Speech Analysis
Project in TTT4225 Applied Signal Processing

Benjamin Strandli Fermann and Even Flørenæs

Spring 2015

Summary

This project is a part of the course *TTT 4225 Applied Signal Processing* and is intended to provide practical experience with design and implementation of systems using various signal processing techniques. The project uses different signal processing tools and methods to implement a vocoder system, which is a compression and decompression system tailored specifically towards speech. We will present three different systems that solve this problem: a basic vocoder using linear predictive coding (LPC) and pitch properties of the speech, a Residual Excited Linear Prediction (RELP) coder with upsampling, and a RELP coder using high frequency regeneration. We will also discuss how using different prediction orders affects the systems. These systems were first implemented and tested in MATLAB and then implemented in C.

RELP using high frequency (HF) regeneration turned out to give the best quality output using the same prediction order on all of the systems using the MATLAB implementation. However, the C implementation of the RELP coder did not work properly, so only the basic vocoder is functional in C. Our system gives a speech output where it is easy to both hear and understand what is being said at high compression levels. Future work with the project would include fixing the RELP coder in the C implementation and creating an implementation that allows for real-time processing rather than batch, because this is necessary in most practical applications.

Contents

1 Introduction

The course *TTT 4225 Applied Signal Processing* focuses on design and implementation of signal processing system based on multiple modules using different signal processing algorithms and tools including filter analysis, correlation, frequency spectrums, modeling and estimation of real-world processes. These goals are realised through this project on speech analysis. We want to use models for speech to create a system that can effectively analyze and regenerate speech based on these models. This will enable very efficient compression of speech.

Efficient speech compression leads to lower bandwidth demands for speech transmission. This means higher capacity for mobile phone base stations and less power consumption. It also means voice-over IP can use less bandwidth where this is an issue. All kinds of speech-based digital communication can benefit from this. It might even lead to increased range on phones, walkie-talkies, etc. by using the extra freed up bandwidth for better error correction.

The report is organized in four parts: theory, our task, implementation and conclusion. In chapter 2 the theoretical background for the project and our task are described. In chapter 3 the tasks given in the project is described. Both the general tasks and also a group specific task. The chapter also contains figures with detailed system overviews which is the foundation for our implementation. The following chapters 4, 5 and 6 describes our implementation of the project divided into three parts: System implementation, MATLAB implementation and C implementation. The system implementation chapter describes which implementation choices we did, and was used in both the MATLAB and C programming. The MATLAB and C implementation chapters will focus on programming language specific implementation choices, and results of the implemented systems. Finally the conclusion is given in chapter 7.

2 Theory

This chapter will explain the theoretical background which is needed for solving the given tasks in the project.

2.1 Linear prediction

The vocal tract of a human can be viewed as a series of connected uniform, lossless tube sections. Analysis of this model gives results which corresponds with an autoregressive process, AR-process[1]. To estimate speech as an AR-process we need the speech to have the same properties for every part of the speech clip. If we divide the speech sample into small time intervals the statistical properties will be approximately the same for every time interval. With this property speech can be seen as a stationary process. By using the vocal model and analysis in limited time periods we can estimate speech data by using a FIR filter structure P-th order linear predictor

$$\widetilde{x}(n) = \sum_{k=1}^{P} a_k x_i[n-k] \tag{1}$$

The prediction error is then given by the difference between the original data and the linear predicted data

$$e_i(n) = x_i(n) - \widetilde{x}_i(n) \tag{2}$$

To optimize the predictor we want to minimize the power of the prediction error given in equation (3). To find the optimal solution we can use the power of prediction error as an object function ,J, and derivate J with respect to a prediction coefficient a_l.

$$J = E_i = \sum_{i=0}^{N-1} e_i^2(n) = \sum_{i=0}^{N-1} (x_i(n) - \widetilde{x}_i(n))^2 \tag{3}$$

$$\frac{\partial J}{\partial a_l} = \frac{\partial}{\partial a_l}(\sum_{n}(x_i(n) - \sum_{k=1}^{P} a_k x_i(n-k))) = 0$$

$$\sum_{n} 2(x_i(n) - \sum_{k=1}^{P} a_k x_i(n-k))(-x_i(n-l)) = 0$$

$$-\sum_{n} x_i(n)x_i(n-l) - \sum_{n}\sum_{k=1}^{P} a_k x_i(n-k)xi(n-l) = 0$$

Equation (4) can now be reduced by the definition of the autocorrelation expressed in equation (5).

$$\sum_{k=1}^{P} a_k \sum_{n} x_i(n-k)x_i(n-l) - \sum_{n} x_i(n)x_i(n-l) = 0 \tag{4}$$

$$\widehat{r}_x(k) = \sum_{n=0}^{N-k-1} x_i(n)x_i(n+k) \tag{5}$$

$$\sum_{k=1}^{P} a_k \widehat{r}_x(|l-k|) = \widehat{r}_x(l); l = 1, 2, ..., P \tag{6}$$

The result in equation (6) gives the basis for the Yule-Walker equations described in equation (7) and (8). The autocorrelation functions in equation (6) can easily be described as matrices depended of the variables l and k.

$$\mathbf{R} = \begin{pmatrix} \widehat{r}(0) & \widehat{r}(1) & . & . & . & \widehat{r}(P-1) \\ \widehat{r}(1) & \widehat{r}(0) & . & . & . & \widehat{r}(P-2) \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ \widehat{r}(P-1) & \widehat{r}(P-2) & . & . & . & \widehat{r}(0) \end{pmatrix}$$

$$\mathbf{r}^T = [\widehat{r}(1)\ \widehat{r}(2)\ ...\ \widehat{r}(P)]$$

$$\mathbf{a}^T = [1\ a(1)\ a(2)\ ...\ a(P-1)]$$

The arrays above for **R**, **a** and **r** describes the elements of equation (6), and can be joined to define the first Yule Walker equation in equation (7). In equation (8) the error variance in the model is described. The description of the error variance in equation (8), which is the Yule-Walker second equation, is found by using equation (6) in the case of $l = 0$.

$$\mathbf{Ra} = \mathbf{r} \tag{7}$$

$$\sigma_{e_i}^2 = r(0) + \mathbf{a}^T\mathbf{r} \tag{8}$$

This formulas gives the basis for linear prediction which can be used in for example estimation of speech. Practial use of linear predictors can be made efficient by using programming algorithms like Levinson-Durbin as described in algorithm 1.

Algorithm 1 Levinson-Durbin recursion

```
procedure LevinsonDurbin(r,P)
    $E \leftarrow r(0)$
    $b(0) \leftarrow 1$
    for $i \leftarrow 1, P$ do
        $k(i) \leftarrow r(i)$
        $a \leftarrow zeros(P+1)$                ▷ Array $a$ of length $P+1$ is set to zero
        $a(0) \leftarrow 1$
        for $j \leftarrow 1, (i-1)$ do
            $k(i) \leftarrow k(i) + b(j)r(i-j)$
        end for
        $k(i) \leftarrow -k(i)/E$
        for $j \leftarrow 1, (i-1)$ do
            $a(j) \leftarrow b(j) + k(i)b(i-j)$
        end for
        $a(i) \leftarrow k(i)$
        $E \leftarrow (1 - |k(i)|^2)E$
        for $j \leftarrow 0, i$ do
            $b(j) \leftarrow a(j)$
        end for
    end for
    return $a$                                   ▷ Return the filter coefficients
end procedure
```

2.2 Detection of voiced and unvoiced sounds

Voiced sounds like vowels and fricatives have a distinct pitch, this periodic nature gives speech short-time stationary properties that we can analyse by looking at a short time frame. Knowing the frequency range of a human voice, we can limit our analysis to look for pitch periods that are 2-20 ms long. Finding this pitch period can be achieved through auto-correlation. Voiced signals are periodic, so the auto-correlated signal will also be periodic as in Figure 1. Unvoiced signals(Figure 2) do not have this property, which makes it suitable to decide whether a speech segment is voiced or unvoiced. The period of the auto-correlated signal corresponds to the pitch period, and the relative amplitude of the peaks gives a measure of how voiced the speech is.

2.3 Hamming window

Window functions are used to smooth over the frequency response of signals. Both rectangular and non-rectangular window functions can be used for windowing. By using a non-rectangular window one can remove sudden and unexpected results in the impulse response, and reduce the height of the ripples. With the consequence of having a wider transition band[2].

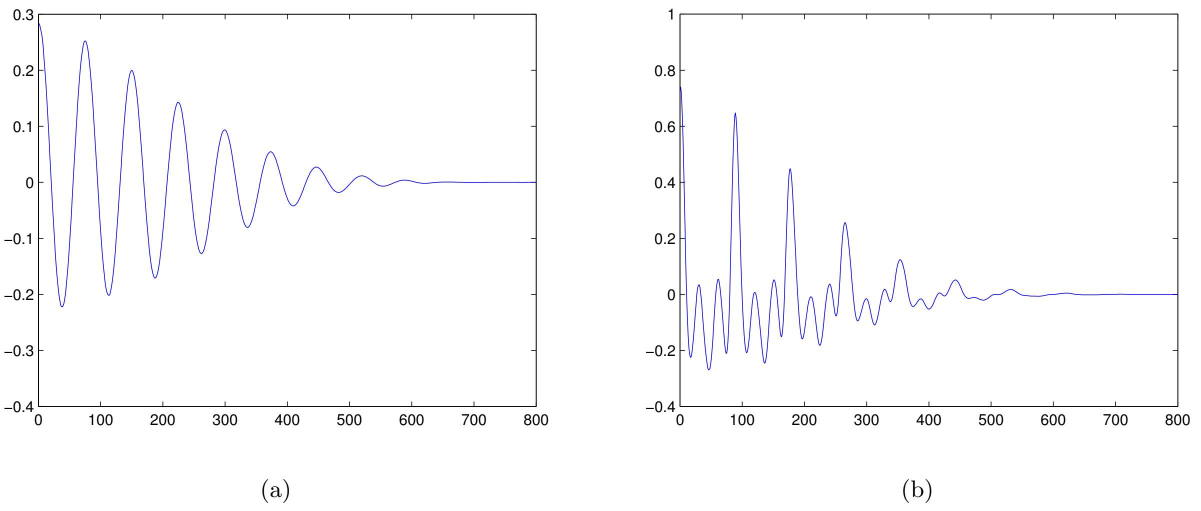


Figure 1: Auto-correlation of voiced sounds

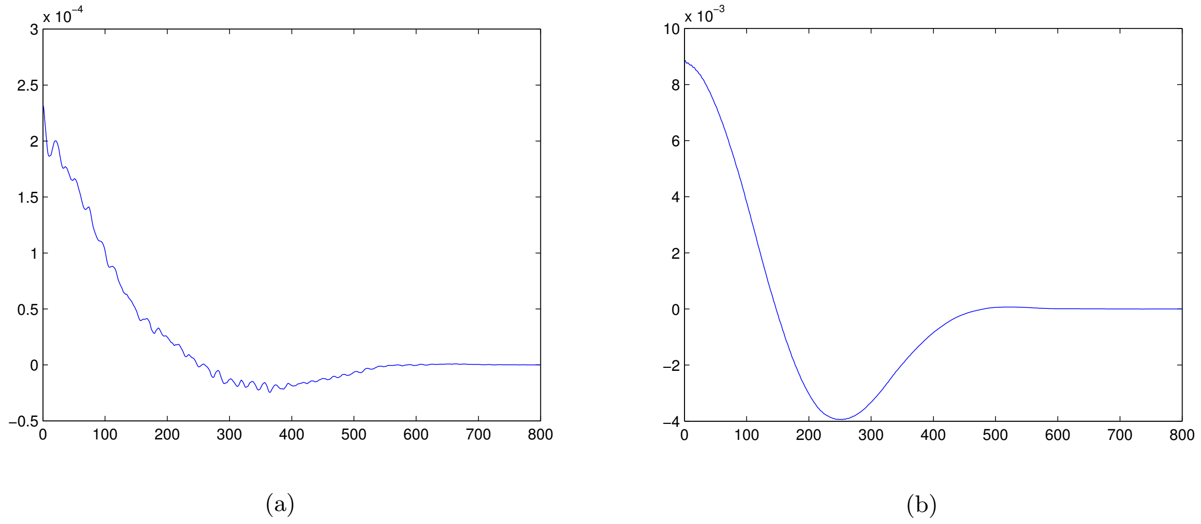


Figure 2: Auto-correlation of unvoiced sounds

A Hamming window is a non-rectangular window described by the formula in (9). The Hamming window is formed as a sinusoidal as shown in Figure (3) with maximum value 1 given in the center the period given by N.

$$w(n) = 0.54 - 0.46cos(2\pi\frac{n}{N}), 0 \leq n \leq N \tag{9}$$

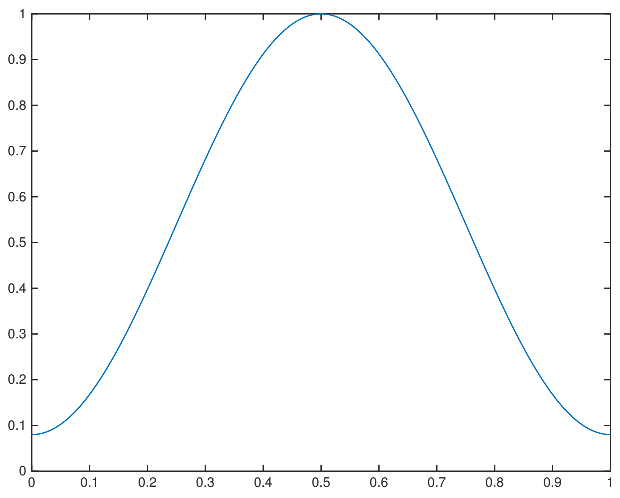


Figure 3: A Hamming window with 10000 samples

2.4 Decimation and interpolation

By decimating the signal, the bitrate is reduced and can later be regenerated (with loss of high frequency information) through interpolation. This is possible because the properties of speech are well known, so high frequency information can be generated based on the low frequency information retained after decimation.

