Object-Oriented Programming

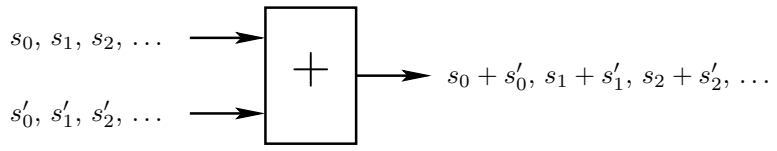
Project Statement

Academic Year 2019-2020

This project can be done by groups of up to two students and must be submitted to `oop@montefiore.ulg.ac.be` *for April 26th at the latest. Projects returned after the deadline will not be corrected. Plagiarism is not tolerated, in line with the policy of the university (*`https://matheo.uliege.be/page/plagiat`*).*

On music CDs and other audio media, sound is usually represented as (large) sequences of *samples*, which are numerical values that approximate the original signal captured at the time of recording. Such sample sequences can be modified using *filters* to induce audio effects such as echo or reverberation.

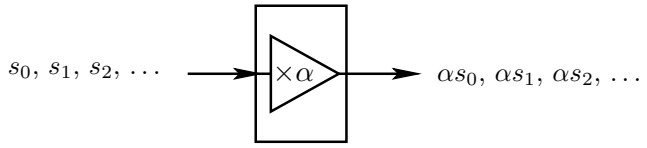
A filter can be defined by means of a *block diagram*, composed of some number of *blocks* interconnected in a specific way. A block has some fixed number of *inputs* and *outputs*. The sample sequences generated at each output are expressed as functions of the sequences present at the inputs. For instance, the *addition block* depicted below computes the sum of two input sequences:



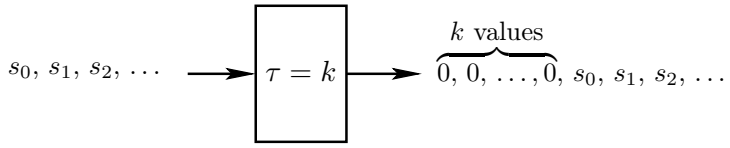
Addition block

The computation performed by such a block is carried out sequentially: the first step is to consider the two input samples s_0 and s'_0 at the head of the input sequences, and to compute their sum $s_0 + s'_0$, which forms the first sample of the output sequence. Then, the same operation is performed on s_1 and s'_1, yielding $s_1 + s'_1$, and so on.

Two additional elementary blocks are defined in a similar way:



Gain block

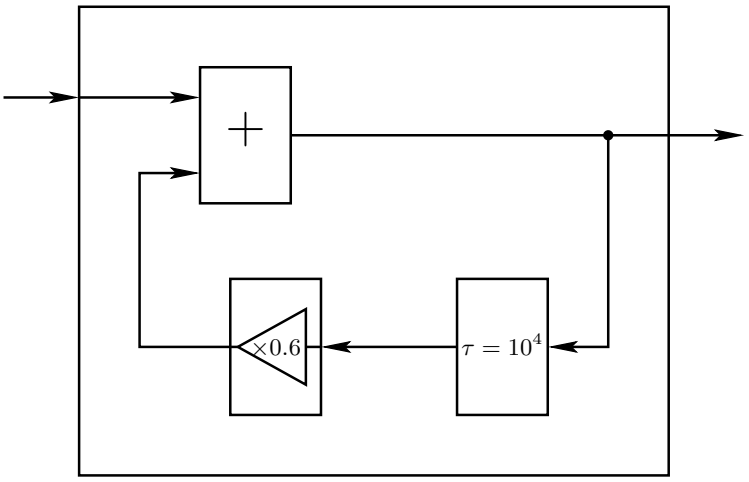


Delay block

The *gain block* multiplies the input samples by a real parameter α. The *delay block* delays an input sequence by k samples, where k is a strictly positive integer parameter. This means that the i-th sample of the output sequence becomes equal to the $(i-k)$-th sample at the input if $i > k$, and to 0 otherwise.

Blocks can be interconnected in order to form a filter, which can itself be represented as a block. The connection rules are as follows: in a given filter, each input of a block or output of the filter must be connected to either an output of a block, or an input of the filter. Filter inputs and/or block outputs can never be connected together. Finally, each *cycle* in such a block diagram, i.e., each path from a point to itself that follows connections and crosses blocks from input to output, must pass through at least one delay block.

An example of valid block diagram is shown below.



This block diagram implements a so-called *echo filter*, with one input and one output. Such a filter could appear as a component of a more complex block diagram.

