A.I. LEGO® Sorter

Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree in Bachelor of Science
in
Robotics Engineering
By

Andrew Schueler

Peter Donaldson

David Jin

Date: March 22, 2019

Project Advisors:

Professor Craig Putnam, Professor Brad Miller

A.I. LEGO Sorter Major Qualifying Project

Andrew Schueler, Peter Donaldson, and David Jin

Abstract

The goal of this Major Qualifying Project is to develop a robotic system to autonomously separate, identify, and sort a multitude of LEGO pieces. The solution developed is a three-part sorting apparatus which utilizes complex mechanical design, computer vision (CV), and convolutional neural networks (CNNs) to serialize, classify, and distribute hundreds of unique part combinations. The completed mechanism is capable of processing a large input of unsorted components and fully sorting them by user-defined metrics.

CONTENTS

List of Figures

I. EXECUTIVE SUMMARY

The primary challenge of sorting LEGO bricks is the sheer volume and diversity of parts. Typically, individuals are unable or unwilling to spend the time and effort required to sort them by hand [24], resulting in large amalgamated buckets containing thousands of parts from unique sets. The system developed in this MQP will be capable of sorting bricks autonomously, eliminating the need for human involvement. Prior research shows only one established project that attempted the sorting of LEGO bricks using artificial intelligence [12]. This project used pneumatic actuators to physically move and sort parts. The posted description did not include documentation for the sorter's physical design or the neural network implementation.

The final product consists of 3 main components: the serializer (Module 1), the classifier (Module 2), and the distributor (Module 3). The modules are coordinated through the ROS software ecosystem. ROS, or Robot Operating System [19], is an abstraction of communication for actuators and sensors and controllers in a robotic system. This means that programmers do not have to worry about writing socket client/server [16] programs and networking protocols [15]-[22] while programming and testing a robot. ROS allows for modular software that enables roboticists to add more sensors, actuators, or more robots into a software ecosystem [20].

The serializer takes an arbitrarily large amount of LEGO bricks and outputs a stream of evenly spaced parts to a conveyor belt where cameras can identify and classify the parts. The second module houses the cameras utilized in the localization and classification of parts as they traverse the conveyor. A combination of 3D scanning and prior training of the neural network will allow the classifier (Module 2) to classify parts from any angle and orientation. Due to the extensive possible combinations of unique LEGO parts available (more than 7,000 unique pieces) [14], the current system focuses sorting a smaller subset: in the future, the AI software can be trained and updated to classify a larger batch.

Some LEGO bricks, including LEGO Technic parts (a subsection of LEGO), come in widely varying sizes and shapes. These will not be included in the sorter's knowledge base until the mechanics of the sorter are able to support parts that are more difficult to contain. The AI's classification set can be expanded to any number of parts, but requires individual training for each new addition. This means that a system capable of sorting 100 different types of LEGO bricks will need to be trained with a knowledge base for each of those parts. The target volume will expand and contract as the sorter goes through further testing. Once passed through the conveyor system, the classified parts enter the mechanical distributor, which uses their classification to organize them into an assortment of storage containers. Each container will store one category, which can be specified by the user. These sorting categories can utilize any combination of a brick's part identification number or color. If there is a part that does not fit into any predetermined category, then it will be sorted into a separate container with no specific identifier.

II. INTRODUCTION

This project was chosen for its reach into the three major subtopics of robotics engineering: electrical engineering, mechanical engineering, and computer science. The intricacies of the physical design, including the sorting system that will evenly distribute multiple parts consistently, and the end manipulator, which is used for complex kinematic actions, make for a challenging mechanical endeavor. All of the system's major components require custom circuit design and power management. In terms of software design the system utilizes artificial intelligence/neural networks and computer vision, two topics that hold particular interest for the MQP group. The practicality and value of this design as applied to the real world, both for the MQP team, and for larger applications in industry, show the innate value of an AI powered sorting mechanism. This project also allows for expansion into other fields of sorting and classification. LEGO bricks were chosen for this application due to their unique shapes and colors, as well as their high availability. Although sorting LEGO bricks is not an industrial problem, this type of technology can be implemented in multiple other domains of industrial engineering. For instance, the sorting of bolts or sockets in a tool shop would increase productivity and decrease routine human labor. However, for the scope of this project, LEGO bricks provide a reliably diverse data set for color, size, and shape.

A. Inspiration

Each member of this group was brought up with LEGO. They introduce many young students to engineering challenges and mechanical design. However, the variety and quantity of the parts make both sorting and organizing them a momentous task. For large-scale applications, like workshops and schools, where LEGO toys are used to teach students the basics of robotics and mechanical engineering, extremely large quantities of LEGO bricks must be collected and placed back into kits and sets at the end of a session. This fact introduces the challenge this project seeks to rectify.

III. BACKGROUND

Before commencing the design phase of this MQP, the team conducted research into similar applications of hybrid AI/mechanical sorters for processing small parts. In particular, the research focused on current LEGO brick sorters. The topic of sorting LEGO parts, due to their universal presence in homes and schools, has been a popular topic for engineers in the LEGO community.

Many projects have been conducted, some successful, that try to fully automate the process of serializing and classifying these parts. The research conducted covers some of these, focusing on those that attempt to use computer vision and artificial intelligence to classify bricks, rather than purely mechanical sorters.

IV. DESIGN OVERVIEW

The sorting system envisioned to solve this complex issue consists of three major components, which will be referred to as Modules 1 through 3 (see Figure 1). Each of these is a unique electro-mechanical system designed to carry out a specific part of the sorting process.

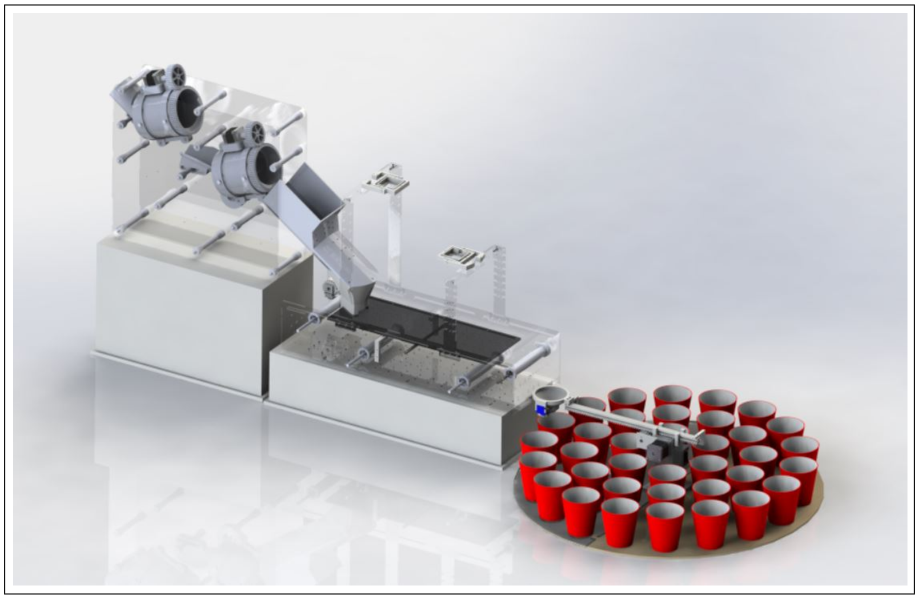


Fig. 1: Full System: Modules 1-3

A. Module 1

Module 1 is the serializer (see Figure 2). It is designed to take an input of multiple unsorted LEGO parts of variable size and color, and process them into a single stream of parts, the speed of which is configurable through software.

On the software side, Module 1 utilizes a combination of ROS messages to signal the movement of parts falling through the module's serializing tumblers, and to advance parts according to requests provided by Module 2. Module 1 also responds to a timer on the main controller, providing single parts to Module 2 at a defined interval of 10 seconds (if another request has not already been received).

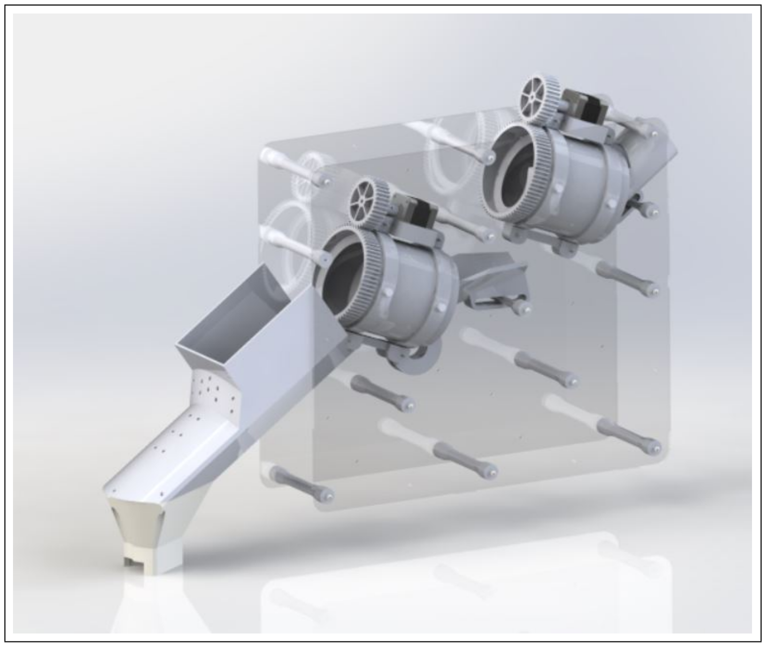


Fig. 2: Module 1: Serializer

B. Module 2

Module 2 is the classifier (see Figure 3). It is within Module 2 that the bulk of the sorting process takes place. Through the use of computer vision, multiple neural networks (the design of which is covered below), and a granular-control conveyor belt system, this module carries parts from the output of Module 1, through a classifying camera fixture, to the input of Module 3.

In terms of software, Module 2 communicates through ROS with both Module 1 and 3. It signals to Module 1 when it is capable of processing a new part on the belt, and it alerts Module 3 when it has completed the classification of the previous part (which indicates that Module 3 now must deliver a classified part to its respective container). Module 2 also delivers image messages from the classifying camera to a neural network classification node on the main controller, receiving in return the classification of the unknown part.

C. Module 3

Module 3 is the distributor (see Figure 4). It takes as input the classification (by color and part ID number) of the incoming LEGO part, and the part itself, and then uses polar positioning

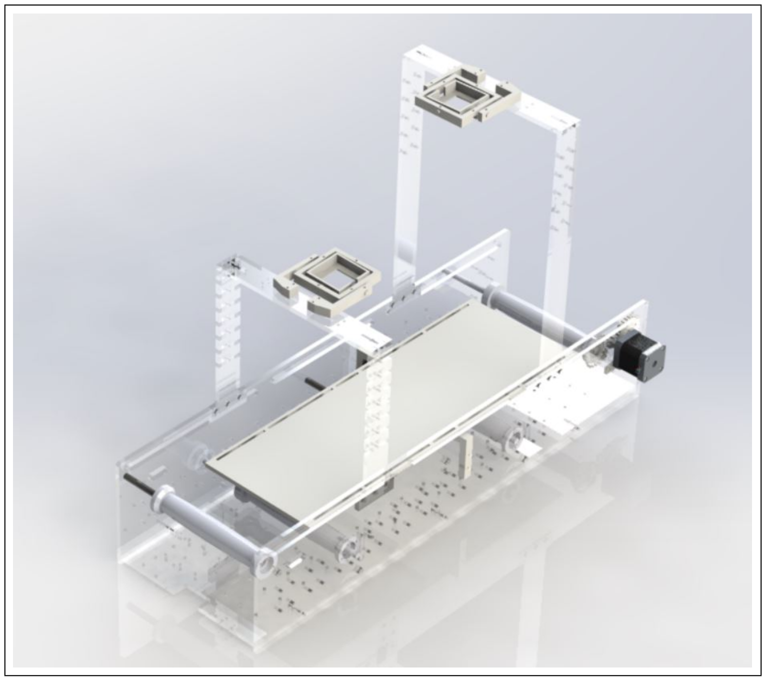


Fig. 3: Module 2: Classifier

to locate the appropriate storage container for the specific classification. It then transfers the part to this position and returns for the next brick from Module 2.

Module 3 is controlled by a path generator or controller on the main board. It receives an alert signal from Module 2 to signify the presence of a new part, and a message containing the part's classification. The path generator then takes this information and provides the motor commands for Module 3 to deliver and drop the part into its designated holder. Then Module 3 returns to its original position and signals to Modules 1 and 2 that it is capable of processing a new part.

The three modules communicate with each other via an event-driven operating system. Each module is capable of sending signals and triggers to the others via the main controller in order to ensure a smooth flow of parts without backups or delays.

The next sections describe, in detail, each of the systems major components, starting with the design process for the neural networks implemented in the classification portion of this project.

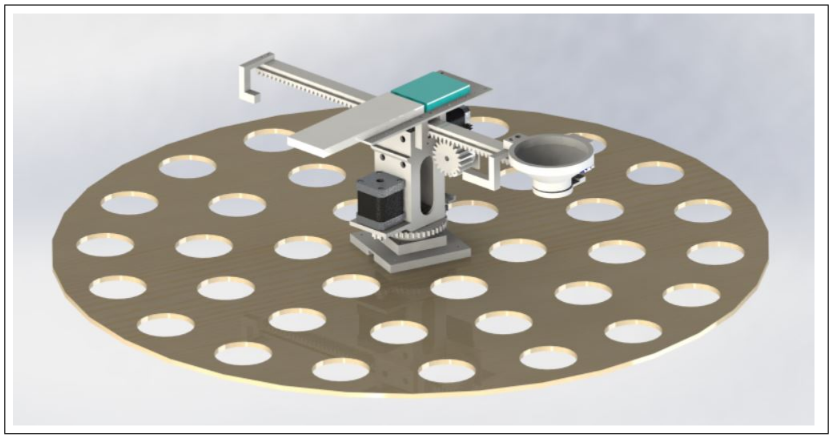


Fig. 4: Module 3: Distributor

V. NEURAL NETWORK

The driving component of an artificial intelligence-based sorting solution is, as expected, the AI. The strategy for the design of the neural networks used in Module 2 involves three major steps. First, construct two networks: one that can be used to identify features on a high resolution image of a part and classify it from any angle, and then a second that can accurately distinguish between similar colors using a low-resolution image. Next, collect enough data so that any part can be identified from any angle or orientation after proper training. Third, incorporate these networks as classifiers to Module 2 and tune the lighting/camera positioning to replicate the collected training data. By completing these steps the classifying module will be able to identify any part in the sorting subset by its two defining metrics: part ID number, and manufacturer-defined color.

A. Data Collection

The first step in designing a classification neural network for many distinct classes is ”good” data. ”Good” data is defined by its quality, both in the physical quality of the training images, and in the sufficient variation of the full set. As determined in early research, no data is currently available for the classification of LEGO bricks that will satisfy the requirements of this system, especially considering the expected variations in the part’s positions and orientation when they pass under the classifying camera. This necessitates a customized solution, one that can capture vast amounts of visual data in high quality, with similar background detail and lighting, that is universally representative of the parts being classified at any angle or orientation. As a result of this revelation, the design for the Scanner was conceived.

1) Initial Efforts: The initial design for the classification network was a fully-connected neural network (see Figure 5) that utilized procedurally generated images from the internet to train itself in part identification. This process required the construction of a network, scripts to identify relevant data for each type of part, scripts to capture said data and save it for use by the network, a procedure to complicate the saved data to account for unpredictable lighting, angles, and camera changes, and finally, a script to automate the training process. This would allow the network to autonomously teach itself how to properly identify parts through Google images' massive existing database.

This strategy was first implemented on non-LEGO bricks, as a proof of concept. The network was designed with 3 fully connected layers and configured to classify 4 categories of images: cats, dogs, cars, and houses. A separate script was written to open a web browser and search for the particular object within the browser's image field. The same script then utilized the browser's developer options to collect the cURL values (a command line tool for transferring data via URLs) for each image, and enter them into the host system's command line to be downloaded individually to the respective classification folder. Once completed for the 4 major classes, this data was divided into training, test, and validation data, and run through the network. The results of these tests were acceptable, but not outstanding. Accuracies were noticeably low (around 80 percent), and mistakes in classification were more common than was acceptable within the scope of the MQP.

2) Data Variance: The issues present in the fully-connected approach amplified themselves when applied to LEGO pieces. The first issue was immediately clear. When using Google's search engine to identify specific parts by their part ID number, there is no way to guarantee an even distribution of colors and angles for parts. Tagged images of these parts often originate from a retailer's website, making it unlikely that the angles the pictures are taken from are sufficiently random to train a network for classification at any orientation. A possible solution to this is a data variation script, "Variate.py", which was implemented to alter and warp the images to provide the network more information than previously possible. Through the use of hue and saturation filtering, mirroring, cropping, resizing, and warping these images, 20 separate images could be generated for each single image fed to the network (see Figure 6). This had a marginal positive effect on the final network's accuracy.

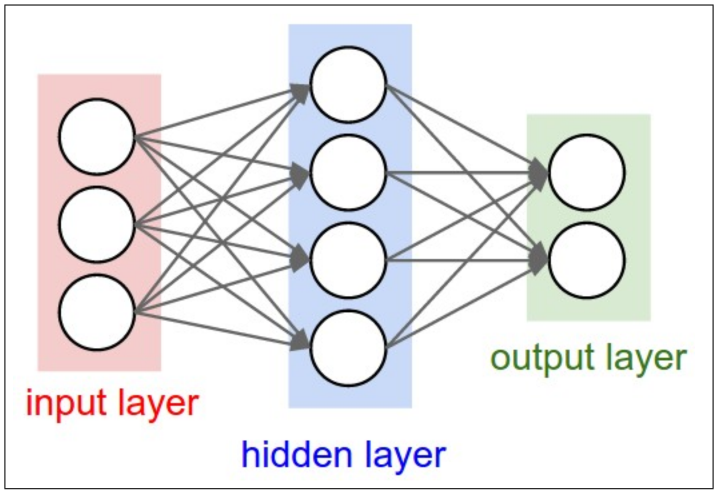


Fig. 5: Model of a Fully Connected Network

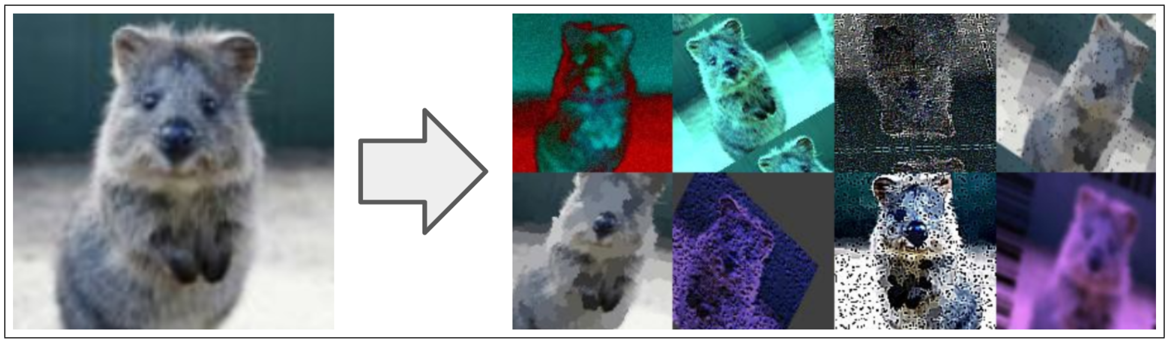


Fig. 6: Data Variance for a CNN

The next major issue with this approach is that the volume of images for each individual part to be categorized changes dramatically from part to part. LEGO's famous 3001 piece (a common 2x4 brick) has a vastly larger image database than other less common pieces. This caused an issue when trying to identify LEGO bricks with slants and curves, as the availability of images for these parts is significantly lower.

3) Version 2: This initial design stage helped to highlight some of the major targets for the next network developed. First, the data would be generated by hand, on a proprietary system. This would guarantee image quality and integrity over enormous data sets, and would also allow customized angles and lighting for best-case scenario classifications. Second, the network would use a structure geared specifically towards feature recognition. This is vital, as the only way to distinguish between LEGO parts is through the quantity and position of their individual features. Dogs, cars, cats, and houses are all fundamentally different enough for identification in a fully-connected network. LEGO pieces, however, have many similarities, and require more complex computer vision-based learning for accurate classification. Finally, the size of the training database would be dramatically increased, so as to guarantee that any part classified would have a sufficiently large training set (something that is difficult to ensure when utilizing an existing set such as one obtained through Google search). This would also necessitate the use of hardware acceleration for training, as the large datasets would require more time and processing power for efficient training cycles.

4) The Scanner: The Scanner is a custom piece of hardware than can be used to consistently replicate the view of the camera on Module 2's conveyor belt (see Figure 7). It is designed as a rotating platform, made from the same material as the belt used in Module 2. The platform rotates under the view of a camera, which is fixed at a specific angle and distance from the platform's rotational center. Parts are placed in the center of the platform as it spins, and a separate script captures images of the part from the camera's view. This continues for a specified number of images (which is equivalent to a certain number of rotations). The part is then flipped to one of its other possible landing orientations and the process is repeated. This is done for every possible landing position, of every possible color, of every possible piece being classified. All of this data is categorized in folders and saved to a hard drive dedicated to data collection.

Mechanically, the Scanner is fairly simple. All of its components are 3D printed using either ABS or PLA plastic, and the cover for the spinning platform is a disk cut from a scrap section of the sanding belt used on Module 2. The base platform has a cutout for a Nema 17 stepper motor, which connects to a sequence of two bevel gears, meshed at 60/30 degree angles. The second of these gears is connected to an axle mounted in the base, which is pressed into the platform above it. The speed ratio of the stepper motor to the rotating disk is 2:1, which allows for more torque (which reduces vibration), and more granular control. On one side of the Scanner's main platform is a mount for an articulated arm, which includes t-slots to secure itself to the main body of the Scanner. The three joints in the arm are adjustable, and connect to a terminating camera mount. The mount and attached camera can be positioned at 45 degrees to guarantee all parts are distinguishable from any orientation. This mount was initially designed for use with the Stereolabs Zed camera, but was later repurposed for use with the Picamera V2.

