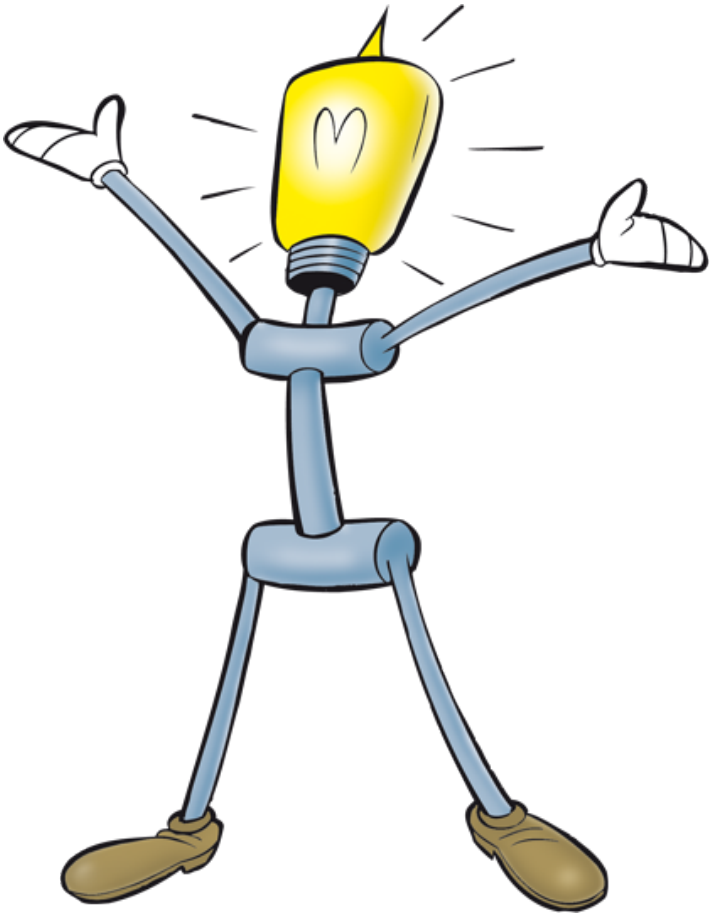
Little Helper

Pitch Correction/Auto tuner

F18-ETISB



Mads Christian Rosendahl
201509250

Lasse Østerberg Sangild
20114274

Advisor
Kim Bjerge

Abstract

This project has had the aim to create a real time auto tuner for use on stage during live performances. The scope was narrowed to get a better understanding of the necessary steps, and the end result focuses on creating a pitch shifter. Starting with shifting only a single sine and presenting thoughts on a model which could be used to determine a complete tone.

A Matlab model based on determining the Instantaneous Frequency using the Hilbert Transform and generating a new sine has been created.

The Instantaneous Frequency finder has been implemented on the ADSP-BF533 using the CrossCore framework and the real time aspect of the issue has been tested. A single sine input frequency can be determined and a new one generated in a maximum of 3.21 ms.

Contents

Reading guide

The report will start with a description of the requirements for the system and a use case for how the system can be used and in what context the system is thought to be used in. After the requirements there will be a plan for the project. The plan is made to show, how the different steps in the development process will be, with many small goals, that will increase in difficulty.

After the planning, the project will start with the first goal in the plan, to pitch shift a pure sine. Here all the thoughts will be described, and a analysis, design, modelling, implementation and test is written for this goal. The next goal in the planning to pitch shift a tone, is then described with possible solutions and ideas as well as problems. There are appendix which will be referenced to, for example to see the Matlab code for the modelling section.

Lastly a discussion of the hole project has been made with a following conclusion to round the project off.

Chapter 1

Introduction

We all know the feeling when your favorite song appears in the radio, and you can't stop yourself from singing along, even though you know, and everyone knows that it sounds like a cat fight. Little Helper is going to make even the most horrifying singer into a decent singer. The goal of Little Helper is to help the singer, so if the singer can't hit a tone, Little Helper will help with this. Little Helper will change the incoming sound and change it to the nearest tone, that way the user will never have a problem hitting an A' again. But this is not all Little Helper can accomplish. In today's modern Danish rap music, it is very popular to use auto tune in a way, so it is obvious. Therefore Little Helper will have the option to 'over' auto tune the incoming sound if the user wishes it. This way even a person with no singing skills, can become a rapper, just with a little help from little helper.

1.1 Terminology

Below in table 1.1 is shown a list of terms used throughout the report describing each name for clarification purposes.

Name	**Description**
Little Helper	The systems name.
IF	Instantaneous Frequency
IP	Instantaneous Phase

Table 1.1: List of terminologies.

Chapter 2

Requirements

In this chapter the requirements for the system will be described. A single use case has been defined, which describes the functionality and the actor's interaction with the system Little Helper. These use case establish the foundation of the requirements specification.

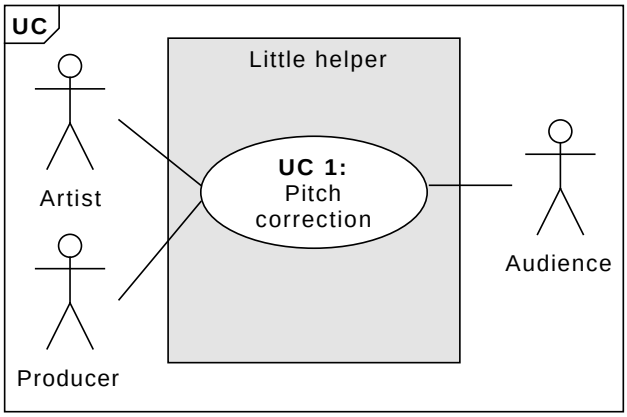


Figure 2.1: Use Case diagram.

2.1 Actor descriptions

Name of actor	Artist
Type	Primary
Description	Is a live performer. The goal for the Artist can be either to be corrected into perfect pitch (e.g. from 443 Hz to 440 Hz), or to create the robotic sound often associated with auto tune (e.g. Cher - Believe).

Table 2.1: Actor description of Artist.

Name of actor	Producer
Type	Primary
Description	Decides how the sounds are during recordings. The producer needs to be able to change the amount of auto tuning used during the recording, to find the right balance.

Table 2.2: Actor description of Producer.

