AINT308 - OpenCV Assignment 1 2022

Student No. 10618407
School of Engineering,
Computing and Mathematics
University of Plymouth
Plymouth, Devon

Fig. 1: Example Car Image

```
for (int i = 0;i<Car.rows; i++) { // runs through all the rows in the image
    for (int j = 0;j < Car.cols; j++) { // runs through all the columns in the image
        Vec3b PixelValue = Car.at<Vec3b>(i,j); // stores the RBG values in the PixelValue vector

        // checks if the blue in the pixel is at least 1.5x higher than the other colours
        if((PixelValue[0] > 1.5 * PixelValue[1])&&(PixelValue[0] > 1.5 * PixelValue[2])){
            // pixel is blue
            blueCount++; // add the pixel to the blue count
        }
        // checks if the green in the pixel is at least 1.5x higher than the other colours
        else if((PixelValue[1] > 1.5 * PixelValue[0])&&(PixelValue[1] > 1.5 * PixelValue[2])){
            // pixel is green
            greenCount++; // add the pixel to the green count
        }

        // checks if the red in the pixel is at least 1.5x higher than the other colours
        else if((PixelValue[2] > 1.5 * PixelValue[1])&&(PixelValue[2] > 1.5 * PixelValue[0])){
            // pixel is red
            redCount++; // add the pixel to the red count
        }
        else {
            // inconclusive - do not add any values to the count
        }
    }
}
```

Fig. 2: Task 1 Code - Colour Checking

functionalities?

***Abstract*—Machine vision is a mature technology that is becoming more prevalent within modern engineering practises. It is being utilised more in the rapidly evolving fields of autonomy and automation. This report outlines some of the functionality of a popular C/C++ based computer visions library *OpenCV*. The Assignment has been split into three tasks; Task 1, Task 2, and Task 3. The first task, Task 1, is to evaluate the colors of pixels in a picture to determine the colour of a given object in the frame (car). The second task, Task 2, was to track on object across multiple frames of a video to track its motion (swinging pendulum). The final task, Task 3, was to identify and cross correlate components on a circuit to check for any missing components.**

Keywords:

Computer Vision, OpenCV, Object Detection, C++, Object Tracking

I. Task 1: Colour Sorter

A. Introduction

The first task was to evaluate the colour of a given object within a frame color within the RGB colour space. For this task, the colour of the car within a frame is being evaluated. An example of the type of image can be seen in Fig. 1.

B. Solution

Fig. 2 shows the solution to the first task, the Colour Sorter.

OpenVC includes a built in method for retrieving the values of a pixel in an image. The code begins by iterating through two loops, one to look at all the rows and the second to go through all the columns. If one of the R(red), G(green), or B(blue) value is more than 1.5 times larger than the others the pixel is deemed to be that colour. This is done to help distinguish between the vibrant colour of the car and what could be considered the background in the image. Once this pixel has been checked, if the colour is within the given threshold, it is added to a running total for each of the colours.

The value of 1.5 was chosen because it was deemed to be a significant enough proportion larger that it was definitive in selecting the colour. If the value was too high, none of the colours would ever have been selected. If the value was too low, it would potentially get confused between the RBG values, especially for colours with a similar spread.

After the whole image is checked, the counts are totalled and the highest value is deemed to be the colour of the car. This can be seen in Fig. 3.

To test the algorithm further, a larger dataset of pictures of cars was used. 40 more images of cars were added to the dataset, alongside the original 30 that was provided. The new dataset was initially used to train and test machine learning models. [1] These were added to test how to algorithms functionality was maintained up without the use of highly cropped and optimised inputs.

Overall this method worked for the dataset that was provided, but not as well for the added dataset.

INCLUDE VALUES OF FINDINGS?

The original dataset was selected such that the car made up a majority of the frame and the background was, for the most part plain as to not sway the results.

