Taymer Computer Vision Challenge

Applicant: Ahmad Kaddoura

Enviornment: Visual Basic, Qt Creator, and OpenCV 2.4.11 (earliest build I could find)

GitHub repo: https://github.com/akaddoura/taymer_challenge

Email: ahmadmkaddoura@gmail.com

Date Submitted: 16 Feb, 2021

Summary

To complete this challenge a simple pipeline was built with two main functions designed to tackle the two tasks. A ***measure_diameter()*** function would apply a simple binary threshold mask which was tested to fit the cable test images as accurately as possible before taking the measurement by locating the nonzero values across a line in the image.

For the second task, two functions were designed to complete it. The first one, ***find_defect()***, would use canny edge detection and morphology functions to narrow down the contour count in each test image to the defect areas which were then filtered and isolated with boxes. At which point the second function, ***classify_defect()***, would classify the defect based on an average of pixel color intensity and the bounding box size ratio.

While I don't consider the solution perfect and rather biased for the test images, I did attempt to further improve it by testing other approaches which are detailed in the report.

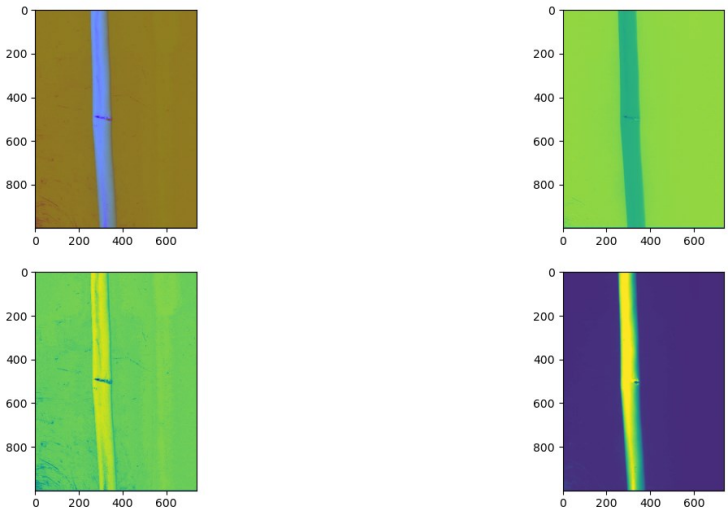
Initial Approach

On first seeing the problem and the different test images included, I believe the general intuition would be to jump towards trying different threshold techniques due to the clear contrast between the cable and the background and the white visibility around the defects. In that sense, I was sure I would be able to complete the first task by a simple binary threshold and I laid out a trial and error plan to test out the best option for isolating the defects in a mask.

As for defect classification, I decided it's best to wait and see how they will look up close after detection. I considered a sliding bar to find the black pixels which would classify a pinhole and cut against a scratch but even early on I wasn't sure I could build a robust enough detector given the low resolution of the images and the different wires and colors which could be seen in one of the cuts.

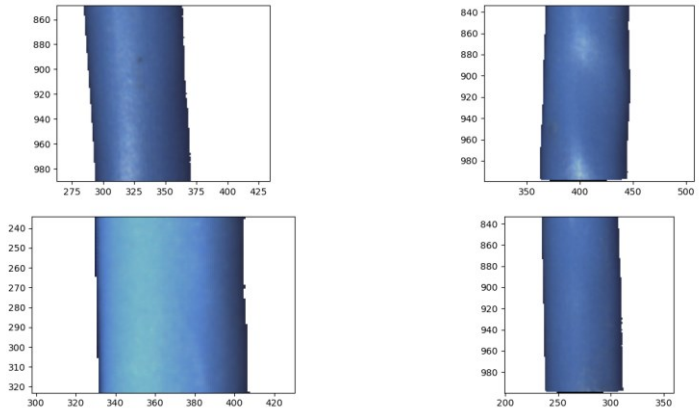
Prototyping

I took the approach of building a fast and dirty prototype to test out the different threshold functions and observe the results. As you can see in the figure below, I looked over the different color spectrums to get a sense of the contrast and light values across each image.



1 HSV Color Space and it's three channels

Of the different color spaces, LUV and HSV felt promising enough to pursue further. At this point I also set up at track bar functionality to speed up prototyping and went through the various techniques like trying to use directional sobel thresholds to mask over the prominent vertical elements of each image. The track bars also made quick work of the first task as I discovered I could apply a relatively good binary threshold over the cable itself as can be seen below.



