On Board Vision System for Swarm Robotics

By: Ezra Idy

Professor: Vikram Kapila

Date: 9/4/18

Table of Content:

Acknowledgement:

I would first like to thank Professor Vikram Kapila for providing me with the space and resources needed to complete the project. I would also like to thank Sai Prasanth Krishnamoorthy for allowing me to take part and contribute in his swarm project. I would also like to thank him for all the help and advice he has provided me over the span of the project. Lastly I would like to thank my fellow lab mates for the constant support that they provided me.

Abstract:

The aim of this project was to create a hybrid swarm system that can move into different formations, using different approaches for neighbor detection and identification. The TurtleBots that were used for this project had all the necessary onboard sensors needed to complete the tasks. The robots operated through the Robotic Operating System, ROS, and they used OpenCV to address tasks related to image processing and object detection. The two distinct approaches of neighbor detection included a servo motor camera system and a 360° camera system. Both systems were able to detect and identify the neighboring objects surrounding the agents. The agents in both systems were also able to move into the desired formation through the help of human interaction and a user interface.

1. Introduction:

In the past decade the field of swarm robotics has taken huge strides in both research and industrial fields. As described in [1], a swarm robotics system is "the study of how large number of relatively simple physically embodied agents can be designed such that a desired collective behavior emerges from the local interactions among agents and between the agents and the environment." In a swarm robotics system, multiple robots need to communicate and collaborate with each other in order to accomplish a given task. Furthermore, when discussing swarm robotics, one must consider two distinct systems of communication: a centralized swarm robotic system and a decentralized swarm robotic system. Each system is desirable depending on the parameters of the environment as well as the application of the swarm system.

In a centralized swarm system, the swarm agents are connected to a central computer, where the central computer is tasked with processing the data collected from the swarm agents as well as monitoring the tasks of the agents. Using an external sensor, usually connected to the central computer, the agents are able to locate themselves relative to their environment and move to their intended location. Because each agent needs to be connected to the central computer, it is undesirable to scale a centralized swarm system to a large number of agents, thus there is a limit to the number of agents that could be present in a centralized swarm system. Another challenge centralized systems have to face is if a shutdown occurs within the central computer. Since all the agents are connected to the central computer, if the central computer is down the agents are unable to continue with their assigned task. Therefore, centralized systems are ideal for environments that are predetermined and known, such as warehouses or malls, where external sensors can be placed to monitor the agents, and where a central computer can be nearby to process and analyze the data. However, this type of system is undesirable when the environment is unknown to the swarm system, such as in search and rescue areas.

In the case mentioned above, a decentralized swarm system may be of better use. As the name implies, a decentralized swarm system has no central computer monitoring the agents in the field. Each agent is its own entity, and has its own onboard sensors and microcontrollers to process the data collected from the environment. Using their many sensors, the agents are able to interact with each other and the environment in order to complete the necessary goal. Since there is no connection to a centralized computer, the number of agents acting in the system can be increased with little effect on the performance of the swarm. Although, a decentralized system addresses the drawbacks of a centralized system, the system still faces its own issues. One such issue is that the swarm agents are only aware of their local tasks. Since there is no central computer and external sensors monitoring the system, the agents are unaware of global task assigned to the swarm. Instead, each agent completes a system of local tasks that when combined will lead to the completion of the systems global task. Thus if one agent goes down, the system will still operate and agents will still complete local tasks; however the global task of the system will still be jeopardized.

Based on the results obtained from [2], swarm robotics systems are still in their early stages despite the major advances. [2] breaks down real world swarm robotic systems into eight issues. It also shows that 77% of the research in swarm robotics only focuses on four of the eight issues. The aim of this project is to create a hybridized swarm system that tackles most of the issues expressed in [2], especially the issues that are under researched. This project also compares the performance of a servo motor camera swarm system design to that of a 360° camera swarm system design. The swarm system created is a hybrid system, thus it has characteristics of both a centralized and decentralized swarm system. The agents in the swarm are decentralized by nature, however an on looking supervisor can act as a central figure and task the agents to change into different formations.

2. Hardware

2.1 TurtleBot3:

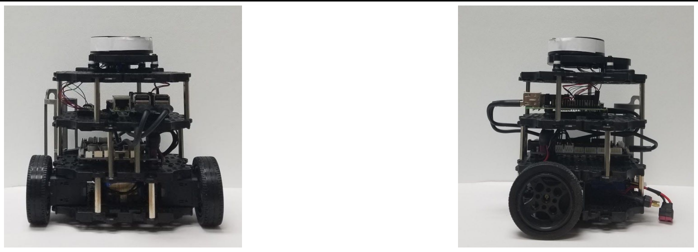


Figure 1**: Front and side view of a TurtleBot3 with no attachments**

The TurtleBot3 is a new generation mobile robot that is modular, compact, and customizable [3]. Created by Robotis, the TurtleBot3 is equipped with both an OpenCR board as well as a Raspberry Pi 3 board. The robot is a differential drive mobile vehicle, and uses Dynamixel X-Series motors to drive its fixed wheels. With no customized attachments, the robot's only sensor is a 2D LIDAR sensor. One can see the front and side view of the TurtleBot3 in Figure 1 above.

2.2 2D LIDAR Sensor:

Figure 2**: 2D LIDAR sensor**

A LIDAR, Light Detection and Ranging, is a sensor that uses light as a pulsed laser to measure variable distances [4]. With the data obtained from the LIDAR, one can map and receive information about the surrounding area. The LIDAR sensor was critical to the system, and acted as the primary sensor of the swarm agents. With the sensor, each robot was able to collect and process information about the surrounding environment. Using the data obtained from the

LIDAR, one was able to obtain the polar coordinates of objects relative to the agent, as well as track multiple objects that were being detected. However, the sensor was unable to determine the identity of the objects being detected, such as the difference between an obstacle in the environment and a neighboring swarm agent. Thus, in order to further understand the surrounding environment, additional sensing was needed through the help of a secondary sensor.

2.3 XBee:

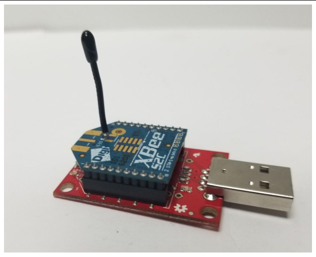


***Figure 3*: XBee device**

The XBee sensor, shown in Figure 3, was used to allow the robots to arrange themselves in different formations. Each swarm agent was given an XBee that could receive the appropriate information from an external user. The protocol that was being used by the XBee was known as Protocol 802. With Protocol 802 the XBees acted as a radio system, thus when the host sent the signal for the desired formation all the agents that were in the vicinity were able to receive the formation command as long as they were tuned to the correct frequency.

2.4 Camera Sensor:

A Raspberry Pi camera V2 was used for the vision system of the TurtleBot3 swarm agents, and acted as the secondary sensor for the swarm agents. The image produced by the camera had a resolution of 1280X960. With the camera system, the swarm agents were able to identify and label the objects obtained from the LIDAR sensor. Thus, the agents could determine whether the objects detected by the LIDAR were fellow agents or surrounding obstacles. This was done with the use of the AprilTag fiducial markers located on the neighboring agents. AprilTag markers are a robust and flexible visible fiducial system that allow full 6DOF localization of features from a single image [5], and can be seen on the agents in Figure 7 and Figure 13. Thus, when the camera detected an AprilTag marker on a neighboring object, the object was recognized as a neighboring agent within the given system. Using the Raspberry Pi camera, two distinct camera systems were designed when trying to implement a vision system on the agents: a servo motor camera system and a 360° camera system.

