Author

A 40 Bits Per Second Lexeme-based Speech-Coding Scheme

Anshuman Chadha
Information & Computer Science

For Anshuman Chadha, this research experience has definitely influenced his future. After finishing his research project and completing a degree in Information & Computer Science, he plans to attend graduate school with the hopes of working further with speech and linguistics technologies. In addition to the proposal and grant writing skills he has developed, this project also gave him the opportunity to improve his technical writing skills. To supplement to his academic work, Anshuman interns at Raytheon Systems as a software engineer. In his spare time, he enjoys reading Tom Clancy and James Clavell novels. Anshuman is also one of two UCI undergraduates selected to present his research at 2004 UC-Day in Sacramento.

Abstract

Speech coding is digital representation of the speech sound. Traditional speech-coding schemes handle acceptable quality speech at bit rates that exceed 2,400 bits per second (bps). This project is based on an extremely low bit-rate speech-coding scheme, on the order of 40 bps. In order to get such a low compression rate, all recognized speech is coded at the lexeme (word level), with low-level elements such as tone and frequency completely disregarded. The lexemes are coded using a probability-based compression mechanism. This coded data is then decoded and decompressed using text-to-speech. Results from this experiment showed that while there were errors introduced by the speech recognition engine, listeners were often able to recover from such errors by inferring what was trying to be expressed. Results also showed that errors made by the listeners in recognizing the synthesized samples were highly dependent on the content of the samples, especially with regard to the familiarity the listeners had with the topic. Applications of this method include speech storage and communications over low-capacity channels.

Key Terms

- Huffman Tree
- Lexeme
- LPC Algorithm
- Speech Coding

Faculty Mentor

The manipulation of information in its digital form is one of computer science's fundamental gifts to all other sciences. Things get interesting and fun when the data is related to what human beings can do as in speech communication. It turns out that we, humans, have a relatively low bit rate when communicating through speech. Anshuman's project confirmed the rate of 40 bits per second that has been suggested before as the rate of verbal communication. We all know that there's a lot more to speech than the words, but this project focused only on the words. Compared to the high bandwidth computers out there, we don't say much!

Cristina Videira Lopes
School of Information & Computer Science

Introduction

Digital telephone and cellular technologies are possible largely because of the use of speech coding in the transmission of speech signals. Speech coding is a digital representation of the speech sound that provides efficient storage, transmission, recovery, and perceptually faithful reconstruction of the original speech (Papamichalis, 1987). There are two conflicting factors involved with speech coding: quality of the speech signal versus size of the digital representation. Due to the large amounts of data that needs to be coded, a high bit rate is required for a more natural representation of human speech. If memory or capacity of the channel is of concern, then the bit rate can be lowered at the cost of poorer quality of the reconstructed speech signal, due to the smaller amount of data that is coded.

The characteristics of speech signals, namely the frequencies, allow compression algorithms to operate at 2,400 bits per second (bps) with acceptable quality. The Mixed Excitation Linear Prediction (MELP) coder described in the work of Supple is the current government standard for voice communications (1997). MELP is based on the traditional Linear Prediction Coding (LPC) model (Markel and Gray, 1976), with added features for improved performance. The generally accepted compromise between quality and performance is 2,400 bps. Hence, modern speech coders fall under two categories: those that code above 2,400 bps, and those that code below 2,400 bps. Coders above 2,400 bps are used for applications where the emphasis is placed on the quality of the speech. Coders below 2,400 bps emphasize signal compression and are known as "low bit-rate coders," and those below 1,000 bps are known as "very low bit-rate coders." The method presented in this paper is a speech-coding scheme that codes on the order of 40 bps, and is thus categorized as an "extremely low bit-rate coder."

Segment Voice Coder

Previous research has already been done in the area of low bit-rate speech coding. Signal processing filters are often used to simplify the speech waveform, reducing the amount of data needed to code the speech signal. These filters extract features from the speech samples, such as the pitch and frequency. The coder described in the work of Lee and Cox uses a database of speech segments (1999). When the subject speaks, the speech is segmented as a set of unit references to a database. At the receiving end, the synthesis of the original speech involves recognizing which units from the database were part of the original speech, concatenating the units, factoring in the pitch period, and outputting the speech. The outcome is a type of segment vocoder (or "Voice Coder"), a coder that codes on the order of 3,000 bps with output that sounds similar to the original speech. For this coder to function, text transcripts of the speech must be available *a priori*. This is because text transcripts are used to create the time alignment between the speech waveforms and the spoken text, in order to factor in the pitch periods appropriately, and give the reconstructed speech its naturalness.

