1 Introduction

For this project we were required to design a method for representing 16KHz speech waveforms at a rate of 1800 parameters per second. A number of possible methods were considered. An obvious simple solution would be to lowpass filter the speech signal to meet the 1800 parameters per second requirement. This would reduce the high frequency content in the speech but would still retain frequencies below 900Hz which would still provide intelligible speech. While this would provide a solution, it seems to be a cheap way out.

As a result, we also consider a number of other possibilities. These included adaptive predictive coding, adaptive transform coding, sub-band coding using adaptive bit allocation, sub-band adaptive predictive coding, and vector quantization. It was at this point that we realized that we needed to set some design objective in conjunction with picking a compression approach. Motivated by the generally warm, fuzzy feeling from *Linear Predictive Coding* (LPC) in the third project, we set the following design goal:

> DEVELOP A SPEECH COMPRESSION TECHNIQUE THAT PRODUCES REASONABLY INTELLIGIBLE MALE SPEECH WITH AS FEW PARAMETERS PER SECOND AS POSSIBLE.[1]

[1]We limited ourselves to male speech since all of our training/testing speech was spoken by male speakers.

2 Design Process

Throughout this section we use the “sun” sound bite from the first project to help illustrate our motivation for various design decisions. We resampled the speech signal at 16KHz in order to ensure an optimal match with the LPC codebook that we assume was trained on 16KHz speech data. Figure 1 shows the original “sun” signal.

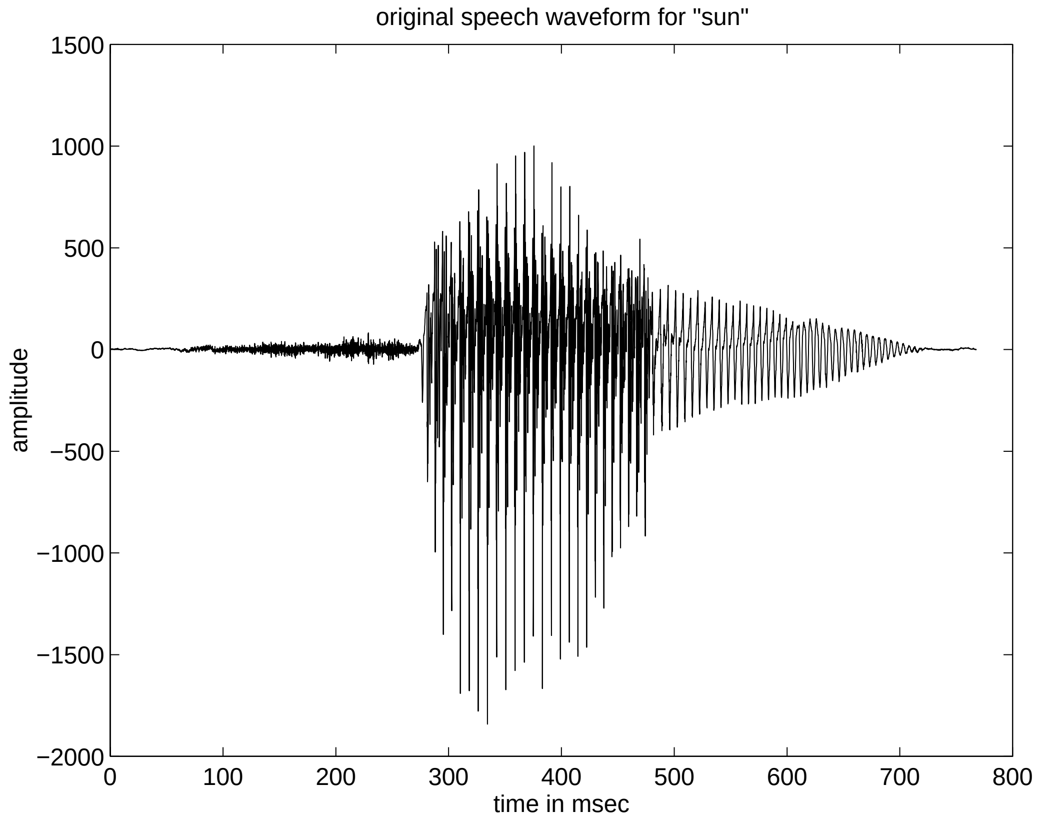

Figure 1: Original speech waveform for “sun”

2.1 Vocal Tract

Our first design decision (other than choosing our design goal) found early and unanimous agreement. We settled on using LPC to model the vocal tract. Furthermore, we restricted our LPC model to a twenty pole filter characterizing 30 msec speech frames. This restriction allowed us to take advantage of the previously trained *Vector Quantization* (VQ) codebooks that we used in the third project. At this point the vocal tract model was fixed as VQ on LPC coefficients of non-overlapping, Hamming windowed, 30 msec speech frames. As in the third project, we used the Euclidean distance metric on the cepstral coefficients to select the ~~apropriate~~ appropriate codeword from the "all_males" VQ codebook.

The remainder of the design process involved modeling the error signal.

2.2 Excitation

We model the error signal generated by the LPC vocal tract analysis as the excitation component of the speech waveform. We will use "excitation signal" and "error signal" interchangeably. A wide variety of excitation models exist in the literature. In this section we will describe a number of approaches that we considered. We will also describe some of the results for the ones we actually implemented.

On the extreme ends lie two options. One option is to ignore the excitation and just use the vocal tract information to reconstruct the signal. We call this approach *complete ignorance*. This approach is appealing in that allows our compression scheme to achieve a parameter rate of just over 33 parameters per second. While the compression rate is extremely good, the quality of the speech (as perceived by a human) is rather low. In fact, the output signal is identically zero. This occurs because the LPC coefficients are weighted by the zeros in the error signal. At the other extreme is a method to model the excitation with all 1800 per second of the available model parameters. This could be done in a way similar to was described above where the compression operation only involved lowpass filtering. Here we model the excitation signal by lowpass filtering the error signal from the LPC modeling to a rate that requires $1800 - 34 = 1766$ parameters. This results in a sampling rate for the excitation signal that is just under 900 Hz. While much of the a frequency content is lost, the key component (the pitch frequency) is retained. Although this approach holds promise for producing high quality speech, we did not implement it because it would not meet our design goal.

Since the *complete ignorance* approach aligned more closely with our design goal, we return to it to try to salvage it by introducing some modifications. With this return come a number of methods. Methods that we call *serious ignorance*, *moderate ignorance*, and a family of methods labeled *mild ignorance*.

Serious ignorance involves one slight modification to the *complete ignorance* method. Instead of completely ignoring the excitation signal, in this approach we calculate the standard deviation of the excitation signal over the entire speech segment. This increases the parameter rate only slightly. Assuming a speech segment of two seconds results in a parameter rate under 34 parameters per second. When reconstructing the signal, we generate white noise with the calculated standard deviation and use it at the excitation signal. The *moderate ignorance* approach is very similar to this except that we now calculate the standard deviation over each frame. This results in a parameter rate of 67 parameters per second. Both of these approaches are founded on the premise that the LPC modeling is a whitening process and the resultant error signal (which we assume to be our excitation signal) is white noise. While this works well for unvoiced speech, it does not perform well for voiced speech. Even so, it is interesting to note that the resultant speech is significantly intelligible. This makes sense because we all know that whispered speech is significantly intelligible yet contains no voiced speech. In fact, the reconstructed speech using the *serious ignorance* method (see Figure 2) and the *moderate ignorance* method (see Figure 3) do sound much like whispered speech.

