BLOCK STACKING ROBOT

Team Members

Abigail Egya-Mensah
Frank Laubach

PROBLEM STATEMENT

The ability to manipulate objects in the environment is an important challenge in the field of robotics. Therefore, this proposal aims to develop a program that enables a robot to stack blocks or create some kind of structure, with the potential to demonstrate the potential of robotics in various industries. More specifically, the goal for this project is to design, develop and evaluate a program that allows a robot to stack blocks, with a focus on using implemented computer vision algorithms for object and pose detection. Two computer-vision algorithms are used in this project: one based on Harris corner detection, and one based on Haar cascades.

This project's objective is to enable the robot to accurately perceive block positions and orientations and stack them in the desired configuration. The program will be tested and evaluated using the Emika Franka arm in the PyBullet simulator, which provides a safe and controlled environment to simulate the robot and the blocks. The initial proposal used the Gazebo simulator, but it was decided to switch to PyBullet after running into issues with out-of-date repositories and MoveIt integration in Gazebo. The code to integrate the Emika Franka arm in PyBullet was sourced from HW6 from the CS 5335 class, with the addition of novel computer vision methods.

Performance will be evaluated through running trials for the project baseline and for Harris Points and Haar Cascade methods. For each trial, 3 blocks will be randomly generated in the workspace. The method will then be run, and return 3 positions and orientations. The arm will then move to grab each block, and stack them in a 3x1 column. Grasp success, detection success, and method time will be measured for each trial, and 20 trials will be run. The successful implementation of this project will contribute to the development of practical applications for robots in various industries, including manufacturing, construction, and logistics.

BLOCK STACKING ROBOT

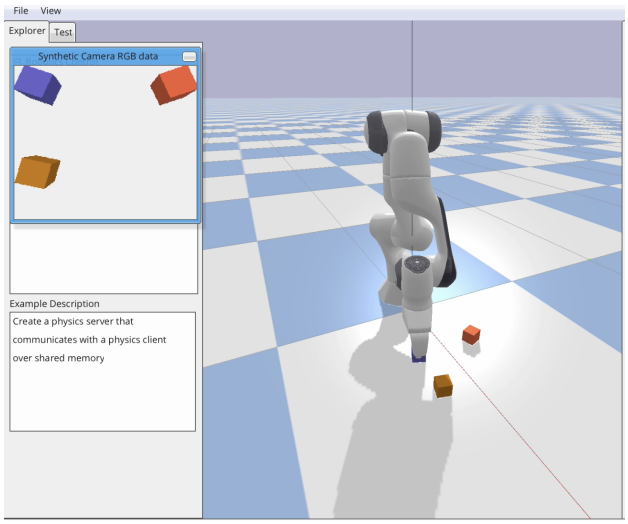
METHOD

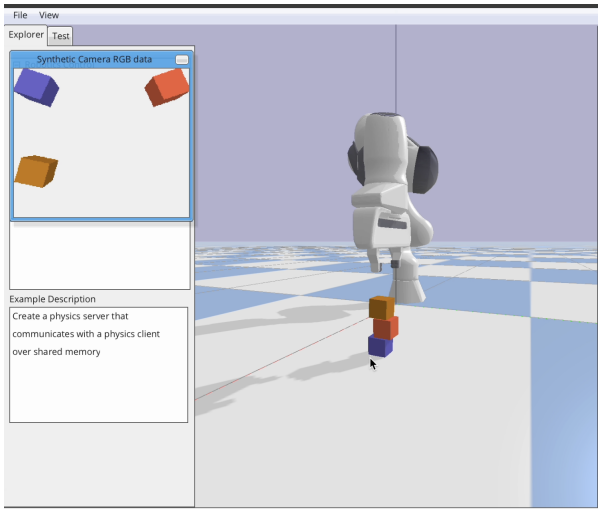
BASELINE

A baseline is a simple and often naive model or approach that is used as a reference point for comparison with other, more complex models or approaches. The goal of using a baseline is to establish a minimum performance that needs to be surpassed for a more complex or sophisticated model to be considered worthwhile. This project's baseline method involves simply spawning the blocks at random locations and orientations, then giving those locations directly to the block-stacking loop.

