Real-time Audio Separation of Human Voices

EECS 452 – Digital Signal Processing Design Lab
University of Michigan, Ann Arbor

Tainon Chen
tainonc@umich.edu

Zhirui He
hezhirui@umich.edu

Nathan Hellstedt
nhellste@umich.edu

Haotian Qiao
qhaotian@umich.edu

Erik Sangeorzan
ejsang@umich.edu

Yuxin Zhong
yuxinz@umich.edu

Table of Contents

Introduction

Context

Our project aims to solve the problem of real-time audio source separation in the context of human conversations, which is a generally unsolved problem in the field of digital signal processing and broadly referred to as the "cocktail party problem" [1]. In a cocktail party, guests at each table are having their own conversations. It is not difficult for a human to focus on a single conversation, such as the one happening at their current table, owing to a combination of human speech processing capabilities and attention control. The ease with which humans can separate audio in a noisy environment is not found in computers — once the conversations are mixed together, it is a significant challenge for a computer to separate the mixed audio into its constituent parts. The goal of our project is to reconstruct the individual voices comprising a conversation in real-time.

Prior Art

There are two existing methods for similar problems which our group was drawn to: source separation and adaptive filtering optimized with machine learning.

Method 1: Blind Source Separation.
The first approach is Blind Source Separation (BSS) [2], used to describe the type of problem that involves separating a signal composed of a linear mix of other signals. Similar to the Fourier transform, which separates signals by frequency, BSS aims to separate signals by independence.

This can be achieved by using matrix methods to transform the input signal into a vector space which maximizes their separation. Independent Component Analysis (ICA) is one such method. Key to ICA are the assumptions that the underlying signals are independent, and are linearly mixed, which can be represented by matrix multiplication between the input signal matrix and a mixing matrix, to obtain the output signals. From here, we attempt to find an inverse mixing matrix which, once applied to the output signals, yields a maximally independent set of signals, presumably the input ones.

ICA draws upon two primary fields in signals processing. The first is in how exactly the separation between audio signals should be represented, and can include topics such as Gaussianity or log-likelihood of the signals. These fall under probabilistic methods. The second lies in how to maximize the chosen measure of separation. Gradient ascent is traditionally used, while modern approaches have found success in fixed point iteration. These overlap with matrix and numerical methods.

Method 2: Adaptive Filtering with Machine Learning.
The second method we considered is using adaptive filters with machine learning. Adaptive filters are filters in which the transfer function is adjusted in real-time by solving an optimization problem [3]. The optimization problem is defined by a cost function, which can be adjusted to suit the needs of the adaptive filter. Adaptive filters see use in current settings with variable background noise, where their adaptiveness yields better results than non-adaptive filters.
One example of an adaptive filter setup involves two input channels: one for the mixed signal (consisting of target audio and background noise), and one for the noise to be separated out

(primarily background noise) [4]. To maximize the distinction of the two above inputs, the mixed signal should attempt to maximize its target audio component, which can be accomplished by placing it as close to the source as possible. Likewise, the background noise signal should attempt to maximize background noise, but not to the extent that its captured audio is significantly different from the background noise of the mixed audio.

Adaptive filters as a standalone are already very relevant for real-time signal processing. However, by applying them to this project, we would be applying them to filtering human voices from other human voices. This is difficult to accomplish since human voices are more irregular than other types of background noise, such as from a car. Additionally, there may not always be a highly distinguishable frequency difference. Therefore, and via lack of team experience on this subject, machine learning of this variety was not utilized in our project.

Method 3: Adaptive Filtering with Neural Networks.
Implementing neural networks with adaptive filters [5] has been explored in the past, but there appears to be limited literature in its use in real-world audio applications. When they do apply this approach to audio processing, existing literature seems limited to successful separation of human voices from music, not human voices from other human voices.

