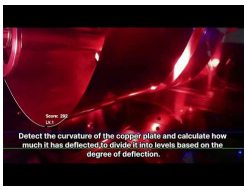
LAB: Tension Detection of Rolling Metal Sheet

Date: 2024-May-07

Author: Gyeonheal An (21900416)

Github: https://github.com/AnGyeonheal/DLIP_GH

Demo Video: 

I. Introduction

1. Objective

This LAB deals with measuring the deflection of a rolling metal sheet caught by a roller using a simple machine vision system.

The tension during rolling can be determined by assessing the curvature of the metal sheet using the camera.

Both the chamber's surface and the metal sheet exhibit specular reflection, potentially resulting in virtual objects appearing in captured images. Therefore, a set of machine vision algorithms must be developed to accurately identify the metal sheet's edge and determine its curvature and tension levels.

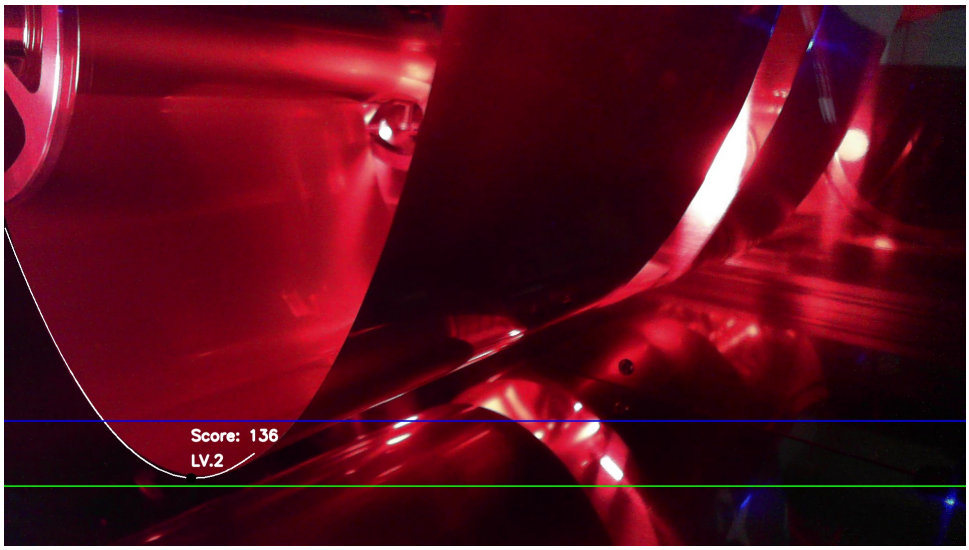


Fig 1. LAB: Tension Detection of Rolling Metal Sheet

2. Problem Conditions

- Measure the metal sheet tension level from Level 1 to Level 3.

    - Use the minimum y-axis position of the metal sheet curvature
    - Level 1: >250px from the bottom of the image
    - Level 2: 120~250 px from the bottom of the image
    - Level 3: < 120 px from the bottom of the image
- Display the output on the raw image
    - Tension level: Level 1~3
    - Score: y-position [px] of the curvature vertex from the bottom of the image
    - Curvature edge

3. Preparation

Software Installation

- Pycharm JetBrain

Dataset

Three levels of rolling metal sheet. + Video

Dataset link: Download the test image

II. Algorithm

1. Overview

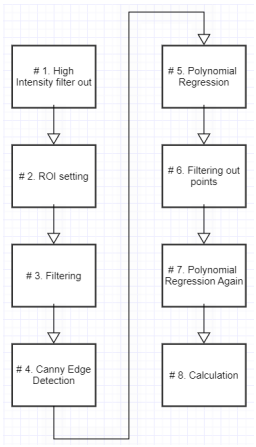


Fig 2. Algorithm Flowchart

The algorithm was developed to achieve the specified objectives seamlessly even in videos where the curvature of the copper sheet changes in real-time.

The primary challenge in videos with changing frames was the reflection of lighting on the copper plate. To mitigate sensitivity to lighting effects, the following Image Processing steps were implemented:

Regions with strong intensity caused by reflected lighting were removed, and a Region of Interest (ROI) was defined. Appropriate filters and edge detection techniques were applied to obtain the coordinates required for Polynomial Regression.

Subsequently, Polynomial Regression was used to generate the graph of a 2nd degree polynomial. Outlying coordinates distant from this graph were filtered out, and regression was conducted again.

