CHAPTER 5
IMPLEMENTATION AND TESTING

5.1 Implementation

This project utilize the java programming language. This project use the Euclidean distance algorithm to find out the similarity value in each fingerprints types by seeing the proximity value of the image, as well as using the canny edge detection algorithm to recognize the fingerprint patterns. The size of the sample image used was 300x350 pixel.

Below is code that was used in the implementation of canny algorithm.

```
private void computeGradients(float kernelRadius, int kernelWidth)
{

  float kernel[] = new float[kernelWidth];
  float diffKernel[] = new float[kernelWidth];
  int kwidth;
  for (kwidth = 0; kwidth < kernelWidth; kwidth++)
  {
     float g1 = gaussian(kwidth, kernelRadius);
     if (g1 <= GAUSSIAN_CUT_OFF && kwidth >= 2) {
            break;
     }
     float g2 = gaussian(kwidth - 0.5f, kernelRadius);
     float g3 = gaussian(kwidth + 0.5f, kernelRadius);
     kernel[kwidth] = (g1 + g2 + g3) / 3f / (2f * (float) Math.PI * kernelRadius * kernelRadius);
        diffKernel[kwidth] = g3 - g2;
   }

   int initX = kwidth - 1;
    int maxX = width - (kwidth - 1);
   int initY = width * (kwidth - 1);
   int maxY = width * (height - (kwidth - 1));

   for (int x = initX; x < maxX; x++) {
     for (int y = initY; y < maxY; y += width) {
       int index = x + y;
       float sumX = data[index] * kernel[0];
       float sumY = sumX;
       int xOffset = 1;
       int yOffset = width;
       for (; xOffset < kwidth;) {
         sumY += kernel[xOffset] * (data[index - yOffset] + data[index + yOffset]);
         sumX += kernel[xOffset] * (data[index - xOffset] + data[index + xOffset]);
         yOffset += width;
         xOffset++;
       }

       yConv[index] = sumY;
       xConv[index] = sumX;
    }

  }

  for (int x = initX; x < maxX; x++) {
    for (int y = initY; y < maxY; y += width) {
       float sum = 0f;
       int index = x + y;
       for (int i = 1; i < kwidth; i++) {
          sum += diffKernel[i] * (yConv[index - i] - yConv[index + i]);
       }

       xGradient[index] = sum;
    }

}

for (int x = kwidth; x < width - kwidth; x++) {
  for (int y = initY; y < maxY; y += width) {
       float sum = 0.0f;
       int index = x + y;
```

```
      int yOffset = width;
      for (int i = 1; i < kwidth; i++) {
        sum += diffKernel[i] * (xConv[index - yOffset] - xConv[index + yOffset]);
        yOffset += width;
      }

      yGradient[index] = sum;
   }

}

initX = kwidth;
maxX = width - kwidth;
initY = width * kwidth;
maxY = width * (height - kwidth);
for (int x = initX; x < maxX; x++) {
  for (int y = initY; y < maxY; y += width) {
      int index = x + y;
      int indexN = index - width;
      int indexS = index + width;
      int indexW = index - 1;
      int indexE = index + 1;
      int indexNW = indexN - 1;
      int indexNE = indexN + 1;
      int indexSW = indexS - 1;
      int indexSE = indexS + 1;

      float xGrad = xGradient[index];
      float yGrad = yGradient[index];
      float gradMag = hypot(xGrad, yGrad);

         //perform non-maximal supression
      float nMag = hypot(xGradient[indexN], yGradient[indexN]);
      float sMag = hypot(xGradient[indexS], yGradient[indexS]);
```

```
            float wMag = hypot(xGradient[indexW], yGradient[indexW]);
            float eMag = hypot(xGradient[indexE], yGradient[indexE]);
            float neMag = hypot(xGradient[indexNE], yGradient[indexNE]);
            float seMag = hypot(xGradient[indexSE], yGradient[indexSE]);
            float swMag = hypot(xGradient[indexSW], yGradient[indexSW]);
            float nwMag = hypot(xGradient[indexNW], yGradient[indexNW]);
            float tmp;

            if (xGrad * yGrad <= (float) 0 /*(1)*/
               ? Math.abs(xGrad) >= Math.abs(yGrad) /*(2)*/
                  ? (tmp = Math.abs(xGrad * gradMag)) >= Math.abs(yGrad * neMag - (xGrad + yGrad) * eMag) /*(3)*/
                  && tmp > Math.abs(yGrad * swMag - (xGrad + yGrad) * wMag) /*(4)*/
                  : (tmp = Math.abs(yGrad * gradMag)) >= Math.abs(xGrad * neMag - (yGrad + xGrad) * nMag) /*(3)*/
                  && tmp > Math.abs(xGrad * swMag - (yGrad + xGrad) * sMag) /*(4)*/
               : Math.abs(xGrad) >= Math.abs(yGrad) /*(2)*/
                  ? (tmp = Math.abs(xGrad * gradMag)) >= Math.abs(yGrad * seMag + (xGrad - yGrad) * eMag) /*(3)*/
                  && tmp > Math.abs(yGrad * nwMag + (xGrad - yGrad) * wMag) /*(4)*/
                  : (tmp = Math.abs(yGrad * gradMag)) >= Math.abs(xGrad * seMag + (yGrad - xGrad) * sMag) /*(3)*/
                  && tmp > Math.abs(xGrad * nwMag + (yGrad - xGrad) * nMag) /*(4)*/) {
         magnitude[index] = gradMag >= MAGNITUDE_LIMIT ? MAGNITUDE_MAX : (int) (MAGNITUDE_SCALE * gradMag);

      } else {
            magnitude[index] = 0;
```

```
        }
      }
    }
}
```

