Systémes Numériques 2^{nd} year - SNum-2
2023 - 2024

Digital Signal Processing Project

Autotune generation

Robin Gerzaguet
robin.gerzaguet@enssat.fr

•
https://gitlab.enssat.fr/rgerzagu/2a_sysnum_dsp_project_autotune
•

École Nationale Supérieure de Sciences Appliquées et de Technologie
Université de Rennes
Technopole Anticipa • Lannion

Contents

Chapter 1

Positioning and objectives

1.1 Positioning

The purpose of the project is to apply audio processing techniques to generate, modify and synthesize audio signals. The application will be embedded on a Zybo z7 with an ARM Cortex A9 processor. The board has an audio input and an audio output. In a first step, the processing will be done on synthesized signal whose samples are stored in the memory of the board. In a second step, you can directly apply the processing technique on the audio signal recorded from the misc !

The project will be iterative. For each iteration, you will first have a **design** part in a high level language (Matlab) and an **implementation** part on the Zybo board. You will have to secure one iteration before going to the second one.

Secure iteration

A secure iteration is when the code on the Zybo works and is validated by the technical manager (i.e the teacher). You can go to the next iteration only if the current iteration is validated.

1.2 Project steps

The purpose of this Digital Signal Processing (DSP) is to study and implement different signal processing algorithms dedicated to autotune generation and audio synthesis in an embedded devices. The project will be done into several parts

1. **Iteration 1**: Pure audio signal analysis.
   In this iteration, you will work on raw audio signals. You will have to take in charge your Matlab development flow and the Zybo development flow. On the simulation part, you will investigate the time-frequency characteristic of synthesized audio signals. On the implementation part, you will have to check that you can store/load signals and use the audio interfaces of the board.
   Duration: 4 hours
2. **Iteration 2**: Pitch synthesis.
   In this second part, you will have to generate a synthesized signal that emits a "pitched" sine wave. After studying how to do this in Matlab, you will have to implement the pitch generation on the Zybo board and be able to listen to the synthesized (and robotized) audio signal. All the processing have to be in FP !
   Duration: 8 hours
3. **Iteration 3**: Pitch detection and pitch shifting

In the third iteration, you will have to detect the presence and the location of a pitch and shift it to a grid of pre-determined pitch. This is how a (very basic) autotune will work. You will have to autotune synthesized signal and implement this on the Zybo board. If you test with a recorded signal, you may encounter some artefacts.
Duration: 4 hours

4. **Iteration 4**: Enhanced autotune
In the last iteration, you will have to refine the previous algorithm to have a much better autotune, with better result on real signals. Every described refinement will give you extra point, but remember that only implemented algorithms on the Zybo will count !
Duration: 4 hours

1.3 Evaluation

The evaluation will be done from a report at the end of the project

- Each validated iteration give some points.
- Only implementation part count. The audio quality will be assessed
- The final report will be in English and will be evaluated as a part of the work.
- The project will be a GIT project and you have to put on the master branch a Tag for each iteration.

1.4 Project structure

The project structure is constituted by several folder with the following structure

- `Articles`: This folder contains usefull sources related to autotune analysis. The different references of this document are present in this folder.
- `Base_Sound`: In this folder you can find `.wav` files associated to synthesized or recorded audios that can be used for Matlab study.
- `Base_Sound_C`: In this folder you can find `.c` and the `.h` files associated to synthesized or recorded audios that can be used for embedded processing.
- `Zybo_PTS_Application`: This is where you have to embed your processing to be run in the ARM ! This folder is detailed in the ARM processor chapter.

Chapter 2

Matlab & Zybo methodology

2.1 Analysis under Matlab

You are now massively familiar with Matlab development. Matlab is a high level language that allows rapid prototyping and easy exploration. It is mainly dedicated for signal processing, image processing and telecommunication and many of its feature is granted through the use of proprietary toolboxes.

2.1.1 Signal study in Matlab

In the project you mainly have to be able to process audio signals. These signals are stored as `wav` signal. This format is a little special and the signal can be loaded in Matlab with the command

```
1 matlab> (sig,fs) = audioread("my_file.wav");
```

Note that there are two output parameters, the signal `sig` and the sampling frequency `fs`. The code you will have to develop should adapt itself to the recorded frequency so use parameters instead of hardcoded values.
In your report you will have to produce some graphics of the signal (especially the time frequency grid, explained afterwards). For each graphics be sure you have labels (for both x axis and y axis) `xlabel('');` and `ylabel('');`, a title (`title('')`) and legend (`legend('')`. If you are plotting an heatmap, be sure you add the bar (z axis) to your plot with `colorbar();`. For any of these functions you should read the documentation first using `help` or `doc` followed by the name of the function.

2.1.2 Fixed point in Matlab

The processing you will do on the Zybo board will only use Fixed Point arithmetic. It means that some of quite simple functions will require a specific implementation when on an embedded device. You want to apply some kind of filtering (with an Infinite Impulse Filter (IIR)) on your audio signal ? In Matlab, it can be easy with `filt(b,a,x)` where `b` is the denominator of the IIR, `a` is the numerator of the filter and `x` is the signal to filter. On the ARM processing system, you will have to implement the complete filter in Fixed Point domain. Not impossible (you have done this in the Lab!) but far from trivial and a little time consuming.

