ECE 383: Final F Report

Justin Havas

December 14, 2018

Duke Community Standard

1. I will not lie, cheat, or steal in my academic endeavors;
2. I will conduct myself honorably in all my endeavors; and
3. I will act if the Standard is compromised.

DCS - Justin Havas

Assignment

Overall Layout of findGrasp() Function

The findGrasp() function that I created can easily be broken down into three stages. The first stage is generating the gradient of the depth image at every pixel, and computing the magnitude and direction of that gradient. This gradient was calculated using the central difference numerical approximation of the derivative. This approximation states that the partial derivative of a function f with respect to a variable z_i in $\mathbf{z}$ can be approximated as:

$$\frac{\partial f}{\partial z_i} = \frac{f(\mathbf{z} + \delta \mathbf{e}_i) - f(\mathbf{z} - \delta \mathbf{e}_i)}{2\delta}$$

where $\mathbf{e}_i$ is all zeroes except for at position i which has a 1. In order to apply this formula to pixels in a depth image, we chose δ to be equal to 1, and plugged in the depth image function as the function f to get the partial derivative in the x-direction and y-direction:

$$\frac{\partial f}{\partial x} = \frac{\text{depth}[y][x+1] - \text{depth}[y][x-1]}{2} \quad \text{and} \quad \frac{\partial f}{\partial y} = \frac{\text{depth}[y-1][x] - \text{depth}[y+1][x]}{2}$$

The ordering of $\text{depth}[y][x+1] - \text{depth}[y][x-1]$ and $\text{depth}[y-1][x] - \text{depth}[y+1][x]$ was determined based off the fact that we wanted to make sure the right and upward directions of the gradient were positive. Once this operation was completed, each pixel would have a gradient $\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix}$. We then computed the norm and direction of the gradient by the following equations

$$||\nabla f|| = \sqrt{\frac{\partial f}{\partial x}^2 + \frac{\partial f}{\partial y}^2} \quad \text{and} \quad \theta = atan2(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x})$$

This led us into the next stage of the findGrasp() function which goes through every pixel and sees if the pixel it is currently analyzing has a "strong enough" gradient magnitude based off a threshold that will be talked about in the Quality of the a Candidate Grasp section of this report. If the gradient magnitude at that pixel is in fact "strong enough", then we must begin the search for another pixel that is no further than the maximum gripper width away that also has a "strong enough" gradient magnitude and a gradient direction θ that is "nearly" pointing in the opposite direction (180°). If we are able to find this pixel then some additional analysis is run such as checking that the path between the two pixels on the screen stays consistently flat, and that the mid point between the two pixels lies on an orange, red, or yellow surface. This means that my implementation of findGrasp() does not pick up the sides of the box as possible candidate grasps, but would if this color check was commented out. There was actually a bit of math that had to be done at this stage as well since potential grasps would be found based off the location of where the grasping surfaces should be placed. This is problematic since the findGrasp() function must return the grasps by the location of the center of the grasp, an orientation of the grasp with zero radians corresponding to the grasping surfaces being parallel to the x-axis, and the width of the grasp. The center of the grasp was determined by the following midpoint formula:

$$\left(\frac{2x + d_x}{2}, \frac{2y - d_y}{2}\right)$$

where x and y correspond to the coordinates of the pixel we are currently on and d_x and d_y correspond to the distance in the x and y direction that lead us to the other "strong enough" pixel. The width of the grasp can trivially be seen as equal to $\sqrt{(d_x^2 + d_y^2)}$. As for the orientation of the grasp, we had to realize that the grasping surfaces, for let's say a horizontal grasp (0°), should be oriented vertically (90°). This led to the following formula for computing the orientation of a grasp:

$$\theta_{grasp} = \left(\frac{\pi}{2} - atan2(d_y/d_x)\right) \% \, \pi$$

where % stands for the modulus operator. This % π became important when comparing angles later on in the scoring function.

