Embedded Machine Vision and Intelligent Automation

Exercise 4

Sarthak Jain

July 19, 2019

Performed on Jetson Nano

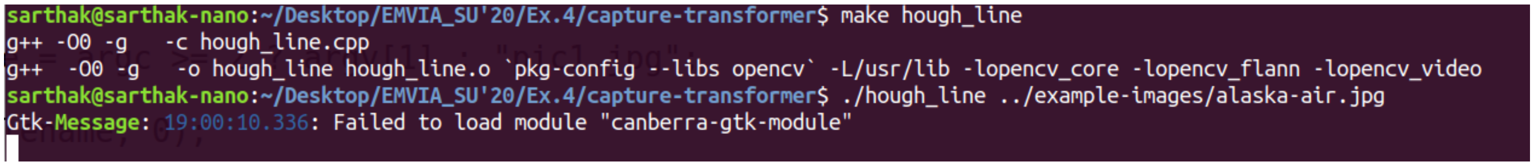
Question 1

1. The paper suggests the problem of using Hough's slope-intercept parameterization. However, in some particular cases, like that of a strictly vertical line, the slope becomes unbounded. The suggested solution is to use normal parameterization. In this way, they have obtained a set of generalizations:
   a. A point in the picture plane corresponds to a sinusoidal curve in the parameter plane.
   b. A point in the parameter plane corresponds to a line in the picture plane.
   c. Points lying on the same straight line in the picture plane correspond to a set of curves passing through a common point in the parameter plane.
   d. Points lying on the same curve in the parameter plane correspond to lines through the same point in the picture plane.
2. In order to reduce computational load, the paper considers only specific regions and ranges over which to quantize the points in the parameter plane. They treat the quantized region as as two-dimensional array, and with each instance of (θ, ρ), update the array. In this way, they show that the number of computations for 'n' figure points has reduced from n^2 to n*d, where d is the range of values θ can take.
3. The paper discusses two alternate interpretations of the point-curve transformations introduced. One interpretation is that ρ locates the projection of the point in the figure plane onto a line in the parameter plane passing through the origin with slope θ. From this, they conclude that if a number of points lie close to a line 'l', their projections onto the line normal to 'l' are nearly coincident.
   The other interpretation discussed by them is that if we follow a curve in the parameter plane, we are actually keeping track of all the lines passing through a particular point, and whenever another such point recurs, it is simply a matter of incrementing the value in the 2-D array.

To summarize, the methods discussed in the paper can be generalized for any family of curves, not just a family of straight lines. The paper attempts to analyze the base-level problem in computer vision of detection of straight lines. The problem is that in computer vision, straight lines are sometimes disjointed, which leads to separation of points, making them co-linear points instead of a straight line. This can be solved by finding the lines formed by all points, but this method is too computationally intensive. The paper goes on to consider the normal parameterization as a method of detecting straight lines, as compared to Hough's method of slope-intercept parameterization. The paper proposes a more computationally efficient of the Hough transform, by reducing the computational load greatly, and can be considered a huge contribution to computer vision.

Question 2

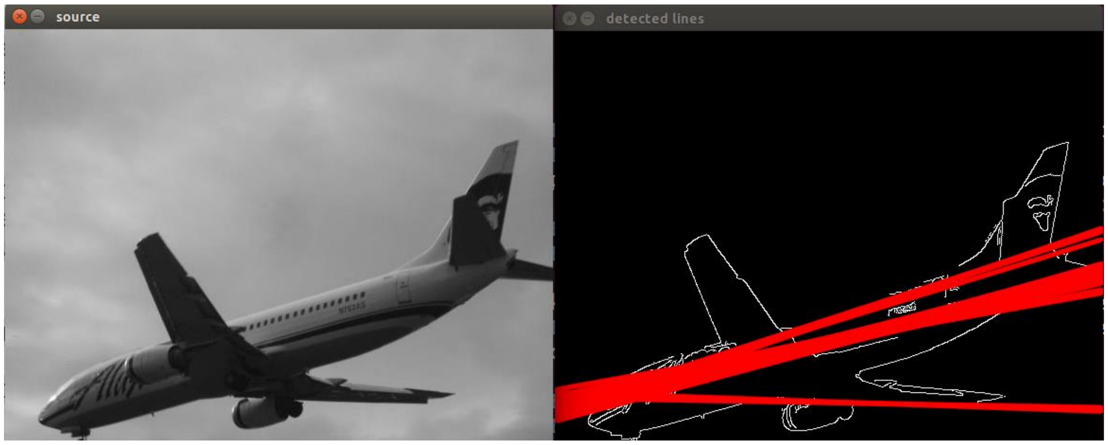
The program attempts to extract the Hough Line features from a given feature, and draws actual lines along them to make them recognizable. There are two methods of doing this as well, the first is discussed below.



```
sarthak@sarthak-nano:~/Desktop/EMVIA_SU'20/Ex.4/capture-transformer$ make hough_line
g++ -O0 -g   -c hough_line.cpp
g++  -O0 -g   -o hough_line hough_line.o `pkg-config --libs opencv` -L/usr/lib -lopencv_core -lopencv_flann -lopencv_video
sarthak@sarthak-nano:~/Desktop/EMVIA_SU'20/Ex.4/capture-transformer$ ./hough_line ../example-images/alaska-air.jpg
Gtk-Message: 19:00:10.336: Failed to load module "canberra-gtk-module"
```