The modified coder described in Lee and Cox uses a slightly different approach for coding speech (2002). For this coder, a database of speech blocks is used. Each block is approximately 10 milliseconds in length, and the database is based on the speech patterns of a particular speaker. This approach does not require speech transcripts to be available *a priori* because the speech blocks are already modeled on the voice of the speaker. This approach results in a larger database and a larger encoding. The bit rate of this coder was found to be 580 bps while maintaining natural and intelligible sounding speech.

Phonetic Vocoder

The use of a phonetic vocoder results in an even more compact coding scheme. The system described in Tokuda *et al.* recognizes the phonemes of speech, which are the smallest units in a language that can cause a difference in meaning (e.g. the "h" sound in "hill" versus the "m" sound in mill) (1998). This system codes the phoneme index, the duration of the state (e.g. phoneme duration, silence duration), and the pitch information about the phoneme, as opposed to a segment coder, which uses a database of speech blocks unique for each particular speaker. Consequently, two speech coders are produced: one coding at about 150 bps, and another coding at about 70 bps. The 150 bps coder was found to have the same quality as a 400 bps coder, while the 70 bps coder was found to have a lower quality than the 400 bps coder, though the synthesized speech remained intelligible.

Syllable Level

Another approach is to code at the syllable level. Hirata and Nakagawa proposed a coder that utilizes speech recognition to parse speech input and recognizes all syllables (the system was built around the Japanese language, which is composed of more than 100 unique syllables) (1989). Each syllable is then coded, along with the duration, power and pitch, with only 16 bits per syllable. Reconstruction of the speech involves decoding the code and concatenating the syllables. The coded features are applied to the reconstructed speech and are used to keep the syllables from sounding disjointed. While this approach works well in theory, actual experimentation showed that the intelligibility of the phrases was less than 70%.

A commonality between these three systems, the segment coder, voice coder, and syllable-level coder is that, within the limitations of low bit-rate speech coding, the goal is to reconstruct the original speech signal as closely as possible. Ideally, the resulting speech signal sounds like the signal produced by the person who speaks. For that, several real-time features of the speech, such as pitch and speech duration along with the speech units, must be coded.

The speech-coding scheme in this study is similar to the one proposed by Hirata and Nakagawa in that a speech recognition approach is also used to process the speech signal (1989). However, unlike the other coding schemes, our coder does not attempt to preserve or reconstruct the real-time non-verbal information of the speech signals; it simply targets the words. For that reason, this approach can be classified as lexeme coding (a meaningful word level unit of a language). Further, since some words are used more commonly than others, entropy coding, specifically the Huffman scheme, ultimately can be used to create the codes (1952). Eventually, speech may be synthesized using the voice features of the speakers and stored statically in the decoder (rather than coded along with the messages). With the lexeme coder, speech coding becomes a matter of English text compression.

In the lexeme coder, the words of speech are coded, but not real-time features, such as pitch, tone and duration. In many situations, such non-verbal information of speech is irrelevant. For instance, most handheld devices also have limited memory, and so an efficient data compression algorithm that preserves the lexemes is critical – even more so than the exact reproduction of the original speech signal. This is because the goal of recording a memo is to store a piece of data for future reference. The quality of the speech does not matter as long as the data is transmitted correctly. This is also true in places such as command and control centers during military campaigns, where orders are issued live to units in the field. Here, the overall purpose of communication is the transmission of orders – whether or not the orders sound human-like at the receiving end does not matter. For recording memos and for transmitting orders to military units, using a speech coder that codes and transmits speech features, such as pitch and duration, would be wasteful, when transmitting just the lexemes would be sufficient.

