection method (https://en.wikipedia.org/wiki/Bisection_method)

(b) Secant method (https://en.wikipedia.org/wiki/Secant_method)

Answer 2. This one is quite self explanatory. I make two functions that take in two bounds as doubles and a pointer to a function that returns a double (The function that uses the secant method also takes in an integer that is used as the number of iterations of the method to perform). The wikipedia pages for both methods have instructions for the algorithms for each respective one, so all that was left to do is to translate that into c++. For the main file that includes the test parameters and functions, see **here**. Results of the tests cases are shown below.

```
Our function is x^3 - 8
The root of our function using the Bisection Method bounds (0,3) is: 2
The root of our function using the Secant Method bounds(3,5) is: 2

Our function is x - 3
The root of our function using the Bisection Method bounds (1,4) is: 3
The root of our function using the Secant Method bounds (1,4) is: 3

Our function is (x - 11)^3
The root of our function using the Bisection Method bounds (1,4) is:
Error: We cannot use the intermediate value theorem, root(s) not present in range
-1
The root of our function using the Secant Method bounds (1,10) is: 10.9589

Our function is (x - 11)^3
The root of our function using the Bisection Method bounds (1,11) is: 11
The root of our function using the Secant Method bounds (1,11) is: 11
```

Problem 3 (20 points). Design and implement a ‘`Money` class for calculations involving dollars and cents where arithmetic has to be accurate to the last cent using the 4/5 rounding rule (.5 of a cent rounds up; anything less than .5 rounds down). Provide addition (`Money + Money`), subtraction (`Money - Money`), multiplication (`Money * int` and `Money * double`), division (`Money / int` and `Money / double`), and equality operators. Represent a monetary amount as a number of cents in a `long`, but input and output as dollars and cents, e.g., $123.45. Do not worry about amounts that don't fit into a long. Define corresponding input `(>>)` and output `(<<)` operators.

Answer 3. My implementation of the money class has one private member `long` that I call `value`. This represents the value of our money object via cents. We allow for negative money. The only exceptions

thrown within this class are during the overload of the >> operator, in which our program will terminate if the input stream does not match a numeric value or money representation. For example, "$3.49" and "4007" are allowed, but "eleven" is not and will cause termination. If the istream has the dollar sign "$" prefix, it will consider the rest of the istream as denoted in dollars, if it does not, then it will be considered denoted in cents. For our previous examples, the two `Money` objects with `long values` of 3490 and 4007 would be created respectively. Like the previous problems, the test cases are inside the main file. The header file for our class can be seen **here**, with their implementations **here**, and the main file **here**. Results of the test cases are seen below.

```
Test constructors:
$0.00
$50.00

Test operator+ overload:
m1 is $100.00
m2 is $50.00
m1 + m2
$150.00

Test operator- overload:
m1 is $100.00
m2 is $50.00
m1 - m2
$50.00

Test operator* overload (1):
$100.00
m1 multiplied by 3
$300.00

Test operator* overload (2):
$100.00
m1 multiplied by 2.5
$250.00

Test operator/ overload (1):
$100.00
m1 divided by 24
$4.17

Test operator/ overload (2):
$100.00
m1 divided 2.5
$40.00

m1 is $100.00
m2 is $100.00
Test operator== overload:
m1 and m2 have equal values

m1 is $100.00
m2 is $1.23
Test operator!= overload:
m1 and m2 do not have equal values
```

```
m1 is $100.00
m2 is $1.23
Test operator< overload:
m2 < m1

m1 is $100.00
m2 is $1.23
Test operator<= overload:
m2 <= m1

m1 is $1.23
m2 is $1.23
Test operator> overload:
m2 !> m1

m1 is $1.23
m2 is $1.23
Test operator>= overload:
m2 >= m1

<Denotes as cents by default, prefix of '$' denotes as dollar>.
Note: non-numeric inputs throw std::invalid_argument and prompt program termination.
Please input a value (ex: $4.20 or 690):eleven
terminate called after throwing an instance of 'std::invalid_argument'
  what():  stod
```

