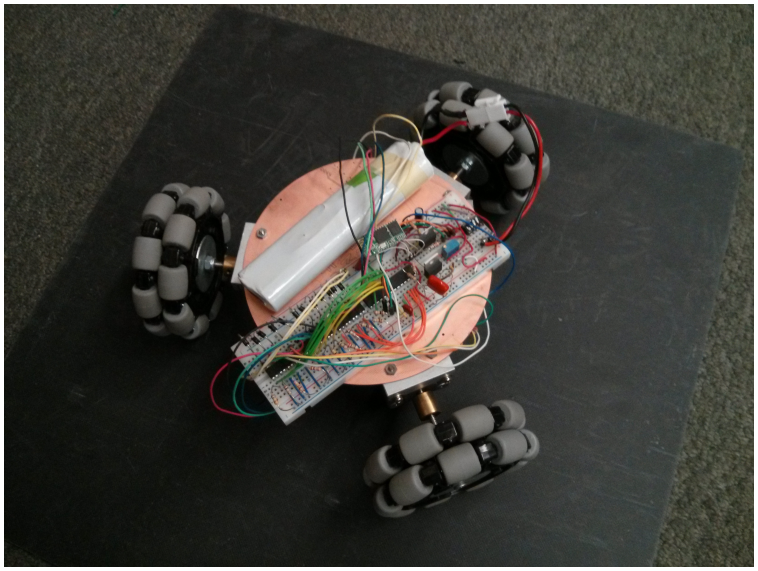

ECE533 Final Report

# 2-Phase Unipolar Stepper Omnidirectional Tri-Axis Holonomic Drive-Train Robot

*Authors:*
Forrest Flagg
Aaron McCollough
Jason Monk

May 9, 2013

# 1 Introduction

The field of robotics is rapidly expanding as new and different robots are created to perform a variety of tasks. Many robots employ various locomotion techniques for navigation, the most popular of which is a four wheel, two axis setup. This works great for many applications but has some drawbacks when trying to perform complicated maneuvers. This is where the omnidirectional, holonomic drive train comes in. By employing three omni wheels on three individually controlled drive shafts, a holonomic drivetrain enables the robot to at once move in any direction and rotate in place without having to change its orientation. This is a huge improvement over conventional 4 wheeled robots. This remarkable drive system was the inspiration for this project and the reason we chose to build a 2-Phase Unipolar Stepper Omnidirectional Tri-Axis Holonomic Drive-Train Robot.

# 2 Circuit

The figure below shows the schematic for the board.

