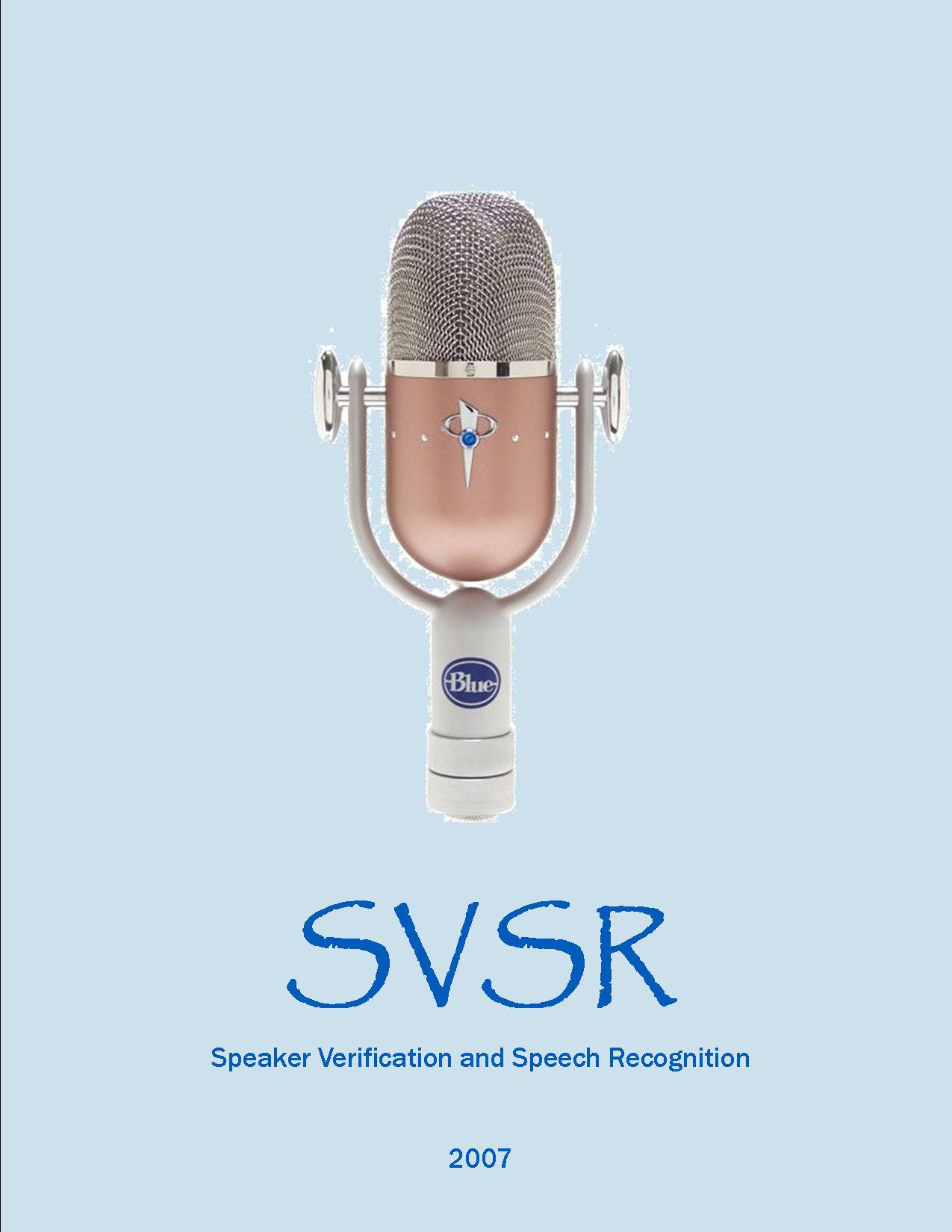
****

**بسم الله الرحمن الرحيم**

**SVSR Staff**

**Loai Qawareeq**

**Aladdin Sweidan**

**Thanks to:**

* **Dr. Loai Malhis**
* **Dr. Raed Al Qadi**
* **Instructor Haya Samaneh**

**And thanks to**

**Lab Supervisors: Othman Othman and Aladdin Menwer.**

**For their great help, allowing us to use labs facilities.**

**And thanks for our friends who stood by our side throughout this project.**

**Introduction**

**Our SVSR Project Is Mainly Divided Into:**

* Text-Dependent Speaker Verification
* Speaker-Independent Command Recognition

**Speaker Identification:**

* What?

Speaker Identification is the process of confirming or verifying person identity by his speech. Our Text-independent speaker identification system aims to recognize the previously-trained users when they say the word(s) they already trained on.

* Why?

Speech in general is becoming more and more a developing area of research; there are several possible applications for speaker verification in security and identification systems, for example: Home Access, Car Access, Automatic Teller Machines.

* How?

The main idea is simple. Train the user and extract his/her “voice characteristics”. When the user comes back for confirming identity then it is just a process of comparing the live data with the stored data and deciding the best candidate. This is can be referred to as the general approach, and as everybody knows… Devil in the Details!!

