Articulated Robot Arm
Dobot V1.0

Andrew Showers, Dylan Elliott, Peter Horak

Rensselaer Polytechnic Institute

December 16, 2016

Table of Contents

Executive Summary

This report describes how we use the *Dobot Arm V1.0* to perform object manipulation tasks in a limited workspace. We attach a camera to the Dobot arm in order to provide visual feedback. The arm is therefore able to detect, locate, and track objects within its workspace. The results in this project demonstrate fundamental abilities such as control and visual detection as well as higher level behaviors. The software environment is developed to be highly customizable to allow for rapid development of further capabilities.

This report is organized into three main sections. First, the control interface for the Dobot is introduced and explained, outlining how the Dobot communicates with a user and navigates around its workspace. Second, the vision system is defined and optimized, demonstrating that the added sensory information can provide the Dobot with detailed information about its surroundings. Finally, the integration of control and vision with high-level code is demonstrated with the behavior model. These three sections combine to result in a stable robotic arm platform capable of performing a variety of tasks.

Introduction

The Dobot arm is a 4 degree of freedom (DOF) robotic arm. It supports a variety of interchangeable, downward facing end effectors such as a metal gripper and an air pump powered suction cup. We develop our project concurrently with students a project from MIT called *Duckytown*. *Duckytown* consists of a small scale model town in which mobile robots called *Duckybots* autonomously transport rubber duckies while following traffic laws. The goal of our project is to provide a means of loading rubber duckies onto the Duckybots using the Dobot arm as the manipulator. In order to achieve this goal, we add a camera to the Dobot arm to provide visual feedback, allowing the arm to detect the presence of mobile robots and determine their location within the field of view.

We organize the project into three subsystems that extend the out-of-box capabilities of the Dobot arm; Control, Vision, and Behavior. The behavior subsystem is the highest level subsystem depends on kinematic control and vision. Figure 1 shows the subsystem dependencies.

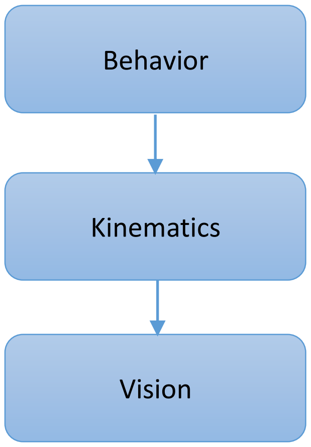

Fig. 1. Subsystem dependencies

The ability to visually locate a Duckybot and load a rubber ducky onto the Duckybot demonstrates simple object manipulation similar to many pick-and-place applications. Thus, our project extends the applications of the Dobot arm to many other tasks such as debris removal, object stacking, object sorting etc.

Hardware

The design and orientation of the Dobot V1.0 arm can be seen in Figure 2 [1]. The arm features a base joint with a range of [-135°, +135°], a rear-arm joint with a range of [-5°, +85°], a forearm joint with a range of [-10°, +95°], and an end effector joint with a range of [-90°, +90°]. The mechanical design of the Dobot is such that the end effector will stay level during movement, therefore the only rotation that can occur to the end effector during movement is about the vertical axis. The Dobot utilizes an in-house low level controller consisting of an Arduino mega 2560 and an FPGA running proprietary code. The serial communication protocol for the Arduino is open sourced though, which allows for the development of applications starting at this level. We attach a Logitech C270 webcam to the end effector of the Dobot, which provides a downward facing view of the workspace. The suction cup end effector option is used for this project because it can easily grasp the top of a rubber ducky.

Software

This project is developed in Python 2.7 and is multi-platform, provided your OS can run Python and administrator access is granted to utilize serial ports. Python is chosen due to the simplicity it offers when developing multi-subsystem projects and communicating over serial ports. Additionally, OpenCV is used to handle and process the webcam video frames, as OpenCV is a well-documented image processing library that is very efficient.

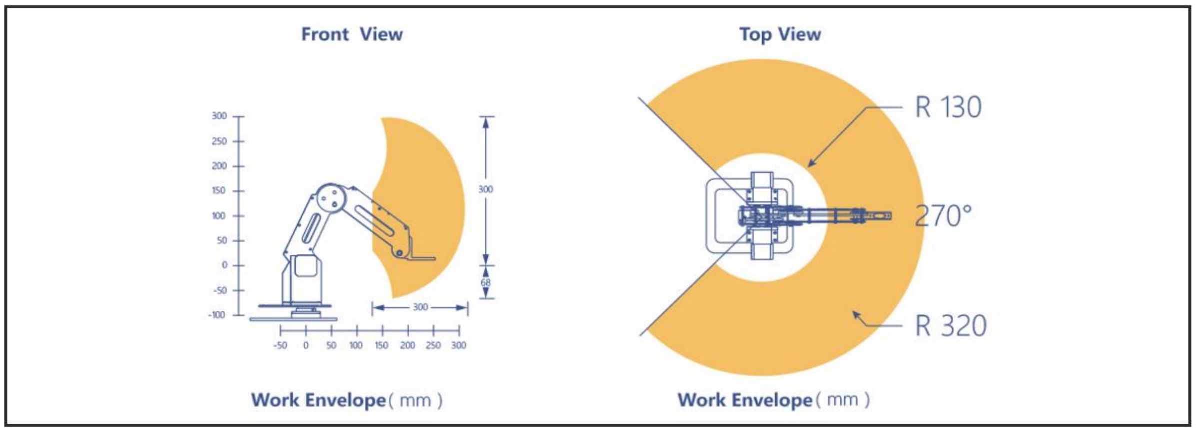

Fig. 2. Dobot specifications (dobot.cc) [1]

Overview of Approach

The ability to detect and locate a Duckybot and load a rubber ducky onto the robot involves multiple underlying capabilities. First, the ability to move the robot arm to a desired location is obviously an important baseline requirement. This capability involves the integration of software to encode and decode messages being sent and received on the Dobot's serial port. We utilize a low level set of Python modules to provide this communications framework to the Dobot. We mark objects with printed AR tags in order to detect and locate them in the Dobot's workspace. The AR tags provide information such as their location, orientation and identity to the Dobot. These capabilities are organized into a set of python modules in order to allow robust cooperation between kinematic control and situational awareness. This high-level development allows for rapid prototyping of useful capabilities and easy debugging.

Subsystem: Control

The purpose of the control subsystem is to provide an interface to the Dobot that can be used by the high-level behavioral code. The control subsystem includes the Dobot serial interface, kinematics, and path planning modules.

Serial Interface

We use pyDobot by Nikolas Engelhard [2] as the basis for our serial interface for the Dobot. It provides methods to send desired joint angles to the Dobot and get the current joint angles of the Dobot among other things. It works as a wrapper for the Dobot serial protocol as described in the Dobot Communication Protocol [3]. In order to pick up duckies, we extend the pyDobot serial interface to turn the suction cup pump on and off. We also include three new features to prevent undesired behavior that can occur with the original version of the pyDobot serial interface.

First, we check that requested joint angles are within the Dobot's range of motion before sending them to the Dobot. Failing to perform this check can cause the Dobot to become stuck in poses outside its range of motion and cause the stepper motors to slip. We use conservative joint constraints based on empirical testing of the Dobot's motion limits. Equations 1-4 lists these joint constraints.

$$-135° \leq \theta_0 \leq 135° \quad (1)$$

$$0° \leq \theta_1 \leq 60° \quad (2)$$

$$0° \leq \theta_2 \leq 60° \quad (3)$$

$$35° \leq \theta_2 - \theta_1 \quad (4)$$

Second, our serial interface blocks until the Dobot completes a requested movement. This prevents the high-level behavioral code from sending commands too quickly, which can overflow the Dobot's internal command buffer and cause it to skip waypoints and behave unpredictably. Our solution works by checking the Dobot's current joint angles against the target joint angles and waiting in a loop until they agree to within 0.001°. This feature eliminates the need to hard-code delays in our behavioral code.

