Application Note: The Digital Signal Analysis based on MATLAB and Arduino Micro-controller

By Haolin Yang

Introduce:

This note will discuss the particular subject of DSP with MATLAB and Arduino Micro-controller that I has worked on in this class. The detailed content will include how to improve the MATLAB code of last quarter, how to use micro controller to do DSP and suggestion for dealing with audio cable problem.

Part one: MATLAB Improvement

To start with, our team decide to use MATLAB code that professor provided in last quarter because we think this way is much easier than using micro-controller. But the code can not realize the real-time analysis. We must use the audacity to record the output and SYNC signal and input the produced wav file into MATLAB, which is really low-efficient and complicated. So I use function of audioread and recordblocking() to realize the record the signal and output as a wav file. Then use function of audioread to load this file, which means only one click of start, we can just get the result of DSP, no longer wasting time to transfer file from audacity to MATLAB. Because of the large volume of data, sometimes speed of analysis of computer is really slow, so we can use gpuArray function to make computer do the calculation with GPU instead of CPU. This is mainly because the GPU of computer has multiple courses to meet requirement of analysis of image information compared to just several courses in the CPU. Finally to deal with the problem of fuzzy of result plot, I use function of Z=Z/abs((max(max(Z)))) and Z=20*log10(Z) to do the sharpen operation. The theory behind this method is that taking the form of logarithm can help to increase the data break, which help to enhance the color spacing and make the plot clear. But this method also can increase the noise, so please use it carefully.

Part two: Try to use Arduino Micro-controller to do DSP.

Actually, our team plan treat this way as a extra discovery for DSP. Because micro-controller has only one course and cannot operate and calculate matrix directly as MATLAB. So the key problem for the part is how to do the FFT on the micro-controller and how to plot the results clearly. I choose teensy 3.2 because it has very high speed and I also very familiar with it last quarter. The detailed parameter I will post in the appendix. Arduino provides powerful math library which enable us to use the already-made FFT library and I also found the way to how to build up the spectrum to show results with python and arduino on the website of adafruit(https://learn.adafruit.com/fft-fun-with-fourier-transforms?view=all). Unfortunately, I cannot test whether this way can work properly because we encounter a problem if

collecting digital data from radar system, which wastes a lot of time.

Part third: Suggestion for deal with problem of collecting digital data

We found our signal was shorten or two channel cannot be separated when we connect to audio cable to the computer with integrated soundcard. So I suggest future students use computer with separated micro input. If this way do not work, a large desktop computer can also meet your requirement.

Appendix A :MATLAB code and explain

```
close all;
clear all;
recObj = audiorecorder(Fs, 16, 2);
disp('Start speaking.')
recordblocking(recObj, 10);
disp('End of Recording.');
q= getaudiodata(recObj);
plot(q);
data = 0;
figure;
plot(q);
```

For this part, we use audiorecorder function to realize the recording SYNC and output signal for 10 second and recording format is a audio file with 2 channels, 16bits and example frequency is 44100Hz and output is a matrix named ‘q’.

```
load handel.mat
filename = 'handel.wav';
audiowrite(filename,q,Fs);
clear q Fs
[q,Fs] = audioread(filename);
sound(q,Fs);
```

This part is the key to connect recording section to the digital analysis section. Frist we load a intermediate named handel and input the digital data into this matrix. Finally using audiowrite function to output a wav file with frequency with 44100 Hz

```
FS=44100
c = 3E8;% speed of light
Tp = 12.8E-3; %(s) pulse time
N = Tp*FS; %# of samples per pulse
fstart = 2300E6; %(Hz) LFM start frequency
```

```
fstop = 2560E6; %(Hz) LFM stop frequency
BW = fstop-fstart; %(Hz) transmti bandwidth
f = linspace(fstart, fstop, N/2); %instantaneous transmit frequency
%range resolution
rr = c/(2*BW);
max_range = rr*N/2;
The aim of part is to set the parameter and figure the max distance range.
[Y,FS] = audioread('1.wav');%the input appears to be inverted
trig = -1*Y(:,1);%SYNC signal
%[Y,FS] = audioread('HU3.wav');
s = -1*Y(:,2);%radar signal
%parse the data here by triggering off rising edge of sync pulse
count = 0;
thresh = 0;
start = (trig > thresh);%convert data into binary
d=size(start,1)-N;
for ii = 100:(d)
if start(ii) == 1 & mean(start(ii-11:ii-1)) == 0
%start2(ii) = 1;
count = count + 1;
sif(count,:) = s(ii:ii+N-1);
time(count) = ii*1/FS;
end
end
disp('end_for')
%subtract the average
ave = mean(sif,1);
for ii = 1:size(sif,1);
sif(ii,:) = sif(ii,:) - ave;
end
zpad = 8*N/2;

%RTI plot
figure(10);
dbv = fft(sif,zpad,2);
v=dbv;
S =abs(v(:,1:size(v,2)/2));
m =mean(mean(v));
Z=S-m;
Z=abs(Z);
%Z=Z/abs((max(max(Z))));
%Z=20*log10(Z);
U=linspace(0,max_range,zpad);
imagesc(U,time,Z);
```

```
colorbar;
ylabel('time (s)');
xlabel('range (m)');
title('RTI without clutter rejection');

%pulse cancelor RTI plot
figure(20);
sif2 = sif(2:size(sif,1),:)-sif(1:size(sif,1)-1,:);
v = fft(sif2,zpad,2);
p=v;
R = linspace(0,max_range,zpad);
abv=p(:,1:size(v,2)/2);
m =mean(mean(p));
J=abs(p-m);
J=J/abs((max(max(J))));
J=20*log10(J);
K=J(:,1:size(J,2)/2);
imagesc(R,time,K);
colorbar;
ylabel('time (s)');
xlabel('range (m)');
title('RTI with 2-pulse cancelor clutter rejection');
```

In this part, we use two function J=J/abs((max(max(J)))) and J=20*log10(J) to amplify the distance between each two number, but its tradeoff is also to amplify the noise, which has a bad influence on the final output.

Appendix B: Detailed parameters of teensy 3.2

Feature	Teensy 3.0	Teensy 3.2 Teensy 3.1	Units
Price	19.00	19.80	US Dollars
Processor Core Rated Speed Overclockable	MK20DX128VLH5 Cortex-M4 48 96	MK20DX256VLH7 Cortex-M4 72 96	 MHz MHz
Flash Memory Bandwidth Cache	128 96 32	256 192 256	kbytes Mbytes/sec Bytes
RAM	16	64	kbytes
EEPROM	2	2	kbytes
Direct Memory Access	4	16	Channels
Digital I/O Voltage Output Voltage Input	34 3.3V 3.3V Only	34 3.3V 5V Tolerant	Pins Volts Volts
Analog Input Converters Resolution Usable Prog Gain Amp Touch Sensing Comparators	14 1 16 13 0 12 2	21 2 16 13 2 12 3	Pins Bits Bits Pins
Analog Output DAC Resolution	0 -	1 12	Pins Bits
Timers FTM Type PWM Outputs PDB Type CMT (infrared) Type LPTMR Type PIT (interval) Type Systick RTC (date/time) **	11 Total 2 10 1 1 1 4 1 1	12 Total 3 12 1 1 1 4 1 1	 Pins
Communication USB Serial With FIFOs High Res Baud Fast Clock SPI With FIFOs I2C CAN Bus I2S Audio FIFO Size	 1 3 1 3 2 1 1 1 0 1 4	 1 3 2 3 2 1 1 2 1 1 8	
** RTC requires a 32.768 kHz crystal & 3V battery. See the Time Library for details.			

