Digital Signal Processing to Detect Catalytic Converter Theft

GreatGrey-52
8/8/2022

Table of Contents

GreatGrey-52/Detect_Recip_Saw: Detect Reciprocating Saw (github.com)

PURPOSE AND DESCRIPTION:

This project is intended to use Digital Signal Processing to Detect Catalytic Converter Theft. The project started after a neighbor's catalytic converter was stolen from the apartment parking lot. Once that happened, I started researching Digital Signal Processing and signal detection.

The Objective Signal of the project is from a battery-powered reciprocating saw operating at approximately 3,000 strokes per minute. The environment is a busy ("noisy") apartment parking lot. In this environment, the objective signal is challenging to classify at varying ranges while other noises in the environment interfere with or resemble the saw's signal.

A couple of notes:

- The work by Velardo, Lys, gabemagee, GianlucaPaolocci, and others has been very helpful to explore this issue
- The version included with this post makes progress to characterize the objective signal; false alarms may still occur (and collected) with this version
- Initial theories that looked for consistent events in the Objective Signal led to a high rate of false alarms; when I switched the approach to look for inconsistent events, the distinction between Other Signals and the Objective Signal became clearer

GOALS:

The initial design goals of the project were:

1. Detect a reciprocating saw within the area of a small parking lot (within a 20-90ft radius)
2. Use readily available parts
3. Minimize listen/process ratio
4. Distinguish a saw signal from normal sounds in the environment
5. Build a foundation for an Artificial Intelligence approach

To be safe and enduring, I request two things:

1. If you observe catalytic converter theft, DON'T CONFRONT, call 911 (many thieves are armed)
2. In online discussions, please don't give away procedural weaknesses; technical discussions should suffice

SIGNAL SAMPLING:

Sample signals for this project included both objective signals and other signals. To collect sample signals, I recorded various events in or near the parking. I placed the microphone in one place and left it there for both sample recordings and software version tests. The microphone was mounted outside a second-floor window, right above an apartment roadway with parking spaces across the roadway. The software collects Alarm (false alarms) data which went into the design process described in "CURRENT VERSION."

OBJECTIVE SIGNALS:

GreatGrey-52/Detect_Recip_Saw: Detect Reciprocating Saw (github.com)

For objective signals, I recorded a neighbor using a popular brand, battery-powered reciprocating saw cutting a piece of steel exhaust pipe. I also recorded a second popular brand, corded reciprocating saw (same strokes per minute as battery-powered) cutting the same type of pipe.

To vary samples, my neighbor and I set the cutting fixture at different distances from the microphone, ranging from 20-90 feet. At my neighbor's suggestion, we even set the cutting fixture under my truck to reproduce the signal as best as possible. This suggestion was important because the objective signal is different when the saw is under the vehicle.

In the Software Design section, Figures 1 through 3 illustrate the software tests comparing Other Signals to Objective Signals.

OTHER SIGNALS:

The other signals came from intentional recordings of non-sawing events and [false] alarm wav files during version tests. The intentional recordings included car traffic, trash trucks, lawn mowers, and weed trimmers. The false alarm signals included paint crews moving 3-story ladders, cars with powerful stereo systems, cars with high-performance exhaust systems, cars driving through puddles, water sprinklers, and even movers walking on an aluminum loading ramp.

CURRENT VERSION:

The software versions evolved from a cyclic process. As I tested versions, the system collected alarm event data in the form of wav files. Even though the events were confirmed as "False Alarms," the data went into designing and testing the next version. The cyclic process included the following steps:

- Sample objective signals under similar conditions
- Design an approach
- Test live (inject objective signal events)
- Gather alarm data
- Analyze circumstances
- Adjust the plan

False alarms still occur with this version. The saw signal is a complex "noise" with many variables. The variables include distance to the microphone, make/model of saw, type of blade, number of teeth per inch, material of the exhaust system, charge on the battery, and others. Alternate attempts to differentiate the saw signal from a parking lot full of noises are in the section "ALTERNATE CLASSIFICATION APPROACHES ATTEMPTED."

It seems I am at the point where Velardo would say, "This needs to be a Deep Learning project." During his Deep Learning series, Velardo even says Deep Learning is "ideal for 'complex' problems." The section "Next Steps" lists the general plan for future versions. I am open to feedback to improve this in any way.