Zed Camera:

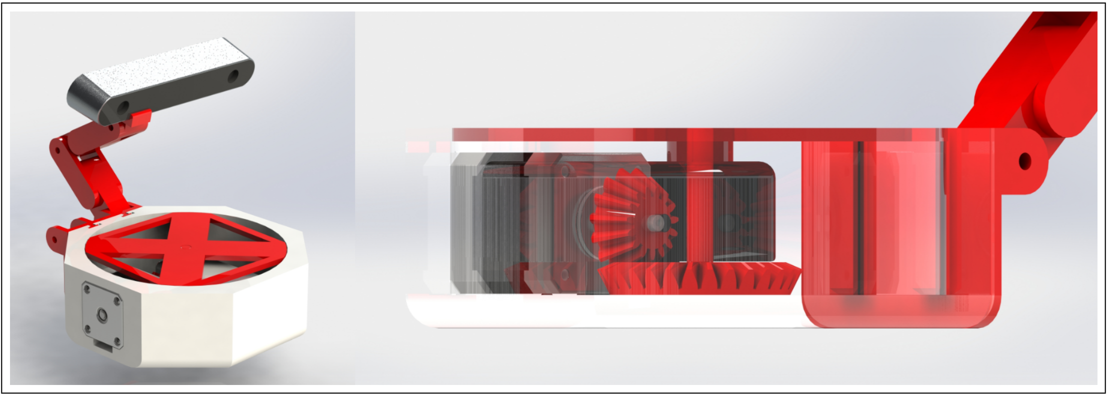


Fig. 7: Renderings of the Scanner

The Scanner was initially implemented to use Stereolabs' Zed camera, a dual sensor camera with depth sensing capability. This was changed in testing after the discovery of the Zed's high minimum focal distance, which prevented usable data captures at any distance below 10 inches. Data collected within this focal range would be blurry, and data captured outside of it would be of a resolution too low to be useful. This fueled the decision to change cameras entirely and adopt the Picamera V2.

Picamera V2:

The Picamera V2 is a proprietary camera module designed for the Raspberry Pi (see Figure 8). It has an 8 megapixel sensor, and interfaces with the PI via a ribbon cable. Most importantly, the lens is adjustable (with the use of an aftermarket tool), and the sensor is capable of macro-focus. This allows for highly detailed stills to be captured at very close range (see Figure 9). This is ideal for this application as small details and features on individual LEGO parts are easier to identify in high resolution.

5) Scanner Software: The Scanner operates on a Raspberry Pi 3b running Robot Operating System (ROS) as a networking tool for transferring images to the main controller (a Jetson TX2 developer kit serving as a small form-factor Linux pc) where all the data is stored and organized.

This project utilizes various code repositories for the individual components of the Scanner and sorter system. The Scanner's key repository is named ros_Scanner_pi. It creates a ROS node to capture images from a Raspberry Pi camera and publish them to ROS. The system also utilizes a repository called ros_Scanner_not_pi, which is tasked with handling the logic for receiving the Pi's image data and saving it to a local directory on one of the device's drives. The ROS

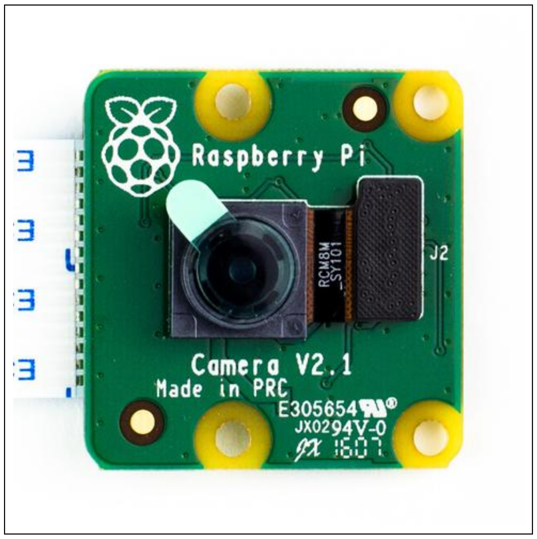


Fig. 8: Pi Camera V2

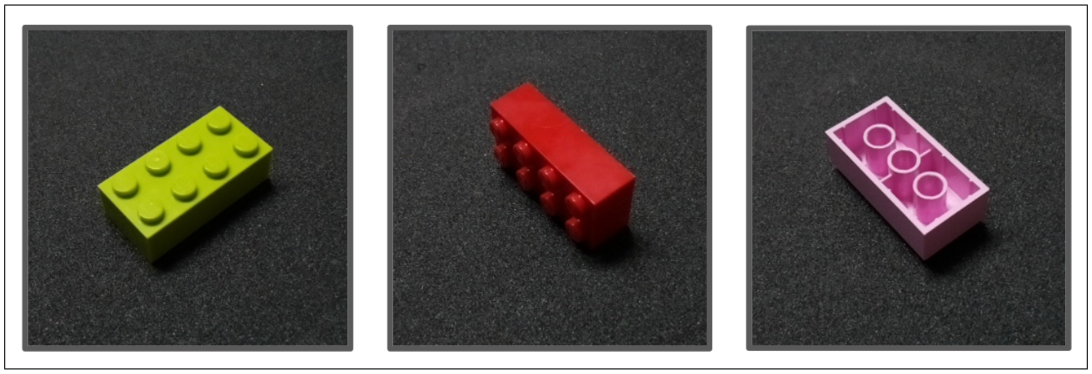


Fig. 9: Macro-Focus Image Capture on the Picamera

node provided in this package can run arbitrarily on any Linux device, provided it has a hard drive or solid state drive (SSD) with enough space to hold all of the pictures, and networking capabilities (which is a guaranteed with ROS and the appropriate ports).

ROS has a metapackage (a special type of ROS package which contains one or more related packages with shared dependencies on core functionality) to handle transporting images as messages called image_common. This metapackage contains a package called image_transport, which allows a user to capture raw images on the Raspberry Pi and deliver them, as a ROS message, to another ROS node (in this case the Scanner node). The package supports raw images, jpeg compression, and other compression formats. This abstraction of the ROS message system allows the user to manually configure the compression parameters of their choice. The ROS publisher for the Scanner's image topic automatically publishes to several compression topics handling these formats. In this case, the Jetson controller subscribes to the raw format, and the publisher only publishes raw images, while the package makes the other compression types available to the user.

Originally, Python ROS nodes were to be used for the image capture, as most other nodes are written in Python and the current software ecosystem revolved around the Python programming language. However, research showed that Python was not officially supported by the image_common metapackage, meaning image nodes would need to be written in C++. This necessitated that the Pi's camera also be controlled in C++. Although it is possible to link Python scripts to image streams, current methodologies are only capable of manipulating and republishing images, and do not possess the ability to behave as the image source. Future implementations of such a vision system could theoretically utilize only Python, but documentation for this technology is limited, and current C++ ROS nodes function amicably. The Pi camera does not have an official C++ library, so a 3rd party library maintained by Universidad de Córdoba [5] was used to interface with it. The implementation of this library allows the Jetson controller to treat the Pi and subsequent camera nodes as a USB webcam, simplifying the data collection process for image topic subscribers. The node receiving the image is then abstracted so that saving image data simply requires reference to a directory the script has permission to edit. The most significant drawback of this abstraction is latency. The publisher for the Pi camera averages approximately 3.5 frames per second, meaning the average scan time (collecting 1400 images) is between 7 and 8 minutes. This time can vary based on part size and color, as each image capture process can take longer. When the scan completes, the terminal running the operation reports the event, and the user then runs it again with a new part configuration.

6) The Collection Process: In order to collect data, the Scanner is placed in a well-lit room with few shadows. Windows are covered up to mitigate changes in lighting and sudden fluctuations in color. In order to ensure that the parts are sufficiently lit, an LED ring light around the Raspberry Pi camera is dimly lit to add a soft white light, brightening shadows cast by the part on the Scanner belt. The Scanner is turned on, along with the Raspberry Pi and connected Pi camera. The ROS node created to capture and send images is also initialized, along with the Jetson TX2 controller, which is facilitating the data capture. A part is placed in one of its possible landing positions (in the case of the 3001 or 2x4 brick, this can be any of the brick's six faces) in the center of the rotating platform. When the data collection script is activated on the Jetson, the Pi begins to collect photos. 740 images are collected before a flag is triggered, the platform stops, and the script terminates. The part is then flipped, and the process can begin again on a different side of the brick. Most bricks have three possible unique landing positions, but some have as many as five, and others as few as two. The whole process is carried out for each color of each brick. This is a very lengthy process, but it guarantees extremely high quality images for any and all orientations of each part. The collection of these images was completed over the course of two weeks by members of the team working in shifts.

7) Data Organization and Storage: The training process described above, when applied to all parts in all colors at all possible angles, yielded approximately 720,000 individual images, totalling roughly 300GB of storage at 256x256 pixels per image. These images were categorized first by part, then by sub-directories of each color, and then by further directories for each possible landing orientation (see Figure 10). In order to translate this data from its raw state to directories and data that the networks can utilize as training input, multiple scripts were created. Rather than manually transfer, process, and resize data for the network's usage, these scripts automate the process saving time and limiting organizational mistakes.

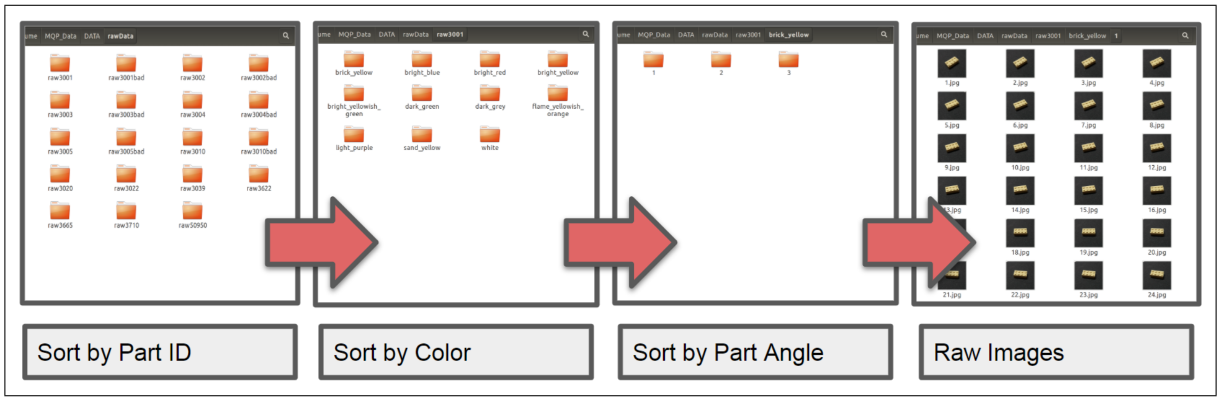


Fig. 10: Raw Data Storage and Organization

The first script to run is "resizeImages.py". This script is responsible for three things. First, it cycles through all of the images for all colors and all angles of a single part. For each of these parts it binarizes the image using the CV2 python library, and finds the centroid of the part. It then centers it in the middle of the frame, and mirrors the sides outwards, to account for any cropping that had to be done. This image is then cropped to a desired aspect ratio (1:1), and resized to a 128x128 pixel image with the part directly in the center (see Figure 11). This is the same process that occurs when a single part travels under the classifying camera on Module 2.

This image is then transferred to another part-specific folder in the ”resized” directory, which contains data that is formatted to be used by the network. This process is then repeated until each part has its own ”resized” folder. This data can then be organized into training, testing, and validation data to be used in training the neural networks.

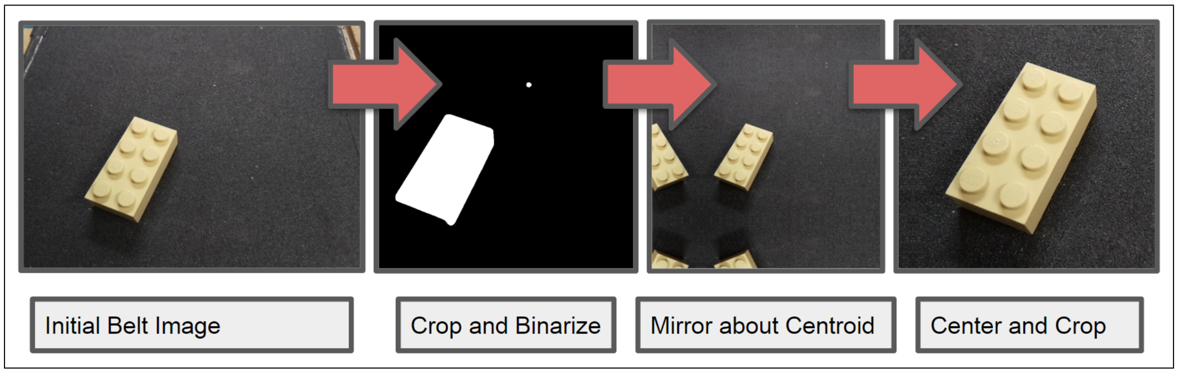


Fig. 11: Data Pre-Processing for Serial Classifier

The next script to run is the ”CreateData.py” script, which takes the resized data and distributes it into the training folders for the network. It uses a user-defined split of 70% training data, 20% testing data, and 10% validation data, splitting random images off into these folders until it has completed the whole set for a single part (see Figure 12). This is done in each of the training, testing, and validation folders, so that each contains an equal number of random images for each LEGO piece.

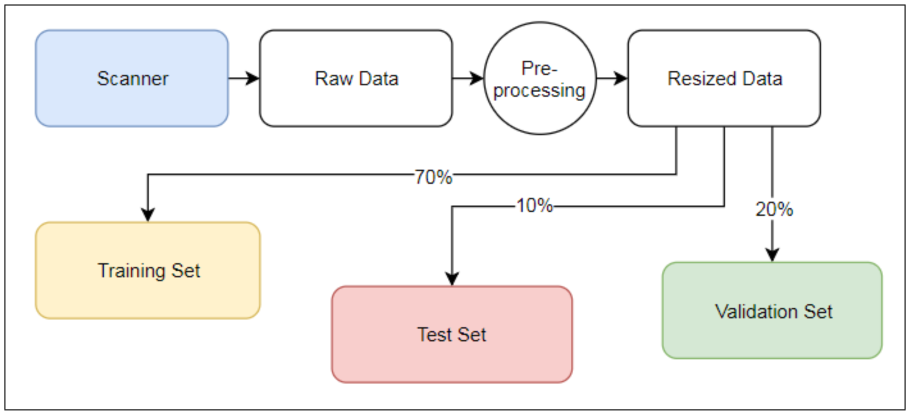


Fig. 12: Image Data Flow

There are two different versions for both of these scripts, one for creating part ID data, and the other for creating color data. The difference is the color data scripts distribute the data into folders by color rather than the part's ID number, and also resize them to a mere 16x16 pixel resolution, as a high resolution image is not necessary for color classification and only hurts turnaround time for training and classifying (see Figure 13).

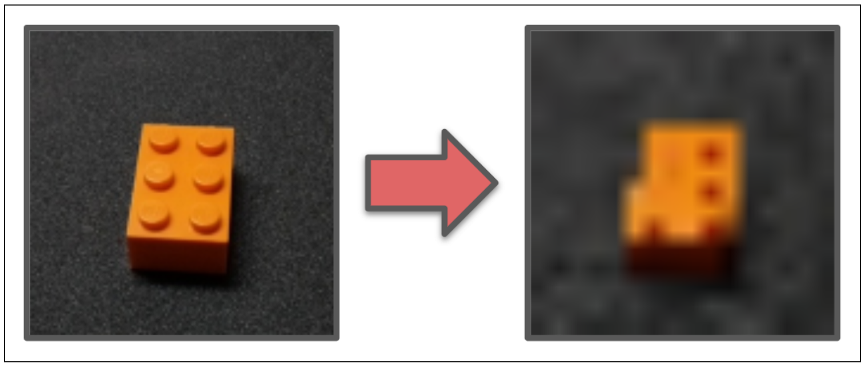


Fig. 13: Data Pre-Processing for Color Classifier

B. Network Design

1) Software: In order to develop the classification networks for both the color and serial classifiers, Keras, an open-source neural network library for Python, was utilized. Keras uses higher-level abstraction and a more intuitive interface to allow users to develop deep-learning neural networks using the TensorFlow API, another open-source software package for numerical computation and machine learning. Keras supports the design and tuning of multiple types of networks. The classifiers used in this system are both convolutional neural networks. The convolutional neural network, or CNN, is a type of fully-connected network used specifically for analyzing visual image data. In order to improve the performance of network training for this project's large data set, Keras was also configured to run in tandem with Nvidia's CUDA platform. This enables hardware acceleration for libraries like Keras, utilizing video ram (VRAM) on a dedicated graphics card (GPU). In this case all training was run using CUDA 9.0 and an Nvidia GTX 1060 GPU with 6GB of dedicated VRAM. Installing CUDA yielded an estimated 50 percent time savings on large training runs, enabling the use of much larger datasets. This allowed the team to train networks on the full dataset of roughly 720,000 images without outsourcing to large computing clusters.

2) Convolutional Neural Network: A traditional fully-connected neural network classifies images by employing layers of neurons, which function like the synapses of a biological brain, to process and relay input signals to other neurons. Although similar to its fully-connected counterparts, a convolutional neural network differentiates itself in a number of ways. The primary difference is the use of convolution filtering on the input image matrices. Although it uses the same input structure as a fully-connected network, requiring one neuron for each of the image's input pixels, it treats the intermediary (hidden) layers differently. The CNN enacts a style of perceptron (a mathematically-modelled neuron) known as ”multi-layered perceptrons” to identify and classify the features of an image. This is done in contrast to the fully-connected network, which treats the whole image as a single feature. The convolutional filter is designed to replicate a more human, or animal, style of visual classification, using many more limited, sectioned views (kernels) of the image at a time. Each of these convolved sections is then used to generate input array for the next hidden layer, which behaves the same way, scanning the image with a particularly-sized kernel and weighing each kernel against the learning database sequentially (see Figure 14). CNNs also enable pooling layers, which are utilized to combine the outputs of the previous hidden layer into a single input for the next. This allows for a reduction in computation for the next hidden layer's neurons, as well as progressively reduces the spatial value of the data being processed.

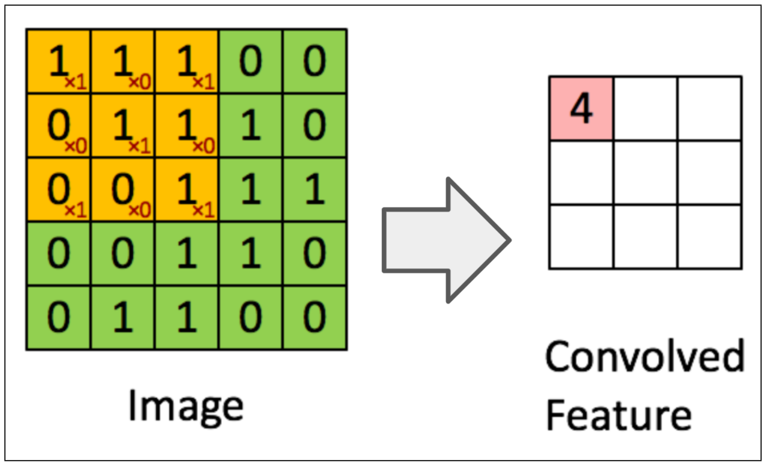
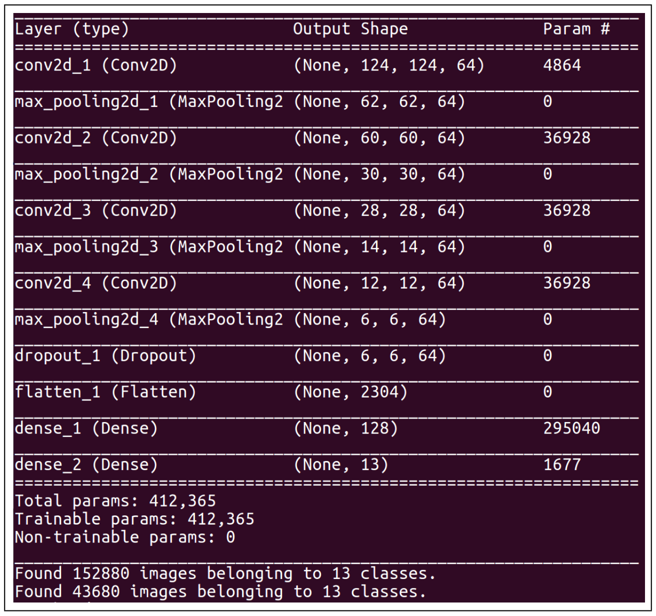


Fig. 14: Convolutional Filtering

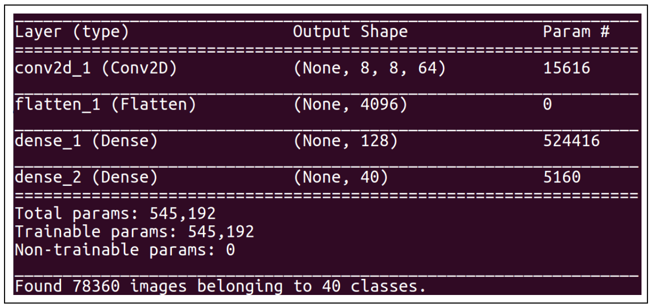
3) Part ID Classifier: The part ID classification network utilizes four two-dimensional convolution layers (see Figure 15). The first of these utilizes a larger convolution kernel of 5x5, whereas the following layers utilize a 3x3 kernel. The input shape of the first layer is 128x128x3 (the resolution of the input image where every pixel has an R, G, and B value). All of the network's hidden layers are separated by two-dimensional max pooling layers. A max pooling layers is a special type of pooling layer which reduces sections of a matrix to their maximum values. The classifier is terminated by a fully-connected layer with only 128 inputs and a final dense layer with outputs proportional to the number of input classifications. This network utilizes a dropout coefficient of 0.4, which is relatively high for an image classifying CNN. This ensures that the network does not overfit the data across extended training runs and multiple epochs (completion of the training process for the given data). Overfit networks are trained so aggressively that they struggle to classify images outside of their training sets, even if they are visually similar to the trained images. Due to the large data set, the network's learning rate is also fairly low, 0.01, which means it reacts more slowly to rapidly changing feature-weights (which can occur during extensive training). This helps the network train effectively over the entire data set.