Following this, another round of Polynomial Regression was performed to identify the local minimum of the refined 2nd degree polynomial graph, enabling the calculation of the Level and Score of the rolling metal sheet.

2. Procedure

#1. High Intensity Filter Out

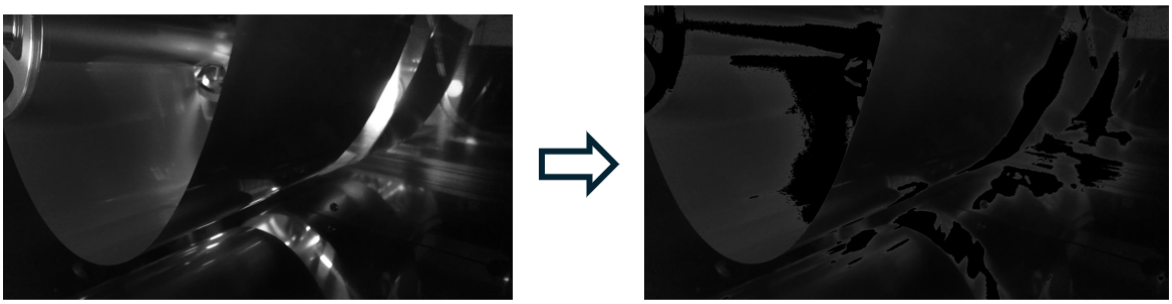


Fig 3. High Intensity Thresholding

In the above image, you can see pixels with intense intensity due to lighting reflecting on the copper plate.

These pixels interfere with the analysis of the shape of the copper sheet, so they have been processed to black by adjusting the threshold value.

```
# 1. High Intensity filter out
threshold_value = 50
src[src >= threshold_value] = 0
```

#2. ROI Setting

Fig 4. ROI (Region of Interest)

We analyzed the curves of the copper plate by setting the Region of Interest (ROI) for focused analysis.

If the ROI is excessively large, errors may occur in the results due to interference, while if it is excessively small, there may not be enough Edge coordinates obtained for regression.

Therefore, we specified an ROI of 500 x 630 size, as shown in the above image.

```
# 2. ROI setting
x, y, w, h = 0, 400, 500, 630
roi = src[y:y + h, x:x + w]
```

#3. Filtering

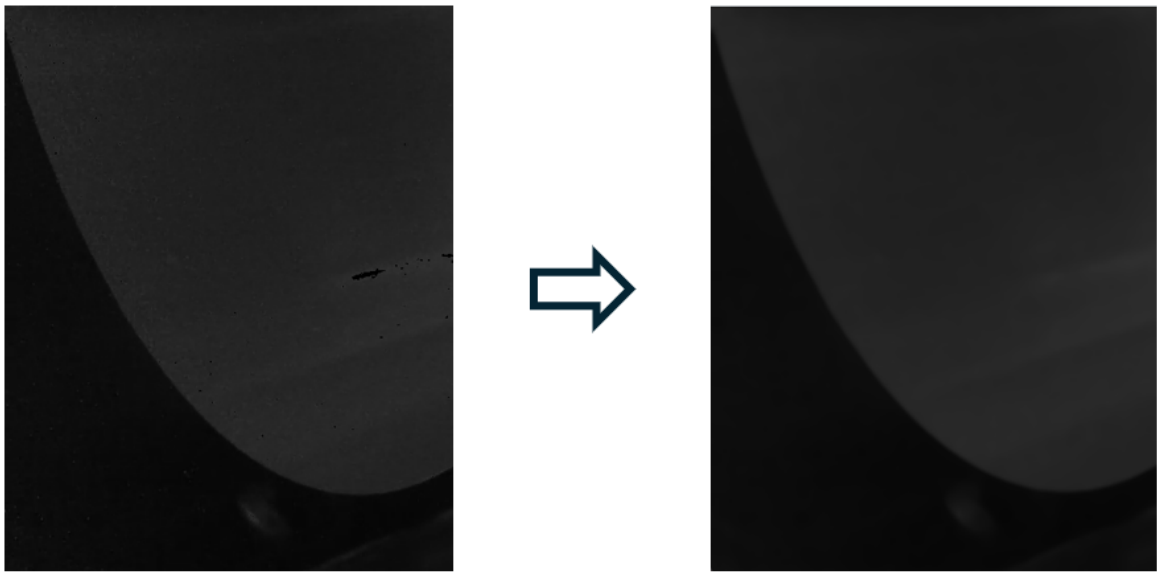


Fig 5. Filtered Image

In the above image, you can observe very small particles.

