Formal Report 2
Special System

Caleb Markley
apollo (Xylophone-Playing Robot)

EEL5666: IMDL
Instructors: Dr. A. Antonia Arroyo, Dr. Eric M. Schwartz
TAs: Andy Gray, Josh Weaver, Nick Cox

Table of Contents

1. Abstract

I am building a xylophone-playing robot that comes up with its own melodies. The special system for my robot is the algorithmic melody generation algorithm. The system consists of a xylophone, 12 small solenoids that will be used to play the xylophone, a Sharp IR sensor to detect the proximity of onlookers, an Arduino Due to drive the solenoids and read the IR sensor, a laptop with a wired connection to the Arduino, a webcam built in to the laptop for color detection, and a MIDI keyboard connected to the laptop through USB. From a software perspective my system consists of a Markov model that I weight using the sensor info along with a key and tempo selected using the MIDI keyboard.

2. Introduction

Inspiration is a powerful resource in art, including in music. Some great creative works have been written by building on the work of others. This composition by inspiration was the motivation for my special system. My robot needed an autonomous system for creating and playing melodies inspired by great songwriters and influenced in real-time by the robot's surroundings. I decided to pursue using a Markov chain with songs from popular music with great melodies forming the probabilities used by the model. I also use a MIDI keyboard, and IR distance sensor, and a webcam to allow the robot to react to its surroundings and alter the model. In this paper I will discuss the sensors used, the theory behind my algorithm, my development process, and the results I have so far. I've also included my code for reference.

3. Theory

My special system uses Markov chains to algorithmically compose melodies. Markov chains take an input state and compare it to a chain that gives a set of potential next states in the chain. Each potential next state has a different probability determined by a learning set. Figure 1 shows an example of a very simple Markov chain from [1]. For the chain given in Figure 1, given state A the probability that the next state will be state E is 0.4 while the probability that the next state will be state A again is 0.6. While this is a very simple example, one can

see how this kind of model could be applied to generate probabilities for which notes will follow specific notes given a good set of initial data.

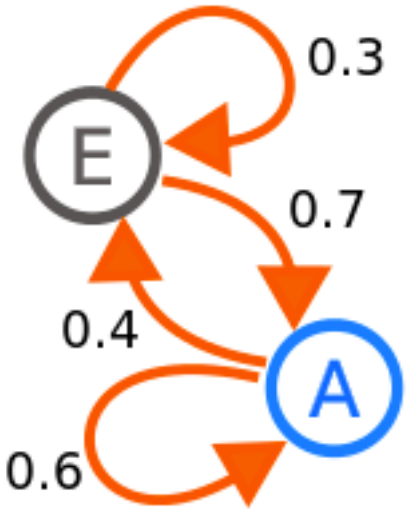

Figure 1, a simple Markov chain from [1].

While my prototype uses only the song "Twinkle Twinkle Little Star" as a learning set, the learning set will eventually be developed from the works of great melody-writers. This system would form the basis for the melody generation. I also plan to test a more complex model that is still based on a Markov chain. Instead of each node of the chain being made up of single notes each node will be made up of note phrases to hopefully make the generated melody seem more human and melodic. I could also potentially impose a verse-chorus structure, as most popular songs have, in order to make the composition seem more natural or human-like as well.

4. Sensors

While the majority of the complexity of this system is software-based, there are a few sensors that are necessary. The sensors used for this robot are a Sharp long-range IR sensor and a webcam built into my laptop. Each sensor influences a different aspect of the system. To understand what the sensors are doing one must first understand what the MIDI keyboard input does.

The MIDI keyboard is used to set the key of the composition as well as the initial tempo. Simplistically speaking the tempo of a piece of music is how fast it is and the key determines what notes fit in the piece well. By pressing a key on the MIDI keyboard 8 times the key is set to note played and the initial tempo is set to the average time between key presses. The sensors start with this initial framework and dynamically changes it based on the robot's environment.