The final stage of the findGrasp() function was to score the grasps in order to return the best grasp candidates. This was done by creating a scoring helper function called scoreGrasp() which takes in the entire list of potential grasps described above and creates a list of counts for each grasp. How it works is each grasp is compared to every other grasp in the list, and we count how many other grasps are similar to the current grasp and store this information in the zero index of the count list. Also, when we determine another grasp is similar to the current grasp, we check if this other grasp's coordinates of the grasping surfaces are larger or smaller than the coordinates previously saved. If so, this information is saved in the other indexes of the count list. This allows the current grasp being analyzed to build a range of grasps that it has been matched as similar to. In theory, the grasps in the center of each cube face should have the highest similarity count values and widest ranges saved for grasping surface coordinates. This idea is explained more in the Generating Grasp Candidates section of the report. Once we have computed this list of counts for every grasp in our potential grasps list, we only consider specific grasps that are unique meaning they have a similariy count of zero or grasps that have high similarity counts and different ranges for coordinates of their grasping surfaces. This generates a score list which usually returns to around 20 final grasp candidates. This list of grasps is then passed into a function that gets the N best grasps. How this function works is it will prioritize getting the two highest scoring grasps for each block (which totals to 6 grasps). If N is greater than 6, it will then return the grasps from score list that have the highest similarity counts. Finally, we return these N best grasps from the findGrasp() function.

How do you judge the quality of a candidate grasp? What parameters, such as weights, are used by this estimate?

Determining if a gradient magnitude of a pixel was "strong enough" to be considered an edge was done by printing out the rgb color, and gradient magnitude of every pixel on a line that passed through some blocks. This allowed me to see that most magnitudes tend to be about 0.015 when it detects an edge. Therefore, we used this weight to tell our program when to begin searching for other pixels that had at least the same gradient magnitude. We also determined if two pixels could be combined to form a grasp candidate by seeing if their gradients pointed "nearly" in the opposite direction. We used the same procedure described above to see what the value "nearly" could be and we determined that the difference between the directions of the two gradients should be within 0.2 radians of 3.14.

There were also many other weights used during the scoring function of the grasps and determining the N best grasps that were determined simply by trial and error, and observing which values produced the most consistent results.

Analyze how certain your scoring function correlates with the physical 3D geometry of a gripper and a potential object to grasp.

My scoring function does a pretty decent job with finding potential objects to grasp. However, it is not the part of the code that determines what grip sizes are used in order to correlate with the physical 3D geometry of the gripper. This is done in the second stage in the overall layout of the findGrasp() function which is basically search. Before we begin this search, we call the gripper_max() function which is able to calculate the maximum gripper width and the approximate width of a block in pixels. We use these parameters in the search algorithm to only search for pixels within the maximum gripper length as potential partners to form a grasp. We also only allow potential grasps that are nearly the same width as the block to be added to our potential grasp list.

Now, the method we created for finding the gripper_max does include some strong assumptions. This is because it was written with the purpose of only being executed in the qhigh configuration. This is because the code looks across the region of the image in which we have identified a block that is perpendicular to the camera. Because we hard coded this region that the function is supposed to exclusively look at, we decided that it was safer to define the gripper maximum and block width based off the values we calculated from the qhigh configuration. This means that my function is able to find grasps that correlate with the physical 3D geometry of a gripper only if the configuration is approximately at the same height as the qhigh configuration. However, we would like to reiterate that the gripper_max() function would be able to return the gripper maximum and block width if the region it was being told to look at for a block was modified by the user of the findGrasp() function.

Since we've taken into account the physical 3D geometry of the gripper in an earlier stage of the findGrasp() function, this allows our scoring function to solely work on finding the grasps with the most similarities and making note of that. The scoring function also takes notes of grasps that have a similarity score of zero since this corresponds to grasps that are unique and should be further analyzed.

How do you generate grasp candidates? What parameters, such as the number of samples, or resolution of a grid, are used by this procedure?

Final grasp candidates are chosen based off the number of samples or other grasps that are similar to it. This procedure was briefly described in the Overall Layout of the findGrasp() function, but the idea is pretty simple. A potential grasp that has a very high amount of potential grasps that are similar to it should be a good candidate since we know that we are trying to grasp blocks which have many pixels on a block's flat surface that return possible grasps. An example of this can be seen in the figure on the next page.

