More Than Meets the Ears: The Voice Transformers

18-551, Fall 2006
Group – 1, Final Report

Ramanpreet Singh Pahwa (rpahwa@andrew.cmu.edu)
Teng Ji Lim (tlim@andrew.cmu.edu)
Marcus Chen Caixing (caixingc@andrew.cmu.edu)

Table Of Contents

1. **Background**
   - 1.1. Introduction
   - 1.2. Applications
   - 1.3. Prior 18551 Projects
     - 1.3.1. Spring 2001
     - 1.3.2. Spring 2004

2. **Implementation**
   - 2.1. Training Part
     - 2.1.1. Overview
     - 2.1.2. The TIMIT database
     - 2.1.3. Dynamic Time Warping (DTW)
     - 2.1.4. Extraction of LPCs/LSFs and excitation spectrum
     - 2.1.5. Summary
   - 2.2. Transformation Part
     - 2.2.1. Overview
     - 2.2.2. Pre-emphasis
     - 2.2.3. Windowing
     - 2.2.4. Input LPCs and LSF
     - 2.2.5. Estimation of Weights
     - 2.2.6. Target LSF and LPCs
     - 2.2.7. Target Vocal Tract Mapping
     - 2.2.8. Target Excitation
     - 2.2.9. Output Speech in Frequency Domain
     - 2.2.10. De-emphasis
     - 2.2.11. Summary
   - 2.3. Post processing Part
     - 2.3.1. Pitch Scale Modification
     - 2.3.2. PSOLA on the Output Speech
     - 2.3.3. Final Output speech
     - 2.3.4. Summary
   - 2.4. Attempted Improvements

3. **Data Flow**

4. **Speed and Memory Issues**
   - 4.1. Memory
   - 4.2. Speed

5. **Results**

6. **Discussion and Further Work**

7. **References**

8. **Appendix: Schedule and Assigned Tasks**

1. Background

1.1 Introduction

Voice transformation is the process of modifying a speaker's voice (referred to in voice conversion terms as the source) to make it sound as if it was spoken by a specific (target) speaker. This transformation involves modifications to any aspect of the signal like fundamental frequency of voicing, duration, energy, and formant positions that carries speaker identity. When we speak we emit a speech pattern containing at least two kinds of information: our meaning (message) and our identity. Voice transformation enables us to transform the speaker's speech pattern into the target's speech pattern while preserving the original content (message). Hence, it transforms how something is said without changing what is said.

1.2 Applications

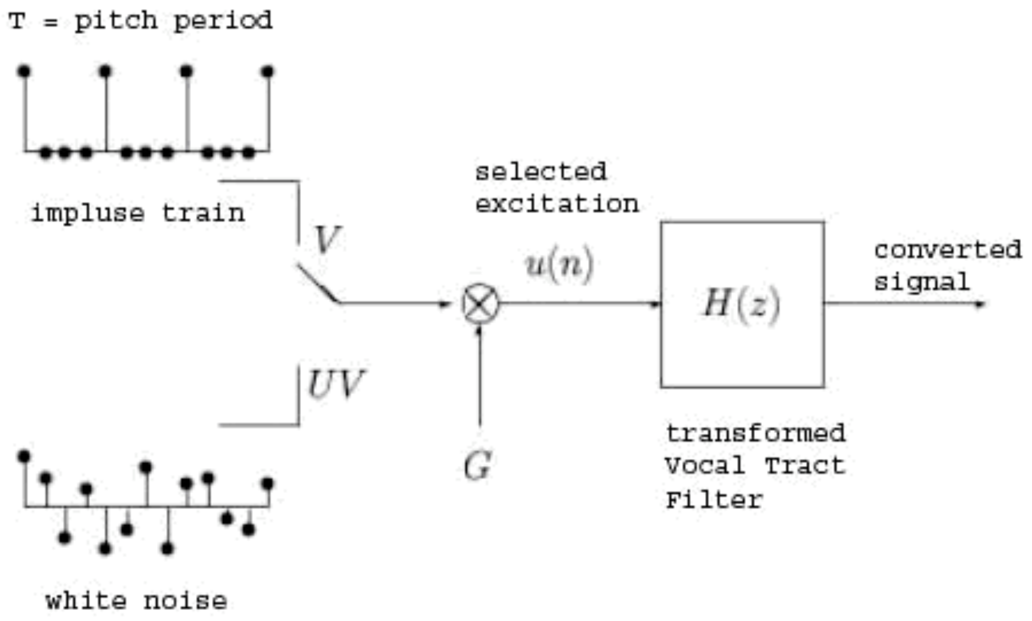
Voice transformation has numerous applications, such as the areas of foreign language training and military. A TTS (text to speech) system with voice transformation technology integrated can produce many different voices. In cases where the speaker identity plays a key role, such as dubbing movies and TV-shows, the availability of high quality voice transformation technology is very valuable as it allows the appropriate voice to be generated, in different languages, without the original actors being present. In karaoke, voice transformation can be used to make the singer's voice sound more similar to the original singer's voice. Furthermore, Voice transformation can be used to create "spy toys" that allows users to mask their voice to trick the person listening on the other side of the phone.

1.3 Prior 18551 Projects

When we started this project, we researched previous projects and found two projects, one in Spring 2001 and the other in Spring 2004, that were similar to ours.

1.3.1 Spring 2001

Group 1 in Spring 2001 did a project on "Speech Morphing for Space Marines". The group's algorithm was based on modification on LPC coefficients and the excitation pitch. First, they divided the input signal into smaller frames of about 20ms each and found the LPC coefficients in each frame. Then they determined if the current frame was a voiced or unvoiced sound. For voiced sounds, they used a pitch detection algorithm to determine the pitch of that part of the speech. The voice conversion was then performed based on Cumulative Density Function (CDF), which is described in [5]. The LPC coefficients were then replaced with the corresponding target coefficients. The final output speech is synthesized using the new coefficients and an impulse train with the target pitch for voiced sounds or white noise for unvoiced sounds as excitation. The following figure describes the final stage of their algorithm:



The group reported that that quality of the re-synthesized speech was often poor and contained pops and cracks. They also mentioned that sometimes the output speech was unintelligible and often didn't sound like the target speaker. Also, their algorithm was text dependent i.e. their algorithm expected the target and source speakers to speak the same text.

One reason why their output was often unintelligible might be that they were using LPC coefficients for the voice conversion. LPC coefficients are very sensitive and often prone to distortion. Relatively small changes in the representation of the LPC coefficients results in a large change in the pole locations of the vocal tract filter model. Hence, this often leads to an unstable filter.