2 Weighted cable image over the binary mask

The jaggedness of the binary mask was not ideal but I was later able to refine it further. This approach of course does have its own drawbacks as can be observed in one of the test cases where the defect occurs on the right side of the cable, which does affect the line continuity of the mask and will likely give incorrect measurements should the diameter be measured across one such case. This however could be filtered out if multiple diameter measurements were taken, averaged and compared over time to discard edge cases.

Detecting Defects

Through trial and error, there were two approaches which seemed most promising to me. One was an 'inRange' threshold over the HSV spectrum to form a tight lower and upper bound against the blue color in an HSV image. The other was a simple Canny edge detection followed by some further image processing to isolate the defects.

First approach: HSV Filtering

Filtering over the HSV looked most promising at a glance. As can be seen below, each defect was completely isolated in its mask which felt like a perfect solution. However, the contour shapes left by this filter were unworkable as the lines were perhaps a bit scattered around the defects and unconnected.

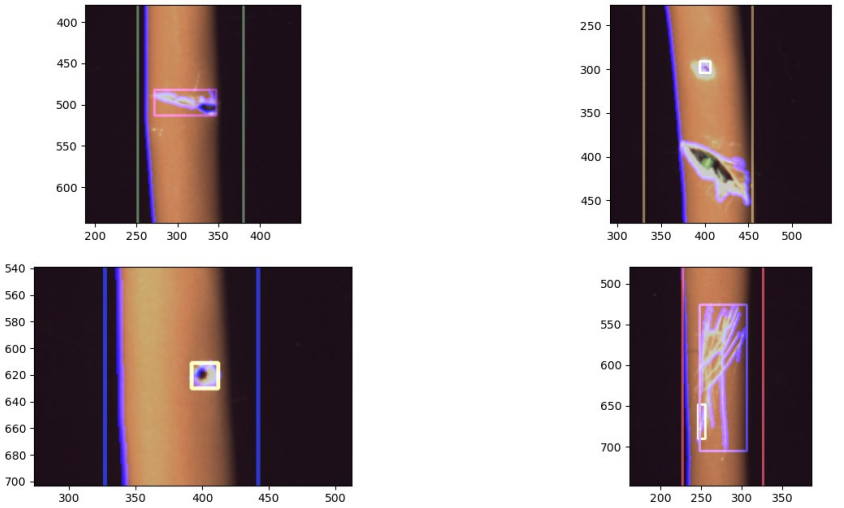
3 HSV filter results

This was partially due to the fact that some defect clarity was sacrificed in order to filter out the higher than normal white values in one of the images across the cable with the single pinhole. Sparingly, trying to find contours by both **external retrieval** and **simple approximation** yielded far too many uncluttered contours which would be difficult to work with.

Second Approach: Canny Edge Detection and Morphology

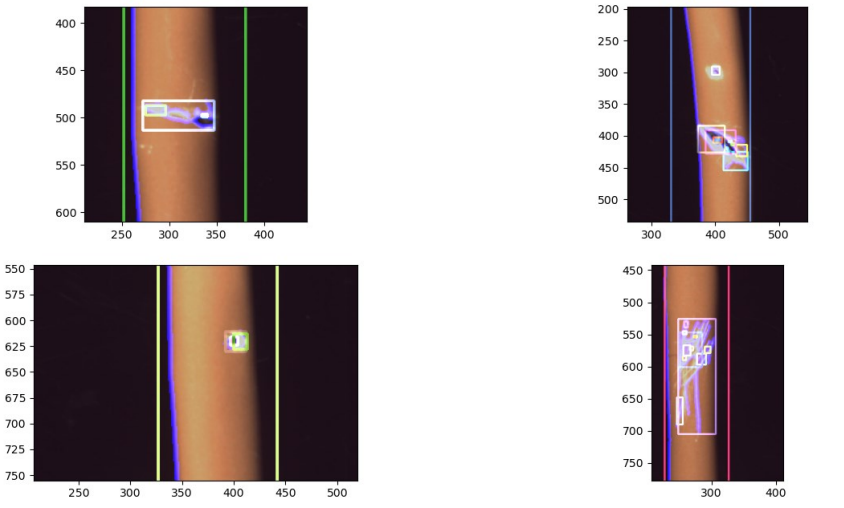
The other solution proved to be more attainable with even simpler thresholds to mess around with. Due to the natural image contrast, the grayscale values and applying Gaussian blur did reasonably well but it felt lacking compared to the HSV filter. However, upon applying a dilute morph to expand the canny lines

around the defects and further eroding the exterior ended up yielding a low amount of contours (2 per image in case of external retrieval) which was a significant improvement over the HSV filter, with the only caveat that one bounding box running the length of the cable would have to be filtered out.



