LING 446 Fundamentals for Speech Signal Processing and Analysis

Spring 2022 Final Project

Instructor: *Yan Tang*
Teaching Assistant: *Yinglun Sun, Shuju Shi*
The Department of Linguistics, UIUC

April 26, 2022

1 Project summary

This is the final project for LING 446 Fundamentals for Speech Signal Processing and Analysis. It aims to combine some important topics covered in this course and apply what you have learned to solve problems in a simulated situation. Therefore, it is scenario-based and consists of several tasks. Each task may or may not depend on other tasks' output(s). The programming language that you can use in this project is limited to **Python**. Apart from specific built-in libraries and those provided by the instructors (see below), the use of any third-party libraries or code in the submission without approval is strictly restricted and will significantly reduce your grade. The weight distribution of the tasks is detailed in Sec. 4 and Sec. 6.

Performance on this project makes up **15%** of your final grade in this course. The project is due at **23:59:59 Saturday, May 14, 2022**. The late work and penalty-free late policies stipulated in the syllabus of this course will not be applied to this project. Therefore, any form of late submission will lead to a loss of the entire 15% from the final grade of this course.

The instructors' office hours remain in the weeks commencing May 01 and 08 to provide the necessary support. Read this document carefully; it is your responsibility to understand and implement all the requirements of the tasks. The instructors are not obliged to respond to or answer any questions whose answers have already been clearly stated in this document. This assignment is designed to *assess* your ability of applying your knowledge acquired in this course. Therefore, the instructors will only answer questions for clarification on this document, and for possible hints; the instructors will not be responsible for checking or confirming the correctness of your solutions and outputs before the project is due.

2 Contents in the project repository

In your project repository, the contents are organised as follows:

```
/ (the repository root)
├── LING446_FP22.pdf (the current file)
├── lib
│   ├── DSP_Tools.py
│   └── Audio.py
└── resources
    └── gated
        ├── hvd_001.wav
        ├── ...
        └── hvd_099.wav
```

```
    example_colocated.wav
    example_separated.wav
    ssn.wav
  signals
    IR
      BRIR_n.wav
      BRIR_s.wav
    utterance
      hvd_001.wav
      ...
      hvd_100.wav
  FP1_spSpkr.py
  FP2_mkNoise.py
  FP3_mkSti_SNR.py
  FP4_mkSti_Binaural.py
  FP5_RUN.py
```

Besides this document – `LING446_FP22.pdf`, all the resources provided include:

- `lib`: a directory containing LING446_DSP Python modules you need.
- `resources`: a directory containing one folder – `gated` with 50 WAV files inside – as the outputs from Sec. 4.1 and the reference output (`ssn.wav`) from Sec. 4.2. These files are provided to aid you in completing the subsequent tasks which depend on these files, and you are not able to produce them by yourself. Also read Sec. 4.5. There are also two sample outputs, `example_colocated.wav` and `example_separated.wav`, from Sec. 4.3 and Sec. 4.4 respectively.
- `signals`: a directory where all material signals are stored. The directory `utterance` contains the raw WAV files for 100 speech utterances that are going to be processed in Sec. 4.1. The directory `IR` keeps the two binaural room impulse response files for speech and noise signals required in Sec. 4.4.
- `FP1_spSpkr.py`: a template for the task in Sec. 4.1.
- `FP2_mkNoise.py`: a template for the task in Sec. 4.2.
- `FP3_mkSti_SNR.py`: a template for the task in Sec. 4.3.
- `FP4_mkSti_Binaural.py`: a template for the task in Sec. 4.4.

- `FP5_RUN.py`: a template for the task in Sec. 4.5.

In the entire project, you can only import the files in the `lib` directory, as well as *numpy*, *scipy*, *matplotlib* and Python stock libraries. Unless specifically clarified, all the ready-made methods are available in `lib/DSP_tools` and `lib/Audio`.