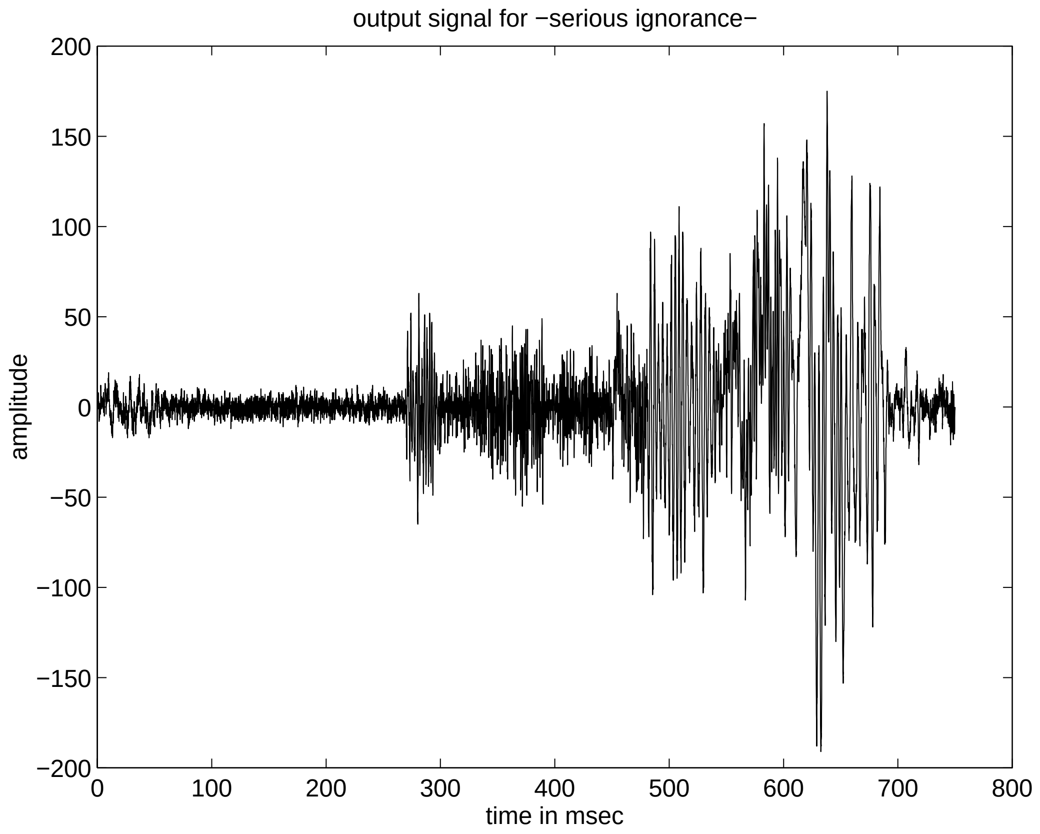

Figure 2: Output for “sun” using *serious ignorance*

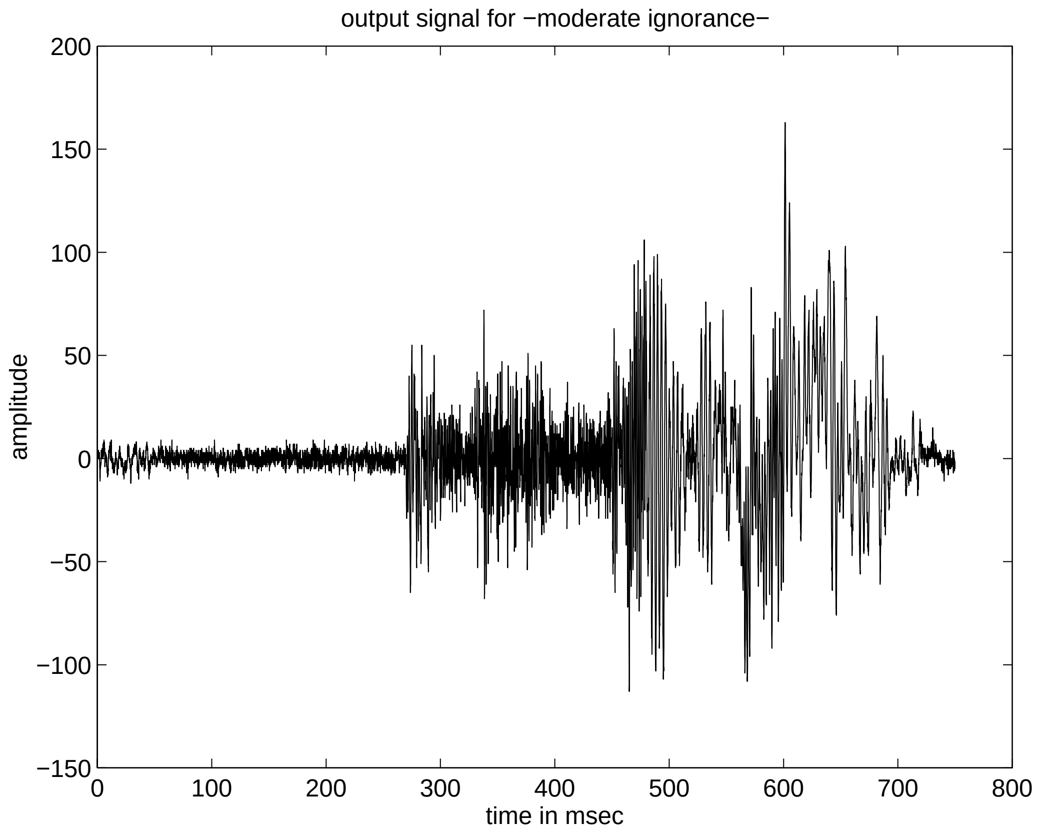

Figure 3: Output for "sun" using *moderate ignorance*

In both the *serious ignorance* and *moderate ignorance* approaches we assume that the entire speech segment is unvoiced. In nearly every case of speech, this assumption is invalid. In order to improve on the quality of the reconstructed speech we describe a family of speech compression techniques that do not assume that the entire speech segment to be unvoiced. In order to remove this assumption we need to perform two tasks – classify each frame as voiced or unvoiced and estimate the pitch period for voiced frames. A plethora of techniques have been developed for performing these tasks , and many variations can be had on each technique. We initially drew our ideas from Rabiner et al. (Rabiner et al. 1976).

A comma should not be placed before a conjunction (and/but) unless it separates two complete sentences. If the conjunction does separate two complete sentences, then a comma should be placed before the conjuction.

Among our pitch detection alternatives were cepstral analysis, autocorrelation methods (center clipping prior to autocorrelation calculation (CLIP) and autocorrelation performed on the LPC error signal (SIFT)), a slightly modified autocorrelation method called Average Magnitude Differences Function (AMDF) which subtracts instead of multiplying in the autocorrelation summation, and a parallel processing method based on an elaborate voting scheme. We immediately dismissed the parallel processing method due to its complexity and little promise of significantly superior performance. Based on our design objective we proposed to use the pitch detection algorithm that produced the most perceptually pleasing results. McGonegal (McGonegal 1977) reported that of these methods, AMDF offered the best results. At this point it is necessary for us to write a "weaselly" sentence or two to explain why we didn't actually do this. The bottom line is that a different group did this and we listened to their results and found that they weren't much different from ours using the cepstral analysis method.

While it is true that a number of methods exist for performing pitch detection, we chose to limit our implementational exploration to cepstral techniques. We did so because of the ease of implementation and intuitive attractiveness. We implemented the cepstral analysis as outlined in our second project. The cepstral coefficients are then used to determine whether the frame contains voiced or unvoiced speech. If the speech is determined to be voiced, an estimate of the pitch period is also obtained. By default our algorithm focuses on the cepstral coefficients representing the frequency range from 100 to 270 Hz.[2] Our algorithm calculates the mean value of nonnegative coefficients in this range. If the peak value is greater than 1.5 times that of the mean value, the speech segment is classified as voiced speech and the pitch period is set based on maximum valued coefficient and is stored as the first excitation modeling parameter. If the peak value is less than 1.5 times that of the mean value, the speech segment is classified as unvoiced speech, and the first excitation modeling parameter is set to zero. In either case, the standard deviation of the excitation signal is calculated and stored as the second excitation modeling parameter.

This processing results in two model parameters for each frame. While it would be possible to arbitrarily chose the frame size for the excitation modeling, for simplicity we chose to remain consistent with the frame length used in the vocal tract modeling, i.e., 30 msec. As a result, we have three parameters for every 30 msec frame or just under 100 parameters per second.

We reconstruct the excitation signal as follows. For an unvoiced frame the excitation signal is white noise with standard deviation equal to the second excitation parameter. For a voiced

[2] Due to the speaker dependent nature of the cepstral approach to pitch detection, we have included an input parameter to adjust this as needed.

frame we generate a periodic signal using the function

$$e_n = r_n + \frac{\alpha n}{1+\alpha n^2} \bmod \gamma$$

where r_n is white noise sequence with the same standard deviation as the excitation signal, α determines the steepness of the slope, and γ is the pitch period. This function provides a periodic excitation signal that retains a white noise component approximating that of the excitation signal. The vocal tract and excitation information are combined via:

$$s_n = e_n - \sum_{k=1}^{20} b_k s_{n-k}$$

where e_n is the excitation signal and b_k are the LPC codebook coefficients.

We performed cepstral analysis on the original signal (henceforth referred to as SCEP *mild ignorance*) and on the excitation signal (henceforth referred to as ECEP *mild ignorance*). The SCEP *mild ignorance* method provided useful results; however, the ECEP *mild ignorance* method is unable to detect voiced frames. Unfortunately, we did not have time to fully explore why this is happening. In any case, the analysis is the same for both methods. The only difference is the signal analyzed. Figure 4 presents the sound bite “sun” after processing by the cepstral analysis on the original signal.

