TRADE-OFF ANALYSIS

Gaurav Chauhan(2018CS50406), Rishi Sarraf(2019CS10393)

April 1, 2021

Metrics

Our trade-off analysis uses 3 metrics as defined below:

0.1 Runtime

Runtime is the total time spent by the program in computing the queueDensity. We calculate this time with the help of chrono library using chrono::high_resolution_clock in our program.

0.2 Error

This is the RMS error between the queue Density output of a method and that of the baseline. Calculated as

$$error = \sqrt{\sum_{n=1}^{k}(y_n^0 - y_n^i)^2/k}$$

where i = 1,2,3,4 depending on the Method, k = 1148 in our analysis (taking every 5th frame).

0.3 Utility

We define the a single valued utility as

$$utility = 1/1 + e^{\alpha}$$

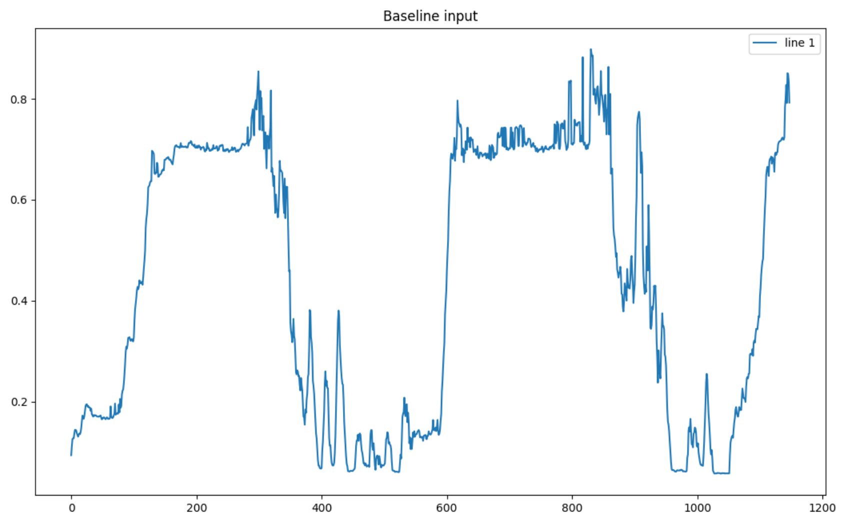
where

$$\alpha = \theta.error$$

here theta is estimated by trial and error for each method to obtain a good trade-off curve.

1 Method 1(Subsampling frames)

In this method we run the code at a baseline of 3fps and then skip to the nth frame skipping n-1 frames. The values of these n - 1 frames is estimated using weighted linear interpolation between the two points they lie between. The graph produced for the baseline case is as follows:



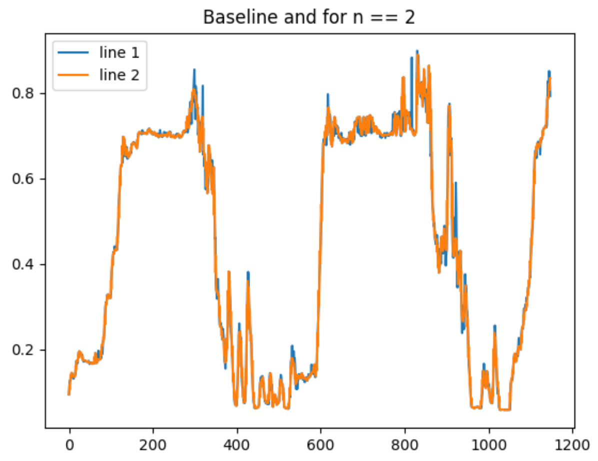
Method 1 optimizes the computation by letting go intermediate values. For e.g if earlier all the frames were being computed now only the 0th, (n)th, (2n)th..... and so on will be computed. We have linearly estimated the values which were skipped by using the value which arises by joining the two points.

The error estimated is using the RMS method, the estimated values are used to find the error (keeping original data as baseline). We find the root mean square values of the estimated value and note the values in Table 1 below.

