Homework 1

Taylor Welker

U0778812

taylormaxwelker@gmail.com

I. Edge Extractor

Algorithm Summary:

This algorithms is based off of RANSAC and is used to find fit line segments to edges found with an canny edge detection algorithm. An input image is first processed with the Canny Edge Detector in order to extract the pixels from the image that correspond with edges (between objects in the image). Once the pixels of interest have been found and collected, two points are chosen at random until a pair are found that are within a certain maximum distance from each other. This pair of pixels is used to create a line model (of the form ax + by + c = 0) that we use to fit pixels to the line. Using this model, we iterate through every pixel of interest until we find all of the pixels that fall within a certain distance threshold from the line. If the number of pixels that fit the line are more than a prescribed minimum limit, then the pixels of interest are removed from our set, and a line is added to our new image and labeled with a random color. This process is repeated until we run out of pixels of interest to work with, or until we have reached a maximum iteration value.

Corresponding function files:

extractEdges.m – The function that runs the Edge Extractor algorithm as described above.

Pseudocode

```
Ifinal = extractEdges(inputImg):
Initialize parameters
Take measurements on the dimensions of the input image
iedge = canny_edge_detection(inputImg)
For each pixel in iedge
        if pixel == 1
                Add pixel to EDGE_SET
Get number of pixels in EDGE_SET
while (iterations < MAX) && (Num_pixels_in_EDGE_SET > 0)
        Get a random pixel (using getRandomPixel.m)
        while (give_up_on_2nd_pixel != 0)
                Try to get a second pixel (using getRandomPixel.m)
                % The 'give up' is to help avoid endless loops

        If we didn't give up
                Compute the line model
                Create an INLIER set, and INDICES_TO_REMOVE set
                For each pixel in EDGE_SET
                        est = getLineEst
```

```
                    if abs(est) < MIN_POINTLINE_DIST
                        add pixel to INLIER
                        store pixel index in INDICES_TO_REMOVE
            If (size(INLIER) > MIN_LINE_PIXEL_NUM)
                    Add INLIER to LINE_SET
                    Remove pixels corresponding to INDICES_TO_REMOVE from EDGE_SET

Ifinal = zeroes(size(inputImg)):
For each line in LINE_SET
        Get a random color
        For every pixel in the line from LINE_SET
                Assign the pixel's value to the random color

Return ifinal
```

getLineEst.m – A helper function for the edge extractor. This function simple returns the expected value to fit the line model ax + by + c = 0 given a pixel with a corresponding (x,y) value, and the (a,b,c) values corresponding to the line model. This value is used by the extractEdges.m function to compute how well the pixel in question fits the line model.

Pseudocode

```
est = getLineEst(pixel, a, b, c):
return (pixel(x) * a + pixel(y) * b + c)
```

getRandomPixel.m – A helper function for the edge extractor. This function randomly obtains a pixel value from the EDGE_SET for the extractEdges.m function can use them to find an appropriate line model.

Pseudocode

```
Pixel = getRandomPixel(EDGE_SET):
Get number of pixels in EDGE_SET
Get a random number (between 0 and 1)
Index = randNumber * number_pixels_EDGE_SET
Index = ceil(Index) % This is to make it a whole number
Pixel = EDGE_SET(Index,:)
Return Pixel
```

Results:

After fine-tuning the parameters corresponding with the algorithm (e.g. the maximum distance between two pixels required to make a line model, the maximum distance between a pixel and the line model to indicate a 'good fit', the minimum number of pixels required to create a line, and the sensitivity of the canny edge detector), I was able to obtain several major edges and filter out a little bit of the noise. Obviously, this algorithm is far from perfect, and my results are also, but in each image, you can see significant line trends and tend to not see lines that aren't as prominent. You can see some pairs of close parallel lines will be the same color, because I extended the range at which pixels can fit the line in order to better account for noise in the image. While this is not ideal, I considered it a worthy sacrifice in order to avoid segmenting longer lines.