The Sharp IR distance sensor is used to detect how close spectators are to the robot. The closer spectators get, the "faster" the robot will play, which is accomplished by

increasing or decreasing the tempo. The listed range for the sensor is 20 to 150cm, which is ideal for detecting how close onlookers get to the robot. Figure 2 shows the IR sensor values for different distances. The initial tempo is unaltered at the median value recorded by the sensor during testing. When the something is closer than that distance to the sensor the tempo will increase and when something is farther away the tempo is decreased. The datasheet for the sensor is given in [2].

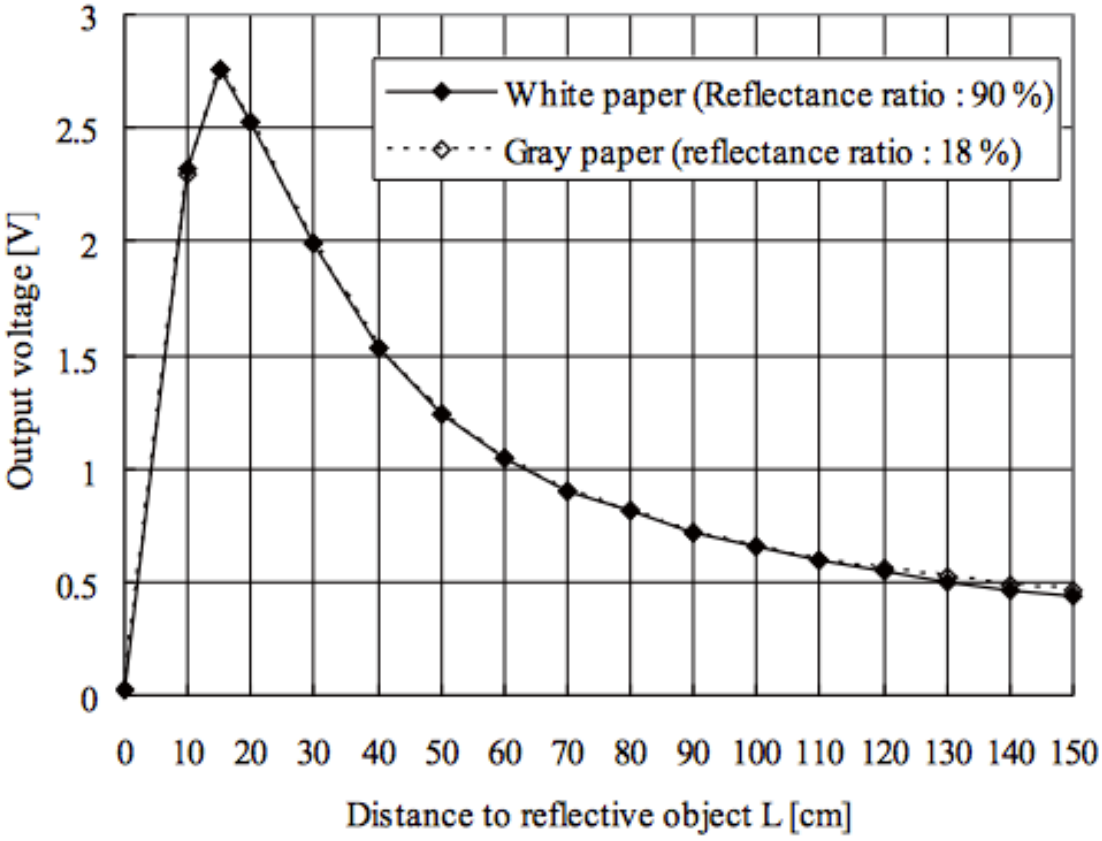

Figure 2, a graph of the IR sensor output voltage versus the distance from an object with a 3V reference as shown in [2].

The webcam is the standard iSight webcam built into a mid-2009 MacBook Pro. This is used to find the primary color in the room the robot is in. I currently determine if the primary color is red, blue, green, or white/none. The color of the robot's environment weights the probabilities of note durations generated by the Markov chains. While the details of how Markov chains work will be discussed in the next section, the color essentially influences if longer notes (like whole notes and half notes) and shorter notes (like eighth notes and sixteenth notes) will be more or less likely than they normally are at a relevant point in the Markov chain.