Advice 1: Explore before making choices

Analysis is build for that. When you implement a processing, you have to think of what comes next. If you can tolerate complicated processing blocks in the analysis, you cannot rely on too much sophisticated and un-implementable blocks. Try, test, and check if the processing is required. If it is, find the less complex version of it. Does a 32 IIR filter taps required when you can have a 16 taps FIR ?

Advice 2: Prototype before implement

Coding and debugging in C can be tricky sometimes, especially on a remote board as the Zybo is. You have to implement your processing in fixed point domain and you can do this on a first stage on Matlab (same way it is done in the lab). You should have some version of the processing that takes only `Int16` inputs, and returns `Int16` outputs with all the Fixed Point operations. It will be easier to implement the C side and also easier to debug (for instance, you can compare the output of the C operations and the Matlab Int operations).

Advice 3: Don't be too greedy

Processing blocks can be split into smaller pieces. Implement pieces per pieces, and validate all pieces

- On Matlab side by testing different signals
- On the Zybo side by validating blocks after blocks

2.2 Implementation on ARM device

After each iteration done in analysis domain you wil switch to the implementation part on the Zybo board. At this stage you should have a working processing block in Matlab, tested on several files. You should have defined the input parameter and buffers, and the output parameters and buffers. You should also have all the Fixed Point format and even a Fixed Point implementation in Matlab. You are now ready to implement them on the embedded board used in the project.

2.2.1 Zybo architecture

The board used in this project is a Zybo board based on the Zynq 7010 architecture. This architecture is a System On Chip (SoC) taht will be studied in details on next year. A SoC is a mixed architecture that contains

- A Processing System (PS) based on a Cortex A9 ARM processor. This is where you will work as all the processing units will be implemented in C language and run on this GPP. For this you will use a Software Development Kit (SDK) that will compile the software you will write for the Zybo and run the code on the board.
- A Programmable Logic (PL), a FPGA that can be synthesized to apply different task. In our project the FPGA is used to have some interface with the Zyboo buttons and to have interface with audio input and audio output. As an additional note, some of the processing you will embed in the software part (in the processor) can be done more efficiently in the hardware part: it would for example be the case of the Fast Fourier Transform that could be synthesized on the PL and used as a co-processor.
- A large DDR3 memory that will be used to store the samples (offline mode) and record the samples (online mode). This memory is shared with the different interfaces through Direct Access Memory (DMA)

You can find on Figure 2.1, the design used on this project. You do no need to have a deep look into as most of the IP used on this chain will make sense next year.
As you can see, the design is constituted of several blocks, synthesized in hardware. The processor is materialized by the `processing_system` IP while the memory access is constituted by the AXI DMA and the AXI interconnect. You can also see the buttons and the audio peripherals for both record and playback. As a side note, this design is important as we have to be able to have the interfaces at the software level. From the programmer perceptive, you have have some libraries and code that will allows you to pilot the interface (audio and buttons) in a similar fashion as you have done on the micro-controller project last year.

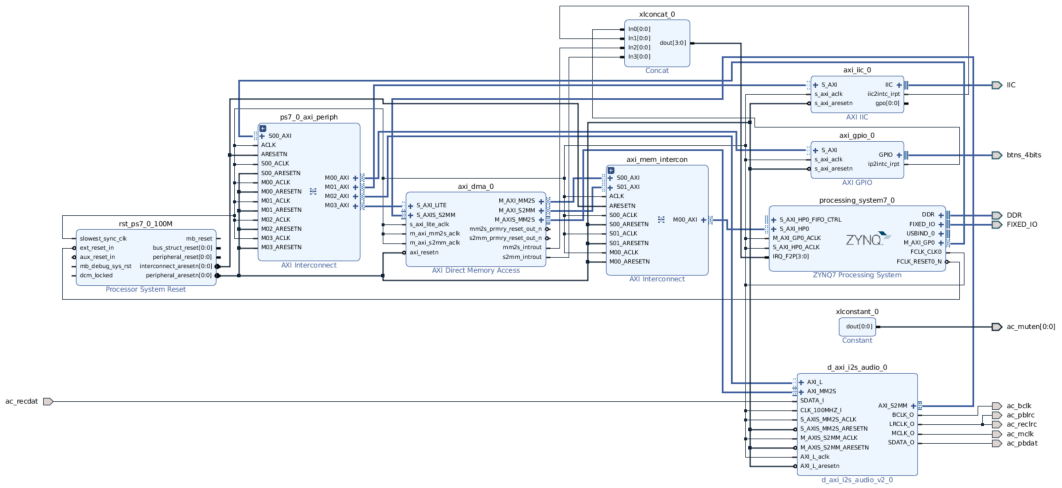


Figure 2.1: Proposed SoC design in the project scope

2.2.2 Using Vivado and SDK

Be sure you are on Linux and open a terminal. Source the setup script on the top of the folder

```
1 users@¬/Project_PTS:./launch_project.sh
```

If the file cannot be evaluated run a `chmod +x source_project.sh` in the terminal first. Then move into the application directory

```
1 users@¬/Project_PTS: cd Zybo_PTS_Application
```