The neural network would also require a large input dataset to optimize, prior to deploying it to the Raspberry Pi (RPi). We attempted to generate our own datasets, by recording the audio from multiple devices (laptop, phone) playing in the same room, in hopes that this would also enable us to easily identify the “correct” separated signal, for easily analyzing the error in our adaptive filter. A drawback to this method is that the trained filter would only be as good as the data it was trained on. It’s possible that this filter would only perform well on mixed signals that have individual components similar to the test data, which will limit the applicability of our filter on a larger range of inputs, making a large dataset all the more important. This approach was ultimately not employed in our project due to these constraints and our limited experience with neural networks.

Our Approach

The final product takes in mixed audio data as initial input, and plays back separated audio as final output. ICA was employed to perform the audio separation. Additionally, a combination of hardware and software were required to complete this project.

Source Separation via ICA

FastICA.
FastICA is the core of the project. The main theory of it is the central limit theorem, which indicates that the sum of several independent signals is more 'Gaussian' than each independent signal. Therefore, to separate the sources, we need to apply transformations to make each separated signal as non-Gaussian as possible. There are several ways to measure the Gaussianity of signals, such as kurtosis and negentropy. Since kurtosis is more sensitive to outliers in the signal, we decided to use the negentropy approach.

The input to our ICA function is N channels of mixed audio. The steps of ICA are as follows: First, centralization is used to make the average value of each input signal zero, to assist in obtaining the covariance matrix. Second, a whitening transformation is performed using eigendecomposition to make the covariance matrix of the N channel array diagonal. Finally, we iteratively update the vectors from the transformation matrix to maximize the negentropy of the separated signals. Once each iteration is done (the vector is not still determined), we perform Gram-Schmidt orthogonalization to ensure that this vector won't converge in the same direction as any other already-determined vectors. The resulting N vectors compose a transformation matrix that can be applied to the mixed audio input to reconstruct the individual channels.

Windowing and Splicing.
Our team made a few modifications to traditional source separation to make the project more applicable for real-time separation. We window the signal repeatedly and compute the FastICA output for each slice of the signal, similar to the STFT. That way, FastICA can be performed without requiring the full input audio signal. This approach, however, brings up an additional issue — the order of the separated signals may be different for each window. Therefore, we developed and implemented a method to splice all the windows pertaining to the same signal back together, after source separation. This was performed by including an overlapping section between consecutive runs of FastICA. The overlapping sections in the beginning of the new FastICA output could then be matched to the corresponding end of the previous FastICA output, by identifying the component with the minimum difference. In this way, the FastICA output will be continuous for each signal.

Implementation: Hardware

The main components of our hardware system are: one Raspberry Pi, two Teensy boards with AudioShield, two CP2102 units (USB to TTL converter), two TRS to TRRS converters, two microphones (sampled at 48 kHz) with 3.5mm cables to audio jack (to transmit the 3.5mm cable to jump line) and one USB hub. The communication between AudioShield and Teensy is I2S protocol which uses DMA, and the communication between Teensy and Rpi is UART (8 data bits).

