AINT308 - Machine Vision and Behavioural Computing
Coursework 1 Report

Student No. 10613591
School of Engineering,
Computing and Mathematics
University of Plymouth
Plymouth, Devon

***Abstract*—Machine Vision is field of study who's applications are becoming rapidly more prevalent amongst contemporary technology, with advancements having large implications in a wide variety of fields. This report details the use of a popular open-source machine vision library, OpenCV, in three different applications: Identifying the most common colour in an image, tracking a moving object within a video, and matching sub-components of an image to a larger image.**

Keywords:

Machine Vision, OpenCV, Object Tracking, C++

I. Task 1 - Colour Sorter

For accompanying large figures, see appendix B

A. Introduction

The first task involves creating a system able to label a dataset of car colours red, green, or blue.

B. Solution

As part of its suite of features, **OpenCV** includes the ability to retrieve individual pixel's values. The specific values available will vary based on the colour space used. For example, working in the *RGB* colour space enables a pixel's *Red,Green* and *Blue* values to be retrieved. Using this information, it is possible to calculate the modal colour of an image.

```
if ((b>1.5*g) && (b>1.5*r) && (b>125)){
    bluecount++;
    if ((x>cols*(lowlimit)) && (x<cols*(highlimit))){
        bluecount = bluecount + 3;
    }
}
else if ((g>1.5*b) && (g>1.5*r) && (g>125)){
    greencount++;
    if ((x>cols*(lowlimit)) && (x<cols*(highlimit))){
        greencount = greencount +3;
    }
}
else if ((r>1.5*b) && (r>1.5*g) && (r>125)){
    redcount++;
    if ((x>cols*(lowlimit)) && (x<cols*(highlimit))){
        redcount = redcount +3;
    }
```

Fig. 1: Pixel testing code

The code in figure 1 performs the checks needed to assertain a pixel's primary colour. Initial attempts simply checked which of three component values were highest. In practice, this method was inaccurate as it had no way to differentiate between between a near-white cell (such as a background cloud) which would have near equal *RGB* values, and one that belonged to the subject (in this case, the car).

To allow the algorithm to differentiate between background elements and the subject, two methods were employed. Firstly, a pixel was only counted if that colour's value was a significant margin higher than the other values. In figure 1 this margin is 50%. Additionally, the pixel's value must be greater than half the colour space's maximum value, in this case 125. This prevents background objects from being counted.

Additionally, extra weighting is applied to pixels within a central rectangle on the screen. This area can be changed by editing the values $highlimit$ and $lowlimit$. By adding this additional weighting to central areas, background areas are further discounted, helping to prevent false positives by valuing true positives more.

With the provided testing dataset of thirty red, blue, or green cars this solution is 100% accurate, correctly identifying the subject's colour in every image. This is a marked improvement from the simple method used in the initial attempt, which would often get confused at reflections in windows (these would be reflecting the sky, which would add bias towards blue), background walls (especially brick, which would bias the results towards red), and any natural backgrounds (adding green bias).

To further demonstrate the improvements performance, an additional dataset was sourced of ninety images, thirty of each colour. The results of this testing are visible in table I.

TABLE I: Expanded dataset testing results

Colour	Result
Red	30/30
Green	28/30
Blue	30/30

For an overview of the algorithm's operation, see figure 6.

C. Limitations

This solution is extremely limited in its current format. The solution has no form of object recognition, meaning that it cannot differentiate between the car that it is trying to find the colour of, and any other objects inside the image. The middle-area weighting attempts to control for this, however it assumes that the subject of the image is centre-frame, which is something that cannot be said for all images of cars. This solution also has no way to deal with multiple cars in one image, especially ones of different colours.

Additionally, using *RGB* values is extremely limiting for colour recognition. The algorithm could not be expanded to detect additional colours (such as secondary colours made of a combination of red, green, and blue) without introducing a large amount of error. A major disadvantage of *RGB* values is that they can vary greatly based on the lighting conditions present in the image. Other colour spaces isolate the brightness (or luma) value, allowing for changes in lighting to be compensated for. A further problem with *RGB*'s representation of colour is that the hue is described as a combination of all three values, making it hard to isolate specific colours, as it is extremely rare for any physical colour to exist in purely one colour channel.

(a) Image 6 (Green)

(b) Image 25 (Green)

```
Incorrect - image 6
Blue Count = 36
Green Count = 468241
Red Count = 699637
```

(c) Image 6 pixel count

```
../Task1/add_green/25.png
Incorrect - image 25
Blue Count = 3
Green Count = 480
Red Count = 815
```

(d) Image 25 pixel count

Fig. 2: Failed recognition images and stats

Figures 2a and 2b show images within the expanded dataset that the algorithm failed to categorise correctly, and their respective pixel counts in c and d. These images were both categorised as red despite the improvements made listed above. This is due to the large amounts of red coloured background elements (the reflection of the brake lights in a and the road surface in b).

D. Further Improvements

A major improvement that could be made is switching colour space to *HSV* instead of *RGB*. *HSV* has numerous benefits for colour recognition when compared to *RGB*, which are further elaborated on in section II-B.

Additionally, changing to *HSV* would allow for more colours to be recognised, as only the colour's *hue* value would need to be programmed in for the algorithm to recognise it. This would allow for theoretically any colour to be recognised. In comparison to *RGB*, where as each colour is a mixture of all three channels, this is much simpler to implement.

E. Conclusion

This solution is adequate when the dataset is tightly controlled and consists only of red, green, or blue cars, in centre frame, with colour-neutral lighting. Due to the limitations of both the approach and colour space, anything more than marginal performance increases would require significant changes to the methodology. Changing colour space to *HSV* would provide a large increase to accuracy, as well as allow for the solution to be expanded to cover more colours.

