Samarth Negi
Internship Report
26 June 2018 - 28 July 2018

Problem Solving using Computer Vision

Conducted under
Mr. Munirathnam Pavuluri
Ananth Technologies

Computer Vision is essentially replicating the way we see the world in a mathematical fashion , while drawing out conclusions and making intelligent

Computer Vision is an interdisciplinary field focussed on teaching a computer to process image and video data .
Inherently computers only posses the knowledge of basic logic functions . CV is an attempt to make computers aware of the visual aspect of nature .
It is a subset of Digital Image Processing and has been garnering attention due to its uses in the fields of AI as being able to see is a very human concept and having a computer be able to see the same way as humans and make rational decisions is one of the cornerstones of AI .

While vision and the idea of perceiving things visually comes naturally to humans as stimuli thanks to the millions of years of evolution , teaching a computer to see and analyse this data is a very difficult task built over many layers of abstraction , and that is only on the software level . A lot of tasks are offloaded to standard libraries and predefined functions , using efficient mathematical techniques . This allows the programmer to focus on the problem solving aspect .

1
OpenCV

INTRODUCTION

OpenCV is a software library first released in 2000 by Intel .
The main goal was to achieve efficient image processing using simple functions to abstract complex mathematical operators and functions allowing a programmer to focus on the algorithm development part.
Since all digital images are essentially multidimensional arrays , opencv uses matrix computational methods to work on the raw data .

The core OpenCV library is written for C++ but I chose to work on its python port called opencv-python . While the speed of processing and efficiency takes a hit , the readability and the import modules of python compensate for most of the losses .

The OpenCV library contain modules for image and video imports from different sources even including webcams as default input source . Image analysis algorithms are built into openCV as functions with adjustable parameters for fine tuning the algorithms .

OFFICIAL DOCUMENTS

https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_setup/py_intro/py_intro.html

GITHUB PORTING LINK

https://github.com/opencv/opencv

All the programming has been done using python 3.5x and the code is backward compatible up to 3.x .
OpenCv is imported as a module called cv2 and any 3.x version will be able to run the code as is .
All additional test footage is derived from youtube , and the links referenced .

2
Raw Data

The raw data for all analysis will be images and videos . They are imported as multi-dimensional arrays from the system into the programme.

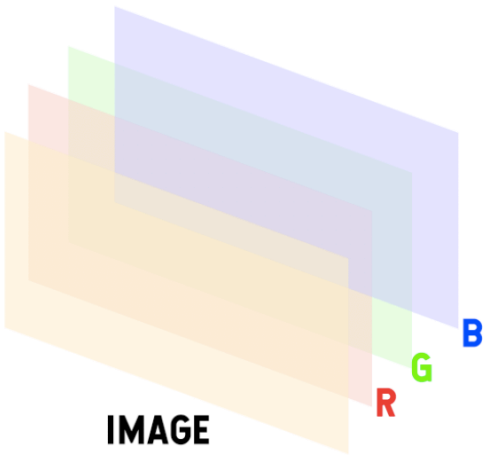
IMAGE PROCESSING

Images are essentially 2d arrays , each point in the array representing a pixel in the screen which is the intensity of white at that pixel . This forms simple greyscale images(black and white) .
To form colour images , several channels overlap over this 2d array , wherein the 2d array provides the light intensity and the channels provide the colour data at each pixel .

ARRAY CHANNELS

The number and type of these channels determine the colour space of the photo .
We have many different standards used for different media purposes . The most common one is the RGB . This matrix is basically a 3D array .
Example
[1080 , 720 , 3]
1080 , 720 = dimensions
3 = channels r , g , b

Other commonly used channels include cmyk , hsv .



VIDEO PROCESSING

Similarly videos are processed as frames . Each frame being an image . These group of images when played back at some speed (measured in fps) is called a video .
In OpenCv , we handle video processing just like we handle images but processing is done for each frame in succession as opposed to only one frame in case of an image data .
The speed of video playback in opencv depends on the processing involved ie. faster the processing of each frame , faster the frames are played back after processing .

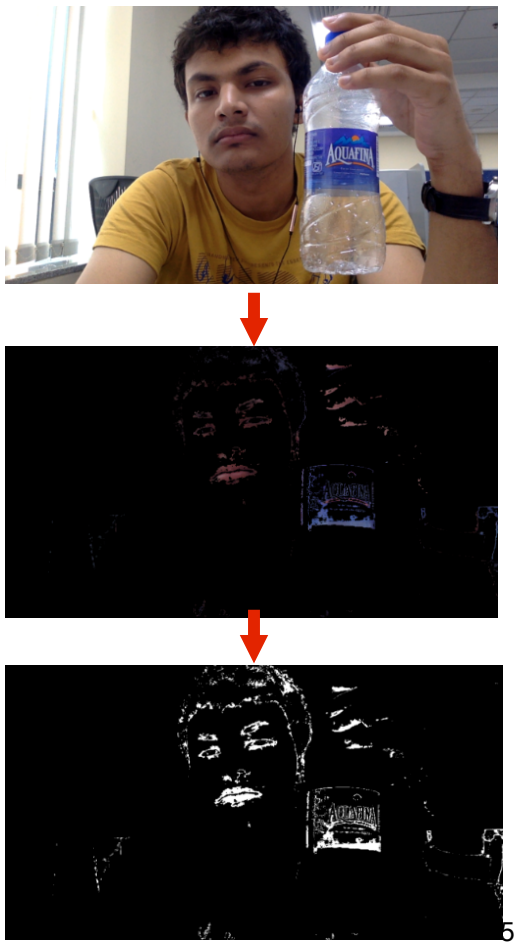
4
Techniques Used

IMAGE DATA AND ROI

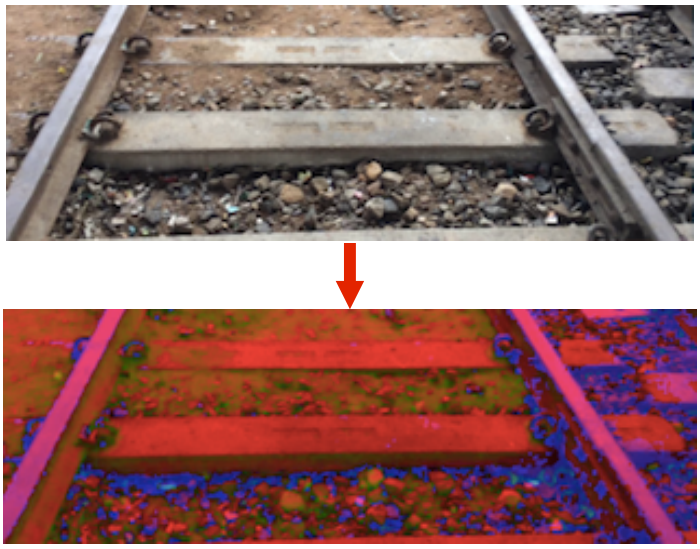
In each frame ,there will always be some redundant data . Since this pixel data is also being processed putting a load on the system , we try to reduce this redundant data . This is achieved by reducing the field of interest . In cv terms we call it ROI or region of interest .
We basically only take some part of the original image to process for each frame .
In case of a video we do it for every frame .
Now we have a single frame data in the form of an array that we can can work on .