5. Operation

The system starts with pressing a key on the MIDI keyboard 8 times as previously mentioned. This sets the key and the initial tempo. Using the learning set as a Markov chain notes are chosen sequentially. When each note is about to be played the sensors are read. The distance read in from the IR sensor increases or decreases the initial tempo when an object is closer to the sensor or farther from the sensor, respectively. The main color detected by the iSight webcam weights the note duration probabilities as explained in the previous section. Once the sensor readings have been dealt with the note is played and the system looks at the node for the note played to start the chain process once again.

6. Conclusion

The special system works well. It is able to compose a melody that is clearly influenced by both sensors. However, there are a few bugs to work out and some improvements that can be made. For some reason the first four notes of the composition being the same every time, which needs to be fixed. There is also a timing bug when stopping the algorithm and starting it again without resetting the program that causes the tempo to be 10 times too fast. I would also like to make the timing between notes more accurate.

As mentioned in the previous sections, I would like to experiment with more complex algorithms to see if I can create more human-sounding melodies. I can try using note phrases instead of single notes as nodes in the Markov chain and imposing a verse-chorus song structure. I also plan to have a much larger list of songs as the learning set. With improvements like these I think the already nearly satisfactory performance could be even better.

7. References

[1] http://en.wikipedia.org/wiki/File:Markovkate_01.svg
[2] https://www.sparkfun.com/datasheets/Sensors/Infrared/gp2y0a02yk_e.pdf

8. Code

main.cpp

```
#include "ofMain.h"
#include "testApp.h"

//========================================================================
int main( ){

    ofSetupOpenGL(640,480, OF_WINDOW);          // <--------
```

```
setup the GL context

    // this kicks off the running of my app
    // can be OF_WINDOW or OF_FULLSCREEN
    // pass in width and height too:
    ofRunApp( new testApp());

}
```

testApp.h

```
#pragma once

#include "ofMain.h"
#include "ofEvents.h"
#include <sys/time.h>
#include <random.h>

#include "ofxOpenCv.h"
#include "ofxMidi.h"

#define C   0
#define Cs  1
#define D   2
#define Ds  3
#define E   4
#define F   5
#define Fs  6
#define G   7
#define Gs  8
#define A   9
#define As  10
#define B   11
#define REST 12

#define WN  0
#define DH  1
#define HN  2
#define DQ  3
#define QN  4
#define DE  5
#define EN  6
#define DS  7
#define SN  8
#define TS  9

#define HIT 15000
```

```
class testApp : public ofBaseApp, public ofxMidiListener
{

public:

    void setup();
    void update();
    void draw();
    void exit();
    void newMidiMessage(ofxMidiMessage& eventArgs);
    void color();
    void tempo();
    void setupMarkov();
    void markov();
    void transpose();
    int  findWeight(vector<int> chain, int rnd);
    void weightTempo();


    ofArduino           ard;
//arduino varaible
    bool                bSetupArduino;
//flag variable for setting up arduino once

    ofVideoGrabber      vidGrabber;
//needed to get camera image
    ofxCvColorImage     colorImg;
//used to display camera image
    int                 h,w;
//height and width of the camera
    int                 r,g,b,color_count;
//used for rgb calculations
    string              main_color = "White/None";
//the main color captured by the camera

    int                 distance_val = 0;
//analog in from IR sensor

    clock_t             last_time = 0;
//last saved time
    clock_t             now_time;
//current time
    clock_t             test_time = 0;
//time for testing purposes
    float               time_diff;
//difference between current time and last saved time
    float               rest;
//us between quarter notes before subtracting HIT time
    int                 tempo_ind = 0;
//index for tempo array
```

```
    float               tempo_arr [8];
//array used to determine tempo (rest values stored here)

    int                 key;
//key of composition (C=0,C#=1,...)


    vector< vector< int > > markov_note;            //an array to
build a markov chain; 12 + (none) possible previous notes
    int                 prev_note = 12;
//the previous note selected
    int                 curr_note;
//the current selected note
    vector< vector< int > > markov_del;             //an array to
build a markov chain; 10 + (none) possible previous delays
    int                 prev_del = 10;
//the previous delay selected
    int                 curr_del;
//the current delay selected

    int                 del_lengths[10];
//length of each delay (whole=0,dotted_half=1,...)
    int                 w_del_len[10];
//weighted delay lengths
    int                 past_note_del;
//

    int         whole,half,quarter;
//note delay values
    int         eighth,sixteenth,thirtysecond;
//note delay values
    int         dotted_half,dotted_quarter;
//note delay values
    int         dotted_eighth,dotted_sixteenth;
//note delay values

    string notes[12]  =
{"C","C#","D","D#","E","F","F#","G","G#","A","A#","B"};
    string delays[10] = {"WHOLE", "DOT HALF", "HALF", "DOT
QUARTER", "QUARTER", "DOT 8TH", "8TH", "DOT 16TH", "16TH",
"32ND"};

    ofxMidiIn midiIn;
    ofxMidiMessage midiMessage;

    int note;

private:

    void setupArduino(const int & version);
```

```
    void digitalPinChanged(const int & pinNum);
    void analogPinChanged(const int & pinNum);
    void updateArduino();
};
```