A video of the code in operation can be seen in this link

```
    cout<<"The blue value is " << blueCount <<endl;
    cout<<"The green value is " << greenCount <<endl;
    cout<<"The red value is " << redCount <<endl;

    // if blue is the higest count the car is blue
    if((blueCount > redCount)&&(blueCount > greenCount)){
         cout<< "The car is blue" <<endl;
    }

    // if green is the higest count the car is green
    else if((greenCount > redCount)&&(greenCount > blueCount)){
        cout<< "The car is green" <<endl;
    }

    // if red is the higest count the car is red
    else if((redCount > blueCount)&&(redCount > greenCount)){
        cout<< "The car is red" <<endl;
    }

    //display the car image untill x is pressed
    while(waitKey(10)!='x'){
        imshow("Car", Car);
    }

}
```

Fig. 3: Task 1 Code - Output

C. Further Improvements

Although this task worked for the given dataset, there are ways to improve the colour detection accuracy. RGB models are typically not used for this type of task, with the Hue, Saturation, and Value (HSV) model being preferred. Initially the RGB model was built and optimised to be used on display screens, whereas the HSV model was developed to mimic how a human interpret colours. Since HSV models are able to separate colour vales, saturation, and lightness, operations can be more easily performed on the hue itself. Because we are more interested in the colour of the cars and not the lightness, a HSV colour model would be an objectively better solution. [2]

Another improvement is to make the algorithm detect the outline of the car. This would eliminate the issues caused by having a background that could be mistaken for the body of the car. An example of this can be seen in Fig 4. Open CV is capable of detecting cars in images and videos, albeit trivially. Although trivial, it could be used to draw a box around the car thus eliminating the background surrounding the car. [3]

The red car with a green background may confuse the simple sorting algorithm and lead to a false result (the algorithm may think the car is green). This is the issue with using such a basic classification algorithm.

The final improvement that could be partially solved by using the HSV colour model, but that would also require a far more complex solving algorithm [4], is that cars are not just red, green, or blue. They come in many different colours (including white, black, and silver) and also in many different shades and brightness of colour. By isolating just the RGB values and evaluating which is highest, a lot of granularity is lost. Using this method it is impossible to detect the shade of the car and even the type of red, green, or blue the car may be, if it is one of these colours at all. Using

Fig. 4: Example Car with undesirable background

the blend of colours and evaluating them against known RGB values, further colours of car could have been identified with a significantly higher range of colours being available (Up to 16M different combinations for an 8-bit RGB value). This method however would require some form of 'lookup table' that would be used to compare the RGB pixel value to. This adds complexity to the task, but would allow for a better overall result.

D. Conclusion

Overall this task was successful for the limited dataset provided. This dataset only included Red, Green, and Blue cars. These cars were placed in centre of the frame with neutral lighting. Due to the limited nature of both the colour space used and the approach, anything more substantial than minor performance increase would require an overhaul of the methodology used. Changing from the *RGB* to the *HSV* colour space would increase the accuracy of the colour detection and would allow the solution to be expanded to a fuller range of car colours.

Finally, outlining the car and omitting the background would reduce the likelihood of the background of the image influencing the detected colour of the car.

II. Task 2: Colour Tracker

A. Introduction

The second task was a colour tracker. It was set to track a specific colour within the frame. This colour was attached to the end of a pendulum. The results were recorded and the relative angle of the pendulum was recorded. This angle was recorded in both radians and degrees.

B. Solution

Fig. 5 is the code used to locate the colour in the image.

The colour in the frame was first converted from the RGB colour space to the HSV colour space, a still of this can be seen in Appendix A. The justification for this has been outlined above, it makes colours easier to identify in images [2]. After identifying all of the correctly coloured pixels in

```
int thickness = 2;
int radiusOfCircle = 20;

line(Frame, // target frame
     point_1, //starting point for line
     point_2, //ending point for line
     Scalar(0, 0, 255), // colour of line
     thickness); //line thickness

// used to convert the frame to the HSV colour space
cvtColor(Frame, FrameHSV, COLOR_BGR2HSV);

// vectors for the upper and lower limits of the green for the card
// this is the colour that the frame will be tracking
Vec3b greenLowerBound(30, 50, 70);
Vec3b greenUpperBound(95, 255, 255);

inRange(FrameHSV, greenLowerBound, greenUpperBound, FrameFiltered);
Moments moment = moments(FrameFiltered, true);

// point in the centre of the rectangle found
Point pendulumPoint(moment.m10/moment.m00,
        moment.m01/moment.m00);
```

Fig. 5: Colour Location Code

TABLE I: Upper and Lower Bounds for Hue, Saturation, and Brightness for Colour Tracking

Bound	Hue	Saturation	Brightness
Lower	30	50	70
Higher	95	255	255

no capital?

the frame, a rectangle was drawn and a point was identified in the middle of this rectangle. The values selected for the upper and lower bounds of these colours can be seen in Table. I The hue value was chosen such that any inconsistencies in the colour of the rectangle would not result in it being unmarked. The brightness and Saturation values go from almost 0 to their maximum value 255. This was to allow changes in brightness of the image due to shadows or lighting changes. These values allowed the rectangle to be detected without any objects being highlighted too. The isolated rectangle can be seen in Appendix B.

