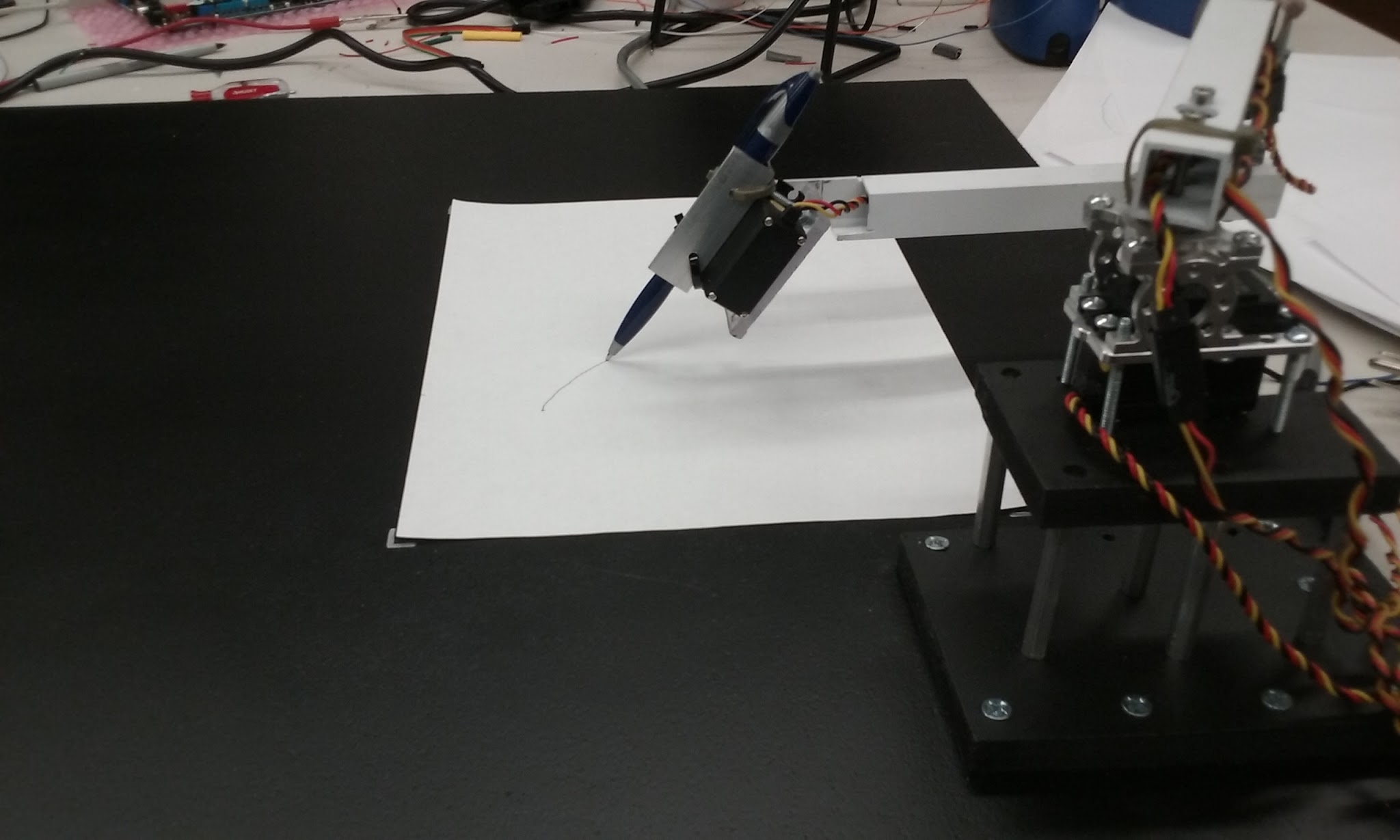
**Robot Artist**

ECE 470 - Final Project Report

By: Bryan Johnson & William Edds

Prepared For: Shadi Alawneh



**Introduction**

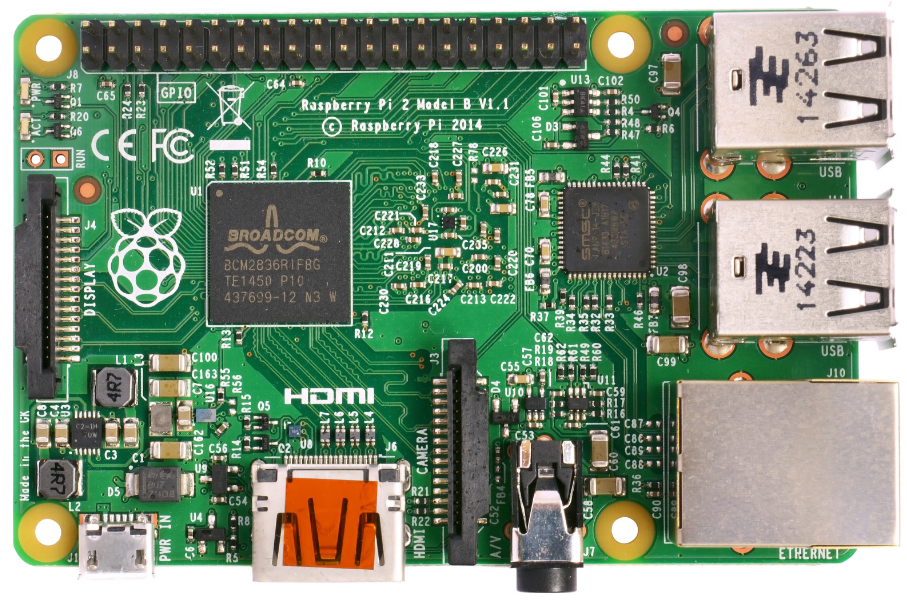
Mankind has been fascinated with art for tens of thousands of years. Until recently, art has been a uniquely human interest. With the advancement of robotics and the power of computer vision, robots are now able to participate and possibly outperform man in this ancient activity.

The goal of this project is to create a two-dimensional robotic arm that is able to draw an image taken with a camera. This is done through manipulation of the original image using an edge detection algorithm, converting coordinate systems, and inverse kinematics. This combination of steps results in a robot arm that is able to draw beautiful portraits, in the same fashion as a human being.

In order to create a robot that is able to draw any digital image, there are several steps that must be executed. These steps involved different disciplines, including but not limited to computer vision and using a microcontroller. First, an image (taken by a camera or one that is already saved on the computer) is processed using edge detection, the pixels are translated to Cartesian coordinates, each location is entered into an inverse kinematics algorithm which creates three output angles for each servo, and these angles are sent via SPI from the computer (Raspberry Pi 2) to the microcontroller, where they are finally converted into PWM signals for each servo. Computer vision will be explained in detail first, and then the process of using a microcontroller to control the robot will be described in detail.

