General Instructions

1. Download `Practical02.zip` from the course website.

   Extract the file under `[CppCourse]\[Practicals]` folder. Make sure that the file structure looks like:

   ```
   [CppCourse]
      -> [boostRoot]
      ...
      -> [Practicals]
         -> [Practical01]
         -> [Practical02]
            -> Practical01Exercises.hpp
            -> ...
            -> [Src]
               -> ComplexNumber.cpp
               ...
   ```

2. Open the text file `[CppCourse]\CMakeLists.txt`, uncomment the following line by removing the `#`:

   `#add_subdirectory(Practicals/Practical02)`

   and save the file. This registers the project with `cmake`.

3. Run `cmake` in order ot generate the project.

4. The declaration of the functions to be implemented are in `Practical02Exercises.hpp` and in `ComplexNumber.hpp`. Create a `cpp` file for each function and add them to the project.

5. Implement the functions into the newly added `.cpp` file under the `[Src]` folder. Do not modify any of the other files.

6. Compile and run your code. If the minimum requirements are met, an output text file will be created. Hand in your `.cpp` files and the output file

   `Practical02_output.txt`

   via Moodle.

7. The types `CoefficientFunction`, `Equation`, `DVector`, `NumericalStep` and `PayoffFunction` are defined in `Practical02Exercises.hpp`.

Exercise 1

```
double eulerStep(double dVal,
        double dTime,
        const DVector & drivingNoise,
        const Equation & euqation);
```

This function implements the Euler-Maruyama step for a scalar valued SDE driven by a 1-dimensional Brownian motion:

$$dS_t = a(t, S_t) + b(t, S_t)dB$$

The function takes the following arguments

- `dVal`: initial stock price S
- `dTime` time t
- `drivingNoise` contains $(\Delta t, \Delta B)$
- `equation` a vector of function pointers containing the coefficient functions (a, b)

The function returns

$$S + a(t, S)\Delta t + b(t, S)\Delta B.$$

Exercise 2

```
double milsteinStep(double dVal,
          double dTime,
          const DVector & drivingNoise,
          const Equation & euqation);
```

This function implements the Milstein step for a scalar valued SDE driven by a 1-dimensional Brownian motion:

$$dS_t = a(t, S_t) + b(t, S_t)dB_t.$$

The function takes the following arguments

- `dVal`: initial stock price S
- `dTime` time t
- `drivingNoise` contains $(\Delta t, \Delta B)$
- `equation` a vector of function pointers containing the coefficient functions

$$\left(a, b, \tfrac{\partial}{\partial S} b\right).$$

The function returns

$$S + a(t, S)\Delta t + b(t, S)\Delta B + \tfrac{1}{2} b(t, s) \tfrac{\partial}{\partial S} b(t, S) \left[(\Delta B)^2 - \Delta t\right]$$

Exercise 3

```
DVector MonteCarlo3(double dS0,
          double dT,
          double dR,
          Equation const& equation,
          NumericalStep const& numericalStep,
          unsigned long int iNumberOfSteps,
          unsigned long int iNumberOfPaths,
          PayoffFunction const& payoffFunction);
```

`MonteCarlo3()` is a somewhat generalised version of `MonteCarlo2()`. This version takes a vector of coefficient functions (defining the SDE) and the numerical method (e.g. Euler-Maruyama, Milstein).

- `dS0` initial stock price
- `dT` time to maturity
- `dR` risk-free interest rate
- `equation` contains the coefficient functions, e.g. (a, b) or $(a, b, \frac{\partial b}{\partial S})$
- `numericalStep` defines the numerical scheme (Euler-Maruyama or Milstein)
- `iNumberOfSteps` number of steps (of equal length) taken by the method
- `iNumberOfPaths` number of trajectories generated
- `payoffFunction` defines the payoff function

The function returns a vector with two entries. The first entry is the Monte-Carlo estimate, the second entry is the standard deviation of the Monte Carlo estimate (and the standard deviation of a sample).
Note: the header file `Utils/UtilityFunctions.hpp` inside the `utils` namespace contains two functions that one might find useful.

```
    void NormalDist(std::vector<double> &vArg);
```

```
    double NormalDist();
```

The first function takes a vector by reference and fills it up with standard normals. The second function takes no argument but returns one single standard normal variable.

Exercise 4

The class `ComplexNumber` is defined in `ComplexNumber.hpp`. The implementation of some member functions can be found in the lecture notes. Implement all the declared member functions.

Exercise 5

The file `Practical02.cpp` contains the implementation of the function

```
TestStrongConvergence()
```

Have a look and try to figure out what it does.