Human Tracking Robot

Arjun Keerthi
Vanderbilt University
Nashville, TN, USA
arjun.b.keerthi@vanderbilt.edu

Rodman Zhu
Vanderbilt University
Nashville, TN, USA
rodman.y.zhu@vanderbilt.edu

Abstract

An autonomous robot that identified and followed a person was built and programmed. ROS was used to control the robot, while OpenCV was used for the identification and location of the human subject. Stereopsis was attempted, but lacking camera hardware limited the success of this approach. While a full-body model was planned to be used, ultimately a pre-trained face tracking model was used with a Haar Feature-based Cascade Classifier. In the end, a monocular approach was used to calculate distance and heading for the robot.

1. Related Work

A. Stereopsis and Depth

Stereopsis was implemented as a first attempt to attain depth information about the scene immediately in front of the robot. Various camera configurations were considered, including placing the cameras vertically and horizontally. Implementations of vertical stereo cameras have found that such a configuration can lead to more accurate pose estimation due to the removal of any dependency on robot rotation[1]. Additionally, vertical stereo configurations increase usable horizontal information at the expense of decreased vertical range[2]. However, our robot benefits from a maximal vertical range due to its low stature, making such a configuration less beneficial.

Multiple algorithms exist for computing the disparity map between the left and right cameras. Such algorithms attempt to match blocks of pixels from one image to the other. OpenCV's StereoBM implementation by Konolige minimizes the sum of absolute differences between windows across the pair of images[3], [4]. This algorithm runs efficiently on even low-resource machines but can be inaccurate. OpenCV also provides the StereoSGBM algorithm by Hirschmuller that, in addition to block matching, enforces smoothness constraints between neighboring pixels and tends to give more accurate results[5], [6].

Post-processing of disparity maps was also considered, as the computation of disparity maps can lead to holes where matches between the left and right images could not be found. OpenCV provides WLSFilter that attempts to match disparities along edges in the original image to attain a more smooth and accurate disparity map[7]. Mean and median filtering are additional methods for filling holes that are also explored in this report[8].

Disparities maps are used to compute depth of features in the scene. Assuming an ideal camera setup, with the image planes of each camera parallel and in line with one another, depth out to a point in space can be computed via:

$$z = \frac{fb}{d}$$

where z is the depth, f is the focal length, b is the baseline length between the two cameras, and d is the disparity at that pixel[9]. With more realistic camera setups, where such an idealization is almost impossible to recreate, we can use the following equation:

$$[x,\ y,\ z,\ w]^T = Q * [x,\ y,\ d,\ 1]^T$$

where (x, y, z) gives the 3D coordinate in space, (x, y) is the pixel coordinate in the image, d is the disparity at that pixel, and Q is defined as:

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & -1/T_x & (c_x - c'_x)/T_x) \end{bmatrix}$$

where cx and cy are the camera center coordinates of the left camera, f is the focal length of the left camera, Tx is the baseline from the left camera to the

right camera, and *cx'* is the x-coordinate of the center of the right camera image[10], [11]. OpenCV provides a method that implements this procedure (reprojectImageto3d()), but fails to account for holes in the disparity map, giving many infinite values, A manual implementation of this procedure is utilized to avoid this problem. The inverse relation between the disparity and depth is another motivation for the investigation into methods for filling holes in the disparity map.

B. *Human Identification*

Human tracking was a core function in the original planned idea. A few approaches were researched and examined before a final decision was made. The initial approach was to perform human pose tracking, much like the Xbox Kinect, which is used for many pose tracking applications[12]. This resulted in the discovery of the OpenPose project from CMU, which uses neural nets to detect body keypoints in real time[13]. However, the hardware requirements for running this software was much too high for the planned hardware platform, so the idea was scrapped.

The next two approaches that were looked into were Haar Feature-based Cascade Classifiers and Histograms of Oriented Gradients (HOG)[14]. Although they are both very early forms of human detection systems, they were considered because they ran much faster than the neural-net based OpenPose and were accurate enough for the purposes of this project. Although both of these approaches were relatively simple to implement due to OpenCV's built in functions for both of these approaches, of the two, a Haar Feature-based Cascade Classifier approach was chosen as it was the faster running of the two, something that was valued considering the limited hardware.

Haar Feature-based Cascade Classifiers identify Haar Features in an image, which are essentially small kernels that detect edge, line, or center-surround features[15]. These kernels are used to identify an object that is trained for identifying a specific object with a series of "positive" and "negative" samples[16]. The downsides of this approach is that the accuracy of this type of object detection decreases with less samples, so many samples are needed for a more accurate result.

Methods

A. *Overview*

ROS was utilized to coordinate the components of the robot. The video_stream_opencv driver was used to stream images from the Logitech C270 webcam (right) and the raspicam node was used for the Raspberry Pi Camera V2 webcam (left). These nodes published Image messages that were subscribed to by both the stereo_image_proc node as well as the object detection system (recognition.py). The cameras were calibrated using the stereo calibration procedure provided by ROS, which saved the camera parameters to a .yaml file that is used by the camera drivers and stereo nodes. Specifically, the stereo_image_proc node utilizes the image streamed from the left and right cameras and their corresponding camera matrices to compute and publish rectified left and right images. These images are subscribed by the stereopsis node (move_robot.cpp, admittedly misnamed), which computes the disparity map and depth, publishing this information as well. The object detection system detects the distance and angle to the human in the frame (see next section), also publishing this information. The depth, distance, and angle are all subscribed to by the motor control node (control_motors.py), which interfaces with the encoders and motors. This node utilizes a PID loop to adjust outputs to the motors based on the incoming information and attempt to move towards the human. A diagram displaying this is shown below:

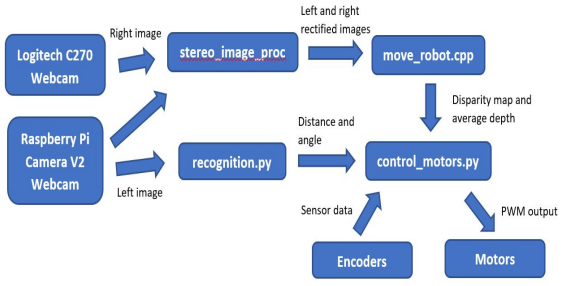

Figure 1: Robot Control Flow

B. *Stereopsis*

A C++ script (move_robot.py) was written in order to perform block matching and compute the disparity map on the incoming pair of images. To analyze the

performance of different algorithms, multiple implementations were included in this script. First, OpenCV's StereoBM and StereoSGBM block matching algorithm was included. The parameters for these matchers were derived from an article by Jay Rambhia[4]. There are numerous configuration options for these matchers, but important one for this camera setup are numberOfDisparities, disp12MaxDiff, speckleWindowSize, and speckleRange. The numberOfDisparities parameter specifies the range of pixels to slide the block window over when trying to find the best match. A large value allows for more range of depths, which is suitable for the relatively imprecise setup of the robot's cameras. The disp12MaxDiff parameter specifies the maximum allowable difference in disparities when comparing in the left-to-right and right-to-left directions. Small value here is good in order to select true matches and filter out wrong selections. The speckleWindowSize specifies the size below which a collection of pixels is considered a speckle (and not viable for a match). Choosing a small value here allows for greater features to be considered for matches. Finally, the speckleRange parameters chooses how close disparity values must be for pixels to be part of the same component[17].

Next, in an attempt to improve on the disparity maps, various filtering and blurring procedures were utilized. Gaussian blurring was done on the resulting disparity map to check if transitions between holes and valid disparities could be smoothened and it was helpful. This was accomplished using OpenCV's GaussianBlur() function. Similarly, box filtering was also utilized with the boxFilter() function from OpenCV[18]. For both these functions, a 5x5 kernel was used.

Additionally, morphological operations were also considered. Motivated by wanting to fill the holes in the disparity map, the grayscale morphological close operation was utilized as this helps fill holes in the foreground of the image, which includes regions assumed to have high disparities (points closer to the image plane should have higher disparities). Also, the grayscale morphological opening was considered as this can fill holes in the background. To implement these operations, OpenCV's morphologyEx() function was used[19]. For these operations, a 5x5 structuring element was used, populated by ones (used OpenCV's getStructuringElement() function to construct this).

Further investigating hole-filling methods, mean and median filtering methods were used. These filtering methods involve replacing each pixel with either the mean or median of the neighborhood (5x5 region was used in this implementation). To minimize distortions in regions without holes, a zero-mask was used to only apply the filter at hole regions (where disparity map is zero)[8]. These filters were applied repeatedly over the disparity map until either no more holes were found (no pixels left at zero), or a set number of iterations was met. Initially, a Python implementation was developed but proved to be too slow for real-time computation. As a result, C++ was used in order to make these methods viable. OpenCV slicing of cv::Mat objects was used to isolate neighborhoods of each pixel and cv::mean() was used to compute the mean of each neighborhood. To compute the medians, the neighborhoods were flattened into a vector and the std::nth_element() function was used.

Also, OpenCV's WLSFilter functionality was also implemented in order to filter the disparity map[7]. However, this required installation of the opencv-contrib modules, which was only available in Python for this robot. As a result, this filtering system ran far too slowly to be viable as a real-time solution for this robot and was not investigated further.