The pseudocode for this project is as follows:

1. Prepare simulator
2. Spawn three (7cm x 7cm) cubes with random locations and orientations
3. Spawn camera, retrieve an image of the workspace, then implement a method
4. Retrieve coordinates and orientation from the method
5. Run through the block-stacking loop 3 times using the new coordinates
    - The Franka arm will navigate to the new positions and intermediate positions using inverse kinematic methods





Stacking with fixed positions.mov

The 2 other methods implemented in this project can be found below:

- Haar Cascade
- Harris Corner Detection

BLOCK STACKING ROBOT

HAAR CASCADE

Haar Cascade function is usually used for face and eye detection but in this project it was used to detect the blocks in the input image. It works by applying a set of pre-defined Haar features to the image and using them to classify whether a region of the image contains the object of interest or not. It is designed for object detection based on image features, and does not have a **built-in capability for detecting the orientation of objects.** This function was implemented in Python using the OpenCV library. To train the classifier for block detection in the image, the following steps were followed:

First, the positive samples are generated using the opencv_createsamples command. This command takes as input a single positive image (-img parameter), a file containing a list of negative images (-bg parameter), a file to store the sample information (-info parameter), the number of samples to generate (-num parameter), and the dimensions of the samples (-w and -h parameters). In this case, 90 positive samples of size 7x7 are generated from the image 'img22n1.png' and stored in the 'info.txt' file.

```
cmd = "/opt/homebrew/opt/opencv@3/bin/opencv_createsamples -img
/Users/biggles/Documents/Code/Blocks/pos2/img22n1.png -bg negative.txt -info info.txt -num
90 -w 7 -h 7 -vec positive_samples.vec"
os.system(cmd)
```

Next, the classifier is trained using the opencv_traincascade command. This command takes as input the positive samples generated in the previous step (-vec parameter), a file containing a list of negative images (-bg parameter), the number of positive and negative samples to use (-numPos and -numNeg parameters), the number of stages to use in the classifier (-numStages parameter), and the dimensions of the samples (-w and -h parameters). In this case, a classifier is trained using 90 positive and 10 negative samples of size 7x7, and 10 stages are used.

```
cmd = "/opt/homebrew/opt/opencv@3/bin/opencv_traincascade -data classifier -vec
positive_samples.vec -bg negative.txt -numPos 90 -numNeg 10 -numStages 10 -w 7 -h 7"
os.system(cmd)
```

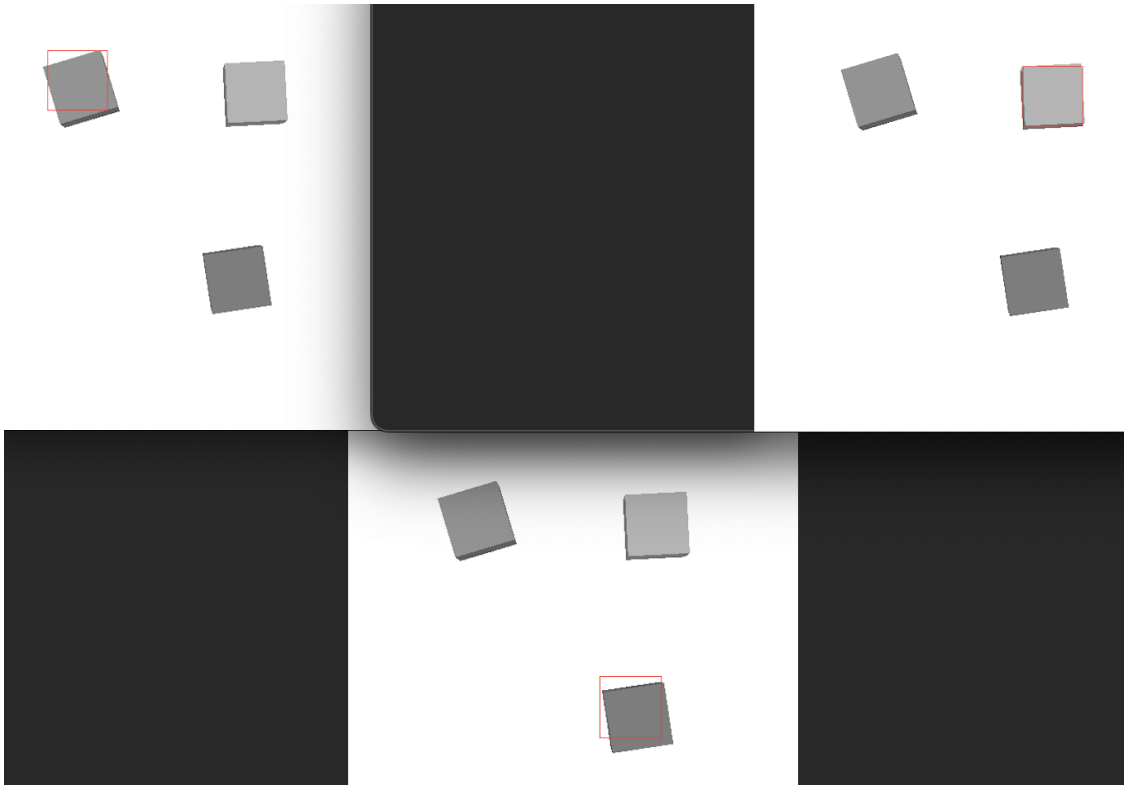
Finally, the trained classifier is tested on a new image using the detectMultiScale method of the CascadeClassifier class. This method takes as input a grayscale image, and several parameters to control the detection process such as the scale factor, the minimum number of neighbors, and the minimum and maximum sizes of the objects to be detected. In this case, the image 'img.png' is loaded and the classifier is used to detect blocks of size between 60x60 and 80x80 pixels.

```
cascade = cv2.CascadeClassifier('classifier/cascade.xml')
img = cv2.imread('img.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

