Canny Edge Detector

Yucheng Qian | yq791 | N17384422

1.Source code file name: CannyEdgeDetector.java

2.How to run the program:

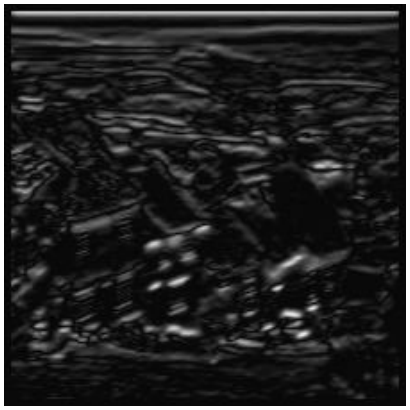
This is a standard Java program, to run the program, you can copy & paste the source code to a Java running environment such as Eclipse or IntelliJ. You can create a new project and put this java file into 'src' folder. Then you should put the test image into the project directory. (My default way is to create a folder called 'image' under the project directory and put test images in it. Don't forget to rename the image if necessary, the file name should correspond to the input source in the code). The generated images will be stored in the same folder with a filename - 'output'.

3. Image results for each step:

Note: Both images have threshold T1 = 15 and T2 = 30.

image1

after_gaussian_smoothing

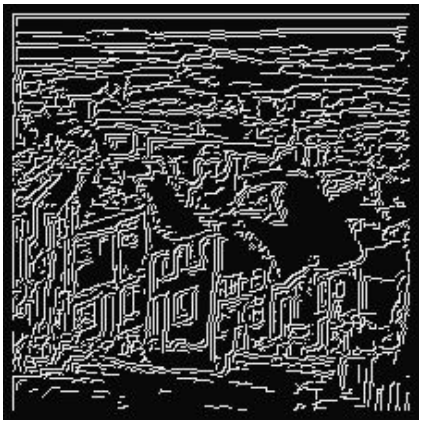


gradient_horizontal

gradient_vertical

gradient_magnitude

non_maxima_suppression



double_thresholding

image2

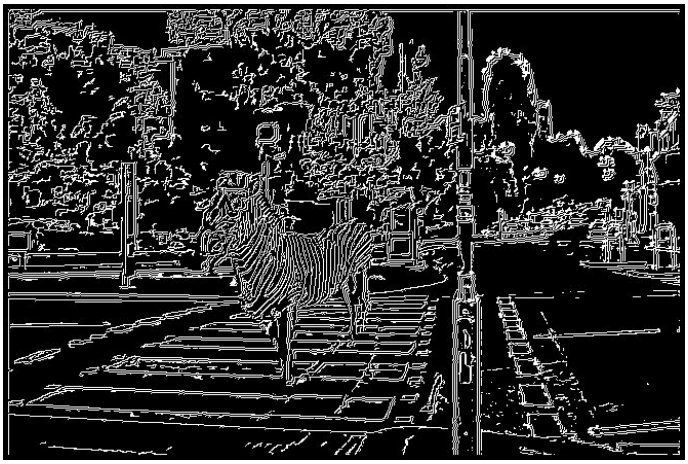
after_gaussian_smoothing

gradient_horizontal

gradient_vertical

gradient_magnitude

non_maxima_suppression



double_thresholding

4. Source Code

```
import java.awt.Color;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
import java.io.File;
import java.io.IOException;

public class CannyEdgeDetector {
    private static String image_path = "./image/test2.bmp";
    private static double normalize_value = 140.0;
    private static int[][] gradientAngle;
    private static double[][] gaussian_Mask = new double[][] {
        {1.0, 1.0, 2.0, 2.0, 2.0, 1.0, 1.0},
        {1.0, 2.0, 2.0, 4.0, 2.0, 2.0, 1.0},
        {2.0, 2.0, 4.0, 8.0, 4.0, 2.0, 2.0},
        {2.0, 4.0, 8.0, 16.0, 8.0, 4.0, 2.0},
        {2.0, 2.0, 4.0, 8.0, 4.0, 2.0, 2.0},
        {1.0, 2.0, 2.0, 4.0, 2.0, 2.0, 1.0},
        {1.0, 1.0, 2.0, 2.0, 2.0, 1.0, 1.0},
    };
    private static int[][] sobelX = new int[][] {
        {-1, 0, 1},
        {-2, 0, 2},
        {-1, 0, 1}
    };
    private static int[][] sobelY = new int[][] {
        {1, 2, 1},
        {0, 0, 0},
        {-1, -2, -1}
    };

    public static void main(String[] args) {
        // read the image
        BufferedImage image = readImage(image_path);
        // apply Gaussian filter
        gaussianSmoothing(image);
        // gradient operations - including non-maxima suppression
        getGredients(image);
        // thresholding
        thresholding(image);
        // save image
        writeImage(image);
    }

    private static void thresholding(BufferedImage img) {
        // select two thresholds so that T2 = 2T1
        int T2 = 30;
        int T1 = 15;

        // set up white color
        int white = new Color(255, 255, 255).getRGB();
        // set up black color
        int black = new Color(0, 0, 0).getRGB();

        for (int y = 0; y < img.getHeight(); y++) {
            for (int x = 0; x < img.getWidth(); x++) {
                Color color = new Color(img.getRGB(x, y));
                //If < T1 let it = 0
```

```
            if (color.getRed() < T1) {
                img.setRGB(x, y, black);
            }
            //If > T2 let it = 255
            else if (color.getRed() >= T2) {
                img.setRGB(x, y, white);
            }
            // else we find if there is a valid neighbor
            else {
                if (y >= 1 && y <= img.getHeight() - 2 && x >= 1 && x <= img.getWidth() - 2) {
                    boolean hasQualifiedNeighbor = false;
                    // check 8-connected neighbors
                    for (int i = -1; i <= 1; i++) {
                        for (int j = -1; j <= 1; j++) {
                            if (x >= 0 && x < gradientAngle.length && x + i >= 0 && x + i < gradientAngle.length
                                && y >= 0 && y < gradientAngle[0].length && y + j >= 0 && y + j < gradientAngle[0].length) {
                                    //
                                    if (new Color(img.getRGB(x + i, y + j)).getRed() > T2
                                            && Math.abs(gradientAngle[x][y] - gradientAngle[x + i][y + j]) <= 45)
                                        hasQualifiedNeighbor = true;
