Uvox: A Novel Text-to-Speech System

Student: Eric Schmidt
Supervisor: Boris Katz
6.UAP Final Paper
December 2015

Abstract

Modern text-to-speech systems employ a few major methods when synthesizing speech, each with its own benefits and drawbacks. This paper outlines a novel system, named Uvox, which addresses the drawbacks of these techniques and aims to create a middle ground by combining principles of unit selection synthesis and formant synthesis. At the core of the system are a unit selector/concatenator and a vocoder. By using these two stages in speech synthesis, we can achieve a final sound that is easily controllable but also maintains a natural quality.

Background

Interest in synthesizing human speech sounds has existed for centuries. The first models of the human vocal tract were produced in 1779, and were capable of reproducing the five vowel sounds [a], [e], [i], [o], and [u][1]. After this came many attempts at building mechanical speaking machines, but the advent of audio signal processing allowed much progress to be made. Electronic systems emerged in the 1950s, and since then, a few major methods have been used to synthesize human speech.

Relevant to the current work are the methods of *unit selection synthesis* and *formant synthesis*. In unit selection synthesis, a database of individual sound recordings is maintained and used in order to construct novel utterances. For example, in producing the word 'home', which can be phonetically represented as [hom], the system would locate recordings of the sounds [h], [o], and [m]. It would then (ignoring the fine details) concatenate these sounds, producing a

result similar to [hom]. Formant synthesis, on the other hand, does not use pre-recorded sound and instead relies on acoustic properties of human speech to completely synthesize sounds using additive synthesis. Thus, in producing the word 'home', such a system uses basic waveforms such as noise and oscillators to create sounds with the same frequency spectra as [h], [o], and [m], and blends these together by modifying the power of the pertinent frequencies over time.

Both techniques of course have drawbacks. Unit selection synthesis requires a large database of recorded sounds, perhaps even more than one recording of each sound in order to ensure that sounds can be chosen that blend together in a non-jarring way when concatenated. In fact, many unit selection techniques use diphones[2] – recordings of the transitions between sounds – in order to achieve good results. As one may imagine, maintaining a database of diphones is expensive because the amount of memory used is quadratic in the number of sounds of the language. On the other end, formant synthesis requires prior knowledge and precise control of a large number of parameters, including formant frequencies, pitch, and volume of speech sounds.

Thus we see that with the current methods described above, we have the following options. We can more easily generate speech by simply looking up individual recordings and stringing them together, which produces a natural-sounding result but requires large amounts of memory and offers little fine control over the output speech. Alternatively, we can generate speech waveforms from scratch, which offers very intricate control of the result but requires more computational power and produces a more artificial sounding result.

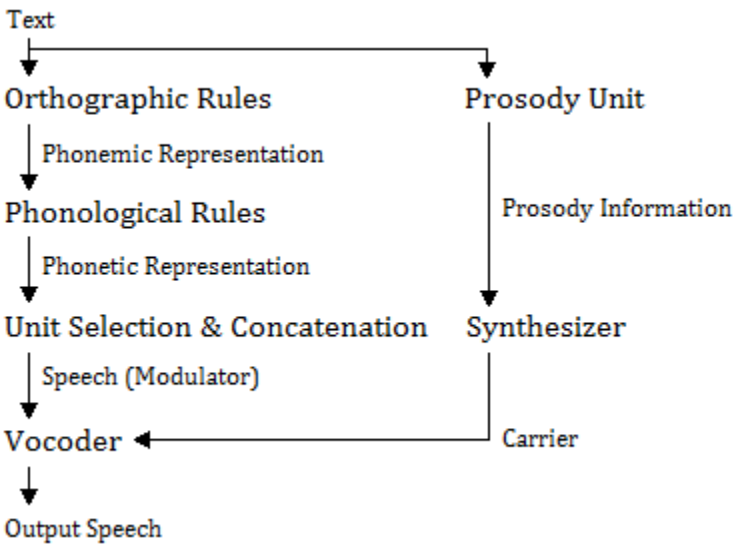
System Description

Uvox (so named as an abbreviation for *Unit Selection + Vocoder*) is a text-to-speech system that uses ideas of both unit selection synthesis and formant synthesis, with the aim of finding a

middle ground between the two techniques. The underlying method of forming utterances is unit selection, with a stage of vocoder (explained below) applied afterward. This allows for less computation when combining sounds, and maintains a degree of realism by using recorded human speech. The addition of the vocoder stage offers the finer control that formant synthesis does, allowing the system to control the prosody of the output speech. As will be apparent from the structure of the system, this two-stage approach also offers a modularity that gives Uvox the potential to be a very robust system.