After the centre point of the rectangle had been identified, the angle of the pendulum to its centre point was then calculated. This was done using basic trigonometric identities using the x and y coordinates of the point. The point was located using the `moments` method which is part of the standard OpenCV library. It is used to find the centre of any shape. The angle calculation in radians is shown below in Fig 6.

Once the angle in radians had been calculated, converting the angle to degrees was trivial. This conversion can be seen in Fig 7.

Once the the angle was outputted onto the frame in both degrees and radians. This was done using the `putText()` method. The text outputted onto each frame can be seen in

```
// calulates the angle of the pendulum in radians
double angleInRadians = (atan2(pendulumPoint.x, pendulumPoint.y )/ M_PI_4) - M_PI_4;
char angleInRadiansOutput[30];
sprintf(angleInRadiansOutput, "Angle in radians: %f", angleInRadians);
```

Fig. 6: Calculating the angle of the pendulum in radians

```
// calcualtes the angle of the pendulum in degress
double angleInDegrees = angleInRadians * 180 / CV_PI;
char angleInDegreesOutput[30];
sprintf(angleInDegreesOutput, "Angle in degrees: %f", angleInDegrees);
```

Fig. 7: Converting Angles From Radians to Degress

Fig. 8: Angle Output on Video Frames

Fig 8.

Alongside the output on each of the frames, the angle of the pendulum was printed to the terminal. This is done in the following way: `cout << "Pendulum angle:" << angleInDegrees << endl;`

At the end of each loop for each frame, the angle (in degrees) is then saved to an Comma Seperated Values (CSV) file. THis was done using the following method: `DataFile << angleInDegrees << endl;`

The values saved into the CSV file, were then combined into a graph to show the angle of the pendulum against time. This can be seen in Fig. 9.

The full size graph of angle against time can be seen in Appendix C.

It can be seen from the graph that the angle of the pendulums swing is not centred around 0 degrees, there is an offset of about 12.5 degrees. This is caused by the testing set up being slightly offset to the camera. This causes distortion in the results that can be seen as the offset.

The full code for task 2 is shown in Appendix D.

Whilst the code is simplistic it does manage to track the pendulum well and doesn't struggle to identify the end of the pendulum in the frame. Despite this, there was nothing else green within the frame, this may have caused issues if this were to be the case.

A video of the code in operation can be seen in this link

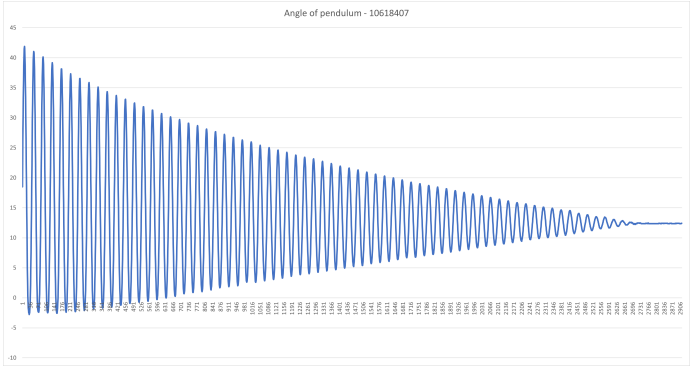


Fig. 9: Angle of Pendulum Against Time

C. Further Improvements

As suggested in the previous section, the improvements for this task revolve mainly around reducing the chance of another object of similar colour in the frame being detected. This could be done in many ways. The most simple way of doing this would be to reduce the chance of targeted body to be a unique shape to everything else in the image. Although *OpenCV* does provide framework for recognising objects in multiple rotations [5], it was be computationally less intense if the shape have multiple lines of symmetry. Another possibility for increasing the speed of image recognition is to train a neural network to detect rotated objects. [6] Another way to improve results, and add complexity would be to add another distinctive colour to track and making the program look for these two adjacent colours. This approach could be scaled to a complex but robust pattern of colours.