It should open a graphical user interface with a panel with all the files and an Eclipse based interface you should be comfortable with. In this folder you have several sub-folder associated to the hardware exportation (description of the board) and the minimal operating system we deploy to be able to run our application (board support package). The application we develop is in the folder `Zybo-Z7-10-DMA/src`. In `Zybo-Z7-10-DMA/src`, we have sevreal subfolder described here

- `audio`: in this folder there are the C files required to capture and play audio buffers with DMA
- `dma`: in this folder you can find the utilities associated to the memory management (DMA fill and flush)
- `iic`: in this folder there are the I2C drivers
- `fixFFT`: in this folder you can find a Fixed Point implementation of the FFT (up to 1024). It takes 2 arrays as input (real and imaginary parts) and returns an inplace FFT.
- `platform`: in this folder is described the platform driver associated to the SoC design
- `printers`: in this folder you can find some basic function to print arrays, that can be useful for UART debugging
- `timers`: in this folder you can find the functions associated to the `tic()` and `toc()` routines, for benchmark your code
- `userio`: in this folder you can find the buttons and LED drivers

In addition to these folders, you have also the following files

- `main.c`: the main file for the source code and a basic example
- `main.h`: prototype associated to `demo.c`
- `processing.c`: where you should put the processing function you will define
- `processing.h` associated to `processing.c`
- `data_file.c` is the file for a audio signal, associated with `data_file.h` for the global extern. You can change the content of this file by any c files in the `Base_Student_c`. Be sure that the file is entitled `data_file.c` or that the `main.c` include the correct header file if it is not.

The code you will have to write should run on the Zybo board which has a cortex A9 (ARM 32 bits). As a consequence, you cannot use the local compiler of your computer (such as with the linux command `gcc`) as it would compile for the computer chip (which belongs to the x86_64 family). Thus, in order to be able to generate an `elf` file for the Zybo you have to use a **cross-compiler**, embedded in the SDK.

Error in SDK launch

If no window appears and you get the following message

```
log4j:WARN No appenders could be found for logger ...
    (org.eclipse.jgit.util.FS).
log4j:WARN Please initialize the log4j system properly.
```

It means that something is wrong with the SDK metadata. Navigate in the `Zybo_PTS_Application` folder and remove the `.SDK` folder

```
users@¬/Project_PTS: cd Zybo_PTS_Application
users@¬/Project_PTS: rm -rv .sdk
```

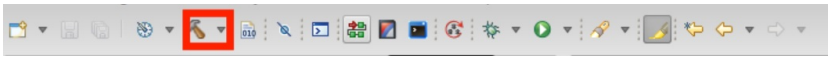
If you can an empty window, then click on `import project` and import all the projects of folder `Zybo_PTS_Application`

Have a closer look on `main.c` and more especially the header of the file. Note that you will have two modes associated to the offline mode, the online mode and the benchmark mode.

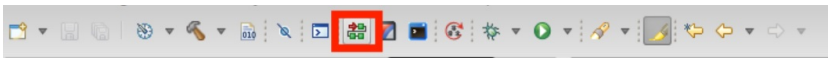
```
/************************** Run mode  ******************************/
#define RUN_MODE 0
// 0 ==> Offline mode: Processing (autotune) on pre-loaded signal ...
    (data_file.c)
// 1 ==> Online mode : Record signal, perform autotune and then render audio
// 2 ==> Benchmark mode : benchmark autotune on pre-loaded signal
/*******************************************************************/
```

To run an application on the Zybo board, several step should be done sequentially. All these steps are important and will be explained in detailed in *System On Chip* lecture.

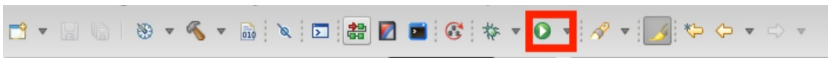
1. be sure that your application can compile first. To do so click on the hammer icon on the top left on the SDK. Have a look on the terminal log to explore any warning or compilation error.

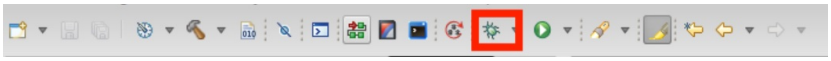


2. Second step is to program the FPGA. For us it will allow to instantiate the DMA (used to store data from the recording part) and the I2S interface for the microphone and the speakers.



3. The application can be run. You can use two modes: the run mode and the debug mode. For the run mode use the green arrow and choose run > launch on hardware. If you want to see the `printf` routines, be sure an UART session is launched independently (see Section 2.2.3). If you want to be able to place breakpoints, use the `debug mode` with the green spiders.





2.2.3 Debugging and Remote interactions

When testing and debugging an application it can sometimes be difficult to monitor and debug each component of your algorithm. In order to better test your code and control that everything has a correct behaviour you can do the following steps

1. Launch execution and display some intermediate variables in the terminal that you will see with a remote connection. This is definitely a "debug by printf" and should used with caution and with parsimony.
2. The second step is to directly use the debugger (the spider button in the interface). By setting breakpoints at the appropriate place, all variables, memories, buffers and registers can be monitored to be sure that everything is done as you want.
3. The last possibility is to send data from the ARM processor to a remote Matlab session. By using some functions provided in `printers.c` you will be able to directly get vectors from the ARM to a Matlab session, which can be of interest to test that the computing is done as you want (for instance the filtering or the Shannon energy computing). Note that exchanging data through UART can be quite long so send small arrays only when you feel that it can validate your work.