1.3.2 Spring 2004

Group 9 in Spring 2004 did a project on voice transformation: "Hey! Stop sounding like me!!" They mentioned all these problems faced by Spring 2001 group and tried to implement an algorithm that used Linear Spectral Frequencies (LSF) instead of LPC coefficients to represent the formant frequencies. The reason they used LSF is because:

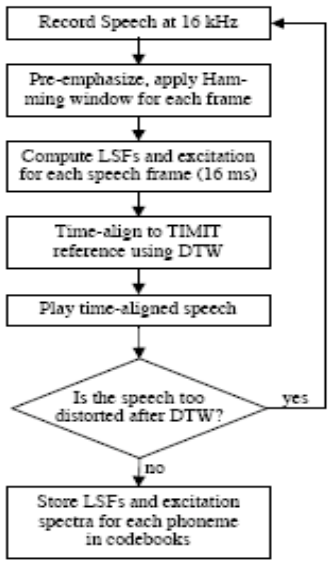
1) LSF have a nice quantization property: they can be interpolated from one speech frame to the next one unlike LPC coefficients

2) They are quite de-correlated and the values change slowly from one speech frame to another frame. Thus, this allows a more accurate inter-frame prediction

3) They are "naturally ordered" (highest to lowest) and hence the synthesis filter is more stable.

They process signals at 16 kHz and 16 bits per sample. They pre-emphasize the speech and decompose it into 256 samples of 16ms each. They divided the input signal into smaller frames of about 16ms each and found the LPC coefficients in each frame. They then convert the LPC coefficients to LSF.

For the training part, they use Dynamic Time Warping (DTW) to time-align the speech signals. This makes phoneme extraction as accurate as possible. They then find the Average LSF and excitation spectrum for each phoneme. After this, they calculate the estimated weights that will be used to find the target LSF from source LSF. Then, they use these weights to calculate target LSF and hence, target LPC coefficients. They model the target speaker's vocal tract by using these estimated LPC coefficients. After this, they calculate the filter using the weighted combination of codebook filters. The output speech is then de-emphasized. The following flow chart describes the training part of their algorithm:


Record Speech at 16 kHz
Pre-emphasize, apply Hamming window for each frame
Compute LSFs and excitation for each speech frame (16 ms)
Time-align to TIMIT reference using DTW
Play time-aligned speech
Is the speech too distorted after DTW?
yes
no
Store LSFs and excitation spectra for each phoneme in codebooks

The group was successful in making the algorithm text independent (the speaker and target can speak entirely different lines now). However, the output sounded very robotic and noisy. They owed this to their implementation of excitation signal and lack of more detailed codebooks. They also mentioned that PSOLA (Pitch-Synchronous Overlap-Add) and energy scaling would have been very helpful in improving the quality of the output.

2. ***Implementation***

Our System consists of three parts: A training phase, a conversion phase and a post-processing phase. All three parts process the speech signal at 16 kHz, 16 bits per sample. For the first two parts, the speech signal is decomposed into frames where each frame consists of 256 samples of 16ms each. In the conversion phase, each frame is processed separately and sequentially. On the other hand, a few frames are processed together in the post-processing phase.

2.1 Training Part

2.1.1 Overview

The training part, in a nutshell, is where both the source and target speakers speak a series of sentences, which are then analyzed, in order to extract information about how each phoneme for each speaker sounds. To do this, we need to somehow know **what** phonemes are being spoken, as well as which segments of that speech correspond to those phonemes (i.e. **when** the phonemes are spoken). The process of identifying what phonemes occur at what time positions in a speech is known as **phoneme segmentation**. Automated phoneme segmentation itself is a difficult research problem, and is beyond the scope of our project. Instead, we use sentences from the TIMIT database with known phoneme transcriptions, make the source and target speakers speak those sentences and then time-align our data using a technique known as Dynamic Time Warping (DTW) to the TIMIT data so that we have data from the 2 speakers that follows the known transcriptions. At this stage, we can then extract the phonemes, and use them to build codebooks representative of the characteristics of each speaker's speech.

In the training part, we use a 16th order LPC analysis to build codebooks containing 16 LSF and glottal excitation spectrum for each phoneme. We used 60 phonemes. The entire training part was done in C on PC.

2.1.2 The TIMIT database

As stated earlier, phoneme segmentation is a complicated procedure. In [2], Arslan used Sentence Hidden Markov Models (Sentence HMMs) to detect phonemes in order to build the codebooks. For our project, we decided instead to use the TIMIT database as it is a very standardized speech database and it contains the phoneme transcriptions for every sentence.

The TIMIT database consists of speech from over 400 unique speakers and the phonetic transcriptions for each speech file. For every speaker, there are ten phonetically diverse sentences available, from which a representative codebook can be constructed.

Unfortunately, although the TIMIT speech files are labeled with a .WAV extension, they are not in the standard WAV file format that MATLAB can read. We used a program called Sound eXchange (SoX) to convert it to the normal WAV format. The now-playable WAV files are used later for the source and target speaker's reference while recording their voices.

2.1.3 Dynamic Time Warping (DTW)

During the training part, the source and target speakers are asked to speak the sentences at the same rate as the reference voice in TIMIT database. Even though we play the reference recording simultaneously as the speech is recorded, it is not possible for a person to speak the sentence at the same rate as the reference speaker. To overcome this problem, we implemented Dynamic Time Warping (DTW) to time align the person's speech.

