**Michael Williams, 211087798, mw1992**

**Andrew Lau, 212905253, Pellost**

**November 21 2016**

**EECS4422**

FINAL REPORT

FOR JENGA STACKING SYSTEM

***INTRODUCTION***

This report focuses on the critical components of the Jenga stacking system and the workings of the entire program itself. In this report, we discuss the critical components, how they are relevant to the overall program and how they were implemented. We also include details of the problems we faced in building each of the components and provide visuals where applicable of how and when each component affects the entirety of the program. After talking about the components we then detail and showcase the final results of the entire program itself, explaining overall how it works, how it incorporates each component and program’s result. All code and diagrams for the components and the main class running the components is available at the end of the lab, in the appendix.

***PROGRESS***

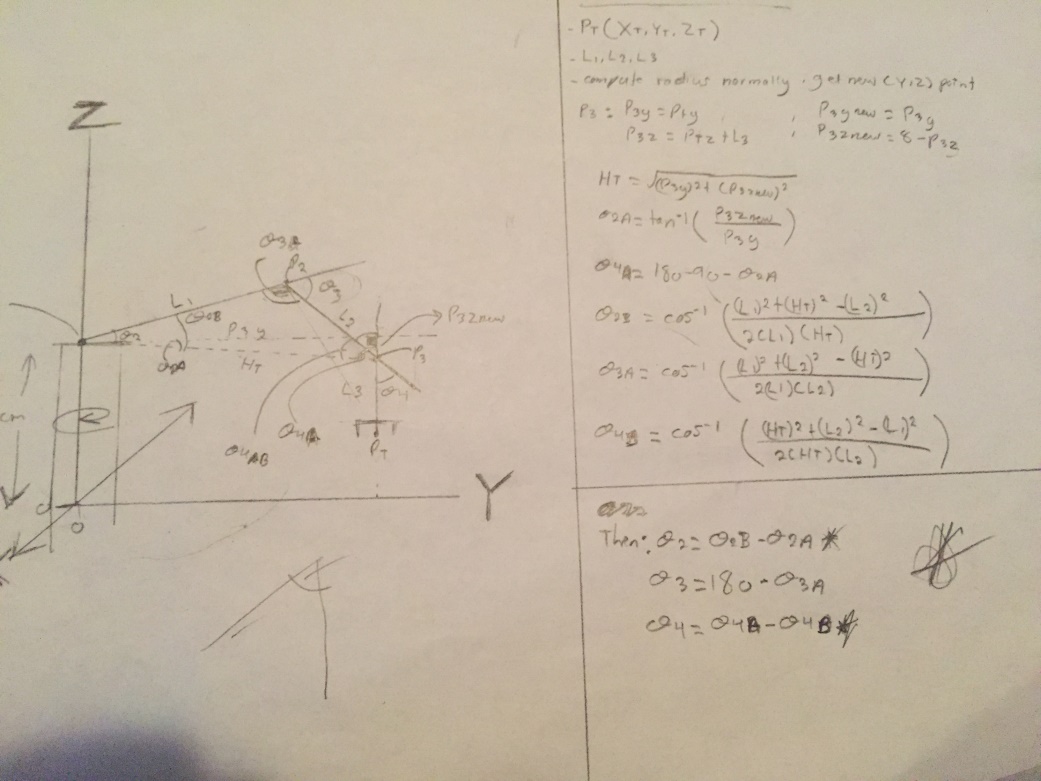
The first of the components completed is the component for moving and picking up objects in the plane by means of the robotic arm. The movement of the robotic arm was programmed using the inverse kinematic formulas derived in the first assignment, as a starting point or template. We altered this code, which originally moved the robot arm to any reachable point in a two dimensional plane, to form a program that moves to a point attainable in the three dimensional plane instead. The exact methodology is shown in the appendix but basically we assign a new parameter for the z-axis. This parameter, given by the user, is used in combination with the adjustments made to the kinematic formulas which allow us to accurately place the object at a point one the 3d plane. To do this, the problem is broken into 2 parts: the rotation, and the distance. First, we use the x and y coordinate to figure out the angle of rotation the arm will have to make, such that it is facing the block it is about to pick up. Then, we assign the radius as the new Y coordinate for the object, and use this Y along with the original z given, to compute the angles needed to pick up the object.