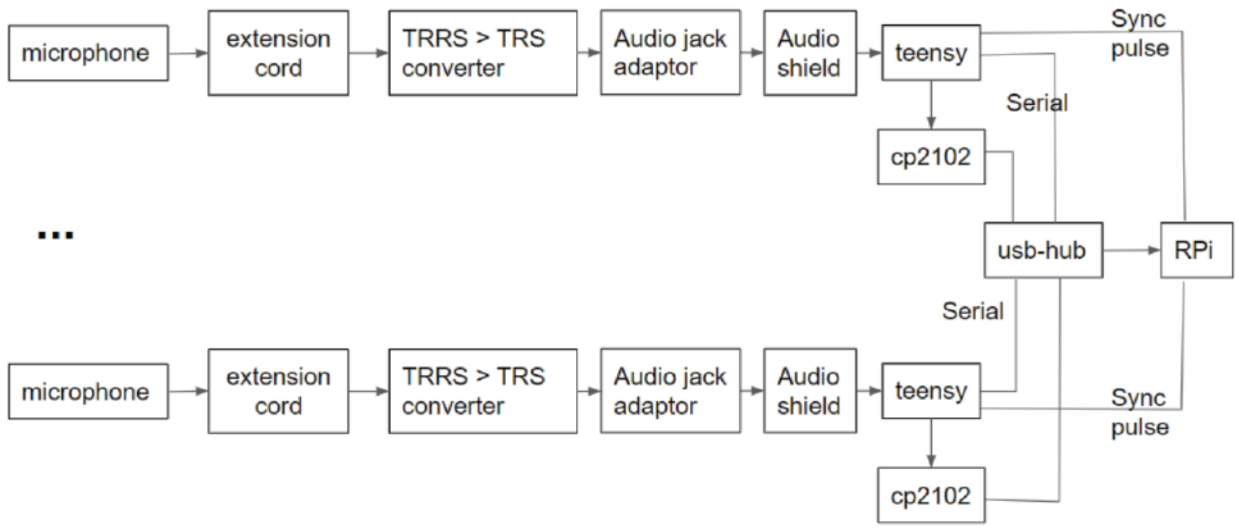

Figure 1: Diagram of our functional hardware system.

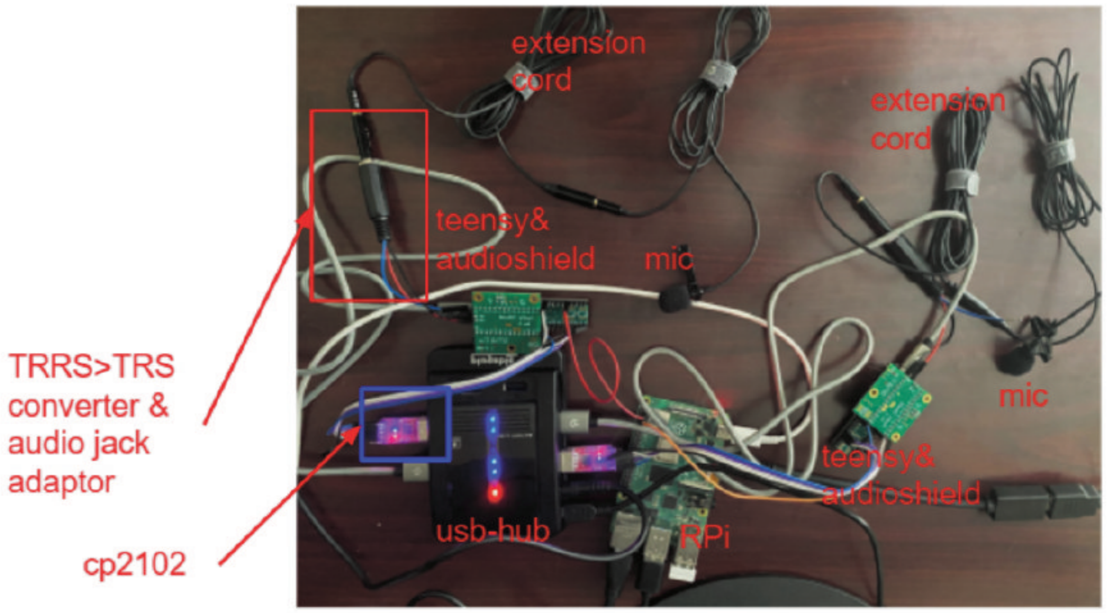

Figure 2: Picture of our functional hardware system.

Initial testing indicated that attempting to transmit the sampled data at 48 kHz led to significant data loss. We determined that because the standard for a voice-frequency transmission channel is 8 kHz, we could safely send data at 12 kHz while maintaining all relevant information. The final norm of the signals is 12 kHz 16-bit integers, which requires a 6 kHz low pass filter to be applied to the microphone data. To prevent the overrun of the buffer, we need at least 12*16=192 kbps. Thus, we set the baud rate as 500 kbps.