A final improvement could be to remove the bias using some software defined bias that could compensate for the skew of the equipment and any form of effect caused by the camera lens, such as fish-eye. *OpenCV* already has baked in methods of correcting for skews such as fish-eye. [7]

D. Conclusion

In similar fashion to part one, this solution works well within the controlled environment of the video. For this to be a more universal approach it's robustness would need to be improved. This could be achieved by adding additional tracking methods or increased target complexity, as discussed in the section above.

III. Task 3: Cross Correlation

A. Introduction

The third task is to use cross correlation to identify missing components on a PCB. Cross correlation is a method of mathematically measuring the similarity of two signals. It is used to measure how similar two samples are, on a sample by sample basis. [8] When using *OpenCV*, the process of cross correlation is known as Template Matching. [9]

Template matching is considered to be one of the easiest forms of object detection that only requires a few lines of code to utilise. Alongside it's easy of use, it is computationally efficient and requires no thresholds to operate. [9]

The template matching in this task was used for identifying missing components on a Raspberry Pi 3B board. There was a list of ten components to check, shown in Appendix E. These ranged from the main Central Processing Unit (CPU) on the board down to small Surface Mounted Technology (SMT) capacitors.

B. Solution

Fig. 10 shows the solution to the third task, Cross Correlation.

The code iterates through a loop to perform template matching on all of the components. The algorithm looks at the first component in the list *Component0*, and does template matching on the whole image to check whether the component

```
// 1st loop with missing components
 cout << "\nPCB with Missing Components" << endl;
//loop through component images
for(int n=0; n<10; ++n){

    //read PCB and component images
    Mat PCB = imread(Path+"PCB.png");
    Mat Component = imread(Path+"Component"+to_string(n)+".png");

    //================Your code goes here=====================
    // get the image output from match
    Mat imageMatch;
    matchTemplate(PCB,
                  Component,
                  imageMatch,
                  TM_SQDIFF_NORMED);

    // values and locations of matching sections
    double maxValue, minValue;
    Point maxLocation, minLocation;
    minMaxLoc(imageMatch, &minValue, &maxValue, &minLocation, &maxLocation);

    // check if the matched pixels for components are below the minimum acceptable amount
    if (minValue > 0.009) { cout << "Component " << n << " not found!" << endl; }

        Point componentRectange(minLocation.x + Component.cols, minLocation.y + Component.rows);
        rectangle(PCB, minLocation, componentRectange, Scalar(255, 0, 0), 2);
        cout << "Component " << n <<  " found!" << endl;

    //display the results untill x is pressed
    while(waitKey(10)!='x'){
        imshow("Target Component - 10618407", Component);
        imshow("PCB - 10618407", PCB);
    }

}
```

Fig. 10: Template Matching Code With No Components Missing

is in the image or not. This is done by by checking the first template image (Fig. 11), and trying to match it with something in the source image (Fig. 12). At each point, the match is calculated using the equation in Fig. 13. This then gives a metric of how *'good'* or *'bad'* the match is. For each location, a computed result is stored in a resultant matrix. 'Bright' locations on the resultant matrix indicate the best mactches and darker areas indicate very little correlation. [9]

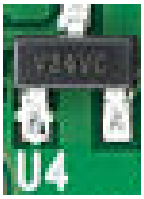


Fig. 11: Template Component 0

$$R(x,y) = \frac{\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

Fig. 13: Template Matching equation for a normalised correlation coefficient

When this code was run on the image, all of the components except *Component 8* were found. This is the expected output for the PCB as this was the only component missing. This can be seen in Fig. 14.