II. Task 2 - Object Tracking

For accompanying large figures, see appendix C

The second task involves tracking a pendulum, and using the observed displacement to calculate the pendulum's angular deviation from it's vertical rest position.

A. Introduction

In order for the angle to be calculated, the pendulum's motion must be tracked. Fortunately, the pendulum arm has a green target affixed to the end. By isolating this colour from the background, and calculating the centre of mass(CoM) using OpenCV's *moments* functionality, the angle can be calculated using trigonometry.

B. Solution

The first stage of the solution is to find all the pixels that match the colour of the target on the pendulum arm. To do this, the colour space is first converted to *HSV, **H**ue, **S**aturation, **V**alue. HSV* has numerous benefits over *RGB* for colour recognition[1], primarily that colours can easily be isolated using the *hue* value, unlike in *RGB* where a colour is dependant on the values of all three channels.

HSV uses three values to describe a colour: *hue*, *saturation*, and *value*. Light changes will only affect two of these values; the *hue* value remains constant. This is better suited for colour recognition applications as it means changes in lighting, as well as natural minor differences in colours, can be compensated for with much tighter ranges than in any other colour space.

To isolate the target, *OpenCV*'s *inRange* function is used. This function takes four arguments, an input matrix, a vector of lower bounds, a vector of upper bounds, and then an output matrix. The output matrix is monochrome, highlighting areas that lie within the provided bounds in white, and any that do not fall within these bounds in black. See appendix C to see the output of the program.

TABLE II: *HSV* thresholds used inside the *inRange* function

Property	Lower Threshold	Upper Threshold
Hue	70	80
Saturation	30	255
Value	30	255

As shown by table II, the *saturation* and *value* variables have an extremely wide range to them, while still maintaining high accuracy. The narrow *hue* range is used to isolate the specific shade of green used on the target.

The centre of mass is calculated using the *moments* function, then displayed as a point. To further aid the user, a cross hair is drawn over the centre of the target, with a line connecting it to the fulcrum at the top of the pendulum arm. For reference, an additional vertical line is drawn, representing an approximation of the vertical rest position.

The current angle is calculated by taking the arctangent, or inverse tangent, of the pendulum line.

$$angle = arctan(pendulum) - \frac{\pi}{4} \tag{1}$$

An offset of $\pi/4$ is added in order to centre the graph around the 0 mark. Additionally, the angle is also displayed in degrees.

$$angle = x * (180/\pi) \tag{2}$$

Figure 8 within appendix C shows the output graph. Comparing the output graph to the observed behaviour of the systems shows that the results are as expected, with the pendulum acting equivalently to a dampened system returning to its rest state after receiving a stimulus.

C. Limitations

This implementation of object tracking does have its limitations. Firstly, it relies on the tracked target being a known colour, and also no other objects of a similar colour being observed by the input. This could be improved by using an additional technique to identify the target, for example, looking for a unique shape using edge detection, or making the target more complex than a single colour, such as two colours in a tiled pattern.

Ideally, if visual detection is to be used, the target must be as unique as possible, to reduce the chances of objects within the camera's line of sight having enough similarity to confuse it. The current implementation relies on no objects being a similar shade of green, which works for the current scenario, but may be unreliable for less controlled applications.

An additional issue with the optical based measurement employed here is that it relies on the target swinging on a path orthogonal to the camera. As shown in the output graph, there is a bias towards the positive direction, caused by the testing apparatus being slightly offset from the camera, distorting the results. This could be compensated for in software, however it would be complex to implement, and the reliability of such a solution cannot be speculated on.

D. Further Improvements

As discussed in section II-C, suggested improvements revolve around reducing the chance for non-target objects that have similar properties. This would involve increasing the relative complexity of the targets. For ease of implementation, the targets should have as many orders of rotational symmetry as possible, to avoid having issues with having to recognise every rotational permutation of the target. *OpenCV* provides documentation on recognising rotated images[2]. Additionally, a neural network could be trained to recognise the targets in any orientation or offset quickly enough to perform in real-time[3]. An easy way to increase the complexity would be to add an additional distinctive colour, and then have the software look for the two colours adjacent to each other.

A Kalman filter could be implemented to improve the tracking performance[4]. The filter works by using the previous measurements and observations to estimate the state of this system, in this case, the pendulum's deviation. This would be helpful for any case where the video may not be as high quality, or the target may not be as clear.

While potentially outside the scope of the task, a generalised method for compensating for offset apparatus would increase the validity of the results, and help make any tests performed with this system more repeatable. Software based algorithms exist that can perform this function with minimal latency [5].

E. Conclusion

As per task one, this solution works extremely well in the controlled environment present in the video, however would need increased robustness - provided either by an increase in target complexity, or additional tracking methods - to succeed in a less controlled environment.

III. Task 3 - Cross Correlation

For accompanying large figures, see appendix D

The third task involves using cross correlation to identify if components are present on a PCB.

A. Introduction

Cross correlation in signal processing is used to measure the similarity between two signals, with the result being a function of displacement of the first signal relative to the second. The higher the result, the more closely related the two series are, with a perfect match meaning the two signals are effectively equal[6].

Cross Correlation has numerous applications in image processing, such as searching a large image for a known feature. By using the *OpenCV matchTemplate* function, the point on an image that has the highest correlation can be identified, effectively revealing the location of the desired feature.

B. Solution

The solution uses Normalised Square Difference Matching, presented as the argument *'SQDIFF_NORMED'* in *OpenCV*. Using this matching method, the area with the best match will be presented as a local minima[7], the code in figure 9 shows *matchloc* being assigned the coordinate value of *minloc*, which is passed by address into the *minMaxLoc* function call.

$$R_{sq\_diff\_normed} = \frac{\sum_{x',y'}[T(x',y') - I(x+x',y+y')]^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \bullet \sum_{x',y'} I(x+x',y+y')}} \tag{3}$$

TABLE III: Error values of different components.