Implementation: Software

There were three primary parts to the software, all of which were run on the Raspberry Pi. The first was UART processing, which converted UART data to integer audio data. The second was ICA, which separated sources. The third was a user interface to control playback and plot the waveforms reconstructed by ICA.

UART Processing.
Using multicore processing, one core for each teensy, this runs in parallel to continuously read in UART data from the USB hub, and convert the byte stream to 16 bit integer representation.

Audio Separation.
This section of the code continuously reads in integer audio samples, from the previous step. Once a sufficient number have been read in, it uses the aforementioned FastICA with windowing and splicing to perform source separation. Additionally, the code would periodically plot and display the waveform of the audio signal, as shown in Figure 3, to help the user identify which output channel corresponds to which audio signal, for playback purposes.

User Interface.
The PyQt user interface we developed, as shown in Figure 3, plays back the audio separated in the previous step. It contains basic controls for playback and channel selection. There is a short delay of a few seconds when playing back audio, since input audio must first be processed via FastICA.

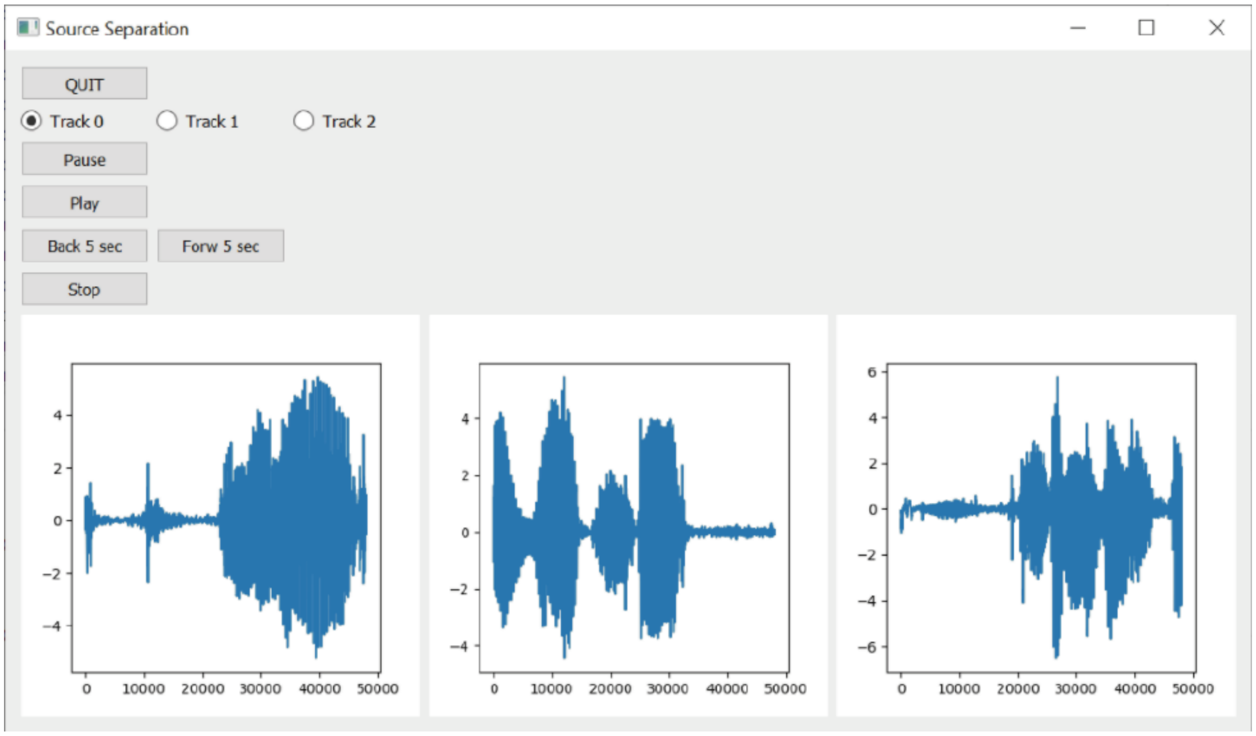

Figure 3: User interface

Technical Issues

Serial Data Transmission

As mentioned, we used 16-bit integers, transmitted at 12 kHz. In order to send 16-bit integers with UART, we need to send the upper 8 bits and the lower 8 bits separately. Additionally, for improving the performance of reading from serial ports on the Raspberry Pi side, we decided to use the "readline( )" function, which means we append the '\n' at the end of each package. So, to avoid the irregular length of packages, we replace the value '10' with '12', and it has been found through testing that negligible loss will occur in the transmission. And at the Raspberry Pi side, we will reconstruct each pair of 8 bit integers to the original 16 bit integer.

For multi-Teensy transmission, we rely on the USB hub to deal with the multiplexing issue. The USB to TTL converter will transfer the output from the hardware serial port to USB protocol, and they will be connected to the USB hub. On the Raspberry Pi side, it is just like we have more USB ports, so we don't need to worry about the transmission order of several Teensys.

Multiple Teensy Issues

To synchronize recording to start at the same time, we use several GPIO pins on the Raspberry Pi and set them to high level when starting, and the Teensys will monitor their GPIO; when they receive a high level, the transmission starts.

Also, receiving from several Teensys was another issue. We have tried both using receiving sequentially and the multiprocessing method. However, both methods led to significant data loss during transmission. We adopted the multiprocessing method to assign two different CPU cores to receive data in parallel, and it performed fairly well.

