Sound Classification and Localization by Acoustic Sensing on Raspberry Pi 4

Jaidon Lybbert, University of Washington

January 24, 2024

Abstract

In this article a network of distributed acoustic sensing nodes is used to localize a sound signature using a time-difference-of-arrival (TDoA) algorithm. Accurate wall-clock times are obtained by GPS at each node, and a frame of acoustic data is recorded through a microphone on each node. Time differences are determined by correlation, and the resulting system of equations solved by the Chan-Ho algorithm to determine location. In a concurrent process, each node performs classification inference on a data frame using a pre-trained convolutional neural network (CNN). The resulting class identification is used to assign a process model to the tracked object for use in an Extended Kalman Filter (EKF) applied to the Chan-Ho localization result to make the final estimate of object position and trajectory.

1 Introduction

This document is organized as follows. In Section 2 4 5 6. I conclude in 7 by discussing the overall takeaways from my experiments and things I would like to try in the future.

2 Background

3 Kalman Filters

The Extended Kalman Filter (EKF) is a well-described algorithm, where discrete samples from one or more sensors is combined with an analytic model of a system to estimate system state. A block diagram of the Kalman Filter from *Kalman Filters for Beginners* by Phil Kim [?] is shown in Fig 1.

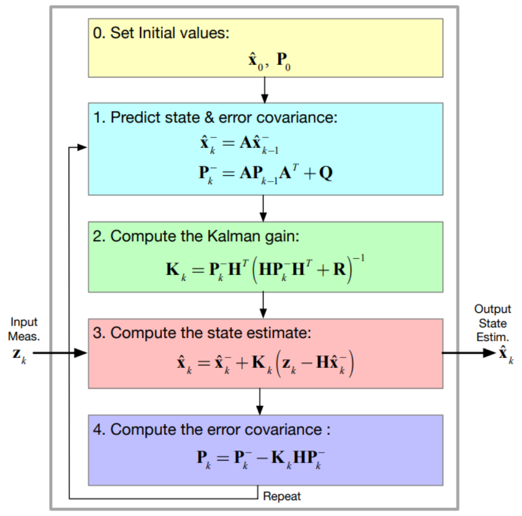

0. Set Initial values:
$\hat{\mathbf{x}}_0, \mathbf{P}_0$
1. Predict state & error covariance:
$\hat{\mathbf{x}}_k^- = \mathbf{A}\hat{\mathbf{x}}_{k-1}^-$
$\mathbf{P}_k^- = \mathbf{A}\mathbf{P}_{k-1}\mathbf{A}^T + \mathbf{Q}$
2. Compute the Kalman gain:
$\mathbf{K}_k = \mathbf{P}_k^-\mathbf{H}^T\left(\mathbf{H}\mathbf{P}_k^-\mathbf{H}^T + \mathbf{R}\right)^{-1}$
Input Meas.
$\mathbf{z}_k$
3. Compute the state estimate:
$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + \mathbf{K}_k\left(\mathbf{z}_k - \mathbf{H}\hat{\mathbf{x}}_k^-\right)$
Output State Estim.
$\hat{\mathbf{x}}_k$
4. Compute the error covariance :
$\mathbf{P}_k = \mathbf{P}_k^- - \mathbf{K}_k\mathbf{H}\mathbf{P}_k^-$
Repeat

Figure 1: Block diagram of the Kalman Filter algorithm. $\hat{\mathbf{x}}_0$ represents the initial state.

4 Methods

4.1 Environment Setup

For classification, I will be using TensorFlow Lite

4.2 TensorFlow Lite

TensorFlow Lite (TFLite) is a lightweight version of the TensorFlow runtime for loading machine learning models and running inference on input data.

TFLite runs on resource-constrained devices like the Raspberry Pi. I use TFLite for loading a pretrained model for classification of bird species by sound. The model of choice is taken from the BirdNET project, and contains 6,000 species. The input to the model is a 3 second long frame of audio sampled at 48kS/s, and the output is an 6k element array containing the class estimates. The model has a secondary input which takes in latitude, longitude, and the week of the year (1-52).

