Speech Synthesis

Joakim Ysing lan11jys@student.lu.se
Ricardo Gomez soc14rgo@student.lu.se
Kevin Froimson ksf5fk@virginia.edu

March 10, 2015

Contents

1 Introduction **3**

2 Implementation **3**
2.1 Proof of concept 3
2.1.1 Block Processing 3
2.1.2 Speech signal 4
2.1.3 LPC 5
2.1.4 Speech Synthesis (Excitation Generation) 5
2.1.5 Speech Synthesis (Output signal Generation) 6
2.2 DSP Implementation 7

3 Problems **8**
3.1 Problems in Matlab 8
3.2 DSP problems 9

4 Results **9**

5 Conclusions **9**

A Matlab code **12**
A.1 SpeechSynth.m 12
A.2 processLev.m 12

B C code **14**
B.1 autoRegressiveFilter.c 14
B.2 ETIN80.c 14
B.3 GWN.c 17
B.4 levinsonDurbin.c 17

1 Introduction

The purpose of the project was to research and implement speech synthesizing on a SHARC ADSP-21262 digital signal processing unit (DSP). Since the DSP was an unfamiliar platform and containing a lot of hardware limitations the project was split into two main parts.

The first part consisted of implementing the speech synthesizing in Matlab. Working with Matlab compared to the DSP presented a lot of simplifications. Code written in Matlab can contain non-causal or anti-causal filtering. Running the code on a computer gives access to a seemingly unlimited amount of memory and processing resources while the DSP has very limited amount of memory and processing resources. We can gain access to the entire signal at all times in Matlab where as the DSP must be run with realtime demands. The Matlab implementation was done mostly as a proof of concept to make sure that the method was correct before working on the DSP.

The second part of the project was the actual implementation on the DSP. Since we already had the method written in Matlab this step was consisted mainly of converting the Matlab code into c code with a few additional problems.

2 Implementation

2.1 Proof of concept

In this Project various goals were pursued. The main one was speech synthesis. This basically means being able to produce speech signals from an computer generated excitation signal. To address this problem, a first study of the speech signal, together with the study of the different signal processing blocks used is presented.

2.1.1 Block Processing

Block processing is a kind of processing that is widely used in DSP implementations. Whereas in sample processing, one sample is processed at a time (as it is the case of an FIR filter with a continuous input data stream), in block processing as seen in figure 1,

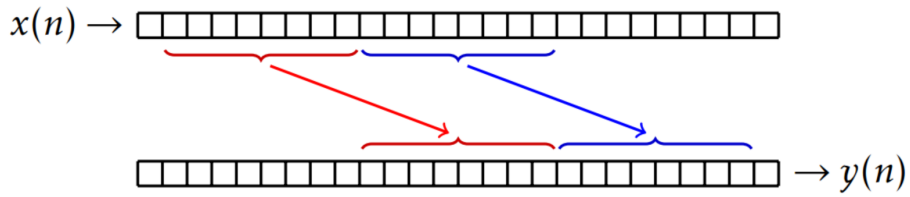

Figure 1: The processing of several signals at a time $X(n)$, referred to as blocks, lead to an increased delay in the output $Y(n)$ as can be seen in the figure.

N samples are processed at a time and thus, the system wait to buffer all of them before it starts processing them.

There are many reasons of why this kind of processing is done this way. In this example, the main reason is the characteristics of the speech signal. As

will be described in the next sections, some parameters of the speech signal are going to be estimated, such as pitch, formants, etc. It is obvious that, during a normal speech, all this characteristics change over time, and thus they can't be estimated just once for the whole speech. Then, it is neccesary to look for some time lapse that guarantees us (with certain precision) that, during that time, most of this characteristics are going to remain constant.

In the example of speech, it is widely used the 20ms time lapse. However, because of the fact that this project deals with digital signals, this time has to be converted to samples. In this conversion, the sampling frequency takes place. Depending on that sampling frequency, the amount of samples buffered in each block will change. It is often the case that, due to a high sampling frequency, it is desirable to perform some kind of downsampling (as in this case) to reduce the processing load to the DSP.

However, apart from the constant-parameters time lapse, there are other variables that influence the optimal block size. These can be included in the trade-off between time resolution and real-time processing. Ideally, it would be desirable to have the block as small as possible (with the upper bound that gives us enough periods to estimate the pitch). The reason for that is because it is desired to be able to synthesize every single millisecond of voice with the particular characteristics of each millisecond. Having a larger block will lead to a loss in terms of resolution, and thus some fast variations will disappear. Also, because of the fact that the 20ms time lapse is not synchronized with when the speech signal changes, it is usual that one block contains two different kinds of speech signal, each one with different pitch, etc.

Nevertheless, when implementing the algorithm in a DSP, the processing time has to be taken into account. As it will be seen, the main time consuming task is the Levinson-Durbin algorithm. The Levinson-Durbin algorithm is calculated for every single block. Thus, the smaller the block, the more often Levinson-Durbin is calculated. In this project, real-time processing is intended. Thus, there is a bound on how fast Levinson-Durbin algorithm can be performed. Too small blocks will lead to a situation where there is not enough time to perform all the processing algorithms before the next signal block is ready to be processed, making impossible the real-time implementation of the desired algorithm.

Lastly, there is also a relation between the perceived delay at the output of the DSP and the block size. The reason for that is because at first, the system has to wait for an entire block to be stored in memory to be processed. Thus, that amount of time has to be added to the delay related to the signal processing at the DSP. However, in this particular project, a small increment on the output delay is not really a problem, and thus this constraint hasn't been taken into account.

2.1.2 Speech signal

The speech signal is an specific kind of signal produced naturally by humans when speaking. From a signal processing point of view, speech signal can be regarded as a kind of signal which is composed by spectral resonances (also called formands), some periodic excitation (voicing), some noise, transients and amplitude modulation.

Various methods have been proposed to model the speech signal generation. However, the one used in this project is the source filter model.

This model basically separates the speech synthesis into two main components: the source, which is basically a pulse train with a certain pitch and with or without fritation noise; and the filter, which includes the resonance terms. Thus, to sum up, it has been seen that speech can be modelled as the output of a certain system (that will be approximated as LTI) excited by pulses and/or noise. An example of a typical block of speech can be seen in figure 2.

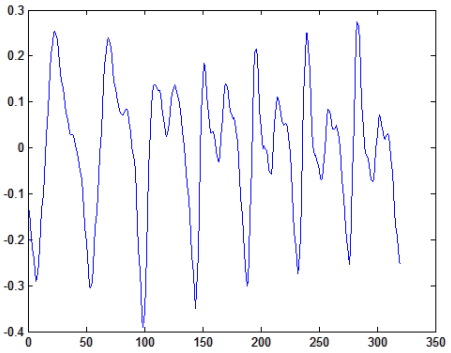

Figure 2: A typical block of a normal speech signal. Horizontal axis is the discrete signal number on the block and the vertical axis is the amplitude.

2.1.3 LPC

Once the main idea and modeling of the speech signal have been presented, it is time to be able to estimate the different parameters that characterize the speech generation model to be able to effectively synthesize speech signal, as intended in this project. For this purpose, the Linear Predictive Coding algorithm is used[3]. The reason of why this kind of method is used to address this kind of problem is that it efficiently estimates the speech parameters as they have been modeled before: as a linear system excited by some glottal pulses and noise. Thus, LPC works directly with the expression of the source-filter model it has been presented.

The specific algorithm used to solve this LPC is the Levison-Durbin algorithm. The basic idea is that, after completing the processing with the Levinson-Durbin algorithm, the linear prediction filter is obtained, which represents the vocal tract (all the correlated terms) and some residual signal, which represents the uncorrelated signal (it is obtained as the difference between the real recorded signal and the predicted signal, which just contain correlated terms).

