Big brain matrix eigenvalue lightspeed fourier transform for the great good solar light

A 4-bit waltz in frequency space

Pushkar Mohile (180260027) and Sankalp Gambhir (180260032)

Indian Institute of Technology, Bombay

Abstract. Identification of sounds has immense applications in the embedded systems space, ranging from simple detection of sounds to complete voice transcription. Being able to do this on low power devices is an area of active interest. We present an approach to this problem involving bypassing a complete Fourier Transform and approximating its results using a cross-correlation based approach pruning a tree of (preset) frequencies. Our method returns present frequencies with reasonable accuracy whilst maintaining the speed expected of such an embedded system.

1 Project Details

1.1 Motivation

Why try to make better Fourier Transforms? Fourier transforms are the most important tools used in data processing. The relative amplitudes of the signal at different frequencies give us a lot of characterising data of the signal, and changing the signal in frequency space has a lot of applications, like auto-tuning, removing noise etc.

$$\tilde{f}(\omega) = \int_{-\infty}^{\infty} e^{-i\omega t} f(t) dt \ . \tag{1}$$

Fourier is Overkill. Looking at the structure of the Fourier transform, it is easy to see that it is closely resembles the idea of a correlation, essentially extracting from the signals its *similarity* to a given frequency. These coefficients, gathered using all frequencies, can produce a one to one mapping to a function space in the frequency realm. However, the entire function is far too much information. For most tasks involving categorization and identification of sounds, fingerprinting is more than good enough. That is, considering the delta response of the system, its restriction to a tiny subset of points in frequency space.

As such, simply extracting the info for these frequencies alone is sufficient, and can be done whilst incorporating several statistical approximations. This is easily done with a correlation of the signal with a pure mode.

However, when frequencies are aggregated, the phase difference is absorbed into the coefficients of multiple frequencies, but in the correlation scenario, no such hope exists. This is where cross-correlation comes in.

$$(f * g)(t) \triangleq \int_{-\infty}^{+\infty} \overline{f(\tau)} \cdot g(t+\tau) d\tau \tag{2}$$

By considering the similarities of shifted signals, we can infer more precise information about their similarity in space, by completely disregarding the temporal dimension. This is immensely useful in matching an externally sourced wave of unknown phase with a synthetic mode of known or unknown phase to identify the wave.

In particular, the global maximum of the cross-correlation may be taken as the phase independent correlation of two signals.

$$corr(f, g) \triangleq \max((f * g)(t))_t \tag{3}$$

The main idea of this project is to extract some characteristic audio data from the signal without having to resort to memory and computationally expensive fourier transforms. The 2KiB SRAM of the Arduino is the biggest botleneck here for doing any processing. In order to do this processing, we came up with multiple optimizations.

1.2 Optimizations

Cross Correlations Instead of doing a FFT on the data, we will extract a few characteristic frequencies components by correlating the signal with a set of frequencies. The expression for crosscorrelation of 2 discrete signals x, y is given by

$$R_{xy}[k] = \sum_i x[i]y[i-k] \tag{4}$$

Along with some normalization. The harmonics form a linearly independent set and return zero-correlation when the product is integrated over several time periods, i.e.

$$\frac{1}{\pi}\int_0^{2\pi} sin(n_1 x)sin(n_2 x) = \delta_{nn'} \tag{5}$$

Where δ is the usual Kronecker delta. We can normalize the correlation with the auto-correlation at 0 to get a number between -1 and 1 that characterized the coefficients.

$$c_{xy} = \frac{R_{xy}[0]}{\sqrt{R_{xx}[0]R_{yy}[0]}} \tag{6}$$

However, the above expression does not account for the phase shift ϕ between the harmonics which reduces the correlation by a factor of $cos(\phi)$Therefore we modify check the signal with phase shifted test harmonics by modifying the expression to:

$$c_{xy} = \frac{max\{R_{xy}[k]\}}{\sqrt{R_{xx}[0]R_{yy}[0]}} \tag{7}$$

This ensures that the loss due to potential phase shift is avoided by getting an estimate of the phase. Further, these correlations avoid the complex multiplications required in FFT calculations while giving more characteristic data.

Memory Management We store the signal using an array of 4 bit numbers, i.e. numbers from -7 to 7. This has multiple benefits, such as being able to outright store more numbers, and more subtly, perform calculations in a smaller time period as the Arduino processor is based around an 8-bit bus. Instead of requiring multiple clock cycles to process 16 / 32 bit datatypes like `float` adn `int`, we can process the signals much faster.

Frequency Space Pruning Beginning with a Dirac comb in frequency space, we prune a lattice tree of the frequency power set via depth first binary search. This bypasses the computationally expensive cross correlation calculations for a lot of (wrong) frequencies.

1.3 Major blocks

The flow of control begins at the bottom left of Figure 1, with a mic providing audio input, which is captured by MATLAB (above) and converted to an array of integers. In ideal conditions and availability of modules, this would be done on the arduino, with little to no performance detriment. The time between serial inputs (based on the baud rate) is utilised to process the incoming data into the format used through the program, which is as in the Doobit storage system.