Code Organization and Data Flow

An earlier iteration of our code contained all of UART processing, FastICA, and playback in the same Python script, and used Python's threading library to run them simultaneously. This resulted in issues with playback, whenever the code attempted to perform FastICA while playback was also occurring. When this happened, an ALSA buffer underrun error message would appear, followed by distortions sounding like static interrupting the playback. This was resolved by treating the Python scripts as different processes, rather than threads.

There was a related issue regarding the transfer of data between software steps: From the UART processing code to ICA to playback. We opted for simply saving and loading .npy files for each step, due to the high speed of .npy reads and writes, and the compatibility of .npy files with our data.

ICA Performance & Speed

When designing our ICA algorithm, we anticipated that the ICA algorithm would need to compute output in less than half the time of the window length. Specifically, if ICA is run on blocks of length 10000 sampled at 12KHz, then the ICA algorithm needs to compute in less than 0.4sec. Quick computation frees us computing power for the other intensive processes being run concurrently and helps to prevent ALSA buffer underrun in the playback function. This requirement was initially difficult to meet, however, one significant optimization allowed us to meet our time-performance goals. This optimization was saving the mixing matrix computed in the previous ICA iteration and then passing it in to the next iteration of ICA. Using the information from a previously optimized mixing matrix allowed for quicker convergence relative to starting each iteration with a random mixing matrix. This ended up reducing ICA computation time by roughly 50%.

Results

Simulations with Linear Mixtures

Figure 4 and Table 1 demonstrate the results from a linear mixture of a conversation with three male speakers. Figure 4 confirms that the waveform of the output channels match those of the input channels, and the low MSE values further indicate that our ICA algorithm is effective in reconstructing the output channels. Note that part of our testing includes listening and comparing the input channels and the ICA output. When ICA was performed on a linear mixture, there was no noticeable difference in the playback of the input and ICA output channels.

An important aspect of our project was implementing ICA in real-time, which required us to window the audio signal, perform ICA on the windowed audio block and then splice the current output with the output computed from previous windows. Figure 5 shows that our method of windowing and splicing is equally effective in reconstructing the signals relative to computing ICA on the entire signal. This confirmed that we could move ahead on our real-time goals, and allowed us to implement the audio playback feature in the user interface.