With these types of applications in mind, the lexeme coder was implemented and tested. The method, implementation, and experimental results as well as the limitations of the methods and how it can be improved were discussed.

Tools and Methods

Several tools are utilized for the lexeme coder, all of which are commercial-grade. Figure 1 shows the three layers of the system.

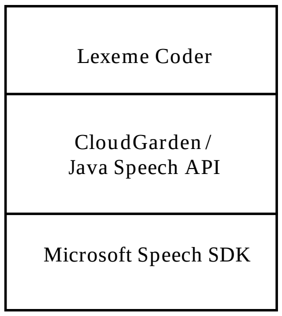

Figure 1
Architecture layers of the system

The Microsoft Speech Software Development Kit (SDK), which provides freely distributed Text-to-Speech (TTS) and Speech Recognition (SR) engines, is used as the backbone of the project (Microsoft Speech, 2002). To easily access the two engines, an open-source Java system called CloudGarden is also utilized (CloudGarden Talking Java, 2002). CloudGarden is a software package that implements the Java Speech Application Programming Interface (API), allowing for simple access to the lower software layer (Java Speech API Programmer's Guide, 1998).

A list of English words and their corresponding frequencies of use served as another tool that was needed to conduct this research. These statistics were used to create the coding of the words using the Huffman scheme. The Edict Virtual Language Centre provided the 5,000 most frequently used words in the English language and their respective frequencies for this project, which were acquired from the Brown Corpus Manual (2002). The corpus contains 500 text samples of about 2,000 words, resulting in a list of more than one million words. The categories of the text samples were diverse, ranging from politics to religion to excerpts from romance novels. These text samples were analyzed to find the frequencies of words that are most commonly utilized in American literature. The 5,000 words were ordered and listed in a small file that was parsed by the lexeme coder. Table 1 shows the first 10 lines from the word file.

Creation of the Huffman Tree

The frequency information is used to create an efficient coding of the lexemes. Figure 2 shows an example of how Huffman coding differs from "non-varying" or "fixed" coding (Hamming, 54).

Table 1
First 10 entries of the word list.

Word Order	Word	Frequency %
1	the	6.8872
2	of	3.5839
3	and	2.8401
4	to	2.5744
5	a	2.2996
6	in	2.1010
7	that	1.0428
8	is	0.9943
9	was	0.9661
10	he	0.9392

Fixed code representation	S1	S2	S3	S4
	00	01	10	11
Huffman code representation, assuming the probabilities of occurence as shown	S1	S2	S3	S4
	45%	30%	15%	10%
	0	11	101	100

Figure 2
A comparison of Huffman and fixed code representations

The fixed code representations of S1 through S4 each use two bits (0s and 1s) to represent four unique values, while the Huffman code representations range from one to three bits. Although S3 and S4 of the Huffman code require three bits instead of two, probability dictates that S1, which requires only a single bit, will be coded most often. Therefore, because S1 occurs 20% more often than S3 and S4 combined, the resultant Huffman code becomes smaller than the fixed length code.

The key data structure used in the lexeme coder is the binary tree, which consists of a data structure composed of "tree nodes." Each tree node can have zero, one, or two "children." The node at the highest level of the tree is called the "root node." Nodes with one or two child nodes are known as "branch nodes," and nodes with no children are known as "leaf nodes."

In this method, the first step in coding speech is to establish the codes of the words. First, the word file (Table 1) is parsed, and the words and their respective frequencies of use are loaded into the system. Each word and its respective frequency are loaded into a tree node object, and those nodes are stored sequentially in a list. Since the words are sorted in descending order according to their frequency within the file, the ordering of the words is preserved in the list. Once the system has parsed all the words, it builds the binary Huffman tree, as the pseudo code excerpt shows.

```
1   while (there is more than one element in the list)
2       get the last and second to last nodes of the list
3       create a new branch node
4       make the last node the left child of the new node
5       make the second to last node the right child of the
        new node
6       calculate the frequency of the new node
7       put the node back in its appropriate position in the list
```

In line two above the last two nodes (i.e. the two nodes with the lowest frequencies) are pulled from the list. In line three a new tree node object is created, and in line four and five the two pulled nodes are associated with this new node. The node with the lowest frequency is made the left child of the new node, and the node with the second-to-lowest frequency is made the right child of the new node. The frequency calculated in line six is the sum of frequencies of the two pulled nodes. In line seven a search is performed on the list to find the appropriate position to put the new node, so that the ordering of the nodes is preserved. Figure 3 illustrates this process.