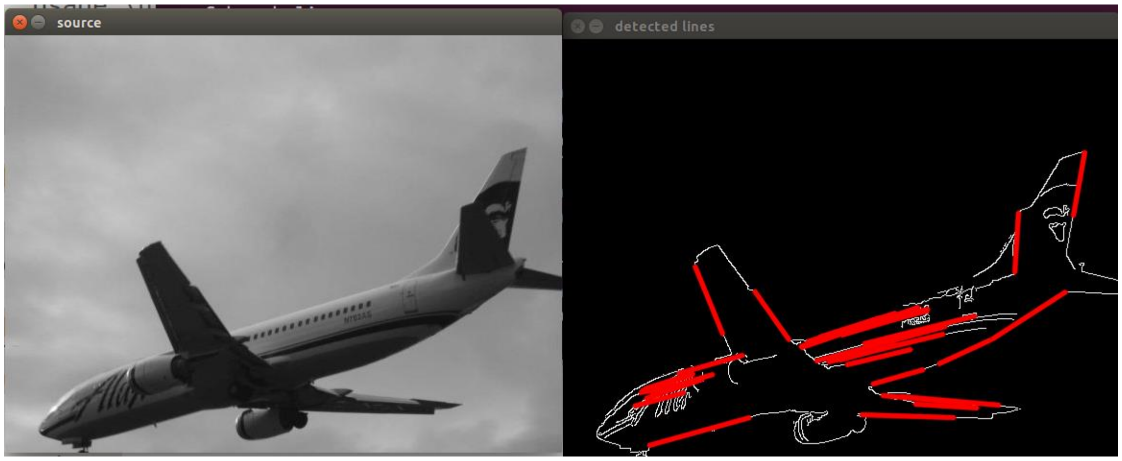
Build and test

1. The Standard Hough Transform identifies a set number of pixels identified as M edge points in the given parameter space. It detects all colinear points by considering the points and finding their normalized parameters. If coincident, the line is considered continuous and the next point is checked for collinearity.



Standard Hough transform

2. The Probabilistic Hough Transforms differs in the manner that it does not consider all M edge points, but a small subset m of them. It works on these randomly selected points, and computed lines between them. The smaller the value of m, the less accurate the detection of lines. If m is kept at an optimum value to make sure that even small lines are detected rather than being discarded like by the Standard Hough Transform, Probabilistic Hough Transform can prove to be a better method.



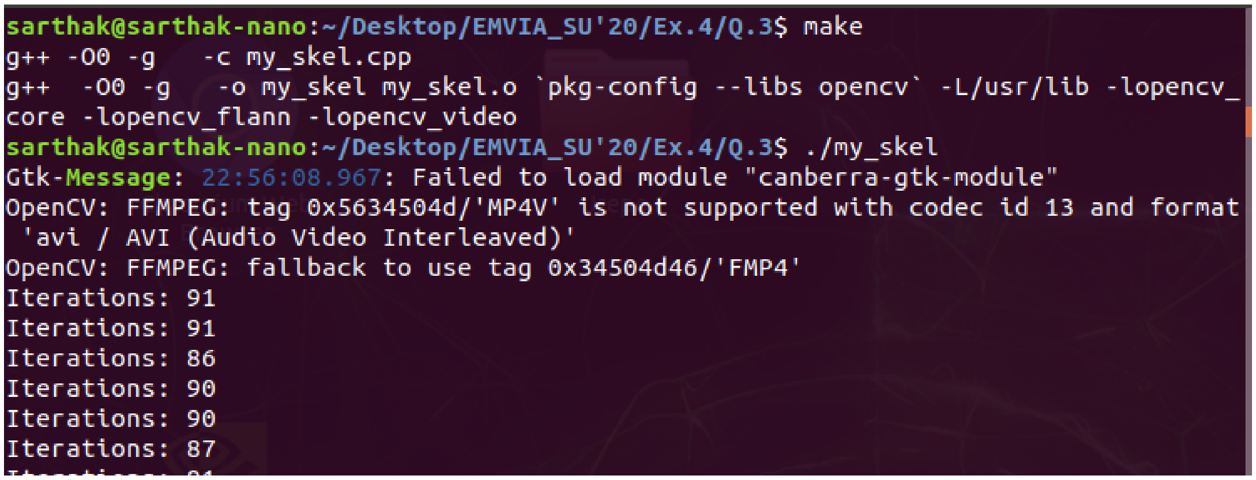
Probabilistic Hough Transform

The code works by first using the Canny edge detector as a preprocessor. The standard Hough transform computes the lines array as described above, and the points to be connected are computed using the function x*Cos(θ) + y*Sin(θ) = ρ. They are connected using the 'line' function.

The Probabilistic Hough transform on the other hand uses a smaller subset of points, and instead of using the above formula, draws lines between each instance of the array lines. This leads to a better detection of lines in the image, as can be seen.