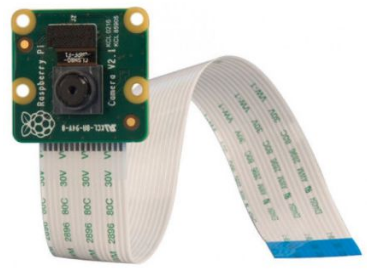


***Figure 4*: Raspberry Pi camera**

2.5 Arena:

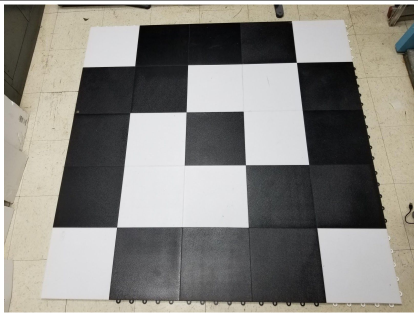


***Figure 5:* Arena**

Testing was done in a well-lit room. The agents were operating on a 5'X5' grid made from interlocking tiles as shown in Figure 5 above.

3. Servo Motor Camera System:

The servo motor camera system consisted of a Raspberry Pi camera attached to an X-Series Dynamixel motor. When the LIDAR detected an object within the set range of the sensor, the servo motor would move to the angle at which the object was detected. Once the camera was at the correct angle, the camera would take an image and processes it to determine the identity of the fellow neighbor. The camera was angled slightly downward, as shown in Figure 7, so that the lens was in line with the AprilTag markers located on the sides of the neighboring agents. The image taken with the camera was a high resolution image, having a size of 1280X960 and around 1.2 Megapixels.

3.1 Servo motor:

***Figure 6*: Dynamixel X-Series motor**

The servo motor used for the servo motor camera system design was a X-Series Dynamixel motor. The Dynamixel X-Series is a new line-up of high performance networked actuator module [6]. The motor came with various feedback and control methods. It was able to be customized using the OpenCR board provided by Robotis. Unlike the X-Series motors used for the differential drive of the robot, the motor used for the camera system was programmed to the position control setting, and the max velocity and max torque were adjusted accordingly. With this system design, another moving part was added to the TurtleBot3, thus making the agents more complex.

3.2 First Design:

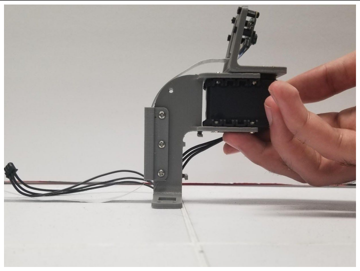


***Figure 7*: Servo camera system mount**

Figure 7 above displays the complete design for the servo motor camera mount. The mount was designed so that the camera and motor were hovering above the LIDAR sensor. One can see that the Raspberry Pi camera was angled slightly downward in the mount design. This design choice allowed for a reduction in the camera range, as well as to a line the camera with the AprilTag markers which were placed on the side of the TurtleBot3 agents, as shown in Figure 8. However, with this design approach, the camera mount was blocking the back of the LIDAR, thus creating a blind spot located behind all the agents. Also, in order to keep the camera wire from breaking, a restriction was placed on the servo motor. Thus the servo motor could only move from a range of -135° to 135°. Lastly, due to the high resolution of the image, the range of the neighbors had to be adjusted accordingly.

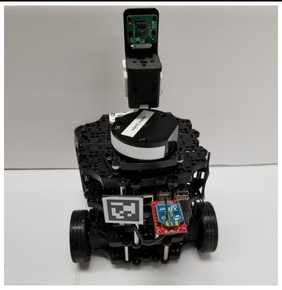


***Figure 8*: TurtleBot3 with servo camera system attachment**

Figure 8 above shows the TurtleBot3 with the servo motor camera attachment. As mentioned earlier, the AprilTag markers were located on the side of the robot alongside the XBee communication.

3.3 ROS RQT:

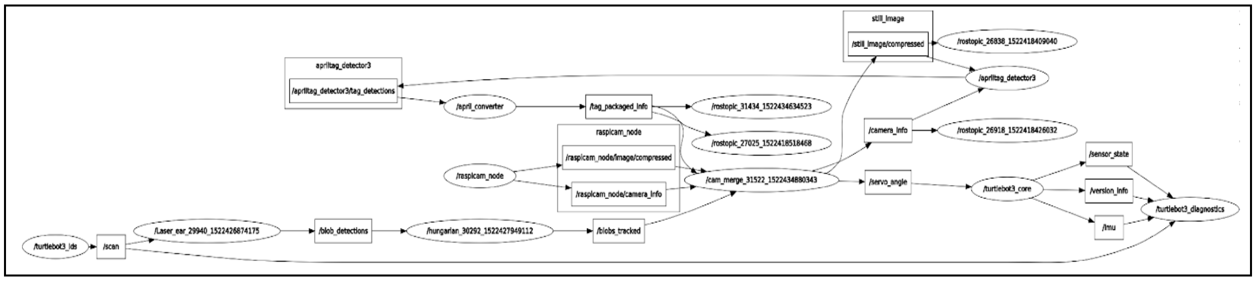

still_image
/still_image/compressed
/rostopic_26838_1522418409040
apriltag_detector3
/apriltag_detector3/tag_detections
/apriltag_detector3
/april_converter
/tag_packaged_info
/rostopic_31434_1522434634523
raspicam_node
/rostopic_27025_1522418518468
/camera_info
/rostopic_26918_1522418426032
/raspicam_node/image/compressed
/raspicam_node
/raspicam_node/camera_info
/cam_merge_31522_1522434880343
/servo_angle
/turtlebot3_core
/sensor_state
/version_info
/imu
/turtlebot3_diagnostics
/turtlebot3_lds
/scan
/Laser_ear_29940_1522426874175
/blob_detections
/hungarian_30292_1522427949112
/blobs_tracked

Figure 9: **RQT Graph of the servo motor camera system**

Figure 9 above displays an overview of the data processing happening within each agent in the system. The LIDAR data is sent to the *cam_merge* node through the scan topic shown above. As the information moves from node to node, the data is being processed so that when it reaches the *cam_merge* node, the data is in the correct format. Once in *cam_merge* the servo motor can move to the angle of an object detected from the LIDAR. Meanwhile, the camera image and information is being sent to the *cam_merge* node as well. While in *cam_merge* the image is made into a still image and then sent to the *apriltag_detector3* node to check to see if an AprilTag marker has been detected. If detected, the markers information is sent back to *cam_merge*, so that the LIDAR data can be updated. Once all objects have been detected one can insert Xbee communication devices and start doing linear formation control. The XBee signal is sent through the *xbee_com_node* thus allowing the formations to change. When receiving specific commands the *formation_node* will adjust the velocity of the TurtleBot3 so that the agents can enter the desired formation.

3.4 Servo Motor Movement:

Using the angle information provided from the LIDAR, the servo motor was able to move to the correct position. Due to the design, the servo motors movement was limited to only rotate between -135° and 135°. This limit was placed to protect the camera wire from snapping.[1]

3.5 Agent Tracking:

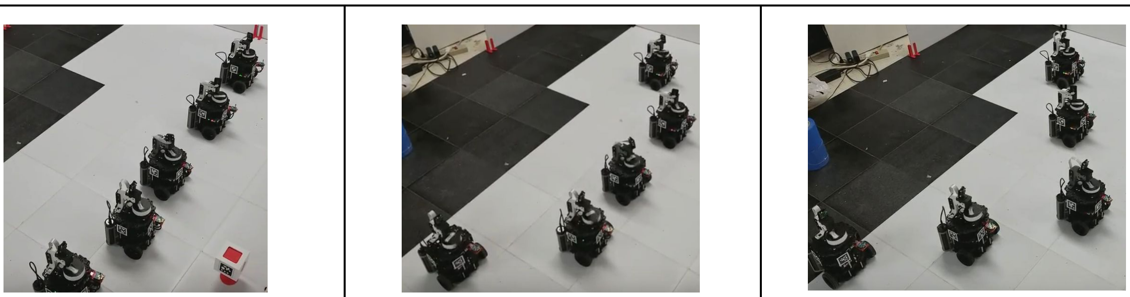
When the LIDAR detected an object, the object was given an identity of 99, which indicated that the object was considered unknown. When an unknown object appeared, the servo motor would rotate so that the camera was in the direction of the unknown object. Once at the correct location, the camera would take an image and try to identify an AprilTag marker. If a marker was detected within the image, the identity of that object would then change to the AprilTags unique number and that object would be considered as another agent in the swarm. However, if a marker was not detected, the object would be considered an obstacle with an identity of 999.[2]

3.6 Formation Control:

The one dimensional linear formation control implemented on the system operated using a proportional controller. Once all the neighboring objects were identified, the agents were able move into the desired formation that was being broadcasted. The formation control being used modeled a rooted out branching graph theory approach, thus all the agents moved in accordance to the leader agent located in the middle. The different formations can be seen in Figures 10 below. Since the system was using a decentralized system method for detecting neighboring objects, the swarm continued to move, and change formations, even if some agents were lost.