Problem 4 (20 points). Design and implement a EuropeanOption class. It should have properconstructor(s) and hold information such as option type (call or put), spot price (of the underlying asset), strike price, interest rate, volatility (of the underlying asset), and time to maturity. Don't accept illegal values. Implement a getPrice() function which gives the price of the option using the Black & Scholes formula:

$$
\begin{aligned}
C(S_t, t) &= N(d_1)S_t - N(d_2)Ke^{-r(T-t)} \\
P(S_t, t) &= N(-d_2)Ke^{-r(T-t)} - N(-d_1)S_t \\
d_1 &= \frac{1}{\sigma\sqrt{T-t}}[ln(\frac{S_t}{K}) + (r + \frac{\sigma^2}{2})(T-t)] \\
d_2 &= d_1 - \sigma\sqrt{T-t}
\end{aligned}
$$

where

- $C(S_t, t)$ is the price of a call option;
- $P(S_t, t)$ is the price of a put option;
- $N()$ is the cumulative distribution function of the standard normal distribution;
- $T - t$ is the time to maturity (expressed in years);
- S_t is the spot price of the underlying asset;
- K is the strike price;
- r is the risk free rate (annual rate, expressed in terms of continuous compounding);
- *sigma* is the volatility of returns of the underlying asset;

Hint: Don't try to implement $N()$ (it is difficult). Do no reinvent the wheel.

Answer 4. For my implementation of `EuropeanOption` I create six `private` member variables, all of which are typ `double` except one which is a `bool`. I design two constructors, one with preset default values, and one

in which each private member of the `EuropeanOption` object can be inputted manually. I represent the type of option through an `enum` of `PUT` and `CALL` and store it in the one `bool` member, which I label `t`. All the other members are `time_to_maturity`, `spot`, `strike`, `risk_free`, and `vol`. They all take the form of `double` and each do what their name implies. The `time_to_maturity` member represents the amount of time left until maturity, `spot` represents the spot price of the underlying asset for our option, `strike` represents the strike price of our option, `risk_free` represents the risk_free rate (annual and continuously compounded), and `vol` represents the volatiility of the option's underlying asset's returns. Each member has a `public` getter and setter. The last method is `public`, denoted by `getPrice()` and gives me the option price using the black scholes equation mentioned previously. I overload the `<<` operator to allow for the output of `EuropeanOption` objects and use a CDF function from https://en.cppreference.com/w/cpp/numeric/math/erfc. My default constructor makes a call option with a time to maturity of 1, spot price of $100, strike price of $100, risk free rate of 2.5%, and volatility of 10%. We do not allow for options with a `time_to_maturity`, `spot`, or `strike` less than 0 and will throw an exception if attempted. The `<<` calls all getters and `getPrice()`. As always the header is found **here**, implementations **here**, and main w/test cases **here**. The main function with its output is small enough to display so we will do so below:

```
#include "EuropeanOption.h"
#include "std_lib_facilities.h"

int main()
{
    EuropeanOption op1;
    cout << op1 << endl;
    EuropeanOption op2(PUT,0.079452,158.28,167.5,.0099,.2048);
    cout << op2 << endl;
    EuropeanOption op3(CALL,-1,100,100,.01,.05);
    cout << op3 << endl;
    return 0;
}
```

```
Option Type: CALL
Time to Maturity: 1.000000 unit(s) of time
Spot Price: $100.00
Strike Price: $100.00
Risk Free Rate: 2.50%
Volatility (Sigma): 10.00%
BSM Price: $5.30


Option Type: PUT
Time to Maturity: 0.079452 unit(s) of time
Spot Price: $158.28
Strike Price: $167.50
Risk Free Rate: 0.99%
Volatility (Sigma): 20.48%
BSM Price: $9.92


Error: illegal_value exception. Tau, Spot Price, or Strike Price cannot be below 0.
terminate called after throwing an instance of 'int'
```

Problem 5 (20 points). Add a function to the `EuropeanOption` class that, given an option price, computes the implied volatility of the option using the bisection method. Do the same with the secant method (adding yet another function). Using the provided Options.txt file containing data for European options with different

strike values, compute their implied volatilities using both bisection and secant methods. You should output your results to a new .txt file, containing all the content from the original file plus two new columns with the implied volatilities computed by you. Comment on your results.