Third, our serial interface checks whether new requested joint angles differ from the last angles sent. If they do not, the serial interface will not send the joint angles. If the suction cup pump state changes in the new command though the joint angles do not, then the serial interface sends a jog packet to turn the pump on or off without changing the joint angles. While not required in theory, in practice this added functionality eliminates an issue where the Dobot can become unresponsive after receiving duplicate joint angles.

We also provide a wrapper script that allows keyboard-based control of the Dobot using both absolute joint angles and jog mode. In jog mode, the user sends a joint and speed to the Dobot, which rotates that joint until commanded to stop. We use the curses library to manage user input and output. The keyboard control can record Dobot poses, which is convenient for calibration purposes.

pyDobot combined with our modifications provide a reliable interface to send joint angles to the Dobot. However, it is often convenient to work in Cartesian coordinates. The next section discusses how the kinematics module can be used to translate between the Cartesian coordinates of the Dobot's end effector and joint angles.

Kinematic Model

Figure 3 depicts the kinematics of the Dobot. The Dobot is unusual in that it defines the second joint angle relative to the rotating base frame rather than the preceding arm link. Additionally, the Dobot has a decoupled architecture that maintains the end effector in a fixed orientation relative to the rotating base.

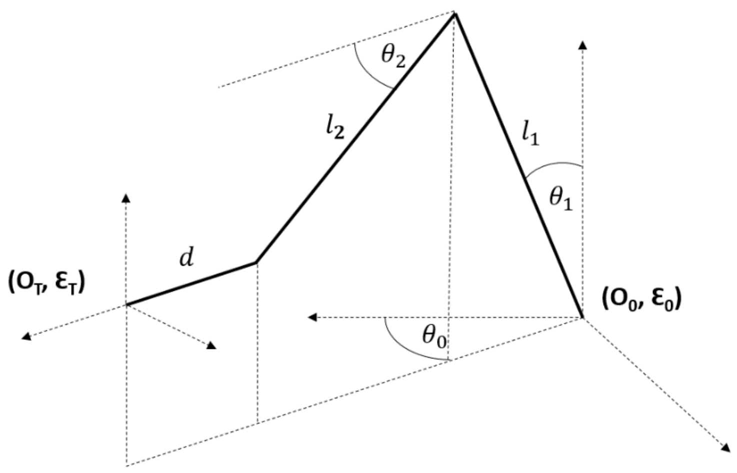

Fig. 3. Dobot kinematics

Our forward kinematics routine calculates the Cartesian coordinates of the end effector using Equations 5-7. Joint angles θ_1 and θ_2 determine the z-axis height and x-y radius of the end effector relative to the origin frame 0. The base angle θ_0 then determines the rotation of the end effector in the x-y plane. The Dobot User Manual [4] specifies the link lengths as $l_1 = 135$mm, $l_2 = 160$mm, $d = 55$mm.

$$x = cos(\theta_0)(l_1 sin(\theta_1) + l_2 cos(\theta_2) + d) \tag{5}$$

$$y = sin(\theta_0)(l_1 sin(\theta_1) + l_2 cos(\theta_2) + d) \tag{6}$$

$$x = l_1 cos(\theta_1) - l_2 sin(\theta_2) \tag{7}$$

Our inverse kinematics routine calculates the joint angles of the Dobot using equations 8-10. Equation set 8 is obtained by applying the Pythagorean Theorem to the Dobot’s geometry with r and ρ defined in Figure 4. The first two equation in set 9 are derived from the law of cosines for the triangle formed by the two links of the Dobot's arm depicted Figure 4. The third equation in set 9 is from trigonometry. With the definitions from equations 8-9 it is possible to determine the Dobot’s joint angles from equation set 10. Because of the Dobot's limited range of motion, the second possible solution for the inverse kinematics (elbow reversed) is unreachable and so does not need to be considered.

$$r = \sqrt{x^2 + y^2} \qquad \rho = \sqrt{(r-d)^2 + z^2} \tag{8}$$

$$\alpha = cos^{-1}\left(\frac{l_1^2+\rho^2-l_2^2}{2l_1\rho}\right) \qquad \gamma = cos^{-1}\left(\frac{l_1^2+l_2^2-\rho^2}{2l_1l_2}\right) \qquad \varphi = tan^{-1}\left(\frac{z}{r-d}\right) \tag{9}$$

$$\theta_0 = tan^{-1}(y/x) \qquad \theta_1 = 90° - \varphi - \alpha \qquad \theta_2 = 180° - \varphi - \alpha - \gamma \tag{10}$$

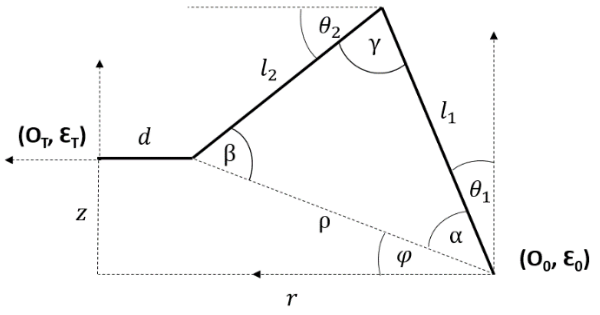

Fig. 4. Law of cosines for inverse kinematics

For development purposes, we include unit tests in the kinematics module, which check the forward kinematics against some known points and then check that the inverse kinematics correctly map the output of the forward kinematics back to the original angles given.

Our kinematics module contains a triangle mesh of the Dobot to be used for collision detection. The mesh consists of four parallelepipeds, which represent bounding boxes for the Dobot's first arm link, second arm link, end effector, and camera attachment. The kinematics module includes a method that returns the meshes after transforming them into the origin frame 0 for a given Dobot pose. Equations 11-13 describe this transformation where $link_1$, $link_2$, and end represent the meshes for the two arm links and the end effector (including the camera). The meshes are rotated independently (besides the base rotation of everything) because of the Dobot’s decoupled architecture. In equations 12 and 13 the meshes must be offset by the preceding link lengths. Figure 5 shows the mesh in its zero configuration.

$$R_z(\theta_0)R_y(\theta_1)link_1 \tag{11}$$

$$R_z(\theta_0)\big((R_y(\theta_1)l_1 + R_y(\theta_2)link_2\big) \tag{12}$$

$$R_z(\theta_0)\big((R_y(\theta_1)l_1 + R_y(\theta_2)l_2 + end\big) \tag{13}$$

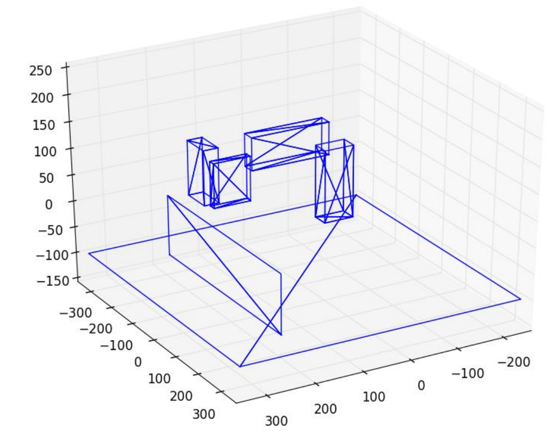
Fig. 5. Dobot and environment mesh models

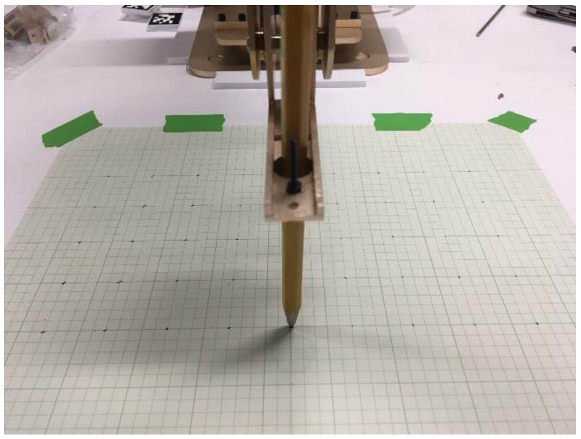
Fig. 6. Arm calibration setup

Model Validation

We validate the kinematics model and parameters by comparing the output of the forward kinematics with a known reference. Figure 6 shows our setup for such a test. By manually controlling the Dobot to move the tip of the pencil to known points on the graph paper, we generate a set of joint angles and known end effector positions (27 in total). Figure 7 compares the pencil positions predicted by the forward kinematics (red) and the positions known from the graph paper (blue). Systematic error can be observed at larger base angles in the left plot. This error exists because the pencil has a slightly different horizontal offset than the suction cup end effector (about 48mm vs 55mm respectively). If we adjust the offset d in the kinematics model, this error is eliminated as shown in the right plot.

