

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

A 4-bit dance in frequency space

Pushkar Mohile (180260027)¹ and Sankalp Gambhir (180260032)¹

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 a new 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

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 bottleneck 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

$$R_{xy}[k] = \sum_i x[i]y[i - k] \quad (1)$$

Along with some normalization. The harmonics form a linearly independent set and return 0 correlation when the product is integrated over several time periods, i.e.

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

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.

$$c_{xy} = \frac{R_{xy}[0]}{\sqrt{R_{xx}[0]R_{yy}[0]}} \quad (3)$$

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:

$$c_{xy} = \frac{\max\{R_{xy}[k]\}}{\sqrt{R_{xx}[0]R_{yy}[0]}} \quad (4)$$

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, ie numbers from -7 to 7. This has several benefits: a) You can store more numbers b) You can perform the calculations in a smaller time period as the Arduino processor is 8-Bit. Instead of requiring multiple clock cycles to process 16 / 32 Bit datatypes like float 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.

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.

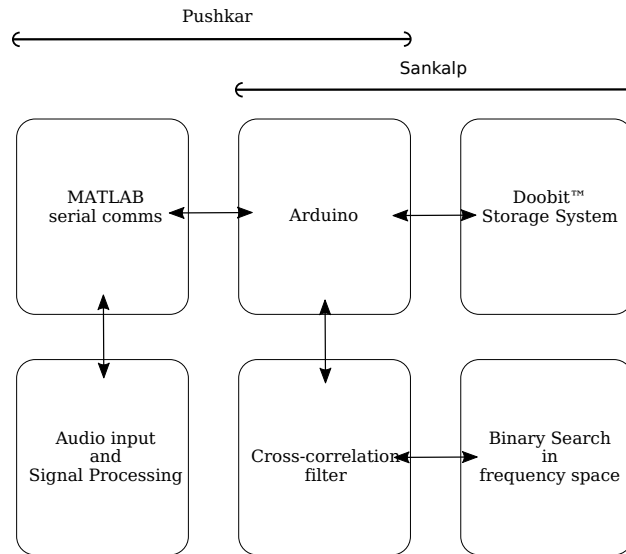


Fig. 1: Block diagram and work distribution.

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 terribly 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
6         this->storelow(x);
7         this->storehigh(y);
8     }
9
10    void storelow(int8_t);
11    void storehigh(int8_t);
12
13    int8_t getlow();
14    int8_t gethigh();
15
16    int16_t operator*(doobit& b);
17 };

```

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 evaluation 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 significant bits of the storage **data** respectively. We look at one of the functions:

```
1 void doobit::storelow(int8_t x){
2     this->data &= 0b11110000; // clear for storage
3
4     x += 7; // remove signed component
5     assert(!(x & 0b11110000) && "doobit range violation");
6
7     this->data |= x;
8 }
```

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:

```
1 int8_t doobit::getlow(){
2     int8_t x = (data & 0b00001111); // bitmask
3     return (x-7); // reinsert sign
4 }
```

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 correlation 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.

```

1 int16_t operator*(doobit& b){
2     auto highprod = this->gethigh() * b.gethigh();
3     auto lowprod = this->getlow() * b.getlow();
4     return (highprod + lowprod);
5 }

```

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

2 Main components and Inventory

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
- LEDs for displaying frequencies (optional extra)

3 Results

3.1 Synthetic Data

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

$$\begin{aligned}
 c &= \{c_1, c_2, \dots, c_n\} \\
 \omega &= \{\omega_1, \omega_2, \dots, \omega_n\} \\
 \phi &= \{\phi_1, \phi_2, \dots, \phi_n\}
 \end{aligned}$$

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

```

1 auto f_gen = [w, c, p](int x){
2     float sum = 0;
3     for(int i = 0; i < wlist.size(); i++){
4         sum += c[i] * sin(w[i]*x + p[i]);
5     }
6     return 7.0*sum/float(std::accumulate(c.begin(), c.end
7     (), 0));
8 };
9 auto f = new signal[SIZE];
10
11 for(int x = 0; x < SIZE; x++){
12     f[i] = signal(f_gen(2*x), f_gen(2*x + 1));
13 }

```

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 2**. 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 unaffected 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 continuously 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.

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× 0.5 [†] , 0.8 [†]	0.3, 0.5, 0.8
0.4	0.3, 4× 0.5 [†] , 0.8 [†]	0.3, 0.5, 0.8
0.5	0.3, 4× 0.5 [†] , 0.8 [†]	0.3, 0.5
0.6	0.3, 4× 0.5 [†] , 0.8 [†]	0.5
0.8	0.3, 4× 0.5 [†] , 0.8 [†]	∅
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. 2: Experiments with varying synthetic inputs and thresholds.

[†]. Phase shifted. Weight unit unless otherwise specified

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.

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.

3.3 Proofs

Code and compilation screenshots of code compiled and running on arduino

Arduino and Humans photo video uploads

NCC Threshold (t)	Input signals	Output (kHz)
0.1	0.4 kHz (generated using a phone speaker)	INSERT
0.1	0.5 kHz (generated using a phone speaker)	INSERT
0.1	0.6 kHz (generated using a phone speaker)	INSERT
0.2	0.6 kHz (generated using a phone speaker)	INSERT
0.3	0.6 kHz (generated using a phone speaker)	INSERT
0.5	0.6 kHz (generated using a phone speaker)	INSERT
0.1	Speech	INSERT
0.1	SOME SHIT CHORD	INSERT
0.1	SOME SHIT CHORD	INSERT
0.1	SOME SHIT CHORD	INSERT

Fig. 3: Experiments with real inputs and thresholds.

Appendix A Arduino Code

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

```

1 // main.ino
2
3 #include "ArduinoSTL.h"
4
5 #define assert(x) (void*)0
6 #define SIZE 200
7
8 // bit mask storage
9 struct doobit{
10     uint_fast8_t data;
11
12     doobit(int8_t x = 0, int8_t y = 0){ // handles all our casts too
13         this->storelow(x);
14         this->storehigh(y);
15     }
16
17     void storelow(int8_t);
18     void storehigh(int8_t);
19
20     int8_t getlow();
21     int8_t gethigh();
22
23     int16_t operator*(doobit& b){
24         auto highprod = this->gethigh() * b.gethigh();
25         auto lowprod = this->getlow() * b.getlow();
26         return (highprod + lowprod);
27     }
28 };

```



```

29 typedef doobit signal;
30
31 // functions and constants
32 int correlation(signal*, signal*);
33 int crosscorrelation(signal*, signal*, int = 0);
34 std::vector<float> checkcorr(signal*, std::vector<float>);
35
36 std::vector<float> freq{};
37
38 const float corr_threshold = 0.1;
39
40 // data input
41 bool recording = false;
42 uint16_t recorded = 0;
43 String text;
44 uint8_t has_num = 0;
45 uint8_t k[] = {0, 0};
46
47 signal f[SIZE];
48
49 void setup(){
50     Serial.begin(9600);
51
52     // clear data just in case
53     for(int i = 0; i < SIZE; i++){
54         f[i] = 0;
55     }
56
57     for(float i = 0.1; i < 1.0; i += 0.1){
58         freq.push_back(i);
59     }
60 }
61
62 void loop(){
63
64     if(recording){
65         // recording data
66         if (Serial.available()){
67             text = Serial.readStringUntil('$');
68             k[has_num] = text.toInt();
69             has_num++;
70         }
71         if ((has_num >= 2) && recorded < SIZE) {
72             recorded++;
73             f[recorded] = signal(k[0], k[1]);
74             has_num = 0;
75         }
76         if(recorded >= SIZE){
77             Serial.write("G"); // we Good
78             digitalWrite(13,1);
79             recording = false;
80         }

```

```

81     }
82     else{
83         // calculate with the data
84
85         // f coming from data
86         /*
87         auto f_gen = [](int x){
88             return (7.0*(sin(0.3 * x) + 4*sin(0.5 * x) + sin
89             (0.6*x + 0.6))/6.0);
90             };
91
92         for(int i = 0; i < SIZE; i++){
93             f[i] = signal(f_gen(2*i), f_gen(2*i+1));
94         }
95         */
96
97         auto wpresent = checkcorr(f, freq);
98
99         for(auto w : wpresent){
100             Serial.println(w);
101         }
102
103         Serial.println("TEST$$$$$$$");
104     }
105     return;
106 }
107
108 // definitions
109
110
111 void doobit::storelow(int8_t x){
112     this->data &= 0b11110000; // clear for storage
113
114     x += 7; // remove signed component
115     assert(!(x & 0b11110000) && "doobit range violation");
116
117     this->data |= x;
118 }
119
120 void doobit::storehigh(int8_t x){
121     this->data &= 0b00001111; // clear for storage
122
123     x += 7; // remove signed component
124     assert(!(x & 0b11110000) && "doobit range violation");
125
126     this->data |= (x << 4);
127 }
128
129 int8_t doobit::getlow(){
130     int8_t x = (data & 0b00001111); // bitmask
131     return (x-7); // reinsert sign

```

```

132 }
133
134 int8_t doobit::gethigh(){
135     int8_t x = ((data & 0b11110000) >> 4); // bitmask and shift
136     return (x-7); // reinsert sign
137 }
138
139
140 int correlation(signal* f, signal* g){
141     int sum = 0;
142
143     for(int i = 0; i < SIZE; i++){
144         sum += f[i] * g[i];
145     }
146     return sum;
147 }
148
149 int crosscorrelation(signal* f, signal* g, int m){
150     int sum = 0;
151
152     if(m >= 0){
153         for(int i = 0; i < SIZE - m; i++){
154             sum += f[i] * g[i+m];
155         }
156         for(int i = 0; i < m; i++){
157             sum += f[i+SIZE-m] * g[i];
158         }
159     }
160     else{
161         m = -m;
162         for(int i = 0; i < m; i++){
163             sum += f[i] * g[i+SIZE-m];
164         }
165         for(int i = m; i < SIZE; i++){
166             sum += f[i] * g[i-m];
167         }
168     }
169     return sum;
170 }
171
172 std::vector<float> checkcorr(signal* f, std::vector<float> wlist){
173
174     if(wlist.size() == 0) return wlist;
175
176     float maxcorr = -1;
177
178     auto g_gen = [wlist](int x){
179         float sum = 0;
180         for(auto w : wlist){
181             sum += sin(w*x);
182         }
183         return 7.0*sum/float(wlist.size());

```

```

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

```

Appendix B MATLAB

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

```
1 function carr = getAudioAndSend2()
2 a = audiorecorder(10000,8,1);
3 recordblocking(a,1); %200 data points recorded
4 carr = getaudiodata(a,'int8')
5 Ard = serial("COM4","BaudRate",9600);
6 fopen(Ard);
7 for i = 1:200
8     fprintf(Ard,'%d\n',int2str(carr(i+1500)));
9 end
10 fscanf(Ard)
11 fclose(Ard);
12 end
```