For depth perception, the two formulations discussed previously were implemented. For the ideal camera model, the images were iterated over and the formula was applied at each pixel using the required constants. For the reprojection approach, the Q matrix was constructed before iterating through the image and performing the multiplication required. Due to issues with visualizing the point cloud of the 3D points, the norm distance from the origin of our system to these points was used. The left camera was assumed to be the origin of the system, so it was hypothesized that computing the distance of each 3D coordinate from the origin would give the depth.

With the intent of saving computation, the disparity map was divided into a 3x3 grid, and depth was computed on the middle rectangle. This was done also to alleviate the poor performance of the depth calculations (see Experiments). Also, for performing stereo calibration, ROS

calibrate_camera.py script was used in order to find the left and right camera matrices that were used by the stereo_image_proc node for rectification[20].

C. *Haar Feature-based Cascade Classifiers for Human Detection*

When starting work on the Haar Feature-based Cascade Classifiers, a simple tutorial was used to test the viability of this approach. The code from this tutorial was modified and inserted into a test script that was provided in the course documents[21]. By simply creating a CascadeClassifier object in OpenCV and calling the detectMultiScale() function on an image, facial recognition was achieved, and a box drawn around the detected face using the rectangle() function. The parameters used for the detectMultiScale() function were as follows: a grayscale image from the video stream as the image input, 1.1 as the scaleFactor, and 3 as the minNeighbors parameter. The scaleFactor determines the steps of scale used for attempting object detection, which is necessary as the model is trained on images of an exact size. A scaleFactor closer to 1 results in a higher chance of detecting the object. The minNeighbors parameter determines how many neighbors an identified object must have to be considered an object, due to how the Haar Cascade Classifier works. This decreases the number of false positives. These values were found to be the most accurate. The frontal face model included with OpenCV was used for this step.

Once it was determined that this approach could conceivably work, research was done into training a model for identifying a standing human facing toward the camera. The pre-included full body models in OpenCV seemed to be trained for CCTV camera footage, which was not the perspective necessary for this project. This model was tested, but found to be inoperable for this application. Next, a couple tutorials were followed in an attempt in training a model, but was unsuccessful in creating a working model[22][23]. Due to this, the model training idea was scrapped, as it would have required more time than was available, and given the remote nature of the project, was deemed unachievable.

A couple other pre-trained models were attempted, including the upper body and lower body included models. However, these also did not yield favorable results. As a result, the idea for simply using face tracking for human identification and tracking was implemented. To do this, the simple face-tracking model was used to identify faces in the video stream, and identify distances and angles using a monocular approach.

D. *Monocular Depth Estimation*

Due to the inaccurate stereopsis results, a monocular approach was used to identify the distance and angle to the human subject from the robot. A simple algorithm was created using the physical layout of the camera sensor and lens.

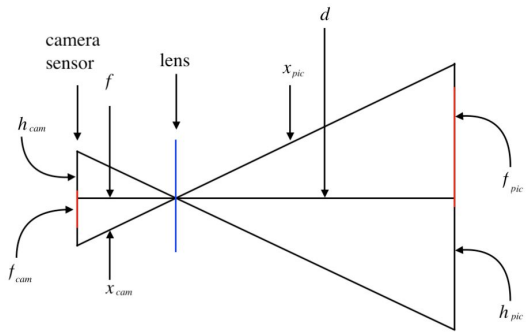

Figure 2: Pinhole Camera Model for Distance Estimation

As shown in the image above, a pinhole camera model was used to estimate distance to the human subject. A few of the variables are known. f is the focal length of the camera, which was found to be 3.04 mm from the camera specifications of the particular camera[24]. f_{pic} is the measured height of the human subject's face, found to be around 205 mm. The height was used instead of the width since the Haar Cascade Classifier tended to put tighter bounds around the top and bottom of the face, probably because the model was trained on square images. h_{cam} is the height of the camera sensor, equal to 3.67 mm, which was also found from the camera specifications page. Using these values, simple geometry can be used to calculate the estimated distance from the camera. When a frame from the video stream is taken, the box surrounding the detected face is found using the Haar Classifier. Then, using ratios, we can

find the real world height of the image, h_{pic}. Since we know that the ratio of the pixel height of the face to the pixel height of the image is equal to the ratio of the real height of the face to the real height of the image, we can conclude that:

$$h_{pic} = f_{pic} * h_{pix} / f_{pix}$$

where h_{pix} is the pixel height of the image and f_{pix} is the pixel height of the face. Next, the Pythagorean Theorem is used to find that:

$$x_{cam} = \sqrt{(\frac{h_{cam}}{2})^2 + f^2}$$

Because we know that the two triangles are similar, we can find using simple ratios that:

$$x_{pic} = x_{cam} * h_{pic} / h_{cam}$$

Now that we have the side lengths for the hypotenuse and base of the right triangle, can find *d*, the distance to the human object, using the Pythagorean Theorem again:

$$d = \sqrt{(\frac{h_{pic}}{2})^2 - x_{pic}^2}$$

This output distance was in meters. This was coded into a function that accepted the face height in pixels, image height in pixels, sensor height, and real face height as parameters. The predefined coefficients were saved as global variables and used every time the function was called. This allows for calibration to a different camera. The function then performed these calculations using some numpy functions and normal arithmetic.

E. Monocular Angle Estimation

A similar approach was used in finding the angle of the human subject in relation to the robot, as illustrated below:

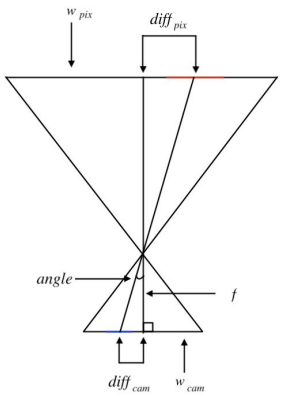

Figure 3: Pinhole Camera Model for Angle Estimation

Once again, we have a few given variables. *f* is still the focal length at 3.04 mm, and w_{cam} is the width of the camera sensor, 2.76 mm, also found from the camera datasheet. $diff_{pix}$ is the offset between the center of the image and the center of the detected face, whereas $diff_{cam}$ represents the physical distance on the camera sensor. $diff_{pix}$ was found by finding the center of the detected face, then subtracting this value from the center of the image. A negative $diff_{pix}$ meant that the subject was to the right of the robot, whereas a positive $diff_{pix}$ meant that the subject was to the left, but when calculating the angle, the absolute value was taken and the sign added back on after finding the angle. Using ratios again yields the following formula:

$$diff_{cam} = w_{cam} * diff_{pix} / w_{pix}$$

This gives us the actual offset from the center on the camera sensor. Using this offset and focal length with some basic trigonometry results in the following:

$$angle = \arcsin(diff_{cam} / f)$$

As stated before, the sign of the angle was again added to the angle before publishing to the robot. The output was in radians. Similar to the distance estimation, these formulae were put into a function that accepted the face recognition box x coordinate, box width, image width, and sensor width as parameters. It then performed these calculations

with a combination of numpy functions and normal python arithmetic.

F. Determining Motor Output

The control_motors.py script was used to control the motors. It subscribes to the depth, distance, and angle information published by other nodes in the system. It also reads ticks detected by the encoder sensors mounted onto each motor. The angle to the human subject is used to calculate the left and right velocities of the robot's wheels. The equations are given below:

$$velocity\_right = w(RADIUS\_OF\_ARC\_TO\_DRIVE + WHEEL\_BASE/2)$$
$$velocity\_left = w(RADIUS\_OF\_ARC\_TO\_DRIVE - WHEEL\_BASE/2)$$

where w is the angle to turn[25]. However, when the angle is zero, the equations above give zero for the wheel velocities. To address this, when the angle is close to zero (within 0.005 radians of 0), the left and right velocities are both manually set to 0.20 meters/second. Next, the distance to the subject is used to scale the velocity values (i.e. larger distance should correspond to higher velocities and vice versa). If the distance is sufficiently close to zero (within 0.005 meters of 0), the motors are stopped. Additionally, the depth information is also included, though weighted much less heavily due to its imprecision (see Experiments). Since low depth values should correspond to higher velocities, the depths were inverted, multiplied by a scale factor, and then subtracted from the velocities. Finally, the velocities for each wheel were converted to a target number of ticks per interval.

The PID loop has each iteration run for 0.8 seconds, during which interval the number of ticks read from the encoders are tracked using the GPIO.add_event_callback() function provided by the RPi.GPIO Python module. Pulse width modulation was used to control the speed of the robots.

Experiments

A. Robot Setup

A Raspberry Pi 4 was used as the microcontroller for this robot. The L298N motor driver was used to allow the Pi to interface with the motors. The bottom clear acrylic frame, wheels, and motors were sourced together in a kit. A top platform was cut from an acrylic sheet. Speed encoders are mounted onto each motor. Waveshare photointerrupter sensors are mounted directly above the speed encoders in order to record each rotation tick. The GPIO pins on the Pi were connected to both the L298N board as well as the photointerrupter sensors, utilizing a small breadboard to facilitate these connections. The Omars 10000 mAh power bank was used to power the Pi, providing 5V at 3A. A 5-AA battery pack was connected to the L298N to power the motors. The Logitech C270 and Raspberry Pi Camera V2 webcams were used on this robot. Images of the robot are included below.