The decimated signal consists of samples from the original signal with a constant interval of $D-1$ samples between each of the corresponding samples in the original signal, where D is the decimation factor. When interpolating back to the original sample rate, the missing $D-1$ samples between each known sample are substituted with zeros as shown in Figure 4.

In signal processing a signal may be decimated to reduce the bitrate of the signal, as explained in the section above. If we after transmission or processing wish to regenerate the signal with interpolation we will loose the high frequency

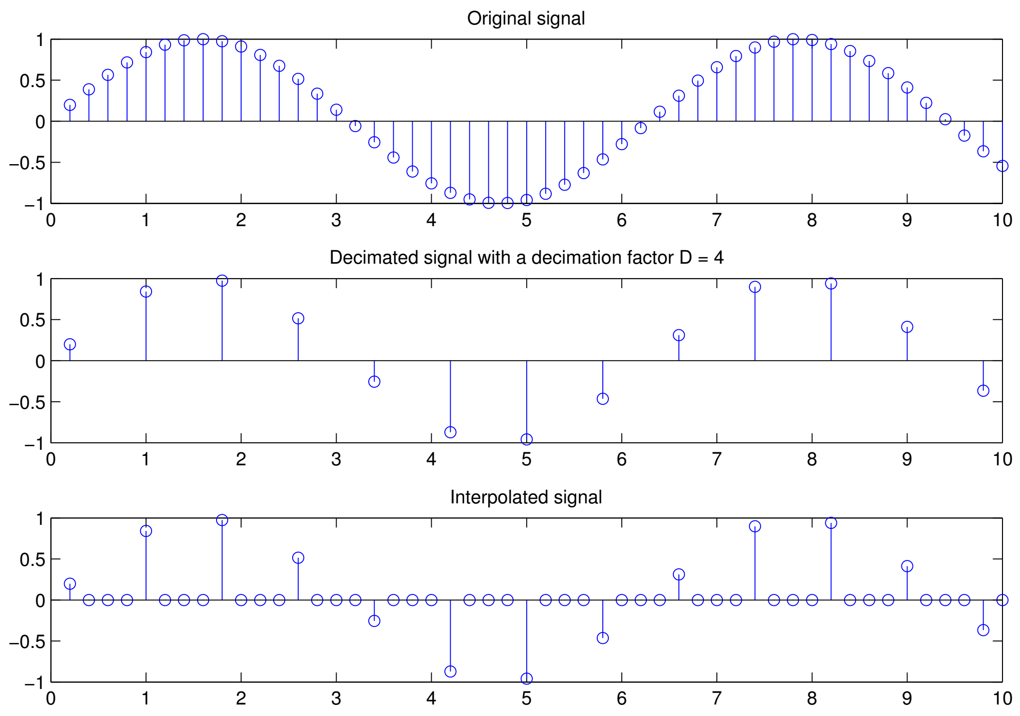


Figure 4: Example of a signal before and after decimation and after interpolation

information. The frequency spectrums of a decimated signal and a decimated and interpolated signal is represented in Figure 5. The figure shows how the high frequency information in the interpolated contains only the information of the low frequencies. The spectrum of the interpolated signal will also get repeating spectrums containing the information from the decimated signal. The repeating spectrums will also be inverted in interpolated signal[3].

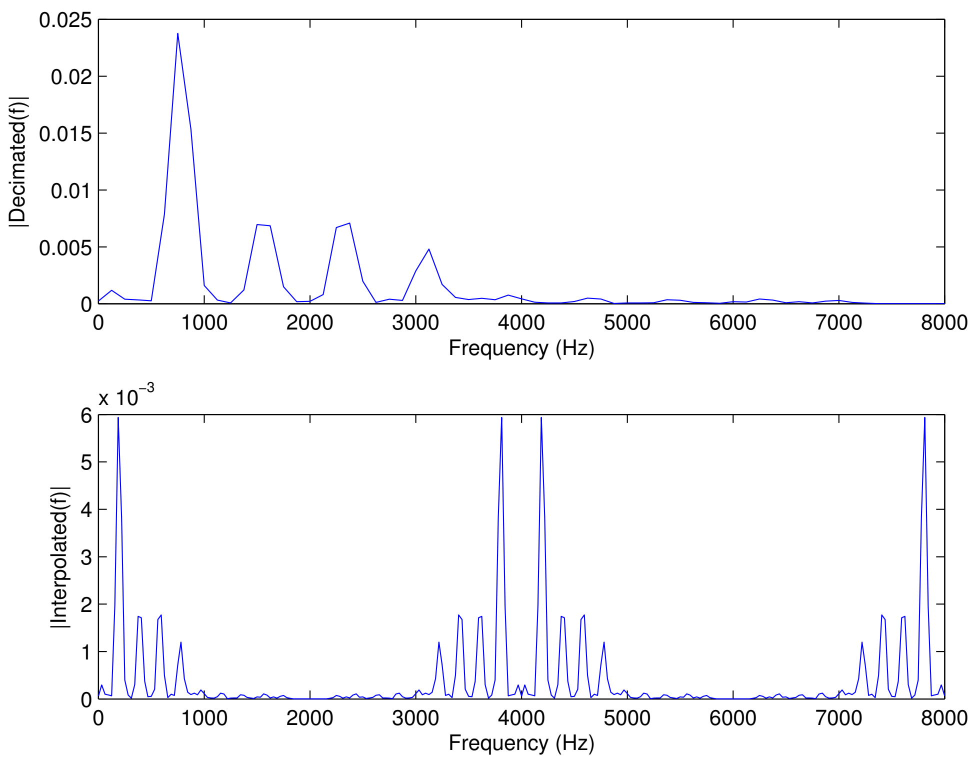


Figure 5: Single-sided amplitude spectrum of a downsampled signal and one signal both downsampled and upsampled

3 Our task

The problems given in this project are divided into tasks given to all the groups and a group specific task. The tasks given to all the groups concerns detecting voiced and unvoiced sounds, and implementing both a basic vocoder and a Residual Excited Linear Prediction (RELP) coder with prediction order $P = 14$. The group specific task which we are going to solve is to test our systems with different values for the prediction order.

3.1 Basic vocoder

In Figure 6 our system for basic vocoder is shown. In the figure the vocoder is divided into different blocks to make it easier to implement in both MATLAB and C code. The system in Figure 6 will take a small segment of the audio signal and shape it using a Hamming window. The resulting segment x_{filt} will then be used to generate filter coefficients using linear predictive coding (LPC), and generate either a pitch signal (pulse train with intervals based on pitch) or white noise based on the pitch properties. This pitch/noise signal will then be filtered using the LPC coefficients and finally low-passed filtered to remove undesirable high frequency components beyond the range of human speech.

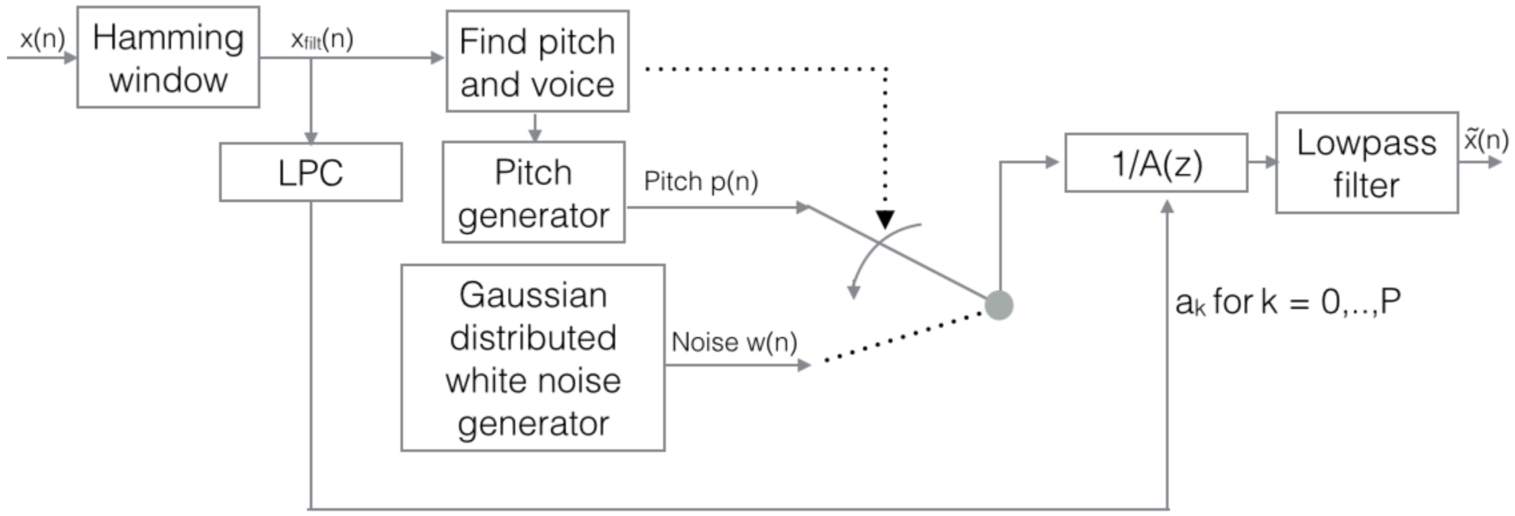


Figure 6: A system overview of the basic vocoder

3.2 Residual Excited Linear Prediction Coder

The system to the RELP coder is shown in Figure 7. The system overview gives the basis for MATLAB and C implementation. The system will take a small segment of the audio signal and shape it using a Hamming window. The

resulting signal will then be used to find filter coefficients using LPC. We will filter this using the LPC coefficients to find the predicted signal, which we then subtract from the unfiltered signal to find the prediction error. This prediction error will then be low-pass filtered to allow for decimation without aliasing.
We will then interpolate the decimated signal back to the original sampling frequency. This causes the low-frequency spectrum from the decimated signal to be replicated and inverted in the high frequency bands. This is undesired so we will low-pass filter the signal once more giving us the signal x_{LP}. We still want high frequency components and to create this we use half-wave rectification on x_{LP}. This signal is in turn filtered using filter coefficients obtained by using LPC-analysis on the rectified signal with a low prediction order. This creates desirable high frequency components which we will isolate using a high-pass filter. We will then combine this high frequency signal with the low frequency signal x_{LP} after making gain adjustments to normalize the two signals. Finally this will be filtered using the LPC coefficients found at the start of the system to produce the reconstructed signal.

