# Algorithms in Signal Processors
# Audio Applications
# 2006

DSP Project Course
using
Texas Instruments TMS320C6713 DSK

Dept. of Electroscience, Lund University, Sweden

# Contents

# Part I
# Acoustic Cancellation of a Sinusoid Signal

Simon Hultgren, Jörn Jacobsson,
Mattias Nilsson, Per Nylander

**Abstract**

This paper is the result of a course in algorithms for digital signal processors. Our goal has been to develop an adaptive filter that acoustically cancels a sinusoidal tone in an arbitrary room. This goal has proven to be more challenging to achieve than we first thought. The report will cover a presentation of the problem and our solution. It will also cover the theoretical aspects of the algorithms we have chosen to implement, as well as our actual implementation in C. There is also a chapter on simulation, in which we describe the results we have achieved in simulation and tests using only part of the algorithm. Finally there is the result chapter in which we discuss our results and what could be done next.

# 1 Algorithms

In our project we use several different algorithms. Which all are based on the Least Mean Square-algorithm (LMS). To understand the LMS-algorithm it is important to know that it derives from the Steepest Descent-method (SD).

## 1.1 Steepest-Descent Method

The SD-method is a recursive method that uses a feedback system to solve a Wiener-filter iteratively in a step by step manner. The positive effects of the SD-method are that the method will converge to the Wiener solution without having to solve any of the Wiener-Hopf equations and also not having to invert the correlation matrix of the input vector. The basic idea of the SD-method is to calculate the gradient of the cost function and then to calculate the new filter coefficients in the opposite direction of the gradient.

### 1.1.1 The SD-method step by step

1. $R$ and $p$ are known. $R$ is the correlation matrix of the input vector $u(n)$ and $p$ is the cross-correlation vector between the input vector and the desired response $d(n)$.

2. Initialize the filter coefficients $w(0)$.

3. Calculate the gradient $\Delta J(n)$,
   where $\Delta J(n) = \sigma_d^2 - w^H p - p^H w + w^H R w$. The gradient will then become $\Delta J(n) = -2p + 2Rw(n)$.

4. Update the filter coefficients in the opposite direction of the gradient with $w(n+1) = w(n) + 0.5\mu(-\Delta J(n))$.

5. Repeat step three and four until the optimal solution of the Wiener-filter has been achieved.

Since this is a feedback system it is important to ensure that the system is stable. This is controlled by the stepsize parameter $\mu$ and the correlation matrix $R$. The stepsize parameter $\mu$ has to be between $0 < \mu < 2/max(\lambda)$ where the eigenvalues $\lambda$ come from the correlation matrix $R$. The largest eigenvalue can be related to the steepest descent in figure 1 but also to the shortest axis. If $\mu$ is chosen as above the method will converge and it will find the optimal solution. If $\mu$ is chosen bigger than the criteria the method will diverge and the optimal solution will not be found.

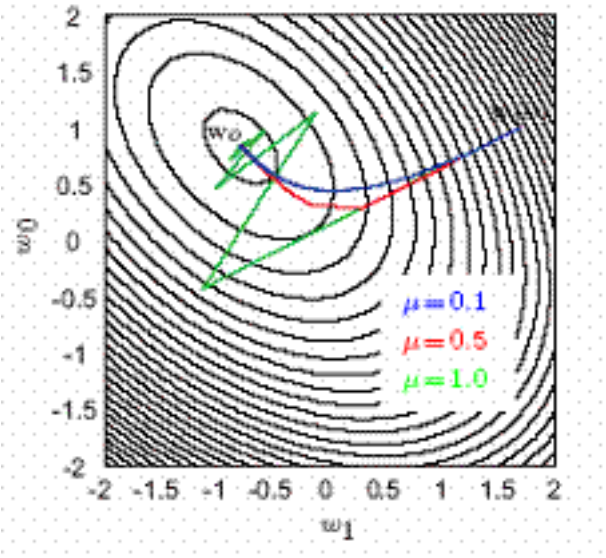


Figure 1: Steepest-Descent Method

### 1.2 Least-Mean-Square Algorithm

The difference between the LMS-algorithm and the SD-method is that in the SD-method the signals statistics are known but in the LMS-algorithm the statistics are unknown. The statistics in the LMS-algorithm are estimated which makes the LMS-algorithm an adaptive filter. Since the statistics are estimated a gradient noise will appear and the LMS-algorithm will not reach the Wiener solution but it will come very close to it. $R$ and $p$ are here estimated as

$$R(n) = u(n)u^H(n) \tag{1}$$

$$p(n) = u(n)d^*(n) \tag{2}$$

where $u(n)$ and $d(n)$ is the input vector and the desired response (see figure 2). In the LMS-algorithm the estimated values of $R$ and $p$ are used to calculate $\Delta J(n)$, due to this estimation the filter vector $w(n)$ will also become an estimation. The gradient vector becomes

$$\Delta J(n) = -2p(n) + 2R(n)w(n) \tag{3}$$

and we get a new relation for updating the filter coefficients vector

$$w(n+1) = w(n) + \mu * u(n)e^*(n) \tag{4}$$

where

$$e^*(n) = d^*(n) - u^H(n)w(n) \tag{5}$$

The iterative procedure is started with an initial guess $w(0)$.

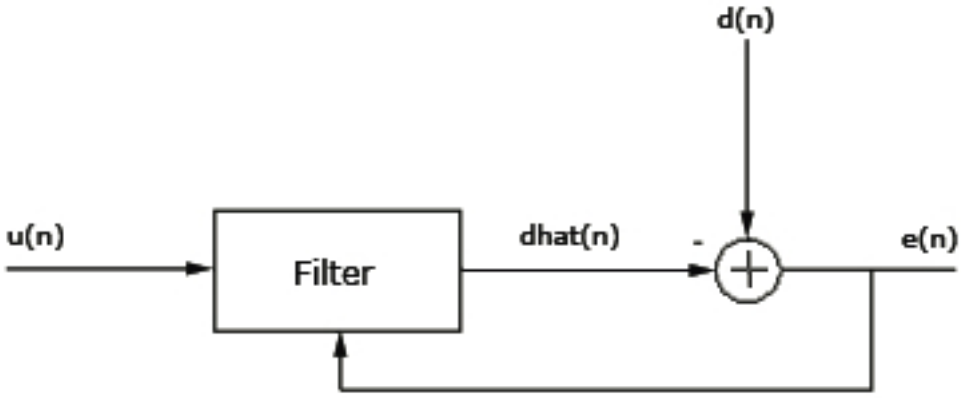


Figure 2: Least-Mean-Square chart

### 1.3 Normalized LMS

Since the LMS-algorithm suffers from a gradient noise when $u(n)$ is large, it is good to be able to overcome this error. Normalized LMS could be used instead. Normalized LMS works almost in the same way as the LMS-algorithm with the only difference that the product vector $u(n)e^*(n)$ is normalized with respect to the squared Euclidean norm of the input vector $u(n)$. The filter coefficient update equation then becomes

$$w(n+1) = w(n) + \frac{\mu}{\| u(n) \|^2} \cdot u(n)e^*(n) \tag{6}$$

### 1.4 Leaky LMS

Leakage is introduced to increase the robustness of the LMS-algorithm. If some of the filter coefficients drift then leaky LMS can be used to prevent this from happening. Leaky LMS comes with an increase in hardware and a degradation in performance. The update of the filter coefficient vector is

$$w(n+1) = (1 - \mu \cdot \alpha)w(n) + \mu \cdot e(n)u(n) \tag{7}$$

As you can see this is equivalent to adding a white-noise sequence of zero mean and variance $\alpha$ to the input $u(n)$.

### 1.5 Block LMS

Instead of updating the filter coefficient vector with the gradient vector for every sample as in the LMS-algorithm, we can update it for every L:th

sample. This is called a Block LMS. The improvement is that we get a better estimation of the gradient vector. This does not imply that we get a faster convergence. So why do we want to use Block LMS instead of the conventional LMS-algorithm? The answer lies not in the accuracy of the algorithm but in the complexity. With Block LMS the computational complexity is reduced and we get a faster system. This in turn mean that we can increase the sample rate of the system.

# 2 Solution

## 2.1 Presentation of the problem

Our project assignment has been to cancel an acoustic sinusoid by using a real time DSP routine in a certain place in a real environment. The sinusoid disturbance should be sent out from a separate speaker in a room, and a second speaker (connected to our DSP) should compensate for the disturbance. The position of a microphone should decide where in the room the cancellation should take effect.

## 2.2 Using negative interference to cancel the tone

The starting point of the whole assignment is the basic property that a sinusoid is a highly correlated signal, this makes it easy to identify and also to predict its behaviour. This fact makes the cancellation of a sinusoid possible. To be able to manually cancel out a sinusoid in a point you basically only need to know the frequency of the signal. You also need to be able to introduce yet another sinusoid with the same frequency, shift its phase and adjust the amplitude. Once the two sinusoids have the same amplitude and are in counter phase the cancellation is a fact. This is called negative interference. So in theory it was actually a pretty easy assignment, but there are some additional reality factors which we had to address. The most important additional factor was the property of the environment. In theory the environment is usually rather simple, e.g. it is echoless. In reality it can be far more complex, the sinusoids will yield echoes.

## 2.3 Our Model

The first step we made was to find a suitable model in the theory of adaptive filtering. The obvious choice was to start out from the interference cancellation model [1] (see figure 3). The first modifications we made were to declare the microphone's recording as the error ($e$) and then replace the error calculation with the microphone. Next we introduced two speakers and succeeding channels ($G1, G2$) preceding the microphone (see figure 4). Our next main concern was determining what we would use as reference signal in our model. In our first version (see figure 5) we made a bad choice. Our plan was to use the recorded error as reference, this way we would get the same frequency in both signals without any extra calculations or using the same source. The obvious problem with using the error signal as reference is that our assignment was to minimize the error. When the error is minimized so is the reference signal. The coefficients in the filter would then compensate to create a large enough output to cancel the disturbance. This would cause the filter coefficients to diverge. We were also concerned that minimizing the internal error might not minimize the error at the microphone. Another

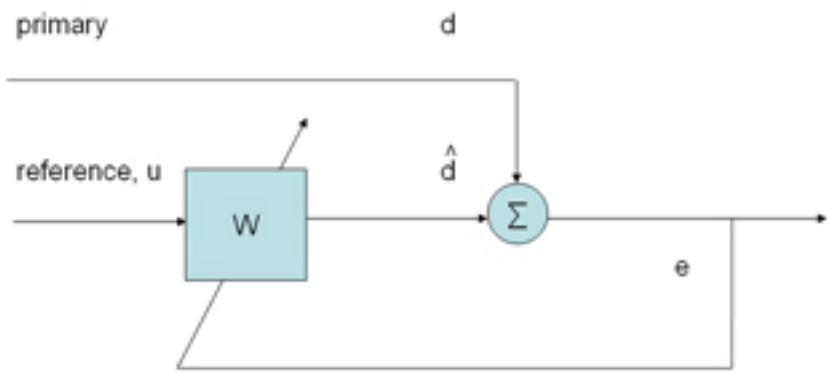


Figure 3: Theoretical interference cancellation model

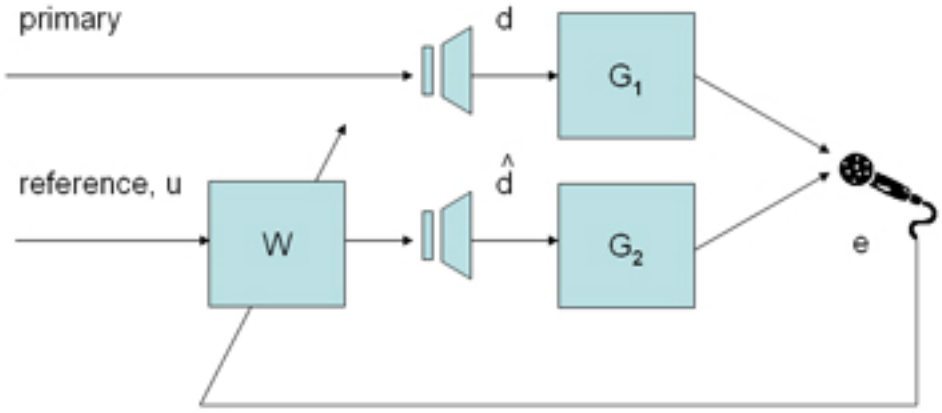


Figure 4: Our interference cancellation model

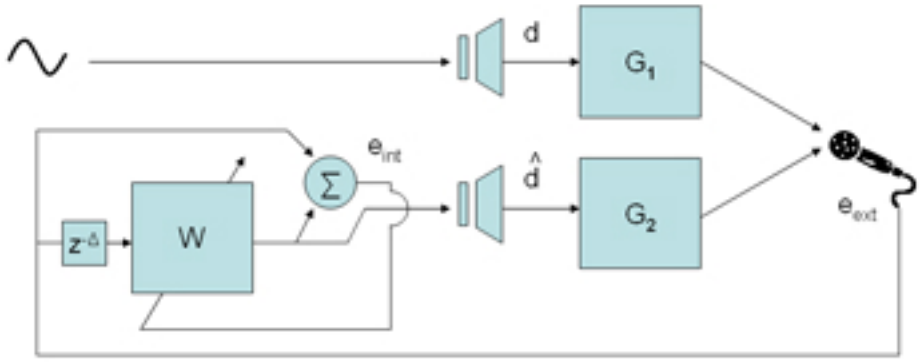


Figure 5: Our first draft

idea we discussed was to estimate the frequency of the measured error signal at certain points in time and then create a new reference sinusoid with the estimated frequency. This would introduce some extra calculations, and would also bring extra complexity to the routine. The frequency has to be measured before we have minimized the sinusoid and then frozen until the sinusoid changes frequency. Once the sinusoid changes frequency the error will increase and the algorithm should remeasure the frequency. We finally decided to use the same source for the primary and the reference signal, i.e. the same sinusoid would both be sent out through the disturbance speaker and sent in to the DSP. We added Additive White Gaussian Noise to the reference signal to avoid parameter drift (see figure 6). In later implementations our guess is that a frequency estimation model would be desirable. To see how the error signal behaved during the development process we used a mixing table, so that we could listen to the sound at the microphone. We also used oscilloscopes to analyse our signals and filter coefficients.

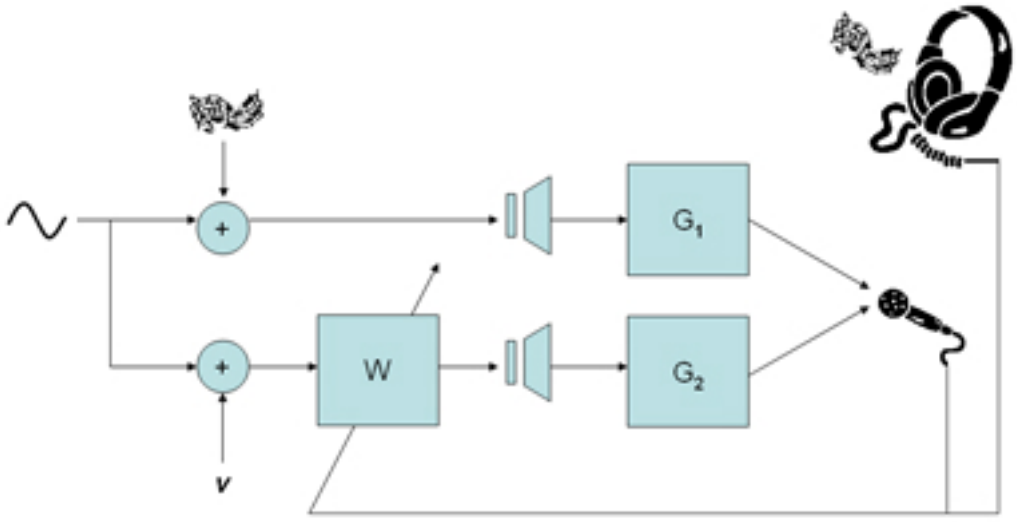


Figure 6: Our final model

# 3 Simulation

## 3.1 Matlab

As a first step we wanted to evaluate our solution by simulation. A rigid simulation would include filters with varying impulse responses to be applied to the interference signal and to the output of the FIR filter. These filters would simulate two aspects. One is the delays caused by the limited speed of sound in air and by having samples processed block wise in the signal processor. Another aspect is that both the interference signal and the cancellation signal are propagated through the room and result in multiple signals reaching the microphone. Due to the short deadline we had to focus on the actual implementation in C, and chose to only simulate the actual LMS algorithm. This was easily done and the result was promising. At this point we did not know that we were going to have to process samples block wise, so we did not simulate Block LMS.