GreatGrey-52/Detect_Recip_Saw: Detect Reciprocating Saw (github.com)

HARDWARE DESIGN:

The hardware for this project consisted of:

- Raspberry Pi 3B with a real-time clock
- High-Sensitive Weatherproof Preamp Microphone
- USB Video Capture Card (audio interface only)
- USB Computer Speaker
- Mouse, Keyboard, and Monitor

SOFTWARE DESIGN:

The software program consists of two primary functions: *main* and *proc_data* with some support functions. The *main* incorporates multiprocessing and queues to reduce the listen-to-process ratio as much as possible. Even with multiprocessing, there is about 10% overhead for each record cycle (0.4-0.5 seconds of process overhead for each 5.0-second recording).

The *main* records five-second signal samples, queues the sample for the next call to proc_data, checks for an alarm on the previous sample, and starts over.

- The *main* records a 5-second audio stream from the mic. The *main* design started with Joshua Hrisko's article "Recording Audio on the Raspberry Pi with Python and a USB Microphone" at https://makersportal.com/blog/2018/8/23/recording-audio-on-the-raspberry-pi-with-python-and-a-usb-microphone. Librosa apparently does not work directly with a live recording. I followed mvniemi's (github) recommendation to adapt using Librosa's utility "buffer_to_float". The "buffer_to_float" conversion prevents the need to write each sample to storage and then read the file for processing.
- The recording steps are in the *main* so that when *proc_data* reports an Alarm on the previous signal, that signal may be written to a wav file.
- Many recorded samples had an initial "pop" at the beginning of the recording. For now, the program slices the first 0.01 seconds off the front of the signal array. If there is a better microphone or setup, I am all ears.

During each cycle, the *proc_data* function *gets* the previous signal data from the queue and performs tests. Velardo's lesson, "Deep Learning 11 - Preprocessing Audio" was a starting point for the proc_ data design. Velardo's other lessons and Librosa's feature descriptions helped explore the environment further.

A couple of notes about the tests in *proc_data.*

- All tests and criteria are based on a 5-second sample
- Table 1 describes the Software Tests and Figures 1-3 illustrate the test-by-test differences between Other Signals and the Objective Signal
- Several of the tests (such as Spectral Flatness, Amplitude Envelope, etc.) go beyond extracting a level of the measure (such as an average). The range of change or the number of changes in the 5-second sample, along with the measure level, distinguished the Objective Signal from many

other signals in the environment. For example, a trash truck driving by the microphone with Power Take-off switched on, generates a complex noise. Still, the flatness measure (tonality) varied very little between individual values. The Objective Signal had a much wider variation between values

- Several of the tests employed windows over the sample waveform to observe trends in average or maximum measure level

The tests in *proc_data* are described in "Table 1: Software Tests" which describes Test #, Data Domain (time or frequency), and Test Description.

Table 1: Software Tests

Test #	Data Domain	Test
1.	Time	**Average Amplitude:** **Measure Definition:** Amplitude is the peak point of a wave **Test Extracts:** Average Amplitude of waveform **Test Description:** The test keeps sample signals with an average amplitude between 0.0028 and 0.01 (above ambient noise and below a trash truck parked below the microphone).
2.	Time	**Spike Test:** **Measure Definition:** A spike is a rapid change in waveform amplitude, both above and below the x-axis. Whereas a traditional envelope has Attack, Decay, Sustain, and Release phases, the spike has very short Attack and Decay phases. **Test Extracts:** **M**ax difference (down and up) between signal peaks. **Test Description:** The Spike Test rules out waveform peak-to-peak amplitude changes that are outside the range -0.035 to 0.035.
3.	Time	**Is Moving:** **Measure Definition:** The Is Moving test divides the sample into windows and looks for steady changes in a waveform's maximum root mean square amplitude through the sample. Per Wolfram Mathworld, root mean square "is the square root of the mean of the values." **Test Extracts:** Moving Object test extracts differences in maximum route mean square amplitude (by window) over a waveform. The test then counts the number of sign changes in max amplitude direction (+/-) across the windows. **Test Description:** Moving Object rules out signals that are steadily growing (no change in max amplitude direction), fading (no change in max amplitude direction), or growing then fading (1 to 2 changes in max amplitude direction). An objective signal varies more than 2 times in max amplitude direction over a five-second sample. The Librosa website had some great starter code on Root Mean Square to explore this area. The maximum Root Mean Square (per window) of the signal reduced the influence of small spikes on the analysis of overall amplitude direction and major direction changes (+/-). In Stackoverflow, janneb's elegant solution to check for sign changes reduced the original number of lines of code.
4.	Time	**Spectral Flatness:** **Measure Definition:** Librosa's Spectral Flatness measures the tonality of a waveform and the tonality coefficient ranges from 0 (tone) to 1 (white noise-like). **Test Extracts:** Average tonality coefficient and the number of changes in tonality coefficient greater than 0.001

