7 Assignment P5 [Assignment ID: `multirate`]

7.1 Preamble (Please Read Carefully)

Before starting work on this assignment, it is **critically important** that you **carefully** read Section 1 (titled "General Information") which starts on page 1 of this document.

You are **very strongly advised** to start working on this assignment as far in advance of the assignment submission deadline as possible. This assignment is not something that can be successfully completed in only a few days before the submission deadline. Start early enough to allow yourself sufficient time to seek help from the instructor (or teaching assistant) when you encounter problems. You have been warned.

In this assignment, **all of your code must be fully commented**. Marks will be deducted for code that is not fully commented. For examples of what constitutes fully commented code, please refer to the source code for the `wireframe` program in Listing D.5 (in Section D.11.1 on pages 512–524) of the textbook (2013-09-26 version).

7.2 Topics Covered

This assignment covers material primarily related to the following: multirate signal processing, integer sampling rate changes, rational sampling rate changes, filter bank, transmultiplexer.

7.3 Part A (Audio Processing)

1. `changeRate` **PROGRAM.** In this problem, we consider sampling-rate changes by an integer factor.
   (a) Write a program called `changeRate` that can perform sampling rate changes by an integer factor. That is, you need to implement M-fold downsampling, upsampling, decimation, and interpolation for arbitrary integer M. Also, you need to implement each of decimation and interpolation with and without using polyphase methods. You should use the `SPL` library to assist you in your solution. The program should consist of a single source code file called `changeRate.cpp`. The command line interface for the `changeRate` program is as follows. The command line takes the following five parameters (in order):
      i. The type of sampling-rate change operation to be performed. The valid values are: `downsample` (for downsampling), `upsample` (for upsampling), `decimate` (for decimation), and `interpolate` (for interpolation).
      ii. An indication of whether polyphase techniques should be employed. The valid values are: `naive` (do not use polyphase methods) and `polyphase` (use polyphase methods). Although this parameter must be specified for all operations, its value is only actually used for the decimation and interpolation operations. That is, the value of this parameter is ignored for downsampling and upsampling operations.
      iii. The integer factor by which the rate will be decreased/increased.
      iv. The name of the input audio file in WAV format.
      v. The name of the output audio file in WAV format.

   For example, to downsample the audio signal in the file `input.wav` by a factor of 4, and write the output to the file `output.wav`, the `changeRate` program would be invoked as follows:

   ```
   changeRate downsample naive 4 input.wav output.wav
   ```

   To decimate the audio signal in the file `input.wav` by a factor of 2 using polyphase-based techniques and write the output to the file `output.wav`, the `changeRate` program would be invoked as follows:

   ```
   changeRate decimate polyphase 2 input.wav output.wav
   ```

   The exit status of the program (i.e., the return value of the `main` function or the parameter passed to the `exit` function) should be zero upon success and one upon failure (i.e., if any error condition is encountered).
   Your code should gracefully handle invalid command line parameters. The program should also print the elapsed time taken for the downsampling/upsampling/decimation/interpolation operation. (Do not include the time required to read/write audio files.) You should use the `chrono` functionality of the standard library for making the timing measurements. You do not need to do anything fancy to handle signal boundaries. (Do not bother with periodic extension or any such schemes.) The antialiasing/antiimaging filters for the decimation and interpolation operations can be designed with the `lowpassFilter` function. (Be careful as to what the passband gain of the filters should be.) You will need to use your judgement in choosing the

filter parameters so that the frequency responses are good enough to avoid any noticeable distortion. Do not overdesign the filters used in the program. It should be possible to meet the specified MSE requirement (for $M \leq 4$) with filters having length less than 3000. **Do not exceed this length.**

To test the decimation and interpolation parts of your code, you should (amongst other things) run the program twice, performing M-fold interpolation followed by M-fold decimation with $M = 4$, and then compare the resulting audio signal to the original signal. In terms of mean-squared error (MSE), the two audio signals should be very close. By "very close", the MSE should be less than 4×10^{-6}. In terms of what your ears hear, the two audio signals should sound identical.

