General Instructions

1. Download `Practical04.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]
      -> ...
      -> [Practical04]
        -> BinaryOperators.hpp
        -> ...
        -> [Inl]
           -> Interpolate.inl
        -> ...
        -> [Src]
           -> Practical04.cpp
           -> ...
```

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

   `#add_subdirectory(Practicals/Practical04)`

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

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

4. The solutions are to be implemented into `.cpp` files under the `[Src]` folder as specified below. The `operator()` member function of the class `Interpolate` is to be completed in the `[Inl]\Interpolate.inl`.

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

   `Practical03_output.txt`

6. The following files are to be submitted via Moodle.

   `Practical04_output.txt`

   `BinaryOperators.cpp`

   `Interpolate.inl`

   `SparseVector.cpp`

The following might be useful. Let $D_t = D_{0,t}$ denote the discount function at time 0 with maturity t, then D_t can be expressed in terms of the annualised yield Y_t and the Libor rate L_t as follows:

$$D_t = \exp(-tY_t) = \frac{1}{1+tL_t}.$$

Exercise 1

The following member function of the class `DF2Yield` has been declared in `BinaryOperators.hpp`.

```
1    double operator()(const double dDiscountFactor, const double dTimeToMaturity)
         const;
```

The function takes two arguments

- `dDiscountFactor` a discount factor
- `dTimeToMaturity` the time to maturity

and returns the corresponding annualised yield.
Add the implementation of this function to `BinaryOperators.cpp`.

Exercise 2

The following member function of the class `Yield2DF` has been declared in `BinaryOperators.hpp`.

```
1    double operator()(const double dYield, const double dTimeToMaturity) const;
```

The function takes two arguments

- `dYield` an annualised yield
- `dTimeToMaturity` the time to maturity

and returns the corresponding discount factor.
Add the implementation of this function to `BinaryOperators.cpp`.

Exercise 3

The following member function of the class `DF2Libor` has been declared in `BinaryOperators.hpp`.

```
1    double operator()(const double dDiscountFactor, const double dTimeToMaturity)
         const;
```

The function takes two arguments

- `dDiscountFactor` a discount factor
- `dTimeToMaturity` the time to maturity

and returns the corresponding Libor rate.
Add the implementation of this function to `BinaryOperators.cpp`.

Exercise 4

The following member function of the class `Libor2DF` has been declared in `BinaryOperators.hpp`.

```
1    double operator()(const double dLibor, const double dTimeToMaturity) const;
```

The function takes two arguments

- `dLibor` a spot Libor rate
- `dTimeToMaturity` the time to maturity

and returns the corresponding discount factor..
Add the implementation of this function to `BinaryOperators.cpp`.

Exercise 5

The class `Interpolate` has been declared in `Interpolate.hpp`. This class forms a framework for interpolation of the following type. Given input vectors, $\mathbf{x} = (x_1, \ldots, x_n)$ (assuming $x_1 < \cdots < x_n$) and $\mathbf{y} = (y_1, \ldots, y_n)$ and $\mathbb{R}^2 \to \mathbb{R}$ functions f_1 and f_2, the interpolation is a function F defined by

$$z_i = f_1(y_i, x_i),\ i = 1, \ldots, n$$

$$F(x) = \begin{cases} f_2(z_1, x) & \text{if } x < x_1 \\ f_2(z, x) & \text{if } x_i \le x < x_{i+1} \\ f_2(z_n, x) & \text{if } x \ge x_n \end{cases}$$

where in the middle case, z is the linear interpolation of the values z_i, z_{i+1} over the interval $[x_i, x_{i+1}]$:

$$z = z_i + (z_{i+1} - z_i)\frac{x - x_i}{x_{i+1} - x_i}.$$

Note that the vector $\mathbf{z} = (z_1, \ldots, z_n)$ is computed by the constructor, furthermore the vectors $\mathbf{x}$ and $\mathbf{z}$ are stored as data members in the class `Interpolate`.
This interpolation function is to be implemented in `[Inl]\Interpolate.inl` (where currently you can find a dummy implementation):

```
1  template<typename PreOp, typename PostOp>
2  double Interpolate<PreOp,PostOp>::operator()(double dX) const
3  {
4    return double(); //dummy implementation
5  }
```

Note the special syntax. This is because this class is a template class. The template arguments `PreOp` and `PostOp` are assumed to be function objects with binary operations implemented (e.g. `DF2Yield`, `Yield2DF`). An instance of `PreOp` serves as f_1 and is used by the constructor when $\mathbf{z}$ is computed. The compiler also creates an instance

of `PostOp` and stores it as data member `m_postOp`. This member is to be used as f_2 in the function to be implemented.
Hint:You might consider using the STL algorithm `lower_bound` to find the index i such that $x_{i-1} \leq x < x_i$ given $\mathbf{x}$ and x. For reference, use

`http://www.cplusplus.com/reference/algorithm/lower_bound/`

Notes: The actual template types `PreOp` and `PostOp` are to be specified when a particular instance of `Interpolate` is declared. Template objects behave unusually, the usual separation of the header and the implementation does not work. The simplest solution is to keep the implementation in the header. Template functions and template classes will be covered in details in lecture 8.

Exercise 6

The following declarations are in `SparseVector.hpp`.

```
typedef std::map<unsigned int, double> SparseVector;
SparseVector & operator+=(SparseVector & svArg1, const SparseVector & svArg2);
SparseVector operator+(const SparseVector & svArg1, const SparseVector & svArg2);
SparseVector & operator-=(SparseVector & svArg1, const SparseVector & svArg2);
SparseVector operator-(const SparseVector & svArg1, const SparseVector & svArg2);
SparseVector & operator*=(SparseVector & svArg, const double dArg);
SparseVector operator*(const SparseVector & svArg, const double dArg);
SparseVector operator*(const double dArg, const SparseVector & svArg);
std::ostream & operator<<(std::ostream & os, const SparseVector & svArg);
```

The exercise is about implementing these operators into `SparseVector.cpp` (except for the `<<` and `+=` operators that are already implemented). Use the implementation of the `+=` operator as an example. Also recall the implementation of the global functions defining features of the `ComplexNumber` class of the previous practical.
Note, the vectors are sparse, i.e. the sparse vector should not store zero entries. Therefore, each operation should remove zero entries in the case any of them is created.
Discussion questions.

- Why is this only a naive implementation of sparse vectors? What is missing? What is not safe?
- How could it be improved to implement a proper sparse vector container?