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

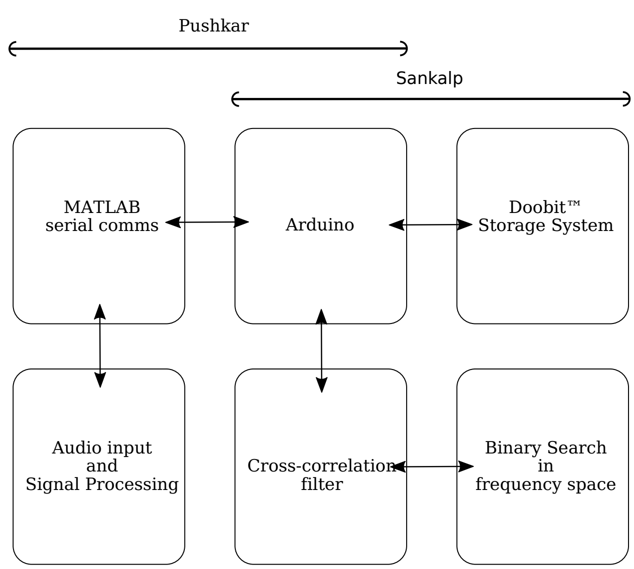

Fig. 1: Block diagram and work distribution.

1.1 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

not good

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 <cmath>        // for sqrt
#include <vector>       // for vector...
#include <algorithm>    // the OTHER std::move
#include <cstdint>      // ALL the ints
#include <cassert>      // testing

#define SIZE 100

// custom typing -- forward declaration
struct doobit;

// 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 = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8};
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};

typedef doobit signal;
signal f[SIZE];

void setup(){
    Serial.begin(9600);

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

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(0.8 * 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){
            printf("%f ", w);
        }

        printf("\n");
    }

    return;
}

// definitions

// 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);
  }
};

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 = -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));
  }

  auto norm_coeff = sqrt((correlation(f, f) * correlation(g, g)));
  norm_coeff = 1/norm_coeff;

  for(int i = -SIZE+1; i < SIZE; i++){
    auto corr = crosscorrelation(f, g, i);
    maxcorr = maxcorr > corr ? maxcorr : corr;
  }

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

  if(maxcorr*norm_coeff < corr_threshold) return {};

  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()));

  std::move(wr.begin(), wr.end(), std::back_inserter(wl));

  return wl;
}
```

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
```