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 124, 124, 64)      4864
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 62, 62, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 60, 60, 64)        36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 30, 30, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 28, 28, 64)        36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 12, 12, 64)        36928
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 6, 6, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               295040
_________________________________________________________________
dense_2 (Dense)              (None, 13)                1677
=================================================================
Total params: 412,365
Trainable params: 412,365
Non-trainable params: 0
_________________________________________________________________
Found 152880 images belonging to 13 classes.
Found 43680 images belonging to 13 classes.
```

Fig. 15: Keras Network Structure (Part ID Classifier)

4) Color Classifier: The color classifier is very similar to the part ID classifier in design (see Figure 16). The main differences are the input shape, which is only 16x16x3, and the removal of three two-dimensional convolution layers. These are replaced with a single convolution layer which utilizes a larger 9x9 kernel. This is necessary to facilitate the averaging of pixel color values, which helps more accurately classify one of the colors in the target classification set of 40. The max pooling layers are also removed as they are not needed (since only one convolution layer is present). The dropout factor is also removed, as the colors tend to overfit significantly less than the part IDs. The color classifier utilizes the same activation functions and learning rate as the part ID classifier.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 8, 8, 64)          15616     
_________________________________________________________________
flatten_1 (Flatten)          (None, 4096)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 128)               524416    
_________________________________________________________________
dense_2 (Dense)              (None, 40)                5160      
=================================================================
Total params: 545,192
Trainable params: 545,192
Non-trainable params: 0
_________________________________________________________________
Found 78360 images belonging to 40 classes.
```

Fig. 16: Keras Network Structure (Part Color Classifier)

C. Training

1) Training and Testing the Serial Classifier: The training process for the neural network for identifying the part's ID number is fairly straightforward. The two folders for training and validation data are provided to the network, and it is allowed to run its full training cycle. When this is complete, the fully trained network is then saved to an ".h5" file, which preserves both the weights and the structure of the network. This network is then fed all of the data from the "testing" folder, and its responses are cross referenced with the known IDs of the part images. The total accuracy of each part folder's inferenced ID is then calculated, and the resulting accuracy rating from 0 to 100 is reported, along with the most commonly guessed ID for each part type (this is utilized in debugging to determine the network's estimate when it is more than 50 percent incorrect). This provides accurate metrics for the reliability of the network, as well as indicating the network's most common estimates in the event of a poor classifier. Another technique implemented is the use of "bad data" in testing. This is done by capturing images of parts that are poorly-lit, off-center, or somehow not perfectly representative of the actual parts view. These images are then provided to the network as testing data. If it is able to reliably categorize these "bad" images, the user can be confident that the network will be able to correctly classify the parts in real-world conditions. When a training run has finished, the percentages are averaged and tabulated in a document to track how the network's accuracy responds to the structure and parameter changes of the previous run. The parameters of the network are then changed and the training is recommenced. When the average accuracy of the full network (on the validation data, testing data, and bad data) reaches a desired threshold (in this case the target was 95% accuracy), the ”.h5” file is saved under a separate name and transferred to the main controller for use in Module 2.

2) Training and Testing the Color Classifier: The training and testing process for the color classifier is very similar to that of the serial classifier. There are only two major differences: the number of classifications for color is significantly higher, and, due to the significantly reduced file sizes of the color testing data, the training takes substantially less time (a few minutes per run rather than half an hour). In testing the output of the color network significantly higher accuracy measures were achieved (near 99% on even ”bad data”), and the training process took dramatically less time due to previous experience in training the serial classifier and the much faster turnaround time on training.

VI. Module 1

Module 1 is the system component designed to serialize parts (see Figure 17). It utilizes a cylindrical hopper to take a large input of unsorted parts and divide them into a serialized stream of individual bricks for Module 2. The system must isolate the provided LEGO bricks and eject them from the module with relatively constant spacing. The module utilizes two part tumblers in series, rotating at different speeds, and delivers parts to Module 2 upon reception of requests from the main system controller. This section covers the mechanisms and software employed to achieve this.

A. Physical Design

1) Casing: Module 1 hardware is encased by two side panels (see Figure 18). The side panels were laser cut out of 6 mm acrylic. The side panels are designed with holes for support structures and wiring mounts as well as slots for the adjustable part tumblers. The support structures use 4 mm holes while the drums use 10 mm holes and a curved 10mm slot. The two panels are held together by 3D printed structural supports that span the width of the housing. The clear acrylic allows the user to track the progress of the parts through the module, as well as identify jammed parts and other issues.

2) Hopper: The hopper, although seemingly simple, is a complex component both in function and in design (see Figure 19). Large quantities of LEGO pieces are loaded through the main opening, and the movement/vibration of the first drum must sequentially draw parts from the

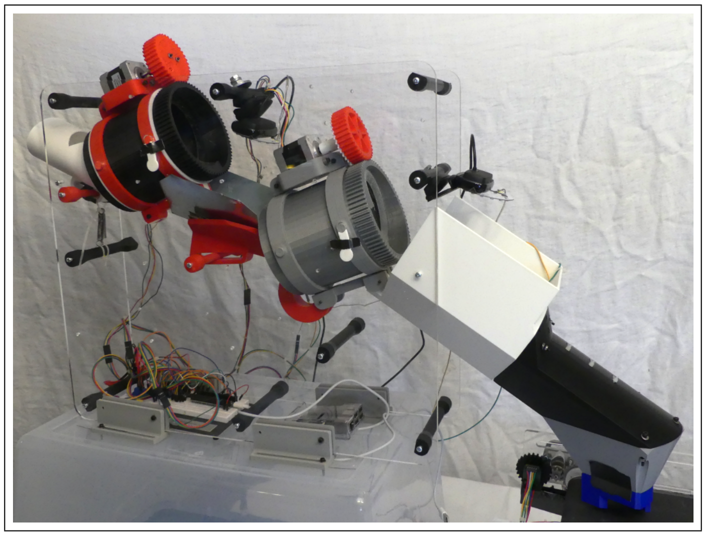


Fig. 17: Module 1

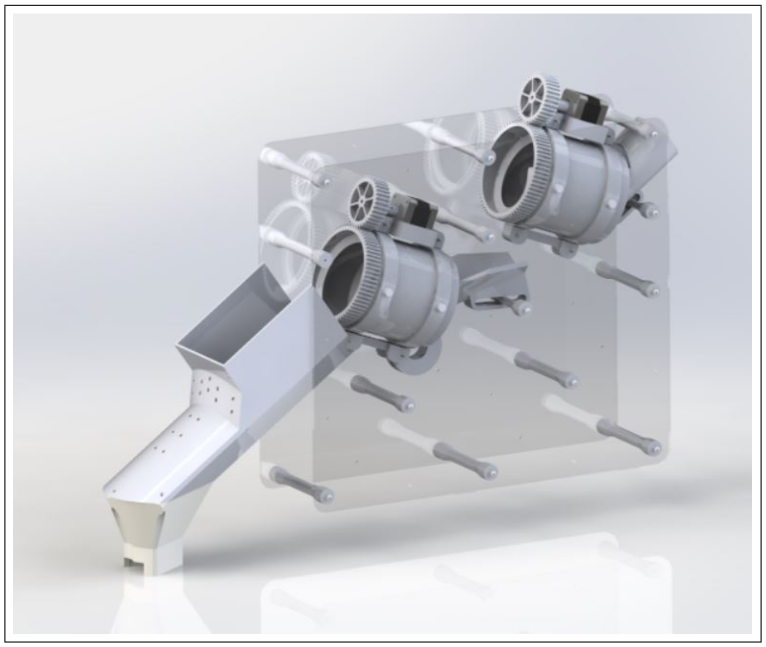


Fig. 18: Module 1 Housing

opening in the hopper's base. The design must also mitigate jamming and ensure that parts can exit cleanly into the drum so as to not climb the sides of the tumbler. In order to avoid this, the exit of the hopper was designed to fill the circular gap of the first drum completely. Sanding and tolerancing were used to guarantee that the hopper does not add extra sliding friction to the inside of the drum. This friction is also reduced by mounting the hopper on a slide, which both allows the hopper to float in the opening, and also translates more of the drum's vibration to the hopper, dislodging stubborn parts.

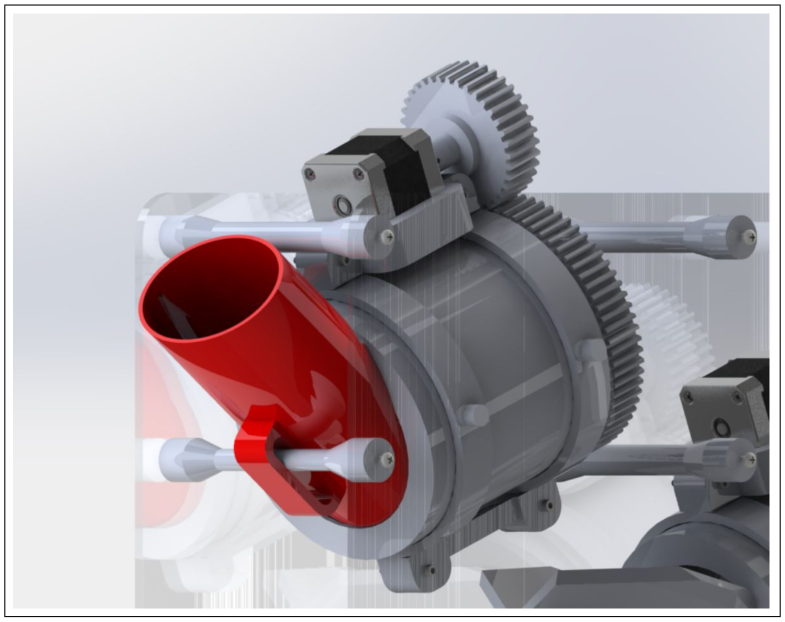


Fig. 19: Hopper Design and Mount

3) Part Tumblers: The main serializing component of Module 1 is the part tumbler, which is structured like a threaded cement mixer (see Figure 20). Module 1 houses two of these tumblers for serializing parts in different stages. The first tumbler in sequence (the one that immediately follows the hopper) is designed to hold many more parts than Tumbler 1. This drum turns very slowly, so as to slow down the movement of parts through it as much as possible, as well as to reduce the draw from the hopper. The second tumbler turns approximately four times faster so that, if multiple parts exit the first tumbler at once, it is more likely that they will be separated in the second drum. This means that in most cases, only one part will exit the second tumbler before both tumblers are stopped. The tumblers' threaded bodies are terminated in custom gear teeth, which interface with the stepper-driven gear above. The motor driving the first tumbler uses a much higher current than the second so that every step is a more jerky/erratic movement. This helps the parts to stay in the base of the tumbler, rather than climbing the sides and falling backwards towards the hopper. Both tumblers are mounted in two, two-part bearings, which interface with grooves in the drum, and are filled with lithium grease for reduced friction. Both tumblers are mounted in a groove in the side panel which enables 30 degrees of tilt. This allows for fine-tuning of the hopper angle to determine the most effective part translation strategy.

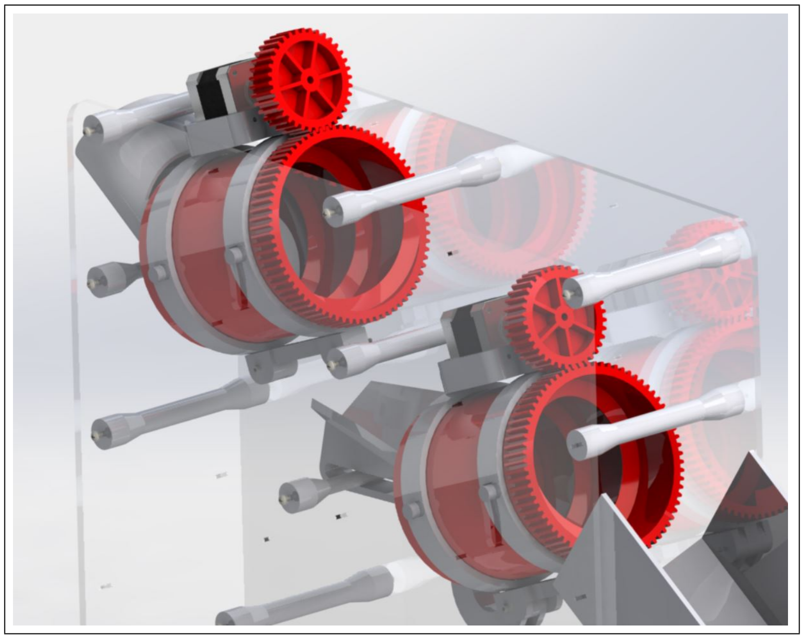


Fig. 20: Serializing Tumblers

4) Slides: The tumblers in Module 1 are connected by slides, which are 3D printed ramps used to translate falling bricks from one tumbler to the next, and then to Module 2 (see Figure 21). The slides use a 95 degree angle, which means reduced contact when a rectangular part falls. This limits how often the parts will become trapped in the slides. Both slides are designed to bridge the casing from side panel to side panel. Parts often fall in unpredictable trajectories, and the wider slide ensures these rogue bricks will be redirected to the center of the following drum. The first slide is mounted on a sliding 4-bar mechanism, which tilts according to the angle of the second tumbler. This ensures the slide will always correctly overlap with the entrance of the second drum. The second slide in the series is much larger, as this translates the part from the second drum all the way to Module 2's conveyor. It also includes mounting holes, which allow the user to mount led strips inside the slide so the brightly lit slide can be more easily monitored in visualization software. This slide is terminated in two parts, the first is a tapered exit, which reduces some of the forward velocity of the falling part. The second is a triangular exit, with a rectangular opening. This is used to stop the the falling part entirely and ensure that it exits onto the slide in one direction.

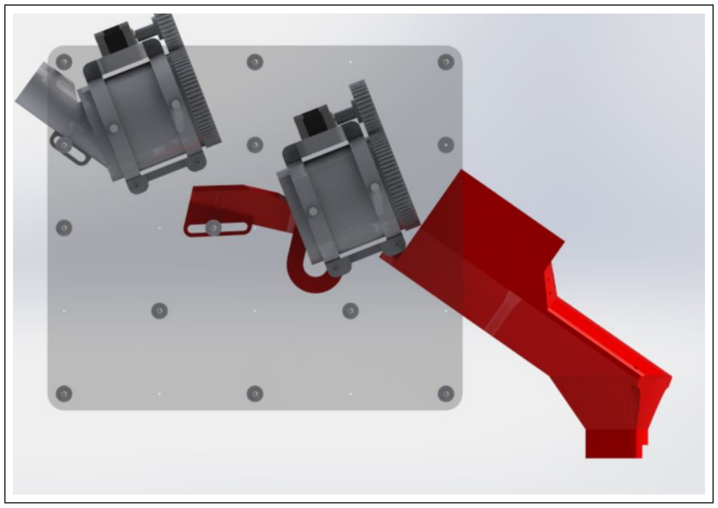


Fig. 21: Slide Designs (1 and 2)

5) Stepper Drive: Module 1 utilizes two NEMA 17 stepper motors to turn the tumblers. These motors are controlled separately through two individual drivers. This necessitates the creation of two, nearly identical motor controller instances in software to fully control the tumblers. This allows for individually addressable tumbler rotation velocities, something that became vital while tuning Module 1 to behave optimally. After extensive testing, it was determined that the optimal speed configuration for the first tumbler was much slower than the second. The first tumbler requires more vibration, which ensures that pieces are dislodged from the hopper and do not climb the sides of the tumbler's internal walls. The second tumbler is designed to eject parts as quickly as possible. This was achieved through reduced vibration and a much higher rotational velocity. The individualized software controllers and tunable current inputs for the two stepper motors allow for both of these tumblers to operate within their ideal functions. This helps to ensure that bricks exit Module 1 properly serialized.

B. Electrical Hardware

1) Teensy 3.6: Module 1 utilizes two dedicated NEMA 17 stepper motors for granular control of the two tumblers. Each stepper motor requires dedicated hardware clock signals and a microprocessor to function with proper feedback control. Although the first module already uses a Raspberry Pi that is capable of controlling image pipelines, these devices do not have the necessary control I/O to drive stepper motors. In order to drive a single stepper motor, 8 I/O pins are required for a NEMA 17 and its respective driver. There are the step, direction, and enable pins that connect directly to digital output lines on the Teensy. There are also three microstepping pins that must be set to the same voltage as the microcontroller logic (Teensy) voltage, 3.3 volts, to enable full microstepping. Although full microstepping is not always utilized (Tumbler 1 utilizes more microstepping than Tumbler 2), the pins were kept available for testing purposes. The stepper driver also needs consistent access to microcontroller logic voltage so that it will properly interpret the control signals going to the stepper driver from the Teensy (step, direction, enable, and microstepping signals). The pulse width of these signals must be extremely accurate, meaning that dedicated hardware clocks are required to maintain the proper signals on the step pin. These requirements demand the use of extra devices to control the flow of parts in Module 1. This device also requires ROS compatibility to interface with the rest of the control system. This limitation restricted the scope of possible co-processors.

After further consulting, Teensy's 3.6 microcontroller became an obvious choice. The Teensy 3.6 has 48 I/O lines available in a small, breadboard-compatible package. It also has multiple dedicated hardware clocks, which are applicable to stepper motor control. Only the larger and more expensive standard Arduino microcontrollers possess this functionality. The Teensy also has a hardware floating point unit (FPU) for complex mathematical operations. Arduinos emulate these FPUs in software, making calculations slower. The 3.6 also boasts clock speeds in excess of 200 MHz whereas standard Arduinos have a maximum of 16 MHz. This means that, when the Teensy is communicating with a host machine over USB, it will maintain significantly more bandwidth for ROS messaging and sensor reading and actuation. The Teensy also has more digital interrupt pins then any Arduino model, as every digital I/O pin can be multiplexed as an interrupt pin. This means that the state machine that runs on it can be programmed to operate in near real-time. There are also more PWM (pulse-width modulation) pins available on the Teensy than similar Arduino boards. Combining these technical superiorities with a lower price and dramatically smaller form factor makes the Teensy 3.6 microcontroller the obvious choice for motor control and actuation in Module 1 (see Figure 22).

This is not to say, however, that there are no drawbacks to utilizing the smaller, more powerful Teensy. The most obvious drawback is the fact that the Teensy is a 3.3-volt device. Most sensors for electronics of this class are 5-volt devices. This limits power delivery to the Teensy to micro-USB cables, as the power supplies used for the motors and lights run at 5-12V. In order to interface with other 5 volt devices (as is the case for Module 3), it is necessary to use logic level shifters to convert data lines between voltages. Another lesser-known issue with using Teensys is

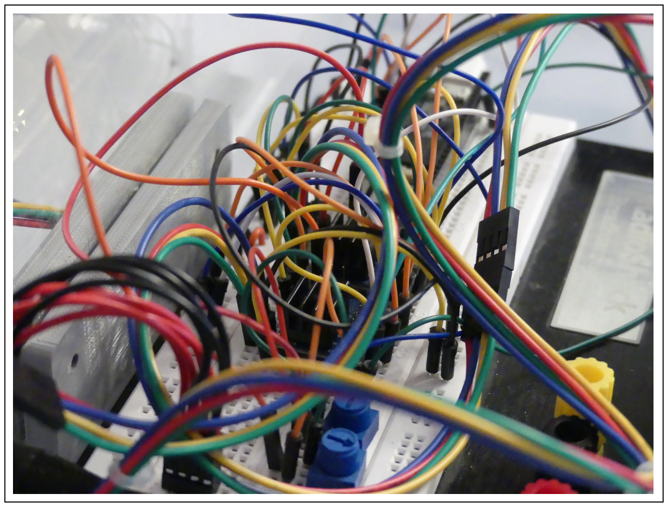


Fig. 22: Teensy Circuitry for Module 1

that the layout of 48 pins makes soldering stackable headers impossible without modifications. The Teensy has 2 rows of 24 pins each, while most stackable headers are a maximum of 12 pins long. In order to solder stackable headers to the Teensy, the headers must be physically modified to double up in adjacent slots. This adds time and sometimes error to the electrical consolidation of the Teensy circuitry.

C. Computer Vision

1) Motion Detection: Module 1 utilizes two USB cameras to control the movement of parts through the module (see Figure 23). Each of these runs an OpenCV (an open source library for computer vision in Python) script designed to detect motion on each of the two slides. The motion detection script is designed to view an input image from one of the cameras and compare it to the first image processed by the detector. If the input image, after being simplified through Gaussian blur, exceeds a differential threshold from the old image, the detector will publish a flag for motion. This means that as a part falls through the slide, the camera situated above will repeatedly test the value of the image against its previous self, and if the image changes, motion is assumed to have taken place. This works for all part colors and shapes. The motion detector constantly publishes to a ROS topic for motion, sending a boolean variable representing a movement or lack thereof. Once motion is detected in either slide, the first tumbler will stop. If motion is detected through the secondary slide, both tumblers stop. This ensures that only one part can exit Module 1 within the desired time frame. Module 1 then publishes a message to

Module 2, informing it that a part is currently available, and Module 1 can start moving once more.