We also use a damped least-squares method (MATLAB's *lsqcurvefit* function) to estimate the model parameters: link lengths, angle offsets, and angle scale factors. Allowing the fit to adjust all three link lengths leads to odd results, possibly due to overfitting, so we hold l_1 and l_2 fixed with the values from the Dobot User Manual [4]. The least-squares fit returns $d = 53.42$mm; -0.02°, 1.22°, and -2.48° for the angle offsets; and 1.00, 0.98, and 1.05 for the angle scale factors. The fit reduces the mean distance between the forward kinematic solutions and reference positions from 3.08±1.55mm (s.d.) to 2.20±1.13mm (s.d.).

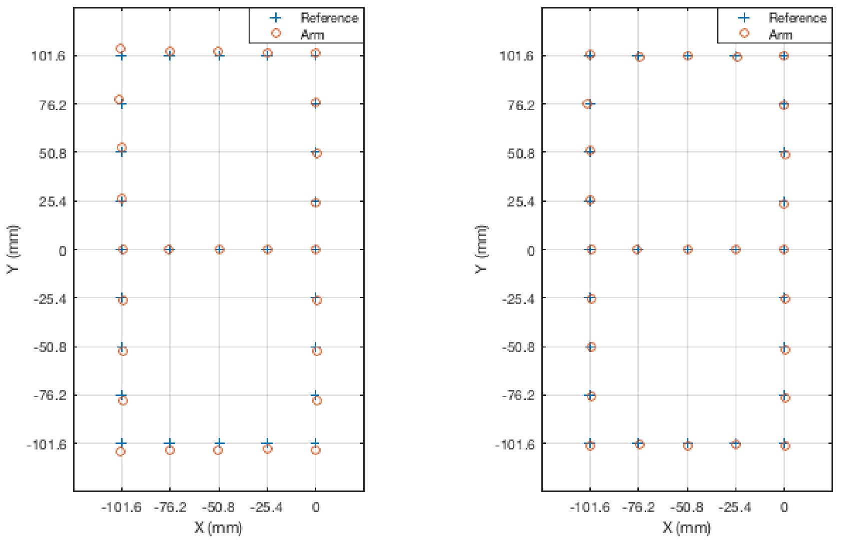

Fig. 7. Arm calibration results with correct (right) and incorrect (left) end effector offsets

We use a second dataset of 39 calibration points to check the repeatability of these results. For the second dataset, the least-squares fit returns $d = 41.477$mm; -0.13°, -0.94°, and -4.35° for the angle offsets; and 1.01, 1.00, and 1.04 for the angle scale factors. The fit reduces the mean distance error from 1.09±0.77mm (s.d.) to 0.55±0.27mm (s.d.). We expect the end effector offset d to differ between experiments because we fix the pencil to slightly different positions each time. However, the discrepancies for the angle offsets and gains suggest that the least-square fit is overfitting noise in the data. This result is not entirely surprising. When we measure the positions of four points on the graph paper twice, we observe distance errors of 0.50, 0.35, 0.61, and 0.25mm between the pairs of samples due to human error in positioning the Dobot's end effector.

Our kinematics module can calculate forward and inverse kinematics for the Dobot. The forward kinematics agree with experimental measurements up to at least 1mm, beyond which we cannot reliably measure whether the kinematics are in error. This functionality is sufficient for basic control of the Dobot, which internally performs its own trajectory planning. However, for more sophisticated path planning, we implement a probabilistic roadmap. To generate collision-free paths, the roadmap relies heavily on our collision detection module, which we will now describe.

Collision Detection

Out collision detection module works by checking for intersections between each triangle in the Dobot's mesh model and each triangle in the obstacle models. Figure 8 illustrates two intersecting triangles and their face normal vectors, which define the planes in which each lie. We use a wrapper class to keep track of obstacle meshes and provide a unified interface for checking collisions and displaying the obstacle and Dobot meshes.

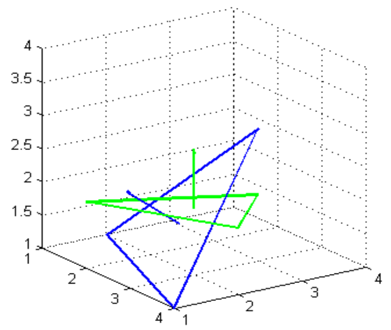

Fig. 8. An example of two intersecting triangles

The intersection tests themselves are performed using an algorithm described by Tomas Moller [5]. Algorithm 1 outlines the main steps in the algorithm. Given two triangles, it returns TRUE if they intersect and FALSE if they do not. Tomas Moller's paper [5] describes each step in detail except for the 2D line-line intersection and vertex-in-triangle tests indicated with asterisks, so only those will be elaborated upon here.

Algorithm 1

```
If all the vertices in either triangle lie to one side of the plane defined by the other
    Return FALSE
If the triangles lie in the same plane
    Project the triangles onto the most closely aligned coordinate plane
    If any of the edges intersect between the two triangles*
        Return TRUE
    If either triangle contains a vertex of the other*
        Return TRUE
    Otherwise
        Return FALSE
Otherwise
    Find the line of intersection of the triangles
    Project the triangles onto the coordinate axis most closely aligned with this line
```

```
**If** the triangles overlap
    Return TRUE
**Otherwise**
    Return FALSE
```

We use a 2-dimensional line-line intersection test from [6] when the triangles lie in the same plane. It attempts to solve for the point of intersection of two line segments to determine if one exists. Its derivation is similar to the sub-problem approach to inverse kinematics discussed in class in that it uses properties of vectors (specifically, that the cross product of a vector and itself is zero) to eliminate one variable and solve for the other, which together define the point of intersection.

We check whether a triangle contains a point (in 2 dimensions) by expressing the point's position as a weighed sum of two edges of the triangle. If the triangle is defined by vertices $\boldsymbol{t}_1, \boldsymbol{t}_2, \boldsymbol{t}_3$ and $\boldsymbol{p}$ indicates the point, then equation 14 captures this idea. The matrix M formed by the two triangle edges can be inverted to determine the weights w_1 and w_2. If the weights are greater than zero and have a sum less than one, then the point is contained within the triangle.

$$(\boldsymbol{p} - \boldsymbol{t}_1) = [(\boldsymbol{t}_2 - \boldsymbol{t}_1) \quad (\boldsymbol{t}_3 - \boldsymbol{t}_1)] \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = M\boldsymbol{w} \tag{14}$$

The matrix inversion approach in the above step may not be the most efficient solution. However, it highlights an important property of the overall intersection test algorithm: it makes the common case fast. In the Dobot and obstacle scenario, many triangles will fall completely to one side or the other of each other's planes and thus be eliminated quickly. Even when run in Python on a 2.2GHz Intel Core i7, the algorithm performs about 17,000 intersection tests per second when checking for collisions between the Dobot mesh and meshes for a table and a wall. Specifically, in 15 runs of 100 configurations with 144 triangle comparisons each, it performs an average of 17,220 intersection tests per second with a standard deviation of 1,129.

Path Planning

With collision detection in hand, we use a probabilistic roadmap to perform path planning with obstacle avoidance. Our code generates the roadmap as follow. First, it randomly samples the configuration space of the Dobot (joint angles) until it has found some specified number of configurations that are valid and collision free and added them to a graph. We generally use 50-150 for the number of desired configurations. We use the NetworkX library to represent the graph [7].

Second, the code generates a KD-Tree using the end effector positions corresponding to each of the configurations. We use the KD-Tree implementation provided by SciPy [8]. A KD-Tree is a binary tree used to divide up points based to their locations. It works by subdividing the hyperspace containing the points along a coordinate axis at each node in the tree. The structure of the KD-Tree is helpful when searching for nearby points because it may be possible to quickly eliminate points in many branches of the tree from consideration. This characteristic allows the KD-Tree to speed up the next step in generating the probabilistic roadmap.

