Real Time Signal Processing: Report for the Autotune Project

Brice CONVERS[a], Simon LAURENT[b]

[a]*Grenoble INP Phelma SICOM, brice.convers@grenoble-inp.org,*
[b]*Grenoble INP Phelma SICOM, simon.laurent@grenoble-inp.org,*

Abstract

This report explores in detail the use of the *rtaudio-4.1.1* API, focusing on the *duplex* file. In the field of real-time computation, we look at dynamic allocation and saving of data, and the creation of an autotune, notably with the addition of a circular buffer. We then take a closer look at sound synthesis, highlighting the importance of phase for optimal coherence.

1. Introduction

The aim of this report is to provide a concise but thorough summary of our approach. We will detail our implementation choices and the results obtained, while trying to provide as many interpretations as possible.

2. Basic API operation

2.1. *Uses of the audioprobe and duplex files*

We began by studying the basic operation of the *rtaudio-4.1.1* API. Once the project had been compiled, we turned our attention to two files in particular: *duplex* and *audioprobe*.

The purpose of the *audioprobe* file is to probe the audio capabilities of our operating system. We can extract information such as the version of the API and operating system, the audio peripherals available, the number of audio outputs, the audio formats supported, and the different sampling rates supported. In our case, we have chosen to use a single output, as in this project we are not really interested in stereo sound. And we choose a sampling rate of **48000 Hz**.

Then for the *duplex* file, its native role is to receive the audio data transmitted by the microphone and send it to the output devices. It can be headphones or loudspeakers. We operate in real time, which means that the program can read and write audio data at the same time during execution. This is the file we are going to use for our audio processing, to apply transformations to our microphone signal.

2.2. *Uses of the inout function*

In the file, we have a function *main* which, among other things, is responsible for executing an infinite loop for acquiring sound information in real time. More specifically, within this loop, the *inout* function is a callback function used by the *rtAudio-4.1.1* library to process audio data in real time. It is called whenever there is enough audio data to fill the buffer.

More specifically, the *inout* function checks whether a buffer overflow has occurred. It extracts the necessary data structures from the data pointer. It then saves the data in the input buffer for later use. It calls the *autotuneProcess* function to apply the transformations required for autotune. The *inout* function then copies the processed signal to the output buffer, allowing you to hear the effects of processing in real time. Finally, the function saves the data in the output buffer for final display at the end.

3. Writing a File - Observing the Evolution of Variables

3.1. *Dynamic allocation*

Note that we do our dynamic allocations in the main loop called *main*, before the start of the infinite loop. This is important, as it allows us to allocate arrays for each variable only once. If this allocation were to be made in *inout*, it could lead to memory overflow.

3.2. *Data backup with the buffer_dump function*

To save our data from the buffers, we have written a function called *writeBuffer* in the *main* after the *inout* output to save all the data from processing. The data is saved in a binary file. The next step is to run a python script to convert these binary files into graphs and audio files.

4. Signal analysis

4.1. *Experimental Visualization*

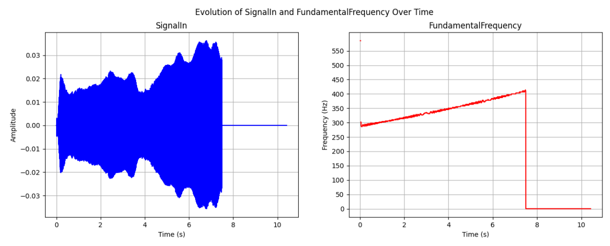


Figure 1: Input signal (SignalIn) on the left and measurement of fundamental frequency as a function of time on the right for a linear frequency variation from 290 Hz to 410 Hz. Figure name : [Q11]SignalInFundamentalFrequency_buffer_2.png

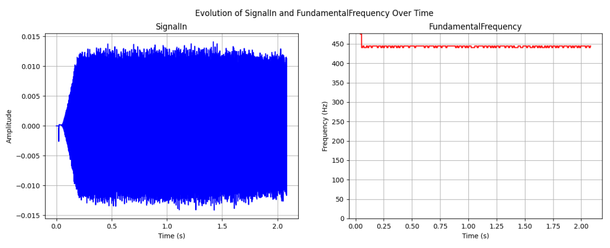


Figure 2: Input signal (SignalIn) on the left and measurement of fundamental frequency as a function of time on the right for a pure tone at 440 Hz. Figure name : [Q11]SignalInFundamentalFrequency_const.png

