ECE 578 Final Project Report

Portland State University

Devon Mickels

Robot Arm That Reads and Plays Music

1.0 Introduction

The goal of this project was to make use of a robot arm and have it be able to read music and interpret that music into a series of motions that would allow it to play an instrument to recreate that music. For this project, I chose to use a baby xylophone as the instrument of choice as it was cheap and easy for the robot to use.

1.1 Implementation Choices

This robot was constructed using both C++ and Python to create the functioning code. C++ was used to create a base framework for moving the robot arm in its native Arduino environment and provides a series of equations to solve for joint angles given the movement location. This C++ file also opens up a means of serial communication that allows me to pipe in instructions for the robot through a serial monitor, or in my case via my Python scripts that handle the rest of the application.

Python is used for the image processing of the music sheet and makes use of OpenCV to recognize key features on the music sheet, such as notes, note order, and rests. All of this information is then streamed through the serial monitor to the base C++ class allowing for the robot to move to locations that correlate with the respective note.

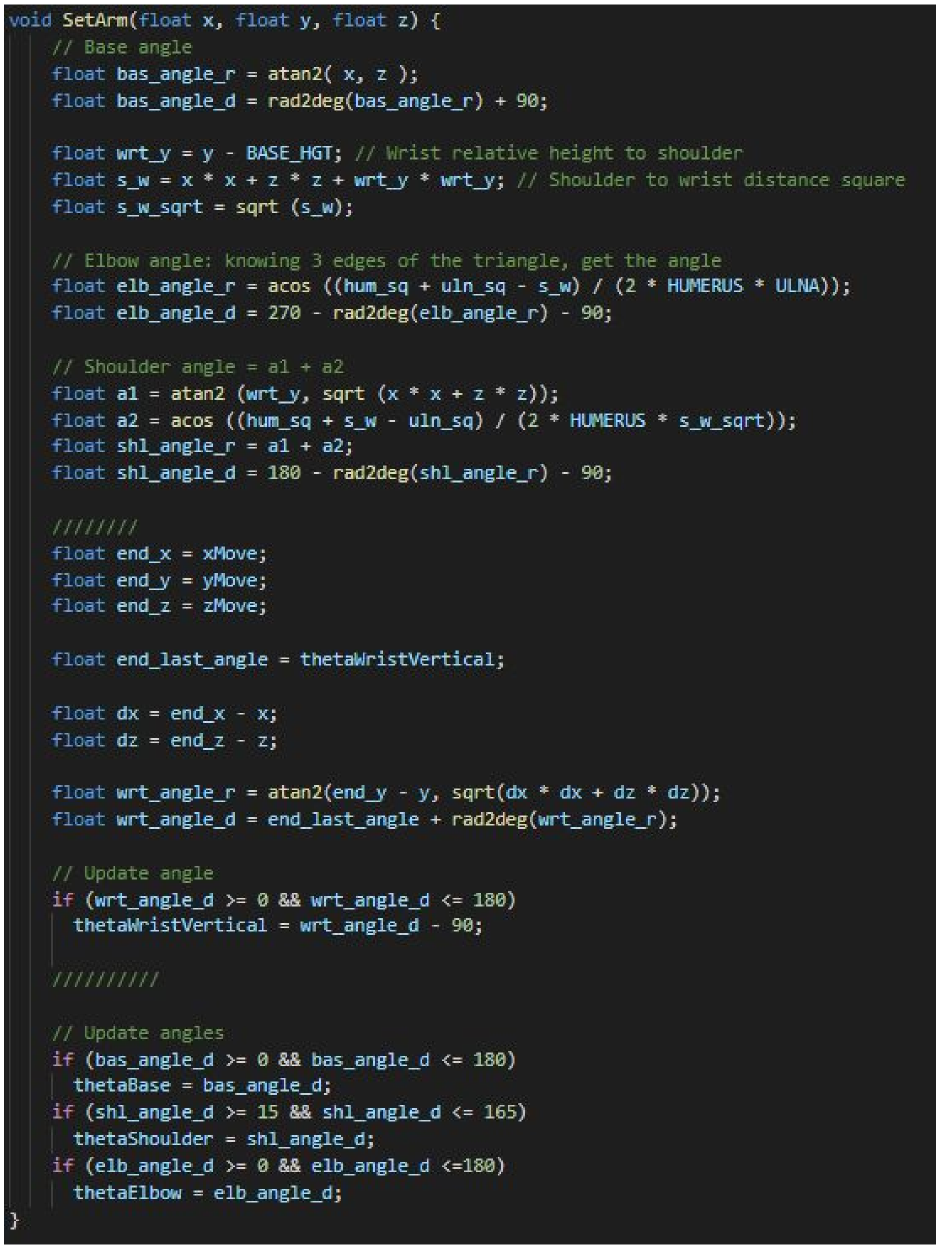
2.0 Base Framework

As previously mentioned, the base framework for the robot movement was constructed in C++ using the Arduino library. The main purpose of this program is to serve as the Inverse Kinematics solver, as well as the serial decoder to process all instructions sent to the robot arm.

2.1 Inverse Kinematics Solver

The Inverse Kinematics solver is a simple series of equations that makes use of Trigonometry relations in order to solve the joint angles given a world space. This rudimentary approach doesn't give precise cartesian coordinates but instead tends to operate on a spherical coordinate system around the arm due to lack of details in the equations, as well as the low precision that is possible given the servos in the robot arm being used. While both of these things could be replaced, those being the robot arm and the inverse kinematic equations, I found that they worked as needed for this intended application and was able to make due.

