FE 522 Assignment 1

Lun Li

October 20, 2019

For all problems below, we centralize utilities in utilities.h *and* utilities.cpp*. There are two categories of utiities, one is miscellaneous utilities, dealing with, for instance, string operations, the other one is math utilities, facilitates various of basic calculations.*

1 Random Number Generator and Its Statistics

The distributions choose are 1) Uniform distribution; 2) Normal distribution; 3) Poisson distribution; 4) Gamma distribution; and 5) Binomial distribution. The sample can be generated through *randomGen* function in *utilities.h* with signature

```
vector<double> mathUtils::randomGen(vector<double> const & parameters, int const& sampleSize, string const& distType)
```

It returns a vector samples if followiong arguments are supplied:

- *paramters* specifies the parametrization of distribution function, e.g., a vector $\{0, 1\}$ of normal distribution stands for a normal distribution with mean 0 and standard deviation 1;
- *sample size* determines the size of sample required to be generated from corresponding distribution;
- *ditType* selects which distribution the user is interested in;

All distributions are available in standard *std* library. As for various statistics calculation, the *statistics* function in *utilities* is implemented, which takes in sample generated and returns the statistics summary. The main code-flow is

Algorithm 1: Number Generator and Its Statistics

```
Input: Global Config $C$, Distribution Config $D_i$, for $i = 1, ..., 5$, outputFile $f$
for Each in Distribution do
  sample = randomGen($D_i$, $C$)
  stats = statistics(sample)
  f $\Leftarrow$ stats
end for
```

The output is outlined below (see *Table 1*), which is verified to be close to theoreticl statistics.

Dist Type	Mean	S.T.D	Median
Uniform	0.500966	0.290359	0.501455
Normal	1.00837	1.50009	1.01124
Poisson	3.9595	2.00949	4
Binomial	1.9951	0.998637	2
Gamma	1.98416	2.0003	1.32536

Table 1: Statistics Table for Random Variable

2 Root Finding

Two main functions are implemented for root finding, one for construction/valuation of polynomial, one for numerical recipes:

1. *Polynomial(var, parameters):* It takes variable x and a list of coefficients of polynomials $\{a_i\}$,

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots a_nx^n \tag{1}$$

2. *Bisection/secant(initialBounds, functor, epsilon, maxIter):* For bisection and secant, intialBounds specify the initial bounds to start off the algorithm, functor is the function pointer wrapped by *std::function*, epsilon is the threshold for successive x_i's and maxIter is the maxmium iterations.

The main program is driven by *inputFile.txt*, which has the following name-value pair structure: the first part specifies the polynomial:

$$0 : a_0, \quad 1 : a_1, \quad \cdots \quad , n : a_n \tag{2}$$

This stands for $p(x) = a_0 + a_1x + \cdots + a_nx^n$. The second parts determines initial guess for respective algorithms:

$$\textit{Bisection Lower Bound} : b_0, \quad \textit{Bisection Upper Bound} : b_1,$$
$$\textit{Secant Lower Bound} : b_2, \quad \textit{Secant Upper Bound} : b_3,$$

The main code-flow reads: The output is summarized in *Table 2*.

Algorithm 2: Root Finding

```
Input: inputFile.txt
for Each line in inputFile do
    coefficients, initialGuess = extractor(line)
    functor = std::bind(polynomial, placeHolder, parameter)
    res1 = bisection(initialGuess, functor)
    res2 = secant(initialGuess, functor)
end for
```

Polynomial	Bisection	Secant
$x^2 - 612 = 0$	24.7386	24.7386
$x^3 - x - 2 = 0$	1.52138	1.52138
$x^3 - 9x^2 + 20x - 13 = 0$	6.04892	6.04892

Table 2: Root Finding for Polynoimal

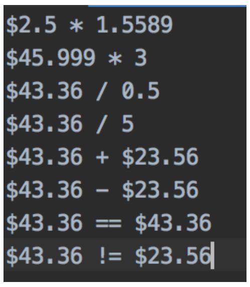
3 Money Class

A money class is implemented as declared below:

```
class money {

public:
    // constructor
    explicit money(string & exRep);
    // overload == operator
    bool operator ==(money const& m);
    // overload != operator
    bool operator !=(money const& m);
    // overload + operator
    money operator +(money const& m);
    // overload - operator
    money operator -(money const& m);
    // overload * operator
    money operator *(double n);
    // overload / operator
    money operator /(double n);
    // overload * operator
    money operator *(int n);
    // overload / operator
    money operator /(int n);

    // friends
    // overload << operator (output)
    friend ostream& operator<<(ostream & os, money const & m);
    // overload >> operator (input)
    friend istream& operator>>(istream & input, money & m);

    // conversion function
    static long strToLng(string & strRep);
    static string longToStr(long longRep);

    // member variables
    long m_internalRep;
};
```