Question 3

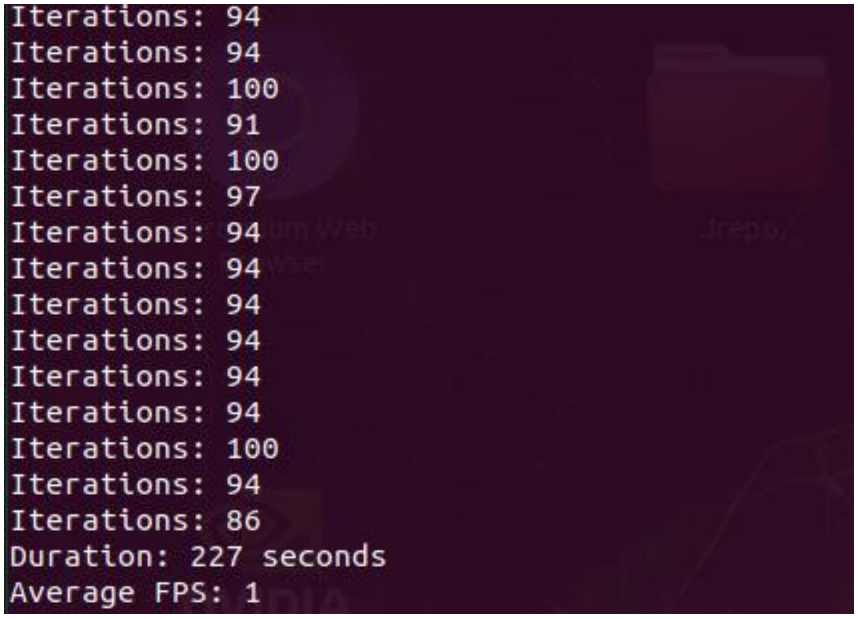
The code follows a simple pattern of program, similar to Prof. Siewert's 'skeletal.cpp'. The only difference is that it takes its input frames from a camera, processes them with the skeletal transform, and then re-encodes them into a video. The program first converts the input frame into grayscale, then thresholds the same using the OpenCV thresholding function with the threshold value set to 100, converting the frame to a bitmap format. This threshold was arrived at with a lot of testing, and depends greatly on the background as well as the lighting conditions. In my particular lighting conditions, 100 was found to be sufficient. Image subtraction is carried out next, since the foreground is generally darker than the background. Subtraction leads to the foreground actually coming to the fore, being given a pixel value of highest. Median blurring is carried out to remove any possible noise.



```
sarthak@sarthak-nano:~/Desktop/EMVIA_SU'20/Ex.4/Q.3$ make
g++ -O0 -g   -c my_skel.cpp
g++  -O0 -g   -o my_skel my_skel.o `pkg-config --libs opencv` -L/usr/lib -lopencv_
core -lopencv_flann -lopencv_video
sarthak@sarthak-nano:~/Desktop/EMVIA_SU'20/Ex.4/Q.3$ ./my_skel
Gtk-Message: 22:56:08.967: Failed to load module "canberra-gtk-module"
OpenCV: FFMPEG: tag 0x5634504d/'MP4V' is not supported with codec id 13 and format
 'avi / AVI (Audio Video Interleaved)'
OpenCV: FFMPEG: fallback to use tag 0x34504d46/'FMP4'
Iterations: 91
Iterations: 91
Iterations: 86
Iterations: 90
Iterations: 90
Iterations: 87
```

Build and test

Now, the main logic of the skeletal transform comes into picture. A flag is set to ensure the program does not run on a black image which has no intensity at all. A loop is run for a count of 100, in which the frame is first eroded, then dilated, using a 3x3 structuring element. The function erode removes border pixels about any object in the image in the 3x3 neighborhood. However, this leads to loss of data. To regenerate the data, dilation is carried out, which extends the actual object by a 3x3 neighborhood. Now, subtraction of this dilated image is carried out from the median-filtered image, leading to only the excess part having been added left in the new image. This frame is stored in a Mat object, and in each iteration, the excess boundary is added to the Mat object. At the end of the loop, we are left with a close enough approximation to a skeleton, along the boundaries of the original image objects. The skeletal objects are re-encoded into a video, and a sample screenshot of skeleton is provided below, along with the build and run of the program.



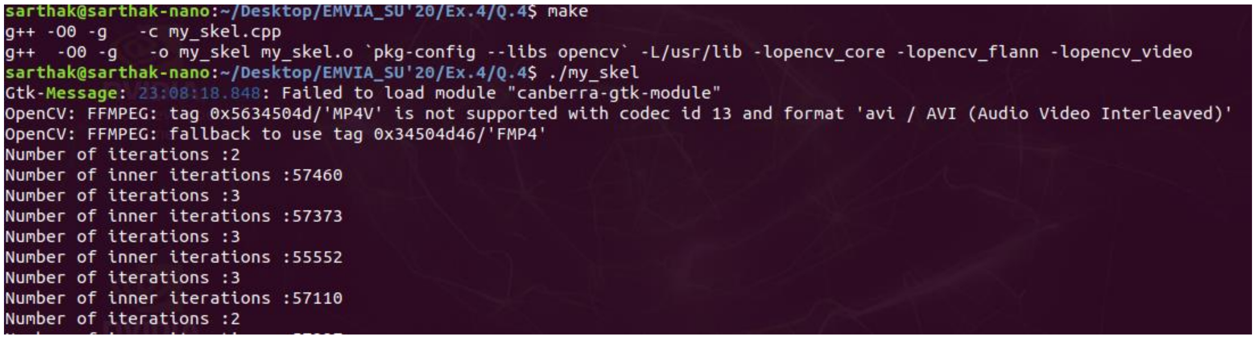
```
Iterations: 94
Iterations: 94
Iterations: 100
Iterations: 91
Iterations: 100
Iterations: 97
Iterations: 94
Iterations: 94
Iterations: 94
Iterations: 94
Iterations: 94
Iterations: 94
Iterations: 100
Iterations: 94
Iterations: 86
Duration: 227 seconds
Average FPS: 1
```

Screenshot showing output video with 1 Frame/second

Commenting on the execution efficiency of the code, an **average FPS rate of 1 is obtained**, even though the camera FPS is 30. This makes sense, since the number of iterations required over the entire frame are found to be greater than 100 at times, at which point the loop breaks because of our limiting condition. In each iteration, a set of computationally intensive functions is carried, which has the effect of greatly slowing the rate at which the program can process camera's input frames, leading to loss of frames. However, the skeletal transform obtained is quite accurate.

Question 4

The aim of this question was to design our own implementation of a skeletal transform, rather than using the OpenCV APIs like erode and dilate. This was performed courtesy of a thinning algorithm, reference for which is given to Chapter 9 of Computer and Machine Vision, by E.R. Davies. The program begins in the same manner as the previous one, by performing pre-processing operations.

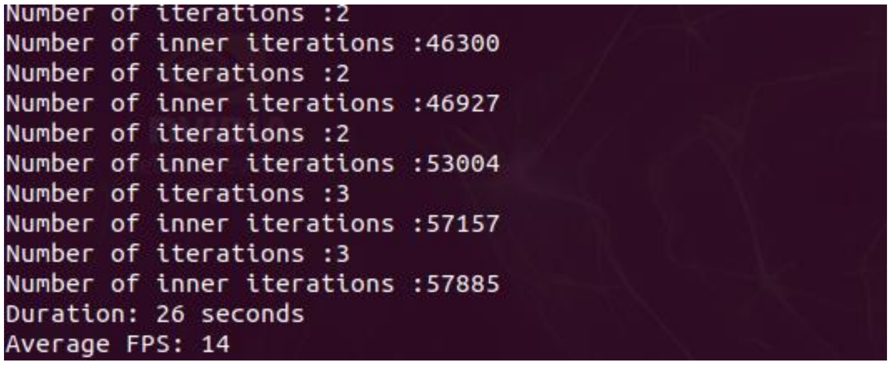


```
sarthak@sarthak-nano:~/Desktop/EMVIA_SU'20/Ex.4/Q.4$ make
g++ -O0 -g   -c my_skel.cpp
g++  -O0 -g   -o my_skel my_skel.o `pkg-config --libs opencv` -L/usr/lib -lopencv_core -lopencv_flann -lopencv_video
sarthak@sarthak-nano:~/Desktop/EMVIA_SU'20/Ex.4/Q.4$ ./my_skel
Gtk-Message: 23:08:18.848: Failed to load module "canberra-gtk-module"
OpenCV: FFMPEG: tag 0x5634504d/'MP4V' is not supported with codec id 13 and format 'avi / AVI (Audio Video Interleaved)'
OpenCV: FFMPEG: fallback to use tag 0x34504d46/'FMP4'
Number of iterations :2
Number of inner iterations :57460
Number of iterations :3
Number of inner iterations :57373
Number of iterations :3
Number of inner iterations :55552
Number of iterations :3
Number of inner iterations :57110
Number of iterations :2
```

Build and test

The main logic involves going over each pixel in the frame, and checking if its intensity if at maximum. If it is, the pixel is checked for the condition of being a redundant border pixel. If the pixel is found to be connecting two parts of the image, it must be retained. This function is determined using a number called the crossing number. To ensure that border pixels are only removed in one of four cardinal directions a time, north, south, east and west points are computed, and reduced to zero intensity only along these directions.

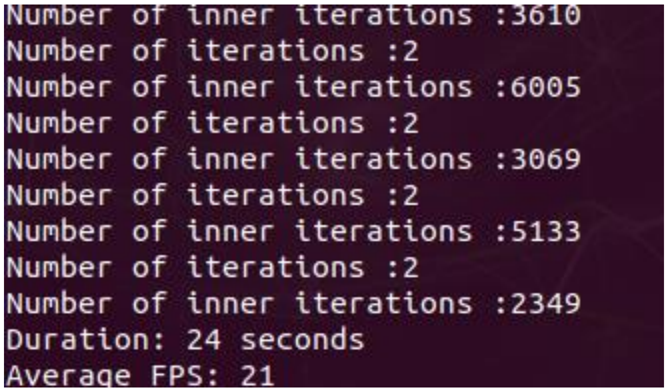
A small modification in this code as compared to the previous one brings about a large change in the efficiency of computing FPS. Instead of waiting for 100 iterations, the loop over each pixel is only iterated as many times as a white pixel exists in the modified image. Once all white pixels are removed or have been operated upon, the loop ends, when it can find no border pixel which is also white. Due to this, the number of iterations was reduced from 100 in the previous question's program, to less than 5. Since the number of iterations is less, the frame rate is also found to be more efficient, giving an **average FPS of 14**, with the camera having an FPS of 30.



```
Number of iterations :2
Number of inner iterations :46300
Number of iterations :2
Number of inner iterations :46927
Number of iterations :2
Number of inner iterations :53004
Number of iterations :3
Number of inner iterations :57157
Number of iterations :3
Number of inner iterations :57885
Duration: 26 seconds
Average FPS: 14
```

