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 are Fourier Transforms imp? Write st.

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.

$$(f * g)(t) \triangleq \int_{-\infty}^{+\infty} \overline{f(\tau)} \cdot g(t+\tau) d\tau \tag{1}$$

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.

$$corr(f, g) \triangleq \max((f * g)(t))_t \tag{2}$$

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

$$R_{xy}[k] = \sum_i x[i]y[i-k] \tag{3}$$

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.

$$\frac{1}{\pi}\int_0^{2\pi} sin(n_1 x)sin(n_2 x) = \delta_{nn'} \tag{4}$$

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 coeficients.

$$c_{xy} = \frac{R_{xy}[0]}{\sqrt{R_{xx}[0]R_{yy}[0]}} \tag{5}$$

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:

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

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.

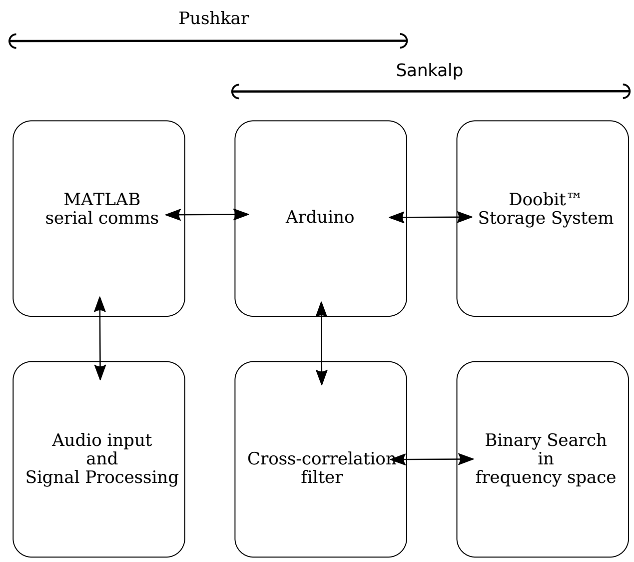

Fig. 1: Block diagram and work distribution.

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.

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.

```
// 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.

1.5 Serial Communication and the Great Horse Race

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

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

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

```
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 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 uneaffected as expected.

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

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†	$\emptyset$
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 with varying synthetic inputs and thresholds.
†. Phase shifted. Weight unit unless otherwise specified

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.

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.

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.

3.3 Proofs

Code and compilation screenshots of code compiled and running on arduino

Arduino and Humans photo video uploads

Fig. 4: Code compiled and running synthetic tests. Left, Pushkar; right, Sankalp.

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);
  }

```

Fig. 5: Code running. Left, Pushkar; right, Sankalp.

```
  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);
  }

  // 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;
}

void doobit::storehigh(int8_t x){
  this->data &= 0b00001111;  // clear for storage

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

  this->data |= (x << 4);
}

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

int8_t doobit::gethigh(){
  int8_t x = ((data & 0b11110000) >> 4); // bitmask and shift
  return (x-7); // reinsert sign
```

```
}


int correlation(signal* f, signal* g){
  int sum = 0;

  for(int i = 0; i < SIZE; i++){
    sum += f[i] * g[i];
  }
  return sum;
}

int crosscorrelation(signal* f, signal* g, int m){
  int sum = 0;

  if(m >= 0){
    for(int i = 0; i < SIZE - m; i++){
      sum += f[i] * g[i+m];
    }
    for(int i = 0; i < m; i++){
      sum += f[i+SIZE-m] * g[i];
    }
  }
  else{
    m = -m;
    for(int i = 0; i < m; i++){
      sum += f[i] * g[i+SIZE-m];
    }
    for(int i = m; i < SIZE; i++){
      sum += f[i] * g[i-m];
    }
  }
  return sum;
}

std::vector<float> checkcorr(signal* f, std::vector<float> wlist){

  if(wlist.size() == 0) return wlist;

  float *maxcorr = new float(-1);

  auto g_gen = [wlist](int x){
          float sum = 0;
          for(auto w : wlist){
            sum += sin(w*x);
          }
          return 7.0*sum/float(wlist.size());
        };

  auto g = new signal[SIZE];

  for(int i = 0; i < SIZE; i++){
```

```
    g[i] = signal(g_gen(2*i), g_gen(2*i + 1));
  }

  float* norm_coeff = new float(0);
  *norm_coeff = sqrt(((double)correlation(f, f) * (double)correlation(g, g)));
  *norm_coeff = 1/ (*norm_coeff);

  for(int i = -(SIZE/5) + 1; i < SIZE/5; i++){
    auto corr = crosscorrelation(f, g, i);
    *maxcorr = *maxcorr > corr ? *maxcorr : corr;
    if (*maxcorr < corr){
      *maxcorr = corr;
      if((*maxcorr)*(*norm_coeff) > corr_threshold) break;
    }
  }

  // clean memory just in case it isn't deallocated
  // before recursion else we run over quota
  delete[] g;

  if((*maxcorr)*(*norm_coeff) < corr_threshold){
    delete norm_coeff;
    delete maxcorr;
    return std::vector<float>{};
  }
    delete norm_coeff;
    delete maxcorr;

  if(wlist.size() == 1) return wlist;

  auto wl = checkcorr(f, std::vector<float>(wlist.begin(), wlist.begin() + (wlist.size()/2)));
  auto wr = checkcorr(f, std::vector<float>(wlist.begin() + (wlist.size()/2), wlist.end()));

  wl.insert(wl.end(), wr.begin(), wr.end());

  return wl;
}
```

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); %200 data points recorded
d = getaudiodata(a,'int8');
%d = round(7.0*sin(0.6*(1:1:500)));
indices = find(d > 7);
d(indices) = 7;
indices = find(d < -7);
d(indices) = -7;
plot(d);

Ard = serial("COM3","BaudRate",115200);
fopen(Ard);
for i = 1:430
  fprintf(Ard,'%s\n',int2str(d(i)));
  pause(3/100);
end
pause(1);
for i=1:15
    y = fscanf(Ard,'%s');
    fprintf('%s\n', y);
end
fclose(Ard);
end
```