Below is the code that makes up the series of equations for solving the joint angles.



```
void SetArm(float x, float y, float z) {
    // Base angle
    float bas_angle_r = atan2( x, z );
    float bas_angle_d = rad2deg(bas_angle_r) + 90;

    float wrt_y = y - BASE_HGT; // Wrist relative height to shoulder
    float s_w = x * x + z * z + wrt_y * wrt_y; // Shoulder to wrist distance square
    float s_w_sqrt = sqrt (s_w);

    // Elbow angle: knowing 3 edges of the triangle, get the angle
    float elb_angle_r = acos ((hum_sq + uln_sq - s_w) / (2 * HUMERUS * ULNA));
    float elb_angle_d = 270 - rad2deg(elb_angle_r) - 90;

    // Shoulder angle = a1 + a2
    float a1 = atan2 (wrt_y, sqrt (x * x + z * z));
    float a2 = acos ((hum_sq + s_w - uln_sq) / (2 * HUMERUS * s_w_sqrt));
    float shl_angle_r = a1 + a2;
    float shl_angle_d = 180 - rad2deg(shl_angle_r) - 90;

    ////////
    float end_x = xMove;
    float end_y = yMove;
    float end_z = zMove;

    float end_last_angle = thetaWristVertical;

    float dx = end_x - x;
    float dz = end_z - z;

    float wrt_angle_r = atan2(end_y - y, sqrt(dx * dx + dz * dz));
    float wrt_angle_d = end_last_angle + rad2deg(wrt_angle_r);

    // Update angle
    if (wrt_angle_d >= 0 && wrt_angle_d <= 180)
      thetaWristVertical = wrt_angle_d - 90;

    //////////

    // Update angles
    if (bas_angle_d >= 0 && bas_angle_d <= 180)
      thetaBase = bas_angle_d;
    if (shl_angle_d >= 15 && shl_angle_d <= 165)
      thetaShoulder = shl_angle_d;
    if (elb_angle_d >= 0 && elb_angle_d <=180)
      thetaElbow = elb_angle_d;
}
```

Figure 1: Inverse Kinematics Solver Code

2.2 Serial Decoder

The serial decoder is a simple series of checks made to any command passed over the open serial port. This allows for simple text commands to be sent from the Python script that can be translated into movements of the robot arm.