Name of actor	Audience
Type	Secondary
Description	The recipient of the live performance or studio recording. Is interested in hearing the music as the Artist/Producer intended it to be heard.

Table 2.3: Actor description of Audience.

2.2 Use Cases

Figure 2.1 shows the defined use cases and the different actors involved in each of them.

2.2.1 Use Case 1 - Pitch correction

Name	Pitch correction
Goal	To move an off tune sound onto a piano scale tune.
Initiation	Singing commences.
Actors and Stakeholders	Artist, Producer and Audience.
Precondition	Resolution and scale is set.
Post-condition	Output sound is moved to the nearest on-pitch frequency, determined by the resolution and scale.
Main scenario	1. Artist starts singing off pitch. 2. Artist/Producer engages Pitch correction. **EXT1** Artist/Producer is not satisfied with set scale or resolution. 3. Output sound is on pitch according to the set resolution.
Extension	**EXT1** 1. Sound is not as Artist/Producer wants it. 2. Artist/Producer changes resolution or scale. 3. Output sound is on pitch according to the set resolution.

2.3 Functional Requirements

A MoSCoW based on the above use case is presented below to account for the functional requirements.

MUST

- Little Helper must shift the primary frequency of the input signal to the nearest step on the desired scale.
- Little Helper must do the processing quickly, so there is no audible delay in the sound.
- Little Helper must do the processing so there is no noticeable noise when the input frequency changes.

SHOULD

- The user should be able to change how 'effective/strong' the auto tune is by pressing buttons on the system.
- The user should be able to turn auto tuner on/off by pressing a button.
- Little helper should have a LED indicating if it is on or off.
- Little helper should have 4 LED's indicating which level the auto tuner is on where 0 LED's on is low. and 4 LED's on is high.

COULD

- Little Helper could be configured by a GUI, so the user can change all the different parameters, such as scale, force and speed.
- Little Helper could take noisy environment into account, when finding the primary frequency.

WON'T

- Little Helper won't make you sing like Freddie Mercury.

2.4 Non-functional Requirements

R1 Little Helper must process audio signals in the C-major scale (261.626 Hz to 493.883 Hz).

R2 Little Helper must correct the audio signal to the nearest step on the scale $\pm$2.5 Hz.

R3 Little Helper latency should be less than 50 ms.

R4 The filter algorithm must be realized using build-in fixed point fractional types (fract16, fract32).

R5 Little Helper should fill less than 16 kB.

R6 The filter should be optimized in speed performance for the B533.

R7 The filter should have a maximum DSP load of 95 %.

2.5 Derived Requirements

DR1 Little Helper must sample with a samle rate of 48 kHz. Derived from R1, where $f_{max} = 1046.5$ Hz then minimum sampling frequency must be $fs_{min} = f_{max} \cdot 2 = 2093$ Hz.

DR2 The filter latency must be delayed by a maximum of 2000 samples. Derived from R3 where the latency is less than 50 ms, $f_s = 48$ kHz, $t_d = 41$ ms.

DR3 The filter must use a maximum of 11 875 cycles of DSP processing per sample. Derived from R6, R7, DR1 and maximum clockspeed of the BF533 [2, p. 3]. $\frac{600\,\text{MHz}}{48\,\text{kHz}} \cdot 95\,\% = 11\,875$.

Chapter 3

Development Plan

To be able to investigate more methods, the project has been split up into stages, each aiming to bring the project closer to the end result of an auto tuner.

- Pure sine corrected to the nearest of two tones.
- Chirp corrected to the nearest of two tones.
- Chirp corrected to C major.
- Handling up to third harmonic being the dominant frequency.
- Voice fundamental frequency corrected to C major.
- Voice fundamental, first and third harmonic, corrected to C major.
- Remove noise and correct voice to C major.

Chapter 4

Pitch shift pure sine

This chapter deals with the issue of moving a pure sine of a frequency between 245 Hz to 521 Hz (approximately subtracting and adding half the difference between the two endpoints, e.g. $C_4 - \frac{D_4 - C_4}{2}$) onto the nearest fundamental frequency of the C-major scale[11].

Name	Frequency (Hz)
C_4	261.626
D_4	293.665
E_4	329.628
F_4	349.228
G_4	391.995
A_4	440.000
B_4	493.883

Table 4.1: Table of C-major fundamental frequencies.

4.1 Analysis

The issue of changing the frequency of a pure sine wave can be handled in a number of ways:

- Changing the sampling frequency.
- Creating a new sine at the target frequency.
- Interpolating or decimating the input (only works if $f_{target} = \frac{f_{meas}}{n} \vee f_{target} = n \cdot f_{meas}$, where n is an integer greater than zero.)

Changing the sampling frequency is easily done in Matlab by following the recipe

1. Find the input frequency.

2. Find the ratio between the input and the target frequency.

3. Multiply the sample frequency by the ratio between the input and target frequency.

4. Postpad with extra samples starting at the right phase if the ratio is larger than 1. Remove samples if ratio is less than 1.

5. Play at new sampling frequency.

Although Matlab is able to play at different sample rates, the Blackfin has a very limited number of sample frequencies to choose from, making this approach unusable[1, p. 19].

Creating a new sine at the target frequency seems to be the most versatile possibility, as you can easily control the frequency and the phase between blocks. The drawback is that calculating the sine is an expensive operation. The steps required are

1. Find the input frequency.

2. Create new sine at target frequency.

3. Play output sine.

Creating a new sine at the target frequency can be made less computationally intensive by calculating sine only at some points and interpolating the rest, or using a look-up table. It is also possible to only calculate one period of the sine at the target frequency and play that part the required number of times. This most likely gives a phase different from 0 as the block ends, but the phase can be saved and used to calculate the sine of the next block. This is illustrated in figure 4.1, where the points from 0 to 2π is calculated and interpolated, but the next period from 2π to 4π is just a copy of the period from 0 to 2π. The last part creates an offset, where the phase can be determined based on the amplitude of the last two points. This phase can then be used to create the first period of the next block.

Interpolating or decimating the input is a very simple solution, which only works if $f_{target} = \frac{f_{meas}}{n} \vee f_{target} = n \cdot f_{meas}$, where n is an integer greater than zero.

