ASSIGNMENT 1

282.762 Robotics and Automation

Caelan Vandermey, 14087877

Table of Contents

Table of Equations

Table of Figures

1. Introduction

This report details a way of programming a Canny Edge Detection algorithm and compares it to the built in OpenCV Canny edge detection function.

A canny edge detector program can be broken down into four main parts;

1. Noise filtering
2. Finding the intensity-gradient of the image
3. Non-maximum suppression
4. Hysteresis

Noise filtering is completed using a Gaussian filter. The intensity-gradient of the image is found by using a pair of Sobel filters. Non-maximum suppression is completed by comparing the gradients and taking the strongest edge. Hysteresis is accomplished through finding the white pixels and making the connected grey pixels white as well.

Section two explains the code in more detail and section three compares the output of the program to that of the built in Canny function.

2. Program Explanation

This section of the report will detail the how and why of the program. This is broken into six sections; program setup, the Gaussian filter, Sobel filter and manipulation, non-maximum suppression, hysteresis and image outputs.

2.1. Program Setup

The setup of the program is where additional libraries are included, namespaces are declared and variables are created for use in the program. This can be broken into two parts; global and local. The global setup is made for use within the entire program and the local setup is for use within only the main function.

2.1.1. Global Setup Code

The code shown in Figure 1 handles adding additional libraries, namespace declaration and some variable creation. The libraries included are the entirety of the OpenCV library (for simplified image manipulation), cmath (for square root and power functions) and cstdio (for debugging the program). The namespaces used are cv (for the OpenCV functionality) and std (for the debugging process).

The variables are declaring pi, for use in the Sobel manipulation, and declaring my white (wT) and black (bT) thresholds for the hysteresis process. The thresholds are float types, despite being able to be standard int type, so that there is no type conversion occurring later in the program.

```
1   #include <opencv2\opencv.hpp>
2   #include <cmath>
3   #include <cstdio>
4
5   using namespace cv;
6   using namespace std;
7
8    float pi = 3.1415926535897;
9
10   float wT = 75;
11   float bT = 20;
```

Figure 1 Global Setup Code

2.1.2. Local Setup Code

Figure 2 shows the setup code in the main function. This assign and prepares the variables that will be used in the program. “x” and “y” are index variables for when the images have operations being run on a pixel by pixel basis.

The “Mat” type variables are all instances of matrices that will be used to hold the image data. Line 20 in the code is reading the given image into the program and converting it directly to greyscale.

Lines 22 to 29 show the preparation of the remaining matrices so that the image data will be in the correct format. They are assigned to have the same number of rows and columns at the input image and they are all in the 32-bit float format, to preserve accuracy in the calculated data.

```
16      Mat image, imageGaussian, imageX, imageY, imageG, imageT, imageS, imageH, imageF, imageC, imageA;
17      int x = 0;
18      int y = 0;
19
20      image = imread("Image 1.jpg", CV_LOAD_IMAGE_GRAYSCALE);
21
22      imageGaussian = Mat(image.rows, image.cols, CV_32F);
23      imageX = Mat(image.rows, image.cols, CV_32F);
24      imageY = Mat(image.rows, image.cols, CV_32F);
25      imageG = Mat(image.rows, image.cols, CV_32F);
26      imageT = Mat(image.rows, image.cols, CV_32F);
27      imageS = Mat(image.rows, image.cols, CV_32F);
28      imageH = Mat(image.rows, image.cols, CV_32F);
29      imageF = Mat(image.rows, image.cols, CV_32F);
```

Figure 2 Local Setup Code

2.2. Gaussian Filter

After reading the image in as a greyscale image, the edges are now softened using a Gaussian blur filter. The built in Gaussian Blur function has been used as this would result in the same output as a 2Dfilter pass with a hard-coded Gaussian kernel. This output is placed into “imageGaussian” for use later in the program.

A 3x3 Gaussian filter is used instead of a 5x5 filter as the 3x3 does blur the image, but not enough to remove too much data from the final output. This medium blur allows for the softer edges to be removed and for the harder edges to still propagate through for further processing.

The image is then converted into a “CV_32F”, which is the 32-bit float for the image. This is required as the image returned by the “GaussianBlur” function is in “uchar”, which is an 8-bit number, which will not be accurate enough for later calculations.

```
31      GaussianBlur(image, imageGaussian, Size(3, 3), 0, 0);
32      imageGaussian.convertTo(imageGaussian, CV_32F);
```

Figure 3 Gaussian Filter Code

2.3. Sobel Filter and Manipulation

The blurred image is then passed through the two Sobel filters (X and Y). This is done using the Sobel function (shown in Figure 4), where line 34 creates the X component of the image and line 35 creates the Y component of the image with 3x3 Sobel filters. The outputs are placed into “imageX” and “imageY”.

