Recognition of Sound Sample Sequences Using Waveform Analysis: Detecting Stutter Words in an Audio Stream

Abstract

The focus of this research is an investigation into waveform analysis of an existing audio file. The investigation developed and tested an algorithm that identifies a given specified word/section within an audio stream file. The user is prompted to pick a word that needs to be searched for either through a .wav file or through specifying a section within the currently represented audio file. After the word specification is set, an audio file can be searched through. The possible word transitions and sections matching the desired word will be marked and tagged in a visual manner on the audio stream, allowing the user to see where the positions if the target sound were found. Once the matches have been identified, the developed software will allow the user to play each section to see if there are any false positives within the search. Several classes were created in helping to develop the search method. The research included an investigation into new algorithms to search large audio streams for sample sounds. An application of the project was used to identify stutter words uttered during an interview.

Objective

The objective of this project is to develop a general single word search algorithm that allows for identification and removal of a specified word within an audio file through waveform analysis. The application of this objective is to reduce time for manual editing of audio streams. A specific use for this program is the identification of excessive stutter words in a voice recording.
The program will be written in the language of Java.

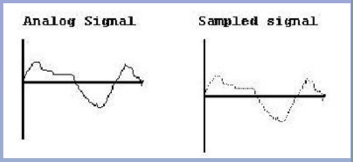
Challenges

- Most speech programs focus on real-time capabilities. This project is focused on the identification in pre-existing audio streams.
- Accuracy is key for the success of the algorithm. Low accuracy could actually hinder the search process rather than improve it.
- Waveform analysis of speech is difficult due to different styles and tones. This project investigated and designed an algorithm based on principles of waveform analysis.

Background

Sampling (Signal Processing)

•Sampling is the reduction of a continuous wave signal into a discrete wave signal. Samples are taken at equal time intervals and those are used to reconstruct the original signal. Since noise is considered a continuous wave, sampling is used in order to create a discrete form. Data loss may occur when converting a continuous signal to a discrete form if the sampling rate is too low. Therefore, a minimum sampling rate should be calculated.

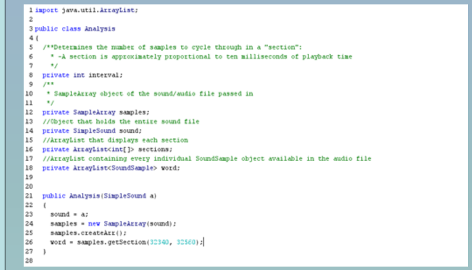


An analog signal and its sampled form.

•According to the Nyquist-Shannon sampling theorem, a perfect reconstruction requires a sampling frequency of at least twice the maximum frequency of the audio stream. (e.g., 8 kHz requires 16 ,000 samples/second).

Analysis

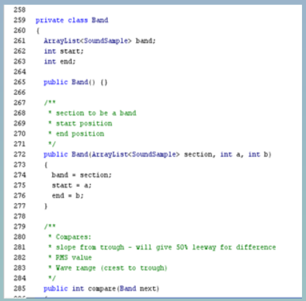
• Class Analysis encompasses all of the methods that pertain to waveform analysis.



```java
import java.util.ArrayList;

public class Analysis
{
  /**Determines the number of samples to cycle through in a "section":
    * -A section is approximately proportional to ten milliseconds of playback time
    */
  private int interval;
  /**
   * SampleArray object of the sound/audio file passed in
   */
  private SampleArray samples;
  //Object that holds the entire sound file
  private SimpleSound sound;
  //ArrayList that displays each section
  private ArrayList<int[]> sections;
  //ArrayList containing every individual SoundSample object available in the audio file
  private ArrayList<SoundSample> word;


  public Analysis(SimpleSound a)
  {
    sound = a;
    samples = new SampleArray(sound);
    samples.createArr();
    word = samples.getSection(32340, 32560);
  }

```

A section from the Analysis class.

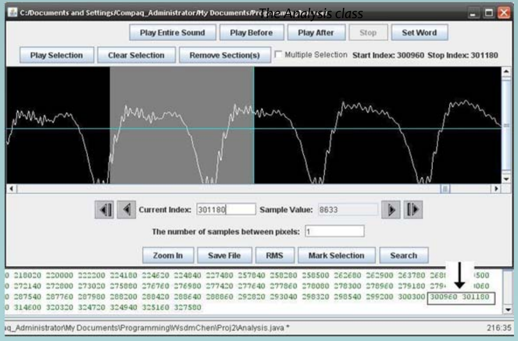
Part of the Band class, a nested class within Analysis. An object is created that represents a single time band.



```java

  private class Band
  {
    ArrayList<SoundSample> band;
    int start;
    int end;

    public Band() {}

    /**
     * section to be a band
     * start position
     * end position
     */
    public Band(ArrayList<SoundSample> section, int a, int b)
    {
      band = section;
      start = a;
      end = b;
    }

    /**
     * Compares:
     * slope from trough - will give 50% leeway for difference
     * RMS value
     * Wave range (crest to trough)
     */
    public int compare(Band next)
```

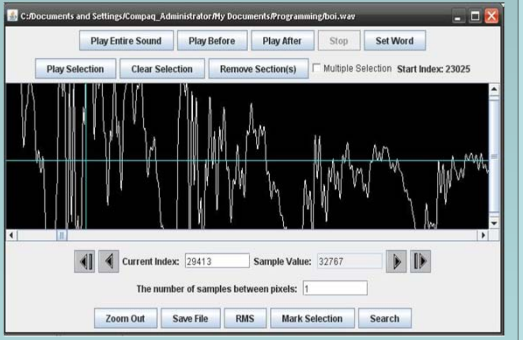
•The audio wave is split into time bands of relatively similar duration (~5 milliseconds).
•The final step is to perform automated waveform analysis on each individual band, recording the measurements stated.