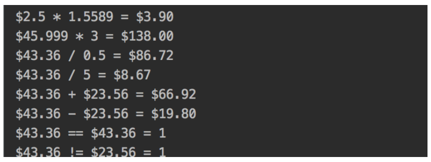
The essential thing in this class is operator overloading, $+, -, *, /, ==, != , <<, >>$. The main idea behind implementation is the following: constructor takes in a string representation of money and internally converts to a long representation, we call it internal representation, recorded by class member $m\_internalRep$. Notice, in case we have dollar string that is more granullar than cents, we use class member function *strToLng* to execute rounding-4-6 and string to long conversion. Now for all algebraic operators, the calculations are all done in terms of internal representation. Since we want to return money object, we convert internal representation, long data type, back to string and use constructor to instantiate the return object. For logic operator, it is even simpler, we just need to overload the operator to have it comparing the internal representation. The I/O operators are also straightforward by passing around internal representation to ostream and istream, respectively.

The simple cases are provided by inputFile.txt with following answer returned:



```
$2.5 * 1.5589
$45.999 * 3
$43.36 / 0.5
$43.36 / 5
$43.36 + $23.56
$43.36 - $23.56
$43.36 == $43.36
$43.36 != $23.56
```

Figure 1: Money Class Test Case



```
$2.5 * 1.5589 = $3.90
$45.999 * 3 = $138.00
$43.36 / 0.5 = $86.72
$43.36 / 5 = $8.67
$43.36 + $23.56 = $66.92
$43.36 - $23.56 = $19.80
$43.36 == $43.36 = 1
$43.36 != $23.56 = 1
```

Figure 2: Money Class Output

4 European Option Class I

The main thrust is to implement the European Option Class:

```
class EuropeanOption {

public:
    // member function
    // constructor 1
    EuropeanOption(string       & optType,
                   double const & spotPrice,
                   double const & strikePrice,
                   double const & interestRate,
                   double const & volatility,
                   double const & timeToMaturity,
                   bool doNotThrow = true);

    // check constraints
    void checkConstraints(bool doNotThrow);
    // get_price
    double getPrice();

    // member variable
    string   m_optType;
    double   m_spotPrice;
    double   m_strikePrice;
    double   m_interestRate;
    double   m_vol;
    double   m_timeToMaturity;
};
```

The constructor takes all necessary information to setup an European option and its calculation, i.e., all member variables are initialized. The constructor is also responsible for checking validity of each parameters, such as, time to maturity cannot be negative. The optional variable *doNotThrow* allows us to suppress error or not. If we throw, the program will break in the middle way, otherwise, it will gives warning but carry subsequent calculation. Such handling of exceptions are wrapped up as *exceptionHandle* that is avilable in *utilities.h*. The *getprice()* function is the analytical core, which basically calculate Black-Scholes value for the option.

The input file is of name-value pair structure, each line specifies one option parameters, e.g.,

optType:call, spotPrice:158.28, strikePrice:155.00, interestRate:0.0099, volatility:0.2084, timeToMaturity:0.5

The main program simply loop through options in the input file and price them one by one, output is shown below:

```
call Option with Strike 155 and time to maturity 0.5 is priced at : 11.3415
call Option with Strike 155 and time to maturity 1 is priced at : 15.4804
```

5 European Option Class II

We implemented another constructor that leaves volatility "uninitialized". This is done by declaring:

static double INF = std::numeric_limits<double>::infinity();

and assign it to member m_vol. Now, in the member function $getPrice()$, it has responsibility to first check if all variables are initialized and make sure their validities. This is achieved by calling member function $assertionVar()$ that utilizes $exceptionHandle()$ utility in *utilities.h*. We add an optional parameter to $getPrice()$ to suppress throwing of exception, if doNotThrow is turned off, it will throw when volatility is unspecified.

