Synthesis of speech with a DSP

Karin Dammer Rebecka Erntell Andreas Fred Ojala

March 16, 2016

1 Introduction

In this project a speech synthesis algorithm was created on a DSP. To do this a method with linear prediction and Levinson-Durbin recursion to find filter coefficients for an IIR filter was used. Then a pulse train and/or Gaussian noise were processed through the filter to recreate an approximation of the original speech signal. A prototype was made in MATLAB and then translated component by component to C code which is what the DSP uses. To find a sufficient number of filter coefficients different filter lengths were evaluated. In the final version of the filter 12 filter coefficients were used.

2 Theory

To synthesize speech, a known speech signal is processed through an FIR linear predictive error filter which gives an estimated error signal. The error signal will in the vowels contain a pulse train with the same pitch as the speech. The frequency of the pulse train is then extracted from the error signal and a similar pulse train is created and fed into an inverse, all pole, IIR filter. The result is a synthetic version of the speech signal. This method is used to synthesize speech and in e.g. vocoders. The inverse filter can also be fed with e.g. a Gaussian noise signal or a pulse train with constant frequency to obtain different effects in the recreated speech. The Gaussian noise will give a good reproduction of consonants and a pulse train, which has a frequency, will be better for vowels. Block diagram of filter

2.1 Linear predictive coding

A linear prediction filter uses the previous samples from a signal to estimate the next sample. In a linear prediction error filter the estimated signal is then compared to the real signal, and subtracted from it, and en error signal is obtained. To compute the filter coefficients the Levinson-Durbin algorithm can be used.

2.2 Levinson-Durbin

The Levinson-Durbin algorithm is an algorithm that is used to solve a linear equation system involving a Toepliz matrix. A Toepliz matrix is a diagonally

symmetrical matrix with a constant value on the dialonal elements. The prediction filter coefficients are the solution $\boldsymbol{w}_f$ in the Wiener-Hopf equation

$$\boldsymbol{R}\boldsymbol{w}_f = \boldsymbol{p} \tag{1}$$

$$\boldsymbol{R} = E(\boldsymbol{u}(n-1)\boldsymbol{u}^H(n-1)) \tag{2}$$

$$\boldsymbol{p} = E(\boldsymbol{u}(n-1)u^*(n)) = \boldsymbol{r} \tag{3}$$

where $\boldsymbol{R}$ is the autocorrelation matrix and $\boldsymbol{p}$ is the covariance vector. The coefficients of the prediction error filter is then the solution to the augmented Wiener-Hopf equation

$$\begin{bmatrix} r(0) & \boldsymbol{r}^H \\ \boldsymbol{r} & \boldsymbol{R} \end{bmatrix} \begin{bmatrix} 1 \\ -\boldsymbol{w}_f \end{bmatrix} = \begin{bmatrix} P_M \\ \boldsymbol{0} \end{bmatrix} \tag{4}$$

where P_M is the prediction error power. The filter coefficients for a M:th order filter $\boldsymbol{a}_M$ is thus $[1, -\boldsymbol{w}_f]$. The algorithm uses the filter coefficients from the previous filter (of (M-1):th order) to determine the filter coefficients for this filter and can thereby be seen as a recursive function. The filter coefficients can be evaluated in the following way.

1. Initialize $\quad \Delta_0 = r(1), \quad P_0 = r(0)$ (5)

2. For $i = 1, .., m \quad \kappa_m = -\dfrac{\Delta_{m-1}}{P_{m-1}}$ (6)

$$a_{m,0} = 1 \tag{7}$$

$$a_{m,k} = \sum_{k=1}^{m} a_{m-1,k} + \kappa_m a_{m-1,m-k} \tag{8}$$

$$\Delta_m = r(m+1) + \sum_{k=1}^{m} a_{m,k} r(m+1-k) \tag{9}$$

$$P_m = P_{m-1}(1 - |\kappa_m^|2) \tag{10}$$

2.3 Signal processor

The DPS used was Analog Devices' ADSP-21262 (fig 1), which is a 32-bit programmable digital signal processor. It runs a Sharc processor core on a clock frequency of 200 MHz and has 2 Mbit of SRAM and 4 Mbit of ROM integrated (Analog Devices 2012). The DSP was put on a breakout card and mounted in a case with four buttons and audio input and output connectors.