3 Project scenario

Conducting a perceptual listening experiment for speech and hearing research involves several steps. The early-stage preparations are critical because the quality of the work consequently determines the accuracy of the outputs from the experiment. Preparation work usually includes corpus recording, transcription and annotation, corpus analysing, stimulus generation and so on, which usually can be rather tedious and onerous if the work has to be done manually. In this project, you are going to help with some of the preparation work by automating the process using computer programs.

This project mainly focuses on early-stage preprocessing of recordings, generating noise maskers and creating experiment stimuli, which can be used in an experiment for speech perception in noise and reverberation. You will be working on the following raw materials:

- WAV files for 100 speech utterances (`signals/utterance`)[1] produced by two male speakers, Speaker A and B, 50 utterances each. One of the metafiles is missing so that files associated with different speakers cannot be separated. Besides, due to some hardware issues during recording, the WAV files cannot be further processed by algorithms such as endpoint detector and F0 tracker until they are somehow cleaned up.

4 Tasks

4.1 Separating WAV files by speaker: `FP1_spSpkr.py` (30%)

To separate the WAV files by speaker, fundamental frequency (F0) is a useful cue to distinguish the voices, especially when the speakers' F0s fall in different frequency ranges. After empirically measuring some of the WAV files of the two speakers, we learned that the F0s of the two voices are approximately 120 Hz (Speaker A) and 150 Hz (Speaker B). In theory, we can readily separate the WAV files into

[1]This is the relative path in the project folder

different voices by calculating the mean F0 of each recording. However, some low-frequency noise was introduced to Speaker A's recordings, due to the recording hardware malfunctioning. The low-frequency noise has led to an estimation error of 30-40 Hz above Speaker A's true F0, overlapping with Speaker B's F0 range. As a consequence, the F0 detectors we implemented based on autocorrelation have shown estimations far from being reliable. If you wish to see how the algorithm breaks down, you can run `DSP_Tools.getPitchContour()` through the WAV files under `signals/utterance` and observe the estimated F0s. You may also compare the results to what Praat[2] suggests – you should see that Praat is prone to the same mistake.

Furthermore, some of the WAV files in `signals/utterance` are preceded and tailed by silences of random durations. Those silences can affect F0 tracking and subsequent analyses; hence they must be removed from the signals. Usually, the onset and offset of the speech part can be precisely located using the endpoint detector based on energy and zero-crossing rate. Because of the low-frequency noise introduced to the recordings, the endpoint detector we programmed in lab13 has failed to process those WAV files directly. One example is shown in Fig. 1.

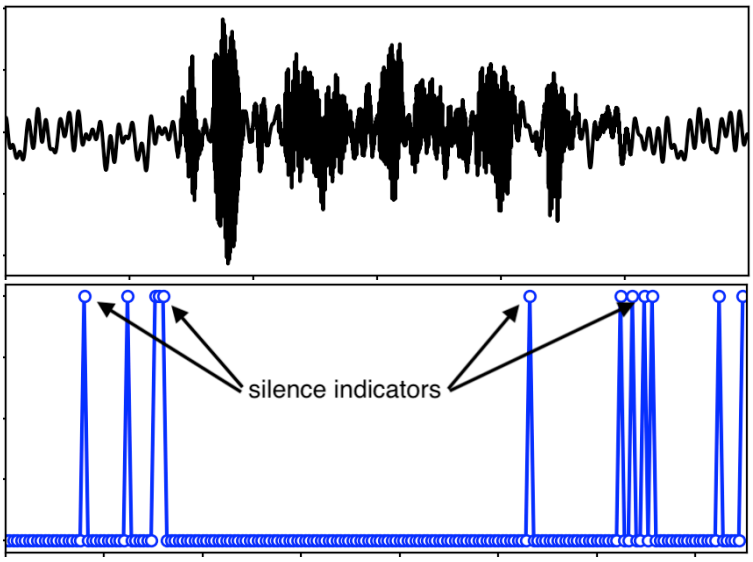

Figure 1: Raw waveform before preprocessing (upper) and the silence indicators `findEndpoint()` (lower)

