Automated Robotic Can Retrieval

Sam Burkhart,Dakota Ward,Khyati Sinha
ECE 544
Spring 2017

Design Report

Introduction:

The Automated Robotic Can Retrieval is a robot control designed on Digilent's new product, the Pynq-Z1 board. The robot utilizes OpenCV computer vision techniques for object detection ,distance measurement, color detection . Our goal was to design a robot which can identify a can of a given color(red in our case) and take it and drop it off to a target location.

Hardware Design:

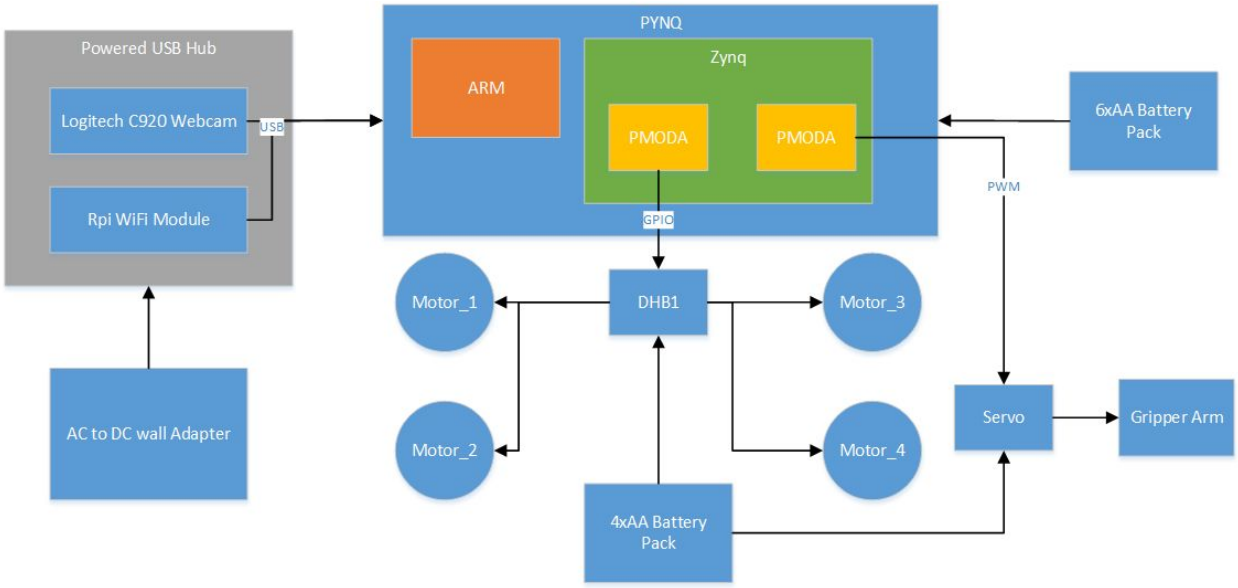
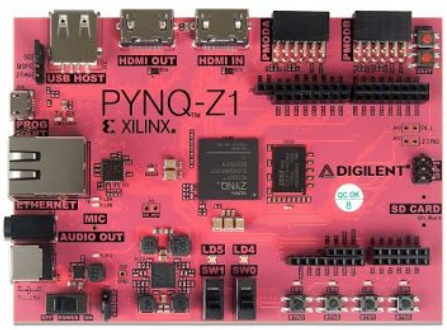
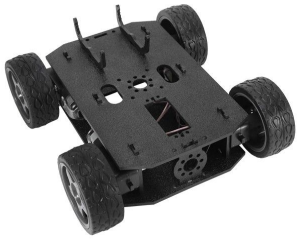


Figure 1: *Hardware module diagram*

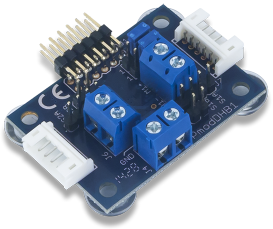


The core development platform used for this project is the Digilent/Xilinx PYNQ board. The board contains a Xilinx Zynq FPGA fabric connected to a Dual-Core ARM Cortex A9 processor. The ARM processor contains a full linux kernel with a Python stack allowing the use of modern programming paradigms such as Jupyter Notebooks and interactive code development. The board contains 1.3M FPGA gates, 512MB DDR3 memory, micro SD card slot (for OS disk), Ethernet, USB, an Arduino header, and two DigilentPMOD connectors plus

several other features not used in this project. A dedicated 6xAA battery pack was used to power the PYNQ board using a DC barrel connector input. This allowed the USB port to be used for connection to the linux terminal without worrying about the power supply.