Third, the path planning code finds the nearest five neighbors of each node using the KD-Tree and attempts to connect them with a path. Specifically, it evenly samples the configuration space between the points for collisions. If none are detected, it adds an edge to the graph containing the configurations. The left plot of Figure 9 shows the nodes and edges of a probabilistic roadmap generated with this procedure. The configuration nodes have been mapped to the corresponding end effector positions for visualization purposes.

The roadmap generation is slow, taking up to minutes. However, once it is generated for a fixed environment, it is possible to relatively quickly plan paths between points in the Dobot's configurations space. Given a set of start and end angles, the planning code adds them to the graph of configurations and attempts to connect them to their 5 nearest neighbors determined from the KD-Tree. It then searches for a path between the start and end nodes using Dijkstra's algorithm as implemented in the NetworkX library [7]. Dijkstra's algorithm finds the shortest path between two nodes in a graph if one exists. It is similar to a breadth first search except that it can handle edges with different weights, which is the case here since not all configuration nodes are equidistant. Once the path search is complete, the planning code removes the start and end nodes from the NetworkX graph and returns either the series of configurations representing the path or an empty list if none exists. The right plot of Figure 9 shows a path generated with this procedure (red) along with nodes from the probabilistic roadmap (green).

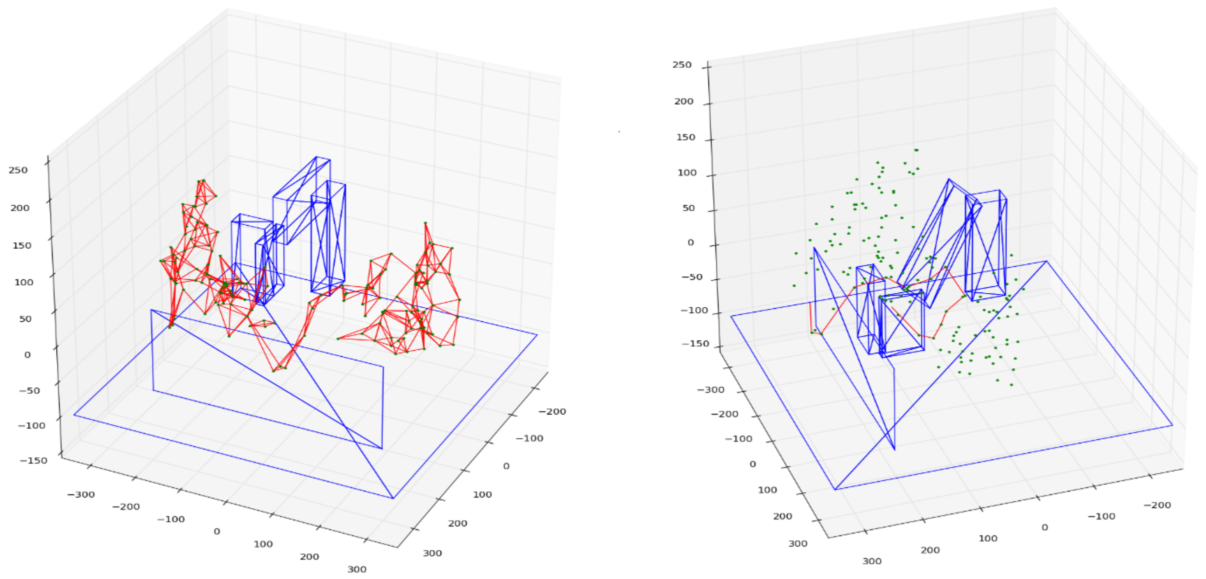

Fig. 9. Probabilistic roadmap (left) and an example path through it (right)

Subsystem: Vision

The Dobot determines the pose of objects by detecting AR tags with the Logitech webcam attached to its end effector. Like the Dobot's end effector, the webcam is mounted to be downward facing and is oriented such that its rotation with respect to the Dobot's frame (R_{0C}) during start-up in the zero-configuration is given by:

$$R_{0C} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

In this configuration, it is easy to interchange between perspectives because the transpose of R_{0C} is R_{0C} itself and therefore R_{0C} = R_{C0}. Additionally, since the Dobot is mechanically configured to keep its end effector level during movement, the rotation of the end effector with respect to the camera frame (R_{CT}) can easily be calculated by multiplying two rotations about the

Dobot's z-axis with R_{co}. These z-axis rotations correspond to the rotation of the Dobot's base joint and the Dobot's end effector joint as these joints are limited to rotation only about the Dobot's z-axis.

Camera Calibration

The Logitech C270 camera is calibrated using the Caltech Camera Calibration Toolbox for Matlab[9]. This toolbox outlines a procedure to determine the intrinsic and extrinsic characteristics of a camera using a pinhole camera model. The Logitech camera is rigidly mounted in order to collect a number of pictures of a checkerboard pattern which are used to solve the Perspective-n-Point (PnP) problem. The checkerboard pattern chosen is black and white colored with 25.4mm x 25.4mm squares and is displayed on a laptop screen in order to minimize distortion caused by a warped image surface. The orientation of the laptop with respect to the camera is varied for each picture in order to create a dataset of 12 unique images of the checkerboard display. Following the procedures of the Caltech toolbox, the corners of each checkerboard image are extracted starting from the top left corner and moving clockwise. Using the actual size of each square in the checkerboard pattern (25.4mm), the toolbox is able to extract the corners of each square in the checkerboard pattern and construct a relative coordinate frame. The toolbox allows for tuning of distortion parameters in order to line up the initial corner calibration if it is initially a little off. Using the relative coordinate frame, the Caltech toolbox calculates camera parameters in two steps. Camera parameters are initially calculated using a closed form solution and then a non-linear optimization scheme is used to calculate optimized parameters. The resulting extrinsic parameters are used to develop a 3D model of the checkerboard pose with respect to the camera which can be seen in Figure 10. The resulting intrinsic parameter matrix (*K*) and distortion coefficients (*dist*) for the Logitech C270 camera are:

$$K = \begin{bmatrix} 810.06 & 0 & 325.40 \\ 0 & 810.76 & 249.02 \\ 0 & 0 & 1 \end{bmatrix}$$

$$dist = [0.01584 \quad 0.37926 \quad -0.00056 \quad 0.00331 \quad 0]$$

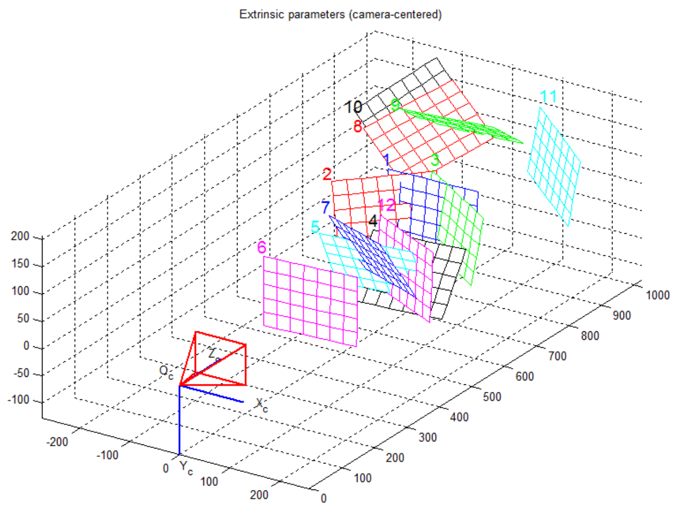

Fig. 10. Extrinsic camera parameters for the Logitech C270

Marker Detection

After the camera has been calibrated, image processing is performed using OpenCV 3.0. OpenCV has a large amount of documentation and is nicely paired with Python's ability to handle linear algebra. The OpenCV library provides a large repository of helper functions that perform computer vision tasks such as edge detection, shape detection, color thresholding and many more features. Additionally, OpenCV allows for pixel-level manipulation of images as numpy arrays, so that custom image processing functions can be developed. Integration with a USB camera is trivial in OpenCV, as camera devices are automatically indexed and are initialized by passing their index as an argument to OpenCV image capture or video capture functions. Recording video is therefore as easy as initializing the camera device to video capture, writing a Python loop and grabbing a frame during each run through the loop. Additional flexibility may be offered by OpenCV functions through the direct control of a USB cameras frame rate and image capture size (if the camera device supports changing these properties). The Logitech C270 allows for the frame rate to be manipulated, therefore an additional degree of freedom is granted when optimizing code performance.