Some examples of what these commands look like:

M10,20,30,	- Moves the robot arm to position (10, 20, 30)(x, y, z).
J10,0,0,	- Moves the robot arm 10 x units from its current position.

D	- Drums at the current location, moving down then back up.
E or Q	- Rotates the hand of the robot arm.
O or C	- Opens or Closes the robot arms hand.
G	- Gets the current location.
H	- Returns to the home position.
R	- Resets the robot arm.

3.0 Python Controller

An additional Python file was created to serve as an interface between the music sheet processor, and the base framework. This file consisted of simplified commands that could be called from the music sheet processor, such as playing a specific note or resting, as well as breaking up the music sheet processor's output and sending each note to the controller one at a time.

The main goal of this additional Python controller was to further abstract the controls of the robot from the music processor, allowing for the music processor to remain abstract and reusable in future assignments. As such, the Python controller handled opening the serial connection, breaking up the input notes and sending them one at a time, and storing needed movement locations of the specific notes.

4.0 Music Sheet Processing

4.1 Overview

The Music Sheet Processor is written in Python and makes use of the OpenCV library in order to process an input image, sheet music, and provide a string of letters and rests that correspond to those shown in the sheet music. While the hope was to process a real piece of sheet music in the end, I was unfortunately unable to get that far. The current input file style can be seen below in Figure 2.

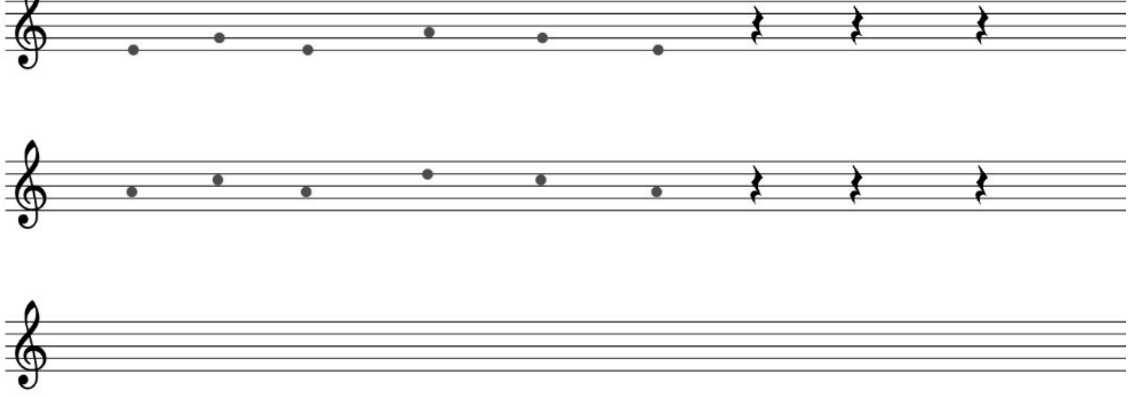


Figure 2: Style of Input File

As shown above, the notes being processed are simply just a circle in the respective location of the intended note, and quarter rests make up the icon used for resting. The input file supports a total of 10 rows (3 are shown in Figure 2) with little spacing required between each of the notes, allowing for some rather long pieces of music to be processed and played.

4.2 Note Recognition and Detection

Note recognition and position detection is handled by two simple steps. The first involves using the HoughCircles function in OpenCV to find all the of the notes shown in the input image. From here, these notes are added to an array as an object that is tagged with their corresponding coordinates on the image. The array then goes through two sorts. The first, sorts notes based on the vertical position, labeling the top line of notes as 0, the next as 1 and so on. The array is then sorted within each line of notes to array them in increasing order of their x coordinates. This provides a complete list of the notes and rests shown in the image in a left to right, top to bottom order which is what you would expect when reading a sheet of music.

4.3 Implementation Analysis

The start of the image processing starts with reading in the input file of choice, which provides the song of choice recreated on the provided image template. From here, the treble clefs are located and removed from the image to avoid any potential conflicting errors when detecting circles, or lines within the program.