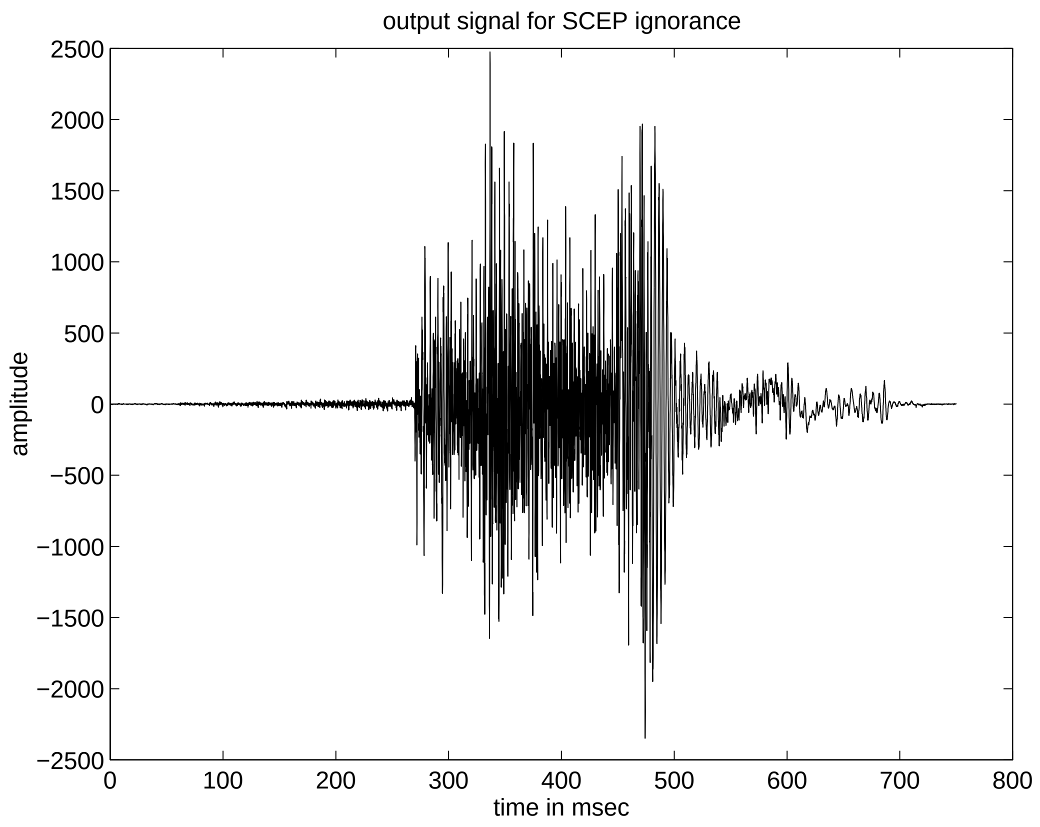

Figure 4: Output for “sun” using SCEP *mild ignorance*

While the plots thus far are instructive, plots of the excitation signal only provide a clearer view of the excitation signal modeling. These plots are included in Figures 5 – 7 for the original excitation signal, the excitation modeled by *moderate ignorance*, and SCEP *mild ignorance* respectively. It should be obvious that the SCEP *mild ignorance* approach provides a much better model for the excitation.

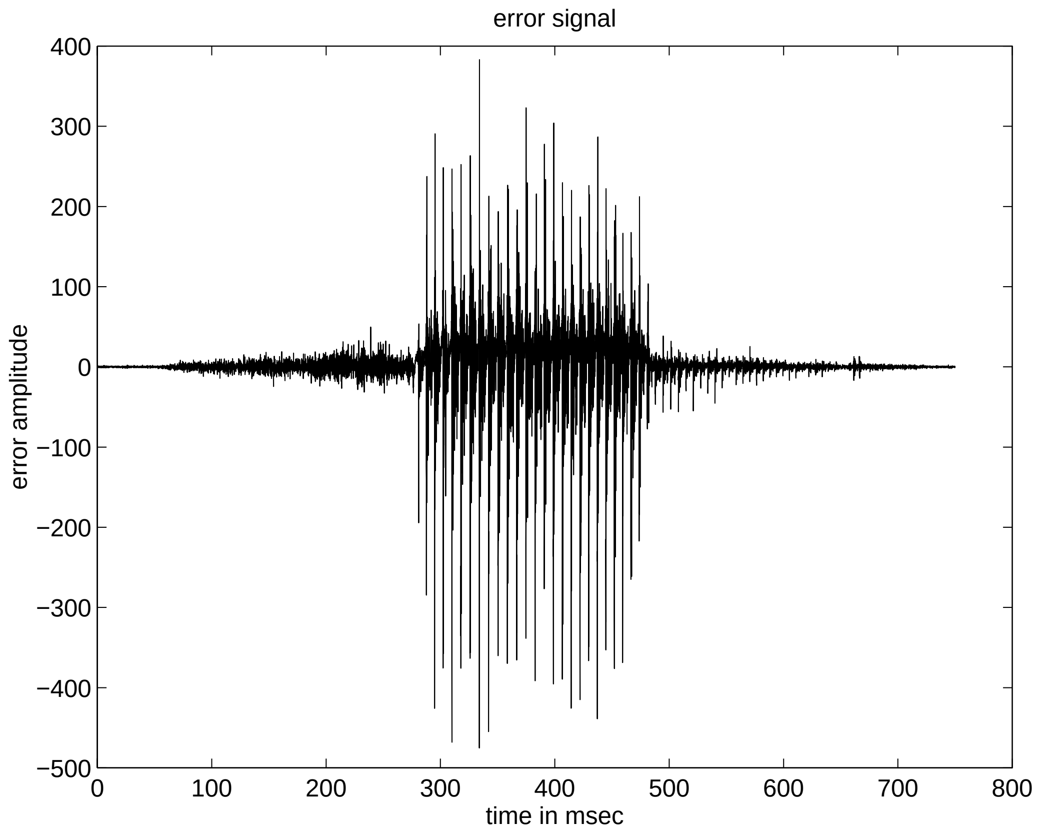

Figure 5: Original excitation for “sun”