We also re-organize code a little bit to minimize duplication and easier for generalization. A new member function is added into class – *blackScholes*, i.e.,

double blackShcoles(double volatility = INF, double spotPrice = INF, double strikePrice = INF, double interestRate = INF, double timeToMatrutiy = INF, string optType = "");

If any parameters is overriden, it will use the overrides otherwise it uses value from class members. When the function is called by $getPrice()$, no overides are required, while when computing implied volatility through $getImpliedVol()$, in the calibration process, we do need to vary volatility,i.e.,

double getImpliedVol(double const& optPrice, string const& method = "bisection", double lower = 0., double upper = 1., double const& epsilon = 1e-6);

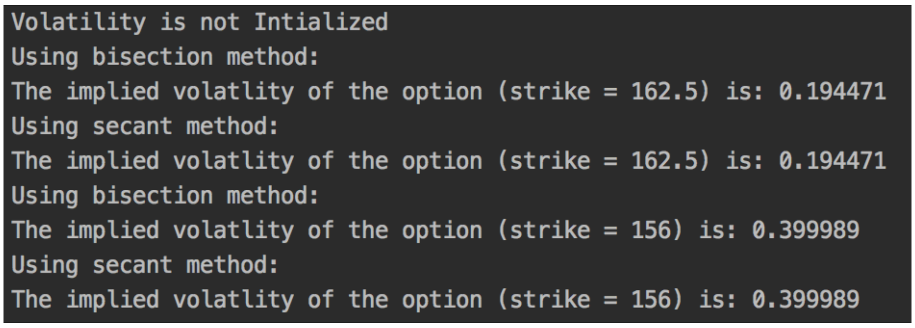
To explain arguments: *optPrice* is the target price to hit, method is default to "bisection", for the rest, they are solver configuration parameters as we introduced in *Section 2*. Notice, we are using the same

Algorithm 3: Get Implied Volatility

```
Input: Target Price $p$, numerical method $m$
// objectiveFun to track error
objectiveFun(vol, p)
    return blackSchole(vol) - p
// get implied volatility
functor $f$ =bind(objectiveFun, placeHolder, optPrice)
**if** m = "bisection" **then**
    return bisection(f)
**else**
    return secant(f)
**end if**
```

root finding function as in *Section 2* and calling them from utilities. To enable class member function $getImpliedVol()$ to use it, we have to use $std :: bind()$, that's why in polynomial root finding problem, we wrap up function pointer to be used here, the alternative for *Section 2* would be using raw function pointer, i.e., $double(*funPtr)(double)$.

Lastly, we provide a strike override API, $setStrike()$, through which user can override value of class member $m\_strikePrice$. After resetting, the implied volatility shall be re-calculated via $getImpliedVol()$. Below is the program output for 1) initialized vol; 2) calculation of volatility with two methods; 3) reset strike and calculation of implied volatility again.



```
Volatility is not Intialized
Using bisection method:
The implied volatlity of the option (strike = 162.5) is: 0.194471
Using secant method:
The implied volatlity of the option (strike = 162.5) is: 0.194471
Using bisection method:
The implied volatlity of the option (strike = 156) is: 0.399989
Using secant method:
The implied volatlity of the option (strike = 156) is: 0.399989
```

Figure 3: European Option Class II Output

6 European Option Class III

This is an application of $getImpliedVol()$ to generate volatility smiles, i.e., $\sigma(K)$, for $K \in \mathcal{K}$. In the inputFile.txt, we are given market quotes bid/ask and other required information to calculate implied volatility per strike, in the meanwhile, the market traded implied volatilities are also presented. In theory, if we reverting the mid-price, we shall get very similar results as market traded level, which we will see in indeed the case later on. Let us illustrate by pseudo code:

Algorithm 4: Generate Volatility Smile

```
Input: Input File $f$, output file $f'$
// open input file and initialize an output file
f.open(), f'.open()
// loop through input file while writing to outputfile
while Not End of $f$ do
    // get schema when reading first line
    if FirstLine in $f$ then
        schema = extract(FirstLine, 1)
        schema' = schema + col(Bisec) + col(Secant) //augmentSchema by two columns
        f' $\Leftarrow$ schema'
    else
        Values = extractor(line, i)
        // use a map to record each argumnts from leach row
        for Each arg, value in Schema, Values do
            map[arg] = value
        end for
        opt = EuropeanOption(map)
        vol = getImpliedVol(map["Target Price"])
        f' $\Leftarrow$ vol
    end if
end while
```

We loop through the input file, for the first line, we extract the schema, augment it by two more columns (bisection vol and secant vol) and write it into the output file as new headers. For each line that has actual data, we use a map to record and feed it into $EuropeanOption$ constructor and $geImpliedVol()$ to invert target price. The results are again written into output file line by line. Below is output and graph of volatility smile.