Appendix C:code of Arduino

you can also see the detailed instrument and download the code on the website https://learn.adafruit.com/fft-fun-with-fourier-transforms?view=all

```
// Audio Spectrum Display
// Copyright 2013 Tony DiCola (tony@tonydicola.com)
// This code is part of the guide at http://learn.adafruit.com/fft-fun-with-fourier-transforms/
#define ARM_MATH_CM4
#include <arm_math.h>
#include <Adafruit_NeoPixel.h>
////////////////////////////////////////////////////////////////////////////////
// CONIFIGURATION
// These values can be changed to alter the behavior of the spectrum display.
////////////////////////////////////////////////////////////////////////////////

int SAMPLE_RATE_HZ = 9000;                // Sample rate of the audio in hertz.
float SPECTRUM_MIN_DB = 30.0;             // Audio intensity (in decibels) that maps to low
LED brightness.
```

```
float SPECTRUM_MAX_DB = 60.0;              // Audio intensity (in decibels) that maps to high LED brightness.
int LEDS_ENABLED = 1;                      // Control if the LED's should display the spectrum or not.  1 is true, 0 is false.
                                           // Useful for turning the LED display on and off with commands from the serial port.
const int FFT_SIZE = 256;                  // Size of the FFT.  Realistically can only be at most 256
                                           // without running out of memory for buffers and other state.
const int AUDIO_INPUT_PIN = 14;            // Input ADC pin for audio data.
const int ANALOG_READ_RESOLUTION = 10; // Bits of resolution for the ADC.
const int ANALOG_READ_AVERAGING = 16;  // Number of samples to average with each ADC reading.
const int POWER_LED_PIN = 13;              // Output pin for power LED (pin 13 to use Teensy 3.0's onboard LED).
const int NEO_PIXEL_PIN = 3;               // Output pin for neo pixels.
const int NEO_PIXEL_COUNT = 4;             // Number of neo pixels.  You should be able to increase this without
                                           // any other changes to the program.
const int MAX_CHARS = 65;                  // Max size of the input command buffer


////////////////////////////////////////////////////////////////////////////////
// INTERNAL STATE
// These shouldn't be modified unless you know what you're doing.
////////////////////////////////////////////////////////////////////////////////

IntervalTimer samplingTimer;
float samples[FFT_SIZE*2];
float magnitudes[FFT_SIZE];
int sampleCounter = 0;
Adafruit_NeoPixel pixels = Adafruit_NeoPixel(NEO_PIXEL_COUNT, NEO_PIXEL_PIN, NEO_GRB + NEO_KHZ800);
char commandBuffer[MAX_CHARS];
float frequencyWindow[NEO_PIXEL_COUNT+1];
float hues[NEO_PIXEL_COUNT];


////////////////////////////////////////////////////////////////////////////////
// MAIN SKETCH FUNCTIONS
////////////////////////////////////////////////////////////////////////////////

void setup() {
```

```
  // Set up serial port.
  Serial.begin(38400);

  // Set up ADC and audio input.
  pinMode(AUDIO_INPUT_PIN, INPUT);
  analogReadResolution(ANALOG_READ_RESOLUTION);
  analogReadAveraging(ANALOG_READ_AVERAGING);

  // Turn on the power indicator LED.
  pinMode(POWER_LED_PIN, OUTPUT);
  digitalWrite(POWER_LED_PIN, HIGH);

  // Initialize neo pixel library and turn off the LEDs
  pixels.begin();
  pixels.show();

  // Clear the input command buffer
  memset(commandBuffer, 0, sizeof(commandBuffer));

  // Initialize spectrum display
  spectrumSetup();

  // Begin sampling audio
  samplingBegin();
}

void loop() {
  // Calculate FFT if a full sample is available.
  if (samplingIsDone()) {
    // Run FFT on sample data.