Component	NMSD Error
U4	0.000642183
C70	0.000428351
U2	0.000984659
L2	0.000689957
Q1	0.000671107
C97	0.000465317
C87	0.000600116
U1	0.00133448
U13 (Missing)	0.0435289
U13 (Present)	1.52745e-07
L8	0.00050902

minMaxLoc is a built-in *OpenCV* function that finds the global minimum and maximum in an array, and their respective positions[8]. The extracted coordinates are used to draw a rectangle around the components position, with the size extracted from the component's associated *png* file.

```
cout << matchloc << endl;
cout << minval << endl <<endl;
if(minval >0.01){
    cout << "Component number " << n<<" not found" << endl;
    count -= 1;
}
```

```
[598, 100]
0.0435289

Component number 8 not found
[377, 470]
0.00050902

Components present: 9/10
```

Fig. 3: a) Error value check
b) associated terminal output.

The error threshold in figure 3 was chosen using experimentation. Inspection of the provided PCB image shows **U13** to be the missing component. This value was chosen by comparing the output values for present components with that of **U13**, and selecting a threshold that would correct isolate the missing component.

Table III shows the comparative error values between different components. It can be seen that the missing component, **U13**, has an error value an order of magnitude higher than the present components. To test the functionality of the solution, **U13** was edited back onto the board. The second iteration of the code performs the same checks against the complete board, and correctly identifies all components, reporting that all are found. Figure 4 shows the corresponding terminal output , while figure 10 shows the results of the second execution.

```
[377, 470]
0.00050902

Components present: 10/10
```

Fig. 4: Terminal output of the second execution, comparing against the complete board.

C. Limitations

The major limitation of this implementation of template matching is that it does not account for differences in scale or orientation between the images to be matched. This means the images must match exactly, marginal differences in lighting or angle will cause greatly increased error values, potentially enough to cause an object to not be recognised. Whilst scale changes can be controlled for, any non-affine transformations (that is, transformations that do not preserve parallel lines), cannot be.

D. Further Improvements

Differences in scale between the template image and the image to be searched can be rectified by performing the template match in a loop, wherein the subject image is reduced in size slightly every iteration, until a match is found. This can be augmented by performing edge map detections as opposed to just grey-scale representation, which would reduce issues with lighting changes, however would not help in the case of non-affine transformations[9].

In the case of non-affine transformations, such as distortions in the image, a better approach would be to use an alternative to template matching. One such alternative is to use Local Binary Patterns[10] to represent the features in the template and the image to be searched as Object Codes, comparing between the two [11]. This approach has greater performance than standard template matching, and the use of SIFT(Scale Invariant Feature Transform)[12] and SURF (Speeded-Up Robust Features)[13] improves performance while allowing the algorithm to correctly identify the features regardless of any transforms applied to it.

E. Conclusion

This solution performs well in the controlled environment, with the template IC images being extracted directly from the source image. Under less than ideal conditions, this solution's performance would quickly drop off, as combinations of noise, image distortions, and changed lighting conditions would affect it. This could be improved by augmenting the template matching with edge map detection, or using a more robust approach such as Object Codes.

APPENDIX

REFERENCES

[1] Nathan Glover. *HSV vs RGB*. 2016. URL: https://handmap.github.io/hsv-vs-rgb/.

[2] OpenCV. *Feature Matching*. Documentation. OpenCV, 2017.

[3] Zihang Dai Mingxing Tan. *Toward Fast and Accurate Neural Networks for Image Recognition*. 2021. URL: https://ai.googleblog.com/2021/09/toward-fast-and-accurate-neural.html.

[4] Rahmad Sadli. *Object Tracking: Simple Implementation of Kalman Filter in Python*. 2020. URL: https://machinelearningspace.com/object-tracking-python/.

[5] Xiaochuan Zhao et al. "An image distortion correction algorithm based on quadrilateral fractal approach controlling points". In: *2009 4th IEEE Conference on Industrial Electronics and Applications*. 2009, pp. 2676–2681. DOI: 10.1109/ICIEA.2009.5138693.

[6] Control Station. *What is Cross-Correlation?* 2014. URL: https://controlstation.com/blog/cross-correlation-cross-correlation-isolate-controller-interaction-issues/.

[7] Gary Bradski Adrian Kaehler. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. 2016.

[8] OpenCV. *Operations On Arrays*. 2022. URL: https://docs.opencv.org/3.4/d2/de8/group__core__array.html#gab473bf2eb6d14ff97e89b355dac20707.

[9] Adrian Rosebrock. *Multi-scale Template Matching using Python and OpenCV*. 2015. URL: https://pyimagesearch.com/2015/01/26/multi-scale-template-matching-using-python-opencv/.

[10] Liangzong He. "Texture classification using texture spectrum". In: *Pattern Recognit.* 23 (1990), pp. 905–910.

[11] Yiping Shen et al. "A Fast Alternative for Template Matching: An ObjectCode Method". In: *2013 2nd IAPR Asian Conference on Pattern Recognition* (2013), pp. 425–429.

[12] D.G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision 60* (2004).

[13] Herbert Bay et al. "Speeded-Up Robust Features (SURF)". In: *Computer Vision and Image Understanding* 110.3 (2008). Similarity Matching in Computer Vision and Multimedia, pp. 346–359. ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2007.09.014. URL: https://www.sciencedirect.com/science/article/pii/S1077314207001555.

A. Github Repository

For the full code, please see the linked github repository HERE.