Figure 1 - lineDetect1.bmp

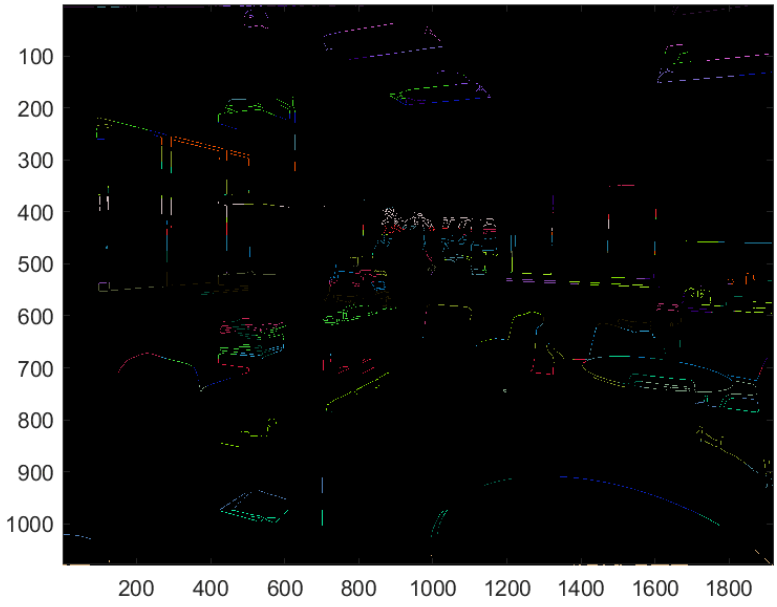

Figure 2 - Output from lineDetect1.bmp

Figure 3 - lineDetect2.bmp

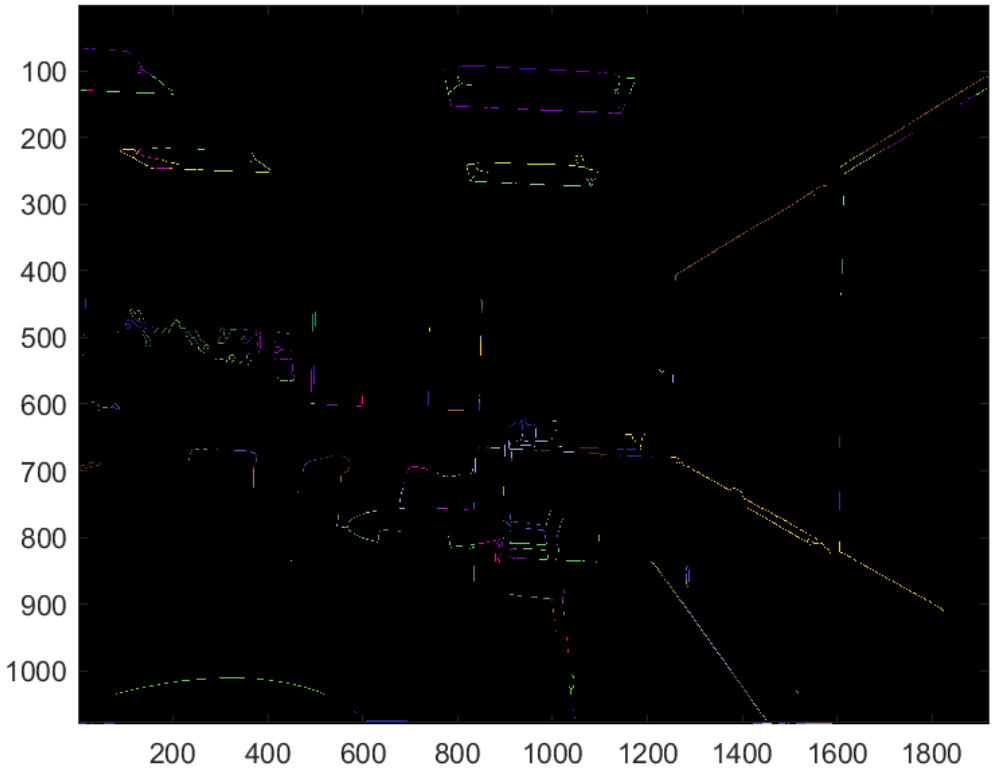

Figure 4 - Output from lineDetect2.bmp

Figure 5 - lineDetect3.bmp

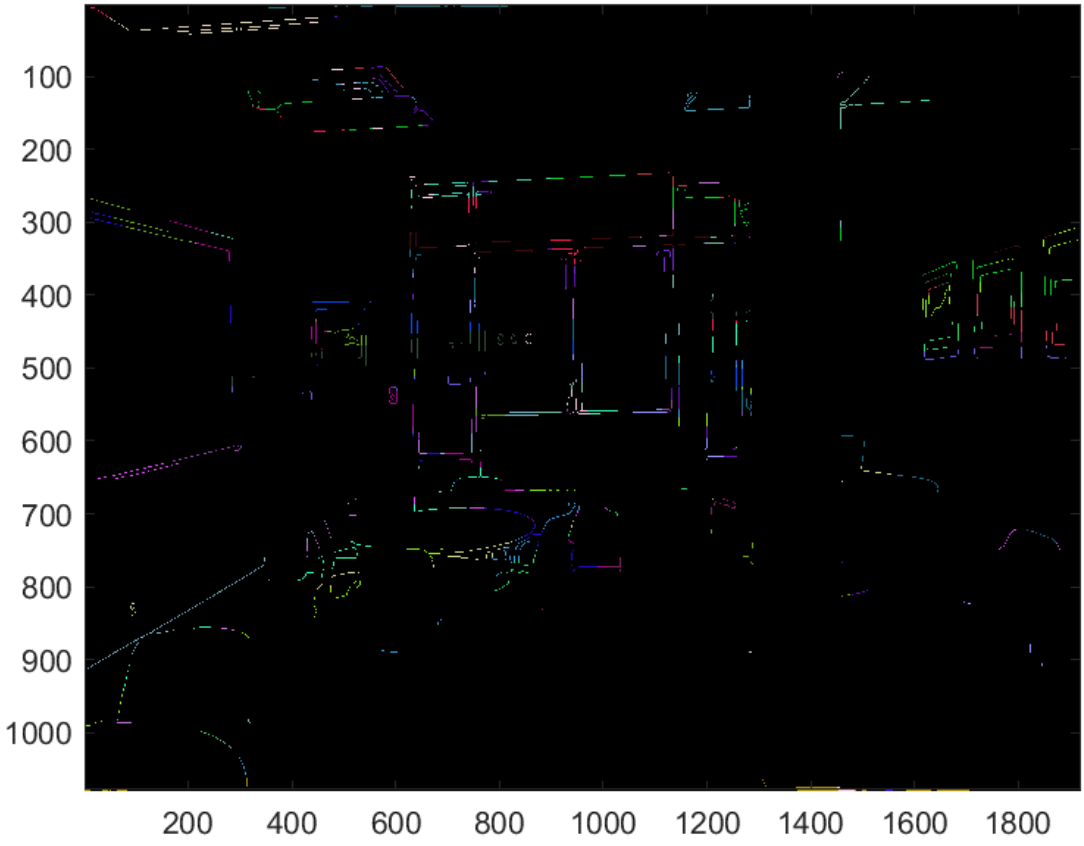

Figure 6 - Output from lineDetect3.bmp

II. Sky Segmentation

Algorithm Summary:

The sky segmentation algorithm is much simpler than the other two. In this algorithm, color thresholds are set; a maximum and minimum integer value for each RGB value that can correspond to a sky-colored pixel. We simply iterate through each pixel of a given input image, and check to see if each of its RGB values fall within the corresponding maximum and minimum threshold values. If all three lie within the thresholds, then the pixel is considered to be a "sky pixel", and is assigned the color white (255,255,255) on the output image. Any pixel that has and R, G, or B value that lies outside of its threshold is considered to be a 'non-sky' pixel, and is assigned the value of the color black (0,0,0).

Corresponding function files:

skySegmentation.m – The function that runs the sky segmentation algorithm described below.

Pseudocode

```
output = skySegmentation(inputImg, imageFilterSelction):
Convert inputImg to an imageMatrix
If(imageFilterSelection == 1)
        Use the RBG thresholds for detectSky1.bmp
Elseif(imageFilterSelection == 2)
        Use the RGB thresholds for detectSky2.bmp
Else
        Use the RGB thresholds for detectSky3.bmp
Get the dimensions of the image
Create another image to be the output and set every pixel value to (0,0,0)
For each pixel in the inputImg
        Get its RGB values
        Create 3 flags, one for red, one for green, and one for blue
        If pixel's red value is within R_MIN and R_MAX
                Set red flag
        If pixel's green value is within G_MIN and G_MAX
                Set green flag
        If pixel's blue value is within B_MIN and B_MAX
                Set blue flag
        If all three flags are set
                Set the corresponding pixel in the output image to be (255,255,255)

Return the output image
```

Results:

I had to fine tune the RGB thresholds for each individual image in order to maximize the accuracy of the algorithm. Therefore, other images that are input into the algorithm may return subpar results. However, I was able to clearly identify the sky in each image, and only suffered a little bit of noise coming from the windows of building that reflected the sky or really bright lighting due to the sun hitting certain buildings directly.