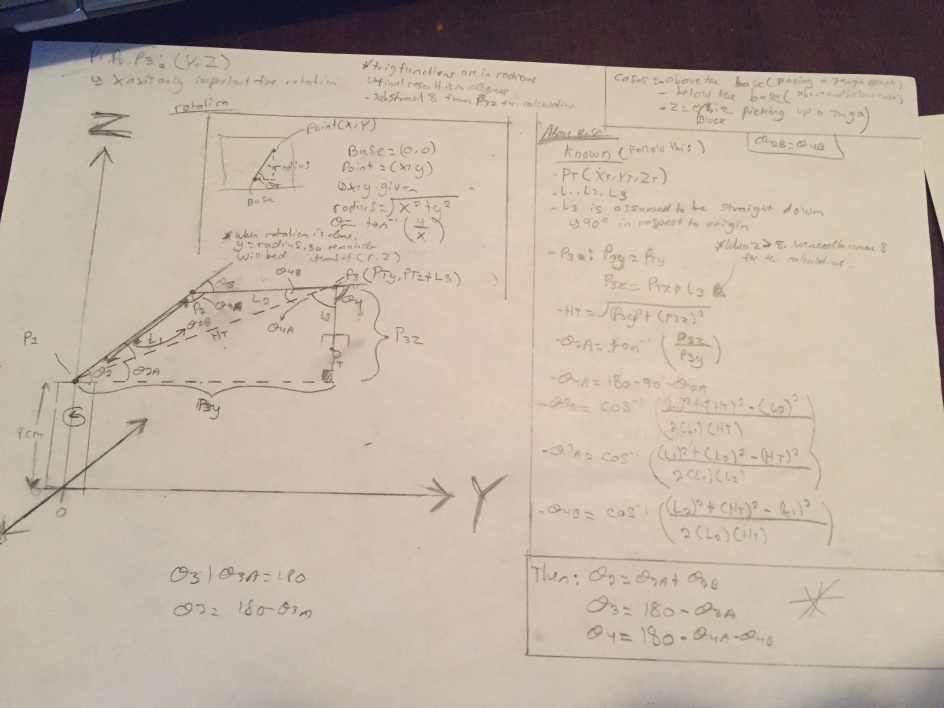
above are images depicting the arm moving to a -90 and 90 degree radius



Image depicts arm rotating to proper location, given the x and y component

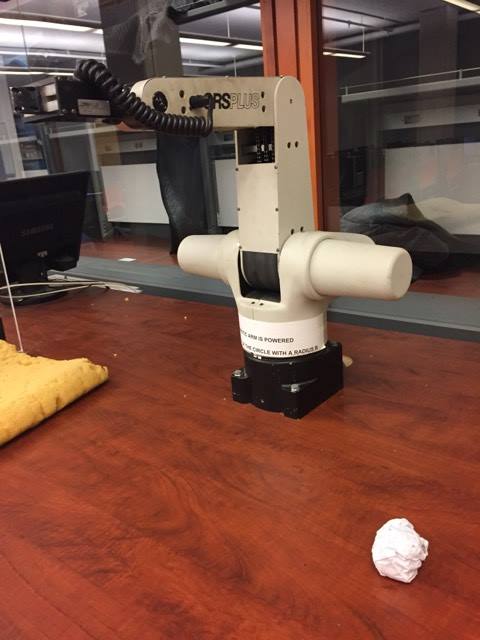
Applying the changes to the existent kinematic formulas involved figuring out the angles of the joints at any desired point in 3d space. Below are images with the instructions followed for computing such angles. To summarize, we were to take the given information (the arm lengths and desired position) and, assuming the wrist of the robotic arm is always perpendicular to the y axis, formed triangles, whose angles were then solved for.

