University of Plymouth

OpenCV Assignment 2

AINT308

James Rogers and Chunxu Li

james.rogers@plymouth.ac.uk

chunxu.li@plymouth.ac.uk

Rev 1.1

Overview

These labs are software based and can be done remotely. During lab times technical support is available online via Teams, and in person in SMB 302. If working remotely, make sure to follow the set up guide to install the relevant software and libraries.

Marking

Assignment 2 is worth 50 marks (50% of the module).

The tasks are worth 20 marks each and are broken down as such:

- **Solution (6 marks):**
    - **(0-1)**: Code barely fulfils the task specification.
    - **(2-4)**: Code mostly fulfils the task specification, but is not overly robust.
    - **(5-6)**: Code fulfils task specification, and extra measures are added to make the solution more robust.

- **Method (6 marks):**
    - **(0-1)**: Little to no effort explaining the code or background on theory.
    - **(2-4)**: Some theory, and an explanation of the code.
    - **(5-6)**: A comprehensive background in relevant theory, and a detailed explanation of the code with appropriate justifications.

- **Evaluation (6 marks):**
    - **(0-1)**: Little to no effort assessing the performance of your solution.
    - **(2-4)**: A measure of performance is identified (such as: error rate, false positives / negatives, etc.) and data is collected to assess your solution.
    - **(4-5)**: A measure of performance is well explained and justified, and a significant amount of data is collected to assess your solution. This may involve making your own data sets to evaluate your solution in a broader scope.

- **Conclusion (2 marks):** Based on your evaluation (with data and numbers to back it up), what conclusions can you make from your solution, and what would you improve on?

An **extra 10** marks are awarded for including the following in the report:

- **Videos (2 marks):** Must contain links to YouTube videos of your code working. The videos don't need to be complicated, just a short clip demonstrating your code functioning as the task describes.
- **Flow Diagrams and Code Snippets (3 marks):** Flow diagrams and snippets will be needed to explain your code, marked on how clear and informative they are.
- **References (3 marks):** All quotes, information, and diagrams used in the report should be clearly cited.
- **Acceptable English and Grammar (2 marks):** Not marked strictly but do proofread before submitting.

Deadline for code and report submission: 16:00 – 12/05/2022

Task 4: Disparity Mapping

A robot is to be designed to navigate an environment, and thus must detect its surroundings. Rather than using LiDAR or ultrasonic sensors, the designer has opted to use stereo cameras. Calibrate the cameras and use disparity mapping to build a 3D representation of the room.

Tasks:

- Use the calibration images to calibrate the cameras. Disparity maps are very sensitive to lens distortion and misalignment so this is essential.
- Use the calibration result to correct the stereo images from the folder "Distance Targets"
- Combine left and right images into a disparity map, where the brightness of each pixel relates to the distance of that pixel to the camera.
- The Distance targets are labelled with a known distance to the camera, use this fact to plot the targets disparity to its distance in excel, and calculate how its disparity relates to its distance in cm. Write a formula that can convert between them: Distance=F(Disparity).
- With this formula, estimate the distance to each of the unknown targets, and create a distance map, where the brightness of each pixel is the distance in cm, up to a maximum of 255cm.

Notes:

This task has multiple parts which need to be done in order:

Camera Calibration

To use stereo cameras, they must be calibrated to each other to correct for both lens distortion (intrinsic error), and physical misalignments (extrinsic error). These can be calculated at the same time using the **"StereoCalibration"** program provided by OpenCV. This was included as a Qt project in the task folder.

The program requires image pairs (taken from the left and right cameras) of a checkerboard target at different positions and orientations. These have been provided in the folder **"StereoCalibration /Calibration Images"**.

To link the images to the calibration program, paths to each image must me listed in the "image_list.xml" file, which can be found under "Other Files" in the Qt project. There is an open space labelled "<imagelist>", here is where you put your paths ordered left-right-left-right… etc:

"../StereoCalibration/Calibration Images/left0.jpg"
"../StereoCalibration/Calibration Images/right0.jpg"
"../StereoCalibration/Calibration Images/left1.jpg"
"../StereoCalibration/Calibration Images/right1.jpg"
…and so on.

Build and run the program. It will load each image pair in turn and detect the grid on the box. It will then calculate the corrections needed to align both cameras. This may take some time, but you should end up with an RMS error of ~0.8 and an epipolar error of ~1.2.