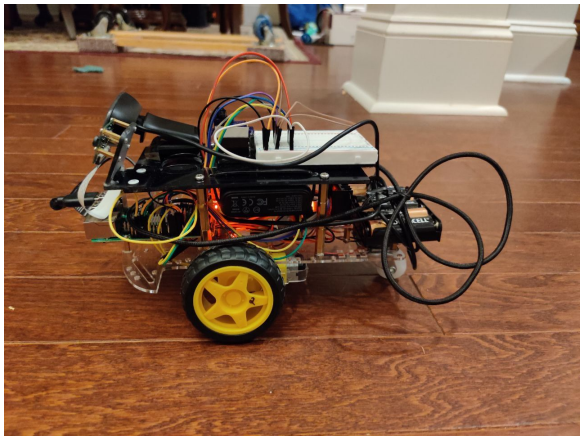

Figure 4: Robot Side Profile

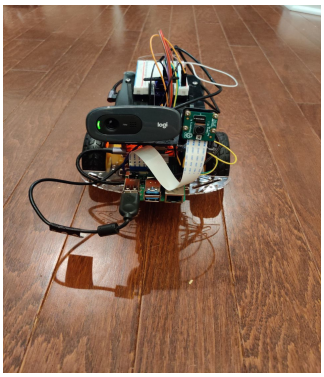

Figure 5: Robot Front View

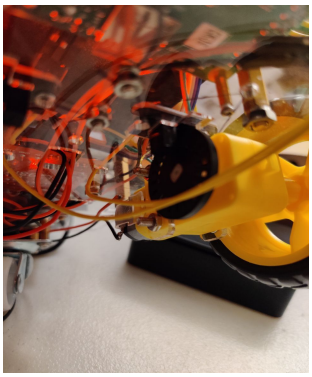

Figure 6: Motor

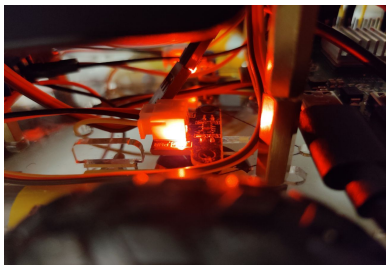

Figure 7: Photointerrupter sensor

B. Stereopsis Results

The stereo_image_proc node published rectified left and right images, of which an example pair is given below:

Figure 8: Raspberry Pi Left Rectified Image

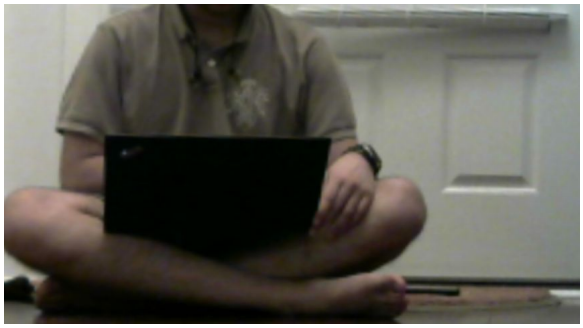

Figure 9: Logitech C270 Right Rectified Image

Though not apparent in the above images, the rectified image for the Raspberry Pi Camera V2 is significantly zoomed in that the raw version. This is because the focal length of the two cameras used for this robot were significantly different. The Logitech C270 webcam was set to be zoomed in more than the Raspberry Pi camera.

Next, the StereoBM and StereoSGBM block matching algorithms were run on the robot and an example pair of the results are given below.

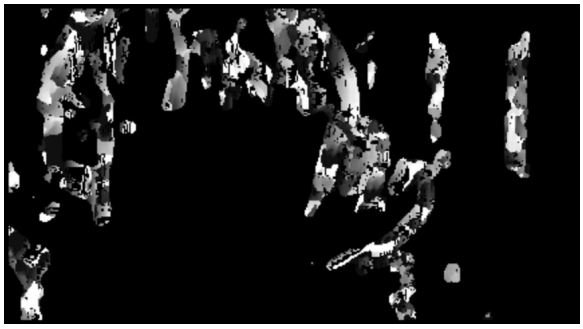

Figure 10: StereoBM Result

Figure 11: StereoSGBM Result

It is apparent the numerous black regions in the disparity map where the algorithms are unable to find matching regions and thus result in holes. For example, in the person's lab is a laptop that is completely devoid in the disparity maps. Additionally, the white door in the background is also completely black, suggesting that the algorithms were unable to compute disparities at these locations as well. Additionally, as expected, the StereoBM algorithm ran much quicker than the StereoSGBM and was able to display the disparity results in real time. However, the StereoSGBM algorithm was very slow on the Raspberry Pi. However, surprisingly, the results from the StereoBM algorithm were more representative of the scene compared to the disparity map from StereoSGBM. Also, the lost visual region on either side of the image (black bars) are much larger in the StereoSGBM disparity map. Neither version was particularly performant, largely due to the difference in focal lengths in the webcams and imprecise mounting (with the hardware available, the cameras could not be mounted perfectly so the lens aligned horizontally).

Next, the filtering techniques were tested. Below, the results for the Gaussian blur and box filter of the disparity map are shown.

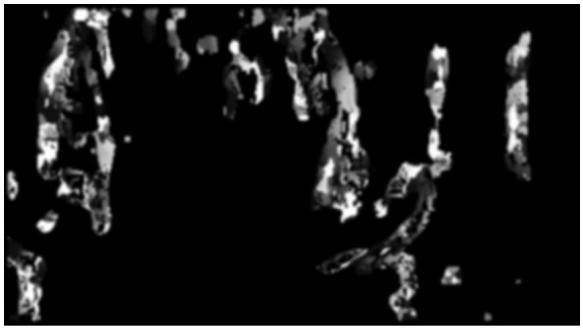

Figure 12: Gaussian Blur

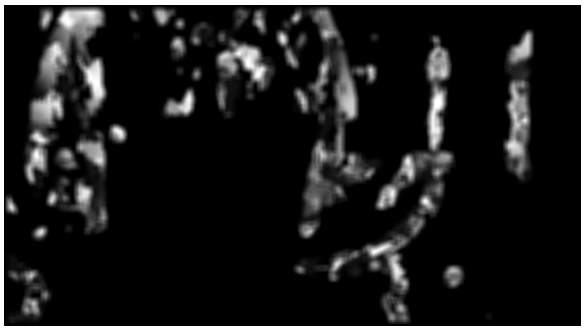

Figure 13: Box Filter

Here we see that the filtering does smooth the disparity maps a bit, especially the box filter. However, they do little towards significantly filling the holes in the map. In the case where regions are actually very far away (and thus have disparities very close to zero), such filtering could be useful to make the transitions smoother. It does appear that though the box filtering seems smoother, the Gaussian filter retains better the contours in the disparity map.

Next, the results of morphological close and openings were retrieved. The resulting disparity maps are displayed below.

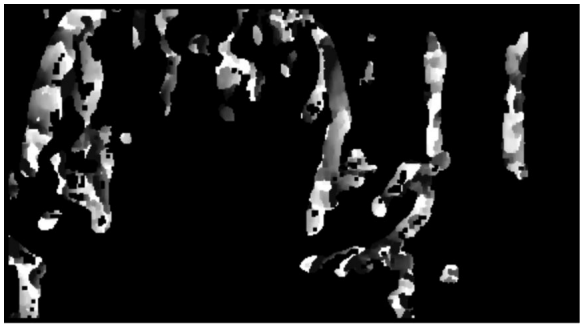

Figure 14: Morphological Opening

The morphological opening does reduce some small patches that were present in the original StereoBM results (especially near the top of the image). The morphological close also reduced small patches (not pictured) and was a bit better at closing some of the small holes near the top of the image. However, morphological operations such as these are not very effective in closing large holes like those in the middle of the image.

The mean and median filtering procedures were tested next. The results for an example pair are given below:

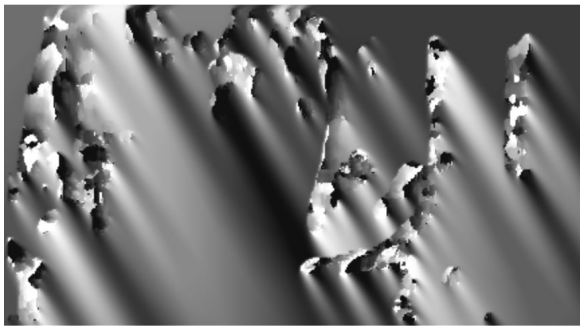

Figure 15: Mean Filtering

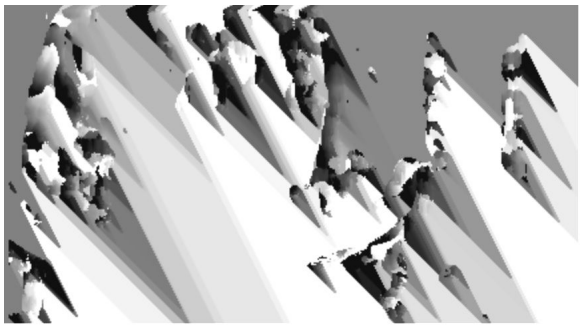

Figure 16: Median Filtering

The both filtering versions fill up the holes in the image. The boundaries of the human subject are also visible in these filtered disparity maps. However, a glaring problem with this method is that it cannot decipher between real holes and holes that need to be filtered. If this was done in a hallway, for example, the robot would be hesitant to move forward since the disparity map fills the void the hallway takes up with nonexistent disparities. The mean filtering approach does seem more representative of the scene, with the holes filled up smoother than in the median filtering. However, these approaches are not aware enough to warrant global use in actual applications.