In this section, we wanted to test our program's calculation of the fundamental frequency of our input signal. To do this, we input a pure tone signal whose fundamental frequency varies linearly with time. We also carried out tests with a pure tone at a constant fundamental frequency (440 Hz). The results can be seen in Fig. 1 and Fig. 2. We can observe very good precision of the order of a tenth of a Hertz, which is very satisfactory for our autotune. However, we can also observe slight frequency fluctuations on Fig. 1. In section 4.3 we will look at a circular buffer implementation to improve the results.

4.2. *Fundamental Frequency Theoretical Studies*

The fundamental frequencies that can be measured are limited. The lower limit is linked to the sampling frequency and buffer size. In fact, the minimum frequency that can be measured corresponds to the inverse of the maximum measurable auto-correlation time. This frequency is given by equation 1, with F_s the sampling frequency equal to 48000Hz, N_b the size of our buffer equal to 512. Subsequently, tests will be carried out with these values.

$$f_{0_{min}} = \frac{1}{\frac{\frac{N_b}{2}}{F_s}} = \frac{2 \cdot F_s}{N_b} \tag{1}$$

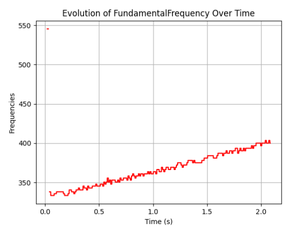
To measure lower frequencies, we can play with the buffer size or sampling frequency, but this remains limited.

The maximum frequency that can be measured is linked to Shannon's sampling theorem. This frequency is given by the equation 2.

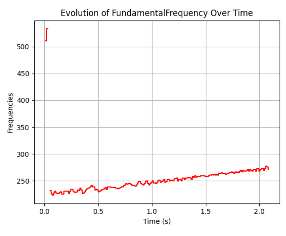
$$f_{0_{max}} = \frac{F_s}{2} \tag{2}$$

4.3. *Circular Buffer implementation*

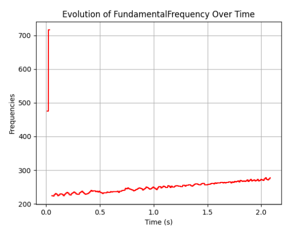
In this section, we'll display various curves with the addition of a circular buffer. We will vary the size of the circular buffer in order to observe the consequences. We have chosen not to systematically include the input signal with the curves, as the latter is generally very similar. We use more or less the same input signal each time, and as we do not have pure sound, this temporal signal does not necessarily provide a great deal of information. However, a complete plot is given in Fig. 3. We can notice high values for each test at the begining. The reason is due to the sound click generated by the keyboard. We need to pay attention to the ordinate scale in our interpretation because of this phenomenon.



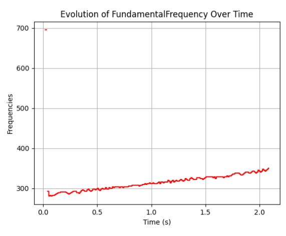
(a) Ring Buffer size is 2



(b) Ring Buffer size is 10



(c) Ring Buffer size is 20



(d) Ring Buffer size is 100

Figure 3: Display of fundamental frequency calculation for different circular buffer sizes. Figure name : [Q12]FundamentalFrequency_i.png

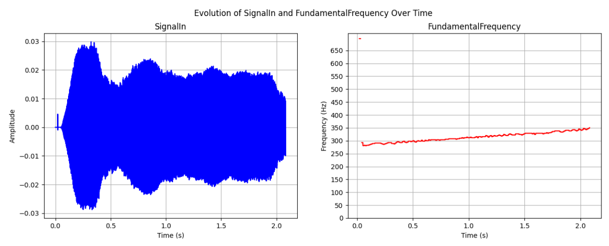


Figure 4: Display of the fundamental frequency calculated from the same input signal (SignalIn), which varies linearly in frequency. For this example, the ring buffer size has been set to 100. Figure name : [Q12]SignalInFundamentalFrequency_100.png

The purpose of adding a ring buffer is to better estimate the fundamental frequency in our program. Instead of calculating it relative to the current buffer, the aim is to average the different fundamental frequencies found over time. The larger the size of the ring buffer, the more we average over the past. With the results in Fig. 3, we can see that with the ring buffer implementation, we can smooth out a little the micro-fluctuations in the fundamental frequency. The larger the ring buffer, the closer the noisy curve should be a straight line.