**Speech Recognition:**

Speech recognition can be classified into 3 different types:

1. Speaker-Dependent
2. Speaker-Adaptive
3. Speaker-Independent

**Speaker dependent:** A speaker dependent system is developed to operate for a single speaker. These systems are usually easier to develop, and more accurate, but not as flexible as speaker adaptive or speaker independent systems.

**Speaker Adaptive:** A speaker adaptive system is developed to adapt its operation to the characteristics of new speakers. Its difficulty lies somewhere between speaker independent and speaker dependent systems.

**Speaker Independent:** A speaker independent system is developed to operate for *any* speaker of a particular type (e.g. American English). These systems are the most difficult to develop, most expensive and accuracy is lower than speaker dependent systems. However, they are more flexible.

Our SVSR is a speaker-independent command recognition system. It aims to make the windows environment automated by human voice. We applied our algorithms on 15 windows commands. Enjoy!

**Chapter I: Matlab and Visual Studio**

Signal processing needs huge capabilities of math manipulation, specifically matrix manipulations, and Matlab is there to provide these capabilities. But, Matlab does not have advanced techniques in creating a GUI windows application, instead Visual Studi presents these features. So, the idea was to get advantage of matlab processing features and in the same time the programming and graphical features of Visual Studio.

So, how to integrate matlab with visual studio?

In general, there are 3 solutions we encountered during research period, these include:

1. **COM approach**

MATLAB exposes itself as a COM Automation [Server](http://www.codeproject.com/KB/dotnet/matlabeng.aspx), so we can use the .NET COM Interoperability to use its functions: we can work directly using a Type Library or using only the Dispatch interface. The progid of the server is Matlab.Application.

1. **DDE: Dynamic Data Exchange**

The Dynamic Data Exchange is a quite old but powerful service of Windows that lets applications communicate and exchange data. It's the ancestor of the whole COM and so it's the grandfather of the whole .NET. Again, as with the COM solution, we have a client, our .NET program, and the server, the MATLAB DDE Server, that communicates.

A DDE communication is made with a Service Name, like Matlab or WinWord, about a topic of discussion, like System or Engine, and is based on the exchange of data elements called items.

The problem with the DDE exchange is the data types used to exchange information, because it uses the Windows clipboard data types. Specifically MATLAB supports:

* Text for operation results, and matrix data exchange (gulp!);
* Metafilepict for image transfer;
* XLTable for MATLAB-Excel data exchange;

1. **C/C++ Shared Library [ DLL ]**

The direct access to the MATLAB C API is the best solution in terms of performance and features, just let use P/Invoke and some unsafe pointer operations.

The C library internally uses Unix pipes or the COM to communicate with the main MATLAB instance, but in this case the data transfer is done using blocks of raw memory instead of the SAFEARRAY. So we can use the unsafe capabilities of C# to transfer efficiently matrices from and to MATLAB, it's just necessary to remember that MATLAB stores data in column wise order.

WE CHOSE TO USE THE LAST SOLUTION due to its read/write speed compared to the DDE or COM.

**Steps to create and use the library:**

The following steps summarize the process of creating a C shared library and calling it from a Windows Forms Application.   
  
**1. In MATLAB, select a compiler with the MBUILD command.**mbuild -setup  
**2. Compile your M-files into a C shared library with the MCC command.**

mcc -B csharedlib:mylib <M-files>

The -B csharedlib option is a bundle option that expands into -W lib:<libname> -T link:lib. The -W lib:<libname> option tells MATLAB Compiler to generate a function wrapper for a shared library and call it libname. The -T link:lib option specifies the target output as a shared library.   
  
**3. Add the MCR library, your library, the library path and the include directories to your MSVS project. Copy the MATLAB generated CTF- and DLL-files to your project directory.  
  
4. Call the MCR (MATLAB Component Runtime) and library initialization and termination routines from your project code.** To initialize the shared library and the MCR, call:  
mclInitializeApplication(NULL,0);  
mylibInitialize();  
After all the calls to your shared library functions, call:  
mclTerminateApplication();  
mylibTerminate();  
5. Build and run your project in MSVS.

The following detailed steps create an example C shared library that is called from a Microsoft Visual Studio 2005 project.   
  
**1. Save the following MATLAB code to a file called "foo.m":**function y = foo(x)  
y = x\*10;  
**2. Execute the MBUILD command at the MATLAB prompt:**mbuild -setup  
The version number of the Microsoft Visual Studio 2005 C/C++ Compiler is 8.0. Select this compiler.  
  
**3. Generate a C shared library from foo.m with the MCC command:**mcc -B csharedlib:foolib foo.m  
This will generate the following files:

foolib.ctf  
foolib.h  
foolib.dll  
foolib.lib  
foolib.exp  
foolib\_mcc\_component\_data.c  
foolib.c  
foolib.exports

**4. In Microsoft Visual Studio 2005, go to File->New Project. In the Project Types field, select Visual C++. In the Templates field, select Windows Forms Application. In the Name field type "test". Click OK.   
  