These particles can be detected in Edge Detection, so we used Median Filter and mean Blur to remove them.

```
# 3. Filtering
for i in range(3):
blur = cv.medianBlur(roi, 15)
blur = cv.blur(blur, (5, 5))
```

#4. Canny Edge Detection

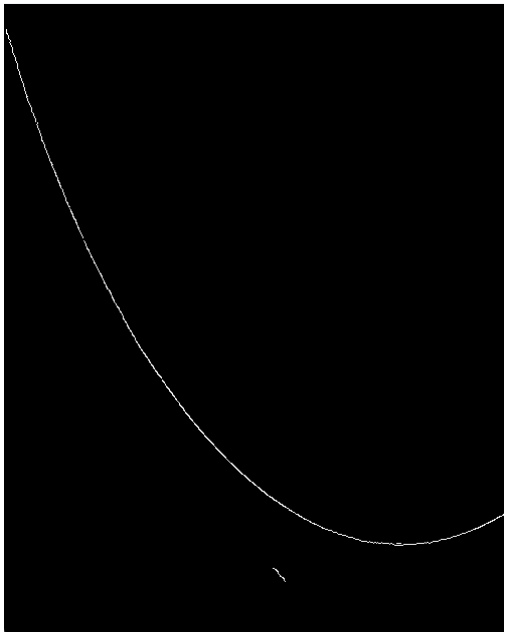


Fig 6. Edge Detection (Canny)

The image above is a photo detecting the curves of the copper sheet through Edge Detection.

While it's possible to divide the Level and assign a Score based on this image, since the curvature of the copper sheet changes, the reflection of the lighting and the intensity of the curve also vary. Therefore, we need to create a detection system that is less sensitive to these changes.

```
# 4. Canny Edge Detection
canny = cv.Canny(blur, 0, 30)
y_coords, x_coords = np.where(canny == 255)
```

#5. Polynomial Regression

The function used for Polynomial Regression is the `polyfit()` function, which represents a polynomial regression model. This model is utilized to explain and predict the relationship between observed data points using polynomials. Since the curve of the copper sheet we aim to obtain is similar to a 2nd-degree polynomial, we used a 2nd-degree polynomial for the regression.

```
# 5. Polyfit (2nd Order)
coefficients = np.polyfit(x_coords, y_coords, 2)
polynomial = np.poly1d(coefficients)
y_fit = polynomial(x_coords)
```

#6. Filtering out points

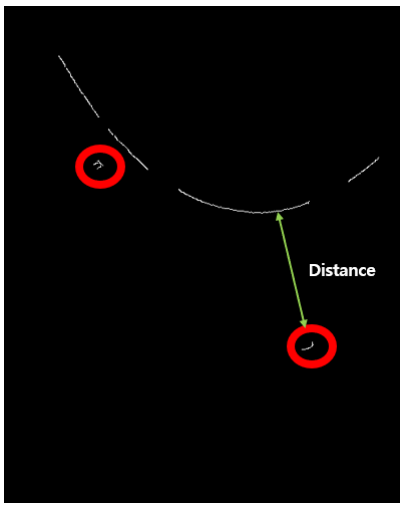


Fig 7. Explain Image of filtering out points

In Polynomial Regression, as shown in the image above, there are other points outside the curve.

These points, which were not filtered out during the filtering and thresholding processes, can introduce errors in drawing the curve if they are included in the calculation of polyfit.

To address this issue, we removed these outlier points during the polyfit calculation process based on how far they are from the 2nd-degree function obtained from the initial Polynomial Regression.

```
# 6. Filtering out points that are far away from the Polyfit graph beyond a
threshold.
distances = np.abs(y_coords - y_fit)
distance_threshold = 30

x_filtered = []
y_filtered = []
for i in range(len(x_coords)):
if distances[i] < distance_threshold:
x_filtered.append(x_coords[i])
y_filtered.append(y_coords[i])
```

#7. Polynomial Regression Again

The filtered coordinates obtained from the above process were subjected to Polynomial Regression again to obtain a new 2nd-degree polynomial.

```
# Refitting with refined data
coefficients_filtered = np.polyfit(x_filtered, y_filtered, 2)
polynomial_filtered = np.poly1d(coefficients_filtered)
```