The above image depicts the exact methodology to determine the location of an object below the robot base



The above image depicts the exact methodology to determine the location of an object above the robot base

One problem encountered was when factoring the base. It was assumed that the coordinate system used the robot base as its origin. Thus, the top of the base, where the shoulder joint is located, would be at coordinate (0,8,0), where the points reflect the X, Y, and Z coordinates respectively. Its important to note that the computations differ slightly when finding the angles for an object above or below the base of the robot. This is why the angles had to be computed twice.



Below are a few images diagrams to show the robotic arm at work. The first depicts the object we wish to pick up with the robotic arm alongside the actual arm being used. The object to be picked up Is a regular piece of paper crushed into a ball.



This next images shows that, just as in the previous assignment, the arm can move to a specified space and pick up the given object at that space.

With this solved, we then focus on the next component of angle handling. By this, I am referring to the angle the claw must be adjusted, in order to coincide with the angle of the object and pick it up with relative ease. To find such an angle, we take the individual object and using its pixels, create a covariant matrix which is then applied and manipulated to calculate the angle of the object. To do such, we use the lineFitplay.mm file as a template for computing the orientation of the objects

In terms of actually computing the orientation, we use the basic idea from the professor’s code, with our own implementation. We first create a covariant matrix, using the means of the objects seize, in terms of length and width. This matrix contains the squared mean of x, the square mean of y, the mean of xy. This is shown in the appendix below. Then, we apply the covariant matrix to the built in function, which effectively compute the shortest and longest line segment of the object. Then, we subtract the short line segment from the covariant matrix which results in a vector, perpendicular to the short line sector, or the direction of the long line segment.



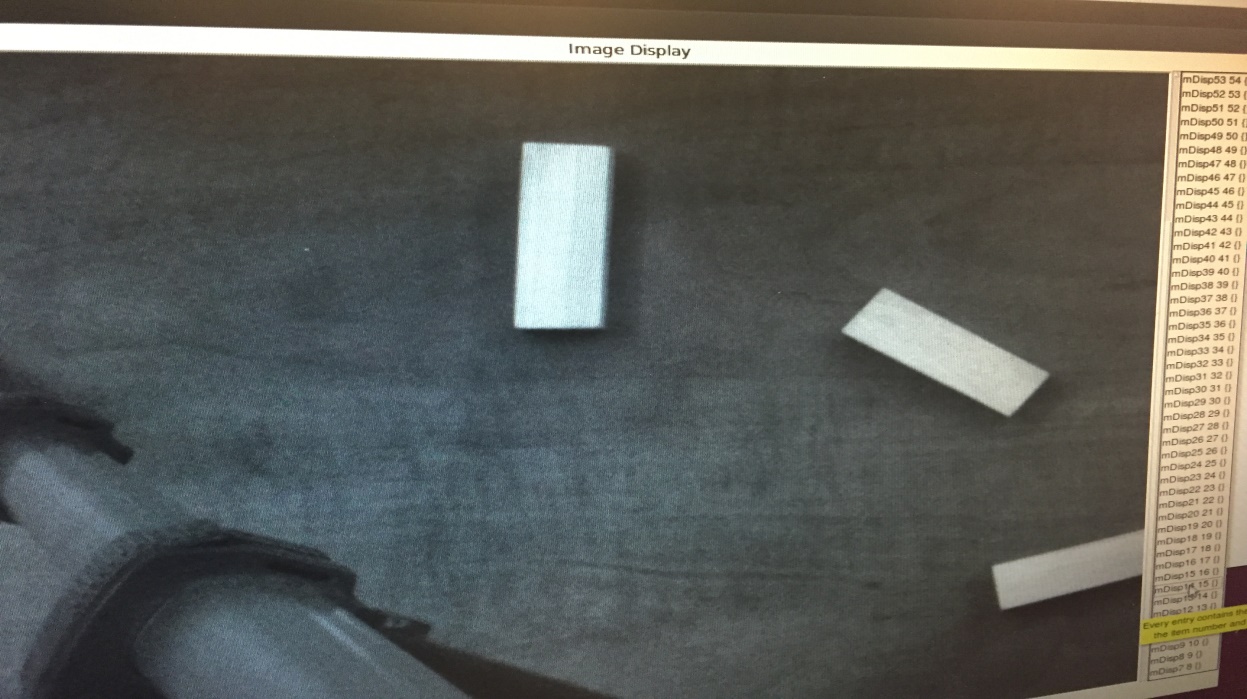
The image above is an example of the types of angles that are regularly computed by the function