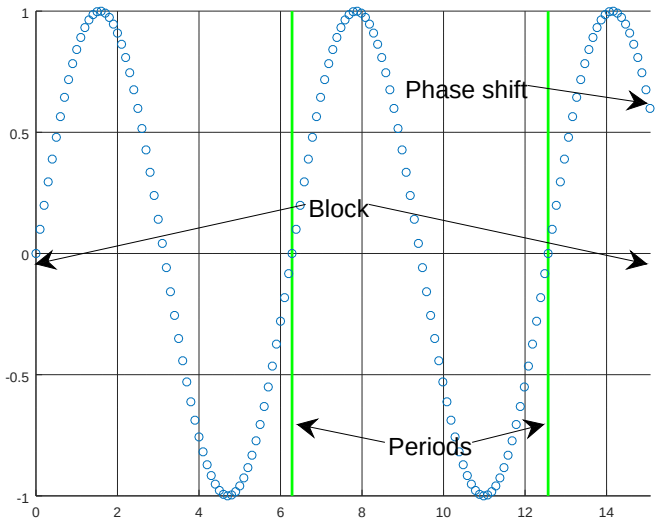


Figure 4.1: This figure shows the envisioned way of copying previously created periods, to avoid recalculation, and the ability to determine the phase for the next block to start at.

4.2 Design

To fulfil the requirements specified, the system is split into three parts: Preprocessing, Signal processing and Post-processing, as seen in figure 4.2.

Preprocessing handles the analog to digital conversion and preprocessing allowing compliance with Non-functional requirement R1 and R2 for the Signal processing block. This means Preprocessing takes the input, processes it and outputs a discrete signal which can be processed and modulated.

Signal processing finds the frequency/tone based on the frequency found it shall be moved to the nearest corresponding C-major tone, shown in table 4.1.

Post-processing handles the digital to analog conversion.

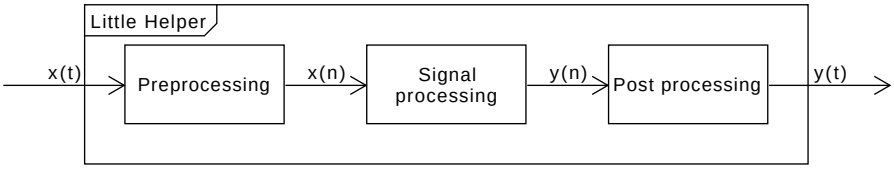


Figure 4.2: Overall design of Little Helper.

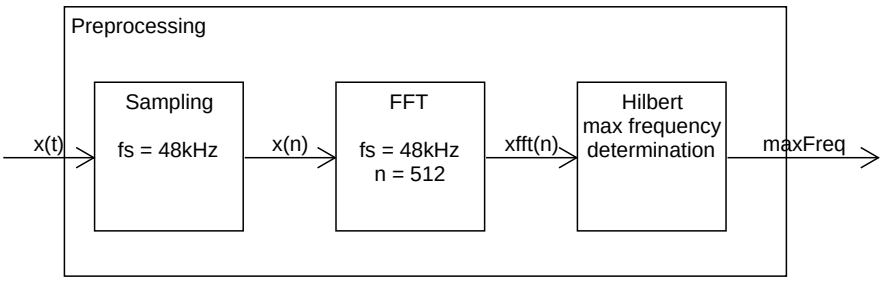


Figure 4.3: Detailed description of Preprocessing in Little Helper.

4.2.1 Detailed description of the design parts

4.2.1.1 Preprocessing

On figure 4.3 it can be seen that the analog signal will be sampled with a sampling frequency of 48 kHz. The discrete signal wil be passed on to the FFT, which will make an FFT of the signal. The FFT signal, xfft, will be passed on to the Hilbert max frequency determination block. The Hilbert transform will find the dominating frequency in the signal. Hilbert transform will only work when the input signal is a pure sine wave, if there are more sine waves mixed together it will find the average frequency and therefore a misinterpretation of the signal will be made. Hilbert transformation requires many heavy calculations and deviations which will result in a product quantization error. The found max frequency, maxFreq will be passed on to the signal processing block.

4.2.1.2 Signal processing

Figure 4.4 shows the detailed idea of the signal processing part.

The maxFreq will be passed on to a Sinus generator, where it will be determined which frequency to go to on table 4.1. A sine wave with the found frequency will then be made with as few points as possible. The generated sine wave, y(n), will be interpolated so the signal gets a lot of values in the sine wave, this method has been chosen with the thought that generating a sine wave with many points is more costly then interpolating. The new full sine wave will be passed on to a circular buffer. The idea behind the circular buffer is to avoid calculating the same sine wave again and again. If the maxFreq appears to be the same multiple times, the sine wave in the circular buffer can be reused thus we do not have to generate or interpolate a new sine. The output from the circular buffer will be the new complete discrete signal.

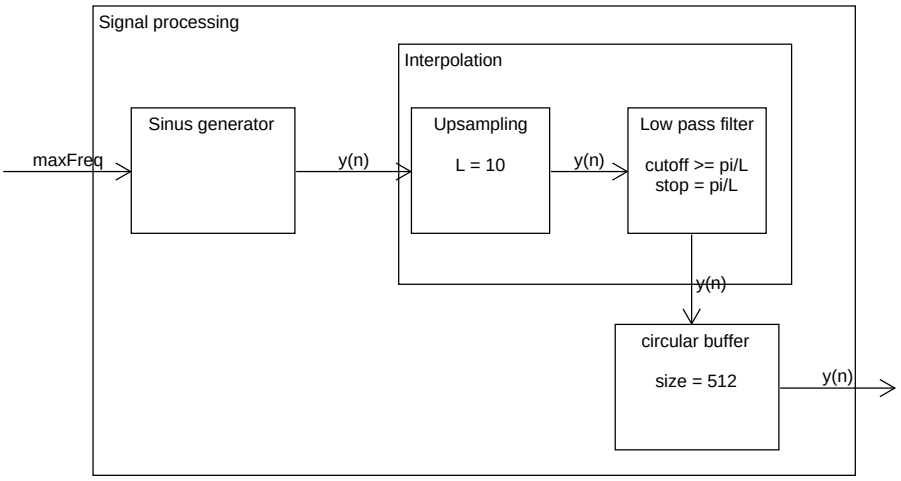


Figure 4.4: Detailed description of Signal processing in Little Helper.