To create the magnitude and angle values, the program analyses each pixel in turn (using nested for loops). Lines 39 and 40 extract the “x” and “y” components from their respective images and places them into “imageG” for the magnitude and “imageT” for the angle.

Line 41 uses Equation 1 to calculate the magnitude of the pixel’s gradient and stores the value in “imageG” for later use. Line 42 uses Equation 2 to calculate the angle of the pixel’s gradient and line 43 stores this angle in “imageT” for later use.

```
34          Sobel(imageGaussian, imageX, CV_32F, 1, 0, 3);
35          Sobel(imageGaussian, imageY, CV_32F, 0, 1, 3);
36
37          for (y = 0; y < imageX.rows; y++) {
38              for (x = 0; x < imageX.cols; x++) {
39                  float gx = imageX.at<float>(y, x);
40                  float gy = imageY.at<float>(y, x);
41                  imageG.at<float>(y, x) = sqrt(pow(gx, 2) + pow(gy, 2));
42                  float ang = atan2(gy, gx) * 360 / (2 * pi) + 180;
43                  imageT.at<float>(y, x) = ang;
44              }
45          }
```

Figure 4 Sobel Filter and Manipulation Code

$$G = \sqrt{G_x^2 + G_y^2}$$

Equation 1 Magnitude Calculation

$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Equation 2 Angle Calculation

2.4. Non-Maximum Suppression

Non-maximum suppression is the removal of data that is not the strongest to find the sharpest edges. Each pixel in the image has a corresponding magnitude and angle. This angle will fall into one of four pairs of angles (shown in Figure 6) where each colour is a pair for comparison (i.e. the green pixels are compared with the grey pixel if the angle is between 67.5° and 112.5° or 247.5° and 292.5°). The pixel pair and the current pixel are compared to see if the current pixel has the largest magnitude. If the current magnitude is the largest, then the value is taken to the next image; otherwise, the current pixel is set to zero in “imageS”. Figure 5 shows the code that implements this. The nested for loops the control the inspected pixel slightly different in the way they are ranged. They begin one row and column in from the edge and end one row and column before the other edge. This prevents the program from trying to access data outside of “imageG”.

```
47      for (y = 1; y < imageG.rows - 1; y++) {
48          for (x = 1; x < imageG.cols - 1; x++) {
49              float g = imageG.at<float>(y, x);
50              float t = imageT.at<float>(y, x);
51              if ((((t <= 22.5)) || ((t > 337.5))) || ((t > 157.5) && (t <= 202.5))) {
52                  if ((g > imageG.at<float>(y, x - 1)) && (g > imageG.at<float>(y, x + 1))) {
53                      imageS.at<float>(y, x) = g;
54                  } else {
55                      imageS.at<float>(y, x) = 0;
56                  }
57              }
58              else if (((t > 22.5) && (t <= 67.5)) || ((t > 202.5) && (t <= 247.5))) {
59                  if ((g > imageG.at<float>(y - 1, x + 1)) && (g > imageG.at<float>(y + 1, x - 1))) {
60                      imageS.at<float>(y, x) = g;
61                  } else {
62                      imageS.at<float>(y, x) = 0;
63                  }
64              }
65              else if (((t > 67.5) && (t <= 112.5)) || ((t > 247.5) && (t <= 292.5))) {
66                  if ((g > imageG.at<float>(y - 1, x)) && (g > imageG.at<float>(y + 1, x))) {
67                      imageS.at<float>(y, x) = g;
68                  } else {
69                      imageS.at<float>(y, x) = 0;
70                  }
71              }
72              else if (((t > 112.5) && (t <= 157.5)) || ((t > 292.5) && (t <= 337.5))) {
73                  if ((g > imageG.at<float>(y - 1, x - 1)) && (g > imageG.at<float>(y + 1, x + 1))) {
74                      imageS.at<float>(y, x) = g;
75                  } else {
76                      imageS.at<float>(y, x) = 0;
77                  }
78              }
79          }
80      }
```

Figure 5 Non-Maximum Suppression Code

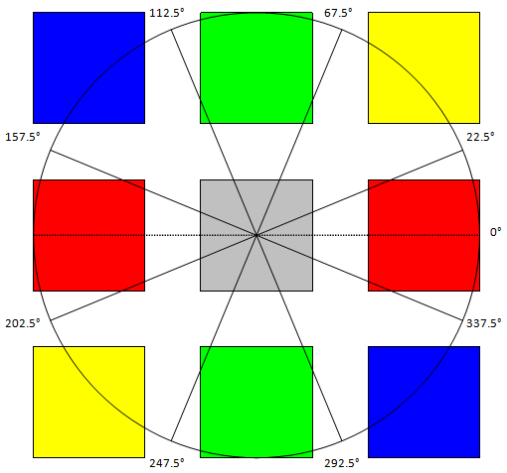

Figure 6 Non-Maximum Suppression Pair Diagram

2.5. Hysteresis

The hysteresis code is broken down into two part; the main function code and the recursive function code.