Hint 1: Use the *mid price* ([bid + ask] / 2) as the option price in your calculations.

Hint 2: The *DaysToMaturity* column of the Options.txt file represents the number of continuous days (not business days) until maturity

Answer 5. For this implementation of `EuropeanOption`, I add a new `private` member variable called `market_val` which is a `double` that represents the mid-point of the bid ask spread of the traded option. Our `<<` operator also accounts for this new value, which gives us the market price vs the black-scholes-merton price instead of just the calculated BSM price. We also implement a function method `f(double x)` that will be used as a placeholder in the calculation of our implied volatility for both the bisection method and secant method. Speaking of which, I also add two new `public` functions that return a `double` representing the implied volatility of the `EuropeanOption` object using the bisection and secant method, named `impliedVolBisection()` and `impliedVolSecant()` respectively. I used the following **article** to base my implementation. The bounds for the bisection method is the `vol` of the object $\pm$ 0.1. The bounds for the secant method are .25 and 1 with 8 iterations. Below is the code for their implementation.

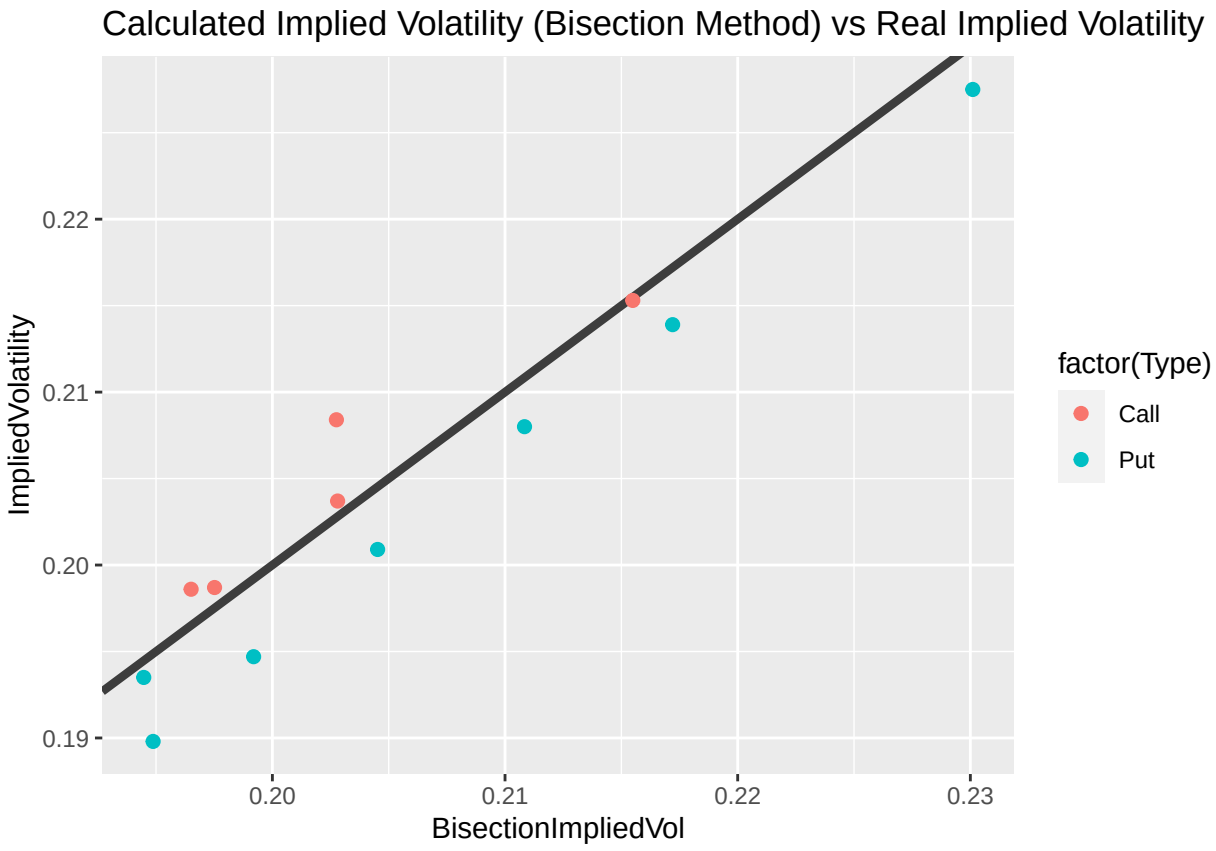
```
double EuropeanOption::f(double x){
    double d1 = (1/(this->vol*sqrt(this->time_to_maturity)))*(log(this->spot/this->strike)
      + ((risk_free + (pow(x,2)/2))*this->time_to_maturity));
    double d2 = d1 - (x * sqrt(time_to_maturity));
    if(this->t){
        return (N(d1)*this->spot) - (N(d2)*this->strike*exp(-risk_free*time_to_maturity))
        - this->market_val;
    }
    return (N(-d2)*this->strike*exp(-risk_free*time_to_maturity))
    - (N(-d1)*this->spot) - this->market_val;
}

double EuropeanOption::impliedVolBisection(){
    double a = this->vol-.1;
    double b = this->vol+.1;
    if(this->f(a)*this->f(b) > 0){
        cout << "We cannot use the intermediate value theorem, root(s) may have a
        multiplicity > 1 " << endl;
        return -1;
    }
    if(a > b){
        a += b;
        b = a-b;
        a -= b;
    }
    double c = a;
    while(b-a > .000001){
        c = (a+b)/2;
        if(this->f(c) == 0){
            break;
        }
        else if(this->f(c)*this->f(a) < 0){
            b = c;
        }
        else{
```