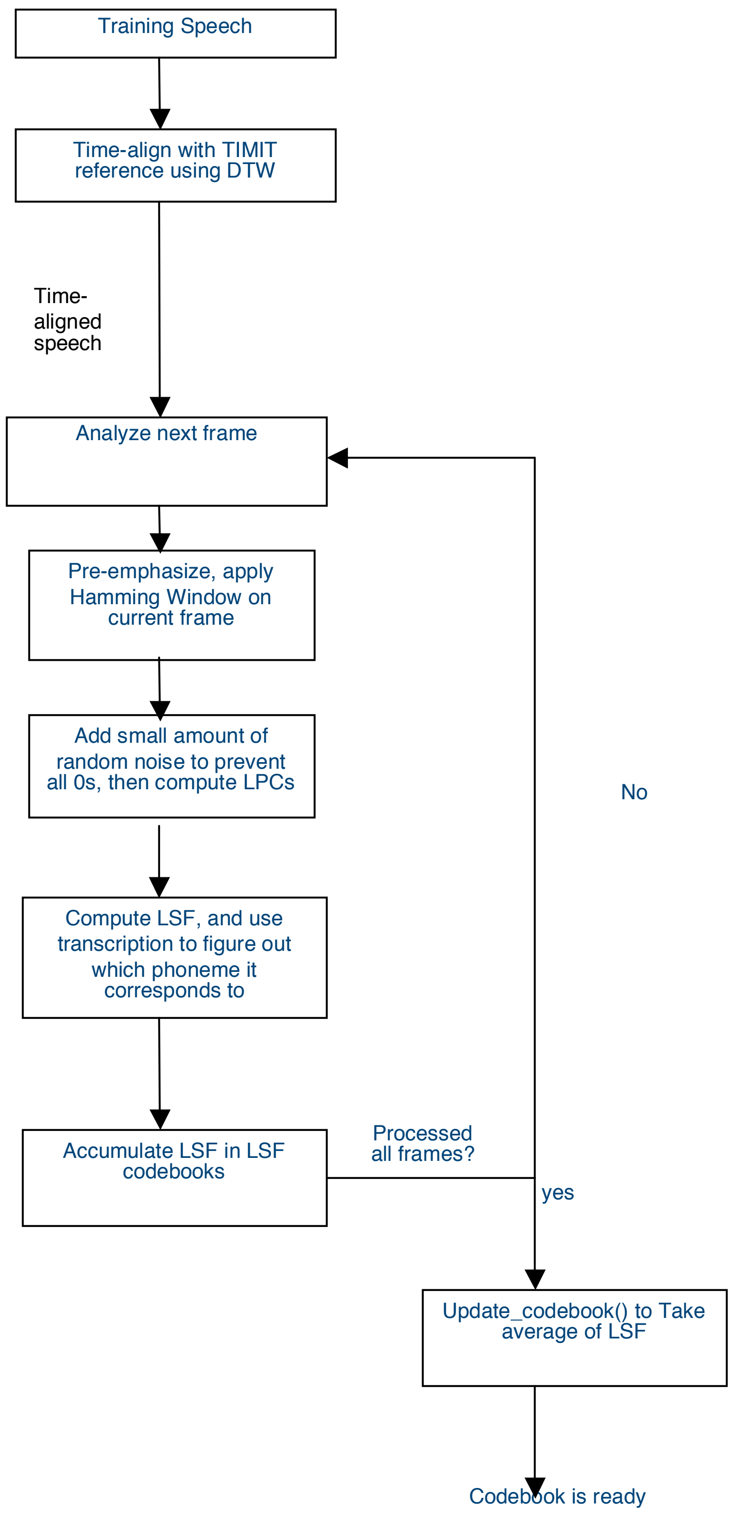
DTW accommodates differences in timing between sample words. The basic principle is to allow a range of 'steps' in the space of (time frames in sample) and to find the path through that space that maximizes the local match between the aligned time frames, subject to the constraints implicit in the allowable steps. The total `similarity cost' found by this algorithm is a good indication of how well the samples match. We implemented this in MATLAB and used the code available on web [3].

2.1.4 Extraction of LSF and excitation spectrum

We divide the input speech signal into frames consisting of 256 floats each. We then perform a 16th order LPC analysis to obtain the LPC coefficients for each frame. We use these LPC coefficients to calculate the LSFs for each frame. We also calculate the excitation spectrum by using the LPC coefficients. As we know the transcription of each sentence we exactly know where the phonemes start and end. We use this information and the ceiling function to find the start and end points of each phoneme. For example, if the phoneme starts at the 500th sample, we ignore the first 12 samples and record the LSF after 512th sample. If the phoneme ends at the 1050th sample, we end it at the 1024th sample. We record the corresponding LSF and store them in the codebook along with the excitation spectrum. The average LSF for each phoneme is then calculated; this is our representation of a phoneme for a speaker.

2.1.5 Summary

The following flowchart shows our implementation of the training part:



Training Speech
Time-align with TIMIT reference using DTW
Time-aligned speech
Analyze next frame
Pre-emphasize, apply Hamming Window on current frame
Add small amount of random noise to prevent all 0s, then compute LPCs
Compute LSF, and use transcription to figure out which phoneme it corresponds to
Accumulate LSF in LSF codebooks
Processed all frames?
No
yes
Update_codebook() to Take average of LSF
Codebook is ready

2.2 Transformation Part

2.2.1 Overview

In the transformation part, the LSF of the input speech signal are computed frame by frame and approximated by a weighted combination of LSF vectors from the source LSF codebook. The output speech is obtained by utilizing the vocal tract generated with the target LSF and the target speech signal.

2.2.2 Pre-emphasis

The input signal is pre-emphasized using the filter $P(z) = 1 - 0.95z^{-1}$. Pre-emphasis increases the energy in parts of the signal by an amount inversely proportional to its frequency. Thus, pre-emphasis helps in amplifying the higher frequencies. This process therefore serves to flatten the signal so that the resulting spectrum consists of *formants* of similar heights. (*Formants* are the highly visible resonances or peaks in the spectrum of the speech signal, where most of the energy is concentrated). The flatter spectrum allows the LPC analysis to more accurately model the speech segment. Without pre-emphasis, the linear prediction would incorrectly focus on the lower-frequency components of speech, losing important information about certain sounds.

2.2.3 Windowing

The pre-emphasized output is then multiplied with a hamming window function given by:

$$s(i) = 0.54 - 0.46 \times \cos\left(\frac{2\pi \bullet i}{n-1}\right) \qquad 0 \le i \le 255, \qquad n = 256$$

The beginning and end of the signals are tapered to zeros. This helps in removing any discontinuities and hence, results in a smoother transition of the signal frame to frame.

2.2.4 Input LPCs and LSF

After applying the hamming window, 16 LPC coefficients are calculated. The LPC coefficients are obtained by minimizing the sum of the squared components of error e(n) between the original speech signal and estimated speech signal. We used the Levinson-Durbin algorithm to calculate the LPCs. The LPC coefficients are then converted to line spectral frequencies (LSF).

We initially used SPTK (Speech Processing Tool Kit) to calculate the LPC coefficients and LSF. However, sometimes the LPC coefficients we obtained were very inaccurate. Also, the code for the SPTK was unwieldy and involved hidden function calls that we suspected were not using memory efficiently and sometimes gave unexpected results in our implementation. Foreseeing having trouble in importing the libraries used by SPTK to DSK, we decided not to use SPTK for our project. Instead we used the code written by Spring'04 Group for calculating the LPC coefficients and converting them to LSF.

