TABLE OF CONTENTS

CHAPTER 1 RESEARCH TOPIC ONE 3

1.1 INTRODUCTION 3
1.2 EXPOSITION OF THE MODELS 3
1.2.1 NOTATION 3
1.2.2 MODELING THE DECISION WITH HASH TABLE 4
1.2.3 MODELING WITH UNIGRAMS, BIGRAMS AND TRIGRAMS 6
1.2.4 MARKOV MODEL 8
1.3 RESULTS 10
1.3.1 THE HASH TABLE APPROACH 10
1.3.2 N-GRAM APPROACH 10
1.3.3 MARKOV MODEL 12
1.4 CONCLUSION 12

CHAPTER 2 RESEARCH TOPIC TWO 15

2.1 INTRODUCTION 15
2.2 RESEARCH PREPARING 15
2.3 PROBLEMS RESOLVING 16
2.3.1 GENERATE THE SPECTRA OF THE VOWELS 16
2.3.2 COMPUTER FOR EVERY VOWEL THE FIRST AND SECOND FORMANT 18
2.3.3 MAKE A PLOT OF THE VOWEL IN THE F1 – F2 SPACE 22
2.4 FINDINGS AND CONCLUSIONS 22

APPENDIX A: REFERENCES 23

INDEX OF FIGURES AND TABLES

FIGURE 1: THE TELEPHONE KEYBOARD SHOWING INFORMAL MAPPING BETWEEN THE SET OF DIGITS μ ,..... 4
FIGURE 2: STRUCTURE OF THE TRIE NODE, AND A VIEW AT THE TRIE 5
FIGURE 3: AN EXAMPLE MARKOV MODEL..... 8
FIGURE 4: ONE SPECTROGRAM EXAMPLE OF VOWEL 'A' 16
FIGURE 5: ONE SPECTROGRAM EXAMPLE OF VOWEL 'E'..... 16
FIGURE 6: ONE SPECTROGRAM EXAMPLE OF VOWEL 'I' 17
FIGURE 7: ONE SPECTROGRAM EXAMPLE OF VOWEL 'O' 17
FIGURE 8: ONE SPECTROGRAM EXAMPLE OF VOWEL 'U' 18

TABLE 1: EXAMPLE DECODING SESSION..... 10

Chapter 1 research topic one

1.1 Introduction

In this research topic, I will focus on following contents. Consider a handheld. Every key represents more than one letter. Key 2 represents the letters A, B, C. I can agree that pressing key 2 selects letter a, pressing two times letter b and pressing three times letter c. But I prefer to press only once. Now the received message is ambiguous. Assume that I receive the labels of the keys and a bracket for the space between words. Can I decode the message? If the vocabulary is small, this can be done. Let me assume that our vocabulary consists of the words of two pages of A4 text. Based on this text I can compute the frequency of the letters and combination of letters such as bigrams and trigrams.
There exist three approaches:

a) Hash tables. I can generate a tree/graph with ten branches from every node. At every node I have a list of possible words. I can order these words by frequency. I can compute the most probable solution of a message $k_1, k_2, \cdots, k_n$.
b) Uni-, bi-, trigrams. I can also compute a most probable path using unigrams, bigrams or trigrams.
c) Markov models. I can model the problem as a Markov mode. I can use the Viterbi algorithm to compute the most probable path.

1.2 Exposition of the models

This section will describe the models I used to write the example programs, and will give some rationale for their derivation. I will begin the exposition by introducing the notation used throughout the report. Only the newly introduced denotations are explained, and other standard ones (see: [1, 2]) are presumed known in advance. The next step is describing the deterministic approach given with the hash table. Deriving *n*-gram and Markov models follow.

1.2.1 Notation

Let χ be the set containing letters of English alphabet: $\chi = \{A, B, \ldots, Z\}$. I limit ourselves only to the 26 upper case letters, as the introduction of case distinction adds no value to the word problem considered. Let μ be the set of digits: $\mu = \{2, 3, \ldots, 9\}$, and a mapping $\pi : \chi \mapsto \mu$ function that gives the mapping from the letter of the alphabet to the digit, as informally defined by the keyboard model on Table 1.

Figure 1: The telephone keyboard showing informal mapping between the set of digits μ, And the set of letters of the alphabet χ

I also define the inverse function $\pi^{-1}(m) = \{n \mid \pi(n) = m\}$, which for a given digit gives a set of associated letters. This function, with informal name *word2key* is given in the class.

The set of all known words are called a language vocabulary and denoted as $L \subseteq \bigcup_i \chi^i$. It is a subset of all the words that can be formed from a string of characters of alphabet χ. L is built from a training corpus, i.e. a set of words that are usually taken from a representative text. It will be seen further in the text how building the word list is implemented. Let k be the vector of received keystrokes. This means that $k = (k_1, k_2, \ldots, k_n)$, where each element is from the list of digits μ, i.e. for all $1 \le i \le n$ olds that $k_i \in \mu$. Let $c = (c_1, c_2, \ldots, c_m)$ be the vector of characters from χ, i.e. a word from the language vocabulary L. I also introduce a keyboard vocabulary $\xi(k)$.

For a given vector $k \in \mu_n$, I define: $\xi(k) = \prod_{j=1}^{n} \pi^{-1}(k_j)$

That is to say, for a given vector k of length n, the function ξ returns a set of all the word vectors of length n that can be formed using the letters that correspond to appropriate keys k_j. To conclude, I introduce the extracting a sub vector from a given vector. With a vector $x = (x_1, \ldots, x_n)$, the sub vector $x_{j:i}$ (where $i < j$) is defined as: $x_{j:i} = (x_i, \ldots, x_j)$. The limit case is simply: $x_{j:j} \equiv x_j$.

1.2.2 Modeling the decision with hash table

The first decoding technique I present is purely deterministic. It depends on the existence of a vocabulary L, which is taken from a representative text. This first approach yields a structure that, once built, is well suited for fast searches of words that match a particular input sequence k. The class implements a structure used to build the vocabulary L. I use a *trie*, a tree-like structure in which the out degree of each node is less or equal to the number of elements in the digit set ($|\mu|$). Please refer to Figure 2 for an illustration of the trie structure.