2.1.4 Speech Synthesis (Excitation Generation)

At this point estimated all the parameters of the source-filter model of the speech voice have been estimated: a prediction filter which takes into account all the correlated parameters of the speech signal has been obtained, and the residual noise after extracting the recorded signal to the predicted signal has been calculated and an example of a block can be seen in figure 3.

However, to synthesize speech, the residual noise is not used as the exciting signal: On the contrary, It is necessary to generate a computer-based excitation which has some of the parameters of the residual excitation signal to generate the sound. In this project, the first parameter that was pursued was the pitch of the signal.

When the general waveform of a excitation signal is observed, it can be seen that it is formed by some pulses that are periodic (considering small amount of

time). The inverse of the period of this pulses represents the pitch of the speech signal. Thus, to generate an artificial excitation, it is neccesary to estimate the pitch of the residual signal. There are many methods to do that. In this project, the autocorrelation of the signal was used. By autocorrelating the signal the similarity between the signal and a delayed version of the same signal can be obtained. Thus, if the signal is periodic (and has some periodic components such as pulses), some maximum values in the autocorrelation will be seen as the delay value of the second signal approximates to the period of the pulse train. After that, a simple maximum-value search is enough to calculate the period of the pulses, and thus the pitch.

Then it was time to generate the excitation signal. As described, it will be basically a train of pulses with a period equal to the inverse of the pitch calculated. A typical block of the excitation signal can be seen in figure 4. However, the height of the pulses (the amplitude of them) is a value that has to be estimated as well. This value will have an impact on the energy of the excitation signal, and thus will have an impact in the energy of the synthesize signal. In order to generate a speech signal with the same energy as the speech recorded, the value that comes directly when using Levinson-Durbin algorithm was used for the pulse generation.

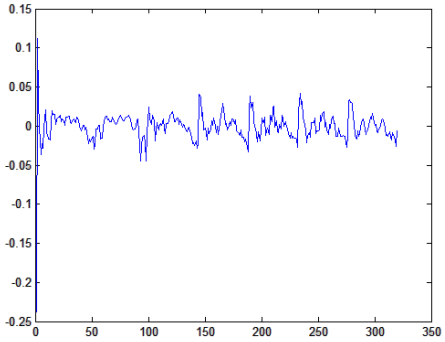

Figure 3: Prediction error of the original signal for one block. Horizontal axis is the discrete signal number on the block and the vertical axis is the amplitude.

Last stage of the excitation generation was the generation of some form of noise. As described in the first section of this theoretical analysis, the excitation signal is not only composed by a train of pulses, but also by some form on noise, representing fricative noise. A speech signal synthesized without this kind of noise will sound too robotic and artificial. In this project, some Gaussian distributed noise was generated, and later on added it to the pulse train to model this behaviour [2].

2.1.5 Speech Synthesis (Output signal Generation)

Once the excitation signal has been generated, it was time to include all the components of the formants that have been estimated with the linear prediction coding algorithm. As has been explained, all that information is stored in a form of a FIR filter, that basically predicts the next sample of a signal. However, it is neccesary to perform the inverse operation: for a given excitation signal, generate the speech signal is desired. In order to perform that, the excitation signal is going to be filtered with the inverse filter that comes out from the Levinson-Durbin algorithm. This means converting the FIR filter into an IIR filter, and then directly filtering the pulse train with noise with it in order to generate the final speech signal.

2.2 DSP Implementation

Since the DSP runs with limitations not found in a regular computer a lot of time was spend trying to find out how to program the DSP and what methods available for it. The main difference, apart from using different languages, was that the DSP didn't have access to a pre-implemented version of the Levinson-Durbin algorithm.

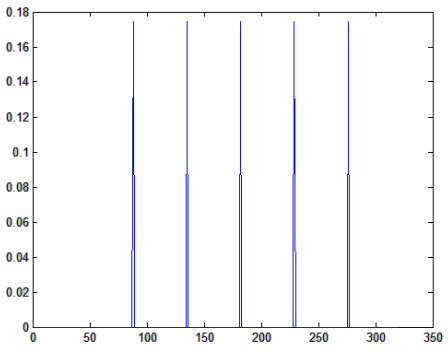

Figure 4: Synthesised error for one block of normal speech. Horizontal axis is the discrete signal number on the block and the vertical axis is the amplitude.