```
blocks = cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(60, 60), maxSize=(80, 80))
```

Once the blocks have been detected using the Haar Cascade classifier, the code calculates the x and y position of the center of each block and converts it to meters based on the provided workspace. The calculated x and y positions are stored in the x_meters and y_meters lists, respectively.

Below is an image of the blocks detected using the Haar function.

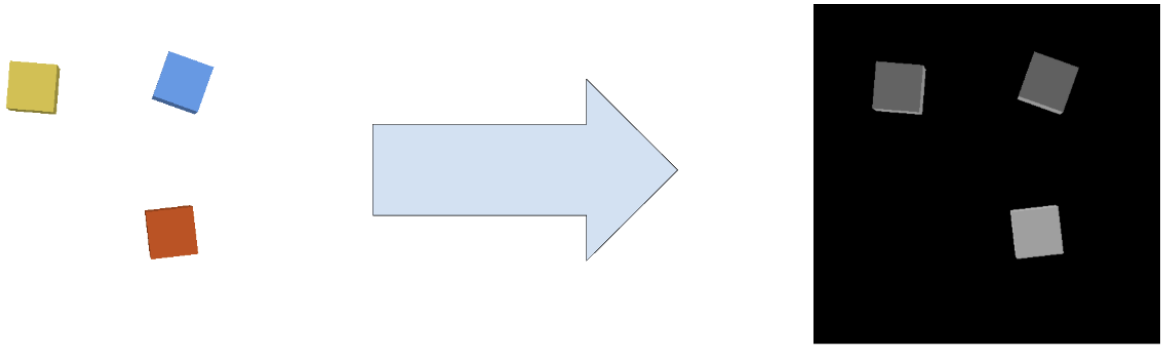


BLOCK STACKING ROBOT

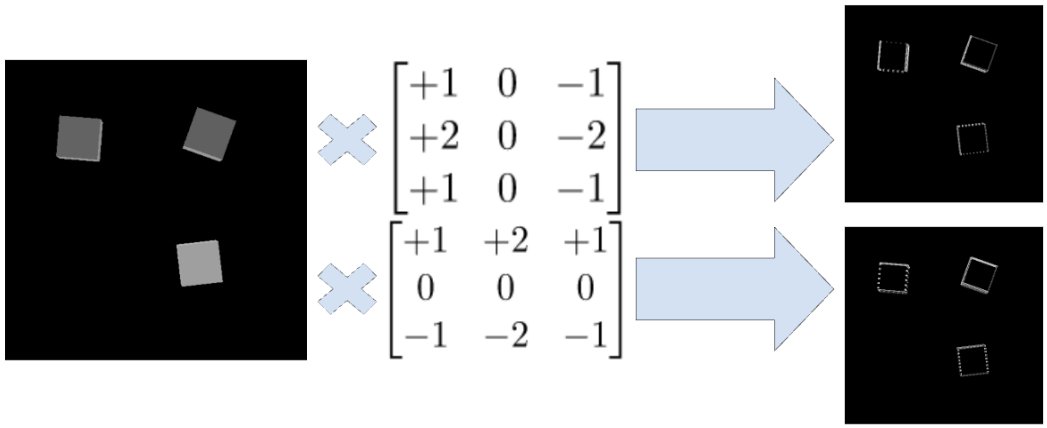
HARRIS CORNER DETECTION

The Harris corner detection algorithm goes through each pixel in an image and compares its value to that of its neighbors. If there is a large difference in both the x and y direction, the algorithm detects a ‘corner’

1. The algorithm is given an image and converts it to grayscale



2. The greyscale image is convolved (with 1 pixel padding) with the sobel operator matrices (directional strength weightings) to give Ix and Iy, which show the strength of change in the x and y directions.



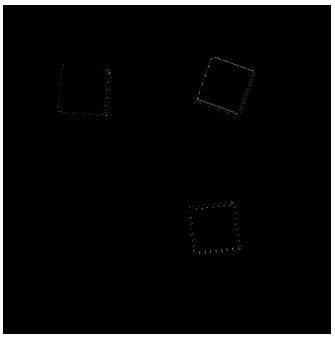
3. Ix and Iy are used to compute the products of derivatives IxIx, IyIy, and IxIy.

4. IxIx, IyIy, and IxIy are used to construct the structure tensor M for each pixel. Then, the determinant and trace of M are used to find the Harris values R (k is a constant).

$$M = \sum_{(x,y) \in W} g(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

$$R = \det(M) - k\,\mathrm{tr}(M)^2$$

5. R values below a certain magnitude are suppressed, resulting in the final Harris corner distribution below.

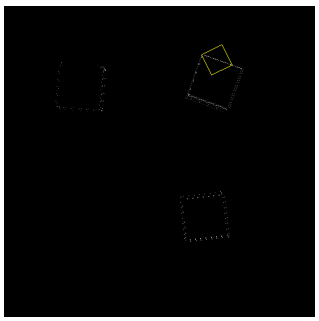


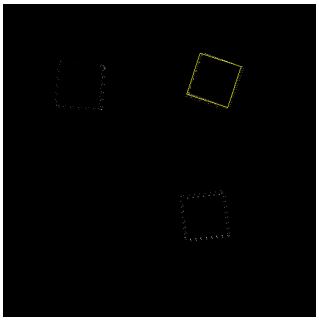
6. In order to find position and orientation, the algorithm will attempt to fit squares onto the collected points. For each point, the algorithm will cycle through all other points, and check the distance between the two. If the other point is within an expected distance (roughly the expected size of the diagonal), the algorithm will proceed.

   From the two points (p1 and p3, on the square's diagonal), two other points (p2 and p4) will be generated to create a square. The centerpoint will also be calculated. The algorithm will again cycle through all other points, and proceed if the test point is within an expected distance to the square's center. 4 lines will be created from the 4 square points, and the lowest distance from the test point to the lines will be recorded. If the distance is below a certain value, a point will be added to the square's 'score'.

7. The algorithm now has a list of possible squares (defined by two points) and each square's 'score' defined by the number of close points. Now the algorithm will find the square with the highest score and extract its coordinates and orientation (from the angle of the diagonal). Before moving on, the algorithm will check for all squares close to the maximum close squares (distance from center to center), and delete them from the squares list. After the close squares are deleted, the square with the next highest score should correspond to another block. The algorithm will go through this cycle three times, and output 3 coordinates and 3 orientations, which will be given to the robot arm.