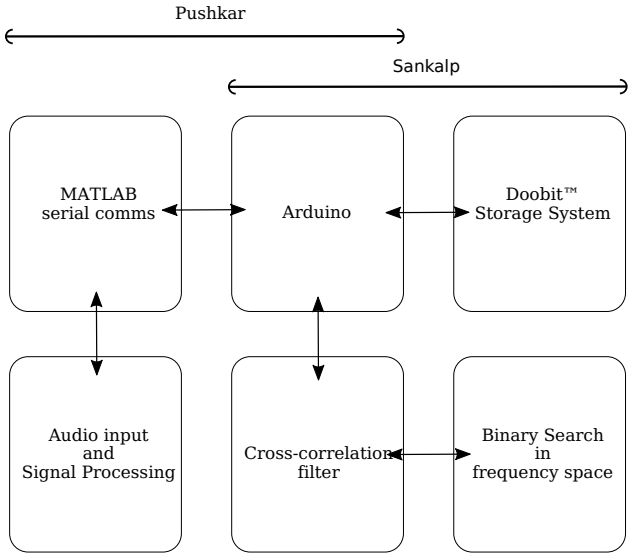

Fig. 1: Block diagram and work distribution.

After the signal has been completely read, a Dirac comb (in frequency space) is generated from a preset range of frequencies. Ideally, a Fourier transform would be utilized for this, but that method carries far too much resolution for just identifying certain frequencies. A limitation that still preserves a huge space of applications. Identification of a combination of frequencies (a trivial extension of the current algorithm, a simple check at the end, or even better if specific, a smaller starting set) could allow for identifying more complicated speech patterns and more processing on subsets of the input (would be stretching the limits of the arduino) could help in preliminary speech recognition, possibly as a low power first gate in an always-on speech recognition system. For example, a lower level circuit that listening for *"Alexa"* and activates a more power hungry and performant circuit for speech recognition.

To avoid processing all the frequencies, we process the entire comb at once, and if it passes a certain threshold, we infer that one or more of the frequencies in the comb is present

in the sample. Proceeding, the comb is split halfway in frequency space, processed depth first, to identify the frequencies present, rejecting the entire set at any point the global maxima of its cross correlation drops below a given threshold.

A possible optimization, abandoned in the interest of time, is to compute the cross correlation at only a few points, utilizing optimization techniques instead of brute forcing to find the maximum. The first thing to do is check whether the correlation function gives us an appropriate response with this sampling rate. In order to do this, we plot the correlation coefficient of a signal and a pure harmonic vs a change in frequency about the pure harmonic for N datapoints.

The code for plotting these graphs in MATLAB is given here:

– Code for correlation of 2 signals

```
1
2  function outp = corel(A,B)
3   outp = 0;
4   normA = 0;
5   normB = 0;
6   for i = 1:length(A)
7       outp = outp+(A(i)*B(i));
8       normA = normA+(A(i)^2);
9       normB = normB+ (B(i)^2);
10  end
11  outp = outp/sqrt(normA*normB);
12 end
```

– Code for getting graphs

```
1      function grp = coreldraw(f1,f2,f3,f4,phi4)
2     t = -0.5:0.005:0.5; %Change the step size to get different
      resolutions
3 signal = round(7/3*(sin(f1*t) + sin(f2*t) + sin(f3*t) )) ; %This
      ensures we have a number between \(\pm7\)
4 df = -200:0.1:200;
5     grp = zeros(length(df));
6
7 for i = 1:length(df)
8 signal2 = round(7*sin(((f4+df(i))*t) + phi4 ));
9 grp(i) = corel(signal,signal2);
10 end
11 plot(df,grp)
12 xlabel('df')
13 ylabel('Correlation Coefficient')
14 s = strcat('Graph of correlation of' ,int2str(f1),'Hz ',int2str(f2)
      ,' Hz' ,int2str(f3), ' Hz with ',int2str(f4),' Hz vs change in
      ',int2str(f4));
15 title(s)
16 end
```

If we first try correlating with 100 data points, we get Clearly, this does not provide a good enough degree of correlation as there are several significant analolous peaks. We would expect only 2 significant peaks correspoinding to 200 and 300 Hz. Next we try

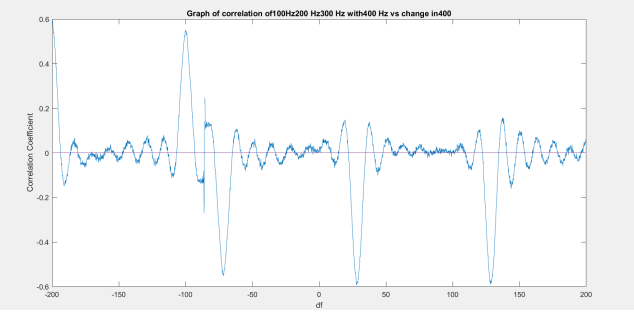

Fig. 2: Correlation with 100 point sampling

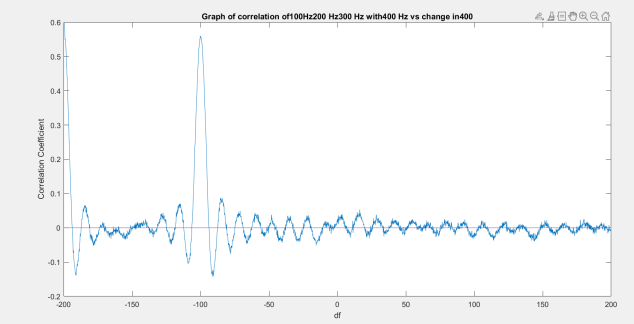

Fig. 3: Correlation with 200 point sampling

with 200 points. This gives us very clear peaks at the right frequencies. It appears that 200 points sampling is good enough. In order to check the resolution limit, we also try a signal where the frequencies are close: The 3 peaks expected are missed. This implies that we cannot separate frequencies very close to each other, and this is the price we have to pay for avoiding a FFT. From these graphs it is clear that a threshold greater than 0.1 will suffice for testing our audio signals.