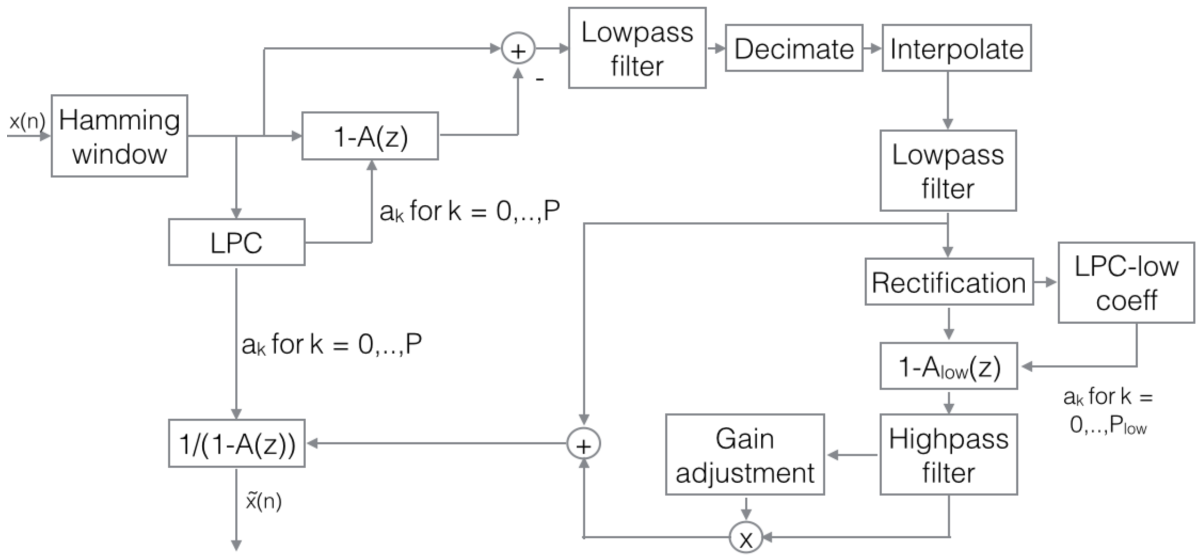


Figure 7: A system overview of the RELP coder

3.3 Testing systems with different prediction orders

Our group specific task is to test our systems, both the basic vocoder and the RELP coder, with different prediction orders in the LPC-analysis. In the general task the prediction order, P, is given as $P = 14$. In our work we will test our systems with $P = 14$ and additionally for $P = 2, 4, 6, 8, 10, 20$. We will try to subjectively decide the consequences of changing the prediction order.

4 System implementation

This chapter will explain which implementation choices we used in our project. Both for the basic vocoder and the RELP coder. The system specification is general for both the MATLAB and C application, and this chapter will describe the system choices which is the basis for the programming part of the project.

4.1 Linear Prediction Coding (LPC)

In order to find the LPC coefficients, the Levinson-Durbin recursion is used as described in algorithm 1. For this system a prediction order P = 14 was chosen, although other orders were also tested (see section 3.3). These coefficients are used to filter the output from the pitch/noise generator ($z(n)$ in Figure 8) in the basic vocoder, but also used in the RELP coder.

4.2 Basic vocoder

To ensure the properties of stationary process we will divided the input signal in smaller time intervals of length 20 msec. The signal $x(n)$ in Figure 6 is a 20 msec interval of the input speech data. The first part of the basic vocoder is to filtrate $x(n)$ by 30 msec Hamming window centered around the 20 msec interval. The filtrated signal $x_{filt}(n)$ gives the basis for the linear prediction in the basic vocoder. The input to the pitch estimation system given in Figure 8 is filtrated with a Hamming window in the same way, but uses a 50 msec window instead.

Pitch estimation

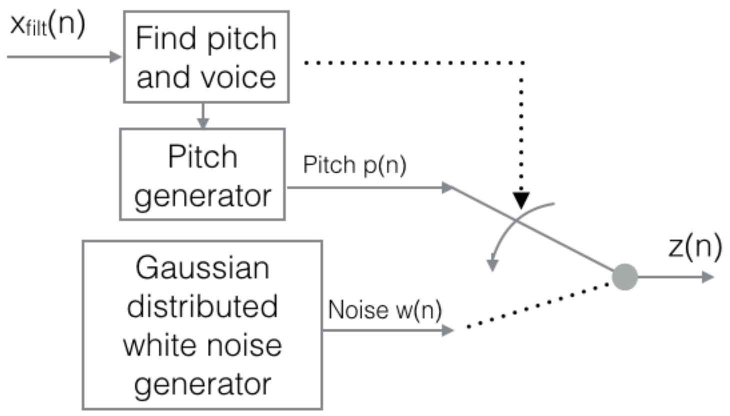
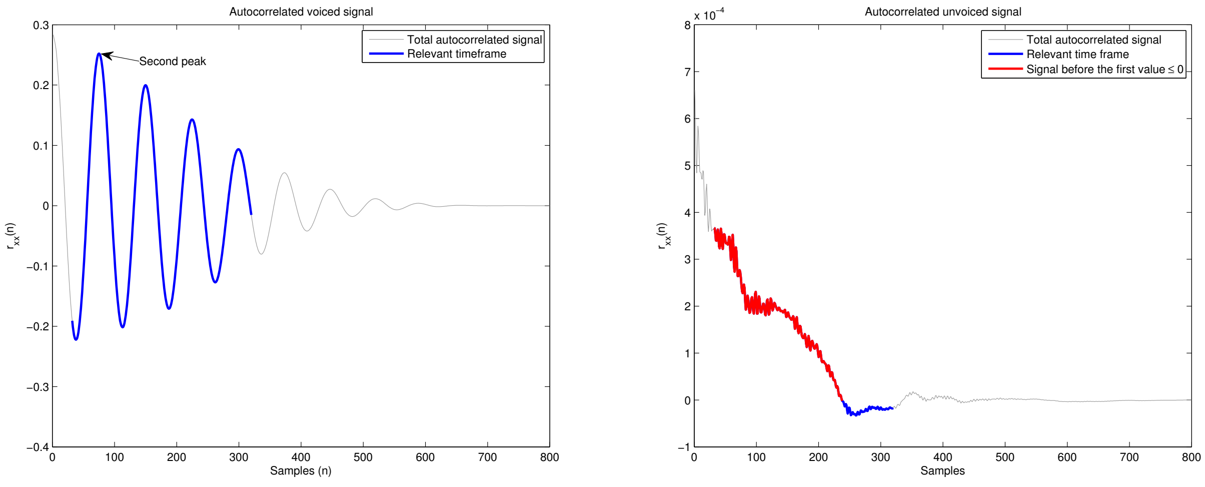


Figure 8: The pitch estimation part of the basic vocoder from Figure 6

In order to find the pitch period and thereby decide whether or not the signal is noised we will first find the auto-correlation of the signal. Observing the auto-correlated signal of voiced speech (e.g. Figure 1) it is clear that all of them go below zero before the second peak, while unvoiced signals get gradually less correlated and never get a significant peak after getting below zero. Additionally, we know that the expected frequency range of the pitch is 50-500 Hz, so it is only necessary to look at a window of 2-20 msec rather than the full 0-50 msec frame from the autocorrelation as shown in Figure 9a. Using the limited



(a) Voiced (b) Unvoiced

Figure 9: Select a time frame to include only relevant pitches.

time frame, it is easy to find the maximum value and the time at that point. However, the auto-correlation of unvoiced and low-pitched speech may decay so slowly that the maximum value is the very first in the time frame and has a relatively high value. In order to avoid this the time frame is further limited by starting at the first sample with a value ≤ 0 as shown in Figure 9b.

The time to the maximum value found, n_{max}, gives us the pitch which is sent to the pitch generator. In order to use the signal from the pitch generator, it is also necessary to make sure the signal is actually voiced. To solve this problem we only use the pitch generator if the maximum value is sufficiently large, large enough to fulfill the inequality in equation (10), where α is a chosen threshold. Otherwise gaussian white noise is used instead as shown in Figure 8.

$$\frac{r_{xx}(n_{max})}{r_{xx}(0)} > \alpha \tag{10}$$

This pitch/noise signal is filtered using LPC coefficients based on x_{filt} as described in section 4.1. To avoid undesired high frequency components beyond the range of human speech, we low-pass filter the signal to obtain the final reconstructed output, $\widetilde{x}(n)$.

4.3 RELP coder

High frequency regeneration

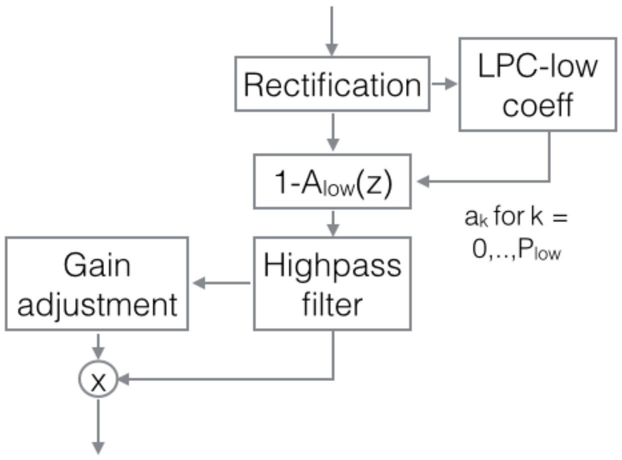


Figure 10: The high frequency excitation part of the RELP coder from Figure 7

The different operations in the RELP coder will have effects on the frequency response to the input signal. To assure that the frequency response on the input is contained on the output, $\widetilde{x}(n)$, of the system, we use the high frequency regeneration method. The high frequency excitation part of the RELP-coder is shown in Figure 10.

The input to the rectification in Figure 10 is the lowpass filtrated output of both decimation and interpolation. The output of the interpolate block in Figure 7 will have the frequency spectrum properties like shown in the Figure 5 in section 2.4. In a speech signal voiced and unvoiced parts of the signal will behave differently concerning high frequency regeneration. For a unvoiced part of a speech signal the output of a decimation and interpolation will be sufficient to represent the high frequency properties, but for a voiced signal we will encounter aliasing after decimation and interpolation. The high frequency part of the RELP coder will ensure to keep the high frequency properties of the voiced signals. First the input to the high frequency excitation will be rectified. The output of the rectifier will be spectrally flattened by coefficients produced by a LPC analysis. The prediction order in the LPC should smaller than for the rest of the system, and is decided to be 4.

To join the high frequency parts with the low frequency we will have to highpass filtrate the output of the LPC analysis. By using the the highpass filter we will produce a high frequent signal which will not change the low frequent properties from the interpolation. In the RELP coder we also do a gain adjustment of the high frequent signal to ensure that the low frequent and the high frequent parts is adjusted, and makes a good representation of both the low frequency and high frequency properties in the output.

4.4 Testing systems with different prediction orders

To find an answer to the group specific task we have made a test function to both the basic vocoder and the RELP coder where we can listen to the output of the system with different prediction order. We have made the test functions to work as double blind tests where the different outputs will be presented randomly to the listener without neither the listener or the researcher knowing the value for P in the coder in the different outputs being presented. With this method we hope to achieve a concluding answer to the question concerning how important the choice of prediction order is, and how the output changes for a set of prediction orders.

The implementation of the test functions will be done in MATLAB. To answer the group specific task the output of our programs and differences between the programs with different prediction order is the most crucial. Which language-based solution we used should in principle be trivial. For that reason we choose the programming language which demands the least work to implement the test functions, and in the most cases that will be MATLAB.

5 MATLAB implementation

This chapter will present the results from our MATLAB implementation. Both the results of the general task and the group specific task will be presented and discussed. The chapter will also briefly explain MATLAB as a programming language and which predefined MATLAB functions we used to make our implementation.

MATLAB is a high-level programming language which means that the language does not need the programmer to for example defining variable or define array sizes. MATLAB is an important tool in signal processing, and generally used for testing. MATLAB is also a C based programming language which means making a C implementation of the MATLAB code is easier than for other programming language that are not C based. The MATLAB implementation is batch. The program will do the all the processes at once and will not be suitable for a real-time application.

5.1 Use of MATLAB functions

MATLAB includes many functions that the user can benefit from in project works. The drawback of using such functions in this project, is that we were supposed to develop a programming solution for both MATLAB and C, and even do we can use pre-defined functions in MATLAB we still need to develop them for C-programming. For that reason we decided in many cases to make our own functions in MATLAB instead of using pre-defined functions to avoid much higher workload on the C part of the programming.

We did still use some predefined functions in our programming. In the basic vocoder and RELP we used functions like wavread, zeros, fir1, filter, decimate, upsample, max, snr and randn.