testApp.cpp

```
/*
 * Arduino and OpenCV
 */

#include "testApp.h"


//--------------------------------------------------------------
void testApp::setup()
{
    // add testApp as a listener
    midiIn.addListener(this);

    //set up serial communication (port,baud rate)
    ard.connect("/dev/tty.usbmodem411", 57600);

    //set width and height of camera image
    w = 640;
    h = 480;

    //rgb sums of image and number of pixels contributing to that
sum
    r = 0;
    g = 0;
    b = 0;
    color_count = 0;

    //set up video grabber for camera
    vidGrabber.setVerbose(true);
    vidGrabber.initGrabber(w,h);

    //allocate for displaying camera image
    colorImg.allocate(w,h);

    midiIn.listPorts();
    midiIn.openPort(0);
    midiIn.ignoreTypes(false, false, false);
    midiIn.setVerbose(true);

    // listen for EInitialized notification. this indicates that
```

```
    // the arduino is ready to receive commands and it is safe to
    // call setupArduino()
    ofAddListener(ard.EInitialized, this,
&testApp::setupArduino);
    bSetupArduino     = false;   // flag so we setup arduino when
its ready, you don't need to touch this

    setupMarkov();
    srand(time(NULL));

    while(tempo_ind < 7) { ard.update(); }
}

//--------------------------------------------------------------
void testApp::update()
{
    if (tempo_ind < 7)
    {
        ard.update();
    }
    else
    {
        //update Arduino
        updateArduino();

        //determine color (main_color updated to "Red", "Green",
"Blue", or "White/None")
        color();

        //display IR reading
        distance_val = ard.getAnalog(0);
        cout << "Distance Value = " << distance_val << endl;

        markov();

        weightTempo();

        transpose();

        clock_t time_now = clock();

        //TODO: Fix Timing
        //TODO: Fix First 4 Notes

        //cout << "Time: " << time_now - test_time << endl;
        //cout << endl;
        cout << "Note: " << notes[curr_note] << endl;
        cout << "Length: " << delays[curr_del] << endl;
        cout << endl;
```

```
            test_time = time_now;

            ard.sendDigital((2 + curr_note),1);
            usleep(HIT);
            ard.sendDigital((2 + curr_note),0);
            usleep(w_del_len[curr_del]);
        }
    }

    //--------------------------------------------------------------
    void testApp::setupArduino(const int & version)
    {

        // remove listener because we don't need it anymore
        ofRemoveListener(ard.EInitialized, this,
    &testApp::setupArduino);

        // it is now safe to send commands to the Arduino
        bSetupArduino = true;

        // print firmware name and version to the console
        ofLogNotice() << ard.getFirmwareName();
        ofLogNotice() << "firmata v" << ard.getMajorFirmwareVersion()
    << "." << ard.getMinorFirmwareVersion();

        // set 2-13 for solenoids out
        ard.sendDigitalPinMode(2, ARD_OUTPUT);
        ard.sendDigitalPinMode(3, ARD_OUTPUT);
        ard.sendDigitalPinMode(4, ARD_OUTPUT);
        ard.sendDigitalPinMode(5, ARD_OUTPUT);
        ard.sendDigitalPinMode(6, ARD_OUTPUT);
        ard.sendDigitalPinMode(7, ARD_OUTPUT);
        ard.sendDigitalPinMode(8, ARD_OUTPUT);
        ard.sendDigitalPinMode(9, ARD_OUTPUT);
        ard.sendDigitalPinMode(10, ARD_OUTPUT);
