General Instructions

1. Download `Practical05.zip` from the course website.

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

   ```
   [CppCourse]
      -> [boostRoot]
      ...
      -> [Practicals]
         -> [Practical01]
         -> ...
         -> [Practical05]
            -> BitManipTests.hpp
            -> ...
            -> [Src]
               -> bitManipulation.cpp
               -> ...
   ```

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

   ```
   #add_subdirectory(Practicals/Practical05)
   ```

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

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

4. The header file `Practical05Exercises.hpp` contains the declaration of three functions and a class. Create and add the `.cpp` files under the `[Src]` folder, and implement the exercises into these files.

5. After compiling and running your code - if the minimum requirements are met - an output text file is created:

   ```
   Practical05_output.txt
   ```

6. Hand in the output file and the `cpp` files you created.

7. The files are to be submitted via Moodle.

Exercise 1

The function `Regression()` is one ingredient of the least squares regression based methods for approximating conditional expectations. Consider an economy that is described by k factors $X = (X_1, \dots, X_k)$. Let the set $\{X_{t_0}^{(1)}, \dots, X_{t_0}^{(N)}\}$ be N simulated values of the factors corresponding to time t_0. For each simulated value, we also simulate a possible value at time t_1. Consider a European option with payoff $f : \mathbb{R}^k \to \mathbb{R}$ at time t_1, and let Y denote the vector of simulated payoffs:

$$Y = (f(X_{t_1}^{(1)}), \dots, f(X_{t_1}^{(N)}))^T.$$

Using this data and a set of $\mathbb{R}^k \to \mathbb{R}$ test functions $\{\phi_1, \dots, \phi_r\}$, we aim to estimate the conditional expectation:

$$V(x) = \mathbb{E}\left[f(X_{t_1}) | X_{t_0} = x\right] \approx \sum_{i=1}^{r} \phi_i(x)\beta_i \tag{1}$$

We can use the formula shown in the Numerical Methods 2 lectures:

$$\beta = (\Phi^T\Phi)^{-1}\Phi^T Y$$

where

$$\Phi = \begin{pmatrix} \phi_1(X^{(1)}) & \phi_2(X^{(1)}) & \cdots & \phi_r(X^{(1)}) \\ \phi_1(X^{(2)}) & \phi_2(X^{(2)}) & \cdots & \phi_r(X^{(2)}) \\ \vdots & & \ddots & \vdots \\ \phi_1(X^{(N)}) & \phi_2(X^{(N)}) & \cdots & \phi_r(X^{(N)}) \end{pmatrix}$$

and $\beta = (\beta_1, \dots, \beta_r)^T$.

```cpp
BVector Regression(const BVector & yVals,
         const std::vector<BVector> & factors,
         const FVector & testFunctions);
```

The function takes three arguments

- `yVals` is a `boost` vector of `double`'s, the observed values at time t_1 (that is Y),
- `factors` is an `std` vector of `boost` vector, is observations of the factor values at time t_0, (that is $\{X_{t_0}^{(1)}, \dots, X_{t_0}^{(N)}\}$),
- `testFunctions` is a collection of test functions.

and returns the estimated regression coefficients.
The precise type definitions can be found in the file `Practical04Exercises.hpp`.
For the implementation, use the linear algebra operations defined in the namespace

`boost::numeric::ublas`

In particular, you will need `trans` for transposing matrices, `prod` for multiplying matrices and vectors, `lu_factorize` and `lu_substitute` for solving the linear equation. Do not forget to include the header

`#include <boost/numeric/ublas/lu.hpp>`

Exercise 2

Once we estimated the coefficients of the regression, we can use the formula (1) for pricing. The function `Projection()` implements the formula.

```
double Projection(const BVector & factor,
        const FVector & testFunctions,
        const BVector & coefficients);
```

The function takes three arguments

- `factor` a `boost` vector, describing the factor values at t_0
- `testFunctions` the set of test functions, spanning the estimate
- `coefficients` the regression coefficients, i.e. β

and returns a double, the estimated conditional expectation.