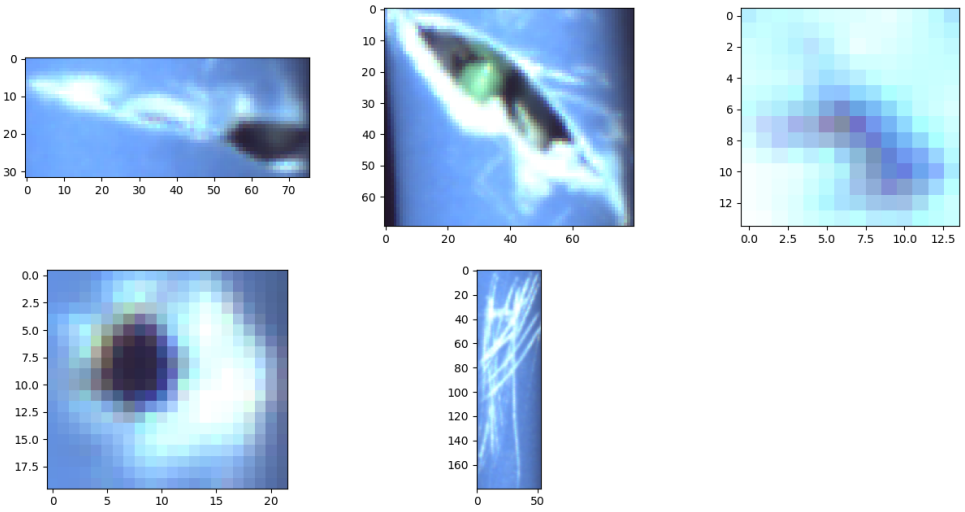
4 Bounding boxes around defects by external retrieval

There was one major issue however in that the second image (top right), the cut was close enough to the edge of the cable as to not be detected as an isolated contour. Several attempts to remedy this by threshold values had little effect and so I set on to another retrieval technique. However, not having really attempted to filter contours by hierarchy before, I elected to use a workaround when choosing to find contours by **Tree Retrieval**.



5 Bounding boxes by Tree Retrieval

This retrieval scenario would result in clusters of bounding boxes around each defect which was perfect. As a workaround, I drew the boxes as filled rectangles on an empty *cv::Mat* and passed it through a binary threshold and find contours by **External retrieval** which resulted in exactly one contour per defect. After which it was simple to obtain rotatedRects by corner approximation.



6 Isolated defects

Classifying Defects

Even disregarding the blurriness of the smaller pinhole defects, every idea I could come up with to classify them felt imperfect or too biased for the current set. Nevertheless, there were clear indicators and features with which I could work.

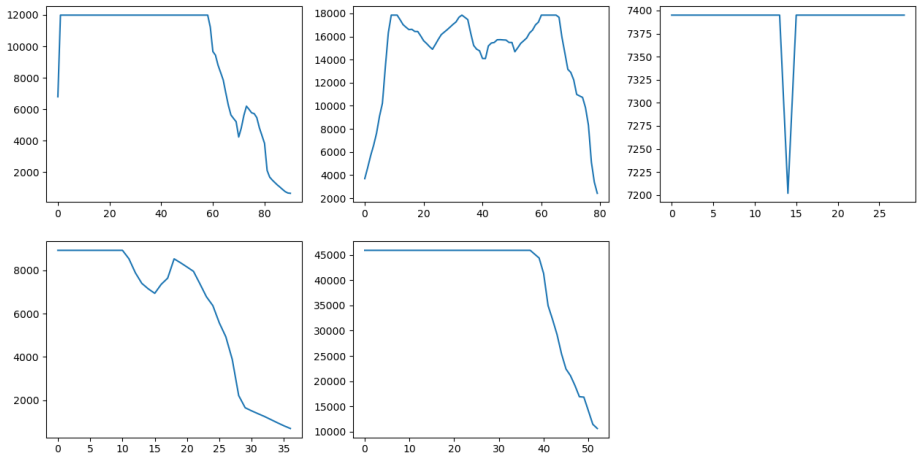
1. The high intensity of white color and absence of black regions across the scratch defect meant I could isolate it by histogram summation and eliminate it as a first option if it doesn't meet a (rather arbitrary, considering the small test images size) certain average color intensity parameter.
2. The cuts and pinholes were characterized by both have black regions in the center and their different aspect rations between the width and height.