Figure 1: ADSP-21262 (Analog Devices 2016b).

The software was put together in Visual DSP++, which is Analog Devices' integrated development environment. It provides a user interface for code editing, project management, compilation, debugging and DSP uploading. Applications can be run either in simulation and emulation modes or programmed to the DSP flash memory. C, C++ and assembler code are supported (Analog Devices 2016a).

3 The MATLAB prototype

To start solving the problem we made a prototype program in MATLAB. To determine the structure of the program MATLABs ready-made functions were used and a simple first prototype were made. In the first draft the a pre-recorded audio file were used as input and the error output from the FIR filter were used as input in the reverse filter to be able to see that the original signal was recreated.

To make the problem solvable the input has to be divided into sections with 320 samples each and put in a matrix where the sections form the columns of the matrix. By dividing the data in small sections way the data in one of the sections can be said to be stationary stochastic which makes it possible to calculate the autocorrelation and covariace. After calculating the autocorrelation the filter coeficcients of the error prediction filter were calculated by the Levinson-Durbin algorithm. The filter coefficients were then used to filter the data sequence through the FIR filter and the error output was analyzed. Three different inputs were used through the inverse filter, a pulse train with constant frequency, Gaussian noise and a pulse train with the same frequency as the input.

4 The DSP application

After verification and validation of the Matlab prototype, the algorithms were translated to C code in order to be run on the DSP.

The DSP application consisted of four separate units. The first one, framework.c, contained preprovided help functions for basic DSP operations and configurations. Its public functions are presented in table 1.

void dsp_init()	Initializes framework.
void dsp_start()	Starts serial codec ports.
void dsp_stop()	Stops serial codec ports.
sample_t* dsp_get_audio()	Returns current audio block.
unsigned int dsp_get_keys()	Returns a bitmask for buttons pressed.

Table 1: Public functions in framework.c.

All algorithms needed to recreate the Matlab prototype were gathered in levinsondurbin.c (table 2). In order to achieve smooth transitions between the processed audio blocks, filter states were saved between invocations. The states were represented as static variables and thus not included in the function calls, as would have been the case in Matlab. For the sake of practice, all mathematical functions, such as filtering, matrix operations and convolution, were made from scratch instead of being imported.

void autocorrelation (float* x, float* Rxx)	Calculates autocorrelation.
float levinson_durbin (float* Rxx, float* coef)	Calc filter coefs, returns pred pwr.
void fir_filter (float* coef, float* x, float* y)	Performs FIR filtering.
void iir_filter (float* coef, float* x, float* y)	Performs IIR filtering.

Table 2: Public functions in levinsondurbin.c.

Different kinds of input for the inverse filtering were generated in input.c (table 3). White noise was produced according to the central limit theorem, which states that the mean of a sufficiently large number of independent random numbers will be approximately normally distributed. The pulse train algorithms were similar to the ones in the Matlab prototype.

void white_noise (int degree, float pp, float* pulses)	Creates and adds white noise.
void static_pulses (float pp, float* pulses)	Creates a const pitch pulse tr.
void dyn_pulses (float pp, float* Ree, float* pulses)	Creates a variant pitch pulse tr.

Table 3: Public functions in input.c.

Main.c, finally, was the executable file in which audio data was read in blocks, filtered and output. It also handled the user interface buttons. Functions are listed in table 4.

void process(int sig)	Audio I/O and filtering at timer interrupts.
static void keyboard(int sig)	Reads buttons at timer interrupts.
void main()	Initiates DSP and interrupts, then idle loops.

Table 4: Functions in main.c.