If successful, it should have created two files in the task folder called **extrinsics.xml** and **intrinsics.xml**. These contain information about the cameras and will allow the next program to correct for lens and positional distortion.

Disparity

Open task4 in Qt and edit lines 13 and 14 to link your files from the previous step. Now the example code should load image pairs from **"../Task4/Unknown Targets"** and combine them to compute a disparity map. This is done with an **SGBM block matching class**, which has been set up for you, but feel free to change the settings to achieve better results.

Now that you have a disparity map, you have distance information about the image pair, but how do you get distance to an object in cm for example? **This is for you to find out**. In the files provided, there is a set of images of a target at known distances: "**../Task4/Distance Targets".** Take the disparity value of the target from the **16-bit disparity image** and note it against the targets known distance. Do this for every image pair and **plot distances vs disparity** (this graph will be expected to be in the report). You will see a relationship between distance and disparity, what is this relationship? Can you create a function which converts disparity into distance? You may want to start with the disparity equation:

$$\boldsymbol{Disparity = \frac{B \cdot f}{Distance}}$$

When you find this function, make a new grey Mat called ***distanceMap***, where each pixel stores the distance in cm, up to 255cm. This will appear to make distant targets brighter and close targets darker. **Use this distance map to predict the distance of all the unknown targets**, by reading the pixel value of the target in each image pair.

Task 5: Self-driving Car Lane Detection

One of the most complex computer vision applications is in the growing field of autonomous driving cars. Cameras around the car capture a great deal of information, however breaking this down into useful data is a huge challenge. In this task, you are required to program a small part of this system, lane detection.

Tasks:

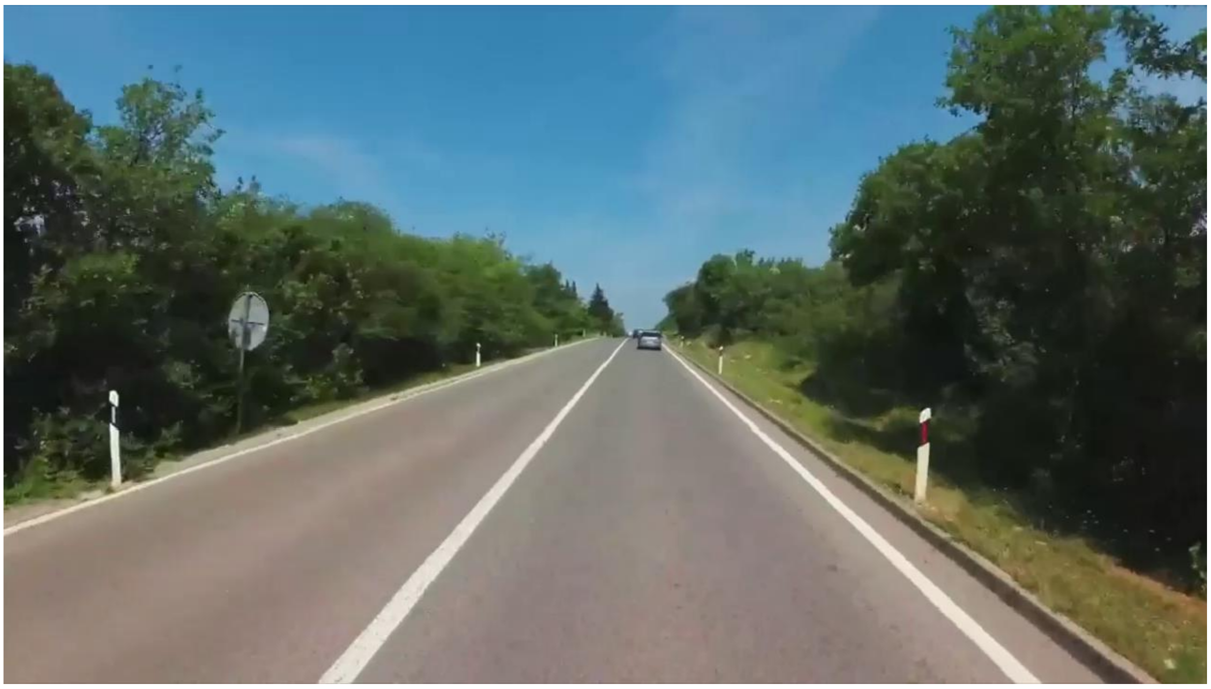
- Load a dashcam video frame by frame.
- Use some method to detect the lane markings either side of the car (this is up to you).
- Display on the video some indication that the lane has been detected.