2.2.5 Estimation of weights

The input LSF vector 'w' is then approximated by a linear combination of the LSF vectors 'Si' of the source codebook. This also helps us in estimating the phonemes that are not present in the codebooks. The method used for estimating the weights 'v_i' is given in [2].

$$h_k = \frac{1}{\min\left(\left|w_k - w_{k-1}\right|, \quad \left|w_k - w_{k+1}\right|\right)} \qquad k = 1,2,\ldots,16$$

The distance 'd_i' corresponding to each codebook is based on the idea that closely spaced linear spectral frequencies which are likely to correspond to formant frequencies are assigned higher weights.

$$D_i = \sum_{k=1}^{16} h_k \times \left|w_k - s_{i,k}\right| \qquad i = 1,2,\ldots,60$$

Based on the distances from each codebook entry, an approximate linear spectral frequency vector can be expressed as a weighted sum of source codebook linear spectral frequencies.

$$v_i = \frac{e^{-\gamma d_i}}{\sum_{t=1}^{L} e^{-\gamma d_t}} \qquad i = 1,2,\ldots,60$$

In [2], L. Arslan reported that the value of ☐ varies from 0.2 to 2. However to simplify the algorithm, we used a constant value for ☐ After several tests, we concluded that ☐ = 2 gives the best approximation of the input LSF vector.

2.2.6 Target LSF and LPCs

As the codebooks are created with the same sentences for every speaker, the **i**th phoneme in the source codebook should correspond exactly to the **i**th phoneme in the target codebook for the conversion (one to one mapping). Therefore, these weights are applied to the LSF vectors $\mathbf{T_i}$ of the target codebook to obtain the corresponding target LSF vector $\mathbf{w_t}$ of the current speech-frame.

$$w_t = \sum_{t=1}^{60} v_i \times T_i \qquad i = 1,2,\ldots,60$$

These target LSF are then converted back to target LPC coefficients using the code written by Spring'04 group.

2.2.7 Target Vocal Tract Mapping

The estimated target LPC coefficients are used to model the vocal tract of the target speaker. In [xxx], the vocal tract filter is expressed as

$$V = \left| \frac{1}{1 - \sum_{k=1}^{16} a_k e^{-jkw}} \right|^{\beta}$$ where a_k are the LPC coefficients.

The Spring'04 group mentioned that value of ☐ is not important as the human ear is insensitive to phase of the vocal tract. Hence, we decided to assign ☐ as 1 to reduce the complexity of the formula and save memory (saving real values compared to complex floats).

2.2.8 Glottal Excitation Mapping

In [1], for transformation of glottal excitation, the weights that we obtained earlier are used to construct a glottal excitation filter which is a weighted combination of excitation codebook filters.

$$H_g = \sum_{i=1}^{60} v_i \frac{U_i^t(w)}{U_i^s(w)}$$

Following the method described above, we obtained an speech which was more like a source speaker. The group in Spring'04 also mentioned this problem and came up with a solution to rectify this problem. They decided to modify the way in which they had transformed the source to the target excitation. They did this by representing the target excitation as a linear combination of the excitation spectra in the target speaker's codebook:

$$G_t(w) = \sum_{i=1}^{60} v_i \times U_{i,t}(w)$$

They mentioned that the quality of the output speech was very bad but it sounded more like the target speaker compared to the source speaker. However, the output was robotic.

We tried another method to overcome this problem. We used the excitation spectra of the source:

$H_g = E_s$, E_s are FFT of the source excitation

The output speech using this glottal excitation mapping had a much better quality. The speech was very easy to understand unlike the output speech from Spring'04 group. Also, this helped us in saving a lot of space in internal memory on DSK. Now, we could move our codebooks to internal memory as we aren't using the target excitation spectra. Thus, this not only helped in improving the quality of the output speech but it also helped us in improving the efficiency and speed of our voice transformation algorithm.

2.2.9 Output Speech in Frequency Domain

The output speech is obtained by applying the vocal tract and glottal excitation filters to the magnitude spectrum of the input speech signal. This gives us an estimate of the DFT correspondence to the target speech signal.

$$Y(w) = H_g(w) \bullet H_v(w)$$

The Spring'04 group implemented the transformation in time domain. They obtained the time domain signals for glottal excitation and vocal tract mapping by applying inverse FFT of these two filters and then used a difference equation to get the output in time domain:

$$y(n) = g_t(n) + \sum_{k=1}^{16} a_{kt}\, y(n-k)$$

They chose to combine the vocal tract and the excitation in time domain because it required less operations and the implementation was very fast. However, as we already had the glottal excitation and vocal tract filters we decided to find the magnitude spectrum of the input speech signal and obtain the out in frequency domain first.
We then obtain the output speech in time domain by applying inverse DFT on Y(w)

$$y(n) = \text{real}\{\text{IDFT}\{Y(w)\}\}$$

2.2.10 De-emphasis

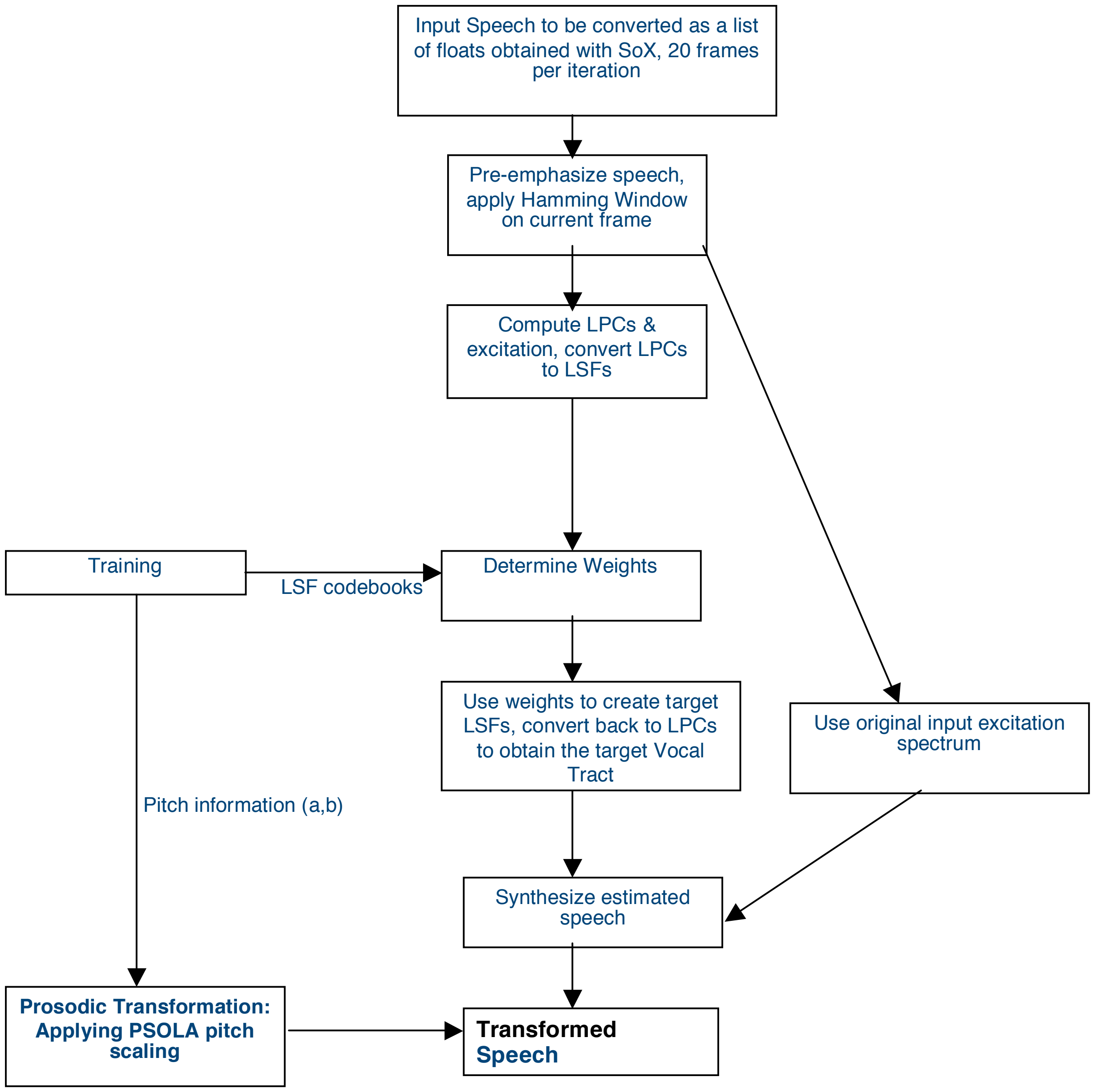
Finally, the pre-emphasis we applied earlier to aid in calculating LPCs is removed from the output speech signal by applying the inverse pre-emphasis filter:

$$P(z) = \frac{1}{1 - 0.95z^{-1}}$$

Before the output is played or sent for post-processing, we scaled the output according to the energy ratio of input frame to the output frame.

2.2.11 Summary

The following flowchart shows our implementation of the transformation part:


Input Speech to be converted as a list of floats obtained with SoX, 20 frames per iteration
Pre-emphasize speech, apply Hamming Window on current frame
Compute LPCs & excitation, convert LPCs to LSFs
Training
LSF codebooks
Determine Weights
Use weights to create target LSFs, convert back to LPCs to obtain the target Vocal Tract
Use original input excitation spectrum
Pitch information (a,b)
Synthesize estimated speech
Prosodic Transformation: Applying PSOLA pitch scaling
Transformed Speech

2.3 Post Processing Part – Prosodic Transformation

We implemented Pitch-Synchronous Overlap Add (PSOLA) to improve the quality of the output speech. PSOLA is a method used to manipulate the pitch of a speech signal to match it to that of the target speaker. In addition to the spectral transformation done in the transformation part, pitch of the output speech is also modified to mimic target speaker's prosodic characteristics.

2.3.1 Pitch Scale Modification

The pitch modification algorithm involves matching the average pitch and range of the target speaker. We can represent instantaneous target speaker's fundamental frequency linearly in terms of source speaker's fundamental frequency.

$$f_t(t) = af_s(t) + b$$

where,

a = $\sqrt{\frac{\sigma_t^2}{\sigma_s^2}}$ ratio of target and source speaker's pitch variance

b = $\mu_t + a\mu_s$ where ☐s and ☐t represent the source and target mean pitch values.

The instantaneous pitch scaling modification factor (☐ t) can be set as

$$\beta(t) = \frac{af_0^s(t) + b}{f_0^s(t)}$$

☐ affects the gender identity of the transformed voice. In general, a ☐ lower than 1 makes the voice sound deeper, even though the fundamental frequency is the same. We found that a good estimate for ☐ which affects the way the formant frequencies are modified, is the midpoint between 1 (no change) and ☐ as the output is very sensitive to changes in ☐ Values of ☐ over 1.5 invariably make the output speech sound like a chipmunk from a children's cartoon, and values below 0.8 make it sound like it was not human.

2.3.2 PSOLA on the Output Speech

We obtained the publicly available MATLAB code from [6]. Our group tested the PSOLA algorithm in MATLAB first. We tried various values for ☐ and ☐ We observed a significant change in the pitch of post processed speech. By just setting some fixed values for ☐ and ☐ we were able to directly change the original male speech signal to female speech signal and vice-versa. After being confident, that PSOLA was robust enough to work well on most of the speech signals, we implemented the algorithm in C.

We tested the C code and it gave us approximately the same output as the MATLAB code.

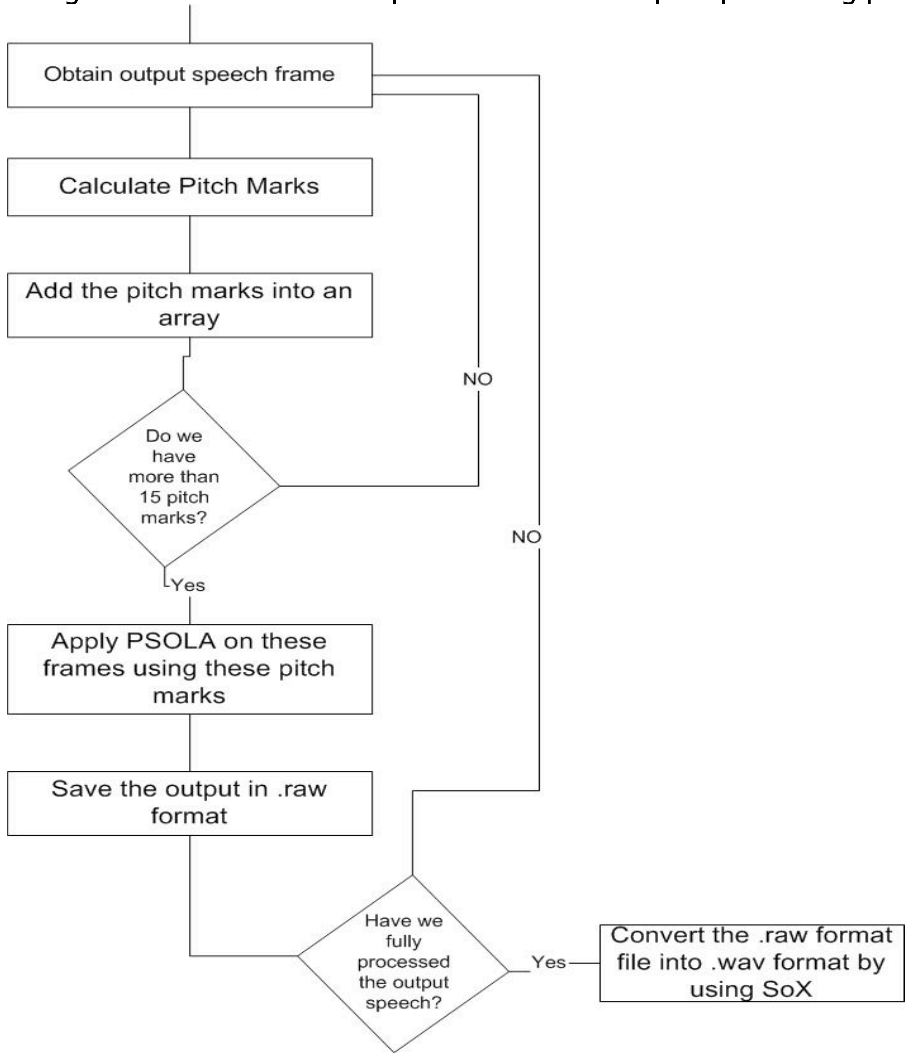
PSOLA calculates the pitch marks in the speech signal by finding the local maxima in an analysis block. The difference between two pitch marks basically represents one pitch period. To improve on the algorithm, we tried to break down the speech signal into smaller chunks and apply PSOLA on each of them before combining the output speech signal into a .wav file. We observed a significant improvement in the output. We further tried to break down the signals to as small as possible to get the minimum number of pitch marks required for PSOLA to work properly. We noticed that PSOLA worked well as long as we had more than 15 pitch marks in the input speech signal. So, while doing the transformation part, we kept track of the pitch marks and as soon as we obtained 16 pitch marks, we estimate the values of ☐ and ☐ and apply PSOLA to that part of the output signal.