The goal of this project is to write a Java library for building and using audio filters. A filter will be represented by an instance of a class implementing the following interface.

```
public interface Filter
{
  int nbInputs();
  int nbOutputs();
  double[] computeOneStep(double[] input) throws FilterException;
  void reset();
}
```

The methods `nbInputs()` and `nbOutputs()` respectively return the numbers n_I of inputs and n_O of outputs of the filter. The method `computeOneStep()` takes as an argument an array containing n_I samples (one for each input), performs one step of computation, and returns an array with the resulting n_O samples (one for each output of the filter). This method should throw a `FilterException` when the filtering cannot be carried out (e.g., because inputs are incomplete). The method `reset()` resets the filter, i.e., prepares it for filtering a new sequence.

The first step consists in writing Java classes `AdditionFilter`, `GainFilter` and `DelayFilter` implementing the corresponding elementary blocks. The constructors of `GainFilter` and `DelayFilter` should respectively admit a `double` and an `int` argument, for specifying the value of their underlying parameter.

The second step will be to write a Java class `CompositeFilter` for creating arbitrary combinations of filters. In addition to being a valid filter, this class should implement the following features:

- A constructor taking as arguments the numbers of inputs and outputs of the composite filter.
- A method `void addBlock(Filter f)` for adding a new block `t` to the composite filter.
- A method `void connectBlockToBlock(Filter f1, int o1, Filter f2, int i2)` for connecting the output `o1` of the block `f1` to the input `i2` of the block `f2`.
- A method `void connectBlockToOutput(Filter f1, int o1, int o2)` for connecting the output `o1` of the block `f1` to the output `o2` of the composite filter.
- A method `void connectInputToBlock(int i1, Filter f2, int i2)` for connecting the input `i1` of the composite filter to the input `i2` of the block `f2`.

Notes:

- Inputs and outputs are numbered starting from 0.
- You are free to define as many auxiliary classes as you need.
- All errors should be handled by throwing an appropriate exception.
- In order to be able to test your code, a class `TestAudioFilter` will be provided. It implements a class method `void applyFilter(Filter f, String inputFileName, String outputFileName)` that applies the filter `f` to a WAV file specified by `inputFileName`, and writes the resulting output in a new file named `outputFileName`. This method only uses the first input and output of the filter, and throws an exception (of unspecified type) in the case of an error.
- The project should also provide an example program demonstrating your library by implementing the echo filter depicted p. 2, and applying this filter to a WAV file using the `TestAudioFilter` class.
- Examples of interesting audio filters and WAV files will be published on the exercise sessions webpage.
- Practical details (how to use `TestAudioFilter`, interface of the demonstration program, submission guidelines, etc.) are detailed on the next page.

Practical details

Using `Filter` and `TestAudioFilter` The `TestAudioFilter` class can be used by including the `audio.jar` archive in your project. This archive also provides the `Filter` interface and the `FilterException` class. All these components belong to the package `be.uliege.montefiore.oop.audio`.

You can get `audio.jar` by downloading `project_basis.zip` on the exercise sessions webpage and unzipping it. This archive also includes a class `Example` showing how to use `TestAudioFilter`.

WAV files The exercise sessions webpage will provide a collection of WAV files to experiment with, all of them containing audio content sampled at 44.1 kHz. (In other words, one second of audio is modeled by 44100 consecutive samples.) You can use this knowledge to calibrate your `DelayFilter` objects.

Demonstration Your demonstration program implementing the echo filter should be named `Demo.java` and should accept two command-line parameters specifying the names of the input and output WAV files. Note that your library will be tested with other filters as well.

Submission rules Your project should be submitted as a ZIP archive. At its root, you must place:

- an empty `bin/` folder,
- a `src/` folder with your `.java` source files,
- the `audio.jar` archive.

You are allowed to provide an optional `README` file (as a simple text file or as a `.md` file) to describe additional features of your library, or specific requirements for compiling and running your project. Any file other than those specified here will be ignored. Additional subfolders in `src/` are accepted if you are using packages.

If you do the project by yourself, your archive must be named

```
oop_lastname_firstname.zip
```

where `lastname` and `firstname` are respectively replaced by your last and first names[1]. If you do your project with a classmate, your archive must be named

```
oop_lastname1_lastname2.zip
```

where `lastname1` and `lastname2` correspond to your respective last names, given in alphabetical order.

You are asked to send your archive for April 26th at the latest to `oop@montefiore.ulg.ac.be`. If you work alone, the subject must be

```
OOP - 2020 - lastname firstname
```

If you work with a classmate, the subject must be

```
OOP - 2020 - lastname1 lastname2
```

In both cases, the names must comply to the same naming conventions as for the ZIP archive.

[1] For all names, only the first letter should be in uppercase.