2.5.1. Hysteresis Main Program

The main function code is shown in Figure 7 and has two main parts. The first part begins by copying the suppressed image from “imageS” into “imageH” so that “imageS” can be shown later. “imageH” is then inspected pixel by pixel in the same fashion as the non-maximum suppression for loops (to ensure the inspected data is not outside of the image). The values are compared to the white threshold (wT) to see if the value can be considered white. If it is white, the program ensures the value is white (by making it 0xFFFF which is an all ones’ situation in a 32-bit float). The program then runs the recursive function, called “rabithole” (which is explained in Section 2.5.2).

The second part is inspecting the same range of pixels, but checking if the value is white (0xFFFF). If this is true, then the pixel is placed in the output image (“imageF”).

```
82      imageH = imageS.clone();
83      for (y = 1; y < imageH.rows - 1; y++) {
84          for (x = 1; x < imageH.cols - 1; x++) {
85              float val = imageH.at<float>(y, x);
86              if (val > wT) {
87                  imageH.at<float>(y, x) = 0xFFFF;
88                  rabithole(&imageH, y, x);
89              }
90          }
91      }
92
93      for (y = 1; y < imageH.rows - 1; y++) {
94          for (x = 1; x < imageH.cols - 1; x++) {
95              float val = imageH.at<float>(y, x);
96              if (val == 0xFFFF) {
97                  imageF.at<float>(y, x) = 0xFFFF;
98              }
99          }
100     }
```

Figure 7 Hysteresis Main Code

2.5.2. Hysteresis Recursive Function

The recursive function is named “rabithole” as when it is called, it will call itself until there are no more grey pixels left in the branch that is being followed.

The function begins by assigning the current pixel to white (0xFFFF) and then checks the eight pixels directly or diagonally adjacent to see if any are grey (brighter than black {greater than bT}, but darker than white {less than wT}). If there are no grey pixels to be assigned to white, then the function ends and the previous set of checks is continued. This ensures that any grey pixel next to a white pixel is also considered to be white.

```
117  void rabithole(Mat* img, int y, int x) {
118      img[0].at<float>(y, x) = 0xFFFF;
119      if ((img[0].at<float>(y - 1, x - 1) > bT) && (img[0].at<float>(y - 1, x - 1) < wT)){
120          rabithole(img, y - 1, x - 1);
121      } else if ((img[0].at<float>(y - 1, x) > bT) && (img[0].at<float>(y - 1, x) < wT)) {
122          rabithole(img, y - 1, x);
123      } else if ((img[0].at<float>(y - 1, x + 1) > bT) && (img[0].at<float>(y - 1, x + 1) < wT)) {
124          rabithole(img, y - 1, x + 1);
125      } else if ((img[0].at<float>(y, x - 1) > bT) && (img[0].at<float>(y, x - 1) < wT)) {
126          rabithole(img, y, x - 1);
127      } else if ((img[0].at<float>(y, x + 1) > bT) && (img[0].at<float>(y, x + 1) < wT)) {
128          rabithole(img, y, x + 1);
129      } else if ((img[0].at<float>(y + 1, x - 1) > bT) && (img[0].at<float>(y + 1, x - 1) < wT)) {
130          rabithole(img, y + 1, x - 1);
131      } else if ((img[0].at<float>(y + 1, x) > bT) && (img[0].at<float>(y + 1, x) < wT)) {
132          rabithole(img, y + 1, x);
133      } else if ((img[0].at<float>(y + 1, x + 1) > bT) && (img[0].at<float>(y + 1, x + 1) < wT)) {
134          rabithole(img, y + 1, x + 1);
135      }
136  }
```

Figure 8 Hysteresis Recursive Function Code

2.6. Image Outputs

The program ends by writing the images into nine ".jpg" files so that the entire process can be observed. Lines 102 to 110 writes the images shown in Figures Figure 10 through Figure 18.