We use an additional set of python modules on top of OpenCV, called *Hampy*[10], in order to detect hamming marker AR tags. The Hampy modules provide a set of functions for generating square hamming markers and detecting if any of these markers are present in a mostly clear image. The marker generation script takes arguments for marker size (in pixels) and marker ID (any number between 0 and 2^28) and generates a png image of the marker. Markers are found using Hampy's *detect_marker* function, which takes an image as an argument and returns a list of *HammingMarker* objects corresponding to any markers found in an image. Each HammingMarker object contains the detected marker ID, and approximate contour and size of the marker. The HammingMarker object also includes a *draw_contour* method which uses the detected marker information to augment the markers contour and ID onto the original image. A screen shot of this augmentation can be seen in Figure 11. Markers for this project are generated to be 25 x 25 millimeters in size.

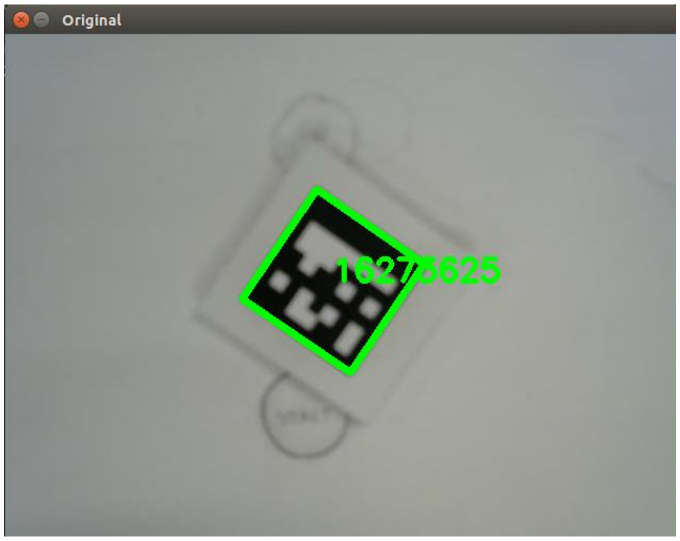

Fig. 11. Detection of a hamming marker using *Hampy* in OpenCV.

Robust marker detection is achieved by considering marker detection rate with respect to the camera frame rate. If the camera is configured to capture video at 20fps, a perfect marker detection rate would be 20 markers/s. Analysis of the marker detection rate on images from the C270 webcam is done using a Python script that takes a set number of video frames and counts how many times a specific marker is detected throughout all the frames. The marker detection rate (MDR) is therefore expressed by:

$$MDR = \frac{\#\,Markers\ Detected}{\#\,Frames\ Taken} * Camera\ Frame\ Rate \qquad (15)$$

Eleven trials of recording 200 frames of raw video at 20fps with the C270 results in an average MDR of 18.91 markers/s for a stationary marker and an average MDR of 9.79 markers/s for a moving marker with moving shadows cast across the marker as the video is taken. Pre-processing of the video frames by gray-scaling the pixels, thresholding them to either full black/white with a decision boundary at 40% pixel strength and applying a slight Gaussian blur yields an average MDR of 19.99 markers/s (5.7% increase) for a stationary marker and an average MDR of 14.10 markers/s (44% increase) for a moving marker with moving shadows cast across the marker as the video is taken. Justification for the choice of 40% for the threshold percentage can be seen in Figure 12, where marker detection rate peaks around the 30%-40% pixel strength threshold range. An upper value of 40% is chosen due to the bright lighting conditions of the Dobot workspace.

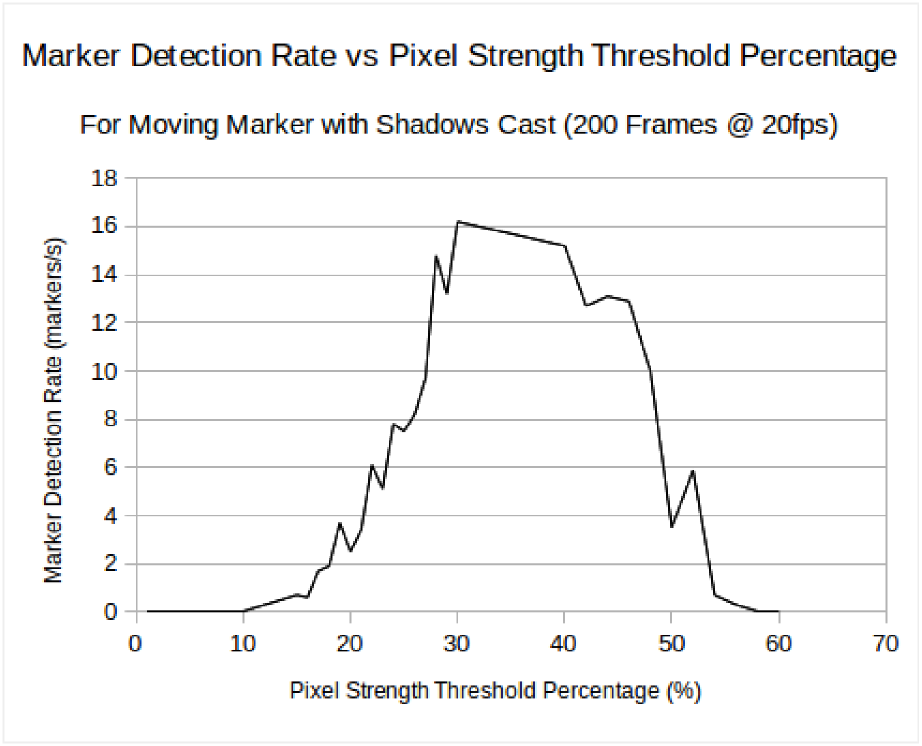

Fig. 12. Marker detection rate as pixel strength threshold is varied.

Marker Location

Using information from detected markers, we can perform a similar task that the Caltech calibration uses, but in reverse, in order to determine the location of a detected marker with respect to the camera frame. OpenCV provides a *solvePnP* function that calculates the pose of an object from its 3D to 2D point correspondences and a given cameras intrinsic parameters and distortion coefficients. By providing this function with the C270's intrinsic parameters and the contours of a detected marker (pixel locations of the four corners) and by defining the detected marker frame to be at the center of the marker and have units of millimeters, the *solvePnP* function returns the pose of the marker with respect to the camera (*rvec*, *tvec*) in mm. *tvec* is a vector from the camera to the center of the detected marker (P_{CA}) and *rvec* is a rotation vector in Rodrigues form, where the direction of the vector is the axis of rotation and the magnitude of the vector is the magnitude of the rotation in radians. A simple call to OpenCV's *Rodrigues* function converts *rvec* to the rotation matrix R_{CA}.