**Computer Vision**

For the computer vision portion of this project, the goal was to have an image as the input of a function or functions, and to have a set of three angles (theta\_one is the angle for the first/shoulder joint, theta\_two is the angle for the second/elbow joint, and theta\_three is the final angle for the wrist joint). This goal was completed by using Python on a Raspberry Pi 2 for these algorithms. The Raspberry Pi 2, which will be referred to as the Pi from this point forward, is a $35 computer that runs a distribution of Linux (Debian/Raspbian in this case) for its operating system. The Pi has a 5MP camera that is sold separately; this camera was used in our final design. All of the image processing and inverse kinematics were executed in Python on the Pi; larger or more intricate images often created many edges, and the inverse kinematics had to be executed one time per pixel. At most, 20,000 pixels had to be used as inputs to the inverse kinematics, which is a very costly task for a CPU. This would have been impossible to execute on the HCS12 microcontroller, which is why using a Pi was necessary and most practical.



**Figure 1.** Raspsberry Pi 2 with four USB ports, HDMI output, camera input, an ethernet port, and the BCM2835 SoC (CPU, GPU, DSP, SDRAM).

The first step in drawing a digital image is to process the image. This is done using Python’s OpenCV libraries. The image is either read from a .jpg file, or an image is taken with the Pi Camera and saved as a .jpg file. The saved image is then processed by converting to grayscale, and using Canny edge detection with proper thresholding (see Figure 2). There are five steps to the Canny edge detection algorithm [1]:

1. Apply Gaussian filter to smooth the image in order to remove noise
2. Find the intensity gradients of the image
3. Apply non-maximum suppression to remove spurious responses to edge detection
4. Apply double threshold to determine potential edges
5. Track edge by hysteresis; finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges



**Figure 2.** A grayscale image [2] before and after being processed with Canny edge detection. When drawing an image, a human will focus on features like edges, so that was the most logical approach for programming a robot to draw an image.