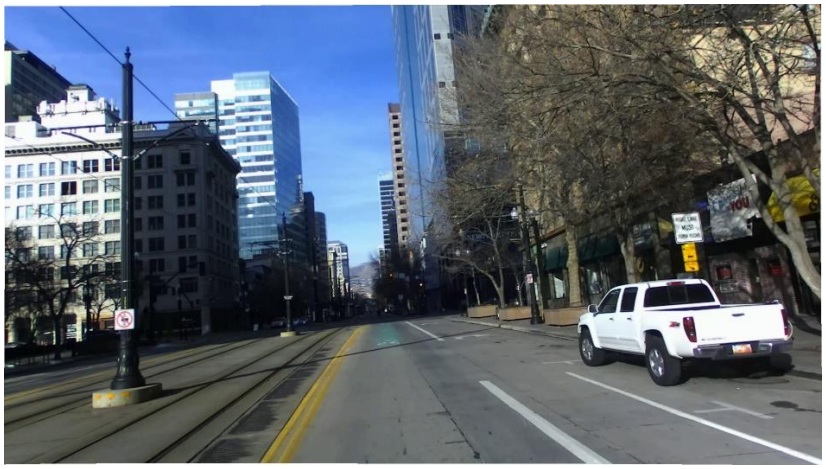

Figure 7 - detectSky1.bmp

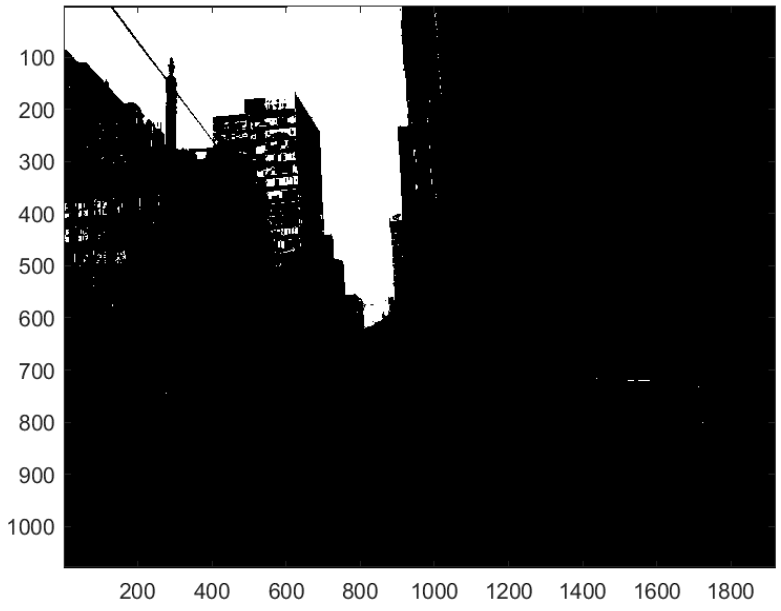

Figure 8 - Output from detectSky1.bmp

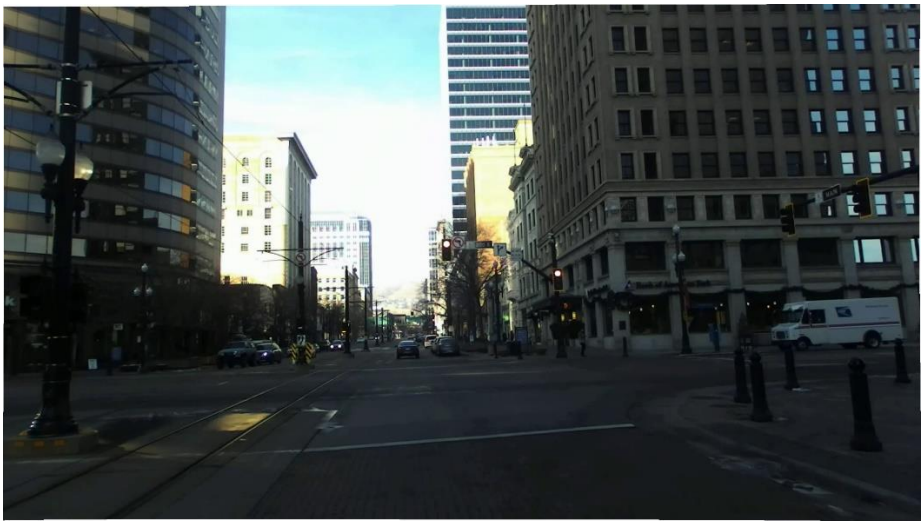

Figure 9 - detectSky2.bmp

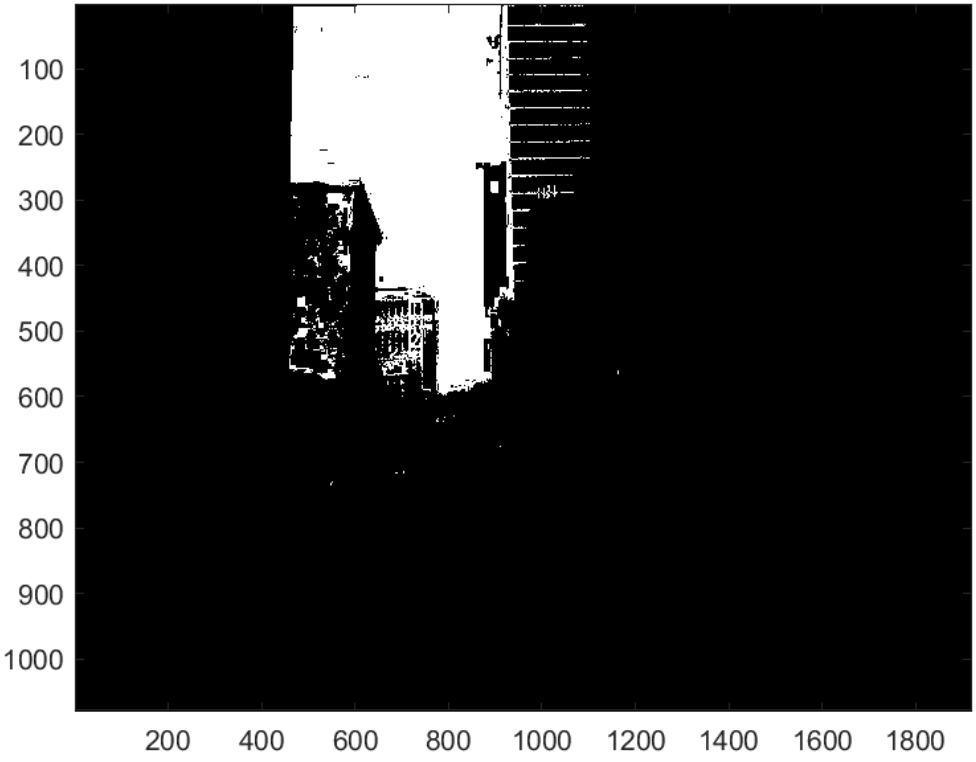

Figure 10 - Output from detectSky2.bmp

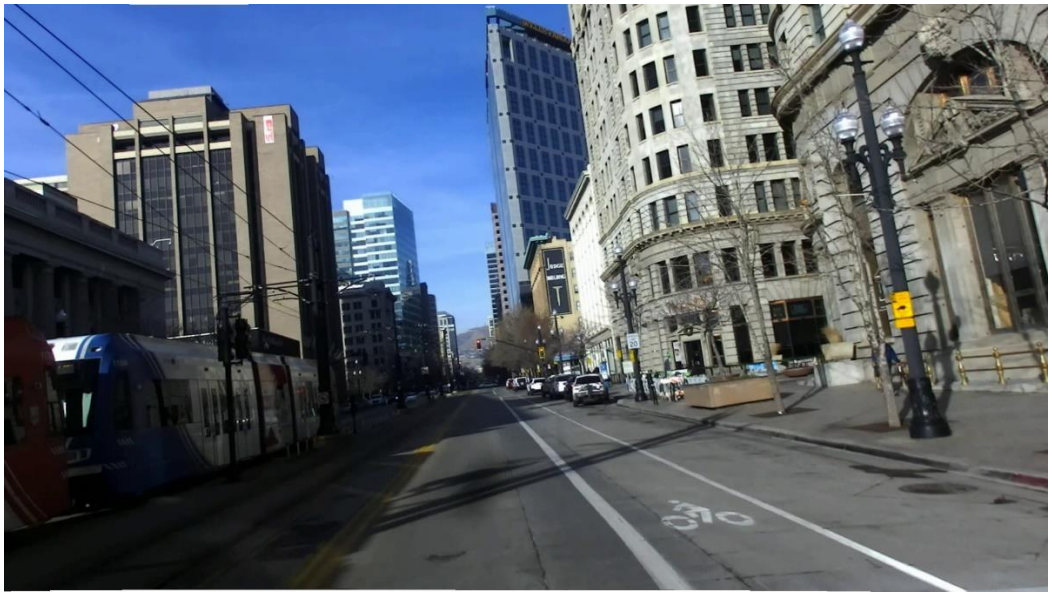

Figure 11 - detectSky3.bmp

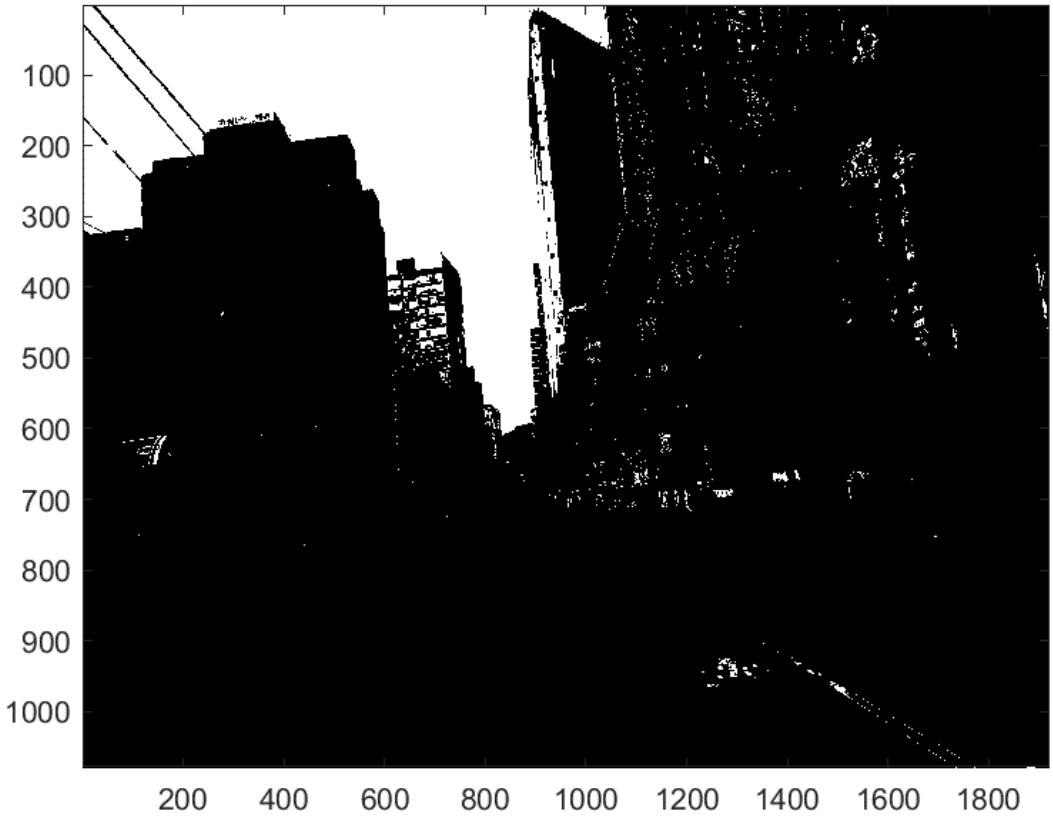

Figure 12 - Output from detectSky3.bmp

III. Stereo Vision Matcher

Algorithm Summary:

The Stereo Vision Matcher algorithm is used to compute and output a 'disparity map' that utilizes a 'left-eye' and a 'right-eye' image of the same scene to create a mapping that describes the relative proximity of each pixel to the camera's coordinate frame. Because the output image that depicts the map is in grayscale, a lighter color pixel (or a pixel with a higher value) has a higher disparity (meaning that they are closer to the camera), while darker color pixels indicate low disparity (which are farther away from the camera).