Inside the compute gradient method, the initial process begins from smoothing, which were the process to smooth the image, then followed by the non-maximum suppression method which aimed to determine the edge points of the image. Next, the hysteresis process was done to make the outlines based on the threshold. If the edge point was >= threshold, then a white outlines will be generated, however if the edge point was <= threshold, then an outlines will not be formed at all / have a black color.

Below is the implementation process of the Euclidean algorithm

```
public static double jarakEuclidean(BufferedImage img1, BufferedImage img2) {
    int gray1, gray2 = 0;
    double jarakEuclid = 0;
    int barisImg1 = img1.getWidth();
    int barisImg2 = img2.getWidth();
    int kolomImg1 = img1.getHeight();
    int kolomImg2 = img2.getHeight();
  if ((barisImg1 == barisImg2) && (kolomImg1 == kolomImg2)) {
  for (int baris = 0; baris < barisImg1; baris++) {
    for (int kolom = 0; kolom < kolomImg1; kolom++) {
       int rgb1 = img1.getRGB(baris, kolom);
       int r1 = (rgb1 >> 16) & 0xFF;
       int g1 = (rgb1 >> 8) & 0xFF;
       int b1 = (rgb1 & 0xFF);
       gray1 = (r1 + g1 + b1) / 3;
       int rgb2 = img2.getRGB(baris, kolom);
       int r2 = (rgb2 >> 16) & 0xFF;
       int g2 = (rgb2 >> 8) & 0xFF;
       int b2 = (rgb2 & 0xFF);
       gray2 = (r2 + g2 + b2) / 3;
       jarakEuclid = jarakEuclid + Math.sqrt(Math.pow(gray1 - gray2, 2));
       }
```

```
        }
    }
    return jarakEuclid;
}
```

Euclidean distance method is used to calculate the distance between the input image and every single image in the dataset. The reference for the Euclidean distance calculation is the pixel point in the image. Smallest distance result will determine the results of fingerprint detection.

5.2 Testing

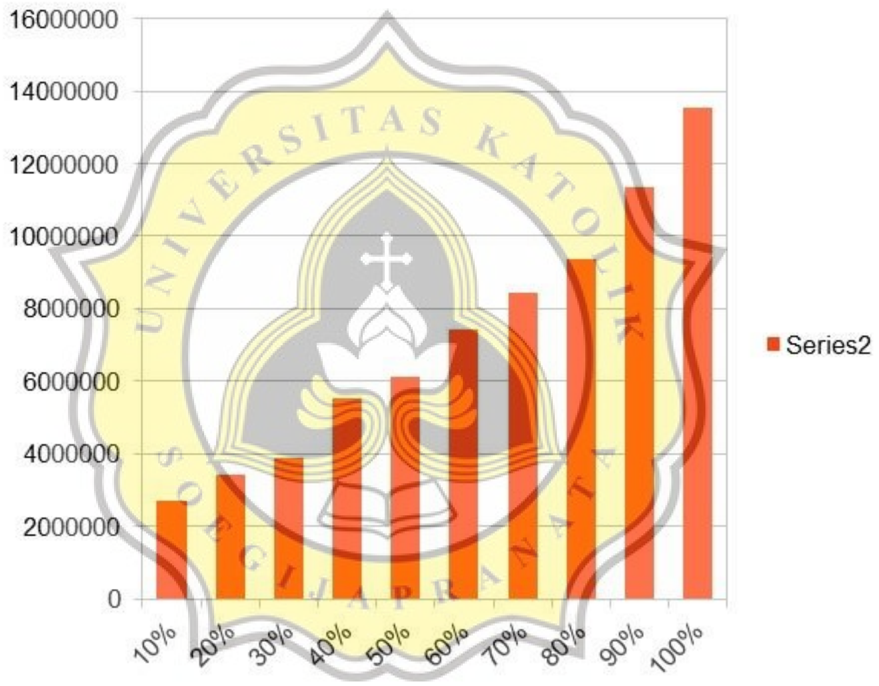

Illustration 12: Graph of the Euclidean Distance Results Based on the Image Noise

The graph above (Illustration 5) is the results of the Euclidean distance based on the image noise, so the higher the percentage of the image noise, the more unrecognizable the image will be.