1.4 Doobit™

Working on the Arduino with signals very quickly turned into a constant battle of resolution and storage, limited by its relatively tiny memory. After msot optimizations we went through, managing memory manually very closely and ensuring sequentialized execution contexts, we were teribly bottlenecked by the storage. To work around this physical limit, we manage each byte of the stored signal manually and instead of one, store two signal data points in every byte. This reduces our data resolution by way of being limited to 4 bits, but grants us unmatched robustness to noise by comparison, by doubling possible data processing capabilities. The system has been affectionately named Doobit.

```
1 // bit mask storage
2 struct doobit{
3     uint_fast8_t data;
4
5     doobit(int8_t x = 0, int8_t y = 0){ // handles all our casts too
```

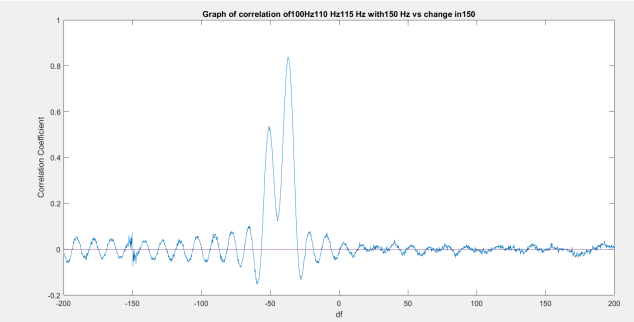

Fig. 4: Correlation with 200 point sampling for 5Hz difference

```
        this->storelow(x);
        this->storehigh(y);
    }

    void storelow(int8_t);
    void storehigh(int8_t);

    int8_t getlow();
    int8_t gethigh();

    int16_t operator*(doobit& b);
};
```

A `struct` provides us fast access with very little memory and performance overhead, something that only becomes more and more negligible as we increase our (now doubled!) data numbers.

Unpacking the code block, `data` is the actual storage, an unsigned 8-bit type, chosen this way to avoid accidental signed interpretation and any unwanted processing by the compiler. Reliance on unsigneds in a case such as this is common even within the compiler itself, where it would convert signeds to unsigneds before evalutation to avoid ambiguity. In particular, a two's complement could scramble the data beyond recognition quite quickly.

The `fast` part of `uint_fast8_t` indicates to the compiler that we are looking for a type that is atleast 8 bits, but is the fastest among those. On an Arduino Uno, of course, this is just the 8 bit unsigned integer, but on more exotic embedded systems this could end up being a 9 or 10 bit type, if not more. This notation allows for some compatibility between systems, though as is with embedded systems, one would try to create more efficient structures that exploit the architecture of those systems.

The storage of the signal is handled by the `storelow()` and `storehigh()` functions, which store data into the 4 least significant and 4 most significantbits of the storage `data` respectively. We look at one of the functions:

```
void doobit::storelow(int8_t x){
  this->data &= 0b11110000;  // clear for storage

  x += 7; // remove signed component
  assert((!(x & 0b11110000)) && "doobit range violation");

```

```
  this->data |= x;
}
```

This function takes in a value `x`, to be stored in the lower 4 bits of `data`, and preparing for it, clears the lower 4 bits via an `AND` operation with a bitmask `11110000`. Following this, a manual conversion is made to ensure `x` is unsigned. The operation moves `x`'s previous range, -7 to +7, to now 0 to 14, with 15 remaining generally unused, rushing to somehow occupy this position leads to little benefit and after much trial to squeeze extra storage out of the system, was abandoned.

The `assert` exists merely for testing purposes to ensure our data can indeed fit in 4 bits. Finally, having ensured `x` has only its lower bits populated, an `OR` operation with the storage inserts it in.

The `storehigh()` function works in a similar manner, albeit with a bit shift and an inverted bit mask to work on the 4 higher bits.

Now remains the issue of retrieving data from this storage, and is done as very much the inverse of how it is stored:

```
int8_t doobit::getlow(){
    int8_t x = (data & 0b00001111); // bitmask
    return (x-7); // reinsert sign
}
```

The *unrequired* data is removed via a bit mask, and would be moved rightwards via a bit shift in the case of `gethigh()`, and its signed nature is restored by shifting it back to the original range.

The conversion to unsigned is quite important to have to not manage the carry bits arising from a two's complement operation. The number -7 in C++ could ambiguously be coming from an `int` (internally `int32_t`) in which case the signed bit is the 32nd, while it could also be coming from an 8 or 16 bit type, making the signed bit unclear and its extraction slow and painful. Asking for a change of range was the fastest of the operations we tested, included several bitwise only operations.