Screenshot showing output video (without background subtraction) with 14 Frame/second

Additionally, it was found that using background subtraction further improves the FPS, since now the number of iterations and number of pixels to be analyzed in each pixel reduces further. **The FPS in the case of background subtraction was found to be 21.**



```
Number of inner iterations :3610
Number of iterations :2
Number of inner iterations :6005
Number of iterations :2
Number of inner iterations :3069
Number of iterations :2
Number of inner iterations :5133
Number of iterations :2
Number of inner iterations :2349
Duration: 24 seconds
Average FPS: 21
```

Screenshot showing output video (with background subtraction) with 21 Frame/second

Here, the difference between outputs due to the two programs is quite evident. The OpenCV top-down implementation is found to be a lot better in terms of quality, as the skeleton is quite prominent, with almost no noisy lines polluting the skeletal frame. However, in terms of efficiency, the bottom-up approach is found to be better, as it is computationally much less intensive, being able to operate over all pixels in a small number of iterations. It manages to capture a high fraction of the frames produced by the camera. The bottom-up approach loses out in terms of quality, where it can be observed in the output that sometimes, a slightly distorted skeleton is produced, consisting of lines along the right-hand and bottom edges of the object. This can be fixed by removing the points as evenly as possible, by iterating over the image, finding the outermost edge pixels, removing only the redundant ones from this set, and repeating. However, this would involve using an edge detection scheme, which would greatly increase the computational load, leading to a similar result as the top-down approach.

Question 5

The paper proposes a distinctive method of recognizing objects in images. The problem arises when images are rotated, leading to problems in identifying the same object. A scale and rotation-invariant transform is required. The Scale Invariant Feature Transform (or SIFT) was proposed with the following four main steps:

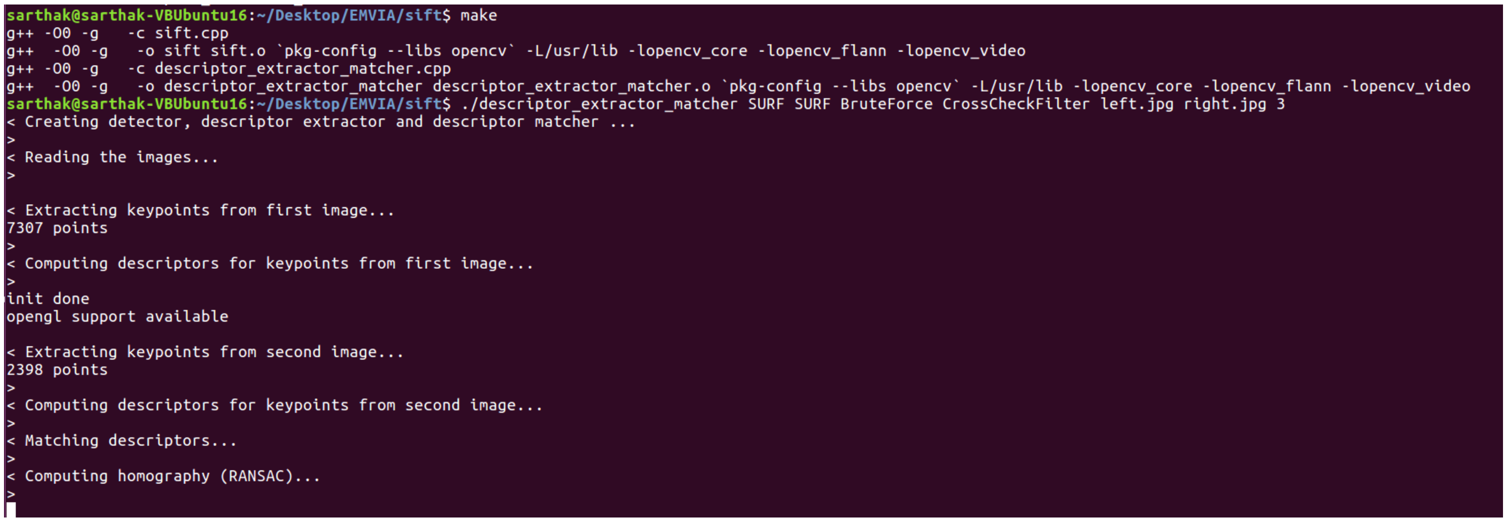
1. Scale-space extrema detection – The problem of varying scale is tackled first. Finding the LoG of images helps with extracting the feature key points, but being intensive, an approximation to LoG is computed by finding the difference between blurred octaves of images, called Difference of Gaussian, or DoG. The local extrema give the possible key points.
2. Key point Localization – The location of the various key points is pin-pointed to specific locations by using Taylor series to expand the scale space. DoG is also sensitive to edge key points, so these are removed by a technique called edge thresholding. In this step, low contrast key points and edge key points are removed.
3. Orientation Assignment – In order to achieve rotation invariance, orientations are assigned to each key point by creating an orientation histogram based on magnitude and direction of all pixels around the key point, and assigning the orientation with highest peak to the key point.
4. Feature Generation – Key point descriptors are created by taking a 16x16 neighborhood around the key point, and divided into 16 sub-blocks. For each sub-block, a histogram is created having 128 values. The bin of the histogram represents a vector to identify the key point's features.
5. The main application of this technique, is obviously, to image and object detection. Since a lot of features may be common between two images, or even objects, initial matches between key points are not the only deciding factor. Clusters of at least three features between key points are identified, and if matched, the result is used to consider matches between key points.