The Implementation of the speech synthesize on the DSP was done with the help of a program called VisualDSP++. To avoid having to turn hardware on and off all the time a simulation for the DSP was set up in VisualDSP++ where we had the additional possibilities of printing out optional data in the terminal for debugging. Given that the exact same method was to be implemented on the DSP as in Matlab made the transition from Matlab code to C code quite easy. There were however a few differences apart from the non-existing Levinson-Durbin method that made the implementation lite less of a direct code conversion. The largest of these was that there was no pre implemented version of an IIR-filter that suited the needs of the project. This however was a relatively quick thing to implement resulting on only a minor delay of the schedule. The C code for the IIR-filter used in the finished program can be seen in appendix B.

Research on the Levinson-Durbin algorithm showed that it solves a system of equations called the Yule-Walker AR equations. This is basically a recursive matrix equation of a toeplitz matrix to find the next step in a prediction filter[1]. A block schematic can be of a prediction filter returned by the Levinson-Durbin method can bee seen in figur 5.

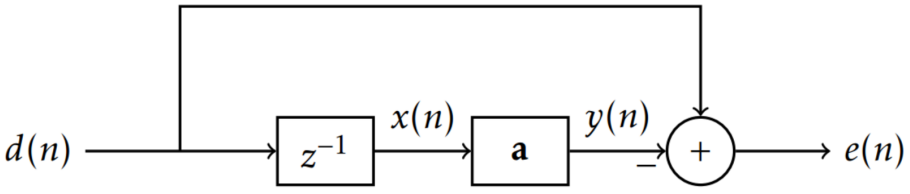

Figure 5: The 1-step forward prediction filter which is returned implemented using the Levinson-Durbin algorithm. a represents the linear predictor coefficients.

The implementation on the DSP uses block of 320 signals at a time which are aligned as a vector. This decreased the problem of solving a recursive equation system of matrices into solving it only for vectors. This meant little differ-

ence in the time it took to implement but requires less calculation and thus less processing time on the DSP. The pseudo code for the projects take on the Levinson-Durbin algorithm can be seen below, with R representing the autocorrelation coefficients and A representing the Levinson coefficients returned from the algorithm. For a full implementation of the algorithm see appendix B.

- Initialize predicted error as $E = R[0]$.
- For i from 0 to 15
  - initialize $temp[0] = R[i+1]$.
  - initialize $sum = R[i+1]$.
  - calculate $sum = \frac{1}{E} \cdot \sum_{j=1}^{i} A[j] + R[i-j+1]$.
  - calculate $temp[j] = A[j] - \frac{sum}{E} \cdot A[i-j+1]$.
  - initialize $temp[i+1] = \frac{sum}{E}$
  - update $E = E \cdot (1 - (\frac{sum}{E})^2)$.
  - update $A = temp$.

3 Problems

3.1 Problems in Matlab

Fortunately, Matlab implementation was simple compared to a direct DSP implementation. The first problem was related to the used function for the LPC algorithm. Instead of using directly the Levinson Durvin algorithm, with returns the coefficients of the whole system (which is composed by the LPC filter plus the subtraction), the LPC function was used, which just returns the LPC filter. Thus, it was neccesary to manually perform the prediction filtering and subtraction from the recorded signal. After having some problems with this approach, and following the teacher's instructions, Levinson as the Matlab function was used.

The second problem encountered was during the excitation generation. At first, a simple approach was implemented that set, for each block, all the positions of the pulses starting from the sample "one". This caused a lot of distortion, due to the fact that between blocks there was no consistency, and thus it was common to have two pulses (the last one of the block k and the first one of the block k+1) too close to each other, and then generating some high frequency distortion terms. It was easily solved by taking into account the position of the last pulse on the generation of the pulse train of the next block, and then including a certain offset to avoid this problem.