B. Task 1 Figures

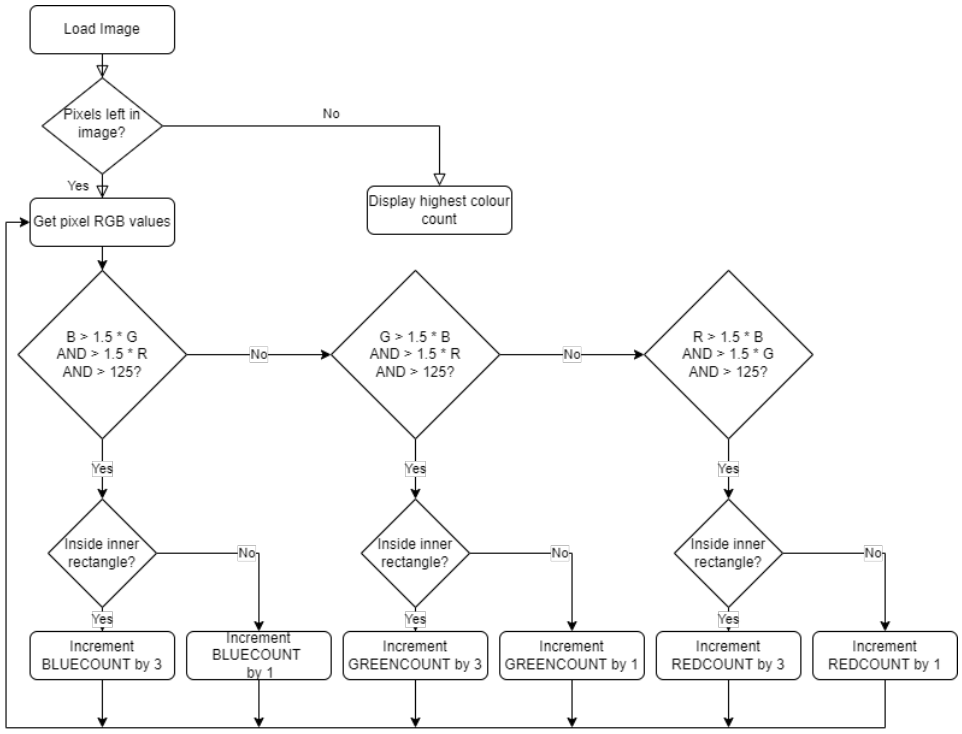


Fig. 5: Flowchart showing the Task 1 algorithm

C. Task 2 Figures

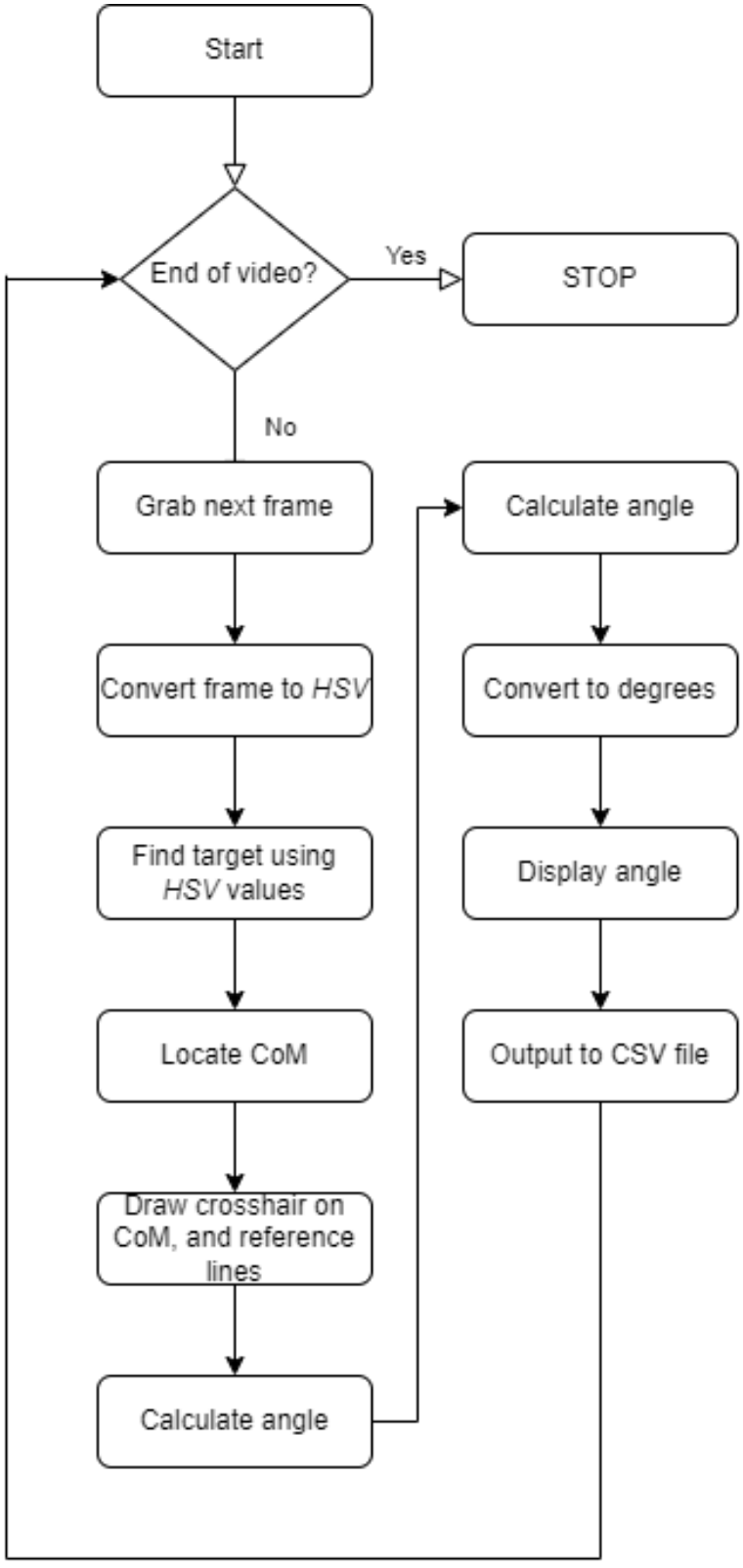
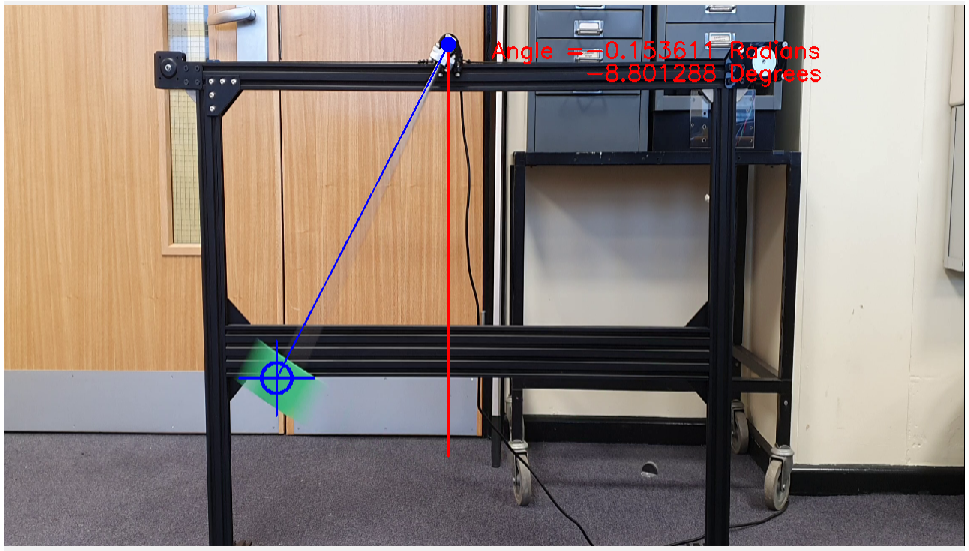