#8. Calculation

We found the local minimum of the 2nd-degree polynomial obtained above and divided the Level according to the Problem Condition (I.2) described earlier, and assigned a Score.

```
for i in range(len(x_new) - 1):
    cv.line(img, (int(x_new[i]), int(y_new[i])), (int(x_new[i + 1]), int(y_new[i
+ 1])), (255, 255, 255), 2)
    if (max_y < int(y_new[i])):  # Finding the point with the lowest y-value on
the graph
        max_y = int(y_new[i])
        max_x = int(x_new[i])
```

```
    if max_y > height - 120:
        cv.putText(img, "LV.3", (max_x, max_y - 25), cv.FONT_HERSHEY_SIMPLEX,
1.0, color=(255, 255, 255), thickness=3,
                   lineType=cv.LINE_AA)
    elif max_y < height - 120 and max_y > height - 250:
        cv.putText(img, "LV.2", (max_x, max_y - 25), cv.FONT_HERSHEY_SIMPLEX,
1.0, color=(255, 255, 255), thickness=3,
                   lineType=cv.LINE_AA)
    elif max_y < height - 250:
        cv.putText(img, "LV.1", (max_x, max_y - 25), cv.FONT_HERSHEY_SIMPLEX,
1.0, color=(255, 255, 255), thickness=3,
                   lineType=cv.LINE_AA)
```

III. Result and Discussion

1. Final Result

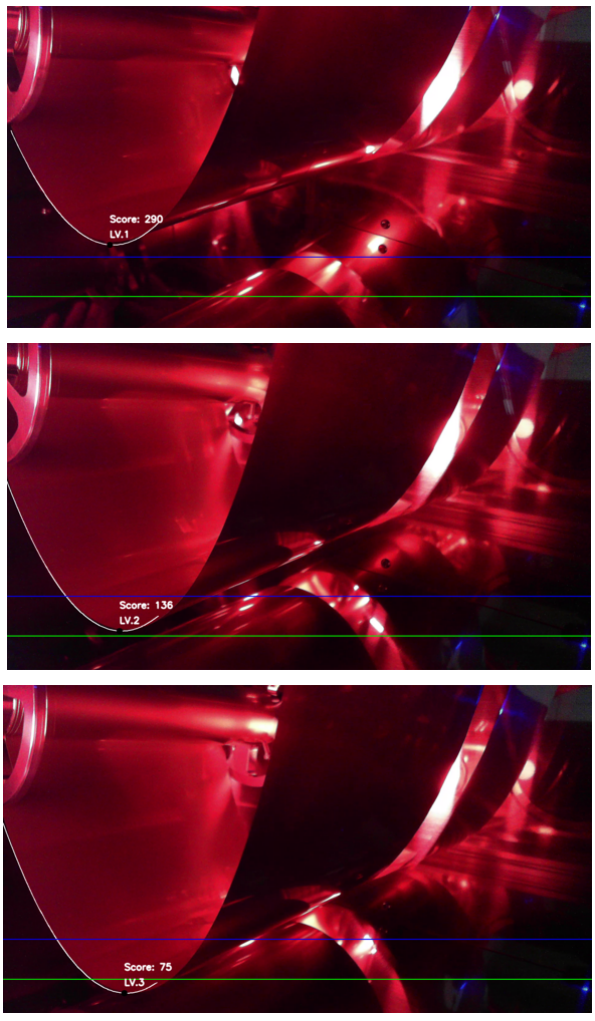
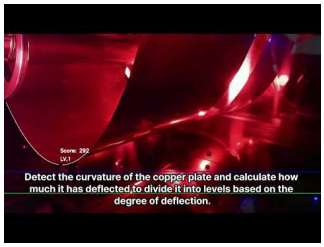


Fig 8. Result Image (LV1, LV2, LV3).

Demo Video Embedded: 

2. Discussion

Items	Accuracy
Level 1.	100%
Level 2.	100%
Level 3.	100%
Video	100%

Table 1. Accuracy of Algorithm

It was observed that the algorithm accurately represents regression graphs even in videos with continuous frame changes, similar to the provided images of the three levels.

Therefore, the algorithm successfully satisfies the conditions of this Lab.

Conclusion

The algorithm was developed to perform the specified objectives even on videos where the curvature of the copper sheet changes in real-time.