Notes:

A video has been provided **"Task5/DashCam.mp4"**, which shows a good case scenario with freshly painted lines on a sunny day. Feel free to use this video or any other, for top marks you'd be required to test the performance of your solution on other videos with less ideal cases.

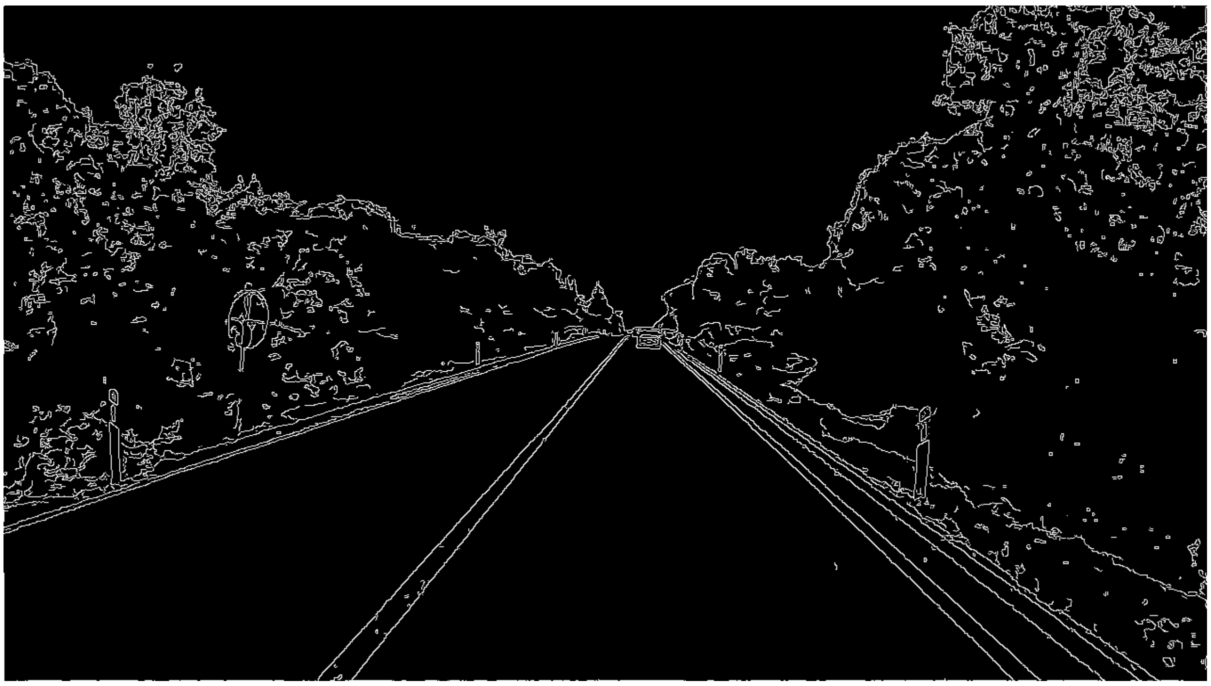
This task is more open than the previous**, so you can choose to program this any way you like.** A basic method will be outlined here for you to build off, but several modifications will be needed to get it working well.

The example code given to you opens the video and plays it frame by frame:



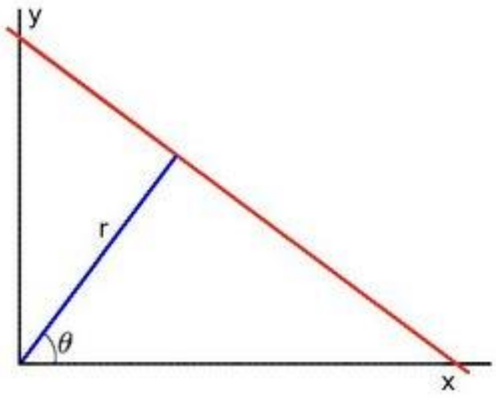
To detect lines, firstly the image needs to be reduced to an **edge map.** This will remove all colour and detail and only leave edges behind. This is done with edge detection. There are many methods of edge detection, such as sobel and difference of Gaussians. **Here we are going to use Canny**, as it produces nice clean single pixel edges that are good for the next step. Here I used 50 for the lower threshold, and 100 for the upper, but feel free to adjust these values to get better results.

```
Canny(inputMat, outputMat, lowerThreshold, upperThreshold);
```