To summarize, the paper describes SIFT key points and extraction of their features. The key points are scale and rotation invariant. A large number of key points can be extracted even from small objects in images, leading to robustness in detection of these objects. Some methods for using these key points in object detection are also proposed, like Hough transform for verification. Other possible applications are 3D reconstruction, robot localization, object tracking and detection.

The paper's contributions to computer vision include a novel technique for object detection, no matter the orientation of the object. Possible directions for the future of this technique could be more efficient object detection and recognition using the learned features as well as extracted features from key points.

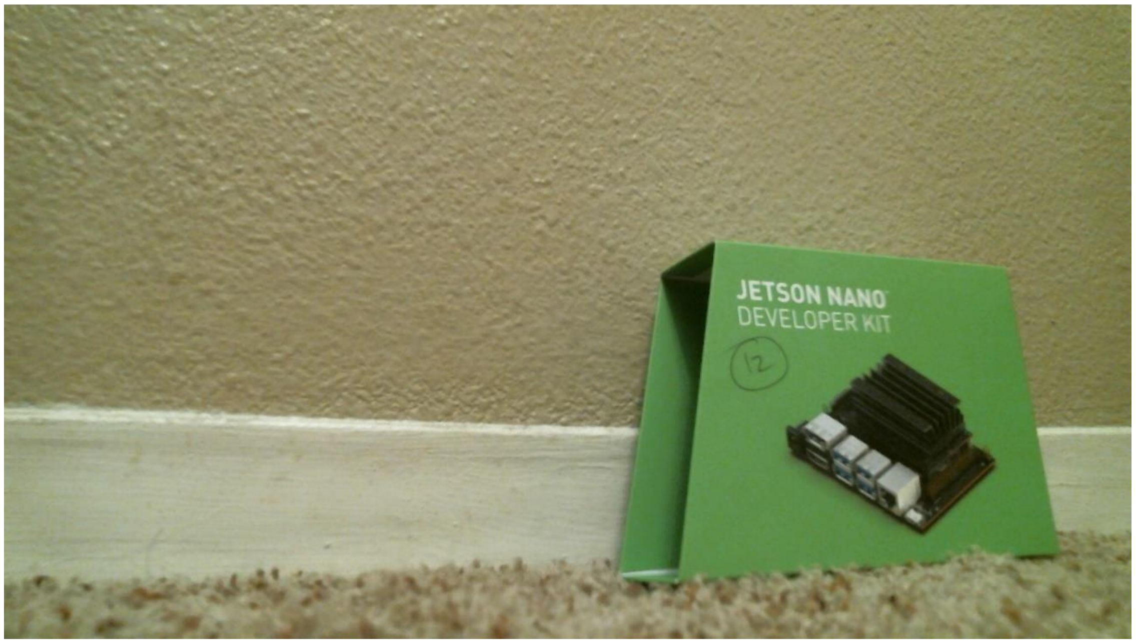
Question 6

The program demonstrates the use of the SIFT algorithm to extract key points between two different images and match the common parts or same object of two images. In this case, the two images are the left and right views of the same object against a plain background. The program proceeds similar to the actual SIFT algorithm. Features are extracted, descriptors are extracted, descriptors are matched, and lastly, the key points are extracted, and image matching done on basis of the extracted key points.



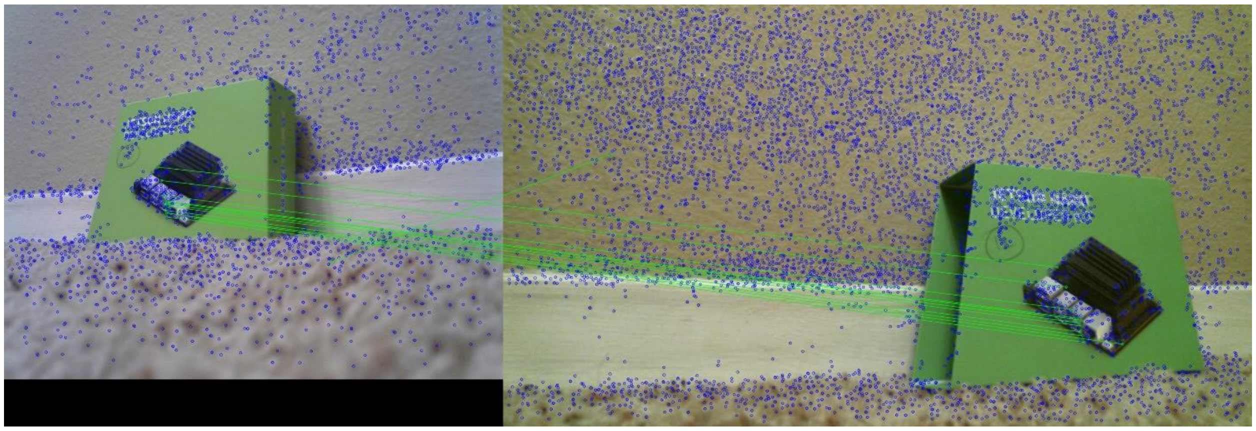
```
sarthak@sarthak-VBUbuntu16:~/Desktop/EMVIA/sift$ make
g++ -O0 -g   -c sift.cpp
g++  -O0 -g   -o sift sift.o `pkg-config --libs opencv` -L/usr/lib -lopencv_core -lopencv_flann -lopencv_video
g++ -O0 -g   -c descriptor_extractor_matcher.cpp
g++  -O0 -g   -o descriptor_extractor_matcher descriptor_extractor_matcher.o `pkg-config --libs opencv` -L/usr/lib -lopencv_core -lopencv_flann -lopencv_video
sarthak@sarthak-VBUbuntu16:~/Desktop/EMVIA/sift$ ./descriptor_extractor_matcher SURF SURF BruteForce CrossCheckFilter left.jpg right.jpg 3
< Creating detector, descriptor extractor and descriptor matcher ...
>
< Reading the images...
>

< Extracting keypoints from first image...
7307 points
>
< Computing descriptors for keypoints from first image...
>
init done
opengl support available

< Extracting keypoints from second image...
2398 points
>
< Computing descriptors for keypoints from second image...
>
< Matching descriptors...
>
< Computing homography (RANSAC)...
>
```