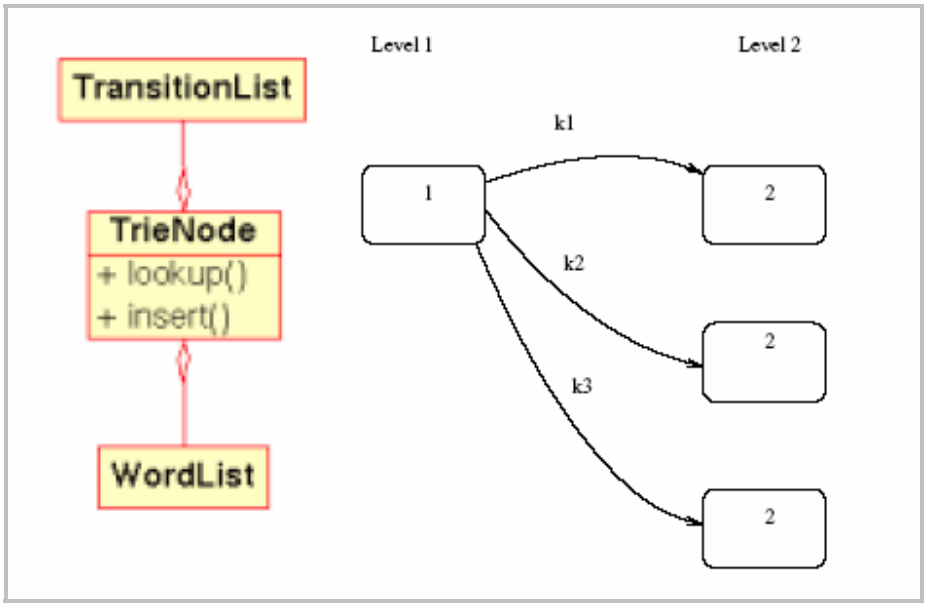


Figure 2: Structure of the trie node, and a view at the trie

The trie encodes all the words in the vocabulary *L*. Each node of the trie can store a word list. The coding is as follows: the links between the nodes of the graph are each weighted by one (and only one) element of the keystroke alphabet μ. The nodes are stratified into levels. The path from the root node to a node on level l contains a sequence of links $(k_1,\ldots,k_l)$, and corresponds to a sequence of l keypresses. The invariant of the trie is: every node in a subtrie on level *l*, corresponding to a list of keypresses $k=(k_1,\ldots,k_l)$, has in their word lists the words whose letters map by function π to the given vector k. If a particular part of the subtrie is empty, say because no words correspond to a particular path, the coding is economized by leaving out the entire subtrie.

The word search is performed as follows. Let there be given an input vector k. The search is started at the root of the trie, the only node inhabiting level 0. I examine element x_1 of the vector and follow the link labeled with the corresponding label to a node on level 1. I repeat the algorithm with this destination node as root node, with an input sequence of $k_{n-1:1}$. A good recursive definition needs the "bottoming out" rule. Once I get to an ill-defined subvector $k_{0:1}$, I bottom out by returning the word list of the node I got to. If at any place the search fails, such as is the case when I get to a node that has no descendants, whereas link following would require me to proceed further, I exit returning an empty set of matching words.

The upside of the choice is that search is fast, as I only need to traverse a single path down the trie. It is also easy to obtain all the words matching the given prefix by finding a node whose path from root encodes the prefix and then getting all the words from the

word lists of the subtrie induced by that node. Adding a new word into the vocabulary is also convenient. I first encode the word by mapping it onto a corresponding keypress vector with function π that I previously defined, and then perform an operation similar to the aforementioned search. In essence I follow the same steps as the search would, but this time I are free to add the missing nodes and links. The new word is then inserted into the word list of the node that is the furthest down the search path.

Functions *lookup* and *insert* are provided for this purpose. The trie can be stored to a file and retrieved when necessary. I use the program *mkwordlist* to build the vocabulary *L*, and the program *hashlookup* to decode. The lookup in hash table retrieves only a list of the matching words, or an empty set if no words match.

The downside of the approach is that at all times I need to have a reference vocabulary at hand. This may be a formidable constraint if I are interested in implementing this coding scheme in a mobile device. Also the encoding is not very efficient, as quite a large overhead is present at each node that adds to the trie "infrastructure" but also the volume. The hash table I used for testing had 1510 words and the footprint was as large as 150kB. Admittedly the encoding was not very efficient, given that the compressed footprint is only 18kB, but it is a good indicator of the storage space needed to host a trie on a memory device. The size increment will surely decrease as the trie gets filled with more and more words, but it is the initial growth that may make it unmanageable on a mobile device. A very interesting way to get around the requirement of having a vocabulary, thus reducing overall storave is the *n*-gram approach of the following section.

1.2.3 Modeling with unigrams, bigrams and trigrams

In the *n*-gram approach to decoding, I substitute the full vocabulary for a table of a predetermined fixed size. Instead of memorizing all the words, I only extract and memorize features. I hope that the features take less memory but still perform fairly well in decoding the words.

More specifically, I chose to memorize the joint probability distribution of a vector of *n* letters. For a given *n* I have an *n*-gram: unigram for *n* = 1, bigram for *n* = 2, trigram for *n* = 3 and so forth. Thus for a given *n* I need to memorize at most $|\chi|^n$ values, using a space that can be considerably less than what is consumed by a trie. The joint probability distribution for an n gram is given by $P(c)$, where $c \in \chi_n$. This function is obtained empirically by analyzing a representative sample text. Note now that I do not operate with L once the distribution is determined, so I can throw away L and work only with $P(c)$.

I now proceed to relate the words in the language to the keypress sequence. Let me first build a vocabulary model for *n*-grams, and then specialize for $n = 1, 2, 3$. I are interested in maximizing the a posteriori probability that given k of length l, the word c was typed in. If I denote c^* as the best match according to this criterion, it is given as:

$$c^* = \arg\max_{c \in x^l} P(c \mid k)$$

This expression requires iteration over the whole *l*-dimensional space of the input sample to compute the inverse maximum. However, the search space can be reduced by noting that I need only examine such c that can be obtained from rolling back from k to c using the function $\xi(k)$. The last equation is now:

$$c^* = \arg\max_{c \in \xi(k)} P(c \mid k)$$

But now I note that since I limited the search space to that of $\xi(k)$, the conditional probability is in fact directly proportional to the marginal: $P(c \mid k) = 1/P(k) \cdot P(c,k)$, and this holds as long as $c \in \xi(k)$ is true. I can therefore maximize just as good:

$$c^* = \arg\max_{c \in \xi(k)} P(c)$$

The only remaining term is $P(c)$, and its decomposition depends upon the model I choose:

$$P(c) = P(c_1,\ldots,c_n) = \prod_{i=1}^{n} P(c_i) \qquad \textit{For unigram model}$$
$$P(c) = P(c_1,\ldots,c_n) = P(c_{2:1}) \prod_{i=2}^{n-1} P(c_{i+1} \mid c_i) \qquad \textit{For bigram model}$$
$$P(c) = P(c_1,\ldots,c_n) = P(c_{3:1}) \prod_{i=2}^{n-2} P(c_{i+2} \mid c_{i+1}, c_i) \qquad \textit{For trigram model}$$

The terms $P(c_{2:1})$ and $P(c_{2:1})$ from the previous equation deserve some attention. They are clearly a telltale sign of asymmetry in the approach. To have completely symmetric formulae, I would be required to substitute the above for the appropriate *n-1*-gram joint distribution. However, I chose not to proceed that way, in hope that the longer the initial *n*-gram is, the better it would determine the starting letter of the word. I chose to go for the logest available *n*-gram instead.

