Music Visualizer using FFT

By

Julian Tang – jt2932

Ameya Phadke – ap6310

Anuj Doshi – ad5092

Abstract: Using the Dynamic Pose-Sensitive Sound Design sensor, as developed in Project 1 for this course, a visualizer for the Audio thus designed was developed. It included an LED RBG Matrix used to visualize incoming audio signals based on frequency bands, as well as reacting dynamically to said frequency bands and signal intensity.

Keywords: FFT, Raspberry Pi, RBG LED Matrix, IMU Tracking

Index

1) Introduction:

This project was born with an idea to develop the Pose-Sensitive Sound Design Sensor developed as part of Project 1. While the sound is generated dynamically as part of the exhibit, it is important to include a visual component for the audience to realize and understand their interactions with the exhibit. The motion of the Sound Design Sensor brings in an element that keeps in line with the theme of the client's exhibit, and the audio frequency band visualizer is used to use this dynamic, unstable nature to represent various different tones and instruments. These LED subspaces react to such preconfigured parameters, including some light incoming light intensity, to provide a unique experience to visitors and audience of the exhibit.

2) Bill of Materials

Component	Quantity	Price
Raspberry Pi 3B+	1	$35.00
Adafruit RGB Matrix	1	$24.95
Adafruit 16x32 LED panel	1	$24.95

Additionally, all the components used to develop the Pose Sensitive Sound Design Sensor were also used.

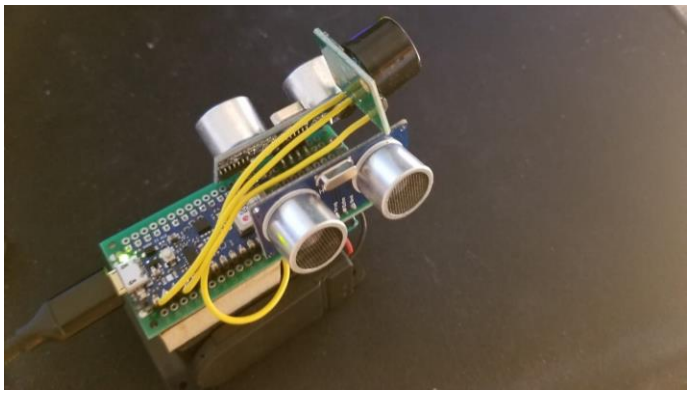


Figure 3: Position Sensor View

Component	Quantity
Arduino Nano 33 BLE	1
Maxbotix MB1010 LV-Maxsonar EZ1	1
Tower Pro SG-90	3
9v Battery	2

3) Working and Implementation

Fast Fourier Transform:
In complex notation, the time and frequency domains each contain one signal made up of N complex points. Each of these complex points is composed of two numbers, the real part and the imaginary part. For example, when we talk about complex sample X[42], it refers to the combination of ReX[42] and ImX[42]. In other words, each complex variable holds two numbers. When two complex variables are multiplied, the four individual components must be combined to form the two components of the product.

The FFT operates by decomposing an N point time domain signal into N time domain signals each composed of a single point. The second step is to calculate the N frequency spectra corresponding to these N time domain signals. Lastly, the N spectra are synthesized into a single frequency spectrum.

The following figure shows an example of the time domain decomposition used in the FFT. In this example, a 16-point signal is decomposed through four separate stages. The first stage breaks the 16-point signal into two signals each consisting of 8 points. The second stage decomposes the data into four signals of 4 points. This pattern continues until there are N signals composed of a single point. An interlaced decomposition is used each time a signal is broken in two, that is, the signal is separated into its even and odd numbered samples. There are Log2N stages required in this decomposition, i.e., a 16-point signal (24) requires 4 stages, a 512 point signal (27) requires 7 stages, a 4096 point signal (212) requires 12 stages, etc.