                                }

                        }
                    }
                    if (hasQualifiedNeighbor) img.setRGB(x, y, white);
                    else img.setRGB(x, y, black);
                }
            }
        }
    }
}

private static void nonMaximaSuppression(BufferedImage img, int[][] sobelX_after, int[][] sobelY_after) {
    int height = img.getHeight();
    int width = img.getWidth();
    // get the gradient angle
    gradientAngle = new int[height][width];
    int[][] averageAngle = new int[height][width];
    // calculate the gradient angle
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            if (x >= 4 && x <= width - 5 && y >= 4 && y <= height - 5) {
                gradientAngle[y][x] = (int) ((180 / Math.PI) * Math.atan2((sobelY_after[y][x]), (sobelX_after[y][x])));
            }
        }
    }
    // calculate the average angle sector
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            if (x >= 4 && x <= width - 5 && y >= 4 && y <= height - 5) {
                if ((gradientAngle[y][x] >= -22.5 && gradientAngle[y][x] < 22.5)
                        || gradientAngle[y][x] >= 157.5 || gradientAngle[y][x] < -157.5) {
                    averageAngle[y][x] = 0;
                }
                else if ((gradientAngle[y][x] >= 22.5 && gradientAngle[y][x] < 67.5) ||
                        (gradientAngle[y][x] >= -157.5 && gradientAngle[y][x] < -112.5)) {
                    averageAngle[y][x] = 45;
```

```
            }
            else if ((gradientAngle[y][x] >= 67.5 && gradientAngle[y][x] < 112.5) ||
                  (gradientAngle[y][x] >= -112.5 && gradientAngle[y][x] < -67.5)) {
               averageAngle[y][x] = 90;
            }
            else if ((gradientAngle[y][x] >= 112.5 && gradientAngle[y][x] < 157.5) ||
                  (gradientAngle[y][x] >= -67.5 && gradientAngle[y][x] < -22.5)) {
               averageAngle[y][x] = 135;
            }
         }
      }
   }
   // apply non-maxima suppression
   for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
         if (x >= 5 && x <= width - 6 && y >= 5 && y <= height - 6) {
            // sector 0 - compare center pixel with left and right pixel
            if (averageAngle[y][x] == 0) {
               if (((x - 1) >= 0 && img.getRGB(x, y) < img.getRGB(x - 1, y)) ||
                     ((x + 1) <= (width - 1) && img.getRGB(x, y) < img.getRGB(x + 1, y))) {
                  img.setRGB(x, y, 0);
               }
            }
            // sector 1 - compare center pixel with up right and down left pixel
            else if (averageAngle[y][x] == 45) {
               if (((x - 1) >= 0 && (y - 1) >= 0 && img.getRGB(x, y) < img.getRGB(x - 1, y - 1)) ||
                     ((x + 1) <= (height - 1) && (y + 1) <= (height - 1) &&
                           img.getRGB(x, y) < img.getRGB(x + 1, y + 1))) {
                  img.setRGB(x, y, 0);
               }
            }
            // sector 2 - compare center pixel with up and down pixel
            else if (averageAngle[y][x] == 90) {
               if (((y - 1) >= 0 && img.getRGB(x, y) < img.getRGB(x, y - 1)) ||
                     ((y + 1) <= (height - 1) && img.getRGB(x, y) < img.getRGB(x, y + 1))) {
                  img.setRGB(x, y, 0);
               }
            }
            // sector 3 - compare center pixel with up left and down right pixel
            else if (averageAngle[y][x] == 135) {
               if (((x - 1) >= 0 && (y + 1) <= (height - 1) && img.getRGB(x, y) < img.getRGB(x - 1, y + 1)) ||
                     ((x + 1) <= (width - 1) && (y - 1) >= 0 && img.getRGB(x, y) < img.getRGB(x + 1, y - 1))) {
                  img.setRGB(x, y, 0);
               }
            }
         }
         // At locations with undefined gradient values and at locations where the center pixel has a
         // neighbor with undefined gradient value, let the output be zero (i.e., no edge.)
         else img.setRGB(x, y, 0);
      }
   }

}

private static void getGredients(BufferedImage img) {
   // use sobel operators to create two new image
   int[][] sobelX_after = getSobel(img, sobelX);
```

```
    int[][] sobelY_after = getSobel(img, sobelY);
    // calculate the magnitude
    getMagnitude(img, sobelX_after, sobelY_after);