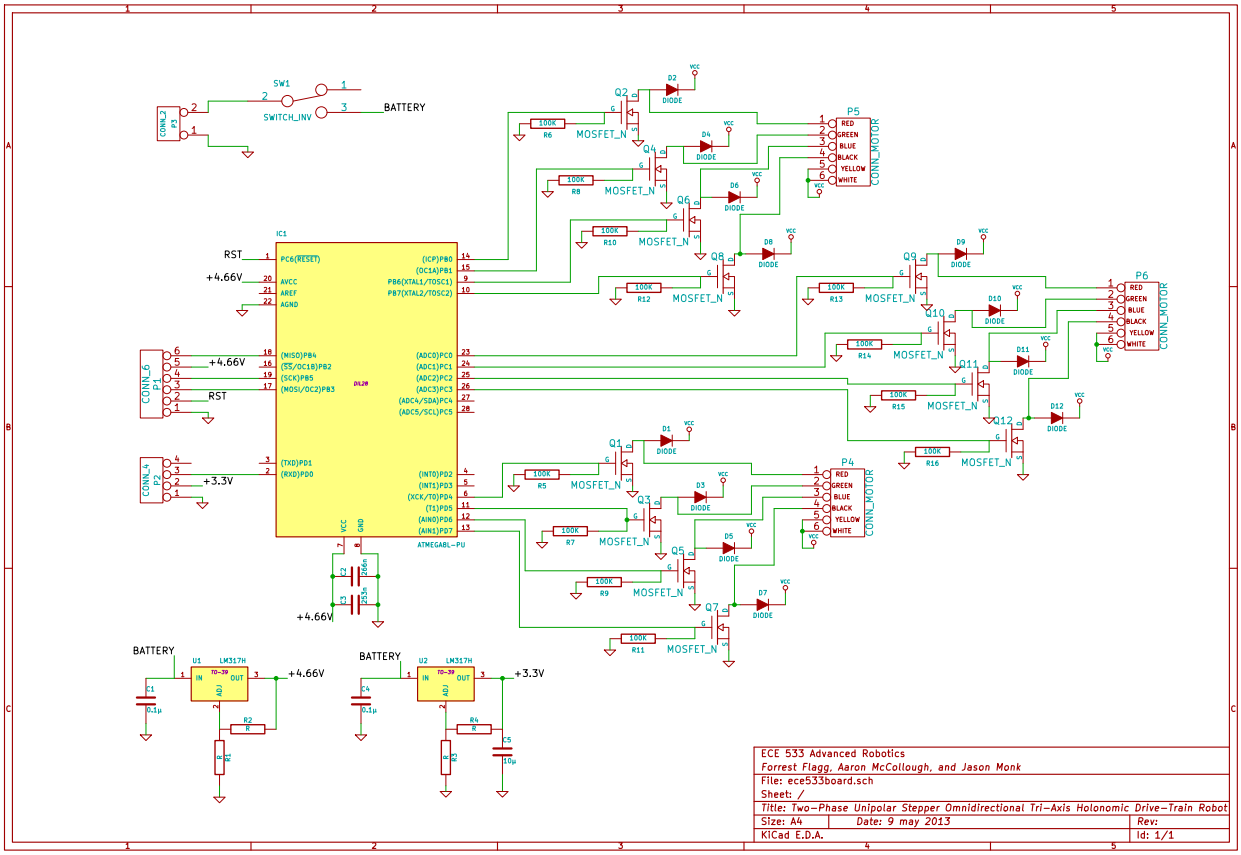


Figure 1: Full Schematic

The microprocessor that drives the robot is an ATmega8L. This chip is a 28 pin PDIP package. The ATmega8L can run at up to 8MHz which is currently being used. It has 8kB of programmable flash memory and 1kB of internal SRAM and can run on voltages between 2.7V and 5.5V.

The MOSFETs which drive the motors are IRFD110. The Vgs specification for these MOSFETs is minimum 2V and maximum 4V. They are driven at 4.66V on the robot to ensure full saturation and allow maximum torque for the motors.

Two voltage regulators are used to supply the three voltage levels needed. The regulators are LM317 linear adjustable voltage regulators. One supplies the 3.3V line that powers the Bluetooth chip and the other supplies the 4.66V rail which runs the microprocessor.

The Bluetooth communication chip is designed to be a drop in replacement for serial communication lines. Only three pins are needed It runs at 3.3V and is specified to run at up to 5.6V, but communication fails not far above 3.3V.

The stepper motors are described in Section 3.

The batteries that power the robot are 7.4V 5200mAh Li-ion rechargeable batteries made by Powerizer.

# 3 Stepper-Motor Driving

A stepper motor is a type of DC electric motor that operates by energizing an electromagnet to pull the motor into a specific position. Several electromagnets are arranged so that when energized in sequence the motor will move forward, and the reverse sequence moves it backwards.

The motors being used are PX-245-2B-C8 (AKA 101 Motors). These are Uni-polar stepper motors, which have several types of driving methods. Figure 2 shows the common driving modes for this type of stepper motor.

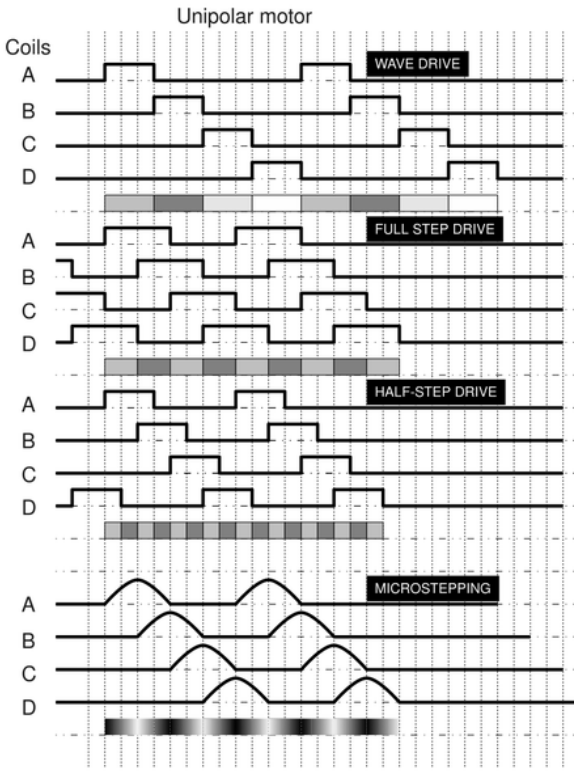


Figure 2: Unipolar Driving Modes from Wikipedia

With the hardware setup described in Section 3.1, we are capable of achieve Wave, Full Step and Half Step Drives. In testing we achieved the greatest torque from the motors from the Half Step Driving pattern, so that is what is currently implemented.