After the leaf and branch nodes have been established, binary codes must be assigned to each word.

```
1    loadFunction(theNode, currentCode)
2        if(theNode is a leaf node)
3            set the binary code of theNode to currentCode
4        else
5            if(the rightChild of theNode is not null)
6                add a "0" to the end of currentCode
7                call loadFunction(rightChild of theNode,
                 currentCode)
8            if(the leftChild of theNode is not null)
9                add a "1" to the end of the currentCode
10               call loadFunction(leftChild of theNode,
                 currentCode)
```

The system calls "loadFunction," which is a recursive function that holds the root node and an empty string object as its parameters. The "theNode" value is the current tree node of the Huffman tree being considered, and "currentCode" is the binary code that has been generated up to this point. Line two checks if "theNode" is a leaf node. If not, then the node's binary code (and in turn the binary code of the word the node is wrapped around) is set to "currentCode," and the function call ends there. If it is not, then the function checks to see how many child nodes "theNode" has. If a right child node exists, then a "0" is added to the binary code and another recursive call to "loadFunction" is made with

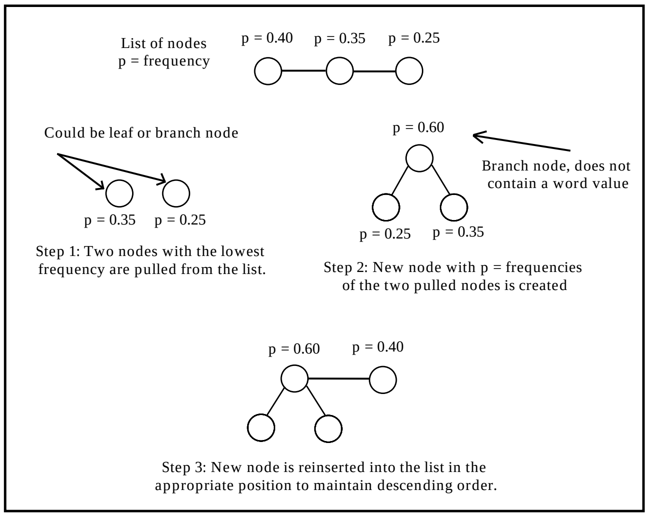
List of nodes
p = frequency
p = 0.40
p = 0.35
p = 0.25
Could be leaf or branch node
p = 0.35
p = 0.25
Step 1: Two nodes with the lowest frequency are pulled from the list.
p = 0.60
Branch node, does not contain a word value
p = 0.25
p = 0.35
Step 2: New node with p = frequencies of the two pulled nodes is created
p = 0.60
p = 0.40
Step 3: New node is reinserted into the list in the appropriate position to maintain descending order.

Figure 3
One process in the creation of the Huffman tree based on the pseudo code in Table 3

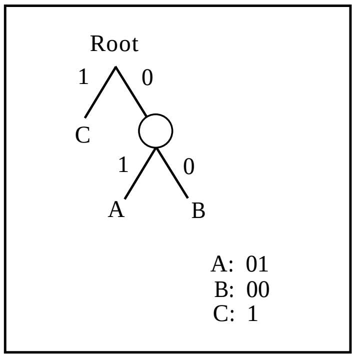

Figure 4
Binary code assignment

"rightChild" and the updated "currentCode" as parameters. If a left child node exists, then a "1" is added to the binary code and a recursive call to loadFunction is made with "leftChild" and the updated "currentCode" as parameters. Thus at every branch node, the function either branches right first and then left, until it reaches a leaf node.

Figure 4 shows an example of how the codes are generated. In Figure 4, a branch left adds a "1" to the code, and a branch right adds a "0" to the code. Note: since the value "C" in the figure has the smallest binary representation, it can be assumed that "C" has the lowest frequency value, based on how Huffman trees are generated for the research system. Table 2 shows a few words that are listed in the word list, and their binary representations.