`wavread`	`wavread` is a function which is used to read in wav-files. In the project we used wav-files of speech to test our system. The wavread-function reads the wav-files, and returns an array with length $F_s * L$, where L is the length of wav-file.
`zeros`	`zeros(`x`,`y`)` generates a matrix of size $x * y$ with zeros as content in every position. In our project we used `zeros` to predefine arrays we used in our MATLAB code.
`fir1`	`fir1(N,2`$\frac{F_c}{F_s}$`)` generates the coefficients for a filter with cut-off frequency F_c, sampling frequency F_s and filter order N. In our code we used the `fir1` function to generate a low-pass filter for the basic vocoder and both low-pass and high-pass filter in the RELP-coder.
`filter`	`filter(`b_k`,`a_k`,`$y(n)$`)` filtrates the input signal based on the coefficients given as b_k and a_k. The coefficients are important in this function. Different coefficients can give either a lowpassfilter, highpassfilter or be output of a linear predictor function.
`decimate`	`decimate(`$y(n)$`,D)` will decimate the input $y(n)$ with decimation factor D. The function works as explained in the theory on shown in Figure 4.
`upsample`	`upsample(`$y(n)$`,I)` will interpolate the input $y(n)$ the factor I. The properties of interpolation is shown if Figure 4. In our project the `decimate` and `upsample` function will be used together.
`max`	`max(`$y(n)$`)` will find the maximum value in array given as input.
`snr`	`snr(`$x(n)$`,`$y(n)$`)` finds the signal-to-noise of a signal $x(n)$ compared with the noise $y(n)$ [4]. The `snr` function is only available in MATLAB R2014.
`randn`	`randn(`x`,`y`)` returns gaussian distributed randomly generated numbers in a matrix of size $x * y$.
`sound` and `soundsc`	`sound(`$y(n)$`,`Fs`)` plays the input $y(n)$ with the given sampling frequency Fs. In `sound` the input $y(n)$ needs to be values between 0 to 1 to avoid clipping in the output sound. The function `soundsc` has same functionality and takes the same inputs as `sound`, but automatically scales the input values in $y(n)$ to be between 0 and 1 and will always avoid clipping.

Table 1: Description of functions used in the MATLAB programming

5.2 Results

We measured the SNR of the two RELP systems using the `snr` function in MATLAB, comparing the original signal with the difference between the original and generated signal at $P = 14$ and found:
0.07dB SNR for RELP with upsampling
0.01dB SNR for RELP with HF regeneration

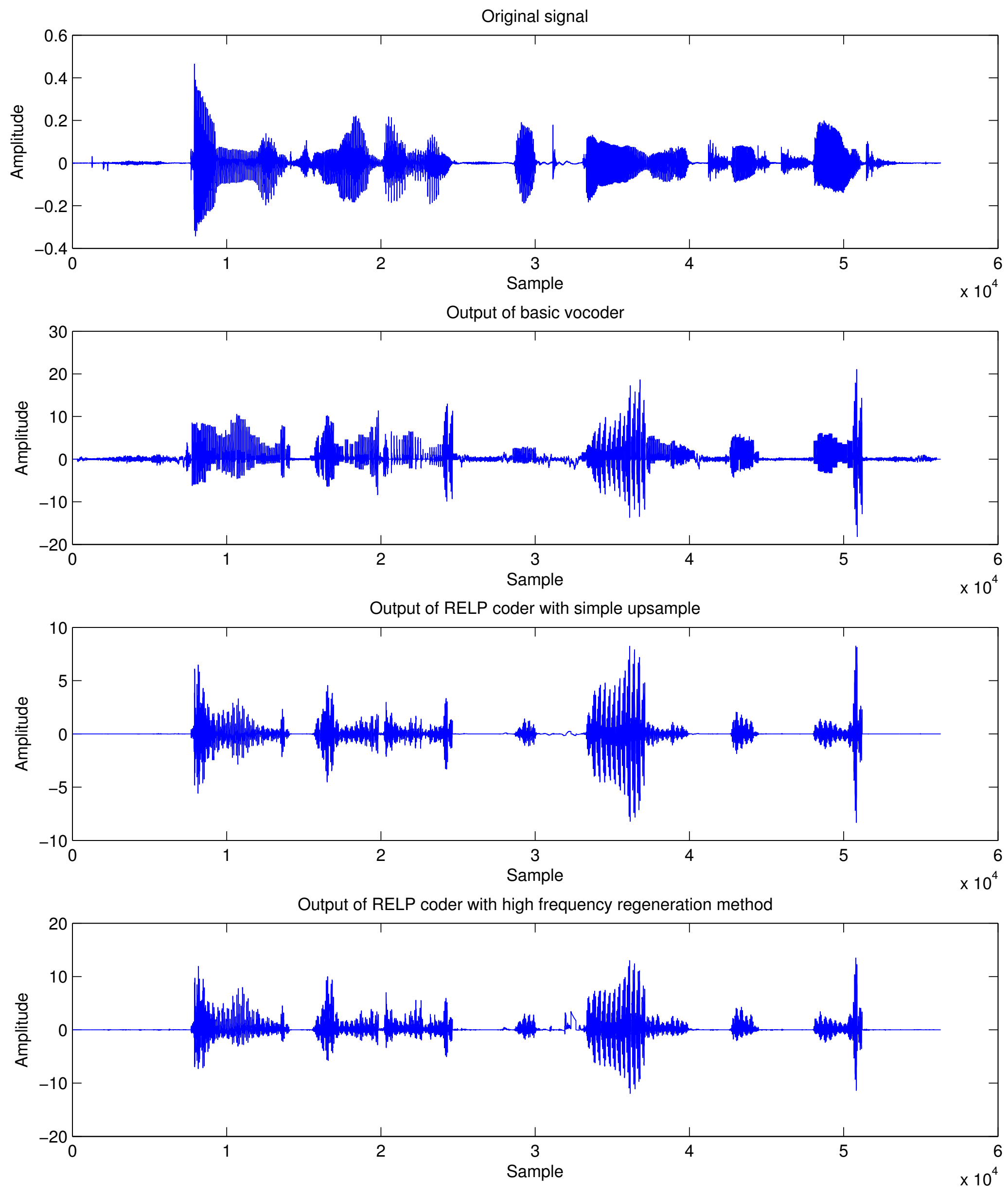


Figure 11: Input signal compared with output for the two systems

Basic vocoder	The output of the basic vocoder has reduced quality compared to the original speech input, but it is still easy to both hear and understand what is being said in the clip. The output sound is noisy and the speech does not sound as natural as in the input.
RELP coder with upsampling	With the RELP coder with upsampling it is also possible to hear and understand the speech output. The sound is lower both in volume and also in frequency. The speech does not sound natural.
RELP coder with high frequency regeneration method	The quality of the output with the high frequency algorithm is better than for both the basic vocoder and the upsampled version of the RELP. The output is still quite noisy, but the speech sounds more natural.

Table 2: Description of the sound quality of the output of the systems

$P = 2$	Very low quality and much noise on the output. Difficult to hear and understand the speech.
$P = 4$	Like for $P = 2$ it is very low quality and contains much noise on the output. Difficult to hear and understand the speech.
$P = 6$	Quality is still very low, and much noise on the output. A little easier to understand the speech.
$P = 8$	Some more noise then in the case of $P = 14$. Easy to understand the speech.
$P = 10$	Little lower quality then for $P = 14$, but very much like the $P = 8$ case.
$P = 20$	Better then for $P = 14$. Easy to understand the speech and quality is relatively good.

Table 3: Testing of basic vocoder system with different prediction orders

$P = 2$	Barely able to understand the speech. Very low quality.
$P = 4$	Like for $P = 2$ barely understandable speech. Noisy and low quality output.
$P = 6$	Better then for the cases with $P = 2$ and $P = 4$, but still very low quality and noisy output. Little easier to understand the speech.
$P = 8$	The quality is quite like for $P = 14$, but a little more noisy. Easy to understand the speech.
$P = 10$	Negligible difference from $P = 8$. Easy to understand the speech.
$P = 20$	Using $P = 20$ instead of $P = 14$ does not enhance the quality on the output in this case. Easy to understand the speech.

Table 4: Testing of RELP coder system with high frequency regeneration with different prediction orders

5.3 Discussion

The output of the two systems, with a testfile *anvsb1.wav* with 24 sec speech sample as input, is presented in Figure 11 compared with the input. The figure shows that the output of the systems gives results which differs from the original speech sample. There are areas where the coders does not estimate the data correctly, like after 50 000 samples both the basic vocoder and the RELP representes the input signal with a large sudden top, but in the original there is no such variation at this point.

The quality of the basic vocoder is limited by the quality of the implementation of the pitch estimation. That shows in the result in Figure 11. The data produced by the basic vocoder is decided by the pitch estimation which sends out a pitch segment or noise segment depending on the speech sample in the time limited period being voiced or unvoiced. This gives a result for the basic vocoder which analyzes 20 msec speech segment at the time and does not represent sudden changes in value precisely. The RELP coder is not depended of detection of voiced and unvoiced speech, which makes the quality of the coding more stable. The coder is therefore able to detect more sudden changes in the speech sample.

Figure 11 also shows that the results for the basic vocoder and the RELP coder has amplitudes which is much higher than the the input signal. The values of the input signal varies between 0 and 1, but the results of our systems have values which varies between 0 and around 15-20. The reason for this effect is that the processing of the speech by using LPC-filtering and low and highpass filtering results in gain for the input signal. By using the `soundsc`-function to present our results the values will be automatically adjusted to vary between 0 and 1, like the input signal. Alternatively we could have manually gain adjusted the output of the systems by dividing every value by the max value making the max value of the output 1.

The results of listening to the outputs presented in table 2 shows that the quality of a speech sample will be quite reduced in both the basic vocoder and the RELP coder. The basic vocoder output is noisy which is a result of the implementation of the pitch estimation. The pitch estimation will estimate unvoiced speech with noise, which is a simplification that in many cases sounds unnatural. There is clear differences between the results for the two versions of the RELP coder. The output of the RELP coder with simple upsampling has clear signs of aliasing where the speech is voiced. In that part the high frequency method gives a better representation of the input.

The quality of the outputs are still quite low compared to the input. Both systems have limitation concerning making a good representation of the input, but some quality loss may originate from our implementation and our system choices both in the pitch estimation and in the high frequency regeneration method.

The results of the group specific task are presented in table 3 and 4. We investigated the effect of different prediction order by running multiple random tests and comparing the different results. The tests clearly shows that the choice of prediction order is crucial to the quality in the coders. For the basic vocoder the quality is too low to be suited for any practial use if $P \leq 6$. The noise level in these cases are high because the prediction filter used will not be able to make a good representation of the original signal. Using $P = 8$ or $P = 10$ in the basic vocoder gives negligible difference from $P = 14$. Using a greater value for P than 14 will enhance the quality of our basic vocoder. For the RELP coder prediciton order $P \leq 6$ will not give a practically usable solution. Using $P = 8$ or $P = 10$ will give relatively the same quality as for the original $P = 14$. Unlike for the basic vocoder using $P = 20$ does not enhance the quality of the output compared to $P = 14$.

The SNR results for the RELP coder with upsampling and HF regeneration shows that the ouputs are very noisy. The noise part of the signal is almost the same power as the signal part. The SNR results also gives that the upsampling version has the best signal to noise ratio, even do the difference between them is rather small. This is quite contradicting to the result of the subjectiv listening described in table 2, and makes us question the SNR. The choice of inputs to the `snr`-function might be wrongly chosen, and gives a SNR which wrongly describes our implementation.

6 C batch implementation

This chapter will discuss the major differences between the C and MATLAB implementation, presentation of the results and a brief discussion of the usefulness of this implementation as both a general system and as a C implementation in particular, and potential for further development.

6.1 Differences from MATLAB

One of the major differences between our C and MATLAB implemetation, is reading and writing audio files. In MATLAB this problem is trivial as it can easily be solved using functions such as `wavread`. However this is not the case for C, our solution was instead to manually read the bytes from the header file to obtain the audio data and convert it from integer to float format.

Additionally the implementation of Levinson-Durbin recursion differs slightly from the MATLAB version (as described in algorithm 1). This change was done to minimize the amount of array operations necessary, as these are very verbose in C because of the need to do operations on each element individually rather than the whole array at once.

The `filter`, `fir1`, `decimate` and `upsample` functions are also not available in C, so we had to implement this functionality ourselves. However, we aimed to keep the over-all structure of the system similar to the MATLAB implementation, so implemented these functions so they would behave in the same way as the MATLAB equivalent.

It turns out that the RELP implementation did not work as intended, scaling the amplitude to the highest peak the volume was too low to hear any speech, only a few pops. The crude solution to this was to scale everything to the average amplitude and clip everything with a larger amplitude. This made it possible hear some of what was said, but ultimately the RELP coder should be considered broken.