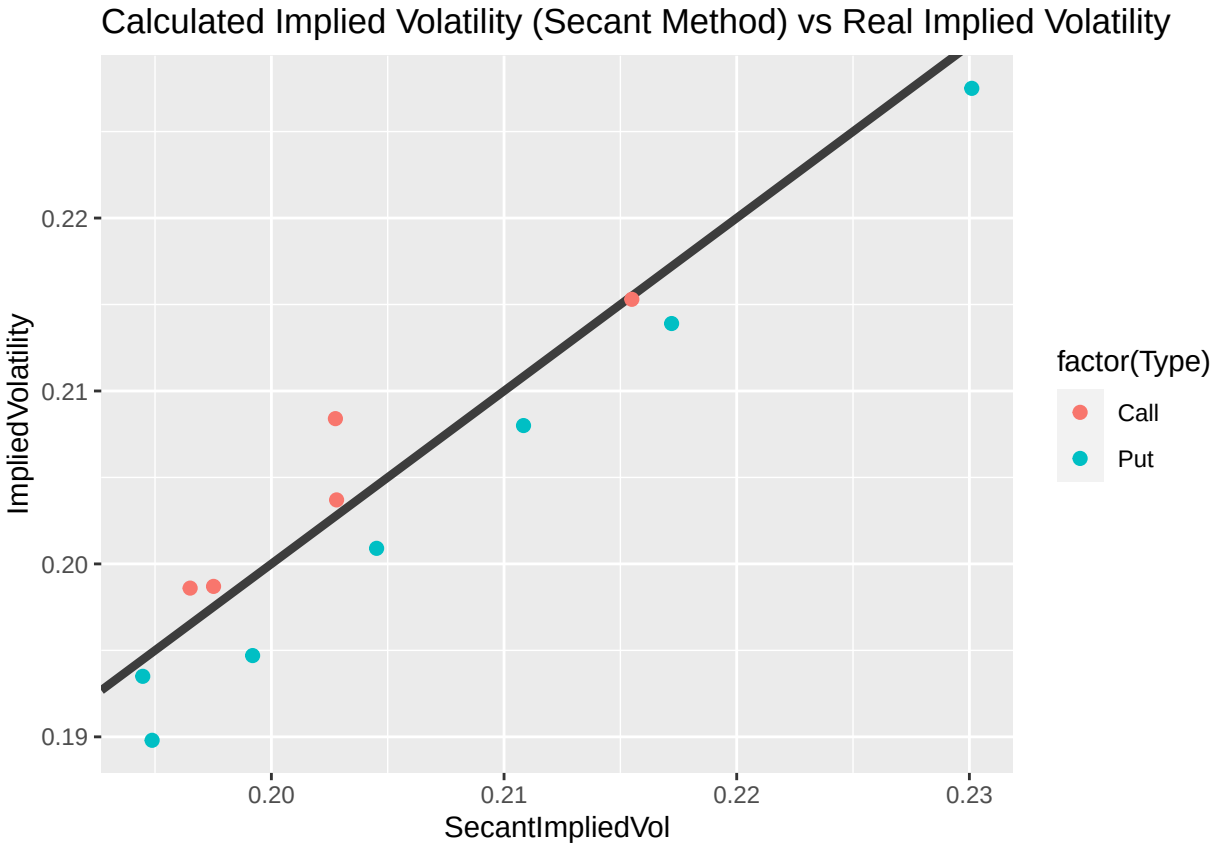
```
            a = c;
        }
    }
    return c;
}

double EuropeanOption::impliedVolSecant(){
    int count = 0;
    double next = 0;
    double a = .25;
    double b = 1;
    while(count < 8){
        if(this->f(b)-this->f(a) == 0){
            return b;
        }
        next = b - (this->f(b)*(b-a)/(this->f(b)-this->f(a)));
        a = b;
        b = next;
        ++count;
    }
    return b;
}
```