The disparity map is computed by iterating through each pixel and finding the disparity value that corresponds with the maximum NCC value. That disparity value is assigned to its pixel in the output image and directly corresponds with its brightness in the disparity map. To find the disparity value, you create two patches or filters (in reality, they are square matrices that hold grayscale pixel values) and compute the NCC value between them. The NCC equation is as follows:

$$NCC(A,B) = \frac{\sum_{ij} a_{ij}.b_{ij}}{\sqrt{\sum a_{ij}^2}\sqrt{\sum b_{ij}^2}}$$

Equation 1 - NCC

Where 'A' and 'B' are the two patches from the image and 'aij' and 'bij' are the corresponding elements at index (i,j) in each matrix. A value of 1 corresponds with a perfect match between the two patches of pixels, while a 0 indicates the opposite. As we try to calculate the disparity value for a certain pixel, we use the maximum NCC value as our indicator for the best matching disparity value.

To determine the patches A and B, we create square 5x5 matrices centered around each pixel (matrices who's bounds extend beyond that of the image are ignored). As we keep the patch on the left image stable, we try a variety of patches shifted from right to left along the corresponding row within the right image. This is because each pixel in the right image is, in general, equivalent to a pixel in the left image, except shifted to the right a certain number of pixels. This shift is our disparity. Once we find the disparity value that results in the largest NCC value, we have found our matching pixel, and map that value to our disparity map.

Once every pixel that has a patch within the limits of the image boundaries has had its disparity computed, the disparities are placed into a grayscale image, and output as an image.

Corresponding function files:

stereoMatching.m – the function that runs the Stereo Vision Matching algorithm as described below.

Pseudocode

```
Output = stereoMatching(leftImg, rightImg):
Initialize parameters
Convert images to grayscale (and from int8 to double)
Get the dimensions of the two input images
Initialize an image with pixel values of 0 to hold our disparity map (outputImg)
For each pixel in both images
        Set bestDisparity = 0
        Set bestNCC = 0

        For disp=1:DISPARITY_RANGE
                Get the limits of the square matrix that we will drag across each image
                        % top = y – EXTEND
                        % bottom = y + EXTEND
                        % leftImg_leftside = x – EXTEND
                        % leftImg_rightside = x + EXTEND
                        % rightImg_leftside = x – disp – EXTEND
                        % rightImg_rightside = x – disp + EXTEND
                Create a skipFlag to signal if we should skip this pixel
                        % Done if indices of matrices are out of bounds
                If any of the limits of the matrices (top, bottom, …, etc…) are out of bounds
                        Set the skipFlag = 1
                If (skipFlag == 0)
                        Patch1 = square submatrix of leftImg using limits described above
                        Patch2 = square submatrix of rightImg using limits described above
                        CurrNCC = NCC(Patch1, Patch2)
                        If (CurrNCC > bestNCC)
                                bestNCC = currNCC
                                bestDisparity = disp
        outputImg(pixel) = bestDisparity
return outputImg
```

NCC.m – a helper function for the Stereo Vision Matching algorithm. Given two matrices (patches), this function computes the corresponding NCC value for the stereo matcher to use.

Pseudocode

Output = NCC(leftPatch, rightPatch):
Calculate the numerator
    Element-wise multiplication between leftPatch and rightPatch
    Num = Sum all the elements of the resulting matrix together
Calculate the denominator
    Square each element of leftPatch
    Square each element of rightPatch
    Sum elements of squared leftPatch
    Sum elements of squared rightPatch
    Den = sqrt(leftSum) * sqrt(rightSum)

Output = Num / Den

Results:

This was truthfully the hardest algorithm to implement, simply because I was unaware of the fact that the image pixel values needed to be converted from int8 to double after performing the rgb2gray on the original image. Once that was done, I obtained very satisfactory results. Overall, the algorithm doesn't take very long to run on the first two pairs of images, but the third and final pair of images the algorithm can take between 5-10 minutes to run. However, the results appear to be satisfactory as nearby objects are colored brighter than those that are not.