6.2 Results

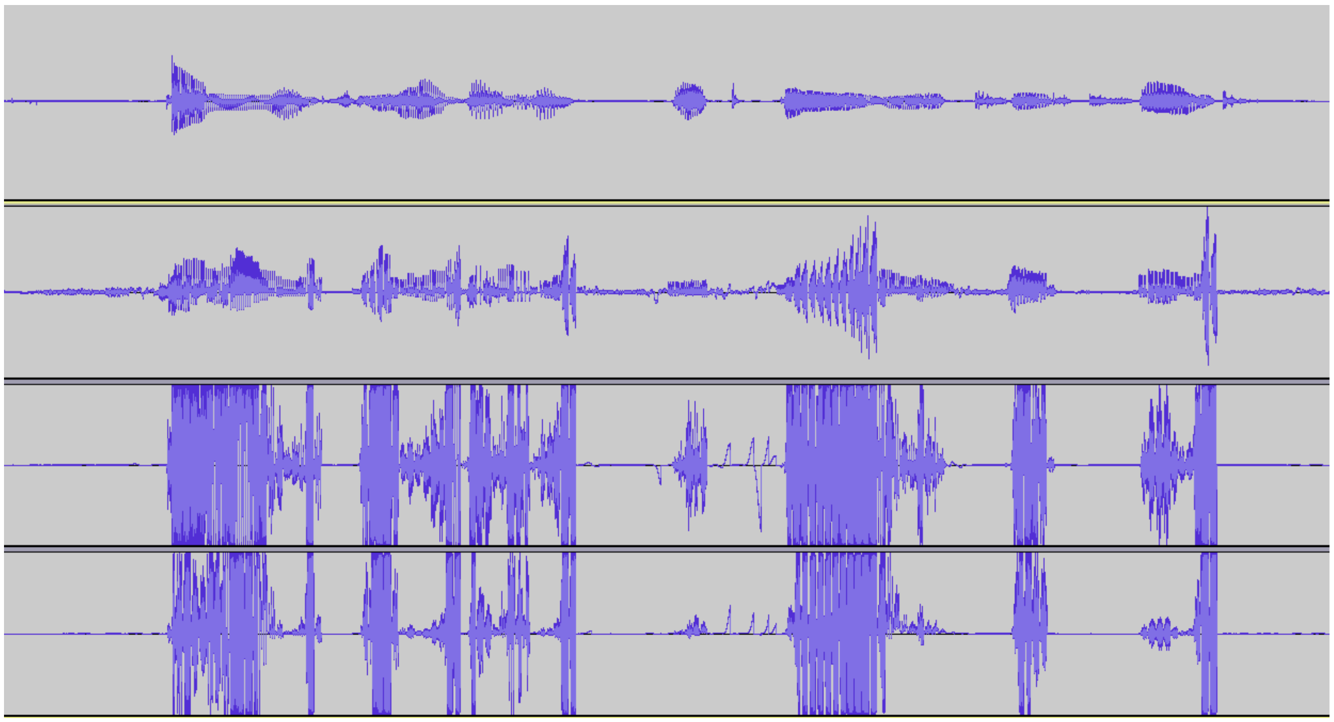


Figure 12: Input signal and resulting signals for the C implementation. First graph(from the top):Input signal. Second graph: basic vocoder. Third graph: RELP coder with simple upsample Last graph: RELP coder with algorithm

Basic vocoder	The output of the basic vocoder has reduced quality compared to the original speech input, but it is still easy to both hear and understand what is being said in the clip. The output sound is noisy and the speech does not sound as natural as in the input. Similar to MATLAB implementation
RELP coder with upsampling	Barely intelligible, heavy clipping, not suited for practical use.
RELP coder with high frequency regeneration method	Not intelligible, heavy clipping, not suited for practical use.

Table 5: Description of the sound quality of the output of the systems - C implementation

6.3 Discussion

While this C implementation aimed to replicate the system in the MATLAB implementation, the results were different. This is caused by some yet unresolved bugs in the implementation. The basic vocoder seems to be identical or very similar to the MATLAB implementation, although the RELP coder is much worse. it is pretty much impossible to understand RELP with HF regeneration, but using upsampling it is slightly better, but still bad.

While these vocoder systems were a good fit for MATLAB's abundance of built-in functions for signal processing and straight-forward handling of arrays, C is still a very useful language for these systems. MATLAB is excellent for prototyping, but as an interpreted language it is reliant on pre-installed MATLAB software to run. While C-code can be compiled to run on pretty much anything, embedded systems included. Being able to transmit intelligible speech with a low bitrate is useful for many kinds of communication, whether it be VoIP with low bandwidth, mobile phones, walkie-talkies or other means of speech transmission where bitrate is limited.

For most practical purposes, a real-time implementation would be necessary, for that reason, this would be a priority for any further development of the system. Because most of the system already operates on small segments of the signal, it should be able to create a real-time system without changing much of the core system. The biggest challenge would be handling I/O, which would involve a complete re-write of the current file I/O. Furthermore there's still potential for improvement as the RELP coder in particular does not work as intended in the C implementation.

7 Conclusion

In this report we have investigated the effects of different signal processing tools and methods in the design and implementation of three different systems for speech analysis and regeneration. We implemented a basic vocoder generating pitch pulses based on the short-time stationary properties of the speech and filter the generated signal based on coefficients from LPC analysis. LPC analysis was also used in the RELP-coder which also used decimation followed by interpolation as simple compression before filtering with LPC coefficients. The RELP coder with HF regeneration added a few more steps after interpolation where it split the signal into a low frequency and high frequency component, again using LPC to regenerate some of the information lost in decimation.

The project has given great knowledge about more practial use of the signal processing theory and tools we have learned from earlier courses like *TTT 4120 Digital Signal Processing* and *TTT 4110 Signal Processing And Communication*. The earlier courses have primarly been theoretical and mathematical, and this project has given an insight into the applied work in the field of signal processing. Working with the project has also been a great way to learn more about C programming. The C implementation lead to hours of testing and error handling for bug problems with read in/out wav files, allocating memory for arrays and implementing MATLAB functions, like the `filter`-function.

We found that RELP with HF regeneration gave the best results in the MATLAB implementation, using a prediction order of $P = 8$ or higher gave intelligible speech of decent quality, higher prediction order gives better quality, but there are diminishing returns. For the C implementation, only the basic vocoder worked as intended. The results of our implementation of the RELP coder in C does not work as our MATLAB implementation, and gives an output speech which is noisy and not usable in it is current form.

Our results show that it is possible to reduce bitrate of speech significantly while still retaining the most important information, but our results also show that the speech quality have great dependencies concerning the implementation of the system and choice of prediction order in the LPC analysis.

8 References

[1] T.Svendsen, *LPC – Linear Predictive Coding*,Presentation in TTT4240 Applied Signal Processing, 2015

[2] Dimitris G Manolakis and Vinay K. Ingle, *Applied digital signal processing*, Cambridge University Press, 2011

[3] T.Svendsen *Applied signal processing -Autoregressive modeling of speech*,Lecture notes,NTNU January 2014

[4] MathWorks documentation, *http://se.mathworks.com/help/signal/ref/snr.html, visited 18.04.2015*, The Mathworks, Inc.

Appendix A Source code

A.1 MATLAB code

basicVocoder.m - main function

```
% Reads audio from a .wav-file specified in the string soundFile, analyze the signal properties
% and regenerate the signal using P filter coefficients.

function x = basicVocoder(soundFile,P)
y = wavread(soundFile); %loading speech signal
Fs = 16000; %Sampling frequency
Fc = 4000; %Cut-off frequency for low-pass filter
N = 8; % Filter order for FIR low-pass filter

alpha = 0.5; %Boundary for what will be seen as voiced
beta= 0.1; %Gain to the noise signal

windowSpeech = hammingWindow(0.03*Fs);
windowPitch = hammingWindow(0.05*Fs);
%Initialize the arrays to be used in the basic vocoder for synthetic speech
pitch = zeros(1,length(y));
noise = zeros(1,length(y));
vocoderInput = zeros(1,length(y));
synthezised = zeros(1,length(y));
%Initialize the last values
last = 1;
lastPulse = 1;

% Find filter coefficients for a N'th order FIR low-pass filter with cut-off frequency Fc.
[bLow,aLow] = fir1(N,2*(Fc/Fs));

% Iterating through the signal, analyzing and regenerating the signal 20ms at a time.
% 'i' marks the middle of the 20ms frame.
for i = Fs*0.03:Fs*0.02:length(y)-0.025*Fs
    % Shape the relevant signal segments using Hamming-windows.
    lastSpeech = i+1-0.015*Fs;
    nextSpeech = i+0.015*Fs;
    lastPitch = i+1-0.025*Fs;
    nextPitch = i+0.025*Fs;
    y_filt = windowSpeech.*y(lastSpeech:nextSpeech);
    y_pitch = windowPitch.*y(lastPitch:nextPitch);

    r_y = autocorr(y_filt);
    % Levinson-Durbin recursion is used on the auto-correlation of the signal to find filter coeffic
    A = LevinsonDurbin(r_y,P);
    % Find the period of the pitch and a voiced constant (between 0 and 1, larger for voiced sounds)
    [pitchPeriod, voiced] = findPitchAndVoice(y_pitch,Fs);

    % Generate a signal using either white noise or pulse trains depending on the voiced properties
    % of the signal segment.
    if voiced >= alpha
        % When the signal is voiced, make a pulse train of "dirac"-pulses with intervals
        % defined by the previously found pitch period.
        last = i+0.01*Fs-pitchPeriod-1;
```

```
            for j = lastPulse:pitchPeriod:i+0.01*Fs
                pitch(j) = 1;
                if j> last
                    lastPulse = j;
                end
            end
        else
            % When the signal isn't voiced, white noise is used as a signal, scaled down with 'beta' gai
            lastPulse = i+0.01*Fs;
            noise(i+1-0.01*Fs:i+0.01*Fs) = beta*randn(1,0.02*Fs);
        end
        vocoderInput = noise+pitch;

        % Adjusts numbers that are too large, in the rare case of white noise getting very large values.
        vocoderInput(vocoderInput > 1) = 1;
        temp = vocoderInput(i+1-0.01*Fs:i+0.01*Fs);
        % Filter the generated signal using the filter coefficients found by the Levinson-Durbin recursi
        synthezised(i+1-0.01*Fs:i+0.01*Fs) = filter(1,A,temp);
    end
    % Low-pass filter the generated signal to remove spurious high frequency elements.
    synthezised = filter(bLow,aLow,synthezised);

    x = synthezised;
    end
```

RELPcoder.m - main function

```
% The function RELPcoder takes a sound file and the order P to be used in the linear prediction as a
% returns to arrays called x_algorithm and x_upsample. x_algorithm is the output of high frequency r
% using the algorithm explained in the task theory. x_upsample use upsampling to regenerate the
% high frequencies.
function [x_algorithm, x_upsample] = RELPcoder(soundFile,P)
    %Take in the data in the sound file
    speech = wavread(soundFile);

    %Initializing the signals that will be estimated
    synthetic_error = zeros(1,length(speech));
    synthetic_speech_upsampling = zeros(1,length(speech));
    synthetic_speech_HFregeneration = zeros(1,length(speech));

    %Declaring the theoretical values for different parts of the RELPcoder
    Fs = 16000; %Sampling frequency
    Fc = 2000; %Filter frequency (to be used in both lowpass- and highpassfilter)
    D = 4; %Decimated factor decided to be 4
    low_p = 4; %Order to be used in low p LP analysis of the refraction signal in high frequency reg
    N = 8; %Order to be used in the produced filters

    %Making a Hamming window, using a self-made function, used for estimation
    windowSpeech = hammingWindow(0.03*Fs);

    %Finding the coefficients for both lowpass- and highpassfilter
    [bLow,aLow] = fir1(N,2*(Fc/Fs));
    [bHigh,aHigh] = fir1(N,2*(Fc/Fs),'high');

    %Indexing the for-loop to start after 0.03*Fs and then use 0.02*Fs as step
    for i = Fs*0.03:Fs*0.02:length(speech)-0.025*Fs
```

```
%LastSpeech and nextSpeech keeps track of the start and end of the intervall to be processed
lastSpeech = i+1-0.015*Fs;
nextSpeech = i+0.015*Fs;

%Filtrating the speech with the window function
speech_filt = windowSpeech.*speech(lastSpeech:nextSpeech);

%Finding the LP-coefficients using a made LevinsonDurbin function which uses the autocorrela
A = LevinsonDurbin(autocorr(speech_filt),P);

%Filtrate the speech with found A-coefficients
speech_est = filter(A,1,speech_filt);

%Finding the difference between speech and LP-estimated data
speech_error = speech_filt - speech_est;

%Storing found result on synthetic_error which has the same length as input signal
synthetic_error(i+1-0.01*Fs:i+0.01*Fs) = speech_error(0.005*Fs+1:0.025*Fs);

%Lowpass filtering the error
speech_error = filter(bLow,aLow,speech_error);

%Decimation of filtrated error
high_freq_regenerate  = decimate(speech_error,D);

%Upsampling the decimated signal. This is the output to filtering in RELPcoder with upsampli
zero_insertion = upsample(high_freq_regenerate,D);

%Dividing into low frequencies and high frequencies as described in the project theory
LF_excitation= filter(bLow,aLow,zero_insertion);
HF_excitation = LF_excitation;
HF_excitation(HF_excitation<0 ) = 0; %Removing negative values of HF_excitation

%Computing the low LP-analysis
low_p_LP_analysis = LevinsonDurbin(autocorr(HF_excitation),low_p);
HF_excitation = filter(low_p_LP_analysis,1,HF_excitation);

%Higpass filtering HF_excitation to avoid higher frequencies
HF_excitation = filter(bHigh,aHigh,HF_excitation);

%Gain adjustment to adjust HF_excitation in relationship to the LF_excitation
%with using the maximum of the auto correlation as gainfactor for both signals
gainHF = max(autocorr(HF_excitation));
gainLF = max(autocorr(LF_excitation));
gain = gainLF/gainHF;
HF_excitation = gain .* HF_excitation;

%Addding both the low frequencies and the high frequencies
synthetic_fullband_residual = LF_excitation + HF_excitation;
%Computing the upsampled solution
speech_upsampling = filter(1,A,zero_insertion);

%Computing the algorithm solution
speech_HF = filter(1,A,synthetic_fullband_residual);

%Storing the values
```

```
        synthetic_speech_upsampling(i+1-0.01*Fs:i+0.01*Fs) = speech_upsampling(0.005*Fs+1:0.025*Fs);
        synthetic_speech_HFregeneration(i+1-0.01*Fs:i+0.01*Fs) = speech_HF(0.005*Fs+1:0.025*Fs);


    end %for

    %Making the output for both methods
    x_algorithm = synthetic_speech_HFregeneration;
    x_upsample = synthetic_speech_upsampling;

    %Computing the Signal to noise - ratio for both methods and printing to screen
    r_a = snr(x_algorithm,speech'-x_algorithm);
    r_u = snr(x_upsample,speech'-x_upsample);
    fprintf('The SNR for upsampled method: %g \n The SNR for algorithm method: %g \n',r_u,r_a);

end %function
```