Before describing the system itself, it is useful to briefly discuss *vocoder*, one of the fundamental components of Uvox. Vocoder is a technique that essentially applies the sonic characteristics of one sound (the *modulator*) to another (the *carrier*). In technical terms, the frequency spectrum of the modulator is computed and used to filter the carrier such that the frequencies present in the modulator are emphasized in the carrier. This produces a new sound that has the timbre of the carrier, but the articulation of the modulator. This is the effect that is often used to produce robotic-sounding voices for music and film, and is achieved by using a voice recording as the modulator and some frequency-rich synthesized sound as the carrier.

With this in mind, the following is an overview of the full system structure:

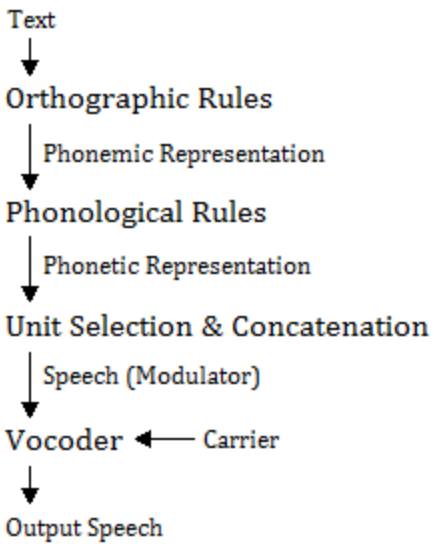


As one can see, the system is divided into six modules. This allows for a high degree of flexibility – individual components can be modified to change the output speech in myriad ways. For example, one would be able to change the accent of the output speech by substituting a different set of phonological rules for the same language. One would also be able to define their own orthography with custom orthographic rules, or modify the timbre of the output speech by adjusting parameters within the synthesizer. These traits make Uvox a system with great potential for usefulness and customization without prohibitively large amounts of extra work.

Implementation

The system outlined in the diagram above is the full theoretical version of Uvox. For the purposes of maintaining a reasonable scope of work to be accomplished in the given time frame, not all components of the system were fully implemented this semester. The goal was to create a proof-of-concept to demonstrate that such a system can be successfully built, and this goal was reasonably achieved. The system has been implemented in Python.

The implemented aspects of the system are shown in the following diagram:



The details of each system will be described below.

Orthographic Rules

The first subsystem encountered by the input text is the one that handles orthographic rules. For a good text-to-speech system, users must be able to enter text in the orthography of the target language. Writing phonetically would easily allow synthesis of any utterance, but using the native writing system of a language keeps Uvox robust and has the additional benefit of modeling the translation from graphemes into phonemes. The job of this subsystem is to convert a written word into a list of phonemes, which is handled by the next subsystem.

To maintain modularity, a simple language to describe orthographic rules was designed. The format of this language is illustrated in the snippet below:

```
// English orthographic rules
ou# -> u
ack -> ae k
eck -> eh k
ick -> ih k
uck -> uh k
ock -> a k
qu -> k u
x -> k s
...
```

The general syntax for a rule is `spelling -> phonemes`, where `spelling` is the sequence of written characters to match (`#` can be used for a word boundary) and `phonemes` is a space-separated sequence of phonemes represented by the characters. The rules apply in order, so longer character sequences should be placed at the top in order to find the maximal matches. The representations of phonemes (such as `ae` or `eh` in the example) are arbitrary but must correspond to the phoneme representations used in the phonological rules, explained in the next section.