   This process can be visualized below: the algorithm will favor the potential square on the right over the square on the left because the right square has a higher score. The square on the left will then be deleted due to the closeness to the square on the right, leaving the algorithm free to detect the other two squares in the layout.





BLOCK STACKING ROBOT

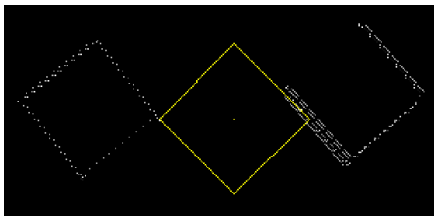
EXPERIMENTS

Performance for each method was evaluated by running 20 trials. For each trial, 3 blocks were randomly generated in the workspace. The method then returned three positions and orientations (or, for the baseline, the positions and orientations were given). Using pre-made inverse kinematic methods, the arm used this information to grab each block and stack them in a column. Grasp success, detection success, and method time was measured for each trial.

	Baseline			Harris Points			Haar Cascades		
Trial #	Successful Detections (/3)	Successful Grasps (/3)	Method Runtime (s)	Successful Detections (/3)	Successful Grasps (/3)	Method Runtime (s)	Successful Detections (/3)	Successful Grasps (/3)	Method Runtime (s)
1	3	3	0.0	2	2	29.3	3	2	0.9
2	3	3	0.0	3	2	44.4	3	2	0.9
3	3	3	0.0	3	3	47.1	3	1	0.7
4	3	3	0.0	3	2	197.8	3	2	0.8
5	3	3	0.0	3	3	121.4	3	2	0.7
6	3	3	0.0	3	2	175.0	3	2	0.8
7	3	3	0.0	3	2	163.7	3	1	0.8
8	3	3	0.0	3	3	86.9	3	1	0.7
9	3	3	0.0	3	3	33.6	3	1	0.5
10	3	3	0.0	3	2	101.4	3	2	0.9
11	3	3	0.0	3	3	50.4	3	2	0.7
12	3	3	0.0	3	3	175.0	3	2	0.6
13	3	3	0.0	3	2	38.7	3	2	0.6
14	3	3	0.0	3	3	72.1	3	2	0.5
15	3	3	0.0	3	2	53.7	3	2	0.6
16	3	3	0.0	3	2	31.1	3	2	0.6
17	3	3	0.0	2	2	197.5	3	2	0.8
18	3	3	0.0	3	3	136.4	3	2	0.2
19	3	3	0.0	3	3	87.6	3	0	0.7
20	3	3	0.0	3	3	107.8	3	1	0.5
Summary:	**100.00%**	**100.00%**	**0.0**	**96.67%**	**83.33%**	**97.5**	**100.00%**	**55.00%**	**0.7**

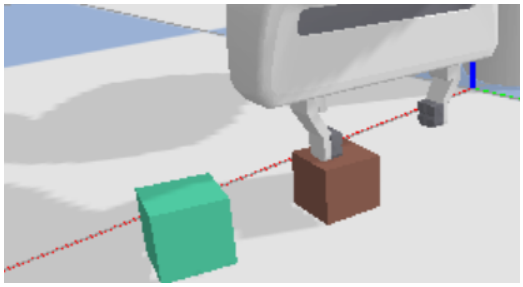
ANALYSIS

The Harris method attained a high detection rate.. The most common mode of failure was caused by the detection of ‘phantom cubes’ formed by points on the edge of two real cubes.



While rare during testing, this mode of failure would be more common in a real environment. In testing, blocks were generated at a good distance apart from each other, and the environment was free of clutter which could lead to more phantom cubes. In a Harris algorithm developed for a realistic environment, a second ‘score’ could be created. The original code fits a square between two points and scores it by counting nearby points, resulting in a list of possible squares. This list could be further scored by counting overlapping squares. A real object, with many points, is likely to generate many overlapping squares, so creating a second score based on overlapping squares could eliminate the phantom object problem.