Using this, we can rotate the wrist in such a way that the claw coincides with the side of the object we wish to pick up. Ultimately, this means that the robot will always pick up the object by its long side. All of our problems in this part of implementation were derived from the creation of the covariance matrix. This turned out to be harder than expected, to mimic that of the professor’s template, in a way that worked with dots representing points, but with full blocks of pixels of an object.

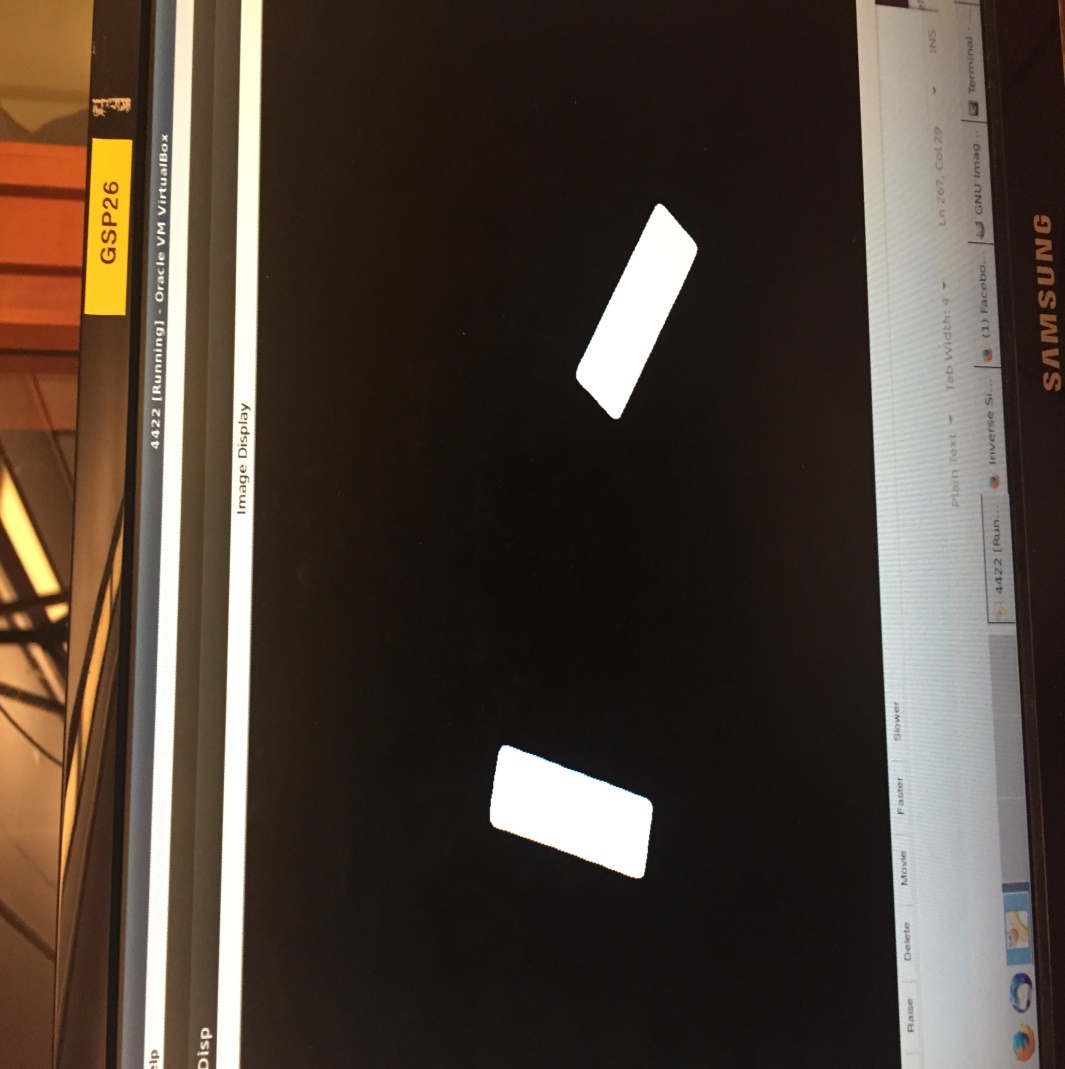
The second component completed is the component for detecting multiple objects on an image and distinguishing the individual objects, for individual use. The purpose of this is simple: The code needs some way of distinguishing the individual blocks so that they can be separately dealt with. By this I am of course referring to separately moving blocks to their destination. Programming this involved the programming of multiple subcomponents, so we broke implementation down into 3 steps. First was taking an image. Second was applying a mask to the image, to distinguish the objects clearly and effectively on the plane. The last part involved the actual task of separating the objects themselves.