The storage could be optimized for several arithmetic operations, but for our case in particular, for the correelation setups, we need only multiplication. As of now, this is done simply by retrieving the numbers individually and multiplying them pairwise. This is done as opposed to attempting bitwise operations as (1), multiplication operations are quite optimized on a circuit level in modern processors anyway, and more importantly (2), 4 bit being such a limited storage type, would cause an overflow for most possible multiplication operands.

```
int16_t operator*(doobit& b){
    auto highprod = this->gethigh() * b.gethigh();
    auto lowprod = this->getlow() * b.getlow();
    return (highprod + lowprod);
}
```

This definition is obviously not compatible with all arithmetic operations, but quite specific for our limited operations, kept this way to prevent overcomplication of the rather heavily utilized routine.

The full code for all the functions may as always be found in the Appendix.

1.5 Serial Communication and the Great Horse Race

The input of the system is handled using MATLAB. This is appealing for several reasons, (1), we can immedietly visualise and process the sound array that we get, making it easy to debu, and (2), we can use the inbuilt system microphone. MATLAB by default offers several different ways to get this audio data, the lowest resolution being a 1000Hz Sampling frequency and 8-bit sampling, using the `audiorecorder` object. Once we record the relevent sample, we extract an array with this data and after doing some basic filtering, to remove any high frequency noise that may affect the signal, we send the data to the Arduino via the Serial port.

The important part here is understanding how the serial communication with the Arduino works. The serial port can only send data character by character to the Arduino and waits for it to be processed. Therefore the processing of the integers is done as follows:

1. Collect multiple audio samples from the microphone. We take 2000 datapoints, assuming a periodic sample. The points are all `int8`.
2. Apply a filter to remove any peaks above ± 7 to ensure that the data doesn't overflow in the arduino. Then we apply a Moving average filter by taking 3 200-datapoint samples from the 2000 datapoints to remove any anamolous noise in the data.
3. In order to communicate this data, we convert the integer Array in MATLAB to strings using the `int2str` method so that it can be sent via the Serial port. The terminator that we use is the newline character '\n'.
4. We open the serial monitor and send the data character by character using the `fprintf` function in Matlab.
5. We check for available data in the Arduino using the `Serial.available()` function.
6. Then we read the data into an array in Arduino using the `Serial.parseInt()` function. We wait for 2 bytes to arrive and store them in a single byte on the arduino to work with our processing scheme.
7. When all the expected data (200 points) are read, stop recording and do the processing to find the matching frequencies.
8. Send the frequencies back to MATLAB to print using the Serial port.

This protocol can be seen in this block diagram

However, we ran into some issues here, where several values were not detected by the Arduino. We tried multiple approaches to solve this

1. We increased the Baud rate to 115200 to ensure that the transmission wasn't causing any loss of data.
2. We added a small delay of 30ms between subsequent transmissions to ensure that the Arduino had time to process the data.
3. The Serial protocol has a buffer that stores the incoming values before they are processed by the Arduino. In the Arduino Uno, this `SERIAL_RX_BUFFER` memory that stores the incoming data from the Serial Port is only 64 bytes. We changed the header files to make the buffer 128 bytes in HardwareSerial.h so that more characters could be stored without losing them in transmission.

In order to ensure that this method is working correctly we have several checks

1. We ran a simple program that turns the pin13 LED On and Off based on whether the character it received was 0 or -7.

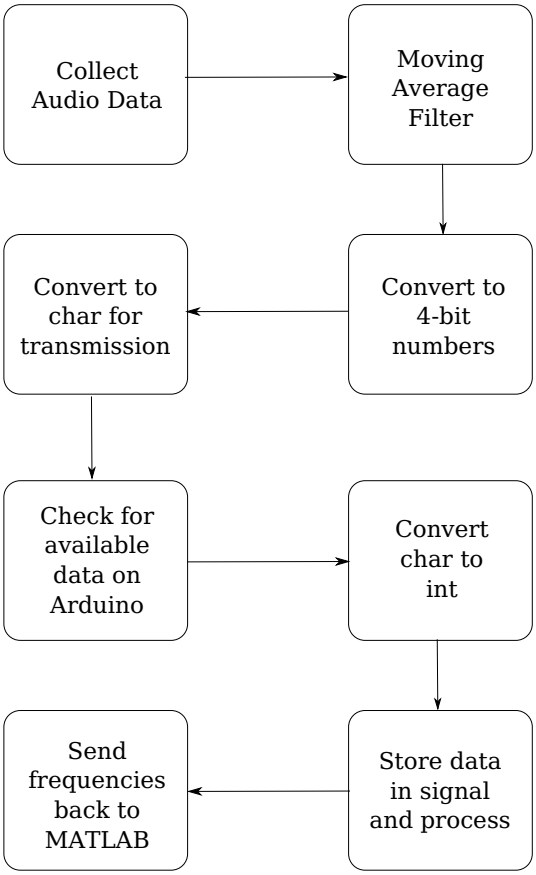

Fig. 5: Block diagram for communication protocol.

2. In our main code, pin 13 is HIGH when waiting for data and recording. When all the expected data has been found, it stops glowing.
3. We also plot the raw data that we get from the Mic to ensure that a reasonable waveform has been detected.