COLOR FILTERING AND MASKING

We can filter out the image using thresholding parameters for shadows and colours
In the example to the left , i've used a combination of colour and shadow for an effect called relevant depth determination . It is an emulation of a preprocessing used in many mobile camera FX applications .
The first frame if the raw data that is being fed from the webcam , the second frame is the thresholded image generated by the cutoff parameters provided to the filter generator.
The final image as we have seen is called **the mask** . Creating these masks according to specification is a very important preprocessing task in opencv as we use this as our baseline for further changing the image and is used multiple times during the programme .



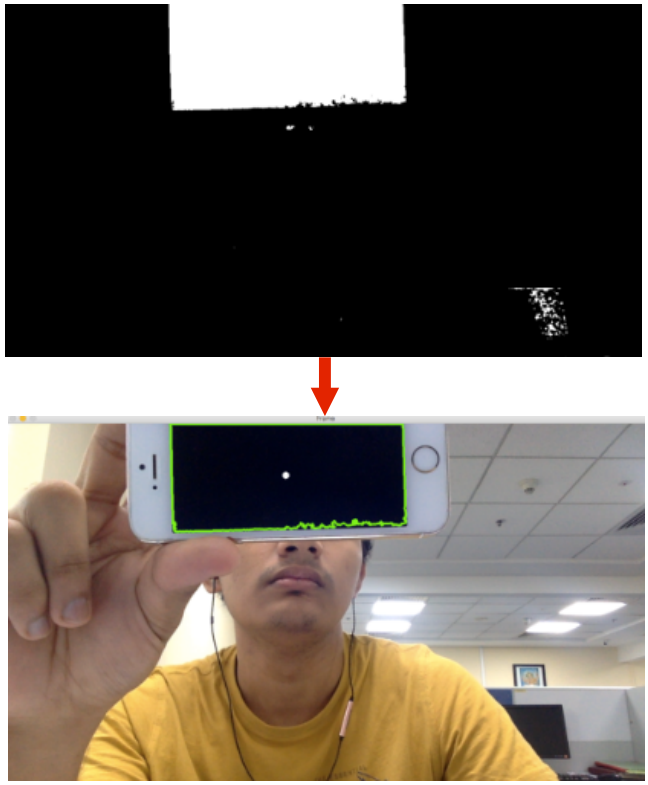
CHANGING COLOURSPACE



Colour is represented in many different ways in digital systems . For standardisation purposes , all these means of representation manifest as channels . The **RGB** channel is the most common one and is similar to what we naturally see . It is a 3 channel system which is based on values of red , blue and green colour . While this standard is very easy to recognise visually by humans , it is very hard to work with logically and develop algorithms for processing . The **HSV** colourspace is much more widely used in analysis , so we generally use it as a starting point .

The above photo is a photo in RGB space and the lower image is the same converted into HSV channel .

CONTOUR DETECTION

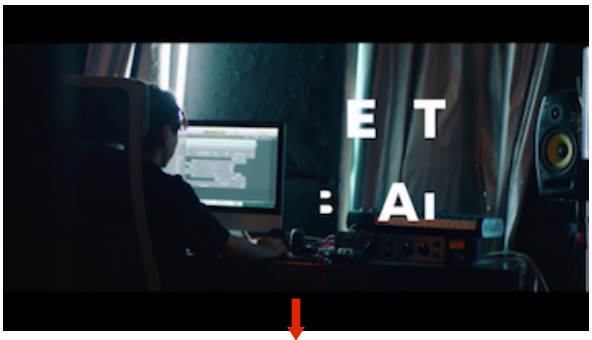


One of the most important features we can extract from an image is the contours present in it. They can give us an insight on how different objects are placed in a photo.
CV have many different ways to find contours in a frame , in the particular example shown i have used a mask to find black coloured contours and threshold it based on area using the **cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE** parameters .

The first image shows the mask filtered out from the original frame and the second image shows the contour drawn by a green line and also representing the centre of contour using a white dot.

EDGE DETECTION

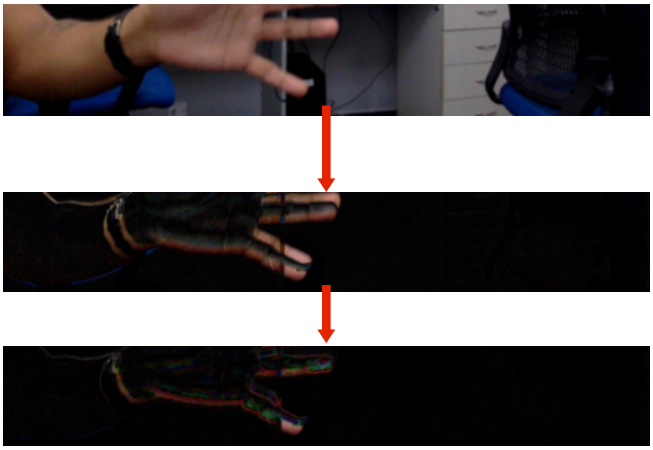
Edge detection is a method of finding out edges in a frame . OpenCV achieves this by processing frames using the a canny operator . Edge detection is also an essential tool in image analysis as edges of processed images helps in segmentation of frames into different regions of interest .
The most common edge detection algorithm used is the canny algorithm which is based on the sobel operator . This operator essentially finds derivative of light intensity on the x-axis and the y-axis and then plots the magnitude above a given threshold . This results in only the edges remaining in the frame .