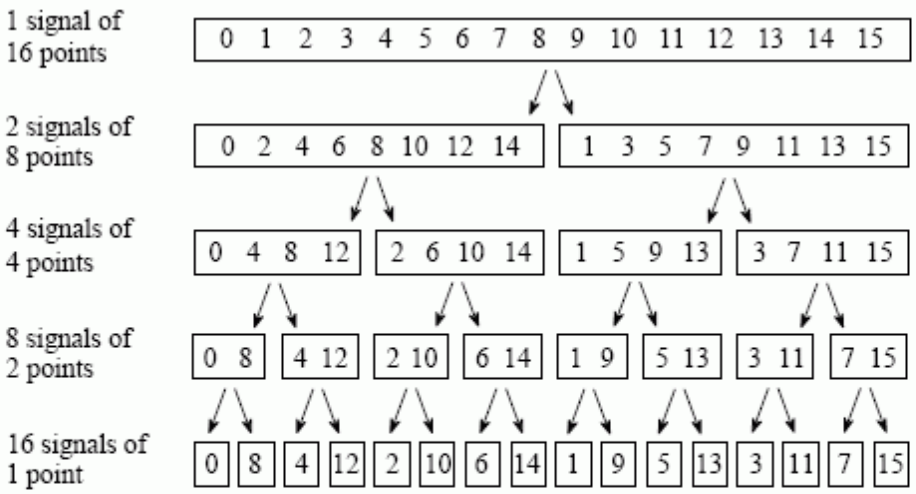
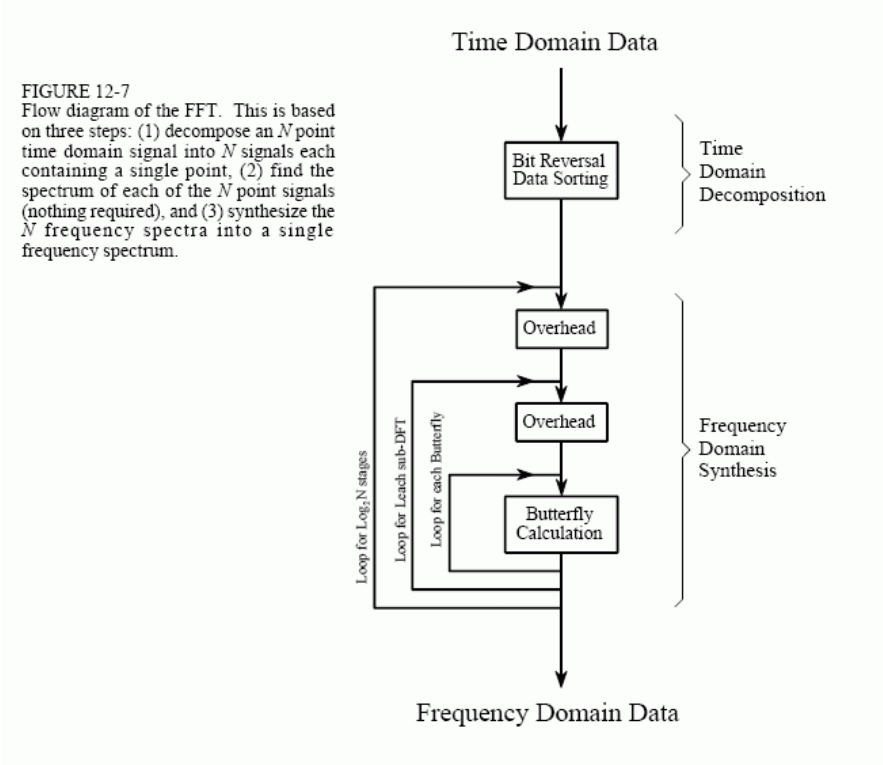


FIGURE 12-2
The FFT decomposition. An N point signal is decomposed into N signals each containing a single point. Each stage uses an *interlace decomposition*, separating the even and odd numbered samples.

The important idea is that the binary numbers are the reversals of each other. For example, sample 3 (0011) is exchanged with sample number 12 (1100). Likewise, sample number 14 (1110) is swapped with sample number 7 (0111), and so forth. The FFT time domain decomposition is usually carried out by a bit reversal sorting algorithm. This involves rearranging the order of the N time domain samples by counting in binary with the bits flipped left-for-right.

The next step in the FFT algorithm is to find the frequency spectra of the 1 point time domain signals. Nothing could be easier; the frequency spectrum of a 1 point signal is equal to itself. This means that nothing is required to do this step. Although there is no work involved, don't forget that each of the 1 point signals is now a frequency spectrum, and not a time domain signal.

The last step in the FFT is to combine the N frequency spectra in the exact reverse order that the time domain decomposition took place. Unfortunately, the bit reversal shortcut is not applicable, and we must go back one stage at a time. In the first stage, 16 frequency spectra (1 point each) are synthesized into 8 frequency spectra (2 points each). In the second stage, the 8 frequency spectra (2 points each) are synthesized into 4 frequency spectra (4 points each), and so on. The last stage results in the output of the FFT, a 16 point frequency spectrum.