5. In the Form that is automatically created, add 2 text boxes and a push button. Change the name of one of the text boxes to "inBox" and the other to "outBox". This can be done by right-clicking on the text box and selecting Properties. Edit the Name property to the appropriate value. You do not need to change the name of the push button.  
  
6. Double click on the push button. This will bring you to the Click function for this button. Paste the following code into that function:**

double input = System::Double::Parse(inBox->Text);  
mxArray \*x\_ptr;  
mxArray \*y\_ptr=NULL;  
double \*y;  
  
// Create an mxArray to input into mlfFoo  
x\_ptr = mxCreateScalarDouble(input);  
  
// Call the implementation function   
// Note the second input argument should be &y\_ptr instead of y\_ptr.   
mlfFoo(1,&y\_ptr,x\_ptr);  
  
// The return value from mlfFoo is an mxArray.   
// Use mxGetpr to get a pointer to data it contains.  
y = (double\*)mxGetPr(y\_ptr);  
  
// display the result in the form  
outBox->Text = ""+\*y;  
  
//clean up memory  
mxDestroyArray(x\_ptr);  
mxDestroyArray(y\_ptr);

**7. Open the test.cpp source file. Add the library header import statement**

#include "foolib.h"

before the Form header input statement

#include "Form1.h"

**8. Locate the main function in the test.cpp file. Add the MCR and library initialization function calls before the Form is opened. To do this, paste this code**

/\* Call the MCR and library initialization functions \*/  
if( !mclInitializeApplication(NULL,0) )  
{  
 MessageBox::Show("Could not initialize the application.");  
 exit(1);  
}  
  
if (!foolibInitialize())  
{  
 MessageBox::Show("Could not initialize the library.");  
 exit(1);  
}

just before the line

Application::Run(gcnew Form1());

**9. Add the library and MCR terminate function calls after the call to open the form, but before the return statement. To do this, paste this code**

/\* Call the MCR and library termination functions \*/  
foolibTerminate();  
mclTerminateApplication();

just after the line

Application::Run(gcnew Form1());

**10. Select your project in the Solution Explorer. Select Project -> Properties. Under Configuration Properties->General, change "Common Language Runtime support" from "Pure MSIL Common Lanaguage Runtime Support (/clr:pure)" to "Common Lanaguage Runtime Support (/clr)". Click OK.  
  
11. Under Configuration Properties->Linker->General category select the Additional Library Directories field. Add the location of foolib.lib, and the following directory:**

$MATLAB\extern\lib\win32\microsoft

Note that $MATLAB should be replaced with the MATLAB installation directory, as returned by typing  
matlabroot  
at the MATLAB command prompt.  
  
**12. Under Configuration Properties->Linker->Input category add the following libraries in the Additional Dependencies field:**

mclmcrrt.lib  
foolib.lib

**13. Under Configuration Properties->C/C++ category->General select the Additional Include Directories field. Add the location of foolib.lib, and the following directory:**

$MATLAB\extern\include

replacing $MATLAB with the MATLAB installation directory as above.  
14. Also under the C/C++ category, select Code Generation. In the Runtime Library, select Multi-threaded DLL. Click OK to close the Properties window.  
  
**15. Go to Build->Build Solution. You should now have built test.exe.**  
**16. Place foolib.dll and foolib.ctf in folder where test.exe is located.**  
If you are using MATLAB 7.3 (R2006a) and you want to run the application from within MSVS, you will also need to change the debug working directory to ensure that the CTF-file is found. To do this select Project -> Properties. Under Configuration Properties->Debugging in the Working Directory field, enter the name of the directory where the DLL- and CTF-files are located. For example if they are in the Debug subdirectory of your project, simply enter the text "Debug".

**Our Project:**

We used a main m-file that we transformed into a DLL library. This file include calls to other m-files depending on the task we wanted to perform.

**MCR: Matlab Compiler Runtime**

The Objective is to create MATLAB applications that take advantage of the mathematical functions of MATLAB, yet do not require that end users own MATLAB. Standalone applications are a convenient way to package the power of MATLAB and to distribute a customized application to your users.

The source code for standalone C applications consists either entirely of M-files or some combination of M-files, MEX-files, and C or C++ source code files.

MATLAB Compiler takes your M-files and generates C source code functions that allow your M-files to be invoked from outside of interactive MATLAB. After compiling this C source code, the resulting object file is linked with the run-time libraries. A similar process is used to create C++ standalone applications.

You can distribute a MATLAB Compiler generated standalone to any target machine that has the same operating system as the machine on which the application was compiled.

For example, if you want to deploy an application to a Windows machine, you must use MATLAB Compiler to build the application on a Windows machine. If you want to deploy the same application to a UNIX machine, you must use MATLAB Compiler on the same UNIX platform and completely rebuild the application. To deploy an application to multiple platforms requires MATLAB and MATLAB Compiler licenses on all the desired platforms.