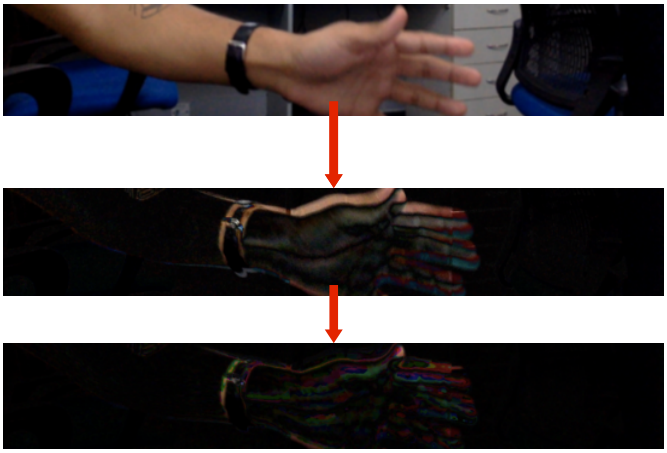


The following example frames are a part of a video , the first frame shows the original raw frame , the second shows the same frame in the grayscale colourspace . since colour intensities can be judged better in black and white photos , the input to the canny operand is normally a black and white image .
The third and final frame shows the canny operator in action where the image is divided into lines showing edges .

IMAGE SUBTRACTION

It is a very effective method of reducing noise in a given frame . The basic principle is to compare 2 consecutive frames and to only show pixels which are changed over the course of frames . This leaves out the relevant data or large objects which can then be mapped or processed over .





The following frames are screenshots of a video stream captured from my webcam , ie. all the processing is dynamic and being done simultaneously.
The first photo represents the main frame and the second photo shows the changing pixels . The third photo shows the result of **bitwise and** operator performed on the first and second photos .
As it is clearly visible i was able to isolate relevant data in the video frame .

4
Application

Problem Statement : To Detect Faults and Breaks in a Train Track Using Video Footage from a Drone .

AIM

The main goal of the project is to detect discontinuities on a train track .
In the current state , the process of track verification is manual and extremely labour intense , this solution is an attempt to automate almost all of that process and bring in more reliable and accountable systems in place to fix the issue .

SPECIFICATIONS

1. For a given video of a train track shot from a drone , the video processing algorithms should be able to reliably detect whether the track is damaged .
2. The solution should account for natural and humans interference in the video and have appropriate measures in place .
3. The solution should be efficient and should be able to work on large data sets .
4. The solution should minimise the need of human intervention .

DEVELOPMENT PROCESS

The entire cycle of development has been documented below starting from development of basic tools and libraries used all through to a working software , including patch notes for each major push with reference to the code being used .
All the code is available in the GitHub repository :

https://github.com/n-s405/train-track-tracking

PHASE 1 : CLASSIFICATION CRITERIA

The stating point for this project was a single image of a train track . The objective being, classification of what counts as a railway track in an image .
I took the following frame as a sample .

In a meeting with my mentor , we came up with 3 parameters which we could classify as being a good indicator for a train track .
The thinking behind this was to logically pen down the things that we as humans use to differentiate the train track from the rest of our vision .

1. **Corners and Edges**
    1. Train tracks always form a continuous edge throughout the image which separates it from the rest of the image .
    2. Edges can be represented as a collection of a large number of corners .
2. **Color**
    1. Regardless of the environment , we can make out a track due to its colour .
    2. Colour of a train track generally does not change abruptly .
3. **Depth**
    1. We as humans can see as 3D , so we can see the depth and make out the tracks from the ground .
    2. While images are 2 dimensional in nature , we can ascertain depth to a certain degree using colour gradients and shadows .

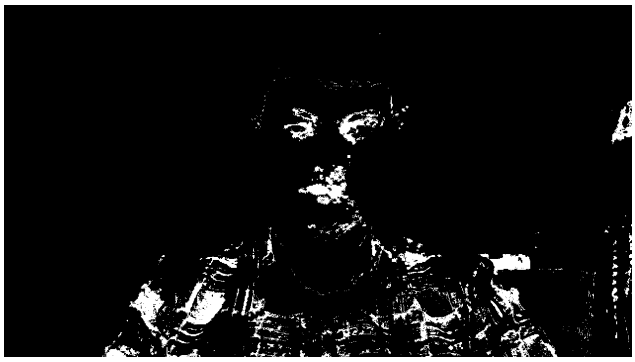
Taking these 3 parameters as my core classification criteria I began developing tools which can help in isolating these from the image and thus giving us a new frame with only the tracks to work on .

COLOR FILTERING

The first task I took up was filtering a certain colour out of the main frame . In this case , i focussed on silver/grey colour range .
All the colour calculations were done in the HSV colour space .

code referece : **27_06_color-filtering.py**
This is a tool i built for colour isolation is to just isolate a random colour from a given video/ image stream . I picked the blue-red colour range for this particular example .
By default it takes the video stream from the webcam and masks out only the blue and red colour range in the frame . It is highly specific and does not work in all conditions but is only set for a certain range that is changeable in code .

INPUT



MASK

OUTPUT

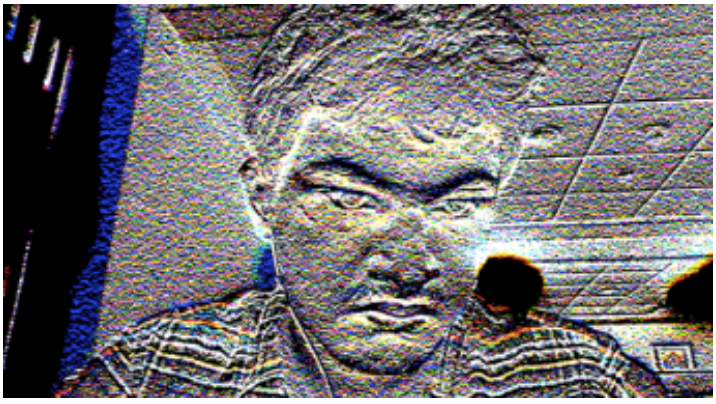
FINDING CORNERS

The next module i built was the edge detector . It is based on the **sobel operator** which is a mathematical operator used to find differentials in a matrix above a certain threshold .
The basic idea is that whenever there is an edge in an image , there will be a drastic change in the colour composition . This change is referred to as the differential and forms the basis of edge analysis . It works on x and y axis separately to give us the edges .

code reference : **28_06_sobel-testing-01.py**
This module basically finds differentials and plots them as an image . The darker parts means a high differential value ie. an edge , a lover value ie. not an edge has a lighter shade .