Namely, once the choice for the start of the word is made, the choices of other letters are very much fixed. So in case the initial letters are guessed wrong by, say, choosing a sequence of unigrams at start, I would have a word that is very likely to start with a vowel, and there would be no subsequent way to get back on right track, as I would then be examining only conditional probabilities. Therefore, to play safe, I chose to substitute the most probable longest available *n* gram at the beginning of the word.

Now the conditional probabilities of the bigram and trigram models can be expressed in terms of joint probabilities, which are known a priori:

$$P(c_{i+1} \mid c_i) = \frac{P(c_{i+1,} c_i)}{P(c_i)} \text{ And } P(c_{i+2} \mid c_{i+1}, c_i) = \frac{P(c_{i+2}, c_{i+1}, c_i)}{P(c_{i+1}, c_i)}$$

All the unconditional probabilities can be gathered from the frequency data of the unigrams, bigrams and trigrams. Note also that the computation of *n*-gram depends on knowing *(n−1)*-gram frequencies, and that I readily determine $P(c_i)$, $P(c_{i+1}, c_i)$ and $P(c_{i+2}, c_{i+1}, c_i)$ by counting within L. Now I can readily maximize over all words attainable by input k and obtain c^*. Since all terms of the products are independent, I can maximize them individually to obtain the global maximum. The sequence obtained in this way is the most probable *n*-gram sequence. Here one needs to handle cases in which prior probability for a sequence is equal to 0. A prior probability of 0 for an *n*-gram sequence means that such a sequence does not appear in the training set. Since I only expect the words coming from the training set, it is not needed to handle this case in more detail than to assign a low non-zero probability to it.

1.2.4 Markov model

Modeling the decoding as a Markov model is an elegant way to encode the letter interdependencies (Figure 3).

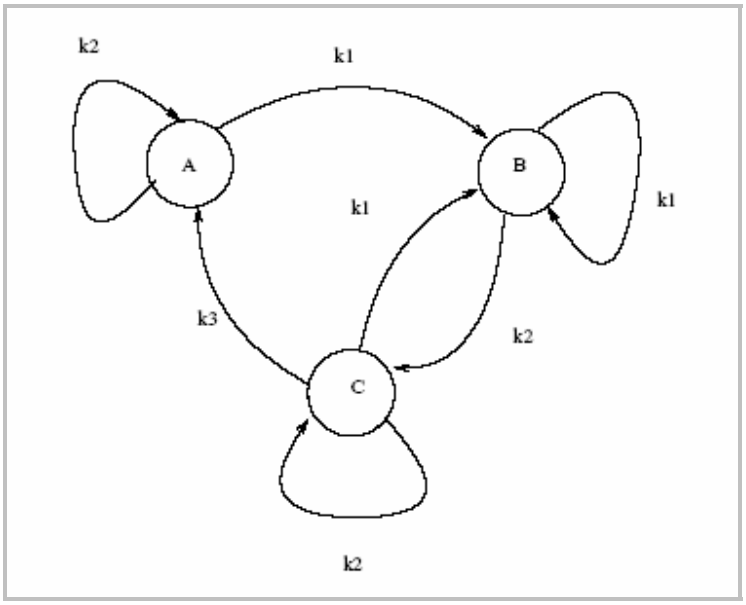


Figure 3: An example Markov model

I are now able to express the probability over the entire sequence. I will work our way towards obtaining the components of the Markov model backwards: starting from the expression that I wish to maximize. I will point to an efficient algorithm to compute the required a posteriori maximum probability. The expression to maximize over words is:

$$c^* = \arg\max_{c \in \xi(k)} P(c \mid k)$$

Which asks a question: "What is the most likely set of transitions in the Markov model that could have produced the keyword k?" The conditional probability expression above can be unpacked as:

$$P(k \mid c) \equiv P(k_{n:1} \mid c_{n:1}) = P(c_n, c_{(n-1):1} \mid k_n, k_{(n-1):1})$$

I emphasized the *n*-th element of both the word and input vectors. The idea underlying the following few lines are to express this conditional probability in terms of pre-computable values and a smaller instance of itself; I know that such an expression is possible according to Viterbi and others. I reshuffle the last equation to eliminate dependency upon k_n, and also drop the conditioning on k_n upon $k_{(n-1):1}$:

$$P(c_n, c_{(n-1):1} \mid k_n, k_{(n-1):1}) = \frac{P(c_n, c_{(n-1):1}, k_n \mid k_{(n-1):1})}{P(k_n \mid k_{(n-1):1})} = \frac{1}{P(k_n)} \cdot P(c_n, c_{(n-1):1}, k_n \mid k_{(n-1):1})$$

By using chain rule I can decouple dependency from last states:

$$\frac{P(c_n, c_{(n-1):1}, k_n \mid k_{(n-1):1})}{P(k_n)} = \frac{P(c_n, k_n \mid c_{(n-1):1}) P(c_{(n-1):1} \mid k_{(n-1):1})}{P(k_n)} = \frac{P(k_n \mid c_n) P(c_n \mid c_{(n-1):1}) P(c_{(n-1):1} \mid k_{(n-1):1})}{P(k_n)}$$

By noting that:

$$P(k_n \mid c_n) = \sum_{c \in \pi^{-1}(k_n)} P(c \mid c_n), \; P(c_n \mid c_{(n-1):1}) = P(c_n \mid c_{n-1})$$

I finally get:

$$P(k \mid c) \equiv P(k_{n:1} \mid c_{n:1}) = \frac{P(c_n \mid c_{n-1})}{P(k_n)} \cdot \sum_{c \in \pi^{-1}(k_n)} P(c \mid c_n) \cdot P(c_{(n-1):1} \mid k_{(n-1):1})$$

I see $P(k_{n:1} \mid c_{n:1})$ expressed in terms of $P(c_{(n-1):1} \mid k_{(n-1):1})$ and pre-computable quantities, $P(c_n \mid c_{n-1})$, which are *transition probabilities* between the states the Markov model. I also see that I need to adopt $c_i \in \chi$ as the states of the model, with $k_i \in \mu$ as transitions. I "bottom out" on the recursion in the previous expression by setting the a priori probabilities of states to:

$$\pi_c = P(c_{1:1} \mid c_{0:1}) = p(c_1)$$

For each of the possible $c_1 \in \chi$, I see that the transition probabilities can be given in terms of the bigram values that I already computed in the previous section. To compute c^* I can find the global maximum with an algorithm that exploits the recursion, such as that of Viterbi.