#### For Windows Environment

**Gather and package the following files and distribute them to the deployment machine.**

| **Component** | **Description** |
| --- | --- |
| MCRInstaller.exe | Self-extracting MATLAB Component Runtime library utility; platform-dependent file that must correspond to the end user's platform. This file is located in the *matlabroot*\toolbox\compiler\deploy\win32 directory. |
| magicsquare.ctf | Component Technology File archive; platform-dependent file that must correspond to the end user's platform |
| Magicsquare | Application; magicsquare.exe for Windows |

Chapter II : Speaker Verification

Speaker Verification is confirming the identity of an individual from his speech. There are several possible applications for speaker verification in security and identification systems. Some examples include automatic teller machines and home or vehicle access. An interesting application might even be voice imitation training.

There are several possible implementations for systems that perform speaker verification. The implementation which most commonly comes to mind is that of the matched filter. There are various characteristics of signals which can be matched. These include magnitude, energy, frequency, linear predictive coding coefficients, and cepstrum techniques.

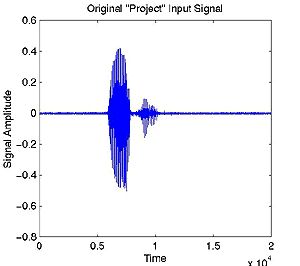
The matched filter is the implementation used for this speaker verification system. The initial consideration for an implementation was to simply matched filter the input waveforms of two signals as opposed to certain characteristics of these waveforms. This implementation consisted of taking two input signals and cutting out the sections of the waveforms where speech existed (based on a visual determination). Then, one waveform was scaled to the length of the other, the two signals were normalized, and the dot product was computed. This implementation presents various, obvious problems. The most significant problem encountered is aligning the phase of the two input signals. This, it turns out, is a very difficult obstacle to overcome. As a result, this implementation was deemed impractical very quickly. The consideration of several other problems experienced in a speaker verification system led to the development of the final implementation.

Problems include the detection of the endpoints of the actual speech in a given waveform and the varying speed or volume of a given speech signal. Several techniques are used to solve the various problems discussed above. A clever endpoint detection algorithm is utilized to determine the beginning and end of speech in any given waveform. Finally, dynamic time warping is used to compensate for the varying speed of a person's speech. This is much more effective than linear scaling to map one signal to another.

Methods and Algorithms used in the project:

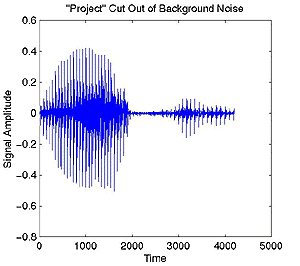
1. End Point Detection

The implementation for the speaker verification system first addresses the issue of finding the endpoints of speech in a waveform. The code which executes the algorithm can be found in the file locatespeech.m. The algorithm finds the start and end of speech in a given waveform, allowing the speech to be removed and analyzed. Our implementation uses this algorithm for the short-time magnitude analysis of the speech. The endpoint detection algorithm is used here, but not in the cutting of the unvoiced regions of the pitch track. It is important to note that this algorithm gives the entire region where speech exists in an input signal. This speech could include voiced regions as well as unvoiced regions. Voiced speech includes hard consonants such as "ggg" and "ddd", while unvoiced speech includes fricatives such as "fff" and "sss". For the short-time magnitude of a speech signal, it is necessary to include all speech which would be located by this algorithm. However, for short-time pitch, one is only concerned with voiced regions of speech. As a result, this algorithm is not used, and instead, we use the energy in the signal to find the voiced and unvoiced regions of the pitch track. This, however, is further developed in the Short-time Frequency section.



The endpoint detection algorithm functions as follows:

1. The algorithm removes any DC offset in the signal. This is a very important step because the zero-crossing rate of the signal is calculated and plays a role in determining where unvoiced sections of speech exist. If the DC offset is not removed, we will be unable to find the zero-crossing rate of noise in order to eliminate it from our signal.
2. Compute the average magnitude and zero-crossing rate of the signal as well as the average magnitude and zero-crossing rate of background noise. The average magnitude and zero-crossing rate of the noise is taken from the first hundred milliseconds of the signal. The means and standard deviations of both the average magnitude and zero-crossing rate of noise are calculated, enabling us to determine thresholds for each to separate the actual speech signal from the background noise.
3. At the beginning of the signal, we search for the first point where the signal magnitude exceeds the previously set threshold for the average magnitude. This location marks the beginning of the voiced section of the speech.
4. From this point, search backwards until the magnitude drops below a lower magnitude threshold.
5. From here, we search the previous twenty-five frames of the signal to locate if and when a point exists where the zero-crossing rate drops below the previously set threshold. This point, if it is found, demonstrates that the speech begins with an unvoiced sound and allows the algorithm to return a starting point for the speech, which includes any unvoiced section at the start of the phrase.
6. The above process will be repeated for the end of the speech signal to locate an endpoint for the speech.



2. Dynamic Time Warping:

What happens when people vary their rate of speech during a phrase? How can a speaker verification system with a password of "Project" accept the user when he says "Prrroooject"? Obviously, a simple linear squeezing of this longer password will not match the key signal because the user slowed down the first syllable while he kept a normal speed for the "ject" syllable. We need a way to non-linearly time-scale the input signal to the key signal so that we can line up appropriate sections of the signals (i.e. so we can compare "Prrrooo" to "Pro" and "ject" to "ject").