2.3.3 Final Output speech

The obtained post processed speech signal is then converted to the playable .wav format by using the SoX program.

2.3.4 Summary

The following flowchart shows our implementation of the post-processing part:



Obtain output speech frame
Calculate Pitch Marks
Add the pitch marks into an array
Do we have more than 15 pitch marks?
NO
Yes
Apply PSOLA on these frames using these pitch marks
Save the output in .raw format
NO
Have we fully processed the output speech?
Yes
Convert the .raw format file into .wav format by using SoX

2.4 Attempted Improvements

Over the course of this project we tried many ways to improve both the sound quality of the output speech, and the similarity of the output speech to the target speaker's speech.

- We tried varying the order of LPC/LSF analysis, but it turned out that 16 was the sweet spot in terms of getting good representative LPCs. Using too many LPCs will result in spurious peaks, while too few will cause formants to be lost. As a rule of thumb, 16 LPCs is a good number for our sampling rate of 16kHz.

- We also tried incorporating glottal excitation transformation filters into our project design as suggested in [1], but the quality became a lot worse, with no apparent benefits.

- Finally, on the day of the demo, we tried a completely new method suggested by Professor Casasent. The details of that method are documented below in the Results section of this report.

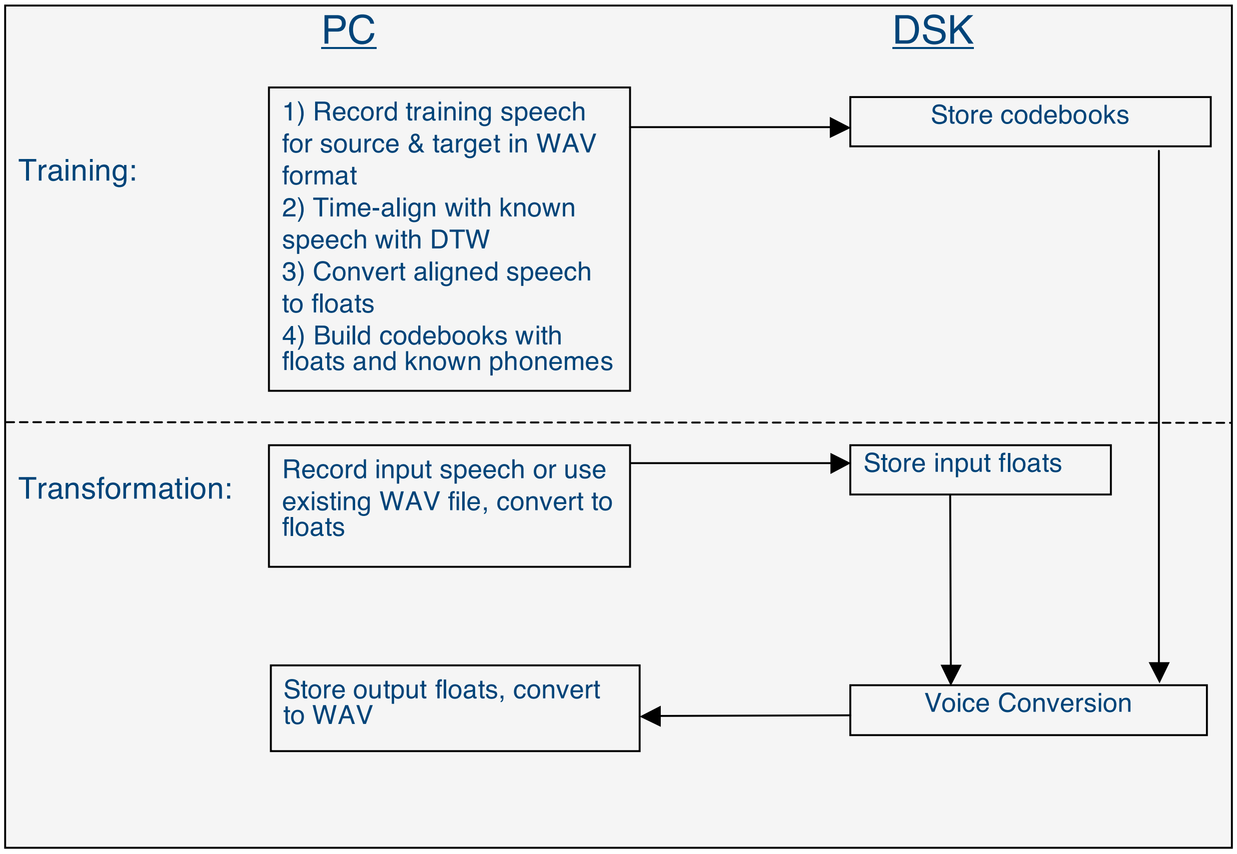
3. Data Flow

To reduce the complexity of the project, the PC was responsible for the training phase of our voice conversion system. We did this as the quality of the voice conversion was our top priority, given the difficulty previous 18-551 project groups had with such projects, instead of more mundane things like user interfaces. Using the PC for the training phase meant that we could utilize existing MATLAB routines to expedite the development of the user I/O for the training phase. This in turn allowed us to concentrate more on the voice conversion itself. Using the PC also made for easier debugging, which was something we had to do a lot over the course of the project.