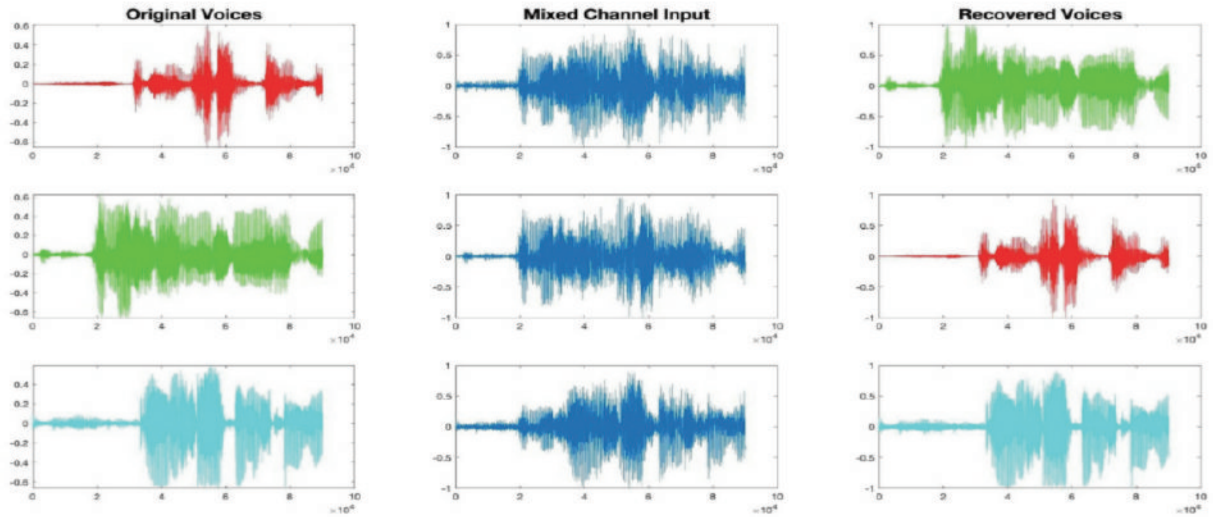

Figure 4: The left column shows the individual inputs of the conversation, the middle column shows the mixed conversation used as ICA input and the right column shows the individual channels reconstructed from ICA.

Voices from Figure 3	Normalized Mean Squared Error
Red	1.4537e-6
Green	0.2050
Cyan	3.8551e-5

Table 1: Mean Squared Error from the ICA reconstruction of the conversation shown in Figure 4.

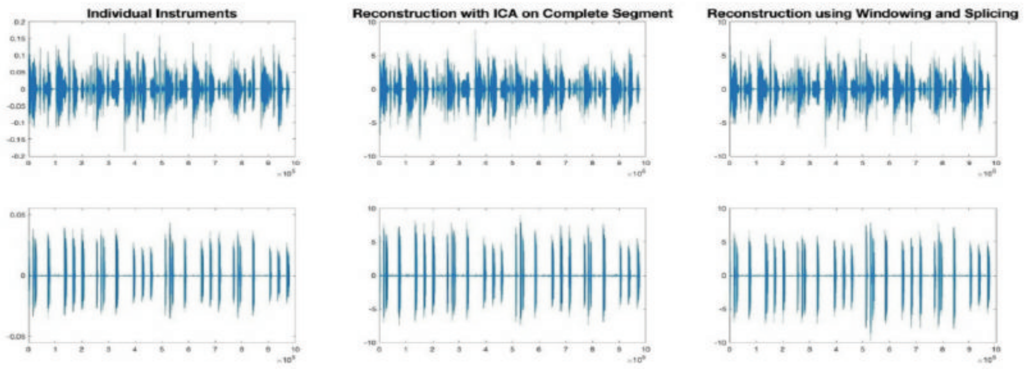

Figure 5: The left column shows the two of the original individual instruments, the middle channel shows the reconstructed channels doing ICA on the entire audio segment and the right channel shows the reconstructed channels using windowing and splicing.