The last problem encountered was a remaining beeping and distortion due to the rapid and abrupt change in the frequency and shape of the excitation signal between blocks. To address this problem, the way the buffering was taking place was changed. Instead of taking non-overlapping blocks, which produced a lot of distortion, overlapping blocks (which is easily achieved by the Matlab's buffer function) were taken. After that, when combining all the results from each block, Hamming windowing was used. However, the result was not as good as expected, so it has been finally removed it as it introduces an important amount of complexity for a not really big improvement on the sound's quality.

3.2 DSP problems

Most of the limitations on the DSP turned out not to be a problem during the project. However an important part when debugging and comparing the step by step results with the ones achieved in Matlab was that the DSP works with $32 - bit$ sized variables whereas Matlab works with $64 - bit$ sized. This in turn resulted in minor differences in the values, something that in it self is of little importance but can be confusing when searching for a small actual error that results in a defective implementation in the end.

As for the Levinson-Durbin algorithm, which was the main part to implement on the DSP, this took quite a lot of time to implement. The most difficult part of the algorithm was to understand what was needed from it. Several implementations were tried until the theory of the algorithm was fully understood and an reduced working implementation was made.

The second largest problem was the IIR-filter. There where three alternatives for pre implemented versions of such a filter for the DSP. All of these required input that was not available or simply gave a result different from the what was needed. Full understanding of these functions was never achieved but instead an autoregressive filter was implemented to suit the need of the project. The full implementation of the filter can be seen in appendix B.

4 Results

Then end result of the project is two output alternatives on the DSP. Both alternatives results in a relatively clear speech signal to the output.

First one produces an output signal constructed from filtering the synthesized error signal back through a autoregressive filter. This produces a signal where it is no problem to hear what is being said when talking into a microphone. There are however smaller breaks in between the actual sounds leading to a sound that that can be compared to a monotonic stuttering or a more robotic voice.

The second alternative was created to counter the problem with the first one. With the introduction of additive white Gaussian Noise that was added to the synthesized error signal the resulting signal became much more fluent. There are however problems with the addition of noise. The signal get noisy and it gets harder to clearly hear what is being said when talking into a microphone. That being said, it produces a signal that is less monotonic and where the breaks in between the signals are evened out resulting in a sound that is much more comfortable to listen to in the long run.

5 Conclusions

Even though the project seemed relatively easy at the start there has been at least one problem with every step in the process. Not only from working with an unknown environment such as the DSP but even in the beginning when working on the proof of concept using the familiar Matlab language there has been frequent problems in the process. This has of course resulted in that the Matlab implementation took a lot longer that initially expected. Instead of the two weeks, which was the planed time, Matlab was used right up until the last

week of implementations. This did result in being able to spend less time with the DSP implementation and therefor only resulting in two quite similar speech synthesis.

Although the resulting outputs are a form of speech synthesis it is not what was expected in the beginning of the project. The sound could have been clearer and there could have been "funnier" alternatives to the ones in the result. This would of course require more time to work with the DSP.

References

[1] http://www.emptyloop.com/technotes/a%20tutorial%20on%20linear%20prediction%20and%20levinson-durbin.pdf. Retrieved 2015-01-25

[2] http://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform Retrieved 2015-02-05

[3] http://www.ece.ucsb.edu/Faculty/Rabiner/ece259/digital%20speech%20processing%20course/lectures_n Retrieved 2015-02-20

A Matlab code

A.1 SpeechSynth.m

```
function SpeechSynth
   [x2, Fs] = audioread ('Brienne.wav');
   x2 = decimate(x2, 6);
   [r c] = size(x2);
   x = x2(1:c);
   Fs = Fs/6;
   xb = buffer (x , 320);
   [M , N] = size ( xb );
   yb = zeros (M , N );
   output=[];
   Zi=zeros(1, 15);
   paddO = 1;

   for n = 1: N
      [temp, Zf, paddOf]=processLev ( xb (: , n ), Fs, Zi, paddO);
      Zi=Zf;
      paddO=paddOf;
      output=[output; temp];
   end

   y = yb (:);
   sound(output,Fs);
end
```