INPUT

SOBEL x-axis

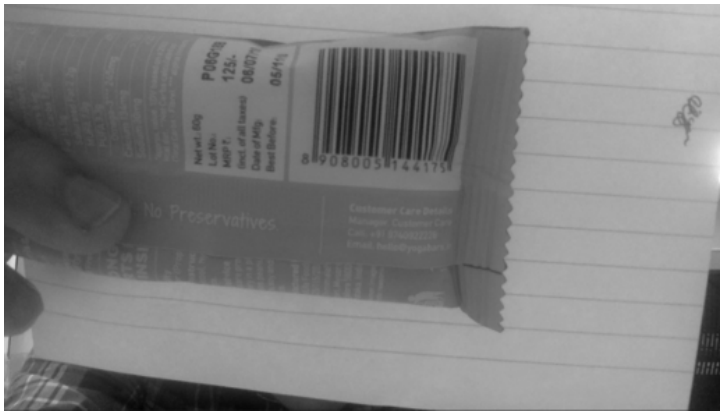


SOBEL y-axis

FINDING CORNERS

The next module was the corner detection module . This is based on the Shi-Tomasi Algorithm for finding corners which in itself a modification of Harrison's algorithm for finding corners .

code reference : **29_06_shiTomassi-corner-testing-01.py**
This module basically plots the strongest N corners in an image with a blue dot from an input image . For the ease of testing , I've chose the webcam as the default input stream . The image is first converted into a black and white photo since it is easier to find corners in a greyscale image .



INPUT

OUTPUT

PRACTICAL TEST 01

This is the first test I performed on a sample track image .
The goal was to get the code to have some level of **track recognition from a still image** .

INPUT : An image of a train track , zoomed in , converted to greyscale

OUTPUT : Excluding the extra corner points plotted , the software was able to detect track perfectly and made a straight line around the grey track .

CONCLUSION : While there was some recognition , this is an extremely ideal condition .The colour aspect was not addressed and there are a lot of false points generated which can skew the real test .

PRACTICAL TEST 02 [core engine mk1]

code reference : **02_07_core-engine-mk1**

In this second iteration of the main programme , I combined all the preceding elements of colour filtering , masking , roi , colourspace shifting such that the system is able to **clearly differentiate the track from the surrounding** in a video file

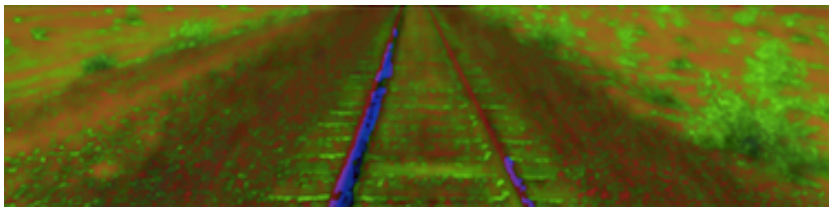
INPUT : A video of a drone footage of a train track , with no preprocessing .

STEP 1 - SELECTING A REGION OF INTEREST

A small region of the video is extracted based on the area where relevant information is present reducing the number of pixels to compute thereby increasing efficiency and reducing the additional junk data

STEP 2 - CHANGING COLOURSPACE

The video is converted to HSV colourspace for easy analysis

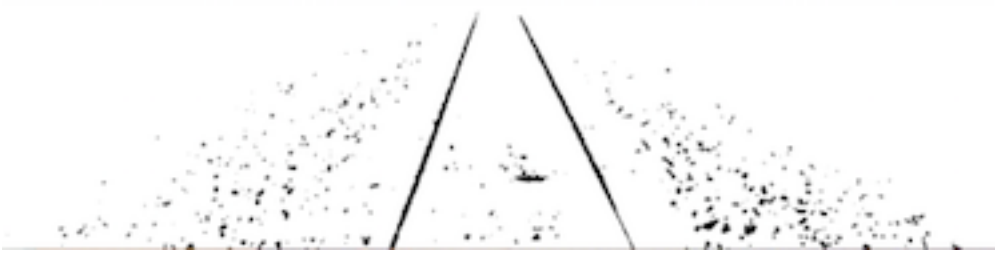


STEP 5 - COLOUR BASED MASK FILTERING

A mask is generated based on the parameters for the colour of the track isolating only a part of the frame thus giving us a heavily reduced frame with only the track and some noise

INVERTED FINAL OUTPUT

The final output of this iteration of the core engine

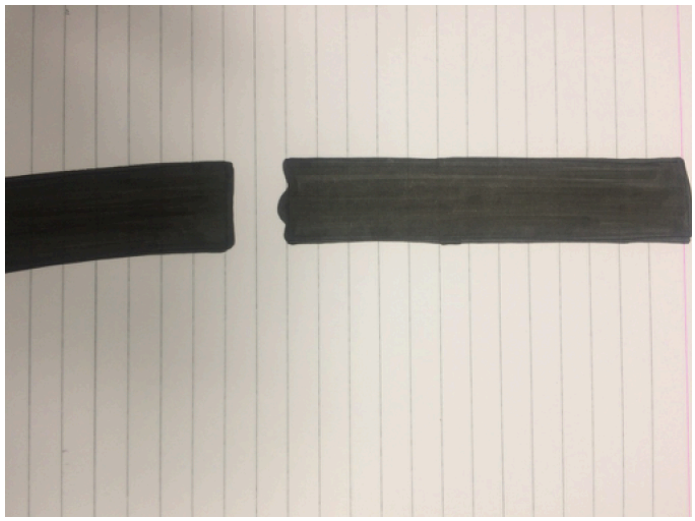


CONCLUSION : This time the reduction of additional data was very accurate and I was able to reduce most of the surroundings and thus isolate the track . While this is an improvement , there is still considerable noise in the frames . This is majorly due to rocks and sand in the surrounding which lie on the same colour range as that of the track .

PHASE 2 : TRACK DEFECT IDENTIFICATION CRITERIA

Now that I am able to narrow my programme view to focus on the tracks , the next problem i tackled was how to actually find if a track is broken at some particular place .

For this i considered a hand drawn image of a simple broken track as shown below .



From this picture I realised that if a track is broken at a location , there will be 2 easily differentiable pieces in the frame ie. 2 different solid shapes that would exist .
If a track were intact it would be one continuous rectangle .

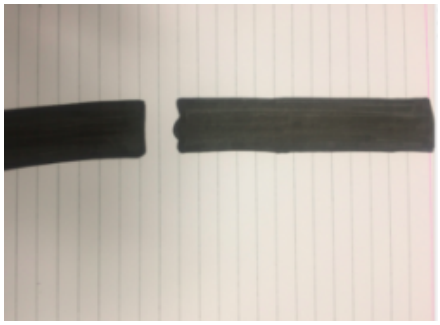
Using that logic as my criteria , i began developing a module to count the number of contours in a given frame . So I ended up with this general rule for finding defects in the track :
If the number of contours in any given frame are more than 1 , then there is a possibility that the track might be broken at that point .

FINDING CONTOURS

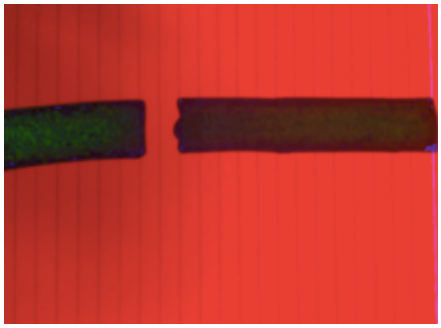
code reference : **03_07_black-contour-counter-photo-01.py** **[image input]**
04_07_black-contour-counter-video-01.py **[video input]**

Any bounding area formed with a continuous line can be referred to as a contour . In CV , finding contours is very easy and can be achieved using a simple library function with threshold parameters . For this application i have placed a constraint on the area to be larger than a set value as the train track will be the largest object on the frame .

For demonstration i have taken my drawing on paper as input and ran it though my contour detection algorithm . The following result plots the contours present and even returns the number of contours present in the terminal window .

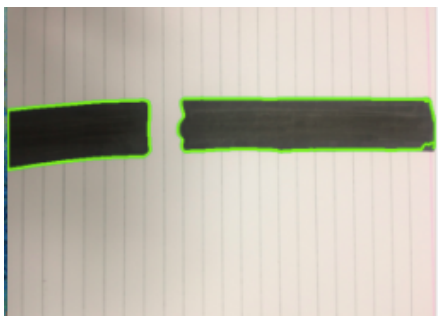


INPUT



HSV

MASK



OUTPUT

CANNY EDGE DETECTION

code reference : **03_07_canny-edges.py**

This is a method to find the edges of a frame , just like the sobel operator but it adds an additional step to it and gives us better result .

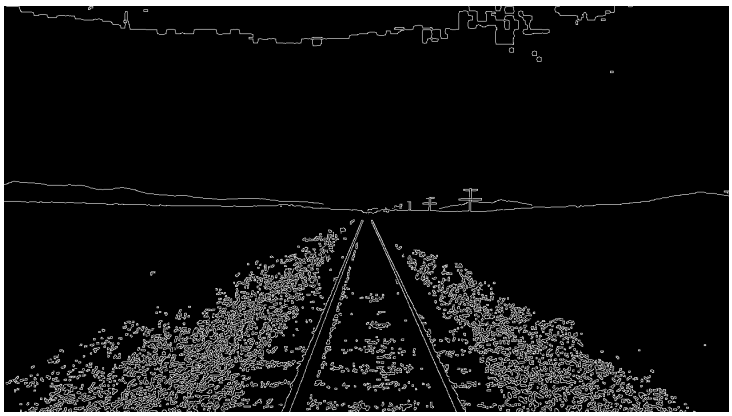
The input of this function is actually the output of a sobel operator as discussed in the previous sections .

It gives better result as it predicts the true edges and gives thin lines in space which can further be used for line detection . It is a technique heavily used in CV applications and is present in almost all image processing tasks in one way or form .

For demonstration of this , we are taking a video of train track as an input and the canny edge detector will return us some plot lines which we can plot on the frame to give accurate lines .

INPUT

COLOUR FILTER AND BLUR



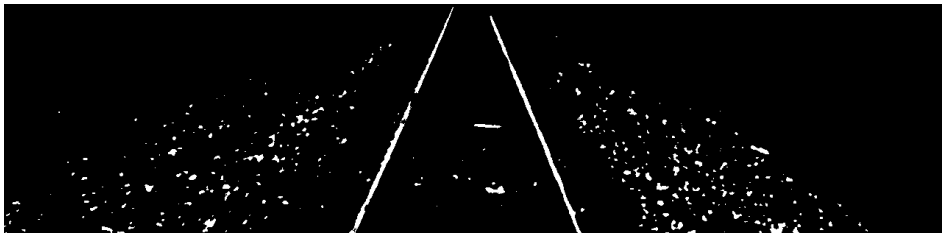
OUTPUT

HOUGH LINE TRANSFORMS

code reference **: 12_07_core-engine-mk2.py**

The next concept I learned about was the hough line transform . It is basically an algorithm to find aligned points in images that create a line . The input can be very varied but the output is always a set of lines given in terms of the end and starting coordinates . Now that we have a clear view of the track , i planned to start measuring the length of the track .
With hough line transform , I generated a set of lines only over the train track with a green colour digital line .
This process works for video as well as an image input stream .

