University of Colorado
Boulder

EXERCISE 4

BY

SIDDHANT JAJOO

UNDER THE GUIDANCE OF:

PROFESSOR SAM SIEWERT,
UNIVERSITY OF COLORADO BOULDER

7/21/2019

BOARD USED: NVIDIA JETSON NANO AND UBUNTU 16.04 VIRTUAL MACHINE.

TABLE OF CONTENTS

QUESTION 1 2
SUMMARY 2
QUESTION 2 3
SCREENSHOTS 3
CODE DESCRIPTION 4
QUESTION 3 5
SCREENSHOTS 5
CODE DESCRIPTION 5
QUESTION 4 7
SCREENSHOTS 7
CODE DESCRIPTION 9
ANALYSIS 9
QUESTION 5 10
SUMMARY 10
QUESTION 6 11
SCREENSHOTS 11
CODE DESCRIPTIONS 12
QUESTION 7 13
SCREENSHOTS 13
CODE DESCRIPTIONS 14
REFERENCES 14

QUESTION 1

SUMMARY

The problem solved in the paper is to detect collinear or almost collinear points forming a straight line in a digitized image. This problem can be solved by testing the lines formed by all the pairs of points. The computation requirement for n points is n2 and is quite large. In order to reduce these computations, hough proposed a better method in order to find concurrent lines. He replaced the slope-intercept method with another method since the slope and the intercept both are unbounded. In addition to this, the slope intercept method gives us an infinite value for vertical lines. He stated that a point can also be represented by an angle and theta of its normal and its algebraic distance from the origin. This method can be used to show that the curves corresponding to collinear figure points have a common point of intersection. The (theta, p) representation of points transforms the problem of finding collinear points into finding concurrent lines. A point in the picture plane corresponds to a sine curve in the parameter plane. These sine curves for different points have a common point of intersection. This signifies that the collinear figure points have different sine curves which intersect at a common point.

In order to reduce computations for finding collinear figure points, we quantize and divide the region of interest into equal parts called cells. The quantized region is treated as a two dimensional array of accumulator. The array holds the number of curves passing through the (r,theta) point i.e the curve passing through the cell. A threshold is assumed for the number of curves passing through the cell in question. If the number of curves in the accumulator is greater than the threshold, it can be said that a line is found in the image and a line is drawn through that point represented by x,y in that specified region. For example, we divide the image into n cells consisting of d1 different values of theta and d2 different values of p. So the total computation to find the collinear points is reduced due to the quantization. This introduces error in finding collinear points but since we do not want the exact lines, we can take advantage of this approach. Now, we just need to check the d1d2 cells in order to find that if the points in that region are collinear.

Some of the limitations for this method are:

1. The final results are affected by choosing the quantization values i.e the number of cells. This is the value of d1 and d2. Smaller the value of d1 and d2, faster is the processing.
2. This method detects a line even if here is discontinuity in the line. Thus if there are unrelated figure points in the image, it will regard it as a line.
3. Threshold selection is of utmost importance in this method since a false line can get detected if the threshold is not set accordingly.

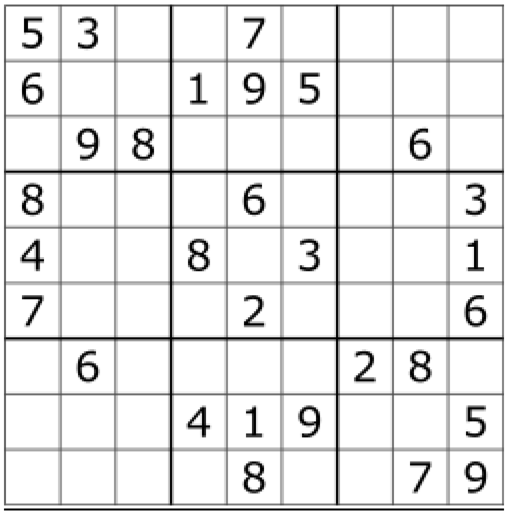
This method can be extended for different configurations of figure points example: Circles. This can be done by choosing a correct parameterization for the family of curves.

This method has contributed to the most important part in computer vision. It has bought down the computations to a great extent. Thus in time constraint scenarios and where accuracy is of not that important, Hough line transform can be used.

QUESTION 2

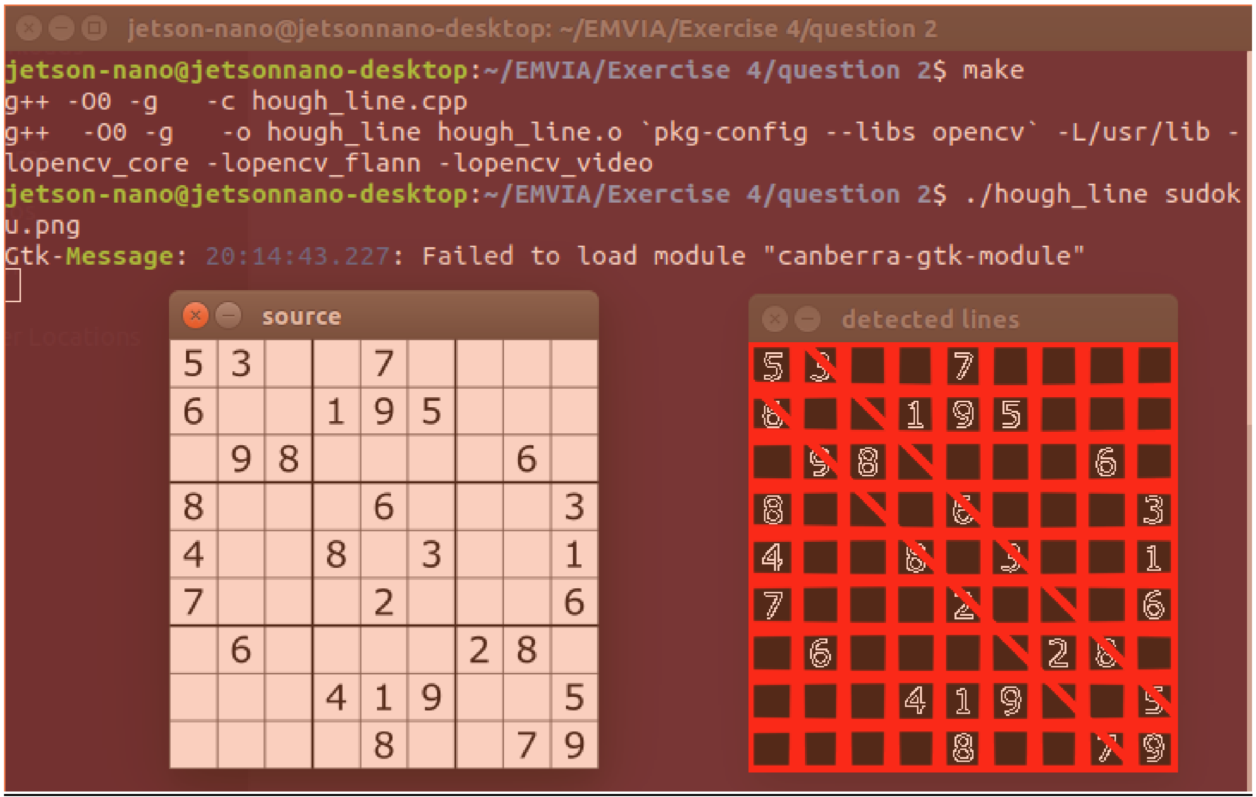
SCREENSHOTS

ORIGINAL IMAGE:



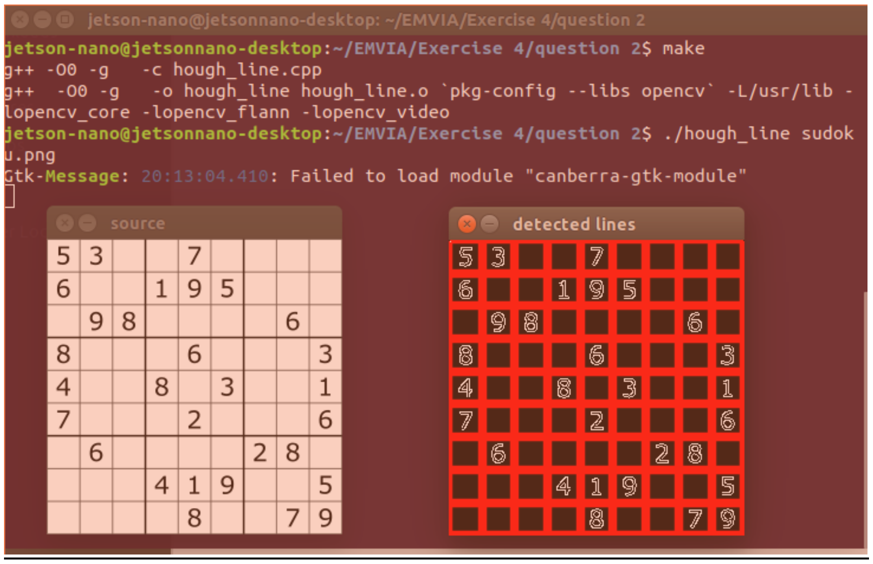
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

STANDARD HOUGH LINEAR IMAGE:



```
jetson-nano@jetsonnano-desktop: ~/EMVIA/Exercise 4/question 2
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/question 2$ make
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
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/question 2$ ./hough_line sudok
u.png
Gtk-Message: 20:14:43.227: Failed to load module "canberra-gtk-module"
```

PROBABILISTIC HOUGH LINEAR IMAGE:


jetson-nano@jetsonnano-desktop: ~/EMVIA/Exercise 4/question 2
source
detected lines

```
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/question 2$ make
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
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/question 2$ ./hough_line sudok
u.png
Gtk-Message: 20:13:04.410: Failed to load module "canberra-gtk-module"
```

CODE DESCRIPTION

This code implements and demonstrates line detection with the help of Hough Lines Transform. In this code, the image is being read in grayscale mode and is followd by Canny function in order to detect edges in the image. The image obtained is then converted to color in order to mark the lines obtained in Houghlines transform. The code has two approaches separated by an if loop in order to achieve hough Line Transform. They are namely: The Standard Hough Transform and The Probabilistic Hough Line Transform. In Probabilistic Hough Lines Transform method, the start and end coordinates of the lines are stored in a vector<Vec4i> variable. This variable is fed as a parameter to HoughLinesP function. This function applies the Hough transform and stores all the start and end points in the vector<vec4i> variable. The lines are drawn on the image using these points.

In case of standard Hough lines Transform, the vector<2f> variable stores the result of the detection. This form of detection gives results in the form of (theta,r). Using the values in vector<vec2f>, one can calculate the rcos(theta) and rsin(theta) values. Finally x,y pairs are calculated using cvRound and lines are drawn between the coordinate pairs.

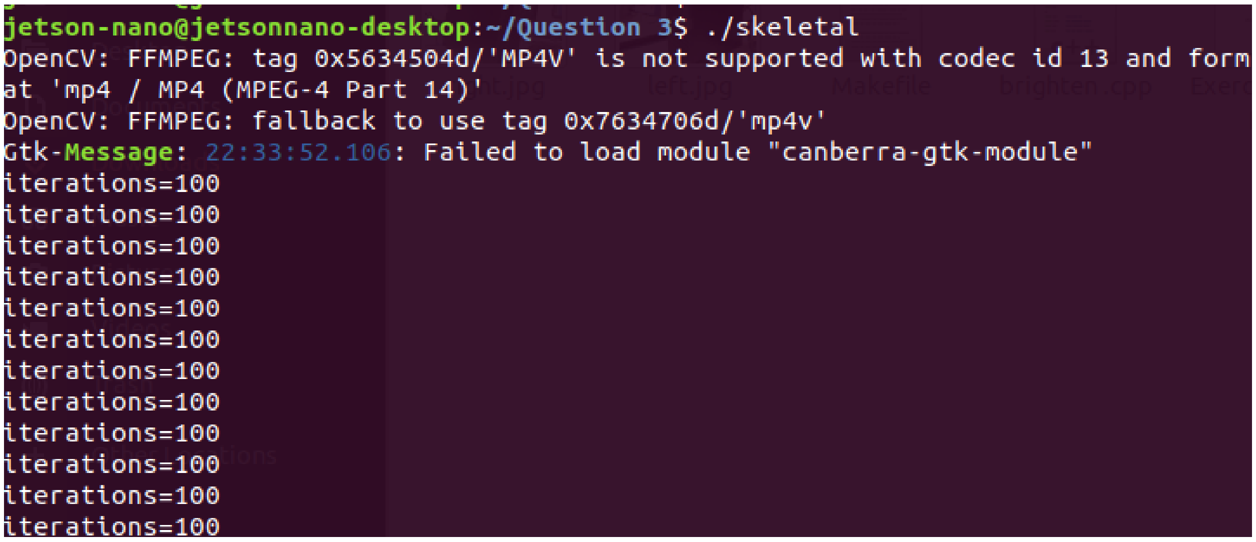
The Probabilistic is much more efficient than the Standard approach because it draws lines between the start and end points on the line as opposed to the Standard Method. The Probabilistic Hough Transform returns line segments rather than the whole line. In case of standard hough line transform, it draws a line if the count in the accumulator array is greater than the threshold for(r,theta) irrespective of any discontinuities. It can be seen from the screenshots that the Probabilistic Hough Transform has yielded better results than Standard Hough Transform.

APPLICATION OF THE INTEREST IMAGE

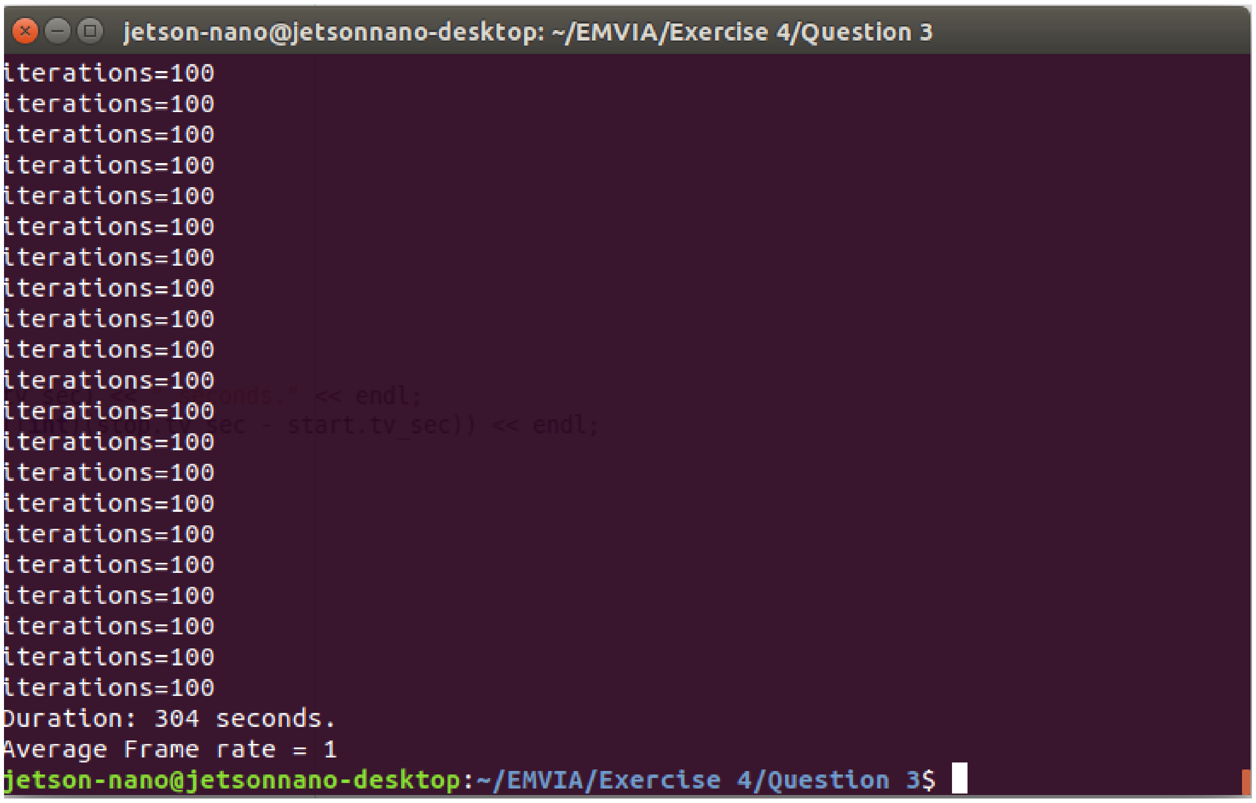
The above image was chosen because it had obvious lines to be detected which would simplify the analysis of Hough Lines Transform.

QUESTION 3

SCREENSHOTS



```
jetson-nano@jetsonnano-desktop:~/Question 3$ ./skeletal
OpenCV: FFMPEG: tag 0x5634504d/'MP4V' is not supported with codec id 13 and form
at 'mp4 / MP4 (MPEG-4 Part 14)'
OpenCV: FFMPEG: fallback to use tag 0x7634706d/'mp4v'
Gtk-Message: 22:33:52.106: Failed to load module "canberra-gtk-module"
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
```



```
jetson-nano@jetsonnano-desktop: ~/EMVIA/Exercise 4/Question 3
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
Duration: 304 seconds.
Average Frame rate = 1
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/Question 3$
```

CODE DESCRIPTION

This code uses the frames obtained from the camera and obtains the skeletal MPEG-4 video at the end. The frame obtained from the camera is converted into grayscale in order to eliminate the colour

channels for further processing. The obtained grayscale frame is converted to binary image so that there can be demarcation of a foreground and background object. This is achieved using the threshold function. The threshold can be decided by using histogram analysis or using GIMP. Threshold depends on the background lighting as well so depending on that a suitable value is chosen. Median filter is then applied in order to eliminate noise and smoothen the image. Now an algorithm is applied to the median fitered image in order to obtain the skeletal form of the Image.
The algorithm is as follows: (These steps are performed in a do while loop until the nonzero pixel count is not zero or number of iterations is less than 100)

1. The first step would be to erode the unnecessary border and thin the image.
2. The next step is to dilate the image obtained on erosion. These steps would eliminate any noise in the image and focus on the foreground object.
3. Subtract the dilated image from the image before erosion.
4. Bitwise_or the image obtained in 3rd step with an empty image.
5. Copy the eroded image obtained in the first step as the first argument to the erode function.
6. Increment the iteration variable.

The above erode and dilate operations are called as Morphological transformations. These operations are performed on binary images. These work by providing a kernel or a structuring element. There can be different sizes to this kernel and that depends upon how much of the foreground object needs to be eroded and the shape of the foreground object.

The basic idea of erosion is to erode the boundaries of foreground object. A pixel in the original image (either 1 or 0) is considered 1 if all the pixels under the kernel are 1 or else it is eroded thus causing erosion. It is useful for removing small white noises. Dilation is just the opposite of erosion. It increases the white region and is also useful in joining broken parts of an object. Dilation is followed by erosion because it removes the noise and shrinks the object. Since the noise is eliminated it cannot come back, dilation is followed by erosion. The image obtained after dilation is subtracted with the original one. The algorithm is implemented till we get a skeleton of the object in place.

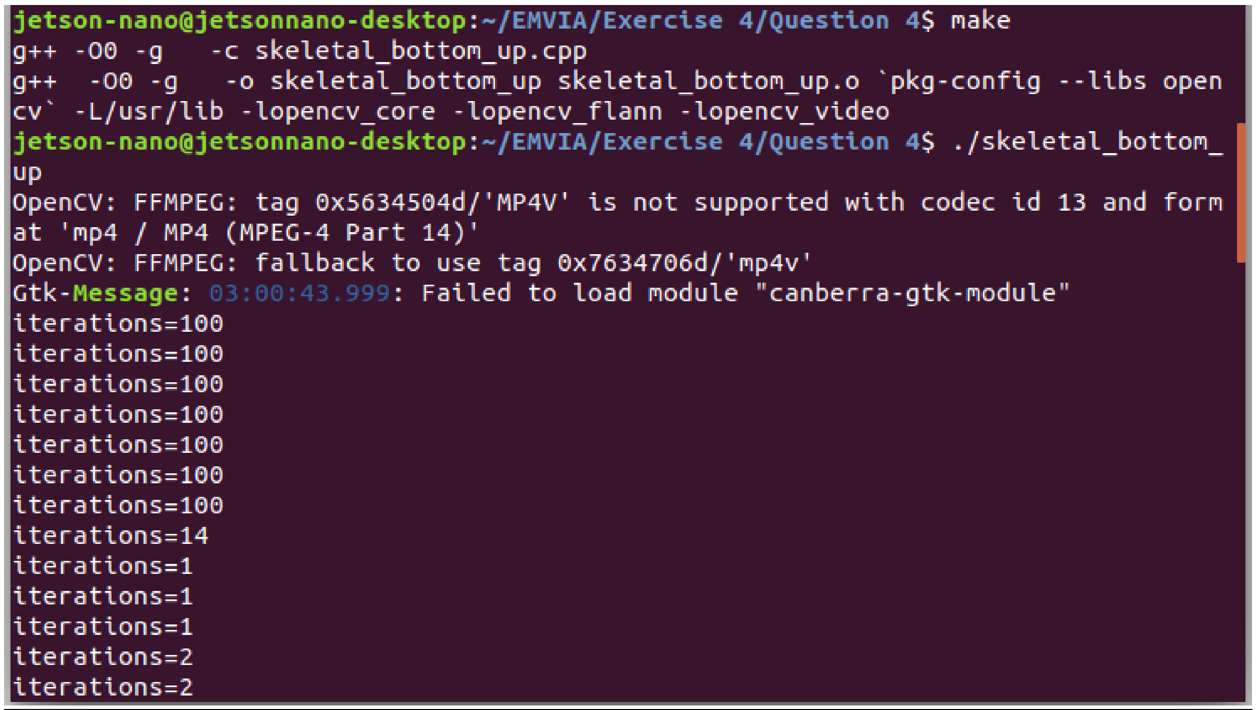
Finally the frames are written to an output file using Videowriter. The FPS obtained here is 1. This is due to recurring computations to thin images. In this code, the maximum times the frame undergoes computation is 100. By reducing this number, the FPS can be increased and this was observed by practically changing the value from 100 to 50. But by changing this number, there can be instances where thinning is not upto the mark and requires computations more than 50 times. The other side to this is that a lot of cycles are wasted by computing 100 times as there can be instances when thinning is completed way earlier. Thus there is a tradeoff between FPS and accurate results.

The video and example frames of the continuous skeletal transformation can be found in the Zip folder.

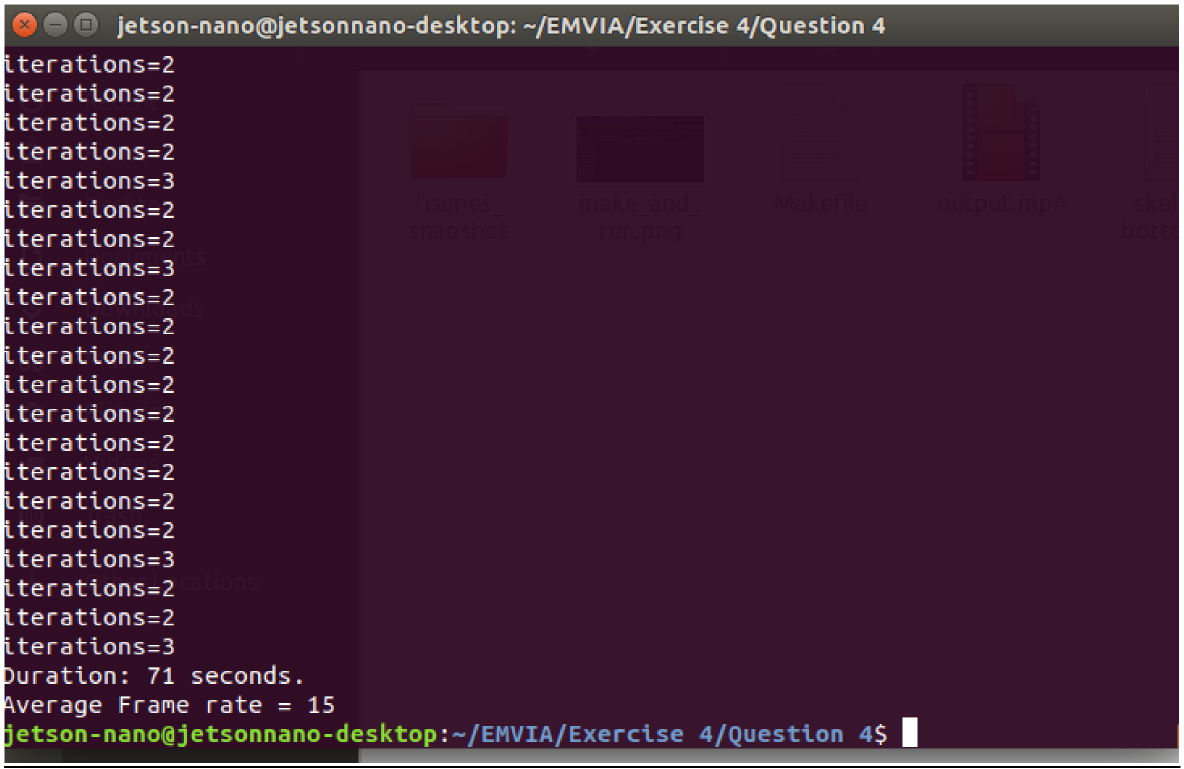
QUESTION 4

SCREENSHOTS

Without frame differencing: Make and run and frame rate images



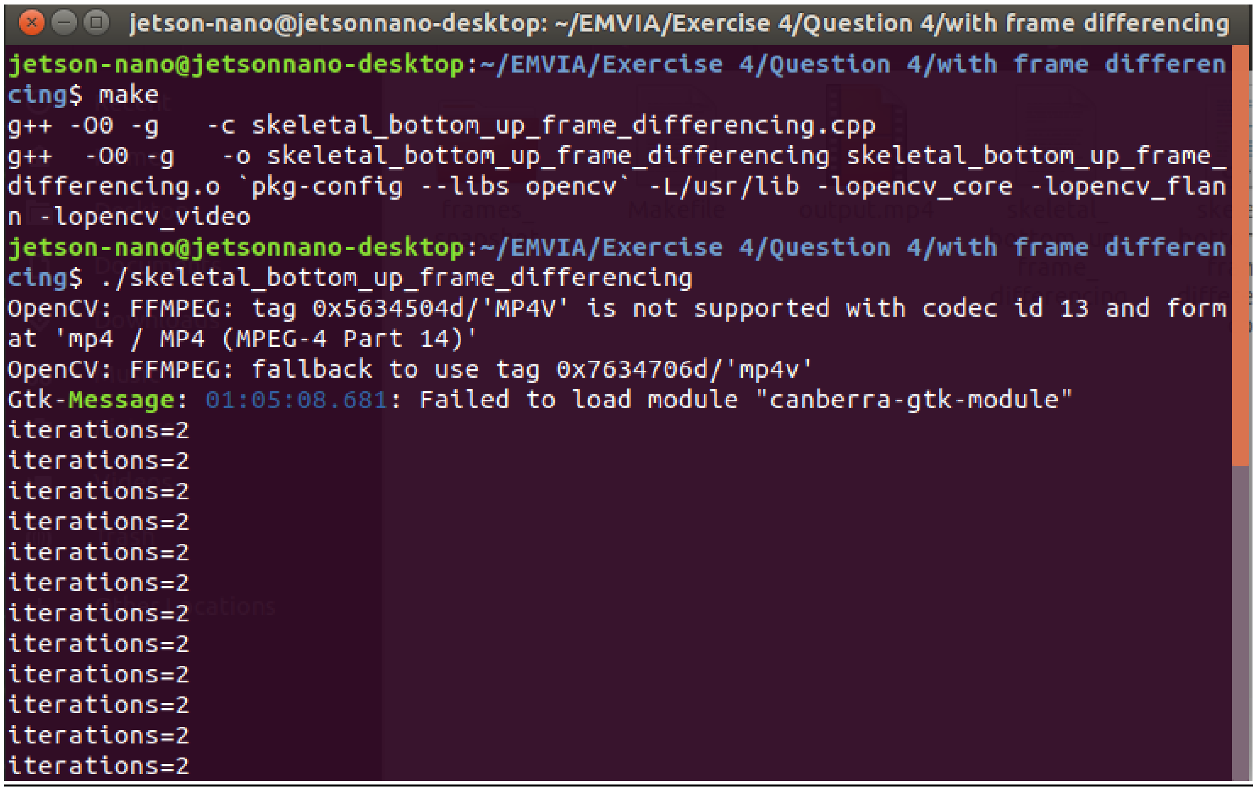
```
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/Question 4$ make
g++ -O0 -g   -c skeletal_bottom_up.cpp
g++  -O0 -g   -o skeletal_bottom_up skeletal_bottom_up.o `pkg-config --libs open
cv` -L/usr/lib -lopencv_core -lopencv_flann -lopencv_video
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/Question 4$ ./skeletal_bottom_
up
OpenCV: FFMPEG: tag 0x5634504d/'MP4V' is not supported with codec id 13 and form
at 'mp4 / MP4 (MPEG-4 Part 14)'
OpenCV: FFMPEG: fallback to use tag 0x7634706d/'mp4v'
Gtk-Message: 03:00:43.999: Failed to load module "canberra-gtk-module"
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=100
iterations=14
iterations=1
iterations=1
iterations=1
iterations=2
iterations=2
```



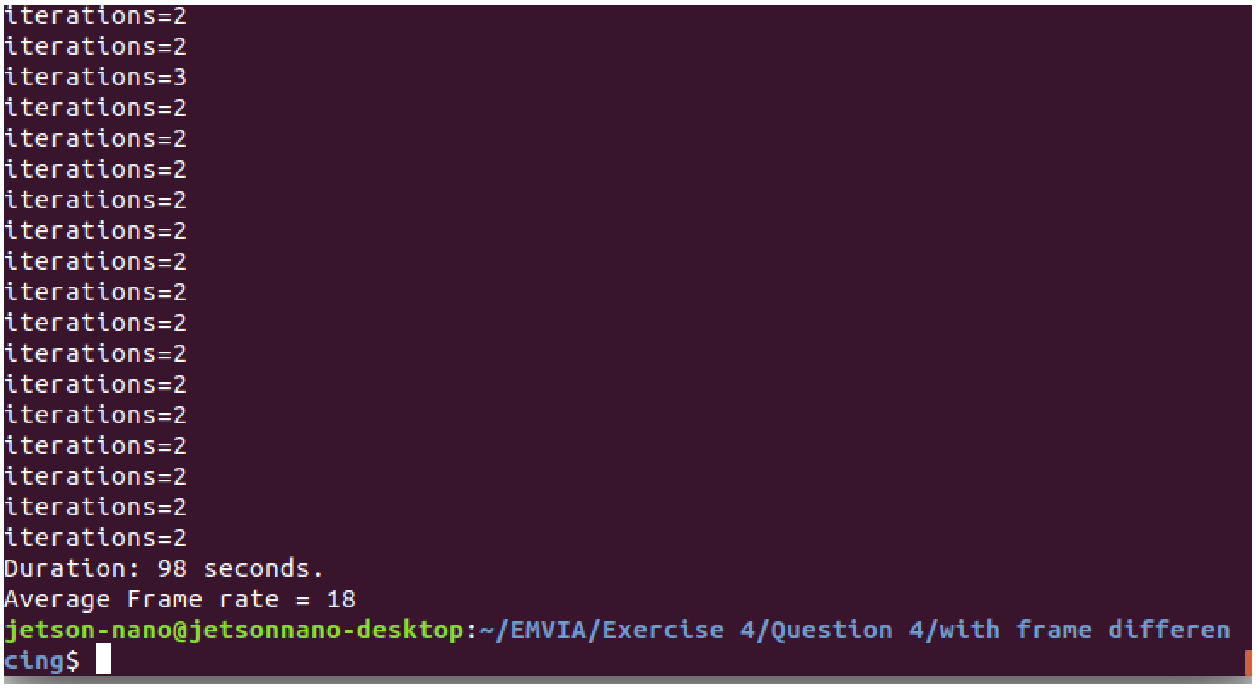
jetson-nano@jetsonnano-desktop: ~/EMVIA/Exercise 4/Question 4

```
iterations=2
iterations=2
iterations=2
iterations=2
iterations=3
iterations=2
iterations=2
iterations=3
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=3
iterations=2
iterations=2
iterations=3
Duration: 71 seconds.
Average Frame rate = 15
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/Question 4$
```

With frame differencing: Make and run and frame rate images



```
jetson-nano@jetsonnano-desktop: ~/EMVIA/Exercise 4/Question 4/with frame differencing
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/Question 4/with frame differen
cing$ make
g++ -O0 -g   -c skeletal_bottom_up_frame_differencing.cpp
g++  -O0 -g   -o skeletal_bottom_up_frame_differencing skeletal_bottom_up_frame_
differencing.o `pkg-config --libs opencv` -L/usr/lib -lopencv_core -lopencv_flan
n -lopencv_video
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/Question 4/with frame differen
cing$ ./skeletal_bottom_up_frame_differencing
OpenCV: FFMPEG: tag 0x5634504d/'MP4V' is not supported with codec id 13 and form
at 'mp4 / MP4 (MPEG-4 Part 14)'
OpenCV: FFMPEG: fallback to use tag 0x7634706d/'mp4v'
Gtk-Message: 01:05:08.681: Failed to load module "canberra-gtk-module"
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
```



```
iterations=2
iterations=2
iterations=3
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
iterations=2
Duration: 98 seconds.
Average Frame rate = 18
jetson-nano@jetsonnano-desktop:~/EMVIA/Exercise 4/Question 4/with frame differen
cing$
```

CODE DESCRIPTION

The screenshots above are for two types of codes, the one that includes frame differencing before skeleton thinning and the one without skeletal thinning.

For the code without frame differencing, the image is read from the camera. The procedure is the same as the 3rd question upto the do while loop. The do while in this question changes completely. The do while loop continues until a false condition is detected inside the loop or the number of iteration exceeds 100. The logic is in such a way that two "for" loops traverse all the pixels in the image. Now if the pixel in consideration has a binary value of 1, then further processing is done or else it moves to the next pixel. The values of sigma and chi are calculated and according to the conditions, the pixel values are changed or unchanged. Finally the frames are recorded and displayed.

In case of frame differencing, the only change is that the frame differencing code is placed at the beginning and the entire without frame differencing code is applied on the image obtained after frame differencing. The only changes in the code are: Threshold is changed to 10 because the differencing brings down the pixel values and the line "binary = 255 - binary" is commented.

ANALYSIS

The FPS obtained for without frame differencing and with frame differencing was 15 and 18 respectively as shown in the screenshot. This change in FPS is due to the fact that frame differencing eliminates most of the pixel data values i.e makes them zero. Thus much of the pixel data does not move into the first if loop thus saving a lot of computations and less data to be worked on. This increases the FPS of the approach with frame differencing. In addition to this the common noise is also eliminated due to frame differencing thus giving us a crisp and clear skeletal transformation.

Comparison between different approaches:

1. Top down: FPS =1
2. Without frame differencing (bottom up): FPS = 15
3. With frame differencing (bottom up): FPS = 18

The quality of 3rd approach turned out to be best with fast processing, highest FPS and good quality output. In terms of quality the first approach which involved opencv method was the best because it had opencv functions which I think are considered to be exhaustive and accurate. In terms of efficiency the 3rd and 2nd approach are almost similar with 3rd one having the highest FPS. The bottom up approach was efficient because it was coded from the bottom and designing the whole algorithm by myself according to the needs.

All in all, top down approach though being slow has the best output and the 3rd code with frame differencing code has the best efficiency with acceptable output.

The video and example frames of the continuous skeletal transformation can be found in the Zip folder.