[1] The code for the movement of the servo motor can be found in the Appendix as Code 1.

[2] The code for the agent tracking of this system can be found in the Appendix as Code 2.



a) Formation 1 b) Formation 2 c) Formation 3

Figure 10: **Formations**

4. 360° Camera System:

The 360° camera system consisted of a Raspberry Pi camera placed underneath a 360° fisheye lens. When the LIDAR detected an object within the given range, the camera would take an image of the surrounding area. If an Apriltag marker was detected, the angle of the marker would be compared to the angle obtained from the LIDAR data. Due to the many transformations done to the image, the resolution was very poor. The dimensions of the image was 1180X200 and had around 0.2 Megapixels.

4.1 360° Attachment:

Unlike the servo motor camera system, the 360° camera system required no moving parts to detect for swarm agents in the area. With the 360° camera system the agents obtained a full 360° image, allowing for all the nearby objects to be detected within one image. The image was then remapped and dewarped, in order for the AprilTag markers to be detected with ease.

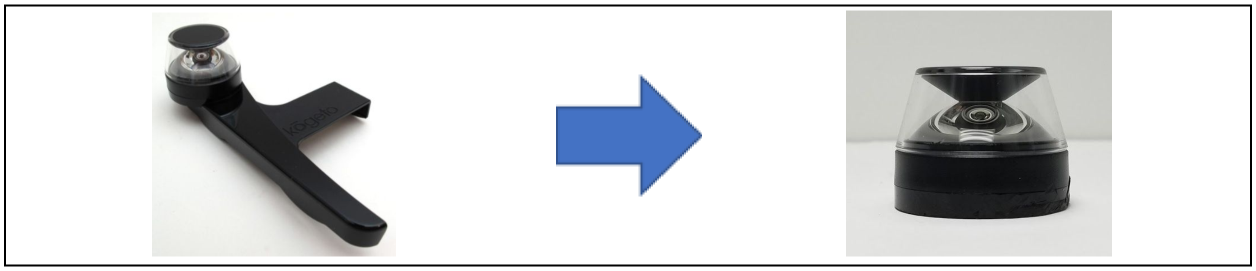


***Figure 11*: 360° adapter**

Figure 11 above shows the 360° adapter used for the second camera based system. The adapter was taken from an iPhone case. Once the 360° lens was separated from the case, a bezel was used to smooth out the rough edges, so that the piece could sit nicely on the camera lens. Figure 12 below shows the part that was designed to hold the 360° adapter. The part shown in the figure below was used for both designs of the 360° camera system.

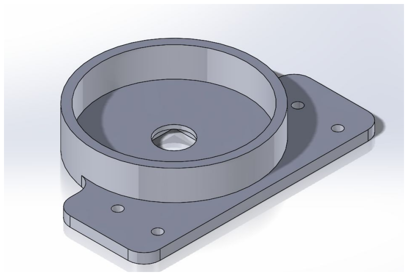


Figure 12: **360° adapter holder**

4.2 First Design:

When creating the pieces needed for the 360° camera system, inspiration was taken from the servo motor system's camera mount design. Thus, the camera was placed on top of the agent and the AprilTag markers were meant to be placed on the side of the agent. When implementing the first design, the camera was shown to detect the tags at up to 0.25 meters. While testing the design, the markers were unable to be detected due to the poor resolution of the camera and due to the low placement of the markers on the neighboring agents. Thus, a redesign of the camera mount had to be made. The first camera mount can be seen in Figure 13 below.

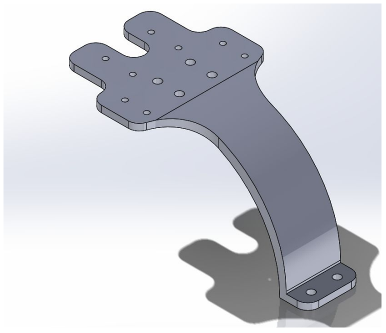


Figure 13: **First 360° Camera mount design**

4.3 Second Design:

When designing the second mount all the flaws that were present in the first design were considered. With this new design, the AprilTag markers were placed between the camera system and the LIDAR, which can be seen in Figure 16. By placing the markers closer to the camera, they were able to be seen well within the range of the cameras detection. As one can see in Figure 14, the redesign was mostly hollow allowing for easy placement and assembly of the camera. Although the redesign solved the issues of the first design, it also came with some minor issues. Since the 360° adapter holder was needed to be placed on top of the mount shown in Figure 14, side flaps were created. These side flaps created shadows over the side markers, making it hard for the camera to detect them. Thus, an add-on-piece was created to avoid further complications of the design. The piece can be seen in Figure 15.

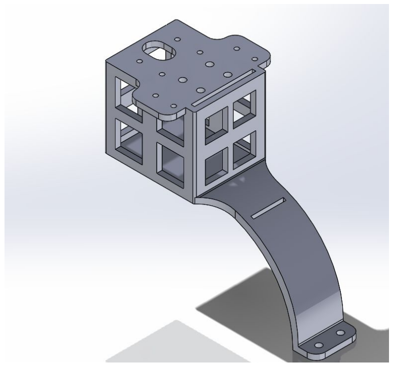


Figure 14: **Second 360° Camera mount design**

The add-on-piece was created to help extend the side AprilTag markers so that there won't be a shadow interfering with the detection. Due to the design of the camera mount the add-on-piece was able to be inserted like a puzzle piece. The piece was custom made to fit nicely with the camera mount. The pieces were also designed to be compatible with both the right and left side of the agent, making it a standard part.

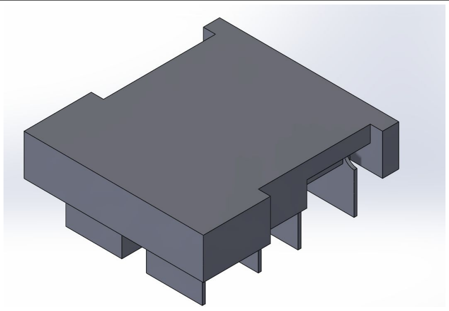


Figure 15: **AprilTag marker add-on-piece**

Figure 16 below shows the side and front view of the TurtleBot3 with the 360° camera system attachment. Due to the sensitivity of the LIDAR, all camera mounts had to be black. The add-on-piece was also printed in black which allowed for better processing of the AprilTags, especially when applying thresholding.