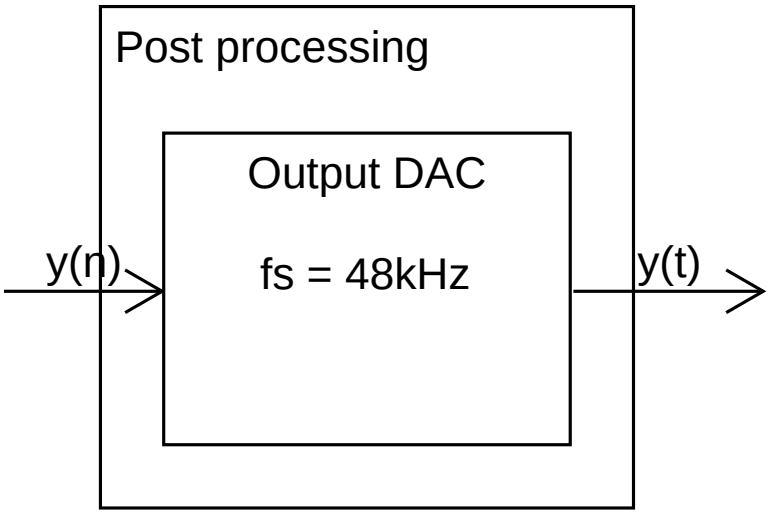


Figure 4.5: Detailed description of Post processing in Little Helper.

4.2.1.3 Post processing

The last phase of the system will take the final discrete signal and through a DAC make it to an analog signal as shown on figure 4.5.

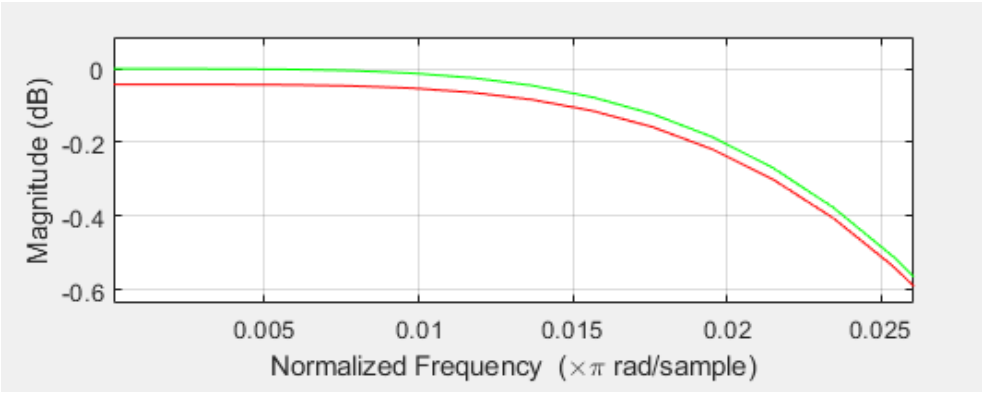


Figure 4.6: Matlab filter coefficients results in the green curve, 1.15 format in the red curve.

4.2.2 Quantization

There will be a quantization error on the IIR low pass filter, but since the filter is not steep and have rather long time to get to the stop band, the low pass filter does not suffer significantly from the quantization, as can be seen in figure 4.6 where the max difference was found to be $<0.5\,\%$. Thus the quantization error for this specific filter is negligible, but the implementation of this filter should be a FIR low pass filter since it is what works with the build in functions.

In the filter there will occur coefficient and product quantization. Coefficient quantization will occur when the filter coefficients are fitted to the 1.15 format. The product quantization will occur when the filter coefficients are multiplied with the data. This will result in quantization since it still has to fit 1.15 format. On figure 4.7 the places where quantization occurs can be seen.

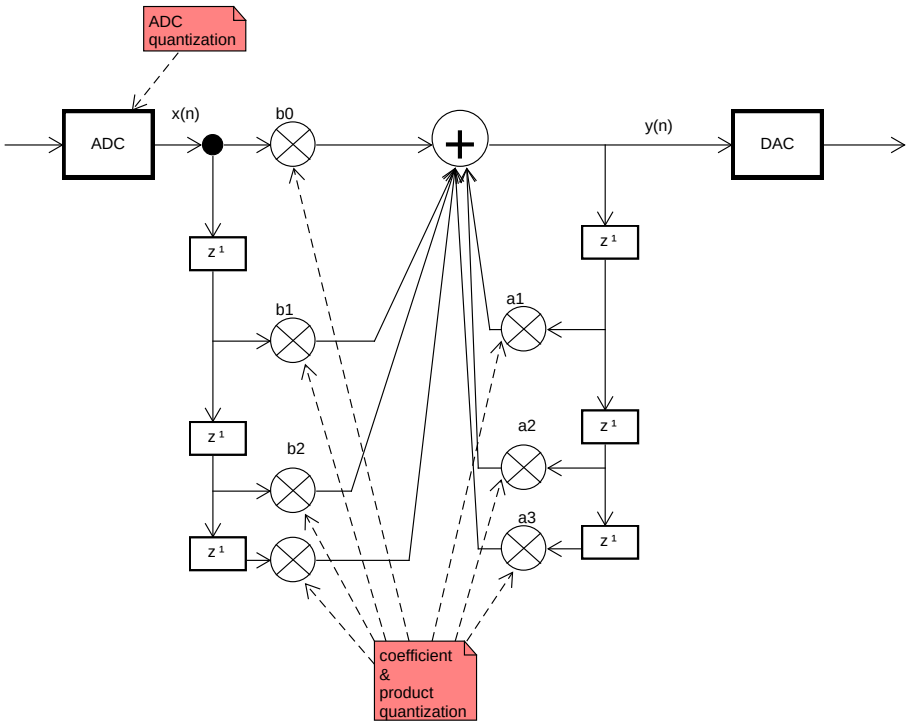


Figure 4.7: Quantization errors Little Helper.

4.3 Modelling

The Matlab code for pitch shifting an input sine of 470 Hz is shown in appendix A. It is primarily based on the description from Matlab [8] and the open source implementation in the Octave Signal package [10].