Fig. 6: Flowchart showing the Task 2 algorithm



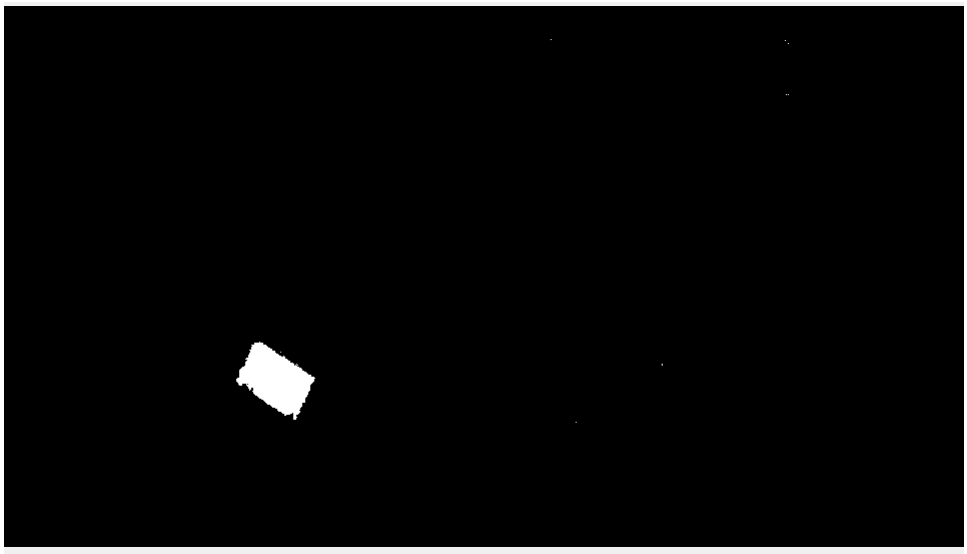


Fig. 7: Single frame of output from Task 2's execution. Note the green target is highlighted in the second frame.

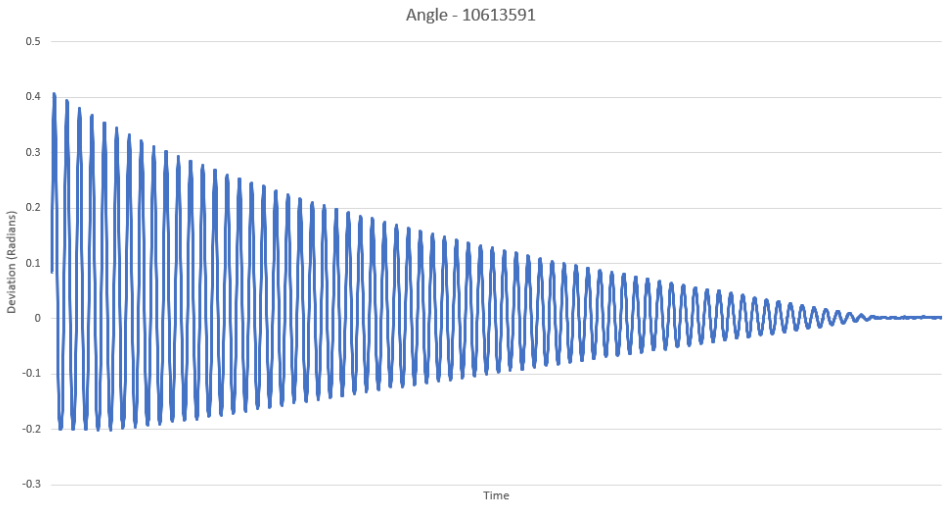


Fig. 8: Output graph showing the pendulum's deviation from the vertical plotted against time. The raw data and graph is available for edit within the archive.