QUESTION 5

SUMMARY

The paper emphasizes on matching different objects or scenes in images. The features are invariant to scaling and rotation and partially invariant to change in illumination and 3D camera viewpoint. These features are matched by using a cascade filtering approach. The cascading approach is such that further operations are only applied if it passes the initial test. There are four major stages in this process called Sgift Invariant Feature Transform:

1. Scale –Space extrema detection.
2. Keypoint Localization.
3. Orientation assignment.
4. Keypoint descriptor.

SIFT generates a large number of features which are stored in databases. These features are compared with the image in question based on the Euclidean distance from its feature vectors. SIFT extracts keypoints and computes its descriptors. The first step is to construct a scale – space and compute laplace of Gaussian in order to compute key points. The scale space is created by using Gaussian Blur. The original image is gradually applied Gaussian blur to create five blurred images. This first set of blurred image is called as the first octave. For the second octave the image is resized to half of the original size and the Gaussian blur is applied to all the resized images and so on till the fourth octave. The amount of blurring in each image depends on the factor k.

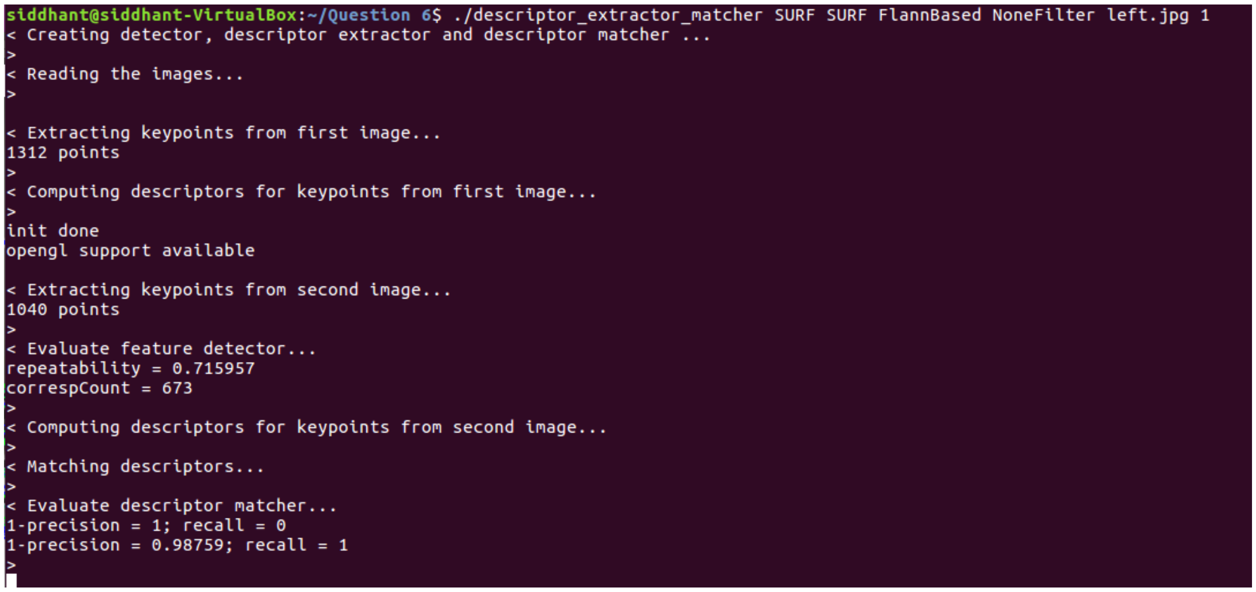
Now in order to obtain the keypoints from the image, Difference of Gaussian(DoG) is used. DoG is equivalent to laplacian of Gaussian. Laplacian of Gaussian images can be computed quickly using the scale space. LoG is calculated by taking the difference between two consecutive scales. These new images created are scale invariant. This subtraction of images is done for all the octaves.

The next step is to find the keypoints which can be found by locating the maxima and minima in the DoG images and finding the subpixel minima and maxima. The maxima/minima is located by scanning the pixels in the neighbourhood of the current, above and the below images. As a matter of fact the obtained points in the above process are approximate maxima and minima. The maxima/minima always lie somewhere between the pixel which is calculated using the taylor series with the help of the available approximate maxima/minima values. The next step is to reduce the number of keypoints. The keypoints are rejected if they have a low contrast or if located on an edge using thresholding. Reduction in keypoints helps improve the efficiency of the algorithm.

Now in order to match features irrespective of their orientation, keypoints are assigned orientations. This is achieved with the help of histograms and with the help of gradient orientations and magnitudes. Finally we create a fingerprint for the feature. A 16x16 neighbourhood around the keypoint is taken and is divided into 16 blocks of 4x4 size. 8 bin orientation histogram is created for each sub-block thus totaling to a total of 128 bins.