hammingWindow.m

```
% Computing a Hamming window of length L given
% the formula for this window type: ham(n) = 0.54-0.46*cos(2*pi*(n/N))
% [Source: http://se.mathworks.com/help/signal/ref/hamming.html]
function ham = hammingWindow(L)
    n = 0:1:L-1;
    N = L-1;
    w = 0.54-0.46*cos(2*pi*(n/N));
    ham = w';
end %function
```

autocorr.m

```
% Auto correlation function which is easier to implement in C
% Finds the auto correlation with the signal energy r(0) as starting point
function r = autocorr(x)
    x = x';
    N = length(x);
    x = [x,zeros(1,N+1)];
    % Stores the auto correlation values in C in the iterations
    C = zeros(1,N);

    % Compute the auto correlation using the formula definiton of correlation
    for k = 1:N
        for n = 1:N
            C(k) = C(k)+ x(n)*x(n+k-1);
        end % for n
    end % for k

    % Copies the wanted area of C to the auto correlation output
    r = C(1:N);
    r = r';

end %function
```

findPitchAndVoice.m

```matlab
% Takes in a short signal x, with a given sampling frequency Fs
% Finds the pitch periode (pP) given as the number of samples and the ratio (vR)
% between the closest "harmonic" and r_x(0)
% The ratio indecates if the signal is voiced or not
function [pP, vR] = findPitchAndVoice(x, Fs)
    %Finds the auto correlation of x
    r_x = autocorr(x);

    % Define the pitch properties for range and frame
    pitchRange = floor(0.002*Fs):floor(0.02*Fs);
    pitchFrame = r_x(pitchRange);

    % Finds where the frame is less then zeros or sets minima as the length of pitchFrame
    % if no minima is found
    minima = find(pitchFrame <= 0);
    if isempty(minima)
        minima = length(pitchFrame);
    end %if

    % Finds the pitch positions
    pitchPos = find(pitchFrame(minima(1):end) == max(pitchFrame(minima(1):end)));

    % Sets the value of output values by computed data
    pP = pitchPos + minima(1) + pitchRange(1) - 3;
    vR = r_x(pP+1)/r_x(1);

end %function
```

LevinsonDurbin.m

```matlab
% The function LevinsonDurbin takes the auto correlation of a signal and the linear prediciton order
% as input, and returns an array of the different coefficients for the wanted value of P
function A = LevinsonDurbin(r,p)

    %Initializes error power stored in r(0) (Matlab 1-indexed)
    E = r(1);

    %The array is to be used to produce A and is initalized and the first value of b is 1
    b = [];
    b(1) = 1;

    for i = 1:p
        %Calculating the i'th reflection coefficient
        k(i) = r(i+1);

        %Initializes a every iteration and manually sets value of a(1) = 1
        a = zeros(i+1,1);
        a(1) = 1;

        %Computing the i'th order predicitor coefficients
        for j = 2:i
            k(i) = k(i)+b(j)*r(i-j+2);
        end %for j

        %Adjusting k by the prediction error power
        k(i) = -k(i)/E;
        %Computing a for found b and k
        for j = 2:i
            a(j) = b(j)+k(i)*b(i-j+2);
        end %for j

        a(i+1) = k(i);
        %Prediction error power for i'th power
        E = (1-abs(k(i))^2)*E;
        %Copying the values of a in to the temporary array b
        b = a;

    end %for i

    %Setting a to the output array A
    A = a';

end %function
```

testVocoder.m - testing function

```
% testVocoder is a test function for basicVocoder for different values of P
% to be used to answer the group specific task.
% The function plays the output of basicVocoder for randomly picked P
% to make the review of quality be as precise as possible
function testVocoder()

    %Different P-values to be tested
    P = [2 4 6 8 10 20];

    Fs = 16000; %Sampling frequency
    %First playing the original signal
    fprintf('Play of original\n');
    soundsc(basicVocoder('anvsb1.wav',14),Fs);
    pause();
    while ~isempty(P)
        % Randomly choosen value of P to be played
        k = ceil(rand*length(P));

        % Find output for a random P
        x = basicVocoder('anvsb1.wav',P(k));

        % Play of output for choosen P followed by a pause
        fprintf('Output for P = %d is now being played \n',P(k));
        soundsc(x,Fs);

        pause();
        % Plot of the signal is also presented to be used in review
        figure;
        text = sprintf('Plot for P = %d',P(k));
        plot(x),title(text);

        % Discard P-values that have been tested
        P(k) = [];

    end %while
end %function
```

testRELP.m - testing functions

```
% testRELP is a test function for RELPcoder for different values of P
% to be used to answer the group specific task.
% The function plays the output of RELPcoder for randomly picked P
% to make the review of quality be as precise as possible
function testRELP()

    %Different P-values to be tested
    P = [2 4 6 8 10 20];

    Fs = 16000; %Sampling frequency

    %First playing the original signal
    fprintf('Play of original\n');
    [x_alg x_up] = RELPcoder('anvsb1.wav',14);
    soundsc(x_alg,Fs);
    pause();
    while ~isempty(P)
        %Randomly choosen value of P to be played
        k = ceil(rand*length(P));

        %Find output for a random P
        [x_alg x_up] = RELPcoder('anvsb1.wav',P(k));

        %Play of output for choosen P followed by a pause
        fprintf('Output for P = %d is now being played \n',P(k));
        soundsc(x_alg,Fs);
        pause();
        %Plot of the signal is also presented to be used in review
        figure;
        text = sprintf('Plot for P = %d',P(k));
        plot(x_alg),title(text);

        %Discard P-values that have been tested
        P(k) = [];

    end %while
end %function
```

A.2 C code

main.c

```
// Signal processing systems for speech analysis
// This program requires extra arguments to run, the syntax is:
// $./<prog> <path_keyword> <method>
// <method> defaults to basic vocoder, but can be changed by "relpUp" or "relpHF"
// <method> is also the name of the output-file
// Example: $./test current relpUp
// There is no Makefile, but this project can be compiled by running:
// $gcc -o test main.c RELP.c basicVocoder.c SignalProcessing.c -lm
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include "basicVocoder.h"
#include "RELP.h"
#include "SignalProcessing.h"


int main(int argc, const char * argv[]) {
    //short* soundData;
    //short* outputData;

    typedef unsigned char BYTE;
    typedef unsigned int DWORD;
    char* file;
    char* outFile;
    if (argc == 1){
        printf("Mangler argument\n");
        return 0;
    }

    // Add file locations for source and target file
    if (strcmp(argv[1], "default") == 0){ // Reads "anvsb1.wav" from current folder
        char fileDir[1024];
        getcwd(fileDir, 1024);
        file = malloc(strlen(fileDir) + 20);
        outFile = malloc(strlen(fileDir) + 20);
        strcpy(file, fileDir);
        strcpy(outFile, fileDir);
        strcat(file, "/anvsb1.wav");
        strcat(outFile, "/basic.wav");
    }if (strcmp(argv[1], "new") == 0){ // Select a file from current folder to read
        char fileDir[1024];
        char input[100];
        getcwd(fileDir, 1024);
        file = malloc(strlen(fileDir) + 20);
        outFile = malloc(strlen(fileDir) + 20);
        strcpy(file, fileDir);
        strcpy(outFile, fileDir);
        printf("Enter name of source file: ");
        scanf("%s", input);
        strcat(file, "/");
        strcat(outFile, "/");
        strcat(file, input);
        if (argc == 3){
            strcat(outFile, argv[2]);
            strcat(outFile, ".wav");
        }else{
            strcat(outFile, "basic.wav");
        }
        printf("%s\n", outFile);
    }else{
        printf("Feil argument\n");
        printf("Husk argument\n");
        return 0;
    }
    // Declaring variables for header-elements
    FILE *soundFile;
    BYTE id[4],id2[4],id3[4],data[4];
    DWORD size,formatSize,sampleRate,bytesPerSec,dataSize;
    short format_tag,channels,block_align,bitsPerSample;

    //Reading the .wav-file, including all header elements.
    if((soundFile = fopen(file,"rb"))== NULL){
        perror("Failed to open file for reading");
    }
    fread(&id,sizeof(BYTE),4,soundFile);
    fread(&size,sizeof(DWORD),1,soundFile);
    fread(&id2,sizeof(BYTE),4,soundFile);
    fread(&id3,sizeof(BYTE),4,soundFile);
    fread(&formatSize,sizeof(DWORD),1,soundFile);
    fread(&format_tag,sizeof(short),1,soundFile);
    fread(&channels,sizeof(short),1,soundFile);
    fread(&sampleRate,sizeof(DWORD),1,soundFile);
    fread(&bytesPerSec,sizeof(DWORD),1,soundFile);
    fread(&block_align,sizeof(short),1,soundFile);
    fread(&bitsPerSample,sizeof(short),1,soundFile);
    fread(&data,sizeof(BYTE),4,soundFile);
    fread(&dataSize,sizeof(DWORD),1,soundFile);

    short* soundData =  calloc (dataSize/2,sizeof(short));
    //short* outputData =  calloc(dataSize/2, sizeof(short));
    fread(soundData,sizeof(short),dataSize/2,soundFile);
    if (fclose(soundFile) != 0){
        perror("Failed to close file\n");
    }

    //Reading the .wav file.
    int wav_length = dataSize/2;
    float y[wav_length];

    // Adjusting gain for float and converting to float format
    int i;
    float gainDown = 1.0/32760.0;
    for (i=1;i<wav_length;i++){
        y[i] = (float) soundData[i]*gainDown;
    }
    float output[wav_length];
```

```
    //Choosing the correct encoding
    if (argc < 3){  // Default to basic vocoder
        printf("Using basic vocoder.\n");
        basicVocoder(y,output,dataSize/2,14);
    }else if(strcmp(argv[2], "relpUp") == 0){
        printf("Using RELP coder with upsampling.\n");
        RELPcoder(y,output,dataSize/2,14,0);
    }else if(strcmp(argv[2], "relpHF") == 0){
        printf("Using RELP coder with HF regeneration.\n");
        RELPcoder(y,output,dataSize/2,14,1);
    }else{
        printf("Using basic vocoder.\n");
        basicVocoder(y,output,dataSize/2,14);
    }

    // Adjusting gain and converting to integer
    float gainUp = 32760.0;
    float tmp;
    for(i=0;i<wav_length;i++){
        tmp = output[i]*gainUp;
        soundData[i] = (short) tmp;
    }

    //Writing the .wav file including header
    soundFile = fopen(outFile,"wb");
    if (soundFile == NULL){
        perror("Failed to open file to be written to");
    }
    fwrite(&id,sizeof(BYTE),4,soundFile);
    fwrite(&size,sizeof(DWORD),1,soundFile);
    fwrite(&id2,sizeof(BYTE),4,soundFile);
    fwrite(&id3,sizeof(BYTE),4,soundFile);
    fwrite(&formatSize,sizeof(DWORD),1,soundFile);
    fwrite(&format_tag,sizeof(short),1,soundFile);
    fwrite(&channels,sizeof(short),1,soundFile);
    fwrite(&sampleRate,sizeof(DWORD),1,soundFile);
    fwrite(&bytesPerSec,sizeof(DWORD),1,soundFile);
    fwrite(&block_align,sizeof(short),1,soundFile);
    fwrite(&bitsPerSample,sizeof(short),1,soundFile);
    fwrite(&data,sizeof(BYTE),4,soundFile);
    fwrite(&dataSize,sizeof(DWORD),1,soundFile);
    fwrite(soundData,sizeof(short),dataSize/2,soundFile);



    fclose(soundFile);
    //Writing the .wav file
    return 0;
}
```