I first referenced the TFLite Python API documentation to attempt loading the model, and running it against a sample audio clip. I quickly found out that the high-level API and provided binaries for the **aarch64** architecture are not well supported by the TensorFlow developers. I shifted my strategy to setting up the BirdNET-Pi project to run on my device. I ran into build and dependency issues, since the developers of the BirdNET-Pi project support a previous release of the Raspberry Pi OS. Instead of re-installing a different OS, I used the source code of BirdNET-Pi as reference to write a minimally working script to get TFLite working on the Pi.

The BirdNET model contains functions that require a piece of software to be compiled in with the TFLite binary. This runtime is not available in the binary provided by the TFLite developers, so I used a version provided by an independent developer who compiles the runtime with this software for the Pi. The TFLite wheel was incompatible with the system installation of Python 3.11 included in the Raspberry Pi OS. I installed **pyenv** to install Python 3.9 and manage the two different versions. The versions of packages required by the TFLite wheel, such as **Numpy** were incompatible with the versions installed by **pip** by default, and changing the version manually broke dependency rules for other packages like **matplotlib**. I created a **requirements.txt** file for pip, and manually wrote the dependency rules to get all packages working.

I took a sample audio **MP3** of a known bird species from an online database, and loaded it into Python with **pydub**, and converted it to a **Numpy** array. I wrote a function to split the file into 3 second chunks. For each chunk, I put the array of samples into the first TFLite input tensor, and put the location and time data into the second input tensor. I timed how long it took for inference, and recorded the estimate result. I plotted the inference execution time over all frames of audio.

4.2.1 GPS

I used the Adafruit Ultimate GPS USB breakout board to get GPS data through **gpsd** [?]. **gpsd** is a Linux daemon which listens on port 2497 of the loopback address by default. When a connection request is recieved, it begins communicating with the GPS module over USB by reading data through the /dev/ttyUSB0 device file, and forwarding it to the TCP/IP port to service the request. By default, **gpsd** is managed through **systemd**. Adafruit recommends disabling this functionality through the **systemctl** interface, and running **gpsd** manually.

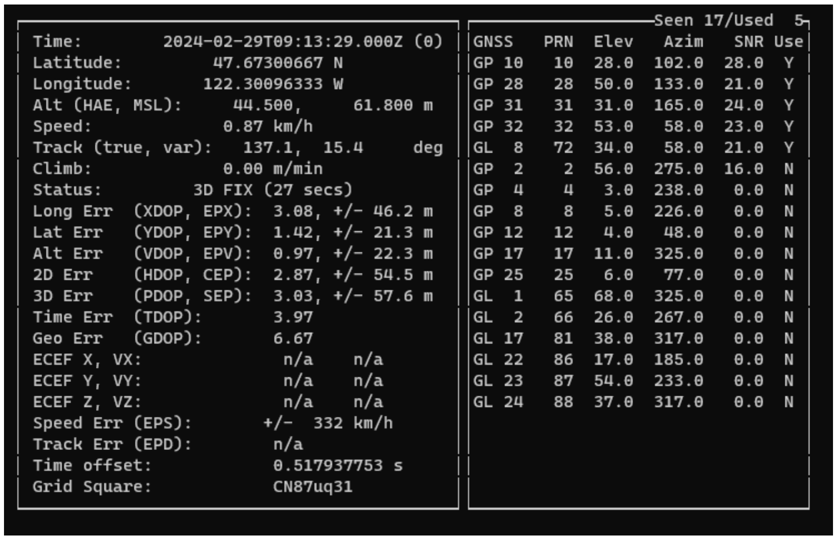
```
sudo apt install gspd gpsd-clients
sudo systemctl stop gpsd.socket
sudo systemctl disable gpsd.socket
sudo killall gpsd
sudo gpsd /dev/ttyUSB0 -F /var/run/gpsd.sock
```