In order to match spellings, the orthography module first pads the input word with boundary markers and then scans across this string from left to right, attempting to match each character sequence in order. Upon finding a match, the corresponding phonemes are added to the phonemic representation and the matcher steps forward to continue checking the remaining text. To illustrate this process, let us consider a brief example.

Consider the set of rules `ome# -> oh m; h -> h`, and the input 'home'. The matcher first pads the input with boundary markers, producing `#home#`. It then begins attempting to match spellings at the first character, `#`. No match is found, so it moves one step forward. A match is found at `h`, so `h` is added to the phonemic representation, and matcher moves forward. Then `ome#` matches with the rest of the padded input, so the phonemes `oh` and `m` are added to the representation and the matcher has considered the entire input string. The resultant phonemic representation is `[h, oh, m]`.

For a language like English, a large set of orthographic rules may be necessary. However, the use of a language to describe the rules allows the user to define a custom orthographic system. This means the user is free to enter input in whatever format is most desirable.

Phonological Rules

Once the phonemic representation of a word has been determined, the next step is to find its phonetic representation. This is achieved by the phonology subsystem, which employs a list of phonological rules that describe the actual produced sounds corresponding to each phoneme or sequence of phonemes. This system has the same benefits as the orthography system – not only does it model the process of translating phonemes to phones, but its modularity keeps the overall system robust and allows the user to customize pronunciations.

A language very similar to that of the orthographic rules is used to describe the phonological rules. The following is an example:

```
// English phonology
ae -> unrounded,low-mid,front
eh -> unrounded,mid,front
ih -> unrounded,high-mid,central
oh -> rounded,mid,back
th -> unvoiced,dental,fricative
dh -> voiced,dental,fricative
p -> unvoiced,bilabial,stop
...
```

The general syntax of a phonological rule is `phonemes -> phones`, where `phonemes` is a space-separated list of phonemes (again, `#` can be used as a boundary marker) and `phones` is a space-separated list of sounds, described by their classification in the International Phonetic Alphabet (IPA). The description of each sound is a comma-separated list of its features in the order `voicing,place,manner` for consonants and `rounding,height,backness` for vowels: for example, the voiced alveolar stop /d/ would be represented as `voiced,alveolar,stop`, and the unrounded high front vowel /i/ would be `unrounded,high,front`. As stated earlier, the representation of phonemes must match those used in the orthographic rules. The phonological rules also apply in order, and the matching algorithm is identical to that used to match orthographic rules.

Let us continue our demonstration with the word 'home'. Once the orthography subsystem has produced the phonemic representation `[h, oh, m]`, the phonological rules may apply. Consider the set of rules `oh -> rounded,mid,back; h -> unvoiced,glottal,fricative; m -> voiced,bilabial,nasal`. The system first pads the phonemic representation with word boundaries, resulting in `[#, h, oh, m, #]`. Then it starts from the left – no match is found for `#`, so it continues. The phoneme `h` matches a rule, so the sound `unvoiced,glottal,fricative` is added to the phonetic representation. The phoneme `oh` also matches, so `rounded,mid,back` is added, and lastly, `m` matches and `voiced,bilabial,nasal` is added to the phonetic representation. The final `#` does not match any rules, leaving us with the phonetic representation `[unvoiced,glottal,fricative; rounded,mid,back; voiced,bilabial,nasal]`.

At its simplest, the list of phonological rules can serve as a map from phonemes to individual phones. However, the language can also be used to describe phonological processes such as final devoicing with rules like `d # -> unvoiced,alveolar,stop`. This modularity allows the user to control the phonology of the language, allowing for any language or even specific accent of a language to be produced using Uvox.

Unit Selection & Concatenation

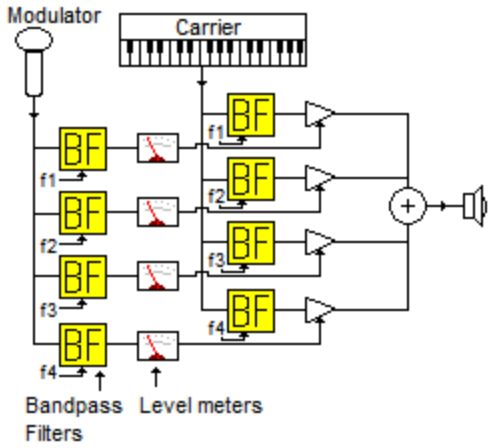
The next subsystem is part of the core of Uvox. This is the unit selector and concatenator, which creates the modulator input to the vocoder stage. It relies upon a collection of pre-recorded sounds in order to construct utterances. Recordings are stored as very short WAV files; in the current implementation, there is one recording per IPA sound used in the English language. In a full implementation, a recording for every sound in the IPA would be stored.