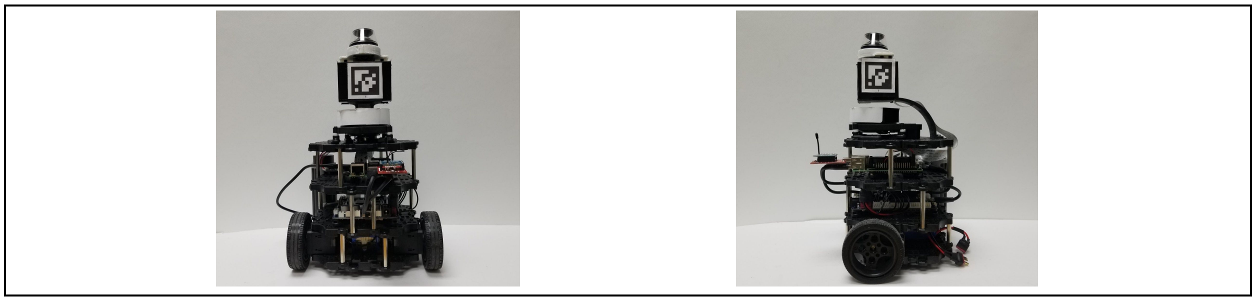


Figure 16: **TurtleBot3 with 360° camera system attachment**

4.4 ROS RQT:

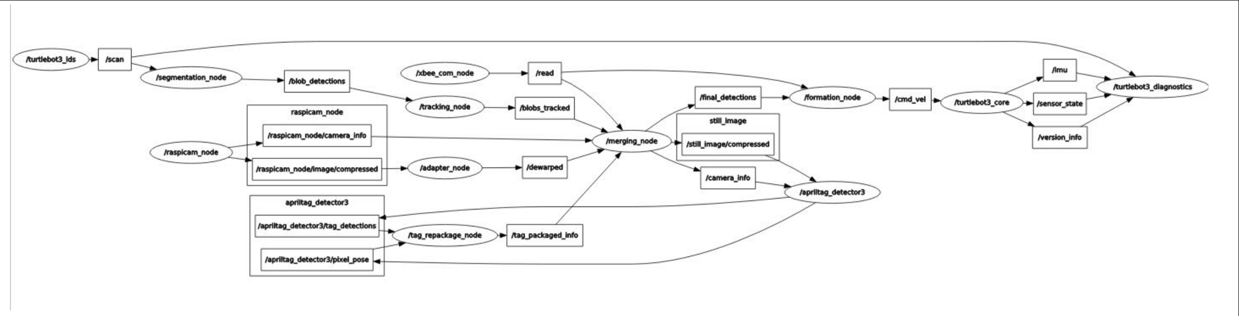


Figure 17: **RQT Graph of the 360° camera system**

Figure 17 above displays an overview of the data processing happening within each agent in the swarm. The LIDAR data is sent to the *merging_node* through the scan topic shown above. As the information moves from node to node, the data is being processed so that when it reaches the *merging_node*, the data is in the correct format. Meanwhile, the camera image and information is being sent to the *merging_node* as well. Before reaching the node however, the image is sent to the *adapter_node* were it is processed and dewarped using OpenCV. While in the *merging_node* the image is made into a still image and then sent to the *apriltag_detector3* node to check to see if an AprilTag has been detected. If detected, the AprilTag information is sent back to the *merging_node*, so that the LIDAR data can be updated. Lastly an XBee signal can be sent through the *xbee_com_node* thus allowing the formations to change. When receiving specific commands the *formation_node* will adjust the velocity of the TurtleBot3 wheels so that the robots can enter the desired formation.

4.5 360° Dewarping:

Figure 18 below displays the image taken from the Raspberry Pi camera when using the 360° adapter. The issue with the image below is that the edges bend due to the adapter being a fisheye lens. Another issue was that the Cartesian position and orientation information received from the AprilTag markers were wrong due to the transformation of the image by the adapter. Thus, in order for the AprilTags to be read properly, the image needed to be dewarped. In order to dewarp the image, a remapping was applied. Thus, the pixels within the appropriate boundary would be remapped to allow for a rectangular image [6]. The boundary for the remapping can be seen in Figure 18 below. Lastly, the dewarped image can be seen in Figure 19. Since there is a known blind spot with the camera design, some shifting of theta was needed to make sure that the image was split at the blind spot.

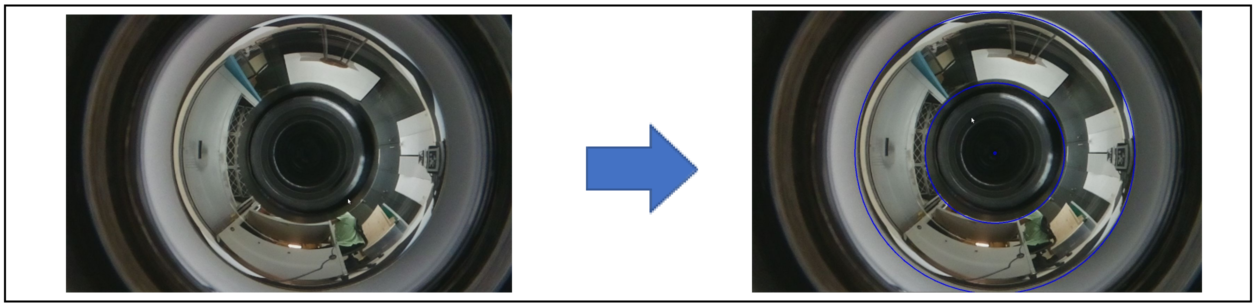


Figure 18: **Applying boundaries to the image**

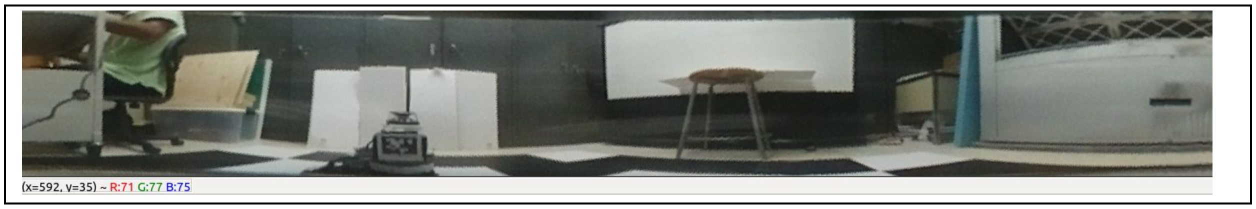


Figure 19: **Dewarped image**

Once the image was dewarped, the AprilTag marker can give the pixel coordinates within the dewarped image. Using the x-position of the AprilTag pixel coordinates and knowing the width of the dewarped image, one could calculate the angle at which the AprilTag was located. Lastly, in order to track the robot the angle of the AprilTag needed to be compared to the angle given by the LIDAR.[3]

4.6 Thresholding:

Even when the image was dewarped, the camera had issues detecting the AprilTag markers due to the lighting of the environment, the poor image resolution, and the distance between the agents. To help with the detection binary thresholding was applied to the image [7]. Thresholding is the concept of converting a colored, or grayscale, image into a black and white image, by providing a limit on the color value of every pixel in the image. By doing this, the pixels in the image will either be 0 or 255, black or white respectively. Since the camera was only being used to detect the AprilTag markers, a color image as shown in Figure 19 was not needed. One could obtain similar or better results with a black and white image. As shown in Figure 20, the threshold point was not 127. The number for the threshold point was obtained through trial and error for the environment in which testing was done. Through testing, it was

[3] The code for dewarping the image can be found in the Appendix as Code 3.

seen that when using 80 as the threshold point, the agents were able to detect the AprilTag markers more easily and the range of the detection was able to extend to around ~0.3 meters.[4]

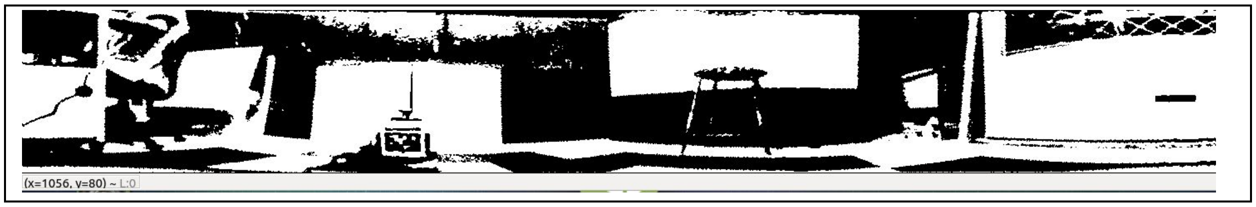


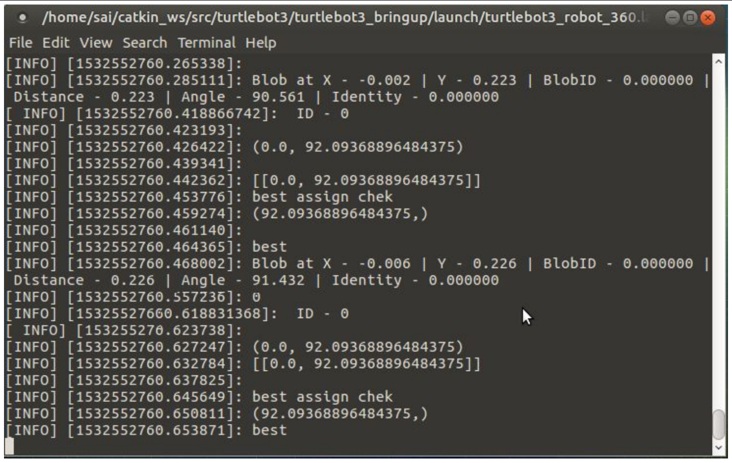
Figure 20: **Threshold image, with cut-off at 80**

4.7 Robot Tracking:

Since the angles were calculated using the pixel position in the dewarped image, as well as with the LIDAR, all that was needed to accomplish tracking was to compare the given angles to each other. Due to the location of the two sensors, as well as the different approaches in calculating the angles it was necessary to apply a tolerance to account for some errors. Thus a tolerance was set to 6.5°, however usually the angles only had a 1° degree difference as shown in the figure below.