We would like to determine the pose of a marker with respect to the Dobot frame (P_{0A}, R_{0A}) in mm, given the position of the marker with respect to the camera frame (P_{CA}, R_{CA}) in mm. This is done by considering the diagram in Figure 13 that shows the vectors that connect various points of interest in the Dobot's workspace. The subscript labels *C*, *T*, *A*, and *0*, correspond to the camera frame, end effector frame, marker frame and the Dobot's frame respectively. From the diagram, we can see through vector addition that,

$$P_{0T} - P_{CT} + P_{CA} - P_{0A} = 0 \tag{16}$$

Since the end effector and the camera are attached to the same body and only a rotation about the Dobot's z-axis can occur to the end effector, P_{CT} will be a constant in this system. Manipulating (16) to solve for P_{CT} yields:

$$P_{CT} = P_{CA} + R_{CT}R_{T0}(P_{0T} - P_{0A}) \tag{17}$$

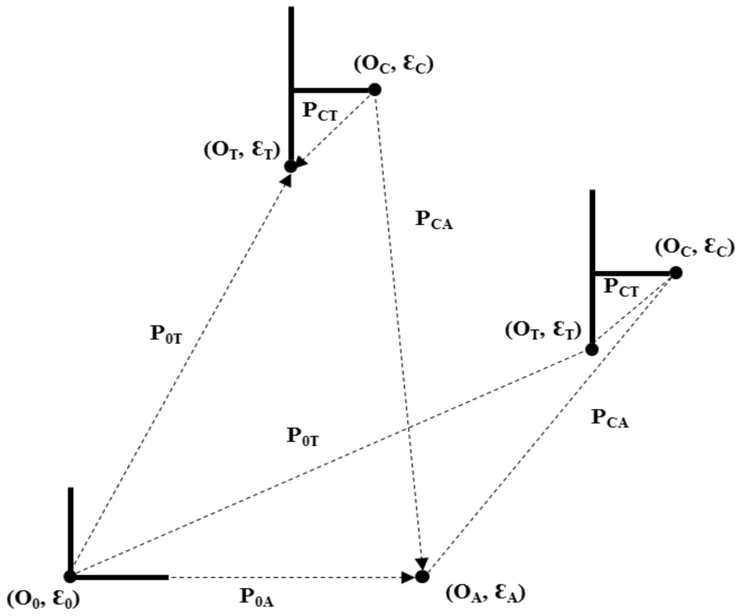

Fig. 13. Points of interest and their dependencies in the Dobot workspace.

Where R_{C0} is now decomposed into R_{CT}, the rotation of the end effector in zero configuration with respect to the camera frame and R_{T0}, the inverse of the current rotation of the end effector with respect to the Dobot's frame (R_{0T}). P_{0T} is provided by solving the forward kinematics of the Dobot's joint angle readings and P_{CA} is provided by solving the PnP problem in OpenCV with the detected marker. In order to accurately measure P_{0A}, the Dobot arm is manually moved to touch the center of the marker at position *A* such that $P_{0T} = P_{0A}$. Solving the forward kinematics again yields a measurement of the position of the Dobot end effector and therefore an accurate measurement of the markers position in the Dobot's frame (P_{0A}). This procedure is performed multiple times to calculate an average value for P_{CT} in mm:

$$P_{CT} = \begin{bmatrix} -10.48 \\ -21.58 \\ 28.42 \end{bmatrix}$$

The Dobot needs to move its end effector along the vector $(P_{TA})_0$, (P_{TA} in the Dobot frame), in order to touch its end effector to the center of the marker in Figure 13. $(P_{TA})_0$ can be solved in a similar manner that P_{CT} is solved for in (17) using vector addition to yield:

$$(P_{TA})_0 = R_{0T}R_{TC}(P_{CA} - P_{CT}) \tag{18}$$

The kinematics system is able to determine the position of a marker in the Dobot frame (P_{0A}), by adding $(P_{TA})_0$ to the current position of its end effector (P_{0T}). To touch or pick up a marker, the Dobot simply moves to the location (P_{0A}). This procedure is repeated for all markers detected within a frame of video in order to determine the location of all markers with respect to the Dobot frame. The Dobot is therefore able to detect, locate and interact with any object in its workspace that is labeled with a marker.

Subsystem: Behavior

In order to perform high-level tasks, well-structured code is a necessity. Figure 14 shows the layout of the behavior and camera modules. The behavior module initializes 2 threads, one for the camera feed and another for sending serial commands. The camera processes images and keeps track of object poses. The camera module can be configured to either capture data continuously or wait until an object pose is requested. The latter approach is ideal when the Dobot has limited computing resources and image processing proves to be too costly.

The behavior module requests user input and verifies that the command exists and is possible at the time of execution. Some commands have no pre-requisites while others are very restrictive. When the pre-requisites are met, the task initiates along with a thread listening for user input. The input is treated as an interrupt signal to allow breaking out of the current task.

From the available tasks that can be performed, some are more complex and build off of smaller tasks. For example, loading duckies onto a Duckybot requires searching for objects, cleaning unknown tags, placing a ducky, and avoiding obstacles. By examining these complex behaviors, a significant portion of the underlying functionality can be uncovered. The following will explore these behaviors in-depth. Note that all tasks assume the following pre-requisites are met: serial communication with the Dobot, camera initialization, and tag registration.

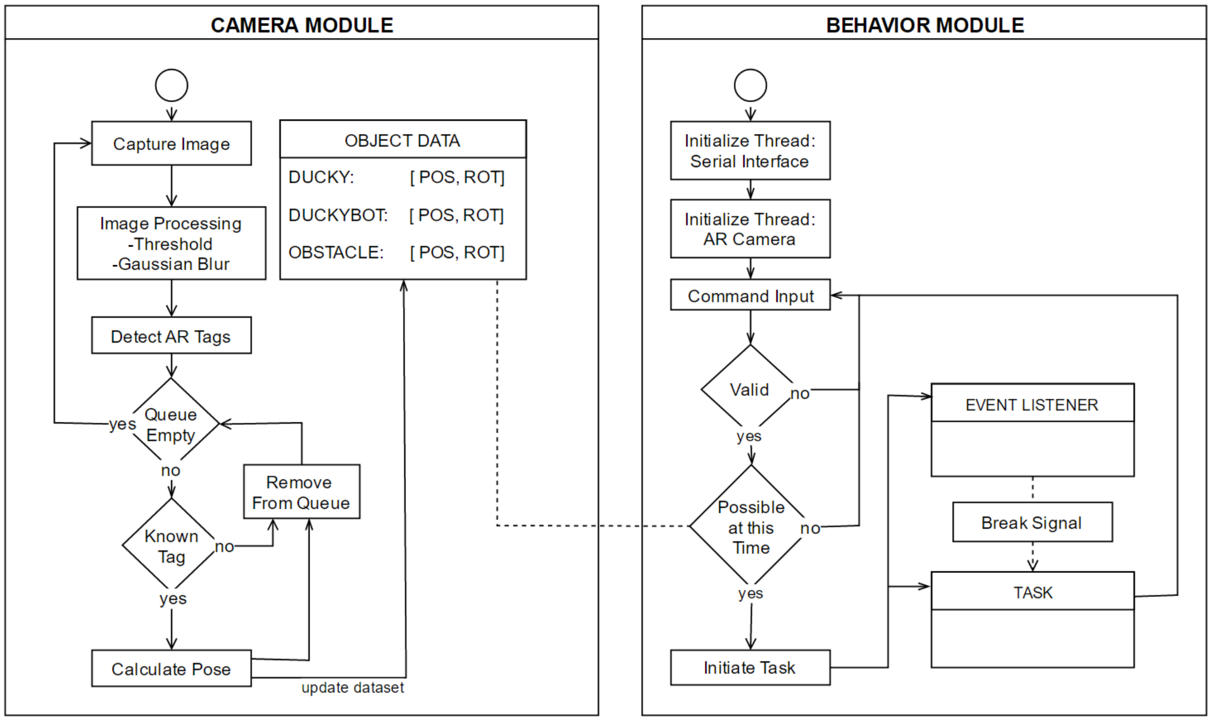

Fig. 14. Logic flowchart of the camera and behavior modules.

Searching (see Appendix B for documented source code) – The Dobot will attempt to find a specified AR tag in the workspace. The user must select an object to find from one of the registered tags in the camera module. Once activated, the Dobot begins rotating around the workspace. Each movement consists of a 5 degree change to the base angle. The other joint angles remain static throughout this process. The angle change is pushed through the serial interface which is configured to be a blocking call, so code execution will only resume once the Dobot has completed the requested movement. This avoids timing issues and provides responsive movements. If the rotation exceeds 100 degrees in a given direction from the zero configuration, the direction is reversed. This gives a total search range of 200 degrees, which is less than the full 270 degree range available to the Dobot. The 70 degree buffer provides some protection against an improper zero configuration. During each rotation, the program queries the camera module for the requested object. If found, the search behavior terminates.