A.2 processLev.m

```
function [R, Zf, paddO] = processLev( fx, Fs, Zi, paddI )
   close all;
   [r c] = size(fx);
   Fmin = 80; %minimum frequency we are expecting as voice
   maxLag = ceil((1/Fmin)*Fs); %Maximum Lag expected
   maxLag = 200;
   length = 15; %Length of the Levinson Filter we want to achieve
   autoXbil = xcorr(fx,fx,length); %Autocorrelation for Levinson
       Calculation
   autoX = autoXbil(length+1:end); %Just the second half is important
   [a predErr]= levinson(autoX); %Calculation of the filter
   [e, Z] = filter(a,1,fx, Zi); %Extraction of the error signal
   Zi=Z;

   acor = xcorr(e,e,maxLag); %Autocorrelation for pitch detection
   acor = acor(maxLag+20:maxLag+75); %For vector Matching
   peak = find(acor == max(acor)) + 1; %Find the peak. +1 for consistency
   peak = round((peak+20));
   Fspeech = 1/(1/Fs * peak); %With Lag, calculate
   errorSynt = zeros(r,1);
```

```
   if(Fspeech > 3000),
      paddO = 1;
   else
      errorSynt(paddI:peak:end) = predErr; %Synthesised error with
          energy conservation
      vector1 = find(errorSynt == predErr);
      last = vector1(end);
      paddO = peak-(260 - last) + 1;
   end;

   additiveNoise = randn(r, 1); %Gaussian Noise Generation
   errorSyntNoise = errorSynt*(24/25) + additiveNoise*predErr*(1/25);

   [speechSyntNoise, Z] = filter(1,a,errorSyntNoise, Zi); %Generate
       speech signal with noise (less robotic)
   Zf=Z;

   R=speechSyntNoise;
end
```

B C code

B.1 autoRegressiveFilter.c

```
float *autoRegressiveFilter (float dm out[], float dm in[], float dm
    coeff[], int length, int size, float dm prev[]) {
   int n, i;

   for(n=0;n<length;n++){
      out[n]=in[n];
      for(i=1;i<=n;i++) {
         out[n]=out[n]-(coeff[i]*out[n-i]);
      }
      for(i=n+1; i<length;i++) {
         out[n]=out[n]-(coeff[i]*prev[length-i+n]);
      }
   }

   for(n=length;n<size;n++){
      out[n]=in[n];
      for(i=1;i<length;i++) {
         out[n]=out[n]-(coeff[i]*out[n-i]);
      }
   }
}
```