When the application started, the main function was called. The DSP was initialized and timed interrupts for reading button values and processing audio were registered. Then an incessant idle loop started. The user would chose between generating unprocessed sound (first button), a static pulse train (second button) or a dynamic pulse train (third button). White noise could be also added (fourth button) before inverse filtering, generating six different operating modes in total.

```
if (input) {
    autocorrelation(x, Rxx);
    pp = levinson_durbin(Rxx, coef);    // pp = prediction power
    switch(input) {
        case ST_PULSES:
            static_pulses(pp, p);
            break;
        case DYN_PULSES:
            fir_filter(coef, x, e);
            autocorrelation(e, Rxx);
            dynamic_pulses(pp, Rxx, p);
            break;
        default:
            // case WH_NOISE
            noise_on = 1;
            for (n = 0; n < DSP_BLOCK_SIZE; ++n)
                p[n] = 0.0;
            break;
    }
    if (noise_on)
        white_noise(6, pp, p);
    iir_filter(coef, p, y);
} else {
    // Pass-through.
    noise_on = 0;
    memcpy(y, x, sizeof(y));
}

// Copy output buffer to left and right audio channels.
for(n=0; n<DSP_BLOCK_SIZE; ++n) {
    audioout[n].left = y[n];
    audioout[n].right = y[n];
}
```

Figure 2: Audio processing in the process() function of main.c.

The audio processing logic is depicted in figure 2. At processing interrupts, a block of audio samples was fetched. For unprocessed sound, input data was sent directly to output. Otherwise, autocorrelation of the samples was performed and the Levinson-Durbin algorithm was used to calculate prediction error filter coefficients from the autocorrelation vector.

In the static pulse train mode, static_pulses() was called. In the dynamic pulse train mode, input data was FIR filtered with the Wiener filter coefficients. The autocorrelation of the resulting error signal was used for calculating the dynamic pulse train with dynamic_pulses(). At this stage, white noise could be added to the pulse trains, or to an empty vector in order to inverse filter pure noise. The resulting vector was IIR filtered with the Wiener filter coefficients and output.

5 Result

The original speech signal consisted of a person saying the words ”she had your dark suit and greasy wash water all year”. The plot of the speech signal can be seen in figure 3. The amount of coefficients from levinsondurbin were chosen to be 12.

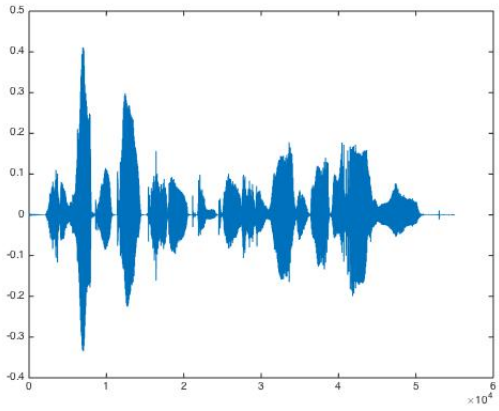

Figure 3: Original input

The filtered error had a very strong correlation somewhere between 50-100 samples, but the most common occurred at around 80 samples. Using this fact, the correlation of the filtered error in the DSP was checked for the highest correlation between the 50:th and 100:th sample. The resulting output from the matlab testing was a robotic voice that had some of the characteristics of the speaker. On the DSP however, some kind of disturbance occurred when using the dynamic pulse train in contrast to that of the matlab testing.

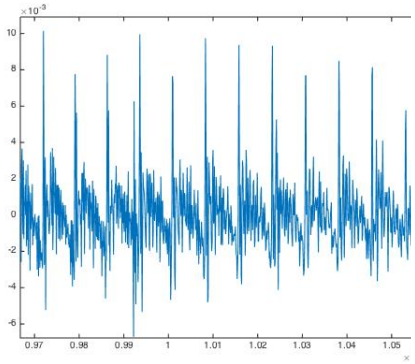

Figure 4: Filtered error.

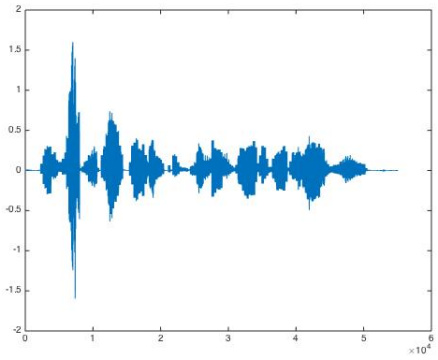

Figure 5: Dynamic pulse train.