## 3.2 Enclosed cancellation

Once we had tested our solution in Matlab we started implementing it in C for the DSP card. Implementation for the DSP card turned out to be very tricky and we experienced a lot of strange behaviours of the signals. To eliminate error sources we made a small alteration to the code, making it possible to calculate the error signal internally instead of reading the er-

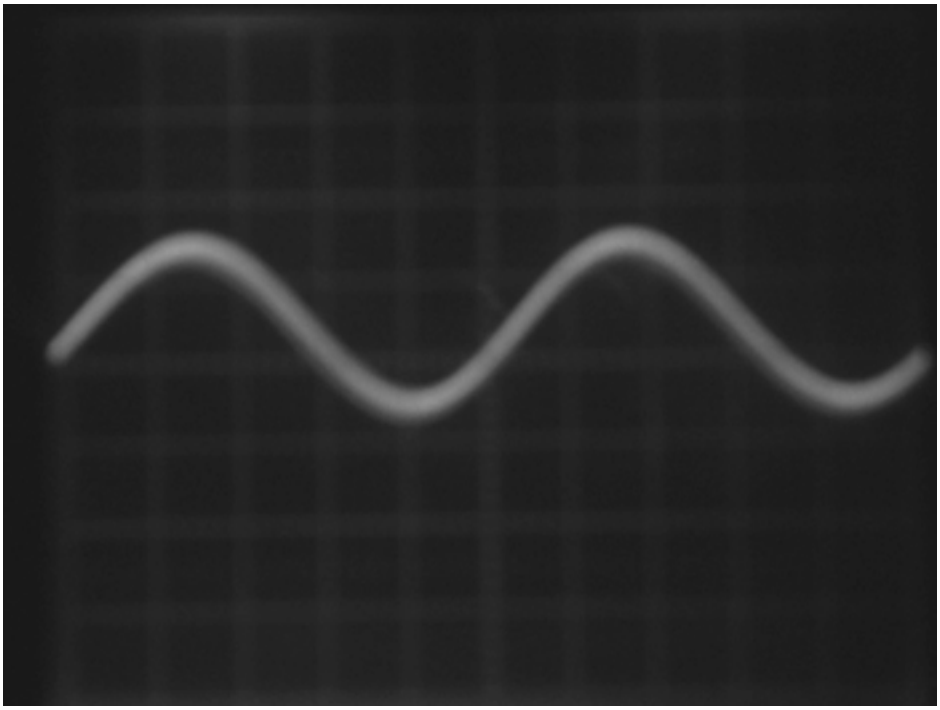

Figure 7: Sinusoid signal from one speaker

ror from the microphone. This gave us the opportunity to make sure the algorithm was correctly implemented. Of course cancelling a sinusoid in a computer program is a much easier task than cancelling it acoustically. Later in the project we switched to Block LMS which meant a slightly different implementation. Also the Block LMS implementation was tested with enclosed cancellation. We also wanted to know what results we could expect using acoustic cancellation. Thus our next simulation was to manually adapt the speakers' positions and loudness.

### 3.3 Manual Adaptation

In Matlab we created two sinusoid signals, the second of them being phase shifted. We then played the signals on two speakers and moved the speakers back and forth to find the best interference position. Once we found suitable positions for the speakers we adjusted the loudness of one of the speakers until we found a minimum in the microphone signal. The difference in the microphone signal was significant and can be viewed in figure 7 and figure 8. We now knew approximately what we could expect to achieve.

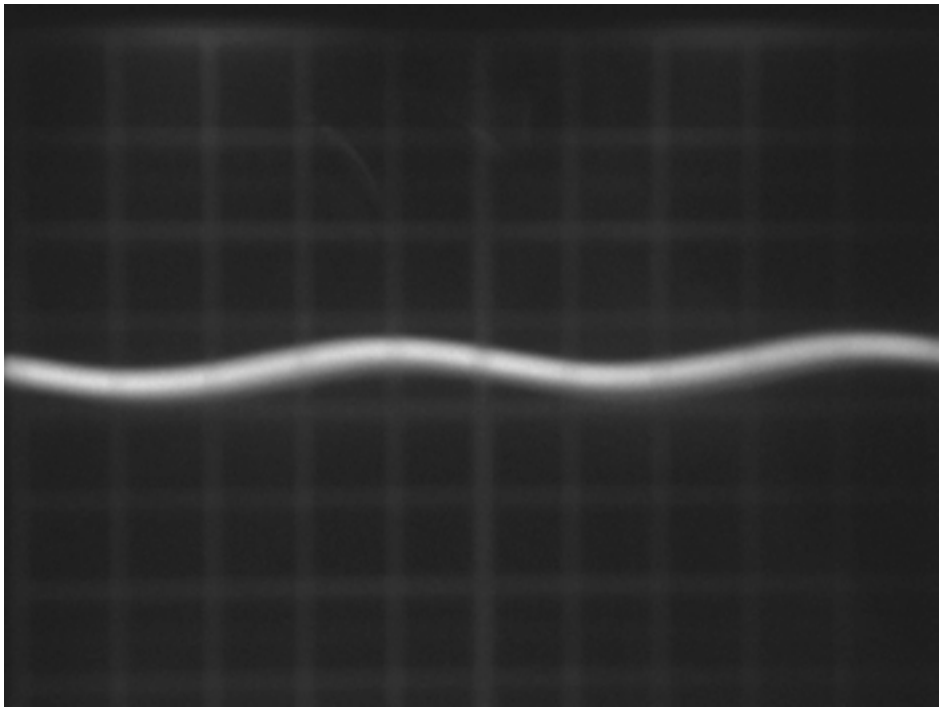

Figure 8: Sinusoid signal cancelled by another sinusoid signal, manual adaptation

# 4 Problems and solutions during our implementation

Our programming problems on the path to the best solution of the given problem can be divided into three categories:

- The acoustic room problem
- DSP problems
- Algorithm problems

## 4.1 The acoustic room problem

The LMS algorithm was our first choice of solution. We implemented the algorithm in Matlab with a rather large step size parameter ($\mu = 0.5$). It worked perfect in Matlab, but when we tried to use this solution with our Digital Signal Processor in an acoustic environment, the filter coefficients diverged and so did our cancellation signal. Even with a small step size parameter we could not get our filter coefficients to converge. The problem with acoustic signals versus signals in Matlab is that acoustic signals bounces in the room and enters our microphone delayed and with different amplitudes. Different positions in the room gave different results. Some positions were worse then others. After measuring the distances from the loudspeaker we made the conclusion that at a distance of half the wavelength, or a multiple of that, our algorithm caused the cancellation signal to diverge. Matlab does not have any of these problems since it runs in a closed environment. More on this in the result chapter.

## 4.2 DSP problems

The DSK 6713 is a floating point Digital Signal Processor. We have had our share of problems trying to program it. Having the benefit of programming C99 in another course this term, we thought this would be an easy task, however we have learned the importance of using as few variables and parameters as possible. The DSP used is rather sensitive and it is important to use only the memory you need and allocate it in your code. If you do not follow these recommendations you risk overwriting one or more of your values. We had several examples of destroyed values when adding a parameter. Minimized code is good.

## 4.3 Algorithm problems

As we mentioned before, the LMS algorithm was our first solution to the given problem. We created the sinusoid that was supposed to be cancelled in Matlab. In Code Composer Studio we programmed our LMS algorithm

together with a predefined sinusoid with the same parameters as in Matlab. Our filter length was 32 and we calculated one sample at a time with a sample rate of 8 kHz. We calculated the best step size parameter in Matlab and divided it by four. The code was compiled and loaded into the DSP. The result was not acceptable. The filter produced a drifting signal that never reached any satisfying result. After discussions and a lot of tests with different parameters, filter lengths and buffer sizes, our conclusion was that the DSP and our computer were timed using different clock signals. Because of this, generating sinus signals with e.g. 400 Hz will give two signals with slightly different frequencies. This lead to another test with no sinusoid generated in the DSP, but instead read from Matlab via one of the channels on line in.

Testing the new solution the cancellation worked when the microphone was positioned at other distances than those mentioned distances of a multiple of half the wavelength. This could be due to that at certain points in the room the reflections of the sinusoid create too complex patterns for the algorithm to cancel. At these points in the room the algorithm diverges.

Normalized LMS was our third solution. Since Normalized LMS is not as sensitive as LMS we thought that this was the correct way to reach the perfect cancellation, but this solution lead to the same effects that we had with the former solutions and no improvement was achieved.

Leaky LMS is an algorithm that punishes high filter coefficients and was for that reason our forth solution. When testing the Leaky algorithm we had a lot of programming errors and therefore temporarily abandoned this solution.

After minimizing and rewriting our code from scratch we decided to try Block LMS. So far our goal had been a buffer length of one sample. We now increased our buffer to 8 samples and it actually worked alright at some distances. Our filter cut the amplitude of the sinusoid to half its original size, but was quite unstable. It was a breakthrough.

We completed our Block LMS with a Leaky- and Normalized-LMS, and reached a more stable filter, but still with the problem that some positions worked better than others. Because of the positioning problem we added Additive White Gaussian Noise to the reference signal to avoid the filter from diverging at frequencies not present in the sinusoid signal. After implementation, our code resulted in a filter that filtered all frequencies, but still only cut half the amplitude.

Still not satisfied we discussed the latest solution with our supervisors and got a tip of how to compile our code more efficient. We now had compiled our code in Code Composer Studios' debug mode. With a compilation method without debug mode we reached a good result. We completed this with a sampling rate of 48 kHz and reached an even better result with more precision.

# 5 Results

We started of this project confident that we would be able to achieve, if not complete silence, at least a considerable reduction of the tone in the area around the microphone. The room in which we have been working is quite small, that gives us the problem with several echoes of the tone reaching the microphone creating a complex pattern for the LMS algorithm to adapt to. The LMS algorithm has one speaker in its use to minimize the amplitude of the sound measured at the microphone. In figure 9 and 10 the upper signal is the received signal at the microphone, the lower is the output from the filter, also sent to the second speaker. The LMS adaptation met our expectations from the manual adaptation (see section 3.3). To further prove the effect of our algorithm we listened to the measured error signal from the microphone. The effect was obvious, we heard a distinct difference in the amplitude of the sinusoid signal when we turned the algorithm on. We were also able to add music to the primary signal and the filter still only cancelled the sinusoid. But it seems to us that the pattern created at the microphone by the reflections of the sinusoid signal sometimes get too complex for the algorithm to cancel with only one speaker. Since our algorithm works flawless when tested with internal cancellation we can not think of another reason for the filter to diverge when the microphone is placed at certain points in the room. Other than that we are pleased with the results of our implementation. It shows that it is possible to cancel a sinusoid acoustically. Analysing and figuring out in detail why the algorithm diverges when the microphone is at certain places in the room could be a good next step for this topic. Perhaps giving the algorithm two speakers in its use to cancel the sinusoid signal would solve the problem.

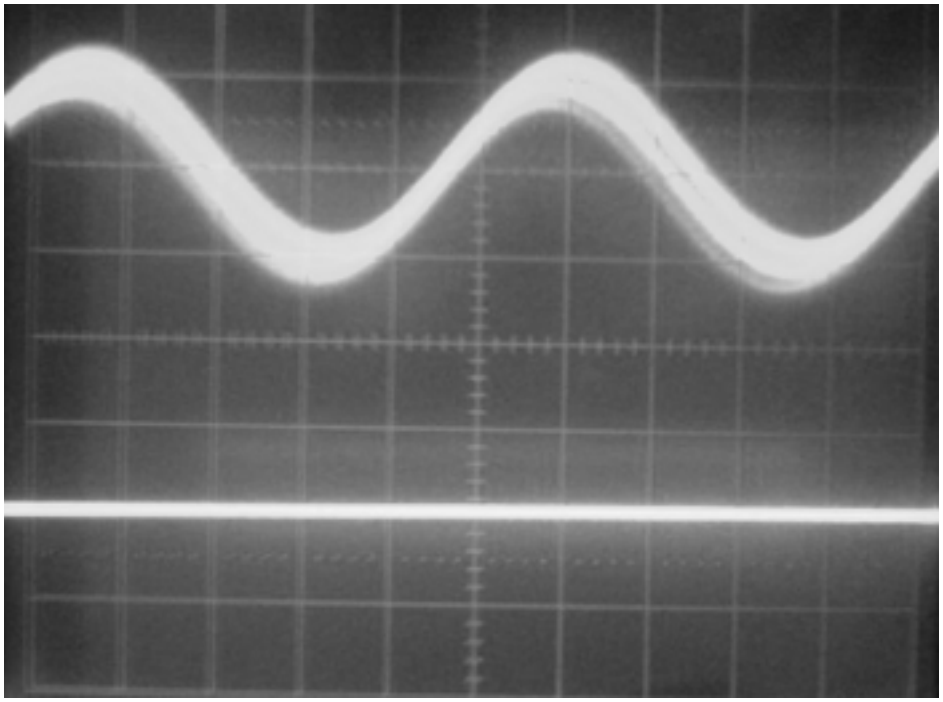

Figure 9: Algorithm not running

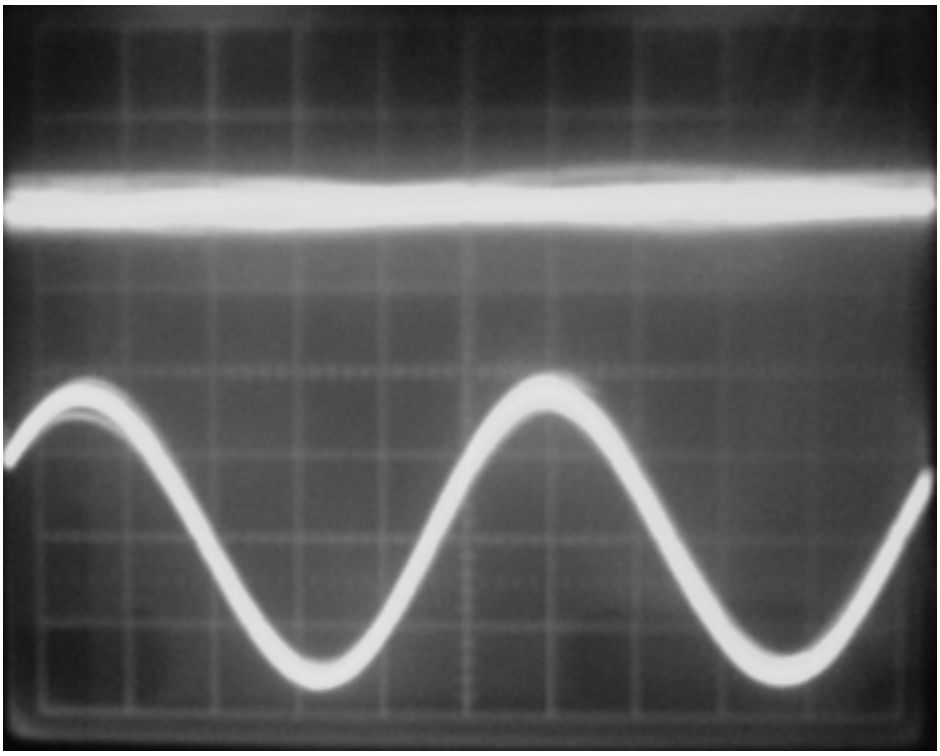

Figure 10: Algorithm running

## References

[1] Simon Haykin, "Adaptive Filter Theory", Fourth Edition, Prentice-Hall, 2001. Hardcover: ISBN 0-13-090126-1

# Part II
# Voice Alteration using LPC-Analysis

Mattias Ahlberg, Karl Eriksson,
Therese Andersson, Christian Larsson

**Abstract**

The main purpose of this project was for us to familiarize with working in a Digital Signal Processing-environment. The task was to alter voice, for example make it unrecognizable. Linear Prediction Coding-analysis were used to create a model of the vocal tract. By first filtering the sampled speech signal through the inverse of this model the voice characteristics could be removed. The output from this filter, the residual, is a whitened version of the input. This residual was then filtered through a modified model of the vocal tract to create a voice different from the original. The system was to be run in real time which introduced restrictions on the complexity of the implementation. Even so, several changes could be made. Among others the sound of the letter *e* was created to sound as an *a*.

# 1 Introduction

Voice alteration techniques attempt to change the characteristic of a speech signal. This transformation of the speech signal can be designed to have some specific characteristics. For example, disguising the original voice of a speaker making it impossible to recognize or alter the frequency properties of the voice. In this paper we present a voice alteration system based on Linear Prediction Coding (LPC).

LPC is a subtractive analysis/resynthesis method that has been successfully used in applications that handles speech and music. In 1978 it was implemented in Texas instruments "Speak and Spell", an educational toy [4]. Today, LPC is used in the Global System for Mobile Communications (GSM), the most common standard for mobile phones in the world.

LPC analyzes a data-frame of the speech and rebuilds an approximation of it. This approximation consists of filter coefficients and a residual. The coefficients is used to represent the frequency spectrum of the signal in the best way possible. When the sampled voice is filtered with these coefficients the residual is acquired. The goal of the LPC-coefficients is to minimize the residual. When transmitting, this approximation requires much less data than the original sampled voice.