First is taking the image. This was very clear cut and simple. We used the professor’s code for grabbing an image using the point grey camera, and saved the image taken for later use. In actuality, 2 images are taken. The first is just the field of view. The second is the same field, containing the objects to be picked up by the robot arm. We used a point grey camera for better manipulation of aperture and focus, and overall flexibility. At this point, the calibration techniques are used so that we can find the relationship between the image taken and the world coordinate system. This is discussed further in the next section, as to not get sidetracked with the implementations of the component.

Above it the first image taken, of the field

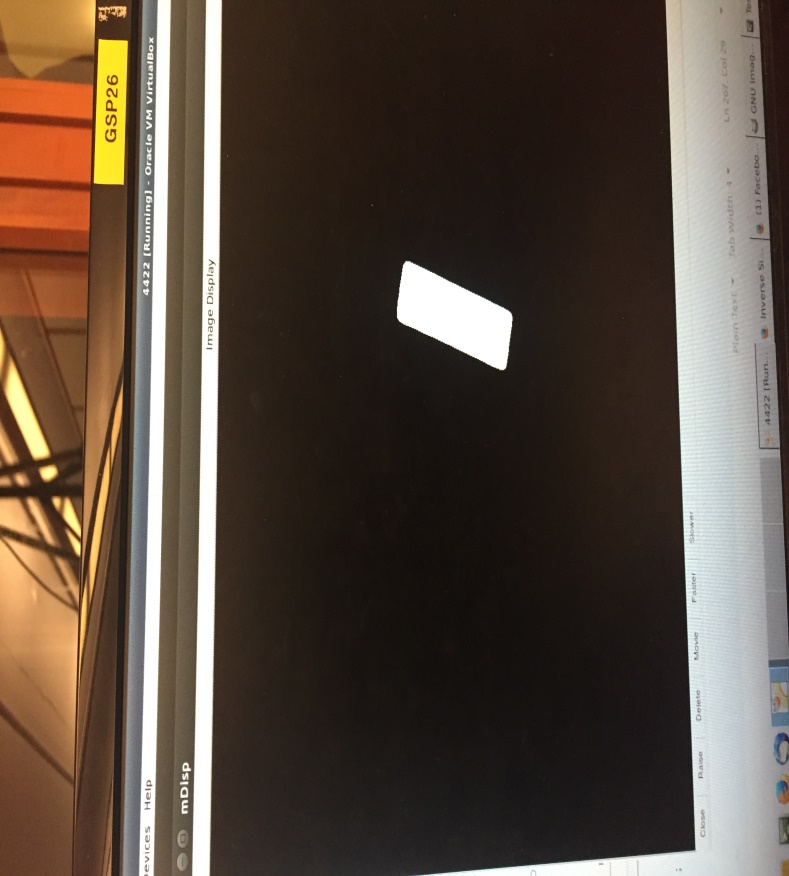


Above is the second image taken, of the field with the objects inside of it

Second, we apply a mask to the image taken, containing the multiple objects, so that we can better separate the individual objects in the last step. The point of obtaining a mask of the original image is to fundamentally have a way of isolating the location of the objects, and differing them from just the ordinary background. This is done by simply taking the 2 images from above ( ie one with and one without the placed objects), and subtracting the 2 images from one another. We then traverse through the image pixels and change any nonzero (or not black) pixel to 255 (or white). The resulting image is one that is entirely black and white, with the white spaces denoted the objects themselves.