## 3.1 Hardware

As shown in the schematic, there is a driver for each of the coils on all of the motors. The motor is driven by connecting one side of the coil to Vcc and the other to a driving MOSFET (IRFD110). The MOSFET is turned on to energize that coil in the motor. Since the motors are primarily inductive, diodes (1N4004) allow the current to feedback into the Vcc rail when the coil is turned off.

## 3.2 Software

To control all three stepper motors a fair amount of asynchronous structure was required in the code. The first step was to setup a timer interrupt that ran at 10 KHz. This interrupt checks if each of the motors needs updating. It does this by decrementing 3 counters and

updating motor values when its respective counter reaches zero, then reseting the counter to the top. When a motor is updated a flag is set to indicate a new motor state is required to be calculated. There is a fourth counter which is used for timing when doing automated testing (i.e. going in a circle).

```
ISR(TIMER1_COMPA_vect) {
    TCNT1 = 0;
    if (--counter1 == 0) {
        PORT_1 = motorState1;
        needState1 = TRUE;
        counter1 = speed1;
    }
    if (--counter2 == 0) {
        PORT_2 = motorState2;
        needState2 = TRUE;
        counter2 = speed2;
    }
    if (--counter3 == 0) {
        PORT_3 = motorState3;
        needState3 = TRUE;
        counter3 = speed3;
    }
    if (--counter4 == 0) {
        needsNewSpeed = TRUE;
        counter4 = speed4;
    }
}
```

Figure 3: AVR Timer Interrupt Handler

When the needState flags are set the main code calculates the next motor state to be queued for output. The next state calculated the same way for each of the motors, either moving forward or backward in the stepping sequence, but the calculation of the port value varies for each of the motors because they reside in different areas of the port.

```
while (1) {
    ...
    if (needState1) {
        currentState1 = getNextState(currentState1, movingForward1);
        motorState1 = getMotorState(currentState1);
        needState1 = FALSE;
    }
    if (needState2) {
        currentState2 = getNextState(currentState2, movingForward2);
        motorState2 = getMotorState(currentState2) << 4;
        needState2 = FALSE;
    }
    if (needState3) {
        currentState3 = getNextState(currentState3, movingForward3);
        state = getMotorState(currentState3);
        motorState3 = ((state & 0x0C) << 4) | (state & 0x03);
        needState3 = FALSE;
    }
}
```

Figure 4: Next State Calculation

The states of the motor are stored in a buffer that can easily be changed to any of the motor driving techniques described above. Then the motor states can be traversed by moving forward or backward through the state array;

```
const unsigned char motorStates[8] =
        {0x01, 0x03, 0x02, 0x06,
         0x04, 0x0C, 0x08, 0x09};

unsigned char getMotorState(unsigned char state) {
    return motorStates[(int)(state & 0x07)];
}
```

Figure 5: Next State Function

A simple helper function controls the movement between states based on the current state and the direction of the motor. It performs an increment or decrement modulo 8.

```
unsigned char getNextState(unsigned char currentState,
                           unsigned char movingForward) {
    if (movingForward) {
        ++currentState;
    } else {
        --currentState;
    }
    return currentState & 7;
}
```

Figure 6: Next State Iteration

# 4 Omni-Directional Control

When moving the robot there are two aspects that need to be considered. The translation component and the rotation component. These actions can be considered separately and then combined after the calculations have been made. There are some caveats to this and they will be discussed in the following sections.

## 4.1 Translation Component

The basic kinematics for each wheel on the robot can be shown in figure 7. The direction vector shown indicates the direction the robot wants to go relative to the wheel. The speed at which the wheel must turn is proportional to the component of that vector that lies perpendicular to the wheels axis, indicated by the red arrow. The wheel magnitude vector can be calculated as shown in the following equation:

$$S = V * \sin(\Theta). \tag{1}$$

Where S is the wheel speed and V is velocity of the robot.