If a component is found on the board it is outlined with a blue rectangle and the corresponding component number it outputted onto the terminal. When the code was run over mul-

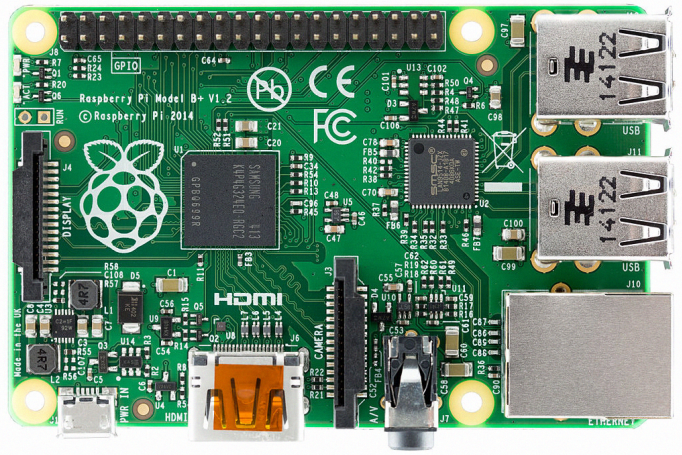
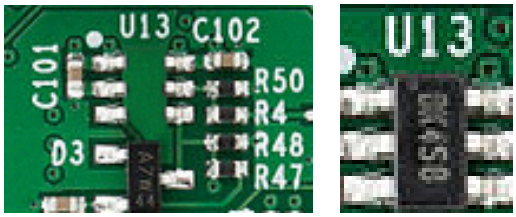


Fig. 12: Source Image



(a) Missing Component Space on PCB (b) Missing Component

Fig. 14: Missing Component and Empty place on PCB

tiple iterations, it reliably managed to identify that *Component 8* was missing.

In order to do a comprehensive test of the software, a picture of the board with the missing component added back on was used a test. Under these conditions, *Component 8* was reliably and repeatably found in the correct location. This result shows that the template matching algorithm was in fact working on all the components and was not missing any on its run through.

A video of the code in operation can be seen in this link

C. Further Improvements

Despite the Template Matching algorithm working, it is not robust. It will not work if there are differing orientations or scales. Fixing the scaling issues is relatively simple and can be done using template matching. To do multi-scaled template matching is simple. The template image is re-scaled and the template matching algorithm is run again. [10] This approach is simplistic and can save a lot of effort in writing and a lot of additional code and dealing with more fancy methods of template matching.

Template matching is not a good solution if the template and source image are in different orientations. Instead another methods can be used such as Canny Edge Detection or Keypoint Detection.

Canny Edge detection is one of the most popular edge detection algorithm developed by John F. Canny. It is a multi-stage process. It starts by utilising a Gaussian filter [11] to reduce edge noise in the image. After this, the intensity gradient of the image is found using a Sobel kernel [12]. This then gets the first derivaive in both the x (G_x) and y (G_y) directions. The Edge Gradient is then calculated from this. After this, Non-maximum suppression [13] is used to remove any unwanted pixels that may not have made up the edge of the object. The final step is Hysteresis Thresholding [14]. This stage is done to determine which edges are actually edges and which are not.

Keypoint detection is used often to mark out the human body. It utilises a heatmap representation of each keypoint i.e. shoulders, hips, hands etc. [15] This type technique is often paired with a neural network to make the detection faster and more accurate. This kind of Keypoint Detection is very powerful and has many uses such as gesture recognition, sign languages understanding, and activity recognition along with many more. [16] Given the scope of this task, keypoint detection using a deep learning network would be deemed to be an over-engineered solution.

D. Conclusion

As per the two other examples, template matching worked well enough for the criteria provided. Despite this, template matching is rather limited and if anything more complex than a difference in scaling, this approach would prove to be ineffective. A more robust and comprehensive solution would be required in the real world.

A combination of the three previous example of cross correlation could provide a lightweight initial set of tests backed up by a more computationally intensive but more robust solution. This could be used to evaluate more complex matching tasks, initially running the more lightweight, but less powerful algorithms before moving towards heavyweight but more thorough methods afterwards.

A video of the code in operation can be seen in this link

References

[1] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3d object representations for fine-grained categorization," in *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013.