To test the device I used the command **cgps -s**, which is a **gspd** client that communicates over the default port and prints out GPS status information to the console, as shown in Fig. 2. It took significant troubleshooting to get this to work properly. What I finally determined to work was editing the configuration of **gpsd** defined in the **/etc/default/gpsd** file, and rebooting.

```
DEVICES=""
START_DAEMON="true"
GPSD_OPTIONS="/dev/ttyUSB0␣-G"
USBAUTO="true"
GPSD_SOCKET="/var/run/gpsd.sock"
```

The other necessary step was physically moving my test setup from the basement to the main floor of my house, since I could not get a reliable connection to the GPS satellites from the basement.

Once I had a reliable connection, I wrote a Python script to collect about 1200 GPS readings of latitude and longitude, while leaving the system stationary, and wrote the readings to a **.csv** file. I wrote a script to convert degrees of latitude and longitude to meters, then plotted a histogram of the differences of each reading from the mean value to evaluate the spatial precision of the GPS reciever.



```
Time:         2024-02-29T09:13:29.000Z (0)    GNSS   PRN  Elev   Azim   SNR Use
Latitude:          47.67300667 N              GP 10   10  28.0  102.0  28.0  Y
Longitude:        122.30096333 W              GP 28   28  50.0  133.0  21.0  Y
Alt (HAE, MSL):      44.500,     61.800 m     GP 31   31  31.0  165.0  24.0  Y
Speed:               0.87 km/h                GP 32   32  53.0   58.0  23.0  Y
Track (true, var):   137.1,  15.4     deg     GL  8   72  34.0   58.0  21.0  Y
Climb:               0.00 m/min               GP  2    2  56.0  275.0  16.0  N
Status:           3D FIX (27 secs)            GP  4    4   3.0  238.0   0.0  N
Long Err  (XDOP, EPX):  3.08, +/- 46.2 m      GP  8    8   5.0  226.0   0.0  N
Lat Err   (YDOP, EPY):  1.42, +/- 21.3 m      GP 12   12   4.0   48.0   0.0  N
Alt Err   (VDOP, EPV):  0.97, +/- 22.3 m      GP 17   17  11.0  325.0   0.0  N
2D Err    (HDOP, CEP):  2.87, +/- 54.5 m      GP 25   25   6.0   77.0   0.0  N
3D Err    (PDOP, SEP):  3.03, +/- 57.6 m      GL  1   65  68.0  325.0   0.0  N
Time Err  (TDOP):       3.97                  GL  2   66  26.0  267.0   0.0  N
Geo Err   (GDOP):       6.67                  GL 17   81  38.0  317.0   0.0  N
ECEF X, VX:              n/a    n/a           GL 22   86  17.0  185.0   0.0  N
ECEF Y, VY:              n/a    n/a           GL 23   87  54.0  233.0   0.0  N
ECEF Z, VZ:              n/a    n/a           GL 24   88  37.0  317.0   0.0  N
Speed Err (EPS):       +/-  332 km/h
Track Err (EPD):        n/a
Time offset:            0.517937753 s
Grid Square:            CN87uq31
```
Seen 17/Used 5

Figure 2: Terminal display of GPS information from the **cgps -s** command.

4.3 Microphone

I use an generic adjustable-gain microphone module to sample the acoustic signals with the ESP8266 sensing node. For my breadboard prototype, I don't have an ADC capable of sampling at the 48kS/s rate required by the TFLite model. Given more time, I would implement the Analog Devices SSM2603 audio CODEC, which has programmable gain and a digital I2S interface. I power the microphone with 3.3V from an ELEGOO breadboard power supply connected to a wall outlet through a 9V wall wart. The microphone output is connected to a simple first-order low-pass antialiasing filter implemented with a resistor and capacitor on the breadboard with a cutoff frequency of 20kHz. The output of the filter is connected to an ADC on the ESP8266. To characterize the microphone noise, I tape over the transducer, then sample the ADC with the ESP8266, and send the samples over to the Raspberry Pi through the web socket interface. I record the samples to a **CSV** to plot the sample distribution.