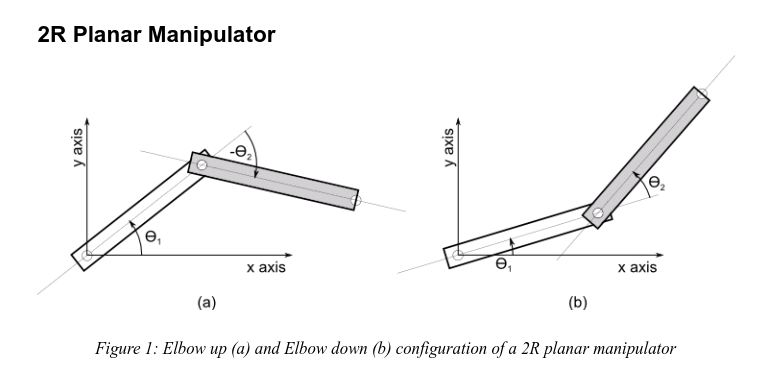
Before using edge detection, the original image is resized down to 352 x 272 pixels (to match the aspect ratio of a 8.5” x 11” sheet of paper) to cut down on processing time. The thresholds for the edge detection are set using two trackbars for the upper and lower values. The higher the thresholds, the fewer number of edges. Because each image is completely different, it is important to be able to change these thresholds easily.

After the edges are found and the new image is saved to a global variable, the find\_pix() image is called. This is where the pixels and lines will be added to a list (pix\_list) to be later drawn. The image is indexed through until a white (edge) pixel is found. This pixel is then added to pix\_list. The neighboring 8 pixels (3 x 3 matrix) are then checked and if any of these pixels are also part of an edge, the follow\_pix() function is called. In this function, the pen is set down (the third servo gets a value of 65 degrees), the current pixel is added to pix\_list, and the follow\_pix() function is called again with the current pixel location.

This process creates a function that will find a white pixel (edge), add it to pix\_list to be drawn, and continue looking for white pixels, adding them to a list, and following the line until it ends. After an edge/line ends, indexing through the image continues right after the beginning of the line started. In addition, any pixel added to pix\_list is deleted so that the same line is not drawn twice. Basically, the image is indexed through starting at the top left corner until a white pixel is found, and that line is followed through til the end, where the find\_pix() function moves on looking for another white pixel (the beginning of a new line).

When the entire image has been indexed, pix\_list is completed and each pixel coordinate (x, y) is sent to the pix\_to\_ik() function to be translated into angles for the servos. This is the longest process in terms of computing time.

This robot is a 2D-planar arm, and the inverse kinematics are fairly straightforward (3, Figure 4). However, this function is called 10,000 to 20,000 times for a single image, so this can be very time consuming. First, the pixel coordinates are translated into Cartesian coordinates. Then the link lengths (a1, a2) are converted to inches. The angles are found using the equations found in [3], and the angles are then checked to make sure they are within bounds. The final servo, controlled by theta\_three, is either 0 or 65 degrees depending on whether or not the pen is down. These angles are appended to an array (angle\_array) each time pix\_to\_ik() is called, and saved to a SQL database (also known as a dataframe). This ￼SQL database will be indexed through and sent via SPI to the HCS12 microcontroller.



**Figure 4.** Overview of how angles are viewed when looking at a 2D planar robot. The right image (b) is the reference used for this robot because both angles are positive. This is a bird’s eye view of the robot, with theta\_one being the shoulder joint, and theta\_two the elbow joint.

**Microcontroller**

Programming the microcontroller presented many unique challenges, each of these requiring time and a very systematic approach to troubleshooting the robotic system as a whole. Issues ran into and resolved include problems with a common ground between the HCS12 and the Pi as well as intermittent issues with SPI and servo errors.

Once the inverse kinematics were run and the three theta values were stored in a SQL table, the Pi (acting as the master) would initiate the transfer of those values via SPI to the HCS12. The program running on the HCS12 initialized the SPI registers on the board, for this particular use case only the PS7 (used for external SPI) , PS6 (used for the SPI clock controlled by the Pi), and PS5 (Master Out Slave In) pins were necessary to facilitate SPI communication. In this system communication was one way, originating at the Pi, and terminating at the HCS12. Since no feedback was used in this system there was no need for the HCS12 to send a signal back to the Pi.