QUESTION 6

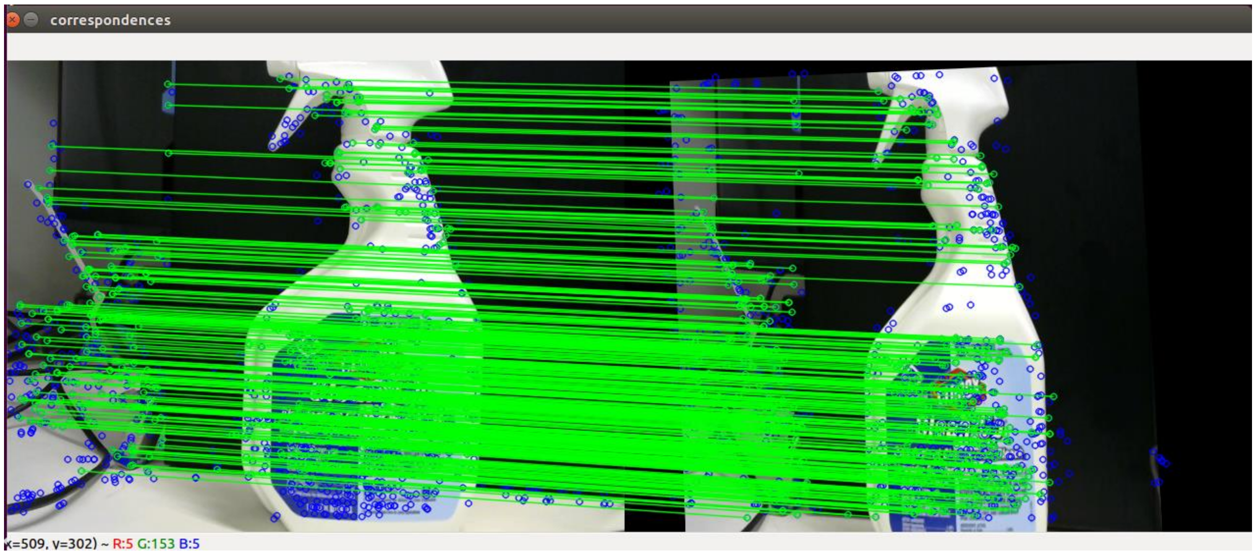
SCREENSHOTS

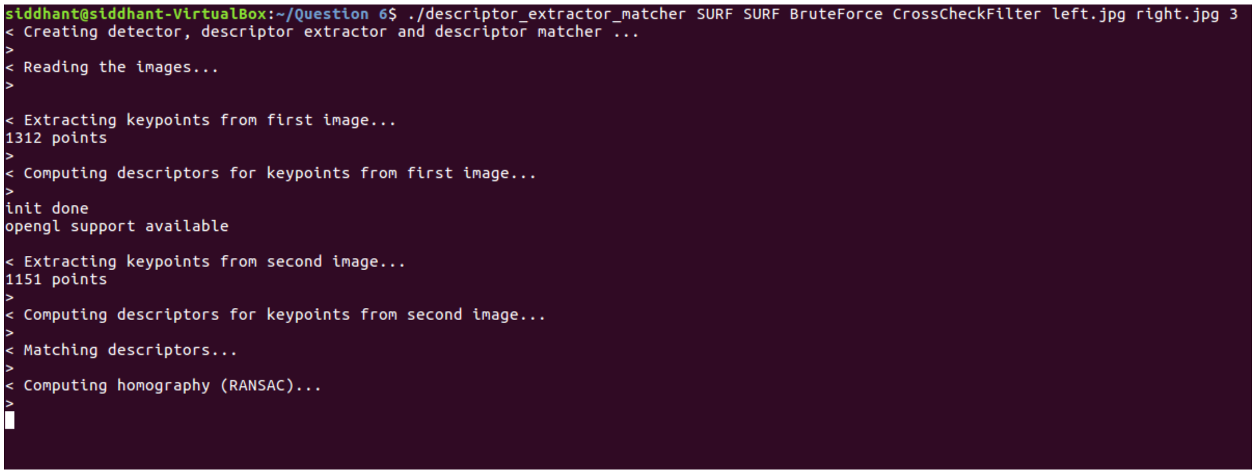


```
siddhant@siddhant-VirtualBox:~/Question 6$ ./descriptor_extractor_matcher SURF SURF FlannBased NoneFilter left.jpg 1
< Creating detector, descriptor extractor and descriptor matcher ...
>
< Reading the images...
>

< Extracting keypoints from first image...
1312 points
>
< Computing descriptors for keypoints from first image...
>
init done
opengl support available

< Extracting keypoints from second image...
1040 points
>
< Evaluate feature detector...
repeatability = 0.715957
correspCount = 673
>
< Computing descriptors for keypoints from second image...
>
< Matching descriptors...
>
< Evaluate descriptor matcher...
1-precision = 1; recall = 0
1-precision = 0.98759; recall = 1
>
```

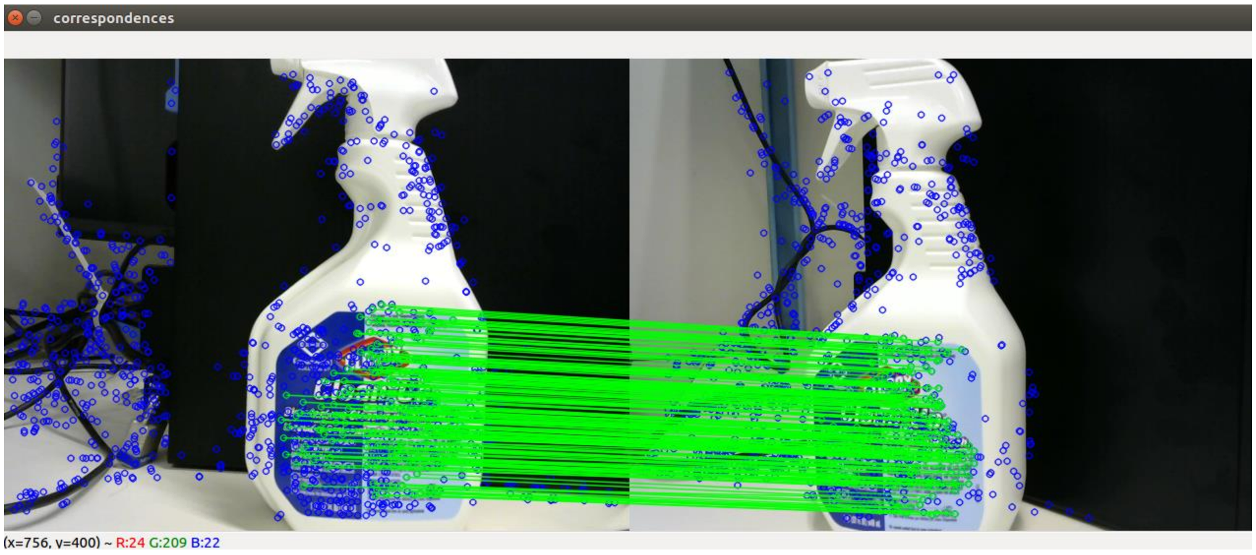




```
siddhant@siddhant-VirtualBox:~/Question 6$ ./descriptor_extractor_matcher SURF SURF BruteForce CrossCheckFilter left.jpg right.jpg 3
< Creating detector, descriptor extractor and descriptor matcher ...
>
< Reading the images...
>

< Extracting keypoints from first image...
1312 points
>
< Computing descriptors for keypoints from first image...
>
init done
opengl support available

< Extracting keypoints from second image...
1151 points
>
< Computing descriptors for keypoints from second image...
>
< Matching descriptors...
>
< Computing homography (RANSAC)...
>
```



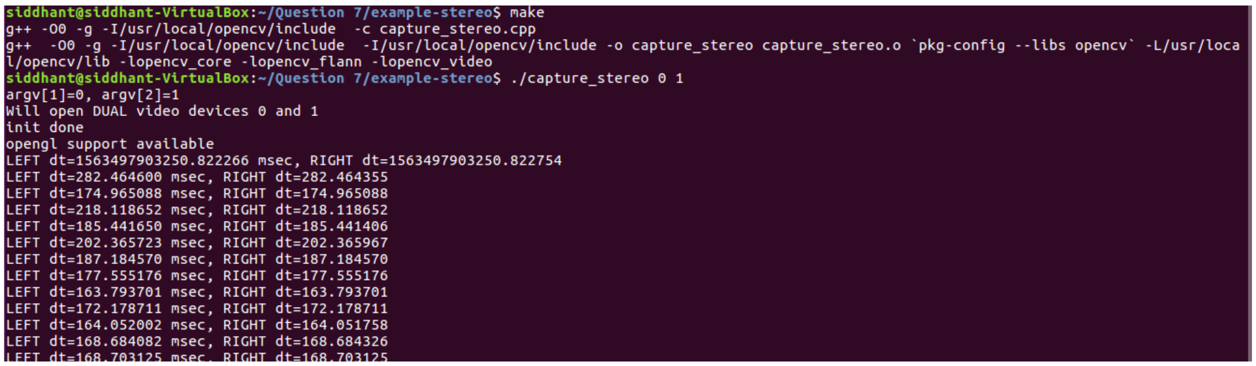
CODE DESCRIPTIONS

This code creates the respective detectors, descriptors, filters by taking respective arguments from the command line. The code exits if any of the above parameters are not created. The next step is to read the images depending on the parameters. If only one image is provided, it creates a copy for the 2nd image by itself. If both images are provided in the code, it reads both images to apply SIFT. The keypoints and descriptors are computed for the first image initially. Next step is to generate a random number and store it in an RNG class. The next step is to call the doIteration function. This function matches the keypoints between the two images. Pressing the ESC key exits the program.

In the doIteration function, the first step is to get a warperspective image by creating an appropriate kernel. Warpperspective image is only created if only one image is provided. This image is used to match keypoints with the original image. However, when both the images are obtained his step is skipped. Depending on if the warpperspective image is created or an image is accepted in the form of command line argument, feature detector is evaluated followed by computing the descriptors and keypoints. Now depending on the filter option provided, respective filter function is executed. The next step is to evaluate generic descriptor matcher followed by finding homography. The final step is to match points, draw matches and display it on the screen.