Ultimately, the role of the ring buffer implementation is to smooth out fundamental frequency fluctuations in order to improve the performance of our autotune. However, if we look at Fig. 3, we can see that its role remains very moderate. Next, as we are going to test our project on speech, we must not average over too long a time window. We will take a size equal to **5**, which corresponds roughly to **53 ms**.

4.4. *Harmonic Analysis*

Once we have our fundamental frequency value, we will calculate the DFT of our signal. The bin corresponding to our fundamental frequency in the calculated DFT is given by equation 3.

$$nf_0 = \lfloor \frac{f_0 \cdot n\_fft}{F_s} \rfloor \tag{3}$$

The DFT of a cosine calculated by FFT gives us two peaks of amplitude $\frac{n\_fft}{2}$ whereas, in theory, we would expect two peaks of amplitude $\frac{1}{2}$, so we need to correct the amplitudes obtained by dividing by *n_fft*.

5. Signal Synthesis

5.1. *First Implementation*

Additive synthesis produces a signal by combining cosines with amplitudes and frequencies corresponding to each harmonic of the signal to be reconstructed. We thus have the equation 4.

$$x(t) = \sum_{n=0}^{\infty} 2A_n \cos(2\pi n f_0 t) \tag{4}$$

We can then visualize our synthesis. To begin with, we record a pure tone at a constant 440 Hz. We can visualize the sound input and output of our program in Fig. 5.

In the example shown in Fig. 5, we are still using the same previous settings.

The first thing we notice in Fig. 5 is that the synthesis does not seem to completely match the look of the original signal. We have more variation in the amplitude of our temporal signal, although this remains slight. What is more, when we listen to this synthesis

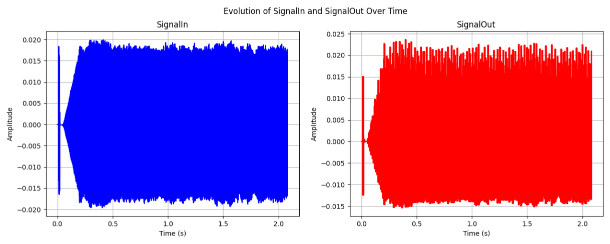


Figure 5: Input signal on the left (SignalIn) and signal synthesis on the right (SignalOut) for re-synthesis of a pure tone at 440Hz without phase recalculation. Figure name : [Q15]SignalInSingalOutWithoutPhase.png

(SignalOut) we recognize a pure sound, but we also have a phenomenon of repetitive high-frequency cuts that strongly disrupt the sound. We can also observe this cutoff by zooming in on Fig. 5.

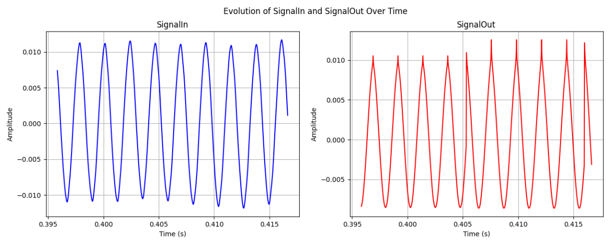


Figure 6: Zoomed display of the input signal on the right (SignalIn) and the synthesized signal on the left (SignalOut) for re-synthesis of a pure tone at 440Hz without phase recalculation. Figure name : [Q15]SignalInSingalOut_zoomed_phase.png

In Fig. 6, we can clearly see the phase problem at time $t = 0.405s$. We have a discontinuity that explains what we are hearing. This discontinuity occurs with each new buffer, i.e. approximately every **10 ms** with our settings. We can also observe that the part of the signal made up of maximums has the shape of a spike. This can probably be explained by the fact that we have a finite number of harmonics. However, we do not understand why the minimum part of the signal does not have the same shape.