Answer 5(cont). As always I will link the header file **here**, the implementation **here**, and the main file **here**. Once implemented, my main function takes in a file called options.txt (provided for the assignment) and parses through the text to create `EuropeanOption` objects, storing them in a `vector`. Afterwards I find the implied volatility using the bisection and secant method for each `EuropeanOption` in my vector and output the original table with every row having the two new implied volatilities appended to them. The output of the text file is in csv format for easy display when put through markdown. Below is a sample of our ouput for our test cases and graphs from our exported text file.

```
Option Type: CALL
Time to Maturity: 0.079452 unit(s) of time
Spot Price: $158.28
Strike Price: $155.00
Risk Free Rate: 0.99%
Volatility (Sigma): 20.84%
Market Price: $5.53 vs BSM Price: $5.62

Volatility Estimate (Bisection Method): 20.2752%
Volatility Estimate (Secant Method): 20.2752%

Option Type: CALL
Time to Maturity: 0.079452 unit(s) of time
Spot Price: $158.28
Strike Price: $160.00
Risk Free Rate: 0.99%
Volatility (Sigma): 19.86%
Market Price: $2.77 vs BSM Price: $2.81

Volatility Estimate (Bisection Method): 19.6507%
Volatility Estimate (Secant Method): 19.6508%

...Collapsed Values...
```

```
Option Type: PUT
Time to Maturity: 0.079452 unit(s) of time
Spot Price: $158.28
Strike Price: $157.50
Risk Free Rate: 0.99%
Volatility (Sigma): 18.98%
Market Price: $3.02 vs BSM Price: $2.94

Volatility Estimate (Bisection Method): 19.4874%
Volatility Estimate (Secant Method): 19.4874%

Option Type: PUT
Time to Maturity: 0.079452 unit(s) of time
Spot Price: $158.28
Strike Price: $162.50
Risk Free Rate: 0.99%
Volatility (Sigma): 19.35%
Market Price: $5.92 vs BSM Price: $5.91

Volatility Estimate (Bisection Method): 19.4471%
Volatility Estimate (Secant Method): 19.4471%
```