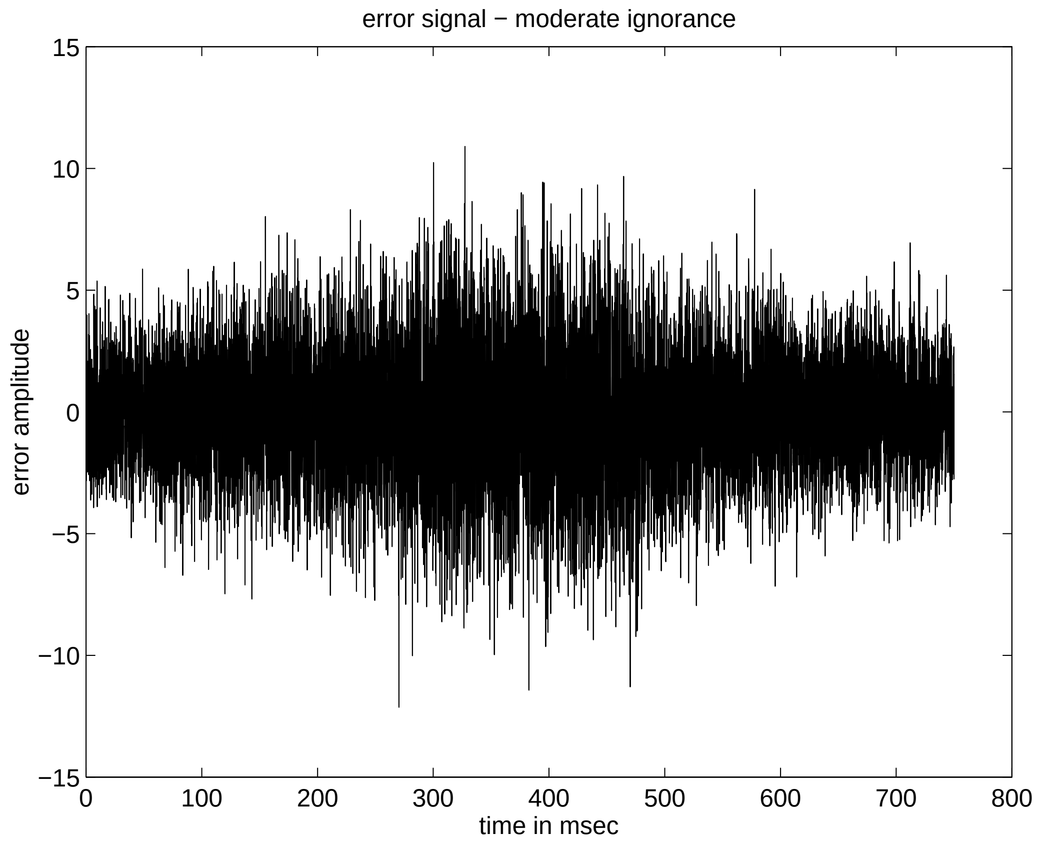

Figure 6: Excitation for “sun” using *moderate ignorance*

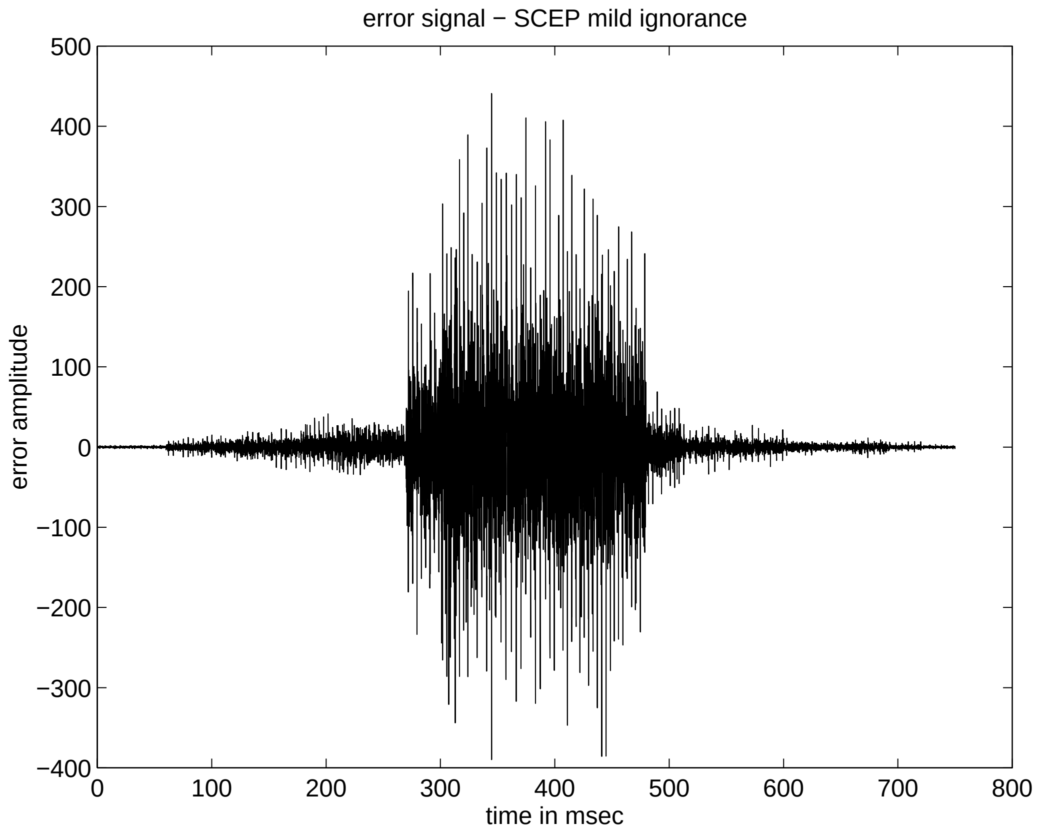

Figure 7: Excitation for “sun” using SCEP *mild ignorance*

3 Discussion

There exist a large number of reasonable approaches for reaching our design goal. We have considered a number of them and have actually implemented a subset of that number. Since our design goal was founded on intelligibility, we concluded that a quantitative evaluation to be of little use in assessing our ability to achieve our objective. Instead we relied on subjective assessments. Our assessments are rather imprecise and are aimed to provide a feel for our experiences as opposed to a definitive argument for a particular approach. Table 1 contains our estimates on the percentage of intelligible speech present for each speech signal for the two methods included in our final program.

There are five approaches that we evaluated — *complete ignorance*, *serious ignorance*, *moderate ignorance*, ECEP *mild ignorance*, and SCEP *mild ignorance*. As its name suggests, *complete ignorance* did not perform very well. The resulting speech waveform was often unintelligible. Although the standard deviation varied significantly from frame to frame, the difference between the *serious ignorance* and *moderate ignorance* intelligibility was not as pronounced as we had expected. Both approaches resulted in reasonably intelligible speech. One implication of these approaches is the lack of any voiced speech. This resulted in the impression that processed speech sounded as if it were being whispered. While this was a significant deviation from the original speech, it did not reduce the intelligibility significantly. It would seem that at this point we had met our design criteria. These approaches allow us to achieve compression rates of 34 and 67 parameters per second respectively while still maintaining reasonably intelligible speech. The two *mild ignorance* methods attempted to reduce the "whisper effect" by including voiced speech frames. These methods increased our parameter burden to 100 parameters per second (still well below the 1800 parameters per second that we were given to work with). The ECEP *mild ignorance* method failed to identify voiced speech. As a result, the output was the same as that of the *moderate ignorance* approach. While the SCEP *mild ignorance* approach was moderately successful in reducing the whisper quality of the speech, there were a few shortcomings. One significant disadvantage was that the threshold was somewhat speaker dependent. This shortcoming is most likely due to our choice of pitch detector. The cepstral pitch detection method is known for it's thresholding ambiguity, and it may be that we could elevate this problem by selecting a different pitch detection method like the AMDF. This could be done with a simple modification and the general compression framework would remain the same. Another disadvantage is that the transitions between voiced and unvoiced occasionally produces an audible artifact. It may be possible to incorporate some sort of transition smoothing to eliminate this; however, we did not explore this option.

	SCEP *mild ignorance*			*Moderate ignorance*		
Sentence number	Speaker number 1	2	3	Speaker number 1	2	3
1	80%	60%	50%	70%	20%	20%
2	60%	70%	70%	30%	50%	30%
3	70%	40%	100%	20%	20%	30%
4	70%	60%	90%	40%	20%	20%
5	80%	80%	90%	40%	10%	20%

Table 1: Percentage of intelligible speech

Our project guidelines made it clear that we were to not concern ourselves with the number of bits required to represent the speech; however, it may be of interest to note that our approach can be easily modified to squeeze the most information out of each bit as possible. We chose to use a 10 bit codebook for the LPC coefficients, but we certainly could have reduced this without much loss of intelligibility. A 6 bit codebook should suffice. As we saw in the comparison between the *serious ignorance* and *moderate ignorance* approaches, the standard deviation estimate is not very sensitive. For the sake of discussion we will assume that we can quantize this estimate to 4 bits. The remaining parameter contains information on the pitch period. We also use this parameter to indicate whether the speech frame contains voiced or unvoiced data. This is done by setting the pitch period equal to zero if the frame contains an unvoiced speech segment. This approach allows us to reserve one quantization level of the pitch period parameter as a flag for unvoiced speech. Because of the narrow range of possible pitch periods, we hypothesize that we can quantize this parameter to 4 bits. Table 2 indicates the parameter and bit rates using these quantization levels for the various approaches that we implemented.

Compression technique	Parameters per second	bit per second
complete ignorance	33.3	200
serious ignorance	$33.3 + 1$	$200 + 4$
moderate ignorance	66.6	667
ECEP *mild ignorance*	99.9	1400
SCEP *mild ignorance*	99.9	1400

Table 2: Compression rates

All of these bit rates could be reduced further by additional coding techniques. For example, the *mild ignorance* techniques could make good use of Huffman coding. It should be evident from Figure 7 that the voiced/unvoiced decision remains consistent for a few frames at a time. As a result, all neighboring unvoiced frames will share the same value for their pitch period parameter. If we store the LPC codebook parameter for all the frames first, then the pitch period parameter for all of the frames next, and then the standard deviation parameter last, the sequence of pitch period parameters should compress significantly whenever a sequence of unvoiced frames appear consecutively.

4 Additional Notes

The entire project was programmed in ‘C’ and the source code is attached at the end of this report. Also, the last page of the report (after the source code) is the “Project 4S Information Sheet.” Our executable code allows two modes of operation. The default mode processes using the SCEP *mild ignorance* method. Using the `+N` flag will cause the program to process the speech data using the *moderate ignorance* method instead. Please refer to the manpage included just prior to the source code, refer to the README file, or run the program with the `-help` option for more information on the command syntax. All of the files for our project can be found in `/home/offset/a/taylor/SpeechStuff`. Some files exist in each directory and the others are symbolically linked. Our program generates ascii speech files. In order to listen to the output converted it to binary speech files and then used a package called “sox” to convert the file to a Sun AU file, and used “audioplay” on the Suns and “send_sound” on the HPs to listen to the output.

5 Bibliography

- L.R. Rabiner, M.J. Cheng, A.E. Rosenberg, and C.A. McGonegal, “A Comparative Performance Study of Several Pitch Detection Algorithms,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-24, no. 5, pp. 399–418, 1976.
- C.A. McGonegal, “A Subjective Evaluation of Pitch Detection Methods Using LPC Synthesized Speech,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-25, no. 6, 1977.