The transmission of a byte of data from the Pi triggered an edge controlled interrupt on the HCS12. It was within this interrupt that the values of the three servo angles were written. One of the more simple yet profoundly important lessons from this project involved clearing the registers on the HCS12 in between processing a new image. This presented a problem in the early stages of the project; if the registers had not been properly cleared, the incoming bytes become offset. Since each bit is shifted in it is very easy for the system to quickly become offset in a way which is impossible to trace after only a few bytes of information. Another important insight came after an issue that was only observed when originating SPI communication on the Pi 3. In order to facilitate communication from the Pi 3 to the HCS12, the BCM 2835 library was used to write a C program that runs on the Pi 3. This program initializes the GPIO pins needed for SPI communication, along with setting the bytes endianes and clock speed. Unbeknownst to us this library was written for the previous model Broadcom chip which was only found on the Pi. This one oversight caused hours of debugging, after the first few bytes of data were received, the spi communication consistently went hay wire. Shifting in invalid values, causing the robotic arm to swing in an unpredictable fashion or jitter in one location endlessly.

Since the speed of the SPI communication far out paced the speed with which the servos could move from location to location; small edges produced a quick flick of the pen. In many cases if the edge was not long enough the next byte would have been written causing the pen not to touch the page in these areas. One proposed solution for this issue was the creation of a buffer system to store values as they were transmitted to the HCS12. The onboard SPI register has a built in transmit buffer which stores three bytes at a time. As the new data is received the buffer is successively overwritten. The speed difference between the SPI and the Servos would be a perfect use case for expanding this buffer, this along with a feedback loop and potentially using a different Microcontroller are among the proposed improvements for the future of the project.

In order to avoid using signed integers over SPI each servo value was offset before being sent. Since each of these offsets were different each servo had a slightly different equation that was used to write the duty cycle needed for each angle. These equations made use of the chosen clock divider, giving the maximum resolution possible to improved drawing performance.

**Results**

The robot we created performed better than we could have hoped for. This project was a huge undertaking, and the fact that we were able to complete this in about three weeks is a feat in itself. There were some mistakes and problems along the way. One problem was the fact that the chip on the Raspberry Pi 3 is relatively new, and does not have a SPI library available. All of the image processing was originally done on the Pi 3 because it has been designed for image processing projects and is therefore much faster than the Pi 2. However, because the SPI protocol was already written in C on the HCS12 and we did not want to start over, we decided to use the Pi 2 and sacrifice processing speed.

Another significant problem was getting the SPI to work reliably on the HCS12. When we were first testing the robot, the robot would draw relatively well for a few minutes, then would go haywire and the servos would attempt to reach impossible angles. The accepted culprit was the SPI, and this problem was fixed to the best of our abilities by changing and cleaning up the power supply with a capacitor from the positive to negative terminals. In addition, a common ground was used for every piece of hardware; this helped to remove ground loops. After these changes were made, the robot was able to execute every set of angles without a failure.

As for future improvements, it would be extremely beneficial to add feedback and control loops such as a PID to smooth out the servo movements. The servos currently have no feedback and the system is an open loop because of the lack of time and this was not required. Given more time, PID control would smooth the trajectory of the robot, making more natural looking lines on the paper. Additionally, the Dragonboard with the HCS12 microcontroller is probably overkill for this project. All that is needed is a simple microcontroller to recieve SPI data and transmit PWM signals. This would be much more simple and cost effective on an Arduino which has better documentation, is cheaper, and more user friendly. However, given that the HCS12 was a requirement for this project, it was benficial working with a microcontroller that required more thought and effort.

**References**

[1] "Canny edge detector", En.wikipedia.org, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Canny\_edge\_detector. [Accessed: 13- Nov 2016].

[2] Bogotobogo.com, 2016. [Online]. Available: http://www.bogotobogo.com/python/OpenCV\_Python/images/Canny/Canny\_Edge\_Detection.png. [Accessed: 05- Dec- 2016].

[3] 2016. [Online]. Available: http://www.hessmer.org/uploads/RobotArm/Inverse%2520Kinematics%2520for%2520Robot%2520Arm.pdf. [Accessed: 18- Nov- 2016].