The main purpose of this project was to get familiar with the TI C6713 DSP-card and the Code Composer Studio environment. In this paper it is shown how a voice alteration system can be implemented on a DSP-card. This project also gave an opportunity to collaborate in a way that may be similar to an engineers working-day.

In section 2 of this paper a description of the different parts of the LPC voice alteration system are presented. Then, the implementation on the DSP is described as well as the problem using real-time speech. This part also handles the different tests made. Results and conclusions are in the last section.

## 2 Theory

A block diagram describing the system is shown in Figure. 1. The input to the system is a real-time sampled speech signal. The blocks on the upper branch are all used to calculate parameters and filter coefficients. It is only in the blocks on the lower branch that the signal is being processed.

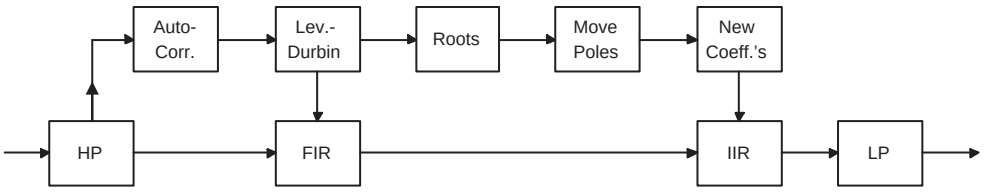


Figure 1: Block model of the system

The first step towards finding the model of the vocal tract is to find the resonance peaks. Speech consists of two types of sounds, voiced and unvoiced. The voiced sound decrease by 6 dB/octav, which results in a falling resonance spectrum. It is crucial to even out the spectrum before analyzing it. If this is not done only the peaks at low frequencies will be found. The straightening is easily done with a high-pass filter as the one seen in equation 1.

$$H_{HP} = 1 - 0.98z^{-1} \tag{1}$$

The coefficient is chosen to match the 6 dB/octav decrease. Automatically, high-pass filtering like this leads to a necessary low-pass filter at the very end of the chain to restore the signal. To avoid instability the coefficient of the low-pass filter is chosen a bit smaller. The low-pass filter is shown in equation 2.

$$H_{LP} = \frac{1}{1 - 0.95z^{-1}} \tag{2}$$

As mentioned in the introduction LPC-analysis is done. The first step towards finding the LPC-coefficients is to calculate the autocorrelation of the signal. The autocorrelation of a function is the cross-correlation of the function itself. It is a well used mathematical formula for finding patterns in a signal. It can be explained as the similarity between two samples separated in time in a signal. An estimation of the autocorrelation function can be expressed as a convolution of a block of samples, equation 3.

$$r_x(k) = \sum_{n=k}^{N} x(n)x^*(n-k) \tag{3}$$

With the autocorrelation the filter coefficients $(a_0, \ldots, a_8)$ that represent the signal in the best way can be calculated by using Prony's method. These coefficients is an estimation of the vocal tract represented as an IIR-filter (Infinite length Impulse Response - filter), equation 1.

$$H(z) = \frac{1}{a_0 + a_1 z^{-1} + \ldots + a_p z^{-p}} \tag{4}$$

Prony's method shows that by solving the linear equations in equation 5 (where $r_x$ is the autocorrelation function of the signal) the error of the model is minimized. By combining these equations with the modelling error in equation 6, a new set of linear equations is obtained. This set is given in matrix form in equation 7. The Levinson-Durbin recursion can easily solve this system. This is a recursive algorithm for solving a general set of linear symmetric equations. Levinson himself, who presented the algorithm, referred to it as a "mathematically trivial procedure" [2]. The full recursion can be seen in figure 2.

$$r_x(k) + \sum_{l=1}^{p} a_p(l) r_x(k-l) = 0 \quad ; \quad k = 1, 2, \ldots, p \tag{5}$$

$$\epsilon_p = r_x(0) + \sum_{l=1}^{p} a_p(l) r_x(l) \tag{6}$$

$$\begin{pmatrix} r_x(0) & r_x^*(1) & r_x^*(2) & \ldots & r_x^*(p) \\ r_x^(1) & r_x(0) & r_x^*(1) & \ldots & r_x^*(p-1) \\ r_x^(2) & r_x(1) & r_x(0) & \ldots & r_x^*(p-2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_x(p) & r_x(p-1) & r_x(p-2) & \ldots & r_x(0) \end{pmatrix} \begin{pmatrix} 1 \\ a_p(1) \\ a_p(2) \\ \vdots \\ a_p(p) \end{pmatrix} = \epsilon_p \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{7}$$

Since the model of the vocal tract is an IIR-filter, the inverse, a FIR-filter (Finite length Impulse Response-filter), is needed to remove the characteristics. After filtering the signal through the FIR-filter, see equation 8, only the residual remains. With this residual the original signal can be restored. If instead the residual is filtered through another model of the vocal tract, a different sound can be obtained. In for example GSM-telecommunications the former approach is used. The latter is what is being done in this project.

$$H(z) = a_0 + a_1 z^{-1} + \ldots + a_p z^{-p} \tag{8}$$

1. Initialize the recursion
   (a) $a_0(0) = 1$
   (b) $\epsilon_0 = r_x(0)$
2. For $j = 0, 1, \ldots, p-1$
   (a) $\gamma_j = r_x(j+1) + \sum_{i=1}^{j} a_j(i) r_x(j-i+1)$
   (b) $\Gamma_{j+1} = -\gamma_j / \epsilon_j$
   (c) For $i = 1, 2, \ldots, j$
   $$a_{j+1}(i) = a_j(i) + \Gamma_{j+1} a_j^*(j-i+1)$$
   (d) $a_{j+1}(j+1) = \Gamma_{j+1}$
   (e) $\epsilon_{j+1} = \epsilon_j [1 - |\Gamma_{j+1}|^2]$
3. $b(0) = \sqrt{\epsilon_p}$

Figure 2: The Levinson-Durbin recursion

The altered model of the vocal tract should have the same structure as the "real" one, seen in equation 1 only with new coefficients. When creating an IIR-filter it is of decisive importance that the filter is stable. The easiest way of deciding whether a filter is stable or not is to check if the poles are inside the unit circle. If this is the case, the filter is stable. To remain full control of the stability when changing the vocal tract model, it is the poles that are being moved sufficiently and not the filter coefficients directly.

A great concern when implementing this system is how to find the roots of a polynomial of arbitrary degree. Bairstow's method is one way of solving it. The approach is to iteratively, using Newton's method, find the roots to a quadratic polynomial. The full polynomial is then divided by the quadratic and the procedure is repeated until all the roots have been found. Since there is a possibility that the algorithm will not converge a limitation must be set on the number of iterations that can be allowed.

Bairstow's method always produce the complex conjugate roots in pairs which is convenient when proceeding in the search for a new model of the vocal tract. Since the poles should be kept complex conjugated also after they have been moved, knowledge of which poles that belong together is of great importance. How the poles should be moved depends on what sound effect the new vocal tract model is going to represent.

After the poles have been placed in their new positions the corresponding filter coefficients should be calculated. The residual created in the FIR-filter

is processed through the new vocal tract model and an altered sound is created. As previously mentioned there is also a low-pass filter to reverse the effect of the high-pass filter.

# 3 Implementation

## 3.1 Algorithms

This project is implemented on a Texas Instruments C6713 DSP-card, using TI:s own implementing environment Code Composer Studio. The DSP-card can not handle output values larger than $\pm 32000$. Because of this a limit restricting the output is needed. If the output value becomes to large the previous value is used instead.

The order of the vocal tract model is set to eight. The algorithm that calculated the poles of the polynomial is the most time consuming. Also, more poles means that greater precision is needed to obtain stability of the IIR-filter. This is why a greater order is not chosen.

When the autocorrelation is implemented, it is taken into consideration that speech is said to be static in 20ms. If a sample rate of 8 kHz is used to sample the signal, 160 samples will be produced during these 20 ms. Since the input vector to the DSP-card only contains 128 samples a buffer is needed. To speed up calculations a circular buffer with 160 values is made.

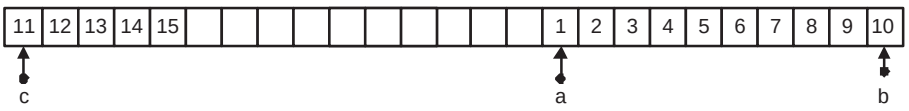


Figure 3: Example of a circular buffer.

In Figure 3 the numbers 1-15 are going to be placed in the buffer starting at point a. When the values reach the end of the buffer (point b) and there is no more room for new values, the next value will be put first in the buffer (point c).

The Bairstow algorithm that is used for finding the roots has one big problem. If it does not converge after a fixed number of iterations, precision is reduced and it starts all over again. The problem is that the precision is reduced towards zero. If convergence is never obtained the algorithm keeps reducing for eternity and gets stuck in an endless loop. To solve this the roots from the previous filter is used when the algorithm does not converge.

To change the poles they are divided up into magnitude and angle instead of imaginary part and real part. Section 3.2 describes in more detail how the angle is moved. Since these new poles are to be used in an IIR-filter a stability check is necessary. If a magnitude is found greater than 0.95 it is set to 0.95 to keep an error margin.

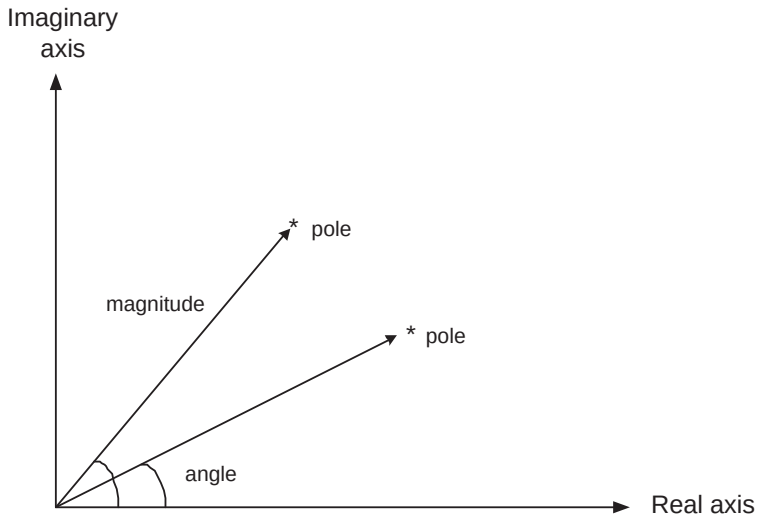


Figure 4: The poles divided up into magnitude and angle in the z-plane

Even though all poles are placed inside the unit circle there is still a possibility that the IIR-filter becomes instable. It is necessary to do a lot of mathematical calculations when the transform from poles to filter coefficients is made. Due to some rounding when multiplying many coefficients the poles can end up wrong. There is a greater probability that an 8-tap IIR-filter becomes instable than four 2-tap IIR-filters, so the latter option is therefore implemented.

As mentioned earlier it is necessary to know which poles that are complex conjugated. The poles are placed by the algorithm in a vector with the complex conjugated pairs next to each other. There is no knowledge of where in the vector a pair is placed since there can be real poles also. In the implementation of this system the vector is sorted with all complex poles placed first and the real poles last. Since four 2-tap IIR-filters is used, four vectors with old output samples needs to be saved. Every time the filter coefficients are updated the poles are also sorted. This leads to that the old output samples saved, can no longer be used and are set to zero.

### 3.2 Tests

A number of different tests were made to analyze what results that can be accomplished. Some of the tests and a short explanation is given below.

- A simple test is to move all the poles up in frequency. Two different frequency changes were tested. In the first test the poles were moved 200 Hz and in the second they were moved twice as far.

- To make the *"e"* sound like an *"a"* the poles within the frequency range 200-400 Hz are moved 200 Hz higher and the poles within 2100-2700 Hz are moved 1600 Hz down in frequency. Figure 5 and figure 6 shows the spectrum of an *"a"* and an *"e"*.

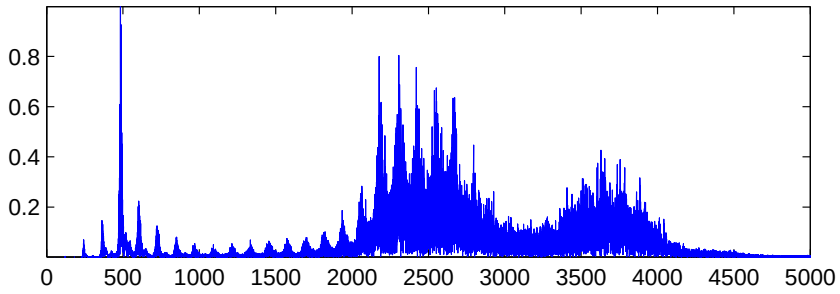


Figure 5: Spectrum of *"e"* up to 5000 Hz

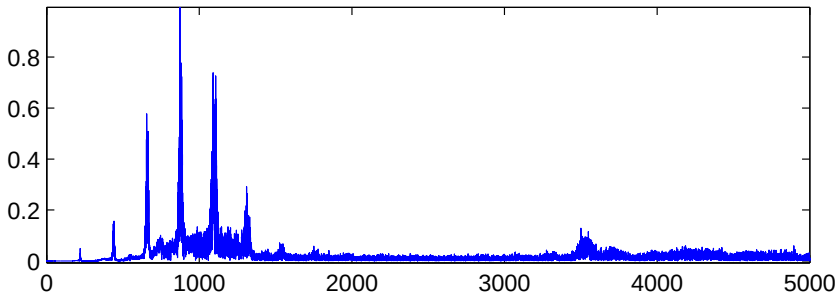


Figure 6: Spectrum of *"a"* up to 5000 Hz

# 4 Results and conclusions

## 4.1 Results

One of the more effective tests was to move all the poles 400 Hz up in frequency. Obviously the voice gets brighter, but it also made it unrecognizable. This result was far better than we ever expected.

Changing an *"e"* to an *"a"* also worked surprisingly well. The result of this test was depending on who was talking. Some voices worked better than others and this is due to the voice characteristics.

There are still some disturbances on the output. The reason why this may happen is because low frequency disturbances are moved into higher frequencies.

## 4.2 Conclusions

When we chose this project, our aim was to make a significant alteration of the voice. Also, an idea was to make one vowel sound like another. A successful result was accomplished.

Instead of a high-pass filter of order two, a high-pass filter of much higher order could be used. This would probably decrease the noise on the output signal.

By implementing a LPC-voice alteration system in real-time significant changes can be made. Even though our tests showed good results there are still both time and processor power for improvements to be made.

In the beginning of this project most of the time was spent on creating functional algorithms. Next step was to try to make these algorithms work together in real-time. From here on almost all time was spent on troubleshooting. When working on a project for a company there is no one to give you the answers. It is up to you to find the solution to the problem. Therefore troubleshooting was very educational and important to learn for future projects.

# References

[1] S.K. Mitra, *Digital signal processing, A computer-based approach*, McGraw-Hill, 2001

[2] Monson H. Hayes, *Statistical digital signal processing and modeling*, John Wiley and Sons, Inc, 1996

[3] Ping-Fai Yang and Yannis Stylianou, *Real time voice alteration based on linear prediction, in ICSLP - 1998, paper 0814*

[4] $http://www.datamath.org/Speech_IC.htm$

# Part III
# Talking orchestra

Johan Jönsson, Martin Kilsgård,
Marcus Lundström, Zoltan Michis

**Abstract**

This report contains a summary of the work that had to be done when implementing a "Talking Orchestra" on a Texas Instrument DSP-card. It will focus on the method of Linear Predictive Coding (LPC) cross-synthesis, as that was the method used in this implementation. The different functions that had to be used, such as auto-correlation and Levinson-Durbin recursion, will be explained both in theory and in the more practical sense when written in C-code. The code was mainly written directly in C but some of it was also tested in MATLAB. Hopefully this report can give others a list of "do's and dont's" when it comes to using the TI C6713 DSP- platform and the environment of the program Code Composer Studio.

# 1 Introduction

The main goal of the project was to successfully create an algorithm that produces a "talking orchestra" effect, using Linear Predictive Coding (LPC). In short the task was to make an instrument ?talk?. The inputs to the system should contain voice from a microphone on one of the channels of "line in" and music from the PC on the other. The output should then be the music with the "talking orchestra" effect. The whole system also has to work in real-time. Only negligible delay of the signal was allowed. To enable this a LPC filter needs to be used to mimic the effects of the oral cavity on a voice. The music can then be modified by applying the filter to it. The rate with which the speech and the music is sampled is crucial for the resolution, and faster sample rate naturally results in higher resolution. The input comes in blocks of 256 samples, 128 on each channel. These had to be separated and a major part of the problem then was to create suitable buffers with the right size for different operations. The finished algorithm, written in C-code, was implemented on a Texas Instrument C6713 DSP- card and the software used was Code Composer Studio.

## 2 Equipment

To solve this assignment following equipment has been used:

- MATLAB version 7.04
- DSP-card Texas Instruments
- Code Composer Studio
- Shure microphone

The DSP-card is connected via the computers USB-port for data transmission. Via this connection binary source files are downloaded to the DSP-card. When the DSP-card is in run mode audio signals are passed from the line- in input to the headphones- out output. Since the goal of this project is to mix voice signals with instrumental signals two inputs are needed. This is accomplished by splitting the stereo signal into two mono signals (voice on the left channel and instrument on the right channel). To play instrument signals the sound recorder program in Windows is used.

# 3 Theory

There are different ways of implementing a "Talking Orchestra", among them are the LPC (Linear Predictive Coding) based approach and CS (cross synthesis). CS was used by early vocoders and is not nearly as accurate as LPC which is today used in nearly all speech analysis applications.

The human voice is generated by the vocal cords and the vocal tract. The vocal cord produces periodic waveforms with many harmonics, these waveforms are filtered and shaped by the vocal tract, which includes the throat and nose. (It resembles a complicated piping system). This is then used to produce differences in harmonic content which results in sounds that we use in speech or song. There are another type of sounds, known as unvoiced or plosive sounds. A typical example is the pronunciation of the letters "t" or "p". These letters are produced without vibration of the vocal cords. They are instead produced by stopping the airflow in the vocal tract.

To be able to implement a "talking orchestra" one must first examine the human speech. The first step is to find the characteristics of voice and measure how its spectral characteristics changes over a certain period of time. This can be done with LPC analysis. These changes in spectral characteristics are mapped and stored for later use. To recreate the voice, one must simply apply the stored characteristics to the carrier wave using an inverse filter (IIR). In this case however, a different source of carrier waves will be used, namely music. When applying the characteristics of speech on the music it will sound like someone is speaking with music.

## 3.1 LPC analysis [2] [3]

To apply the spectral shape of the speech signal to music, an all pole IIR filter can be used. See equation 1 for the equation of an IIR filter. [2]

$$[H(z) = \frac{1}{A(z)} = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + ... + a_8 z^{-8}} \tag{1}$$

To calculate the filter coefficients one must first determine the autocorrelation coefficients $r_x$. Autocorrelation is a useful mathematical tool often used in signal processing, it?s applied to functions or series of values such as time domain signals. It's used to detect recurring patterns in a signal such as periodic waves or in this case a carrier wave. Autocorrelation is actually a cross-correlation with itself and is mathematically described in equation 2 below. [2]

$$r_s(k) = \sum_{i=k}^{159} s(1)s(i-k)k = 0...8 \tag{2}$$

When the $r_x$ coefficients are calculated the Wiener-Hopf equations for the optimum linear predictor can be presented.

$$\begin{bmatrix} r(0) & r(1) & r(2) & r(3) & r(4) & r(5) & r(6) & r(7) \\ r(1) & r(0) & r(1) & r(2) & r(3) & r(4) & r(5) & r(6) \\ r(2) & r(1) & r(0) & r(1) & r(2) & r(3) & r(4) & r(5) \\ r(3) & r(2) & r(1) & r(0) & r(1) & r(2) & r(3) & r(4) \\ r(4) & r(3) & r(2) & r(1) & r(0) & r(1) & r(2) & r(3) \\ r(5) & r(4) & r(3) & r(2) & r(1) & r(0) & r(1) & r(2) \\ r(6) & r(5) & r(4) & r(3) & r(2) & r(1) & r(0) & r(1) \\ r(7) & r(6) & r(5) & r(4) & r(3) & r(2) & r(1) & r(0) \end{bmatrix} \begin{bmatrix} a(1) \\ a(2) \\ a(3) \\ a(4) \\ a(5) \\ a(6) \\ a(7) \\ a(8) \end{bmatrix} = \begin{bmatrix} -r(1) \\ -r(2) \\ -r(3) \\ -r(4) \\ -r(5) \\ -r(6) \\ -r(7) \\ -r(8) \end{bmatrix} \quad (3)$$

The above matrix equation 3 can be solved by using a Gaussian elimination method, a matrix inversion tool (implemented in MATLAB) or the Levinson-Durbin recursion.

To improve the "talking orchestra" one must consider the nature of the human voice, a mans voice has a resonance frequency around 200Hz, a woman around 400Hz. The amplitude quickly drops off (estimated 12dB/octave) which explains the low-pass behavior of human speech. To adequately model the vocal tract a hi-pass filter must be applied to the speech, this results in a more linear function.

# 4 Method

## 4.1 First method

The method suggested in the project description included an LPC (Linear Predictive Coding) analysis of the voice, to modify the characteristics of another sound, e.g. music. The implementation contains a Levinson-Durbin [2] routine that decides the AR (Auto Regression) - coefficients of the voice. These coefficients are used in an IIR (Infinite length Impulse Response) filter for inverse filtering with the other sound, e.g music. This procedure makes the music "talk". See figure 1 for the principle idea of the method.

## 4.2 Second method

In order to improve the effect an additional FIR (Finite length Impulse Response) filter is inserted in front of the IIR filter. The input is convoluted with its on AR- coefficients to get the residue of the input. The residue of the input is then filtered through an IIR filter in the same kind of way as in the first approach. See figure 2 for the principle idea of this other method.

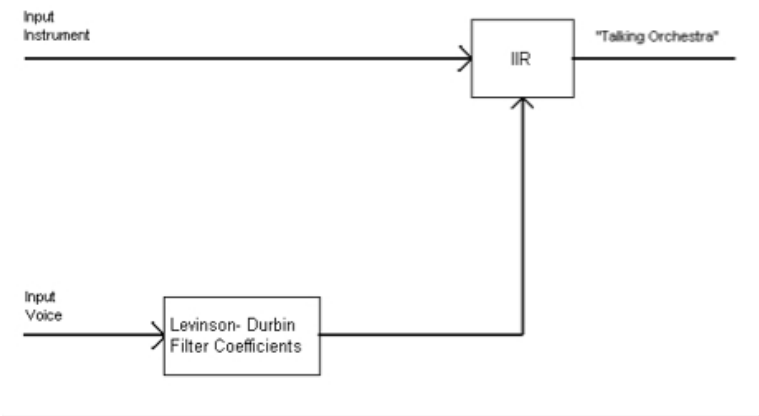


Figure 1: A principle schematic of the first method

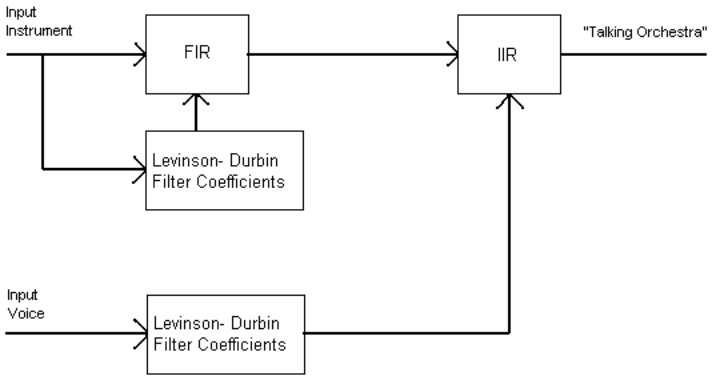


Figure 2: A principle schematic of the second method

## 5 Conclusions and problems

The thought behind the project was to make music sound as if it was talking, a "talking orchestra". The hard part was to decide the length of the buffer in the different stages, and synchronize the system. The most complex work was with the FIR filter and the hardware. The FIR filter was easy to simulate in Matlab, but to programme it in Code Composer Studios was much harder. The old input values had to be saved from the previous call to the present call into the filter. Managing the convolution to work correctly was very difficult. Most of the errors arose because the memory cells were wrongly assigned, or that the input values were wrong summarized in the filter.

We also had some problem with the DSP-card. The first circuit crashed and behaved oddly so we had to replace it. We also got some problem with the line in connector that appeared to have bad contact.

When running our program it crashed after an indefinite time. We allocated the problem to our buffers being to big for the internal memory. After allocating the buffers on the external memory everything worked fine.

## 6 Results

We chose a filter length of 8. It gave us some problem with the IIR filter that diverged so we decided to reduce the length to 6. When simulating the project, with no consideration of the resolution and efficiency, it worked fairly well. To test with other filters Matlab was used to give us the right filter parameters.

By filtering the music with it's on filter coefficients in the FIR filter the music had only the tones left. After the music passed the IIR filter, where the music was filtered with the filter coefficients calculated from the speech, the music sounded as if it was talking. To increase the resolution of the system we had to make the filter coefficients update faster. Because time was an issue this was left out.

# 7 Acquired knowledge

By doing this project the group learned:

- How to use Levinson-Durbin algorithm by pre- calculating the auto-correlation of a signal.
- How a simple function in Matlab can be much more complex to implement in the software on a circuit.
- How to cooperate in a group and plan the work to solve a project.
- How to search for errors in C-code, and systematic work through the program to see what code is correct or incorrect.
- How Digital Signal Processing is implemented in reality and how a digital signal sounds after it has been worked through different filters and algorithms.
- How to structure a mathematical expression to C programming code.
- How to use our knowledge in DSP theory to create a working circuit.
- How to use Latex to make a report.

## References

[1] Monson H. Hayes, *Statistical Digital Signal Processing and Modeling*, Wiley, 1996

[2] Martin Stridh, *Optimal Signalbehandling Labbhandledning*, Department of Electroscience, Lund University, Sweden

[3] *Linear Predictive Coding*, http://www.otolith.com/otolith/olt/lpc.html

# Part IV
# Pitch Estimation

Poyan Daneshnejad, Rickard Hammar,
Oscar Haraldsson, Per Vahlne

**Abstract**

This report describes a method of estimating pithes in audio, and the goal of the underlying project is to implement an algorithm on a DSP that handles audio input in real time. Pitch estimation is used to find certain tones in audio, and a typical application is a guitar tuner, where the input from the guitars string is processed and the output tells you wether to tighten or loosen the strings tension, or if the string is actually on key. The method of choise is called AMDF (Average Magnitude Difference Function), and operates by subtracting samples of the input with increasing lag from the actual one. If the input signal is tonal, this results in significant dips, and the quantity of samples between the dips is related to the frequency of the tone.

# 1 Introduction

Pitch detection may be useful in several areas, like for example, transcribing different kinds of music into a score. Another kind of application may be a guitar tuner, where the output tells you whether you need to tighten or loosen the strings tension, or if the string is actually on key. This paper is made on basis of a project with the goal of implementing a method of detecting the fundamental frequency, pitch, of a spoken, or more accurately, a sung signal. Every algorithm will first be tried out in Matlab environment because of its simplicity and familiarity. When they are confirmed functional, they are converted into C code and implemented on a Texas Instruments C6713 DSP.

# 2 Different methods of pitch estimation

The pitch estimation can be solved in several different ways. It can be made in the time domain or in the frequency domain. After examine a number of reports and comparisons between the different approaches like the FFT method, autocorrelation method, AMDF method and a few more we decided to go with the AMDF method. The first conclusion we made was that there is no method that is the best in detecting all pitches, some methods are preferable when detecting low frequencies and others are better suited for detecting high frequencies. The reason that the AMDF method was chosen was that it seemed to have a good average performance [1]. The FFT method did not seem to be very stabile for our application and the autocorrelation method was the choice of the last year?s project group and we wanted to do a new approach to the problem.

## 2.1 The AMDF method, our method of choice

The two most frequently used pitch detecting algorithms are the autocorrelation method and the AMDF method. They both work in the time domain and are very similar to each other. AMDF stands for Average Magnitude Difference Function. AMDF measures the similarity and periodicity of a given signal. The main advantage of this algorithm is the low computation cost and that it is relatively easy to implement. The AMDF function is given as

$$a(j) = \sum_{i=1}^{N} |x_n[i] - x_n[i+j]|, 1 \leq j \leq max[lag] \tag{1}$$

where the maximum numbers of AMDF values generated in each frame $k$ is $max(lag)$. The difference function is expected to have a strong local minimum if the lag $j$ is equal to or very close to the fundamental period.

For each frame, the lag for which the AMDF is a global minimum is a strong candidate for the pitch period of that frame [2].

### 2.1.1 Disadvantages with the AMDF method

There are some disadvantages in the AMDF approach that needs to be discussed. The main problem in the pitch detection algorithm is the high dependence of the background noise to the magnitude of the principle minimum of the frame. Another problem is the sensitiveness of intensity changes. A certain window length is required to be able to detect the right pitch. The length of the window determines how much the white noise will influence on the calculations. As said in [3] a recommended length of the window is somewhere between 20 and 60 ms. The longer the window is, the less the effect from the noise will be. However, if the window is chosen too large, it will take a lot longer to do the calculations, and since this should be a real-time working program, this is undesired. Another way to eliminate these problems is to use low pass filters.

# 3 The algorithm

The working method of the project is set up in two phases. In the first phase the algorithm is written and tested in Matlab. In the second phase the algorithm is converted from Matlab code to C code in Code Composer Studio. As test signal different tones with different pitches is made in Matlab with known frequencies and then distinguished by the program.

## 3.1 The MatLab algorithm

In the first step of the algorithm we wish to remove undesired noise frequencies from the signal to prevent the algorithm from detecting false frequencies. All frequencies less than 100 Hz are not desired since we can not distinguish them from noise in the AMDF function. In the same way all frequencies above 1 kHz are discarded, since most spoken sounds do not exceed this limit [4]. The lag is chosen to match the lowest detectable frequency that the AMDF function is meant to handle. The lag is dependent of the sample rate of the system witch is set to 8 kHz. The window length is empirical chosen and is a tradeoff between the algorithms ability to handle frequency variations and insensitivity to noise. What the program does is to take one window frame and run it through the AMDF function. A threshold is set and the local minima above this value are not taken into count. The number of minima is detected, and the number of minima is crucial to the performance of the algorithm. Too many will lead to faulty determination of fast frequency variations and this is typical if the frequency is low, since one period last for quite some time. The distance between the dips is related to

the frequency. Since several frequencies may occur at the same time, it is of interest to find the most common distance between the dips, and by doing this the fundamental frequency is detected. Figure 1 shows a typical plot of the AMDF function.

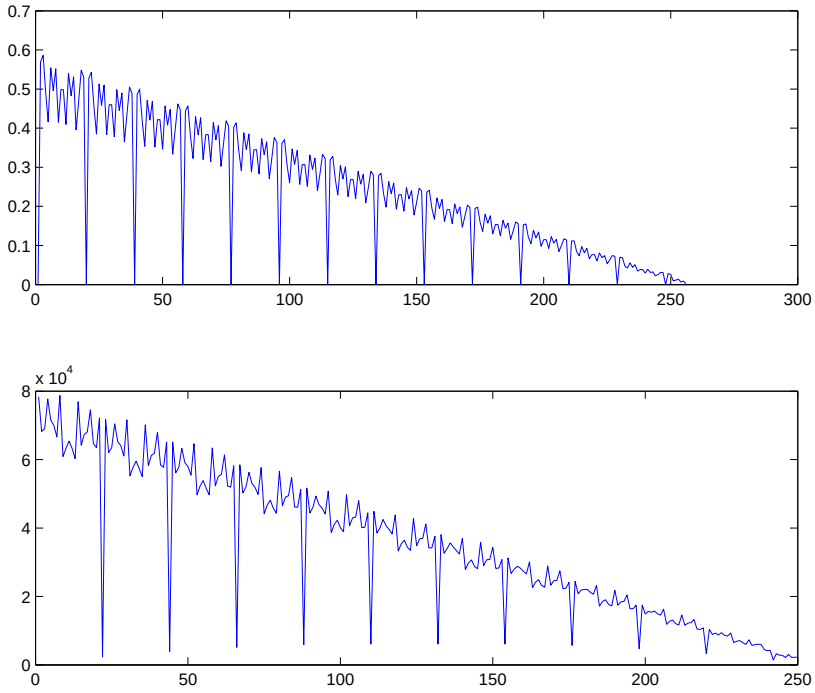


Figure 1: The first picture is a plot simulated in MatLab, the second plot is the signal processed by the DSP. The horizontal axis shows number of samples, and the vertical axis shows the AMDF.

## 3.2 Algorithm for the DSP

The instruction to the DSP has to be in assembly code or in C code. We prefer to use the C language since C coding is much easer than assembly coding. To get access to the processor a program named Code Composer Studio from Texas Instrument is used. At first, an algorithm to access the input to the DSP from the buffer to the processor was created. This algorithm has to copy the samples from de input buffer to our main process, and this has to be done repeatedly. Then an algorithm to detect the frequency from a simple, MatLab generated, sine wave was written. The algorithm has

to be able to make the detection of the pitch in real time, so the sampling of the input must be done at all times. The samples are put into a buffer in the memory. When this buffer is full the calculations in the AMDF function are done. Lastly the results are printed to a file and returned to MatLab for a graphic presentation of the found frequency. These four operations are run in four threads that are triggered by interrupts. The sampling of the input is a hardware interrupt, and the samples are put into audiosrc, which is a certain memory buffer. The calculations thread is triggered by a software interrupt when the audiosrc buffer is full, and here all calculations needed for the detection are done. The print to file operation is triggered when all the calculations are done, and returns values to a continuous running graph, see figure 2.