6 Source Files

6.1 hw4.h

```
/*****************************************************************************
Authors: Varun Madhok and Chris Taylor
Date:    December 6, 1996
File:    hw4.h
Purpose: This header file contains the function prototypes for the
         speech compression application that was part of our
         fourth homework assignment for EE649 -- Speech Processing

Notes: The following subroutines have been copied (mostly) from the
       text 'Numerical Recipes in C' by Press, Teukolsky, Flannery
       and Vetterling.  The source code however has not been submitted.

(float *)vector      : allocates memory for a floating point array;
(double *)dvector    : allocates memory for an array with double elements;
(double *)c_dvector  : allocates memory for an array with double elements
                       with initialization to zero;
```

```
(int *) ivector      : allocates memory for an array with integer elements;
void free_vector     : frees memory allocated for a floating point array;
void free_ivector    : frees memory allocated for an integer point array;
void free_dvector    : frees memory allocated for a double point array;
void dfour1          : carries out FFT on input array. Original array is
                       replaced by the FFT thereof. To work with complex
                       data, the convention used is to assign all real
                       values to the even indices and the imaginary components
                       to the odd indices of the array (assuming first index
                       is zero);
void normal          : white noise generation subroutine with mean 0 and
                       variance 1.
**********************************************************************/

/* Definitions for constants in our simple program.  If this were more
   than an experimental application, these constants should be parameters
   whose values could be selected at runtime. */
#define DEF_DAT 7680
#define SEGMENT_LENGTH 480
#define IN_DEF_FILE "sun.ascii.Z"
#define OUT_DEF_FILE "out.temp"
#define CODE_DEF_DIR "male"
#define DEF_CODEBK_SIZE 2

#if defined(__STDC__) || defined(ANSI) || defined(NRANSI)
  /* fftmag: Calculates the magnitude of an n sample signal s and stores
             the result in mag */
  /* fftmag: Calculates the n point FFT of s and stores the magnitude
             of the result in mag.
             Notes: n must be a power of two with n <= 1024
                    mag stores the magnitude, not the log magnitude */
  int fftmag(double s[], double mag[], int n);

  /* hamm: Calculates the Hamming windowed version of an n sample signal s
           and stores the result in hs (uses float precision) */
  void hamm(float s[], float hs[], int n);

  /* dhamm: Calculates the Hamming windowed version of an n sample signal s
            and stores the result in hs (uses double precision) */
```

```
void dhamm(double s[], double hs[], int n);

/* lpc: Calculates p Linear Predictive Coding coefficients
        b[1], ..., b[p]; (b[0] = 1.0) The LPC coefficients approximate
        the signal x[].
        Convention used:  signs of the b[k]'s are such that the denominator
                          of the transfer function is of the form
                          1+(sum from k=1 to p of b[k]*z**(-k))
        This is the normal convention for the inverse filtering formulation
        errn = normalized minimum error
        rmse = root mean square energy of the x[i]'s
        n = number of data points in frame
        p = number of coefficients = degree of inverse filter polynomial,
            p <= 40 */
int lpc(float x[], int n, int p, float b[], float *rmse, float *errn);

/* voiced_error_gen: Generates a seg_len length voiced error signal,
                     segment, which is a sequence of pulses (with a
                     period of pitch_period/2) corresponding to the
                     excitation signal for voiced speech is generated
                     using the function f(x) = ax/(1+a*x*x).  A constant
                     multiplicative factor based on the standard deviation
                     measured over the actual error signal is used to
                     modulate the signal to the appropriate amplitude.
                     White gaussian noise with a standard deviation of
                     err_stdev is added */
void voiced_error_gen(float *segment, int seg_len, float err_stdev,
                      int pitch_period);

/* unvoiced_error_gen: Generates a seg_len length unvoiced error signal,
                       segment, which is just white noise with a standard
                       deviation of err_stdev */
void unvoiced_error_gen(float *segment, int seg_len, float err_stdev)

/* code_select: Selects the appropriate codebook.
                **real_cep: This is the array of cepstral coefficients generated
                            by the frame over the entire speech signal.
                **code_cep: This contains the codebook for the cepstral coefficients.
                **code_lpc: This contains the codebook for the LPC coefficients.
```

```
                  **codeword: Once the best match between the input word and that
                              from the codebook (cepstral) is found, the corresponding
                              word from the LPC codebook is transferred to 'codebook'
                              as the output to be used in speech generation. */
  void code_select(float **real_cep, float **code_cep, float **code_lpc, float **codeword,
                   int seg_num, int num_codes, int filter_order);

  /* wr_error: If n is zero it prints and error and exists
               otherwise, it prints an okay message and continues */
  void wr_error(int n);

  /* print_directions: Displays usage instructions */
  void print_directions();
#else
  void hamm();
  void dhamm();
  int fftmag();
  int lpc();
  void voiced_error_gen();
  void unvoiced_error_gen();
  void code_select();
  void wr_error(int n);
  void print_directions();
#endif
```

6.2 hw4.c

```
/*************************************************************************
Authors: Varun Madhok and Chris Taylor
Date:    December 6, 1996
File:    hw4.c
Purpose: This file contains the main application for the speech compression
         application that was part of our fourth homework assignment for
         EE649 -- Speech Processing
*************************************************************************/

#include <stdio.h>
#include <math.h>
#include "/home/offset/a/taylor/Src/Recipes/recipes/nrutil.h"
```

```
#include "/home/offset/a/taylor/Src/Recipes/recipes/nr.h"
#include "/home/offset/a/taylor/Src/Recipes/Vrecipes/randlib.h"
#include "hw4.h"
#define MOD_FACTOR 1.5
#define OTHER 0
#define MALE   1
#define FEMALE 2
#define CHILD  3

int main(int argc, char *argv[])
{
  int      i;
  int      j;
  int      k;
  int      N_flag;
  int      pole;
  int      itemp;
  int      num;
  int      seg_len;
  int      seg_num;
  int      filter_order;
  int*     data;
  int      pad_location;
  int      ID;
  int      sampling_rate;
  int      lifter_from_this_sample;
  int      lifter_till_this_sample;
  float    ftemp;
  float    rmse;
  float    errn;
  float*   filter_coeffs;
  float*   ceps_coeffs;
  float    e;
  float*   gen_e;
  float    err_stdev;
  float    err_mean;
  float*   segment;
  float*   windowed_segment;
  int      non_zero_count;
  int     max_index;
  int     pitch_period;
  int     num_codes;
  int     category_is;
  /* long_segment is of length 1024 samples. It comprises the windowed segment
     in the centre padded left and right by an appropriate number*/
  double* long_segment;
  double* fft_segment;
  double  non_zero_sum;
  double  max_samp;
  FILE*   infile;
  FILE*   errfile;
  FILE*   gen_errfile;
  FILE*   cepsfile;
  FILE*   lpcfile;
  float*  gen_err;
  float** real_cep;
  float** code_cep;
  float** code_lpc;
  float** codeword;
  float*  error_signal;
  float*  output_signal;
  char    fname[55];
  char    out_fname[55];
  char    temp_str[90];
```