Therefore, in order to fix this issue, some preprocessing is required on the raw recordings prior to performing endpoint detection and F0 estimation. After some

[2]https://www.fon.hum.uva.nl/praat/

analyses, it turns out that the noise introduced by the hardware only contains frequencies below 30 Hz. In this region, there is not much important information we need to keep in the speech signal, hence it can be lost if necessary. The solution for this issue has become clear: the performance of the endpoint detector and F0 estimator should improve if we can design a filter that can effectively reduce the noise floor in the signal.

Write a Python script, `FP1_spSpkr.py`, to process all the 100 WAV files. First, remove the silences before and after each WAV file using the ready-made endpoint detector, then calculate the mean F0 of the WAV file using the F0 estimator. Finally, save the processed WAV files belonging to Speaker A to a new directory. The requirements for this task are as follows:

- Design a Finite Impulse Response filter to preprocess all the WAV files in the `signals/utterance` directory. For good performance, the cutoff frequency could be set to 15-20 Hz lower or higher (depending on the type of the filter you choose) than the boundary of the region where the noise falls into. Once your filter is designed, it would be useful to examine the frequency response of the filter to make sure it will serve the purpose. An example of the filtered signal and the silence indicators suggested by `findEndpoint()` is provided in Fig. 2

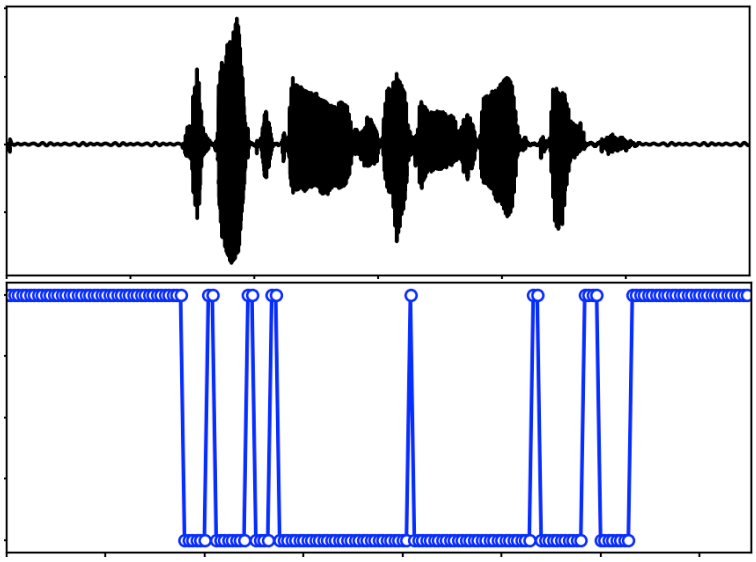

Figure 2: Filtered waveform (upper) and the silence indicators suggested by `findEndpoint()` (lower)

- Perform endpoint detection on the filtered signal, using `DSP_Tools.findEndpoint()` with a window size of 20 milliseconds and the default settings for other parameters.

- Use the silence indicators found by `findEndpoint()` to locate the onset and the offset of the signal. Extract the speech part between the onset and the offset. Supply the speech part to `DSP_Tools.getPitchContour()` for processing with the same windows used above; all other parameters in `DSP_Tools.getPitchContour()` are in default. Compute the mean F0 of the WAV file from the pitch contour as in HW06. Note that the mean F0 must be calculated from voiced segments, in which a valid F0 can be estimated. To separate WAV files by Speaker A from Speaker B, a threshold for comparison can be set. For example, if a WAV file with a mean F0 lower than 140 Hz, this WAV file is highly likely by Speaker A, or by Speaker B otherwise. Save all the WAV files of **Speaker A** (with preceding and tailing silences being removed) in a directory – `gated` – with the original file name and sampling frequency. Your code should first check the target directory's existence before creating it. If it exists, remove the directory along with its contents, then re-create it.