In Figure 21, one can see that the LIDAR detected the object at an angle of 91.432°, while the camera detected an AprilTag marker at an angle of 92.09°. Since these angles are well within the tolerance range, the object's identity changed from 99 to the AprilTag ID which was 0. Once the object was identified, the identity wouldn't change unless the object was lost to the LIDAR. Thus, as long as the BlobID remained the same, the identified object would remain identified. Furthermore, all confirmed obstacles are reset to unconfirmed objects with every 20 iterations of the code. This was done in case the identity of a neighboring agent was overlooked.[5]

[4] The code for thresholding can be found in the Appendix as Code 4.
[5] The code for robot tracking can be found in the Appendix as Code 5.

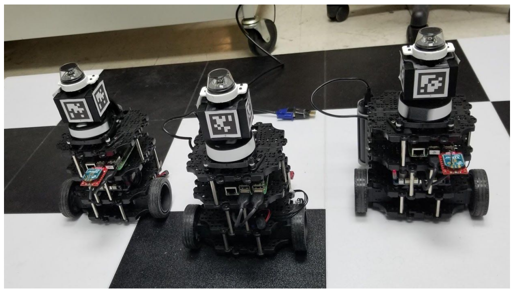
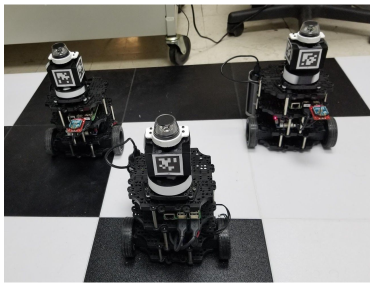

/home/sai/catkin_ws/src/turtlebot3/turtlebot3_bringup/launch/turtlebot3_robot_360.l
File Edit View Search Terminal Help

```
[INFO] [1532552760.265338]:
[INFO] [1532552760.285111]: Blob at X - -0.002 | Y - 0.223 | BlobID - 0.000000 |
 Distance - 0.223 | Angle - 90.561 | Identity - 0.000000
[ INFO] [1532552760.418866742]:  ID - 0
[INFO] [1532552760.423193]:
[INFO] [1532552760.426422]: (0.0, 92.09368896484375)
[INFO] [1532552760.439341]:
[INFO] [1532552760.442362]: [[0.0, 92.09368896484375]]
[INFO] [1532552760.453776]: best assign chek
[INFO] [1532552760.459274]: (92.09368896484375,)
[INFO] [1532552760.461140]:
[INFO] [1532552760.464365]: best
[INFO] [1532552760.468002]: Blob at X - -0.006 | Y - 0.226 | BlobID - 0.000000 |
 Distance - 0.226 | Angle - 91.432 | Identity - 0.000000
[INFO] [1532552760.557236]: 0
[INFO] [15325527660.618831368]:  ID - 0
[ INFO] [153255270.623738]:
[INFO] [1532552760.627247]: (0.0, 92.09368896484375)
[INFO] [1532552760.632784]: [[0.0, 92.09368896484375]]
[INFO] [1532552760.637825]:
[INFO] [1532552760.645649]: best assign chek
[INFO] [1532552760.650811]: (92.09368896484375,)
[INFO] [1532552760.653871]: best
```

Figure 21: **Robot detection**

4.8 Formation Control:

The formation control implemented on the agents operated using a proportional controller. When the neighboring agents were detected, the agents would move into formation depending on what formation was being broadcasted. The formation control being used modeled a rooted out branching graph theory approach, thus the agents would be following the leader, located in the middle. The different formations are shown in Figures 22. Since the system was a decentralized system, the agents would continue to move into formation even if some agents were unable to detect the leader. Lastly the swarm moved in a linear one dimensional formation.

 a) Formation 1	 b) Formation 2

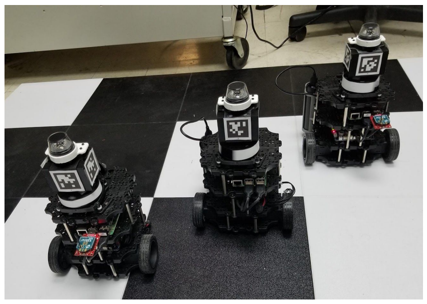
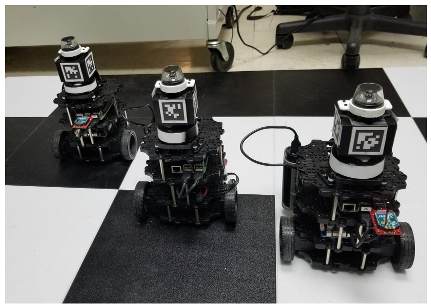
 c) Formation 3	 d) Formation 4

Figure 22: **Formations**

5. Issues:

Although the robots were able to move into formation and identify the surrounding neighbors, there were still pressing issues that should be noted.

5.1 Lighting:

Lighting will always an issue when working with any vision system. In order for the AprilTag markers to be detected quickly and properly, the camera required an appropriate amount of lighting in the environment. Furthermore, due to constant transformations that were taking place, the 360° camera system required an even better source of lighting to detect the AprilTags. The servo motor system also required lighting for detection, however not as much as the 360° system, since the images taken were at a higher resolution.

5.2 Range:

Since the servo motor camera system was using a high resolution image, the range at which it could detect AprilTags was much greater than that of the 360° camera system. Although not formally tested, the servo motor camera system was able to detect AprilTags ~2 meters away, while the 360° camera system was only able to detect AprilTags ~0.24 meters away. When applying thresholding to the 360° camera system the range increased slightly to ~0.3 meters. The reduction in the range was due to the poor resolution of the image after so many transformations have been applied. Due to the servo motor system having a larger range of detection, limitations needed to be applied so that objects were not being falsely identified. Thus, although poor, the range of the 360° camera system was a decent range for which a neighbor could be classified.

5.3 Time delay:

Another issue that was faced when working with the two-different systems were the time delays necessary for the images to be processed. With the servo motor system, the movement of the servo motor took a couple of seconds to get into the necessary position and then the camera needed a couple of seconds to take and process the image. The servo motor then had to repeat the functionality for all the objects detected by the LIDAR before the agents could move into formation. With the 360° camera system, all objects detected with the LIDAR were in the same image, and the angles of the two devices were compared. Since the 360° system only cared for robot detections the tracking time would theoretically be much quicker. However, there was still a time delay that took place within the 360° camera system. The delay happened when the agents were initiated. The dewarping node took a while to load, thus the robots would not start tracking until the camera was up and running. Also due to the poor resolution of the image, it could be some time before the agents were detected. However, since all surrounding obstacles get processed at the same time, in an environment where a lot of objects are being detected, the 360° camera system would theoretically be much faster than the servo motor camera system.

5.4 Blind Spots:

Both systems had a blind spot located in the back of the agents. This blind spot was due to the placement of the camera on top of the LIDAR. Since the camera stand blocked the LIDAR at the back of the agent, a blind spot would occur and the neighbor could not be detected if placed directly behind the robot. The servo motor system had a larger blind spot than the 360° system, due to the safety of the camera wire. However, by creating a blind spot, limitations were applied to both swarm systems.