RELP.c

```c
//
//  RELP.c
//  Speech Analysis C-code
//

#include "RELP.h"
//Including the SignalProcessing.h to use the function defined in SignalProcessing.c
#include "SignalProcessing.h"

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



void RELPcoder(float* data, float* output,int length_data,int P, int choice){

    //Define the parameter values to be used in the processing
    int filterOrden = 8;
    int Fs = 16000;
    int D = 4;
    int low_P = 4;
    int step = 0.02*Fs;
    int halfStep = step/2;
    int speechLength = 0.03*Fs;
    int start = 0.005*Fs;
    float B[1] = {1};

    //Allocating space for all the arrays that is needed in the computation
    //Using calloc sets the value of every array position to zero
    float* syntheticError = (float*) calloc(length_data,sizeof(float));
    float* syntheticSpeechUpsampled = (float*) calloc(length_data,sizeof(float));
    float* syntheticSpeechHF = (float*) calloc(length_data,sizeof(float));
    float* temp1 = (float*) calloc(length_data,sizeof(float));
    float* windowSpeech = (float*) calloc(speechLength,sizeof(float));
    float* dataDecimated = (float*) calloc(speechLength/D,sizeof(float));
    float* speechFilt = (float*) calloc(speechLength,sizeof(float));
    float* speechEst = (float*) calloc(speechLength,sizeof(float));
    float* speechError = (float*) calloc(speechLength,sizeof(float));
    float* speechErrorFilt = (float*) calloc(speechLength,sizeof(float));
    float* ryLF = (float*) calloc(speechLength,sizeof(float));
    float* ryHF = (float*) calloc(speechLength,sizeof(float));
    float* ry = (float*) calloc(speechLength,sizeof(float));
    float* A = (float*) calloc(P+1,sizeof(float));
    float* A_low = (float*) calloc(low_P+1,sizeof(float));
    float* HFexcitation = (float*) calloc(speechLength,sizeof(float));
    float* HFexcitationFilt = (float*) calloc(speechLength,sizeof(float));
    float* HFexcitationLP = (float*) calloc(speechLength,sizeof(float));
    float* LFexcitation = (float*) calloc(speechLength,sizeof(float));
    float* syntheticFullbandResidual = (float*) calloc(speechLength,sizeof(float));
    float* speechUpsampling = (float*) calloc(speechLength,sizeof(float));
    float* speechAlgorithm = (float*) calloc(speechLength,sizeof(float));
```

```
//Defining the filter coefficients for both the lowpass- and highpassfilter
//The coefficients has been computed in MATLAB
float coeffLow[9] = {0, -0.0277, 0, 0.274,0.4974, 0.274, 0, -0.0227, 0};
float coeffHigh[9] = {0, -0.0161, -0.086, -0.1948, 0.7501, -0.1948, -0.0860, -0.0161, 0};

//Defining variables that will be changed in the processing
float gainLF, gainHF, gain;
int i,j,filtering;
float tmp;

//Making the Hamming window using the function in SignalProcessing.h
hammingWindow(windowSpeech,speechLength);

for (i=0.03*Fs; i<length_data-0.025*Fs; i+= step) {
    int lastSpeech = i+1-0.015*Fs;
    int nextSpeech = i+0.015*Fs;

    // Select a frame from the signal using Hamming window
    for (j = 0; j<nextSpeech-lastSpeech; j++) {
        tmp = windowSpeech[j]*data[lastSpeech+j];
        speechFilt[j]= tmp;
    }

    // Finding filter coefficients using LPC
    autocorr(speechFilt, speechLength,ry);
    LevinsonDurbin(ry, A, P);

    filtering = filtrate(speechFilt,speechLength ,B, 1, A, P, speechEst);
    if(filtering == -1){
        printf("Filtrate function failed: Size of A smaller than 1\n");
        return;
    }
    // Find prediction error
    for (j=0; j< speechLength; j++) {
        speechError[j] = speechFilt[j]-speechEst[j];
    }
    for(j= 0; j< step;j++){
        syntheticError[i+j-halfStep] = speechError[j+start];
    }

    // Filtrate speechError
    firFilter(coeffLow, filterOrden,speechError,speechErrorFilt,speechLength);

    // Decimate speechError
    decimate(speechErrorFilt, dataDecimated, speechLength, D);

    // Upsample speechError
    upsample(dataDecimated, speechError, speechLength, D);

    // Low-pass filter the upsampled signal
    firFilter(coeffLow,filterOrden,speechError,LFexcitation,speechLength);

    for (j = 0; j < speechLength; j++){
        HFexcitation[j] = LFexcitation[j];
        if (HFexcitation[j] < 0){
            HFexcitation[j] = 0;
        }
    }
    autocorr(HFexcitation,speechLength,ryHF);
    autocorr(LFexcitation,speechLength,ryLF);

    // Filter the high frequency components using filter coefficients from a low order LPC
    LevinsonDurbin(ry,A_low,low_P);
    filtering = filtrate(HFexcitation,speechLength,B,1,A_low,low_P,HFexcitationLP);
    if(filtering == -1){
        printf("Filtrate function failed: Size of A smaller than 1\n");
        return;
    }

    // High-pass filter the HF signal
    firFilter(coeffHigh,filterOrden,HFexcitationLP,HFexcitationFilt,speechLength);

    // Adjust gain before adding the LF and HF components
    gainHF = 0;
    gainLF = 0;
    for (j = 0; j < speechLength; j++){
        if(ryHF[j]> gainHF){
            gainHF = ryHF[j];
        }
        if (ryLF[j] > gainLF){
            gainLF = ryLF[j];
        }
    }
    gain = gainLF/gainHF;

    // Add LF and HF components
    for (j = 0; j < speechLength; j++){
        HFexcitationFilt[j] = gain*HFexcitationFilt[j];
        syntheticFullbandResidual[j] = LFexcitation[j] + HFexcitationFilt[j];
    }

    // Filter the upsampled signal using the LPC coefficients to find RELP with upsampling
    filtering = filtrate(speechError,speechLength,B,1,A,P,speechUpsampling);
    if(filtering == -1){
        printf("Filtrate function failed: Size of A smaller than 1\n");
        return;
    }

    // Filter the signal obtained through HF regeneration using the LPC coefficients from the fi
    filtering = filtrate(syntheticFullbandResidual,speechLength,B,1,A,P,speechAlgorithm);
    if(filtering == -1){
        printf("Filtrate function failed: Size of A smaller than 1\n");
        return;
    }

    for (j = 0; j< speechLength; j++){
        syntheticSpeechUpsampled[i+j-halfStep] = speechUpsampling[j+start];
        syntheticSpeechHF[i+j-halfStep] = speechAlgorithm[j+start];
    }
}

// choice == 0: Use RELP with upsampling
// choice == 1: Use RELP with HF regeneration
switch (choice){
        case 0:
            for (i = 0; i < length_data; i++){ output[i] = syntheticSpeechUpsampled[i];}
            break;
        case 1:
            for (i= 0; i<length_data; i++){ output[i] = syntheticSpeechHF[i];}
            break;
    }

    // Attempt at making the speech intelligible, amplify everything so average amplitude becomes ma
    // Clip everything else.
    float maxVal = 0;
    float sum = 0;
    for (i = 0; i < length_data; i++){
        sum += fabsf(output[i])/(float)length_data;
        if (fabsf(output[i]) > maxVal){
            maxVal = fabsf(output[i]);

        }
    }
    for (i = 0; i < length_data; i++){
        if(fabsf(output[i]) >= sum){
            output[i] = output[i] / fabsf(output[i]);// maxVal;
        }else{
            output[i] = output[i]/sum;
        }
    }

    // Free dynamically allocated arrays
    free(syntheticError),free(syntheticSpeechUpsampled),free(syntheticSpeechHF),free(windowSpeech);
    free(dataDecimated), free(speechFilt), free(speechEst),free(speechError),free(speechErrorFilt);
    free(ryLF),free(ryHF),free(A),free(A_low),free(HFexcitation),free(HFexcitationFilt);
    free(HFexcitationLP),free(LFexcitation),free(syntheticFullbandResidual),free(speechUpsampling);
    free(speechAlgorithm);
}
```

basicVocoder.c

```
// Takes an audio-input as the array 'data' of length 'length_data', analyzes and regenerates
// the signal based on the signal properties using linear predictive coding (LPC) with 'P'
// filter coefficients and stores the regenerated signal in 'output'

#include "basicVocoder.h"
#include "SignalProcessing.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>


void basicVocoder(float* data,float* output,int length_data, int P){
    srand(time(NULL));                  // Set seed for the random-function used to generate white-n
    //Defining constant variables
    const int Fs = 16000;               // Sampling frequency
    const int N = 8;                    // Filter order for FIR low-pass filter
    const int step = 0.02*Fs;
    const int halfStep = step*0.5;
    const int speechLength = 0.03*Fs;
    const float alpha = 0.5;            // Threshold for voiced or unvoiced
    const float beta = 0.1;             // Gain for white-noise
    const int pitchLength = 0.05*Fs;
    const int end = length_data-0.025*Fs;

    //Defining changing variables
    int lastPulse = 1;
    int lastSpeech, nextSpeech, lastPitch, nextPitch,filtering;
    int i,j,k,l,m;
    //Defining array that will be processed
    float* pitch = (float*) calloc(length_data,sizeof(float));
    float* noise = (float*) calloc(length_data,sizeof(float));
    float* vocoderInput = (float*) calloc(length_data,sizeof(float));
    float* synthezised = (float*) calloc(length_data,sizeof(float));
    float* pitchProperties = (float*) calloc(2,sizeof(float));
    float* randNoise = (float*) calloc(step,sizeof(float));
    float* windowSpeech = (float*) calloc(speechLength,sizeof(float));
    float* windowPitch = (float*) calloc(pitchLength,sizeof(float));
    float* vocoderInputSample = (float*) calloc(step,sizeof(float));
    float* vocoderInputSampleFilt = (float*) calloc(step,sizeof(float));
    float* yFiltrated = (float*) calloc(speechLength,sizeof(float));
    float* yPitch = (float*) calloc(pitchLength,sizeof(float));
    float* ry = (float*) calloc(speechLength,sizeof(float));
    float* A = (float*) calloc(P+1,sizeof(float));
    float B[1] = {1};
    //Making the Hamming Windows for speech and pitch
    hammingWindow(windowSpeech,speechLength);
    hammingWindow(windowPitch,pitchLength);

    // Low-pass filter coefficients, calculated in MATLAB
    float lowCoeff[9] = {0, -0.0277, 0, 0.274,0.4974, 0.274, 0, -0.0227, 0};

    // Iterating through the signal, analyzing and regenerating the signal 20ms at a time.
    //'i' marks the middle of the 20ms frame.
    for (i = Fs*0.03; i<end; i= i+step) {
        //Set boundaries for the signal segments
        lastSpeech = i+1-0.015*Fs;
        nextSpeech = i+0.015*Fs;
        lastPitch = i+1-0.025*Fs;
        nextPitch = i+0.025*Fs;

        // Shape the relevant signal segments using Hamming-windows
        for (j = 0; j<nextSpeech-lastSpeech; j++) {
            yFiltrated[j] = windowSpeech[j]*data[lastSpeech+j];
        }
        for (k = 0; k<nextPitch-lastPitch; k++) {
            yPitch[k] = windowPitch[k]*data[lastPitch+k];
        }

        // Find the auto-correlation of the signal segment.
        autocorr(yFiltrated,speechLength,ry);
        // Levinson-Durbin recursion is used on the auto-correlation of the signal to find filter co
        LevinsonDurbin(ry,A,P);
        // Find the period of the pitch and a voiced constant (between 0 and 1, larger for voiced so
        // and store them as element 0 and 1 respectively in the array 'pitchProperties'
        findPitchAndVoice(yPitch,pitchLength,pitchProperties,Fs);

        // Generate a signal using either white noise or pulse trains depending on the voiced proper
        // of the signal segment.
        if (pitchProperties[1] >= alpha){
            // When the signal is voiced, make a pulse train of "dirac"-pulses with intervals
            // defined by the previously found pitch period.
            int pp = (int) (pitchProperties[0]);
            for (m = lastPulse; m<i+0.01*Fs; m = m+pp){
                pitch[m] = 1;
                lastPulse = m;
            }
        }else{
            // When the signal isn't voiced, white noise is used as a signal, scaled down with 'beta
            lastPulse = i+0.01*Fs;
            // Generate white-noise
            rand_gauss(randNoise,step);
            for (l = 0; l<step; l++) {
                noise[i-halfStep+l] = beta*randNoise[l];
            }
        }

        for (l = 0; l < step; l++){
            vocoderInput[i-halfStep+l] = pitch[i-halfStep+l] + noise[i-halfStep+l];
            // Adjusts numbers that are too large, in the rare case of white noise getting very larg
            if (vocoderInput[i-halfStep+l] > 1){
                vocoderInput[i-halfStep+l] = 1;
            }
            vocoderInputSample[l] = vocoderInput[i-halfStep+l];
        }

        // Filter the generated signal using the filter coefficients found by the Levinson-Durbin re
        filtering = filtrate(vocoderInputSample,step,B,1,A,P, vocoderInputSampleFilt);
        if(filtering == -1){
            printf("Filtrate function failed: Size of A is equal to or smaller then 1\n");
            return;
        }
        for (l = 0; l<step; l++) {
            synthezised[i-halfStep+l] = vocoderInputSampleFilt[l];
        }
    }

    // Low-pass filter the generated signal to remove spurious high frequency elements.
    firFilter(lowCoeff,N,synthezised,output, length_data);

    // The signal should have values from -1 to 1, the signal is adjusted to comply
    // with this constraint
    float maxVal = 0;
    for (i = 0; i < length_data; i++){
        if (fabsf(output[i]) > maxVal){
            maxVal = fabsf(output[i]);
        }
    }
    for (i = 0; i < length_data; i++){
        output[i] = output[i] / maxVal;
    }
```

```
    //Free dynamically allocated memory
    free(yPitch),free(yFiltrated),free(synthezised),free(vocoderInputSample),free(vocoderInput),free
    free(pitch),free(pitchProperties),free(windowPitch),free(windowSpeech),free(ry),free(randNoise),
}
```