We also tried to compare the frequency of our pure tone between SignalIn and SignalOut. We recalculate the fundamental frequency of our synthesized signal with the Numpy python function: *fft.rfftfreq*. Our audio output sounds at **468 Hz**, which is a deviation of **6%** from the signal input. We always obtain this deviation, even after several trials. The interpretation of this result is that, as we do not take phase into account, we have a phase shift of the harmonics in the signal. This will change the fundamental frequency.

5.2. Ajout de la Phase

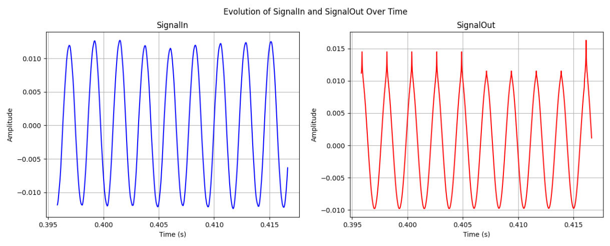


Figure 7: Zoomed display of input signal (SignalIn) on the left and signal synthesis on the right (SignalOut) for pure 440Hz sound. Figure name : [Q17]SignalInSingalOut_zoomed_with_phase.png

In the previous synthesis, we noticed that in our sound synthesis there were abrupt and periodic cuts. We were able to identify that the reason was a phase problem. As we are synthesizing buffers one after the other, we need to re-adjust the phase to ensure sound coherence throughout the recording.

From a theoretical point of view, this means calculating the phase with equation 5.

$$\phi_n^{(i+1)} = \phi_n^{(i)} + 2\pi n f_0^{(i)} \Delta t \tag{5}$$

In equation 5, we need to calculate the phase for buffer i+1 as a function of the phase at buffer i and the fundamental frequency f_0 in Hz of buffer i. In addition, we need to ensure that Δt is in seconds. To do this, we need to recalculate Δt from the digital current index and sampling frequency, as follows: $\Delta t = current_index/F_s$.

Once we have our phase, we can carry out the synthesis with equation 4, which becomes equation 6.

$$x(t) = \sum_{n=0}^{\infty} 2A_n \cos(2\pi n f_0 t + \phi_n) \tag{6}$$

With this new synthesis, we have a very similar result over a trace of several seconds. However, when listening, we no longer have this clipping phenomenon. And when we measure the actual frequency of the output signal, we get **444 Hz**, which is almost exactly what we expected.

In Fig. 7, we can see that we now have perfect transitions between each buffer. For example, we can take a closer look at $t = 0.405s$ and compare with Fig. 6.

6. Autotune

To create autotune, we have combined all the steps described above, and added a fundamental frequency change. This step is implemented with the *changeFundamentalFrequency* function, and is called before synthesizing the output signal. We can also set the number of semitones we wish to correct. Briefly, to apply autotune to the signal, we need to change the signal unity from Hertz to semitone. Then, we round our signal according to the semitone parameter.

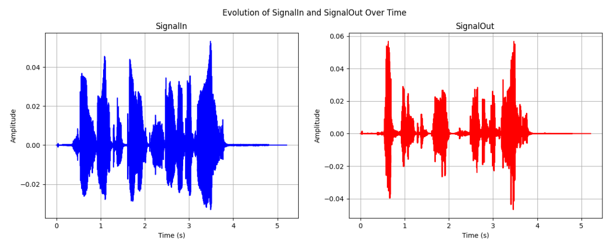


Figure 8: Display for an autotune test on a phrase. The input signal (SignalIn) is on the left and the signal synthesis is on the right (SignalOut). Figure name : [Q18]SignalInSingalOut.png

7. Conclusion

When we listen to our output signal, we can hear the effect of our autotune. However, the sound quality is not ideal, due to the fact that we take a finite number of harmonics when synthesizing the signal. Implementing the circular buffer has enabled us to optimize our signal slightly.

References

[1] Repository GitHub pour le projet Autotune, Brice Convers, Simon Laurent
[2] Le site web de l'API RtAudio par Gary P. Scavone,