There were several problems which I laid out as the roadblocks I'd have to find a way to resolve if I wanted a fit-all solution.

1. The green cable's intersection of the second cut meant any attempt to isolate its black regions would be cut in half which I was quickly able to see through a quick run of different tests.
2. The upper left cut's difficult location at the right side meant the scratch around it would be part of its classification as well, unless I could isolate the black region on its own.
3. The pixilation of the pinholes meant there wasn't much room to work around them without padding the box.

By using histograms, I was able to at least sanity check my assumption that scratch-only defects would have a higher color intensity across the y-axis on average as can be seen below. The bottom middle plot as a much higher average than all the others at 45k pixel values compared to the other defects which were at most below 20k.

Another think made clear by these plots is that the black regions for each defect cause a noticeable dip to appear after if we disregard the low values at the left and right side of each plot.



7 Color intensity histograms

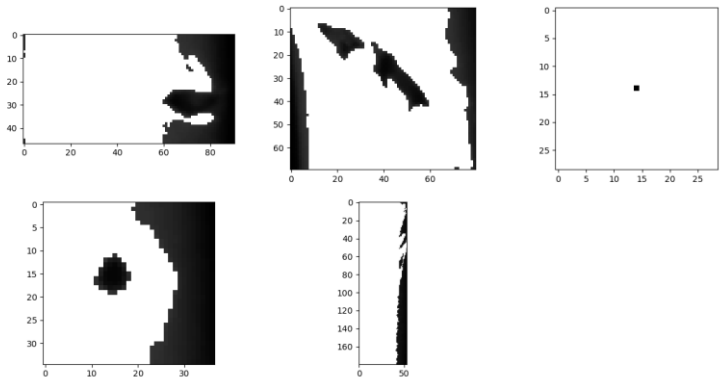
At this point, I did continue trying to isolate the black region so as to properly classify the top left cut without the scratch alongside it, but my attempts were unsuccessful which I'll outline below, but the final solution presented in the code stops here.

I don't think it's the best solution for this particular problem, but I thought it does quite well given the sample size of test images. At one point I even considered applying transformations to the test images and inputting them into a simple Conv network just for fun but I found myself quickly running out of time.

Extra: Isolating the Dark Regions of Cut Defects

First Attempt: Binary threshold

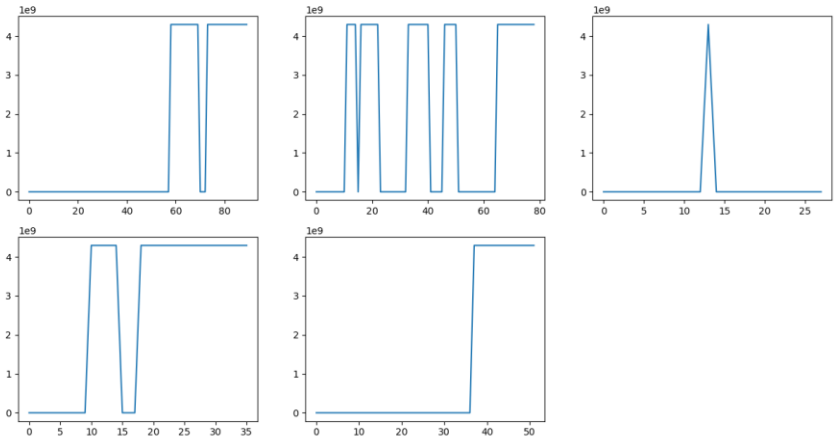
Binary thresholds proved ineffective as there was no balance to be found between the highly pixelated pinholes and the top left cable's persistence in morphing with the dark background, hence not allowing or contours to detect its corners properly.