    arm_cfft_radix4_instance_f32 fft_inst;
    arm_cfft_radix4_init_f32(&fft_inst, FFT_SIZE, 0, 1);
    arm_cfft_radix4_f32(&fft_inst, samples);
    // Calculate magnitude of complex numbers output by the FFT.
    arm_cmplx_mag_f32(samples, magnitudes, FFT_SIZE);

    if (LEDS_ENABLED == 1)
    {
      spectrumLoop();
    }

    // Restart audio sampling.
    samplingBegin();
  }
```

```
  // Parse any pending commands.
  parserLoop();
}


////////////////////////////////////////////////////////////////////////////
// UTILITY FUNCTIONS
////////////////////////////////////////////////////////////////////////////

// Compute the average magnitude of a target frequency window vs. all other frequencies.
void windowMean(float* magnitudes, int lowBin, int highBin, float* windowMean, float*
otherMean) {
    *windowMean = 0;
    *otherMean = 0;
    // Notice the first magnitude bin is skipped because it represents the
    // average power of the signal.
    for (int i = 1; i < FFT_SIZE/2; ++i) {
      if (i >= lowBin && i <= highBin) {
        *windowMean += magnitudes[i];
      }
      else {
        *otherMean += magnitudes[i];
      }
    }
    *windowMean /= (highBin - lowBin) + 1;
    *otherMean /= (FFT_SIZE / 2 - (highBin - lowBin));
}

// Convert a frequency to the appropriate FFT bin it will fall within.
int frequencyToBin(float frequency) {
  float binFrequency = float(SAMPLE_RATE_HZ) / float(FFT_SIZE);
  return int(frequency / binFrequency);
}

// Convert from HSV values (in floating point 0 to 1.0) to RGB colors usable
// by neo pixel functions.
uint32_t pixelHSVtoRGBColor(float hue, float saturation, float value) {
  // Implemented from algorithm at http://en.wikipedia.org/wiki/HSL_and_HSV#From_HSV
  float chroma = value * saturation;
  float h1 = float(hue)/60.0;
  float x = chroma*(1.0-fabs(fmod(h1, 2.0)-1.0));
  float r = 0;
  float g = 0;
```

```
    float b = 0;
    if (h1 < 1.0) {
        r = chroma;
        g = x;
    }
    else if (h1 < 2.0) {
        r = x;
        g = chroma;
    }
    else if (h1 < 3.0) {
        g = chroma;
        b = x;
    }
    else if (h1 < 4.0) {
        g = x;
        b = chroma;
    }
    else if (h1 < 5.0) {
        r = x;
        b = chroma;
    }
    else // h1 <= 6.0
    {
        r = chroma;
        b = x;
    }
    float m = value - chroma;
    r += m;
    g += m;
    b += m;
    return pixels.Color(int(255*r), int(255*g), int(255*b));
}


////////////////////////////////////////////////////////////////////////////////
// SPECTRUM DISPLAY FUNCTIONS
////////////////////////////////////////////////////////////////////////////////

void spectrumSetup() {
    // Set the frequency window values by evenly dividing the possible frequency
    // spectrum across the number of neo pixels.
    float windowSize = (SAMPLE_RATE_HZ / 2.0) / float(NEO_PIXEL_COUNT);
    for (int i = 0; i < NEO_PIXEL_COUNT+1; ++i) {
        frequencyWindow[i] = i*windowSize;
```

```
  }
  // Evenly spread hues across all pixels.
  for (int i = 0; i < NEO_PIXEL_COUNT; ++i) {
    hues[i] = 360.0*(float(i)/float(NEO_PIXEL_COUNT-1));
  }
}

void spectrumLoop() {
  // Update each LED based on the intensity of the audio
  // in the associated frequency window.
  float intensity, otherMean;
  for (int i = 0; i < NEO_PIXEL_COUNT; ++i) {
    windowMean(magnitudes,
               frequencyToBin(frequencyWindow[i]),
               frequencyToBin(frequencyWindow[i+1]),
               &intensity,
               &otherMean);
    // Convert intensity to decibels.
    intensity = 20.0*log10(intensity);
    // Scale the intensity and clamp between 0 and 1.0.
    intensity -= SPECTRUM_MIN_DB;