Exercise 3

In practice we can combine the `Regression()` and `Projection()` into a single pricing object. The class `EuropeanOptionPricer` gives an example of this combination. The main idea is to run the regression once, when the object is initialised, store the regression coefficients as a data member, and later re-use them for projection.

```
  EuropeanOptionPricer(const std::vector<BVector> & factorsAt0,
            const BVector & valuesAtT,
            const FVector & testFunctions);
```

The constructor of the class takes three arguments

- `factorsAt0` a set of simulated initial factors (corresponding to time t_0)
- `valuesAtT` a set of possible discounted payoff values at t_1
- `testFunctions` a collection of test functions

The constructor runs the regression, and saves the coefficients into the `boost` vector data member `m_Coefficients`. Furthermore, the constructor also saves the vector of test functions into the data member `m_TestFunctions`. Implement this member function in terms of the `Regression()` global function.

```
  double operator()(const BVector & factorAt0);
```

The `operator()` member, takes one argument, an instance of X; and returns the estimated option value using the regression coefficients and the test functions. Implement this member function in terms of the `Projection()` global function.

Exercise 4

Once we initialised it, `EuropeanOptionPricer` is useful pricing tool. To initialise it, we need a set of simulated factors at t_0:

$$\{X_{t_0}^{(1)}, \ldots, X_{t_0}^{(N)}\}$$

and a set of possible payoff values at t_1

$$\{f(X_{t_1}^{(1)}), \ldots, f(X_{t_1}^{(N)})\}$$

Consider the particular case, when $X_t = (S_t^1, S_t^2)$, such that

$$\begin{aligned} \mathrm{d}S_t^1 &= rS_t^1\mathrm{d}t + \sigma_1 S_t^1 \mathrm{d}B_t^1 \\ \mathrm{d}S_t^2 &= rS_t^2\mathrm{d}t + \sigma_2 S_t^2 \left[\rho \mathrm{d}B_t^1 + \sqrt{1-\rho^2}\mathrm{d}B_t^2\right] \end{aligned}$$

The function `MonteCarlo4()` generates the payoff values, given a grid of initial stock prices.

```
BVector MonteCarlo4(std::vector<BVector> vS0,
          double dR,
          double dSigma1,
          double dSigma2,
          double dRho,
          double dT,
          Function const& payoff);
```

The function takes seven arguments

- `vS0` an `std` vector of pairs of initial stock prices
- `dR` risk free rate, r
- `dSigma1` the volatility σ_1 of the first stock
- `dSigma2` the volatility σ_2 of the first stock
- `dRho` the correlation of the driving Brownian components
- `dT` time to maturity $t_1 - t_0$
- `payoff` the payoff function f

and returns a `boost` vector of discounted simulated payoff values, such that if the `ith` entry of `vS0` is the vector (S_0^1, S_0^2), then the `ith` entry of the returned `boost` vector is $f(S_{t_1}^1, S_{t_1}^2)$, where $S_{t_1}^1$ and $S_{t_1}^2$ are simulated stock price values at time t_1 with initial value S_0^1 and S_0^2 respectively.

Exercise 5a

The `subtract()` is declared as follows.

```
unsigned int subtract(unsigned int a, unsigned int b);
```

The function takes two `unsigned int` variables and subtracts the second from the first. The implementation is to be based on bitmanipulation operations similar to the `add()` function from the lectures.

Exercise 5b

The `swap()` is declared as follows.

```
void swap(unsigned int & a, unsigned int & b);
```

The function takes two `unsigned int` variables by reference, and swaps their values. This is to be done with using bitmanipulation operations only and strictly without any additional temporary variables.

Exercise 5c

The `BitManipTests.hpp` file is prepared for unit tests that are to be designed and implemented by you. Create two-three tests for each of the `subtract()` and `swap()` functions.
Please hand in `BitManipTests.hpp` together with the test functions.