6. Conclusion:

6.1 Future Work:

Although the agents successfully achieved formation using both of the camera system designs, improvements could still be made to account for the many different issues mentioned in the prior section. While working with the Raspberry Pi camera and the AprilTag detection node, it was clear that proper lighting was needed in order for detections to occur. Thus the environment in which the robots can operate must be unique. One possible way to fix this issue is to create a thresholding with a feedback loop. Thus, the thresholding criteria will be able to be adjusted to allow for detection optimization in an environment. However, by doing so the time delay of the robots might increase. Another possible improvement is to redesign sturdier camera mounts that can help reduce the blind spot.

6.2 Conclusion:

The project demonstrated a swarm system achieving linear formation control using two different vision system. Both systems were able to achieve the desired formations, and the agents were able to detect their fellow neighbors. The 360° camera system achieved detection without any extra moving parts, with a decreased detection time, and with poorer image resolution. However, the max range of the neighbor was drastically decreased, when compared to the servo motor system. The 360° system also worked better in a busier environment, since the camera took an image of the total surroundings as opposed to individual objects. Both systems could be considered novel approaches to a vision based swarm robotics system. Furthermore, both systems were able to interact with humans in an abstract manner, using the XBee communication device.

<u>References:</u>

[1] E. Şahin, “Swarm robotics: from sources of inspiration to domains of application,” in Swarm Robotics Workshop: State-of- the-Art Survey, E Şahin and W. Spears, Eds., Lecture Notes in Computer Science, no. 3342, pp. 10–20, Berlin, Germany, 2005.

[2] Barca, Jan Carlo, and Y. Ahmet Sekercioglu. “Swarm Robotics Reviewed.” *Robotica*, vol. 31, no. 03, 2012, pp. 345–359., doi:10.1017/s026357471200032x.

[3] Cortina, Alfredo, et al. “TurtleBot 3 Burger [US].” *ROBOTIS*, www.robotis.us/turtlebot-3-burger-us/.

[4] US Department of Commerce, and National Oceanic and Atmospheric Administration. “What Is LIDAR.” *NOAA's National Ocean Service*, 1 Oct. 2012, oceanservice.noaa.gov/facts/lidar.html.

[5] Olson, Edwin. “AprilTag: A Robust and Flexible Visual Fiducial System.” *2011 IEEE International Conference on Robotics and Automation*, 2011, doi:10.1109/icra.2011.5979561.

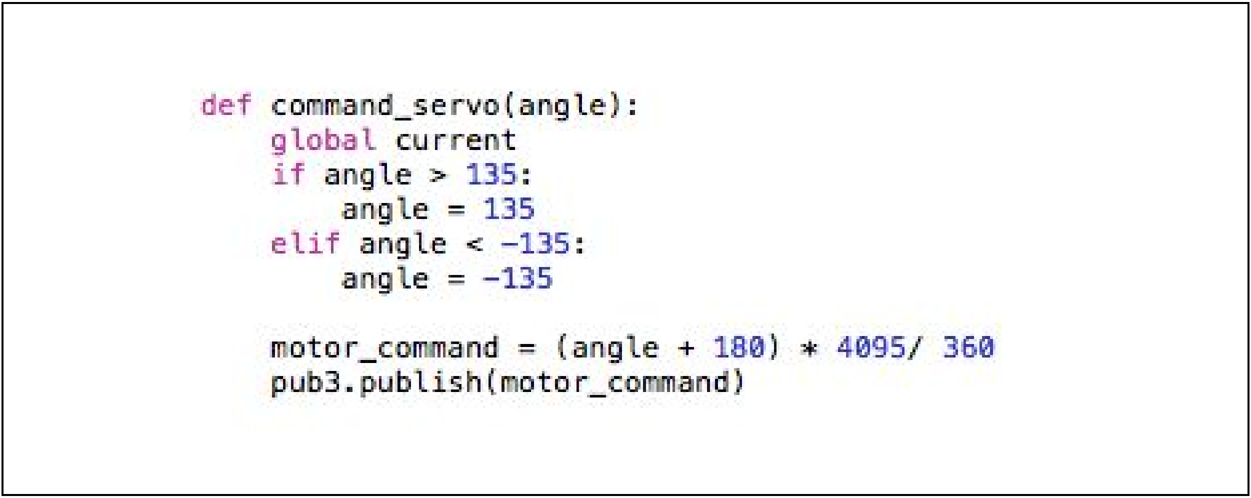
[6] “X Series.” *ROBOTIS*, www.robotis.us/x-series/.

[7] “Image Transformation.” *Learning OpenCV*, by Gary Bradski and Adrian Kaehler, O'Reilly Media, 2015, pp. 162–163.

[8] “Overview - Panorama Generation from the Periphery of a Fisheye.” *Atria Logic*, www.atrialogic.com/dewarping.php.

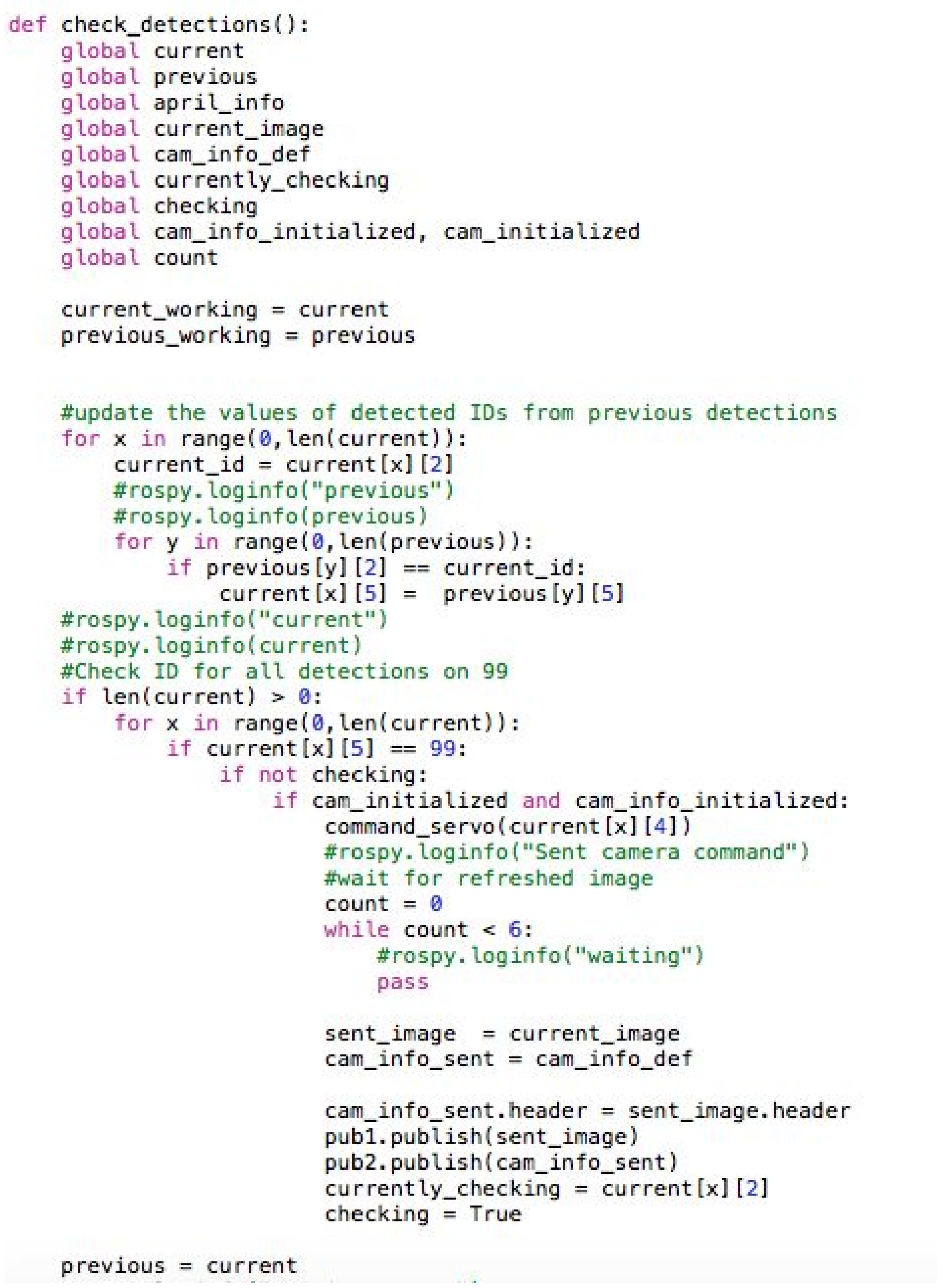
[9] “Image Processing.” *Learning OpenCV*, by Gary Bradski and Adrian Kaehler, O'Reilly Media, 2015, pp. 135–141.