Coding

Once the Huffman tree has been generated and the binary representation for each word is specified, coding a particular word is simply a matter of searching through the list of words for the particular word, and returning the binary code associated with that word.

Table 2
Listed words and their binary representations

Word	Code	Frequency
The	0010	6.8%
Was	111111	0.96%
Dreamed	111101011110000	0.0019%
Mexico	111101011110100	0.0019%

Unfortunately, the lexeme word coding system can present a problem when a word that is not within the list is encountered. To alleviate this problem, a custom coding scheme is utilized for words that are not within the 5,000-word list. For unlisted words, the number of characters of the word is counted, and since all 26 letters of the English alphabet can be coded with five bits, binary representation of each character of the word, along with the binary value of the number of characters of the word, becomes the coding for this particular word. This scheme is shown in the next pseudo code excerpt.

```
1  if word is in the word list
2      return word binary code
3  else
4      create variable returnString
5      add unlisted word flag binary code to returnString
6      add binary representation of the number of
       characters of the word to returnString
7      for each character of the word
8          add binary representation of the character to
           returnString
9  return returnString
```

The "unlisted word flag" binary code added in line five is "0000001;" seven bits that signify the block of binary code as representing an unlisted word. A special leaf node is created in the Huffman tree that can be reached with the path involving six right branches and one left branch ("0000001"). If this leaf node is reached, then the system will know to decode the next block of code for the unlisted word. In line six the binary representation of the number of letters of the word is added to the "returnString" variable (i.e. if the word were seven letters, "0110" would be added at the end of "returnString"). In lines seven and eight, the letters of the word are coded and concatenated one at a time. Finally, in line nine the calculated binary representation for the particular word is returned. Figure 5 shows that

46 bits are required to represent the seven-letter word "prosody."

Decoding

Decoding the binary code is a relatively simple process, with the exception of decoding the unlisted words. To decode words that are in the 5,000-word list, the system simply starts at the root node, branches to the left or right child node of the root (based on the value of the first bit of the code), and repeats the process for the next node and so forth. To decode the unlisted words, the three parts of the binary code must be recognized: the unlisted word flag, the number of characters, and the binary code for each of the characters themselves. The next code excerpt demonstrates this process.

```
1    create new String returnString
2    variable current:= root node
3    for each bit of binary input
4        if bit is 1
5            current:= left child of current
6            if current is a leaf node
7                if leaf node is unlisted word node
8                    read in 4 bits
9                    calculate the number of letters
10                   for each letter
11                       read in 5 bits
12                       calculate letter
13                       add letter to returnString
14               else
15                   add binary value of leaf node to
                     returnString
16           current:= root node
17       else if bit is 0
18           current:= right child of current
19       <repeat lines 6 through 17>
20   return returnString
```