NOTE: Since, when grading assignments, it is practically difficult to distinguish between code that is somewhat incorrect and extremely incorrect, the MSE criterion is likely to be weighted heavily in the marking scheme. That is, **not meeting the MSE requirement will likely result in a significant mark penalty**.

NOTE: For information on how to measure the difference (in terms of MSE) between audio signals contained in WAV files, refer to the document in the file `audioCompare.txt` on the course web site.

NOTE: For testing purposes, several audio files are available from the course web site, including `heart1.wav`, `heart2.wav`, `heart3.wav`, `linkinpark1.wav`, and `voice1.wav`. It is advisable that you utilize all of these files in the testing of your code.

NOTE: For information on how to play audio files on the computers in the lab, refer to the document in the file `using_audio_capabilities.txt` on the course web site.

(b) Let x denote the audio signal in the file `heart1.wav`. Using the `changeRate` program, downsample x by a factor of 4 and also decimate x by a factor of 4. That is, run the commands:

```
changeRate downsample naive 4 heart1.wav output1.wav
changeRate decimate naive 4 heart1.wav output2.wav
```

Listen to both of the resulting audio signals. Describe what you hear. Explain why you hear what you do. Use sketches of signal spectra to assist in your explanation.

(c) Let x denote the audio signal in the file `heart1.wav`. Using the `changeRate` program, upsample x by a factor of 8, and also interpolate x by a factor of 8. That is, run the commands:

```
changeRate upsample naive 8 heart1.wav output1.wav
changeRate interpolate naive 8 heart1.wav output2.wav
```

Listen to both of the resulting audio signals. Describe what you hear. Explain why you hear what you do. Use sketches of signal spectra to assist in your explanation.

(d) Let x denote the audio signal in the file `heart1.wav`. Using the `changeRate` program, decimate x by a factor of 8 using each of the naive and polyphase approaches. That is, run the commands:

```
changeRate decimate naive 8 heart1.wav output1.wav
changeRate decimate polyphase 8 heart1.wav output2.wav
```

Compare the time required for performing decimation in the naive and polyphase cases (being careful to explain if the observed results are to be expected and why). For the purposes of making timing measurements, make sure that your code is compiled with optimization enabled (via the `-O` option to `g++`).

(e) Let x denote the audio signal in the file `heart1.wav`. Using the `changeRate` program, interpolate x by a factor of 8 using each of the naive and polyphase approaches. That is, run the commands:

```
changeRate interpolate naive 8 heart1.wav output1.wav
changeRate interpolate polyphase 8 heart1.wav output2.wav
```

Compare the time required for performing interpolation in the naive and polyphase cases (being careful to explain if the observed results are to be expected and why). For the purposes of making timing measurements, make sure that your code is compiled with optimization enabled (via the `-O` option to `g++`).

7.4 Part B (Audio Processing)

1. **TRANSMULTIPLEXER SOFTWARE.** In this problem, a transmultiplexer will be implemented and used to combine multiple audio signals into a single signal and then split them apart again. Two separate programs will be developed: 1) `multiplex`, which performs multiplexing; and 2) `demultiplex`, which performs demultiplexing. If you would like to share any common code between these two programs, you should use the files `mux.hpp` (for header information) and `mux.cpp` (for non-header information) for this purpose.

(a) Write a program called `multiplex` that takes M (where $M \geq 2$) audio signals, read from files in WAV format, and combines them into a single signal, written to a file in WAV format. The program should consist of a source code file called `multiplex.cpp` and possibly the source code files `mux.hpp` and `mux.cpp` as mentioned earlier. The command line interface for the `multiplex` program is as follows. The command line specifies the following (M+1) parameters (in order):

i. The name of the output file (in WAV format) to which to write the multiplexed audio signal.

ii. The names of the M (where $M \geq 2$) input files (in WAV format) from which to read the signals to be multiplexed together. Each of the M file names is specified as a separate parameter. The frequency bands in the multiplexed signal should be assigned in increasing order, starting with the first signal to be multiplexed.