INPUT



MASK

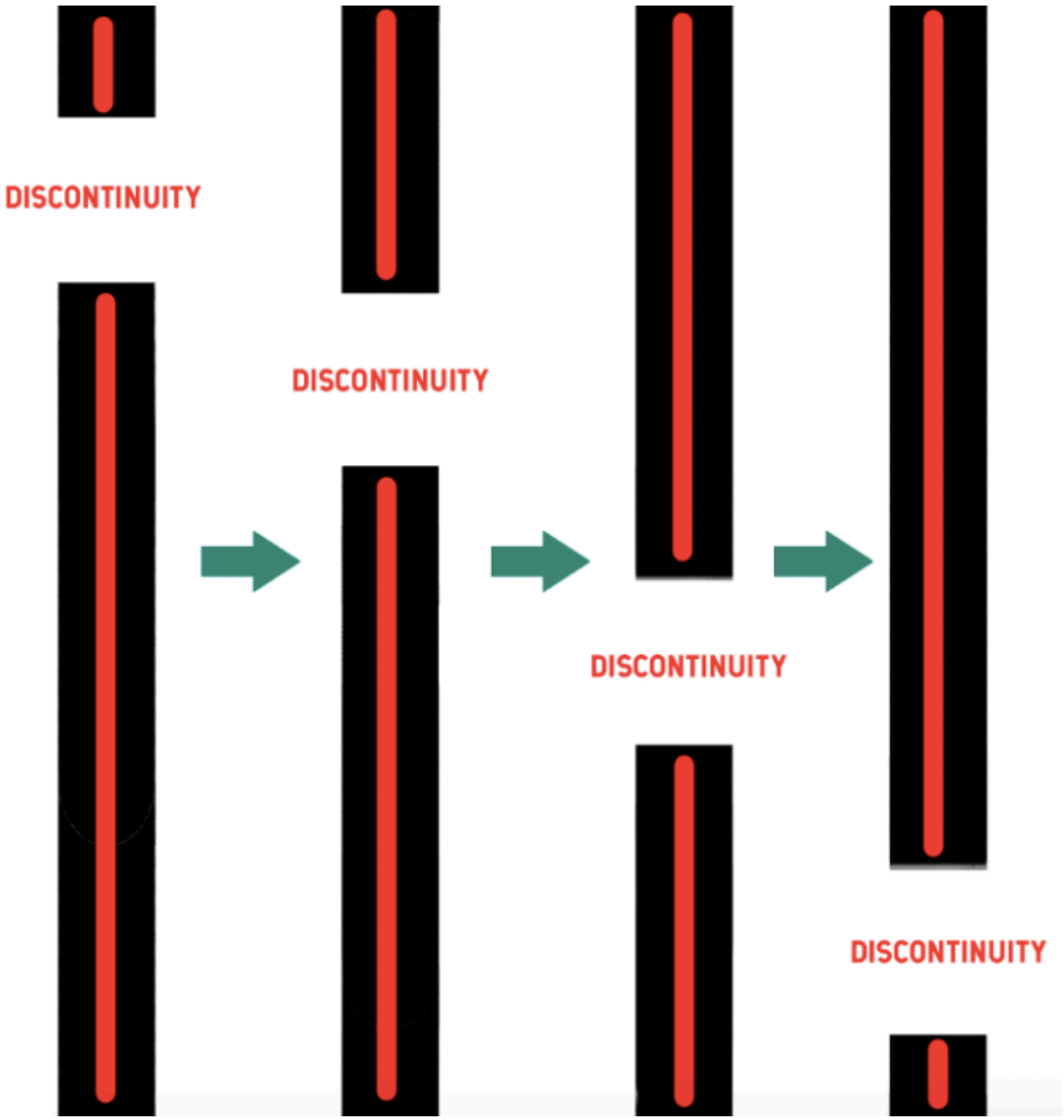
OUTPUT

PRACTICAL TEST 03 [core engine mk3]

code reference : **12_07_core-engine-mk2.py**
In this iteration of the programme , I have developed a working algorithm to test out the working of the programme . In a meeting with my mentor , we came up with a universally applicable rule to classify whether the track has a discontinuity or not , the only given requirement is that the track is identified .

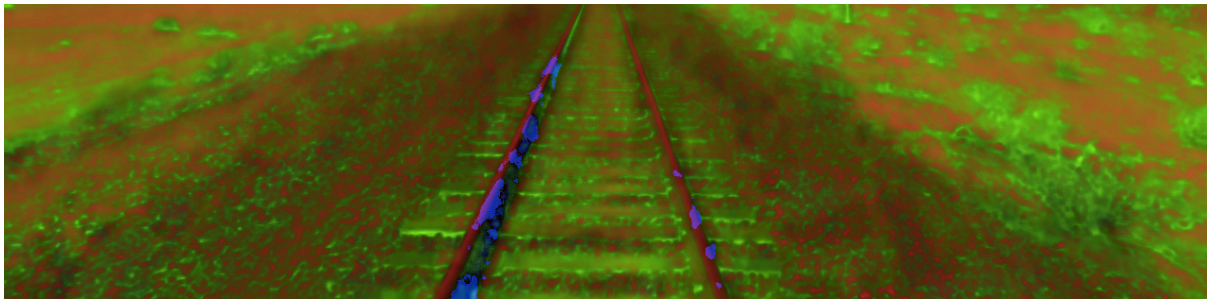
LOGIC USED : In a frame , we plot lines over the tracks using hough line transforms and are constantly calculating and updating the average length of a line , in case of a discontinuity , there will be a drop in the average which we are taking as a classification criteria . This can be illustrated in the image below .

the red lines are the track hough lines used for track length determination

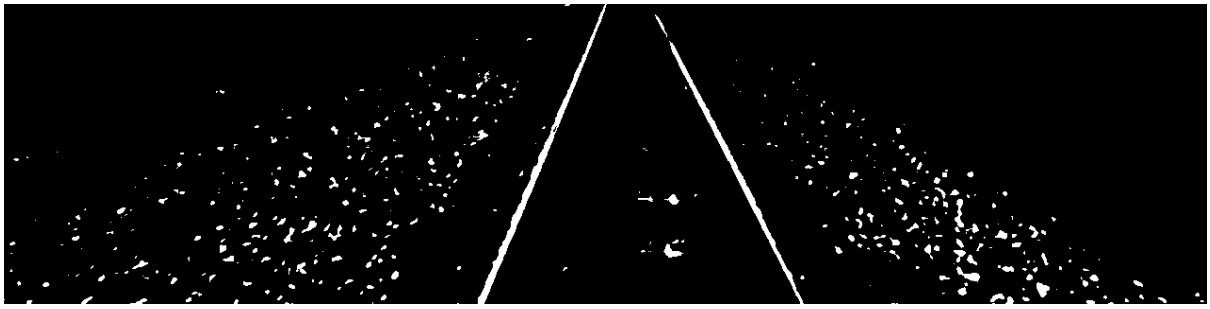


INPUT : A video of a drone footage of a train track , with no preprocessing .

STEP 1 : BASIC PROCESSING
Selecting a region of interest and changing colourspaces.

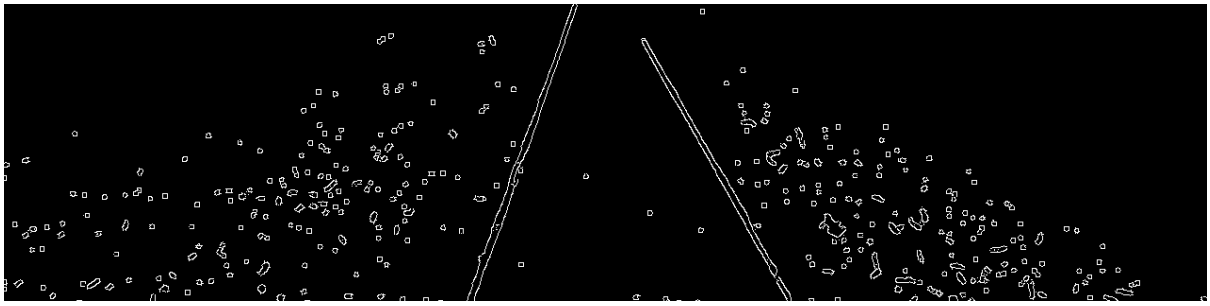


STEP 2 : MASK CONSTRUCTION
Generating a colour filtered mask to construct a better canny output