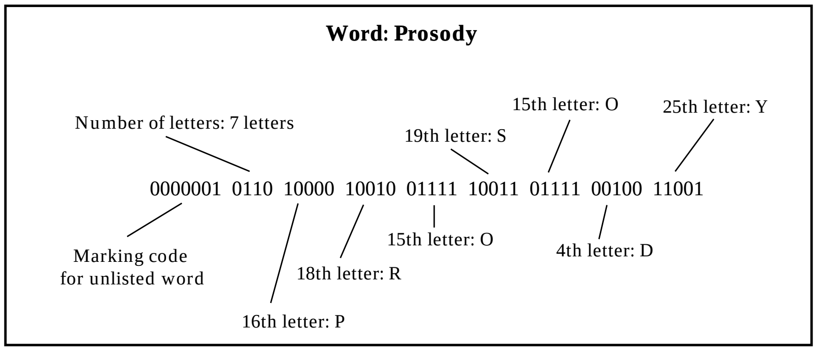

Figure 5
Binary representation of an unlisted word

Line one demonstrates that the "returnString" variable holds the data that will be returned after all the bits have been decoded. "Current" is a reference to the Huffman tree node that is currently under consideration. The reference starts at the root of the tree, and each bit changes the reference to either the left or right child of that particular node. Line three begins the "for" loop that iterates for each bit of the input. If the current bit is "0" (line 17) then the code jumps to line 18. Otherwise it continues to line five. At line five the current bit is "1," indicating a branch to the left. Thus, the reference of "current" changes to the left child of "current." If the node is a leaf node, and if the node is a listed word, then the binary representation of the word will simply be added to "returnString," and the program will continue. However, if it is the unlisted word flag, then the next batch of binary code is decoded. The first four bits are read to determine the number of characters that compose the word. The loop in line 10 will iterate for the number of times equivalent to the number of characters calculated in the previous line. For each iteration, five bits are read in, the character represented by the binary code is determined, and that character is added to the "returnString" variable. At the end the "current" reference is shifted back to the root node of the Huffman tree, so that a new word can be decoded. The loop in lines 6 through 17 is repeated after line 19 with a different "current" reference. At the end of the function, the "returnString" value is returned, which is composed of all the words that were decoded.

System Testing

Speech Samples

To test the capabilities of the system, the diverse test samples were used to better estimate the efficiency of the algorithm. Table 3 shows the six samples that were used for testing purposes, and provides a brief description of their content.

Recognition Error Testing Procedure

A set of rules was developed to determine the number of errors made by the system in recognizing spoken speech. The number of errors is determined by comparing the original text samples to the transcripts of the recognized speech. The two sources are compared word-by-word for any discrepancies. If a word in the text sample is not appropriately recognized by the system, an error is flagged. This includes the following cases: incorrect word or words in place of the correct word, a missing word, or entirely extraneous words. Note that these cases are not cumulative: for example if the words "I do" appear in the recognized transcript, instead of being recognized as "Idol," a single error is counted. Once the number of errors is counted, the percentage error is tabulated by dividing the number of errors by the total number of words of the particular text sample.

Table 3
Diverse text samples used to better estimate how efficient the algorithm is

Sample	Number of Words	Content Description
1	114	Description of how OpenGL (graphics programming library) models light and its different components (ambient, diffuse, and specular light components).
2	89	Description of how OpenGL (graphics programming library) handles perspective project transformation so as to model depth in a 2-D environment of a monitor.
3	96	Explanation of the virtual memory systems that appear in contemporary computers, as well as the issues involved with using it.
4	67	Definition of so-called "legacy software," and the challenge it presents to computer scientists and professional developers alike.
5	88	Passage from a children's story about the difference between astronauts and "aquanauts" (Schaffert, 1983).
6	109	Passage from a children's story about the Norse myth of Thor and the Giants (Pyk, 1983).

Table 4
Bit-rate data

Sample #	# Words	% Words Unlisted	Length of Speech (seconds)	Gzip Compression File Size (bytes)	Lexeme Coder Compression File Size (bytes)	Bit Rate of Coder (bps)
1	114	15%	42	371	245	47
2	89	17%	35	343	220	50
3	96	21%	37	366	195	42
4	67	24%	29	296	173	48
5	88	10%	28	308	148	42
6	109	13%	29	260	173	48

Human Testing Procedure

The last phase of the research is human testing, where human recognition errors are compared to the speech engine recognition errors. Five test subjects sat in front of a computer and listened to the synthesized speech samples. The samples were played one sentence at a time through the Microsoft Text-To-Speech synthesizer. The subjects could ask for the current sentence to be replayed, but they could not go back to a previous sentence, nor could they pause in the middle of a sentence. The test subjects were not made aware of the content of the samples, but they were made aware of possible errors that may be present within the text samples. The test subjects were advised not only to type what they heard, but also to make an attempt to infer information based on the context of the sample and to try correct the errors. For example, if the test subject was able to determine the context of a particular passage to be mountain ranges, and he or she heard the phrase "the apple ate ton mountains are huge," perhaps he or she may be able to infer that "apple ate ton" really is supposed to be "appalachian." Upon completion, perceptual errors were counted by comparing the transcribed texts to the original text samples, not the synthesized transcripts the subjects listened to (i.e. the words that were recognized by the Speech Recognition engine). The same rules for counting errors as mentioned above were used. This study was approved by the Institutional Review Board (IRB) of UCI, under protocol #2003-2947.