The biggest challenge in videos with changing frames was the reflection of lighting on the copper plate. The most crucial process in this LAB was to make the results insensitive to the effects of lighting. Recognizing that copper plates have a very high reflectivity, making the intensity of lighting very high, we adjusted the high-intensity pixels using thresholding. Then, by defining the Region of Interest (ROI), we limited the pixel group we wanted to analyze and were able to obtain coordinates of the curves on the copper plate through image preprocessing. We further filtered the obtained coordinates based on the distance between pixels and conducted polynomial regression analysis to determine the amount of deformation in the copper sheet.

Through this LAB, we learned that the images we need to analyze in practice are affected by lighting conditions and require careful preprocessing to obtain accurate results.

Appendix

```
# * DLIP_LAB3_Tension Detection of Rolling Metal Sheet
# * author: Gyeonheal An
# * Date: 2024-05-07

import numpy as np
import cv2 as cv
```

```
# Open video
cap = cv.VideoCapture("LAB3_Video.mp4")

while True:
    ret, img = cap.read()  # Read the frame

    if not ret:
        break

    # BGR to GrayScale
    src = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    # 1. High Intensity filter out
    threshold_value = 50
    src[src >= threshold_value] = 0

    # 2. ROI setting
    x, y, w, h = 0, 400, 500, 630
    roi = src[y:y + h, x:x + w]

    # 3. Filtering
    for i in range(3):
        blur = cv.medianBlur(roi, 15)
    blur = cv.blur(blur, (5, 5))

    # 4. Canny Edge Detection
    canny = cv.Canny(blur, 0, 30)
    y_coords, x_coords = np.where(canny == 255)

    # 5. Polyfit (2nd Order)
    coefficients = np.polyfit(x_coords, y_coords, 2)
    polynomial = np.poly1d(coefficients)
    y_fit = polynomial(x_coords)

    # 6. Filtering out points that are far away from the Polyfit graph beyond a
threshold.
    distances = np.abs(y_coords - y_fit)
    distance_threshold = 30

    x_filtered = []
    y_filtered = []
    for i in range(len(x_coords)):
        if distances[i] < distance_threshold:
            x_filtered.append(x_coords[i])
            y_filtered.append(y_coords[i])

    # Refitting with refined data
    coefficients_filtered = np.polyfit(x_filtered, y_filtered, 2)
    polynomial_filtered = np.poly1d(coefficients_filtered)

    # Drawing a Polynomial Regression graph
    x_new = np.linspace(min(x_filtered), max(x_filtered), 100)
    y_new = polynomial_filtered(x_new) + 400

    max_x = 0
    max_y = 0
    height = img.shape[0]
    width = img.shape[1]
```

```
    for i in range(len(x_new) - 1):
        cv.line(img, (int(x_new[i]), int(y_new[i])), (int(x_new[i + 1]),
int(y_new[i + 1])), (255, 255, 255), 2)
        if (max_y < int(y_new[i])):  # Finding the point with the lowest y-value
on the graph
            max_y = int(y_new[i])
            max_x = int(x_new[i])
    cv.circle(img, (max_x, max_y), 10, (0, 0, 0), -1)
    cv.putText(img, f"Score: {height - max_y}", (max_x, max_y - 75),
cv.FONT_HERSHEY_SIMPLEX, 1.0, (255, 255, 255),
               thickness=3, lineType=cv.LINE_AA)

    cv.line(img, (0, height - 250), (img.shape[1], height - 250), (255, 0, 0),
2)
    cv.line(img, (0, height - 120), (img.shape[1], height - 120), (0, 255, 0),
2)

    if max_y > height - 120:
        cv.putText(img, "LV.3", (max_x, max_y - 25), cv.FONT_HERSHEY_SIMPLEX,
1.0, color=(255, 255, 255), thickness=3,
                   lineType=cv.LINE_AA)
    elif max_y < height - 120 and max_y > height - 250:
        cv.putText(img, "LV.2", (max_x, max_y - 25), cv.FONT_HERSHEY_SIMPLEX,
1.0, color=(255, 255, 255), thickness=3,
                   lineType=cv.LINE_AA)
    elif max_y < height - 250:
        cv.putText(img, "LV.1", (max_x, max_y - 25), cv.FONT_HERSHEY_SIMPLEX,
1.0, color=(255, 255, 255), thickness=3,
                   lineType=cv.LINE_AA)

    cv.imshow("Regression Image", img)

    if cv.waitKey(1) & 0xFF == ord('q'):  # Press 'q' to quit
        break

cap.release()
cv.destroyAllWindows()
```