- All the new signals saved in the `gated` directory have a root-mean-square (RMS) of 0.01

- If you examine the waveforms in the `resources/gated` directory, you might notice that even among the reference outputs, not all the signals were precisely processed (i.e. the onset or offset of a signal was not accurately located) with an error rate of 2.0%, namely the instructor's algorithm failed in 1 out of the 50 waveforms. Therefore, your algorithm will be allowed to make mistakes without losing points if the error rate is under 16%.

4.2 Making speech-shaped noise: `FP2_mkNoise.py` (25%)

The experiment is designed to investigate listeners' speech perception in speech-shaped noise (SSN) – a temporally-stationary masker. To generate SSN, white noise is filtered using the coefficients of the long-term spectral envelope of the speech corpus, which are estimated using linear predictive coding (LPC). Consequently, the long-term average spectrum of SSN matches that of the corpus.

Write a Python script, `FP2_mkNoise.py`, to generate a one-minute long SSN, and save it as a WAV file. The requirements for this task are as follows:

- Define and implement a method `makeSSN()` to generate a SSN signal. makeSSN() takes four arguments:

- **DIR_sig**: the corpus directory
- **nb_sent** [default: 10]: the number of sentences from the corpus that will be used to estimate the spectrum using LPC
- **duration** [default: 30]: the duration of the output SSN signal in seconds
- **tarRMS** [default: 0.1]: the RMS of the output SSN signal

To implement, *randomly* draw **nb_sent** unique sentences from the corpus directory (the `gated` folder from Sec. 4.1). Concatenate all the sentences into one signal with each of the sentences being RMS-normalised to **tarRMS**. Segment the long signal without overlap between any consecutive windows, using a window size of three times the sampling frequency of the wav files.

Calculate LPC coefficients on all the segments. For each segment, the coefficients are estimated by 100^{th}-order LPC, meaning there will be 101 coefficients returned by `DSP_Tools.LPC()`. The final 101 LPC coefficients are the *average* across those acquired from the individual segments.

Generate a white noise signal using `numpy.random.uniform()` as in HW05. The white noise signal should be **duration**-second long.

Filter the white noise using the LPC coefficients calculated above. Note that the LPC coefficients are the 'a's of the filter. The 'b's of the filter is simply 1, since there is only one input term - the current input.

`makeSSN()` returns the output of the filter with an RMS of **tarRMS**, and the sampling frequency of the SSN signal (i.e. that of the sentences used for LPC estimation)

- Use makeSSN() to generate a 60-second long SSN from all the 50 sentences in the `gated` folder. The SSN signal has an RMS of 0.05. Save the signal as "`ssn.wav`" under a directory `noise`. Your code should first check the existence of the `noise` directory before creating it. If it exists, remove the directory along with its contents, then re-create it.
- Produce a graph with two subplots in a column. The upper plot shows the first 10-second waveform of the SSN signal. Limit the y axis to [-0.3, 0.3]. The x-axis should be time in seconds. The lower plot shows the magnitude spectrum *in decibels* generated from the first 100 milliseconds of the SSN file. The x-axis is frequency in Hz.

- Use `save2PDF()` provided at the top of this template to save the plot to "`etc/T-F.pdf`". `save2PDF()` takes the file name and the figure object as the inputs. Your code should first check the existence of the `etc` directory before creating it. If it exists, remove the directory along with its contents, then re-create it. Note that since the graph will be saved to the target folder, in your final submission calling `matplotlib.pyplot.show()` should be avoided, which halts the execution of the script.