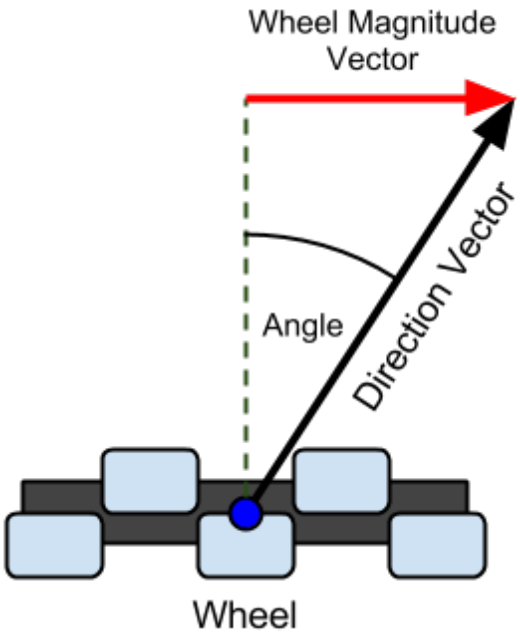


Figure 7: Wheel Kinematics Diagram

To calculate the wheel magnitude vector for the other two wheels, simply add an offset of 120 degrees and 240 degrees respectively to the direction vector angle.

## 4.2 Rotation component

The rotation component is much simpler than the translation component. If you want the robot to rotate in one direction simply run all the wheels in the same direction at the same speed. If the speeds are different the robot will still rotate but there will be some translation as well.

## 4.3 Combining Translation and Rotation

If you want the robot to move some distance and rotate some number of degrees at the same time you can simply calculate the translation and rotation components separately and then sum the results for each wheel. Setting the wheels to this speed will cause the robot to move that distance and rotate that amount but the result of rotating while translating is that the robot will veer off course. This is because the translation is relative to the orientation of the robot and if that orientation is changing so will the direction of the robot.

One solution is to constantly update the robot wheel speeds to accurately reflect the position and orientation of the robot in its current time step so that even as its rotating the

translation calculations are being made based on the robots current angle and not the start angle. This solution is not ideal because it requires many, rapid, updates. In the ideal case it would require an infinite amount of updates. Our solution is to find a translation vector and a rotation speed that can be set to move the robot from any position and orientation to any other with a non-changing wheel speed.

For this to work the robot will have to travel in an arc from one point to the next. The length of the arc is derived from the law of cosines and is given in the following equation:

$$D = \sqrt{\frac{\Delta x^2 \times \Delta y^2}{2 - 2\cos(\Theta)}} \times \Theta. \tag{2}$$

The translation angle of the robot is given by:

$$\Theta_s = \arctan(\frac{\Delta x}{\Delta y}) + \frac{\Theta_r}{2}. \tag{3}$$

Now that we know the distance the robot must travel we can calculate the time that a given operation will take.

$$Time = \frac{D}{S}. \tag{4}$$

The rotation component of the robot can now be calculated.

$$\phi = \frac{\Theta_r}{Time}. \tag{5}$$

Now that the adjusted translation component and the rotation component are known they can be combined to give each wheels individual speed to control the robot.

```
//relative new x and y
float x = rloc.x;
float y = rloc.y;
//relative rotation in radians
float rot = rloc.rot;
Log.e("ROBOT PATH", "L: " + x + " " + y + " Rot: " + rot);

//finalant variables
//Robot radius
final float robot_rad = 4.75f;
//Wheel radius;
final float wheel_rad = 2.0f;
//Maximum Robot Speed
final float max_speed = 6.28f;

float radius,d,fulld;

// linear distance to next point
d = (float) Math.sqrt(x*x + y*y);

// radius of circle that the arc lies on
radius = (float) Math.sqrt(d*d/(2*(1-Math.cos(rot))));

// length of the arc
fulld = radius*Math.abs(rot);

//start_rot = rot/2;
double final_angle = Math.atan2(y,x) + rot/2;

// calculate the time to next state
float time = fulld/max_speed;
double rob_rot = rot * robot_rad / (/*2 * */wheel_rad) / time;
Log.e("ROBOT PATH", "Rotation: " + rob_rot);

// Calculate ticks per second
double tps1 = (Math.sin(final_angle)*max_speed * 1/2 + rob_rot)

double tps2 = (Math.sin(final_angle+RobotControl.off2)*
                              max_speed * 1/2 + rob_rot)
                                    * (180/(0.9*Math.PI));
double tps3 = (Math.sin(final_angle+RobotControl.off3)*
                              max_speed * 1/2 + rob_rot)
                                    * (180/(0.9*Math.PI));
```

Figure 8: Control Implementation

# 5 Wireless Communication

We wanted the robot to be completely wireless as its omni-directional nature is not exactly condusive to having any wires hanging off from it. The robot currently uses a bluetooth transciever to communicate with phones or computers.