Opt Type	Bid	Ask	Spot	Strike	Rate	Days To Maturity	Mkt Vol	Bisec Vol	Secant Vol
Call	5.45	5.6	158.28	155	0.0099	29	0.2084	0.20274	0.20274
Call	2.73	2.82	158.28	160	0.0099	29	0.1986	0.196507	0.196507
Call	1.18	1.25	158.28	165	0.0099	29	0.1987	0.197512	0.197512
Call	0.78	0.84	158.28	167.5	0.0099	29	0.2037	0.202801	0.202801
Call	0.35	0.38	158.28	172.5	0.0099	29	0.2153	0.215491	0.215491
Put	0.37	0.41	158.28	145	0.0099	29	0.2275	0.230074	0.230074
Put	0.61	0.65	158.28	148	0.0099	29	0.2139	0.217169	0.217169
Put	0.86	0.92	158.28	150	0.0099	29	0.208	0.210819	0.210819
Put	1.34	1.4	158.28	152.5	0.0099	29	0.2009	0.204508	0.204508
Put	2.04	2.09	158.28	155	0.0099	29	0.1947	0.199185	0.199185
Put	3	3.05	158.28	157.5	0.0099	29	0.1898	0.194873	0.194873
Put	5.8	6.05	158.28	162.5	0.0099	29	0.1935	0.194471	0.194471

Table 3: Volatility Smile Board

As expected, two numerical methods give same results and its smooth, while the market quotes are non-smooth as defined by actual demand and supply(notice, the there is one repeated point for same strike is excluded).

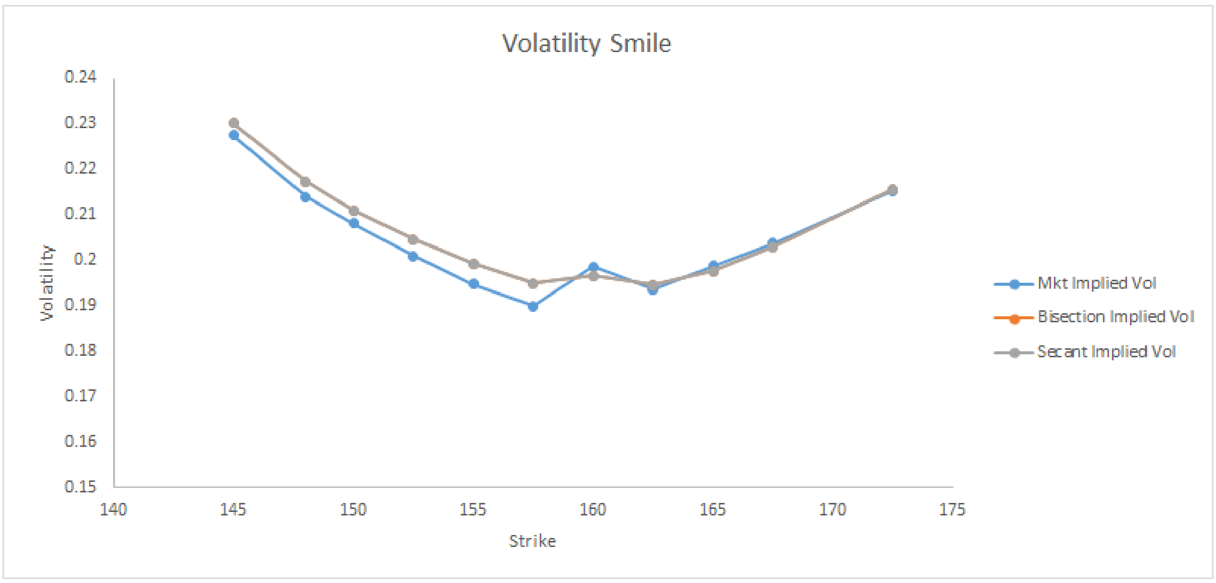


Figure 4: European Option Class II Output