8 Failed attempts at binary thresholds to extract the dark regions along

Second Attempt: differentiation

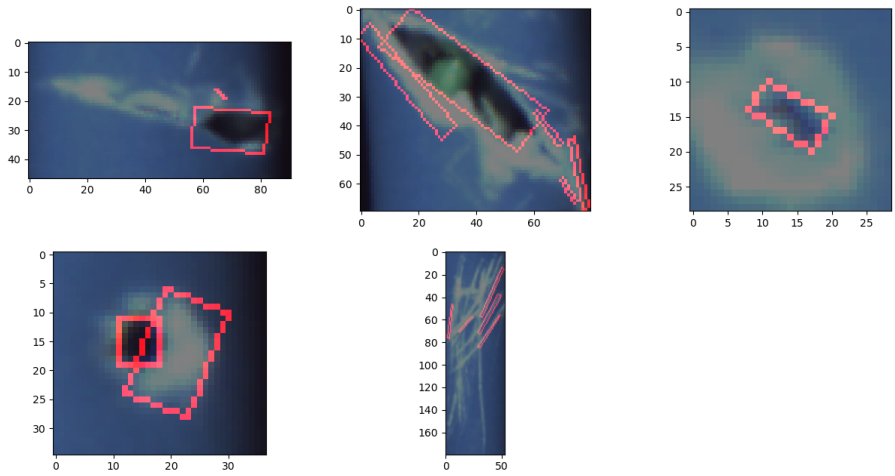
I thought a differentiation of the histogram intensity plots would allow me to at least grab the indices of the dark regions which would at least crop the search region further. The indices however were not accurate enough to attempt this.



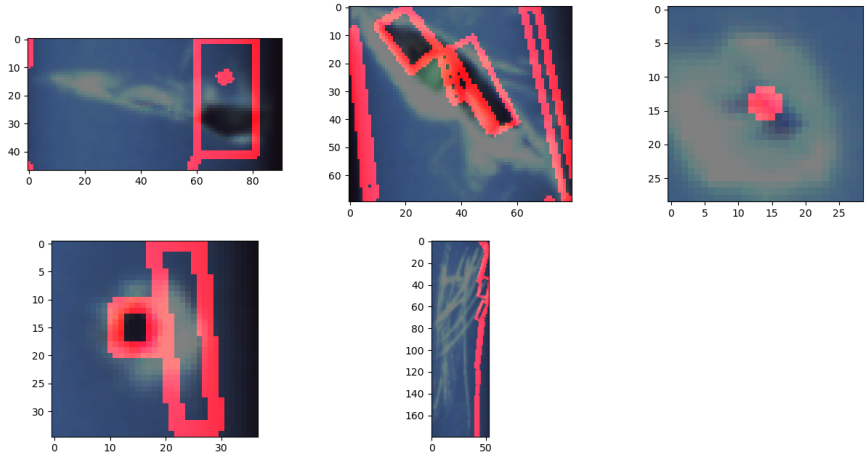
9 histogram differentiation

Third Attempt: Convex Hull

This was easily the most promising of the three as through trial and error I was able to decently cover the most egregious dark regions of both cuts as can be seen in the figure. I did however run in the problem of too many contours and decided it wasn't worth the effort to try and filter them out.



10 Black regions found



11 Black region contours

Source Code

main.cpp

```
#include <iostream>

#include <QApplication>

#include <opencv2/core/core.hpp>
#include <opencv2/core.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/features2d.hpp>
#include <opencv2/core/mat.hpp>

#include "mainwindow.h"

using namespace std;

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();

}
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QDebug>
#include <QFileDialog>
#include <QtGui>
#include <QMainWindow>
#include <QMessageBox>
#include <QPixmap>


#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
```

```
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:

    void on_measure_button_clicked();

    void measure_diameter();

    void on_find_defect_button_clicked();

    void find_defect();

    int classify_defect(cv::RotatedRect &ellipse, const cv::Mat &img);

    void on_load_image_button_clicked();

    void on_measure_save_button_clicked();

    void on_defect_save_button_clicked();

    void on_exit_button_clicked();

private:
    Ui::MainWindow *ui;
    cv::Mat img;

    int xsize = img.rows;
    int ysize = img.cols;

    cv::Mat measure_output;
    cv::Mat defect_outut;
};
#endif // MAINWINDOW_H
```

mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

static void grayscale(cv::Mat &src, cv::Mat &dst);
static void gaussianBlur(cv::Mat &src, cv::Mat &dst, int kernel);
static QImage MatToQImage(const cv::Mat& mat);