This module, despite being at the core of the system, is the simplest. Once the phonetic representation has been created, the unit selector simply finds the sound file corresponding to each phone, and concatenates them in order. Uvox maintains a dictionary mapping sound features to the filenames of the corresponding recordings to facilitate this. It writes the result to a special file that is then used as input to the vocoder. To achieve this in Python, the built-in `wave` module is used to read and write WAV files.

Vocoder

The more complex part of the core of Uvox is the vocoder. As explained earlier, vocoder is a technique that applies the sonic characteristics of a *modulator* sound to a *carrier* sound. The output of the unit selector is a jarring string of separate recordings, so vocoder is used to smooth out the result. This is done by applying the characteristics of the concatenated phones to a synthetic voice sound.

A deeper understanding of how vocoder works is necessary to understand how it was implemented in Python. There are essentially two stages: first the modulator is filtered by a bank of bandpass filters and the level at each frequency is measured; then, the carrier is bandpass filtered at the same frequencies, and the levels at each frequency are multiplied by the levels from the modulator.



The diagram above shows a simple 4-band vocoder. Bandpass filters allow only frequencies around a certain value to pass through. The filter frequencies (`f1`, `f2`, `f3`, and `f4` in the diagram) are evenly spaced across the frequency spectrum. Level meters are used to measure the intensity of the modulator at each of the filter frequencies. These intensities are then essentially used to multiply the intensities of the carrier at the same frequencies. This is how the characteristics of the modulator are applied to the carrier; using a larger number of bandpass filters increases the faithfulness of the carrier to the modulator's articulation. Uvox uses a 20-band vocoder.

The Python vocoder implementation is courtesy of the Pyo library[3], a Python package for digital signal processing. The implementation does exactly the process described above: for each band, it creates a bandpass filter on the modulator, a level meter to follow the intensity of the filter output, and a bandpass filter on the carrier with its output multiplied by the intensity measured by the level meter. This functionality is wrapped in a simple `Vocoder` class that stores the modulator and carrier sounds as properties.

Overall

Now that each subsystem has been described, the overall system functionality can be understood. When the user enters a sentence, the system first splits it into words. Each word then passes through the first two subsystems, resulting in a phonetic representation for each. These phonetic representations are then sent to the unit selector and concatenator, which concatenates individual sounds for each word, and then concatenates each word together with a small period of silence in between. This result is written to a WAV file. The vocoder then reads this file as the modulator, and another WAV file as the carrier, and plays the result.