Tracking (see Appendix C + D for documented source code) – The Dobot will attempt to follow a specified AR tag in the workspace. The Dobot will enter search mode if the tag is not found or is lost during the tracking process. Since it is possible for tags to become undetectable due to brief lighting changes, the tag is only considered unavailable if undetected for 30 consecutive queries. Once detected, P_{CA} in Figure 13 can be calculated. The desired value for P_{CA} is the following:

$$P_{CA\ desired} = \begin{bmatrix} 0 \\ 0 \\ 5 * tag\ size \end{bmatrix}$$

The desired X and Y offsets are 0 since P_{CA} is from the center of the camera to the center of the tag. The desired Z offset is dependent on tag size to ensure the camera hovers high enough so that the tag remains in view. Corrective action will only take place when the magnitude of P_{CA} - $P_{CA\ desired}$ exceeds 1mm. This is implemented in order to ignore fluctuations in the camera readings due to error. In addition to this filter, the correction only carries a 20% weight against the current position. This prevents a faulty reading from disrupting tracking by a significant degree. The only instance in which tracking is stopped is when an interrupt signal is received from the end-user. The overall performance of this behavior is heavily reliant on how fast a tag moves and whether it stays in the video frame. This issue is amplified by the fact that the field of view from the camera is very limited. However, if the tag moves slowly and remains in view, the tracking behavior is reliable. This is because the detection failure rate is rare, as shown in figure 12, and tracking only fails if object detection fails 30 times consecutively.

Loading a ducky – The Dobot will attempt to place a ducky onto a Duckybot. The user must specify whether to enable trash cleanup or obstacle avoidance. This behavior requires the Dobot to interact with objects in the workspace, which is accomplished through using the pump. By activating the pump while the end effector is touching an object, it becomes attached. Assuming the object weighs less than 500g, the max payload for the Dobot, the arm can move to a new position and release the object.

Once the loading task begins, the Dobot searches for the Duckybot. If trash cleanup is enabled, any unknown tags will be placed into a designated garbage bin. Trash detection is performed during each 5 degree increment in the search behavior. When detected, the Dobot picks up the object and releases it over the garbage bin. The arm will return to its previous position and continue the searching process. When a Duckybot is detected, the arm will pick up a ducky from a designated position and release it onto the Duckybot.

When obstacle avoidance is enabled, the path planning module is used (see Appendix A + E for documented source code). The module finds a free path from the starting position to the desired position. This path is represented as a list of positions that must be executed consecutively. If a path is not found, the program terminates in order to protect the Dobot from performing actions that could lead to a collision.

The reliability of the loading behavior is dependent on the accuracy of the static positions for the ducky and camera offset. If the camera offset is changed due to a collision, the release position for the ducky will be incorrect. This issue is mitigated by performing a series of touch tests, where instructions are sent to the Dobot to touch an AR tag. If the position is incorrect, the camera offset can be modified either in the program or physically. A similar test can be performed with the ducky position by sending commands to grab the ducky and reset repeatedly. This form of manual calibration must be performed frequently to ensure reliability, especially after having multiple interactions with the Dobot.

Conclusion

We demonstrate the ability to detect Duckybots using a camera and load duckies onto them, satisfying our primary objective. This report describes the methods employed toward this goal along with the results. We validate our kinematics model to approximately 1mm, measure AR tag detection rates of 14.1 markers/s at 20fps, and demonstrate repeatable task performance. Our module architecture and the high-level organization of our behavioral code has allowed for quick development of new capabilities, such as debris removal and object stacking. We hope it may also accelerate the progress of future users.

In order to further improve object detection, future developments of the project could include a better camera, perhaps with autofocus capabilities to sharpen AR tag images at close range. As this change would come with higher resolution images, the code would have to be re-optimized due to the higher processing overhead these images would require. Additionally, a calibration interpolation technique would need to be employed in order to adaptively change camera intrinsic parameters as focus is adjusted automatically. These developments would greatly improve marker detected at a variety of camera heights.

Our project would also benefit from faster path planning. Collision detection is a naturally parallel process and could be accelerated significantly with parallelization. Graphics cards often work with triangle primitives, so the triangle-triangle intersection algorithm would lend itself well to a GPU implementation. It would also be interesting to investigate incremental collision detection algorithms.

Lastly, low-level control of the Dobot would open up a variety of possibilities. For example, low-level control would make smooth path following possible. In the current setup, the Dobot does not support path following. The arm accepts one set of target joint angles at a time and generates a trajectory to reach them itself.

In this project, we have developed pick-and-place capabilities for the Dobot arm, demonstrated them in experiment, and identified directions for future work. In the process, we have also discovered limitations of the Dobot's current firmware and would investigate improvements to these areas in the future.

References

[1] Dobot specifications. Digital image. Dobot. Web. 13 Dec. 2016. <http://dobot.cc/dobot-armv1/specification.html>

[2] Nikolas Engelhard, pyDobot. GitHub, Inc. Web. 13 Dec. 2016. <https://github.com/NikolasE/pyDobot>

[3] Dobot Communication Protocol. Web. 13 Dec. 2016. <http://dobot.cc/upload/ue_upload/files/2016-05-23/Dobot%20Communication%20Protocol.pdf>

[4] Dobot User Manual V1.1. Web. 13 Dec. 2016. <http://dobot.cc/upload/ue_upload/files/2016-05-20/Dobot%20User%20Manual_201600314_Final.pdf>

[5] Tomas Moller, A Fast Triangle-Triangle Intersection Test. Journal of Graphics Tools, 2(2): 1997. <http://web.stanford.edu/class/cs277/resources/papers/Moller1997b.pdf>

[6] Gareth Rees. Answer posted 19 Feb. 2009. Web. 13 Dec. 2016. <http://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect>

[7] NetworkX. Web. 13 Dec. 2016. <https://networkx.github.io>

[8] KDTree. SciPy Reference. Web. 13 Dec. 2016. <https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html>

[9] Bouget Jean-Yves, Camera Calibration Toolbox for Matlab, (2015). Web. 13 Dec. 2016. <https://www.vision.caltech.edu/bouguetj/calib_doc/>

[10] Rouanet Charles, Simple Hamming Marker Detection using OpenCV, (2014), GitHub repository, <https://github.com/pierre-rouanet/hampy>

Appendix

A Moving end effector to desired position

```
# FUNCTION: move_xyz - Move the end effector to a desired XYZ position
def move_xyz(interface, target, pump_on = False, joint_4_angle = 0, path_planning = False)
:
    # Joint angles needed to reach target XYZ position
    angles = DobotModel.inverse_kinematics(target)
    # No solution was found
    if any(np.isnan(angles)):
        print "Error: No solution for coordinates: ", target
    # Solution found and path planning is disabled
    elif not path_planning:
    # Send serial command to change the joint angles
        interface.send_absolute_angles(float(angles[0]),float(angles[1]),float(angles[2]),
joint_4_angle, interface.MOVE_MODE_JOINTS, pump_on)
    # Solution found and path planning is enabled
    else:
    # Find a feasible path to the desired target
        start = interface.current_status.angles[0:3]
        path = PRM.get_path(start, angles)
    # No path found, future actions may cause collision! Terminate Immediately
    if path == [] or path == None:
        print "ERROR: No Path Found! Aborting..."
        sys.exit()
    # Path found. Iterate through the list of positions
        for p in path:
            interface.send_absolute_angles(float(p[0]), float(p[1]), float(p[2]), joint_4_
angle, interface.MOVE_MODE_JOINTS, pump_on)
```