D. Task 3 Figures

```
Mat matchImage;
matchTemplate( PCB, Component, matchImage, TM_SQDIFF_NORMED );  //Normed Square Difference Matching
double minval, maxval;                                          //Minimum and maximum values
Point minloc, maxloc, matchloc;                                 //coordinates for min error, max error, and the match location
minMaxLoc(matchImage, &minval, &maxval, &minloc, &maxloc);
matchloc = minloc;                                              //SQDIFF returns 0 as a perfect match
rectangle(matchImage, matchloc, Point( matchloc.x + Component.cols, matchloc.y + Component.rows), Scalar(0,0,255), 2, 8, 0);
rectangle(PCB, matchloc, Point( matchloc.x + Component.cols, matchloc.y + Component.rows), Scalar(0,0,255), 2, 8, 0);
```

Fig. 9: Template matching code, showing use of Normalised Squared Difference Matching.

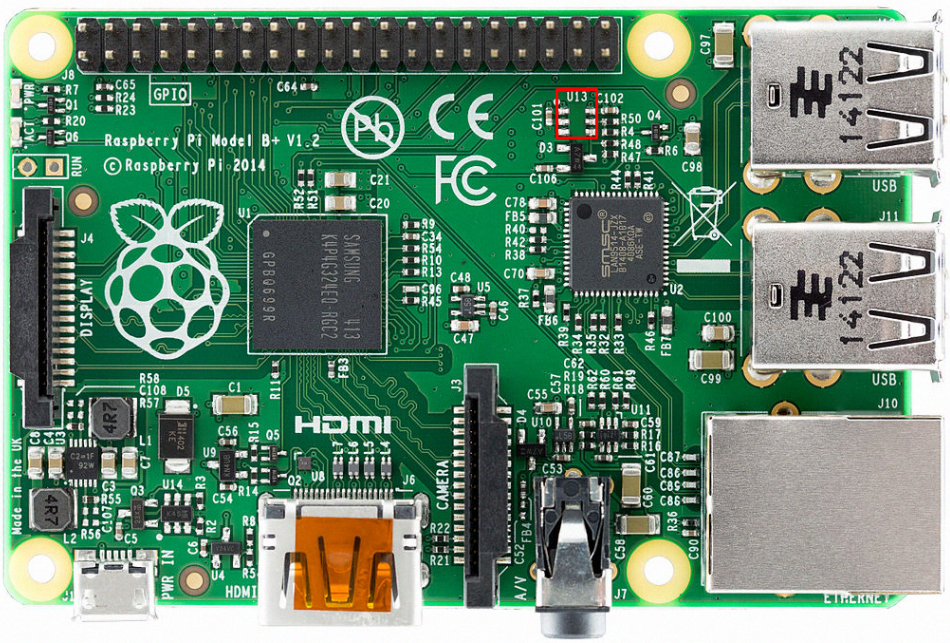


Fig. 10: Task 3 output showing the position of **U13** correctly highlighted as the missing component.

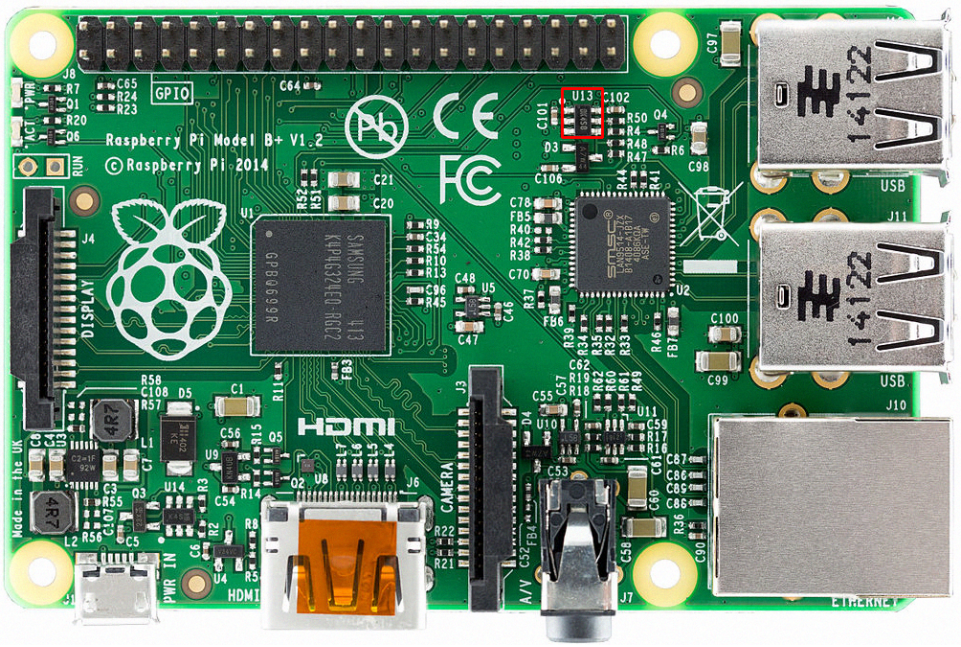


Fig. 11: The edited PCB image with **U13** identified.