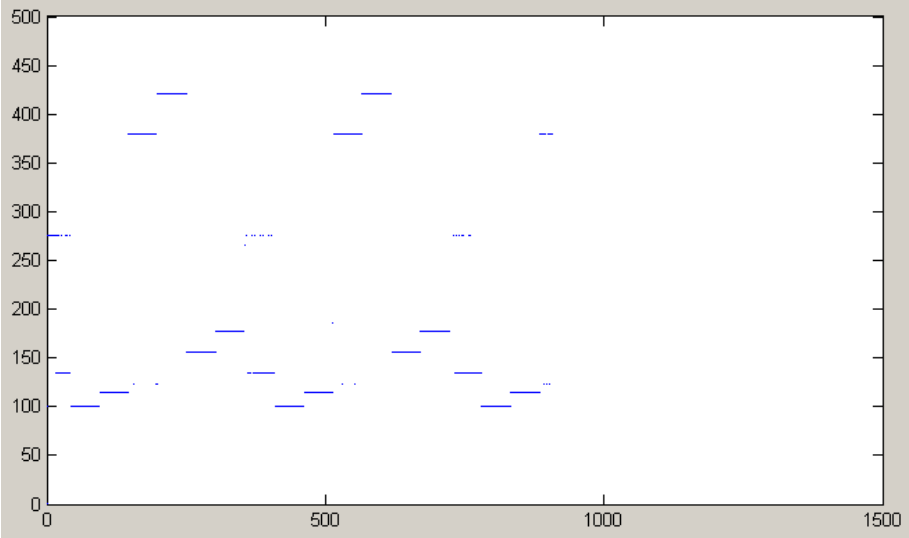


Figure 2: The picture shows the running graph with the data returned from the DSP to MatLab. The horizontal axis shows the time, and the vertical axis shows the frequency of the input audio signal.

### 3.2.1 Dealing with noise in the DSP

If the input is noisy, the distinctness of the dips will be reduced, i.e the dips will not be as deep as they are in the noise-free case. This will cause the algorithm to oversee minimums early in the frame, and only take respect to those situated in the latter part. This is an alias-like phenomena. To deal with this problem, an adjustable sloped threshold is applied. This threshold

is determined by the straight line equation

$$y = kx + 0.75m. \tag{2}$$

For each frame the value of m is determined by the mean value of the ten first samples in the frame. To set the starting point for the threshold we lowered the m by a factor 0.75, which we found was a good value, to exclude the noise.

# 4 Real time data exchange between Code Composer Studio and Matlab

RTDX Real Time Data Exchange is part of the CCS and is used to exchange data between Code Composer Studio and Matlab. RTDX uses an Object Linking and Embedding, OLE, programming interface to transfer debugging information to the host side CCS while the application continues to run in real time. RTDX makes it possible for different threads to send objects between each other without stopping the process. It also allows tracking and visualization of tasks being executed, gathering of real-time statistics on system execution and display of variables in real time [5]. The bandwidth of the RTDX is 8kbit/s.

# 5 The Graphic User Interface

The Graphic User Interface, also called GUI is an extension to the MatLab program. The interface for this project is designed in a way that the user is able to read the variables that the DSP currently uses and also to be able to change them to another desired value. The plot that appears on figure 3 is updated in real time. It can be set to show either frequency or tones in a specific octave. As the plot is printed the user can in ether in real time or afterward gets the values that have been calculated. When the GUI is set to show frequency it is able to show frequencies between 0 Hz up to 600Hz and when it is set to show octave it can show frequencies between 200 Hz up to 600 Hz.

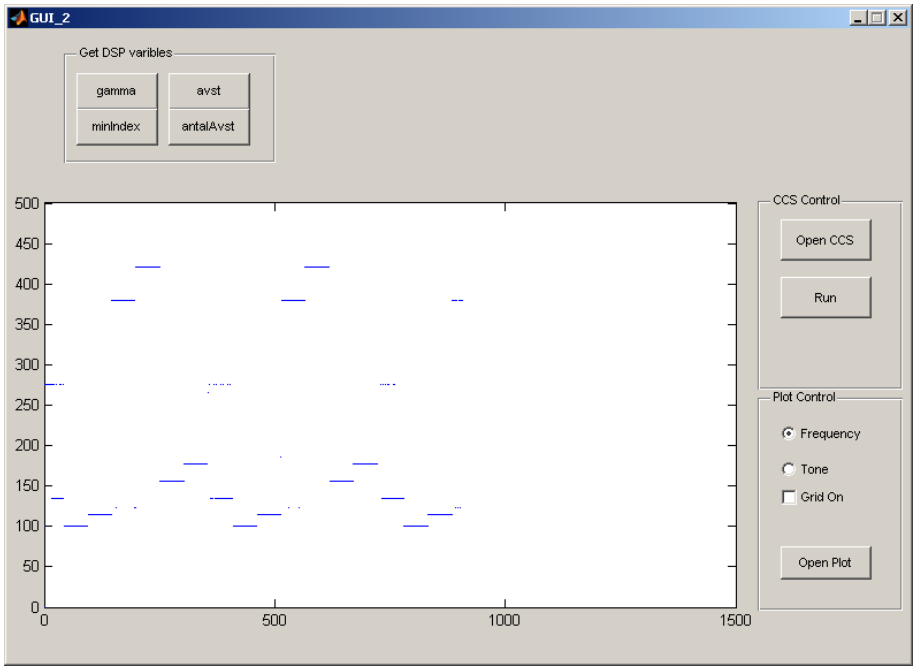


Figure 3: The Graphic User Interface.

# 6 Results and conclusion

This project has by no means been as easy as we predicted from the beginning. During the first weeks we focused on implementing the different algorithms in MatLab. We tried out several methods to find the minimum generated by the AMDF function. Our first approach was to simply concentrate on the first appearing minima and hoping that it was in fact a global minima. That, however, was not always the case of course, and it did not provide a sufficient amount of accuracy to the estimation of the pitch even though the first minima was a global minima. The more noise applied, the worse the function worked. When this did not turn out well, we made another approach, and tried to find the distance between the minimums. The initial thought was to calculate the average distance between the dips. A problem with this approach was that if one minima for any reason was not detected, the average value would become inaccurate. By finding the most common distance, the estimation became fairly on spot, so we decided to go with this approach.

We decided not to use any pre-processing of the signal at all. Mostly because this would increase the CPU load. Instead we only used post-processing where we used two non-linear filters to remove any kind of outliers. The first one is a three-tap median filter which removes single outliers and provides increased robustness to the second filter. The second filter is a five-tap median filter which removes bursts of outliers.

Because of our choice to use a sample frequency of 8 kHz, the accuracy of the function is somewhat poor at estimating frequencies around 1 kHz and above. This method is, because of this inaccuracy, more suitable for speech or singing than for instrumental sounds.

## References

[1] L.R. Rabiner et al. *A comparative performance study of several pitch detection algorithms*, IEEE transactions on acoustics, speech, and signal processing, Vol. ASSP-24, No. 5, October 1976

[2] http://purcell.ecn.purdue.edu/ speechg, 2006-03-01

[3] Y. Gong, J-P. Haton, *Time domain harmonic matching pitch estimation using time-dependent speech modeling*,IEEE transactions on acoustics, speech, and signal processing, Vol. ASSP-35, No. 10, October 1987

[4] http://www.ling.su.se/staff/hartmut/f0_m&f.pdf, 2006-03-01

[5] http://archive.chipcenter.com/dsp/DSP020520F1.html, 2006-03-01

# Part V
# Digital synthesis using static polymorphic techniques

Emil Björnson, Johan Heander, Jonas Åström

**Abstract**

We implement a flexible software FM synthesizer on a TMS320C6713 from Texas Instruments using C++ template mixin programming techniques. Hence oscillators and sound effects can easily be combined in different ways and scheduled down to sample precision.

We show that signal adapted interpolators offer a small but significant improvement in audio fidelity and discuss methods for practically alias free FM synthesis with arbitrary oscillator waveforms.

# 1 Introduction

A synthesizer is an electronic musical instrument that generates sound by different kinds of synthesis. The simplest case would be to modulate the signal from some signal source, for instance a sine oscillator. By altering the oscillator, modulate the amplitude, phase and frequency and by using digital filters different kinds of sounds can be produced.

The telephone inventor Elisha Gray became in 1876 perhaps the first electronic musician by discovering that you can control the sound from a self vibrating electromagnetic circuit. During the 1960s the first synthesizer that could be played in real-time was developed. Those early devices were usually made to experiment with the concepts of modularity.

Most modern synthesizers are based on digital signal processing (DSP) methods. Digital synthesis gives exactly reproducible sound under precise control, but high quality audio synthesis can be very computationally demanding.

# 2 Theory of sound synthesis

## 2.1 Aliasing in discrete real valued signals

Like in all digital signal processing we have to consider which signals that can be uniquely described by its discrete samples on a given sampling frequency. Sampling of the real part of a continuous complex valued signal is a surjective map $(R \longrightarrow C) \longrightarrow R^n$, where $n$ is the number of samples. More than one signal is mapped to each discrete real valued signal and those are said to be aliases of each other.

But aliasing of audio signals is better understood in the frequency domain. Frequencies higher than the sampling rate of the map are aliased onto frequencies below the sampling rate. And since the spectrum of a real valued signal is symmetric, taking the real part of a complex valued signal gives aliasing (negative and positive frequencies cannot be separated).

In audio synthesis, where we want to produce a signal with a specific spectral content, aliasing represents a severe distortion. It must be managed to achieve good sound quality.

Since the amplitudes of spectral components of continuous signals decrease with frequency, a common strategy is to sample at a multiple of the sample rate and apply a lowpass filter with cut-off at the target Nyquist frequency. The oversampled signal is then decimated to the target sample rate.

## 2.2 Wavetable synthesis

In wavetable synthesis the desired waveform is sampled with a given resolution during an integer number of periods. This sample is then resampled and looped at playback to reconstruct the sound. To achieve a more varied sounds different waveforms can be used during different phases in the sound, for instance during the attack, sustain and decay parts. [3]

One problem with wavetable synthesis is if the wavetable is used at a faster rate than it was recorded. Then the high harmonics might exceed the Nyquist frequency and produce aliasing. This problem cannot generally be solved by using a better interpolator in the resampler since it would use too much resources. What is used instead is a technique, that was originally invented in image processing, called MIP mapping [11].

The idea is that instead of only sampling the desired waveform with one single resolution, it is also sampled at half the resolution, one fourth the resolution and so on. Since the table creation is done only once, it is possible to use very expensive resampling techniques with low aliasing. At playback it is sufficient to use a simple polynomial resampler and just select the appropriate MIP map depending on the desired output frequency. This technique has a very low computation cost and uses at most twice the memory of the non-MIP mapped wavetable, regardless of the number of MIP maps.

By using a sufficient wavetable size, the interpolation error of linear interpolation can be made less than the least significant bit. If this uses too much memory, a more complex interpolator can be used together with a smaller wave table. One interesting interpolator is the N-point Enhanced Linear Interpolator ("N-ELI") [2] which uses a small symmetric FIR filter to approximate midpoints between samples from N nearby samples and then linearly interpolates in the upsampled signal.

The interpolation error of ELI is never better than linear interpolation of the wavetable sampled at twice the rate, but is a significant improvement over linear interpolation. Typically, the mean square error of linear interpolation is a factor 20 larger than that of 6-point ELI and a factor 8 larger than that of 4-point ELI.

The FIR coefficients used in ELI can be adapted to the waveform and table size, which gives a small improvement in interpolation error over a non-signal-adapted FIR filters. Using differential evolution [12] we computed optimal coefficients for 4 and 6 point ELI for triangle, square and sawtooth waveforms at table sizes 16, 32, 64, 128, 256, 512 and 1024. The evolutionary algorithm took a little more than 24 hours in total runtime.

## 2.3 Differentiated Parabolic Wave

Differentiated Parabolic Wave ("DPW") [13] is a method for computing a sawtooth wave by differentiating a parabolic wave. The parabolic wave has steeper spectral slope than a sawtooth wave, so sampling a parabolic wave results in significantly less aliasing than sampling a sawtooth wave directly.

The sampled parabolic wave is differentiated to restore the spectral shape to that of a sawtooth wave, which results in a sawtooth wave with reduced aliasing. DWP sounds surprisingly good given the very low computational complexity of the algorithm.

The method was recently presented by [13] but similar methods has been used earlier.

## 2.4 Frequency modulation

Frequency modulation (FM) is the process of modulating the frequency of one set of oscillators by the weighted output of another set. This generates sidebands that depend in a complex way on the waveforms and modulation strengths [10]. An oscillator being modulated and mixed into the output audio is called a carrier, while an oscillator modulating the frequency of another oscillator (for instance the carrier) is called a modulator.

In synthesizers dedicated to FM, modulators and carriers are often chained in an FM "matrix", which specifies modulation strength between each directed pair of oscillators.

Sidebands are created at both positive and negative frequency offsets from the carrier frequency. It can be difficult to implement FM well, since these strong sidebands alias heavily even for moderate modulation strengths. Positive sidebands fold and alias around the Nyquist frequency, while negative sidebands generate negative frequencies that fold and alias around zero if the carrier is real valued.

A common and incorrect viewpoint is that aliasing is an unavoidable part of FM synthesis. [1] To reduce aliasing, oversampling by a factor $\geq 2$ is necessary. By generating the waveform as a full complex signal the negative sidebands can be filtered out before they alias onto positive frequencies. Since sidebands may create components near DC, a highpass filter with cut-off around 30Hz may also be needed.

This makes FM rather expensive to implement for general waveforms and FM matrices with full audio quality. For each output sample, at least four oscillator samples needs to be computed and filtered through complex filter structures.

Many synthesizers do not bother with this, and implement a simple FM that will alias at extreme settings. This is especially common in synths with a subtractive focus where FM is a nice but not central addition.

### 2.4.1 Classical Frequency modulation

The simplest case of FM uses two sinusoidal oscillators and is a well-known technique to generate a harmonic spectrum with low computational cost. This is the standard textbook FM example, but it bears little resemblance to the FM structures actually used in recent FM synthesizers. It is described by the formula

$$y(t) = A\sin(2\pi f_C t + I\sin(2\pi f_M)),$$

where $A$ is the amplitude, $f_C$ is the carrier frequency and $f_M$ is the modulation frequency. $I$ is known as the *modulation index* and is defined as

$$I = \frac{D}{f_M}$$

where the *depth* $D$ controls the amount of modulation. If $I = 0$ the signal $y(t)$ will just be pure sine with carrier frequency. When the index is increased the frequency modulation will produce equally spaced sidebands both above and below the carrier frequency, see Figure 1. The distance between two sidebands is a multiple of $f_M$, and therefore an integer ratio between $f_C$ and $f_M$ will generate a harmonic spectra.

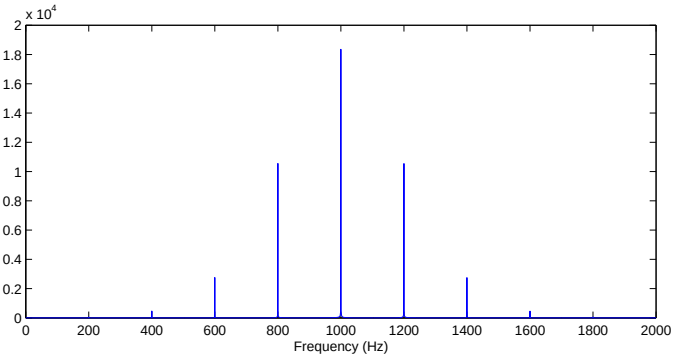


Figure 1: The frequency spectrum when performing the standard textbook FM, using $f_C = 1000$, $f_M = 200$ and modulation index $I = 1$.

The number of harmonic pairs with significant amplitude can be approximated as $I + 1$ and their bandwidth with $2(D + M)$. By choosing $I$ such that no sideband of negative frequency has significant amplitude one can guard against aliasing at a severe cost in expressiveness.

### 2.4.2 Phase Modulation

In practice, FM is often implemented as a varying phase shift. This is also called phase modulation, and has the benefit that no special care need to be taken for modulators with DC offsets. PM and FM are nearly equivalent,

to see this consider that frequency is the derivative of phase with respect to time. Modulating the frequency by a modulator $m(t)$ is equivalent to a time varying phase shift $\int m(t)dt$ and consequently phase modulation by $n(t)$ is equivalent to FM by $\frac{d}{dt}n(t)$.

### 2.4.3 Emulated FM

Instead of doing actual FM synthesis, FM can be emulated with wavetables. A few times per second a new wavetable is generated from the current FM parameters and these wavetables are then crossfaded. Since only a single period of the frequency modulated waveform needs to be generated, this can be done at an extremely high oversampling rate leading to virtually no alias at all after low pass filtering and decimation. Aliasing of negative frequencies folds onto harmonics of the fundamental and gives no distortion (but perturbs the amplitudes of the harmonics).

There is at least one commercial synthesizer using this technique, and it seems to work well in practice.

## 2.5 Oversampling

We have implemented a structure of cascaded oversampling stages, each implementing a doubling of the sample rate. Our audio algorithms can be run at any stage in a flexible manner, to trade computational expense for improved audio quality. The stages are activated and deactivated as needed.

The stages are decimated using a nuttall [8] windowed sinc lowpass filter with cut-off at 0.25. The convolution of a signal with this FIR filter is efficient to compute as the impulse response is symmetric and every even sample of the impulse response is zero.

The filter is linear phase, which was once thought to be an important property for high quality decimation filters. However, it has recently been shown that the pre-echoes inherent in linear phase FIR filters create audible artifacts that subtly degrade audio quality [5].

## 2.6 Envelopes and curves

It is customary to divide the lifetime of a sound into four different parts: attack, decay, sustain and release, (abbreviated ADSR) according to Figure 2. Applying different amplitude curves during the different phases is called using an *envelope*. Apart from amplitude it is also common to change other sound characteristics such as harmonic content and base frequency by individual ADSR envelopes.

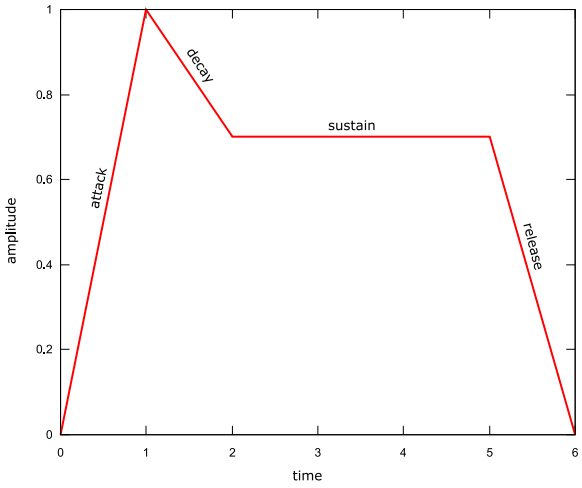


Figure 2: A standard ADSR envelope.

# 3 The MIDI protocol

For the purpose of transmitting musical data between electronic musical devices, MIDI (Musical Instrument Digital Interface) is an old but commonly used communication protocol. MIDI is not used to transmit audio, but to send digital information about the audio and the performance. There is a large collection of standard messages that could be transmitted, including basic sound controllers like note-on, note-off and adjustments of volume, modulation, etc. MIDI can also be used to control different kinds of hardware, like stages lights.

Two important properties of MIDI are the low bandwidth and the multiplex audio channel transmission. Most messages consist of two or three bytes and considering the fact that a standard MIDI port has a signalling speed of 31,250 bits per second, this ensures near real-time accuracy.

Four bits of most messages are used to specify which audio channel used. This makes it possible to separate 16 different audio channels on a single MIDI connection. In other words it's possible to control several different instruments.

If you play a certain note on different MIDI compatible instruments, the transmitted message will be identical. The outputted audio is depending on the channel and on the device where we connect our loudspeakers. Each channel is attached to one of 128 possible instruments (which one can be changed by just sending a message). These instruments could either consist of sampled sounds or be constructed through different kinds of synthesis, or both. Everything is up to the synthesizer that converts the MIDI signal into audio. That's why the same MIDI-signal could sound terrible on a cell

phone, but intriguing on a computer with a good sound card.

The MIDI 1.0 specification doesn't specify what instrument number that corresponds to what kind of instrument. The original thought was that each user should assemble her own bank of instruments. But when MIDI became important in computer multimedia there became a need of portability - each instrument number had to cause the same kind of sound on every system. For this purpose there is specification called General MIDI. It states that the first instrument number corresponds to a piano, and later on there are guitars, bass, strings, etc.

Instruments could be either monophonic or polyphonic. An instrument of the first kind has just one voice, i.e. it can only play one note at the same time (in the same way as a flute). A polyphonic instrument can have multiple voices acting at the same time (which is case of most acoustic instruments).

A collection of MIDI messages can be stored in the computer as a MIDI file for later playback and manipulation.

# 4 Scope

The scope of our project was to implement a digital synthesizer using the TMS320C6713 digital signal processor from Texas Instruments. The minimum requirements we had to fulfill was:

- Support at least two channels with different sound.
- Synthesize the sound in real time.
- Implement ADSR amplitude envelopes.
- Play MIDI-files sent from the host computer.

In addition to these minimal requirements we also aimed at the following:

- A flexible design where many different kinds of oscillators could be combined in different ways.
- An event scheduler with sample precision, for controlling notes, effects and envelope curves.
- Using oversampling and frequency shifting to avoid aliasing during FM modulation.
- Optimize the implementation to be able to use as many voices as possible.
- Also include the ability to play sampled sound, for drums and sound effects.

- Be able to apply curves to all parameters in an oscillator, not just the amplitude.

# 5 Implementation

To be able to fulfill our goals we choose to use C++ instead of ANSI C to program the processor, especially to make the oscillator structure as flexible and modular as we liked. One technique that proved very useful was the concept of *template mixin*, where you make the superclass of a class substitutable:

```
template<class Super> class Base : public Super { ... };
```

This technique makes it possible to write general audio oscillators and audio effects that can be combined in any order to create very complex oscillators.

## 5.1 Oscillator building blocks

The *template mixin* structure makes is possible to divide the software into several layers with well specified functions. The layering consist of a single top layer where the actual sample data is produced. Next follows zero or more middle layers which process and transform the sound produced in the top layer. At the bottom we find the postprocessing layer which converts the samples into the form expected by the rest of the system.

### 5.1.1 An overview of components for different layers

1. Top layer: Sound generation

   **ChebyshevOscillator** A pure sine oscillator approximated by a sixth order polynomial according to the Chebyshev formula[1] given in [14]. The maximal deviation from a the Matlab sine is $4.2 \cdot 10^{-7}$ and the mean deviation is about $2.51 \cdot 10^{-7}$. The Chebyshev approximation is noticeably faster than a standard sine.

   **WavetableOscillator** Uses a wavetable precomputed from the phase and amplitude of the harmonics. The wavetable is MIP mapped to avoid aliasing when played at high frequencies.

   **DPWSawtoothOscillator** Differentialted parabolic wave sawtooth wave [13].

   **DPWPulseGenerator** Variable pulse-width pulse wave oscillator built from two DPW sawtooth oscillators [13], [15].

[1] The polynomial with the smallest maximum deviation from a given function is called *minimax*. They are closely approximated by the Chebyshev formula on $x \in [-1, 1]$ because the error gets smoothly spread over the interval. Since sine has symmetrical oscillations we only need to approximate the first quarter of a period, so we worked with $\sin(\frac{\pi}{4}(x+1))$.

**SampledOscillator** Playbacks a sampled mono sound.

2. Middle layer - Sound transformation

   **TextbookDoubleFM** Standard textbook double-FM structure built from 3 oscillators of a given type.

   **TextbookNestedFM** Standard textbook nested-FM structure built from 3 oscillators of a given type.

   **Textbook2opFM** Standard textbook 2 operator FM structure built from 2 oscillators of a given type.

   **PMMatrixOscillator** A 2x2 phase modulation matrix accepting two general oscillators (i.e. ChebyshevOscillators).

   **ResonantLp** A resonant low-pass filter with variable resonance and cut-off that accepts one oscillator and filters its output.

   **PolynomialWaveshaper** Polynomial waveshaper, accepts one oscillator and distorts its output.

   **Detune** Mixes several mono oscillators with a configurable frequency difference.

3. Bottom layer - Stereo mix

   **MixedGenerator** Produces a stereo mix with envelope and panning given one of the mentioned oscillators.

   **DelayEffect** A simple delay line to give a more natural sound.

## 5.2 Event scheduler and time splicing system

To fulfill the requirement of being able to generate sound in real time, we introduced the concept of *events* as any general function that needed to be handled at some specific time.

The event scheduler consists of a binary heap priority queue of objects representing events. The head of the queue always contains the event that should be processed nearest in the future. An event object consists of a timestamp specifying the time when the event should take place, some kind of data (for example an audio generator or an envelope curve) together with a function that will be called by the scheduler at the specified timestamp.

While generating each buffer for the audio codec, the event scheduler processes the event queue by looking at the head of the queue. If the head should be processed within this buffer, it is removed from the queue and placed in a list. This is repeated until the event queue is empty or the head of the queue has a timestamp beyond the current buffer. Finally the scheduler processes the list of events scheduled in this buffer and calls the callbacks at the specified timestamp.

Audio generation within the buffer is time-spliced [6] around the event times. This is a standard technique and allows synthesis algorithms to be free of timing logic which is completely taken care of in the scheduling layer.

Time-splicing means that if an audio generator has no event that affects it within the buffer, it is called a single time with the full buffer length. If there are pending events affecting the generator the buffer is split into intervals at the event times and the generator called once for each interval with the events executed between the slices.

This allows the audio code to be an efficient 'steady-state' processing loop, and means that timing logic only needs to be written once. To increase efficiency, a minimum slice length can be defined allowing processing loops to be more heavily optimized. (loop unrolling, software pipelining, etc) However, we opted for greater timing precision at the cost of lost optimization opportunities.

## 5.3 Parameter structure

To be able to apply curves to all the parameters of a synthesizing module, we developed a general parameter concept. A parameter consists of a lowpass filtered line segment, specified by an initial value, a final value, lowpass filtering coefficient and the duration in number of samples. Every time the oscillator needs a certain parameter its value is computed using

$$v(t) = x(t) + c(v(t-1) - x(t)) \qquad 0 \leq c < 1$$

where $v(t)$ is the parameter value at time $t$, and $x(t)$ is the ideal parameter value given by the curve without lowpass filtering, and $c$ is the lowpass filtering coefficient.

To be able to use general curves, we used the event scheduler to assign new line segments to a parameter at given times, without resetting the filtering parameters. This makes it possible to apply smooth general curves to any oscillator parameter, which we have used for implementing the general ADSR envelope as well as frequency slides and filter changes during the tone duration.

## 5.4 Memory management

While the object oriented approach we used gave us a very flexible synthesis framework which was scalable up to thousands of simultaneous voices it required a very good memory management to be able to run on the limited platform of the DSP processor. Almost all events and different kinds of audio generators were represented using objects that had to be allocated on the heap. Since the lifetime of these objects were highly dependent on the parameters at construction, such as note duration and instrument type, it

would be difficult to identify one single part of the program responsible for the deletion of the objects.

The simplest solution to this problem is reference counting, where every object has an internal counter that can be incremented and decremented by external objects. In addition to this we also made each object hold a pointer to a deallocation policy, which described how the object should be deleted. This was made to be able to recycle certain objects instead of deleting them completely.

Every method that receives a reference counted object increments the counter in the received object, and decrements it again when it no longer has any interest in it. When the reference count for a given object reaches zero, it automatically uses the deletion policy to either delete itself from the heap or recycle itself in some manner.

## 5.5 Sampled sounds

To playback percussive sounds we use a sample bank stored in SDRAM. When the synthesizer starts up, it resamples the bank to the output sample rate using a high quality interpolated sinc interpolation resampling method [11].

## 5.6 MIDI decoder

The synthesizer was required to play MIDI data received from the host computer. MIDI only transmit the most crucial information about the musical score and the messages is not as exactly timed as our event scheduler. The result is that the capability of the synthesizer is not fully used, especially since we implemented many sound effects that cannot be controlled by the MIDI standard.

MIDI computer files are commonly used to imitate music from acoustic orchestras. If that was our main purpose the best way of getting the audio to sound well (in the meaning sound like a real orchestra) would be to base the instruments on samplings, in the way that modern sound cards does. This were never our intention with the synthesizer, hence we have concentrated our efforts at other kinds of instruments.

At the beginning of the project, a very basic MIDI decoder where implemented using the Matlab MIDI Toolbox [4] to perform most of the decoding. Later on a more complete MIDI decoder was implemented, capable of reading MIDI files sent over the Host Channel. This MIDI decoder is capable of parsing both MIDI type 0, and MIDI type 1, files with an unlimited number of tracks.

The MIDI commands [7] are converted to an internal event type and placed in a queue sorted by the starting time of the events. This queue is then processed bit by bit by a MIDI sequencer module and converted

into sounds schedulable by the event scheduler. This way we could control the amount of events that were transferred into the event scheduler, and it also made it possible to recycle the sound generators after they had finished playing. The recycling was accomplished by substituting the standard deallocator in the reference counting, with a special deallocator that pushed the used sound generators on a stack of free objects, that could then be reused by the MIDI sequencer.

## 6 Conclusions

C++ template mixins is an excellent programming technique that gives flexible and modular programs without compromising the runtime efficiency. We can conclude that this technique excellently matches to a software synthesizer. The template classes we have written can easily be combined in many ways. For example, a classic 2op FM matrix where the carrier and modulator are polynomially waveshaped pulse wave oscillators filtered by resonant lowpass filters can be built by instantiating the existing templates in a new order.

## 7 Further improvements

The construction of a synthesizer is a never-ending project, since you can always add new instruments and sound effects. On the other hand we are satisfied with the performance of our flexible synthesizer design, which makes it easy to add new features. One particular improvement would be to add software support for connecting a MIDI keyboard to the TMS320C6713. That way the real-time capability of the our synthesizer can easily be shown.

## References

[1] Alberts, L.J.S.M. (2005), *Sound Reproduction Using Evolutionary Methods: A Comparison of FM Models*, `http://www.fdaw.unimaas.nl/education/bachelor/conference/5.pdf`

[2] Berman, David, Melas, C.M., Hutchins, R.A., *Enhanced Linear Interpolation for Low Sampling Rate Asynchronous Channels*, 2001 IEEE Globecom Conference Record, vol. 5, p3025-3028.

[3] Bristow-Johnson, Robert, *Wavetable Synthesis 101, A Fundamental Perspective*, `http://www.musicdsp.org/files/Wavetable-101.pdf`

[4] Eerola, T. & Toiviainen, P (2004), *MIDI Toolbox: MATLAB Tools for Music Research*. University of Jyvskyl: Kopijyv, Jyvskyl, Finland. Available at `http://www.jyu.fi/musica/miditoolbox/`.

[5] Green, Steve (2004), *A New Perspective on Decimation and Interpolation Filters*, `http://www.cirrus.com/en/pubs/whitePaper/DS668WP1.pdf`

[6] Kleimola, Jari (2005), *Design and Implementation of a Software Sound Synthesizer*, Master's Thesis Helsinki University of Technology.

[7] N/A *MIDI Technical Fanatic's Brainwashing Center* `http://www.borg.com/~jglatt/`

[8] Nuttall, A. *Some windows with very good sidelobe behavior*, IEEE Transactions on Acoustics, Speech, and Signal Processing Volume 29, Issue 1, Feb 1981, p84-91.

[9] Roads, Curtis (1996). *The computer music tutorial*, The MIT Press.

[10] Schottstaedt, Bill, *An Introduction To FM*, `http://ccrma.stanford.edu/software/snd/snd/fm.html`

[11] de Soras, Laurent (2005). *The Quest For The Perfect Resampler*, `http://ldesoras.free.fr/doc/articles/resampler-en.pdf`.

[12] Storn, Reiner, Price, Kenneth (1995), *Differential Evolution A simple and efficient adaptive scheme for global optimization over continuous spaces*, Technical Report TR-95-012, ICSI.

[13] Välimäki, Vesa, *Discrete-Time Synthesis of the Sawtooth Waveform With Reduced Aliasing*, IEEE Signal Processing Letters, Vol. 12, No. 3, March 2005.

[14] Weisstein, Eric W. (1999-2006), *Chebyshev Approximation Formula* From MathWorld–A Wolfram Web Resource, `http://mathworld.wolfram.com/ChebyshevApproximationFormula.html`

[15] Wright, Joe (2000), *Synthesising band limited waveforms using wavetables*, http://www.musicdsp.org/files/bandlimited.pdf

# Part VI
# Reverberation

Andreas Nielsen, David Pettersson,
Martina Lundh, Milena Aspegren

**Abstract**

The echo environment characteristics of a room is called reverberation. By emulating the reverberations it is possible to create audio acoustics, simulating different enviroments. This effect is one of the most commonly used in music creation. The focus of this project was to implement a digital reverb using the Dattorro algorithm. This algorithm was selected since it is known of having a good sound quality with a fair computational need. Due to the fact that the Dattorro algorithm has several adjustable parameters, a hardware MIDI-interface was developed in order to change these using a standard MIDI-keyboard. The project resulted in a reverb effect with the ability to emulate several different enviroments in real-time.