Finally, the two depth formulations were tested. The results are given below:

Figure 17: Ideal Camera Model

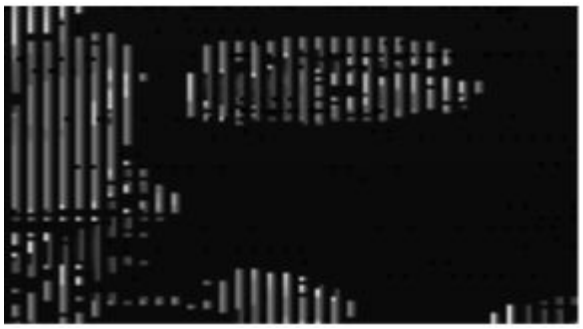

Figure 18: Reprojection Formulation

Clearly, these depth results are not representative of the scene. They correspond to the middle rectangle of the image (when divided into a 3x3 grid), and when matching these to the StereoBM disparity map, we see that the ideal camera model depth results actually do match with the disparity map. The reprojection formulation has some features that seem to appear in the original image, but is not nearly as clear. As a result, the camera setup on this robot does not generate accurate stereo results. This is likely due to the dissimilar camera hardware used. Additionally, when calibrating the camera using ROS camera calibration procedure, the epipolar error was approximately 3 pixels. This is relatively high (when ~0.1 pixels is considered good), suggesting that perhaps a more refined camera calibration procedure would be helpful.

C. Distance and Angle Estimation Results

Testing was done in two phases for the distance and angle estimation functions. First, they were added into a simple test program that performed the calculations in real time. This code was taken from the sample code given on Brightspace. The following image is an example of one frame of results:

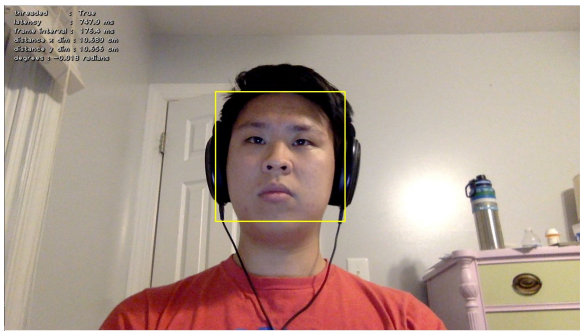

Figure 19: Sample Testing Output

The results of the test revealed reasonable results. Accuracy was not expected at this phase, since the camera specifications of this particular test camera were unknown. All that was tested was whether the program appeared to function correctly, e.g. the angle equalling zero at the center of the image, distance increasing and decreasing with real world distance, etc. Once this was satisfied, the functions were reprogrammed to communicate with the robot using ROS, and the changes were pushed to a repository and tested on the robot.

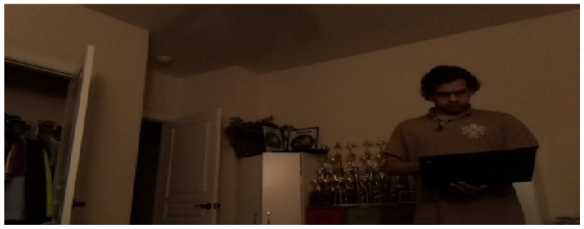

Figure 20: View of Subject From Robot

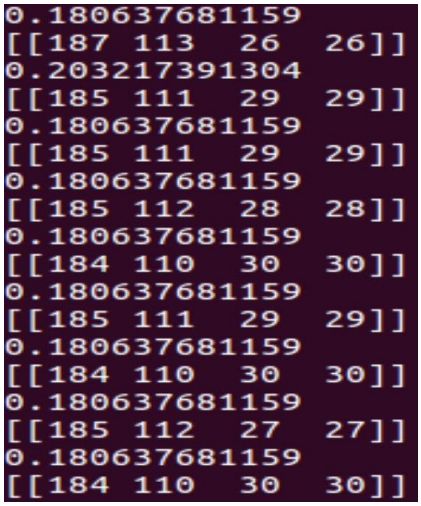

Figure 21: Example Distance Test Results

As shown in the test results, the distance function was found not to be accurate, unfortunately. The distance function appeared to cap out at distances of about 180 cm. This could have been due to the aperture and distortion of the camera lens, as well as the inaccuracies of the pinhole camera model. An undistorted image could have lowered some of these inaccuracies, but given the scale of the inaccuracy, a more drastic change would be needed for accurate estimation of distance. Another possibility would be that at smaller sizes, the facial recognition model did

not perform as accurately, due to the size it was trained on. However, the function did calculate relative size fairly accurately. When the subject moved closer to the camera, the distance value outputted would decrease, and increase as the distance value increased. This made it such that the inaccuracies returned from the distance estimation did not affect the overall function of the final robot due to the way the driving code was implemented.

On the other hand, the angle estimation appeared to be fairly accurate, giving reasonable results for different angles. In addition, when testing it on the robot, the robot would spin the correct wheel to turn it in the direction of the detected face. For this reason, it was weighted more in the final robot driving code.

D. Final Test Results

Due to the low height of the robot, the human subject sat on the floor so that they were in the field of view of the camera on the robot. The robot was placed on the ground in front of the subject such that the subject's face was in the camera's view.

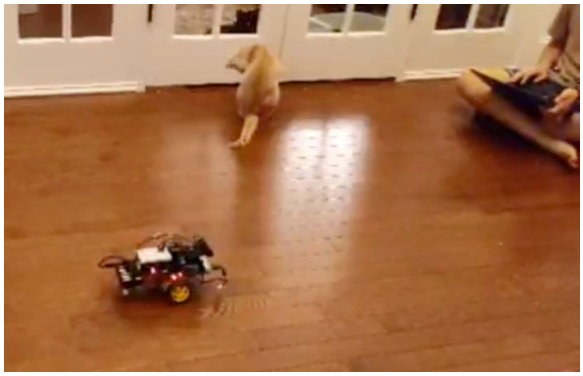

Figure 22: Robot Starting Location

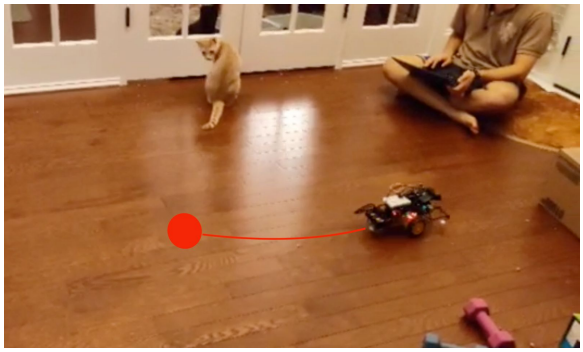

Figure 23: Robot During Movement

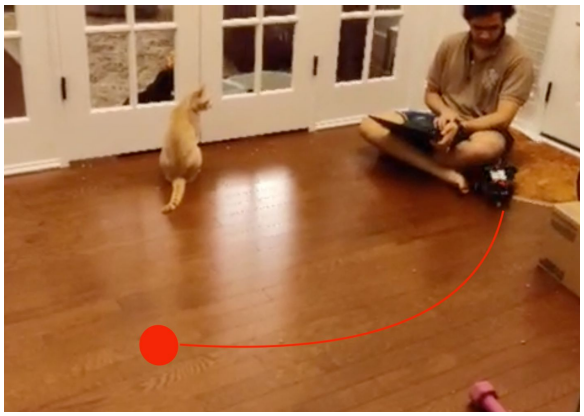

Figure 24: Robot at End of Operation

Depicted in figures 22-24 are an example of a successful run. Although the robot started turning right, due to the uneven weight balance on the wheels, it recognized that the human subject was to the left, and corrected course so that it ended up where the face was detected. However, only about a third of the test runs succeeded, where the robot would reach the subject. In the other third, the robot would turn in the correct direction, but would not reach the subject. In some cases, it would turn too far, missing the target.

The program did not consider where the robot should go if a human face is not detected, so perhaps the robot lost sight of the face and continued turning. The control_motors.py script provides target velocities to the PID loop only when messages are published from the face detection node, so if no face is detected, the previous velocities are retained. Another issue was that the robot wheels could not always gain traction at the beginning of the test. Due to uneven weight distribution and inclined wheels due to low quality parts, the robot would initially start in a random direction depending on which wheel gained traction first. These inclined wheels may have also affected the accuracy of the PID driving code. However, after initially starting off in the wrong direction, the robot would always steer toward the subject, provided that the human subject was still in frame.