Since the correlation of the filtered error were highest between the 50th and 100th samples, the static pulse train were chosen to be placed at every 80:th sample. Since each pulse were located at every 80th sample, the static pulse train didn't need to remember the previous location of pulses due to the fact that 80 is a multiple of the block size, 320 samples. The resulting synthesized speech from the static pulse train were also a typical robotic voice. However, in contrast to the dynamic pulse train case this resulted in a monotonous robotic voice.

When using random noise with variance from the prediction error as the

input to the inverse filtering the resulting output captured more of the noisier structure of the voice. This came at the cost of the synthesized speech becoming much noisier and the characteristics of the speaker completely disappearing.

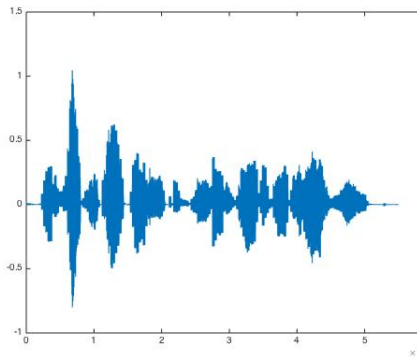

Figure 6: Static pulse train, output.

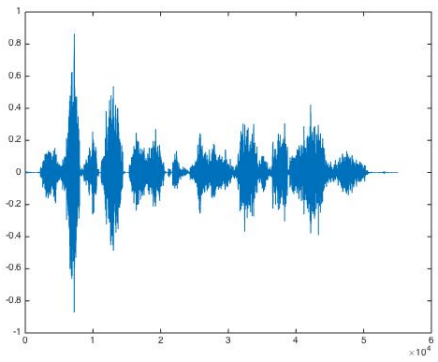

Figure 7: Noise, output.

On the digital signal processor, the combination of dynamic pulse train with noise and a static pulse train with white noise were also implemented. Since both the pulse train and the white noise had the same amplitude, the amplitude of the noise was scaled down with a factor 2.

6 Conclusion

The implementation of the different algorithms went smoothly, both in Matlab and on the DSP, with only some minor tweaking to improve sound quality. When translating from Matlab to C, array indexing needed to be changed which needed to be given some extra thought when arrays were being processed in loops that were nestled in more complex ways. Another tricky part was to match the in- and output formats of the algorithms so that all parts were represented on the same form and fitted together.

We chose to use 12 filter coefficients from the Levinson-Durbin algorithm, as these would suffice. Theoretically, if we would have chosen a higher amount of coefficients the resulting output would be of a higher quality. However, in our testing we could not hear much difference with a higher amount of coefficients.

As a whole, the result was almost as intended. The monotonous robotic voice from the static pulse train worked as expected. The noisy output from the white noise was also expected. The dynamic pulse train worked almost as intended. The error that needs some tweaking is a slight disturbance noise from the dynamic pulse train. We were not able to determine where this came from or how to fix it. We tried different methods, increasing the amount of filter coefficients and increased the length of the filtered error, but to no avail. It would also be interesting to model the white noise input to model more of the characteristics of the voice. However, we had no knowledge how to do this so we decided to not include this in the project.

References

Analog Devices 2012. SHARC Embedded Processor ADSP212-61/ADSP-21262ADSP-21266 Datasheet. URL www.analog.com/media/en/technical-documentation/data-sheets/ADSP-21261_21262_21266.pdf. 2016-03-03.

Analog Devices 2016a. VisualDSP++ 5.1. URL www.analog.com/en/design-center/processors-and-dsp/evaluation-and-development-software/vdsp-bf-sh-ts.html_dsp-overview. 2016-03-03.

Analog Devices 2016b. ADSP-21262. URL http://www.analog.com/en/products/processors-dsp/sharc/adsp-21262.html#product-overview. 2016-03-03.