The solution to this problem is to use a technique known as "Dynamic Time Warping" (DTW). This procedure computes a non-linear mapping of one signal onto another by minimizing the distances between the two.

In order to get an idea of how to minimize the distances between two signals, let's go ahead and define two: K(n), n = 1,2,...,N, and I(m), m = 1,2,...,M. We can develop a local distance matrix (LDM) which contains the differences between each point of one signal and all the points of the other signal. Basically, if n = 1,2,...,N defines the columns of the matrix and m = 1,2,...,M defines the rows, the values in the local distance matrix LDM(m,n) would be the absolute value of (I(m) - K(n)).

The purpose of DTW is to produce a warping function that minimizes the total distance between the respective points of the signals. We now introduce the concept of an accumulated distance matrix (ADM). The ADM contains the respective value in the local distance matrix plus the smallest neighboring accumulated distance. We can use this matrix to develop a mapping path which travels through the cells with the smallest accumulated distances, thereby minimizing the total distance difference between the two signals.

There are some constraints that we would want to place on such a mapping path. We would not want this path to wander too far from a linear mapping (we don't want to stretch or squeeze the signal too much). We should also say that the path can only map points that haven't been covered yet. For example, we can't map point 6, point 5, and then point 6 again. And for boundary considerations we would want the endpoints of the signals to map to each other (I(1) maps to K(1) and I(M) maps to K(N)).

With these constraints in mind we can develop this concept of an accumulated distance matrix. Above we said that ADM values will be the LDM values at that location plus the smallest neighboring accumulated distances. To keep from backtracking over points that have already been covered, we use the Itakura method to define what these "neighboring" points are. The Itakura method says that neighboring points of the point at (m,n) are (m,n-1), (m-1,n-1), and (m-2,n-1). Defining ADM(1,1) to be LDM(1,1), we can write the equation for the other cells in the accumulated distance matrix:

ADM(m,n) = LDM(m,n) + min{ADM(m,n-1), ADM(m-1,n-1), ADM(m-2,n-1)}

With this matrix developed, we can create the path by starting at the value in ADM(M,N) and working our way backwards to ADM(1,1), travelling through the cells with the lowest accumulated distances subject to the Itakura constraints. In order to stay near a linear mapping, another Itakura constraint says that the path can't stay in the same row for two consecutive cell movements (i.e. we can't go from (m,n) to (m,n-1) and then to (m,n-2)).

Our group added the additional constraint that the path cannot choose to skip two rows twice consecutively. We noticed that this technique slightly improved the quality of the time warping.

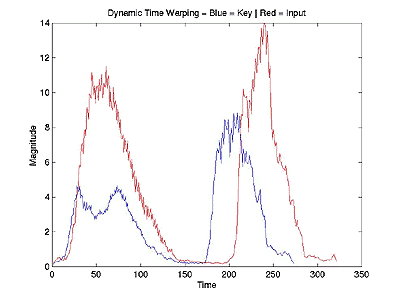
Once the path has been traced out, the signals can be mapped onto each other in the following way:

I onto K:

* warping\_path(i = 1..N) = ADM row number that intersects path at ADM column i
* time\_warped\_I = I(warping\_path)

K onto I:

* warping\_path(i = 1..M) = ADM column number that intersects path at ADM row i
* time\_warped\_K = K(warping\_path)



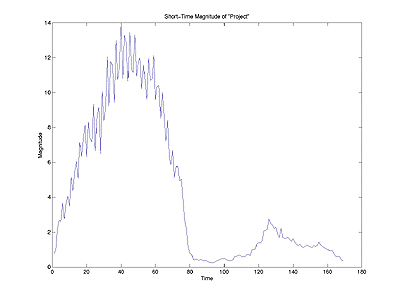
3. Short Time Magnitude:

One portion of the final implementation is the comparison of the short-time magnitude of two speech signals. This operation is implemented in the file magmaster.m.

Using an endpoint detection algorithm, the speech is selected from the two input waveforms. Then, their short-time magnitudes are determined. The short-time magnitude characterizes the envelope of a speech signal by lowpass filtering it with a rectangular window. The magnitude function follows these steps:

* The bounds of the signal are determined and each end is zero-padded.
* The signal is convolved with a rectangular window. As the window is swept across the signal, the magnitude of the signal contained within it is summed and plotted at the midpoint of the window's location. This function, avgmag.m, gives the envelope of the speech.

One magnitude plot discrete time warped onto the other. The dot product of the two waveforms is computed, and this number is divided by the product of the signals' norms as mandated by the Cauchy-Schwarz Inequality. This calculation results in a percentage of how similar one signal is to the other.



4. Short-Time Frequency:

A simple model of the human vocal tract is a cylinder with a flap at one end. When air is forced through the tract, the vibration of the flap is periodic. The inverse of this period is known as the fundamental frequency, or pitch. This frequency, combined with the shape of the vocal tract, produces the tone that is your voice. Variations in people's vocal tracts result in different fundamentals even when the same word is said. Therefore, pitch is another characteristic of speech that can be matched.

To extract pitch from our signals, we make use of a harmonic-peak-based method. Since harmonic peaks occur at integer multiples of the pitch frequency, we can compare peak frequencies at each time t to locate the fundamental. Our implementation finds the three highest-magnitude peaks for each time. Then we compute the differences between them. Since the peaks should be found at multiples of the fundamental, we know that their differences should represent multiples as well. Thus, the differences should be integer multiples of one another. Using the differences, we can derive our estimate for the fundamental frequency.

Derivation:

* Let f1 = lowest-frequency high-magnitude peak  
  Let f2 = middle-frequency high-magnitude peak  
  Let f3 = highest-frequency high-magnitude peak
* Then d1 = f2-f1 and d2 = f3-f2.
* Assuming that d2 > d1, let n = d2/d1. Then our estimate for the fundamental frequency is F0 = (d1+d2) / (n+1).

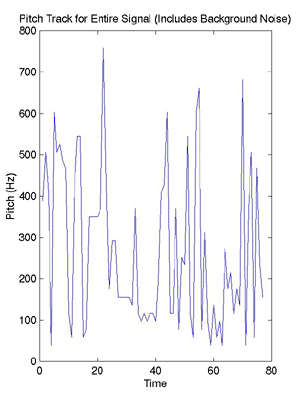
We find the pitch frequency at each time and this gives us our pitch track. A major advantage to this method is that it is very noise-resistive. Even as noise increases, the peak frequencies should still be detectable above the noise. Also, it is easily implementable in MATLAB.

In order to do this, we first find the signal's spectrogram. The spectrogram parameters we have decided to use are a window length of 512 points and a sampling rate of 10000 Hz. We assume that the fundamental frequency (pitch) of any person's voice will be at or below 1200 Hz, so when finding the three largest peaks, we only want to consider sub-1200 Hz frequencies. Thus, we want to cut out the rest of the spectrogram. Before we do this, however, we must use the whole spectrogram to find the signal's energy.

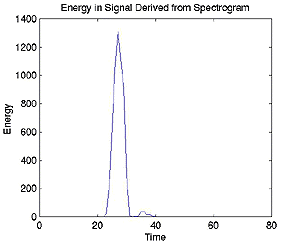
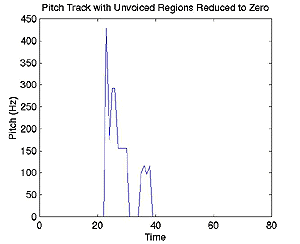
The signal's energy at each time is very important, as it shows the voiced and unvoiced areas of the signal, with voiced areas having the higher energies. Since we are using our pitch track to compare pitch between signals, we want to be certain that we are only comparing the voiced portions, as they are the areas where pitch will be distinct between two different people. A plot of energy versus time can actually be used to window our pitch track, so that we may get only the voiced portions.

To find the energy versus time window, we take the absolute magnitude of the spectrogram and then square it. By Parseval's Theorem, we know that adding up the squares of the frequencies at each time gives us the energy of the signal there. Plotting this versus time gives us our window.

Once this is done, we cut our spectrogram and move on to finding the three largest peaks at each time. A frequency is designated as a "peak" if the frequency directly above and below it both have smaller magnitudes than it does. If a frequency is a peak, then its magnitude is compared to the three magnitude values stored in the "peak matrix" (a matrix of magnitudes and locations for the three highest peaks which start out as zeros at each time). If it is greater than the minimum matrix value, then its magnitude and location replace the magnitude and location of the matrix's smallest peak.



The matrix of peak values and locations at each time is then fed through the fundamental frequency algorithm and we have our uncut pitch track (above)! At this point, we go back to our energy versus time plot and use it to find the energy threshold of the noise and unvoiced areas that we want to cut out of our pitch track. This is done by finding the mean and standard deviation of the very beginning of the signal (assumed to be noise as the person never begins speaking until at least half a second into the recording due to mental processing time) and using these to develop the threshold. Then, the pitch track is windowed with the energy signal, and everything below the threshold is cut out (below). This function, pitch.m, gives a pitch track of the voiced portions of the signal -- it is now ready for comparison with another signal.

Our pitch track comparison program, pitchmaster.m, takes in two signals and finds each of their pitch tracks. It then maps the pitch tracks onto one another using dynamic time warping. After mapping, we take the dot product of the two tracks and divide it by the norms of the tracks as mandated by the Cauchy-Schwarz Inequality to find the percent that they match. This is done twice, mapping the first signal onto the second and then vice versa, and then the highest dot product is taken as the matching correlation.