4.3 Generating stimuli with target speech-to-noise ratio: `FP3_mkSti_SNR.py` (20%)

Once the noise WAV files are ready, the stimuli for the experiment can be created from the given speech materials ("`gated/*.wav`") and the noise file. In this experiment, there are two main conditions. The first one is to test the listener's perception when the speech source and the SSN source are colocated in front of the listener. The second one is to investigate the listener's perception when the speech source and the SSN source are spatially separated.

For a speech-in-noise test, listeners normally listen to speech sentences in the presence of background noise. The levels of both the speech and noise signals are strictly controlled in order to elicit listener responses at different noise levels, with the relative level between the speech and noise signals being particularly important. This relative level can be measured as *speech-to-noise ratio* (SNR) in decibels (dB). Mathematically, SNR can be expressed as,

$$SNR = 10\log_{10}\left(\frac{\sum_{t=0}^{T-1} s^2(t)}{\sum_{t=0}^{T-1} n^2(t)}\right) \tag{1}$$

where s and n are the speech and noise signals, respectively. T denotes the total number of the sample points.

In this task, write a Python script, `FP3_mkSti_SNR.py`, to make 50 speech-noise pairs as the stimuli for the first condition in the experiment. While speech WAV files are given, the noise signal must be *randomly* extracted from "`ssn.wav`" created in Sec. 4.2 for each of the 50 speech WAV files. In each speech-noise pair, the duration of the noise signal must match that of its speech counterpart. One way to do this is to randomly decide a position in `ssn.wav` as the starting index for the noise signal first. Since the length of the speech signal, T, is known, then the position where the noise signal should end in `ssn.wav` can be calculated. The noise signal with a matching T can be subsequently acquired using array/list slicing. It is however worth noting that the randomly-determined position in `ssn.wav` has to be no greater than $(L_{ssn} - T)$, where L_{ssn} is the length of `ssn.wav`, otherwise the noise

signal extracted from that position may *not* be long enough to match the speech signal.

For each speech-noise pair, there should be a random starting index in `ssn.wav` for the noise signal as aforementioned. Under the `etc` directory, you are required to save all the indices as integers along with the file names of the speech signals to a file – `ssn_idx1.txt` – in which the file name and the corresponding index are separated by a single tab "\t" character. The entries in `ssn_idx1.txt` must be sorted in ascending order from "hvd_001.wav" to "hvd_099.wav". See Fig. 3 for example.

```
hvd_042.wav 175403
hvd_043.wav 727588
hvd_044.wav 548104
hvd_045.wav 228529
hvd_046.wav 464542
hvd_047.wav 561564
hvd_048.wav 238144
hvd_049.wav 145656
hvd_050.wav 22731
hvd_051.wav 408786
hvd_052.wav 240377
hvd_053.wav 248032
```

Figure 3: Snippet in an example of `ssn_idx1.txt`

Note that you should not check your outputs against the example, because all the numbers are generated stochastically. `ssn_idx1.txt` will be used during grading to regenerate the noise segments, in order to compare them to those supplied by you.

As per the requirement on the signal levels, all speech-noise pairs must have a target SNR, $SNR_{tar} = -3$ dB. The procedure of adjusting the SNR is similar to setting the theoretical sound pressure level of a signal to a target value. In practice, altering SNR can be achieved by either changing the speech level while maintaining the noise level, or the opposite. In this task, you must implement this by keeping the *speech level* unchanged while altering the noise level. This can be done by scaling the noise signal by a factor k,

$$SNR_{tar} = 10 \log_{10} \left(\frac{\sum_{t=0}^{T-1} s^2(t)}{\sum_{t=0}^{T-1} (k \cdot n)^2(t)} \right) \tag{2}$$

The method to compute SNR, `snr()`, is provided in `DSP_tools.py`. You are expected to define a function, `setSNR()`, at the beginning of `FP3_mkSti_SNR.py`,

and call your method when needed. Your `setSNR()` takes three arguments: (1) the speech signal, (2) the noise signal, and (3) the desired SNR; it then returns variables: (1) the level-adjusted noise signal, leading to the target SNR, and (2) the scaling factor k.