Appendix:



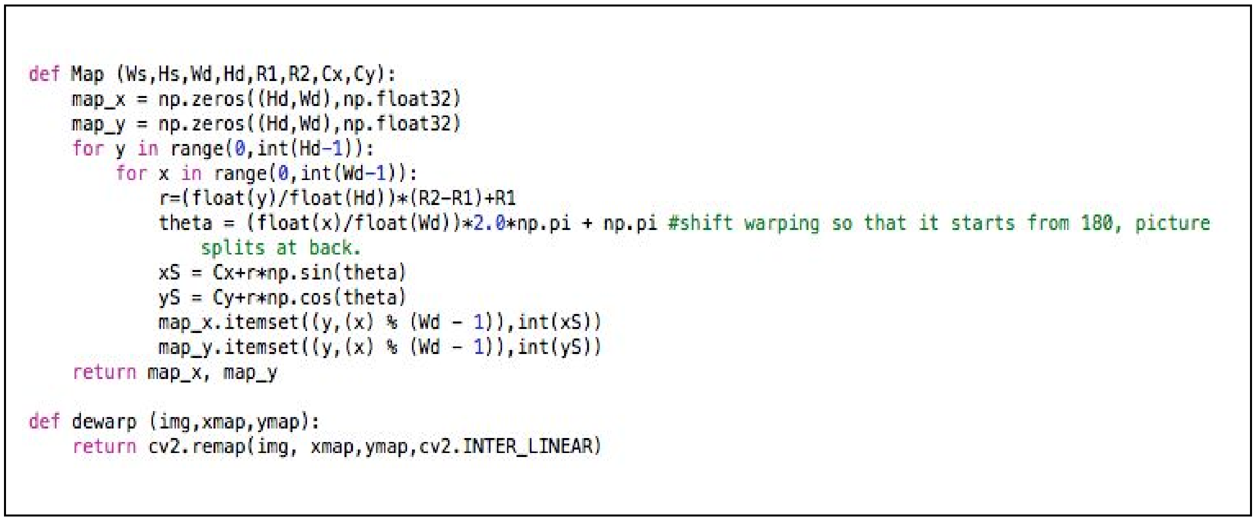
```
def command_servo(angle):
    global current
    if angle > 135:
        angle = 135
    elif angle < -135:
        angle = -135

    motor_command = (angle + 180) * 4095/ 360
    pub3.publish(motor_command)
```

Code 1: **Moving servo motor to detected object**



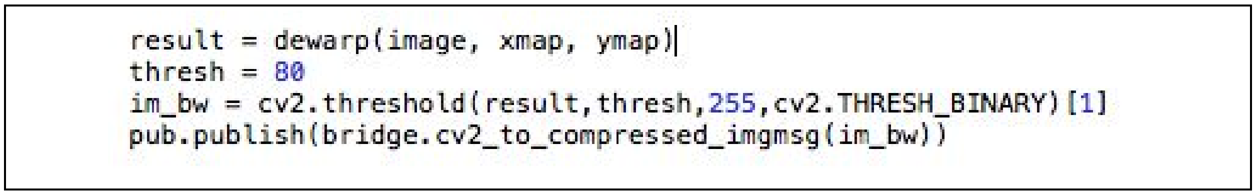
```python
def check_detections():
    global current
    global previous
    global april_info
    global current_image
    global cam_info_def
    global currently_checking
    global checking
    global cam_info_initialized, cam_initialized
    global count

    current_working = current
    previous_working = previous


    #update the values of detected IDs from previous detections
    for x in range(0,len(current)):
        current_id = current[x][2]
        #rospy.loginfo("previous")
        #rospy.loginfo(previous)
        for y in range(0,len(previous)):
            if previous[y][2] == current_id:
                current[x][5] =  previous[y][5]
    #rospy.loginfo("current")
    #rospy.loginfo(current)
    #Check ID for all detections on 99
    if len(current) > 0:
        for x in range(0,len(current)):
            if current[x][5] == 99:
                if not checking:
                    if cam_initialized and cam_info_initialized:
                        command_servo(current[x][4])
                        #rospy.loginfo("Sent camera command")
                        #wait for refreshed image
                        count = 0
                        while count < 6:
                            #rospy.loginfo("waiting")
                            pass

                        sent_image  = current_image
                        cam_info_sent = cam_info_def

                        cam_info_sent.header = sent_image.header
                        pub1.publish(sent_image)
                        pub2.publish(cam_info_sent)
                        currently_checking = current[x][2]
                        checking = True

    previous = current
```

Code 2: **Robot tracking with servo motor camera system**



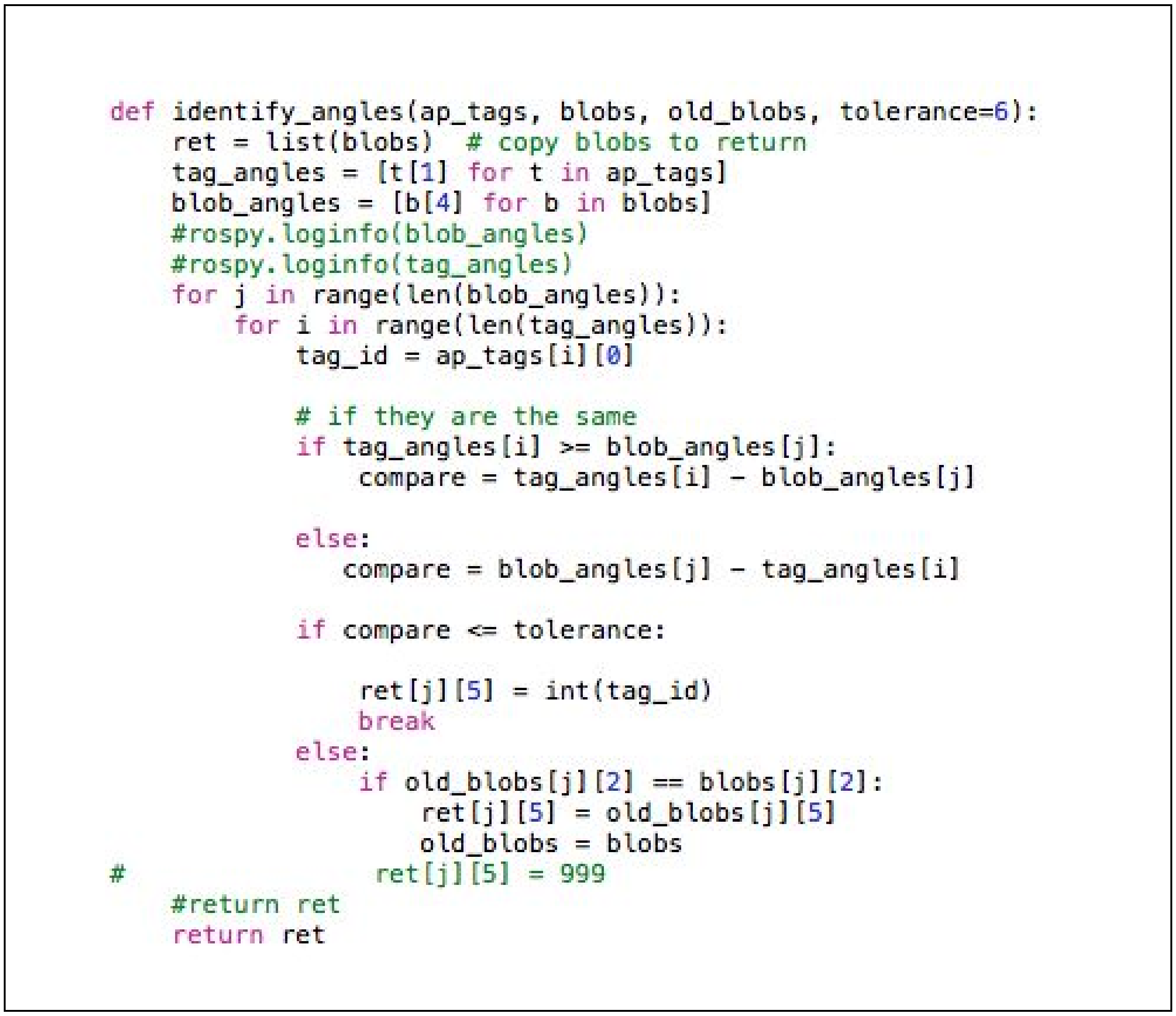
```
def Map (Ws,Hs,Wd,Hd,R1,R2,Cx,Cy):
    map_x = np.zeros((Hd,Wd),np.float32)
    map_y = np.zeros((Hd,Wd),np.float32)
    for y in range(0,int(Hd-1)):
        for x in range(0,int(Wd-1)):
            r=(float(y)/float(Hd))*(R2-R1)+R1
            theta = (float(x)/float(Wd))*2.0*np.pi + np.pi #shift warping so that it starts from 180, picture
                splits at back.
            xS = Cx+r*np.sin(theta)
            yS = Cy+r*np.cos(theta)
            map_x.itemset((y,(x) % (Wd - 1)),int(xS))
            map_y.itemset((y,(x) % (Wd - 1)),int(yS))
    return map_x, map_y

def dewarp (img,xmap,ymap):
    return cv2.remap(img, xmap,ymap,cv2.INTER_LINEAR)
```