5. Training:

When a speaker attempts to verify himself with this system, his incoming signal is compared to that of a "key". This key should be a signal that produces a high correlation for both magnitude and pitch data when the authorized user utters the password, but not in cases where:

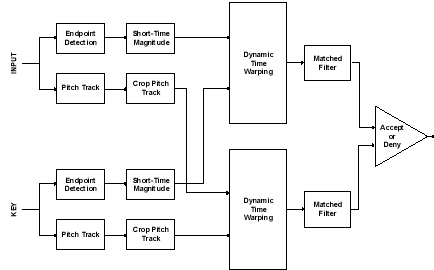
* the user says the wrong word (the password is forgotten)
* an intruder says either the password or a wrong word

To develop such a key, the system is trained for recognition of the speaker. In this instance, the speaker first chooses a password, and it is acquired five separate times. The pitch and magnitude information are recorded for each. The signal that matches the other four signals best in both cases is chosen as the key.

Furthermore, the system will return lower thresholds for matching to magnitude vs. time and pitch vs. time. Thresholds below 0.90 are not returned, and to eliminate the possibility of an extremely high threshold like 0.99, an upper bound is placed on the thresholds of 0.95. Therefore, the thresholds will lie between 0.90 and 0.95.

6. Verification:

Once a key has been established, an authorization attempt breaks down into the following steps:



1. The person speaks a password into the microphone.
2. The short-time magnitude of the signal is found and recorded, as is the pitch track.
3. Each is cropped and dynamically time warped so that (possible) corresponding points on the signals are aligned.
4. Now that the signals rest on top of each other, matched filter (for both magnitude and pitch) to determine a numerical value for their correlation.
5. These numbers are compared to the thresholds set when the key was first created. If both the magnitude and pitch correlations are above this threshold, the speaker has been verified.
6. Allow user to enter top-secret hangout.

To perform the previous steps, an m-file with the necessary function calls, keysigmatch.m, is executed with the lock signal and the thresholds established by makelock.m.

Also we used the following signal analysis , to increase the efficiency of the match. These analysis compare other characteristics of the signal like:

length of the signal , linear predictive coefficients, and energy.

That is the result of the previous implementation give us a percent of nearly 80% of success. The following test was added to the previous implementation, and it consists of the following tests:

1. Length of the signal .

2. Cross\_Correlation of the signals.

3. FFT of the signal.

4. Cross\_Correlation of power spectras.

5. Power spectrum above threshold.

**Chapter III: Command Recognition**

The most common way of giving commands to your computer is, for today, the keyboard. This way keeps your hands constantly working whenever you want to communicate with your PC.

A speech-recognizing computer will have several advantages over the traditional one:

* Hands free communication way.
* Supplying the user with ability to command his PC remotely (even from a cell phone).
* Faster input (typing is, for most users, slower than talking).

The reasons above are only part of the advantages that computer voice recognition could supply.

Those advantages are the main motivating factor to develop reliable voice recognizing system.

**Background**  
  
The main problem with speech recognizing computer is the big variety of commands (words) that exist to command it. When typing, each command is composed of a finite set of symbols, each of them easily recognized by the computer according to the pressed keys on the keyboard. Voice commands, on the other hand, do not consist of predefined components. Each of them has its own, unique voice signature. In the ideal world, recording each word and comparing a new sample to the record will solve the problem. However, this method is not practical since even when produced by a single speaker, the records vary in dependence of speaker’s mood, distance from the microphone, background noise, etc. Any other user will produce a different voice signature for the same command, thus making the recognition process even more complicated.

Many methods were developed in order to overcome this problem; one of them is presented in this project.

**The method:**  
The method we applied in the project is based on hidden Markov chains. The algorithm analyzes the speech signal, divides in into frames. Each frame is processed and then represented by a point in a p-dimension space. A special Markov machine (built and tuned according to the training set of voice samples) then estimates the probability of the current sequence of frames to represent the spoken command. The probability acquired for each command is the used to classify the current voice sample.

The process consists of two parts:

1. Learning part: In this part the voice sample is being processed and its parameters are added to the general statistics of the training set of voice samples. The statistics are then used to represent each command by a special Markov machine that will be used to classify new voice samples in the recognition part.

Picture1

1. Recognition part: In this part, each sample is preprocessed in a way similar to the training part. After the preprocessing, the algorithm tries to match the frames sequence to each one of the ten Markov machines. Each machine represents a command. The probability of the frame sequence to match each of the machines is calculated. The biggest probability tells us which command should represent the current voice sample.

Picture3

Commands set:

Our project is able to recognize up to 15 different windows commands, more commands can be added easily but we were restricted with the time and samples availability.