Your `main` function is over 400 lines long. You really should split this into a number of shorter functions. Doing so would allow you to more easily test, modify, and maintain your code.

```
char    num_codes_string[8];
char    code_fname[15];
char    group_name[5];
char    CODEBOOKS_EXIST;

if (( argc > 1 ) && ( !strcmp (argv [1], "-help" ))) {
  print_directions();
}
/*the default values are assigned here*/
strcpy(fname, IN_DEF_FILE);
strcpy(out_fname, OUT_DEF_FILE);
strcpy(code_fname, CODE_DEF_DIR);
N_flag=1;
pole=0;
num_codes=DEF_CODEBK_SIZE;
strcpy(num_codes_string, "2");
num=DEF_DAT;
filter_order= 20;
seg_len=SEGMENT_LENGTH;
category_is=OTHER;
ID=0;
CODEBOOKS_EXIST=1;
sampling_rate=16000;
/*The for loop below works in the command line arguments into the program */
for(i=1;i<argc;i++) {
  if(!strcmp(argv[i],"-in")) {
    strcpy(fname, argv[1+i]);
  } else if(!strcmp(argv[i],"-out")) {
    strcpy(out_fname, argv[1+i]);
  } else if(!strcmp(argv[i],"-code")) {
    strcpy(code_fname, argv[1+i]);
  } else if(!strcmp(argv[i],"+P")) {
    pole=1;
  } else if(!strcmp(argv[i],"+N")) {
    N_flag=0;
  } else if(!strcmp(argv[i],"-ID")) {
    sscanf(argv[i+1], "%d", &ID);
  } else if(!strcmp(argv[i],"-bksize")) {
      sscanf(argv[i+1], "%d", &num_codes);
    strcpy(num_codes_string, argv[1+i]);
  } else if(!strcmp(argv[i],"-num")) {
    sscanf(argv[i+1], "%d", &num);
  } else if(!strcmp(argv[i],"-segl")) {
    sscanf(argv[i+1], "%d", &seg_len);
  } else if(!strcmp(argv[i],"-samp")) {
    sscanf(argv[i+1], "%d", &sampling_rate);
  } else if(!strcmp(argv[i],"-group")) {
    strcpy(group_name, argv[1+i]);
    if((!strncmp(group_name, "m", 1))||(!strncmp(group_name, "M", 1))) {
      category_is=MALE;
    } else if((!strncmp(group_name, "f", 1))||(!strncmp(group_name, "F", 1))) {
      category_is=FEMALE;
    } else if((!strncmp(group_name, "j", 1))||(!strncmp(group_name, "J", 1))) {
      category_is=CHILD;
    } else {
      category_is=OTHER;
    }
  }
}
if(pole) {
  strcpy(temp_str,"zcat ");
  strcat(temp_str, fname);
  if((infile=popen(temp_str, "r"))==NULL) {
    wr_error(0);
  }
} else {
  strcpy(temp_str, fname);
  if((infile=fopen(temp_str, "r"))==NULL) {
    wr_error(0);
  }
}

strcpy(temp_str, "/home/purcell/c/ee649/Data/p3/codebooks/");
strcat(temp_str, code_fname);
strcat(temp_str, "/cepstral/codebook.");
strcat(temp_str, num_codes_string);
if((cepsfile=fopen (temp_str, "r"))==NULL) {
  CODEBOOKS_EXIST=0;
}
if(CODEBOOKS_EXIST) /* If the codebooks are found in the right location
         the program proceeds as normal otherwise output
         files corresponding to the actual excitation
         signal and the generated excitation are created */
{
  strcpy(temp_str, "/home/purcell/c/ee649/Data/p3/codebooks/");
  strcat(temp_str, code_fname);
  strcat(temp_str, "/lpc/codebook.");
  strcat(temp_str, num_codes_string);
  strcat(temp_str, ".lpc");
  if((lpcfile=fopen (temp_str, "r"))==NULL) {
    CODEBOOKS_EXIST=0;
  }
}
if(CODEBOOKS_EXIST==0) {
  strcpy(temp_str, out_fname);
  strcat(temp_str, ".err");
  if((errfile=fopen (temp_str, "w"))==NULL) {
    wr_error(0);
  }
  strcpy(temp_str, out_fname);
  strcat(temp_str, ".gen");
  if((gen_errfile=fopen (temp_str, "w"))==NULL) {
    wr_error(0);
  }
}
readseed();
/* This set-up determines the range to be left as non-zero in the
   liftering of the cepstrum.  The range varies by gender and age. */
switch (category_is) {
  case MALE :
    lifter_from_this_sample=(int)((float)sampling_rate/200.0);/*200 Hz is used as upper limit*/
    lifter_till_this_sample=(int)((float)sampling_rate/100.0);/*100 Hz is used as lower limit*/
    break;
  case FEMALE :
    lifter_from_this_sample=(int)((float)sampling_rate/275.0);
    lifter_till_this_sample=(int)((float)sampling_rate/180.0);
    break;
  case CHILD :
    lifter_from_this_sample=(int)((float)sampling_rate/285.0);
    lifter_till_this_sample=(int)((float)sampling_rate/180.0);
    break;
  default :
    lifter_from_this_sample=(int)((float)sampling_rate/270.0);
    lifter_till_this_sample=(int)((float)sampling_rate/100.0);
    break;
}
data=(int *)ivector(0, num-1);
error_signal=(float *)vector(1, num);
/*reading data and calculating mean*/
ftemp=0.0;
j=0;
while(j<num) {
  fscanf(infile,"%d", &itemp);
  data[j]=itemp;
  j++;
}
if(pole) {
  pclose(infile);
} else {
  fclose(infile);
}

seg_num=num/seg_len; /*number of segments in the speech file*/
segment=(float *)vector(0, seg_len-1);
windowed_segment=(float *)vector(0, seg_len-1);
long_segment=(double *)dvector(0, (2*1024)-1);
fft_segment=(double *)dvector(0, 1024-1);
filter_coeffs=(float *) vector(0, filter_order);
ceps_coeffs=(float *) vector(1, filter_order);
/*e=(float *)c_vector(0, seg_len-1);*/
gen_e=(float *)c_vector(0, seg_len-1);
real_cep=(float **)matrix(1, seg_num, 1, filter_order);

pad_location=(1024-seg_len)/2;
for(k=1; k<=seg_num; k++) {
  for(j=0; j<seg_len; j++) {
    if(((k-1)*seg_len+j)<num) {
      segment[j]=(float) data[(k-1)*seg_len+j];
    } else {
      segment[j]=0.0;
    }
  }
  hamm(segment, windowed_segment, seg_len);
  /* At this stage calculate the pitch period of the input signal
     thereby classifying segment as voiced/unvoiced*/
  /* Step I - pad windowed segment from the left and right*/
  for(j=pad_location; j<(pad_location+seg_len); j++) {
    long_segment[2*j]=(double)windowed_segment[j-pad_location];
    long_segment[2*j+1]=0.0;
  }
  /* Left pad*/
  for(j=(pad_location-1); j>=0; j--){
    if(((k-1)*seg_len+j-pad_location)>=0) {
      long_segment[2*j]=0.0/*(double) data[(k-1)*seg_len+j-pad_location]*/;
    } else {
       long_segment[2*j]=0.0;
    }
    long_segment[2*j+1]=0.0;
  }
  /* Right pad*/
  for(j=(pad_location+seg_len+1); j<1024; j++) {
    if(((k-1)*seg_len+j)<num) {
      long_segment[2*j]=0.0;
    } else {
      long_segment[2*j]=0.0;
    }
    long_segment[2*j+1]=0.0;
  }
  /* Step II- calculate Fourier Transform*/
  dfour1(long_segment-1, 1024, 1);
  /* Step III- calculate IDFT of log()*/
  for(j=0; j<1024; j++) {
    fft_segment[j]=log(sqrt(long_segment[2*j]*long_segment[2*j]+
                   long_segment[2*j+1]*long_segment[2*j+1]));
  }
  /* Step IV - Lifter operation*/
  for(j=0; j<1024; j++) {
    long_segment[2*j]=fft_segment[j];
    long_segment[2*j+1]=0.0;
  }
  /* Inverse FFT of the log fft_segment is the cepstrum*/
  dfour1(long_segment-1, 1024, -1);
  max_samp=0.0;
  non_zero_count=0;
  non_zero_sum=0.0;
  if((k==ID)||(ID==0)) {
    /* Liftering is done so that the maxima corresponding to the
       pitch is accentuated (if it exists)*/
    for(j=0; j<(1024/2); j++) {
      if((j>lifter_till_this_sample)||(j<lifter_from_this_sample)) {
        long_segment[2*j]=0.0;
      }
      /* The location of the maximum is found and the value coresponding
         to the max is stored*/
      if(long_segment[2*j]>max_samp) {
        max_samp=long_segment[2*j];
        max_index=j;
      }
      if((long_segment[2*j]>=0.0)&&(j<=lifter_till_this_sample)&&
         (j>=lifter_from_this_sample)) {
        non_zero_count++;
        non_zero_sum+=fabs(long_segment[2*j]);
      }
    }
    non_zero_sum/=non_zero_count;
    /* Pitch detection is done here : If the max value is greater than the
       average non-negative signal over the liftered signal, we claim a
       pitch to have been detected*/
    if((max_samp>(MOD_FACTOR*non_zero_sum))&&(N_flag!=0)) {
      pitch_period=max_index;
    } else {
      pitch_period=-1;
    }
    lpc(windowed_segment, seg_len, filter_order, filter_coeffs, &rmse, &errn);
    /* Calculate error--->Initialization*/
    for(j=0;j<seg_len; j++) {
      gen_e[j]=0.0;
    }
    err_stdev=err_mean=0.0;
    for(j=0;j<seg_len; j++) {
      e=0.0;
      for(i=0; i<=filter_order; i++) {
        if(k==1) {
          if((j-i)>=0) {
            e+=filter_coeffs[i]*segment[j-i];
          }
        } else {
          e+=filter_coeffs[i]*(float)data[(k-1)*seg_len+j-i];
        }
      }
      if(!CODEBOOKS_EXIST) {
        fprintf(errfile, "%f\n", e);
      }
      err_mean+=e;
      err_stdev+=e*e;
    }
    err_mean/=(float)(seg_len);
    err_stdev/=(float)(seg_len);
    err_stdev-=(err_mean*err_mean);
    if(err_stdev>0.0) {
      err_stdev=sqrt(err_stdev);
    } else {
      err_stdev=0.0;
    }
    /* At this stage... use the voiced unvoiced decision
       plus standard deviation of the error signal to generate
       an 'error' signal.
       To recap - Parameters used are :
       a. (optional) Voiced/unvoiced flag : 0 if unvoiced, 1 if otherwise;
       b. standard deviation of the error for the frame;
       c. pitch period : -1 if unvoiced, something +ve if voiced; */
    /* An excitation signal is generated as and how we have classified the frame */
    if(pitch_period>0) {
      voiced_error_gen(gen_e, seg_len, err_stdev, pitch_period);
    } else {
      unvoiced_error_gen(gen_e, seg_len, err_stdev);
    }

    for(j=0; j<seg_len; j++) {
      error_signal[(k-1)*seg_len +j] = gen_e[j];
      if(!CODEBOOKS_EXIST) {
        fprintf(gen_errfile, "%f\n", gen_e[j]);
      }
    }

    for(i=1; i<=filter_order; i++) {
      ceps_coeffs[i]=-filter_coeffs[i];
      ftemp=0.0;
      for(j=1; j<=(i-1); j++) {
        ftemp-=(float)j * ceps_coeffs[j]*filter_coeffs[i-j];
      }
      ceps_coeffs[i]+=(ftemp/(float)i);
      real_cep[k][i]=ceps_coeffs[i];
    }
  }
} /* End of k loop -> new segment begins */

if(CODEBOOKS_EXIST) {
  codeword=(float **)matrix(1, seg_num, 1, filter_order);
  code_lpc=(float **)matrix(1, num_codes, 1, filter_order);  /*read codebook LPC*/
  code_cep=(float **)matrix(1, num_codes, 1, filter_order);  /*read codebook CEPS*/
}
/* Freeing memory */
free_ivector(data, 0, num-1);
free_vector(gen_e, 0, seg_len-1);
free_vector(windowed_segment, 0, seg_len-1);
free_dvector(long_segment, 0, (2*1024)-1);
free_dvector(fft_segment, 0, 1024-1);
free_vector(segment, 0, seg_len-1);
free_vector(filter_coeffs, 0, filter_order);
free_vector(ceps_coeffs, 1, filter_order);

if(CODEBOOKS_EXIST) {
  for(i=1; i<=num_codes; i++) {
    for(j=1; j<=filter_order; j++) {
      fscanf(cepsfile,"%f", &ftemp);
      code_cep[i][j]=ftemp;
      fscanf(lpcfile,"%f", &ftemp);
      code_lpc[i][j]=ftemp;
    }
  }
  /* At this stage... have frame by frame data on cepstral coefficients
     have codebooks on lpc and cepstral coeffs.
     Proceed with the association
     Output is stored in codeword */
  code_select(real_cep, code_cep, code_lpc, codeword, seg_num, num_codes, filter_order);
  free_matrix(code_cep, 1, num_codes, 1, filter_order);
  free_matrix(code_lpc, 1, num_codes, 1, filter_order);

  /* Incorporate inverse filtering process */
  output_signal=(float *)vector(1, num);

  for(k=1;k<=seg_num;k++) {
    for(i=1;i<=seg_len;i++) {
      output_signal[(k-1)*seg_len+i] = error_signal[(k-1)*seg_len+i];
      for(j=1;j<=filter_order;j++) {
        /* Generating output using excitation signal
           and LPC coefficients from the codebook */
        if(((k-1)*seg_len+i-j)>=1) {
          output_signal[(k-1)*seg_len+i] -= codeword[k][j]*output_signal[(k-1)*seg_len+i-j];
        }
      }
      printf("%d\n", (int)output_signal[(k-1)*seg_len+i]);
    }
  }
  free_vector(output_signal, 1, num);
  free_matrix(codeword, 1, seg_num, 1, filter_order);
  fclose(lpcfile);
  fclose(cepsfile);
}
free_matrix(real_cep, 1, seg_num, 1, filter_order);
free_vector(error_signal, 1,  num);
if(CODEBOOKS_EXIST==0) {
  fclose(errfile);
}
if(CODEBOOKS_EXIST==0) {
  fclose(gen_errfile);
```

```
  }
  writeseed();

  return 0;
}
```