First to have a remote interaction in order to see the different `printf` you will add, you need to setup an UART connection between your PC and the Zybo board. First you need to know what is the name of the UART port you are using. In a new terminal, type

```
1 users@¬/Project_PTS: ls /dev/tty* | grep USB
```

This should return the path of the USB device. Then you can open an UART session with

```
1 users@¬/Project_PTS: screen /dev/ttyUSB1 115200
```

In order to connect the UART session with a Matlab session, you must first close the Screen session (with <ctrl-A> \). In the Matlab folder you have a sample files that can be used to setup the connection, by updating the port name. It will returns a cell with all the different printf concatenated and you can access to each string element by its index. You can use the function `eval(string)` to evaluate the result in the Matlab main scope. For array obtained by the `print_array_u32` and `print_array_float` (on the Zybo side), you can use (on Matlab) the functions `load_array_float` and `load_array_u32`. Then, you can inspect your data, plot the array, and so forth !

Chapter 3

On Autotune and project iterations

Let's go now to the project iterations. You should refer to the previous parts when it comes to Analysis or Implementation advices.

3.1 First iteration: Signal analysis, basic compilation

A warn advice

First be sure to have read section 2.1.1

3.1.1 Analysis part

In this first part we will investigate and analyze some audio signals. You can perform all your tests and analysis on the different signals proposed in the `Base_Sound` folder.

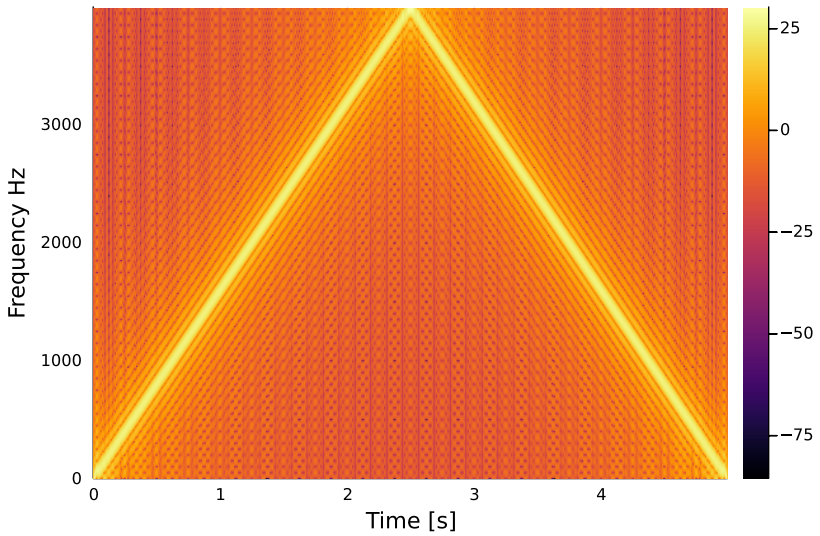
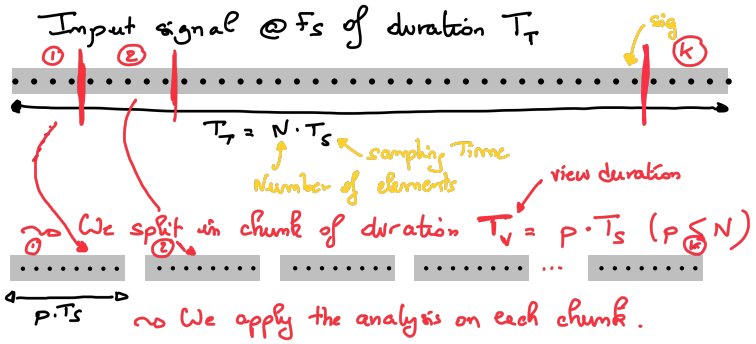


Figure 3.1: Example of a TF grid for the file `synth_sweep_1.wav`

1. Load a file and check the properties of the signal. Plot the signal in time and frequency domain. You can also listen to the audio file (with headphones for the sake of your professor). For each signal, be sure you be able to describe the signal in time and frequency domains.

2. A interesting way to characterize a signal is to investigate its time/frequency component. It corresponds to the analysis of its frequency component on a window in time domain, done iteratively. The representation is thus a matrix which column are the frequency components and the lines are the time axis. A key article of the literatures is the one proposed by Stankovic in 1994 [Sta]. Propose a Matlab function that perform a time/frequency analysis. This function should take a vector, its sampling frequency, an FFT size and a window duration (in samples) and display the time-frequency analysis. For instance, for the file `synth_sweep_1.wav` the TF grid should be represented as in Figure 3.1. A To compute a TF grid, the following steps should be performed:

   (a) Chunk the input signal into different pieces, each of them with desired duration (that we will call a *view*). Note that this duration may not be aligned with the FFT size. This duration should be define in seconds and corresponds to a few millisecond of signal. Note that the few samples of the end can be discarded if it does not fit in a view.