Test #	Data Domain	Test
		Test Description: The Spectral Flatness test keeps within a certain average tonality coefficient range (0.0052 to 0.01) and counts greater than 150 over a five-second sample. The Librosa website had some great starter code for exploring spectral flatness. Again, janneb's solution made the sign change counts much more elegant.
5.	Time	**Amplitude Envelope:** **Measure Definition:** Velardo's amplitude envelop "follows the [max] amplitude of the waveform." **Test Extracts:** Average envelope level and the number of amplitude changes greater than 0.001 over a five-second sample. **Test Description:** The Amplitude Envelope test keeps samples with an average envelope level between 0.005 and 0.06 plus a change count between 1,300 and 1,470. Velardo's lesson "Extracting the amplitude envelope feature from scratch in Python" was very helpful to explore this area.
6.	Frequency	**Spectral Bandwidth:** **Measure Definition:** Librosa refers to the Spectral Centroid when describing Spectral Bandwidth. The centroid defines where the radiated energy is centered. The Spectral Bandwidth feature allows one to compute the pth order from the Spectral Centroid. In this case, the value is 0 order. **Test Extracts:** Spectral Bandwidth minimum and maximum plus the count of the number of changes in amplitude direction (+/-) over a five-second sample. **Test Description:** The test keeps signals with an average Spectral Bandwidth between 12,000 to 15000 Hz and a sign change count between 450 and 500 within a five-second sample.
7.	Frequency	**Spectral Rolloff:** **Measure Definition:** Librosa defines the Spectral Rolloff as the "center frequency for a spectrogram" based on the energy of the spectrum. While exploring the data within this feature, many "false alarms" had continuous activity above an 80% level (0.8 in a normalized scale). The Objective Signals had a very low level of activity above an 80%. **Test Extracts:** Extracts the average change in Spectral Rolloff frequency level and the number of changes greater than 0.8 (normalized 0.0 – 1.0). **Test Description:** The objective signal appeared to change within 165 to 695 Hz on average. After normalizing the Spectral Rolloff array, the objective signal appeared to have only 1 to 20 changes in the 0.8 and above (other signals appeared to have 50 to 200 changes above 0.8 level). The red line in Figure 3 Test 7, illustrates the 0.8 cutoff and the difference between Other and Objective signals