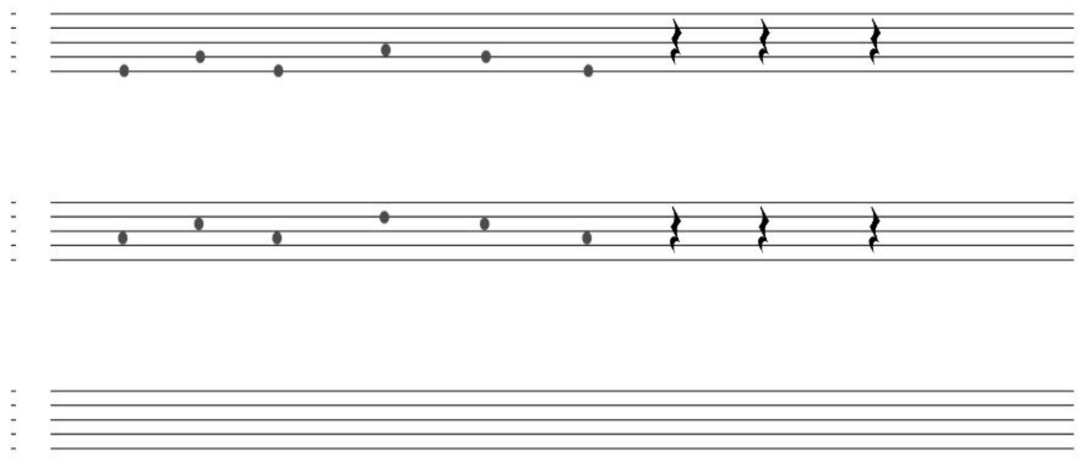


Figure 3: Treble Clefs removed

This effect can be observed in Figure 3, where the previous location of the treble clefs has been whited out. The next step in the process is to identify the rest. In its current state, the program is only trained to recognize quarter rests, making use of the matchTemplate function in OpenCV to find and locate the quarter rest images within the input music sheet. Location recognition for these are the same as the notes.

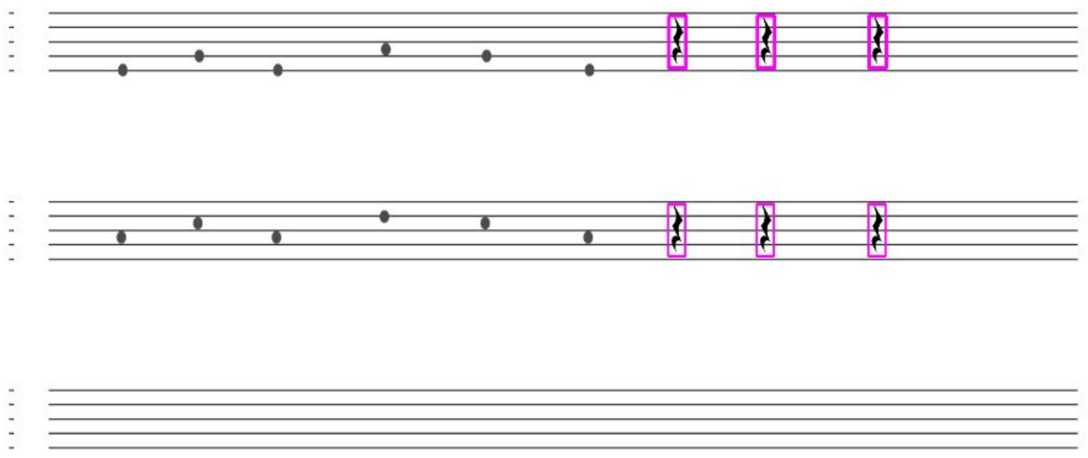


Figure 4: Rests Identified

Once the rests have been identified, we move to the final steps, the first of which is simply finding all of the lines that make up each bar of music. This can be observed in Figure 5 and makes use of the Canny edge detection and HoughLines in order to find and locate each line. These lines are used to give a position on the sheet music, which can then be cross-referenced when determining note location for each of the 10 bars in a single sheet of music.