Listing 4.1 shows pseudo code describing the Matlab model. The blockSize is set, and a Hamming window is generated. The window can be used for each block and it is only necessary to generate it once. The phase starts at zero, but is updated after each loop, allowing for the next block to start with the correct phase shift. The loop loads the samples, applies the window, finds the frequency, finds the closest C-major frequency, generates a sine at the correct frequency and phase and updates the phase.

Finding the frequency is done in five steps, the output of some plotted in figure 4.8:

1. Hilbert Transform
    - Shifts the input signal by 90°.
2. Determine the phase, figure 4.8a.

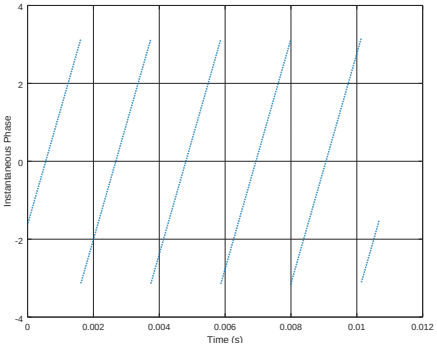
```
        blockSize = 512;
        h = hammingWindow(blockSize);
        phase = 0;

        while(1)
        y = load(blockSize);
        yh = h .* y;
        freqIn = findIF(yh);
        newFreq = pianoFreq(freqIn);
        [out, phase] = genSine(blockSize, newFreq);
```

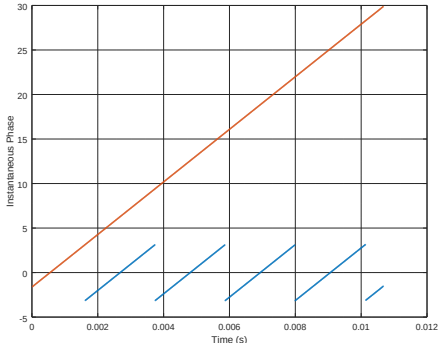
Source Code 4.1: Pseudo code for the Matlab model.

3. Unwrap the phase, figure 4.8b.
4. Calculate the derivative in each point, figure 4.8c.
5. Calculate the mean of the middle portion of the points (in our case the middle third).

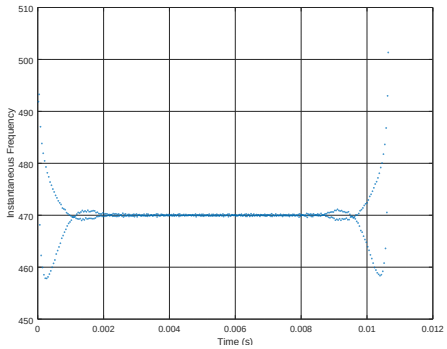
The IF and phase is used to create a new sine. This is done for each new block as shown in figure 4.9. In the model each point is created using a call to `sin()`, this is due to the fact that the lowest frequency, 261.626 Hz will need only $(\frac{48\,000\,\text{Hz}}{261.626\,\text{Hz}})$ =183 points without using interpolation, which does not seem excessive, and should be tried to make the Blackfin implementation simpler.



(a) Instantaneous Phase.

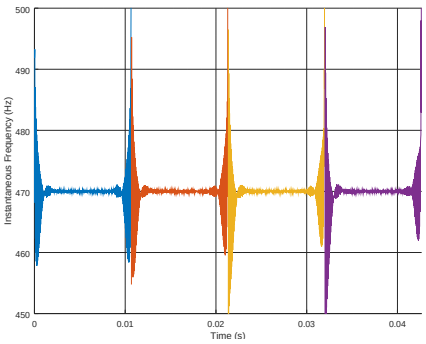


(b) Wrapping Instantaneous Phase (blue), and unwrapped Instantaneous Phase (red).

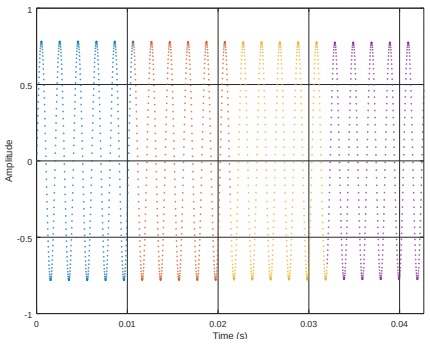


(c) Instantaneous Frequency. Derivative of the unwrapped phase.

Figure 4.8: Different steps of determining the Instantaneous Frequency.



(a) Instantaneous Frequency of the input signal. Each color represents a new block.



(b) Output signal in the time domain. Each color represents a new block.

Figure 4.9: Block processing.

4.4 Implementation

The implementation has been done in the framework made by our supervisor Kim Bjerge. The complete idea of the system has not been implemented on the Blackfin. What has been implemented on the blackfin is the Instantaneous Frequency algorithm, which determines the main frequency in the signal and a sine generator. To the framework has been added a function called `hilbert()` as can be seen below and and a function called `sineGenerator()`. Before the `hilbert()` is called the FFT of the signal will be made using the function

```
void rfft_fr16(const fract16 input[], complex_fract16 output[],
↪  const complex_fract16 twiddle_table[], int twiddle_stride,
↪  int fft_size, int *block_exponent, int scale_method);
```