Figure 13 - left1.png

Figure 14 - right1.png

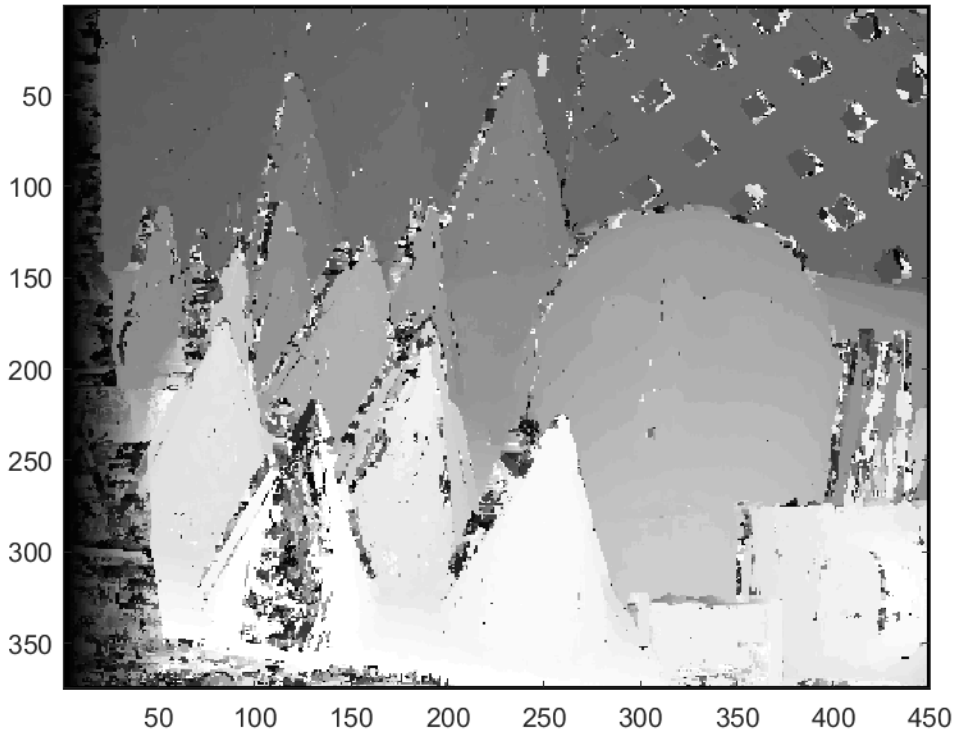

Figure 15 - Output from left1.png and right1.png

Figure 16 - left2.png

Figure 17 - right2.png

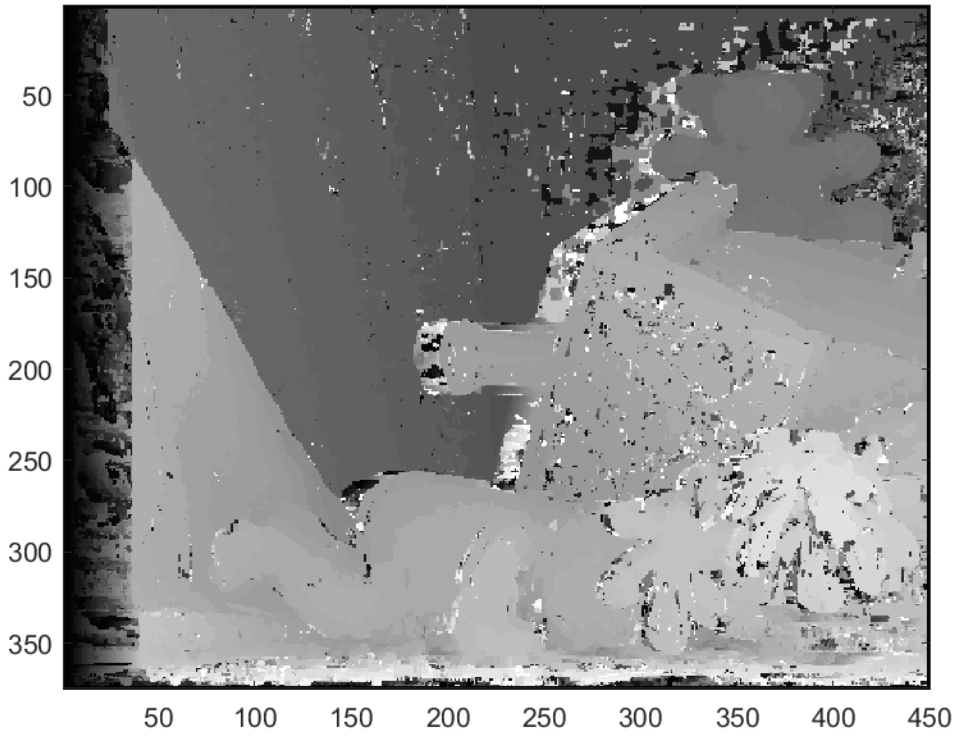

Figure 18 - Output from left2.png and right2.png

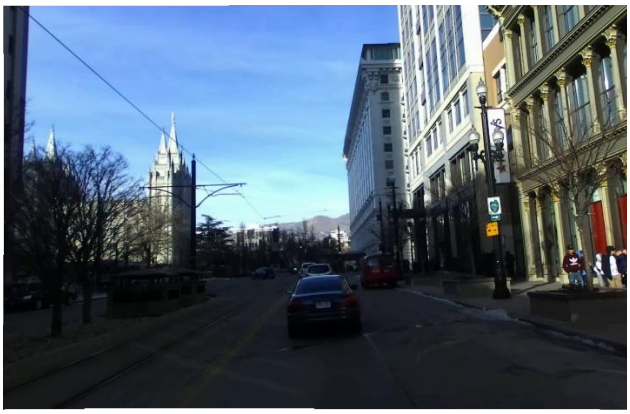

Figure 19 - left3.bmp

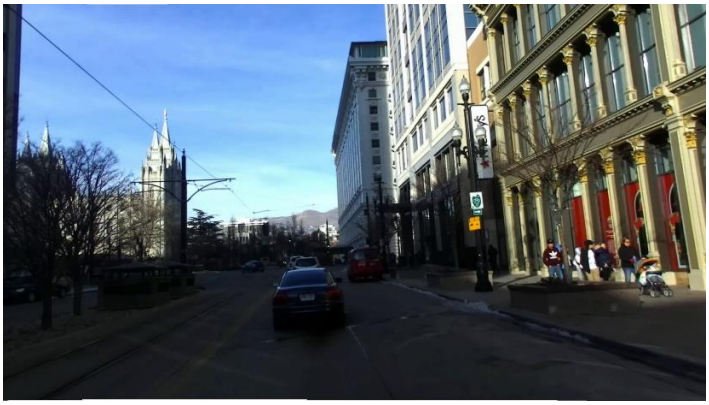

Figure 20 - right3.bmp

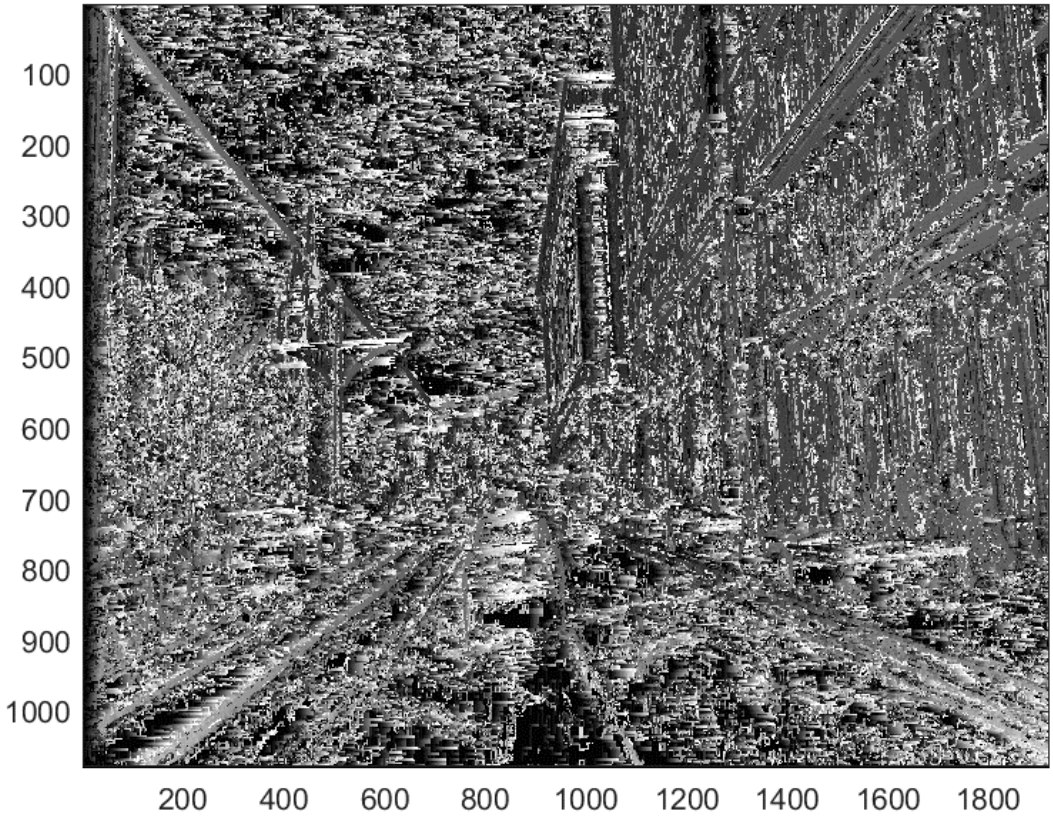

Figure 21 - Output from left3.bmp and right3.bmp

IV. Overall Code Organization

Root >

README.txt

main.m – runs all 3 algorithms using inputs from the 'Inputs' folder, and prints the outputs to the 'Outputs' folder. For other function file descriptions, see their summaries above in sections I, II, and III.

Functions >

extractEdges.m

getLineEst.m

getRandomPixel.m

skySegmentation.m

NCC.m

stereoMatching.m

Inputs >

Input Images (.bmp, .png files)

Outputs >

Results from running main.m (.bmp, .png files)

V. Outside Resources

How to access functions in another folder outside of the MATLAB root:

https://www.mathworks.com/matlabcentral/answers/31113-call-functions-from-subpath

How RGB colors work in MATLAB:

https://www.mathworks.com/company/newsletters/articles/how-matlab-represents-pixel-colors.html

*There may be others that I had forgotten to document. I didn't see the requirement to list all of these sites until the project was almost over. These are the ones I could remember and find. I will be sure to be more careful on the next project.