Process Flow:

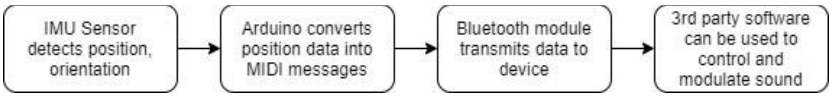


Figure 1: Sensor Process Flow

4) Results

Python Code:

```
import pyaudio
import numpy as np
import audioop
import sys
import math
import struct
from rgbmatrix import RGBMatrix, RGBMatrixOptions,graphics

em1=0.5
em2=0.2
```

```
em3=0.05
em4=0.5
em5=0.5
em6=0.8
em7=0.1
em8=0.1
em9=0.1
emc=0.04
Rprev=0
Gprev=0
Bprev=0
LL1=0
LL2=0
LL3=0
LL4=0
LL5=0
LL6=0
LL7=0
LL8=0
Lmeanprev=0
Mmeanprev=0
Hmeanprev=0

options=RGBMatrixOptions()
options.rows=16
options.cols=32
options.chain_length=1
options.parallel=1
options.hardware_mapping='adafruit-hat'
Dmatrix=RGBMatrix(options=options)
Dmatrix.Clear()
chunk=4096
scale=50
exponent=5
samplerate=44100
power = []



device=2
p=pyaudio.PyAudio()
stream=p.open(format =
pyaudio.paInt16,channels=1L,rate=44100,input=True,frames_per_buffer=chunk,input_device_in
dex=device)
```

```
msize=64
matrix = [0] * msize
def calculate_levels(data,chunk,samplerate):

    fmt="%dh"%(len(data)/2)
    data2=struct.unpack(fmt,data)
    data2=np.array(data2,dtype='h')
    fourier2=np.fft.fft(data2)+0.000001
    power=np.log10(np.abs(fourier2))**4
    power=np.reshape(power,(msize,msize),-1)
    matrix=np.int_(np.average(power,axis=0))
    return matrix

cyc=0
shif=1

while True:
    try:
        data = stream.read(chunk,exception_on_overflow = False)
        matrix=calculate_levels(data,chunk,samplerate)
        matrix=matrix[0:32]
        matrix-=6
        matrix/=24

        matrix-=5
        matrix[matrix<0]=0

        LL=matrix[1]
        LL-=6
        LL/=1
        LL**4
        LL=max(0,LL)

        LM=matrix[2]

        LH=matrix[3]


        Lmean=(LL+LM+LH)/3
        Lmeanema=Lmean*em3+(1-em3)*Lmeanprev
        Lmeanprev=Lmeanema

        LL1=em1*LL+(1-em6)*LL1
        LL2=em1*LL1+(1-em1)*LL2
```

```
LL3=em1*LL2+(1-em1)*LL3
LL4=em1*LL3+(1-em1)*LL4
LL5=em1*LL4+(1-em1)*LL5
LL6=em1*LL5+(1-em1)*LL6
LL7=em1*LL6+(1-em1)*LL7
LL8=em1*LL7+(1-em1)*LL8
#print(LL8)

ML=matrix[5]
ML-=3
ML**2
ML=max(ML,0)
MM=matrix[6]
MM-=3
MM**2
MM=max(MM,0)
MH=matrix[8]
MH-=3
MH**2
MH=max(MH,0)
Mmean=(ML+MM+MH)/3
Mmeanema=Mmean*em4+Mmeanprev
Mmeanprev=Mmeanema


H1=matrix[9]
H2=matrix[13]
H3=matrix[18]
H4=8*matrix[24]
H5=24*matrix[30]
Hmean=(H1+H2+H3+H4+H5)/5
Hmeanema=Hmean*em5+Hmeanprev
Hmeanprev=Hmeanema

B=Lmean/8
B=emc*B+(1-emc)*Bprev
Bprev=B
#print(B)
B=max(B,0)
B=min(B,1.2)
```

```
            R=1.4-B
            R=max(R,0)
            R=min(R,1.2)

            G=1.7-R-B
            G=max(G,0)
            G=min(G,1.2)
            B=B*0.8

            MB=Mmean+0.2*Mmeanema
            HB=Hmean+0.2*Hmeanema

            Mmatrixema=Mmeanema*np.ones((16, 6))
            Hmatrixema=Hmeanema*np.ones((16, 10))


            LL*=0.7

Lmatrix=np.array([[LL5,LL5*0.8,LL5*0.7,LL5*0.6,LL6*0.5,LL6*0.4,LL6*0.3,LL6*0.3,LL7*0.3,LL7*0.
3,LL7*0.2,LL7*0.2,LL8*0.2,LL8*0.2,LL8*0.1,LL8*0.1],
                    [LL5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

[LL4*0.4,LL4*0.4,LL3*0.9,LL3*0.9,LL2*0.9,LL2*0.9,LL1*0.9,LL1*0.9,LL*0.8,LL*0.9,LL,LL,LL,LL,LL,LL
],

[LL4*0.4,LL4*0.4,LL3*0.9,LL3*0.9,LL2*0.9,LL2*0.9,LL1*0.9,LL1*0.9,LL*0.8,LL*0.9,LL,LL,LL,LL,LL,LL
],

[LL4*0.4,LL4*0.4,LL3*0.9,LL3*0.9,LL2*0.9,LL2*0.9,LL1*0.9,LL1*0.9,LL*0.8,LL*0.9,LL,LL,LL,LL,LL,LL