Interestingly, the Harris algorithm grasp rate was lower than its detection rate. This may be due to problems with the premade inverse kinematics method. While the official trials of the baseline method returned perfect results, during earlier testing it was observed that some grasps failed to execute despite being given perfect data. The arm motion loop sends the gripper to a ‘pre-position’ before executing a grasp. This pre-position was observed to sometimes be slightly offset, which results in the grasper pushing the block into the ground instead of grasping it. This issue can be resolved by refining the inverse-kinematics method. However, since this project focuses more on object detection and orientation methods, successful detections are a more important metric than successful grasps.



The Haar Cascade method showed promising results for detecting blocks in images. Based on the data collected from the experiments, it had a detection rate of 3 for all trials, indicating that the function was able to correctly identify all blocks in the images. However, the grasp rate varied between trials, ranging from 0 to 2 out of 3, indicating that the function was not always able to accurately predict which blocks were able to be grasped.

As seen in the above image, the function is able to detect the location of the image, but isn’t able to feed the robot with an oriented image. This is due to the lack of a built-in capability for detecting the orientation of objects. This can be resolved by combining the Haar Cascade function with other computer vision techniques.

Between the two methods, the Harris method has the edge. The Haar method performs faster and more accurately in terms of detection, but suffers from its inability to detect orientation. The Harris method is more able to execute grasps, and is quite good at detection. With some brushing-up, and maybe some speed optimization by reducing the number of points or running on GPU, the Harris method could compete with the Haar method in terms of detection and speed. Alternatively, the Haar method could be combined with the Harris method: the Haar method provides a location to the Harris method, which then fits square in only that location, and returns an orientation. This would result in an algorithm that executes quickly and precisely, combining the strengths of both methods.

BLOCK STACKING ROBOT

REFERENCES

- https://www.cs.auckland.ac.nz/~m.rezaei/Tutorials/Creating_a_Cascade_of_Haar-Like_Classifiers_Step_by_Step.pdf
- http://uu.diva-portal.org/smash/get/diva2:601707/FULLTEXT01.pdf
- Haar Cascade: Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001.
- https://opencv.org/
- https://pybullet.org/
- https://www.baeldung.com/cs/harris-corner-detection
- Singh, Hanumant. “Lecture 13.” EECE 5554