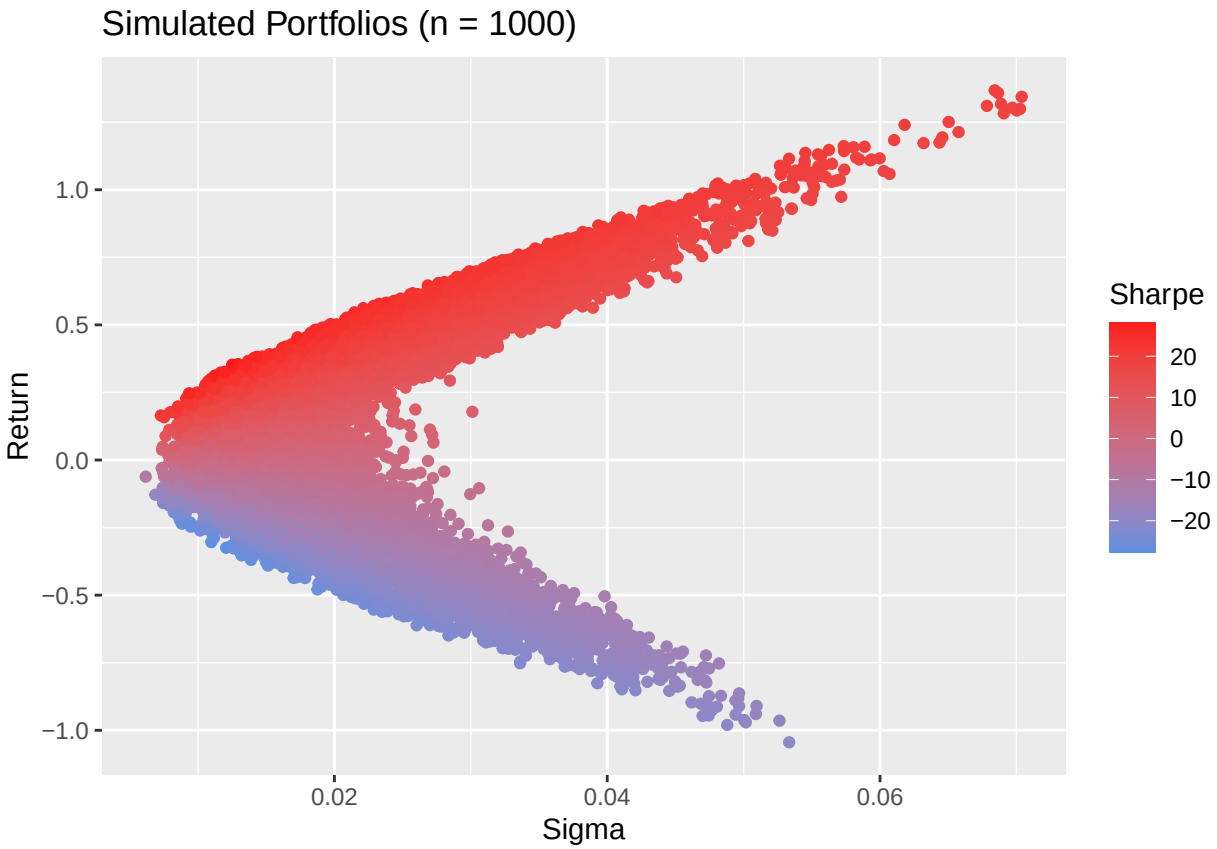
	Type	Bid	Ask	Spot	Strike	Rate	DaysToMaturity	ImpliedVolatility	BisectionImpliedVol
1	Call	5.4500	5.6000	158.2800	155.0000	0.0099	29	0.2084	0.2028
2	Call	2.7300	2.8200	158.2800	160.0000	0.0099	29	0.1986	0.1965
3	Call	1.1800	1.2500	158.2800	165.0000	0.0099	29	0.1987	0.1975
4	Call	0.7800	0.8400	158.2800	167.5000	0.0099	29	0.2037	0.2028
5	Call	0.3500	0.3800	158.2800	172.5000	0.0099	29	0.2153	0.2155
6	Put	0.3700	0.4100	158.2800	145.0000	0.0099	29	0.2275	0.2301
7	Put	0.6100	0.6500	158.2800	148.0000	0.0099	29	0.2139	0.2172
8	Put	0.8600	0.9200	158.2800	150.0000	0.0099	29	0.2080	0.2108
9	Put	1.3400	1.4000	158.2800	152.5000	0.0099	29	0.2009	0.2045
10	Put	2.0400	2.0900	158.2800	155.0000	0.0099	29	0.1947	0.1992
11	Put	3.0000	3.0500	158.2800	157.5000	0.0099	29	0.1898	0.1949
12	Put	5.8000	6.0500	158.2800	162.5000	0.0099	29	0.1935	0.1945