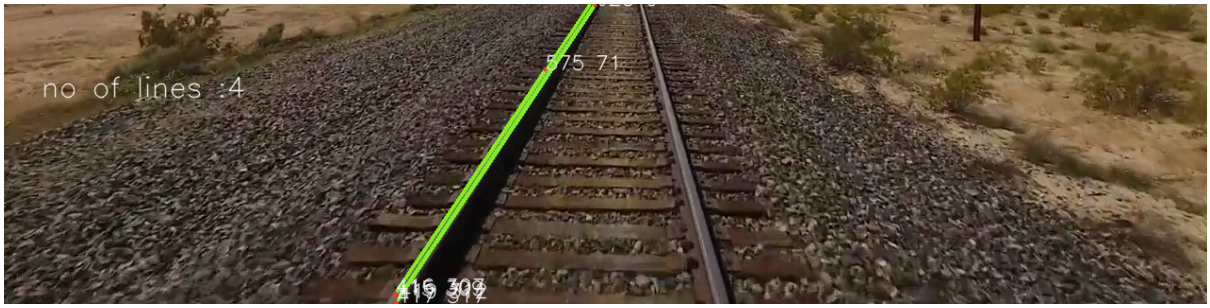


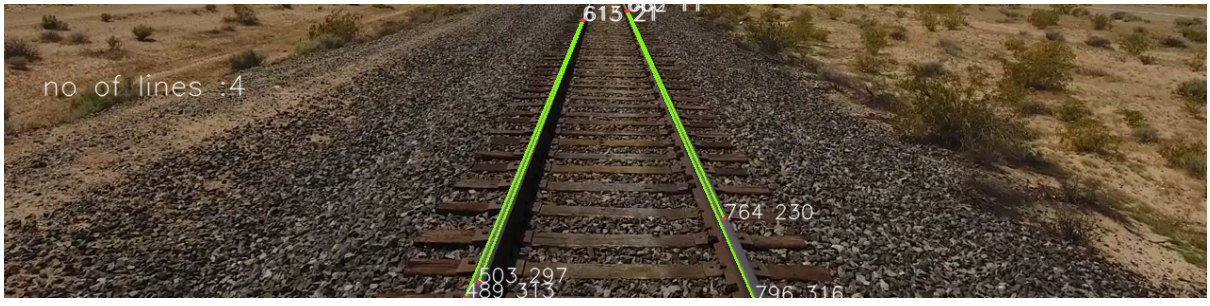
STEP 3 : CANNY EDGES

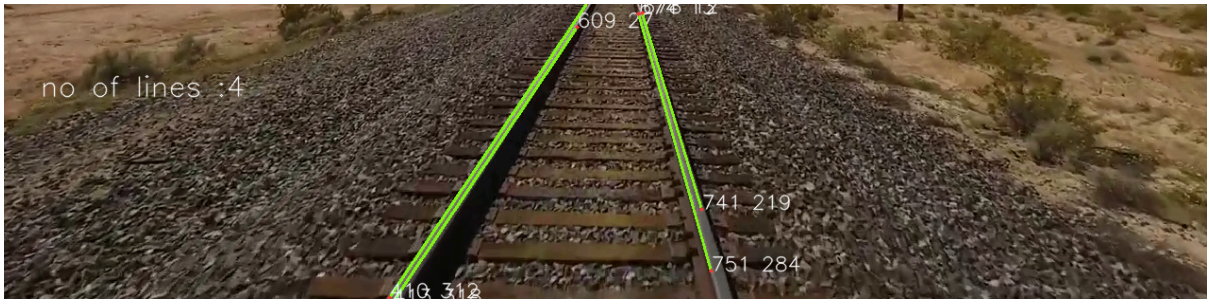
Running the mask through a canny edge detected for generating smooth and continuous edges