As a sanity test, the robot was propped on a box so the motors and wheels could spin without resistance. When running the entire system and standing at different positions, the motors appeared to turn in directions that would cause the robot to turn in

the right directions. For example, standing towards the right of the robot (from the robot's perspective) caused the left wheel to turn faster than the right wheel. As a result, overall, the robot appeared to steer in the correct direction of the subject, but wouldn't always exactly reach the subject.

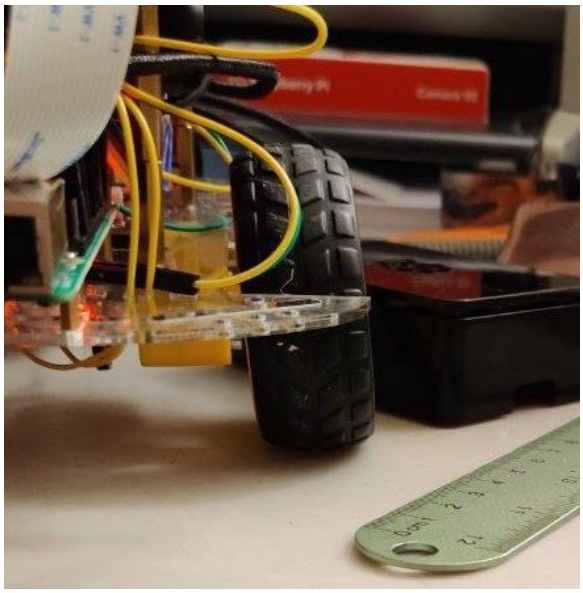

Figure 25: Inclined Wheels

Conclusion

The robot did not perform as well as originally intended, but given that the structure of the project was changed several times after discovering that certain techniques were unfeasible or inoperable, it did perform the original basic aims of the project. Stereopsis was not functional, perhaps due to the extreme difference between the two cameras. A monocular approach achieved satisfactory results given the accuracy of the approach, but stereopsis would have definitely increased the accuracy of the robot. Additionally, better hardware could have helped the motor control script perform better as well. Given more time, a more reliable model could have been trained to more accurately identify a human subject, and use that to more accurately identify location. Overall, despite the restraints of the project, the robot achieved the basic goals of the project.

References

[1] M. E. Ragab and K. H. Wong, "Robot Pose Estimation: A Vertical Stereo Pair Versus a Horizontal One," *International Journal of Computer Science and Information Technology*, vol. 7, no. 5, pp. 19–38, 2015.

[2] S. Singh and B. Digney, "Performance Improvements for Autonomous Cross Country Navigation Using Stereo Vision," Jan. 1999.

[3] "cv::StereoBM Class Reference," *OpenCV*. [Online]. Available: https://docs.opencv.org/3.4/d9/dba/classcv_1_1StereoBM.html#details. [Accessed: 29-Apr-2020].

[4] J. Rambhia and J. Rambhia, "Disparity Map," *Jay Rambhia's Blog*, 29-Mar-2013. [Online]. Available: https://jayrambhia.com/blog/disparity-mpas. [Accessed: 29-Apr-2020].

[5] "cv::StereoSGBM Class Reference," *OpenCV*. [Online]. Available: https://docs.opencv.org/master/d2/d85/classcv_1_1StereoSGBM.html#details. [Accessed: 29-Apr-2020].

[6] "Big different between StereoSGBM and gpu::StereoBM_GPU edit," *Big different between StereoSGBM and gpu::StereoBM_GPU - OpenCV Q&A Forum*. [Online]. Available: https://answers.opencv.org/question/29392/big-different-between-stereosgbm-and-gpustereobm_gpu/. [Accessed: 29-Apr-2020].

[7] "Disparity map post-filtering," *OpenCV*. [Online]. Available: https://docs.opencv.org/master/d3/d14/tutorial_ximgproc_disparity_filtering.html. [Accessed: 29-Apr-2020].

[8] "Computer Vision Emphasis on Filtering & Depth Map Occlusion Filling CS 611 Fall 2012," *Brian Hudson*. [Online]. Available: https://people.clarkson.edu/~hudsonb/courses/cs611/. [Accessed: 29-Apr-2020].

[9] R. A. Peters, "Lecture Notes: Stereopsis (Coplanar).

[10] "Camera Calibration and 3D Reconstruction," *Camera Calibration and 3D Reconstruction - OpenCV 2.4.13.7 documentation*. [Online]. Available: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#reprojectimageto3d. [Accessed: 29-Apr-2020].

[11] "reprojectImageTo3D() in OpenCV," *Stack Overflow*, 01-Dec-1963. [Online]. Available: https://stackoverflow.com/questions/22418846/reprojectimageto3d-in-opencv. [Accessed: 29-Apr-2020].

[12] "Azure Kinect Body Tracking SDK download," *Azure Kinect Body Tracking SDK download | Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/azure/kinect-dk/body-sdk-download. [Accessed: 29-Apr-2020].

[13] "CMU-Perceptual-Computing-Lab/openpose," *GitHub*, 26-Apr-2020. [Online]. Available: https://github.com/CMU-Perceptual-Computing-Lab/openpose. [Accessed: 29-Apr-2020].

[14] M. Vidanapathirana, "Real-time Human Detection in Computer Vision - Part 1," *Medium*, 31-Mar-2018. [Online]. Available:

https://medium.com/@madhawavidanapathirana/https-medium-com-madhawavidanapathirana-real-time-human-detection-in-computer-vision-part-1-2acb851f4e55. [Accessed: 29-Apr-2020].

[15] "Cascade Classification," *Cascade Classification - OpenCV 2.4.13.7 documentation*. [Online]. Available: https://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html. [Accessed: 29-Apr-2020].

[16] "Cascade Classifier Training," *Cascade Classifier Training - OpenCV 2.4.13.7 documentation*. [Online]. Available: https://docs.opencv.org/2.4/doc/user_guide/ug_traincascade.html. [Accessed: 29-Apr-2020].

[17] "Choosing Good Stereo Parameters," *ros.org*. [Online]. Available: http://wiki.ros.org/stereo_image_proc/Tutorials/ChoosingGoodStereoParameters. [Accessed: 29-Apr-2020].

[18] "Smoothing Images," *OpenCV*. [Online]. Available: https://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html. [Accessed: 29-Apr-2020].

[19] "Morphological Transformations," OpenCV. [Online]. Available: https://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html. [Accessed: 29-Apr-2020].

[20] C. Wong, "How to Calibrate a Stereo Camera," ros.org, 28-Jun-2018. [Online]. Available: http://wiki.ros.org/camera_calibration/Tutorials/StereoCalibration. [Accessed: 29-Apr-2020].

[21] T. Khan, "Computer Vision - Detecting objects using Haar Cascade Classifier," Medium, 19-Dec-2019. [Online]. Available: https://towardsdatascience.com/computer-vision-detecting-objects-using-haar-cascade-classifier-4585472829a9. [Accessed: 29-Apr-2020].

[22] T. Ball, "Train Your Own OpenCV Haar Classifier," Coding Robin, 2018. [Online]. Available: https://coding-robin.de/2013/07/22/train-your-own-opencv-haar-classifier.html. [Accessed: 29-Apr-2020].

[23] "Cascade Classifier Training¶," Cascade Classifier Training - OpenCV 2.4.13.7 documentation. [Online]. Available: https://docs.opencv.org/2.4/doc/user_guide/ug_traincascade.html. [Accessed: 29-Apr-2020].

[24] "Camera Module," Camera Module - Raspberry Pi Documentation. [Online]. Available: https://www.raspberrypi.org/documentation/hardware/camera/. [Accessed: 29-Apr-2020].

[25] D. Kohanbash, "Drive Kinematics: Skid Steer & Mecanum (ROS Twist included)," robotsforroboticists.com, 22-Jun-2016. [Online]. Available: http://robotsforroboticists.com/drive-kinematics/. [Accessed: 29-Apr-2020].