Experimental Findings and Results

There were two important categories of data that were collected when testing the lexeme coder. The first is the data rate of the coding scheme, measured in bps. The second is the error rate: the number of recognition errors made by the speech engine, as well as the number of recognition errors made by human listeners. The data rate of the coding scheme is measured as follows: When spoken, the text samples are of a certain length of time and the coding of each sample is saved to a binary file. Next, the bit rate was determined by dividing the size of the bit file by the number of seconds of spoken text. Table 4 shows all the data necessary to determine the bit rate.

As shown in Table 4 the bit rate varies between 40 and 50 for the six samples. As expected, the samples with a higher percentage of unlisted words tend to have a higher bit rate since the unlisted words require the custom-built codes (i.e. sample four with only 67 words has the same bit rate as sample six with 109 words). Nonetheless, the variance is not so high as to suggest that a new coding scheme is needed. Because the synthesized samples were continuous, with breaks only for punctuation, and the speech samples were slowed to a more natural rate of speech, the bit rate would be even lower, on the order of 30 bps or so.

Table 4 also provides a comparison against the GZip tool, as a way of measuring the efficiency of our lexeme coder. GZip

Table 5
Error data for both the speech engine and the human test subjects

Sample #	# Words	% Words Unlisted	% Recognition Errors	% Perceptual Errors	Difficulty Rating
1	114	15%	11%	11%	4
2	89	17%	17%	11%	5
3	96	21%	5%	6%	3
4	67	24%	7%	6%	2
5	88	10%	17%	13%	6
6	109	13%	7%	5%	1

(or GNUZip) is based on the loss-less coder presented in Ziv and Lempel (1978). This GZip functions at the character level, whereas this coder works at the word level, which achieves 30% to 50% more compression than GZip.

Table 5 demonstrates that the percentage of words that are unlisted does not appear to have any bearing on error percentage. For example, samples three and four have the highest percentage of unlisted words, and yet they have some of the lowest numbers of recognition and perceptual errors. By comparing the recognition and perceptual errors, there appears to be a linear relationship: as one increases, so does the other. But variation in error percentage can only be explained by looking at the Difficulty Rating as well. The Difficulty Rating is a subjective measure established for each sample, based on the complexity of the sentence structure, as well as the complexity of the content. The content complexity takes into account the fact that all of the subjects were, in one way or another, part of the Computer Science Department (i.e. students, professors). The difficulty level of the samples were based on how specialized the information was. Sample six is a passage from a children's book about Norse legend, with little or no word complexity. Sample four discusses the problem of legacy software with terminology the majority of people in the computer industry should be able to understand. Sample three discusses virtual memory, another commonly known concept, though with slightly more complex terminology. Samples one and two discuss algorithms used in computer graphics, specifically in the OpenGL library (information that is familiar to most people in the graphics industry). Finally, sample five is another passage from the children's book, that discusses the difference between astronauts and aquanauts. Although the sentence structure is quite simple, this sample was given the highest difficulty rating because the word "aquanaut" is not a word that the Microsoft Speech Engine can recognize. Therefore, it would be hard for the human subjects to understand what the sample is about if one of the key terms never transferred accurately. This sample was the only sample where the Difficulty Rating was based on the recognized speech, not the original passage.

When the system was first developed, it was expected that the listeners would get at least as many perceptual errors as recognition errors, if not more. However, the results show that in four of the six samples the listeners actually transcribed fewer errors than they heard. This observation can be attributed to the listeners' ability to detect many of the recognition errors, and infer what was supposed to be recognized. This can be seen by looking at sample two, where the subjects were able to infer more than a third of the technical jargon that was missed by the recognizer.

In contrast the subjects were able to correct fewer errors in samples five and six, which presented topics with which testing subjects possessed little familiarity. These results imply that it is feasible to code speech at extremely low bit rates by doing a word-level analysis of the speech signals. Furthermore, the results demonstrate that the more understanding the listeners have with the context of the spoken selection, the fewer number of perceived errors will occur, and a larger number of recognition errors will be caught and corrected.