Build and run of SIFT code



Left-side image of object

Right-side image of object



Comparison image of two images

The right-side view image obtained is slightly smaller since the camera resolution is 720p as compared to 1080p for the left-side view image.

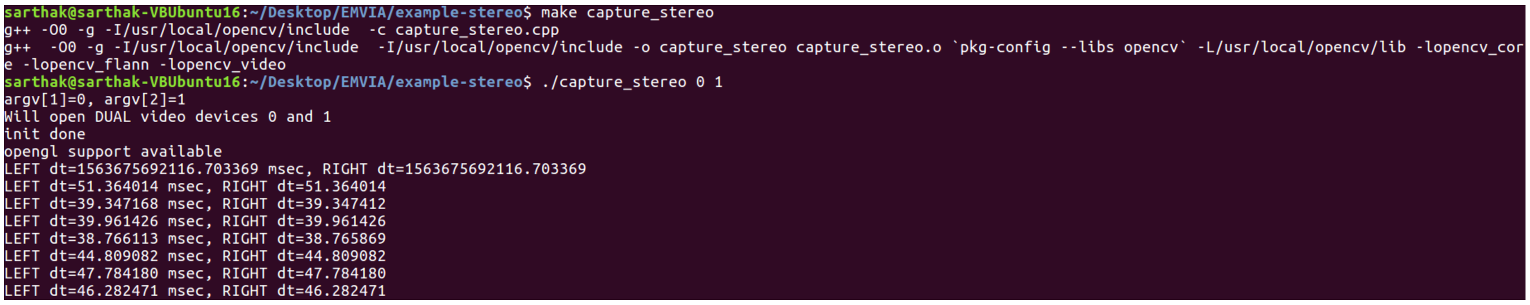
Question 7

The program attempts to capture the disparity between two input images provided. It does so by a technique called disparity estimation, in which for each pixel in the first image, a corresponding pixel in the other image is sought. Of course, the pixels may not lie in the same rows in both images, which is solved by using a multi-grid approach. The program uses a single-fixed sampling grid along with several discrete quantization levels to cause convergence of points lying on same lines.

In terms of the code, the main logical part of the program is the StereoVar constructor used to compute the disparity between two Mat type objects using the variational algorithm described above.

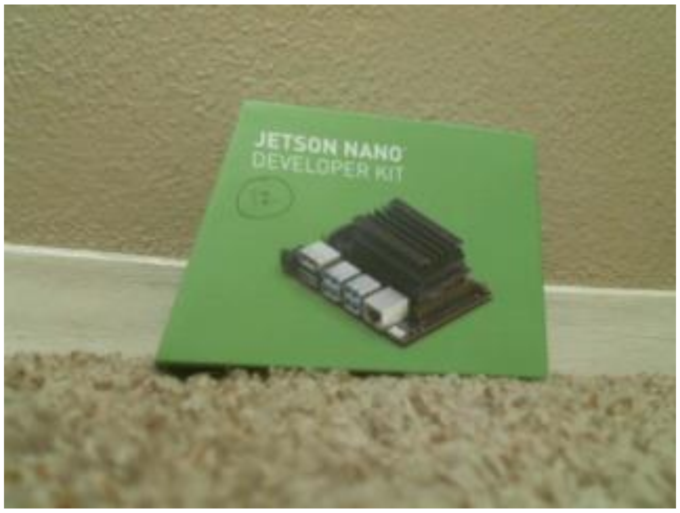
The performance and accuracy parameters depend completely on the elements of the StereoVar class. Some elements like 'levels', 'pyrScale', and 'min/maxDisp' determine the number of blurring layers to be used, the scale of the layers, and the limits of the disparity values. Change in these values will lead to change in the accuracy of the results as well. In short, all these parameters can be used to further optimize the output by determining the level of processing to be done, the level of blurring to be performed, the scale to which the blurring is performed, and so on.

As mentioned above, key points are not used in this particular method of disparity analysis. Of course, using key points would increase the quality of analysis, giving even better results. However, it seems unnecessary, as anyway this method would only be applicable to objects not rotated too much. It is a rotation variant method. Using key points would change this, but if this algorithm is anyway applied to images which are not rotated, there seems to be no need of using key points extraction.