The Actobotics Junior Runt Rover was used as the base robot platform, which includes 4 gear motors with wheels. The 4 wheels are connected to a Dual H-Bridge PMOD (DHB1), where the left wheels are connected to the first motor outputs, and the right wheels are connected to the second motor outputs. By driving two wheels from a single H-Bridge we reduced the number of ports we use (only 1 PMOD for the DHB1) and didn't loose any functionality as there was no clear use-case for individual control of each motor. The front gripper claw is controlled via a standard sized servo connected to the second PMOD port, and mounted using an Actobotics 45 degree dual angle pattern bracket to elevate the gripper arm. The 4 motors are powered through the DHB1 VM/GND terminal with 6xAA batteries providing ~9V. This same battery pack powered the servo for the gripper arm.



Since the PYNQ board doesn't have on-board WiFi or Bluetooth, a USB WiFi dongle was used to connect to the board for control and programming. A USB webcam was used for image collection as well. These were connected to a powered USB hub. We were able to run the entire system off of an unpowered hub, but ran into several issues with the WiFi/Camera that were remedied by full wall power to the powered USB hub. Unfortunately this means the robot was tethered to wall power.

Software Application:

The DHB1 PMOD is connected to the PMODA port on the PYNQ board, and is configured using the PMOD_IO IOP (Input/Output Processor) provided by Digilent/Xilinx. The PMOD_IO module allows you to program the ports of the PMOD as either input or output GPIO ports. You cannot program different ports within a PMOD to Input and output as the same time with this interface. For this reason we configured the enable and direction pins of the DHB1 (Pin0 = Enable1, Pin1 = Direction1, Pin4 = Enable2, Pin5 = Direction2) as outputs. This means that by driving a value to the direction and enable pin the motor will spin at full force in the direction specified. By limiting the time the motor is on, we can adjust the distance the motor travels. We mapped the appropriate mixture of direction and enables to create a motor interface with forward(), reverse(), left(), and right() movement capabilities, with a single parameter for each function telling the robot how long to hold that action (shown in the table below).

Action	Left Motor Enable	Left Motor Direction	Right Motor Enable	Right Motor Direction
Forward()	1	0	1	0
Reverse()	1	1	1	1
Right()	1	0	1	1
Left()	1	1	1	0

Table 1: *Motor signals*

The gripper arm servo is connected to PMODB's pin0 and this pin is configured using the PMOD_PWM IOP. This IOP utilizes the AXI Timer/Counter in PWM mode to drive one of the PMOD pins with a PWM signal. A servo is driven by a pulse of a given length, matching the angle the servo should be held, that needs to be updated every so often. This correlates to a PWM signal where the period of the PWM is the refresh timing, 25ms in our case, and the pulse correlates to the duty cycle of the PWM signal. For zero degrees a standard servo needs a pulse of 500 microseconds, which matches a 2% duty cycle of our 25ms PWM signal. This corresponds with the closed state of the gripper arm. For the open state, we don't need a lot of change in degrees, so a 1ms pulse or 4% duty cycle works well for releasing the gripper. A PWM stop signal is given to the servo when in the release state as there is not holding force required.

All of the functionality described is packaged into a Python class called Robot_Controller.py. This allows it to be used as any other IOP in the PYNQ ecosystem. This is the strength of the PYNQ design, you can develop the functionality and complexity necessary at the level of programmability you are accustomed and/or trained to. You can develop using only Python classes and IOP's that are pre-built, or you can design custom C-code that configures the hardware in more a more detailed manner. This was explored when initially designing a Pmod-HB3 controller for the PYNQ, where we wanted to be able to drive a PWM signal to the motor and treat the direction pin as a GPIO output, and the sensor inputs as GPIO inputs. This was unsuccessful, but demonstrates the ability to design C-level driver functionality and push those packaged C apps in *.bin format into python for use in your python interface. The highest level of customization comes in the ability to create custom Overlay's for the PYNQ board, reconfiguring the FPGA fabric to suite your needs. Since the main OS runs on the dual ARM cores, the bit files for the Zynq configuration can be loaded while the board is running, allowing for rapid configuration changes to be performed on the PYNQ board to suite your needs. This was explored, but not implemented for this design, as the IOP configuration was sufficient for our needs, and the task of modifying the IOP infrastructure was fairly overwhelming. Given more time it would have been beneficial to attempt a fully custom Overlay that we could use for configuring all of the features of the DHB1 Pmod.