1.3 Results

In this section I present the results of the experiments with the programs written as a supplement to this report. The programming languages used was C++ and Java.

1.3.1 The hash table approach

Table 1 gives an example session with the decoding program, *hashlookup*. The utility *word2key* is a helper which enables the tester to type words in plain English instead of permanently allocating one eye for seeking the key mapping. The converted values of the keys are fed into the *hashlookup* program, which never actually sees the words. Instead it only sees the converted values and operates only on them. The vocabulary was built from the hash table whitepaper.hash that is a technical text about 9 pages long.

The testing text was one of the most famous lines in English literature, the notorious dilemma of the prince of Helsingor. This contrast to the intial technical text was deliberately chosen in order to show that the intersection of the two vocabularies is quite a curious set of words bearing general meanings or high utility value. It also emphasizes a weakness of hash table approach: absolute dependency on a vocabulary.

I can see that the exact matches are easily found by the hash table lookup. The lookup is also rather fast but, as discussed earlier, at the expense of the increased storage. The mismatch of the vocabularies is apparent. The text likely to be decoded is only a (small) subset of common words. This means that although the decoder will give the decoded words exactly, it is not able to recognize words outside of the prescribed dictionary. This, together with bulkiness, is a major drawback of the hash table.

1.3.2 n-gram approach

According to the vocabulary file, the program can make the frequency table based on the whole combination of the uni-letter, bi-letters and tri-letters. Using frequency table, probability of table 1: example decoding session. The file whitepaper.hash is the hash table of a technical report of approximately 9 text pages. The lower case letters were typed in from the console, and the decoding by the program was done word for word, with the decoded words in the brackets.

Table 1: Example decoding session

To be or not to be, that is the question.
86: (TO)
23: (BE AD)
67: (OR)
668: (NOT)
86: (TO)
23: (BE AD)

8428: (THAT)
47: (IS)
843: (THE)
78378466: (QUESTION)

Whether tis nobler in the mind to suffer
9438437: (WHETHER)
847: ( )
662537: ( )
46: (IN GO)
843: (THE)
6463: (MIND)
86: (TO)
783337: ( )

The slings and arrows of outrageous fortune...
843: (THE)
754647: ( )
263: (AND)
277697: ( )
63: (OF)
6887243687: ( )
3678863: ( )

Letters can be computed. For example, frequency of the letter"A" in example vocabulary file are 692 and total frequency of the uni-letter are 8897, so the probability of the letter"A" is 0.077779025. Based on different approach, the same stroked keys queue can be decoded to different result. (Such as keys queue "43556", in uni-gram approach is "IELLN", in bi-gram approach is "HELLM" and in tri-gram approach is "HELLO"). If the keys queue can not be recognized, program will print nothing.

The example vocabulary file is some parts of the famous novel. Not like the hash table approach, search is based on the exact word in the vocabulary. In n-gram approach the result can be the random combination of the letters, such as "AA", "ACY", "YES", "HELLO" and so on. Some of them are not the legal words at all. As description in the former part, n-gram approach use probability of the letter to recognize the stroked keys. I use the same stroked keys as hash table approach to see the result. Figure 5 shows the result of the uni, bi and tri-gram approach.

1.3.3 Markov model

I already showed that I need bigram and unigram data to build the Markov model. I give a tabular list of transitions and initial probabilities from the model. The probabilities are obtained from an excerpt from another English literature classic, the novel “Wuthering Heights”. The given data are enough to construct the full Markov model graph.

The frequency graph is given in a separate file being an attachment to this report, due to its size. The files are in the directory results/. Unigram probabilities $P(c_i)$ are initial state probabilities of the Markov model. Transitions are given as $P(c_{i+1}c_i) / P(c_i)$.

1.4 Conclusion

In this report I tried to present our answer to the questions of assignment 2, by presenting both the program code with experiments and the foundations upon which the code was built. I also analyzed the properties of each of the diverse approaches to de-ambiguation of the keypress sequence given some knowledge of the structure of the encoded content, i.e. requirement that decoded messages be valid words of English.

I conclude that the hash table approach will yield accurate decoding as long as the vocabulary is sufficiently large. The vocabulary size may be formidable for mobile devices, however. To alleviate this shortcoming, an alternative approach was analyzed, where the vocabulary is only used to build a fixed size table. Presented results suggest that it may be possible to provide good decoding that only exploits local dependence of the characters within a word, but that it depends heavily on the text used for training.

The obvious upside is that no vocabulary is needed and that the storage space is fixed once the *n* of the *n*-gram model is determined, which would probably be at design time. I did not investigate the decoding error rate dependence upon the richness of training vocabulary, assuming that the decoding quality would increase with enlarging the training set. While this may be so, I do not know whether a saturation point exists, past which no training will yield significant decoding performance boost.

1) To be or not to be, that is the question
86 23 67 668 86 23, 8428 47 843 78378466

Stroked Keys	Uni-gram	Bi-gram	Tri-gram
86	TN	TO	TO
23	AE	AD	AD
67	NS	OR	OR
668	NNT	NOT	NOT
86	TN	TO	TO
23	AE	AD	AD
8428	TIAT	THAT	THAT
47	IS	IS	IS
843	TIE	THE	THE
78378466	STESTINN	RVEPUIMN	STESTION

2) Whether tis nobler in the mind to suffer
9438437 847 662537 46 843 6463 86 783337

Stroked Keys	Uni-gram	Bi-gram	Tri-gram
9438437	YIETIES	WHETIEQ	WHETHER
847	TIS	TIS	THR
662537	NNALES	OMALDS	OMBLES
46	IN	IN	IN
843	TIE	THE	THE
6463	NINE	NIND	NINE
86	TN	TO	TO
783337	STEEES	RVEEEP	STEDER

3) The slings and arrows of outrageous fortune
843 754647 263 277697 63 6887243687 3678863

Stroked Keys	Uni-gram	Bi-gram	Tri-gram
843	TIE	THE	THE
754647	SLINIS	PLINIS	RKINGS
263	ANE	AND	AND
277697	ASSNYS	ARROWS	ARROWS
63	NE	ND	ND
6887243687	NTTSAIENTS	NUTRCHENTS	OTTRAGENTR
3678863	ENSTTNE	ENSUTOF	ENSTUND