After the FFT, the `hilbert()` will use the FFT signal. In `hilbert()` the FFT signal will first be rearranged into the array $zHilbert[N\_FFT]$. The first value and the $N\_FFT/2$ value will be waited less and therefore are multiplied with 0.5 using the optimized function $cmlt\_fr16(a, b)$. From $N\_FFT/2$ and beyond everything is set to zero. After rearranging the inverse FFT of zHilbert is made and placed in m_ifft_output. The inverse FFT is also made using the build in functions.

```
1  void DynamicFilter::hilbert(){
2          // parameters
3          int block_exponent;
4          fract16 instafreq[N_FFT];
5          fract16 meanFreq;
6          fract32 temp = 0;
7          uint16_t dif;
8          complex_fract16 scale;
9          scale.re=0.5; scale.im=0.5;
10         fract32 temp32[512];
11
12         // Hilbert transform
13         zHilbert[0] = cmlt_fr16(m_fft_output[0], scale);
14
15         for (int i = 1; i < N_FFT/2; i++){
16                 zHilbert[i] = m_fft_output[i];
17         }
18
19         zHilbert[N_FFT / 2] = cmlt_fr16(m_fft_output[N_FFT /
           ↪  2], scale);
20
21         // ifft of the hilbert signal
```

```
22          ifft_fr16(zHilbert, m_ifft_output, m_twiddle_table, 1,
            ↪  N_FFT, &block_exponent, 1);
23
24          ////// instant frequency finding //////
25          //first finding the phase
26          for (int i = 0; i<N_FFT; i++){
27                  instafreq[i] = atan2_fr16(m_ifft_output[i].im,
                    ↪  m_ifft_output[i].re);
28          }
29
30          // unwrapping with threshold 2PI
31          temp32[0] = instafreq[0];
32          for(int i = 1; i<N_FFT;i++){
33                  dif = (((int16_t)instafreq[i - 1] -
                    ↪  (int16_t)instafreq[i]) + (1 << 15)) % (2 *
                    ↪  (1 << 15)); //1<<15 = pi
34
35                  temp32[i] = ((int32_t)(dif - (1 << 15)));
36
37                  temp32[i] =
                    ↪  sub_fr1x32(temp32[i-1],        temp32[i]);
                    ↪  // optimization
38          }
39
40          // find the diff
41          for(int i = 1; i<N_FFT;i++){
42                  temp32[i-1] =
                    ↪  (sub_fr1x32(temp32[i],temp32[i-1])) * fs /
                    ↪  (2 * PI_FLOAT); // optimization
43          }
44
45          //finding the mean and therefore the value
46          for(int i = 100; i < N_FFT-99; i++){
47                  temp += temp32[i];
48          }
49
50          meanFreq = ((temp / (N_FFT-200)) * PI_FLOAT) / (1 <<
            ↪  15);
51
52  }
```

Now the Instantaneous Frequency finding starts. First the argument of every value of the IFFT signal is found using `atan2_fr16`. Afterwards the remainder

Function name	Description
`rfft_fr16(input, output, twiddle_table, twiddle_stride, fft_size, *block_exponent, scale_method)`	Calculates the FFT of the input signal
`ifft_fr16(input, output, twiddle_table, twiddle_stride, fft_size, *block_exponent, scale_method)`	Calculates the inverse FFT of the input signal.
`cmlt_fr16(a, b)`	Multiplies two complex fract16 values.
`sub_fr1x32(a, b)`	Subtracts two complex fract32 values.
`atan2_fr16(imaginary_part, real_part)`	Find the argument of a complex fract16 value.

Table 4.2: The used optimized functions.

is found after the subtraction of two arguments. Lastly the difference is found and then a mean of the middle values are used to find the meanFreq.

4.4.1 Optimized functions

In the implementation some of the build in functions have been used, these functions are optimized and should use as few clock cycles as possible to execute. On table 4.2 some of the different build-in functions which are used can be seen.

`twiddle_table` is a table made in the setup of the program. It is $W_n = \cos(\frac{2\pi kn}{N}) - j\sin(\frac{2\pi kn}{N})$. It is made so that the FFT and IFFT are executed faster since these function do not have to calculate this part of the equation but can just take the value from the `twiddle_table`. As long as the size of the FFT or IFFT does not change these values in the twiddle factor does not have to be calculated again.

4.5 Test

4.5.1 Frequency determination

Table 4.3 shows the output of the Matlab model and the Blackfin implementation when determining the frequency of a 512 samples long input signal. The Matlab model was run four times with four different inputs and applied a Hamming window, which the Blackfin implementation did not. As can be seen the Matlab model is very precise in determining the frequency whereas

Test freq (Hz)	Matlab freq (Hz)				Blackfin freq (Hz)
245	245.13	244.93	244.85	245.03	237
350	349.90	349.94	349.88	349.96	349
470	470.00	470.00	470.00	470.00	470
521	521.22	521.18	521.10	521.04	525

Table 4.3: Tested frequencies and the found results using blocks of length 512. Matlab results are the average of 4 blocks of 512 on a signal of length 2048, each block windowed with a hamming window. It has been investigated if the 470 Hz results are due to an error, but it does not seem so.

the Blackfin implementation is slightly of. The Blackfin being not as precise as the Matlab model can be attributed mostly to the lacking window and to quantization errors.

4.5.2 Clock cycle usage

A question to be asked is, "Can the processing of the signal occur before the next set of 512 samples?". To find the answer for this, the clock cycles used of the FFT an the hilbert transform has been measured, and the amount of time at disposal has been calculated.

Time between sample blocks: $td = \frac{samples}{fs} = \frac{512}{48\,\text{kHz}} = 10.7\,\text{ms}$

Measurement of the clock cycles usage: $\frac{1898733 cycles}{512 samples} = 3708 \frac{cyles}{sample}$. Which is less then specified in the requirements.

The Blackfin has a maximum speed at 600 MHz. Which means it will take $\frac{1\,898\,733}{600\,\text{MHz}} = 3.16\,\text{ms}$ so the Hilbert transformation uses less time than the time it takes to sample 512 samples since $3.16\,\text{ms} < 10.7\,\text{ms}$ and it uses less cycles pr. sample then specified in the requirements. Concluding the Hilbert transform takes less time than it has at its disposal.

To generate 183 samples, with the sine generator it takes: $\frac{26\,225 cycles}{183 samples} = 143.3 \frac{cycles}{sample}$. This brings the total time of finding the frequency and generating one period of it up to 3.21 ms, still way below the threshold of 10.7 ms.

4.5.2.1 Optimized clock cycle usage

In the cross core environment it is possible to optimize the code with regards to size or speed. This was done with three different settings for the optimization. Firstly where the size was the main focus of the optimization. Secondly the size and speed had same prioritization in the optimization settings. And lastly where the speed was the focus. The result can be seen on

So if the code is optimized with regards to speed, the amount of cycles used becomes less, but a neglect able amount.

Optimized for:	None	Size	50% Size 50% Speed	Speed
Hilbert transform	3708	3582	3577	3554
Sine generator	143	115	115	115
Total	3851	3697	3692	3669

Table 4.4: The cycle usage with different settings for the optimization function in the cross core environment. Every value is shown in $\frac{\text{cycles}}{\text{sample}}$.

Chapter 5

Pitch shift C-major

5.1 Analysis

In contrast to the pure sine, the C-major consists of several frequencies, as shown in figure 5.1.