The edge map shows clear straight lines where the lane markings are, but all the computer sees are a bunch of black and white pixels. For the program to find these lines, it has to perform some kind of line detection algorithm. **The one used here is Hough line detection.**

Before the code is explained, it's important to know that Hough **line detection represents lines with two values, theta and rho:**



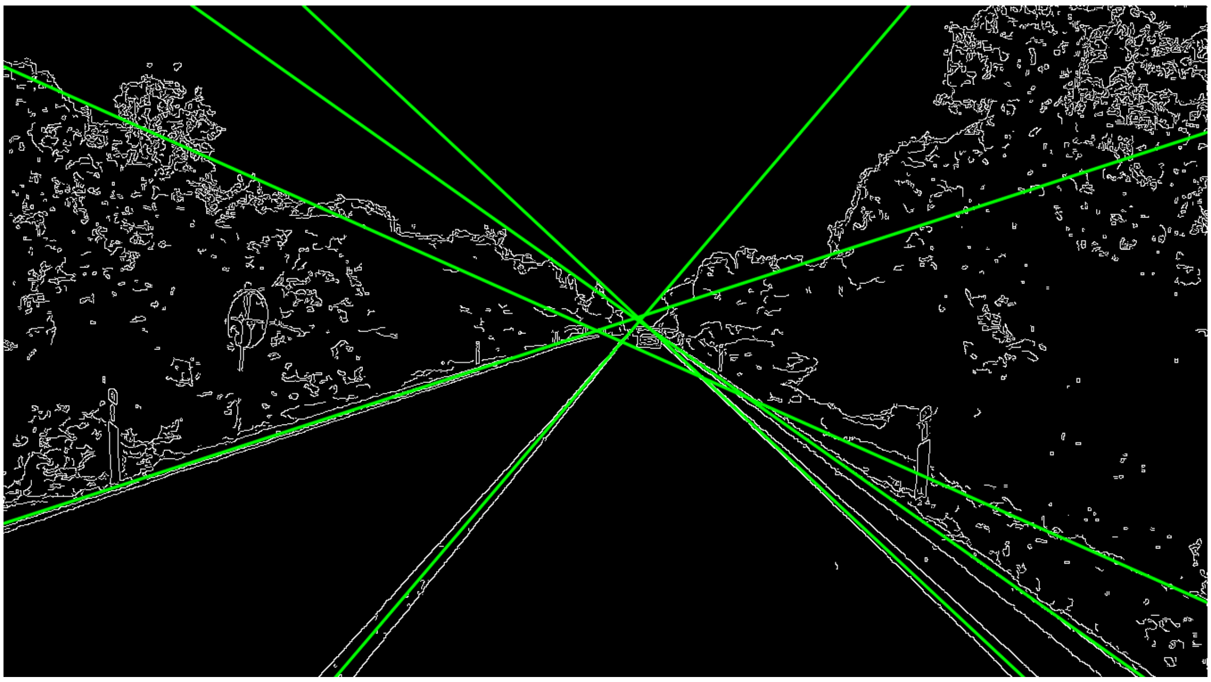
The line as shown in red above, can be described as being r (rho) pixels from the origin, and Θ (theta) degrees around the origin. Because of this **lines are stored in a Vec2f data type**, which stores two floating point values. A vector of lines will look like so:

```
vector<Vec2f> lines;
```

The HoughLines function has many arguments. The first is the edge map from before. The second is the array of lines where the output will be stored. rhoRes is the positional resolution of the detection, in this case 1 will be sufficient. thetaRes is the rotational resolution, M_PI/180.0 will set the resolution to 1 degree. Threshold is how sensitive the line detection is. If you get too many lines then increase this number, and vice versa. Start with a value like 100 and adjust it from there:

```
HoughLines(edgeMap, lines, rhoRes, thetaRes, Threshold, 0, 0 );
```

Since plotting a line based on theta and rho is not trivial, I've included a new drawing function that can do this for you called "lineRT". Using this function to draw the detected lines, you get something like this:



All the lane markers have been detected, along with some extra lines that need to be removed. **How you filter out the extra lines is up to you.** You could have theta and rho thresholds for the left and right lane markings, you could also look for line intersection points, or some other method. Cleaning up the edge map before searching for lines would help a lot too, so feel free to get creative.

After finding the lane markers, **indicate on the original frame some label that shows the detection in action.** Note, you only have to detect the lane you are currently in: