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

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. In order to achieve this processing, we came up with multiple optimizations.

1)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

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

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

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

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

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

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 :

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

This ensures that the loss due to the phase shift isn't captured. 1) Sampling The Arduino has 2kB of SRAM available this means that any calculations we want to do (Array multiplication) must not exceed 2kB. In order to do this, we employ

1.1 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.

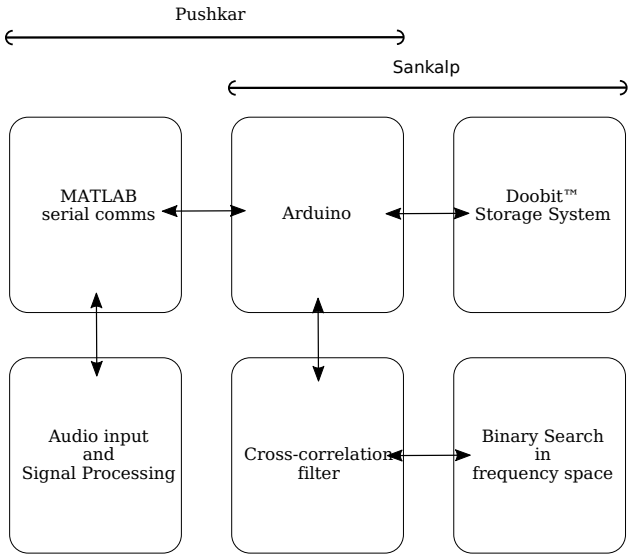

Fig. 1: Block diagram and work distribution.

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.

1.2 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.

```
// 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);
};
```

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.

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
- 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

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

which generate a 'generating' function for the discrete signal later

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

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™.

NCC Threshold (t)	Input frequencies and coefficients (kHz)	Output
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†	$\emptyset$
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. 2: Experiments in varying synthetic inputs and thresholds.
†. Phase shifted. Weight unit unless otherwise specified

screenshots of code compiled and running on arduino
photo video uploads

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;

// 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(9600);

  // 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);
  }
}

void loop(){

    if(recording){
        // recording data
    if (Serial.available()){
      text = Serial.readStringUntil('$');
      k[has_num] = text.toInt();
      has_num++;
    }
    if ((has_num >= 2) && recorded < SIZE) {
      recorded++;
      f[recorded] = signal(k[0], k[1]);
      has_num = 0;
    }
    if(recorded >= SIZE){
      Serial.write("G"); // we Good
      digitalWrite(13,1);
      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);
        }
```

```
101
102         Serial.println("TEST$$$$$$$");
103     }
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   norm_coeff = 1/norm_coeff;
194 
195   for(int i = -(SIZE/5) + 1; i < SIZE/5; i++){
196     auto corr = crosscorrelation(f, g, i);
197     maxcorr = maxcorr > corr ? maxcorr : corr;
198     if (maxcorr < corr){
199       maxcorr = corr;
200       if(maxcorr*norm_coeff > corr_threshold) break;
201     }
202   }
203 
```

```
204   // clean memory just in case it isn't deallocated
205   // before recursion else we run over quota
206   delete[] g;
207
208   if(maxcorr*norm_coeff < corr_threshold) return std::vector<float>{};
209
210   if(wlist.size() == 1) return wlist;
211
212   auto wl = checkcorr(f, std::vector<float>(wlist.begin(), wlist.begin
        () + (wlist.size()/2)));
213   auto wr = checkcorr(f, std::vector<float>(wlist.begin() + (wlist.
        size()/2), wlist.end()));
214
215   wl.insert(wl.end(), wr.begin(), wr.end());
216
217   return wl;
218 }
```

Appendix B MATLAB

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

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