The code for testing the glowing of the LEDs is found here:

```
const int led = 13;
String text;
int k;
void setup() {
                //pinMode(led, OUTPUT);
                Serial.begin(9600);
                k = 0;
}
void loop() {
```

```
  if (Serial.available()){
                    text = Serial.readStringUntil('$');
                    k = text.toInt();
                    if(k == -7){
                        digitalWrite(led, 1);
                    }
                    if(k == 0){
                        digitalWrite(led, 0);
                    }
  }
}
```

The MATLAB code for controlling this unit is found here:

```
Arduino = serial("COM4","BaudRate",9600);
fopen(Arduino) %This opens a monitor
fprintf(Arduino,'%s$',int2str(0));
fprintf(Arduino,'%s$',int2str(-7));
fclose(Arduino);
```

2 Main components and Invectory

The project mainly revolves around processing onboard the Arduino, so not much hardware is required:

- Arduino
- USB Cable for serial communication
- Mic – replaced by laptop with MATLAB here

3 Results

3.1 Synthetic Data

First, tests were run on data generated in situ from a list of frequencies, coefficients, and phase differences

$$c = \{c_1, c_2, \ldots, c_n\}$$
$$\omega = \{\omega_1, \omega_2, \ldots, \omega_n\}$$
$$\phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$$

which generate a 'generating' function for the discrete signal to follow

$$f(x) = \sum_{i=1}^{n} c_i \cdot sin(w_i x + \phi_i)$$

implemented as a lambda-function in code

```
auto f_gen = [w, c, p](int x){
                float sum = 0;
                for(int i = 0; i < wlist.size(); i++){
                    sum += c[i] * sin(w[i]*x + p[i]);
                }
                return 7.0*sum/float(std::accumulate(c.begin(), c.end
    (), 0));
            };

auto f = new signal[SIZE];

for(int x = 0; x < SIZE; x++){
    f[i] = signal(f_gen(2*x), f_gen(2*x + 1));
}
```

The normalising factor may be succinctly written as

$$a_N = \frac{7.0}{\sum_{i=1}^{n} c_i} \ .$$

It simply ensures that the cross correlation of the signals do not blow up and that a single signal agnostic threshold value may be chosen as a metric for matching. The factor of 7 scales the float value to the range (-7, 7) so as to maintain reasonable resolution after conversion to integer type to save memory.

The signal is generated as even/odd pairs to facilitate pairwise storage implemented by Doobit™. The typename `signal` is an alias for `doobit`, kept as such for modularity amongst storage backends, and to possibly expand across architectures, saving major edits.

Playing with the described parameters, experiments were performed, and the results have been tabulated in Figure 6. Since we look for a global maxima of cross correlation, the phase is irrelevant to the results, but for the sake of completeness, an indication has been provided where phase shifts were tested. Several arbitrary values were tested, with the result being uneaffected as expected.

All synthetic tests were performed with a resolution of 1ms, and 400 data points.

3.2 Real Inputs

Not having a microphone module for the Arduino, we resorted to passing a recorded array to the device via MATLAB, opening its own can of worms by way of buffer overflows and race conditions, as described in detail in **??**.

For currently unknown reasons, we had memory overflows with the same size of real data as we ran synthetic tests for. Curiously, in this case, the Arduino continuosly prints `'w'` on the Serial line. After much testing we have linked this behaviour to memory overflow, but are yet to find resources confirming it.

Reducing the size, however, allows us to perform some tests, limited by the data transfer speed as well, with the serial line taking up to 30 seconds to transfer all the data successfully, yet with significant packet loss. The best transfer ratio we were able to obtain after fine tuning the transfer rates and timing was 400 packets received for 420 sent, but at this delicate parameter island, the Arduino would, with roughly half probability, run out of memory. It may be dependent on buffering of incoming packets. Buffering just a few more packets may have been pushing it over the edge.

NCC Threshold (t)	Input frequencies and coefficients (kHz)	Output (kHz)
0.1	0.3, 0.5, 0.8	0.3, 0.5, 0.8
0.1	0.3, 0.5, 0.8†	0.3, 0.5, 0.8
0.1	0.3, 0.5†, 0.8†	0.3, 0.5, 0.8
0.2	0.3, 0.5†, 0.8†	0.3, 0.5, 0.8
0.2	0.3, 0.5†, 0.8†	0.3, 0.5, 0.8
0.2	0.3, 4$\times$ 0.5†, 0.8†	0.3, 0.5, 0.8
0.4	0.3, 4$\times$ 0.5†, 0.8†	0.3, 0.5, 0.8
0.5	0.3, 4$\times$ 0.5†, 0.8†	0.3, 0.5
0.6	0.3, 4$\times$ 0.5†, 0.8†	0.5
0.8	0.3, 4$\times$ 0.5†, 0.8†	$\emptyset$
0.1	0.3, 0.5, 0.35†	0.3, 0.4, 0.5
0.2	0.3, 0.5, 0.35†	0.3, 0.5

Fig. 6: Experiments with varying synthetic inputs and thresholds.
†. Phase shifted. Weight unit unless otherwise specified

For generating required frequencies, the Android app Frequency Sound Generator was used.