B Searching Behavior

```
# FUNCTION: Search - Rotate base by increments of 5 degrees until the desired tag is found
# AR TAGS: DUCKY = 0  DUCKYBOT = 1   OBSTACLE = 2
# If tag_index is left undefined , search will return upon finding any registered tag.
# Cleanup mode places unknown tags in the designated trash bin.
def search(interface, camera, tag_index = -
1, clean_mode = False, path_planning = False):

    # Current base angle is the starting point
    base_angle = interface.current_status.get_base_angle()
    direction = 1

    # EVENT LISTENER
    req_exit = []
    listener = threading.Thread(target=input_thread, args=(req_exit,))
    listener.start()

    # While there is no break signal, continue search
    while not req_exit:
    # Positive angle change direction
        if direction == 1:
            while base_angle < 100 :
        # BREAK SIGNAL DETECTED
```

```
                if req_exit:
                    return None

                # Get object pose data
                data = camera.get_all_poses()

                # If cleanup mode is active and garbage is detected, throw it away
                if clean_mode and data[2] != [None, None]:
            # Position Before Cleaning
            initial_pos = DobotModel.forward_kinematics([base_angle,10,10,0])
            # Pick up the AR tag (garbage tag)
                    garbage_xyz = get_xyz(interface, data[2][0])
                    touch(interface, 2, 1, True)

            # Go to max height above AR tag
                    tmp = np.zeros((3, 1))
                    tmp[:, :] = garbage_xyz[:, :]
                    tmp[2, 0] = MAXHEIGHT
                    move_xyz(interface, tmp, True)

                    # Go to garbage can
                    move_xyz(interface, GARBAGECAN_POS, True, 0, path_planning)
            # Release Garbage
                    move_xyz(interface, GARBAGECAN_POS, False)
            # Return to last position during searching process
            move_xyz(interface, initial_pos, False, 0, path_planning)
                    continue

                # Search mode: Find any tag
                if tag_index == -1:
                    for x in range(0,3):
                        if data[x] != [None, None]:
                            return data[x]
         # Search mode: Find specific tag
                else:
                    if data[tag_index] != [None, None]:
                        return data[tag_index]

        # Rotate base angle by 5 degrees
                base_angle = base_angle + 5
                interface.send_absolute_angles(base_angle, 10, 10, 0)

    # Negative angle change direction
        else:
            while base_angle > -100 :
        # BREAK SIGNAL DETECTED
                if req_exit:
                    return None

                # Get object pose data
                data = camera.get_all_poses()

                # If cleanup mode is active and garbage is detected, throw it away
                if clean_mode and data[2] != [None, None]:
            # Position Before Cleaning
            initial_pos = DobotModel.forward_kinematics([base_angle,10,10,0])
            # Pick up the AR tag (garbage tag)
                    garbage_xyz = get_xyz(interface, data[2][0])
                    touch(interface, 2, 1, True)

            # Go to max height above AR tag
```

```python
                    tmp = np.zeros((3, 1))
                    tmp[:, :] = garbage_xyz[:, :]
                    tmp[2, 0] = MAXHEIGHT
                    move_xyz(interface, tmp, True)

            # Go to garbage can
            move_xyz(interface, GARBAGECAN_POS, True, 0, path_planning)
            # Release Garbage
            move_xyz(interface, GARBAGECAN_POS, False)
            # Return to last position during searching process
            move_xyz(interface, initial_pos, False, 0, path_planning)
            continue

        # Search mode: Find any tag
        if tag_index == -1:
            for x in range(0,3):
            if data[x] != [None, None]:
                    return data[x]
             # Search mode: Find specific tag
            else:
                if data[tag_index] != [None, None]:
                return data[tag_index]

            # Rotate base angle by 5 degrees
            base_angle = base_angle + 5
            interface.send_absolute_angles(base_angle, 10, 10, 0)

            # change direction
            direction = direction * -1

        # user requested to exit
        interface.send_absolute_angles(0, 10, 10, 0)
        return None
```

C Tracking Behavior (Low-Pass Filter Disabled)

```python
# FUNCTION: Track - Follow an AR tag by hovering the camera above it.
# AR TAGS: DUCKY = 0  DUCKYBOT = 1   OBSTACLE = 2
def track(interface, camera, tag_index):
    # EVENT LISTENER
    req_exit = []
    listener = threading.Thread(target=input_thread, args=(req_exit,))
    listener.start()

    while not req_exit:
        # Search if the camera module fails to find the tag for 30 consecutive frames
        searching = True
        for x in range(0,30):
            data = camera.get_all_poses()[tag_index]
            if data != [None, None]:
                searching = False
                break;

        if searching:
            # Search until the desired tag is found
            data = search(interface, camera, tag_index)

        # SEARCH TERMINATED DUE TO BREAK SIGNAL
        if data == None:
            return

    # Follow tag while it is in view
    while data != [None, None] and not req_exit:

        # === Getting Desired XYZ of end effector ===
        # Hover above the tag (Z offest 5 * marker size)
        Pct = np.array([[0], [0], [5 * REGISTERED_TAGS[tag_index][1]]])

        # Calculate difference in desired position and actual position
        Roc = np.array([[0, 1, 0], [1, 0, 0], [0, 0, -1]])
        Pta = np.matmul(Roc, data[0]) - np.matmul(Roc, Pct)

        # Only move if the change is significant (1mm)
        if np.linalg.norm(Pta) > 1:
        # Current XYZ position
        p0t = DobotModel.forward_kinematics(interface.current_status.get_angles())
        # Desired XYZ position
        target = np.reshape(Pta, (3, 1)) + np.reshape(p0t, (3, 1))
        move_xyz(interface,target)

        # Get object pose data
        data = camera.get_all_poses()[tag_index]
```

D Tracking Behavior (Low-Pass Filter Enabled)

```
# FUNCTION: Track - Follow an AR tag by hovering the camera above it.
# AR TAGS: DUCKY = 0  DUCKYBOT = 1  OBSTACLE = 2
def track(interface, camera, tag_index):
    # EVENT LISTENER
    req_exit = []
    listener = threading.Thread(target=input_thread, args=(req_exit,))
    listener.start()

    alpha = 0.2 # weight of new measurements
    P0a_est = None # estimate of AR tag position

    while not req_exit:
          # From kinematics
          angles = interface.current_status.angles[0:3]
          P0t = np.reshape(DobotModel.forward_kinematics(angles), (3,1))
          R0t = DobotModel.R0T(angles)

          # From calibration
          Pct = np.array(CAMERA_OFFSET)
          Rtc = np.array([[0, 1, 0], [1, 0, 0], [0, 0, -1]])
          R0c = np.matmul(R0t,Rtc)

          # From camera
          Pca = np.reshape(data[0], (3,1))

          # Hover above the tag (Z offest 5 * marker size)
          Pca_des = np.array([[0], [0], [5 * REGISTERED_TAGS[tag_index][1]]])
          # correction = -1*np.matmul(R0c, Pca_des - Pca)
          P0a_des = P0t + np.matmul(R0c, Pca_des - Pct)

          # Smoothed estimate
          if P0a_est is None:
                 P0a_est = P0a_des # initialize so no movement required
```

```
            P0a =  P0t + np.matmul(R0c, Pca - Pct) # measured
            P0a_est = alpha*P0a + (1 - alpha)*P0a_est # update estimate

            # If the change in desired XYZ is notable, move to track it
            correction = P0a_des - P0a_est
            if np.linalg.norm(correction) > 1.0:
                    angles = move_xyz(interface, P0t - correction)

            # Get object pose data
            data = camera.get_all_poses()[tag_index]
```

E Path Planning – Returning a path

```
"""
Searches for a path in the PRM between configurations q0 and qf.
Returns a list of configuration tuples describing the path or []
if no path is found.
"""
def get_path(self,q0,qf):
    q0 = np.reshape(np.array(q0),3)
    qf = np.reshape(np.array(qf),3)
    if all(q0 == qf):
        return [qf]

    n0 = len(self.G.node)
    nf = n0 + 1

    # Add the start and end configs to G, so we can just search it
    self.G.add_node(n0,cfg=q0)
    self.G.add_node(nf,cfg=qf)
    for k in [n0,nf]:
        self._connect(k,DobotModel.forward_kinematics(self.G.node[k]['cfg']))

    if not nx.has_path(self.G,n0,nf):
        path = [] # could not find a path
    else:
        nodes = nx.dijkstra_path(self.G,n0,nf,'weight')
        path = [self.G.node[k]['cfg'] for k in nodes]

    # Remove the start and end configs so G remains consistent with tree
    self.G.remove_node(n0)
    self.G.remove_node(nf)

    return path
```