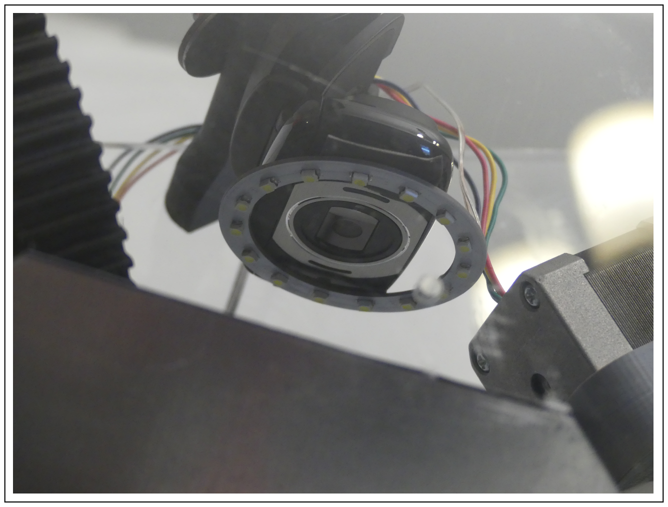


Fig. 23: USB Camera Positioning

2) Lighting: When using aftermarket USB cameras in custom image pipelines, one very common issue is auto-exposure. USB cameras are designed to automatically determine the correct exposure levels for their sensor and adjust it in real time according to changes in lighting. In order to control the lighting within Module 1, LED ring lights are mounted below each USB camera (see Figure 24). This ensures that the USB cameras will always auto-adjust to the appropriate levels regardless of environmental factors like the sun or room lighting. With this strategy implemented, sudden fluctuations in ambient lighting are less likely to cause lapses in part sensing, or accidental triggering of the motion detectors. LED lighting is also used in the base of the second slide to brighten parts as they fall through it. These lights are all controlled on individual 5-volt rails controlled through voltage regulators that allow for variable brightness.

D. Software

Module 1 requires the most straightforward control logic. Module 1 initializes waiting for a request to provide a part. Once Module 1 receives a part request to the ROS topic "/module_1/request_part/", the Module 1 controller will activate both tumblers. While the tumblers are turning, the controller is waiting for either of the USB cameras to detect motion. This detection of motion is how the serializer knows a LEGO piece has gone by. If a LEGO piece passes

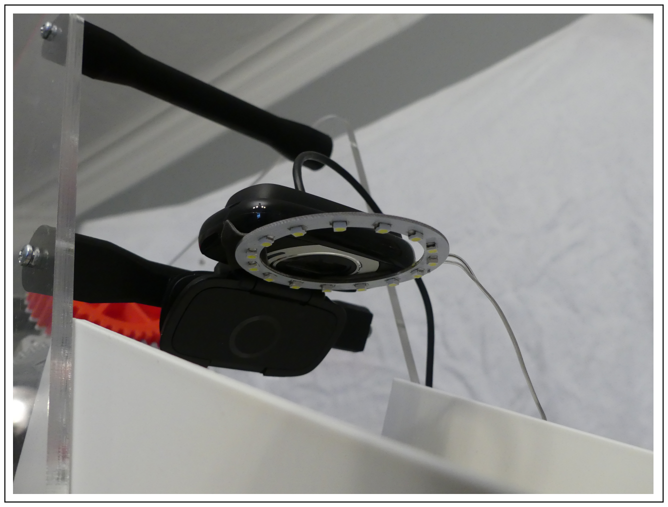


Fig. 24: Module 1 Lighting

by the first camera, then the first tumbler stops. If a LEGO piece goes by the second camera then both tumblers stop. This second scenario indicates that a part has left the serializer. Once a part has exited the second slide, Module 1 publishes to ”/module_1/part_available” to inform Module 2 that there is a part to collect. Next, Module 1 initiates a delay for 1 second so that the Module 1 controller does not detect the same part more than once. In the first rendition of Module 1, image croppers were utilized in the pipeline for each camera to ensure that the sensor would not detect motion outside the sorter housing. These extra ROS nodes added a significant delay in the ROS communication, ensuring no LEGO piece could be double counted as it fell through the slide. Once the mechanics of Module 1 were updated, however, the motion detectors processed images so rapidly, parts would often be registered twice. For this reason, a ROS delay was implemented after Module 1 publishes the availability of a new part.

During the creation of the control software for Module 1, the team encountered many scenarios where syntactical programming was required. One of the major issues encountered involved duplicate ROS node names and topics. If a ROS node is running, and another node has started with the same name, the original node will crash, closing all its communication lines in order to make way for the new process. This is due to the fact that all ROS nodes run on the root namespace (a process-specific directory of named items), unless otherwise specified. As was discovered during development, there are many opportunities to reuse ROS nodes across all 3 modules, due to shared mechanical and digital functions. For example, if two devices (like the two tumblers in Module 1) utilize the same electrical hardware and function in the same manner, it would be ideal to run both of them with the same node. However, as explained before, the first tumbler would initiate properly, and upon booting the second tumbler, would crash. This can also be an issue with ROS topics, but instead of crashing, the nodes will behave unpredictably. This is due to the fact that nodes cannot track the sources of their received messages, as multiple publisher nodes can be publishing to the same topic. As all three modules dealt with this issue, the main controller utilizes a programming paradigm known as 'namespaces' to delineate where ROS nodes can be run and ROS topics can be found. This helps to remap the interactions of multiple identical nodes and topics to their proper publishers/subscribers.

1) Part Request Handling: Module 1 operates entirely through callback functions and timers. As the two major expected run conditions for the sorting system are timed and continuous, the first Module requires handling for both operations. This can be done through ROS topics configured to trigger singular part requests, on a timer, or manually. These messages can be enacted by either a universal timer (typically triggered every 10 seconds), or by Module 2, which can request parts as needed. If Module 2 is ready to accept a part, Module 2 will publish the message to Module 1 asking for a part and Module 1 will then power both drums until the second detector senses a part has entered Module 2. This dual system ensures that a part will always be available when Module 2 is ready, and a constant stream will be provided regardless of temporary jams/delays in other parts of the system.

VII. Module 2

Module 2 is the camera module. The transportation of parts through this module of the sorter requires a simple straight section with 2 cameras overhead. The first camera is a generic USB camera that detects where parts are located on the belt. This camera also counts the number of parts that pass by on the conveyor belt. This information is then fed into the color identifying camera that identifies the characteristics of the part. Once the part has been identified based on some predefined metrics, that part is moved to its final location via Module 3.

A. Physical Design

Although the system is comprised of three distinct modules, they were not designed in the order in which a LEGO brick would pass through them. Instead, the first module to be built was actually Module 2, as it was believed to be the most complicated of the three modules.

The first version of Module 2 was extremely small, and required cutting the 48in sanding belt and reconnecting it to form a 24in loop. In addition to the pulleys for the belt, there was also a raised horizontal platform for mounting the original Zed camera (see Figure 25).

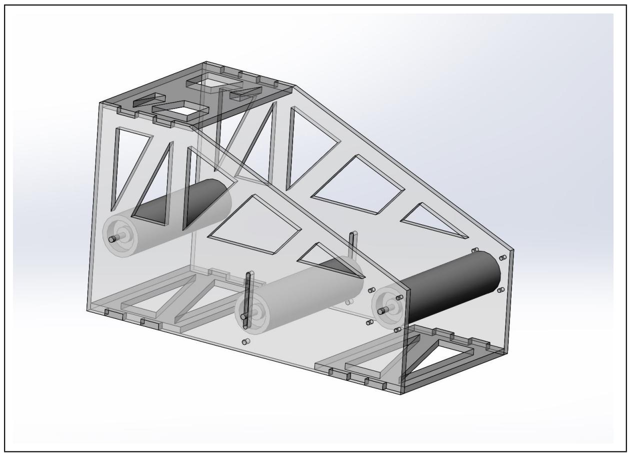


Fig. 25: Original Module 2 Design

The pulleys are 3d printed, and rotate on a steel shaft with nylon bushings. The frame is laser cut from 6mm acrylic, with t-joints for holding the panels together. There are two add-ons to this module, which are a Zed camera mount and a stepper motor gearbox (see Figure 26).

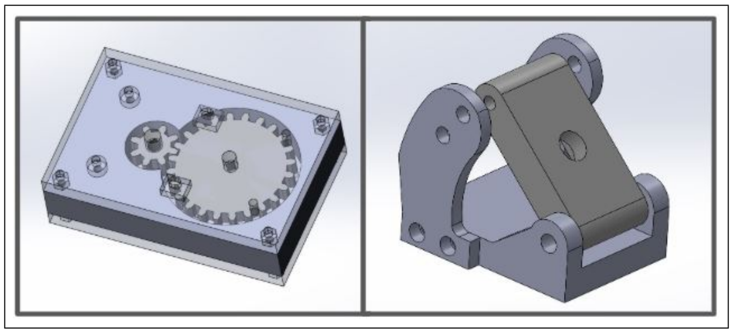


Fig. 26: Module 2 Gearbox and Camera Mount

The gearbox was designed with a 3:1 reduction to help the stepper motor turn the belt. Even with this reduction the stepper motor sometimes struggled to begin rotating the belt. The Zed mount was adjustable and capable of tilting the Zed camera 30, 45, and 60 degrees. Once the assembly was completed, the team was able to see a few problems with the design. The Zed camera is designed for high resolution stereo vision at a distance, which was not important to this application. Instead, the team needed to find a camera with a macro lens, as the Zed camera simply could not focus on the belt at such a close distance. This version of Module 2 was never completed beyond this point, but it acted as a proof of concept.

With the first iteration of Module 2 completed, there were many apparent issues. One major mechanical flaw was that the modified sanding belt could not be properly fastened back together. Both tape and staples were unable to hold the cut belt together for long, and needed to be replaced constantly. Also, the Zed camera mount simply did not allow the camera to function properly due to its proximity to the belt. The second version of Module 2 attempted to remedy some of these problems by using the full 48 length of the sanding belt. This way there would be more distance for the Zed camera to focus on the object on the belt, and there would be no issues with the belt splitting during operation. To do this, the entire module was extended. All the pieces were still laser cut from 6mm acrylic, and utilized U-bolt joints to be fastened together.

There was also a design approach change to the second version of Module 2. Instead of trying to fully design every possible attachment for the module, the side rails were designed with a multitude of holes and slots for mounting future attachments (see Figure 27). The slots would allow for attachments to be moved any distance along the module, requiring less accuracy when designing parts. The ability to finely adjust parts proved to be essential as the design process continued. This version of Module 2 had 5 rollers, and the nylon bushings were replaced with steel ball bearings. The rollers were 3d printed, and rotated much more freely when compared to the first iteration. The bottom 2 rollers were mounted into slots in the acrylic, allowing them to act as tensioners for the belt. The rollers were held under tension by a pair of rubber bands on each axle. However, the belt's tension issues persisted, as parts continued to vibrate and move around unpredictably when in operation.

At this step in the process, the camera mount also underwent changes. Instead of being integrated as part of the main chassis of Module 2, it became a separate structure. The sub-structure can be adjusted along the full length of the module. This allows the user to finely adjust the distance at which the camera's view is oriented on the belt (see Figure 28). Having solved the mounting issue, the team determined that the Zed camera was not a viable option for this application, as the necessary focal distance was far too large, even with the extended belt. Instead, the Zed camera was replaced with a USB webcam. Although not nearly as high resolution, the USB camera is capable of detecting centroids and distinguishing parts from the belt. However, due to its low resolution and distance from the belt, the webcam is not capable

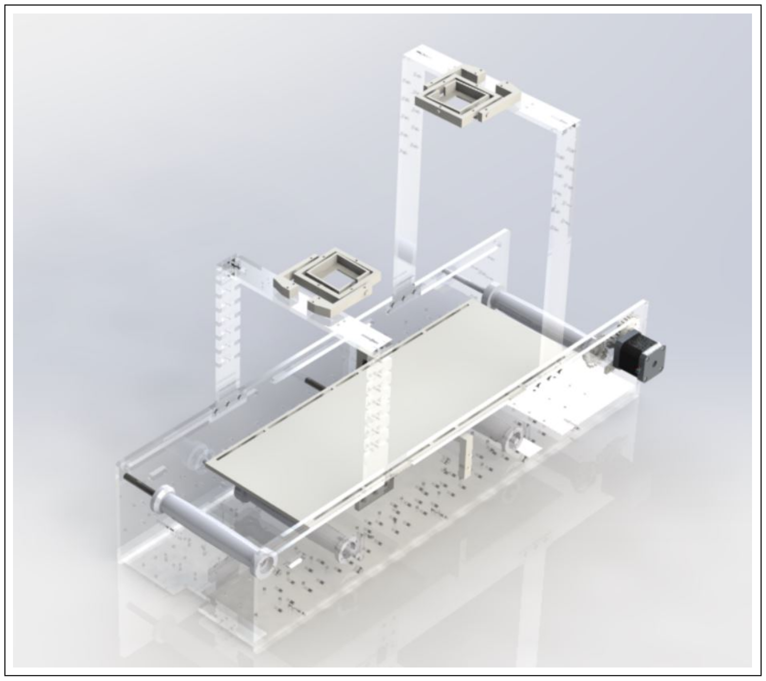


Fig. 27: Second Iteration of Module 2

of generating images for part classification, so a secondary camera is required for capturing part images in higher detail.

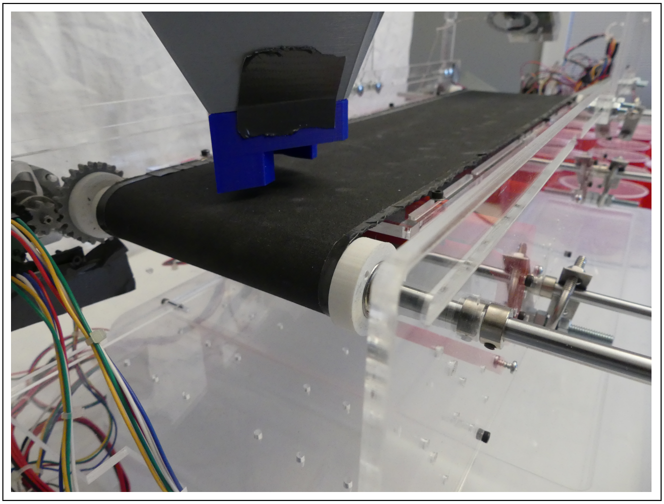


Fig. 28: Module 2 Conveyor Design

With the Zed camera removed, there was a need for two new cameras. One of the cameras was a generic USB webcam, and the other was a Raspberry Pi Camera. The USB Camera is responsible for detecting a part on the belt and determining its centroid. To maximize adjustability, the mount for the USB camera can slide along the entire length of the module (see Figure 29).

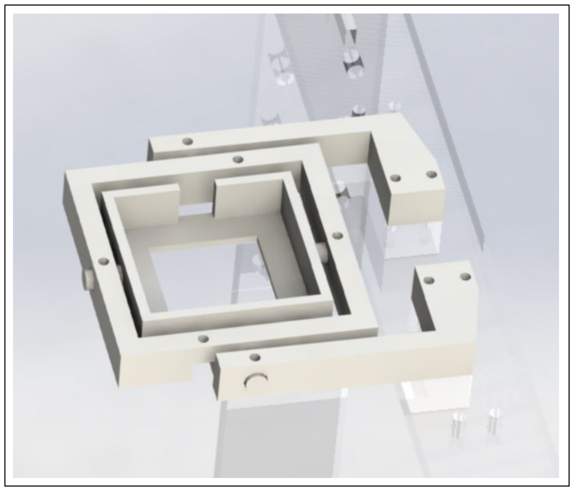


Fig. 29: USB Camera Mount

The second camera is a Raspberry Pi Camera (Picam), which can focus at a much shorter distance than the Zed camera while maintaining high resolution photos. Since the Picam is responsible for classifying the parts by shape and color, it needs even more adjustability than the USB webcam. The mount for the Picam is adjustable along the full length of the module, and also has adjustable height (see Figure 30).

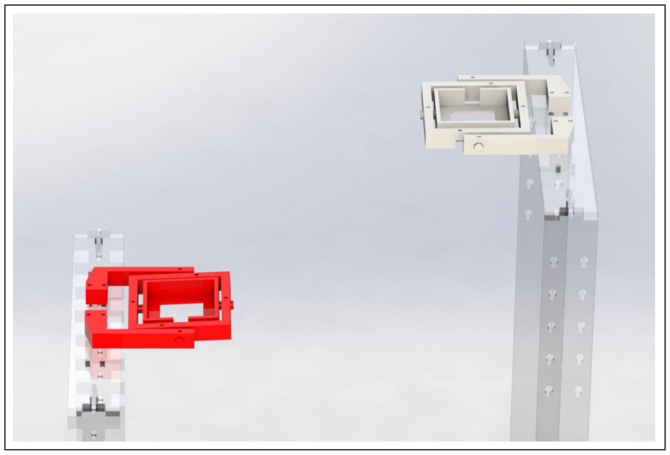


Fig. 30: Pi Camera Mount

Both cameras are mounted in a case that allows for two degrees of freedom. The cases allow the cameras to be tilted along the length of the belt and perpendicular to the belt. These cases are essential to moving the view of the camera to keep the belt centered.

The second iteration of Module 2 discarded the initial gearbox design in favor of a simple gear reduction mounted directly to one of the side plates of the module. The ratio remained the same at 1:3, which combined with the new bearings, allowed the Nema 17 stepper motor to easily turn the belt. There is also an encoder attached to the opposite side of the stepper motor, with the same gear reduction (see Figure 31). The encoder allows distances to easily be measured, and provides feedback to the microcontroller about whether the belt is stopped or moving.

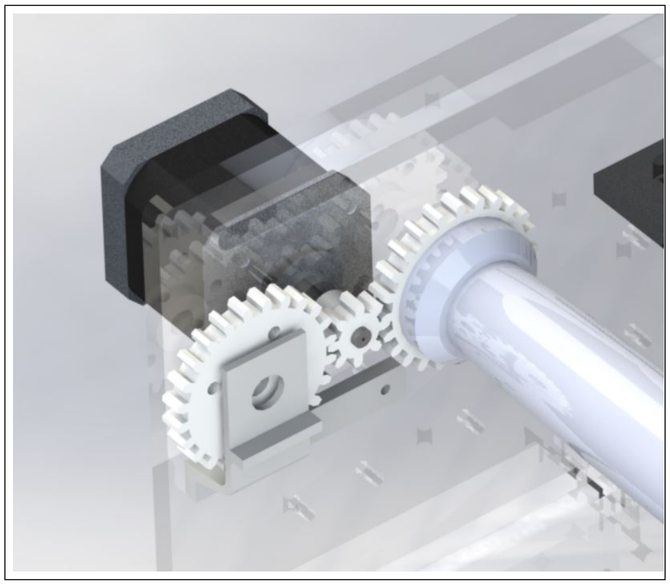


Fig. 31: Gearbox Encoder and Mounting

The final iteration of Module 2 does not have many differences from the second iteration. The only major change was the addition of two more tensioners on the roller opposite of the driven roller. This was a necessary change as the belt was not staying centered on the rollers, and the two new tensioners allowed the belt to be adjusted to remain centered at all times during operation. With these changes, Module 2 was mechanically complete (see Figure 32).

1) Belt Tension: Proper operation of a conveyor belt like the one implemented in this system mandates tightness and consistency in the belt's orientation. This helps to ensure that the belt has no slack, and stays centered in the middle of the pulley, rather than climbing up the sides, hurting the performance of the system and damaging the belt. In order to meet both of these requirements, steel U-bolts were employed as hard tensioners to the static shafts (see Figure 33).

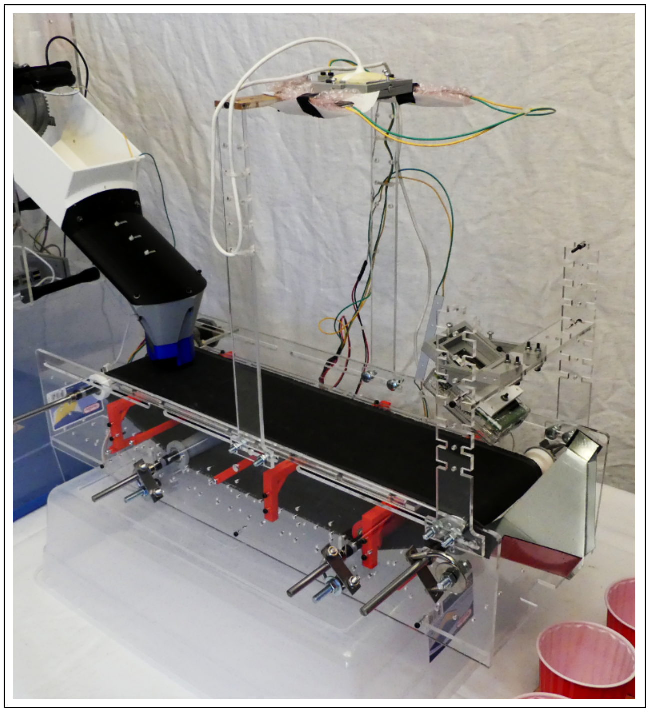


Fig. 32: Module 2 Final Iteration

The U-bolts were the correct dimensions required to wrap around the floating shaft as well as the tensioner peg. Six U-bolts were used to tighten the three floating shafts along Module 2. In order to tighten the belt properly, the belt was run in continuous rotation. The team then observed the belt, noted inconsistencies in its path, and tightened/loosened the bolts accordingly. High tension would reduce the erratic horizontal movement of the belt, but would also increase stress, hurting the performance of the stepper motor and introducing unnecessary friction. Alternatively, loosening the bolts significantly also ensured the belt would stay centered, but would add slack to the center, pushing parts to the sides. This adjustment operation was repeated multiple times before the sorter became stable for extended periods of time.

2) Anti-Trampolining: A major downfall of stepper motors is their tendency to ”snap” between steps when provided high current. This has negative effects on conveyor belts, as this ”snap” motion is translated along the length of the belt. When it occurs at hundreds of steps per second, it causes a vibration throughout the belt's entirety. Module 2's belt system is no exception. Parts travelling along the belt in its original configuration would vibrate and shift sporadically, negating the accuracy of the centroid finder, and hurting the consistency of part placement under the classification camera. In order to mitigate this, an anti-trampolining platform was designed.