   (b) For each view, apply an apodisation function. The apodisation function should be chosen carefully, see the associated DSP lecture part.

   (c) Apply the frequency transform to each view. If the view size is lower than the FFT size you should follow a Zero Padding approach. You can extract the modulus on one side and the angle on another matrix.

   (d) You have now two matrixes as you have frequency components (modulus and phase) for each view. Display the matrix (using `imagesc`) of the modulus (as it is in Figure 3.1). By default, `imagesc` flips the axis and you can use the Matlab command `set(gca,'YDir','normal');`.

   A T/F exposes keys information of the signal characteristics, especially when the signal is not stationary. The duration of the views should be chosen to ensure the signal is stationary in each view. The color indicates the energy level. For Figure 3.1 it shows that you have only one frequency active at a time and that this frequency increases and then decreases. This is a sweep !

3. Perform the same kind of analysis for the other signal. What is the active frequency used in `synth_beep_1` ? What is the active frequency used in `synth_beep_2` ? For this two algorithms, propose a algorithm that automatically extract the main frequency component (a *pitch*) and checks that the result you have are coherent with the T/F grid you have plotted.

3.1.2 Implementation part

It the SDK does not launch

It is possible that nothing happens if you launch the `launch_project` command. It this is the case launch `debug_launch.sh` and import manually all the projects in the workspace.

This first implementation part is minimal. It is mainly as an introduction to the Zybo board and take charge of the design flow, both in compilation and execution.

1. You have a blank project, read carefully the file `main.c` to understand the different part and how you can switch between modes. The two main modes used in the project is the offline mode (when you will play a signal from memory) and the online mode (when you will record a signal and then play it).

2. We use a **Direct Memory Access (DMA)** system to have a memory location where samples can be put from the microphone and take for the speaker. This location is placed at the beginning of default memory location (MEM_BASE_ADDR) on which an offset XAXIDMA_BUFFLEN_OFFSET is required. To ease the DMA access all the take/put functions will use `ptr0`, a pointer that points this precise location. All the interface use standard word size. As the Cortex A9 is 32 bits, it means that we should use `ptr0` as a `u32`. Instantiate the pointer at its correct location in the file (line 156).

   Code location

   In the proposed code skeleton you have the *a priori* location of the code addition and the Iteration index used. In some cases it may require additional update at other places (code in `processing.c` or addition in header files).

3. Write a function `populate_dma(ptr0,sig,size)` that copies an array `sig` at the location where it can be used by the I2S interface. We indeed work at another location and then copy the location of interest at the DMA location. An illustration of the memory mapping is presented on Figure 3.2. Two (very) important remarks

   - We will work on `int16` array (with words on 16 bits called *halfword* words) and we have to write u32 words for audio. A cast will be required to switch from `sig` to `ptr0`.
   - Audio interface is an audio interface which means that we need to write two consecutive words (with same content) to ensure correct audio rendering.

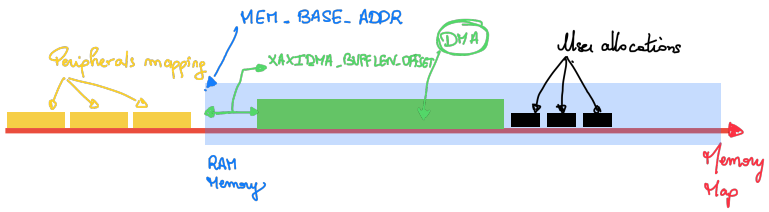


Figure 3.2: Memory mapping and DMA location

You can now compile and check that you can hear the first audio signal.

⚠ TAKE CARE OF YOUR EARS

You have to plug earphone to the HPH out port of the Zybo. If the sampling file (when RUN_MODE is set to 0) have a good volume as the data is appropriately scaled, if you do not implement correctly the filter it may lead to very strong audio signal. Be sure that you don't put the earphones on your ears the first time you launch the application with the filter activated. If the signal that comes from the earphone is sufficiently low, you can have a deeper analysis on the signal quality.

4. You can now try to replace the content of `data_file.c` by another content from the folder `Base_Sound_c`. Any remarks.
5. Enter the debugging mode and put a breakpoint at (around) line 172. Be sure you can have access to internal variables, memory locations and registers. Check also the disassembly code. All these aspects will be important when you will have to actively debug your more complex applications.

End of Iteration 1

What you should have

- Understand T/F grid and plot it for the different signals
- Be able to analyze the signal in time and frequency domain to extract itds key characteristics
- Listen to these signals in Matlab
- Take in charge the development flowgraph on the Zybo
- Setup the I2S interface through the DMA and listen to an audio signal directly on the Zybo

3.2 Second iteration: Pitch synthesis

In this part we will synthetize simple audio signals and play it on the Zybo board.

3.2.1 Analysis part

The purpose of the analysis part is to synthetize two signals that follows specific T/F grid. Note that these signals will be defined by their frequency and duration.