Appendix

All source code can be found on Github: https://github.com/arjunkeerthi/point-bot

A. recognition.py

```
import rospy
import numpy as np
import cv2 as cv
from cv_bridge import CvBridge
from sensor_msgs.msg import Image
from geometry_msgs.msg import Point

cv.setUseOptimized(True)

bridge = CvBridge()

sensor_width = 0.00367
sensor_height = 0.00276
face_height = 0.0205
body_height = 1.8

f = 0.00304
body_classifier = cv.CascadeClassifier('haarcascade_frontalface_default.xml')

# Code based on code taken from
https://towardsdatascience.com/computer-vision-detecting-objects-using-haar-cascade-classifier-4585472829a9 with major modifications

pub = None

def recognition(frame):
    # Convert to OpenCV image and grayscale
    left = bridge.imgmsg_to_cv2(frame, desired_encoding='bgr8')
    gray = cv.cvtColor(left, cv.COLOR_BGR2GRAY)
    # Detect faces
    bodies = body_classifier.detectMultiScale(gray, 1.1, 3)
    distance_x = 0
    distance_y = 0
    (x, y, w, h) = bodies[0,:]
    # Draw the bounding box around the face on the frame
    cv.rectangle(left, (x, y), (x + w, y + h), (0, 255, 255), 2)
    image_height, image_width = left.shape[:2]
    # Calculate distance
    distance_y = distance_calculation(h, image_height, sensor_height, face_height)
    # Calculate angle
    angle = angle_calculation(x, w, image_width, sensor_width)

    pub = rospy.Publisher("/distance_heading/distance_angle", Point, queue_size=1)
    msg = Point()
    print("Distance: ", distance_y)
    print("Angle: ", angle)
```

```
    msg.x = distance_y
    msg.y = angle
    pub.publish(msg)

# This function calculates the distance of the subject based on the relative size of
the detected face
def distance_calculation(box_dim, img_dim, sensor_dim, face_dim):
    real_dim = face_dim * (img_dim/box_dim)

    sensor_side = np.sqrt((sensor_dim/2)*(sensor_dim/2) + f*f)

    pic_side = (real_dim/sensor_dim)*sensor_side

    distance = np.sqrt((pic_side*pic_side)-((real_dim/2)*(real_dim/2)))

    return distance

# This function calculates the angle of the subject based on the relative position and
size of the detected face to the center of the image
def angle_calculation(box_coor, box_dim, img_dim, sensor_dim):
    right = True
    center = img_dim / 2
    diff_pix = center - (box_coor + (box_dim/2))
    print("center: ", center)
    print("box_coor: ", box_coor)
    print("box_dim: ", box_dim)
    print("diff_pix: ", diff_pix)
    if diff_pix > 0:
        right = False
    diff_pix = abs(diff_pix)
    diff_real = (float(diff_pix) / img_dim) * sensor_dim
    print("img_dim: ", img_dim)
    print("sensor_dim: ", sensor_dim)
    print("diff_real: ", diff_real)
    heading = np.arcsin(diff_real / f)
    print("heading: ", heading)
    if not right:
        heading = -1 * heading

    return heading

# This communicates with ROS
def listener():
    rospy.init_node('distance_angle', anonymous=True)
    rospy.Subscriber("/stereo/raspicam_node/image", Image, recognition)
    rospy.spin()


if __name__ == '__main__':
    pub = rospy.Publisher("/distance_heading/distance_angle", Point, queue_size=1)
```

```
listener()
```