One limitation with the lexeme coder is the way that the speech was played back through a synthesized voice. This was unavoidable since only the words from the speech were coded. However, the lack of extra speech features, such as tone or pitch, make the speech completely unnatural and most likely contributes to the number of perceived errors.

Conclusion

The results of this research show that it is possible to code speech at extremely low bit rates using simple tools for the process. We were able to achieve between 80% and 96% recognition accuracy from the speech engine itself, and a 87% and 95% perceptual recognition accuracy. It is possible that the better the speech recognition engine is, the fewer the perceptual errors. Regardless, the lexeme speech coder can be immediately put into use for purposes such as memo recording and retrieving, where the speaker and listener are the same person and share the context of the messages. This system can also be used for military applications where relatively small vocabularies are used, and the ability to transmit orders as quickly as possible is imperative. However, a drawback with the current lexeme coder is the unnaturalness of the synthesized voice. We believe that the coding of pauses and stresses, and the use of speech features such as tone or pitch that are currently ignored, will greatly increase the quality of the synthesized speech while preserving the data rate in the order of 40 bps. That work is currently under investigation.

Acknowledgements

I would like to thank Dr. Cristina Lopes for giving me the inspiration for this project, and for her continual support through the design, development and documentation of the research. I also want to thank the participants who agreed to take part in the human testing. This research project was supported by the Summer Undergraduate Research Program and the Undergraduate Research Opportunities Program at UCI.

Works Cited

Brown Corpus Manual. Ed. Anne Lindebjerg. 11 Sept. 1997. Brown University. 20 Dec. 2002. www.hit.uib.no/icame/brown/bcm.html.

CloudGarden Speaking Java. Ed. Jonathan Kinnersley. Ver 1.6.2. 08 Mar. 2001. CloudGarden. 10 Dec. 2002. www.cloudgarden.com.

Edict Virtual Language Centre. Word Frequency Text Profiler. Edict. 15 Jan. 2003. www.edict.com.hk/textanalyser.

Furui, Sadaoki. Digital Speech Processing, Synthesis, and Recognition. 2nd ed. New York: Marcel, 2001.

Hamming, Richard W. Coding and Information Theory. New Jersey: Prentice, 1980.

Hirata, Yoshimitsu and Seiich Nakagawa. "A 100bit/s Speech Coding using a Speech Recognition Technique." EUROSPEECH-89 1989: 290-293.

Huffman, David A. "A Method for the Construction of Minimum Redundancy Codes." IRE Proceedings 1952: 1098-1101.

Java Speech API Programmer's Guide. Vers. 1.0. 26 Oct. 1998. Sun Microsystems. 06 Jan. 2003. http://java.sun.com/products/java-media/speech/forDevelopers/jsapi-guide/index.html.

Lee, Ki-Seung and Richard V. Cox. "TTS Based Very Low Bit Rate Speech Coder." ICASSP-99 1999: 181-184.

---. "A Segmental Speech Coder Based on a Concatenative TTS." Speech Communication 38 (2002): 89-100.

Markel, J. D. and A.H. Gray Jr. Linear Prediction of Speech. Berlin: Springer-Verlag, 1976.

Microsoft Speech. 10 Sept. 2003. Microsoft Corporation. 26 Dec. 2002. http://www.microsoft.com/speech/

Papamichalis, Panos E. Practical Approaches To Speech Coding. New Jersey: Prentice, 1987.

Pyk, Ann. "The Hammer of Thunder." Secrets and Surprises. New York: Macmillian, 1983. 106-115.

Supple, Lynn M., R. Cohn, J. Collura, and A. McCree. "MELP: The New Federal Standard at 2,400 bps." IEEE International Conference on Acoustics, Speech and Signal Processing. 21 Apr. 1997. Munich, Germany. Volume 2, 1591-1594.

Tokuda, Keiichi, Takashi Masuko, Jun Hiroi, Takao Kobayashi, and Tadashi Kitamura. "A Very Low Bit Rate Speech Coder Using HMM-Based Speech Recognition/Synthesis Techniques." ICASSP-98 1998: 609-612.

Ziv, Jacob and Abraham Lempel. "Compression of Individual Sequences Via Variable-Rate Coding." IEE Transactions on Information Theory 24.5 (1978): 530-536.