Chapter 2 research topic two

2.1 Introduction

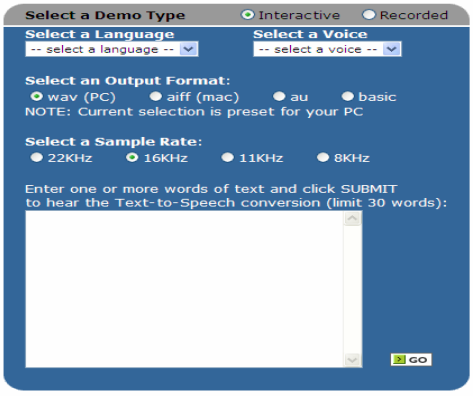
In this research, an automatic speech processing tool – CSLU (Center for Spoken Language Understanding) and MATLAB are used. The latter software is powerful tool for analyzing signal characteristics. The detailed task in this research is:

1) Generate the spectra of the vowels;
2) Compute for every vowel the first and second formant (make at least 5 registrations of every vowel);
3) Make a plot of the vowels in the F1 – F2 space, i.e. take the values of F1 – F2 as coordinates in a 2-dimensonal Euclidean space;
4) Based on the experimental findings, make a conclusion.

2.2 Research preparing

According to the research requirements, I download and install the CSUL in my computer, then learn some function about this software. Especially the tools speech view, by it I can generate the spectral of the vowels easily. The CSUL website provide some interesting information about signal process. Such as background knowledge about spectrogram reading. The website is:
http://cslu.cse.ogi.edu/tutordemos/SpectrogramReading/spectrogram_reading.html

To make sure pronunciations of the vowels are correct, I use some Text to Speech software in internet. In this experiment, I use the software developed by AT&T, naming AT&T Natural Voices Text-to-Speech Engine.
The website is: http://www.naturalvoices.att.com/demos/
The use interface is following:



Using it I got the .wav files about the vowels (a, e, i, o, u).

2.3 Problems resolving

2.3.1 Generate the spectra of the vowels

With the help of the tool Speech View in CSLU, I can generate the spectra gram of the vowels. From the spectrogram example of each vowel I may see for each vowel there are some frequency regions of relatively great intensity, which is also not dependent on time.

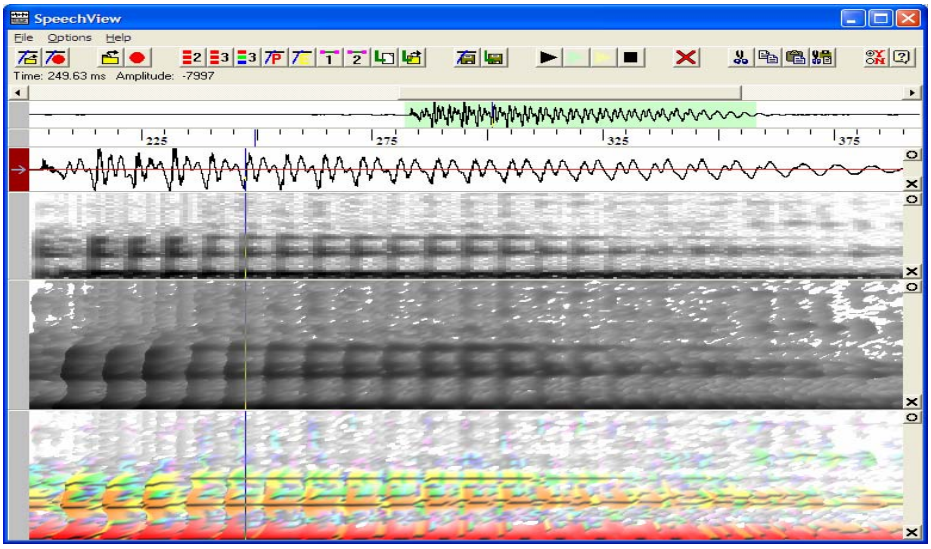


Figure 4: One spectrogram example of vowel 'a'

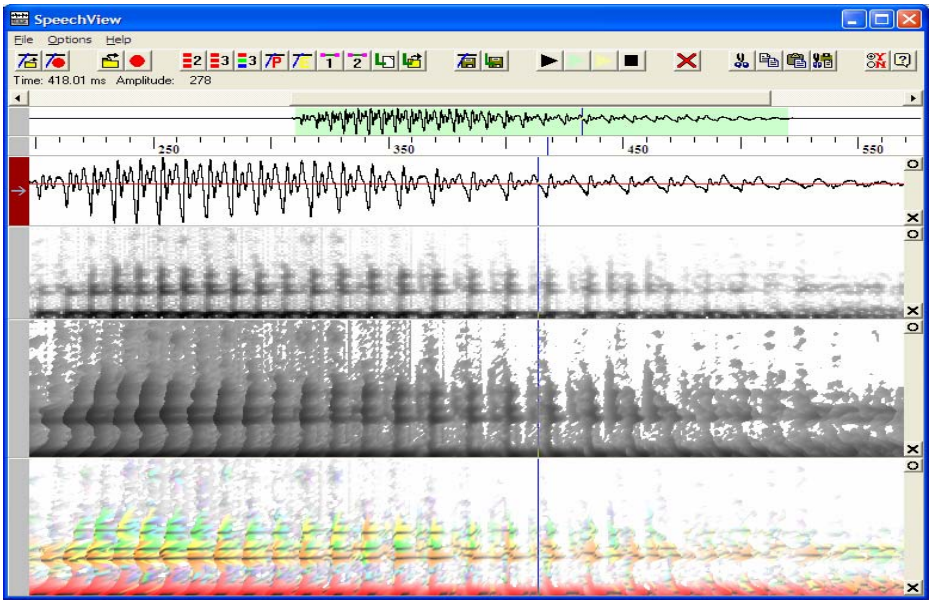


Figure 5: One spectrogram example of vowel 'e'