],

[LL4*0.4,LL4*0.4,LL3*0.9,LL3*0.9,LL2*0.9,LL2*0.9,LL1*0.9,LL1*0.9,LL*0.8,LL*0.9,LL,LL,LL,LL,LL,LL
],

[LL4*0.4,LL4*0.4,LL3*0.9,LL3*0.9,LL2*0.9,LL2*0.9,LL1*0.9,LL1*0.9,LL*0.8,LL*0.9,LL,0,0,0,0,0],

[LL4*0.4,LL4*0.4,LL3*0.9,LL3*0.9,LL2*0.9,LL2*0.9,LL1*0.9,LL1*0.9,LL*0.8,0,0,0,0,0,0,0],
                    [LL4*0.4,LL4*0.4,LL3*0.9,LL3*0.9,LL2*0.9,LL2*0.6,0,0,0,0,0,0,0,0,0,0],
                    [LL4*0.4,LL4*0.4,LL3*0.9,LL3*0.9,0,0,0,0,0,0,0,0,0,0,0,0],
                    [LL4*0.4,LL4*0.4,LL3*0.6,0,0,0,0,0,0,0,0,0,0,0,0,0],
                    [LL4*0.4,LL4*0.4,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                    [LL4*0.4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                    [LL4*0.4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
```

```
[LL5,LL5*0.8,LL5*0.7,LL5*0.6,LL6*0.5,LL6*0.4,LL6*0.3,LL6*0.3,0,0,0,0,0,0,0,0],

[LL5,LL5*0.8,LL5*0.7,LL5*0.6,LL6*0.5,LL6*0.4,LL6*0.3,LL6*0.3,LL7*0.3,LL7*0.3,LL7*0.2,LL7*0.2,
LL8*0.2,LL8*0.2,LL8*0.1,LL8*0.1]])

        Mmatrix=np.array([[0,0,0,0,0,0],
                [ML*0.1,ML*0.1,0,0,0,0],
                [ML*0.4,ML*0.4,0,0,0,0],
                [ML*0.8,ML*0.8,0,0,0,0],
                [ML,ML,0,0,0,0],
                [ML,ML,0,0,0,0],
                [ML*0.8,ML*0.8,MM*0.2,MM*0.2,0,0],
                [ML*0.4,ML*0.4,MM*0.4,MM*0.4,0,0],
                [ML*0.1,ML*0.1,MM*0.8,MM*0.8,0,0],
                [0,0,MM,MM,MH*0.2,MM*0.2],
                [0,0,MM,MM,MH*0.4,MM*0.4],
                [0,0,MM*0.8,MM*0.8,MH*0.8,MM*0.8],
                [0,0,MM*0.4,MM*0.4,MH,MH],
                [0,0,MM*0.2,MM*0.2,MH,MH],
                [0,0,0,0,MH*0.3,MM*0.3],
                [0,0,0,0,MH*0.1,MM*0.1],])


        Hmatrix=np.array([[H5*0.8,H5,H5,H5*0.8,H5*0.6,H5*0.4,H5*0.2,H5*0.1,0,0],
                [0,0,0,0,0,0,H4*0.5,H4,H4,H4*0.5],
                [0,0,0,0,0,0,H4*0.5,H4,H4,H4*0.5],
                [0,0,H2*0.5,H2,H2,H2*0.5,0,0,0,0],
                [0,0,0,0,0,0,H3*0.7,H3,H3,H3*0.7],
                [H1,H1*0.8,0,0,0,0,H3*0.7,H3,H3,H3*0.7],
                [H1,H1*0.8,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0],
                [H1,H1*0.8,0,0,0,0,0,0,0,0],
                [H1,H1*0.8,0,0,0,0,H3*0.7,H3,H3,H3*0.7],
                [0,0,0,0,0,0,H3*0.7,H3,H3,H3*0.7],
                [0,0,H2*0.5,H2,H2,H2*0.5,0,0,0,0],
                [0,0,0,0,0,0,H4*0.5,H4,H4,H4*0.5],
                [0,0,0,0,0,0,H4*0.5,H4,H4,H4*0.5],
                [H5*0.8,H5,H5,H5*0.8,H5*0.6,H5*0.4,H5*0.2,H5*0.1,0,0]])

        Fmatrix=np.hstack((Lmatrix,Mmatrix,Hmatrix))

        Dmatrix.Clear()
```

```
            for y in range(0,32):
                for x in range(0,16):
                    I=20*Fmatrix[x][y]
                    Rint=int(round(R*I))
                    Rint=min(242,Rint)
                    Rint=max(0,Rint)
                    Gint=int(round(G*I))
                    Gint=min(242,Gint)
                    Gint=max(0,Gint)
                    Bint=int(round(B*I))
                    Bint=min(242,Bint)
                    Bint=max(0,Bint)

                    Dmatrix.SetPixel(y,x,Rint,Gint,Bint)

        except ZeroDivisionError as err:
            print('Handling run-time error:', err)
```