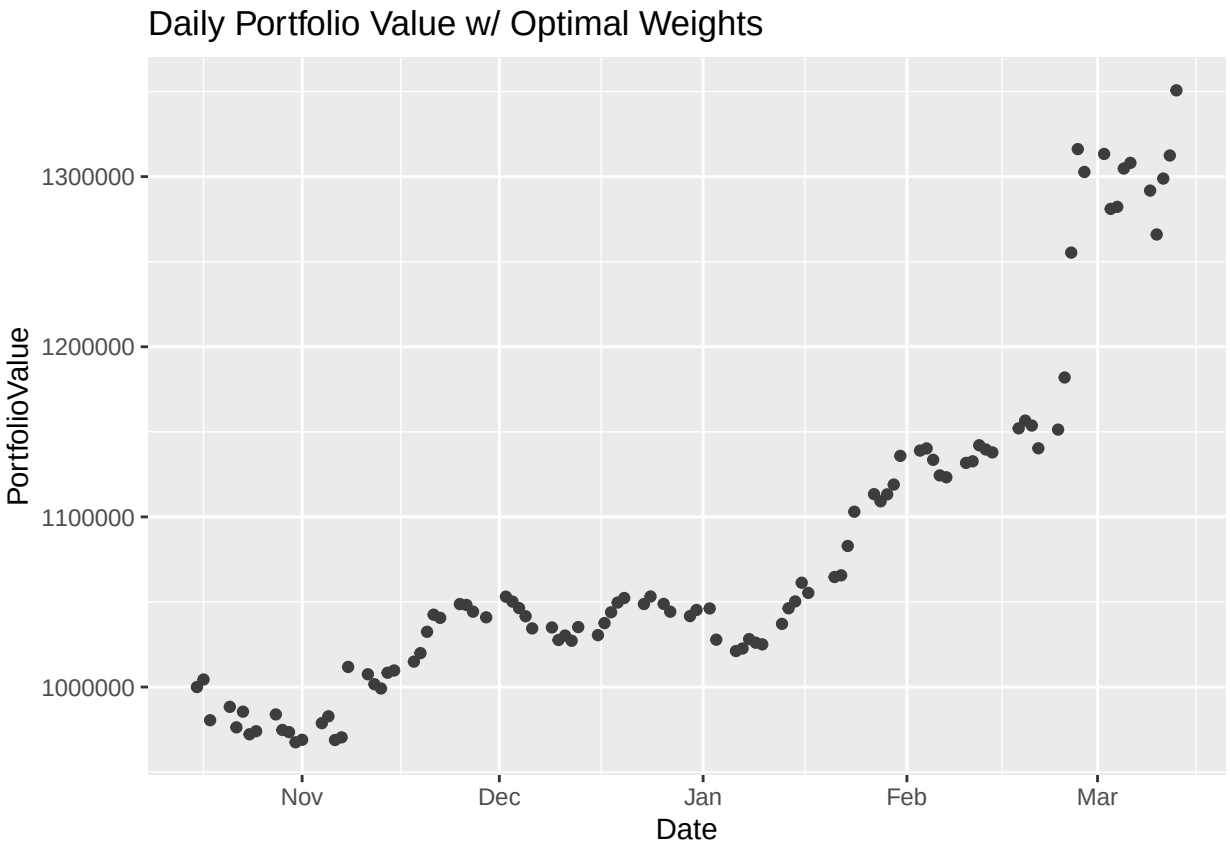
Problem 6 (25 points). Choose 5 or more stocks of your interest and download six months worth of daily stock prices for each stock from Yahoo! Finance (https://finance.yahoo.com) as CSV files. Write a program to pass in the pricing information, then construct a portfolio at the beginning of the 6-month period, consisting of all the stocks you chose. Assume the initial wealth you invest is \$1, 000, 000 and assume you could invest in fractions of a stock, e.g. you could buy 2.46 shares of Netflix (NFLX). **The portfolio weights must be randomly generated.** Calculate the profit if you hold this portfolio for the entire 6-month period without adjusting the portfolio weights. Set different weights to the initial portfolio until you obtain the optimal portfolio. Track the daily values of your optimal portfolio and output them into a CSV file. Use another language (Python, MATLAB, etc) to visualize it. Discuss how you came up with the optimal portfolio.