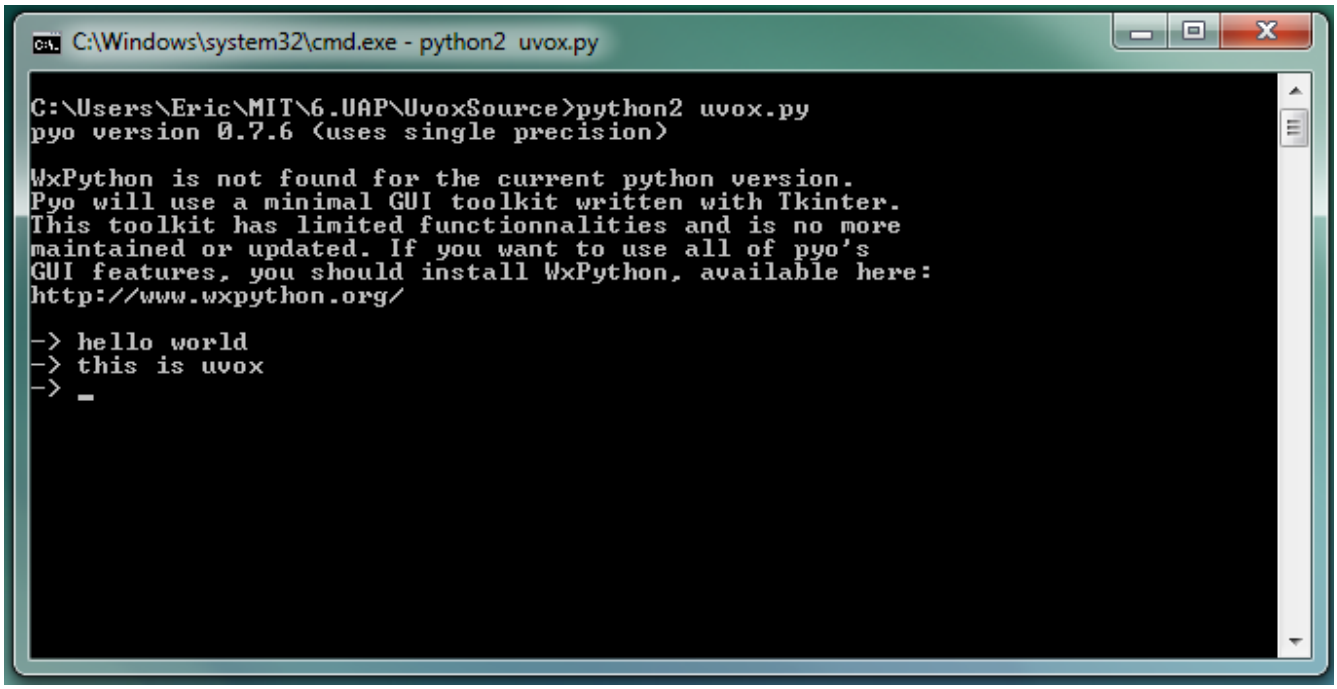
Uvox is written as a Python script and is called from the command line. It accepts four optional arguments: the file to use as the vocoder carrier, the file containing the orthographic rules, the file containing the phonological rules, and a flag determining whether to display the phonemic representation as the speech is played back. The default arguments use white noise as the vocoder carrier, English orthographic and phonological rules, and no printing of the phonemic forms. These arguments allow the results of the system to be easily modified without changing any source code.

Conclusion

Initial results from the first implementation of Uvox have been promising, with many of its utterances able to be understood by humans. There is much room for improvement; the concatenation process currently makes the output speech somewhat choppy, but a better method of blending the sound units together would produce smoother speech. The quality of the individual sound recordings for unit selection could also be better in order to improve the intelligibility of the speech. In addition, a more robust language for phonological rules could be designed to allow the user to match only certain features of a phoneme rather than a specific

phoneme, which is a limitation of the current implementation. In the future, instead of reading the vocoder carrier from a WAV file, the prosody and synthesizer units from the original theoretical design could be built, allowing for more control and context sensitivity of the voice.

Though areas for improvement are many, the goals laid out for this project at the beginning of the semester were satisfactorily achieved. It has been demonstrated that unit selection and vocoder can be combined to produce reasonably good synthesized speech. The core of the system was able to be implemented programmatically in Python, and progress was made on the orthographic and phonological modules. Additionally, the robustness of the system is illustrated by the fact that it can handle English text, which is more difficult than the Dutch text originally proposed for testing. As a new text-to-speech system, Uvox leaves much to be desired, but it shows an interesting technique that can hopefully be further developed in the future.



Uvox running in the command prompt

References

1. S. Lemmetty (1999), “History and Development of Speech Synthesis”, In *Review of Speech Synthesis Technology*. Retrieved from http://research.spa.aalto.fi/publications/theses/lemmetty_mst/chap2.html.
2. C. Hamon, E. Moulines, and F. Charpentier (1989), “A diphone synthesis system based on time-domain modifications of speech”, *Proc. Int. Conf. Acoust., Speech, Signal Proc., Glasgow,* pp. 238-241.
3. O. Belanger (2015), “Pyo: Dedicated Python module for digital signal processing”. Retrieved from http://ajaxsoundstudio.com/software/pyo/.