CS 3630 – Assignment 6

Michael Sobrepera

Alexa Siu

16-April-15

Image Filtering

As a first step the bottom and sides of the grayscale image, captured by the robot, were cropped out. Because the target is above the floor level, and the robot keeps the square centered, this resulted in no data being lost, while removing areas of potential noise. A convolution was then performed using a Gaussian kernel of size 21x21 in order to diminish noise within the image. The image was then converted to a binary image, using a threshold, further reducing the noise level. The Canny edge detector algorithm was then used to highlight edges. A modified version of RANSAC was then utilized to identify lines within the image. In this modified version of RANSAC, any point which was within the tolerance required to vote for a line was removed, given the line obtained sufficient votes to be used. If a line did not obtain a sufficient number of votes to be used, then its points were not removed from consideration. In order to handle varying distances, the RANSAC algorithm was run with an initially high number of votes required. If the algorithm failed to produce the four edges needed to perform further analysis, then the threshold was lowered and the algorithm run again, with all points returned to consideration. This was repeated until the threshold was sufficiently low as to be no longer useful. This iterative reduction in the RANSAC threshold value allowed the system to find the target even from a large distance away, while preventing excessive error at close distances.

Once the lines corresponding to the edges of the square were found, the intersections between the lines were found. Only intersections within the image were kept, yielding the corners of the square.

In order to aid in validation, the lines and points were plotted to the screen as seen in Figure 1. The Image filtering code can be found in ImageFilt.py.

Figure 1: The original image with the lines output from the RANSAC like algorithm shown in green and the points designating the corners shown in blue, along with the edges output from the Canny edge detector.

Controller

Once image analysis had been performed, the points were passed on to a control system. This code can be found in Controller.py. If not enough points were found and the target had not yet been acquired, then the robot was commanded to move forward, this allowed the robot to find the square when it was at far distances from the target. When too many points were found or when too few points were found, but the target had been acquired at a prior point, the robot was commanded to move backward to allow it to get a better vantage point.

When all four points of the square were successfully identified, the controller looked at three different parameters found in VisualServoing.py:

- The center of the square.
- The ratio of the square's leftmost line over the rightmost line (skew ratio).
- The square width.

The goal was for the robot to face directly at the center of the square at a skew ratio of 1:1 and at a known square width in pixels that corresponded to 1 foot.

In order to prevent the robot from losing the target, prior to every movement, it checked to ensure that the square was near to the center of its frame, if it was not, then it would take a correcting action, by turning. The turn was executed over a rotation proportional to the error between the center of the frame and center of the target.

At a skew ratio of 1, the robot was in the correct plane in relation to the square. If the skew ratio was less than zero then the robot needed to move right. On the other hand, if it was greater than zero then the robot needed to move left. The right and left motions commanded were proportional to the error, i.e. the actual skew ratio minus the desired skew ratio of 1.0, scaled by some constant. At the end of the movement, the robot's rotation was approximately the same as at the beginning of the movement, with respect to a global coordinate frame, with only the position changed.

To place the robot the correct distance from the target, we looked at the width of the horizontal lines in the square. At the target depth of 1 foot, the square's width in pixels was known from knowledge of the square's dimensions (~350 pixels). If the recorded width was greater than this value, then the robot was too close to the square. On the other hand, if it was smaller, then the robot was too far and needed to move forward. Similarly to the right and left motions, the commanded forward and backward motions were proportional to the error.

The cycle of taking pictures and moving was repeated until the robot was within a specified error of the goal. A picture was taken after every movement (rotation, lateral skew correction, or axial distance correction).

```
 go = True
targetFound = False

while go:
    picName = s.takePic()
    points = ImageFilt.getPoints(s.directory+picName)
    if len(points)==4:
        targetFound=True
    if len(points)<4 and not targetFound:
        print('too few points, moving forward:\t%i'%(len(points)))
        s.goForward(.2)
    elif not len(points)==4:
        print('too many points, backing up:\t%i'%(len(points)))
        s.goBack(.2)
    elif visualServoing.getCenter(points)[0]>(100+1280/2):
        print('turning right because center at:\t%f'%visualServoing.getCenter(points)[0])
        s.rotate('right',.0005*abs(visualServoing.getCenter(points)[0]-(1280/2)))
    elif visualServoing.getCenter(points)[0]<(-100+1280/2):
        print('turning left because center at:\t%f'%visualServoing.getCenter(points)[0])
        s.rotate('left',.0005*abs(visualServoing.getCenter(points)[0]-(1280/2)))
    elif visualServoing.getSkewRatio(points)>1.05:
        print('going right because skew ratio is:\t%f'%visualServoing.getSkewRatio(points))
        s.goRight(.7,15*abs(visualServoing.getSkewRatio(points)-1))
    elif visualServoing.getSkewRatio(points)<.95:
        print('going left because skew ratio is:\t%f'%visualServoing.getSkewRatio(points))
        s.goLeft(.7,15*abs(visualServoing.getSkewRatio(points)-1))
    elif visualServoing.getDepth(points)<325:
        print('going forward because depth is:\t%f'%visualServoing.getDepth(points))
        s.goForward(.005*abs(visualServoing.getDepth(points)-350))
    elif visualServoing.getDepth(points)>375:
        print('going back because depth is:\t%f'%visualServoing.getDepth(points))
        s.goBack(.005*abs(visualServoing.getDepth(points)-350))
    else:
        go = False
        print('!!!DOCKED!!!')
```

Results

The robot was able to successfully find the target with a high degree of accuracy and dock. In the event that the robot lost the target, it proved robust at reacquiring the target. Due to the scale based nature of the control system, the robot was able to move quickly when it had a large distance to cover and move with precision when it had a short distance to cover. The motor path for the robot starting two feet away from the target, at a 45 degree angle is shown in Figure 2; the robot was able to locate the target, and move towards it, successfully docking. A more challenging task is presented in Figure 3, where the robot had to start a significant distance from the target. The robot initially tracked

noisy data, but was able to recover and dock with the target. A video can be found under the name AFS-MJS-Assignment6.mp4.