The physical design and software design were combined to successfully control the robot manually to retrieve a can and deposit the can in a target location. After some testing and further integration of the autonomous CV based algorithms, some limitations and failures of the physical design were apparent, mainly the limitations of the small toy motors used with this motor platform. The plastic to metal connections of the motors failed under heavy load and heavy drive. A lighter load on the robot or smaller drive would have helped the stability of the motion of the robot.

Computer Vision:

Using the popular open source Computer Vision frame work, OpenCV 3.2, our design utilized python bindings for the OpenCV libraries for image processing.

The main concept of our design was that the robot platform would be able to autonomously detect objects of interest (a soda can, a drop off target), be able to navigate to them, and either pick up the object (soda can) or drop the object off at (the target).The design was constrained by the fact that only one camera would be utilized due to physical space on the robot chassis and additional sensors (i.e. ultrasonic range finder,etc) would not be utilized because of the limited number of pmod slots available on the PYNQ board.

This provided a significant challenge for the computer vision side of the design. Given one camera only and with the camera positioned directly behind the servo claw, the necessary data needed to be extracted from the frames of the Logitech webcam.

The initial approach to developing the computer vision algorithms for our design was a two-pronged strategy. First, classic computer vision algorithms were pursued ; Secondly, Deep learning networks were investigated with the hope that a neural network would be more robust in object detection --whilst the classic CV methods might be quicker to train and implement. The overall idea was to compare the two different approaches, and with the impending deadline, pick the methods that would provide the fastest delivery of functionality.

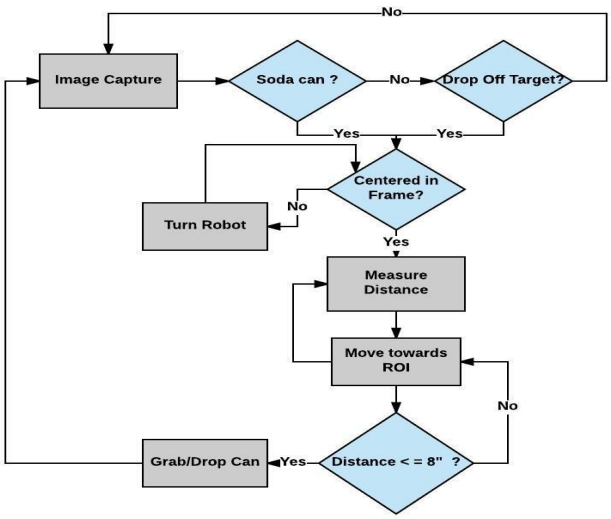


Figure 2: *Object detection flow diagram*

Figure 2 depicts The high level of view of the CV workload:

General algorithm

1. From the video frames, the CV implementation scans for either a can or drop off target (for our purposes a blue poster board).
2. If either object is detected in the foreground of the frame, the relative position of the detected ROI is compared to the absolute width of the web camera's frame and the robot is queued to correct the camera alignment by rotating either left or right to center the object in the camera frame.
3. The distance to the ROI is measured via triangulation (similarity of triangles)

4. The robot is queued to move towards the ROI in increments of 2 inches
5. If the distance measured to the ROI <= 8 inches, and the robot advances to the ROI and will either:
    a. If the ROI is the can: servo claw queued to open position, and a “pickup” attempt is made: the robot advances and closes its claw.
    b. If the ROI is the drop off target: the robot advances to the target and the servo claw is queued to the open position, dropping the soda can.

Can detection:
Initially, the idea was to offload the main workload of object detection on a custom built classifier. Using the Viola-Jones object detection framework, the development of a Haar-cascade classifier was undertaken. The Viola-Jones algorithm uses a sliding window, sized to match the target of interest, to scan the input image. For each subsection of the the input image, a Haar-like feature is calculated. Haar-like features are inputs to basic 2 leaf decision tree classifiers that either output a 1, indicating that the region is a likely candidate of the target or a 0 otherwise.