    // apply non-maximum suppression
    nonMaximaSuppression(img, sobelX_after, sobelY_after);
}

private static void getMagnitude(BufferedImage img, int[][] sobel_x, int[][] sobel_y) {
    int height = img.getHeight();
    int width = img.getWidth();
    int[][] matrix = new int[height][width];
    int max = Integer.MIN_VALUE;

    // get all magnitude values
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int magnitude = (int) Math.sqrt(Math.pow(sobel_x[y][x], 2) + Math.pow(sobel_y[y][x], 2));
            matrix[y][x] = magnitude;
            if (magnitude > max) max = magnitude;
            //img.setRGB(x, y, new Color(magnitude, magnitude, magnitude).getRGB());
        }
    }
    double normalization = max / 255.0;
    // normalize and draw the image
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int value = (int) (matrix[y][x] / normalization);
            img.setRGB(x, y, new Color(value, value, value).getRGB());
        }
    }
}

private static int[][] getSobel(BufferedImage img, int[][] sobel) {
    int height = img.getHeight();
    int width = img.getWidth();

    int[][] new_sobel = new int[height][width];
    int[][] res_sobel = new int[height][width];

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int color = 0;
            if (x >= 4 && x <= width - 5 && y >= 4 && y <= height - 5) {
                for (int i = -1; i <= 1; i++) {
                    for (int j = -1; j <= 1; j++) {
                        // exclude boarder cases. If part of the 3 x 3 mask of the operator goes outside of the image
                        // border or lies in the undefined region of the image after Gaussian filtering, let the output value
                        // be undefined.
                        color += new Color(img.getRGB(x + j, y + i)).getRed() * sobel[i + 1][j + 1];
                    }
                }
            }
            // take absolute value
            new_sobel[y][x] = Math.abs(color);
        }
    }
    // find max value in the matrix
```

```
    int max = Integer.MIN_VALUE;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            if (new_sobel[y][x] > max) max = new_sobel[y][x];
        }
    }
    // divide max by 255
    double normalization = max / 255.0;
    // draw the image
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int value = (int) (new_sobel[y][x] / normalization);
            //img.setRGB(x, y, new Color(value, value, value).getRGB());
            res_sobel[y][x] = value;
        }
    }
    return res_sobel;
}

private static void gaussianSmoothing(BufferedImage img) {
    int height = img.getHeight();
    int width = img.getWidth();

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            double r = 0;
            // exclude boarder cases, if part of the Gaussian mask goes outside of the image
            // border, let the output image be undefined at the location of the reference center.
            if (x >= 3 && x <= width - 4 && y >= 3 && y <= height - 4) {
                for (int i = -3; i <= 3; i++) {
                    for (int j = -3; j <= 3; j++) {
                        Color c = new Color(img.getRGB(x + j, y + i));
                        r += c.getBlue() * gaussian_Mask[i + 3][j + 3];
                    }
                }
            }
            // divide by sum of entries to normalize
            r /= normalize_value;
            img.setRGB(x, y, new Color((int) r, (int) r, (int) r).getRGB());
        }
    }
}

private static void writeImage(BufferedImage image) {
    File tmpFile;

    try {
        tmpFile = new File("./image/output.bmp");
        ImageIO.write(image, "bmp", tmpFile);
        System.out.println("Image saved.");
    } catch (IOException e) {
        System.out.println("Exception occurred: unable to save image.");
        System.exit(1);
    }
}

private static BufferedImage readImage(String path) {
```

```
        BufferedImage image = null;
        File tmpFile;

        try {
            tmpFile = new File(path);
            image = ImageIO.read(tmpFile);
        } catch(IOException e){
            System.out.println("Exception occurred: unable to read the image.");
            System.exit(1);
        }
        return (image);
    }
}
```