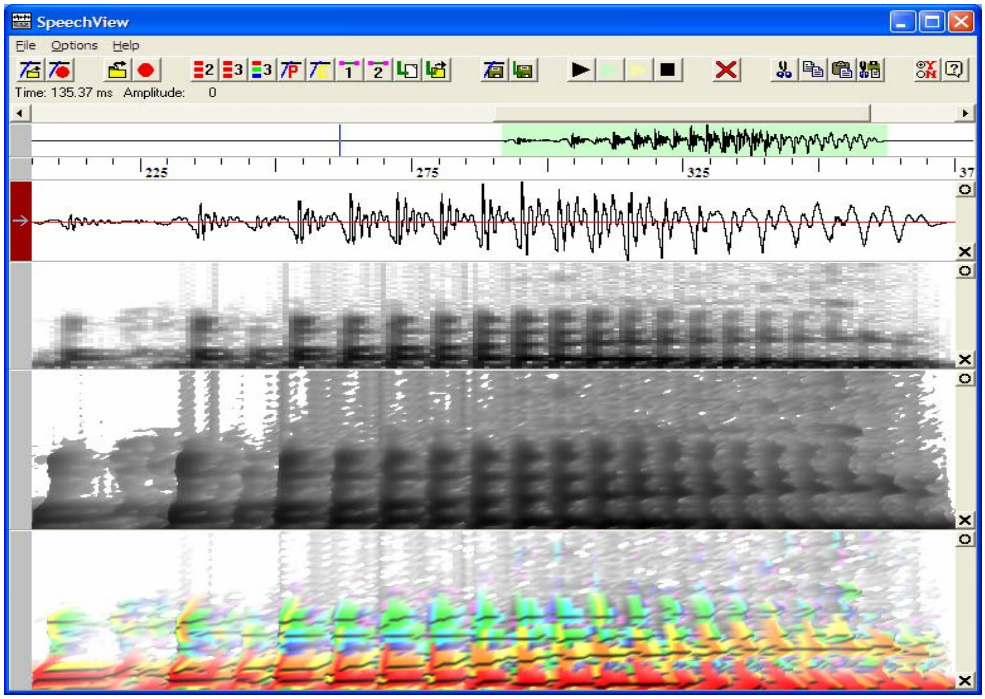


Figure 6: One spectrogram example of vowel 'i'

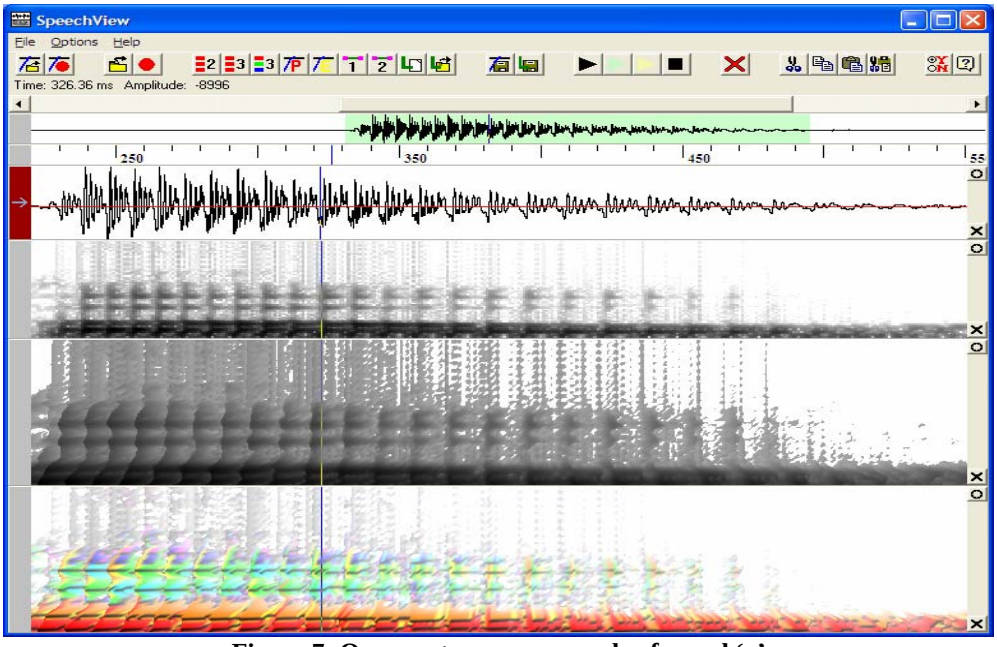


Figure 7: One spectrogram example of vowel 'o'

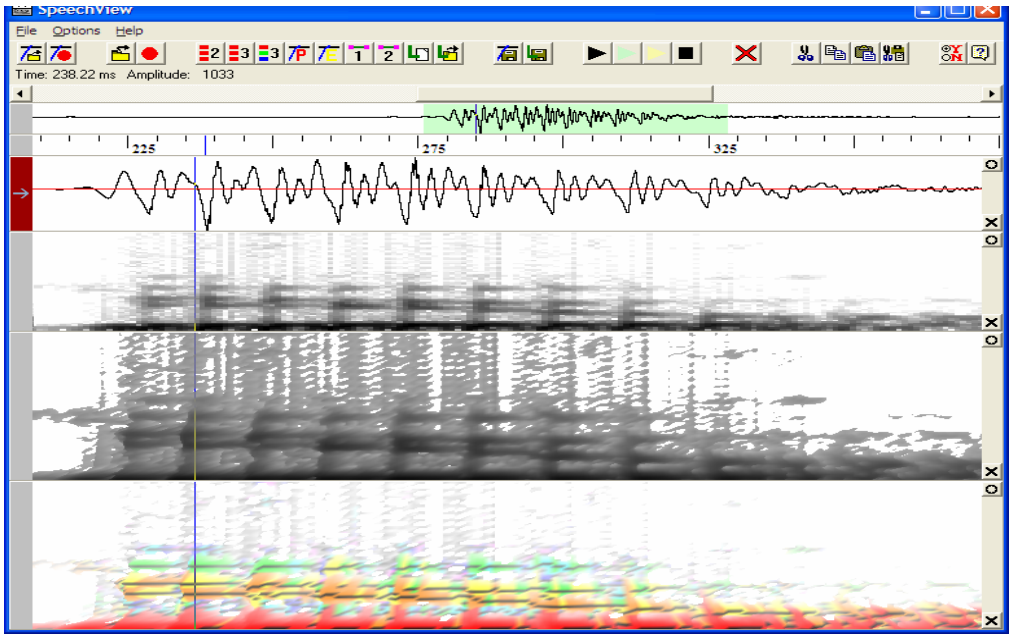
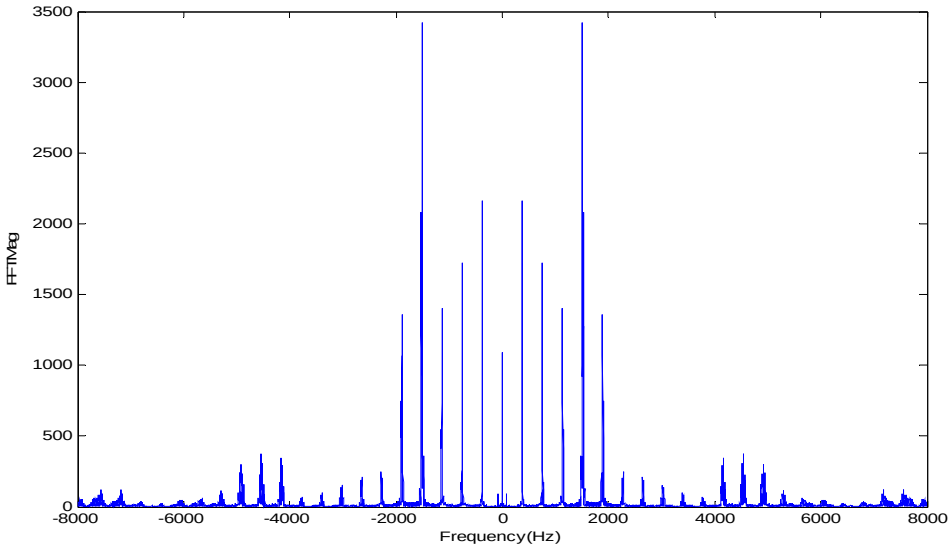