For example, to multiplex the audio signals in the files `sig1.wav`, `sig2.wav`, and `sig3.wav` into a single signal to be output to the file `mux.wav` (with the signals from `sig1.wav` and `sig3.wav` occupying the lowest and highest frequency bands in the multiplexed signal, respectively), the `multiplex` program would be invoked as follows:

```
multiplex mux.wav sig1.wav sig2.wav sig3.wav
```

The exit status of the program (i.e., the return value of the `main` function or the parameter passed to the `exit` function) should be zero upon success and one upon failure (i.e., if any error condition is encountered).

All input signals must be sampled at the same rate. (Your program should check for this.) The input signals do not necessarily need to have the same number of samples, however. If the input signals do not all have the same number of samples, you should use the first N samples of each signal, where N is the number of samples in the file with the least number of samples. (In effect, this will force all of the signals to have the same number of samples, namely N.) No fancy handling of finite-extent signals is required (i.e., you do not need to use periodic extension or any other such method). The transmultiplexer does not need to be an exact PR transmultiplexer. You do, however, need to choose the various analysis/synthesis filters such that the transmultiplexer system does not introduce any noticeable distortion. The `lowpassFilter`, `highpassFilter`, and `bandpassFilter` functions should be used to obtain suitable filters. Do not overdesign the filters used in the transmultiplexer. For $M \leq 4$, it should be possible to meet the specified MSE requirement with filters having length less than 3000. **Do not exceed this length.**

Write a program called `demultiplex` that takes a multiplexed signal produced by the `multiplex` program and splits the multiplexed signal into its original constituent signals. The program should consist of a source code file called `demultiplex.cpp` and possibly the source code files `mux.hpp` and `mux.cpp` as mentioned earlier. The command line interface for the `demultiplex` program is as follows. The command line specifies the following (M+1) parameters (in order):

i. The input file (in WAV format) containing a multiplexed signal generated by the `multiplex` program.

ii. The names of the M files that are to store each of the demultiplexed signals. Each of the M file names is specified as a separate parameter. The file names are specified starting with the name of the file to hold the signal that occupies the lowest frequency band in the multiplexed signal.

For example, to demultiplex the audio signal in the file `mux.wav` consisting of 3 multiplexed signals, into the audio signals in the files `sig1.wav`, `sig2.wav`, and `sig3.wav`, the `demultiplex` program would be invoked as follows:

```
demultiplex mux.wav sig1.wav sig2.wav sig3.wav
```

The exit status of the program (i.e., the return value of the `main` function or the parameter passed to the `exit` function) should be zero upon success and one upon failure (i.e., if any error condition is encountered).

NOTE: Several audio files for testing purposes (including `heart1.wav`, `heart2.wav`, `heart3.wav`, `linkinpark1.wav`, and `voice1.wav`) can be obtained from the course web site.

(b) Test your `multiplex` and `demultiplex` programs with the following test cases:

i. Three signals to be multiplexed given (in order) by `heart1.wav`, `voice1.wav`, `linkinpark1.wav`.

ii. Four signals to be multiplexed given (in order) by `voice1.wav`, `heart1.wav`, `heart2.wav`, `linkinpark1.wav`.

For each test case, confirm that each of the signals obtained after demultiplexing is the same as its corresponding original. By "the same", the MSE should be less than 4×10^{-5}. (Again, please refer to the document `audioCompare.txt` for information on how to measure the difference between two audio files in terms of MSE.) Also, confirm (with your ears) that the signals obtained after demultiplexing are perceptually indis-

tinguishable from the original signals before multiplexing. For each test case, try carefully listening to the multiplexed audio signal. Describe what you hear. Does your observation make sense? Explain. Use sketches of signal spectra to assist in your explanation.

NOTE: Since, when grading assignments, it is practically difficult to distinguish between code that is somewhat incorrect and extremely incorrect, the MSE criterion is likely to be weighted heavily in the marking scheme. That is, **not meeting the MSE requirement will likely result in a significant mark penalty**.