1. Propose an algorithm that is able to construct a similar signal as the one proposed in Figure 3.3. You have to first analyze the image to extract the core characteristics of the signal and then inject these characteristics into a script `generate_synth_sig.m`. You can also load the Matlab figure to better zoom if you want. Remember that the image corresponds to a T/F grid and the associated sine wave has the form $sin(2\pi f t)$ where f is the sine frequency and t the time axis.
2. Listen to this generated signal and call the professor to check the quality is OK.
3. As you have done in Matlab, the signal generation is done with sine wave. On the Zybo board all operations are in Fixed Point and the sine wave should also follows this formalism. To do so we will have to find an estimation of a sine wave that allows us to synthetize kind of a sine wave. We will call this function `fixSine`. This function will take a `Int16` as an input and produce an `Int16`.
   (a) We will use a Taylor approximation for the sine function. Trough Taylor expansion we can quickly find the expression of an approximation of the sine function defined by for x $\in]-\pi/2;\pi/2[$. We will use an intermediate variable $\bar{x}$ and define the approximation as

$$P[\bar{x}] = a \times \bar{x} + b \times \bar{x}^3 + c \times \bar{x}^5 \tag{3.1}$$

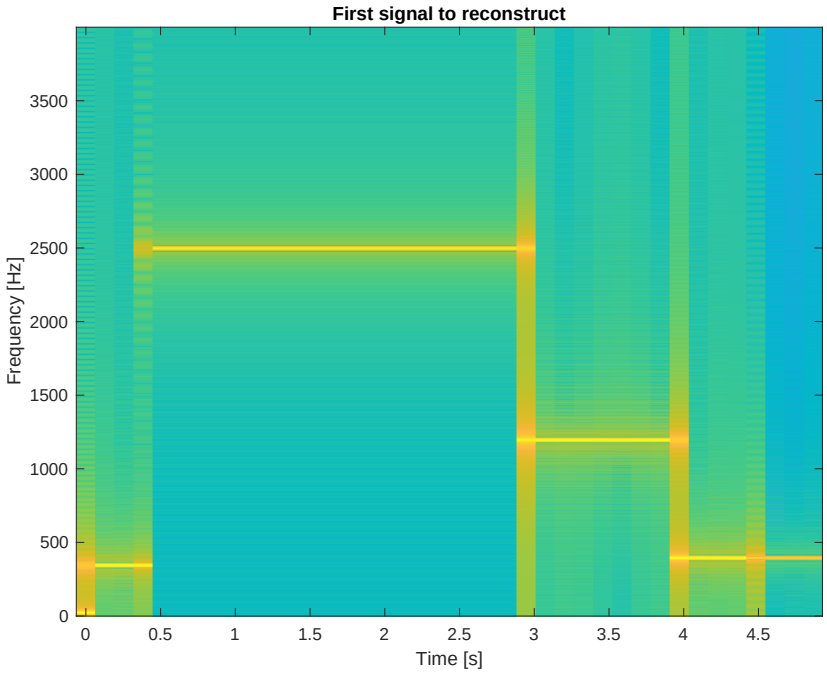


Figure 3.3: T/F grid of the signal to synthetize

with $\bar{x} = \frac{2x}{\pi}$ and a, b, c the Taylor coefficients values. The scaling of x in $\bar{x}$ is important as it will allows to have an input between $]-1:1[$. This is also practical as sine is sometimes computed for multiple of π which will corresponds to values that can be simply expressed using $\bar{x}$. For example $x = \pi/4$ corresponds to $\bar{x} = 1/2$. Deduce what will be the Q format of the input.

(b) Calculate the values of a, b, c and demonstrate that b and c can be expressed as a function of a.

$$b = -2a + \frac{5}{2} \tag{3.2}$$

$$c = a - \frac{3}{2} \tag{3.3}$$

(c) Demonstrate that a can be expressed as $a = \pi/2$. This does not corresponds to the best approximation we can get for an order 5 approximation. Demonstrate that a can also be expressed as $a = \frac{12}{\pi} - \frac{9}{4}$) using

$$\int_0^1 P[\bar{x}]d\bar{x} = \int_0^1 sin\left(\frac{2}{\pi}\bar{x}\right)d\bar{x}$$

Note that this new value for a is close (but different !) to $\frac{\pi}{2}$.

(d) Propose a flowgraph of the fixed point sine function. Calculate all the intermediate Q formats for each input variables. Deduce the final output format.

(e) Implement a fixed point function in Matlab. **This function should take an int16 as input and returns an int16. You can use the function provided that performs multiplication and shifts as it will be on the Zybo**. To perform the multiplication the same way it will be done on the Zybo, use the provided function `mul_int16_int16.m` which applies a single precision multiplication .Represent the 3 versions of the algorithms (the real sine, the floating point approximation and the fixed point calculation on the same figure). The output should be really similar of what is displayed on Figure 3.4. What should you display to have a better comparison between the implementations ?