[2] T. Mamdouh. (2020, Apr.) Color spaces (rgb vs hsv) - which one you should use? [Online]. Available: https://discover.hubpages.com/technology/Color-spaces-RGB-vs-HSV-Which-one-to-use

[3] H. Parmar. (2021) Vehicle detection in python using opencv. [Online]. Available: https://www.codespeedy.com/vehicle-detection-in-python-using-opencv/

[4] A. Zucconi. (2015, Sep) The incredibly challenging task of sorting colours. [Online]. Available: https://www.alanzucconi.com/2015/09/30/colour-sorting/

[5] A. Sears-Collins. (2021, Mar) How to determine the orientation of an object using opencv. [Online]. Available: https://automaticaddison.com/how-to-determine-the-orientation-of-an-object-using-opencv/

[6] D. K. Gupta, D. Arya, and E. Gavves, "Rotation equivariant siamese networks for tracking," in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, pp. 12 357–12 366.

[7] K. Jiang. (2017, Sep) Calibrate fisheye lens using opencv. [Online]. Available: https://medium.com/@kennethjiang/calibrate-fisheye-lens-using-opencv-333b05afa0b0

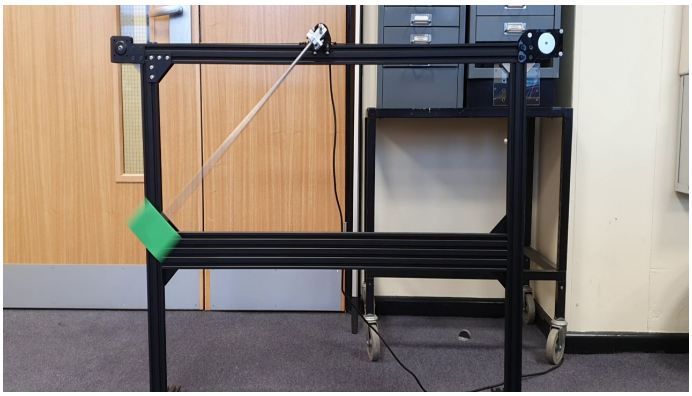
[8] M. Carrick. (2021, Dec) Cross correlation explained with real signals. [Online]. Available: https://www.wavewalkerdsp.com/2021/12/01/cross-correlation-explained-with-real-signals/

[9] A. Rosebrock. (2021, Mar) Opencv template matching (cv2.matchtemplate). [Online]. Available: https://pyimagesearch.com/2021/03/22/opencv-template-matching-cv2-matchtemplate/
[10] ——. (2015, Jan) Multi-scale template matching using python and opencv. [Online]. Available: https://pyimagesearch.com/2015/01/26/multi-scale-template-matching-using-python-opencv/
[11] R. Fisher. (2003) Gaussian smoothing. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm
[12] A. Sears-Collins. (2017, Dec) How the sobel operator works. [Online]. Available: https://automaticaddison.com/how-the-sobel-operator-works/
[13] S. K. (2019, Oct) Non-maximum suppression (nms) - a technique to filter the predictions of object detectors. [Online]. Available: https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c
[14] A. Diwan. (2020, Dec) What is hysteresis thresholding? how can it be achieved using scikit-learn in python? [Online]. Available: https://www.tutorialspoint.com/
[15] V. Belagiannis and A. Zisserman, "Recurrent human pose estimation," in *International Conference on Automatic Face and Gesture Recognition*. IEEE, 2017.
[16] V. Gupta. (2018, Oct) Hand keypoint detection using deep learning and opencv. [Online]. Available: https://learnopencv.com/hand-keypoint-detection-using-deep-learning-and-opencv/

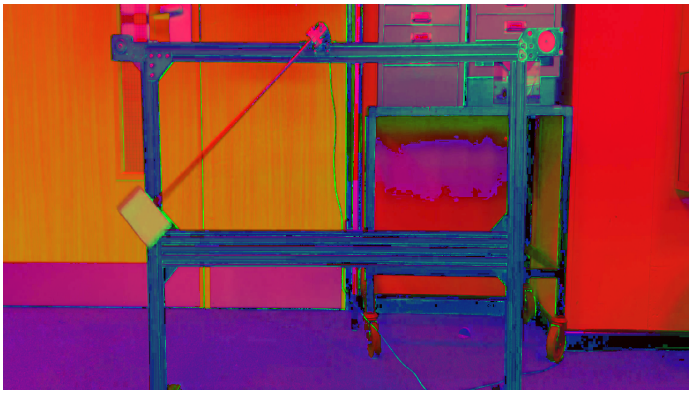
The code can be found on GitHub Here!