SignalProcessing.c - Code to supporting functions

```
//
//  Signal Processing.c
//  Speech Analysis C-code
//
//

#include "SignalProcessing.h"


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>



int rand_gauss (float *x, int N){
    /* Create Gaussian N(0,1) distributed numbers from uniformly distributed numbers using */
    /* the Box-Muller transform as described in
*/
    /* D. E. Knuth: The Art of Computer Programming, Vol II, p 104
*/
    float v1,v2,s;
    int i, j, M;

    M=N/2;

    // Initialize uniform number generator


    // Loop - each iteration generates two Gaussian numbers
    for (i=0; i<M; i++){
        j=2*i;

        // See Knuth or http://en.wikipedia.org/wiki/Box_Muller_transform
        // for algorithm description
        do {
            v1 = 2.0 * ((float) rand()/RAND_MAX) - 1;
            v2 = 2.0 * ((float) rand()/RAND_MAX) - 1;

            s = v1*v1 + v2*v2;
        } while ( s >= 1.0 );

        if (s == 0.0)
            i=i-1;
        else {
            x[j]=(v1*sqrt(-2.0 * log(s) / s));
            if (j+1<N)
```

```
                x[j+1]=(v2*sqrt(-2.0 * log(s) / s));
        }
    }
    return 0;
}

//Creates a hamming window on the empty array hamming in the input of length L
//[Source: http://se.mathworks.com/help/signal/ref/hamming.html]
void hammingWindow(float* hamming,int L){
    //Initialize the variables to be used in the for-loop
    int i;
    float x;
    int N = L-1;
    //Constant used in the formula for Hamming window
    float alpha = 0.54;
    float beta = 0.46;
    for (i = 0; i<=L-1; i++) {
        x = 2.0*3.14*(float)i;
        x = x/(float)N;
        //Making sure make the hamming[i] a float variable by casting
        //the cos-function from double to float
        hamming[i] = alpha - beta*(float)cos((double)x);
    }
}
//Creates the autocorrelation of the input x of length lengthx
//The output correlation is created on the float variable rx
void autocorr(float* x,int lengthx,float* rx){
    //Declare the array and counting variables to be used in the for-loops
    float y[2*lengthx+1];
    int i,j,k,l;
    //Copying values in x(n) to y(n) with length 2*lengthx+1
    //Filling with zeros outside the length of x(n)
    //This is done to ensure we do not getting accessing problems because
    //of indexing outside the array of x(n)
    for (i = 0; i<2*lengthx+1; i++) {
        if (i<lengthx) {
            y[i] = x[i];
        }else{
            y[i] = 0;
        }
    }
    //The values of non-normalized autocorrelation will be stored in C
    float *C = (float *) calloc(lengthx, sizeof(float));
    for (j=0; j<lengthx; j++) {
        for (k=0; k<lengthx; k++) {
            //Definiton of autocorrelation
            C[j] +=  y[k]*y[k+j];
        }
    }
    //Normalized autocorrelation stored in rx(n) where the biggest value
    //r(0) = 1
    for (l =0 ; l<lengthx; l++) {
        rx[l] = C[l]/C[0];
    }
    free(C);
}
//Creates LP-coefficients based on the autocorrelation
```

```
//The length of A is given by prediction order P
void LevinsonDurbin(float* r,float* A,int P){
    //Indexing variables and arrays
    float* b;
    float* k;
    b = (float*) calloc(P+1,sizeof(float));
    k = (float*) calloc(P+1,sizeof(float));
    float E = r[0];
    int i,j,l;
    memset(A,0,(P+1)*sizeof(float));
    //Set the value of first coefficient to 1, and copy it on b[0]
    A[0] = 1;
    b[0] = 1;
    for (i = 1; i<=P; i++) {
        for (j = 1; j<i; j++) {
            k[i] += A[j]*r[i-j];

        }
        //Adjusting k by the prediction error power

        k[i] = (r[i]-k[i])/E;
        A[i] = k[i];
        //Computing b for found A and k

        for (j=1; j<i; j++) {
            b[j] = A[j] - k[i]*A[i-j];
        }
        //Copying the values of the temporary array b in to A
        for (l = 1; l<i; l++) {
            A[l] = b[l];

        }
        //Prediction error power for i'th power
        E = (1- k[i]*k[i])*E;

    }
    //Inverting the coefficients to make it easier to used in filtrating
    for(i = 1; i<= P;i++){
        A[i]= -A[i];
    }
    free(b);
    free(k);
}

//Filtrates x based on the coefficients given in A and B, and returns the output in y
int filtrate(float* x,int lengthx,float* B,int sizeB,float* A,int sizeA,float* y){
    int i,j,k;
    sizeA++;
    memset(y,0,lengthx*sizeof(float));
    if (sizeA > 1){
        for (i = 0; i<lengthx; i++) {
            for (j = 1; j<sizeA; j++) {
                for (k= 0; k<sizeB; k++) {

                    if (k <= i){
                        y[i] += B[k]*x[i-k];
                    }
                    if (j <= i){
                        y[i] -= A[j]*y[i-j];
                    }
                }
            }
            y[i] = (1/A[0])*y[i];
        }
    }else{
        return -1;
    }
    return 0;
}


//Takes a signal x of length N, downsamples the signal with a factor D,
//and puts the output in xDec of length N/D.
void decimate(float *x, float *xDec, int N, int D){
    int i;
    int c = 0;
    for (i = 0; i < N; i += D){
        xDec[c] = x[i];
        c++;
    }
}

//Takes a signal xDec of length N/D, upsamples the signal with a factor D,
//and puts the output in x of length N.
void upsample(float *xDec, float *x, int N, int D){
    int i;
    int c = 0;
    for (i = 0; i < N; i++){
        if (i%D){
            x[i] = 0;
        }else{
            x[i] = xDec[c];
            c++;
        }
    }
}


//Takes in a set of Ncoeffs filter coefficents and filters the input x
//Output xFiltred is the filtered x

void firFilter (float *coeff, int Ncoeffs,
               float *x, float *xFiltred, int n)
{
  int i, j, k;
  float tmp;

  for (k = 0; k < n; k++)  //  position in output
  {
    tmp = 0;

    for (i = 0; i < Ncoeffs; i++)  //  position in coefficients array
    {
      j = k - i;  //  position in input
      if (j >= 0)  //  bounds check for input buffer
      {
        tmp += coeff[i] * x[j];
      }
    }

    xFiltred[k] = tmp;
  }
}

//Determines if a speech is voiced or unvoiced based on the autocorrelation
void findPitchAndVoice(float* y_pitch,int pitchLength,float* pitchProperties,int Fs){
    //Deciding the work area
    int N = floor(0.02*Fs)-floor(0.002*Fs);

    float ry[pitchLength];
    autocorr(y_pitch,pitchLength, ry);
    int i,j,minima = 0;
    float pitchFrame[N];
    int m = 0;
    //We will only use the parts of the autocorrelation which is in the work area
    //defined by N
    for (i = floor(0.002*Fs); i<floor(0.02*Fs); i++) {
        pitchFrame[m] = ry[i];
        m++;
    }
    //Finds where the frame is less then zeros or sets minima as the length of the pitchFrame
    //if no minima is found
    int foundMinima = 0;
    for (j = 0; j<N; j++) {
        if (pitchFrame[j] <= 0) {
            minima= j;
            foundMinima = 1;
            break;
        }
    }
    if(foundMinima == 0){
        minima = N;
    }
    int k,pitchPos = 0;
    float max = 0.0;
    //Finds the pitch positions
    for (k = minima; k<N; k++) {
        if (max< pitchFrame[k]) {
            pitchPos = k;
            max = pitchFrame[k];
        }
        if ((max > ry[0]/2) && (pitchFrame[k] < 0)){
            break; //Stops after first peak/bang
        }
    }

    //Sets the value of output values by computed data
    int pitchPeriod = pitchPos + minima + 0.002*Fs - 3;
    float pitchRatio = ry[pitchPeriod+1]/ry[0];
    pitchProperties[0] = pitchPeriod;
    pitchProperties[1] = pitchRatio;

}
```

RELP.h - header

```
//  RELP.h
//  Speech Analysis C-code
//

#ifndef __Speech_Analysis_C_code__RELP__
#define __Speech_Analysis_C_code__RELP__

#include <stdio.h>
void RELPcoder(float *data, float *output, int length_data, int P,int choice);
#endif /* defined(__Speech_Analysis_C_code__RELP__) */
```

basicVocoder.h - header

```
//  basicVocoder.h
//  Speech Analysis C-code
//


#ifndef __Speech_Analysis_C_code__basicVocoder__
#define __Speech_Analysis_C_code__basicVocoder__

#include <stdio.h>

void basicVocoder(float* data,float* output,int length_data,int P);


#endif /* defined(__Speech_Analysis_C_code__basicVocoder__) */
```

SignalProcessing.h - header

```
//
//  Signal Processing.h
//  Speech Analysis C-code
//


#ifndef __Speech_Analysis_C_code__Signal_Processing__
#define __Speech_Analysis_C_code__Signal_Processing__

#include <stdio.h>

int filtrate(float* x,int lengthx, float* B,int sizeB,float* A,int sizeA,float* y);

void LevinsonDurbin(float* r,float* A,int P);

int rand_gauss (float *x, int N);
```

```c
void hammingWindow(float* hamming,int L);

void autocorr(float* x,int lengthx,float* rx);

void decimate(float *x, float *xDec, int N, int D);

void upsample(float *xDec, float *x, int N, int D);

void firFilter(float* coeff,int Ncoeffs, float* x, float* xFiltred, int n);

void findPitchAndVoice(float* y_pitch,int pitchLength,float* pitchProperties,int Fs);

#endif /* defined(__Speech_Analysis_C_code__Signal_Processing__) */
```