    intensity = intensity < 0.0 ? 0.0 : intensity;
    intensity /= (SPECTRUM_MAX_DB-SPECTRUM_MIN_DB);
    intensity = intensity > 1.0 ? 1.0 : intensity;
    pixels.setPixelColor(i, pixelHSVtoRGBColor(hues[i], 1.0, intensity));
  }
  pixels.show();
}


//////////////////////////////////////////////////////////////////////////////
// SAMPLING FUNCTIONS
//////////////////////////////////////////////////////////////////////////////

void samplingCallback() {
  // Read from the ADC and store the sample data
  samples[sampleCounter] = (float32_t)analogRead(AUDIO_INPUT_PIN);
  // Complex FFT functions require a coefficient for the imaginary part of the input.
  // Since we only have real data, set this coefficient to zero.
  samples[sampleCounter+1] = 0.0;
  // Update sample buffer position and stop after the buffer is filled
  sampleCounter += 2;
  if (sampleCounter >= FFT_SIZE*2) {
    samplingTimer.end();
```

```
    }
}

void samplingBegin() {
    // Reset sample buffer position and start callback at necessary rate.
    sampleCounter = 0;
    samplingTimer.begin(samplingCallback, 1000000/SAMPLE_RATE_HZ);
}

boolean samplingIsDone() {
    return sampleCounter >= FFT_SIZE*2;
}


////////////////////////////////////////////////////////////////////////////
// COMMAND PARSING FUNCTIONS
// These functions allow parsing simple commands input on the serial port.
// Commands allow reading and writing variables that control the device.
//
// All commands must end with a semicolon character.
//
// Example commands are:
// GET SAMPLE_RATE_HZ;
// - Get the sample rate of the device.
// SET SAMPLE_RATE_HZ 400;
// - Set the sample rate of the device to 400 hertz.
//
////////////////////////////////////////////////////////////////////////////

void parserLoop() {
    // Process any incoming characters from the serial port
    while (Serial.available() > 0) {
        char c = Serial.read();
        // Add any characters that aren't the end of a command (semicolon) to the input buffer.
        if (c != ';') {
            c = toupper(c);
            strncat(commandBuffer, &c, 1);
        }
        else
        {
            // Parse the command because an end of command token was encountered.
            parseCommand(commandBuffer);
            // Clear the input buffer
            memset(commandBuffer, 0, sizeof(commandBuffer));
```

```
    }
  }
}

// Macro used in parseCommand function to simplify parsing get and set commands for a variable
#define GET_AND_SET(variableName) \
  else if (strcmp(command, "GET " #variableName) == 0) { \
    Serial.println(variableName); \
  } \
  else if (strstr(command, "SET " #variableName " ") != NULL) { \
    variableName = (typeof(variableName)) atof(command+(sizeof("SET " #variableName " ")-1)); \
  }

void parseCommand(char* command) {
  if (strcmp(command, "GET MAGNITUDES") == 0) {
    for (int i = 0; i < FFT_SIZE; ++i) {
      Serial.println(magnitudes[i]);
    }
  }
  else if (strcmp(command, "GET SAMPLES") == 0) {
    for (int i = 0; i < FFT_SIZE*2; i+=2) {
      Serial.println(samples[i]);
    }
  }
  else if (strcmp(command, "GET FFT_SIZE") == 0) {
    Serial.println(FFT_SIZE);
  }
  GET_AND_SET(SAMPLE_RATE_HZ)
  GET_AND_SET(LEDS_ENABLED)
  GET_AND_SET(SPECTRUM_MIN_DB)
  GET_AND_SET(SPECTRUM_MAX_DB)

  // Update spectrum display values if sample rate was changed.
  if (strstr(command, "SET SAMPLE_RATE_HZ ") != NULL) {
    spectrumSetup();
  }

  // Turn off the LEDs if the state changed.
  if (LEDS_ENABLED == 0) {
    for (int i = 0; i < NEO_PIXEL_COUNT; ++i) {
      pixels.setPixelColor(i, 0);
    }
```

```
        pixels.show();
    }
}
```