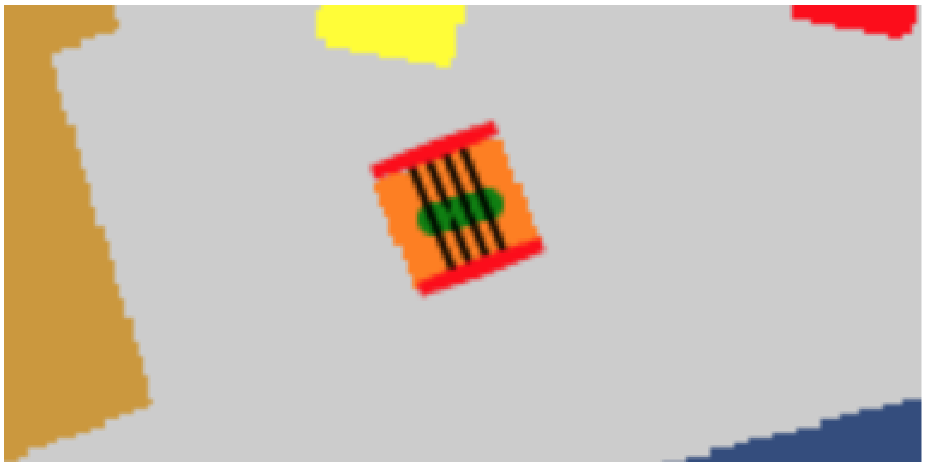

Figure 1: The figure above contains a visual of how many potential grasps can group on sides of a block. The red lines correspond to the grips of the gripper and the green dot corresponds to the center of the grasp.

This concept can be better seen in the figure above. In this figure, there are four candidate grasps all pretty well oriented and positioned to be considered as candidate grasps. The scoring function would go through these four grasps and recognize that each of their counts is 3 since there are three other grasps that are similar to it. Since we can't use similarity counts as a tie breaker inside the score function, we will then go through the ranges of each one and notice that the grasps in the center will have the widest ranges in the x and y direction of the group. This is because the grasps on the ends will have the maximum or minimum coordinates for their group as their own coordinates, so these values will not be saved for their personal ranges. This will cause the center grasps to be chosen instead of the grasps on the outside.

What potential challenges does this pose for a user (e.g., a person working on motion planning) when interpreting your results?

One of the main potential challenges of returning these results in image coordinates is that users must then calculate a transform that will transform these coordinates into camera coordinates and then into world coordinates. This is an extra workload on the user or person working on motion planning because they cannot simply take the image coordinate output that is returned from the findGrasp() function and add them as a constraint to the inverse kinematic objective. Converting from image to camera coordinates is also tricky in the fact that the transform significantly relies on the calibration of the camera to be very accurate. Just a slight difference in the camera calibration would force the motion planner to come up with a new transform.

Unit Testing

One of the main unit tests that I used to test the functionality of each of the three sections we described in the layout (generating gradients, searching for grasps, and scoring) was displaying the grasps that were being returned during each phase. For example, if we wanted to make sure that we had a very diverse set of potential grasps going into the scoring function, we could uncomment the "return graspList" line in the code so that all the potential grasps would be displayed on the screen like so:

```
(1462, 4)
```

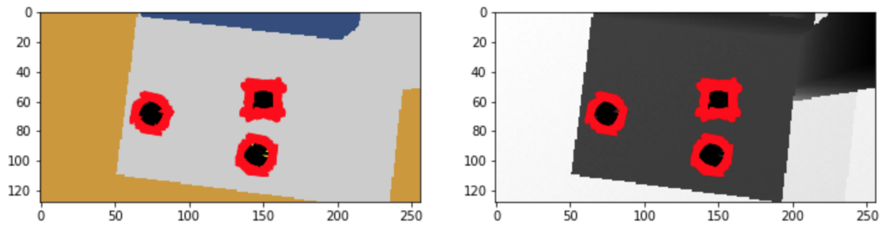

Figure 2: The figure above contains a visual of 1462 potential grasps calculated for this particular configuration.

We could then test the scoring function by seeing if it was able to cut down the number of potential grasps and return candidate grasps like so:

```
(1462, 4)
(17, 3)
```

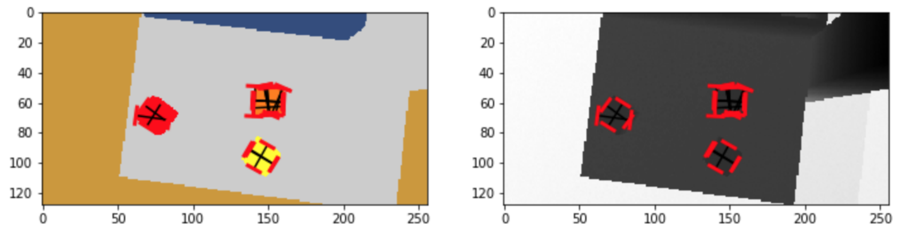

Figure 3: The figure above contains a visual of 17 grasps outputted as the highest scoring in similarity or as the most unique grasps.