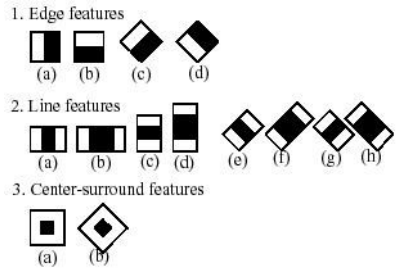


Figure 3: *Haar-features used by OpenCv*

Haar-cascade classifier

Composed of Haar- like features, adjacent rectangular regions at specific locations in the detection window are analyzed. The sum of the pixel intensities of each region is computed and the difference between these sums are used to categorize subsections of an image. The classifier is “cascaded”, meaning it is composed of many classifiers (stages) and can be rescaled to search for the target with larger/smaller dimension. Detection is signaled by a drawn bounding box surrounding the ROI in the image.

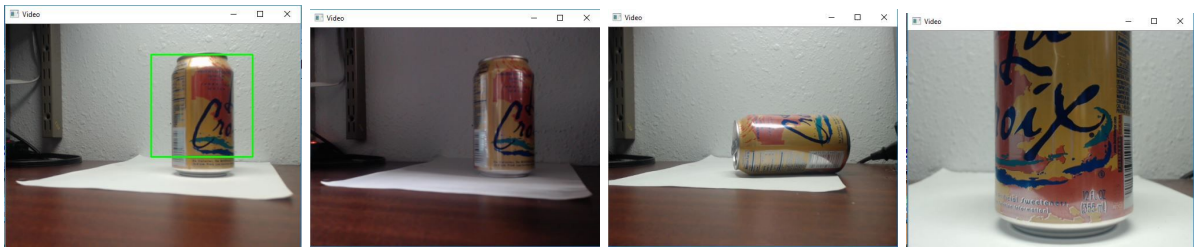


Figure 4: *Haar-cascade classifier results: decent illumination, poor illumination, rotated can, camera close to target*

For the purpose of our design the above Haar-classifier was trained with:

Type	Pos samples	Neg samples	Stages	Min. hit rate	Max False Alarm rate	Max. Depth Weak Tree	Max Weak Trees
Haar	1085	3044	20	0.995	0.5	1	100

Table 2: *Haar-cascade classifier training parameters*

As can be seen in Figure 4, the trained classifier can detect the soda can under certain conditions:

- Minimal background noise : a white background and surface was necessary in preventing false positives from surface reflection and occlusions created by a noisy background.
- Lighting condition: the lighting must illuminate the target sufficiently, poor lighting produced inconsistent detection
- The can must be upright because the Haar-Classifier is not rotational invariant and small deviations from an “upright can position” will throw the detection off.
- The can must be a certain distance away from the camera lens in order to be detected. This is due to the positive training images and their scaling.
- The ROI established by the classifier did not fit the physical soda can. This fact swayed our decision to abandon the classifier because our algorithm required that the defined ROI match the physical dimensions of the target for distance/ frame calibration.

After many trial runs and hours invested in attempting to fine tune the classifier for better results, it was determined that the hit rate of the classifier in its present state was too low. Therefore, another approach was needed.

Feature Detection

Using the OpenCv libraries, an object detection algorithm was devised with the following steps:From the input frames captured by the webcam:

1.Filter and smooth the image:

To eliminate noise and augment the preceding edge/ contour detection functions, the captured frame was:

- Converted to grayscale
- Filtered with the bilateralFilter() function, an edge preserving filter that averages (smooths) away weak difference in pixel values created by noise.
- Thresholded the image in order to separate regions of the image by way of pixel intensities. For our case, 0 (black) and 255 (white) was used.