All real tests were performed with a resolution of 1ms, and 200 data points.

We further noted that the algorithm itself could give us better resolution with more frequencies. However, as the memory in our Arduino was quite strained, we ran into memory overflow issues with garbage data values being sent. Howver, this algorithm can be easily scaled up on better devices by adding more frequencies to test with.

3.3 Proofs

Screenshots for the code compiled and working may be seen in Figure 8 and Figure 9 respectively for the group emmbers, and a photo each of the physical setups is found in Figure 10. These, along with some more photos, and video demonstrations, have been uploaded to the required shared directory, as well as to our own folder, linked here.

NCC Threshold (t)	Input signals	Output (kHz)
0.1	0.4 kHz (generated using a phone speaker)	0.2*, 0.3, 0.4
0.1	0.5 kHz (generated using a phone speaker)	0.3, 0.4, 0.5, 0.6*
0.1	0.6 kHz (generated using a phone speaker)	0.4, 0.5, 0.6, 0.7*
0.2	0.6 kHz (generated using a phone speaker)	0.4, 0.5, 0.6
0.3	0.6 kHz (generated using a phone speaker)	$\emptyset$
0.1	Speech ("Hello Hello")	0.2*, 0.3, 0.4, 0.5*, 0.6
0.1	Middle C (261 Hz)	0.3
0.1	Middle A (440 Hz)	0.3, 0.4

Fig. 7: Experiments with real inputs and thresholds. $*$ indicating the frequency appeared sometimes during several test runs.

Appendix A Arduino Code

Here is the full code sent to the Arduino, verbatim.

```
// main.ino

#include "ArduinoSTL.h"

#define assert(x) (void*)0
#define SIZE 200

// bit mask storage
struct doobit{
  uint_fast8_t data;

  doobit(int8_t x = 0, int8_t y = 0){ // handles all our casts too
    this->storelow(x);
    this->storehigh(y);
  }

  void storelow(int8_t);
  void storehigh(int8_t);

  int8_t getlow();
  int8_t gethigh();

  int16_t operator*(doobit& b){
    auto highprod = this->gethigh() * b.gethigh();
    auto lowprod = this->getlow() * b.getlow();
    return (highprod + lowprod);
  }
};
typedef doobit signal;

```

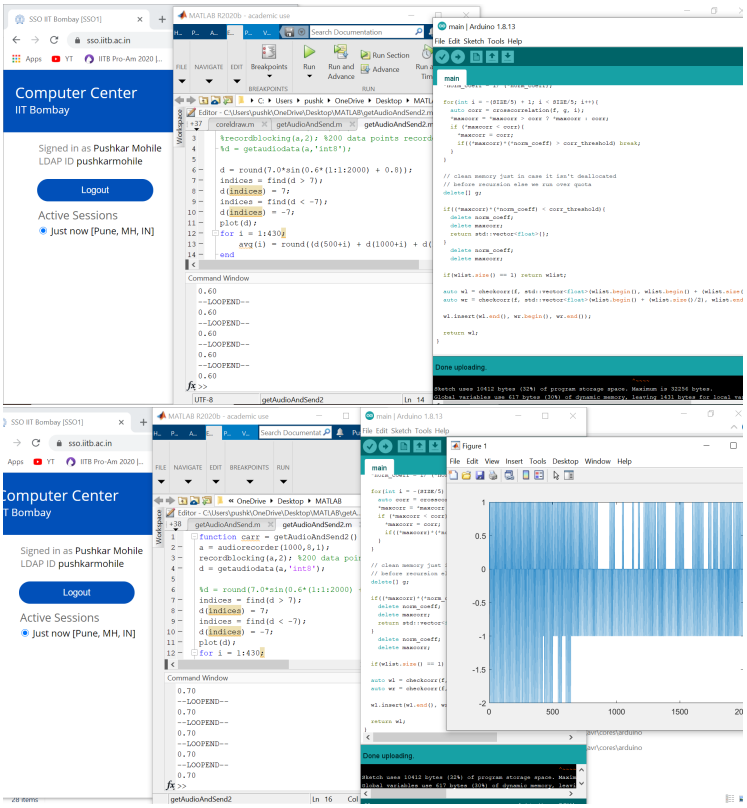

Fig. 8: Code compiled and running tests for Pushkar

```
// functions and constants
int correlation(signal*, signal*);
int crosscorrelation(signal*, signal*, int = 0);
std::vector<float> checkcorr(signal*, std::vector<float>);

std::vector<float> freq{};

const float corr_threshold = 0.1;

// data input
bool recording = false;
uint16_t recorded = 0;
String text;
uint8_t has_num = 0;
uint8_t k[] = {0, 0};

signal f[SIZE];

void setup(){
    Serial.begin(115200);

```

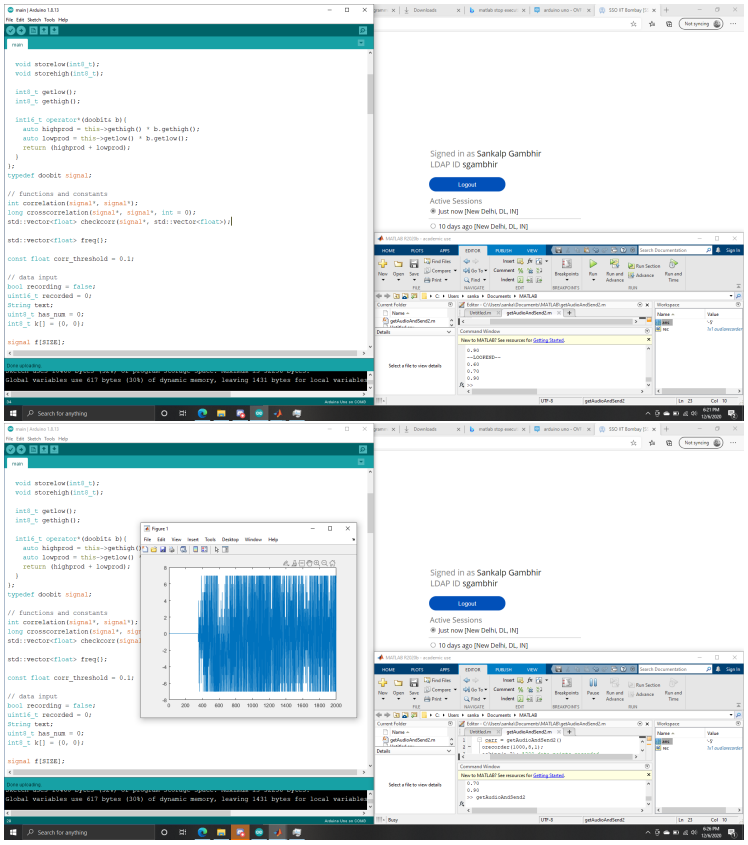

Fig. 9: Code compiled and running tests for Sankalp

```
  // clear data just in case
  for(int i = 0; i < SIZE; i++){
    f[i] = 0;
  }

  for(float i = 0.1; i < 1.0; i += 0.1){
    freq.push_back(i);
  }

  // recording indication
  pinMode(13, 1);
  digitalWrite(13,1);
  recording = true;
}

void loop(){

    if(recording){
        // recording data
        if (Serial.available()){
```

```
          k[has_num] = Serial.parseInt();
          has_num++;
        }
        if ((has_num >= 2) && recorded < SIZE) {
          f[recorded++] = signal(k[0], k[1]);
          has_num = 0;
        }
        if(recorded >= SIZE){
          Serial.write("G"); // we Good
          digitalWrite(13,0);
          recording = false;
        }
    }
    else{
        // calculate with the data

        // f coming from data
        /*
        auto f_gen = [](int x){
                    return (7.0*(sin(0.3 * x) + 4*sin(0.5 * x) + sin
    (0.6*x + 0.6))/6.0);
                    };

        for(int i = 0; i < SIZE; i++){
            f[i] = signal(f_gen(2*i), f_gen(2*i+1));
        }
        */

        auto wpresent = checkcorr(f, freq);

        for(auto w : wpresent){
            Serial.println(w);
        }

        Serial.println("--LOOPEND--");
    }

    return;
}

// definitions


void doobit::storelow(int8_t x){
  this->data &= 0b11110000;  // clear for storage

  x += 7; // remove signed component
  assert((!(x & 0b11110000)) && "doobit range violation");

  this->data |= x;
}

```

```
123 void doobit::storehigh(int8_t x){
124   this->data &= 0b00001111;  // clear for storage
125 
126   x += 7; // remove signed component
127   assert((!(x & 0b11110000)) && "doobit range violation");
128 
129   this->data |= (x << 4);
130 }
131 
132 int8_t doobit::getlow(){
133   int8_t x = (data & 0b00001111); // bitmask
134   return (x-7); // reinsert sign
135 }
136 
137 int8_t doobit::gethigh(){
138   int8_t x = ((data & 0b11110000) >> 4); // bitmask and shift
139   return (x-7); // reinsert sign
140 }
141 
142 
143 int correlation(signal* f, signal* g){
144   int sum = 0;
145 
146   for(int i = 0; i < SIZE; i++){
147     sum += f[i] * g[i];
148   }
149   return sum;
150 }
151 
152 int crosscorrelation(signal* f, signal* g, int m){
153   int sum = 0;
154 
155   if(m >= 0){
156     for(int i = 0; i < SIZE - m; i++){
157       sum += f[i] * g[i+m];
158     }
159     for(int i = 0; i < m; i++){
160       sum += f[i+SIZE-m] * g[i];
161     }
162   }
163   else{
164     m = -m;
165     for(int i = 0; i < m; i++){
166       sum += f[i] * g[i+SIZE-m];
167     }
168     for(int i = m; i < SIZE; i++){
169       sum += f[i] * g[i-m];
170     }
171   }
172   return sum;
173 }
174 
```

```
175 std::vector<float> checkcorr(signal* f, std::vector<float> wlist){
176
177   if(wlist.size() == 0) return wlist;
178
179   float *maxcorr = new float(-1);
180
181   auto g_gen = [wlist](int x){
182           float sum = 0;
183           for(auto w : wlist){
184             sum += sin(w*x);
185           }
186           return 7.0*sum/float(wlist.size());
187         };
188
189   auto g = new signal[SIZE];
190
191   for(int i = 0; i < SIZE; i++){
192     g[i] = signal(g_gen(2*i), g_gen(2*i + 1));
193   }
194
195   float* norm_coeff = new float(0);
196   *norm_coeff = sqrt(((double)correlation(f, f) * (double)correlation(
      g, g)));
197   *norm_coeff = 1/ (*norm_coeff);
198
199   for(int i = -(SIZE/5) + 1; i < SIZE/5; i++){
200     auto corr = crosscorrelation(f, g, i);
201     *maxcorr = *maxcorr > corr ? *maxcorr : corr;
202     if (*maxcorr < corr){
203       *maxcorr = corr;
204       if((*maxcorr)*(*norm_coeff) > corr_threshold) break;
205     }
206   }
207
208   // clean memory just in case it isn't deallocated
209   // before recursion else we run over quota
210   delete[] g;
211
212   if((*maxcorr)*(*norm_coeff) < corr_threshold){
213     delete norm_coeff;
214     delete maxcorr;
215     return std::vector<float>{};
216   }
217     delete norm_coeff;
218     delete maxcorr;
219
220   if(wlist.size() == 1) return wlist;
221
222   auto wl = checkcorr(f, std::vector<float>(wlist.begin(), wlist.begin
      () + (wlist.size()/2)));
223   auto wr = checkcorr(f, std::vector<float>(wlist.begin() + (wlist.
      size()/2), wlist.end()));
```

```
224
225   wl.insert(wl.end(), wr.begin(), wr.end());
226
227   return wl;
228 }
```