These frequencies are evenly spaced, and the solutions presented in section 4.1 are still valid to use. Here the solution of changing the sampling frequency is still not valid due to limitations on the Blackfin, and the primary target of the analysis is if it is possible to use the same approach as in the pure sine implementation: Creating a new tone at the target frequency.

The approach again focuses on the steps

1. Find the input tone.
2. Create new tone at target frequency.
3. Play output tone.

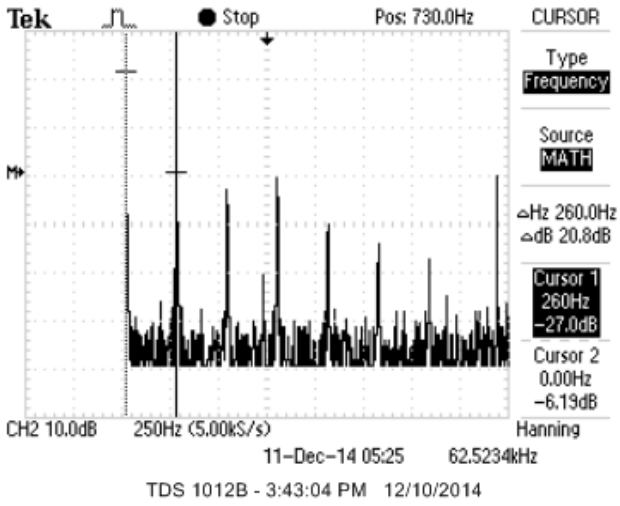


Figure 5.1: Frequency spectrum of C_4 played on a piano [4].

Note	IF
C_4	436.043
D_4	489.442
E_4	549.380
F_4	582.047
G_4	653.325
A_4	733.333
B_4	823.138

Table 5.1: IF values if including the first two harmonic frequencies.

Finding the input tone In the sine part, the frequency was found using Instantaneous Frequency. This approach may still be valid, but would require the ratio between the fundamental and every harmonic frequency to be exactly the same each time. Should the ratios for the fundamental and the first to harmonics be e.g. 3, 2 and 1, the output of the Instantaneous Frequency would be as shown in table 5.1.

Another approach could be to use FFT to determine the tallest peak, leading to a not very accurate measurement, approximately $\pm 50\,\text{Hz}$, bandpass filter this frequency bin, removing everything else, and the calculate the Instantaneous Frequency. The found frequency can then be compared to a table of the C-major frequencies and their harmonics, allowing the tone to be determined. Then the new pitch shifted tone can be created with a number of harmonics.

Chapter 6

Discussion

The original idea for this project was an auto tuner, which should help people with their singing, by changing the tones. The idea for this project originates from listening to todays big idols, both on the live stage and their studio recordings.

An impressive complex and optimized algorithm must have been made for auto tuning to work on live performances. Therefore a big challenge as an auto tuner seemed like a brilliant idea. The project started with making the requirements. After making the requirements and a guidance section with the supervisor, it was clear that this was to ambitious. Hence the project was scaled down to a pitch shifter, which firstly should be able to change the frequency of a pure sine as the incoming signal and later on, be able to handle the harmonics in a tone as well as changing frequency.

The first method to change pure sine to another frequency was generally to change the sample frequency, fs. But before changing the fs, the frequency of the input signal had to be found. Since the frequency area that this project works around is as seen in table 4.1 very close to each other a high resolution of the FFT was needed. And to get a high frequency resolution a lot of samples is needed. But this brings another predicament since with the high fs, the amount of samples will both fill the memory but also use a lot of calculation time. Therefore decimation before making the FFT, was decided this could give a high frequency resolution and doesn't fill the memory. Both of these methods huge flaw is the time delay of approximately one second, and that is only the frequency determination that has been made at that point. This method with decimation, fft and change of fs was manageable in Matlab, where the algorithm was written. But at the implementation stage it was noticed that the fs, on the BF533, only had two possible values, thus a new solution was needed.

The new method was the use of Hilbert transformation to find the frequency with a high frequency resolution and a short amount of processing time. A sinus generator from the $math.h$ library and a circular buffer to minimize the amount of sine waves to compute. The brilliant and astonishing thing about the Hilbert transform, is the extremely high frequency resolution and minimal amount of samples needed. It can be done quick and only with 512 samples,

and even less samples if the use of a window was implemented as shown in section 4.5. The cons of the Hilbert transform is the fact that it can only find the precise frequency of a pure sine wave meaning, if there are several sines mixed together it will find the average. This could probably be handled in some way, if a good description of the harmonics for an instrument as well as each tone is described in the code. Through the Hilbert transformation quantization errors also occurs, and result in a small deviation of the found frequency compared to the algorithm in Matlab. After the Hilbert transformation, the sine wave generator is needed. It was implemented in Matlab and a version on the Blackfin to measure the cycles usage. A problem with the sine generator is, if a tone is the input signal, a generator for all the harmonics are also needed, and could escalate to become very heavy on the computation or memory. To avoid computing a new sine every time the idea of reusing the data in a circular buffer was made up. Since most of this method has not been implemented on the Blackfin, it is still very unclear how the sound will be, the total time delay and how it will work when it is upgraded to handle harmonics as well.

Through the whole process the synergy between algorithm writing in Matlab and afterwards implementation on the Blackfin has been very effective. The synergy has also been used efficient to debug, to find the error in the implemented algorithm, by loading output files and see if they match the Matlab code. This method was very efficient since it was very clear to find the exact place something went wrong, and handle it and in the end an outcome allowing the frequency to be determined.

Chapter 7

Conclusion

In summary the implementation of Little helper on the Blackfin has not been as fulfilling as the goal of the project. The project algorithm had to be changed when the limitations of the Blackfin was found with regards to the ability to change the sampling frequency. Based on this experience, it is necessary to have a good understanding of the used processor. The developer must have the knowledge of the limitations as well as the strengths of the processor. With this information the developer can easily see or predict a future hurdle with the algorithm.

The synergy between Matlab and the implementation on the Blackfin has worked flawlessly. Both in regards to the development of the algorithm, as well as the debugging and implementation on the processor. A troublesome thing with writing the algorithm in Matlab is that without the sufficient knowledge of the processor, the developer might make an algorithm which cannot or is badly optimized on the processor. If that is the case the developer has to go back to the drawing board and make a new iteration.