6.3 code_select.c

```
/***********************************************************************
Authors: Varun Madhok and Chris Taylor
Date:    December 6, 1996
File:    code_select.c
Purpose: This file contains the code_select function which selects the
         appropriate codebook for the speech being processed by the speech
         compression application that was part of our fourth homework assignment
         for EE649 -- Speech Processing
***********************************************************************/
#include <math.h>

void code_select(float **real_cep, float **code_cep, float **code_lpc, float **codeword,
     int seg_num, int num_codes, int filter_order)
{
  int i;
  int k;
  int j;
  float err;
  float emin;
  for(k=1;k<=seg_num;k++) {
    emin = 9999999.9;
    for(i=1;i<=num_codes;i++) {
      err = 0.0;
      /* Measuring difference between the generated codeword and one from the
         cepstral codebook*/
      for(j=1;j<=filter_order;j++) {
        err += (double)fabs((float)real_cep[k][j] - (float)code_cep[i][j]);
      }
      if(err<emin) {
        for(j=1;j<=filter_order;j++) {
          codeword[k][j] = code_lpc[i][j];
        }
        emin = err;
      }
    }
  }
}
```

6.4 wr_error.c

```
/*************************************************************************
Authors: Varun Madhok and Chris Taylor
Date:    December 6, 1996
File:    wr_error.c
Purpose: This file contains the wr_error function which displays an error
         message and exists if n=0.
*************************************************************************/
void wr_error(int n)
{
  if (n==0)
  {
    printf("ERROR␣:%c␣Aborting␣and␣exitting.\n", 0x07);
    exit(1);
  }
  else printf("Flag␣%d␣:␣All␣OK␣...\n",n);
}
```

6.5 print_directions.c

```
/*************************************************************************
Authors: Varun Madhok and Chris Taylor
Date:    December 6, 1996
File:    print_directions.c
Purpose: This file contains the print_directions function which displays
         usage instructions for the speech compression application that
         was part of our fourth homework assignment for EE649 -- Speech
         Processing
*************************************************************************/
```

```
void print_directions()
{
  printf("program␣␣Usage:\n");
  printf("␣␣␣␣␣␣␣␣␣␣−num␣␣␣␣␣n␣␣␣␣␣Number␣of␣records␣in␣testfile␣\n");
  printf("␣␣␣␣␣␣␣␣␣␣−ID␣␣␣␣␣␣n␣␣␣␣␣ID␣of␣segment␣to␣be␣extracted␣(enter␣0␣for␣all)\n");
  printf("␣␣␣␣␣␣␣␣␣␣−bksize␣␣n␣␣␣␣␣Number␣of␣codes␣(size␣of)in␣the␣desired␣codebook␣\n");
  printf("␣␣␣␣␣␣␣␣␣␣−samp␣␣␣␣n␣␣␣␣␣sampling␣rate\n");
  printf("␣␣␣␣␣␣␣␣␣␣−in␣␣␣*char␣␣␣␣in−filename\n");
  printf("␣␣␣␣␣␣␣␣␣␣−out␣␣*char␣␣␣␣out−filename\n");
  printf("␣␣␣␣␣␣␣␣␣␣−code␣*char␣␣␣␣codebook␣directory␣to␣be␣used␣in␣/home/purcell/c/ee649/Data/p3/codebooks/\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Valid␣options␣are␣−>␣male␣(default)\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣female\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣all_males\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣all_females\n");
  printf("␣␣␣␣␣␣␣␣␣␣−segl␣␣␣␣n␣␣␣␣␣segment␣length\n");
  printf("␣␣␣␣␣␣␣␣␣␣−group␣*char␣␣␣group␣name␣to␣decide␣cepstrum␣liftering.\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Valid␣options␣are␣−>␣O␣or␣o␣␣(default);\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣M␣or␣m␣␣␣male;\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣F␣or␣f␣␣␣female;\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣J␣or␣j␣␣␣child.\n");
  printf("␣␣␣␣␣␣␣␣␣␣+P␣␣␣␣␣␣␣␣␣␣␣␣␣use␣popen\n");
  printf("␣␣␣␣␣␣␣␣␣␣+N␣␣␣␣␣␣␣␣␣␣␣␣␣dont␣classify␣voiced/unvoiced\n");
  printf("\nDESCRIPTION\n");
  printf("Default␣input␣file␣␣␣␣␣␣␣␣␣␣:␣%s\n", IN_DEF_FILE);
  printf("Default␣codebook␣dir␣␣␣␣␣␣␣␣:␣%s\n", CODE_DEF_DIR);
  printf("Default␣codebook␣size␣␣␣␣␣␣␣:␣%d\n", DEF_CODEBK_SIZE);
  printf("Default␣number␣of␣records␣␣␣:␣%d\n", DEF_DAT);
  printf("Default␣segment␣length␣␣␣␣␣␣:␣%d\n", SEGMENT_LENGTH);
  printf("Default␣sampling␣rate␣␣␣␣␣␣␣:␣16000␣Hz\n");
  printf("Default␣␣filter␣order␣␣␣␣␣␣␣:␣20\n");
  exit(0);
}
```

6.6 unvoiced_error_gen.c

```
/*************************************************************************
Authors: Varun Madhok and Chris Taylor
Date:    December 6, 1996
File:    unvoiced_error_gen.c
```

```
Purpose:  This file contains the unvoiced_error_gen function which generates
          the voiced error signal for the speech compression application that
          was part of our fourth homework assignment for EE649 -- Speech
          Processing
*************************************************************************/

#include <math.h>
#include <stdio.h>
#include "hw4.h"
#include "/home/offset/a/taylor/Src/Recipes/Vrecipes/randlib.h"
void unvoiced_error_gen(float *segment, int seg_len, float err_stdev)
{
  int i;
  /* The unvoiced excitation signal is just white noise with the
     desired variance */
  for (i=0; i<seg_len; i++) {
    segment[i]=normal()*err_stdev;
  }
}
```