APPENDIX

A. Pendulum in the RGB and HSV Colour Space

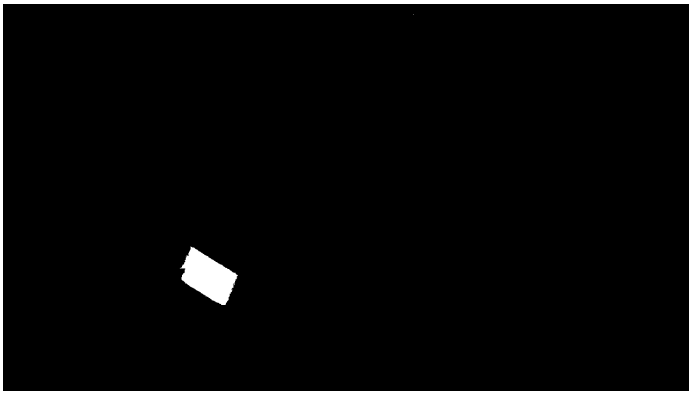


(a) Pendulum in the RGB Colour Space

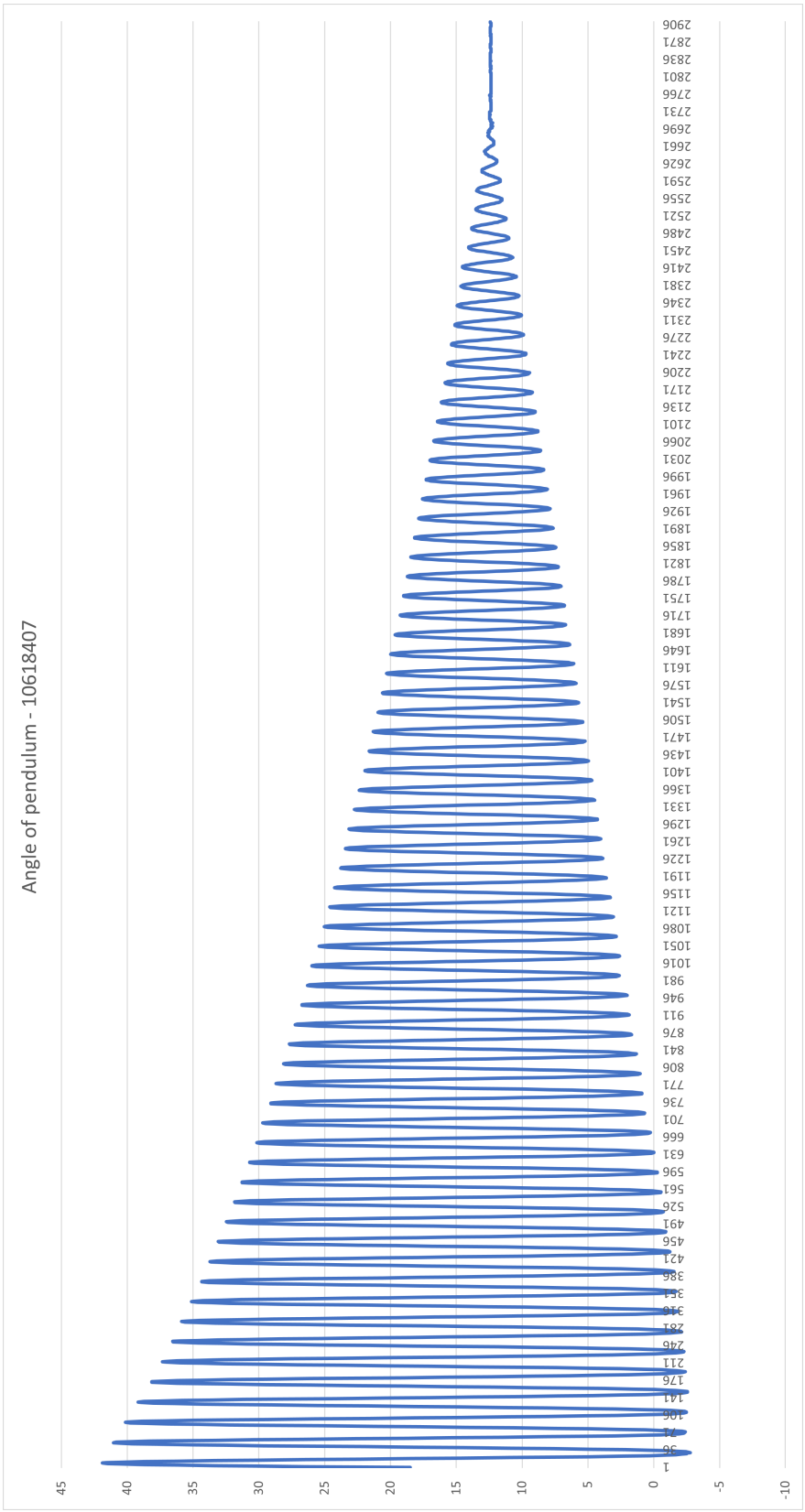


(b) Pendulum in the HSV Colour Space

B. Isolated Rectangle at the End of the Pendulum



C. Full Size Graph of Pendulum Angle Against Time



D. Full Code - Task 2

```
// points used to draw the vertical line for the origin of the pendulum
Point point_1(592, 52), point_2(592, 650);

int thickness = 2;
int radiusOfCircle = 20;

line(Frame, // target frame
     point_1, //starting point for line
     point_2, //ending point for line
     Scalar(0, 0, 255), // colour of line
     thickness); //line thickness

// used to convert the frame to the HSV colour space
cvtColor(Frame, FrameHSV, COLOR_BGR2HSV);

// vectors for the upper and lower limits of the green for the card
// this is the colour that the frame will be tracking
Vec3b greenLowerBound(30, 50, 70);
Vec3b greenUpperBound(95, 255, 255);

inRange(FrameHSV, greenLowerBound, greenUpperBound, FrameFiltered);
Moments moment = moments(FrameFiltered, true);

// point in the centre of the rectangle found
Point pendulumPoint(moment.m10/moment.m00,
        moment.m01/moment.m00);

// draws the circle of the centre of the rectangle
circle(Frame, // target frame
       pendulumPoint, // center of circle
       radiusOfCircle, //radius of circle
       Scalar(0,255,0), // colour of circle
       thickness * 1.5); //thickness of line

// draws the line corrisponding to the pendulum
line(Frame, // target frame
     Pivot, // starting point of line
     pendulumPoint, // ending point for line
     Scalar(255,0,0), //colour of line
     thickness); //thickness of line