The above image shows the mask applied to the image

The last part of building this component is now separating the white objects of the image, so that they can be individually handled by the overall program. Essentially, the robot arm needs a way of targeting one jenga block at a time, so isolating each individual object proved to be an effective method to do so. To isolate these objects, we used the connected components method of Medimath. This function traverses through an image and finds the white pixels. Once a white pixel has been found, it changes the pixel to a value, say 1 for example. Then, it finds all connected, or adjacent white pixels and consequently changes them to the same value as the original, until all connected white pixels, or the pixels of the individual object, have been found. Now, the entire object will be of the same value or intensity. Afterwards, the traversal continues. The same thing is done when another white pixel is found. It and all its connected white pixels are changed to a value. However, this value is set to one that differs from the first object. After the entire image has been searched, we are thus left with an image in which the individual white blocks are replaced with differing colours. We cannot show this because the differing colours are all still so close to black (0), that they would not be distinguishable.

****With this subcomponent complete, alongside the other 2 mentioned above, we effectively now have a methodology for isolating multiple objects within an image. To summarize, it works by taking 2 images, 1 with and one without an object, applying a mask over it, and then using the connected components function to differentiate between them. By applying this, we can distinguish between the jenga blocks, and apply functionality to each individually. This of course refers to picking it up and placing it in the correct end location, without the worry of disruption from the other blocks, in any way.

The image above shows the result of isolating the blocks of the previous image. It is now separated for individual use

Implementing this component involved solving many troublesome problems along the way. The biggest of the problems was the actual implementation of what connected components does. Before using this function, we had developed our own recursion based method in which, upon finding a white pixel, we applied depth first search to it to find all other white connected pixels. The problem with this function was that there is only a certain number of recursive calls the function is allowed to make, and this, after only obtaining a piece of the object, we would hit a max\_recursion error. We implemented a non-recursive way around this, involving the implementation of our own stack and methods, but ultimately, the far more efficient option was the simple and easy to use connected components.

Last, but not least, we now turn attention back to the calibration of the image, explained very briefly above. The point of the calibration is to, given an image, find the relationship between the image and the world coordinate system. In doing so, we can effectively obtain coordinates from the image taken, that can be applied to the robotic arm’s movement and reach. In other words, we can obtain the location of the objects from the image, and convert them to coordinates the robotic arm can move to, in order to reach the block. In doing this, it is assumed that the individual blocks are not stacked on top of one another, or that they are all on a level field

The calibration was done as such: First, we took multiple images for the actual calibration process. The more pictures taken, the better he accuracy of the resulting transformation matrix. Afterwards, these images were applied to the camera calibration toolkit in Mat Lab as to produce effective components for the actual transformation from the image to world coordinate system. After we obtain the extrinsic components, we need to recalculate the function as to suit an image-to-world structure. This result is then applied to the image coordinates whenever we wish to grab another object. We obtain the image coordinates by finding the center of mass, after each object is isolated.

With all the components created and explained, what’s left is to show how they are all used in the overall program, and the results of applying them. Effectively, we combine all of these components together to create a program that takes an image of the field, finds the jenga blocks within the field, picks them up, and then stacks them in a manner such that they stay level. We begin by using the component for multiple object detection. Using this component and its subcomponents, we effectively obtain a calibration matrix for the camera’s view, and take the images of the field, one with and one without the jenga blocks for stacking. Then, after isolating the objects, we apply the calibration matrix to each one, as to find their world coordinates, in terms the robot arm can use. After this, we apply the robotic arm component, which, when fed the world coordinates from the first component, moves to the location specified and pick up the jenga block. From there, we simply run a function to move the blocks to a specified area, where it begins stacking them.



The image above depicts the program running in mode 1, where the blocks are simply stacked one by one, on top of one another



This above image depicts the program running in mode 2, in which the blocks are stacked in the way they are usually stacked when playing Jenga