E. Code Printouts

The following pages contain complete printouts of the code used to implement the solutions outlined above, for reference.

```cpp
//James Rogers Jan 2022 (c) Plymouth University
#include <iostream>

#include<opencv2/opencv.hpp>
#include<opencv2/opencv_modules.hpp>

using namespace std;
using namespace cv;

int main(){

    //Path of image folder
    string PathToFolder = "../Task1/Car Images/";

    //Loop through the 30 car images
    for(int n=0; n<30; ++n){

        //Each image is named 0.png, 1.png, 2.png, etc. So generate the image file path based on n
and the folder path
        string PathToImage = PathToFolder+to_string(n)+".png";

        cout<<PathToImage<<endl;

        //Load car image at the file paths location
        Mat Car=imread(PathToImage);

        //Your code goes here. The example code below shows you how to read the red, green, and blue
colour values of the
        //pixel at position (0,0). Modify this section to check not just one pixel, but all of them
in the 640x480 image
        //(using for-loops), and using the RGB values classifiy if a given pixel looks red, green,
blue, or other.

        //==============example code, feel free to delete=============
        int rows, cols;                                                 //variables to hold the size
of the matrix
        int bluecount = 0,greencount = 0,redcount = 0;                  //pixel counts
        double lowlimit = 0.2,highlimit=0.8;                            //limits for the enhanced
weighting area
        int x=0;
        int y=0;
        Size s = Car.size();                                            //extract the size of the
matrix
        rows = s.height;                                                //number of rows of pixels
        cols = s.width;                                                 //number of columns of
pixels

        for(x=0; x<cols; x++){
            for(y=0; y<rows; y++){
                Vec3b PixelValue = Car.at<Vec3b>(y,x);                   //get pixel value
                //check which value is higher
                int b,g,r;                                               //b g r values
                b = PixelValue[0];
                g = PixelValue[1];
                r = PixelValue[2];
                if ((b>1.5*g) && (b>1.5*r) && (b>125)){                 //1.5 is the weighting
threshold
                    bluecount++;
                    if ((x>cols*(lowlimit)) && (x<cols*(highlimit))){
                        bluecount = bluecount + 3;                      //Enhanced weighting area
counts for triple
                    }
                }
                else if ((g>1.5*b) && (g>1.5*r) && (g>125)){
                    greencount++;
                    if ((x>cols*(lowlimit)) && (x<cols*(highlimit))){
                        greencount = greencount +3;
                    }
                }
                else if ((r>1.5*b) && (r>1.5*g) && (r>125)){
                    redcount++;
```

```
                    if ((x>cols*(lowlimit)) && (x<cols*(highlimit))){
                        redcount = redcount +3;
                    }
                }
            }
        };
        cout << "Blue Count = " << (int)bluecount <<endl;
        cout << "Green Count = " << (int)greencount <<endl;
        cout << "Red Count = " << (int)redcount <<endl;
        if ((bluecount>greencount) && (bluecount>redcount)){
            cout<<"This car is blue" <<endl;
        }
        else if ((greencount>bluecount) && (greencount>redcount)){
            cout<<"This car is green" <<endl;
        }
        else {
            cout<<"This car is red" <<endl;
        }

        //============================================================

        //display the car image untill x is pressed
        while(waitKey(10)!='x'){
            imshow("Car", Car);
        }
    }
    //testing with expanded dataset
    int blue_correct = 0,green_correct = 0,red_correct = 0,result = 0, temp_correct = 0;
    for(int p=0; p<3; ++p){
        temp_correct = 0;
        for(int n=1; n<30; ++n){

            //Each image is named 0.png, 1.png, 2.png, etc. So generate the image file path based on
n and the folder path
            if (p==0){
                PathToFolder = "../Task1/add_blue/";
            } else if (p==1) {
                PathToFolder = "../Task1/add_green/";
            } else {
                PathToFolder = "../Task1/add_red/";
            }
            string PathToImage = PathToFolder+to_string(n)+".png";
            cout<<PathToImage<<endl;
            //Load car image at the file paths location
            Mat Car=imread(PathToImage);
            int rows, cols;                                                 //variables to hold the
size of the matrix
            int bluecount = 0,greencount = 0,redcount = 0;                  //pixel counts
            double lowlimit = 0.2,highlimit=0.8;                            //limits for the
enhanced weighting area
            int x=0;
            int y=0;
            Size s = Car.size();                                            //extract the size of
the matrix
            rows = s.height;                                                //number of rows of
pixels
            cols = s.width;                                                 //number of columns of
pixels

            for(x=0; x<cols; x++){
                for(y=0; y<rows; y++){
                    Vec3b PixelValue = Car.at<Vec3b>(y,x);                   //get pixel value
                    //check which value is higher
                    int b,g,r;                                               //b g r values
                    b = PixelValue[0];
                    g = PixelValue[1];
                    r = PixelValue[2];
                    if ((b>1.5*g) && (b>1.5*r) && (b>125)){
                        bluecount++;
                        if ((x>cols*(lowlimit)) && (x<cols*(highlimit))){
                            bluecount = bluecount + 3;
```

```
                    }
                }
                else if ((g>1.5*b) && (g>1.5*r) && (g>125)){
                    greencount++;
                    if ((x>cols*(lowlimit)) && (x<cols*(highlimit))){
                        greencount = greencount +3;
                    }
                }
                else if ((r>1.5*b) && (r>1.5*g) && (r>125)){
                    redcount++;
                    if ((x>cols*(lowlimit)) && (x<cols*(highlimit))){
                        redcount = redcount +3;
                    }
                }
            }
        };
        if ((bluecount>greencount) && (bluecount>redcount)){
            result = 0;
        }
        else if ((greencount>bluecount) && (greencount>redcount)){
            result = 1;
        }
        else {
            result = 2;
        }
        if (p == result){
            temp_correct++;
        } else {
            cout <<"Incorrect - image " << n <<endl;
            cout << "Blue Count = " << (int)bluecount <<endl;
            cout << "Green Count = " << (int)greencount <<endl;