Code 3: **Remapping of fisheye lens [5]**



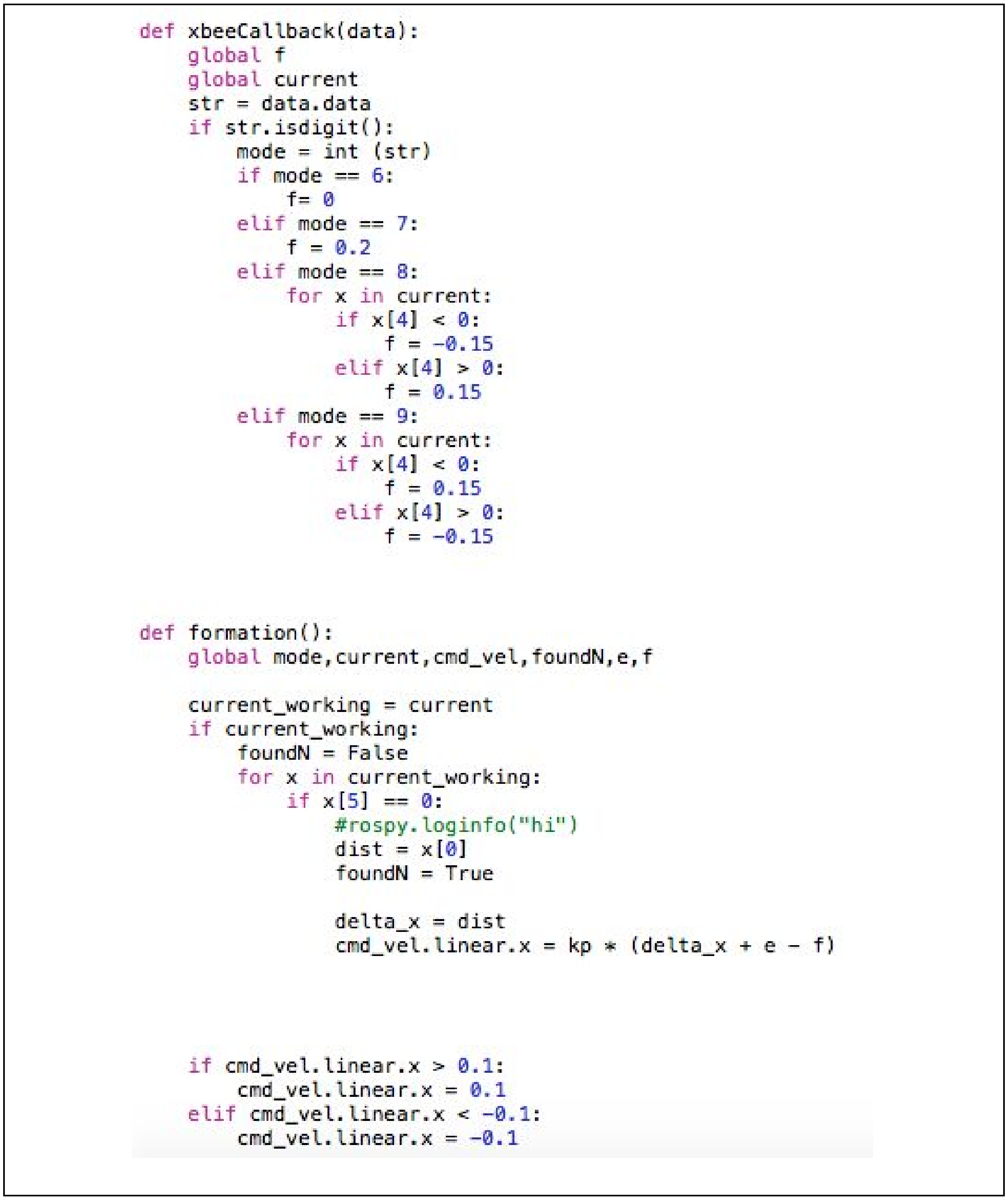
```
result = dewarp(image, xmap, ymap)
thresh = 80
im_bw = cv2.threshold(result,thresh,255,cv2.THRESH_BINARY)[1]
pub.publish(bridge.cv2_to_compressed_imgmsg(im_bw))
```

Code 4: **Thresholding**



```python
def identify_angles(ap_tags, blobs, old_blobs, tolerance=6):
    ret = list(blobs)  # copy blobs to return
    tag_angles = [t[1] for t in ap_tags]
    blob_angles = [b[4] for b in blobs]
    #rospy.loginfo(blob_angles)
    #rospy.loginfo(tag_angles)
    for j in range(len(blob_angles)):
        for i in range(len(tag_angles)):
            tag_id = ap_tags[i][0]

            # if they are the same
            if tag_angles[i] >= blob_angles[j]:
                compare = tag_angles[i] - blob_angles[j]

            else:
               compare = blob_angles[j] - tag_angles[i]

            if compare <= tolerance:

                ret[j][5] = int(tag_id)
                break
            else:
                if old_blobs[j][2] == blobs[j][2]:
                    ret[j][5] = old_blobs[j][5]
                    old_blobs = blobs
#                ret[j][5] = 999
    #return ret
    return ret
```

Code 5: **Robot tracking with 360° camera system**



```python
def xbeeCallback(data):
    global f
    global current
    str = data.data
    if str.isdigit():
        mode = int (str)
        if mode == 6:
            f= 0
        elif mode == 7:
            f = 0.2
        elif mode == 8:
            for x in current:
                if x[4] < 0:
                    f = -0.15
                elif x[4] > 0:
                    f = 0.15
        elif mode == 9:
            for x in current:
                if x[4] < 0:
                    f = 0.15
                elif x[4] > 0:
                    f = -0.15



def formation():
    global mode,current,cmd_vel,foundN,e,f

    current_working = current
    if current_working:
        foundN = False
        for x in current_working:
            if x[5] == 0:
                #rospy.loginfo("hi")
                dist = x[0]
                foundN = True

                delta_x = dist
                cmd_vel.linear.x = kp * (delta_x + e - f)




    if cmd_vel.linear.x > 0.1:
        cmd_vel.linear.x = 0.1
    elif cmd_vel.linear.x < -0.1:
        cmd_vel.linear.x = -0.1
```

Code 6: **Formation control**