The Analysis class



```
0 218020 220000 222200 224180 224620 224840 227480 257840 258280 258500 262680 262900 263780 268
0 272140 272800 273020 275880 276760 276980 277420 277640 277860 278080 278300 278960 279180 279
0 287540 287760 287980 288200 288420 288640 288860 292820 293040 298320 298540 299200 300300 300960 301180
0 314600 320320 324720 324940 325160 327580
```
```
q_Administrator\My Documents\Programming\WsdmChen\Proj2\Analysis.java *     216:35
```

The highlighted section is a single time band of roughly five milliseconds in duration. The size of the band in this particular file is 220 samples.



A transitional point in the audio stream, usually indicating a shift in phonemes.

• Several measurements will be taken to determine if there was a matching sequence in the stream:
  – Root mean square (RMS) of the samples.
  – Crest Factor – a waveform measurement, the crest factor is calculated by dividing the peak magnitude by the root mean square of the sample sequence. This program gave some flexibility by taking the three largest peaks above the line and averaging them.
  – Range (crest-trough) – Finds the value difference from the largest crest to the lowest trough of the sample sequence.
  –Average value – Takes the average value of the sample section.

•The specified bands from the sample are compared to the bands in the audio file. A value is then generated to determine whether the compared bands are a matching set. Note: some measurements are weighted more heavily in determining a match.

•The matching sections are then presented to the user to confirm the selection.

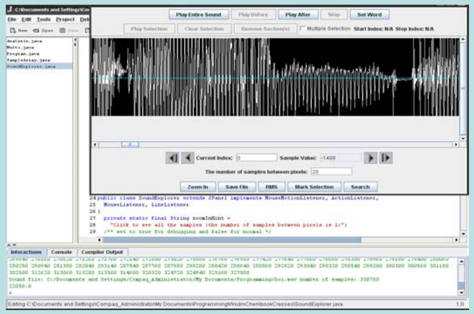
Media Computation Library

The Georgia Tech media computation library provides different classes and files related to media-based interactions. The project utilized the following classes.

• SoundExplorer – Creates an object that presents a waveform representation of the specified audio file. The graphical user interface of the class was modified to allow the user to perform the sample sound search.

• SoundSample – A class that represents one sample of the sound.

• SimpleSound – Represents the entire specified audio file.



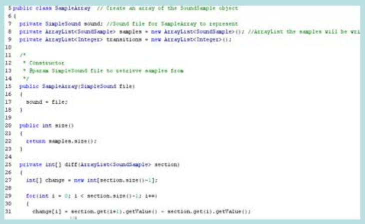
Execution of SoundExplorer class.

Extra Classes

Several classes were created to enhance the existing Media Computation library and in order to permit easier manipulation of other classes.

• SampleArray class – creates an object containing an ArrayList. The ArrayList holds every SoundSample object within the audio stream. This allows for simpler access to a large section of SoundSample objects instead of calling on each individual one.

• Multi class – keeps track of multiple selections and positions within the audio stream.



```java
public class SampleArray  // Create an array of the SoundSample object
{
  private SimpleSound sound; //Sound file for SampleArray to represent
  private ArrayList<SoundSample> samples = new ArrayList<SoundSample>(); //ArrayList the samples will be wri
  private ArrayList<Integer> transitions = new ArrayList<Integer>();

  /*
   * Constructor
   * @param SimpleSound file to retrieve samples from
   */
  public SampleArray(SimpleSound file)
  {
    sound = file;
  }

  public int size()
  {
    return samples.size();
  }

  private int[] diff(ArrayList<SoundSample> section)
  {
    int[] change = new int[section.size()-1];

    for(int i = 0; i < section.size()-1; i++)
    {
      change[i] = section.get(i+1).getValue() - section.get(i).getValue();
```

Portion of the SampleArray class

Applications

- There are a few media based applications in which this program will be useful.
  - Reduce manual editing time by lowering the amount of instances a word is present in the audio file. This would be especially useful with speakers that excessively stutter, or audio where the word occurs a large amount of times.
  - The program could serve as a method of searching through the sound file. Key words can be passed in to be searched for and determine whether it occurs or not. This could be applied to data mining.

Conclusion

- There was success in identification of a sample sound. The words identified correctly ranged from 20-50%. The variance in success is most likely attributed to the sample value range in the audio stream. In some cases, there were no correct identifications in the stream. Files with a very low range created more false matches than those with high ranges.

- Overall, this project showed that it is possible to automatically identify similar words, albeit, it may need a more thorough comparison in order to correctly match them.

Future Work

- *Decrease amount of false transitions and matches*: The main future focus is to reduce any false transitions or matches that may be present during a search.

- *Extended support for audio formats*: Extended support to popular extensions such as .mp3 or .ogg would have greater productivity compared to only supporting .wma.

- *Real-time Speech Recognition*: The program searches through pre-existing audio streams, but more use could be added if it was capable of analyzing speech in real-time.