B. move_cpp.cpp

```
#include <ros/ros.h>
#include <message_filters/subscriber.h>
#include <message_filters/synchronizer.h>
#include <message_filters/sync_policies/approximate_time.h>
#include <image_transport/image_transport.h>
#include <image_transport/subscriber_filter.h>
#include <opencv2/highgui/highgui.hpp>
#include "opencv2/calib3d/calib3d.hpp"
#include <opencv2/imgproc.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/core/utility.hpp>
//#include "opencv2/contrib/contrib.hpp"
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/Image.h>
#include <geometry_msgs/Point.h>
#include <stdio.h>
#include <cmath>

/* Objects for matchers and publisher */
cv::Ptr<cv::StereoBM> sbm;
cv::Ptr<cv::StereoSGBM> sgbm;
image_transport::Publisher disp_pub;
ros::Publisher depth_pub;

void imageCallback(const sensor_msgs::ImageConstPtr& imageL, const
sensor_msgs::ImageConstPtr& imageR) {
   try {

        /* Convert ROS Image messages to OpenCV images, which are then
           converted to grayscale */
        cv::Mat cvImageL = cv_bridge::toCvShare(imageL, "bgr8")->image;
        cv::Mat cvImageR = cv_bridge::toCvShare(imageR, "bgr8")->image;
        cv::imwrite("src/control_robot/left_rectified_image.png", cvImageL);
        cv::imwrite("src/control_robot/right_rectified_image.png", cvImageR);
        cv::Mat grayL, grayR;
        cv::cvtColor(cvImageL, grayL, cv::COLOR_BGR2GRAY);
        cv::cvtColor(cvImageR, grayR, cv::COLOR_BGR2GRAY);
        cv::Rect rect(108, 60, 216, 120);
        cv::Rect rect2(144, 80, 144, 80);
        cv::rectangle(cvImageL, rect, cv::Scalar(0,0,255));
        cv::rectangle(cvImageL, rect2, cv::Scalar(0,0,255));
        cv::imwrite("src/control_robot/left_rectangle.png", cvImageL);
        /* Display left and right rectified images */
        //cv::imshow("left_rect_color", cvImageL);
```

```
        //cv::imshow("right_rect_color", cvImageR);

        /* Run OpenCV's StereoBM block matching algorithm on left and right
           rectified images, then normalize to range 0-255 grayscale (8 bit
           single channel - CV_8U) */
        cv::Mat disp, disp8;
        sbm->compute(grayL, grayR, disp);
        //sgbm->compute(grayL, grayR, disp); /* To run StereoSGBM instead */
        normalize(disp, disp8, 0, 255, CV_MINMAX, CV_8U);

        /* Perform Gaussian blur on disparity map g_iter times */
        /*int g_iter = 1;
        for(int i = 0; i < g_iter; i++) {
            cv::GaussianBlur(disp8, disp8, cv::Size(5,5), 0);
        }*/


        /* Perform grayscale morphological opening on disparity map */
        /*cv::Mat struct_element = cv::getStructuringElement(cv::MORPH_RECT,
cv::Size(3,3));
        cv::morphologyEx(disp8, disp8, cv::MORPH_OPEN, struct_element); */

        /* Get total number of pixels in disparity map */
        int total_px = disp8.rows * disp8.cols;


        /* Run either masked zero elimination mean filtering or
           masked zero elimination median filtering
           Reference: https://people.clarkson.edu/~hudsonb/courses/cs611 */
        /*int iter_count = 0;
        while(cv::countNonZero(disp8) < total_px && iter_count < 20) {
        for(int r = 0; r < disp8.rows; r++) {
            for(int c = 0; c < disp8.cols; c++) {
                if(disp8.at<uchar>(r,c) == 0) {
                    //disp<uchar>.at(r,c) = 1;
                    int r1 = r-2 < 0 ? 0 : r-2;
                    int r2 = r+2 >= disp8.rows ? disp8.rows : r+2;
                    int c1 = c-2 < 0 ? 0 : c-2;
                    int c2 = c+2 >= disp8.cols ? disp8.cols : c+2;
                    cv::Mat A = cv::Mat(disp8, cv::Range(r1, r2), cv::Range(c1, c2));
                    int nz = cv::countNonZero(A);
                    if (nz != 0) {
                        // Mean filtering
                        disp8.at<uchar>(r,c) = cv::sum(A)[0] / nz;

                        //Median filtering
                        //std::vector<uchar> v_flat;
                        //for(int ar = 0; ar < A.rows; ar++) {
                        //    for(int ac = 0; ac < A.cols; ac++) {
                        //        if(A.at<uchar>(ar,ac) != 0) {
```

```
                        //            v_flat.push_back(A.at<uchar>(ar, ac));
                        //        }
                        //    }
                        //}
                        //std::nth_element(v_flat.begin(), v_flat.begin() +
v_flat.size()/2, v_flat.end());
                        //disp8.at<uchar>(r,c) = v_flat[v_flat.size()/2];
                    }
                }
            }
        }
        ++iter_count;
        }*/

        /* Gaussian Blur after mean or median filtering */
        /* int g_iter2 = 1;
        for(int i = 0; i < g_iter2; g++) {
            cv::GaussianBlur(disp8, disp8, cv::Size(5,5), 0);
        }*/

        /* Box filter blur */
        /*for(int i = 0; i < 2; i++) {
            cv::blur(disp8, disp8, cv::Size(5,5));
        }*/

        /* Do grayscale morphological closing on disparity map */
        /*cv::Mat struct_element2 = cv::getStructuringElement(cv::MORPH_RECT,
cv::Size(3,3));
        int morph_close_iter = 1;
        for(int i = 0; i < morph_close_iter; i++) {
            cv::morphologyEx(disp8, disp8, cv::MORPH_CLOSE, struct_element2);
        }*/

        /* Apply jet colormap to disparity before displaying */
        /* cv::applyColorMap(disp8, disp8, cv::COLORMAP_JET);
        cv::imshow("disparity", disp8); */

        cv::imwrite("src/control_robot/disparity_image.png", disp8);

        /* Convert disparity image to ROS Image message and publish to
           "/control_robot/disparity_image" topic */
        sensor_msgs::ImagePtr msg = cv_bridge::CvImage(std_msgs::Header(), "mono8",
disp8).toImageMsg();
        disp_pub.publish(msg);
        /*for(int r = 0; r < disp8.rows; r++) {
            for(int c = 0; c < disp8.cols; c++) {
                disp8.at<float>(r, c) = 1;
            }
        }*/
```

```
        /* TODO: Work on attempts at depth below or move to separate node */

        /* Will only compute depth on middle section (divide disparity map
           into 3x3 grid and compute average depth on middle rectangle) */
        cv::Mat middle = cv::Mat(disp8, cv::Range(80,160), cv::Range(144,288));

        /* Construct Q matrix using camera matrix from camera calibration */
        /*float Q_data[16] = { 1, 0, 0, -214.1453037549374, 0, 1, 0,
-122.8193944709537, 0, 0, -1/0.064, (214.1453037549374-223.5827604021137)/0.064 };
        cv::Mat Q = cv::Mat(4, 4, CV_32F, Q_data);*/

        /* Calculate 3D points for each pixel in disparity map.
           Uses (wx,wy,wz,w) = Q * (c, r, disparity, 1); */
        /*middle.convertTo(middle, CV_32F, 1./16);
        middle.convertTo(middle, CV_32F);
        cv::Mat norm_depth = cv::Mat(middle.rows, middle.cols, CV_32F);
        for(int r = 0; r < middle.rows; r++) {
            for(int c = 0; c < middle.cols; c++) {
                uchar d = middle.at<uchar>(r,c);
                float pts[4] = {c, r, d, 1};
                cv::Mat s = cv::Mat(4,1,CV_32F,pts);
                cv::Mat t = Q * s;
                float t3 = t.at<float>(3,1);
                if(abs(t3) < 0.0001) {
                    if(t3 >= 0) {
                        t3 = 0.0001;
                    } else {
                        t3 = -0.0001;
                    }
                }
                float x = t.at<float>(0,1) / t3;
                float y = t.at<float>(1,1) / t3;
                float z = t.at<float>(2,1) / t3;
                // Currently attempting to just calculate distance of 3D point
                // from origin (assumed to be left camera)
                norm_depth.at<float>(r,c) = sqrt(pow(x,2) + pow(y,2) + pow(z,2));
                //norm_depth.at<float>(r,c) = std::abs(z);
            }
        }
        cv::Mat test_depth;
        normalize(norm_depth, test_depth, 0, 255, CV_MINMAX, CV_8U);
        std::cout << test_depth << std::endl << std::endl;*/

        /* Try to use OpenCV's reprojectTo3D() function - currently giving
           many INF value due to holes in disparity map, hence the above manual
           implementation that deals with case where disparity is close to zero.
           Also reasoning for trying to fill holes in disparity map with
           blurring, morphological operations, and mean/median filtering */
        /*cv::Mat disp32;
```

```
        disp.convertTo(disp32, CV_32F, 1./16);
        cv::Mat output3d(disp32.size(), CV_32FC3);
        cv::reprojectImageTo3D(disp32, output3d, Q);
        cv::Mat test_depth(output3d.rows, output3d.cols, CV_32F);
        cv::Mat ch1, ch2, ch3;
        std::vector<cv::Mat> channels(3);
        cv::split(output3d, channels);
        ch1 = channels[0];
        ch2 = channels[1];
        ch3 = channels[2];
        std::cout << ch1 << std::endl << std::endl;
        cv::pow(ch1, 2, ch1);
        cv::pow(ch2, 2, ch2);
        cv::pow(ch3, 2, ch3);
        cv::Mat norm_depth = cv::Mat(output3d.rows, output3d.cols, CV_32F);
        cv::add(ch1, ch2, norm_depth);
        cv::add(norm_depth, ch3, norm_depth);
        cv::sqrt(norm_depth, norm_depth);
        normalize(norm_depth, norm_depth, 0, 255, CV_MINMAX, CV_8U);*/
        //std::cout << "M = " << std::endl << " " << norm_depth << std::endl <<
std::endl;*/

        /* Another attempt at depth calculation, using
           depth = focal_length * baseline / disparity.
           Fix hole problem by increasing values from 0 to 1 (not good either) */
        middle += 1;
        //std::cout << middle << std::endl << std::endl;
        middle.convertTo(middle, CV_32F);
        float fx = 340.3432526334113;
        float fy = 340.53122558882;
        float b = 0.064;
        cv::Mat test_depth = (fx * b) / middle;
        normalize(test_depth, test_depth, 0, 255, CV_MINMAX, CV_8U);

        /* Display depth map */
        //std::cout << test_depth << std::endl << std::endl;
        //cv::applyColorMap(test_depth, test_depth, cv::COLORMAP_JET);
        cv::imshow("middle_depth", test_depth);
        cv::imwrite("middle_depth.png", test_depth);

        cv::Scalar average = cv::mean(test_depth);
        float avg_depth = average[0];
        std::string s("FAR");
        if(avg_depth < 170 && avg_depth > 130) {
            s = "NEAR";
        } else if (avg_depth <= 130) {
            s = "CLOSE!!!";
        }
        std::cout << s << std::endl;
```

```
        geometry_msgs::Point d;
        d.x = avg_depth;
        depth_pub.publish(d);

        cv::waitKey(30);
    } catch(cv_bridge::Exception& e) {
        ROS_ERROR("Could not convert from '%s' to 'bgr8'.", imageL->encoding.c_str());
    }
}

int main(int argc, char **argv) {
    /* Initialize and start node */
    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;

    /* Create windows for displaying images */
    //cv::namedWindow("left_rect_color");
    //cv::namedWindow("right_rect_color");
    //cv::namedWindow("disparity");
    //cv::namedWindow("depth");
    //cv::startWindowThread();


    /* Initialize parameters for StereoBM block matcher
       Parameters used from: https://jayrambhia.com/blog/disparity-mpas */
    sbm = cv::StereoBM::create(16, 9);
    sbm->setPreFilterCap(61);
    sbm->setPreFilterSize(5);
    sbm->setMinDisparity(-39);
    sbm->setTextureThreshold(507);
    sbm->setUniquenessRatio(0);
    sbm->setSpeckleWindowSize(0);
    sbm->setSpeckleRange(8);
    sbm->setDisp12MaxDiff(1);

    /* Initialize parameters for StereoSGBM matcher.
       Parameters used from: https://jayrambhia.com/blog/disparity-mpas

    minDisparity = -64
    numDisparities = 192
    blockSize = 9
    P1 = 600
    P2 = 2400
    disp12MaxDiff = 10
    preFilterCap = 4
    uniquenessRatio = 1
    speckleWindowSize = 150
    speckleRange = 2
    mode = StereoSGBM::MODE_SGBM
    */
```

```
    //sgbm = cv::StereoSGBM::create(-64, 192, 9, 600, 2400, 10, 4, 1, 150, 2);


    /* Initialize image transport for subscribing to left and right rectified images */
    image_transport::ImageTransport it(nh);
    image_transport::SubscriberFilter subLeft(it, "/stereo/left/image_rect_color", 1);
    image_transport::SubscriberFilter subRight(it, "/stereo/right/image_rect_color",
1);

    /* Publisher to publish disparity map */
    disp_pub = it.advertise("control_robot/disparity_image", 1);
    depth_pub = nh.advertise<geometry_msgs::Point>("control_robot/middle_depth", 1);

    /* Approximate Time Synchronizer is used to get images from left and right
       cameras that approximately have the same timestamp */
    typedef message_filters::sync_policies::ApproximateTime<sensor_msgs::Image,
sensor_msgs::Image> MySyncPolicy;
    message_filters::Synchronizer<MySyncPolicy> sync(MySyncPolicy(10), subLeft,
subRight);

    /* Call imageCallback() on each pair of images recieved */
    sync.registerCallback(boost::bind(&imageCallback, _1, _2));
    ros::spin();

    //cv::destroyAllWindows();
    return 0;
}
```

C. filter_disparity_map.py

```
#!/usr/bin/env python

import rospy
import cv2
import numpy as np
from cv_bridge import CvBridge
from sensor_msgs.msg import Image
from message_filters import ApproximateTimeSynchronizer, Subscriber
cv2.setUseOptimized(True)

bridge = CvBridge()

# Attempt to filter disparity map with WLS Filter (too slow for Pi currently)
def filter_image(imageL, imageR):
   left = bridge.imgmsg_to_cv2(imageL, desired_encoding='mono8')
   right = bridge.imgmsg_to_cv2(imageR, desired_encoding='mono8')

   left_matcher = cv2.StereoBM_create(numDisparities=16, blockSize=9)

   left_matcher.setPreFilterCap(61)
   left_matcher.setPreFilterSize(5)
   left_matcher.setMinDisparity(0)
   left_matcher.setTextureThreshold(507)
   left_matcher.setUniquenessRatio(0)
   left_matcher.setSpeckleWindowSize(10)
   left_matcher.setSpeckleRange(8)
   left_matcher.setDisp12MaxDiff(10)
   """
   wls_filter = cv2.ximgproc.createDisparityWLSFilter(left_matcher)
   right_matcher = cv2.ximgproc.createRightMatcher(left_matcher)
   left_disparities = left_matcher.compute(left, right)
   right_disparities = right_matcher.compute(right, left)
   wls_filter.setLambda(4000.0)
   wls_filter.setSigmaColor(1.0)
   filtered_disp = wls_filter.filter(left_disparities, left,
disparity_map_right=right_disparities)
   filtered_disp = cv2.normalize(src=filtered_disp, dst=filtered_disp, beta=0,
alpha=255, norm_type=cv2.NORM_MINMAX)
   filtered_disp = np.uint8(filtered_disp)
   """
   filtered_disp = left_matcher.compute(left, right).astype('float32')
   filtered_disp *= (255 / np.max(filtered_disp))
   cv2.imshow("filtered_disparities", filtered_disp)
   cv2.waitKey(10)
```

```
def listener():
    rospy.init_node('disparity_listener', anonymous=True)
    left_sub = Subscriber("/stereo/left/image_rect_color", Image)
    right_sub = Subscriber("/stereo/right/image_rect_color", Image)
    ats = ApproximateTimeSynchronizer([left_sub, right_sub], queue_size=10, slop=0.1)
    ats.registerCallback(filter_image)
    #rospy.Subscriber("/control_robot/image", Image, filter_image)
    rospy.spin()


if __name__ == '__main__':
    listener()
```