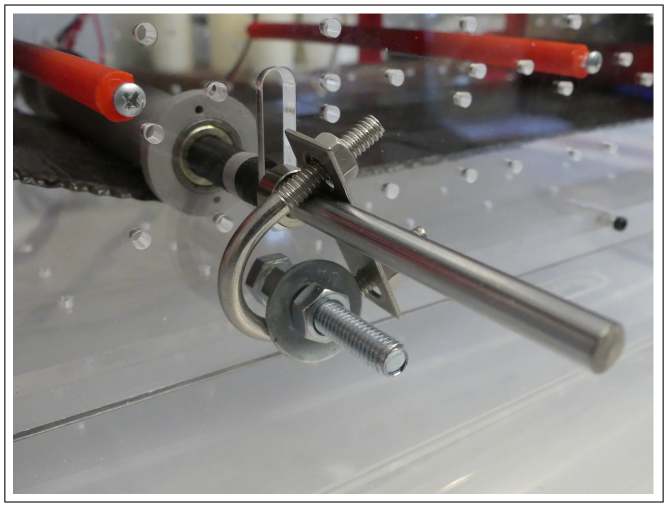


Fig. 33: Belt Tensioner

This is essentially a slab of laser cut acrylic that sits exactly under the majority of the conveyor belt, at the precise level of the pulley's highest point (see Figure 34). The platform is adjustable in height and in horizontal position, and behaves as a type of damper, softening the bouncing and vibration of the sanding belt while still allowing for high velocities.

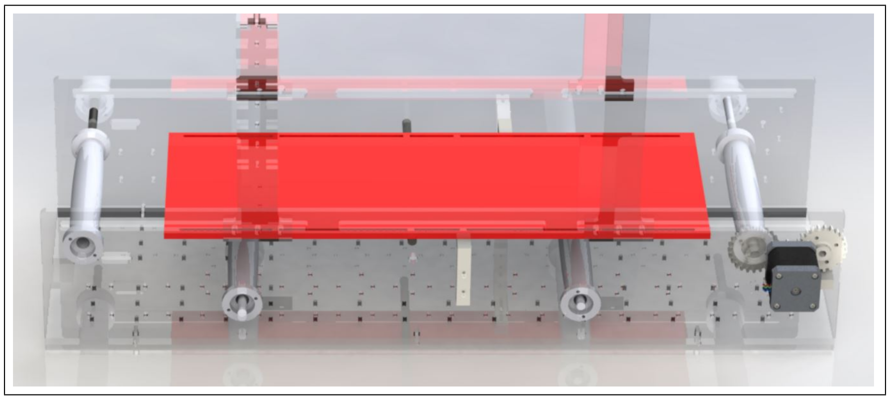


Fig. 34: Anti-Trampoline Platform

3) Stepper Drive: This sorter is driven entirely by stepper motors. These motors have power requirements that must be met to ensure proper operation. The project started with a focus on Module 2 so the only electronics involved were a Teensy microcontroller driving a single NEMA 17 stepper motor. In the first test configuration, it was assumed that, for testing purposes, a single 12 volt and 3 amp supply would be sufficient to power a 12 volt 0.1 amp stepper motor and a Teensy, which operates at 5 volts using negligible amounts of current. It was also assumed that, given the 36 watt power supply, there would not be any issues with current draw. Given these assumptions, the initial stepper motor configuration was configured to draw 12 volts and 2 amps. The initial power supply implemented was a Cross the Road Electronics (CTRE) Voltage Regulator Module (VRM) (see Figure 35).

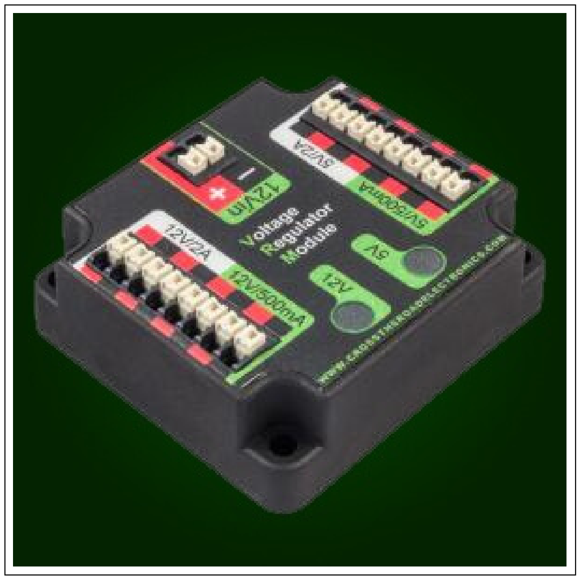


Fig. 35: Module 2 Original Power Supply

After booting up Module 2 for the first iteration of software development, the stepper motor circuit began behaving sporadically. The voltage supply would start clicking loudly when the stepper motor attempted to accelerate, but would then go silent and stop when the motor reached its target angular velocity. Prior to this velocity, however, the motor would vibrate so much that the acceleration could not be controlled through software. When the motor reached this stopping velocity, and then tried to advance, the motor would buzz and click, sounds indicative of a stalling stepper motor. The voltage supply would also click uncontrollably. The software came directly from the library examples, so that was not the problem. The team also tuned the potentiometer for the current limit on the stepper driver, but adding more current or taking away more current would not change the behavior of the motor. This lead the team to the conclusion that the stepper was over-drawing current (amps) from the power supply. It should also be mentioned that there was no load on the motor, ruling out mechanical stalling as a culprit. This experiment led to the conclusion that the power supply must provide each stepper motor driver a current of greater than 2 amps for optimal performance. Since there are several uses for stepper motors throughout the sorter, the team decided it would be advantageous to invest in a substantially more powerful AC/DC power supply.

After a proper power supply was acquired, testing on Module 2 could continue. The left power supply is a dual 12v/5v supply for powering the Teensy (see Figure 36). The unit on the right is the 12 volt 40 amp power supply. This is the main supply that will power the entire sorter. The left power supply is simply for testing purposes and would later be repurposed to power lights and a Raspberry Pi. The middle of the image contains the breadboard-based circuit for the Teensy and stepper motor driver. The USB cable plugs into an Ubuntu laptop for testing purposes. The stepper motor driving code worked admirably once proper power supplies were connected. With the stepper motors working properly, experimentation helped determine the maximum velocity of 815 steps per second and an ideal no-load acceleration of 450 steps/sec^2.

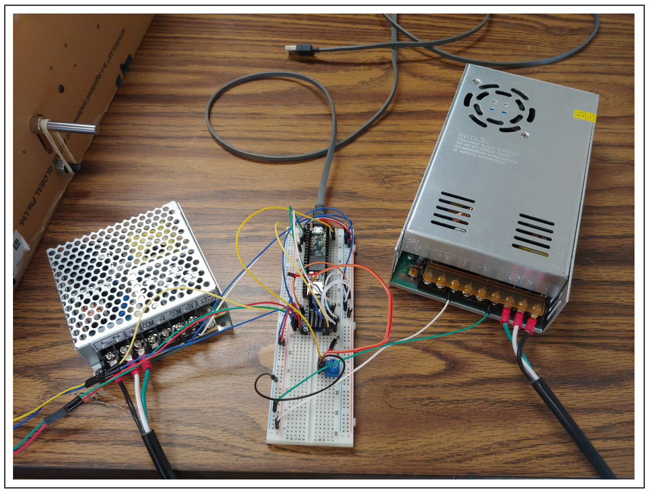


Fig. 36: Sorter Power Supplies

When controlling the stepper motor with ROS, there are some quirks to the software architecture. The communication of the Arduino IDE with the Teensy uses the same serial port as the ROS node, meaning that uploading code to the Teensy will break the communication. Also, if the motor is set to continuous rotation, and the ROS serial node is terminated, and the motor continues to run indefinitely. For future iterations of the software, the Teensy will need to activate a callback function when the node loses connection with roscore. This is a safety issue and therefore demands more thorough research in the future. Although a software E-Stop was implemented later, it would be ideal to devise a more explicit hardware E-Stop on Module 2 to ensure that the mechanical systems can be safely stopped in the case of a major system fault.

4) Camera Mounting: The cameras on Module 2 have very specific mounting requirements. The USB camera for motion detection needs to sit in a position where is sees as much of the belt as possible. If necessary, the image can be cropped so that only a specific part of the belt can be viewed. The mounting stands for both cameras include notches to allow user-configuration of the heights. They are also slotted within the side panels of the module, which allows them to be adjusted for any position horizontally along the belt. Both camera mounts also possess three adjustable degrees of freedom which enables fully configurable camera orientation (see Figure 37).

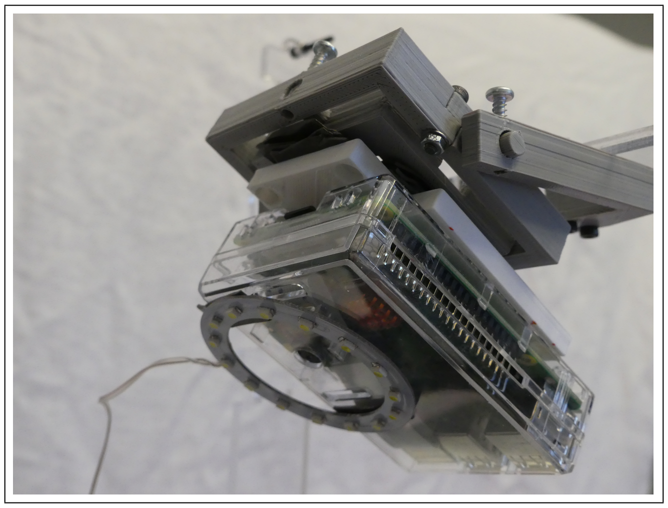


Fig. 37: Module 2 Camera Mount

B. Electrical Hardware

In terms of electrical hardware Module 2 uses a similar configuration to Module 1. It employs a single Teensy 3.6, a stepper driving circuit (as there is only one motor to control in Module 2) including a NEMA 17 and Bourns encoder, and a Raspberry Pi 3b+. The Pi drives communication to the Teensy for motor movement and facilitates the translation of computer vision messages between Module 2 and the main controller. Module 2 also utilizes a simple 5-volt power rail to drive the three LED lights (two light bars, one ring) which illuminate the camera views on the belt.

C. Computer Vision

Most of the computer vision tasks in Module 2 are programmed in the Python programming language due to its rapid work flow, interpreted nature, and excellent data visualization tools. Another reason Python was chosen for these challenges was its excellent base of open source libraries and tools to help simplify image processing. Some of the libraries utilized for this section of the sorter are listed below:

- OpenCV: OpenCV, or the python library CV2, is the primary tool for the secondary camera (used to identify LEGO centroids and colors). It is an open source computer vision library with extensive tools for data visualization, processing, and modification.
- Pandas: Pandas is a python library used to read and write ".csv" (comma-separated value) files. This is used in scripts to record user input and camera data between runs.
- Numpy: A powerful python library for mathematical calculations. In this case it it utilized for large-scale n-dimensional array operations (required for processing high resolution images multiple times per second). It is also used to convert image data into flattened matrices for improved processing speed in the neural network nodes.

1) Centroid Camera: The full system for computer vision on the secondary camera involves three separate python scripts whose purposes are as follows:

- Cropper.py: Opens a simple camera view of the conveyor belt and prompts the user to click around the area that will be cropped out. When the user has completed the outline, they can hit enter to save the points into a csv file called croppoints.csv.
- ColorTuner.py: Opens a simple camera view of the conveyor belt and prompts the user to click around the area they wish to filter out. For every click the prompt will print out the HSV color value of that specific pixel. When the user is satisfied that they have a good average of the color they wish to remove they can hit enter' to save the average of all these hue, saturation, and value components in a csv called output.csv.
- CV2 Test.py: This is the main code for identifying the LEGO brick's centroids and colors. It starts by retrieving the values from both CSV files created earlier (croppoints.csv and output.csv). These are then used to crop out the area outside the conveyor and then filter out the conveyor entirely. Anything on the conveyor is then filtered further to remove holes and inconsistencies. Then the image is binarized and the positions or blobs in the resulting masked are recorded as contours. These contours are then labeled with a green outline and

an averaged centroid. The position of these different contours and their average values are then made available for publication to their respective ROS topics.

2) Tuning and Configurability: The filtering process for the secondary camera's main image is complex and involves multiple stages of editing. To commence the process, the croppoints.csv' file is read in from disk. These points are appended into an array. This array is then run through a fillPoly function, which fills in a polygon between n-points. This creates a cropping mask for later use. The inside of the mask is assigned to 255' (white) and the outside is assigned a value of 0' (black). Next, the averaged filtering color from the ColorTuner.py' program is read in from disk and run through a bounding function. The purpose of this is to assign a proper differential (high and low bound) for the color filtering. At the beginning of the program there is a colorDiff' variable which is used to assign the range in which to filter. The default value is 50, but this is subject to change based on the testing environment. The bounding function generates two new HSV color values where saturation (S) and value (V) range from colorDiff' below the belt's averaged value to colorDiff' above it. The hue (H) value is set from the minimum to the maximum value (0 to 255). It was determined that this does not affect the outcome significantly when lighting is consistent. This marks the beginning of the main while() loop, where the camera will repeatedly capture frames, filter them, and publish them to the user until the esc' key is pressed. The raw capture from the camera is first run through a bitwise AND filter with the cropping array from above. This crops out the unnecessary part of the image. Then, this image is converted from RGB colorspace to the HSV color space used for filtering. A mask is then created, using the "inRange" filter and the bounds created earlier for the color filtering. This mask filters out all of the desired color (in this case the color of the conveyor belt). This mask is then inverted using a bitwise NOT, leaving only the desired components (the LEGO bricks). Next, the cropped image from before is run through a bitwise AND with the color mask. This leaves only the colored objects on the conveyor belt. Now the filtering that improves the image's usability takes place. First, the image is blurred with a median blur. Then the edges are dilated by a ten pixel radius and blurred again. This eliminates salt and pepper effects and holes in the image and smooths out the edges. Finally, the image is run through a contour identifier, which saves the isolated shapes in an array. The position of every pixel in these individual contours are then averaged to determine the centroid, and their edges are expanded to create a circle around them. These circles and centroids are then applied to the image along with text that labels each centroid (see Figure 38). The process repeats approximately 30 times per second until the esc' is pressed, the camera is released, and the program ends.

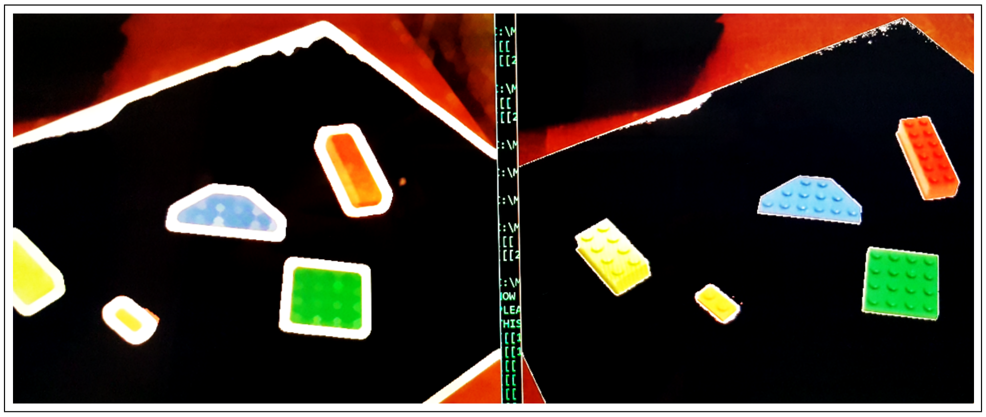


Fig. 38: Centroid View

3) Pi Camera: A Pi Camera V2 captures images for the CNN. This camera has been especially modified with a macro lens so that the images were focused on the LEGO pieces passing by on the belt.

4) Lighting: Module 2 shares similar lighting issues to Module 1. The USB webcam's motion detector requires significantly brighter-than-ambient light conditions to function at normal sensitivity. If there is not enough light in the image, then the parts will not be detected as they pass by on the belt. The classifying Pi camera is also very light-sensitive, but this is more closely related to the quality of the classifying image. The network requires similar lighting conditions to the training data for proper operation. This means that the lighting conditions generated on the Scanner, must be replicated on Module 2's belt. This can be done by implementing the overhead 5V LED lights, as well as a dimly-lit ring light around the Pi camera's lens. In the future, this process could be made more reliable through the use of identical artificial lighting configurations on the Scanner and Module 2, as well as a shrouded data capture design, which would completely eliminate ambient interference.

5) Image Pre-Processing: Before the raw image data from Module 2's Picam can be utilized in the neural network, it must go through data pre-processing. This is essential, as the network is highly dependant on the data closely mimicking that of the training set. It is often the case that a part will move along the belt and will not be perfectly centered. Without pre-processing this can damage the network's accuracy, as the convolutional network is not agnostic to position. The network negates inconsistencies in scaling and orientation, but an image that is mis-centered is much more difficult to classify. Because of this, each image taken from Module 2 is passed through the same processing script as the images on the Scanner. In particular, they pass through a script ”piResize.py”, which is very similar to ”resizeImages.py”, described in the ”Data Organization and Storage” section above. This uses the OpenCV library to crop, center, pad, and then resize the image, creating two separate images. One uses a 128x128 resolution for the serial classifier, and the other utilizes a 16x16 resolution for the color classifier. Both of these images are perfectly centered, so as to guarantee the most accurate estimate in both color and part identity. This script is also ROS enabled, taking the message type for images and then publishing the output to the NetNode, which is used to classify the part currently travelling through Module 2

D. Software

The software development for Module 2 started with the identification of necessary software libraries. The most significant software need is in driving the stepper motor circuit. While the steppers run, the Teensy must maintain communication through ROS, so the stepper library cannot consume all of the controller's hardware clocks. The stepper motor control also requires the use of acceleration, both to reduce jarring motion for the parts on the belt, and because the belt's tension and static friction would strain the motor on a cold start. After searching on the internet, a library called TeensyStep [10] by Github user ”luni64” was discovered, a stepper motor library that fits the requirements for Module 2.

The software to control Module 2 employs the largest number of ROS nodes. First, Module 2 listens to the ROS topic ”/module_1/part_available”, waiting for Module 1 to publish a boolean message of type ”True”, signalling that it is capable of processing a part. Once Module 2 receives the message that there is a part available, it will send a command to the Teensy serial_node.py ROS node from the ROS package rosserial_arduino to start the movement of the belt. The part then passes under the centroid camera, signalling to the Pi that a part is present. The Teensy's motor controls are then assigned a set distance, to advance the part to the classifying camera. The camera then captures and image, sends the image along the image transfer topic to the net node on the main controller, and repeats the described process once more.

In Module 1, the rosserial node only runs ROS topic subscribers. Module 2, however, requires both subscribers and publishers to ROS topics, meaning the troubleshooting for this module is fundamentally different. On an isolated system like a laptop/desktop computer, this does not require special attention. However, while working with Arduinos and Teensy boards, this becomes a major concern. The rosserial node is running on a serial communication protocol, unlike the parallel processes that run on the Jetson, therefore the serial communications must be more robust than on integrated communication lines. If there is data loss on a regular ROS node, the ROS node can simply keep publishing messages so that the subscribing node can eventually recover its signal. With the rosserial node, if there is data loss, the node risks losing track of which data packet is the header packet, meaning the Teensy can no longer identify the beginning of a ROS message. This can lead to catastrophic behavior of the entire ROS system, as initial tests of the ROS node showed that the Teensy would simply drop the connection with the rest of the network altogether, and then attempt a reconnection.

The first major cause for this communication lapse is that ROS messages cannot be published faster than twice in a second (2hz). In order to get around this, a virtual timer library that takes advantage of the hardware clocks was used to synchronize ROS publish method calls with the optimal publish rate for the serial bus. This meant it would not asynchronously break out of the main program loop. Once ROS publishing was limited, a substantial increase in stability of the rosserial node was noticed.

The second issue was that the rosserial node running on the Raspberry Pi would report a version mismatch between the Arduino ROS library running on the Teensy and the rosserial node. After more exploring in the rosserial library, it was determined that there are different versions of the same ROS node from different ROS packages under the rosserial metapackage. The rosserial node running on Module 1 comes from the rosserial_python package while the node that running on Module 2 is from the rosserial_Arduino package. It is not entirely clear why the Arduino version is more stable than the python version, but the assumption is that it is related to the low level implementation being written in C++ rather than Python. C++ compiles to hardware instructions, while Python runs on an interpreter, suggesting that the interpreter cannot run fast enough for the serial bus to send messages to/from an Arduino at the stock 57,000 baud rate. Once these 2 fixes were implemented, more reliable communication with the Teensy was feasible. It should be noted, however, that if a more redundant form of communication was implemented between the Teensy code and the Module 2 controller, the communication errors could be preventable.

After fixing the two prior issues, more limitations of the serial bus were discovered. Often, after delivering two parts through Module 2, and then requesting a third, no indication was received that the Teensy received the commands to start the belt. After experimenting with the Module 2 controller, it became clear that simply publishing a single request to start the belt was not sufficient. The Teensy would begin dropping messages without prior notice or reported errors. This is likely due to the fact that the serial bus is not fast enough to process the volume of messages being published. The Module 2 controller main loop runs at 15 hertz and the baud rate is locked at 57,000 baud, leading to the belief that the system had reached the limits of the rosserial node. In order to rectify this, instead of publishing a single message, the publish message method call was moved into a for() loop that runs five times sequentially. This eliminated the issue of the belt stopping every third part. It should also be noted that the message was of type empty, so there was technically no information being sent. This was done deliberately to preserve the node's bandwidth in order to make space for the float message that tells the belt how far it should rotate before stopping.

1) Part Request: Another key software aspect of Module 2 has to do with its reception of parts from Module 1. As Module 1 has no way to tell how far parts are along the belt on Module 2, it is the responsibility of the Module 2 controller to inform the first module when it is in need of a new part to classify. There are two ways that this can occur. The first involves the part request system. This is a ROS topic published by the Module 2 controller and subscribed to by the Module 1 controller that contains a boolean value. If the value is true, Module 2 is ready to receive a new part for classification, and Module 1 is free to provide one. If it is false, Module 1 should wait until it is true. This generally means that Module 2 already has multiple parts on the belt and is in the process of classifying them. To provide a part at this point would likely lead to a misclassification. The other way Module 2 can receive a part from Module 1 is through a ROS message on a timer. This request message is only subscribed to by Module 1, and it publishes a part request every 10 seconds. This only occurs if Module 2 has not already explicitly requested a part. This means that if at any time Module 2 takes longer than 10 seconds to request a part, one will automatically be provided. If, however, less than ten seconds pass and a new part is requested by Module 1, the timer is reset so as not to flood the second Module with unclassified LEGO bricks. This has proved to be the most time-efficient and reliable part request method.