## 5.1 Bluetooth

The Bluetooth communication chip has a number of features that are not being utilized. The chip supports interfaces for USB, UART, SPI, and PCM. Only the UART interface is currently used. This allows the chip to be a transparent replacement for serial UART lines. The chip handles all of the Bluetooth communication including pairing, the Bluetooth stack, and pin security.

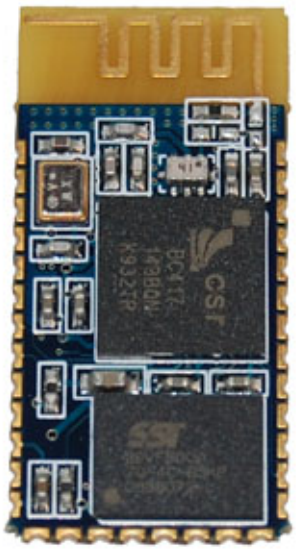

Figure 9: Bluetooth Device Used

## 5.2 Protocol

The communication from the AVR side is extremely simple. The AVR recieves data on the UART at 9600 Baud. At the top of every main loop the code checks for incoming serial communication and stores it in the buffer. When it has received 6 bytes it interprets them as 3 signed integer values, these are then used as delays for each of the motor driving.

```
char index = 0;
int speeds[3];
char* buffer = (char*) speeds;

while (1) {
    if  (UCSRA & (1<<RXC)) {
            buffer[index++] = UDR;

            if (index == 6) {
                    setSpeed1(speeds[0]);
                    setSpeed2(speeds[1]);
                    setSpeed3(speeds[2]);
                    index = 0;
            }
    }
        ...
}
```

Figure 10: Serial Receive

The setSpeed function simply handles the signed number so that positive goes forward and negative goes backwards.

## 5.3 Android

To connect to the bluetooth chip on android some complicated code was required. Figure 11 shows the code that connects to the bluetooth chip described in Section 5.1. It uses reflection to access the function createRfcommSocket, which is no longer available in current android APIs. The code then calls the function on the device which opens a socket communication with it.

```
// hard coded the MAC of our bluetooth chip
device = adapter.getRemoteDevice("00:19:5D:EE:08:05");

// breaks out functions that are no longer broken out for our use
// specifically connect
Method m = device.getClass().getMethod("createRfcommSocket",
                new Class[] { int.class });
bluetoothSocket = (BluetoothSocket) m.invoke(device, 1);

// connect to the board
bluetoothSocket.connect();
```

Figure 11: Android Bluetooth Connection

In practice we had a lot of reliability issues with using Android to connect to this bluetooth device. It would sometimes have problems disconnecting, then problems reconnecting, and required rebooting the phone a lot. The workaround used for this was to use a laptop to connect to the bluetooth device, then use a socket forwarding program (socat).

```
$ sudo rfcomm connect rfcomm0 00:19:5D:EE:08:05
$ sudo socat TCP-LISTEN:8023,fork /dev/rfcomm0,raw,b9600,echo=0
```

Figure 12: Socket Forwarding

# 6 Android App

An Android App was developed to control the robot. There are two modes of control that were supported. The first is referred to as Simple Control and allows the user to direct the robot to move in any direction or turn in any direction. The second mode is referred to as Fancy Control and utilizes multitouch to allow the user to draw a path and orientation for the robot to move.

Figure 13 shows the startup screen for the app. It allows the user to enter either of the modes above. It also allows the user to set the IP for the Socket-Bluetooth bridge as discussed in Section 5.3.


Figure 13: Android App Main Menu

Figure 14 shows an example of setting the IP and Port of the Socket-Bluetooth bridge.