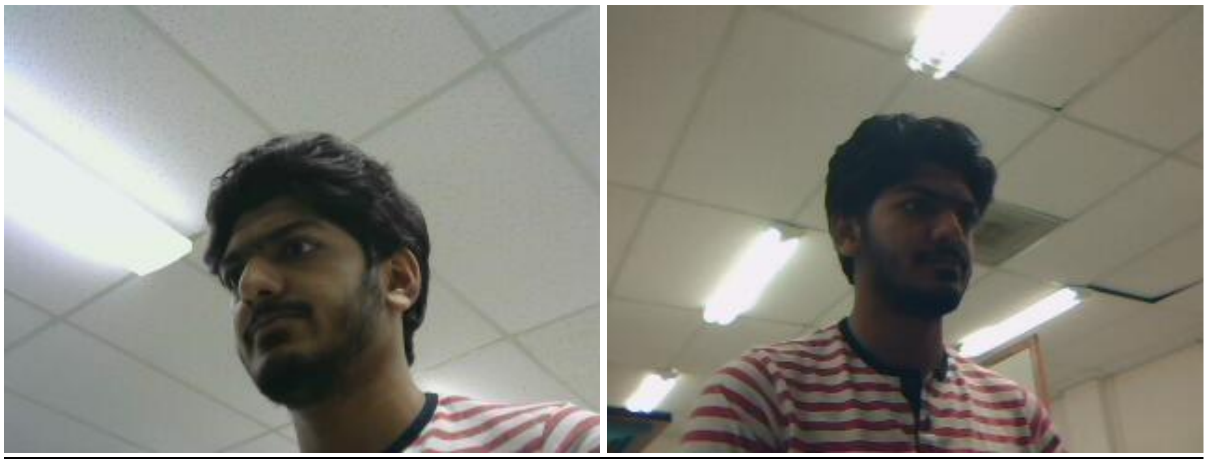
QUESTION 7

SCREENSHOTS

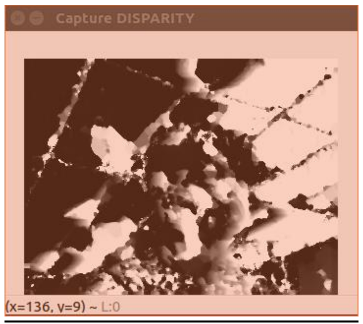


```
siddhant@siddhant-VirtualBox:~/Question 7/example-stereo$ make
g++ -O0 -g -I/usr/local/opencv/include  -c capture_stereo.cpp
g++  -O0 -g -I/usr/local/opencv/include  -I/usr/local/opencv/include -o capture_stereo capture_stereo.o `pkg-config --libs opencv` -L/usr/loca
l/opencv/lib -lopencv_core -lopencv_flann -lopencv_video
siddhant@siddhant-VirtualBox:~/Question 7/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=1563497903250.822266 msec, RIGHT dt=1563497903250.822754
LEFT dt=282.464600 msec, RIGHT dt=282.464355
LEFT dt=174.965088 msec, RIGHT dt=174.965088
LEFT dt=218.118652 msec, RIGHT dt=218.118652
LEFT dt=185.441650 msec, RIGHT dt=185.441406
LEFT dt=202.365723 msec, RIGHT dt=202.365967
LEFT dt=187.184570 msec, RIGHT dt=187.184570
LEFT dt=177.555176 msec, RIGHT dt=177.555176
LEFT dt=163.793701 msec, RIGHT dt=163.793701
LEFT dt=172.178711 msec, RIGHT dt=172.178711
LEFT dt=164.052002 msec, RIGHT dt=164.051758
LEFT dt=168.684082 msec, RIGHT dt=168.684326
LEFT dt=168.703125 msec, RIGHT dt=168.703125
```

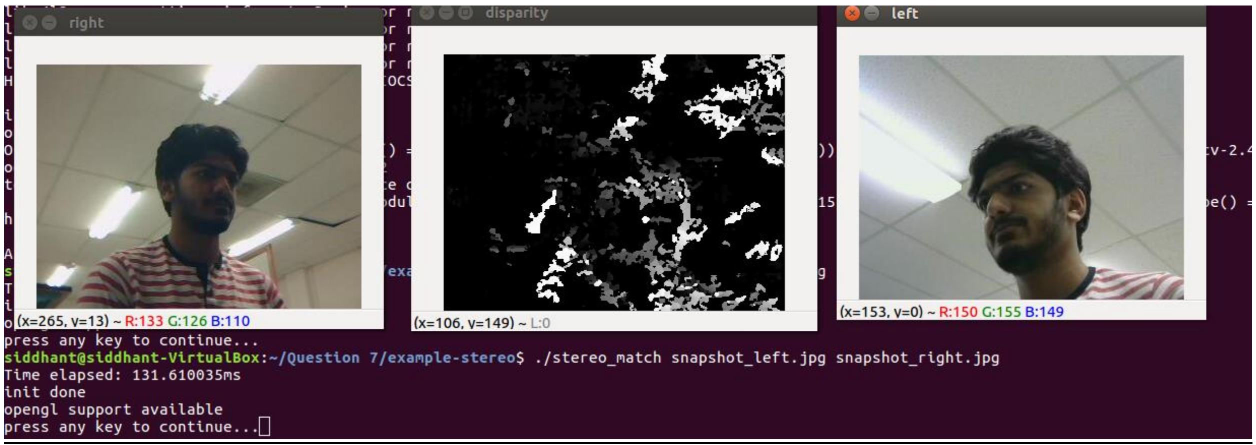
Make and run



Snapshot left and right



Disparity


right
disparity
left
(x=265, y=13) ~ R:133 G:126 B:110
(x=106, y=149) ~ L:0
(x=153, y=0) ~ R:150 G:155 B:149

```
press any key to continue...
siddhant@siddhant-VirtualBox:~/Question 7/example-stereo$ ./stereo_match snapshot_left.jpg snapshot_right.jpg
Time elapsed: 131.610035ms
init done
opengl support available
press any key to continue...
```

Stereo match make and run

CODE DESCRIPTIONS

The capture_stereo code is quite simple where depending on the command line parameters, frames are obtained from the connected cameras. A stereovar class is initialized and properties are assigned to this class. Depending on these parameters, the pixels in the 2 stereo images are compared with each other thus finding the depth map from the calculated disparity field. Timings are noted for the frames captured in both the cameras and disparity is calculated between the two using mystereovar function. The disparity field is reconstructed by using a combination of multigrids and a multi-level adaptive technique for the variational approach.

No, keypoints are not used for passive depth estimation. Yes they can be used but if keypoints are to be used SIFT is the best approach to go about. This method matches the images with same orientation and does it much faster. So introducing keypoints would destroy the notion of faster processing.

REFERENCES

- https://opencv.org/
- https://www.learnopencv.com/hough-transform-with-opencv-c-python/
- https://docs.opencv.org/3.4/da/df5/tutorial_py_sift_intro.html
- http://aishack.in/tutorials/sift-scale-invariant-feature-transform-introduction/
- https://www.informatik.uni-marburg.de/~thormae/paper/ISVC2009stereo.pdf