VIII. Module 3

Module 3 is the system component that underwent the most significant design changes. As the main goal of the system is fairly simple (delivering objects to specific containers) there was lots of freedom in the design stage, and many possible valid solutions. The original plan was a planar parallel manipulator, designed to very quickly deliver the part in 2D space. There was also a plan and physical design for a release mechanism designed like a camera sensor's iris, which would release parts into the containers upon arrival. These designs were adjusted to fit the time frame and scope of the project, and the module was recreated as a crane-like system with more simplistic polar kinematics, and a bucket-style dropper for catching and releasing parts (see Figure 39). These designs are just as rapid as the previous design, they have minimal moving parts and therefore minimal points of failure, and they utilize more simplistic control logic, which helps to make the system more expandable/user-friendly.

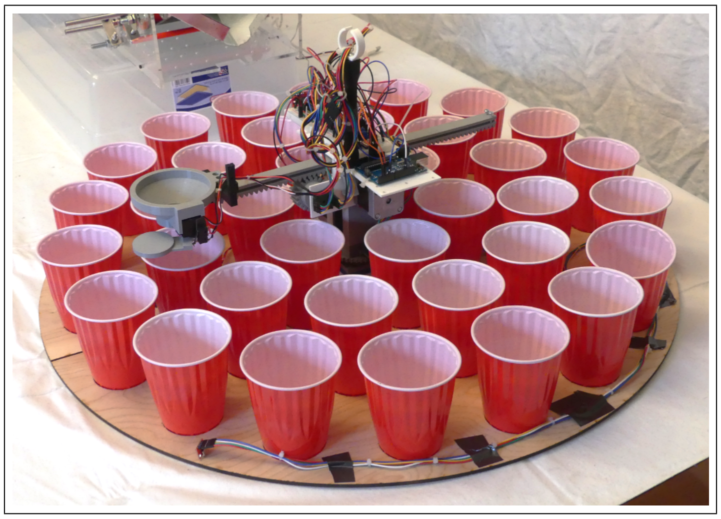


Fig. 39: Module 3

A. Physical Design

Module 3's crane system consists of a solid base, a rotating tower, and an extending arm (see Figure 40). Due to the movement mechanics of the system, Module 3 can be smoothly controlled using polar coordinates. Since the entire module rotates around a base, electrical wiring is handled internally. A slip ring is mounted inside the tower to prevent the wires from becoming wrapped around the module during use. Module 3's final design consists of four major components: the base, the tower, the rack, and the end effector.

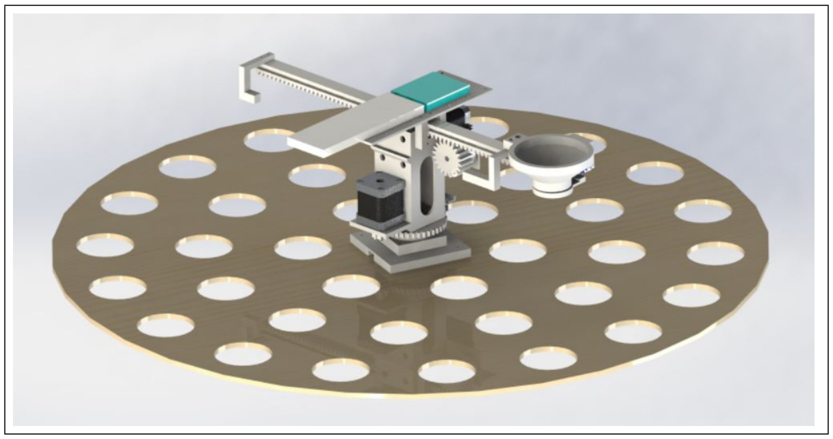


Fig. 40: Module 3 CAD

1) Base: The first design challenge for Module 3 was the base of the system. This base is responsible for organizing the cups which hold the sorted LEGO bricks, and supporting the crane (see Figure 41). By rigidly mounting the crane to the base, the relative location of each of the cups can be kept constant in relation to the crane. This allows each individual cup to have a unique and unchanging polar coordinate associated with it. Without the base, setting up the cups in a consistent fashion would be almost impossible. In addition to positioning the cups, the base also provides a platform for the crane to be mounted on. When the crane is fully extended, there is a significant amount of cantilevered weight, which could cause the entire crane to tip. However, since it is attached to such a large base, the crane will never tip or shift.

2) Tower and Mounting: The crane consists of two major pieces. The rotating tower and the extending arm. The tower is responsible for rotating the assembly along the vertical axis, as well as mounting all the electronics (see Figure 42). Since the tower has to be able to rotate while supporting the weight of the arm, there is a large bearing inside of it. The bearing has an inner diameter large enough to fit a slip ring, and allows for a clean pass through of all the wires in the module. There are many components on the tower, as all the electronics for the entire module are mounted on it. By mounting the Arduino and Teensy on the tower itself, the team was able to greatly reduce the number of wires running out of the tower, as only so many could be attached to the slip ring. The tower is rotated by a Nema 17 stepper and has feedback from an encoder. This allows for very accurate turning, and the tower will never lose its ”home”

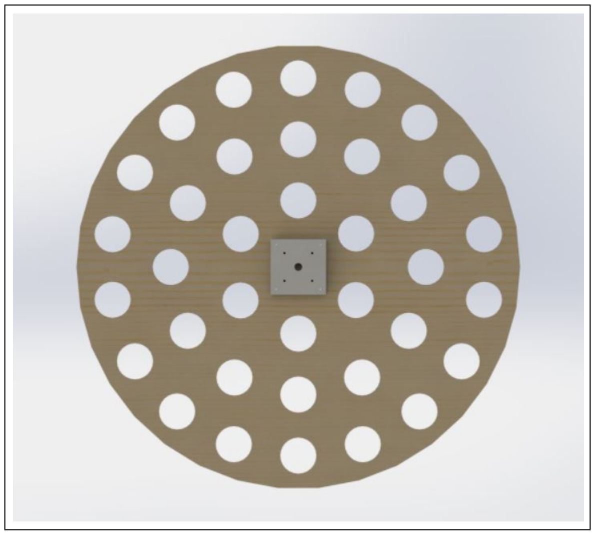


Fig. 41: Module 3 Base Structure

position. At the top of the tower, there is a flat section dedicated to mounting a breadboard and an Arduino. Directly beneath this platform, there is another Nema 17 stepper motor that drives the pinion in a rack and pinion system. The arm on the tower is driven by this rack and pinion, and allows the end effector to be extended or retracted radially from the tower.

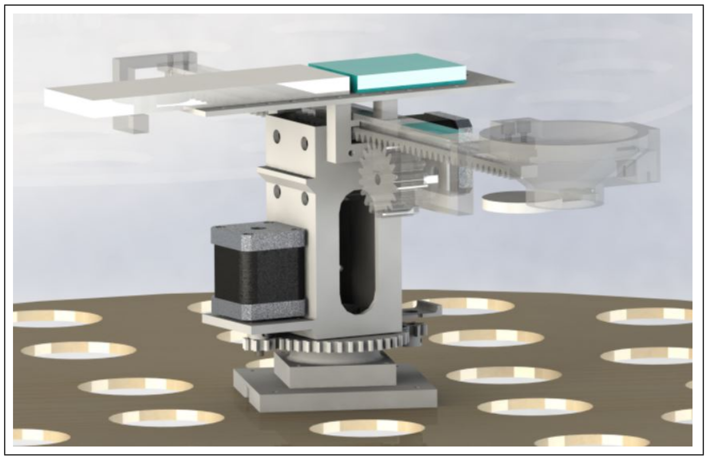


Fig. 42: Module 3 Tower

3) Rack and Pinion: The arm section of the crane is a simple rack gear. On one end of the rack there is the end effector, and on the other end is a mechanical stop (see Figure 43). The rack is driven by a Nema 17 stepper motor, and has an encoder for accurately measuring how far the rack has moved. The rack profile was taken from a model on McMaster-Carr [13], and scaled to fit the tower. Similarly, the pinion was taken from the same source and given the same scaling. There is a limit switch for the rack when it is in its fully retracted position, allowing the team to set a ”home” position for the rack during initialization. The mechanism at the end of the rack is the end effector, which is responsible for holding and dispensing incoming LEGO bricks.

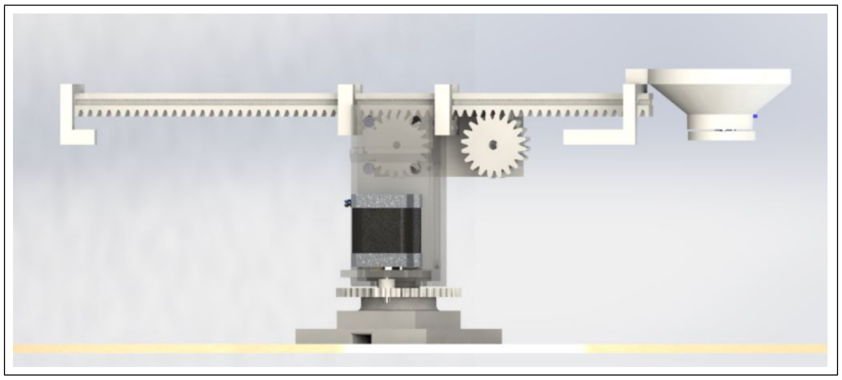


Fig. 43: Module 3 Rack and Pinion

4) End Effector: The end effector for Module 3 acts as a bottomless bucket with a controllable opening (see Figure 44). The end effector includes a mounting position for a 9 gram micro servo. The micro servo swings the cap open and closed allowing parts to exit the effector on command. The servo also includes a limit switch which, when activated, stops the servo’s progress. This is used as a safety measure over extended periods. There is also a hardware end stop connected to the end of the rack gear which activates a limit switch on the tower when the end effector has reached its home position.

B. Electrical Hardware

Module 3 is the most demanding module in terms of electrical requirements. The entire module is controlled by a Teensy, rather than a Raspberry Pi like the others, meaning that all of the sensors and actuators need to be wired accordingly. There are two stepper motors and two Bourns encoders controlling the actuation and localization of Module 3. This means that there are also two stepper drivers in the circuit. There is an endstop to reset the zero configuration of the rack. The endstop is configured as an interrupt so that the zero-ing operation occurs without involving ROS. On the end of the rack there is a servo and another endstop. These devices add more complexity to the end effector as the servo motor requires 5.0 volts while the Teensy is a 3.3

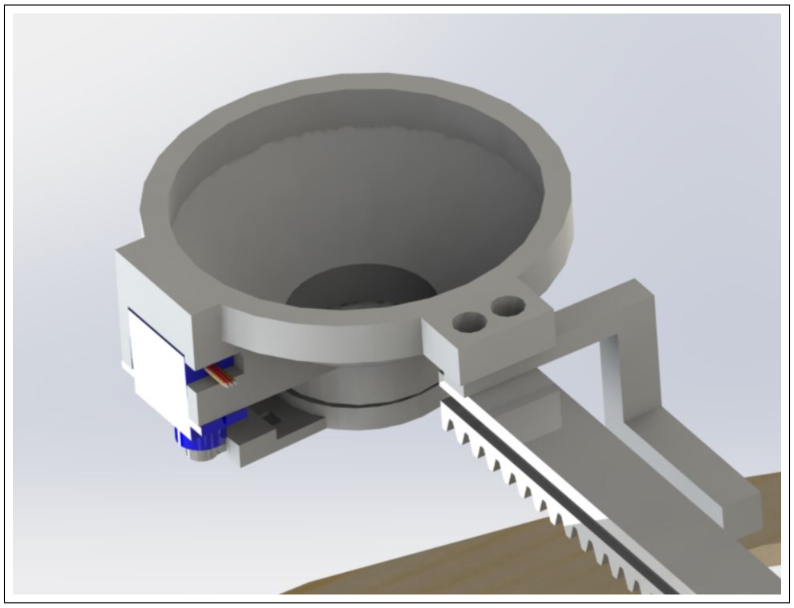


Fig. 44: Module 3 End Effector

volt device. This means that a logic level shifter was required to be able to communicate with the servo. Servos also require constant PWM signals to set a desired position. It is not possible to send burst signals like stepper motors or regular DC motors. This means that a hardware timer and clock divider were required to drive the servo correctly. In trying to integrate the servo motor with the stepper motor library, it was discovered that the two libraries were incompatible. This meant that a separate microcontroller was required to drive the end effector's bucket. This led to the decision to add a regular 5.0 volt Arduino Uno to Module 3's circuitry. Once added, the only control signals requiring level shifting are the interrupt signal to control the Arduino from the Teensy, and the Teensy interrupt pin that reports the state of the bucket. The endstop on the bucket tells the Arduino if the bucket is fully open or not, making the opening/closing of the bucket a closed-loop operation.

1) Integrated Wiring: Unlike the other two components of the system, all of the wiring and electronics for Module 3 are integrated into the actual module (see Figure 45). The reasoning for this design choice is two-fold. First, the positioning of the module within the rings of cups, as well as the low profile of the platform it rests on, makes external electronics and wiring impractical. In order to house the wires and processors utilized, they would either need to be stored under or next to the module's base. This would either require elevating the base, or adding significantly more wiring to the design in order to transfer power and data to the module. The next reason is the elimination of extra wires. This is a challenging task due to the fact that two full motors and a servo are controlled from Module 3's driver, and both a Teensy and Arduino are utilized to activate them. This means that in an ideal case, the wiring would be kept as compact as possible. This can only be done by housing all the electronics on the rotating tower, which begs the question: how will power and data be delivered to the actual module without compromising the wiring when the platform rotates? This dilemma is rectified by the slip ring.

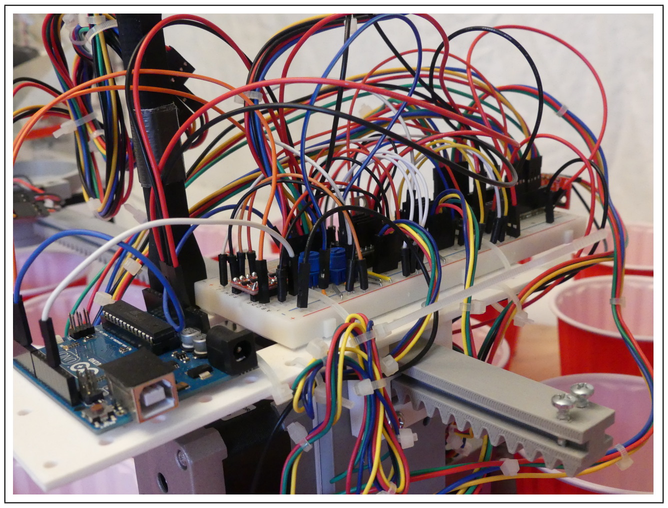


Fig. 45: Module 3 Wiring

2) Slip Ring: Controlling Module 3 was one of the most complicated electrical challenges in configuring the system. It became obvious that many devices would be controlled within Module 3. It was decided that a Teensy would be used to drive the module, and would be connected to a host machine via USB. In order to control the Teensy without compromising the module's wiring, a slip ring in the base of the tower was utilized for the USB communication and power. In testing, the slip ring worked very reliably. Its implementation was not changed over the course of the project.

3) Stepper Control: The stepper motors use the same stepper library as the other two modules. The most significant difference between this module and the others is the fact that both steppers need to be controlled independently of each other.

C. Software

1) Deposit Assignments: Because the sorting system is capable of processing up to 520 unique parts, and there are only 36 deposit locations in Module 3, software is used to determine how classified parts are assigned to physical sorting locations. This is done through the comma-separated value file ”sortArray.csv”. This file associates every line of the file with one of the sorting locations. The first cup is always associated with the misclassified assignment, and is used if a part cannot be classified to one of the other locations. Each subsequent line corresponds with the next cup, and can be filled out with part colors, part ID numbers, or both. If two cups happen to have overlap between assignments, the sorter will place it in the first of the two. This provides users with incredibly granular control over how the sorting system operates, allowing them to fully constrain any cup to as many colors/part ID numbers as they desire. The user can also sort exclusively by color or part ID by swapping in the files ”colorArray.csv” or ”serialArray.csv”. A graphical user interface is currently in development to enable more comprehensive sorting through an interactive interface. Although this is not within the scope of the current MQP, it would add extra value and usability to the overall system.

2) User Control: The ROS software for Module 3 allows for a simple control interface. There are three ROS topics that can be published to and three topics that can be subscribed to. The first ROS topic to publish to is the turntable position ROS topic. This topic takes an angle measure in degrees and then sends the turntable to that angle. The turntable then publishes to another ROS topic stating that the turntable has completed the action of turning. This allows the user to run other operations in the background while waiting for the turntable to complete its motion. The rack and pinion is controlled the same way, except that the rack takes distances in meters. The turntable will take any valid floating point number as an angular position because the input has already been restricted to the range [0, 360]. The rack is restricted to a range of 0 meters to 0.25 meters.

3) Configuration Flexibility: The software and hardware are configured to enable expandability and flexibility of the overall distribution system. This is accomplished through the use of procedurally generated deposit assignments which are extrapolated through variables like the number of containers, their distance from the tower, and the size of the container openings. If the length of the rack, number of rings in the base plate, quantity of overall containers, or many other features of Module 3’s physical design are changed, the software only requires minor variable changes to adapt. This also applies to deposit assignments, as the user is capable of configuring which containers to use and how through the use of simple text files. These files are automatically parsed on start up and mapped to the physical system. This helps to abstract some of the more complex functions of the mechanism and improve the user experience. Future work for this module could also include the development of a simple user interface to further abstract this process.

IX. INTEGRATION

A. *Control System and Networking*

Each module is connected through ethernet to the local area network via an ethernet switch. There are a total of four devices connected to the switch: the Jetson TX2, the Raspberry Pi that controls Module 1, the Pi that interfaces with the USB camera mounted in Module 2, and the Pi connected to the classifying camera at the end of Module 2. These four networked devices communicate with the roscore node running on the Jetson. Module 1 makes use of a Raspberry Pi for the sole purpose of abstracting the rosserial_python node from the Jetson. Module 2 utilizes two Pis for multiple reasons. The first Pi powers the USB camera mounted on the first camera housing in Module 2. This allows it to capture images of a large portion of the belt. The second Pi on Module 2 interfaces with a Pi Camera V2 (for classification), along with the Teensy that drives the belt. Out of all three modules, Module 3 presents the smallest networking challenge. For modules 1 and 2, the Teensy's driving functionality was abstracted from the Jetson by plugging each Teensy directly into their own Raspberry Pi. For Module 3, however, the Teensy is directly controlled by Jetson, meaning that all of Module 3 is driven by the main controller. This is feasible due to the lower processing requirements and the lack of computer vision components in the third module.

1) Jetson TX2: The Jetson TX2 is responsible for running roscore, the rosserial node that abstracts the ROS communication to the Teensy. It also runs a python ROS node called module_3_controller.py that iterates through the state machine driving Module 3. This controller contains the configurable sorting list, and determines the target part positions for pieces classified in Module 2. The Jetson also contains the node for the neural network, receiving an input image from the Pi in Module 2 and outputting both classifications to module_3_controller.py.

2) Networking Between Systems: All systems are connected via ethernet. When a ROS node is initiated on one of the Raspberry Pis, it will immediately begin pinging roscore (which is running on the Jetson) to search for a connection. Once the Pi has established communication with roscore, the ROS node is able to start publishing or subscribing to its respective topics. The data lines not officiated through ethernet are the Teensys, which operate over micro-USB, and the various USB cameras, which do not utilize any ROS code. These ROS nodes are serviced through special packages designed to operate over the USB bus.

B. Electrical Integration

1) Power Delivery: The combination of all three modules requires 12 and 5 volt power. The entire sorter utilizes a 12 volt, 40 amperes power supply. The stepper motors all receive 12 volt power directly off a splitter right off the power rails. The Pis receive power from 12 volt to 5 volt regulators connected to pin 2 for 5 volts and pin 6 for supply ground. Currently, the Pi with the Pi Camera V2 is powered off of AC power because the chassis is blocking the GPIO pins. The lights on modules 1 and 2 are powered directly off of the 12 volt rails. All of the Teensy micro-controllers receive power from the Pis and the Jetson from the USB port that they communicate over.

C. Robot Operating System

The Robot Operating System is an abstraction layer of communication used for linking sensors, actuators, and control boards together in a control system. A typical use case for ROS involves a computer vision system that captures images from a camera and then publishes those images to a ROS topic. This topic is then subscribed to by an algorithm searching for specific features which, when identified, trigger a state machine advance in another ROS node. ROS is also used for linking isolated control systems together so that a main controller can control and monitor multiple lesser systems. This abstraction allows for testing of different control algorithms without disabling the entire system. This modular control scheme enables increased productivity in isolated development environments.

1) Image Topics: This MQP utilizes multiple cameras, therefore there are many ROS topics involved with image pipelines. A ROS node running the usb_cam ROS package's usb_cam_node node will publish images to several ROS topics depending on the image compression desired, such as raw image and JPEG or Theora format. The ROS node that receives images from the Pi Camera V2 only publishes to a single ROS topic because it was custom written in C++ for this MQP. This ROS topic publishes only the raw image format as it is later run through the pre-processing scripts for the classifying neural networks.

2) Neural Network Topics: There are also separate ROS topics designated to the transfer of data relevant to the neural networks. The two main topics utilized are the image transfer topic and the classification topic. One is delivered to the Net Node and the other is published by it. The image data nodes, as described before, transfer a captured image of a part on the belt to the main controller, where pre-processing can be completed in the Net Node. This is then run through both classifiers individually and saves their classifications (both part ID and color) as strings. These strings are then appended and delimited by a comma, and this is published as the classification topic. This topic is then intercepted by a subscriber in the Module 3 controller which can then generate a path for Module 3.