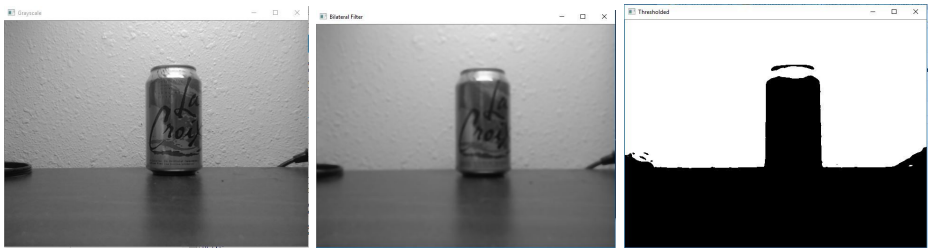


Figure 4: *Image processing sequence: gray scale, bilateral filtering, thresholding*

2. Detect edges of ROI in image: Opencv's Canny() edge detector function was used to detect edges. The algorithm's steps include:

1. Apply a Gaussian filter for noise
2. Apply a pair of convolution masks in order to find the intensity gradient of the image
3. Identify the intensity gradient strength and direction
4. Apply non-maximal suppression, eliminating pixels not considered part of an edge
5. Hysteresis by way of two thresholds, accepting all pixels above the upper threshold.

An additional step of image dilation, with a 3x3 kernel slid across the input image, is applied to strengthen the edges.

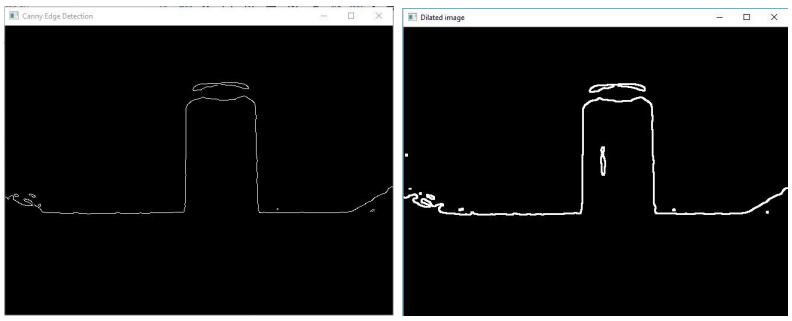


Figure 5: *Image processing sequence: Canny edge detection, dilation*

3.Collect contour data of ROI

With the edge information passed to the OpenCV function findContours() and parameters adjusted to:

- Retrieving only the extreme outer contours
- Storing all contour data into subsequent points: (x1,y1) & (x2, y2) of the contour will be either horizontal, vertical, or diagonal neighbors
- Discarding all contours under a certain length, thereby filtering out small contours generated by noise or pixel regions not necessary for the detection routine

A list of contour information from the frame is returned and the data was used to generate:

1. Shape of resulting external contours: The image processing routine retrieves the contour information and uses the Ramer-Douglas-Peucker algorithm to approximate polygonal curves with specified precision, returning either a “square” or “cylinder” detected from the input image. All other shapes are ignored because the robot only cares about the “drop off target” or “soda can.”
2. From the list of 2D contour points the minAreaRect() function returns the minimum area bounding rectangle of the point set. In our case, if a square/cylinder is detected, then from the data points, a bounding box is generated that fits the shape.

4. Generate bounding box around ROI and generate data from box data points

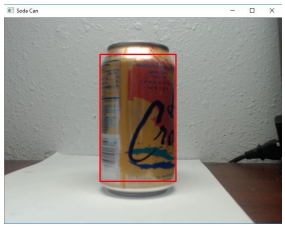


Figure 6: *Image processing sequence: minAreaRect() returns bounding box*

A successful bounding box of the target allows the routine to leverage the dimensions of the box itself to generate the necessary data for distance computation. From the bounding box , the center x position (cX) and maximum width (in inches and pixels) are returned.

5. Calibrate Frame position
In order to establish proper alignment from the robot servo claw to its target (can, drop off area) The relative position of the ROI must be corrected to be centered within the actual camera frame. To do this, the web cameras known frame width/height was found. Using the web camera’s frame center (frame width/2), an offset was created by: frame center - object’s x center.

A negative offset then indicates that the object is left of center, while a positive offset indicates that the object is right of center. The robot is then queued to either turn left or right (based on the offset value) to find center. Once center is found, the robot stops centering and begins to approach the object.

6. Distance measurement:

<u>Similarity of Triangles:</u>
Our design only afforded the use of one camera to determine the distance to the ROI in the captured frame. Using triangulation, a rough approximation of distance was computed.