        ard.sendDigitalPinMode(11, ARD_OUTPUT);
        ard.sendDigitalPinMode(12, ARD_OUTPUT);
        ard.sendDigitalPinMode(13, ARD_OUTPUT);

        //set A0 for IR sensor in
        ard.sendAnalogPinReporting(0, ARD_ANALOG);

        //initialize all solenoids to 0
        ard.sendDigital((2),0);
        ard.sendDigital((3),0);
        ard.sendDigital((4),0);
        ard.sendDigital((5),0);
        ard.sendDigital((6),0);
        ard.sendDigital((7),0);
        ard.sendDigital((8),0);
```

```
    ard.sendDigital((9),0);
    ard.sendDigital((10),0);
    ard.sendDigital((11),0);
    ard.sendDigital((12),0);
    ard.sendDigital((13),0);

    // Listen for changes on the digital and analog pins
    ofAddListener(ard.EDigitalPinChanged, this,
&testApp::digitalPinChanged);
    ofAddListener(ard.EAnalogPinChanged, this,
&testApp::analogPinChanged);
}

//--------------------------------------------------------------
void testApp::updateArduino()
{

    // update the arduino, get any data or messages.
    // the call to ard.update() is required
    ard.update();

    // do not send anything until the arduino has been set up
    if (bSetupArduino)
    {
        //ard.sendDigital(13,1);
        //sleep(1);
        //ard.sendDigital(13,0);
        //sleep(1);
    }

}

// digital pin event handler, called whenever a digital pin value
has changed
// note: if an analog pin has been set as a digital pin, it will
be handled
// by the digitalPinChanged function rather than the
analogPinChanged function.

//--------------------------------------------------------------
void testApp::digitalPinChanged(const int & pinNum)
{
    // do something with the digital input. here we're simply
going to print the pin number and
    // value to the screen each time it changes
    //buttonState = "digital pin: " + ofToString(pinNum) + " = "
+ ofToString(ard.getDigital(pinNum));
}

// analog pin event handler, called whenever an analog pin value
```

```
has changed

//--------------------------------------------------------------
void testApp::analogPinChanged(const int & pinNum)
{
    // do something with the analog input. here we're simply
going to print the pin number and
    // value to the screen each time it changes
    //potValue = "analog pin: " + ofToString(pinNum) + " = " +
ofToString(ard.getAnalog(pinNum));
}


//--------------------------------------------------------------
void testApp::draw()
{
    // draw the incoming, the grayscale, the bg and the
thresholded difference
    ofSetHexColor(0xffffff);
    colorImg.draw(0,0);

    ofSetColor(0);
    }

//--------------------------------------------------------------
void testApp::exit ()
{
    // clean up
    midiIn.closePort();
    midiIn.removeListener(this);
}

//--------------------------------------------------------------
void testApp::newMidiMessage(ofxMidiMessage& msg)
{
    midiMessage = msg;
    note = midiMessage.pitch % 12;

    if (midiMessage.status == 144)
    {
        if (tempo_ind < 7)
        {
            cout << "MIDI Pitch: " << note << endl;
            key = note;

            //calculate time between notes
            if (last_time == 0)
            {
                last_time = clock();
            }
```

```
            else
            {
                now_time = clock();
                last_time = now_time - last_time;
//overwriting last_time to use as a placeholder