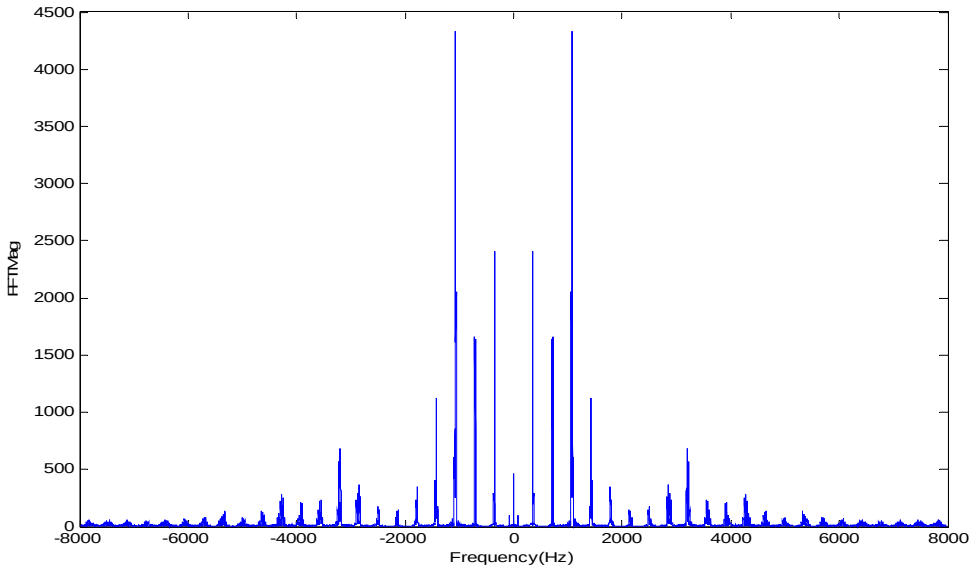
Figure 8: One spectrogram example of vowel 'u'

2.3.2 Computer for every vowel the first and second formant

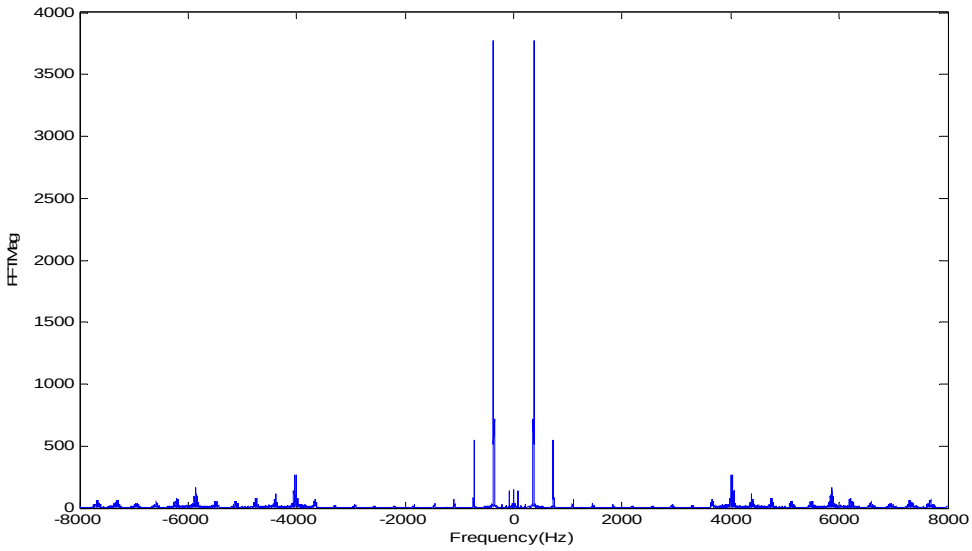
I use MATLAB function ***wavrecord ( )*** to record the pronunciation of five vowels, and then use MATLAB to analyze these vowels spectrogram to learn more detail information.



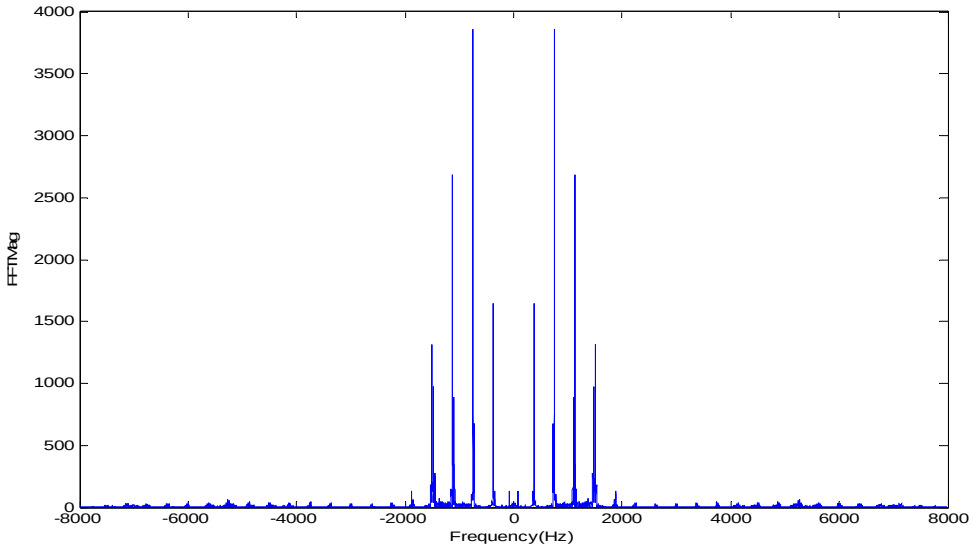
Spectra example of vowel 'A'