STEP 4 : OUTPUT





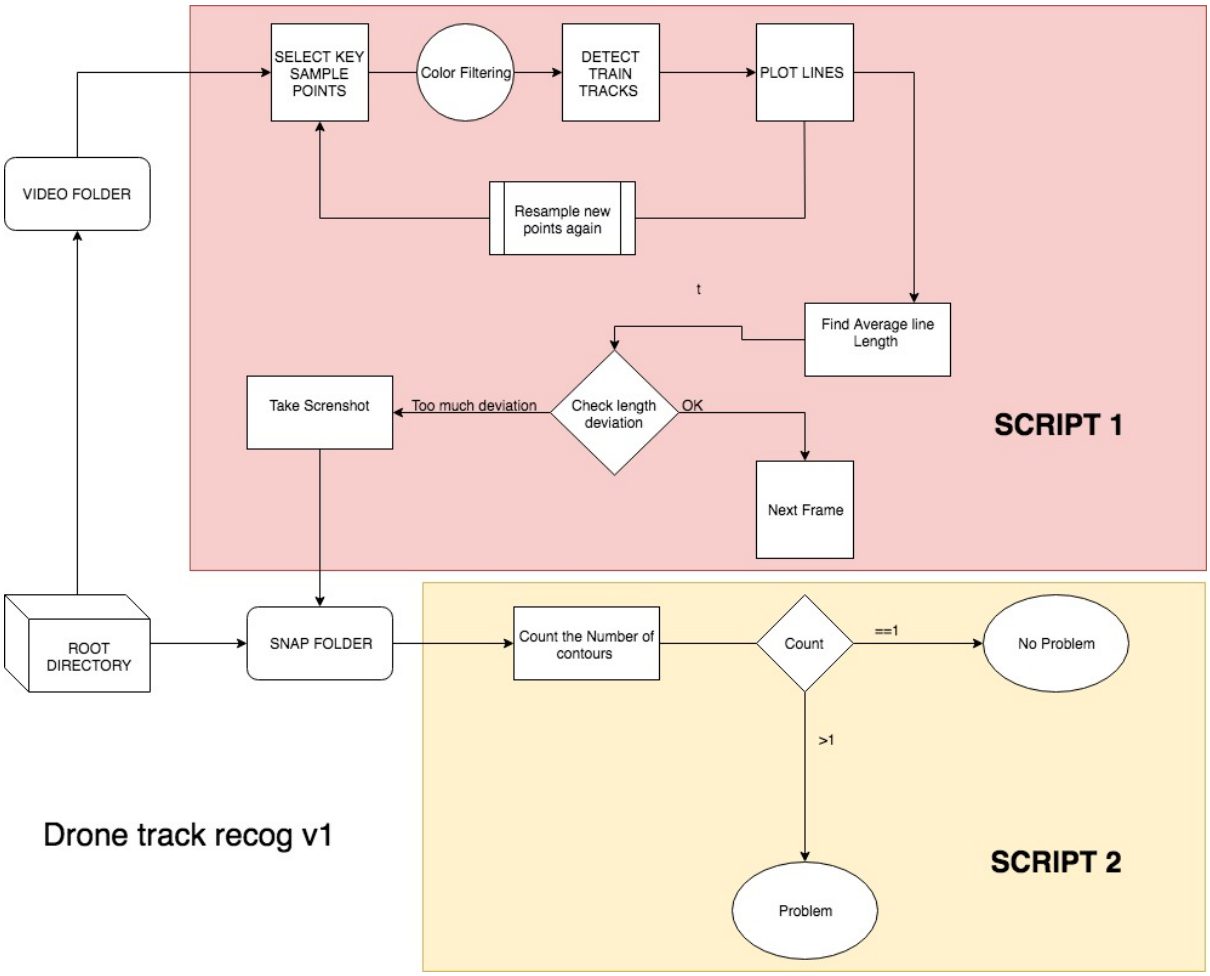


CONCLUSION : This version of the software is extremely accurate and accounts for almost all the specifications we wanted . All but one , changes in environment colour . The problem which I discovered in this software was that it was constrained to tracks which were grey/silver in colour and very susceptible to changes in environment tones and colouring . To make this system fully robust these are 2 very essential key points that needed to be solved .

PHASE 3 : FINE TUNING THE PROCESS

This phase is all about adding redundancies and including special cases within the programme code to make the software easier to use for the end user .
It includes Dynamic Colour Picking and Sampling modules and the Image Frame by Frame Subtraction modules .
These 2 methods were made specifically to include colour tone changes .

Finally , the software will have 2 different scripts running , the first being a preliminary broad range test and the second being a highly specific and accurate testing .
This has been illustrated in the flowchart shown below .



COLOUR SAMPLING

code reference :**17_07_click-mask-generator.py** **[only the masking]**

18_07_core-engine-mk3-2.py **[full analysis included]**

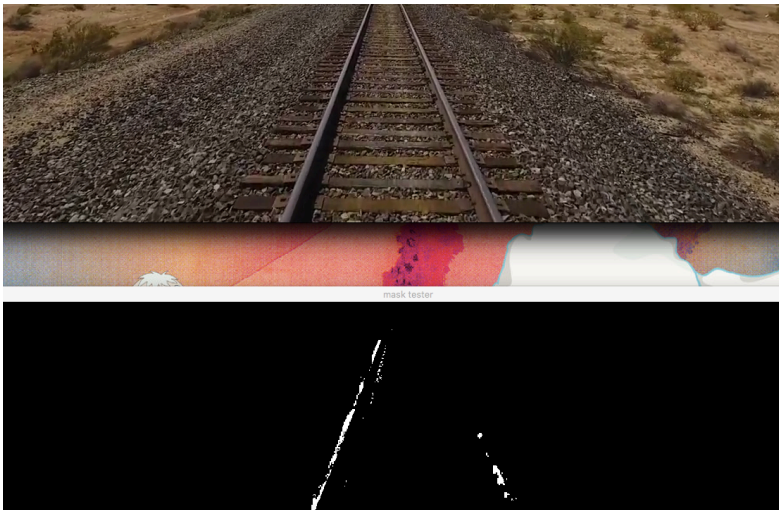
This concept does not stem from any preexisting mathematical theorem or process , it is purely a logic based algorithm with a simple and intuitive approach .

For a given video , press p to pause the video and select sample points for the new track .

This adds a manual step to the process but makes the processing very accurate and once the person has entered the sample points , these points are updated dynamically from the new points generated .

s

INPUT
all windows empty



MASK EDITING WINDOW

OUTPUT WITH DIFFERENT SAMPLE POINTS

POST FRAME PROCESSING

code reference :**19_07_differential-noise-reduction.py** **[for mask overlap]**
23_07_difference-engine-test.py **[for frame overlap]**

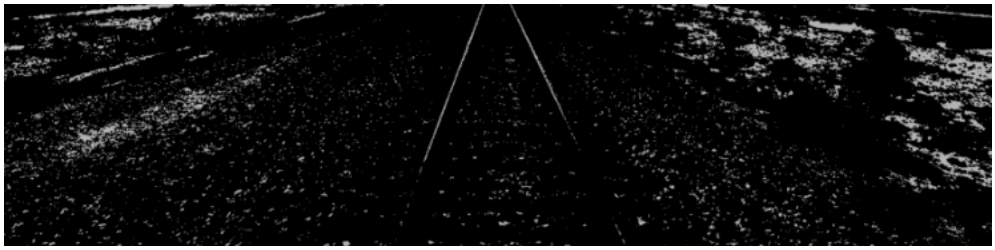
A lot of information can be obtained from comparing concurrent frames . Post frame processing is basically an implementation of a feedback system where i put in the output of the last cycle as the input to the next cycle .

Practically , if we look at the images , the train track from one frame to other are almost constant with respect to their colour and placement in the frame . They might move left to right but that takes process take over a lot of frames . I used this approach and started processing 2 frames , the current and the previous frame .

With this algorithm in mind , i built 2 modules . One for clearing out masks by reducing noise and the other which can help localise the train track in an untrained video .

1. NOISE REDUCTION USING OVERLAP CANCELLATION

INPUT

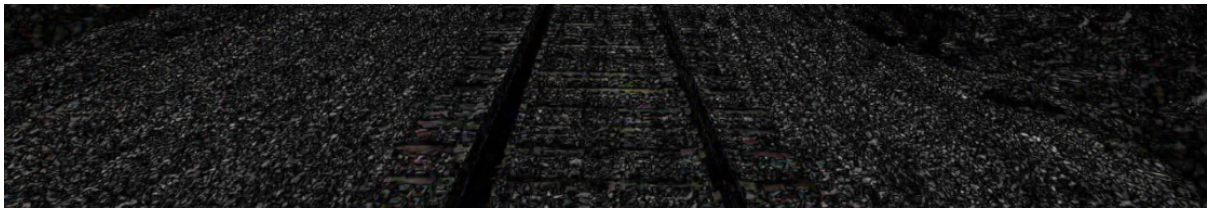


BASIC MASK

NOISE REDUCED OUTPUT

2. DIFFERENCE ENGINE

INPUT



OUTPUT