                time_diff = ((float) last_time) / CLOCKS_PER_SEC;
                rest = time_diff * 1000000;
                last_time = now_time;
                tempo_arr[tempo_ind] = rest;
                tempo_ind++;

                cout << "Rest (ms): " << (rest / 1000) << endl;
            }

            if (tempo_ind == 7) { tempo(); }
        }
        else //reset
        {
            tempo_ind = 0;
            last_time = 0;
        }
    }
}

//--------------------------------------------------------------
void testApp::color()
{
    //background for displaying frame
    //ofBackground(100,100,100);

    //get frame
    bool bNewFrame = false;
    vidGrabber.update();
    bNewFrame = vidGrabber.isFrameNew();

    if (bNewFrame)
    {

        for (int y=0; y<h; y+=10)
        {
            for (int x=0; x<w; x+=10)
            {

                //get pixel color and add to rgb sums
                color_count++;
                int i = (y*w+x)*3;
                r += vidGrabber.getPixels()[i+0];
                g += vidGrabber.getPixels()[i+1];
                b += vidGrabber.getPixels()[i+2];
            }
```

```
        }

        //used to display camera image
        //colorImg.setFromPixels(vidGrabber.getPixels(), w,h);

        //get color averages across camera image
        r = r / color_count;
        g = g / color_count;
        b = b / color_count;
        //cout << "R = " << r << ", G = " << g << ", B = " << b
<< endl;

        //determine main color
        if (r > 100 && g < 100) { main_color = "Red"; }
        else if (g > 100 && r < 100) { main_color = "Green"; }
        else if (b > 100 && r < 100) { main_color = "Blue"; }
        else { main_color = "White/None"; }
        cout << "Main Color = " << main_color << endl;

        //reset color variables
        r = 0;
        g = 0;
        b = 0;
        color_count = 0;
    }
}

//--------------------------------------------------------------
void testApp::tempo()
{
    //find average rest value for quarter note
    rest = 0;
    for (int i = 0; i < 7; i++) { rest += tempo_arr[i]; }
    rest = rest / 7;

    //set the rest values for each note in us
    whole = (rest * 4) - HIT;
    dotted_half = (rest * 3) - HIT;
    half = (rest * 2) - HIT;
    dotted_quarter = (rest * 1.5) - HIT;
    quarter = rest - HIT;
    dotted_eighth = (rest * 0.75) - HIT;
    eighth = (rest * 0.5) - HIT;
    dotted_sixteenth = (rest * 0.375) - HIT;
    sixteenth = (rest * 0.25) - HIT;
    thirtysecond = (rest * 0.125) - HIT;

    del_lengths[0] = whole;
    del_lengths[1] = dotted_half;
    del_lengths[2] = half;
```

```
    del_lengths[3] = dotted_quarter;
    del_lengths[4] = quarter;
    del_lengths[5] = dotted_eighth;
    del_lengths[6] = eighth;
    del_lengths[7] = dotted_sixteenth;
    del_lengths[8] = sixteenth;
    del_lengths[9] = thirtysecond;

    memcpy(w_del_len, del_lengths, sizeof(del_lengths));

    cout << "Rest Avg: " << rest << endl;
}

//--------------------------------------------------------------
void testApp::setupMarkov()
{
    int i;

    //initialize rows for note vector
    for (i = 0; i < 13; i++)
    {
        vector<int> new_row;
        markov_note.push_back(new_row);
    }

    //initialize rows for delay vector
    for (i = 0; i < 11; i++)
    {
        vector<int> new_row;