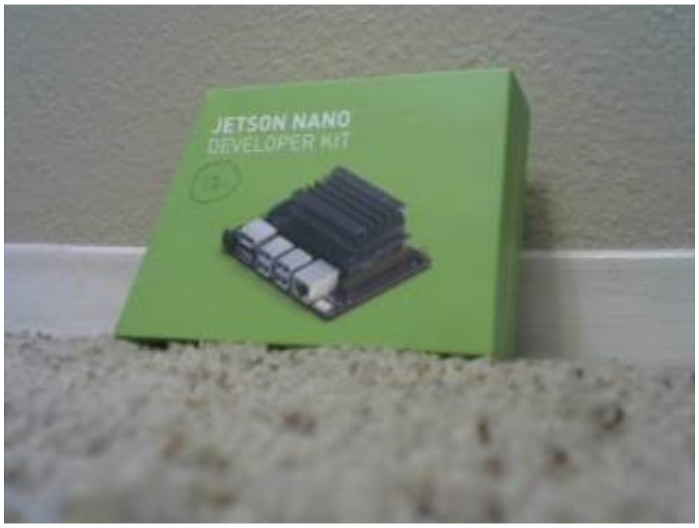


```
sarthak@sarthak-VBUbuntu16:~/Desktop/EMVIA/example-stereo$ make capture_stereo
g++ -O0 -g -I/usr/local/opencv/include  -c capture_stereo.cpp
g++  -O0 -g -I/usr/local/opencv/include  -I/usr/local/opencv/include -o capture_stereo capture_stereo.o `pkg-config --libs opencv` -L/usr/local/opencv/lib -lopencv_cor
e -lopencv_flann -lopencv_video
sarthak@sarthak-VBUbuntu16:~/Desktop/EMVIA/example-stereo$ ./capture_stereo 0 1
argv[1]=0, argv[2]=1
Will open DUAL video devices 0 and 1
init done
opengl support available
LEFT dt=1563675692116.703369 msec, RIGHT dt=1563675692116.703369
LEFT dt=51.364014 msec, RIGHT dt=51.364014
LEFT dt=39.347168 msec, RIGHT dt=39.347412
LEFT dt=39.961426 msec, RIGHT dt=39.961426
LEFT dt=38.766113 msec, RIGHT dt=38.765869
LEFT dt=44.809082 msec, RIGHT dt=44.809082
LEFT dt=47.784180 msec, RIGHT dt=47.784180
LEFT dt=46.282471 msec, RIGHT dt=46.282471
```

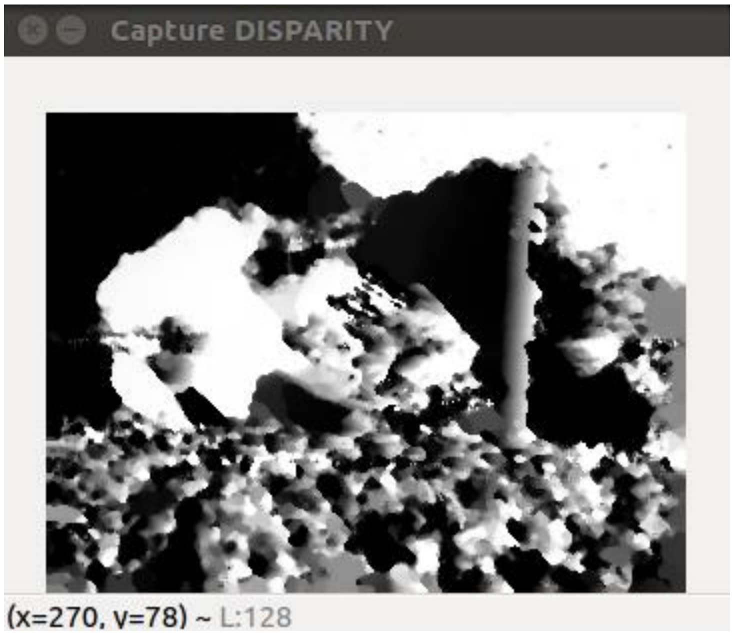
Build and run of stereo capture



Left-side image of object



Right-side image of object



Disparity correspondence analysis of left- and right-side images. The right side of the green box can be seen.

References:

1. Probabilistic Hough Transforms - https://pdfs.semanticscholar.org/58ce/e69ed2033f559f5ba579b48ac4359bcf524c.pdf
2. SIFT: Theory and Practice - http://aishack.in/tutorials/sift-scale-invariant-feature-transform-features/
3. Introduction to Scale Invariant Feature Transform - https://docs.opencv.org/3.4/da/df5/tutorial_py_sift_intro.html
4. Accurate Real-time disparity estimation with variational methods - https://www.informatik.uni-marburg.de/~thormae/paper/ISVC2009stereo.pdf
5. Stereo Correspondence - https://docs.opencv.org/2.4.13.2/modules/contrib/doc/stereo.html
6. Source code of class StereoVar OpenCV - https://jar-download.com/artifacts/nu.pattern/opencv/2.4.9-7/source-code/org/opencv/contrib/StereoVar.java
7. Depth Map from Stereo Images - https://docs.opencv.org/3.1.0/dd/d53/tutorial_py_depthmap.html