Training phase
The training speech is recorded at a sampling rate of 16kHz at 16 bits/sample, time-aligned using DTW to reference sentences and then converted using Sound eXchange (SoX) to single-precision floating point numbers. Alternatively, the user can supply his or her own existing phoneme-transcribed WAV files for training purposes, and then use SoX to convert it to a binary file of 16-bit, 16kHz floats. The PC-side program will then build codebooks based on the known location of the phonemes (using the default reference transcriptions in the first case, and user-specified transcriptions in the second case), and then transfer the codebooks over to the DSK for processing. The DSK then receives the codebooks and stores it in the on-chip memory.

Transformation phase
The user starts the transformation phase by specifying a WAV file from the source speaker to be the input speech to be transformed. The WAV file is converted by SoX to a binary file of 16-bit floats at a sampling rate of 16kHz, which is then sent over to the DSK for voice conversion processing. After the DSK has processed the speech and done

the voice conversion, it sends the transformed speech back to the PC in the same binary float format that it received the input speech in. The PC writes the transformed speech without conversion to disk. Therefore, to listen to the output, conversion back to the usual WAV file format needs to be done using the SoX program.


PC
DSK
Training:
1) Record training speech for source & target in WAV format
2) Time-align with known speech with DTW
3) Convert aligned speech to floats
4) Build codebooks with floats and known phonemes
Store codebooks
Transformation:
Record input speech or use existing WAV file, convert to floats
Store input floats
Store output floats, convert to WAV
Voice Conversion

Data Flow Between PC and DSK

4. Speed and Memory Issues

4.1 Memory

Memory Summary:

Item	Size in bytes
ONCHIP_RAM	
Program Code	0x13a60 = 80,480 bytes
Source Codebook	17 coefficients * 4 bytes/float = 68 bytes
Target Codebook	68 bytes
EDMA paging buffer	20 frames * 256 samples/frame * 4 bytes/sample = 20,480 bytes
SDRAM	
4s of Input speech	16kHz*4s*4 bytes/sample = 256,000 bytes
4s of Output speech	256,000 bytes

Data Size:

Compared to the Spring 2004 group's project, our project takes up much less memory. This was because of two design decisions we made. Firstly, we stored floating-point numbers in a raw format and saved them in binary files. The previous group, however, saved their floating-point numbers in a *text* format. While our format only requires 4 bytes to represent one single-precision floating-point number, theirs required up to 12 bytes for one number, since each digit was represented by an ASCII character, which takes up one byte. Secondly, we did away with the excitation codebooks, electing instead to use the input excitation. This saved us 60 phonemes * 256 floats * 4 bytes/floats = 61,440 bytes per speaker. A consequence of this was that the codebooks could now be stored on the on-chip memory.

a) Codebooks
This is a comparison of codebook sizes for 1 speaker for our system:

- i) Storing excitation information and LSFs (initial approach):
(17 LSFs + 256 excitation values) * 4 bytes/ LSF * 60 phonemes = **65 kB**

- ii) Storing just LSFs (final implementation):
(17 LSFs) * 4 bytes/LSF * 60 phonemes = **4 kB**

b) Input Data
1 frame of 256 samples of audio takes up 256 samples * 4 bytes/sample = 1kB/frame.
In this project, we deal with speech files of approximately 4s in length at a 16kHz sampling rate, i.e. 16kHz*4s = 64k samples. The average speech data thus takes up about 64k samples * 4 bytes/sample = 256kB of memory, and will fit on the onboard RAM without any problems.

4.2 Speed

After profiling individual functions, we discovered that the main bottleneck was the *findFFT_mag()* function, which is called 3 times inside *process_a_frame()*, and not the prosodic transformation as we had previously suspected. With that in mind, we set about doing the following optimizations:

1) Using the optimized TI FFT routine introduced in Lab 2, instead of the FFT code that the previous group used. This resulted in a savings of 40,000 cycles, or 25% of the cycle count of the original findFFT_mag() code at –o3 optimization.

2) Using EDMA paging to page in 20 frames of input speech at a time.

3) Storing the codebooks on internal on-chip memory instead of external memory for faster access.

Other optimizations that could have been done, but were not implemented due to time constraints, include:

i) Precomputing the Hamming window, and storing it in a lookup table on on-chip memory.

ii) Ping-pong buffering to do data transfers while doing the computations for the voice transformation before prosodic transformation.

Our optimizations resulted in a reduction in cycles needed by approximately 37%.

Profiling Summary:

	Without optimization	Full optimizations (-o3, EDMA paging, TI code,etc.)
process_a_frame(): Does voice transformation without prosodic transformation on a single frame	1,750,518	1,101,423
prosodic_transformation(): Does prosodic transformation on the whole array that is the result of calling process_a_frame() on all input data	Total: 81,398,713 Amortized cost per frame: 442,384	Total: 76,526,030 Amortized cost per frame: 415,902
process_a _frame() subroutines		
findFFT_mag()	216,000	110,000
emphasis()	10,204	10.084
de_emphasis()	10,881	10,890
signal2Lpc()	104,566	101,151

5. Results

Demo 1

In the lab demo, we first demonstrated using our system to convert speech from one female speaker in the TIMIT database, to a male speaker in the TIMIT database, and vice versa (i.e. the source could either be the female or the male, and the target would be the other speaker), and we showed the results with and without prosodic transformation The training set was a set of sentences from each speaker from the TIMIT database, and the test set was a sentence from the source speaker that was not in the training set.

The result of this first demonstration did not have the clarity and similarity to the target speaker as we had hoped for, but at least it sounded like speech, albeit from some unknown speaker. Compared to our implementation of the previous group's (Spring 2004) algorithm, the sound quality of the output was definitely better because the excitation signal came from actual speech instead of being a weighted sum of excitations. With prosodic transformation turned on, the pitch of the output speech sounded more like the pitch of the target speaker, but one still could not say that the output speech sounded like it came from the target speaker.

Demo 2

After listening to the above results, Professor Casasent then suggested 2 things to improve our project: 1) checking the phoneme segmentation accuracy of our system, and 2) a new approach which involved storing signals of whole phonemes of the target speaker for a better quality output. We then worked overnight on a MATLAB implementation of this concept that would strive to give insights on these 2 issues.