Finally, we'd be able to see the end result after the getNBest() function is called:

```
(1462, 4)
(17, 3)
(6, 4)
```

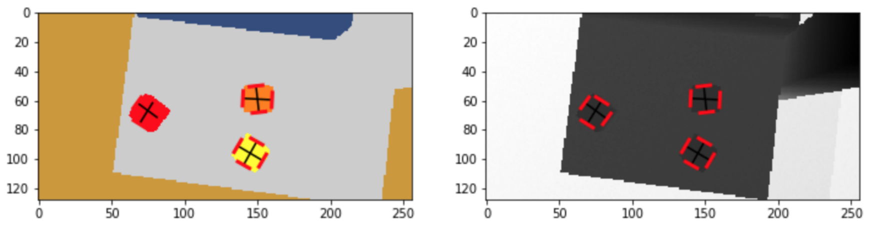

Figure 4: The figure above contains a visual of the 6 best grasps from the score list.

We would then repeat this process for many different configurations, all placing the camera on the gripper to be around same height as the camera in the qhigh configuration.

There were also several unit test functions that we written to test many of the helper functions used during search and scoring. More detail on how they generally worked can be found in the Unit Test Agreeing with Theoretical Behavior section below.

Whether performance is sensitive to certain parameters.

Over the past couple of weeks working on this final, we've come to see that the output of the findGrasp() function is highly sensitive to the weights that we talked about in the Quality of a Candidate Grasp section. This is especially true in the scoring algorithm where everything is dependent on similar grasps being grouped together, so if those parameters are changed around, we may be including or excluding grasps that should or should not have participated in the similarity algorithm.

Whether performance limitations or bottlenecks can be identified.

The main performance bottleneck of my algorithm is the scoring function. This is because it must go through every potential grasp given to it (which is usually in the magnitude of 1000s) and compare it to every other potential grasp in order to compute a similarity score. Once the score for each potential grasp is computed, it loops through each grasp once again determining which ones should be considered for the next stage of the scoring function.

There are many limitations of the algorithm. For example, it determines the maximum grasp length and the block width by observing the change of depth between the grey table and the orange block. Since these blocks are cubes, this change of depth should also be equivalent to the length and width of each face of the block in meters. This function is then able to create a conversion factor that allows us to take in the known maximum gripper length (10cm) and convert that distance into pixel size. However, this function is highly dependent on the robot being in the qhigh configuration since it is in this configuration that the camera of UR5 is almost exactly perpendicular to the orange block.

I'd also say that the performance of the function is also dependent on the orientation of the camera. Through observations seen in different configurations, it seems that the algorithm works really well for blocks that are nearly perpendicular to camera. However, blocks that are not as perpendicular struggle to return a wide array of good candidate grasps. We believe this is because blocks that are nearly perpendicular with the camera are able to provide stronger gradients in the depth image when we find an edge, and are thus able to return much more accurate and diverse grasps than objects not as perpendicular to the camera.

Whether unit tests agree with theoretical behavior.

We created unit tests that checked my essential functions that helped determine potential grasps and score them such as the checkFlat(), checkColor(), gripper_max(), and getNBest() functions. We tested them by creating my own much smaller list of grasps and seeing if each function outputted the expected outcomes using print statements.

Creating unit tests for the entire findGrasp() function was much harder to implement. I often would change the configuration of the robot from the qhigh configuration to see how to algorithm typically ran when the robot was in other configurations. For the most part the algorithm seemed to produce consistent outputs. However, theoretically the algorithm should be able to output the best grasps for each block but it often finds very good grasps for one or two of them, while the grasps for the third block seem to be a bit strange. We believe further tuning of the parameters in the algorithms could help reduce these erroneous outputs from the scoring function.

If you had access to a physical system for testing grasps, how would your performance metrics and test procedure change? How would you be able to use data from testing to improve your grasp detector?

If having access to a physical system, our test procedures would change by including some sort of calibration between the camera and gripper that would also be able to determine the region for my gripper_max function to work in. We'd be able to use data from testing in order to implement a machine learning algorithm that uses techniques such as the Newton's method in order to minimize the error of the scoring function. This calibration would help improve the overall performance of the findGrasp() function by tuning the various weights and parameters in my scoring algorithm so that the error function is minimized. Having access to a physical system would also be helpful by allowing me to see if orienting all the blocks so that they are more perpendicular with the camera improves performance. This would be much easier then moving the robot to a configuration that makes the camera more perpendicular with the blocks.