Once the noise signal is ready, save the speech-noise pairs as a stereo WAV file. To do this, the speech signal and the noise signal need to be combined as a $T \times 2$ 2-D matrix as the data taken by `WAVWriter`. In the saved WAV file, the speech signal should always be on the left channel, i.e. it takes the very first column in the 2-D matrix mentioned above. All the WAV files containing speech-noise pairs must be named using the names of their corresponding speech WAV files, and are saved under `stimuli/SSN_colocated`. Your code should first check the existence of the directory before creating it. If it exists, remove the directory along with its contents, then re-create it. If your signals are generated and saved correctly, they should sound similar to "`resources/example_colocated.wav`".

4.4 Generating stimuli for binaural listening: `FP4_mkSti_Binaural.py` (20%)

The procedure for making stimuli for the second condition in the experiment largely resembles the previous task, except that binaural signals instead of monaural signals need to be created. This involves no more than further convolving the original speech and noise signals with the binaural room impulse response (BRIR) before adjusting the SNR level.

Write a **Python** script, `FP4_mkSti_Binaural.py`, to make 50 binaural speech-plus-noise mixtures as the stimuli on the SSN signal ("`noise/ssn.wav`"). By following the procedure in Sec. 4.3, the commensurate noise signal for each speech signal must be cropped from `ssn.wav` in the same *random* manner. Save all the starting index in `ssn.wav` for each speech signal to a file – `ssn_idx2.txt` – under the `etc` directory as you have done for the first condition.

The BRIRs signals required are under the `signals/IR` directory: "`BRIR_s.wav`" and "`BRIR_n.wav`" are for convolving speech signal and noise signal respectively, as the files decide the spatial location of each source in the space. The two BRIR WAV files have two channels with the first channel for the left ear and the second channel for the right ear. To create a binaural signal, convolution needs to be performed twice – the same speech or noise signal is convolved with the left-ear BRIR and the right-ear BRIR separately. It is worth noting that the output of convolution must be kept in its full length, therefore you might need to alter the "mode" parameter of `scipy.signal.fftconvolve()`. The left-ear signal and the right-ear signal can then be organised in a $T \times 2$ 2-D matrix for both speech and noise signals.

Perform RMS normalisation on the binaural speech signal with a target value of 0.01, before adjusting the SNR between the binaural speech and the noise signal to the same level as for the first condition (i.e. -3 dB). Mix the binaural speech signal and the binaural noise signal by summing them up together; make sure the channels in the two signals match correctly during summation. Finally, save the binaural mixtures to the `stimuli/SSN_separated` directory. The naming convention follows the first condition in Sec. 4.3. Your code should check the existence of the directory first before creating it. If it exists, remove the directory along with its contents, then re-create it. If your signals are generated and saved correctly, they should sound similar to "`resources/example_separated.wav`", in which the speech source is perceived as being close to your left ear while the noise source is close to your right ear, and that the signal in general has a long reverberation tail dying out gradually.

4.5 Running all the scripts in a batch: `FP5_RUN.py` (5%)

If you have successfully completed all the tasks above, each script should be independently executable provided that the required dependencies are available. In this task, write the final script, `FP5_RUN.py`, to perform all the processing in a batch. `FP5_RUN.py` must include executions of the following scripts:

- `FP1_spSpkr.py` (Sec. 4.1)
- `FP2_mkNoise.py` (Sec. 4.2)
- `FP3_mkSti_SNR.py` (Sec. 4.3)
- `FP4_mkSti_Binaural.py` (Sec. 4.4)

All the extra directories required by any other scripts, including "`gated`", '`etc`", '`noise`" and "`stimuli`", must be created automatically by those scripts. During executing `FP5_RUN.py`, it should print out ample prompt messages, indicating the progress of the processing. Fig. 4 shows the folder structure in the repository directory before and after invoking `FP5_RUN.py`.