4.4 ESP8266

Each acoustic sensing node is implemented as an Adafruit Feather HUZZAH ESP8266 MCU which connects to a common WiFi network and communicates with the Raspberry Pi through web sockets. For my prototype, the ESP8266 is connected to my home router. The EEPROM of the ESP8266 is flashed with the MicroPython runtime through a micro-USB cable connected to my laptop using the **esptool** CLI. Configuration of the ESP8266 can be done through the read-evaluate-print-loop (REPL) accessed through a serial port using PuTTY. MicroPython automatically executes the **boot.py** script in the ESP filesystem when the device is powered on, then executes **main.py**. In order to edit the scripts on the ESP8266, I use the **Ampy** command line tool created by Adafruit to modify the filesystem over the serial port. Ampy can be installed through **pip**, then commands can be issued to copy files over by specifying the serial port. I wrote a script **server.py** which is loaded onto the ESP. The script opens a web socket and listens for a connection from the Raspberry Pi, which . When the connection is made, the ESP begins sampling the ADC and stores the samples into a buffer. The buffer is sent over the connection to be recieved by the Raspberry Pi.

4.5 Simulator

The effectiveness of localization based on acoustic signals depends on factors including:

- Sensor noise
- Environment noise
- Sampling quantization
- Sensor and source geometry
- Sound speed
- Reverberation

In order to test my localization algorithm under variation of these parameters, I created a simple simulator in Python to place sensors and sources spatially in the environment, add noise to the signal, simulate the sound propogating from source to each listening sensor, and visualize the resulting hyperbolas of the TDE algorithm and the localization result. The simulator uses the **Pygame** library to graphically represent the environment.

4.6 Time-Difference Estimation (TDE)

To develop the TDE algorithm, I use the simulated time-shifted inputs, and apply simple correlation using the **scipy** Python library. The peak of the digital correlation result between signals recieved at two nodes is used to determine the number of samples the signal is delayed by at one node compared to the other. Based on the sample rate, the time delay is determined. I iterate through all audio frames and compute TDE for each simulated node-pair. The resulting TDEs will be used in the localization algorithm to determine the localization result. I profile the execution time of the TDE algorithm, and write the output to a **csv** file for plotting.

4.7 Localization Estimate

The set of TDEs define a system of hyperbolic equations. When there are exactly 3 TDEs, the equations are represented by a singular matrix, with one solution. With more than 3 TDEs, the matrix is overdetermined, and a least-squares method is necessary to make the final localization estimate. Least-squares is a complicated algorithm to optimize, but well studied, with implementations using matrix decomposition techniques. For this project, I use the **Numpy** Python library, which under the hood, calls a compiled system BLAS library written in C located at **/usr/lib/aarch64-linux-gnu/blas**, and the compiled LAPACK library written in FORTRAN 77 located at /usr/lib/aarch64-linux-gnu/lapack. My profiling tests in the characterization process showed that these implementations take advantage of the Advanced SIMD architecture of the Cortex-A72 CPU for doing tensor operations, and vastly outperformed my best attempts at matrix multiplication in Rust.

4.8 Fast Fourier Transform

The correlation algorithm I used is very basic, but falls short in the real-world where correlated noise like wind and reverberation exists in the signals. The majority of studied localization algorithms use a variation of the Generalized Cross-Correlation Phase Transform (GCC-PHAT) algorithm. Which transform the signal to the frequency domain using the Fast Fourier Transform. For this project, I didn't have time to implement a more complicated localization algorithm, but I did profile the FFT algorithm provided by the

Numpy and SciPy libraries to evaluate the CPU time it would require, since the FFT accounts for the majority of required FLOPs in these algorithms.

5 Results

6 Discussion

7 Conclusion

Note *All my code and project files for this and future reports, can be found on my GitHub repository [**?**].*