13 – Learning libsndfile

Université Jean Monnet – Saint-Étienne 2026/2027

Objective

In this exercise, you will use the `libsndfile` library to build a simple audio gain processor.

The program will:

1. Open an audio file.
2. Read its samples using `sf_read_float`.
3. Store the samples in a `std::vector<float>`.
4. Multiply every sample by a constant gain.
5. Clip samples that exceed the valid audio range.
6. Write the processed samples to another audio file.

The input and output files should contain floating-point samples in the range approximately $-1 \leq x \leq 1$.

Prerequisite

Download this archive. It contains a `CMakeLists.txt` to build the exercises and link against the `libsndfile` library.

Problem 1 Opening an audio file

The library `libsndfile` defines two types: `SF_INFO` and `SNDFILE`.

Before opening a file, a variable of type `SF_INFO` and a variable of type `SNDFILE *` should be instanciated:

```
1 SF_INFO info{};
2 SNDFILE * file = nullptr;
```

Open an audio file for reading using:

```
1 SNDFILE* sf_open (const char *path, int mode, SF_INFO *sfinfo) ;
```

Consult the documentation to determine:

- which constant should be passed as `mode` to open a file for reading;
- the arguments expected by `sf_open`;
- how to detect whether opening the file failed.

Questions

1. What type is returned by `sf_open`?
2. How can you check whether the file was opened successfully?
3. Where does `libsndfile` store information such as the number of channels and sample rate?

Print the following information after opening the file:

- sample rate;
- number of channels;
- number of frames.

For example:

```
Sample rate: 44100 Hz
Channels:    2
Frames:      123456
```

Problem 2 Reading audio samples

The function

```
sf_count_t sf_read_float (SNDFILE *sndfile, float *ptr, size_t items) ;
```

can be used to read audio samples as `float`s.

The samples are interleaved. For example, for a stereo file:

```
L0 R0 L1 R1 L2 R2 ...
```

where each pair corresponds to one audio frame.

Knowing that the number of samples in a file is: $N_{\text{samples}} = N_{\text{frames}} \times N_{\text{channels}}$, create a vector large enough to contain the complete audio file:

```
std::vector<float> samples(...);
```

Use `sf_read_float` to fill the vector.

Questions

1. Why is the number of samples different from the number of frames?
2. For a stereo file containing 1000 frames, how many `float` values are required?
3. What does `sf_read_float` return?

Check that the number of samples actually read is what you expected.

Problem 3 Applying gain

Add a gain parameter to your program.

For example:

```
float gain = 2.0f;
```

Every sample should be multiplied by this value: $y[n] = g\,x[n]$ where g is the gain. For example, if $x[n] = 0.25$ and $g = 2$, then $y[n] = 0.5$.

Modify every element of your `std::vector<float>`.

Exercise

Try the following gain values:

- $g = 0.5$
- $g = 1.0$
- $g = 2.0$
- $g = 5.0$

Listen to the resulting files.

What happens when the gain is greater than 1?

Problem 4 Clipping

After applying gain, some samples may fall outside the valid range: $-1 \leq x[n] \leq 1$.

For example:

```
0.8 * 2.0 = 1.6
```

The resulting value cannot be represented as a normal normalized audio sample without exceeding the expected range.

Write a function `void clip(std::vector<float>& audio);` that performs hard clipping:

$$y[n] = \begin{cases} -1 & \text{if } y[n] < -1, \\ y[n] & \text{if } -1 \leq y[n] \leq 1, \\ 1 & \text{if } y[n] > 1. \end{cases}$$

Questions

1. What is the difference between gain and clipping?
2. Why does clipping potentially introduce distortion?

Problem 5 Writing the processed audio

You now need to create a new audio file. Use `sf_open(...)` again, but this time open the file for writing.

You will need an `SF_INFO` variable describing the output file. The output should preserve the input file's:

- sample rate;
- number of channels;

Choose a suitable WAV format. For example, the `SF_INFO` structure may contain:

```
SF_INFO output_info{};

output_info.samplerate = info.samplerate;
output_info.channels = info.channels;
output_info.format  = ...;
```

Consult the `libsndfile` documentation to determine the appropriate format constant.

Write the samples using:

```
sf_count_t sf_write_float(SNDFILE *sndfile, float *ptr, size_t items);
```

Remember that `sf_write_float` expects a number of frames, not necessarily a number of individual samples.

Questions

1. How many frames should you pass to `sf_write_float`?
2. How do you check whether writing the file succeeded?

Problem 6 Closing the files

Once you have finished reading or writing, close the files with:

```
int sf_close(SNDFILE *sndfile);
```

Make sure that both the input and output files are eventually closed.

Problem 7 Complete program structure

Your final program should have approximately the following structure:

```
#include <sndfile.h>

#include <iostream>
#include <vector>
```

```
5
6  int main()
7  {
8      // 1. Open input file
9
10     // 2. Read audio information
11
12     // 3. Allocate std::vector<float>
13
14     // 4. Read samples using sf_read_float
15
16     // 5. Apply gain
17
18     // 6. Clip samples
19
20     // 7. Open output file
21
22     // 8. Write samples using sf_write_float
23
24     // 9. Close files
25
26     return 0;
27 }
```

Problem 8 Error handling

Your program should handle at least the following errors:

- The input file cannot be opened.
- The output file cannot be created.
- The number of samples read is unexpected.
- Writing the output file fails.

`libsndfile` provides functions for obtaining error information.

Investigate:

```
1  const char* sf_strerror(SNDFILE *sndfile);
```

and use it to print a useful error message when an operation fails.

For example, an error message could look like:

```
1  Could not open input.wav:
2  System error : No such file or directory.
```

Problem 9 Command-line arguments

Modify your program so that it can be invoked as:

```
./gain input.wav output.wav 2.0
```

where:

- `input.wav` is the input file;
- `output.wav` is the output file;
- `2.0` is the gain.

Use `argc` and `argv` to retrieve these arguments.

Your program should display a usage message if the wrong number of arguments is provided.

For example:

```
Usage: ./gain <input> <output> <gain>
```

Problem 10 Optional improvement: process the file in blocks

The previous implementation loads the entire audio file into memory.

This is convenient, but it is not necessary.

Instead, you can process the audio in blocks using:

```
std::array<float, 4096> buffer;
```

The general idea is:

1. Read a block with `sf_read_float`.
2. Apply the gain to the samples in the block.
3. Clip the samples.
4. Write the block with `sf_write_float`.
5. Repeat until the complete file has been processed.

This approach has the advantage that memory usage is independent of the length of the audio file.

Challenge

Rewrite your program so that it never stores the entire audio file in a `std::vector`.

Instead, use a fixed-size buffer such as:

```
std::array<float, 4096> buffer;
```

Be careful with the fact that the final block may contain fewer samples than the buffer can hold.