3) Launch Files: There are multiple types of ROS nodes deployed in driving the sorter. Due to the complexity of the system, and the multiple independent controllers utilized, there are many processes that must be initiated simultaneously. As a result of this, the system utilizes launch files, which combine the functionality of the individual ROS node launches into one easily executable operation. For example, in order to initiate Module 2, the user must start the neural network node, the two camera nodes, and the controller for the stepper motor on the Teensy. This is reduced to a single launch file which automatically executes all three operations. This methodology is repeated for the controllers on modules 1 and 3. The use of launch files dramatically simplifies the process of operating this system as their inclusion means only four Jetson terminals and one terminal per Pi are required to operate the sorter. Launch files are also useful in referring to network file paths on the controller's hard drive, like the neural network's '.h5' file. The network's ROS node demands an input path to the network file via arguments. Instead of hard-coding the location of the network file within the ROS node, the launch file is able to store the path of the network file, and pass it to the node using the syntax below.

```
<node pkg="my_package" name="my_name" type="my_script.py"
args="/path/to/network/file.h5" output="screen"/>
```

D. Control Flow

1) High-Level Process: The high level process begins with with the initialization of modules 1 and 3. Modules 1 and 3 must boot first, as they act as slave systems to Module 2's controller. Once the system is fully operational, Module 2 begins requesting parts from Module 1 via a ROS timer publisher. The publisher requests a part from Module 1 every 10-15 seconds. Once a part has been provided to Module 2, Module 1 will publish to Module 2's controller that the part is available for classification. Once Module 2 receives this signal, it will advance the part along the belt to the Pi camera's view where it can be classified. Once the neural network has identified the part, Module 2 will then provide Module 3 with the part's classification. Module 2 then ejects the classified part to Module 3, where it can be distributed to its final location in the circle of cups surrounding Module 3.

2) RQT Graph: The following graph displays the overall layout of the ROS communications throughout the entirety of the system (see Figure 46). This includes nodes, topics, publishers, and subscribers.

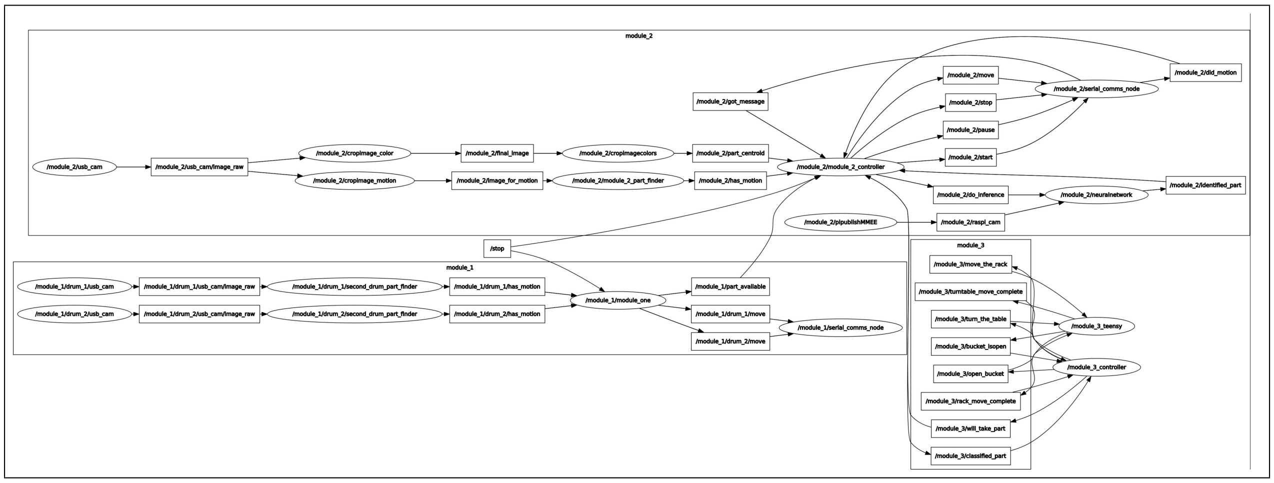


Fig. 46: Full System RQT Graph

Figure 47 shows the RQT graph for Module 1. Here the reader can see that there are duplicate ROS node types. They were renamed in the launch file to separate Tumbler 1 and Tumbler 2. They were initially called drum_1 and drum_2, but the function remains unchanged. The main controller node for Module 1 is the module_one node. This node is responsible for communicating between modules 1 and 2. The two USB cameras also utilize their own identical nodes (drum_1/usb_cam and drum_2/usb_cam) which are responsible for processing the slide images and detecting motion. This then sends a trigger message to the module_one controller which stops the respective tumbler and then, if a part has been detecting falling through the secondary slide to Module 2, informs Module 2's controller of the new part on the part_available topic. Module 1's controller is also subscribed to the /stop node which can be seen in the overall RQT graph. This node sends a stop signal to both Module 1 and Module 2's controllers when it receives a user stop input. This is based around a heartbeat signal which is repeatedly triggered every two seconds. Signalling stop on the user side stops the heartbeat, and if either module does not receive the signal for more than six seconds, it shuts down.

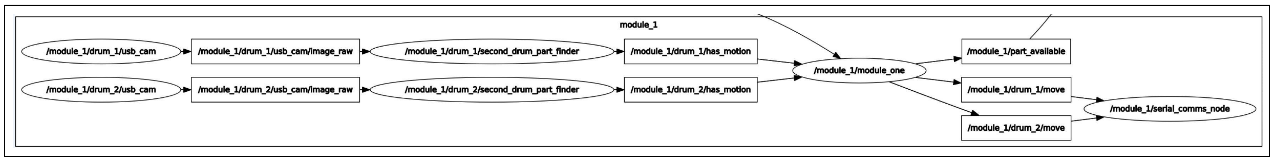

module_1
/module_1/drum_1/usb_cam
/module_1/drum_1/usb_cam/image_raw
/module_1/drum_1/second_drum_part_finder
/module_1/drum_1/has_motion
/module_1/drum_2/usb_cam
/module_1/drum_2/usb_cam/image_raw
/module_1/drum_2/second_drum_part_finder
/module_1/drum_2/has_motion
/module_1/module_one
/module_1/part_available
/module_1/drum_1/move
/module_1/drum_2/move
/module_1/serial_comms_node

Fig. 47: Module 1 RQT Graph

Figure 48 shows the RQT graph for Module 2. This module has the most complicated ROS ecosystem, as it must communicate with the main controller, Module 1, and Module 3's controllers. It also must communicate with the net node or /module_2/neuralnetwork on the do_inference topic in order to infer part identifications. The controller for Module 2 is called module_2_controller, and is subscribed to the topics for motion and position of parts on the belt, the current movement state of the belt, and the inference generator for the part classifications. It publishes only to the belt controls (named move, stop, pause, and start). Module_2_controller is also subscribed to Module 3's controller, in order to stay aware of Module 3's operating state, which can be busy or ready.

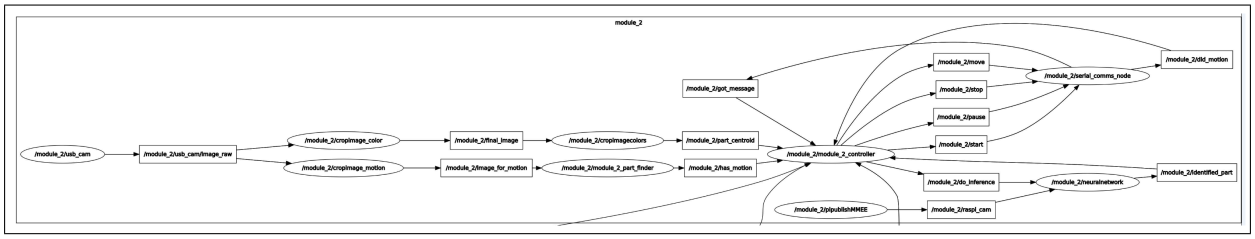


Fig. 48: Module 2 RQT Graph

Figure 49 shows the RQT graph for Module 3. Module 3 utilizes two ROS nodes, labeled module_3_controller and module_3_teensy. The main controller is responsible for informing the Module 2 controller that it has returned to its home position and is prepared to receive a part (on the will_take_part topic). It also subscribes to the classification topic provided by Module 2's controller, which passes in the data required to place the part in a container. Once module_3_controller is aware it contains a part and has determined the correct sorting location, it utilizes a state machine to publish commands to the module_3_teensy ROS node, which contains the code for the stepper motor controllers. The Teensy then sends motor signals to the rack, turntable, and end effector, reporting their movement's completion back to Module 3's main controller. The state machine continues until all movements are completed (implying the part has been dropped in the appropriate container and the end effector has returned to its initial position), then module_3_controller once again publishes that it is capable of processing a new part on the will_take_part topic.

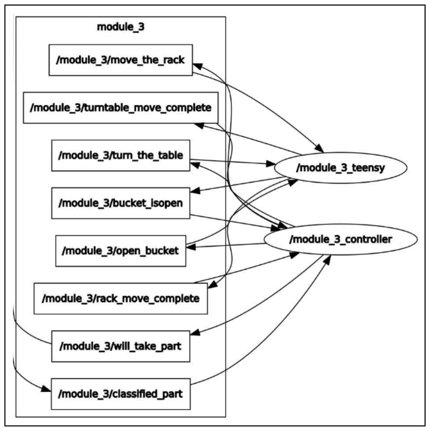


Fig. 49: Module 3 RQT Graph

X. TESTING

A. Testing Module 1

The software to control Module 1 consists of two unique ROS nodes and three non-unique ROS nodes. The ROS nodes unique to Module 1 are the Raspberry Pi camera node, and the entire Module 1 controller node.

The Teensy ROS node simply serves the requests to start and stop rotating the tumblers. The Raspberry Pi camera node simply publishes images to ROS. The Module 1 controller node takes in the information from the motion detectors and the LEGO sorter controller and then activates/deactivates the tumblers accordingly. The other two non-unique nodes consist of the generic webcam publisher node and the motion detector node. The USB webcam node was already written for ROS. It simply publishes webcam images to ROS. The motion detector nodes subscribe to the camera nodes and then publish to a topic specifying motion or the lack thereof.

Many issues were encountered while programming Module 1. The most significant issue is performance, specifically in camera frame rates. Originally, all six nodes relating to Module 1 were designed to run on the Raspberry Pi. After some testing, however, this was deemed impossible. The Raspberry Pi would receive information via a ROS topic a full second after it was sent. This performance is simply unacceptable so, the launch files were rewritten and these ROS nodes were relocated to the Jetson to accommodate these deficiencies.

In order to remedy these issues multiple different configurations were tested. The first configuration implemented involved moving the motion detectors and the main Module 1 controller to the Jetson. This appeared to improve the issue initially, but shortly the system returned to its prior sluggish behavior. By echoing the motion sensor ROS topic it was possible to see that parts would travel past the Pi camera without being announced for nearly two additional seconds. If parts traveling along Module 1 could move undetected, the ROS nodes would have to be reconfigured so that the Raspberry Pi runs as many nodes as possible without performance losses. For comparison, the USB camera published images to ROS at around two frames per second. The Raspberry Pi camera published an image once every second.

The next attempt involved moving the USB camera to the Jetson so that the Raspberry Pi only operated one camera and the Teensy ROS node. After starting up the nodes, the performance changes were immediately clear. The USB camera now published at around 20-30 frames per second. The Raspberry Pi camera, however, published an image once every 3-5 seconds. It is unclear why the frame rate dropped further when the hardware was subjected to even less of a workload, but it was clear that the solution was to offload all image processing to the Jetson. After this test, it was decided that both cameras would be changed to USB webcams running off of the Jetson, as this would eliminate all heavy processing from the Raspberry Pi, leaving only the rosserial node running. Once both cameras were replaced, frames were processed fast enough for the motion detectors to sense parts moving across the ramps in real time.

It should be noted that USB webcams running on Linux platforms can use the v4l Linux driver for hardware interfacing, while the Raspberry Pi camera must utilize a 3rd party library, which allows the camera to be programmed in C++. This is the same ROS node that is utilized in capturing images for the LEGO part database. The Raspberry Pi camera, therefore, is optimized for capturing extremely accurate pictures with its manually adjustable focus. The USB cameras are designed to process as many frames per second as possible, and procedurally adjust their focus. Considering the use case of Module 1, the webcams are a superior choice as they prioritize speed rather than accuracy.

B. Testing Module 2

As was the case with Module 1, most issues relating to the testing of Module 2 are ROS-related. Looking through the documentation for ”rosserial” on ROS’ main website, there is documentation for many derivatives of ”rosserial”, including rosserial_python, rosserial_arduino, and others. When Module 2 was first constructed and programmed with ROS, it was not unusual for the Teensy rosserial_python ROS node to crash, reporting error messages regarding ”rosserial” versions and baud rates between the Teensy and the host device. This is due to the fact that rosserial_python and rosserial_Arduino are two different implementations of communication with Arduino devices (and their derivatives) over ROS. When utilizing the rosserial_python version, the Teensy could not maintain communication with roscore for more than three seconds. Once the rosserial_python version was replaced with the rosserial_arduino version, the Teensy stopped reporting version errors.

The next issue with Teensy and ROS communications involved publish rates. The module controllers for all three modules run at rates ranging from 5 Hz to 15 Hz. After much experimentation, it was discovered that the publishing of messages on Arduinos requires virtual timers to limit the publish rate. It is assumed that this problem has to do with the serial bus baud rate, but because the team has little experience with Linux kernel editing and serial bus testing, it was most sensible to control message publishing via virtual timers. After searching the internet, the virtual timer library provided on this website [2] was identified as a solution to the publishing problem. Once the Teensy had begun publishing at discrete intervals without interrupting its own hardware, it was able to maintain communication with the host device without packet loss. After much testing, the team discovered that the maximum publishing rate was once every quarter second. So as to not strain Module 2, the virtual timer was set to half a second. Module 1 did not suffer from this issue as Module 1 only subscribes to ROS topics; there is no publishing of messages. Module 3, however, also required the use of virtual timers. Both Module 2 and Module 3 incorporate digital encoders, so removing unnecessary computational stresses from their respective Teensys was a high priority.

Unfortunately, these were not the only Teensy-related issues. ROS nodes running on the Jetson publish exponentially faster to other Jetson ROS nodes than to rosserial nodes. This communication lag demanded the use of feedback in ROS messages. This feedback message would simply publish a ROS message of type ”Empty” back to the controller, indicating that the target node had received the command. The controller could then publish to start the belt, wait, and then publish again if the message was not received. This ”Empty” message publishing occurs within the same virtual timer as the belt position and ROS debugging publishers.

C. Testing Module 3

The testing process for Module 3 focused on the individual tasks of turning the tower, extending the rack, and opening/closing the end effector. Testing the turntable function involved publishing the location of the turntable to ROS, in order to convey the desired position to the Teensy. Once the current angle of the turntable is identified (through the use of encoders), the Teensy calculates the direction in which the tower must rotate to ensure that Module 3 is on the shortest path to its target position. In order to stop correctly, the controller mandates a delta value for the difference between current and desired angles. Floating point numbers cannot be compared for equality directly, so the upper and lower bounds were applied to the desired position and used as a range for the turntable to stop within. The faster the rotation, the larger the delta value required to detect the completed motion. The turntable’s hardware functionality was also made compliant in that the turntable can be turned manually while the stepper motors are not driving. This allowed for more rapid testing of the turntable’s functionality. Once the the tower was reliably rotating to within two degrees of the target position, testing focus was directed towards the rack.

The rack was tested in a manner very similar to that of the turntable. After the software was completed, along with the accompanying state machine, and the appropriate ROS nodes and topics were linked between the Module 3 controller and Module 3 itself, false classification commands were sent to the controller to mimic the behaviour of the system. The responses were then gauged against the input, and constants were adjusted to ensure that the movement of the rack matched perfectly with the radii of the rings in Module 3’s base. The rack has hardware limits that must be taken into account during both testing and operation in order to ensure that the motors do not overshoot the physical limits of the gear. This necessitated the creation of launch commands to drive the rack in reverse to its home position at the start of every run. This ”homing command” allowed the encoder to record the starting position of the rack once the endstop has triggered a limit switch on the tower. This starting position was then used as a relative zero for all future measurements, ensuring $<$1mm accuracy on the rack’s positional adjustments.

The testing process for the bucket was similar, and involved sending signals to the Teensy, which would in turn send a signal to the 5V Arduino to trigger the servo's open and close. This was very reliable. Once this step was complete, effort was focused on increasing the overall speed of the system, which was done by parallelizing the movements of the turntable and the rack, as well as removing ROS delays and timing delays from the underlying code. The delta values for the position of the turntable and rack were set liberally at first, giving a large amount of freedom to the final position of the end effector. This was then reduced progressively, until the maximum speed/minimum offset pairing was identified. With Module 3 capable of keeping pace with the demands of the overall system, the module was integrated back into the full sorter and tested in tandem with the other modules.

XI. RESULTS

1) Speed: The sorting frequency target for this MQP is to classify and distribute each part in 15 seconds or less. This means that, when a part exits Module 1 (which can happen at unpredictable intervals), it should take no longer than 15 seconds to reach its final destination. In its initial state, without applying speed optimizations, the sorter requires 45 seconds to move a part from the exit of slide 2 to its final destination in Module 3. After tuning motor speeds and timing intervals in modules 2 and 3, this time was reduced to 13 seconds, well within the target for the project.

Due to the unpredictable nature of parts in the tumblers of Module 1, it is difficult to guarantee a turnaround time for larger sets of parts (as this sorter is geared towards providing low-maintenance sorting for large batches of LEGO pieces). However, the expectation is that since part requests to Module 1 are on a ten second timer (or faster, depending on the speed of Module 2), and modules 2 and 3 regularly require less than 13 seconds to fully process a brick, sorting a kilogram of LEGO pieces (or roughly 700 parts) should require roughly two and a half hours. This means that from the end of the average workday (9am-5pm working hours) to the beginning of the next day, or 16 hours, the system could output approximately six and a half kilograms of sorted bricks. This, however, assumes an expansion of Module 1's hopper to accommodate such a large input (while avoiding jams), and that the containers of Module 3 are expanded as well. The current system is equipped to handle smaller tasks but is easily scalable.

2) Accuracy: The tables below demonstrate the accuracies (as a percentage of the overall image database) across the most common part types in the project's target classification set.

Note the lower accuracy values of the 3001 and 3003 bricks, which are difficult to distinguish due to their identical top faces to the 3020 and 3022 bricks. 72 percent is the worst observed accuracy across all part IDs, and 100 percent is the highest. The 72 percent measurement was found when testing the 3001 piece, a 2x4 studded brick, whereas its 2x4 plate counterpart, the 3020 piece, yielded 100 percent accuracy.

TABLE I: Accuracy of the Part ID Classifier

Part Serial:	Accuracy (%)
3001	72
3003	73
3002	74
3039	74
3622	93
3005	94
3010	95
3665	96
3004	98
50950	98
3022	99
3710	100
3020	100

Training the part ID classifier was an extremely demanding task. Due to the massive size of the training data set, a single run through all the data (or an ”epoch”) could take upwards of 50 minutes. However, in order to train the data for maximum accuracy, multiple training epochs were required. So, a script was created to sequentially increase the number of epochs and then run a the training process. This process starts at one epoch and ends when the network begins to overfit (the testing accuracy starts to decrease). The graph below shows the accuracies of each of the part ID classifications and how they fluctuate across multiple epochs (see Figure 50). The notable developments here are in the progress of the 3001 and 3003 LEGO pieces. These are brick pieces with stud patterns 2x2 and 4x2. Their accuracies tend to suffer, rarely exceeding 80 percent. Although this is actually not terrible for a classifier with 13 classes, it is something the team sought to correct or at least explain. The reason for this seemingly low accuracy has to do with another class in the node: the 3020, and 3022 parts. These parts are the ”plate” equivalents to the 3001 and 3003 pieces. This means they have an identical stud pattern, but are slightly shorter. These plate variants have very high accuracy (>95 percent). It was discovered that as trained individually, both types of parts, bricks and plates, are very easy to classify, but, when trained together, the plates tend to be a much more popular classification. This could likely be resolved through the addition of a second camera to judge height, or possibly more direct lighting on the camera-facing plane of the incoming part.

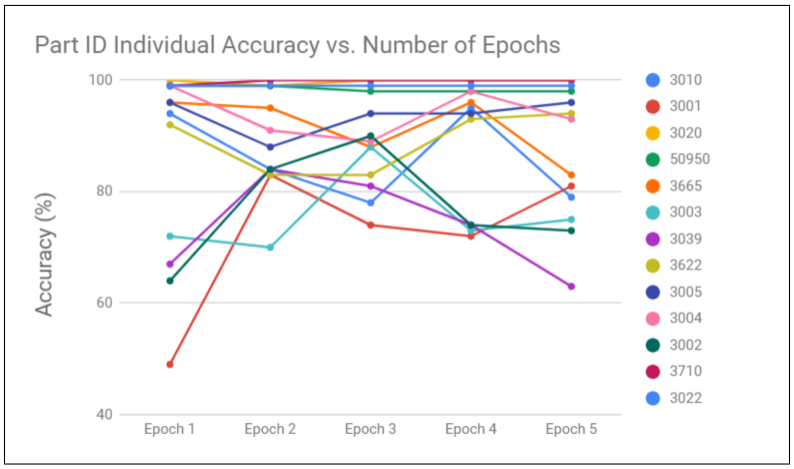


Fig. 50: Individual Part ID Accuracy across Epochs

Figure 51 demonstrates the average accuracies of the entire network on all part IDs across five epochs. Although it is less clear in Figure 50, here the reader can recognize how the accuracy of the total classifier converges on 90 percent at the fourth epoch. This is less evident in the previous graph because some accuracies, in particular those of the brick pieces, decrease as they enter the fourth epoch. The maximum achieved accuracy is 89.7 percent overall. This could be dramatically increased to 96+ percent with the addition of a metric to faultlessly distinguish between the plate and brick pieces.