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}
```

```
MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::measure_diameter()
/* This function uses a precise threshold over the cable jacket to measure
 * the cable diameter in pixels. Due to few pixel inconsistencies in the
mask,
 * which are in the range of 1-3 pixels, it could be made better by averaging
 * the diameter above and below the line where the measurement is taken.
 *
 * Function will measure the diameter and display the result.
 */
{
    if (this->img.empty())
    {
        QMessageBox msgBox;
        msgBox.setText("No image loaded.");
        msgBox.exec();
        return;
    }

    // copy image to draw on
    cv::Mat measureImg = this->img.clone();
    // image containers
    cv::Mat gray;
    cv::Mat blur;
    cv::Mat thresh;

    ysize = measureImg.rows;

    // grayscale the image
    grayscale(measureImg, gray);

    // apply gaussian blur
    int blurKernel = 3;
    gaussianBlur(gray, blur, blurKernel);

    // apply a binary threshold mask tuned to cover the cable's length
    int threshold_low = 60;
    int threshold_high = 255;
    cv::threshold(blur, thresh, threshold_low, threshold_high,
cv::THRESH_BINARY);

    // vector of splits required to measure the cable at 3 different points
    int split = ysize/4;
    std::vector<int32_t> splits = {split, split*2, split*3};

    // line container
    cv::Mat line;
```

```
    // loop through each split and get the diameter
    for (int s : splits)
    {
        // line splice at split s
        line = thresh.row(s);

        // obtain non zero values representing the cable's length
        cv::Mat nonZero;
        cv::findNonZero(line, nonZero);

        // get the indices of the first zero on the left and right side
        int cableLeft = nonZero.at<std::int32_t>(0, 0);
        int cableRight = nonZero.at<std::int32_t>(nonZero.rows - 1, 0);

        // subtract the right from left to obtain diameter
        int diamaeter = cableRight - cableLeft;

        // drawing on measureImg
        std::string text = QString("Diameter:
%1").arg(diamaeter).toUtf8().constData();
        cv::line(measureImg, cv::Point(cableLeft, s), cv::Point(cableLeft -
15, s), cv::Scalar(0, 0, 255), 1);
        cv::line(measureImg, cv::Point(cableRight, s), cv::Point(cableRight +
15, s), cv::Scalar(0, 0, 255), 1);
        cv::putText(measureImg, text, cv::Point(cableRight + 25, s),
cv::FONT_HERSHEY_DUPLEX, 0.5, cv::Scalar(0, 0, 255));

        // image display
        QImage measureImage = MatToQImage(measureImg);
        this->measure_output = measureImg;
        this->ui->image_display->setPixmap(QPixmap::fromImage(measureImage));
    }
}

void MainWindow::on_measure_button_clicked()
{
    measure_diameter();
}

int MainWindow::classify_defect(cv::RotatedRect &rectangle, const cv::Mat
&img)
/* Input
 *  ellipse: a rotated rectangle where the defect was detected.
 *  img: the original img Mat required for warping.
 *
 * Output
 *  int corrosponding to the defect classification.
 *
 * Function will warp the rotatedRect into an up-right rectangle.
 * This will allow us to average the sum of each column in the defect area
 * for classification.
 */
{
```

```
    cv::Mat matrix, warped, warpedGray;

    float h = rectangle.size.height;
    float w = rectangle.size.width;

    cv::Mat hist(cv::Mat::zeros(1,w,CV_32S));

    // points of image to be warped
    cv::Point2f src[4];
    rectangle.points(src);
    // warp destination
    cv::Point2f dst[4] = { {0.0f, h - 1},
                           {0.0f, 0.0f},
                           {w - 1.0f, 0.0f},
                           {w - 1.0f, h - 1.0f} };

    // warp matrix
    matrix = cv::getPerspectiveTransform(src, dst);
    cv::warpPerspective(img, warped, matrix, cv::Point(w,h));

    // grayscale warp so white pixel intensity can be summed
    grayscale(warped, warpedGray);

    // sum all the pixel values in each column then take the mean
    cv::reduce(warpedGray, hist, 0, 0, CV_32S);
    int avgIntensity = cv::sum(hist)[0]/w;

    // hyperparameter for scratches
    if (avgIntensity > 18000)
        return 2;  //scratch
    else
    {
        float ratio;
        if (h > w)
            ratio = w/h;
        if (w >= h)
            ratio = h/w;
        if (ratio <= 0.85)
            return 1; // cut
        else
            return 0; // pinhole
    }
}

void MainWindow::find_defect()
{
    if (this->img.empty())
    {
        QMessageBox msgBox;
        msgBox.setText("No image loaded.");
        msgBox.exec();
        return;
    }
```

```
    // copy image to draw on
    cv::Mat defectImg = this->img.clone();
    // image containers
    cv::Mat gray;
    cv::Mat blur;
    cv::Mat edges;

    int ysize = img.rows;
    int xsize = img.cols;

    // grayscale image
    grayscale(img, gray);

    // apply gaussian blur
    int blurKernel = 3;
    gaussianBlur(gray, blur, blurKernel);

    // use canny detection to find edges of defects
    int canny_low = 60;
    int canny_high = 255;
    cv::Canny(blur, edges, canny_low, canny_high);

    // apply morphology to accentuate edges
    cv::Mat im_erode;
    cv::Mat im_dilate;

    cv::Mat morphKernel = cv::getStructuringElement(cv::MORPH_RECT,
cv::Size(3, 3));
    cv::dilate(edges, im_dilate, morphKernel);
    cv::erode(im_dilate, im_erode, morphKernel);

    // containers for cv::findContours() output
    std::vector<std::vector<cv::Point>> contours;
    std::vector<cv::Vec4i> hierarchy;

    // find contours
    cv::findContours(im_erode, contours, hierarchy, cv::RETR_TREE,
cv::CHAIN_APPROX_SIMPLE);

    // defining a new black image to draw clustered rectangle on in order to
combine them later
    cv::Mat groupImg = cv::Mat(ysize, xsize, CV_8UC3, cv::Scalar(0,0,0));

    // loop over the contours to find bounding boxes and draw rectangles
    for (int i = 0; i < contours.size(); i++)
    {
        // containers
        std::vector<std::vector<cv::Point>> contourPoly(contours.size());
        std::vector<cv::Rect> boundingRectangles(contours.size());

        // find arclength and approximate poly corners for contours
        float perimeter = cv::arcLength(contours[i], true);
```

```
        cv::approxPolyDP(contours[i], contourPoly[i], 0.02*perimeter, true);

        // find the bounding rectangles
        boundingRectangles[i] = boundingRect(contourPoly[i]);
        // calculate area of each box to filter out faulty detections
        float height = boundingRectangles[i].height;
        float width = boundingRectangles[i].width;
        float area = height * width;

        // draw only the good boxes
        if (area < 10000)
           cv::rectangle(groupImg, boundingRectangles[i].tl(),
boundingRectangles[i].br(), cv::Scalar(255,255,255), -1);
    }

    /* In order to combine the multiple bounding boxes cluttered over one
defect,
     * we threshold and find contours again through cv::RETER_EXTERNAL which
will
     * simply combine all the boxes into one large contour around the fault.
     * Much of the process is repeated with slightly different parameters.
     * */

    // grouping image containers
    cv::Mat groupGray;
    cv::Mat groupBlur;
    cv::Mat groupThresh;

    // grayscale the rectangles image
    grayscale(groupImg, groupGray);

    // apply gaussian blur
    gaussianBlur(groupGray, groupBlur, blurKernel);

    // apply a binary threshold mask
    int groupThresholdLow = 50;
    int groupThresholdHigh = 255;
    cv::threshold(groupBlur, groupThresh, groupThresholdLow,
groupThresholdHigh, cv::THRESH_BINARY);

    // containers for cv::findContours() output
    std::vector<std::vector<cv::Point>> groupContours;
    std::vector<cv::Vec4i> groupHierarchy;

    // find contours
    cv::findContours(groupThresh, groupContours, groupHierarchy,
cv::RETR_EXTERNAL, cv::CHAIN_APPROX_NONE);

    // vector to store ellipses data for final defect classification
    std::vector<cv::RotatedRect> minEllipse(contours.size());

    // loop over new contours
    for (int i=0; i < groupContours.size(); i++)
```

```
    {
        // containers
        std::vector<std::vector<cv::Point>> contourPoly(contours.size());
        std::vector<cv::Rect> boundingRectangles(contours.size());

        // approximate poly
        cv::approxPolyDP(groupContours[i], contourPoly[i], 1, true);

        boundingRectangles[i] = boundingRect(contourPoly[i]);