Arduino Code:

```
#include <ArduinoBLE.h>
#include <Arduino_LSM9DS1.h>

byte midiData[] = {0x80, 0x80, 0x00, 0x00, 0x00};


BLEService midiService("03B80E5A-EDE8-4B33-A751-6CE34EC4C700");
BLECharacteristic midiCharacteristic("7772E5DB-3868-4112-A1A9-F2669D106BF3",
                        BLEWrite | BLEWriteWithoutResponse |
                        BLENotify | BLERead, sizeof(midiData));



///EMA///
float emaalpha=0.5;
///EMA///

///US///
const int trigPin =9;
const int echoPin =10;
float duration, distance;
const int trigPin2 =5;
const int echoPin2 =6;
float duration2, distance2;
///US///
```

```
//Interpretation//
int mpdown;
int mpup;
int mrright;
int mrleft;
int myaw;
int mus1;
int mus2;
int mpd;
int mgx;
int mgy;
int mgz;
int macx;
int macy;
int macz;

//Interpretation//

 int marray[14]= {mpdown, mpup, mrright, mrleft, myaw, mus1, mus2, mpd, mgx, mgy, mgz,
macx, macy, macz};
 int channel[14]={69,70,71,72,73,75,76,77,78,79,80,81,82,83};
 int lastmarray[14]={0};
 int lastmarray2[14]={0};



void setup() {
 // initialize serial communication
 Serial.begin(9600);
 pinMode(LED_BUILTIN, OUTPUT);
 if (!BLE.begin()) {
  Serial.println("starting BLE failed!");
  while (true);
 }
 BLE.setLocalName("N33_BLE");
 BLE.setAdvertisedService(midiService);
 midiService.addCharacteristic(midiCharacteristic);
 BLE.addService(midiService);
 BLE.advertise();

  ///US///
 pinMode(trigPin, OUTPUT);
 pinMode(echoPin, INPUT);
```

```
  pinMode(trigPin2, OUTPUT);
  pinMode(echoPin2, INPUT);
  ///US///

  IMU.begin();

}

void loop() {
  // wait for a BLE central
  BLEDevice central = BLE.central();


  if (central) {

    digitalWrite(LED_BUILTIN, HIGH);

  /////// AUS1 /////////

  distance = analogRead(A0);

  ///////  IR1  /////////



  distance2 = 127*digitalRead(12);

  float x, y, z;      //accelerometer
  float mx, my, mz;   //magnetometer
  float gx, gy, gz;   //gyroscope
    IMU.readAcceleration(x, y, z);
    IMU.readMagneticField(mx,my,mz);
    IMU.readGyroscope(gx, gy, gz);
    double xb = float(x);
    double yb = float(y);
    double zb = float(z);
    double roll = atan2(yb,zb)*57.3;
    double pitch = atan2((-xb),sqrt(yb*yb+zb*zb))*57.3;
    //MAGNETOMETER IS TOO SENSITIVE TO MY DEVICES, FLOORED
    double yaw = 0;

  ///////// Sensor2MIDI interpretation //////////
  mpdown=Cmap(pitch,10 ,80 ,0);
  mpup=Cmap(pitch,-10 ,-80 ,0);
```

```
mrright=Cmap(roll,10 ,80 ,0);
mrleft=Cmap(roll,-10 ,-80 ,0);