Appendix B MATLAB

The code used inside MATLAB to record and send audio to the Arduino.

```
function carr = getAudioAndSend2()
  a = audiorecorder(1000,8,1);
  recordblocking(a,2); %2000 data points recorded

  d = getaudiodata(a,'int8');

  % clip the data at +- 7
  indices = find(d > 7);
  d(indices) = 7;
  indices = find(d < -7);
  d(indices) = -7;

  % show data
  plot(d);

  % connect to arduino
  Ard = serial("COM3","BaudRate",115200);
  fopen(Ard);

  % send 400 and change samples
  for i = 1:430
    fprintf(Ard,'%s\n',int2str(d(i)));
    pause(3/100); % prevents race condition!
  end

  pause(1);

  % read the output
  for i=1:15
      y = fscanf(Ard,'%s');
      fprintf('%s\n', y);
  end

  fclose(Ard);
end
```

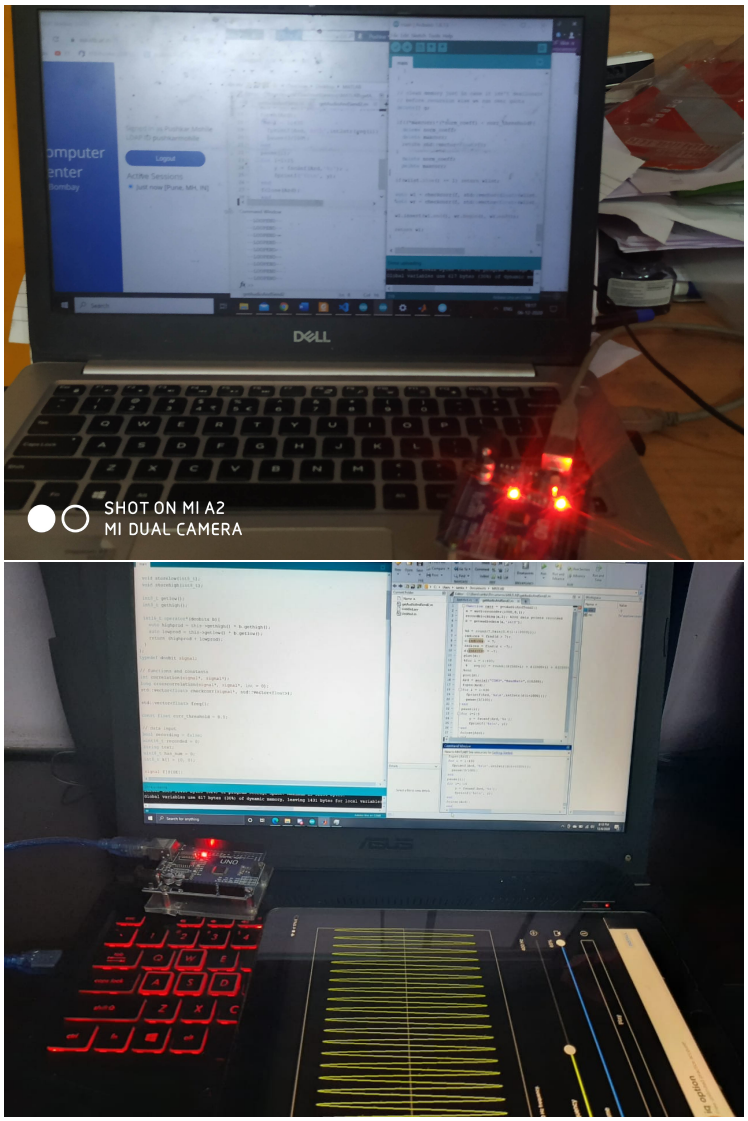

Fig. 10: Physical setup. Top, Pushkar; bottom, Sankalp.