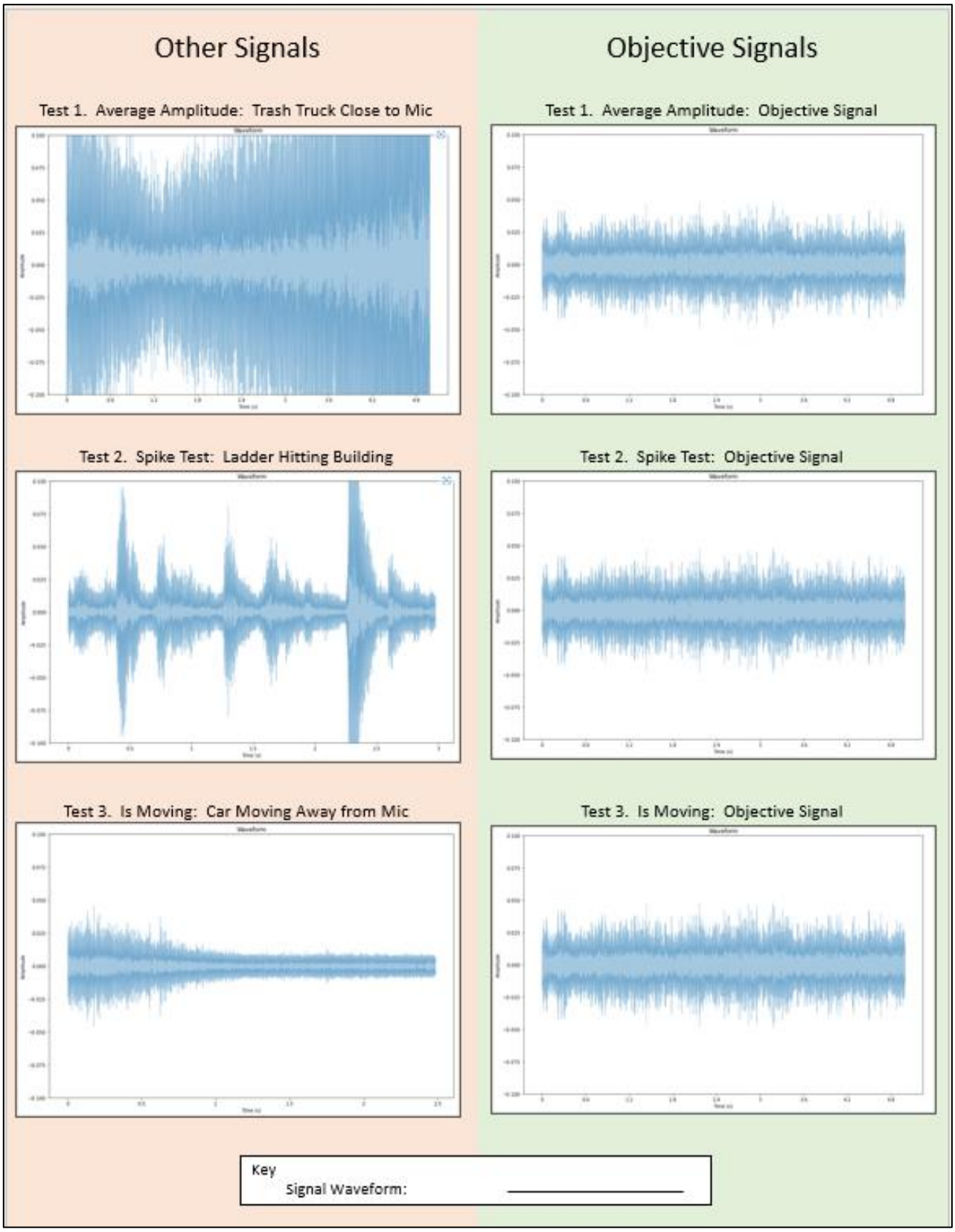

Figure 1: Software Tests with Other and Objective Signals

[GreatGrey-52/Detect_Recip_Saw: Detect Reciprocating Saw (github.com)](https://github.com/GreatGrey-52/Detect_Recip_Saw)