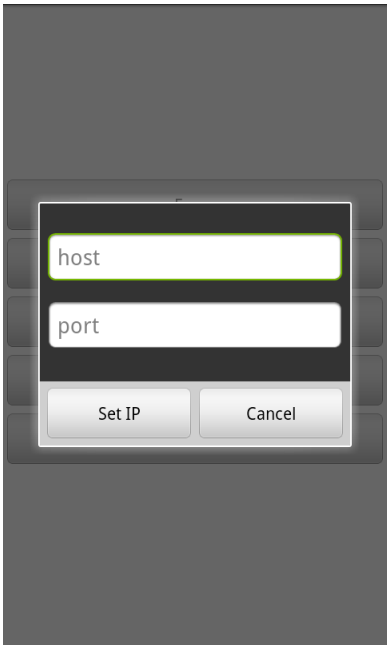


Figure 14: Android host Selection

## 6.1 Surface View

Both the Simple and Fancy Control use a Surface View to have direct control to drawing on the screen. The Surface View creates a Surface Holder which allows access to the canvas for the screen. Figure 15 shows how the rendering thread gets the canvas from the surface holder.

```
public void run() {
        long lastTime;
        while (mRun) {
                lastTime = System.currentTimeMillis();
                if (holder != null) {
                        canvas = holder.lockCanvas();
                        if (canvas != null) {
                                drawFrame();
                                holder.unlockCanvasAndPost(canvas);
                        }
                }

                long diff = delay - (System.currentTimeMillis() - lastTime);
                try {
                        if (diff > 0) {
                                sleep(delay);
                        }
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

Figure 15: Socket Forwarding

The function drawFrame varies from Simple to Fancy control but both of them use the canvas created to draw directly onto the screen.

## 6.2 Multitouch Control

Handling multitouch on android is a fairly simple process. Whenever the Current Activity receives a onMotionEvent call, the Activity passes it off to a class to track the location of all fingers on the screen.

```
public void updateTouch(MotionEvent event) {
        if (isMultitouch) {
            final int pointerCount = event.getPointerCount();
            for (int i = 0; i < pointerCount; i++) {
            final int action = event.getAction();
            final int actual = action & MotionEvent.ACTION_MASK;
            final int id = event.getPointerId(i);
            if (actual == MotionEvent.ACTION_POINTER_UP ||
                        actual == MotionEvent.ACTION_UP ||
                        actual == MotionEvent.ACTION_CANCEL) {
                unTouch(id);
            } else {
                touch(id, event.getX(i), event.getY(i));
            }
            }
        } else {
                if (event.getAction() == MotionEvent.ACTION_UP) {
                        unTouch(0);
                } else {
                        touch(0, event.getRawX(), event.getRawY());
                }
        }
}
```

Figure 16: MultiTouch Tracking

## 6.3 Simple Control

The simple controller draws a circle and a bar on the screen. It then looks for touches within the circle or the bar. When a touch is within the circle it moves the robot in the direction from the center of the circle to the finger, at a speed proportional to the distance from the center. The bar performs similarly except it controls the rotation of the robot and only operating on the X axis.

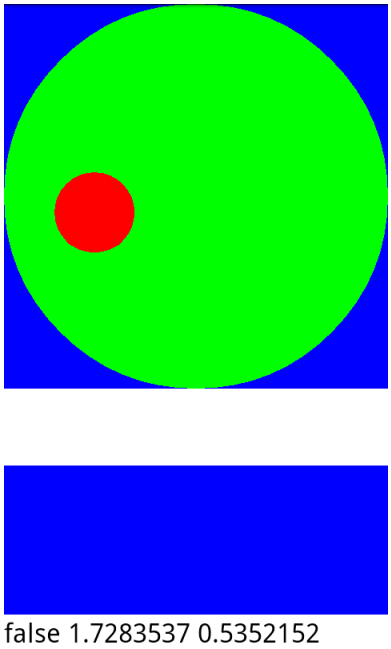


Figure 17: Android Simple Control Interface

The bottom of the screen shows true/false indicating whether the app has successfully connected to the TCP Socket/Bluetooth. This can help significanty when debugging the connection to the robot.

## 6.4 Fancy Control

The Fancy Controller has a much simpler display to it. It draws a blue background to draw on, red dots for each of the fingers detected by multitouch, and the path that has been detected so far. Figure 18 shows the display with all three of these in effect.

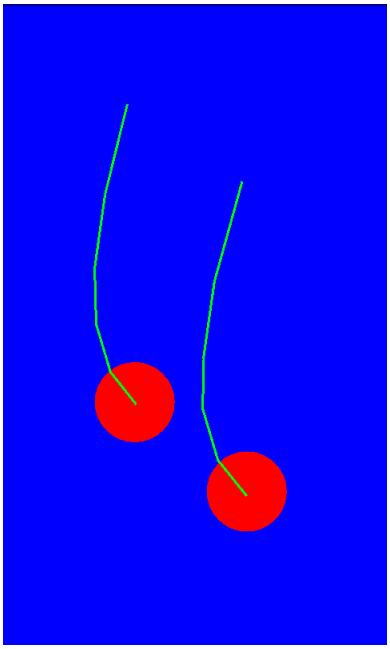

Figure 18: Android Fance Control Interface

The path is created by sampling the finger locations every 1/4th second. When there are 2 fingers on the screen the path is recorded. The path is interpreted as the point half way between the fingers being the desired robot location, and the angle of the vector between them as being the position. The point at which the fingers are set down on the screen is assumed to be (0, 0, 0), which is the current position of the robot. When the number of fingers on the screen changes to anything besides 2 the path for the robot is calculated and the robot follows it. Figure 19 shows the code that gets called every 1/4th second to check the number of fingers/locations of those fingers.

```
private void updatelocations() {
  int numactive = 0;
  Location l1 = null;
  Location l2 = null;
  Point point;
  for (int i = 0; i < TouchController.MAX_POINTS; ++i) {
    point = touchcontroller.getPoint(i);
    if (point.isSet()) {
      ++numactive;
      if (l1 == null) {
        l1 = new Location(point.getX(), point.getY());
      } else {
        l2 = new Location(point.getX(), point.getY());
      }
    }
  }
  synchronized (locs) {
    if (isupdating) {
      if (numactive != 2) {
        isupdating = false;
        path = new RobotPath(locs, TouchActivity.robotControl);
        path.start();
      } else {
        if (Math.max(l1.dist(locs.get(locs.size() - 1).first),
                     l2.dist(locs.get(locs.size() - 1).second))
                     > THRESH) {
          locs.add(new Pair<Location, Location>(l1, l2));
        }
      }
    } else {
      if (numactive == 2) {
        locs.clear();
        locs.add(new Pair<Location, Location>(l1, l2));
        isupdating = true;
      }
    }
  }
}
```

Figure 19: Finger Tracking

# 7 Future Work

This project has a number of ways in which it could be improved in the future. This section details a few that we have thought about and partially explored, and how we think they could help out the robot.

## 7.1 Protocol

The current protocol allows for a fair amount of error to occur. When the information being sent is offset by 2 bytes then the robot directions will be rotated by 180 degrees, when offset by 1 byte some very undesirable behavior will occur. We would like to modify the protocol to be encapsulated by start and end bytes. This would allow for a more robust communication protocol as it would allow us to use a circular buffer on the robot to ensure no loss of data.

There is also no error checking done by the robot. A delay of any value can be sent to the motors, however the motors do not have enough torque to handle a delay value any less than 40. Some input checking could improve operation of the robot.

## 7.2 Image Processing

The original plan for this project was to have an overhead camera to track the robots position. The camera would identify hands and the robot and make the robot attempt to avoid being grabbed by the the hands. The robot was planned to be identified by a cover on the top of it with blue and green dots (both of which avoided the major color spectrum of skin tones). This camera tracking could be a very good addition to the project.

## 7.3 Motors

The current stepper motors work fairly well, but they have several drawbacks. They are quite heavy relative to the rest of the robot. The motors also dont have quite enough torque for the wheels. This can result in slipping and doesnt allow the robot to move over even slight bumps in the surface. The most noticeable drawback is the power consumption. They draw a significant amount of current, between 2A and 3A. Moving to DC motors would fix many of these issues. Suitable motors have been found which offer a slight increase in torque but for significantly less weight and current draw. The increased torque along with the dramatic weight reduction would allow the robot to be much more mobile and should address the issue of movement on non-uniform surfaces.

There are a few issues with using the DC motors however. It would require an entirely different system for driving the motors than is currently employed. Each motor would need

an h-bridge. The driving for each of the h-bridges would require three PWM signals to control each motor individually. This means that either software PWM would be required (at lower frequency) or a more powerful microcontroller would need to be used (one with three PWM channels).

Also, a different mounting system would need to be devised as the motors did not come with mounting brackets and the current mounting brackets are not compatible with the DC motors.