myaw=Cmap(yaw,0 ,360 ,0);
mus1=Cmap(distance,20 ,100 ,0);
mus2=Cmap(distance2,1 ,16 ,0);
delay(50);
mpd=0;
mgx=Cmap(gx,0 ,25 ,0);
mgy=Cmap(gy,0 ,25 ,0);
mgz=Cmap(gz,0 ,25 ,0);
mpd=0;
macx=Cmap(xb,0 ,1 ,0);
macy=Cmap(yb,0 ,1 ,0);
macz=Cmap(zb,0 ,1 ,0);
delay(50);
marray[0]= {mpdown};
marray[1]= {mpup};
marray[2]= {mrright};
marray[3]= {mrleft};
marray[4]= {myaw};
marray[5]= {mus1};
marray[6]= {mus2};
delay(12);
marray[7]= {mgx};
marray[8]= {mgy};
marray[9]= {mgz};
marray[10]= {mpd};
delay(12);
marray[11]= {macx};
marray[12]= {macy};
marray[13]= {0};


///EMA///
for (int i=0; i<14; i++) {
  marray[i]=(emaalpha*marray[i]+(1-emaalpha)*lastmarray[i]);
  lastmarray[i]=marray[i];
  delay(1);
}
///EMA///
///////// SendMIDI, only on significant change from last/////////
for (int i=0; i<8; i++){
if(abs(marray[i]-lastmarray2[i])>6){
midiCommand(0xB0,channel[i],marray[i]);
```

```
 lastmarray2[i]=marray[i];
 delay(1);
 }
 }
  for (int i=8; i<11; i++){
 if(abs(marray[i]-lastmarray2[i])>24){
 midiCommand(0xB0,channel[i],marray[i]);
 lastmarray2[i]=marray[i];
 delay(1);
 }
 }
 for (int i=11; i<14; i++){
 if(abs(marray[i]-lastmarray2[i])>6){
 midiCommand(0xB0,channel[i],marray[i]);
 lastmarray2[i]=marray[i];
 delay(1);
 }
 }
 ///////// SendMIDI, only on significant change from last/////////

 }

 digitalWrite(LED_BUILTIN, LOW);
}


void midiCommand(byte cmd, byte data1, byte  data2) {

 midiData[2] = cmd;
 midiData[3] = data1;
 midiData[4] = data2;

 midiCharacteristic.setValue(midiData, sizeof(midiData));
}

int Cmap(double input,float minin,float maxin,int invert){
int output;

if(invert==1){
input = constrain(input,minin,maxin);
output = map(input,minin,maxin,127,0);
}
else{
input = constrain(input,minin,maxin);
```

```
output = map(input,minin,maxin,0,127);
}
return output;
}
/////// DUAL SIDE US SENSORS REPLACED BY IR SENSOR, REDUCING SIZE AND INCREASES
RESPONSE RATE BY >20x /////////
//float PulseIn2(const byte pin, const byte state, const unsigned long timeout = 1000000L);
//
//
//#define WAIT_FOR_PIN_STATE(state) \
//  while (digitalRead(pin) != (state)) { \
//    if (micros() - timestamp > timeout) { \
//      return 0; \
//    } \
//  }
//
//float PulseIn2(const byte pin, const byte state, const unsigned long timeout) {
//  unsigned long timestamp = micros();
//  WAIT_FOR_PIN_STATE(!state);
//  WAIT_FOR_PIN_STATE(state);
//  timestamp = micros();
//  WAIT_FOR_PIN_STATE(!state);
//  return micros() - timestamp;
//}
```