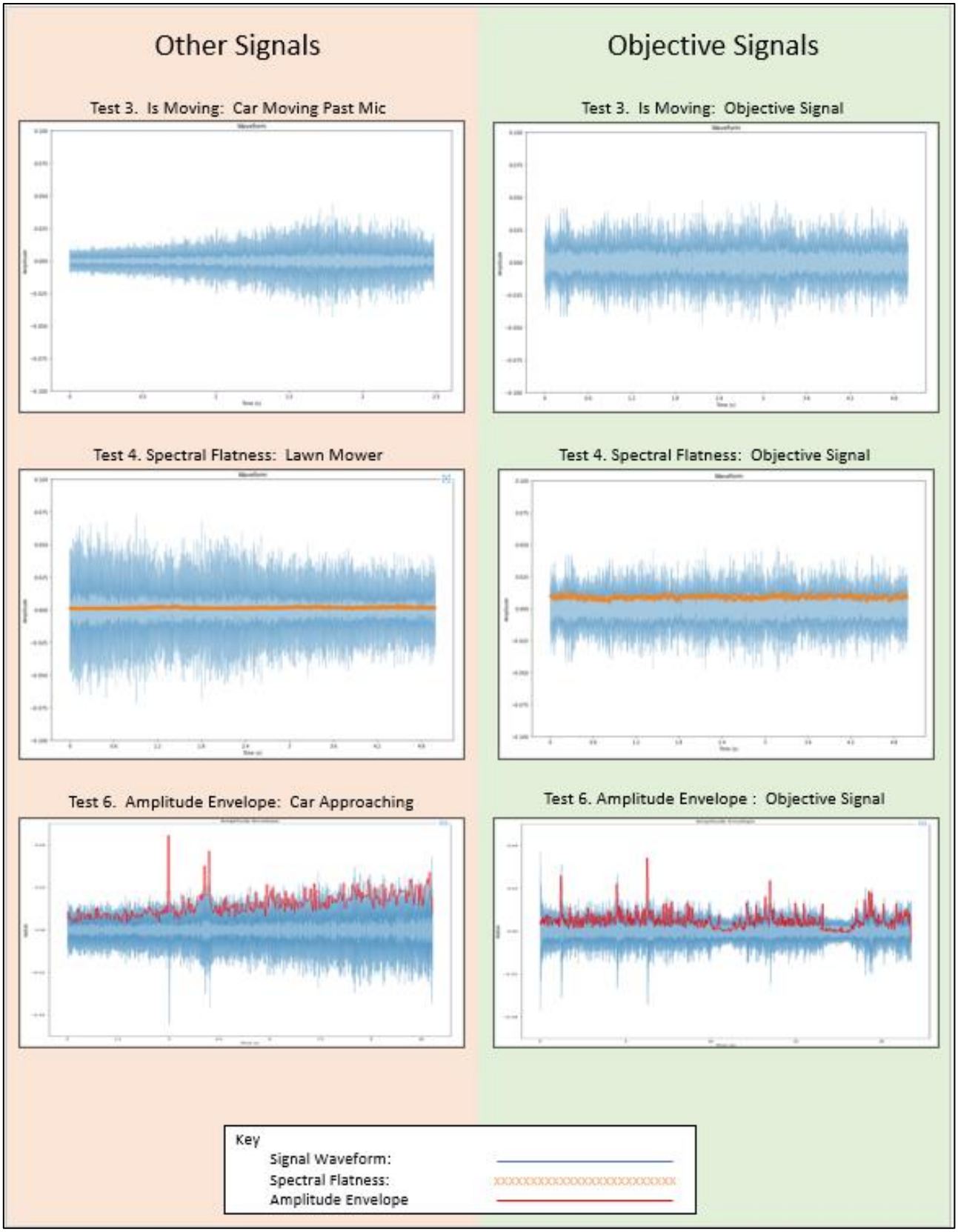

Figure 2: Software Tests (cont.) with Other and Objective Signals

GreatGrey-52/Detect_Recip_Saw: Detect Reciprocating Saw (github.com)

Figure 3: Software Tests (cont.) with Other and Objective Signals

[GreatGrey-52/Detect_Recip_Saw: Detect Reciprocating Saw (github.com)]

The software also included three support functions:

1. *ignoreStderr* suppresses Standard Output (audio) messages which were unrelated to the recording channels and detracted from the output. This function came directly from matthais post on StackOverflow, https://stackoverflow.com/questions/36956083/how-can-the-terminal-output-of-executables-run-by-python-functions-be-silenced-i#36958122
2. *write_alarm_wav* provides main a way to write a wav file after each alarm. The alarm wav allows for post "Alarm" analysis and collection of false alarm data for developing future versions
3. *sound_alarm* provides main a way to send a wav alarm signal to the speaker and repeat the sound until a keyboard interrupt

ALTERNATE CLASSIFICATION APPROACHES ATTEMPTED:

The project tested several alternate classification approaches with the limitations noted.

- Non-linear regression: During design, I attempted to use non-linear regression a polynomial expression on a Spectrum Plot. The Objective Signal had pronounced magnitude levels around 1100, 1500, and 1700 Hz. Limitation: The magnitude levels, across the frequency range, varied too much between samples to discriminate between Other signals.
- Detect frequencies above 5000 hertz. The microphone could detect frequencies greater than 5000 hertz at about 20 feet. Limitation: As the cutting fixture moved farther away from the mic, the higher frequencies became too low in amplitude to process against the background noise.
- Isolating frequencies of interest. On spectrum plots, the objective signal had pronounced magnitude readings around four to six frequencies. Limitation: Both the frequencies and magnitudes were irregular across objective signal samples. This irregularity included signals at the same range and signals at different ranges.
- Searching for apparent harmonics. After analyzing frequencies for close-in recordings (approximately 10ft), the reciprocating saw appeared to produce harmonics. Limitation: Again, as the cutting fixture moved farther away from the mic, the higher frequencies became too low in amplitude to process against the background noise.
- Harmonic Percussive Signal Separation (HPSS): The Librosa HPSS feature produced some interesting graphs which illustrated distinctions between saw noise and false alarms.
- Spectrograms and Mel Spectrograms: Velardo's Spectrograms and Mel Spectrograms had some interesting graphs which illustrated some distinctions between saw noise and false alarms.

NEXT STEPS:

The next steps include:

- Collecting more objective and false alarm samples
- Building a Deep Learning approach based on Velardo's and GianlucaPaolocci's work