Note: You should not use maximum return as your objective. Any other definition of "optimal" is encouraged as long as you have your own investment philosophy.

Answer 6. For this part, I first download our six month data with Yahoo! Finance as CSVs and place them in an input folder located in the root project directory. The tickers I chose were **RTN**, **AAXN**, **BRK-A**, **NFLX**, and **VIR**. I generate the random portfolio using similar methods found in Problem 1. To generate a vector of random portfolio weights first I generate 5 random real values. I find their sum and divide by each element by said sum. This causes our vector to equal 1. To remove leveraged weights, we find the sum of the absolute values of the elements in the vector and again divide each element by the sum. We do this because we include negative values in our random number generation to account for short positions. The absolute value of each element in the vector is used so that we know we only use capital that we have and not any leveraged or borrowed capital (ex: weights like <-3,1,1,1,1>). To find the optimal portfolio we will be maximizing our portfolio sharpe ratio. The sharpe ratio is defined as $\frac{E(R_i)}{\sigma_i}$ where $E(R_i)$ is the risk premium of our financial vehicle and σ_i is its volatility. We use the six-month treasury rate of our starting data, which at 10/16/2019 was 1.64%. We conclude we've reached a maximum sharpe ratio when the difference between the current max value and the previous max value is under some noise ϵ, which we put to be 0.004. We also have another exit clause that stops generation after creating 1,500,000 portfolios. To find portfolio return and variance we use the two respective formulas $E(r_p) = \sum w_i r_i$ and $\sigma_p^2 = \sum \sum w_i w_j cov(i,j)$ Because we use the sum of the log returns of each of the assets for our cumulative returns, we have to convert them back to discrete returns to find an accurate portrayal of our daily portfolio value. We can find the main file with everything in it **here**. I also generate 1000 random portfolios and track their return, volatility (σ), and sharpe for graphing purposes. The code below shows an example output of our main file (the actual main file outputs the contents of the inputed CSVs to ensure they get read properly but that is left out).

```
Test #0 New Sharpe: 22.7022
Test #7 New Sharpe: 23.1787
Test #9 New Sharpe: 23.9199
Test #393 New Sharpe: 24.9062
Test #638 New Sharpe: 25.0173
Test #1093 New Sharpe: 25.3689
Test #1791 New Sharpe: 25.4437
Test #2909 New Sharpe: 25.7068
Test #3211 New Sharpe: 26.7392
Test #4490 New Sharpe: 27.4884
Test #717477 New Sharpe: 27.5197

Best_Weights = [-0.468793,0.208904,-0.00500761,0.236627,0.080668,]
Return: 0.343293, Sigma: 0.0124744
```


Simulated Portfolios (n = 1000)
Return
1.0
0.5
0.0
−0.5
−1.0
0.02
0.04
0.06
Sigma
Sharpe
20
10
0
−10
−20


Daily Portfolio Value w/ Optimal Weights
PortfolioValue
1300000
1200000
1100000
1000000
Nov
Dec
Jan
Feb
Mar
Date