Financial Programming in C++: Homework Assignment 4

Fall 2025,

MSQF, Fordham University

Due: September 25th, 2025

Problem 0

Unzip the file `HW4.zip` it should include the following files:

- **Problem 1**:
  - `discount.h` and `discount0.cpp`
  - `test_discount0.cpp`
- **Problem 2**
  - `discount.h` and `discount0.cpp` (same as in problem 1)
  - `forward.h` and `forward.cpp`
  - `volatility.h` and `volatility.cpp`
  - `black_scholes.h` and `black_scholes0.cpp`
- **Problem 3**:
  - All filles from problem 2.
  - `test_risk0.cpp`
- **Problem 4**: No input files

Problem 1: Swap Present Value

You implemented the swap present value function in **Problem 1** of **Homework 3**:

The present value of a **receiver swap** is given by:

$$\mathrm{pv} = \sum_{i=1}^{N-1} R \mathrm{d}T \mathrm{df}(t_i) + R \mathrm{d}t \mathrm{df}(T) + \mathrm{df}(T) - 1,$$

and present value of a **payer swap** has the opposite sign:

$$\mathrm{pv} = -\sum_{i=1}^{N-1} R\mathrm{d}T\mathrm{df}(t_i) - R\mathrm{d}t\mathrm{df}(T) - \mathrm{df}(T) + 1.$$

In this homework the discount factor will be given by:

$$\mathrm{df}(T) = e^{-r(T)T}$$

where $r(T)$ is linearly interpolated from the input term structure by the function:

```
double discount(double t, int tenors,
                const double *Ts, const double *rs);
```

as implemented in file `discount0.cpp`.

Problem 1.1

Rename file `discount0.cpp` into `discount.cpp`.

Implement in file `discount.cpp` the `swap_pv` function with signature:

```
double swap_pv(bool is_receiver,double R, double T, double freq,
               int tenors, const double *Ts, const double *rs);
```

The implementation is as on homework 3, but using a term structure of interest rates for discounting.

[HINT] Make sure you use a correct implementation of the swap pv formula. You can reuse your own solution for HW3 or the provided instructor's solutions to HW3.

Problem 1.2

Rename the file `test_discount0.cpp` into `test_discount.cpp`.

In `test_discount.cpp`'s `main` function compute the size of the expiries array (`Ts`) in file `test_discount.cpp` using the `sizeof` operator.

Problem 1.3

For each expiry from $T = 0.25$ to $T = 10$ years, inclusive, in quarterly increments of 0.25 years, compute:

- The swap rate for expiry T.
- The swap present value of a receiver swap with annual coupon frequency and expiry T.
- The present value of a bond with coupon equal to the swap rate paid annually and expiry T.

Output to the terminal with the following outputs:

- A header line with the names of the computed fields separated by commas.
- One line per expiry T with the following fields separated by commas:
  - The swap maturity T.
  - The swap rate R for maturity T.
  - The swap present value.
  - The bond present value.

Verify that, for each expiry T, the swap present value for a swap with rate equal to the swap rate is zero.

Verify that the present value of a bond a coupon equal to the swap rate has present value equal to one.

Problem 1.4

Compute and execute the program `test_discount`. Capture its output into file `test_discount.csv`.

Problem 2: Option Greeks

In this problem we will compute option greeks using market consistent inputs.

When level of rates $r(T)$ and volatility $\sigma(T)$ are no longer flat, the option greeks are approximated using *finite differences* as follows:

- **Option Value**:

  $$\text{value} = V(\text{df}, F, \sigma)$$

  where V is the value of the option as implemented by function (in `black_scholes0.cpp`)

  ```
  double bs_price(bool isCall, double K, double T,
                  double df, double F, double sigma);
  ```

  where:

  - df is the discount factor at option expiry T as computed by the `discount` function.
  - F is the forward price at expiry T as implemented by the `forward` function.
  - σ is the volatility of the underlying asset at expiry T as implemented by the `volatility`.

- **Delta**:

  $$\Delta = \frac{V(\text{df}, F_+, \sigma) - V(\text{df}, F_-, \sigma)}{2\text{d}S}$$

  where

  - dS $= 0.01S$, where S is the spot price of the underlying asset.
  - F_+ is the forward (as computed by `forward`) with spot $S + \mathrm{d}S$.
  - F_- is the forward (as computed by `forward`) with spot $S - \mathrm{d}S$.

- **Gamma**:

$$\Gamma = \frac{V(\mathrm{df}, F_+, \sigma) - 2V(\mathrm{df}, F, \sigma) + V(\mathrm{df}, F_-, \sigma)}{(\mathrm{d}S)^2}$$

  where F_+ and F_- are as above.

- **Vega**:

$$\text{vega} = V(\mathrm{df}, F, \sigma + \mathrm{d}\sigma) - V(\mathrm{df}, F, \sigma)$$

  where $\mathrm{d}\sigma = 0.01$.

- **DVO1**:

$$\text{DVO1} = V(\mathrm{df}_r, F_r, \sigma) - V(\mathrm{df}, F, \sigma)$$

  where

$$\begin{aligned} \mathrm{d}r &= 0.0001 \\ \mathrm{df}_r &= \mathrm{df}\, e^{-\mathrm{d}rT} \\ \mathrm{F}_r &= F\, e^{\mathrm{d}rT} \end{aligned}$$

Problem 2.1

Rename the file `black_scholes0.cpp` into `black_scholes.cpp`.

Implement the `bs_risk` function in `black_scholes.cpp` with signature (see header file `black_scholes.h` for details):

```
double bs_risk(bool is_call, double K, double T,
               double S,
               int tenors, const double *Ts,const double *rs,
               const double *ds, const double *sigmas,
               double &delta, double &gamma, double &vega, double &DV01);
```

Problem 2.2

In a option portfolio, each position will consist of number **units** (shares) in an option as defined by its type (call/put), expiry and strike .