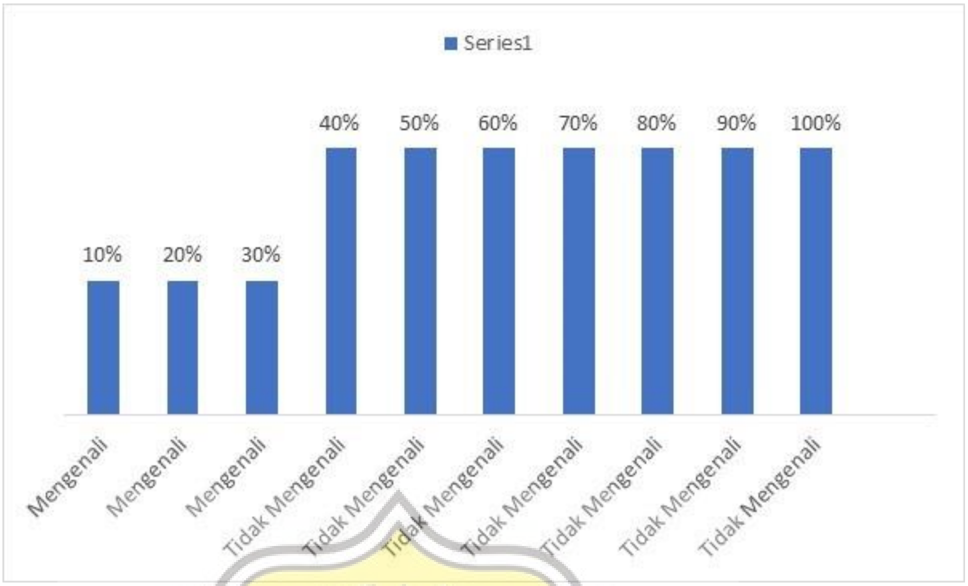

Illustration 13: Conclusion Graph of Euclidean Result

Graph above (illustration 6) is the conclusion graph from the previous Euclidean result graph. It can be concluded that Euclidean can recognize the image until 30% noise. If the noise percentage is more than 30%, then it will no longer be able to recognize whose fingerprint is in the input image.

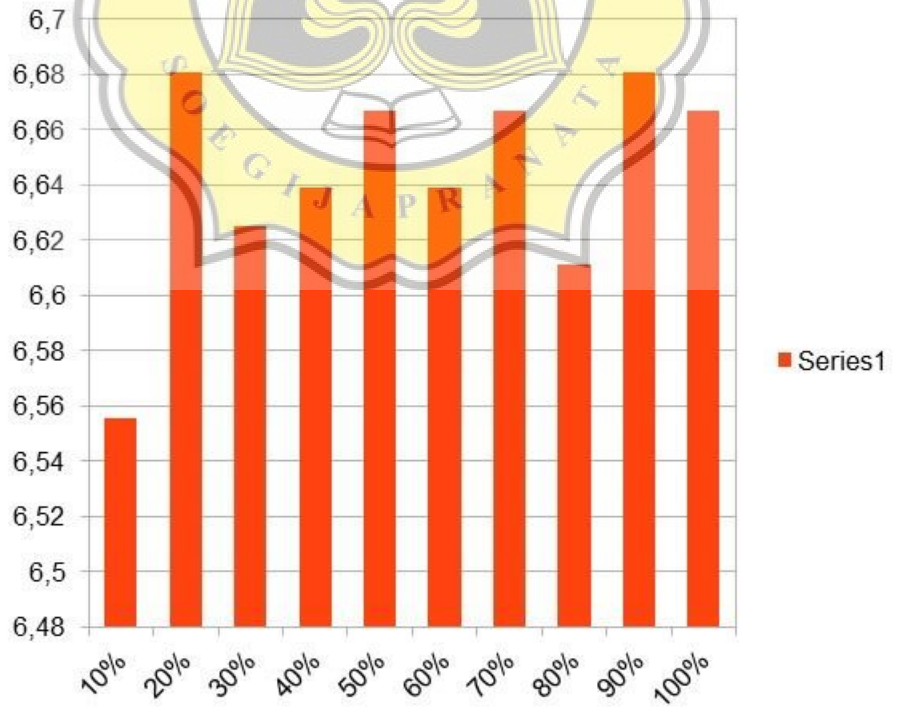

Illustration 14: Graph During the Euclidean Process

Graph above (illustration 7) is the graph about the processing time during the Euclidean running process. So in average, the program that I made needs 6 seconds to process the image of each fingerprints.