            cout << "Red Count = " << (int)redcount <<endl;

            //display the car image untill x is pressed
            while(waitKey(10)!='x'){
                imshow("Car", Car);
            }
        }
    }
    //offload temp result into the correct result variable
    if (p==0){
        blue_correct = temp_correct;
    } else if (p==1){
        green_correct = temp_correct;
    } else if (p==2) {
        red_correct = temp_correct;
    }
}
cout << "blue correct = " << blue_correct+1<<"/30" <<endl;
cout << "green correct = " << green_correct+1<<"/30" <<endl;
cout << "red correct = " << red_correct+1<<"/30" <<endl;

}
```

```cpp
//James Rogers Jan 2022 (c) Plymouth University
#include<iostream>
#include <fstream>
#include<opencv2/opencv.hpp>

using namespace std;
using namespace cv;

#define PI 3.14159265
#define halfPI 1.570796325

int main()
{
    namedWindow("Task 2 - 10613591", WINDOW_NORMAL);
    VideoCapture InputStream("../Task2/Video.mp4");                    //Load in the video as an input stream
    const Point Pivot(592,52);                                         //Pivot position in the video

    //Open output file for angle data
    ofstream DataFile;
    DataFile.open ("../Task2/Data.csv");

    //loop through video frames
    while(true){

        //load next frame
        Mat Frame;
        InputStream.read(Frame);

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

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

        //=====================================================Your code goes here=====================================================
        //this code will run for each frame of the video. your task is to find the location of the swinging green target, and to find
        //its angle to the pivot. These angles will be saved to a .csv file where they can be plotted in Excel.
        int HueLower=70;
        int HueUpper=80;
        int SatLower=30;
        int SatUpper=255;
        int ValLower=30;
        int ValUpper=255;

        Mat FrameHSV;
        cvtColor(Frame,FrameHSV,COLOR_BGR2HSV); //convert to HSV
        Mat FrameFiltered;
        Vec3b LowerBound(HueLower, SatLower, ValLower);
        Vec3b UpperBound(HueUpper, SatUpper, ValUpper);
        inRange(FrameHSV, LowerBound, UpperBound, FrameFiltered);

        Moments m = moments(FrameFiltered, true);
        Point p(m.m10/m.m00, m.m01/m.m00);
        Point p_left(m.m10/m.m00, (m.m01/m.m00)-50);                                    //left of crosshair
        Point p_right(m.m10/m.m00, (m.m01/m.m00)+50);                                   //right
        Point p_up((m.m10/m.m00)+50, (m.m01/m.m00));                                    //top
        Point p_down((m.m10/m.m00)-50, (m.m01/m.m00));                                  //bottom
        //vertical line point
        Point vert_line_end(592,600);
        //text location
```

```
        Point p_text(650,70);
        Point p_text2(774,100);
        //vertical line
        line(Frame, Pivot, vert_line_end, Scalar(0,0,255),2);
        //
        circle(Frame, p, 20, Scalar(255,0,0), 3);
        circle(Frame, Pivot, 10, Scalar(255,0,0), -1);
        line(Frame, p,Pivot, Scalar(255,0,0), 2);
        line(Frame, p_left, p_right,Scalar(255,0,0), 2);
        line(Frame, p_up, p_down,Scalar(255,0,0), 2);

        //angle calculations
        double angle = (atan2(p.x,p.y))-M_PI_4;                                          //
PI/4 offset since zero axis is in the negative y direction
        double angle_degrees = angle * (180/M_PI);                                       //
Conversion to degrees for display purposes
        string text = "Angle ="+to_string(angle)+" Radians";                             //
String construction
        string text_degrees = to_string(angle_degrees)+" Degrees";
        putText(Frame, text, p_text, FONT_HERSHEY_SIMPLEX, 1, Scalar(0,0,255),2);        //
Put text on frame
        putText(Frame, text_degrees, p_text2, FONT_HERSHEY_SIMPLEX, 1, Scalar(0,0,255),2);      //
Roughly lines up with line above


        cout << angle <<endl;                                                            //
Print angle to console
        DataFile << angle <<endl;                                                        //
Send the output to the data file
        Mat3b combined,FilterRGB;
        cvtColor(FrameFiltered,FilterRGB,COLOR_GRAY2BGR);

        hconcat(Frame,FilterRGB,combined);                                               //
combine the original and hue-shifted video feeds into one window
        //
================================================================================================================
==========================

        //display the frame
        imshow("Task 2 - 10613591", combined);
        //imshow("Task 2 - 10613591", Frame);
        waitKey(10);
    }

    DataFile.close(); //close output file
}
```

```cpp
//James Rogers Jan 2021 (c) Plymouth University
#include<iostream>
#include<opencv2/opencv.hpp>

using namespace std;
using namespace cv;

int main()
{

    //Path to image file
    string Path = "../Task3/PCB Images/";
    // check both images
    for(int i=0; i<2;i++){
        cout << "LOADING IMAGE - ";
        if( i == 0){
            cout << "PCB.png" <<endl;
        }
        else {
            cout << "PCB_COMPLETE.png" << endl;
        }
        int count = 10;
    //loop through component images
        Mat PCB;
    for(int n=0; n<10; ++n){

        //read PCB and component images
        if(i == 0){
            PCB = imread(Path+"PCB.png");
        }
        else{
            PCB = imread(Path+"PCB_COMPLETE.png");
        }
        Mat Component = imread(Path+"Component"+to_string(n)+".png");

        //================Your code goes here====================
        Mat matchImage;
        matchTemplate( PCB, Component, matchImage, TM_SQDIFF_NORMED );  //Normed Square Difference
Matching
        double minval, maxval;                                          //Minimum and maximum values
        Point minloc, maxloc, matchloc;                                 //coordinates for min error,
max error, and the match location
        minMaxLoc(matchImage, &minval, &maxval, &minloc, &maxloc);
        matchloc = minloc;                                              //SQDIFF returns 0 as a
perfect match
        rectangle(matchImage, matchloc, Point( matchloc.x + Component.cols, matchloc.y +
Component.rows), Scalar(0,0,255), 2, 8, 0);
        rectangle(PCB, matchloc, Point( matchloc.x + Component.cols, matchloc.y + Component.rows),
Scalar(0,0,255), 2, 8, 0);

        cout << matchloc << endl;
        cout << minval << endl <<endl;
        if(minval >0.01){
            cout << "Component number " << n<<" not found" << endl;
            count -= 1;
        }
        //display the results untill x is pressed
        while(waitKey(10)!='x'){
            imshow("Target", Component);
            imshow("PCB", PCB);
            imshow("Result",matchImage);
        }
    }
    cout << "Components present: " << count << "/10" << endl;
    }
}
```