The final algorithm which uses the Hilbert transform is very impressive, with the high frequency resolution, and the surprisingly fast computation of a signal block. Of course it has it's constraints with regards to the problem when more sines are mixed together as a tone from a piano or a guitar is. The next part of the system, the sine generator, was not fully implemented so the effectiveness or constraints to this part is unexplored, but it works in Matlab. The issues there might be with speed, memory space and quantization are not present in Matlab, but are highly important on the Blackfin platform. It would have been really interesting to implement the next parts of the system to discover if there are fatal issues or if it is a flawless system.

To conclude the project has been formed over frequent iterations and many ideas, with many hurdles. The outcome, a simple sine pitch shifter has a partly working system on the Blackfin and a full functional implementation in Matlab.

Bibliography

Books

[6] H.S. Kasana. **Complex variables**. ISBN: 9788120326415. URL: https://books.google.dk/books?id=rFhiJqkrALIC&pg=PA14&redir_esc=y&hl=da#v=onepage&q&f=false.

Datasheets

[1] **AD1836A Data sheet**. Rev. A. Analog Devices. URL: http://www.analog.com/media/en/technical-documentation/data-sheets/AD1836A.pdf.

[2] **ADSP-BF531/BF532/BF533 Data sheet**. "Rev. I". Analog Devices. URL: http://www.analog.com/media/en/technical-documentation/data-sheets/ADSP-BF531_BF532_BF533.pdf.

[3] **C/C++ Compiler and Library Manual for Blackfin®Processors**. Rev. 5.4. Analog Devices. URL: http://www.analog.com/media/en/dsp-documentation/software-manuals/50_bf_cc_rtl_mn_rev_5.4.pdf.

Webpages

[4] Victor Fei, Olivia Gustafson, and José Villegas. **Piano Sight-Reading Assistant**. URL: https://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/f2014/org5/website/website/index.html.

[5] François Grondin. **Guitar Pitch Shifter**. URL: http://guitarpitchshifter.com/index.html.

[7] Katja. **Pitch Shifting**. URL: http://www.katjaas.nl/pitchshift/pitchshift.html.

[8] Matlab. **Hilbert Transform and Instantaneous Frequency**. URL: https://se.mathworks.com/help/signal/ug/hilbert-transform-and-instantaneous-frequency.html.

[9] A. Nagel. **It's just a little pitch correction**. URL: http://www.columbia.edu/~agn2114/index.html.

[10] Octave. **hilbert()**. URL: https://octave.sourceforge.io/signal/function/hilbert.html.

[11] **Piano Key Frequencies**. URL: https://en.wikipedia.org/wiki/Piano_key_frequencies.

Appendix A

Matlab Model

```
%% Load data in stead
Fs = 48000;
y = csvread('x_signal245.txt');
y(:,2) = []; % Remove extra column
blocksize = 512;

% Check if blocksize is valid
assert(ismember(blocksize, [2^7, 2^8, 2^9, 2^10, 2^11]) &&
↪  blocksize <= length(y));

% Create window
h = hamming(blocksize);

% Initial phase offset
phase = 0;

% Create figure for the IF of different blocks
figure
hold on

% Display target, input, new frequencies
disp("Input\t\tTarget\tNew");

for i = 0:(length(y)/blocksize - 1)
  y_block = y((i * blocksize + 1):((i + 1) * blocksize));

  ts = 1/Fs;
  T = length(y_block)*ts;
  t = 0:ts:(T - ts);

  % Apply window
  y_block = y_block .* h;
```

```
32
33    %% Home made
34    scale = 2^15;
35    z = getIF(y_block, Fs, scale);
36
37    % Find frequency
38    freq(i + 1) = (mean(z(floor(length(z)/3) :
      ↪  end-ceil(length(z)/3)))/(scale))*pi;
39
40    % Find nearest C-major frequency and create signal
41    pianoFreq = findPiano(freq(i + 1));
42    [out((i * blocksize + 1):((i + 1) * blocksize)), phase] =
      ↪  genSine(blocksize, Fs, pianoFreq, phase, max(y_block));
43
44    % Check the frequency of the new signal
45    z_new = getIF(out((i * blocksize + 1):((i + 1) *
      ↪  blocksize))', Fs, scale);
46    newFreq(i + 1) = (mean(z_new(floor(length(z_new)/3) :
      ↪  end-ceil(length(z_new)/3)))/(scale))*pi;
47
48    % Print the result
49    disp([num2str(freq(i + 1)), "\t", num2str(pianoFreq), "\t",
      ↪  num2str(newFreq(i + 1))])
50
51    % Plotting the IF of the different blocks
52    plot([i*blocksize + 1:((i+1)*blocksize -
      ↪  1)].*(1/Fs),z.*(pi/(2^15)));
53  end
```

Appendix B

Hilbert

```
function z = getIF(y, Fs, scale)
  %%%%
  % Returns the instantaneous frequency for any time
  % y is the input signal in time domain
  % Fs is the sampling frequency
  % scale is the scaling, primarily used for debugging and
  ↪  comparison with CC
  % Set scale to pi for normal scaling, 2^15 for CC comparison

  % hilbert
  N = length(y);
  y=fft(y);
  % Division by 2, in stead of multiplication to keep
  ↪  everything below 1 on crosscore
  y=[y(1, :)/2;
     y(2:N/2, :);
     y(N/2+1, :)./2;
     zeros(N/2-1, 1)];
  z=ifft(y);

  %% Angle
  z = atan2(imag(z), real(z));

  %% Unwrap
  % Scale to fit CC
  z = (z/pi)*scale;

  for i = 2:N
    x = z(i - 1) - z(i) + scale;
    y = 2 * scale;
    dif(i) = rem(x, y);
    if (dif(i) < 0)
```

```
      dif(i) = dif(i) + 2 * scale;
    end
    z(i) = (dif(i) - scale);
    z(i) = z(i - 1) - z(i);
  end

  %% Diff
  for k = 2:(numel(z))
    z(k - 1, 1) = z(k) - z(k - 1);
  end
  % Remove last diff
  z(end) = [];

  z = Fs / (2 * pi) * z;
end
```