Like the part ID classifier, the color classifier was trained over a large number of epochs. This was possible due to the much smaller data set used to train for color. Although there are the same number of total images, the size of each is 256 times smaller, meaning each run of the classifier only required approximately a minute of training time. Multiple epochs were tested and the accuracies of the network were recorded for each individual color as well as the overall average. Figure 52 shows the accuracy of all 40 color classifications across five epochs. The meaningful feature of this graph is that although the accuracies seem erratic at times, it is

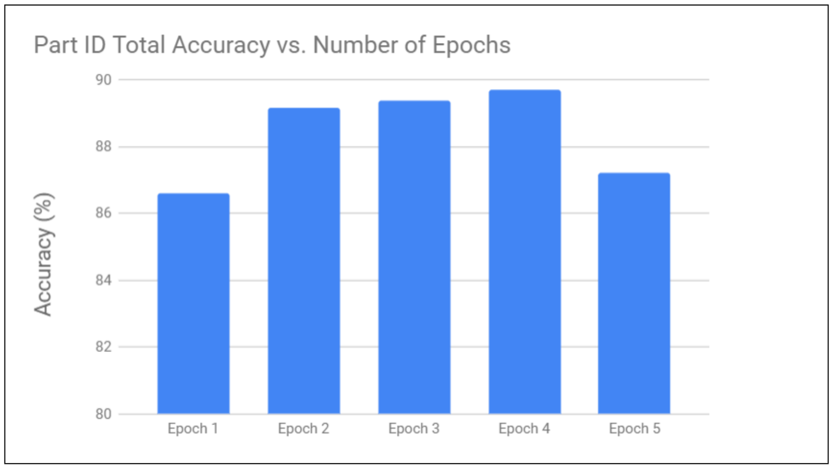


Fig. 51: Average Part ID Accuracy across Epochs

obvious that they are converging towards a maximum value of 100 percent between epochs 1-4, whereas they once again begin diverging at epoch 5. This indicates overfitting of the data, which means the network has too heavily weighted its training set, and is no longer effectively able to classify new information. The maximum average accuracy is achieved at four epochs and rests at 98.7 percent, a respectable result for such a large number of classifications.

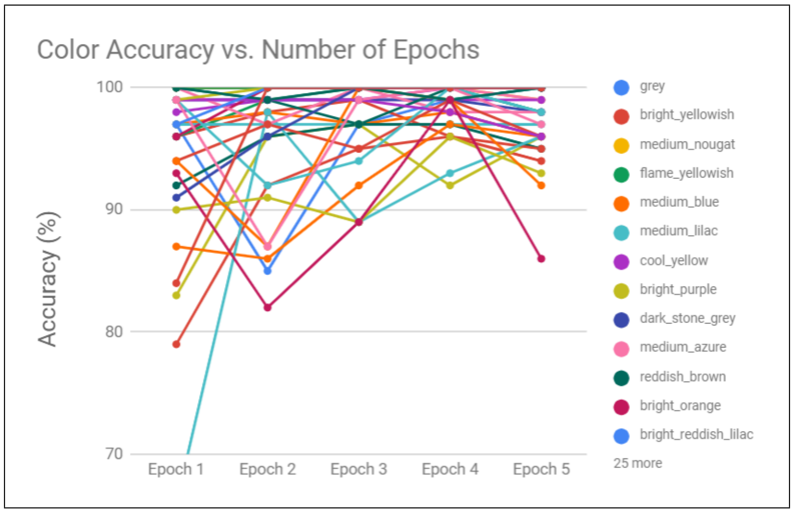


Fig. 52: Individual Color Accuracy across Epochs

Figure 53 demonstrates the average accuracies of the entire network on all colors across five epochs. This chart makes the point of overfitting far more evident, as the accuracy drops a full percent on the fifth epoch. This facilitated the choice to utilize the network trained at 4 epochs in the neural network ROS node for Module 2.

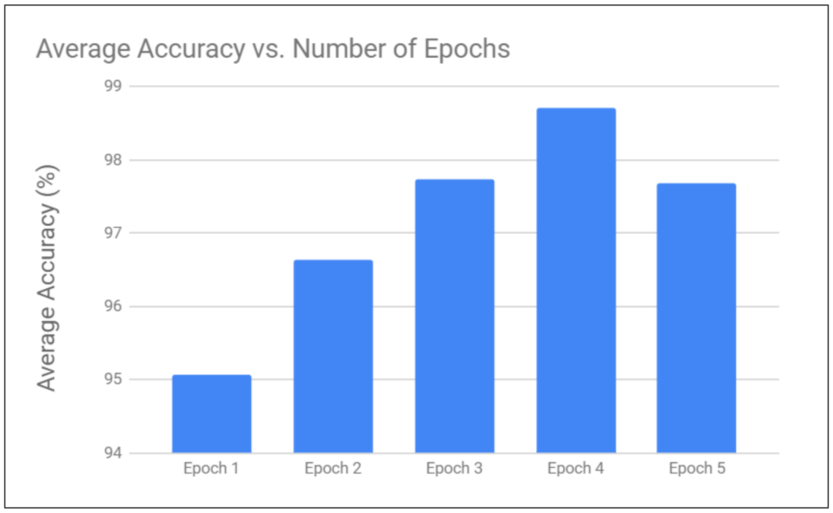


Fig. 53: Average Color Accuracy across Epochs

XII. SOCIAL IMPLICATIONS

The purpose of an industrial robotic system, in general, is to eliminate labor that fits into one of three categories: dirty, dull, or dangerous [11]. In the case of this venture, the work in question is undoubtedly dull. Sorting through hundreds or even thousands of unique parts with the goal of correctly organizing them is time-consuming, low-skill labor. The challenge in this situation is that although the work may be low-skill, it is still technically very complicated. Unlike traditional pick and place tasks, it is actually quite difficult for a machine to distinguish between the hundreds of different LEGO part color and type combinations. Even a human would struggle to determine the difference between ”grey”, ”light grey”, ”very light grey”, and ”pearl light grey” with the naked eye. Without picking up and rotating the part in space in front of a computer vision system, many individual part types are so similar they could easily be confused in a purely mechanical sorting system. The purpose of the LEGO sorter is to apply modern technology to a tedious and outwardly simple process, eliminating the need for human input, and therefore preserving man-hours for more productive work.

XIII. CONCLUSIONS

This project set out to automate the process of sorting large batches of unclassified LEGO bricks. The goal was to be able to classify 500+ unique LEGO bricks by their part ID and color.

This would be done in a modular sorting system, targeting youth centers and schools, who could make use of a system like this overnight to negate the need for manual sorting of parts utilized during the work day. The solution derived involved three modules, each individually responsible for a part of the sorting process. Module 1 would serialize large batches of parts into a single stream, Module 2 would localize and classify the part by color and its part ID, and Module 3 would deliver the part to a desired sorting location. The full system is designed utilizing computer vision for motion detection, fine motor controls, and multiple neural networks. It uses Robot Operating System to communicate between each of its modules, tracking and signalling the process of parts through the overall system. The final system is capable of fully processing a received part in under 13 seconds, and classifies color and the part ID with accuracies 98.7 and 89.7 percent accuracy respectively. Although there is room for improvement in both the design and the accuracy of the system, as a new MQP and provided the scope and breadth of such a project, the resulting system is both functional and technically sound. The team believes that there is ample potential for future expansion of the project, and that the current system is an excellent basis for future innovation in artificial intelligence, physical design, and automation.

XIV. FUTURE WORK

A. *General Improvements*

Due to the large scale of this project, and the fact that this was a new MQP with no previous work to build off of, there are many opportunities for optimization and design changes. Many of these center around physical consolidation of the device and ease-of-use changes to the system that would make it more marketable. The first general improvement that could be made would be in price. The full system cost upwards of $3,000 to build. This is primarily due to the high costs of microcontrollers, Linux computers, the Jetson TX2 (the wires, headers, and connector housings alone cost $200), and also a monitor, keyboard, mouse, and KVM switch setup. Each module uses its own dedicated microcontrollers, in some cases a combination of more than one (both modules 1 and 2 require a Raspberry Pi and a Teensy 3.6 to operate, and Module 3 utilizes a Teensy and an Arduino). This simplifies the modular aspect of the project, as each can be tested and developed independently, but it dramatically increases the overall price. More time could be dedicated to finding alternative solutions to the electrical problems described above. Also, more of the controls that were outsourced to the Teensys and Raspberry Pis could be moved back to the Jetson. This would not only simplify ROS communications, it would also reduce the wiring conflicts of the overall system, and reduce costs. It is also possible that with a more light-weight neural network and computer vision implementation, the entire system could be run off of a less powerful controller, like the Jetson and a single Pi, albeit at the cost of performance and communication bandwidth.

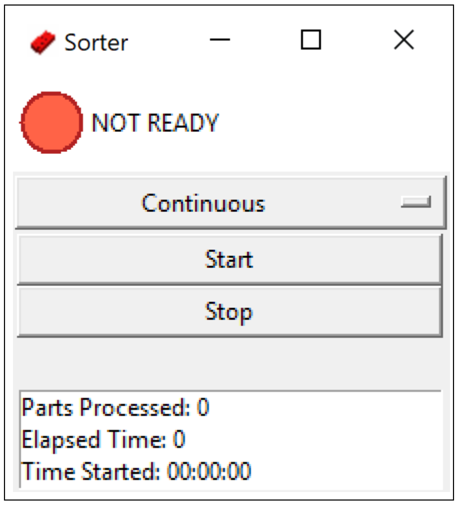
The second major overall improvement to the system would be form factor. As the target of this project was to make the system modular, little thought was put into consolidating the physical design into the smallest possible space. However, once the modules are integrated they consume what would to most be considered an impractical amount of room. A whole MQP could be dedicated to minimizing the current system to a more sensible form. A common theme was the idea of stacking the three modules vertically in a server rack-type enclosure, that would allow the user to keep the sorting capabilities of the original design, while avoiding the space constraint issues of the horizontal mounting. Module 2 could also be dramatically reduced in size. The current form factor is simply limited by the size of sanding belt purchased, and could be hypothetically reduced by more than half.

Another overall improvement for ease-of-use is actually partially developed already. This being a GUI to drive the entire system. Currently, the system requires at least some expertise of the source code to operate. Although this is reasonable to expect of robotics students experimenting with it, a more marketable design would require a much more abstracted, high-level interface with all the functionality a user would expect. This would include a configurable sorting parameter list, as well as intuitive stop/start/pause commands, and real-time usage metrics and feedback (see Figure 54). This would dramatically improve the usability and appeal of this system as a commercial product.

B. Network Possible Improvements

Another area for improvement is in the classifying neural networks. This software is the heart of the MQP, and without reliable classification, it is problematic to utilize. Although the current system has incredibly reliable classification for both part ID numbers and colors in ideal situations, it still struggles in unreliable lighting and brightness conditions. It also tends to struggle with certain specific parts like 2x4 Bricks and 2x4 Plates, which appear nearly identical and therefore report very similar feature recognition outputs (see Figure 55).

For the lighting concerns, the most obvious solution would be to enclose both the Scanner (for data collection), and the entirety of the classifying module (Module 2) within a shroud,



Parts Processed: 0
Elapsed Time: 0
Time Started: 00:00:00

Fig. 54: Sorter GUI

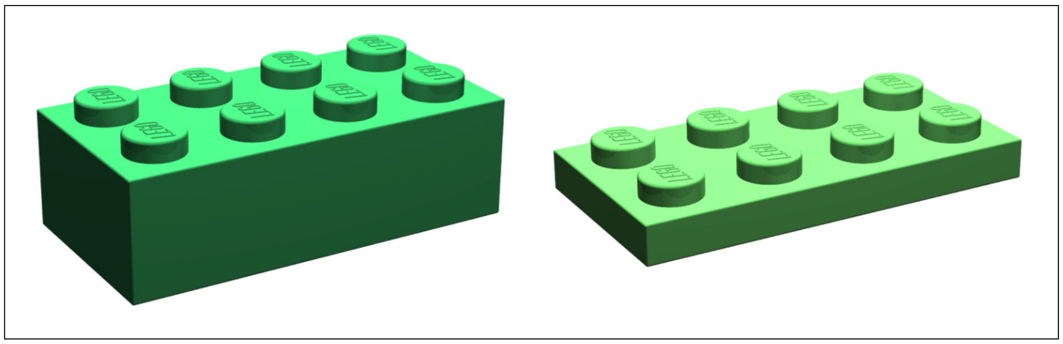


Fig. 55: A 2x4 Brick (3001) and a 2x4 Plate (3020)

with constant reliable lighting. This would guarantee that no unexpected shadows or variable lighting could compromise the classification of the part or the colors. This would be a fairly simple modification, and would not require any changes to the actual network structure. Another issue that arose relates to the classification of physically similar parts, like the two described above. This could be solved in two ways, but as this is a visual imagery issue, both involve the implementation of a new camera. First, a second camera could be added to Module 2, with a horizontal profile view of the passing part. Then, the network could be trained for both types of images simultaneously, and when passed through the network, it would utilize the second view to identify features like profile surface area and general size. This would solve the issue of the 45 degree view misclassification. This would however, vastly increase the training overhead, as double the training and data collection would be required to prepare the network for each individual part. Another camera would also need to be added in a similar position on the Scanner. A simpler solution to this issue would be to use a depth-capable camera. This would generate a point cloud data structure for each part, rather than a 2-dimensional image. Very little would have to be changed in the network structure to incorporate this change. The value of this implementation would be that from any angle, the height and physical volume of the brick could be taken into account as a feature, solving the issue of misclassifying visually similar parts from one angle. The reason this was not employed in this version of the project is that currently, depth-sensing cameras only operate at much larger distances and are incapable of macro-focus. Any product currently available that would be capable of these is far too expensive to be included in the scope of this project. It is possible that future camera technology will make this a much simpler and less costly upgrade.

C. Module 1 Possible Improvements

Module 1 is the only section of the full system where purely mechanical improvements can be made. The current control system is very reliable (in terms of serializing components and consistently providing parts to Module 2), but it could be improved upon with further iterations. In particular, although the system is currently capable of sorting 15 bricks that are completely unique physically, it could be possible to further expand this to accommodate even more bricks. This could include parts from LEGO's Technic line which has smaller and more intricate pieces, the use of which is common in beginner robotics sets. This could be done with updates to the tumblers and the slides, or perhaps some other type of unexplored system like a shaker table (another proposed approach for Module 1's serializing hardware). This could enable a more reliable, higher-paced serializer with even more input options. The module could also be reduced physically, while maintaining the same functionality. This would help to cut back on some of the empty space in Module 1's chassis, further compressing the overall form factor of the system.

D. Module 2 Possible Improvements

Module 2 is the system component that underwent the most physical design iterations, but still has some room for improvement. The current system relies on a sanding belt for transporting parts across a relatively long distance (compared to the other modules). This travel distance could be reduced, and the time loss could be reclaimed in a number of ways. The first method would be to relocate the cameras used for classification and localization closer together, along with moving them closer to the edge of the belt. The current system does not make use of the belt's full length, and does not need extra travel length to improve the performance of the system. In fact, due to vibration in the stepper motors, the larger the travel distance on the belt, the less likely the part will be in a reliable position by the time it has reached the classification camera. In the first iteration of Module 2, the belt was half as long as the current iteration. This setup would operate satisfactorily if the cameras were closer together and right up to the edge of the belt, however, this would require a more reliable way to append shorter belt sections, something that was attempted earlier in the MQP. This truncated system would also require more efficient timing, and the control software for both the delivery of parts to Module 2, and the classification of parts would have to be significantly more sensitive, due to the lower capacity of the shorter belt. In an ideal situation, where Module 1 can deliver singular parts with 100 percent reliability, Module 2 would simply be a bucket or tray with cameras above it. The part would then be classified and immediately dropped into Module 3 for distribution. The belt system currently employed is designed to move the part directly in front of the camera at the correct position, but with more reliable part delivery, this entire step could be eliminated. This simplification and size-reduction of the overall system could be a goal for future MQPs.

E. Module 3 Possible Improvements

Module 3 is another area where improvements could be made. The improvements to Module 3 would be primarily in speed and packing density. In order to improve speed, it is likely that another physical manipulator would be required. Other ideas explored are a 3D gantry system, or a planar parallel manipulator. Either of these would be more stable and hypothetically faster than the current solution, but they would also be significantly more expensive, and would not dramatically improve the packing density of the storage containers. The solution to this is similar to that of the solution for consolidating the entire system: adding a third dimension to the sorting array. This would constitute giving the manipulator a third degree of freedom, and developing some sort of elevator system to reach more arrays of containers. This could easily be done by a 3D gantry, similar to the type of system found in most 3D printers. Although this would add cost, it would expand the sorting capabilities from 36 individual parametrized assignments to as many as would physically fit in the designated space. As LEGO pieces (and other sortable objects) come in far more than 36 permutations (the current system can sort more than 520 unique parts), this would be a valuable addition to the overall system.

XV. ACKNOWLEDGMENTS

A. Worcester Polytechnic Institute

B. Primary Advisor: Professor Craig Putnam

C. Secondary Advisor: Professor Bradley Miller

REFERENCES

[1] Abdelfattah, A. (2017, July 27). Image Classification using Deep Neural Networks - A beginner friendly approach using TensorFlow. Retrieved October 18, 2018, from https://medium.com/@tifa2up/image-classification-using-deep-neural-networks-a-beginner-friendly-approach-using-tensorflow-94b0a090ccd4

[2] Avdweb. (2018, September 28). Non-blocking Virtual Delay timer for the Arduino. Retrieved March 19, 2018, from https://www.avdweb.nl/Arduino/timing/virtualdelay

[3] Brick Owl. (2019). LEGO Bright Green Plate 2 x 4 (3020). Retrieved March 1, 2019, from https://www.brickowl.com/catalog/LEGO-bright-green-plate-2-x-4-3020

[4] Brickformula. (2014, April 23). Lego Parts Sorter Version 1.0. Retrieved November 21, 2018, from https://www.youtube.com/watch?v=m7Gs6-6p7qw

[5] Cedricve, & Funkmeisterb. (2018, May 31). Cedricve/raspicam. Retrieved January 8, 2019, from https://github.com/cedricve/raspicam

[6] Cs231n. (2019). CS231n Convolutional Neural Networks for Visual Recognition. Retrieved October 26, 2018, from http://cs231n.github.io/neural-networks-1/

[7] CTR Electronics. (2019). Voltage Regulator Module P/N: 14-868277. Retrieved February 19, 2019, from http://www.ctr-electronics.com/control-system/vrm.html

[8] Deeplearning Stanford. (2013, April 8). Feature extraction using convolution. Retrieved March 7, 2019, from http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

[9] EtherCAT Technology Group. (2019). ETG News. Retrieved February 2, 2019, from https://www.ethercat.org/default.htm

[10] Luni64. (2019, March 15). Luni64/TeensyStep. Retrieved December 12, 2018, from https://github.com/luni64/TeensyStep

[11] Marr, B. (2017, October 16). The 4 Ds Of Robotization: Dull, Dirty, Dangerous And Dear. Retrieved March 12, 2019, from https://www.forbes.com/sites/bernardmarr/2017/10/16/the-4-ds-of-robotization-dull-dirty-dangerous-and-dear/#22630cd83e0d

[12] Mattheij, J. (2017, June 23). How I Built an AI to Sort 2 Tons of Lego Pieces. Retrieved October 3, 2018, from https://spectrum.ieee.org/geek-life/hands-on/how-i-built-an-ai-to-sort-2-tons-of-lego-pieces

[13] McMaster-Carr. (2019). Gears and Gear Racks. Retrieved February 5, 2019, from https://www.mcmaster.com/gears

[14] MEGAFACTORIES. (2011, August 17). LEGO facts. Retrieved March 3, 2019, from https://www.nationalgeographic.com.au/history/LEGO-facts.aspx

[15] Modbus. (2019). Modbus Organization. Retrieved January 18, 2019, from http://www.modbus.org/

[16] Oracle. (2017). What Is a Socket? Retrieved February 24, 2019, from https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html

[17] Pimoroni. (2019). Raspberry Pi Camera v2.1. Retrieved January 14, 2019, from https://shop.pimoroni.com/products/raspberry-pi-camera-module-v2-1-with-mount

[18] PJRC. (2019). Servo Library. Retrieved February 10, 2019, from https://www.pjrc.com/teensy/td_libs_Servo.html

[19] ROS.org. (2018, August 8). ROS/Introduction. Retrieved October 11, 2018, from http://wiki.ros.org/ROS/Introduction

[20] ROS.org. (2019). Is ROS For Me? Retrieved October 11, 2018, from http://www.ros.org/is-ros-for-me/

[21] Rouse, M. (2018, June). What is UDP (User Datagram Protocol)? Retrieved February 6, 2019, from https://searchnetworking.techtarget.com/definition/UDP-User-Datagram-Protocol

[22] Rouse, M. (2019, February). What is TCP/IP (Transmission Control Protocol/Internet Protocol)? Retrieved February 27, 2019, from https://searchnetworking.techtarget.com/definition/TCP-IP

[23] Sparkfun. (2019). Bourns Absolute Encoder (EAW0J-B24-AE0128L). Retrieved January 13, 2019, from https://www.sparkfun.com/products/15036

[24] The Brothers Brick. (2018, December 28). Sorting LEGO - how do you actually get it done? Retrieved January 12, 2019, from https://www.brothers-brick.com/2010/06/27/sorting-LEGO-how-do-you-actually-get-it-done/

XVI. APPENDICES

A. Supplementary Materials

Accompanying this project is a folder containing all software and CAD work completed by the MQP team. This is available to the public, and contains organized folders which span all aspects of the sorter's design.

Included within is:

- CAD: All final design and SolidWorks files for all modules of the sorter and Scanner, as well as other accessory designs which are not directly part of it. Also includes engineering drawings, '.stl' files, renders of the components and modules, the assemblies for the main structures, and their sub-assemblies, including motors and cameras.
- Scripts: All testing scripts for computer vision, data organization, initial data capture, variation, image pre-processing, and more. Legacy scripts that were not used in the ROS implementation, and scripts excluded from the final system entirely like the preliminary UI and web browser data collector are also included.
- System Images: Real life images of the full system, as well as image documentation and videos of the design process.
- Report Images: All images utilized in the report.
- Network Files: All files associated with the network design for both the color and part ID sorter and saved network '.h5' files. Also includes legacy network designs like the original fully-connected network.
- ROS Code: All ROS code for each module as well as the Scanner and network nodes. All ROS CV scripts, and Arduino/Teensy logic.
- Training Data: A significant portion of the original training data is included for future training/reference. Includes the training, validation, and testing data folders for both the part ID and color networks. This data has already undergone pre-processing.

Each folder contains a README.txt file which describes its contents. The contact information of each team member is also available for continuations or questions about the project.