// calulates the angle of the pendulum in radians
double angleInRadians = (atan2(pendulumPoint.x, pendulumPoint.y )/ M_PI_4) - M_PI_4;
char angleInRadiansOutput[30];
sprintf(angleInRadiansOutput, "Angle in radians: %f", angleInRadians);

// calcualtes the angle of the pendulum in degress
double angleInDegrees = angleInRadians * 180 / CV_PI;
char angleInDegreesOutput[30];
sprintf(angleInDegreesOutput, "Angle in degrees: %f", angleInDegrees);

// outputting the angle in radians onto the screen
putText(Frame, //target image
        angleInRadiansOutput, // text to be outputted
        Point(800, 30), // top-left position of text
        FONT_HERSHEY_DUPLEX, // font
        1.0, // size of text,
        Scalar(0, 255, 0), // text colour
        thickness);

// outputting the angle in degrees onto the screen
putText(Frame, //target image
        angleInDegreesOutput, //text
        Point(800, 60), // top-left position of text
        FONT_HERSHEY_DUPLEX, // font
        1.0, // size of text
      Scalar(255, 0, 0), // text colour
        thickness);

// prints the angle of the pendulum in degrees to the terminal
cout << "Pendulum angle:  " << angleInDegrees << endl;

// writes the angles in degrees to a CSV file to be plotted
DataFile << angleInDegrees << endl;

//if frame is empty then the video has ended, so break the loop
if(FrameFiltered.empty()){
    break;
}

//video is very high resolution, reduce it to 720p to run faster
resize(FrameFiltered,FrameFiltered,Size(1280,720));
```

E. Template Matching Components

(c) Component 0

(d) Component 1

(e) Component 2

(f) Component 3

(g) Component 4

(h) Component 5

(i) Component 6

(j) Component 7

(k) Component 8

(l) Component 9