These commands include:

* 1. Close
  2. Desktop
  3. Hyper terminal
  4. Information
  5. Log off
  6. Media player
  7. Notepad
  8. Paint
  9. Pin ball
  10. Search
  11. Shutdown
  12. Sound volume
  13. Start menu
  14. System restore
  15. Task manager

**Results:**

**The main feature that a command-recognition system should have is to be speaker-independent, our SVSR guarantees achieving this feature. Increasing number of samples for each command from different people is going to increase the efficiency of recognizing the command from any user.**

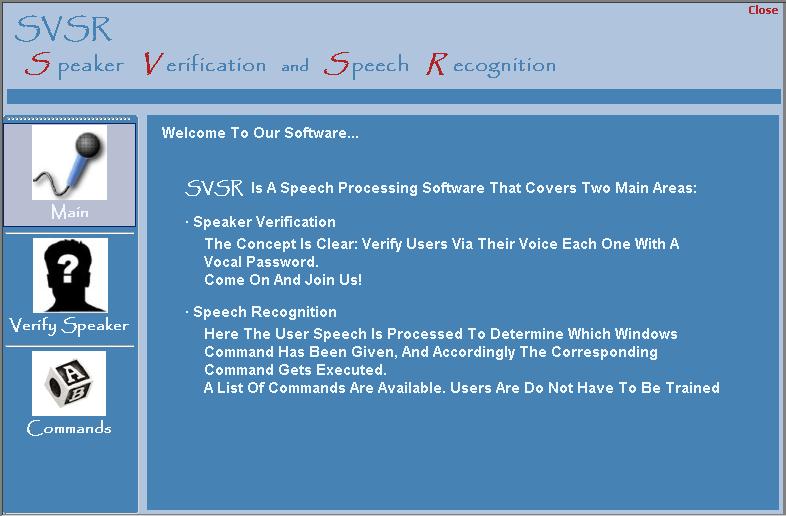
**Chapter IV: How to use our software in steps:**

1. **SVSR Launching:**

This snapshot shows the main window of SVSR during the initializing process. It requires a moment to connect to the matlab compiler. During this initializing the program is just waiting and the features are disabled.



After the initialization is complete the SVSR features are available and ready to run.



1. **Verify Speaker:**

This is the first part of our SVSR. User can access it from the side menu.

User – as we can see – can choose between Training and Verifying.

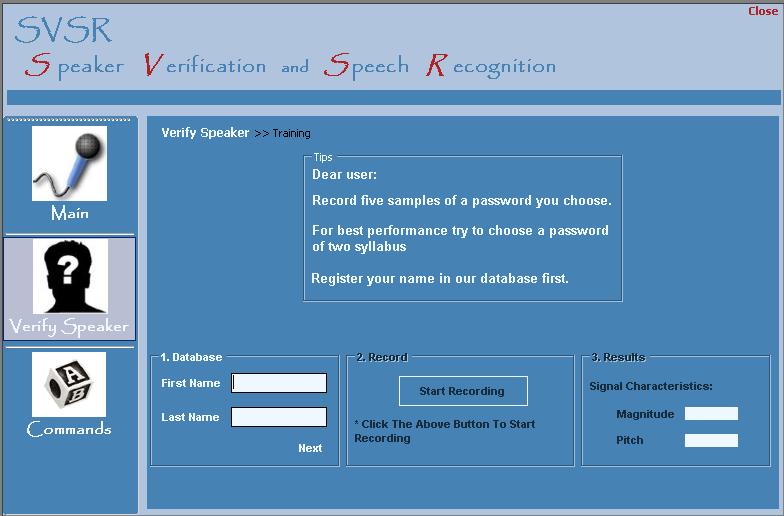


2.1 **Training:**

Here the user is going to choose some vocal password of his choice. The user will record five samples of that password, his name and his signal characteristics will be saved in our database.

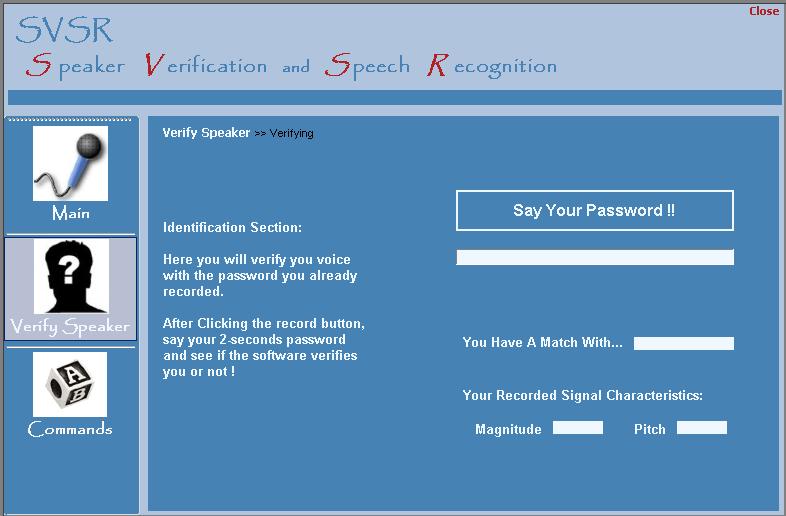
Here there are three steps

1. Saving the name of the user.
2. Recording the 5 samples.
3. Previewing the user’s characteristics.



* 1. **Verifying the user**

It’s simple. The user enters here to see if the SVSR recognizes his voice based on the previously-recorded password. The user say the 2-seconds password after pressing the button, SVSR processes it and returns the result found into the boxes shown.



1. **Command Recognition**

This is the second part of the SVSR. The idea is to make your windows operating system automated by voice. The user can choose one of the 15 command we have chosen to execute. Just say the command in a period of 2-seconds after clicking the button.