A part of the expected outputs, all contents in `gated` from running `FP1_spSpkr.py`, and `ssn.wav` from running `FP2_mkNoise.py` are provided in the `resources` directory, in order to maintain the coherence of this project. This means that even if you are not able to complete the task in Sec. 4.1, you can move on to the subsequent tasks. Likewise, if you are having trouble generating `ssn.wav` in Sec. 4.2, you can skip it and continue to the remaining tasks. Similarly, instead of calling the non-functional scripts in `FP5_RUN.py`, you need to come up with some code to create the missing folders and copy the corresponding contents into the appropriate locations

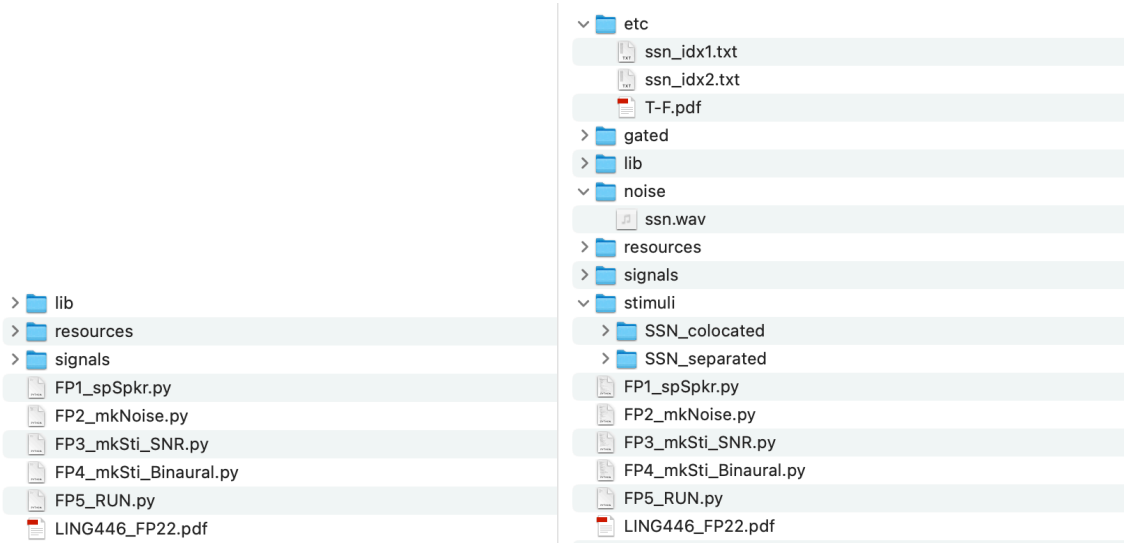

Figure 4: Contents in the project folder before and after running `FP5_RUN.py`

to keep the workflow going. However, your `FP5_RUN.py` must print out messages to STDOUT when the execution of any of the above scripts is skipped. By doing so, you will still receive points.

5 Submission

The Github Classroom is set to shut down the repositories for this project *one second* after the due time **23:59:59 Saturday, May 14, 2022**. This means any submission attempts after that time will be rejected. Therefore, do make sure you push back all your work before the deadline; there will be no extension allowed in any circumstances.

Besides what was originally in your repository while checking out, you must push back all the directories and files generated by executing your scripts upon submission. See the graph on the right in Fig. 4 for details. The instructors may or may not run your code again while grading. Therefore, any missing files from your submission will be assumed to be due to the scripts responsible for producing the files not being functional.

6 Grading

A summary of weight distributions of all the parts in the project:

- **30%**: `FP1_spSpkr.py`
- **25%**: `FP2_mkNoise.py`
- **20%**: `FP3_mkSti_SNR.py`
- **20%**: `FP4_mkSti_Binaural.py`
- **5%**: `FP5_RUN.py`