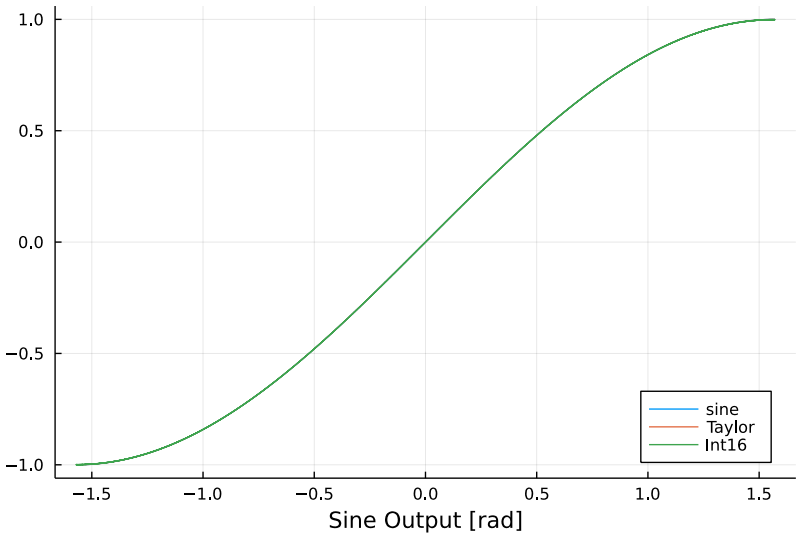
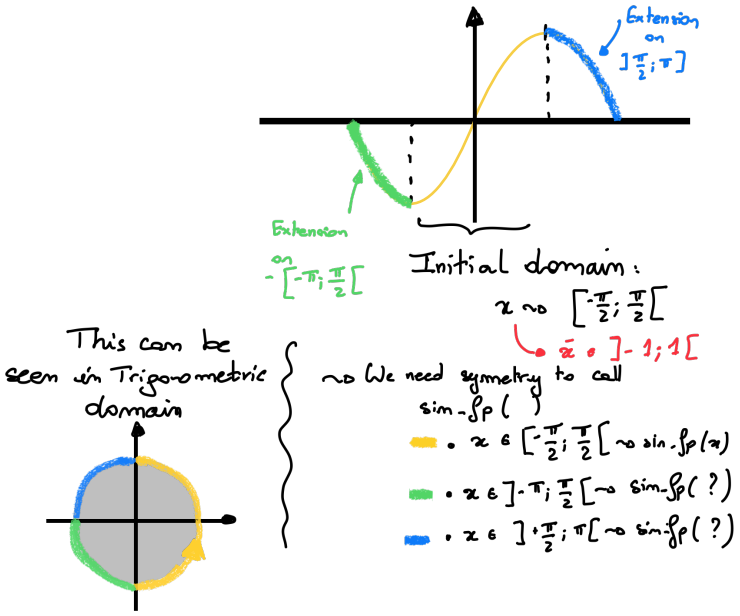


Figure 3.4: Comparison between real sine, approximation and fixed point implementation.

(f) The Matlab function is capable to generate a sine for an angle between $-\pi/2$ and $\pi/2$. It will not be enough to be able to synthesize a complete sine wave. Using the symmetry properties of sinus, propose a function $sin_f p_v 2$ in Matlab that still takes an Int16 but on another format (to be defined) such that we can approximate the sinus for an x between $]-\pi;\pi[$. This is a *domain extension* that will allow us to generate sine waves more easily.



(g) We can now generate a sine wave for a given frequency Fc at a sampling frequency Fs. The trick here is to remind that a sine wave applied on $2\pi Fct$ corresponds to a rotation in the trigonometric circle. We just have to find the appropriate angle, which is directly a function of

Fc and Fs. As we work a sampling frequency Fs, the sine we have to generate has the form $sin(2\pi\frac{Fc}{Fs}t)$. To generate a vector that corresponds to a sine wave at the frequency Fc we have to apply the sine function to an angle that is iteratively calculated. Explain why you will need to calculate $\omega = 4Fc/Fs$ on a Q(16,1,14) format. How can modular arithmetic can help you to ensure the result will still be in the sine definition domain ? When prototyping on Matlab, use the provided function `add_int16_int16` that emulates the modular arithmetic of the Zybo adder. Write a Matlab function that generates a sine wave for a given frequency and check that the audio rendering is satisfying.

3.2.2 Implementation part

Few questions but lot of works ! Do not hesitate to go back to Matlab if you need additional tests especially for the Fixed Point sine generation.

Float you say ?

You work now on the Zybo. Despite being a powerfull board, we may want also to implement your algorithms on very low power (and tiny) devices. It means that all the C code that will be embedded **cannot hold any floating point values**. No π values, no decimal division only integers manipulation !

1. Write the Fixed Point sine function you have designed in the previous part. You should test the function for different key values. Check that the format are respected and that there is no overflow. Note that we use haldfword precision and that the multiplication should be done with the given function `mul_int16_int16`. This function should apply the sine to one value (an int16).
2. Write a function that creates a sine wave with a desired frequency. It should takes an array as an input, the desired frequency, the sampling frequency and the size of the array. Populate the DMA with this created array and checks that this sine wave is correctly synthesized by the speakers.
3. Write a function associated to the the signal of question 3.2.1.1. Check that you can hear them from the Zybo with a similar quality to what you add in Matlab.