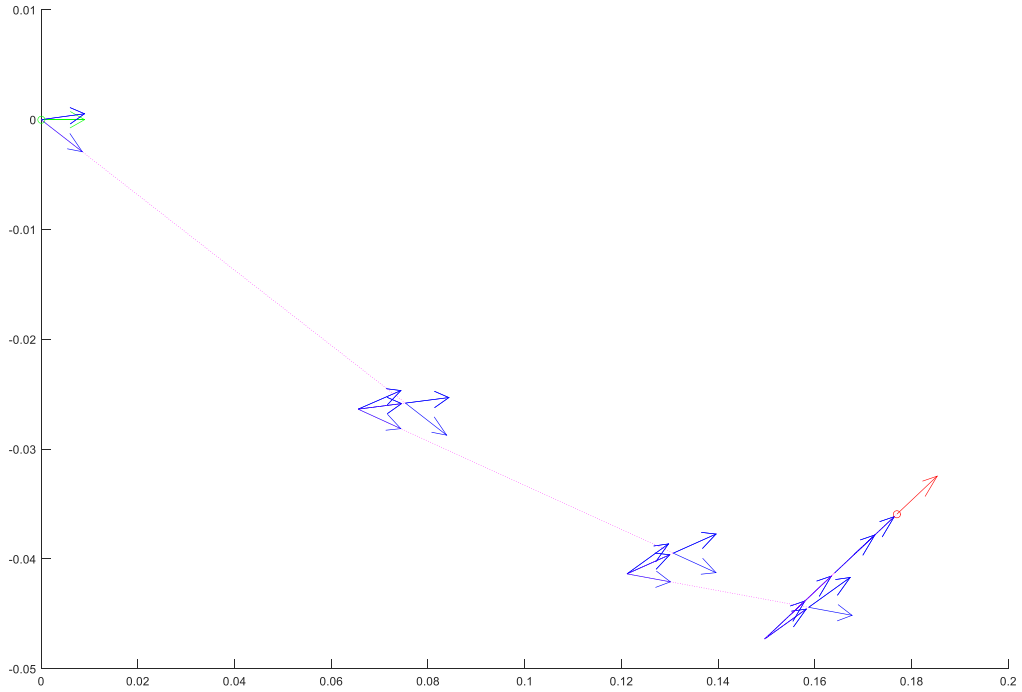

Figure 2: Robot path beginning two feet away from the target at a 45 degree angle from the target normal. Arrows represent the pose of the robot, the green arrow is the starting pose and the red arrow is the ending pose. The magenta line represents the path of the robot. All values are taken from wheel odometry and time data.

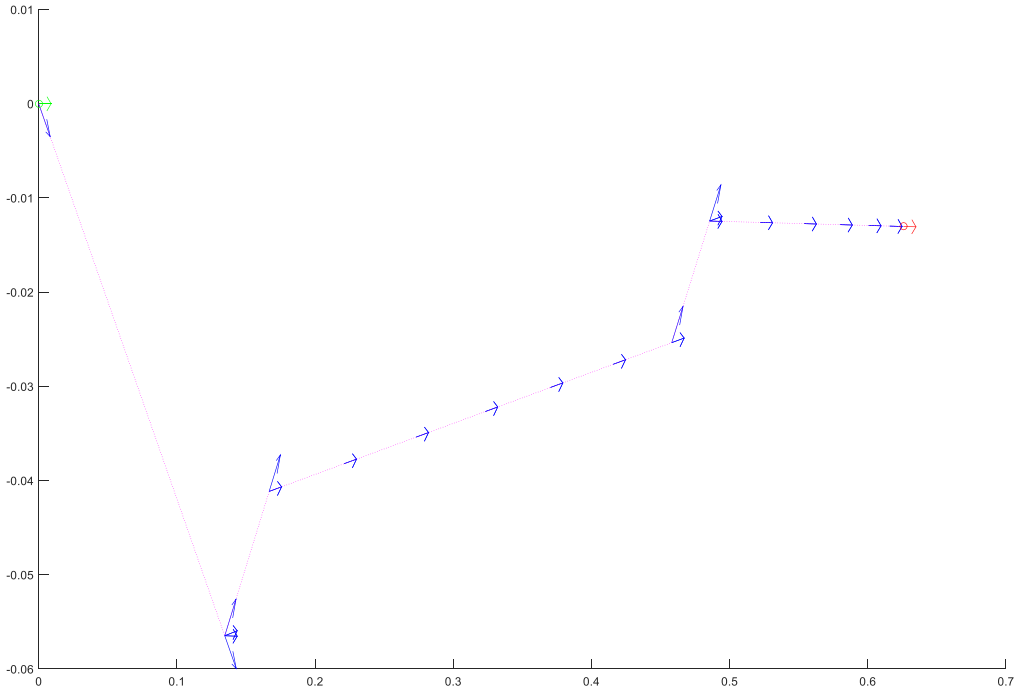

Figure 3: Robot path beginning from approximately 5 feet from the target. It can be seen that initially the robot acted on noisy data, but that it recovered and was able to dock with the target. Arrows represent the pose of the robot, the green arrow is the starting pose and the red arrow is the ending pose. The magenta line represents the path of the robot. All values are taken from wheel odometry and time data.