        // fit ellipses over the defects
        minEllipse[i] = fitEllipse(groupContours[i]);

        // draw ellipses on img
        ellipse(defectImg, minEllipse[i], cv::Scalar(0,0,255), 2);
        // classify which type of defect
        int defect = classify_defect(minEllipse[i], img);
        std::string text;
        switch (defect)
        {
        case 0:
            text = "Defect: Pin Hole";
            break;
        case 1:
            text = "Defect: Cut";
            break;
        case 2:
            text = "Defect: Scratch";
        }

        // drawing functions
        int center_x = minEllipse[i].center.x;
        int center_y = minEllipse[i].center.y;

        cv::putText(defectImg, text, cv::Point(center_x + 60, center_y),
cv::FONT_HERSHEY_DUPLEX, 0.5, cv::Scalar(0, 0, 255));
        QImage defectImage = MatToQImage(defectImg);
        this->ui->image_display->setPixmap(QPixmap::fromImage(defectImage));
        this->defect_outut = defectImg;
    }
}

void MainWindow::on_find_defect_button_clicked()
{
    find_defect();
}

void MainWindow::on_load_image_button_clicked()
{
    // image path
   QString filePath = QFileDialog::getOpenFileName(this,
        tr("Open Image"), "/Desktop", tr("Image Files (*.png *.jpg *.bmp)"));
```

```
    std::string imagePath = filePath.toLocal8Bit().constData();

    // read image
    this->img = imread(imagePath, cv::IMREAD_COLOR);

    // if image fails to load for any reason, display message
    if (img.empty())
    {
        QMessageBox msgBox;
        msgBox.setText("Could not load image.");
        msgBox.exec();
    }

    //display image
    QImage display;
    display.load(filePath);
    QPixmap displayMap;
    displayMap.fromImage(display);
    this->ui->image_display->setPixmap(QPixmap::fromImage(QImage(img.data,
img.cols, img.rows, img.step, QImage::Format_BGR888)));
}

// Image functions
static void grayscale(cv::Mat& src, cv::Mat& dst)
{
    cv::cvtColor(src, dst, cv::COLOR_BGR2GRAY);
    return;
}

static void gaussianBlur(cv::Mat& src, cv::Mat& dst, int kernel)
{
    cv::GaussianBlur(src, dst, cv::Size(kernel, kernel), 0);
    return;
}

static QImage MatToQImage(const cv::Mat& mat)
// credit to eyllanesc on stackoverflow for this great function!
{
    // 8-bits unsigned, NO. OF CHANNELS=1
    if(mat.type()==CV_8UC1)
    {
        // Set the color table (used to translate colour indexes to qRgb
values)
        QVector<QRgb> colorTable;
        for (int i=0; i<256; i++)
            colorTable.push_back(qRgb(i,i,i));
        // Copy input Mat
        const uchar *qImageBuffer = (const uchar*)mat.data;
        // Create QImage with same dimensions as input Mat
        QImage img(qImageBuffer, mat.cols, mat.rows, mat.step,
QImage::Format_Indexed8);
        img.setColorTable(colorTable);
        return img;
```

```
    }
    // 8-bits unsigned, NO. OF CHANNELS=3
    if(mat.type()==CV_8UC3)
    {
        // Copy input Mat
        const uchar *qImageBuffer = (const uchar*)mat.data;
        // Create QImage with same dimensions as input Mat
        QImage img(qImageBuffer, mat.cols, mat.rows, mat.step,
QImage::Format_RGB888);
        return img.rgbSwapped();
    }
    return QImage();
}

void MainWindow::on_measure_save_button_clicked()
{
    if (this->measure_output.empty())
    {
        measure_diameter();
    }
    QString fileName = QFileDialog::getSaveFileName(this, "Save File",
                                "/home/measureoutput.jpg",
                                "Images (*.png *.xpm *.jpg)");
    cv::imwrite(fileName.toUtf8().constData(), measure_output);
}

void MainWindow::on_defect_save_button_clicked()
{
    if (this->defect_outut.empty())
    {
        find_defect();
    }
    QString fileName = QFileDialog::getSaveFileName(this, "Save File",
                                "/home/defectoutput.jpg",
                                "Images (*.png *.xpm *.jpg)");
    cv::imwrite(fileName.toUtf8().constData(), measure_output);
}

void MainWindow::on_exit_button_clicked()
{
    this->close();
}
```