B.2 ETIN80.c

```
#include <processor_include.h>
#include <sysreg.h>
#include <signal.h>
#include <string.h>
#include <filters.h>
#include <stdio.h>
#include <stdlib.h>
#include <stats.h>

#include <math.h>

#include "framework.h"

static float X[DSP_BLOCK_SIZE];     // temporary input signal
static float Y[DSP_BLOCK_SIZE];     // temporary output signal
static int program, maxLag=201, length=15, paddO, paddI, maxY, maxout;
static float Zi[15];

void process(int sig){
    int n, i;

   srand(time(NULL));
```

```
sample_t *audioin = dsp_get_audio();
sample_t *audioout = dsp_get_audio();

for(n=0; n<DSP_BLOCK_SIZE; ++n) {
    X[n] = audioin[n].left;
}

float R[length+1];
autocorrf(R, X, DSP_BLOCK_SIZE, length+1);

for(i=0;i<sizeof(R);i++){
  R[i]=DSP_BLOCK_SIZE*R[i];
}

float A[length+1], Ep;
for(i=0;i<sizeof(A);i++){
  A[i]=0;
}
levinsonDurbin(A, R, length, &Ep);

float Ar[length+1];
for(i=0;i<sizeof(A);i++){
  Ar[i]=A[length-i];
}

float E[DSP_BLOCK_SIZE], state[length+2];
for(i=0;i<sizeof(state);i++){
  state[i]=0;
}

fir_vec(X, E, Ar, state, DSP_BLOCK_SIZE, length+1);

float acor[maxLag];
autocorrf(acor, E, DSP_BLOCK_SIZE, maxLag);

for(i=0;i<sizeof(acor);i++){
  acor[i]=DSP_BLOCK_SIZE*acor[i];
}

int peak=0;
float max=0.0, temp=0.0;
for(i=19;i<75;i++){
  temp=acor[i];
  if(temp>max){
     max=temp;
     peak=i;
  }
}
peak=peak+2;

float Fspeech=((float)DSP_SAMPLE_RATE/((float)peak+1)),
    errorSynth[sizeof(E)];
int last;
for(i=0;i<sizeof(errorSynth);i++){
```

```
      errorSynth[i]=0;
   }
   if(Fspeech>3000){
      paddO=0;
   }else{
      for(i=paddI;i<sizeof(errorSynth);i=i+peak+1){
         errorSynth[i]=Ep;
         last=i;
      }
      paddO=peak-(260-last)+2;
   }
   paddI=paddO;

    float Y[sizeof(X)];
    if(program==1) {
        autoRegressiveFilter(Y, errorSynth, A, length+1, sizeof(X), Zi);
    } else if(program==2) {
      float result;
      for(i=0;i<sizeof(errorSynth);i++){
         GWN(&result);
         errorSynth[i]=(errorSynth[i]*0.90)+(result*Ep*0.10);
      }

      autoRegressiveFilter(Y, errorSynth, A, length+1, sizeof(X), Zi);
    } else if(program==3){
      autoRegressiveFilter(Y, errorSynth, A, length+1, sizeof(X), Zi);
    } else {
      memcpy(Y, X, sizeof(X));
    }

    // 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];
    }
}

static void keyboard(int sig){
    unsigned int keys = dsp_get_keys();

    if(keys & 1) {
        program = 1;
    } else if(keys & 2) {
        program = 2;
    } else if(keys & 3) {
      program = 3;
    } else if(keys & 4) {
      program = 4;
    }
}

static void timer(int sig){

}
```

```
void main(){
   paddI=0;
   paddO=0;
   maxY=0;
   maxout=0;

    dsp_init();

    interrupt(SIG_SP1, process);
    interrupt(SIG_USR0, keyboard);
    interrupt(SIG_TMZ, timer);
    timer_set(9830400, 9830400);
    timer_on();

    dsp_start();

    for(;;) {
        idle();
    }
}
```

B.3 GWN.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

static float PI=3.1415926536;

float *GWN (float *result) {
   float temp1=0, temp2=0, randMax=0.3, random;
    int p=1;
    while(p>0){
      random=rand()%1600000;
      temp2=(random/randMax);
      if(temp2==0){
         p=1;
      } else {
         p=-1;
      }
    }
    random=rand()%1600000;
    temp1=cosf(2.0*PI*random/randMax);
    *result=sqrtf(2.0*logf(temp2))*temp1;
}
```

B.4 levinsonDurbin.c

```
float *levinsonDurbin (float dm out[], float dm in[], int length, float
    *Ep) {
   int i, j;
   float temp[length+1], E, sum, k, A[length+1];

   E=in[0];
   for(i=0;i<=length;i++){
      temp[i]=0;
      A[i]=0;
   }
   for(i=0;i<length;i++){
      for(j=1;j<i+1;j++){
         temp[j]=0;
      }
      temp[0]=1;
      sum=in[i+1];
      for(j=1;j<=i;j++){
         sum=sum+(A[j]*in[i-j+1]);
      }
      k=-sum/E;
      for(j=1;j<=i;j++){
         temp[j]=A[j]+(k*A[i-j+1]);
      }
      temp[i+1]=k;
      E=E*(1-(k*k));
      for(j=0;j<=length;j++){
         A[j]=temp[j];
      }
   }
   for(i=0;i<=length;i++){
        out[i]=A[i];
   }
   *Ep=E;
}
```