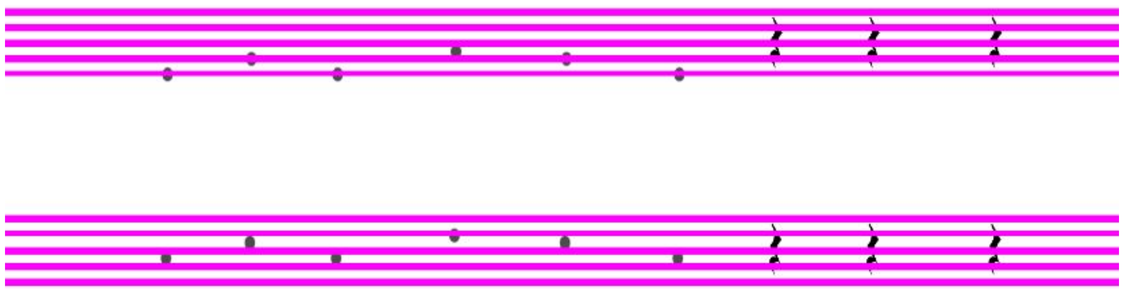


Figure 5: Lines Identified

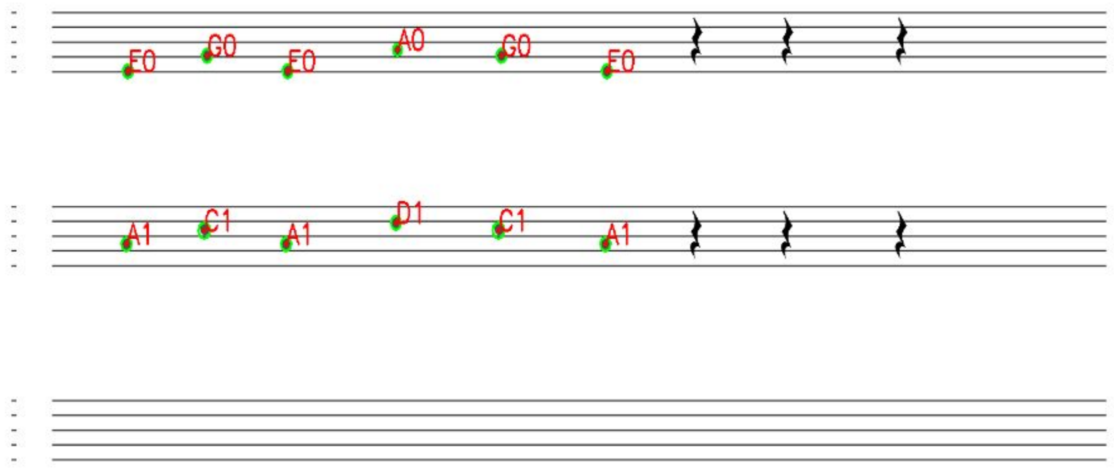


Figure 6: Notes Identified

The final step involved identifying the notes on the page, which in this case are small circles and not full notes. These notes are found using the HoughCircles function from OpenCV and are then passed through a series of sorts to identify their x and y locations, allowing for a final ordering of notes to be assembled.

```
['E', 'G', 'E', 'A', 'G', 'E', 'R', 'R', 'R', 'A', 'C', 'A',
'D', 'C', 'A', 'R']
```

Figure 7: Output of Recognized Notes

The order observed in Figure 7 depicts the array that makes up the output from the observed sheet of music. This array is then forwarded to the Python controller to decode and send appropriate commands over serial for the robot arm to play the correct note. It is important to note that each letter only has one octave, as all E's are clumped together, etc. This is because of the restraint of the musical instrument being used, as it only has one of each note and no sharp or flat options. 'R' in this case is used to represent a rest.

Source code can be found on GitHub.

Demo and explanations: https://www.youtube.com/watch?v=lYeRmwsj23U