End of iteration 2

What you should have

- Be able to analyze signal from the T/F grid to deduce what you have to generate
- Synthesize audio signals from characteristics in floating point
- The same in Fixed point with a Fixed Point sine function
- A pure tone synthesis on the Zybo board

3.3 Third Iteration: Pitch detection and Resynthesis

3.3.1 Analysis part

1. We define a pitch as a perceptual property of sounds that allows their ordering on a frequency-related scale. More commonly, pitch is the quality that makes it possible to judge sounds as "higher" and "lower" in the sense associated with musical melodies. Pitch is a major auditory attribute of musical tones, along with duration, loudness, and timbre (thank you Wikipedia). In short for us it will corresponds to "special" frequencies on the grid that belong to a special group of

harmonic. Autotune aims to "lock" frequencies around some of this special frequencies and in this part we will do this on synthetized signals. You can read the paper in the Articles folder [DLCMS].

2. We need a baseline for our pitch (i.e a list of the lock frequency). We propose to use the C minor table and you can find the vcalues of the harmonics frequencies for instance `https://pages.mtu.edu/~suits/notefreqs.html`. Be sure you understand why we define these frequencies and how we will use them
3. Write an algorithm that takes a pure sine as an input and synthesize an output whose pitch is on the C minor table. You may need two functions: one for the *pitch detection* and one for the *pitch synthesis*.
    - *Pitch detection* aims to detect the location in the spectrum with the most energy. It can be done by for instance auto-correlation or frequency analysis. Discuss on the advantages of the methods and remember that you will have to implement the block on the Zybo. Based on the location of this max, the closer pitch from the C Minor table can be calculated
    - You can now replace the signal by a synthesized signal with the pitch frequency from the minor table.
4. Check that the system works for instance on `synth_beep_1.wav`. Same question for `synth_beep_2.wav`. Any remarks ?
5. Update your function to perform the pitch synthesis per block of input signal instead on the complete signal. Check the result on `synth_beep_3.wav`. If you hear beats it means that you have a phase issue. The signal phase should indeed be continuous. Update the Matlab code to be sure the phase is continuous.
6. Test now on a continuous signal using the different `sweep` files.

3.3.2 Implementation part

1. Implement the pitch shifting algorithm on the Zybo board. You should read and use the `fixFFT` library. Remember that this function only support FFT up to 1024 bins.
2. Test the function on the 3 different beep signals (define on the .c and .h files)
3. Test finally on sweep files. Be sure that the rendering is satisfactory, especially that you do not hear more beats than in the analysis part.

End of iteration 3

What you should have

- A functional pitch shifting in Matlab and in Zybo !

3.4 Fourth iteration: Autotune

3.4.1 Analysis part

1. Check your pitch shifting algorithm on a real signal. Any comments ?
2. The issue is that we completely resynthetize a pitch. We should perform *pitch shifting* which corresponds to transform the input signal (and not resynthesize it). Propose an algorithm that detect the pitch but shift then the location of the maximum to the C minor grid instead of replacing the signal.
3. Test your methods on the different signals and try with real signals. Remember that the phase should be maintained to limit the beep.

3.4.2 Synthesis part

1. Implement the refined algorithm on the Zybo board.
2. The fun begins ! You can now switch to the RUN_MODE= 1, record you own voice and apply the autotune on your recorded samples.
3. For the final version of your processing, after all you refinement, benchmark your code by setting RUN_MODE= 2. Note that the processing time should be lower than 5 second as it corresponds to the record time. If the benchmark shows higher values, it means that you will not be able to perform real time autotune and you should update your code (and make it simpler or with less allocations) to ensure you hold the stringent time limit we have.

Optional

The rest is purely optional. If you have everything done by now, well played !

This is not the end. Search for algorithms refinement and and you can propose several modifications to your algorithm. Remember that only algorithms on the Zybo board will be evaluated.
Some hints that to be pursued

- You can enrich the generated signal through harmonics detection as we have only here use the fundamental. In such a case, you need to shift the harmonics by multiple of the pitch shift initially estiamted
- A key element to upgrade your algorithm is to use the fact that you have different part of the audio band that can be processed independently. It is the core of Filterbank analysis. In Filterbank, you can analyse and filter subband. You can build a filterbank that mirrors the C minor table to enhance the pitch detection and the pitch shifting [KKL].

Appendix

Table 3.1: Glossary

ADC	Analog to Digital Converter
BPM	Beast Per minute
DAC	Digital to Analog Converter
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
GPP	General Purpose Processor
PL	Programmable Logic
PS	Processing System
SIMD	Single Instruction Multiple Data
SoC	System On Chip
T/F	Time Frequency Grid

Bibliography

[DLCMS] Patricio De La Cuadra, Aaron S. Master, and Craig Sapp. Efficient pitch detection techniques for interactive music. In *ICMC*.

[KKL] Geon-min Kim, Chang-hyun Kim, and Soo-young Lee. Implement Real-Time Polyphonic Pitch Detection and Feedback System for the Melodic Instrument Player. In Tingwen Huang, Zhigang Zeng, Chuandong Li, and Chi Sing Leung, editors, *Neural Information Processing*, Lecture Notes in Computer Science, pages 140–147. Springer.

[Sta] L. Stankovic. A method for time-frequency analysis. 42(1):225–229.