The width of the object(W) to be detected is measured and we set the object at a known_distance(D) from the camera.The apparent width in pixels (P) of the object and the focal length(F) of the camera is calculated as follows: F=(P x D) / W

This was done empirically and a small script for camera calibration was run. Using a tape measure and making sure the soda can/target was centered in the frame, a known distance was recorded and the absolute width of the object was measured with calipers. Using the findContours() function returned data that generated the pixel dimensions of the object of interest. The distance to the object was then calculated by : D = (F x W) / P

This method generated a reasonable approximation of the real distance between the camera lens and the physical object. The limitations of this technique became apparent when the camera was in less than 8” of the object , due to the object overpowering the image foreground, which invalidated the measurement.

Therefore, it was deemed that anything under 8” was considered --very close to the object, since the servo claw extended 5” from the where the camera was housed on the robot chassis.

7. Color Detection

The captured frame is converted to the HSV color space .The lower and upper range of the color to be detected on the HSV scale is defined and thresholded with the original image, using the inRange() function, with results stored in arrays. These arrays are checked for non zeroes; for, if the HSV pixel data is present in the array then the target color is present in the ROI. Otherwise the array is zero and the ROI doesn’t contain the targeted color.
For Our Design purposes, the colors considered were: blue (drop-off target) and red (coke can).

Deep Learning network:

Our goal to incorporate a deep learning network proves to be overly ambitious as there were considerable challenges setting up the framework and required libraries. The learning curve proved to be too steep when faced with our 3 week development time frame. Therefore, after unsuccessful attempts, this approach was abandoned.

Implementation Results:

Individually, the different modules of our design were successfully implemented and showcased our intended functionality.

1.The motor was able to successfully navigate to a can, pick it up, and deposit it manually to a target. This proved out the physical design and functionality, which was encapsulated in a Python class.
2. Object recognition could successfully differentiate between the soda can and the drop off target (blues square poster board)
3. Camera calibration and distance measurement provided a rough approximation to the target
4. Color recognition could successfully differentiate between a red coke can and a blue poster board.

Our design was unsuccessful in the following areas:
1. The completion of automated motor control/navigation was partial because of physical malfunctions of the robot itself (motor/wheel failures)
2. Integration of the computer vision routines with the PYNQ board:

The details of our problems encountered are listed in**:** Appendix A: Issues and Challenges

References:

[1] (n.d.). Retrieved May 1, 2017, from http://www.pynq.io/

[2] Junior Runt Rover™. (n.d.). Retrieved May 1, 2017, from https://www.servocity.com/junior

[3] Standard Gripper Kit A. (n.d.). Retrieved May 1, 2017, from https://www.servocity.com/standard-gripper-kit-a

[4] 45° Dual Angle Pattern Bracket. (n.d.). Retrieved May 1, 2017, from https://www.servocity.com/45-dual-angle-channel-bracket

[5] Pmod DHB1: Dual H-bridge. (n.d.). Retrieved May 1, 2017, from http://store.digilentinc.com/pmod-dhb1-dual-h-bridge/

[6] (n.d.). Retrieved May 10, 2017, from http://docs.opencv.org/3.2.0/

[7] (n.d.). Retrieved June 1, 2017, from http://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html

[8] Rezaei, M. (2015, October 2). [PDF]. Creating a Cascade of Haar-Like Classifiers: Step by Step. Auckland: University of Auckland.

[9] Sinhal, K. (2017, January 23). Home. Retrieved May 25, 2017, from https://www.learnopencv.com/training-better-haar-lbp-cascade-eye-detector-opencv/

Appendix A: Issues and Challenges

Haar-cascade classifier:

Training the haar-cascade classifier is a time intensive task requiring thousands of positive and negative samples of images. The positive samples, for increased accuracy of the classifier, must be manually cropped or annotated with an ROI bounding box.

To obtain the requisite number of training images, ImageNet , an online database of images for machine learning was accessed via a Python script that used the urllib library to download the supplied urls of the targeted subject (i.e. “soda cans”). Many of the URLS were corrupt and the download process took time.

When selecting positive training images it was important to constrain the soda to a single orientation (in our simplest case: upright cans without any rotation) because the haar-cascade classifier is not rotational invariant. Additionally, large occlusions of the images were edited out to reduce polluting the positives with excess noise.