Testing with Microphone Input

In the end, we failed to separate the nonlinear sources from the sound recorded by the microphones. This is due to the fact that the mixture of audio recorded from the microphones introduced nonlinearities, such as echo and harmonic distortion. Our ICA algorithm struggles to separate these mixtures because they violate the assumption that the input is both linear and independent. Potential solutions to address nonlinear elements are discussed in the Future Work section.

Delay in Real-time ICA and Playback

Although such code would, in theory, play back audio in real-time, we encountered issues with immediate playback. Choosing to play back audio “greedily”, whenever audio was available, would result in many calls to sounddevice.play(). This resulted in the audio stuttering, due to the brief delay between each call to sounddevice.play(). Although these interruptions couldn’t be completely eliminated, they could be lessened by reducing the number of times sounddevice.play() is called. The simplest solution was to introduce a user-tunable parameter which would only play back audio when the number of available samples was greater than the number of samples as indicated by this parameter. This results in a larger initial delay, while populating audio data for the first call to sounddevice.play(), but results in fewer future calls to sounddevice.play(). thus improving audio quality during playback.

Future Work

More Microphones

Receiving data from more microphones still needs to be tested. The Raspberry Pi only has 4 CPU cores and one core is for the main function of ICA, which means only 3 cores are left for receiving microphone data, which may be a bottleneck for the receiving. Also, further work is needed to optimize the receiving code more to make receiving not rely quite so much on the multiprocessing method, which has led to data loss.

Accomodation of Broader Frequency Spectrum

We used 500 kb/s baud rate in our project because we encountered significant data loss in UART transmission at higher baud rates. Theoretically, a 500 kb/s baud rate could support transmission of audio sampled to 30 KHz. Further testing is required to determine if our system can support these higher transmission rates and whether increasing the transmitted frequency spectrum will lead to better separation results.

Nonlinear Source Separation

Given more time to work on this project, our team believes nonlinear source separation would be achievable — and possibly while used alongside ICA methods. Nonlinearities introduced to our recordings include echo and time delay due to spatial differences between the microphone. If we’re able to remove time-variant disturbances, such as echo, through acoustic echo cancellation, then post nonlinear separation methods, like the one proposed in [6] would provide a promising platform from which we could continue to increase the robustness of our source separation algorithm. The advantages of handling nonlinear features in such a way is that it wouldn’t require the use of neural networks or machine learning solutions that require a priori knowledge to train the system. However, as mentioned in the discussion on prior art, many state of the art source separation solutions use neural networks to train and optimize the separation algorithm. Training a source separation algorithm to handle nonlinear feature spaces before deploying the algorithm to the Raspberry Pi is a useful and potentially powerful method that should be explored in greater depth. A TPU may also be employed to speed up computation on the Raspberry Pi.

References

[1] K. J. Woods and J. H. McDermott, “Schema learning for the cocktail party problem,” *Proceedings of the National Academy of Sciences*, vol. 115, no. 14, 2018.
[2] Xi-Ren Cao and Ruey-Wen Liu, "General approach to blind source separation," in *IEEE Transactions on Signal Processing*, vol. 44, no. 3, pp. 562-571, March 1996, doi: 10.1109/78.489029.
[3] T. Adali and P. J. Schreier, "Optimization and Estimation of Complex-Valued Signals: Theory and applications in filtering and blind source separation," in *IEEE Signal Processing Magazine*, vol. 31, no. 5, pp. 112-128, Sept. 2014, doi: 10.1109/MSP.2013.2287951.
[4] B. Widrow and S. Stearns, *Adaptive signal processing*. Prentice-Hall: Englewood Cliffs, 1985.
[5] Ying Tan, Jun Wang and J. M. Zurada, "Nonlinear blind source separation using a radial basis function network," in *IEEE Transactions on Neural Networks*, vol. 12, no. 1, pp. 124-134, Jan. 2001, doi: 10.1109/72.896801.
[6] C. Jutten and J. Karhunen, “Advances in blind source separation (BSS) and independent component analysis (ICA) for nonlinear mixtures,” *International Journal of Neural Systems*, vol. 14, no. 05, pp. 267–292, 2004.