Basically, our new implementation only stored target phonemes. Instead of doing phoneme segmentation, we went one step further: use input speech which we know the phoneme transcription of (courtesy of the TIMIT database), and replace, phoneme by phoneme, input speaker phonemes with target speaker phoneme signals stored in the training phase. This killed two birds with one stone: we would see the result of the system with *perfect* phoneme transcription, using the new approach Professor Casasent had suggested. If even with perfect phoneme transcription, the new approach failed, then incorporating it into our system with less-than-perfect phoneme transcription would be futile.

As it turned out, the result of the above implementation was a choppy output speech. The words were almost impossible to recognize due to the sudden changes in pitch and phoneme duration. Sentences usually follow a predictable pitch contour and cadence, with the pitch and speed of each phoneme partially dependent on the tone of the sentence, but in this case the pitch and speed of each phoneme was arbitrary since they were taken from different training sentences. We postulated that detailed pitch and duration analysis could have been applied to these phonemes within the PSOLA framework to try to synthesize natural-sounding speech, but did not implement that as that would have been enough for a whole new 18-551 project!

6. Discussion and Future Work

Although we reused the input speaker's excitation signal in the synthesis of speech, the output speech quality was still vastly different from the quality of the input speech. The words were intelligible but the voice speaking it was gravelly and of indeterminate identity. One reason for the poor synthesis of speech was that the representation of a speaker's phoneme by taking the average LSFs per frame was not good enough. Analysis of the whole phoneme, instead of breaking up the phoneme into frames is required to capture a good quality representation of every phoneme. This may require the algorithm to adapt to different lengths of representative codebook entries, and can increase the complexity.

Also, the TIMIT database, despite providing ten phonetically-diverse sentences for each speaker, does not have enough data to enable us to capture phonemes pronounced in different contexts. For example the word "really" can be pronounced very differently, depending on whether the speaker is using a normal tone, or a sarcastic tone. The signal properties of the word and its phonemes would differ greatly depending on which context it was spoken in.

We also found that enabling the prosodic transformation helped in terms of scaling the fundamental frequencies and formants to be more similar to that of the target speaker's. Female to male conversions generally sounded more natural, while male to female conversions sounded more like male to child conversions. Synthesizing female voices is difficult as empirical evidence has shown that they are harder to model.

Arguably, our group was among the ones which spent the most number of hours in the 18-551 lab, due to the wide scope of the project. Getting a working implementation of the algorithm not even including prosodic transformation itself was a whole 18-551 project in Spring 2004, and we not only had to do that, but to also incorporate pitch detection of speech and pitch shifting into our system, which in itself is a difficult research problem. In retrospect, we could have focused on certain areas of the project, each a possible project in their own right, and traded away breadth for depth. Areas we identified that could be the basis of further work include:

1) Robust automated pitch marking for speech:
The quality of the pitch marking of the estimated speech left much to be desired. This was because of the simple algorithm we used, as well as the signal quality of the estimated speech. Pitch marking of speech is an active research topic, and a good pitch marking system will improve the estimates of the source and target speaker fundamental frequencies.

2) Voice transformation of good quality speech by using pitch analysis:
One of the ideas we had was to find the average pitch of each phoneme a speaker pronounces and use that in our analysis, but unfortunately we did not have enough time to come up with an algorithm robust enough for our purposes to detect pitches in speech. Transformation of an input speech based on detecting the difference in phoneme pitches between source and target, while preserving the original pitch contour

of the input pitch, or even somehow estimating the pitch contour for the target speaker (estimating how a target speaker's accent could affect the way he spoke the sentence!) could be the basis of future work.

3) Phoneme Segmentation: In the 2nd demo that we did, we established that phoneme segmentation was not the main problem in this project. Nevertheless, incorporating a phoneme segmentation algorithm will remove the dependency on DTW, and allow for more flexibility in the training part.

7. References

1) "Voice Conversion by Codebook Mapping of Linear Frequencies and Excitation Spectrum", L. Arslan & D. Talkin, in Proc. EuroSpeech'97, pp 1347 -1450.

2) "Speaker Transformation Algorithm using Segmental Codebooks (STASC)", L. Arslan, Technical Report, Washington D.C., 1998

3) "Dynamic Time Warp (DTW) on MATLAB", Dan Ellis, Columbia University.

4) SPTK Speech Processing Toolkit., Nagoya Institute of Technology, Japan : http://kt-lab.ics.nitech.ac.jp/~tokuda/SPTK/

5) "Text Independent Voice Transformation", 18-551 Group 18, Spring'99

6) Implemented PSOLA code in MATLAB http://sheldon.hygred.com/elec484/project/index.htm

8. Schedule and Assigned Tasks

Week	Date	Work done
1	Oct 15th – 21st	All of us read the papers by Kain and Arslan and tried to understand exactly what needs to be done for voice transformation.
2	Oct 22nd – 28th	Teng Ji worked on understanding the SoX program, Marcus started writing the code for training part, and Raman worked on SPTK and tried to understand their implementation for calculating LPC coefficients and LSF .
3	Oct 29th – Nov 4th	Teng Ji tested our output for training phase to the one obtained with MATLAB and read the DTW algorithm, Marcus finished writing the training part and edited parts of the code related to codebook mapping, Raman tested the SPTK code with MATLAB and ported the code into our algorithm.
4	Nov 5th – 11th	We all prepared the presentation slides. Teng Ji adapted the DTW code into out algorithm, Marcus finished building the codebooks and started coding for the transformation part, and Raman wrote the functions for hamming, windowing, glottal excitation and vocal tract mapping.
5	Nov 12th – 18th	After looking at the output, we met Professor Casasent and he suggested to us various ways to improve our output speech quality. Marcus took out the code for target excitation and tweaked his code to follow this new algorithm, Teng Ji researched on other ways we could improve the output quality and Raman looked into the PSOLA references to find some code for PSOLA implementation.
6	Nov 19th – 25th	Marcus imported the online available code for LPC and FFT into our project after we decided not to use SPTK anymore, Teng Ji and Raman tested the PSOLA code in MATLAB and started writing the DSK code for transfer functions.
7	Nov 26th – Dec 2nd	Raman finished writing the PSOLA code in C and tested it with MATLAB output. Marcus redesigned the code into PC and DSK-side code. Teng Ji worked on profiling our implemented functions.
8	Dec 3rd -9th	We all worked together to prepare the slides for our oral presentation and wrote the final report. Marcus and Teng Ji tweaked the PC-side code, and wrote a GUI in MATLAB so that it could take in a speaker's voice from a microphone connected to the computer for the training part.