The "Canny" function is then run on the original image and written to a tenth image for comparison in Section 3. This image is shown in Figure 19.

```
102      imwrite("image02_Grey.jpg", image);
103      imwrite("image03_Gaussian.jpg", imageGaussian);
104      imwrite("image04_XComponent.jpg", imageX);
105      imwrite("image05_YComponent.jpg", imageY);
106      imwrite("image06_Magnitude.jpg", imageG);
107      imwrite("image07_Theta.jpg", imageT);
108      imwrite("image08_Suppressed.jpg", imageS);
109      imwrite("image09_Hysteresis.jpg", imageH);
110      imwrite("image10_Final.jpg", imageF);
111
112      Canny(image, imageA, 75, 20);
113      imwrite("image11_CannyAlgorithm.jpg", imageA);
114      return 0;
```

Figure 9 Image Output Code

Figure 10 Greyscale Image ("image")

Figure 11 Blurred Image ("imageGaussian")

Figure 12 X Component of Gradient ("imageX")

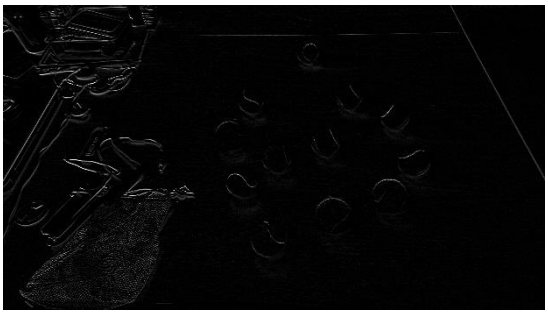

Figure 13 Y Component of Gradient ("imageY")

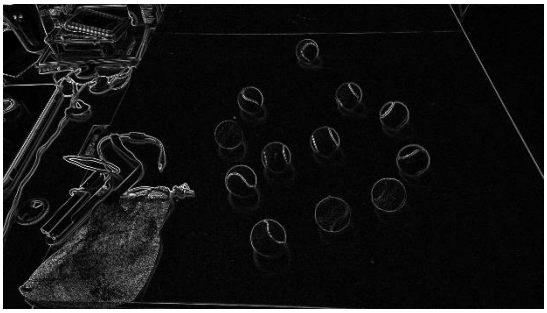

Figure 14 Gradient Magnitude ("imageG")

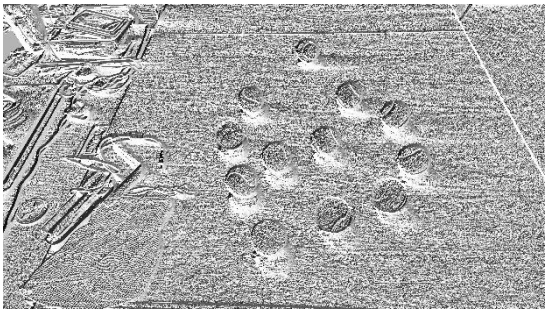

Figure 15 Gradient Angle ("imageT")

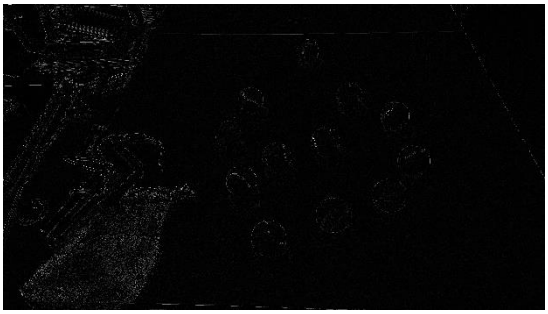

Figure 16 Suppressed Image ("imageS")

Figure 17 Hysteresis Image ("imageH")

Figure 18 Program Output ("imageF")

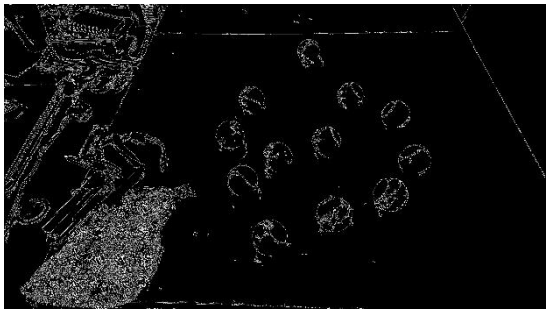

Figure 19 Canny Algorithm Output

3. Algorithm Output Comparison

Figure 18 is the output from the created algorithm and Figure 19 is the output from the “Canny” function. Figure 18 shows much less noise compared to Figure 19, while maintaining the more important edges. Figure 19 also shows more edges than what exist in the image (the three lower tennis balls do not have edges on their faces like what is shown in Figure 19).