# 1 Introduction

Everyone who has been in a church or in a concert hall may have noticed the special acoustics there. This phenomena is called reverberation, or short reverb, and depends on the size of the room and the materials of the walls. These special effects occur when the sound reflects against the walls before it reaches the listener as illustrated in figure 1.

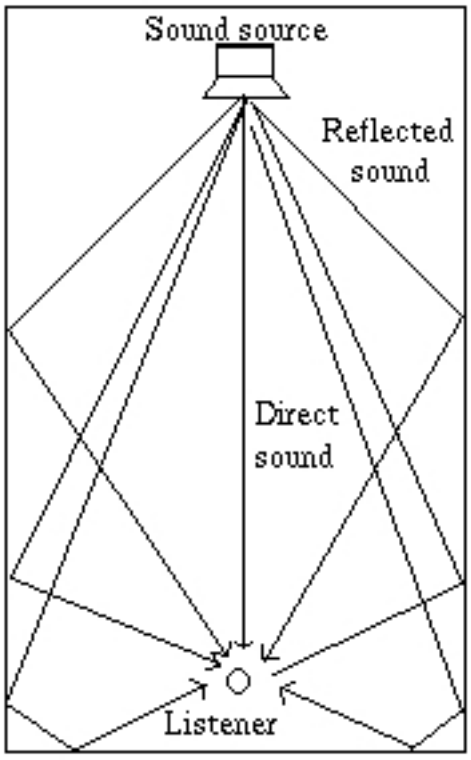


Figure 1: Reflections in a room with sound source and listener.

The reverb can be divided into three different parts; direct sound, early reflections and late reflections. The reflections, which are delayed, have a lower amplitude than the sound source and high frequencies have been absorbed in both the air and the reflecting surfaces. All this leads to a "halo" effect of the sound which describes the characteristics of the room.

Reverb is often used within the music industry to give the sound special effects and give the music more depth or make it lighter. Moreover reverb can be used to create a perfect virtual acoustic room which can not be made in a real physical room.

## 2 Algorithms

The technique behind the reverb effect algorithm is a mixture of time delays and filters. Manfred Schroeder was the first one to implement an artificial reverb algorithm in the 1960's. He used recursive comb filters and allpass filters which is the easiest way to create reverb. Schroeder's latest design was multitap delay line, which simulated the early reflections of the sound in a hall. This design was later to be used in most commercial reverberators. [1]

Improvements of the Schroeder algorithm has been made by i.e. Moorer, Gardner, Dattorro and Jot. Their algorithms are briefly described below.

- Moorer's algorithm was made to simulate air absorption by substituting the feedback gains of Schroeder's algorithm with first order recursive lowpass filters. By reconnecting the output taps of the FIR filter to the recursive part, the time density could be increased and the complexity of the late reverberation could be improved without using more computer calculations.

- To improve the Schroeder algorithm, Gardner used nested and networked allpass filters, which gave a flexibility, and did not require a lot of processing power. However, the design structure was not so analytic and the reverb had flutter and ringing. Gardner created three types of algorithms designed to simulate three different rooms.

- Dattorro was the first one to use a algorithm that contained both earlier well-known elements and new ones. His block-schedule had a non-recursive and a recursive part and the output was a sum of many taps.

- Jot used feedback delay networks (FDNs) in his algorithm, where the delays were represented in a matrix. The input signal is fed to all the inputs of the FDN and the output is the sum of the FDN outputs. The algorithm gives a smooth late reverberation and with enough time density for the order n=12 and n=16 of the feedback matrix.

In this project Dattorro algorithm was chosen because of good sound quality, quite easy implementation and the fact that it did not require too much processing power. [2]

## 3 Dattorro algorithm

The algorithm can be implemented with several types of sound sources and therefore a lot of different kind of reverberations can be created. As mentioned before the algoritm contains both a non-recursive part and a recursive part.

### 3.1 Non-recursive part

The non-recursive part includes a predelay, a lowpass filter and four allpass filters.. This first stage will create an echo density based mainly on two parameters; *input diffusion 1* and *input diffusion 2*. The use of predelay enables a direct sound followed by a delayed echo. The lowpass filter is used to reduce high frequencies with the parameter *bandwidth*.

### 3.2 Recursive part

The recursive part, which Dattorro called the tank, re-circulates the signal by using two cross coupled lines with identical structures. These correspond to modeling the effect in a real room and in figure 2 all the steps in the algorithm can be seen. The two lowpass filters and the gains, located in the tank, have the function of controlling the frequency dependency and decay properties, which are comparable to absorption and damping in a real room. The parameters that are used are; *decay diffusion 1*, *decay diffusion 2* and *damping*. Decay is used for the late reflections of the echo and is controlled by the parameter *decay*. [2] [3]

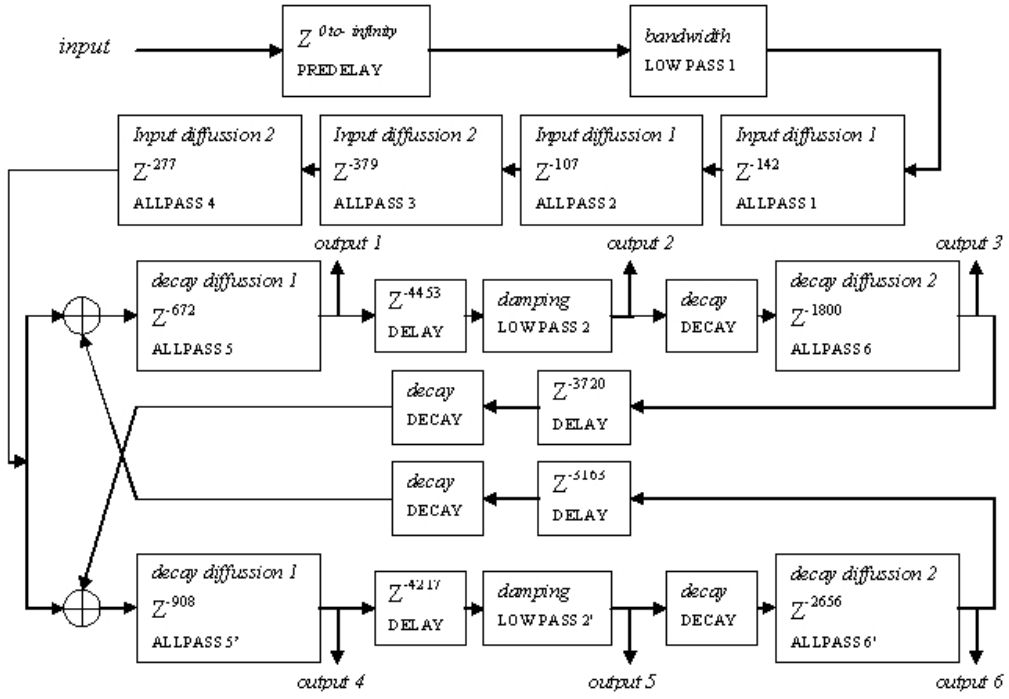


Figure 2: Block diagram for the Dattorro algorithm.

The left and right channel has different taps named output 1-6 in figure 2, which are to be combined in order to get the total output. How the outputs should be combined and scaled is described in the table below.

| *Left output* | | | *Right output* | | |
|---|---|---|---|---|---|
| *Node* | *Delay* | *Sign* | *Node* | *Delay* | *Sign* |
| output 4 | 266 | + | output 1 | 353 | + |
| output 4 | 2974 | + | output 1 | 3627 | + |
| output 5 | 1913 | - | output 2 | 1228 | - |
| output 6 | 1996 | + | output 3 | 2673 | + |
| output 1 | 1990 | - | output 4 | 2111 | - |
| output 2 | 187 | - | output 5 | 335 | - |
| output 3 | 1066 | - | output 6 | 121 | - |

Table 1: The combination and scaling of the left and right output in the Dattorro algorithm.

### 3.3 The low pass filter

One of the components in the algorithm is the lowpass filter which in the recursive part simulate air absorption. The transfer function is

$$H(z) = \frac{1-g}{1-gz^{-1}}, \tag{1}$$

where g is the gain. The block diagram is shown in figure 3.

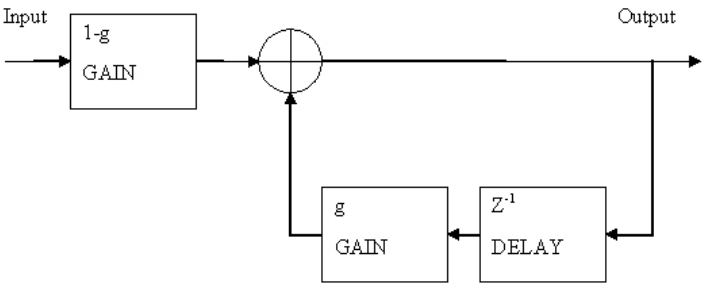


Figure 3: Block diagram for the lowpass filter.

### 3.4 The allpass filter

Another component is the allpass filter which has the transfer function

$$H(z) = \frac{g + z^{-k}}{1 + gz^{-k}}, \tag{2}$$

where g is the gain and k is the order of the filter as seen in figure 4. The different values of k can vary depending on where the allpass filter is located in the algorithm, see figure 2. [3]

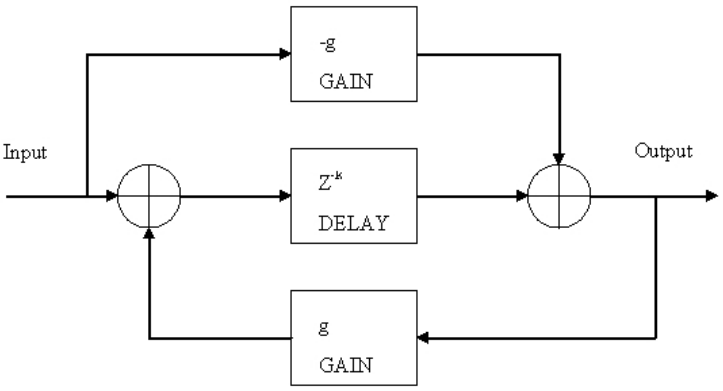


Figure 4: Block diagram for the allpass filter.

## 4 Implementation

In the first approach the algorithm was implemented using C programming language. The use of C language resulted in an algorithm with some drawbacks. The code was not easy to overview and making changes in realtime were unnecessary hard. To solve these problems it was a natural decision to use C++ instead of C. By changing language it enabled object orientated programming. This was a big advantage since the Dattorro algorithm is built by using blocks of delays, allpass- and lowpass filters. Each block could be implemented as an own class object, which simplifies the structure of the code. The class objects could then easily be connected in a chain, transferring samples to another.

### 4.1 Memory management

Since the Dattorro algorithm uses many delays, there is a need of optimized buffer management. This problem can be solved using circular buffers which

speeds up data access and minimizes buffer memory. A circular buffer uses three pointers: memory location, read and write. Each time data is written to the circular buffer the write pointer is incremented. When the pointer reaches the end of the buffer, the pointer wraps back to the start. The read pointer is incremented in the same manner as the write pointer. This allows re-use of allocated buffer memory and needs only a few instructions to access buffer data.

In order to simulate environment reverberations, different delay length are required. A late reflection can vary from a few milliseconds up to a couple of seconds. Assuming that a sample rate of 44 kHz is used, the late reflection could be represented by anything from 200 to 80000 samples. Since the internal heap size in C6713 DSK is limited, usage of external memory is required. In order to use this memory in Code Composer Studio, MEM-library have to be included. The allocation of external memory is made by calling MEM_calloc() and de-allocated with MEM_free().

## 4.2 Class design

The class hierarchy in C++ enables inheritance that can be very useful when building algorithm blocks. This is why the project in an early stage focused on implementing a super class that could be inherited by each algorithm block. The resulting UML-diagram is shown in appendix figure 8. By connecting blocks to each other, it is possible to create a chain structure. This structure simplifies implementation of the Dattorro algorithm. Another benefit of using classes is that the constructor enables initial calculations for each block, thus minimizing unnecessary computations for each sample.

## 4.3 Hardware design

Due to the lack of making true real-time changes to parameter values in Code Composer studio, the decision was to develop a hardware MIDI-interface. This would make it possible to change parameters via a standard MIDI-keyboard.

The MIDI protocol uses a USART at a speed of 31.25 kBaud/s with one start bit, eight data bits and one stop bit. [6] In order to connect the DSP-card to the MIDI-interface, the DSP-card had to be extended with an USART. By using a 8-bit microcontroller (PIC16F876) the USART data was throughput to the DSP-card $I^2C$-interface. The MIDI-interface consists of only one optocoupler (6N138) to isolate the MIDI-line from the other hardware and a few passive components (see appendix figure 9). By using the MIDI-hardware, MIDI-messages could be received using the integrated $I^2C$-interface in the C6713 CPU. In Code Composer studio the CSL_I2C module had to be included in order to use the $I^2C$-API.

# 5 Results

The Dattorro algorithm uses many adjustable parameters for setting environment reverberation. One project objective was to emulate reverberations of different rooms. Finding suitable parameters for the emulation was harder than expected. Using the MIDI-interface enabled change of the parameter values in real-time without compiling and rebuilding the project. By printing the parameter values, when changes through MIDI were made, the values could swiftly be found.

Emulation of reverberation using the Dattorro algorithm resulted in impulse responses, which take into account the air and reflection material absorption. The resulting impulse responses for different rooms are presented in figure 5- 7. In the impulse response for the large room (see figure 6) it is also possible to see the digital delay, which was added to the algorithm in order to emulate larger rooms.

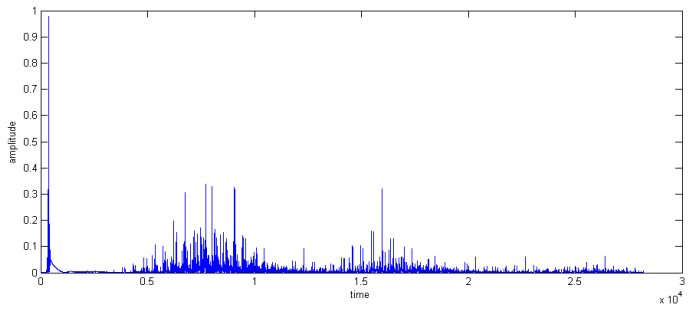


Figure 5: Impulse response for our simulated small room

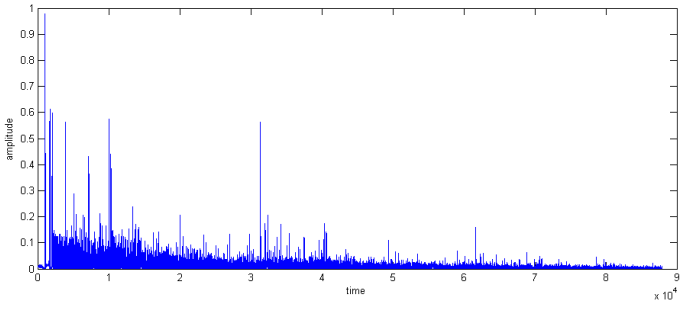


Figure 6: Impulse response for our simulated large room

In the beginning of the project the plan was to implement main parts of the Dattorro algorithm within two weeks. The rest of the time was reserved for trouble shooting and debugging, this due to the fact that software devel-

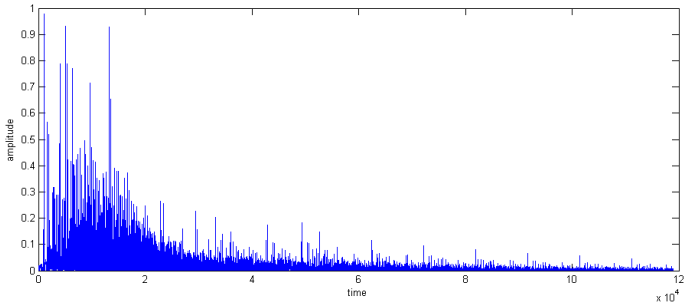


Figure 7: Impulse response for our simulated bathroom

opment is somewhat complex in embedded systems. Managing to keep the original plan permitted an excess of MIDI-hardware design.

Since the Dattorro algorithm uses a lot of resources and the fact that C++ language was used, which significantly increased the complexity, it was found hard to use higher sample rates. In order to use acceptable sample rates, optimizations and tweaking had to be made. The bottleneck in the implementation was the memory access, which was counteracted by merging algorithm blocks. Better results was achieved by disabling debugging and tweak the settings in DSP-BIOS.