D. control_motors.py

```
#!/usr/bin/env python

import rospy
#import cv2
import numpy as np
import RPi.GPIO as GPIO
import time
import threading
#from cv_bridge import CvBridge
from geometry_msgs.msg import Point
#from sensor_msgs.msg import Image
#from message_filters import ApproximateTimeSynchronizer, Subscriber

#cv2.setUseOptimized(True)

#bridge = CvBridge()

# Setup
GPIO.setmode(GPIO.BOARD)
IN1 = 15
IN2 = 16
IN3 = 18
IN4 = 22
ENR = 33
ENL = 32
ENCL = 11
ENCR = 13

# Setup pins for motors (1,2 -> right, 3,4 -> left)
GPIO.setup(IN1, GPIO.OUT)
GPIO.setup(IN2, GPIO.OUT)
GPIO.setup(IN3, GPIO.OUT)
GPIO.setup(IN4, GPIO.OUT)
GPIO.setup(ENR, GPIO.OUT)
GPIO.setup(ENL, GPIO.OUT)

# Setup pins for encoders
GPIO.setup(ENCL, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(ENCR, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

# Setup pins for pwm on enable lines
motorR = GPIO.PWM(ENR, 100)
motorR.start(0)
motorL = GPIO.PWM(ENL, 100)
motorL.start(0)

# Store ticks overall and during each interval
left_ticks = 0
```

```
right_ticks = 0
curr_left_ticks = 0
curr_right_ticks = 0

# Robot parameters
diameter = 0.067 # meters
ticks_per_rev = 20
drive_radius = 0.3 # meters
wheel_base = 0.131 # meters
speed_scale_factor = 10
depth_scale_factor = 0.1

# Initially, want robot at rest
target_left_ticks = 0
target_right_ticks = 0

# PID constants
kp = 0.5
kd = 0.2
ki = 0.1

# Interval between correction updates
interval = 0.8 # seconds

# Estimate for depth straight ahead from robot (middle of image from webcam)
# Initialize to some high value
middle_depth = 10000

# Flag to stop robot
stop = False

# Handle for thread
move_thread = None

# Turned
turn = False

# Calculate the left and right wheel velocities in order to turn a given angle
# with a specified radius for the arc driven and wheel base
# Reference: http://robotsforroboticists.com/drive-kinematics
def get_left_velocity(angle):
   return angle * (drive_radius + wheel_base / 2)


def get_right_velocity(angle):
   return angle * (drive_radius - wheel_base / 2)


def velocity_to_ticks(vel):
   return vel * ticks_per_rev * interval / diameter
```

```
def calc_speed(ticks, time):
   return (diameter * ticks) / (ticks_per_rev * time)


# Callbacks to record every tick encoder reads from wheels
def left_encoder_callback(channel):
   global left_ticks, curr_left_ticks
   left_ticks += 1
   curr_left_ticks += 1


def right_encoder_callback(channel):
   global right_ticks, curr_right_ticks
   right_ticks += 1
   curr_right_ticks += 1


def move():
   global curr_left_ticks, curr_right_ticks
   GPIO.add_event_detect(ENCL, GPIO.RISING, callback=left_encoder_callback,
bouncetime=10)
   GPIO.add_event_detect(ENCR, GPIO.RISING, callback=right_encoder_callback,
bouncetime=10)

   # Start the robot at rest
   left_duty = 0
   right_duty = 0
   left_prev_error = 0
   right_prev_error = 0
   left_sum_error = 0
   right_sum_error = 0
   left_end_ticks = 0
   right_end_ticks = 0
   left_adj = 0
   total_time = 0

   #while total_time < 8:
   while stop is False:
       motorL.ChangeDutyCycle(left_duty)
       motorR.ChangeDutyCycle(right_duty)
       curr_left_ticks = 0
       curr_right_ticks = 0

       # Right (forward: 1=low, 2=high)
       GPIO.output(IN1, GPIO.LOW)
       GPIO.output(IN2, GPIO.HIGH)
       GPIO.output(ENR, GPIO.HIGH)
```

```
        # Left (forward: 3=low, 4=high)
        GPIO.output(IN3, GPIO.LOW)
        GPIO.output(IN4, GPIO.HIGH)
        GPIO.output(ENL, GPIO.HIGH)
        time.sleep(0.8)

        # PID correction

        # Compute error
        left_error = target_left_ticks - curr_left_ticks
        right_error = target_right_ticks - curr_right_ticks
        print("curr_left_ticks: ", curr_left_ticks)
        print("curr_right_ticks: ", curr_right_ticks)
        print("curr_left_speed: ", calc_speed(curr_left_ticks, 0.8))
        print("curr_right_speed: ", calc_speed(curr_right_ticks, 0.8))
        print("left_error: ", kp*left_error)
        print("right_error: ", kp*right_error)

        # Add correction
        left_duty = min(max(left_duty + kp*left_error + kd*left_prev_error +
ki*left_sum_error, 0), 100)
        right_duty = min(max(right_duty + kp*right_error + kd*right_prev_error +
ki*right_sum_error, 0), 100)
        print("left_duty: ", left_duty)
        print("right_duty: ", right_duty)
        print()

        # Update errors for derivative and integral terms
        left_prev_error = left_error
        right_prev_error = right_error
        left_sum_error += left_error
        right_sum_error += right_error
        total_time += interval
        #if total_time > 6:
        #    left_end_ticks += curr_left_ticks
        #    right_end_ticks += curr_right_ticks
        #if total_time > 1:
        #    left_adj = 0

    # Stop
    #print("error between wheels: ", abs(left_end_ticks - right_end_ticks))
    print("Total time: ", total_time);
    GPIO.output(ENR, GPIO.LOW)
    GPIO.output(ENL, GPIO.LOW)
    motorR.stop()
    motorL.stop()
    GPIO.cleanup()
    print("Left ticks: ", left_ticks)
    print("Right ticks: ", right_ticks)
```

```
callback_count = 0

def motor_callback(point):
   global target_left_ticks, target_right_ticks, drive_radius, stop, callback_count,
turn

   callback_count += 1
   distance = point.x
   print("Distance received: ", distance)
   angle = point.y if not turn else 0
   print("Angle received: ", angle)

   if callback_count > 5:
       turn = True

   # If we want to continue going straight (angle ~ 0), then just set
   # vel_left = vel_right
   if abs(angle) < 0.008 and distance > 0.02:
       vel_left = 0.2
       vel_right = 0.2
   elif angle < 0:
       vel_right = get_left_velocity(abs(angle))
       vel_left = get_right_velocity(abs(angle))
   else:
       vel_left = get_left_velocity(angle)
       vel_right = get_right_velocity(angle)
   print("Initial vel_left: ", vel_left)
   # If distance to destination is less than 0.02 meters (1 in), stop
   # If distance is greater than 1 meter, cut off to 1 for speed purposes
   distance = min(distance, 1)
   if distance < 0.005:
       stop = True
       distance = 0
       move_thread.join()
   speed_scale = distance * speed_scale_factor

   # Get target velocity adjusted for how far away destination is
   # (i.e. larger distance corresponds to higher speed and vice versa)
   vel_left *= speed_scale
   vel_right *= speed_scale
   print("vel_left after distance scaling: ", vel_left)
   # Adjust for estimated depth ahead of robot, but give smaller weight
   # due to inaccuracy of depth estimation
   scaled_depth = (100 / middle_depth) * depth_scale_factor
   vel_left -= scaled_depth
   vel_right -= scaled_depth

   # Get target ticks for left and right wheels in interval
   target_left_ticks = velocity_to_ticks(vel_left)+15
   target_right_ticks = velocity_to_ticks(vel_right)
```

```
    print("target_vel_left: ", vel_left)
    print("target_vel_right: ", vel_right)
    print("target_left_ticks: ", target_left_ticks)
    print("target_right_ticks: ", target_right_ticks)


def depth_callback(point):
    global middle_depth
    middle_depth = point.x
    print("Depth received: ", middle_depth)


def dummy(point):
    print("In dummy: ", point.x)


def listener():
    rospy.init_node('control_motor_listener', anonymous=True)
    rospy.Subscriber("/distance_heading/distance_angle", Point, motor_callback)
    rospy.Subscriber("/control_robot/middle_depth", Point, depth_callback)
    rospy.spin()


if __name__ == "__main__":
    move_thread = threading.Thread(target=move)
    move_thread.start()
    listener()
    stop = True
    move_thread.join()
    #GPIO.output(ENR, GPIO.LOW)
    #GPIO.output(ENL, GPIO.LOW)
    #motorR.stop()
    #motorL.stop()
    #GPIO.cleanup()
```