        markov_del.push_back(new_row);
    }

    int c_arr[12] = {2,0,0,0,0,0,0,3,0,0,0,0};
    int d_arr[12] = {2,0,2,0,1,0,0,1,0,0,0,0};
    int e_arr[12] = {1,0,4,0,4,1,0,0,0,0,0,0};
    int f_arr[12] = {0,0,0,0,5,4,0,0,0,0,0,0};
    int g_arr[12] = {0,0,0,0,0,4,0,4,0,2,0,0};
    int a_arr[12] = {0,0,0,0,0,0,0,2,0,2,0,0};
    int n_arr[12] = {1,0,0,0,0,0,0,0,0,0,0,0};

    int half_arr[10] = {0,0,0,0,4,0,0,0,0,0};
    int quarter_arr[10] = {0,0,4,0,30,0,2,0,0,0};
    int eighth_arr[10] = {0,0,1,0,1,0,2,0,0,0};
    int nd_arr[10] = {0,0,0,0,1,0,0,0,0,0};

    markov_note[C].insert(markov_note[C].begin(),c_arr,c_arr+12);
    markov_note[D].insert(markov_note[D].begin(),d_arr,d_arr+12);
    markov_note[E].insert(markov_note[E].begin(),e_arr,e_arr+12);
    markov_note[F].insert(markov_note[F].begin(),f_arr,f_arr+12);
    markov_note[G].insert(markov_note[G].begin(),g_arr,g_arr+12);
```

```
    markov_note[A].insert(markov_note[A].begin(),a_arr,a_arr+12);

markov_note[12].insert(markov_note[12].begin(),n_arr,n_arr+12);


markov_del[HN].insert(markov_del[HN].begin(),half_arr,half_arr+10
);

markov_del[QN].insert(markov_del[QN].begin(),quarter_arr,quarter_
arr+10);

markov_del[EN].insert(markov_del[EN].begin(),eighth_arr,eighth_ar
r+10);

markov_del[10].insert(markov_del[10].begin(),nd_arr,nd_arr+10);

}

//-------------------------------------------------------------
void testApp::markov()
{
    int total_weight = 0;
    float weight;
    float slow_weight = 1.0;
    float fast_weight = 1.0;
    for(int i = 0; i < markov_note[prev_note].size(); i++) {
total_weight += markov_note[prev_note][i]; }
    int rand_n = random() % total_weight;
    curr_note = findWeight(markov_note[prev_note], rand_n);
    prev_note = curr_note;

    vector<int> temp = markov_del[prev_del];
    if (main_color == "Red") { slow_weight = .25; fast_weight =
4; }
    else if (main_color == "Blue") { slow_weight = 4; fast_weight
= 0.25; }
    else if (main_color == "Green") {slow_weight = 0.25;
fast_weight = 0.25; }

    for (int i = 0; i < 4; i++)
    {
        weight = temp[i] * slow_weight;
        temp[i] = weight;
    }
    for (int i = 6; i < 10; i++)
    {
        weight = temp[i] * fast_weight;
        temp[i] = weight;
    }
```

```
    total_weight = 0;
    for(int i = 0; i < temp.size(); i++) { total_weight +=
temp[i]; }
    int rand_d = random() % total_weight;
    curr_del = findWeight(temp, rand_d);
    prev_del = curr_del;
}

//--------------------------------------------------------------
void testApp::transpose()
{
    int temp_note = curr_note + key;

    if (temp_note > 12) { temp_note = temp_note % 12; }

    curr_note = temp_note;
}

//--------------------------------------------------------------
int testApp::findWeight(vector<int> chain, int rand)
{
    for(int i=0; i < chain.size(); i++)
    {
        if(rand < chain[i]) { return i; }
        rand -= chain[i];
    }

    assert(!"should never get here");
}

//--------------------------------------------------------------
void testApp::weightTempo()
{
    float weight;
    if (distance_val >= 368)
    {
        weight = 368 / (float)distance_val;
    }
    else {weight = (735 - (float)distance_val) / 368;}

    memcpy(w_del_len, del_lengths, sizeof(del_lengths));

    for(int i = 0; i < 10; i++)
    {
        w_del_len[i] = w_del_len[i] * weight;
    }
}
```