## References

[1] Curtis Roads, *The computer music tutorial*, ISBN 0-252-18158-4, page 472-484, 1996

[2] Ola Lilja, *Algorithms for reverberation - theory and implementation* Master Thesis LTH, 2002

[3] Fernando A. Beltrn, Jos R. Beltrn, Nicolas Holzem, Adrian Gogu, *Matlab Implementation of Reverberation Algorithms* http://homepage.te.hik.se/personal/tkama/audio_procME/papers /matlab_reverb.pdf, 2006-01-24

[4] W.G. Gardner *The virtual acoustic room* Master Thesis MIT, 1992

[5] Texas Instrument *DSP - Digital Signal Processing* http://www.ti.com, 2006-01-24

[6] Jeff Glatt *MIDI Specification* http://www.borg.com/ jglatt/tech/minispec.htm, 2006-02-23

# Appendix

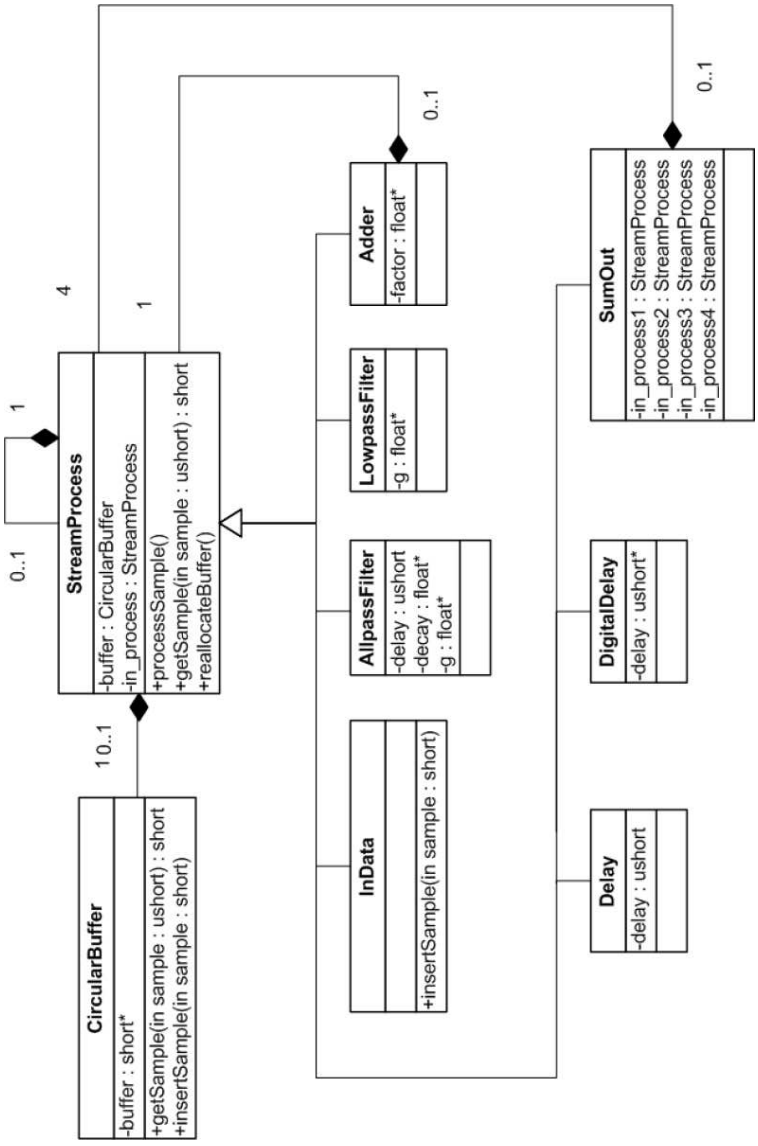


Figure 8: UML-diagram of algorithm blocks

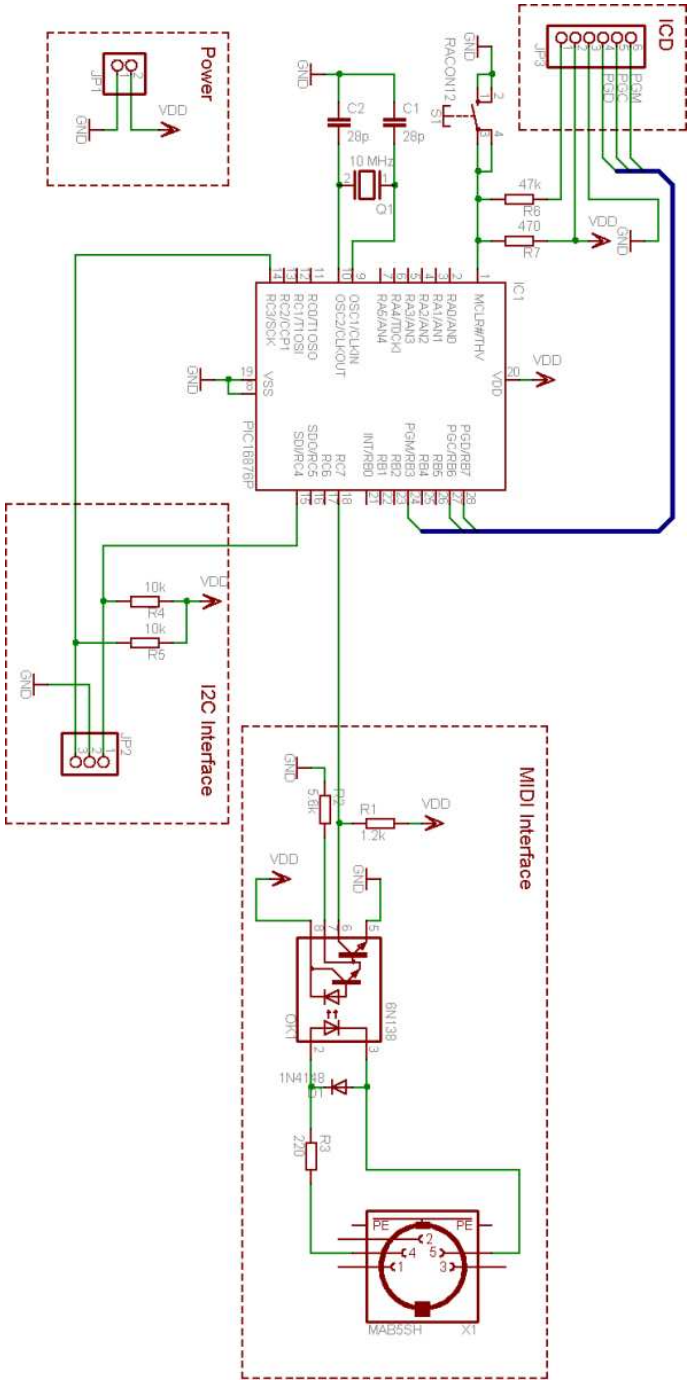


Figure 9: MIDI-Interface schematic

# Part VII
# Chorus

Sagar Jagtap, Chakradhar Kulakarni

**Abstract**

Chorus is a time delay algorithm used to 'thicken' sounds. We duplicate the effect that occurs when many musicians play the same instrument and same music part simultaneously. Musicians are usually synchronized with one another, but there are always slight differences in timing, volume and pitch between two instruments. Such chorus effect can be re-created digitally by adding time varying delayed result together with the input signal.

# 1 Introduction

In this report a brief theoretical background on chorus effect is given followed by the strategy for implementing it on DSP-card from Texas instruments. The theory is given in chapter 2 and implementation is described in chapter 3. A discussion on the difficulties encountered during the project is given in chapter 4.

## 2 Signal processing behind the chorus

Chorus is generated by adding the delayed sample and the current sample. But in order to get some real effect we have to vary few parameters of delayed samples namely

a Delay value.

b Gain of the delayed sample.

b Frequency of delay variation.

In order to get above variable parameter we think about periodic mathematical function like Sine wave, Cosine wave, Triangular wave or Square wave. But for real effect we need smooth variation of above parameters. So we choose to select the Sine wave. One more reason is that Code Composer Studio supports Sine function so it is easy to implement in C.

### 2.1 Delay value variation

We are interested in 20 ms to 40 ms delay variation, so we convert it in terms of delayed sample. In this case sampling frequency is 32 KHz.

for 20 ms delay = 640 delayed sample.

for 40 ms delay = 1280 delayed sample.

We take constant delay of 30 ms (960 delayed samples) and add that with variable delay of 10 ms (320 delayed samples) which is a sine function of amplitude 320.

$$delay = constantdelay + variabledelay * sin(2 * 3.14 * freq/Fs * t + phase) \quad (1)$$

Here 't' is periodic sample time of sine wave. In figure 1 we can see the graphical representation of variable delay.

### 2.2 Gain of the delayed sample

Gain of the delayed sample can be varied by external interface of MATLAB. But the gain of the delayed value should be less than one.

### 2.3 Frequency of variation of delay

Frequency of sine wave indicates frequency of variation of delay. For chorus effect we need very very small frequency of delay variation. This parameter can be varied by external interface of MATLAB.

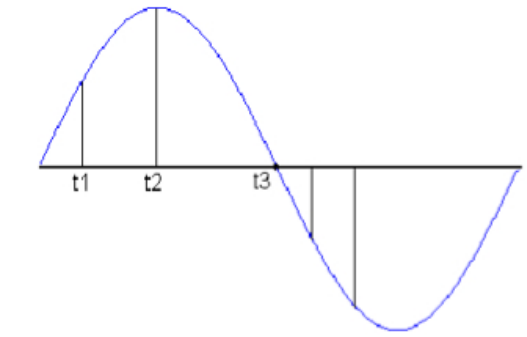


Figure 1: Sinusoidal variation in Delay

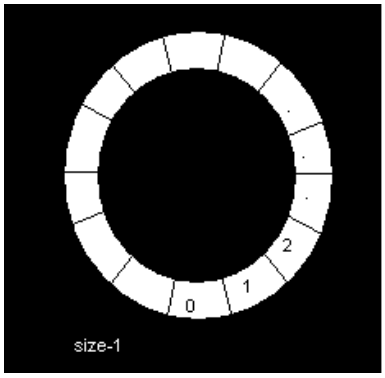


Figure 2: Circular buffer

# 3 Implementation in C code

Development environment is Code Composer Studio (CCS) for DSP-card C6713 by Texas Instrument Compiler of Code Composer Studio supports various C, C++ library function.

## 3.1 Algorithm

1 Split the samples into left and right channel.

2 Storing the samples from respective channel to particular circular buffer.

3 Call the various functions for data processing.

4 Send the processed data to output port.

### 3.1.1 Split the samples into left and right channel

In order to split the input stream on left and right, we pass the even number data to left channel and odd number data to right channel. The input data stream is a combination of left and right channel.

### 3.1.2 Storing the samples from respective channel to particular circular buffer

1 Initially both address pointers are at $Size - 1$ location of the buffer.

2 For each sample the pointer of respective buffer is decremented to next location after storing the data.

3 If current location is minus one location, then it moves to $Size - 1$ location.

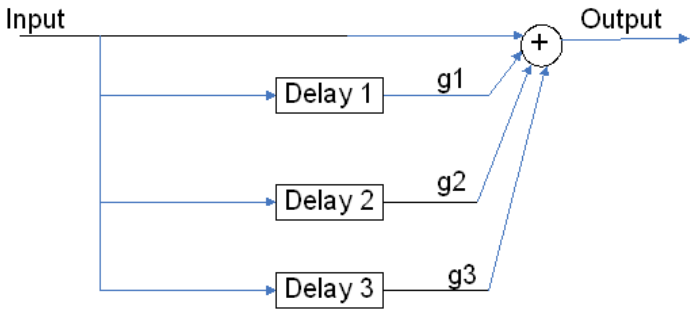


Figure 3: Echo-chorus block diagram

In short this is first in first out (update old data by overwriting) storage buffer as shown in figure 2.

### 3.1.3 Functions for data processing

We implement the following functions for data processing.

1 float sinewave(int t, int amplitude, short phase, short freq)

2 float echochorus(short *xvec, int xveclength)

3 float allpassfilter(short *vec, int inputIndexx, int xxveclength)

Call the sinwave function for each delay line in echochorus function. The block diagram of echochorus function and allpassfilter is shown in figure 3 and figure 4 respectively.

In figure 3 and figure 4 the gain value is set as constant value less than one.

### 3.1.4 Memory Handling

To be able to have the delay large enough the sample vector has to be of at least the length of the delay. With higher sample rate the vector size grows rapidly and more memory is required. Since the cache memory is not enough, the vector has to be allocated on external memory. It is slower than cache but it is fast enough in this application. To be able to use external memory, it has to be declared with the name given in memory section of DSP/BIOS-Config. In this case it is called EXTERNALHEAP (EXTERNALHEAP in DSP/BIOS-Config). The wanted variable also has to be declared. This is done by the following two rows that are top of .c-file amongst the other declaration.

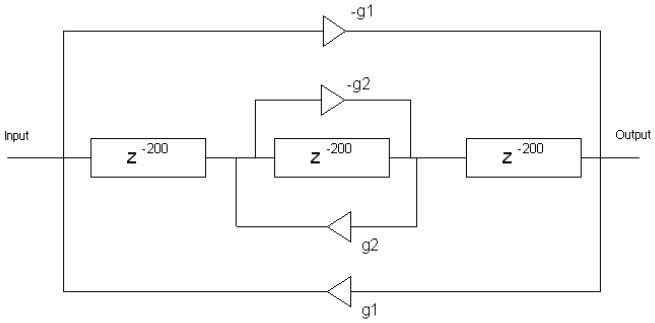


Figure 4: Allpass-filter block diagram

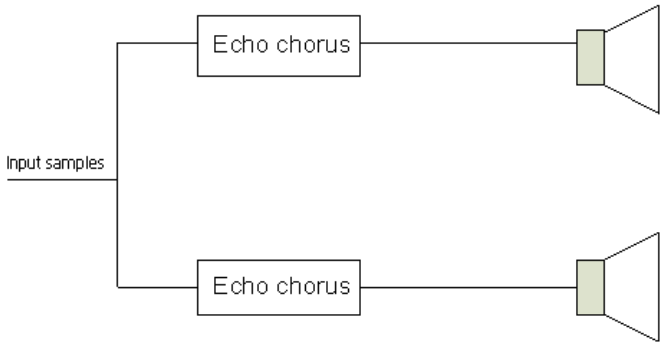


Figure 5: Echo-chorus configuration

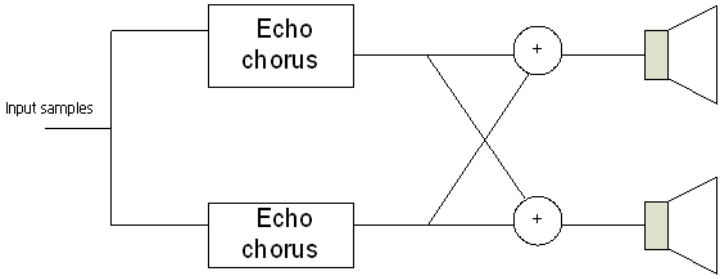


Figure 6: Echo-chorus configuration with cross gain

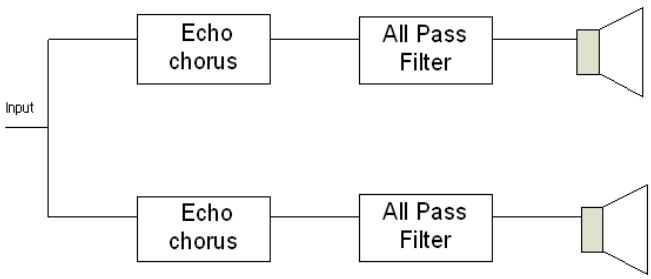


Figure 7: Cascading of echo-chorus and allpass-filter

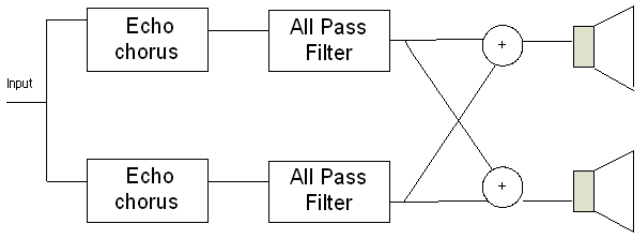


Figure 8: Cascading of echo-chorus and allpass-filter function with cross gain

```
 xvec = extern Int EXTERNALHEAP;
{xvec = short*xvec}
```

In main() the vector has to be allocated size (4000)

```
xvec = (short*)MEM_alloc(EXTERNALHEAP, 4000*sizeof(short), 0);
```

Now this *xvec* can be used as a usual variable declared in the cache, can be used in the program as a usual variable declared in the cache.

## 4 Conclusion

In real time signal processing we have to compromise with memory. But still there is some methods we can reduce memory requirement. We can go for database management and update the database as soon as we use the value from database. At the same time we have to think about system delay like memory access time. Digital signal processing is just arrangement of standard functions like FIR filter, IIR filter, delays, FFT so we have to write code with help of data structure so that we can flexibly change functionality with less effort. One important point about data type of variable which may cause malfunctioning as well as unwanted memory usage.

## References

[1] S.K. Mitra, *Digital signal processing, A computer-based approach*, McGraw-Hill, 2001.

[2] Roads,Curtis (1996). *The computer music tutorial*, The MIT press.