6.7 unvoiced_error_gen.c

```
/*************************************************************************
Authors:  Varun Madhok and Chris Taylor
Date:     December 6, 1996
File:     voiced_error_gen.c
Purpose:  This file contains the voiced_error_gen function which generates
          the voiced error signal for the speech compression application that
          was part of our fourth homework assignment for EE649 -- Speech
          Processing
*************************************************************************/
#include <math.h>
#include <stdio.h>
#include "hw4.h"
#include "/home/offset/a/taylor/Src/Recipes/Vrecipes/randlib.h"
void voiced_error_gen(float *segment, int seg_len, float err_stdev, int pitch_period)
{
  float var;
  float mult_factor;
```

```c
  float ftemp;
  float const_factor;
  int i;
  int j;
  int num_peaks;
  var=err_stdev*err_stdev*(float)seg_len;
  num_peaks=(int)((float)seg_len/(float)pitch_period);
  mult_factor = 0.95*sqrt(var/(float) num_peaks);
  const_factor=10.0;
  j=0;
  for(i=0; i<seg_len; i++) {
    if(j<(int)pitch_period) {
      ftemp=(float)j-(float)pitch_period/2.0; /* This assures that the peaks shall
              occur near about pitch_period/2.0 */
      /* The sequence of pulses corresponding to the excitation signal for voiced speech
         is generated using the function f(x) = ax/(1+a*x*x). A constant multiplicative
         factor based on the standard deviation measured over the actual error signal is
         used to modulate the signal to the appropriate amplitude.
         White gaussian noise (pseudo-random) is added. */
      segment[i]=err_stdev* normal()+sqrt(const_factor)*mult_factor*ftemp/(1.0+const_factor*ftemp*ftemp);
    } else {
      j=0; /* Once the count over the pitch_period is exceeded, the counter is reset*/
      segment[i]=0.0;
    }
    j++;
  }
}
```

6.8 hamm.c

```c
/***********************************************************************
Authors: Varun Madhok and Chris Taylor
Date:    December 6, 1996
File:    hamm.c
Purpose: This file contains the hamm function which applies a Hamming window
         to the n sample signal for the speech compression application that
         was part of our fourth homework assignment for EE649 -- Speech
         Processing
***********************************************************************/
```

```
#include <math.h>
#define PI 3.14159265

void hamm(float s[], float hs[], int n)
{
  double omega;
  double w;
  int k;

  omega=2*PI/(n-1);

  for(k=0; k<n; k++) {
    w = 0.54 - 0.46 * cos(k * omega);
    hs[k] = s[k] * w;
  }
}
```

6.9 dhamm.c

```
/***********************************************************************
Authors: Varun Madhok and Chris Taylor
Date:    December 6, 1996
File:    hamm.c
Purpose: This file contains the hamm function which applies a Hamming window
         to the n sample signal for the speech compression application that
         was part of our fourth homework assignment for EE649 -- Speech
         Processing
***********************************************************************/
#include <math.h>
#define PI 3.14159265

void dhamm(double s[], double hs[], int n)
{
  double omega;
  double w;
  int k;

  omega=2*PI/(n-1);

  for(k=0; k<n; k++) {
    w = 0.54 - 0.46 * cos(k * omega);
    hs[k] = s[k] * w;
  }
}
```

6.10 fftmag.c

```
/***************************************************************************
Authors: Varun Madhok and Chris Taylor
Date:    December 6, 1996
File:    fftmag.c
Purpose: This file contains the fftmag function and some helper functions
         which calculate the magnitude (not log magnitude) of an n point
         signal for the speech compression application that was part of
         our fourth homework assignment for EE649 -- Speech Processing
***************************************************************************/

#include <stdio.h>
#include <math.h>
#define PI 3.14159265
#define c_mag(c1)    sqrt((c1.r)*(c1.r) + (c1.i)*(c1.i))

/* A structure to hold a complex number */
typedef struct {
  double r;
  double i;
} COMPLEX;

/* Authors: Varun Madhok and Chris Taylor
   Date:    December 6, 1996
   Purpose: Returns the product of two complex numbers c1 and c2 */
COMPLEX c_mult(COMPLEX c1, COMPLEX c2)
{
  COMPLEX c3;

  c3.r=c1.r*c2.r - c1.i*c2.i;
  c3.i=c1.i*c2.r + c1.r*c2.i;
  return c3;
```

```
}

/* Authors: Varun Madhok and Chris Taylor
   Date:    December 6, 1996
   Purpose: Returns the sum of two complex numbers c1 and c2 */
COMPLEX c_add(COMPLEX c1, COMPLEX c2)
{
```

You should just put the function prototypes up here and put do the implemetation after fftmag since fftmag is the function of interest in this source file.

```
  COMPLEX c3;

  c3.r=c1.r + c2.r;
  c3.i=c1.i + c2.i;
  return c3;
}

/* Authors: Varun Madhok and Chris Taylor
   Date:    December 6, 1996
   Purpose: Returns the difference of two complex numbers c1 and c2 */
COMPLEX c_sub(COMPLEX c1, COMPLEX c2)
{
  COMPLEX c3;

  c3.r=c1.r - c2.r;
  c3.i=c1.i - c2.i;
  return c3;
}

/* Authors: Varun Madhok and Chris Taylor
   Date:    December 6, 1996
   Reference: Steiglitz, Introduction to Discrete Systems */
int fftmag(double s[], double mag[], int n)
{
  int i;
  int j;
  int m;
  int l;
  int length;
  int loc1;
  int loc2;
  double arg;
  double w;
  COMPLEX c;
  COMPLEX z;
  COMPLEX f[1024];

  for(i=0; i<n; i++) {
    j=0;
    for(m=1; m<n; m += m) {
      if(i % (m+m) >= m)
      j += n/(m+m);
    }
    f[i].r=s[j];
    f[i].i=0;
  }

  for(length=2; length <= n; length += length) {
    w = -2.0*PI/(double)length;
    for(j=0; j<n; j += length) {
      for(l=0; l<length/2; l++) {
        loc1=l+j;
        loc2=loc1+length/2;
        arg=w*l;
        c.r=cos(arg);
        c.i=sin(arg);
        z=c_mult(c,f[loc2]);
        f[loc2]=c_sub(f[loc1],z);
        f[loc1]=c_add(f[loc1],z);
      }
    }
  }

  for (i=0; i<n; i++) {
    mag[i] = c_mag(f[i]);
  }
}
```

6.11 lpc.c

```
/************************************************************************
Authors: Varun Madhok and Chris Taylor
Date:    December 6, 1996
File:    lpc.c
Purpose: This file contains the lpc function which calculates the LPC
         coefficients that approximate the signal x.  The function is
         used by the speech compression application that was part of
         our fourth homework assignment for EE649 -- Speech Processing
************************************************************************/

#include <stdio.h>
#include <math.h>
#define MAX_LPC_ORDER 40
#define EVEN(x) !(x%2)

int lpc(float x[], int n, int p, float b[], float* rmse, float* errn)
{
  int   i;
  int   k;
  float reflect_coef[MAX_LPC_ORDER+1];
  float auto_coef[MAX_LPC_ORDER+1];
  float sum;
  float temp1,temp2;
  float current_reflect_coef;
  float pred_error;

  for(i=0; i<=p; i++) {
    sum = 0.0;

    for(k=0; k< n-i; k++) {
        sum += (x[k] * x[k+i]);
      }

      auto_coef[i] = sum;
    }

    *rmse = auto_coef[0];

    if(*rmse == 0.0) {
      return 1;       /* Zero power.  */
    }
```

You should only have one `return` statement for each function. Having multiple `return` statements makes your function more vulnerable to defects being introduced into your code when it is modified in the future.

```

    pred_error = auto_coef[0];
    b[0] = 1.0;

    for (k=1; k<=p; k++) {
      sum = 0.0;

      for(i=0; i<k; i++) {
        sum += b[i] * auto_coef[k-i];
      }

      current_reflect_coef = -sum/pred_error;
      reflect_coef[k] = current_reflect_coef;
      b[k] = current_reflect_coef;

      for(i=1; i <= (k-1)/2; i++) {
        temp1 = b[i];
        temp2 = b[k-i];
      b[i] += current_reflect_coef * temp2;
      b[k-i] += current_reflect_coef * temp1;
    }

    if(EVEN(k)) {
      b[k/2] += current_reflect_coef * b[k/2];
    }

    pred_error *= (1.0 - current_reflect_coef * current_reflect_coef);

    if(pred_error <= 0.0) {
      return 2;        /* Non-positive prediction error */
    }
  }

  *errn = pred_error;
  return 0;            /* Normal return */
}
```

Assignment Grade Summary

Score	Category
30/30	Meeting specifications
15/20	Technical quality
29/30	Narrative report
5/5	Internal documentation (comments)
0/7	Activity log
3/3	Submission procedure
4/5	Spelling and grammar
86/100	Assignment Grade