The number of units is positive is we own (are long) the option, and negative if we sold (are short) the option.

When computing greeks, we need to scale the risks by the number of units of exposure of each position::

$$\begin{aligned}
\widetilde{\text{value}}_i &= \text{units}_i\, \text{value}_i \\
\widetilde{\Delta}_i &= \text{units}_i\, \Delta_i \\
&\vdots \\
\widetilde{\text{DV01}}_i &= \text{units}_i\, \text{DV01}_i
\end{aligned}$$

where

- i is the index of the option position in the portfolio.
- units_i is the number of units of option i in the portfolio
- value_i, Δ_i, etc are the greeks **per unit** of option i as computed by the function `bs_risk` implemented in Problem 2.1.
- $\widetilde{\text{value}}_i$, $\widetilde{\Delta}_i$, etc are the greeks scaled by the size of the position.

Implement function `bs_risk_portfolio` in `black_scholes.cpp` with signature (see header file `black_scholes.h` for details):

```
void bs_risk_portfolio(
             int noptions,
               const int *units,
               const bool *is_call,
               const double *K,
               const double *T,
               double S,
               int tenors, const double *Ts,const double *rs,
               const double *ds, const double *sigmas,
               double *value,
               double *delta, double *gamma, double *vega, double *DV01);
```

Problem 2.3

The value and Greeks (Δ, Γ, vega, and DV01) of a portfolio are additive (the total portfolio exposure is the sum of the greeks of all its positions).

Implement the function `total_risk` in file `black_scholes.cpp` with signature (see header file `black_scholes.h` for details):

```
double total_risk(
             int noptions, // risk inputs
                const double *values,
                const double *deltas,
                const double *gammas,
                const double *vegas,
                const double *DV01s,
                double &delta,  // risk outputs
                double &gamma,
                double &vega,
                double &DV01);
```

Problem 3: Option Greeks II

In this problem we will implement a `test_risk` program that computes greek exposure of an option portfolio given market data.

0.1 Problem 3.0

Rename the file `test_risk0.cpp` into `test_risk.cpp`.

For the rest of this problem, add your code to the `main` function of file `test_risk.cpp`.

Problem 3.1

In the file `test_risk.cpp` allocate space (statically, no need to use dynamic memory) for the following arrays:

- `value`
- `delta`
- `gamma`
- `vega`
- `dv01`

Make sure each array has enough space to store the greeks of the input portfolio as defined in `test_risk.cpp`.

Problem 3.2

Use the function `bs_risk_portfolio` implemented in Problem 2.2 to compute the greeks of the portfolio.

Problem 3.3

Output the option details (units, call or put, strike, expiry), position value and greeks for each position in the portfolio. One position per line, with fields separated by commas.

Include a header line with the field names.

Problem 3.4

Compute the total value and total greek exposure of the portfolio using the function `total_risk` implemented in Problem 2.3.

Problem 3.5

Output the total value and greeks of the portfolio in a single, comma separated, line.

Include a header line with the names of the reported fields.

Problem 3.6

Compile and execute the program `test_risk`.

Capture the output in a file name `test_risk.csv`.

Problem 4: Dynamic Arrays

In this problem we will create a new executable called `statistics`.

The program will expect a command line argument that specifies the number of data inputs provided by the user.

The command line should be as follows:

```
./statistics <count>
```

where `<count>` is the number of data points to be input by the user.

The program will prompt the user to input `<count>` floating point numbers and report their mean and standard deviation.

Problem 4.1

Create a new file named `statistics.cpp`.

In the `main` function of file `statistics.cpp`:

- test that the number of arguments received from the command line is correct.
- if the number of arguments is not correct, output a clear **error message** describing the correct usage and return immediately with an error code.
- convert the single input argument into an integer value stored in a variable named `count`.

  Use the function `std::stoi` to convert the input string into an integer value.

[**HINT**] Use only the single argument version of `std::stoi`, ignore any default arguments.

Problem 4.2

Using operator `new []` allocate memory for an array named `data` of `count` doubles.

Write a for loop that for `count` times:

1. Prompts the user to input a number in the terminal using `std::cout`.
2. Reads a double value from the terminal using `std::cin` and stores its value in the current element of the `data` array.
3. If the value entered by the user was negative the program should terminate with an informative error message.
4. Accumulate the value of the current element into a variable named `sum`.

Once the loop is complete, compute the mean of the input data as:

$$\text{mean} = \frac{\text{sum}}{\text{count}}$$

Output the mean to the terminal using `std::cout`.

problem 4.3

Compute the standard deviation of the input data as:

$$\text{std} = \sqrt{\frac{1}{N-1}\sum_{i=0}^{N-1}(x_i - \text{mean})^2}$$

where N is the size of the input data array and x_i are the values inputted by the user (stored in the `data` array).

Output the standard deviation to the terminal using `std::cout`.

Program 4.4

Make sure that you free the memory allocated for the `data` array using operator `delete []` along **all control flow paths** of the program.

Program 4.5

Compile and execute the program `statistics`:

What result do you get when you call the program as

```
./statistics 5
```

amd then input the following numbers:

```
1,2,3,4,5 ?
```

write your answer to this question as a comment at the end of the file `statistics.cpp`.

Submissions

You must submit The following files in a zip file to complete your homework:

- **Problem 1**: Files `discount.cpp`, `test_discount.cpp` and `test_discount.csv`.
- **Problem 2**: The file `black_scholes.cpp`.
- **Problem 3**: The file `test_risk.cpp` and `test_risk.csv`.
- **Problem 4**: The file `statistics.cpp`.