Deep Learning network:

Our initial impression, that we would be able to utilize a deep learning network to aid in can/brand name recognition, met many challenges. For one, the toolchain setup and platform dependencies were formidable as we tried installing:

- Neon : an open source Python based DL framework developed by Intel
  Neon supports Linux and there exists “unsupported” methods to get the framework up and running on Windows. After many hours spent trying to successfully install , we actually started running out of time for the other implementations which were still undone. So, we proceeded with the classic CV implementation for object and color detection instead of training a neural network for the same purpose.

- Keras & Tensor Flow: Another option which we tried was using Keras, a deep learning library for training neural network . We tried installing Keras with Tensor Flow backend but faced similar sorts of trouble in setting up the library.

Integration

Many issues became apparent when trying to integrate the CV:

a. The main interface to the user, Jupyter notebook, did not provide a clear method of displaying the video captured by the web camera in real time. Trying many suggested options, the bottleneck was that the video frames displayed in the Jupyter notebook cell were latency heavy. This was very problematic because it made debugging the CV routines impossible --without a clear idea of what the camera and CV routines were actually registering, integrating then became a blind guessing game.

b. The camera angle and position changed dramatically from developing the CV routine on a laptop with a stand alone web camera to testing the routine on the robot itself. The angle of the camera in relation with the target object is critical because the feature detection algorithm’s critical data is edges, and contour arcs. If the target object takes up too much of the foreground in the camera frame, then the detection of a “square” or “cylinder” becomes inconsistent -- leaving the robot doing nothing as it requires a positive identification of one of these objects to perform its next action.

c. Lighting conditions played a critical factor in object detection. Most of our development took place in the Capstone Lab -- a well lit environment. Variations of lighting tended to affect the object detection routine.

d. WiFi issues hampered development as wifi was intermittent at PSU.

e. Motor malfunctions caused issues with reliable turning of the robot. As the weight of the robot was substantial and the motors driven heavily, the plastic motor hubs stripped from the thin axles causing several wheels to spin in their axles rather than apply force to the ground.

Appendix B: Future Improvements

Computer vision:

In the end, the CV implementation used for integration proved too brittle to be successfully ported to the PYNQ board. Without regarding the interface and mechanical problems encountered, with more time, future development could consist of the following:

- Completion of the haar-cascade classifier: the classifier was a valid approach, as its performance is generally faster than the feature detection methods used. With more fine tuning, the classifier could produce the results desired. This would entail adding more positive/negative training samples, using the bounding box cropping feature of the Opencv trainer toolset, and making sure its scaling is correct in relation with the target image from the camera frame.

  Additionally, the classifier could be used in conjunction with the feature detection algorithms. This would provide a fast ROI detection, as the classifier could be the first stage in detection; and then hand its results to the feature detection routine. This would speed up object identification.

- Other methods of feature detection could be pursued : SURF, SIFT , which offer scale and rotational invariant methods of defining key features of the target object and fast performance in feature matching

- Adding sensors or an additional camera to help with camera calibration/distance measuring. Having one camera only proved to be brittle as well, because of the camera location on the robot chassis in relationship to its viewing angle. A second camera positioned higher and farther back could provide a better choice for the main object detection routine; leaving the camera by the servo claw to aid in close up measurements and the “gripping” action of picking up/ dropping off a can. Distance measurement could also become more accurate by utilizing the stereo camera technique.
- The feature detection routine could be augmented with consideration of the edge cases where the target object takes up the foreground of the camera frame. This might require a different set of distance calibration and shape detection for this inevitable case, when the robot closes in on its target.

Robot Design:

- Custom overlay allowing all of the features of the DHB1 to be utilized in Python, including sensing the motor speed and direction as well as controlling the wheel speed, not just the duration and direction of the wheels. This would require a custom IP that interfaced with the DHB1 utilizing PWM generation and detection to provide feedback on the motor state.
- A pressure sensor could have been added to the gripper arm to prevent crushing of empty cans or spilling of full/open cans.
- An armature could be added allowing for the can to be picked-up rather than sliding around the surface, which only works for fairly smooth surfaces.
- A lower power camera could be used to reduce weight and power consumption.
- A larger platform could have been used to allow for a can ‘payload’ where multiple cans could be retrieved and organized from a pile at a time.
- An ultrasonic range finder would have assisted in close range distance detection allowing for fine-tuning the can retrieval operation.
- The physical possibilities are endless, which is one of the reasons for the physical design choices.