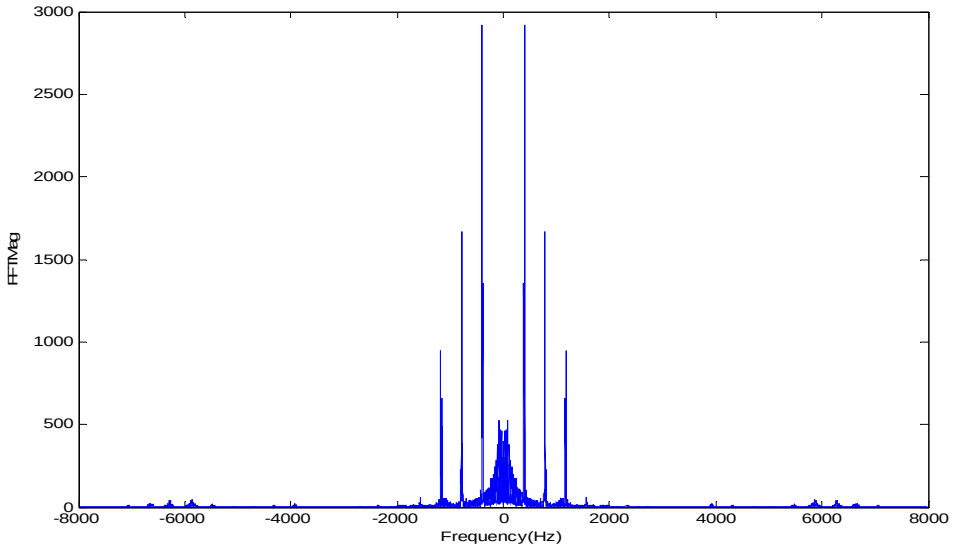
Spectra example of vowel 'E'



Spectra example of vowel 'I'



Spectra example of vowel 'O'

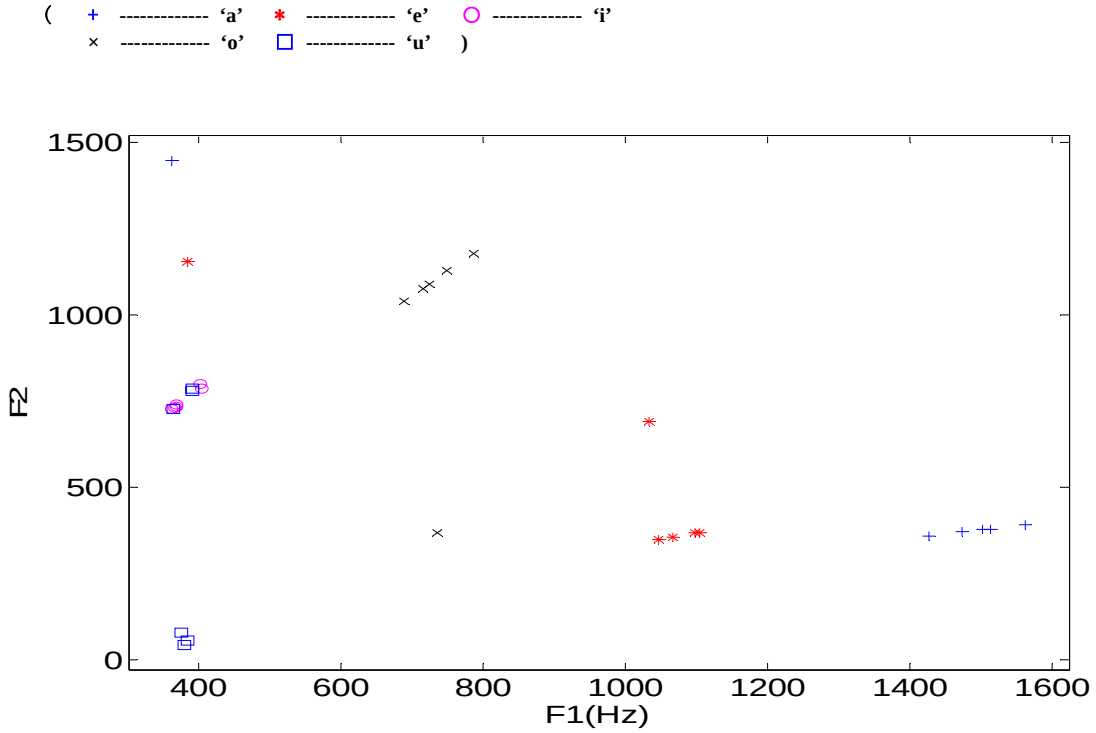


Spectra example of vowel 'U'

According to the analysis of the spectra gram about the vowels, I can define their first and second formant.

Vowel	1st Formant (Hz)	2nd Formant (Hz)
A	1504	378
	361	1447
	1564	391
	1515	377
	1427	357
	1474	370
E	1046	349
	385	1153
	1034	689
	1105	369
	1099	369
	1068	355
I	402	800
	404	786
	366	730
	362	725
	368	734
	369	740
O	715	1075
	690	1040
	736	367
	724	1089
	750	1126
	787	1176
U	365	727
	376	79
	380	42
	390	780
	385	57
	392	784

2.3.3 Make a plot of the vowel in the F1 – F2 space



With this scatter plot I may easily draw a classifier to determine the 5 vowel classes based on the first and second formant.

2.4 Findings and Conclusions

1) A visual representation of vibrations typical of those in human speech is called a speech waveform. (such as Figure 4 to 8)
2) Speech consists of vibrations produced in the vocal tract. The vibrations themselves can be represented by speech waveforms. It is not possible to read the phonemes in a waveform, but if I analyze the waveform into its frequency components, I obtain a spectrogram which can be deciphered.
3) Each phoneme is distinguished by its own unique pattern in the spectrogram. For voiced phonemes, the signature involves large concentrations of energy called formants. Formant values can vary widely from person to person, but the spectrogram reader learns to recognize patterns which are independent of particular frequencies and which identify the various phonemes with a high degree of reliability. For example, the monophthong vowels have strong stable formants; in addition, these vowels can usually be easily distinguished by the frequency values of the first two or three formants, which are called F1, F2, and F3. In my practical work, the vowel can be determined by the formant 1st and 2$^{nd.}$ But for vowel "U", I'd better use three or four formants to get more accurate.

Appendix A: References

[1] S. Russel, P. Norvig: Artificial Intelligence, A Modern Approach, Prentice Hall, Englewood Cliffs (NJ), 1995.

[2] A. Gersho, R. M. Gray: Vector quantization and signal compression, Kluwer Academic Publishers,Norwell (MA), 1992.

[3] CLSU Spectrogram Reading;

[4] MATLAB Manual & Demo;