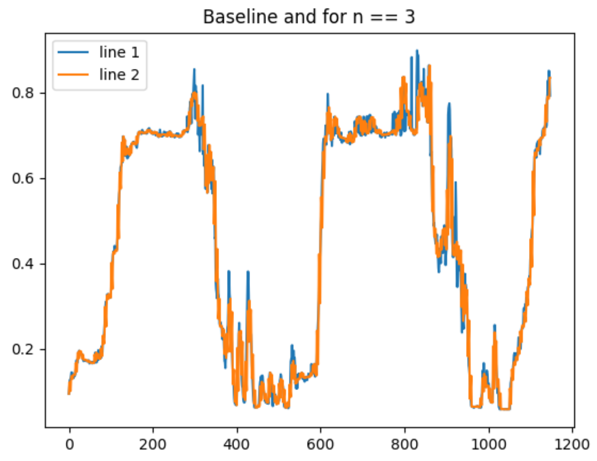
The queue density graph for various values is shown as below(the orange is the estimated graph obtained by skipping frames and the blue one is the output graph of baseline input values):

1.1 Parameters

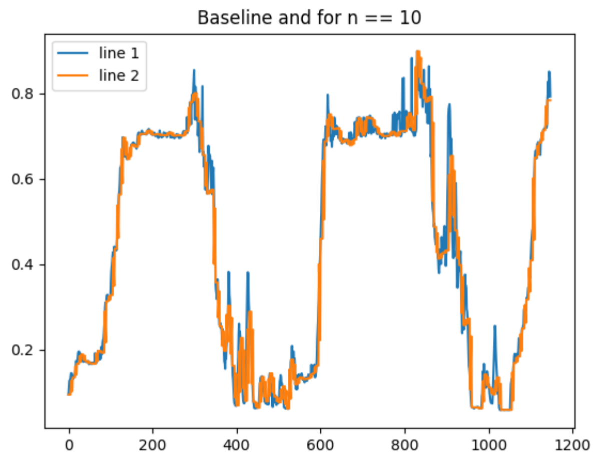
This method takes a single integer values parameter(n= number of frames skipped). We have analysed the behaviour of this method with the following parameters:
for n == 2:



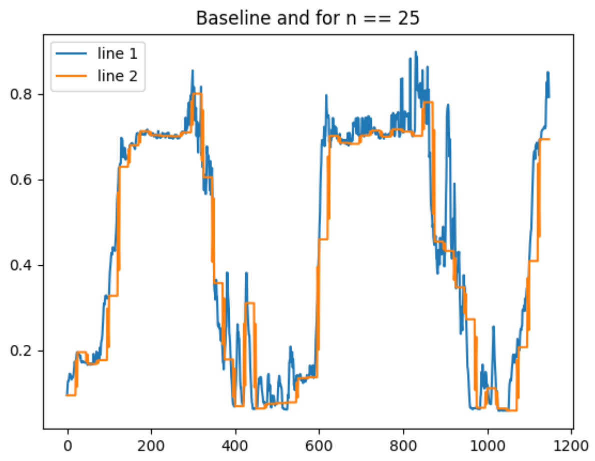
for n == 3:



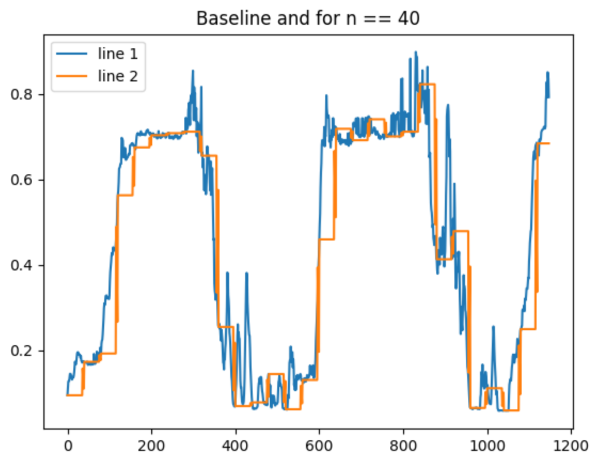
for n == 10:



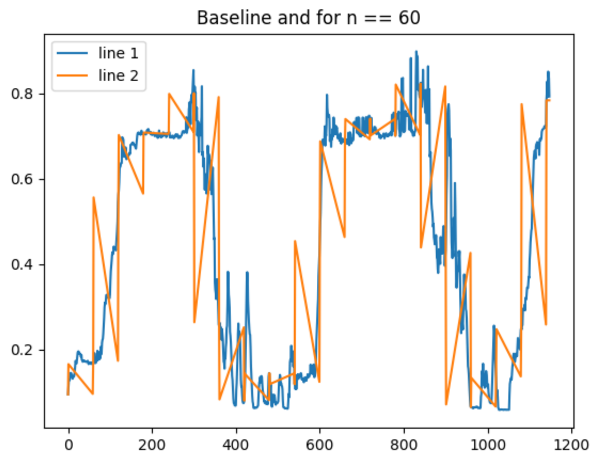
for n == 25:



for n == 40:



for n == 60:



1.2 Trade-off:

Table 1: n vs time

Value of n(number of frames skipped)	Time taken in microseconds	error (RMS)
n == 1	30840090	0
n == 2	27059414	0.017
n == 3	25652809	0.0265
n == 5	25254792	0.0351
n == 7	24098526	0.0496
n == 10	23460330	0.0627
n == 13	23148135	0.0734
n == 15	22951533	0.0778
n == 20	22792384	0.0892
n == 25	22437487	0.0940
n == 40	22271877	0.1183
n == 50	20944003	0.1442
n == 60	20112003	0.1521

————————

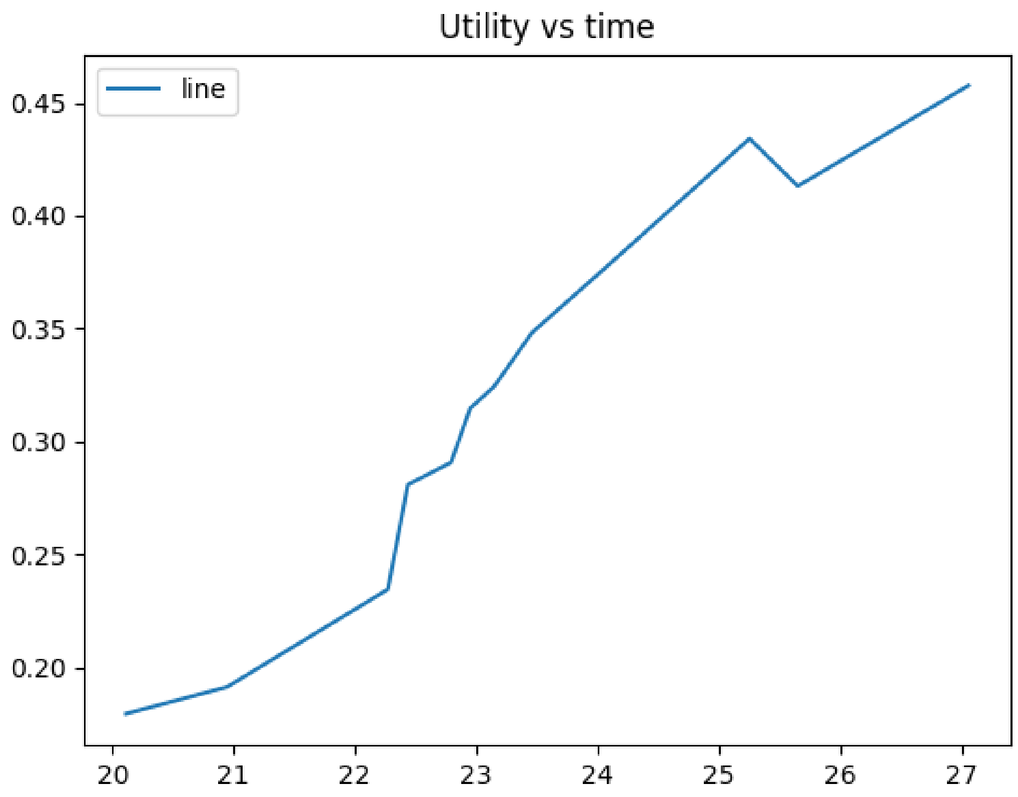
We define utility as:

$$utility = 1/1 + e^{\alpha}$$

where

$$\alpha = \theta.error$$

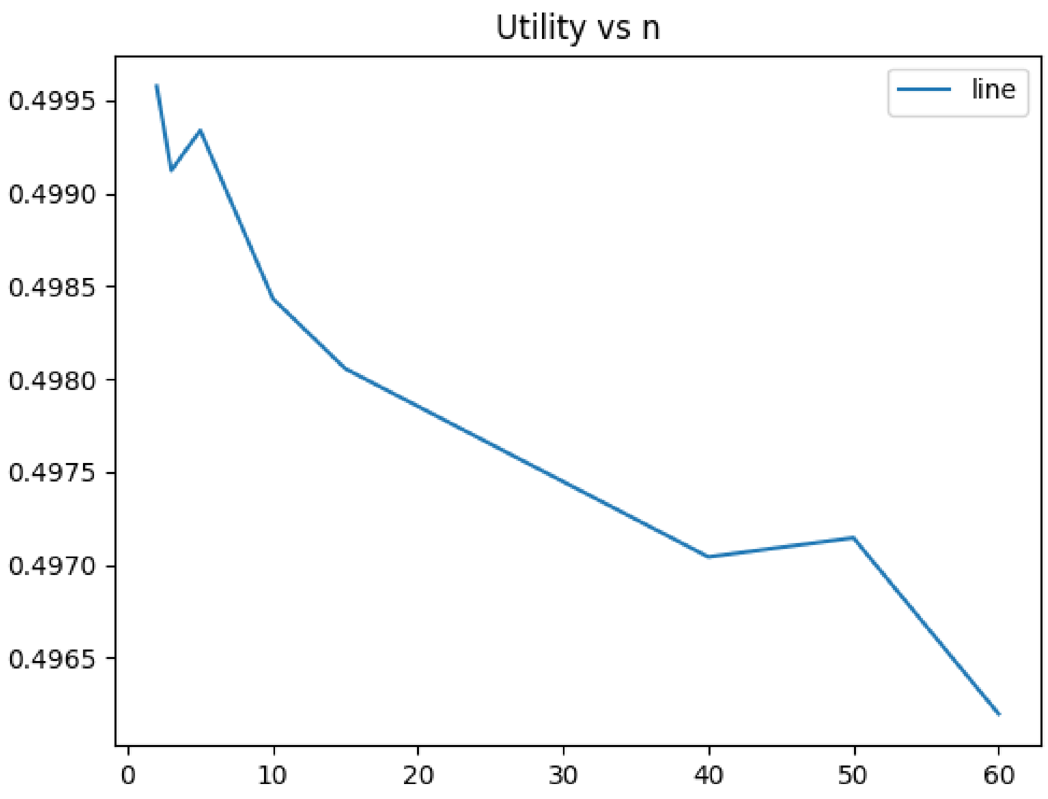
We understand that if error increases the utility would decrease(given theta is positive). So we plot the utility versus time graph to understand the optimisation quality.



We observe that when the time is more the value of the utility is more. This can be easily seen as the time is more when the number of skips is low and hence the utility is more.

(for above graph $\theta = 10$)

We also study the graph of utility vs the parameter(i.e. n the skip factor). We see that as the number of skips increase the error increase and the utility decreases. This is seen by the graph obtained as shown below:



(for above graph $\theta = 0.1$)

2 Method 2(Resolution Reduction)

In this method we have reduced the quality of the image. This can be done by reducing the quality in the x direction or the y direction. We have taken into account the x_factor and y_factor for the same and the outputs observed for the x,y pair are as follows:

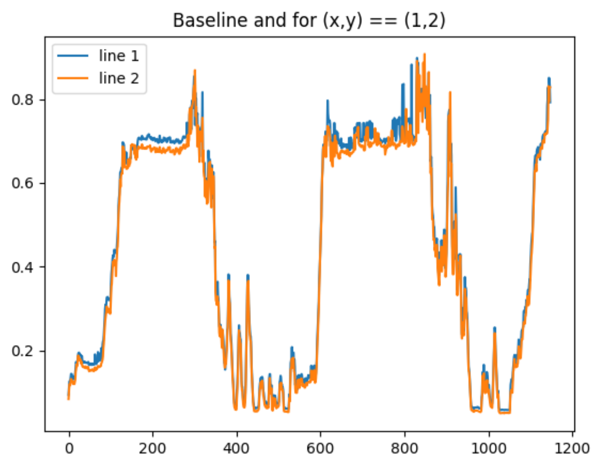
Table 2: n vs time

x_factor, y_factor	Time taken in seconds	error (RMS)
(x,y) == 1,1	34	0
(x,y) == 1,2	33	0.0250
(x,y) == 2,1	33	0.0164
(x,y) == 2,2	33	0.0278
(x,y) == 3,3	32	0.0305
(x,y) == 4,4	31	0.0393
(x,y) == 5,5	31	0.0458
(x,y) == 7,7	30	0.0626
(x,y) == 10,10	29	0.0865
(x,y) == 20,20	29	0.1329
(x,y) == 30,30	28	0.1538
(x,y) == 40,40	32	0.1858

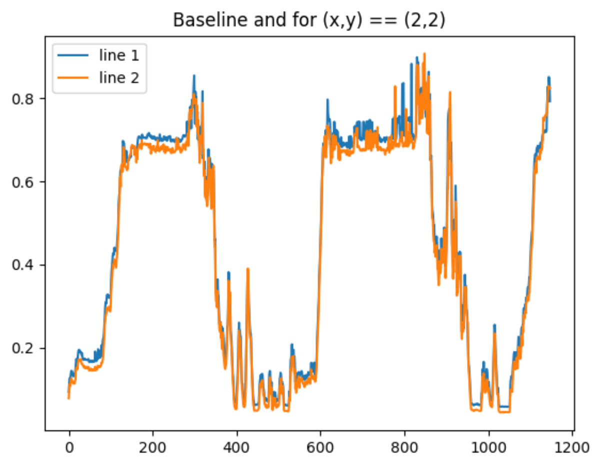
2.1 Parameters

This method takes 2 parameters, x_factor and y_factor(the factor by which resolution is to be decreased/divided along each axis). The following are some of the observations by varying parameters:

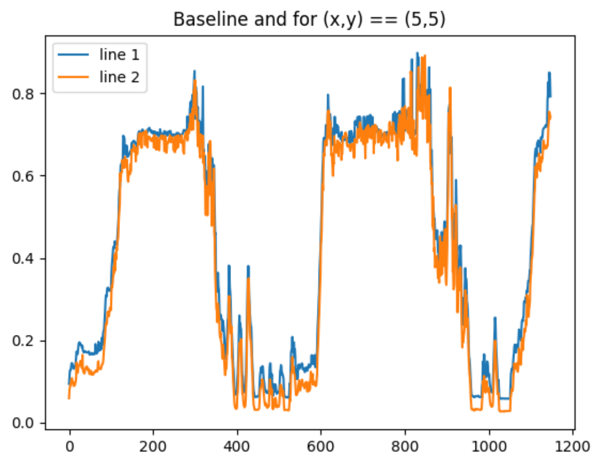
The graph obtained for (1,2) is:



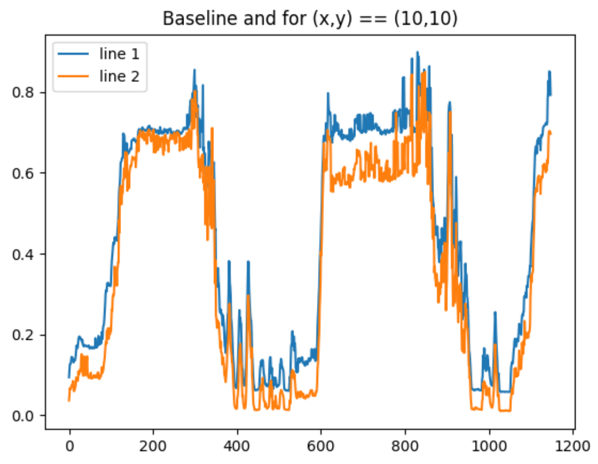
The graph obtained for (2,2) is:



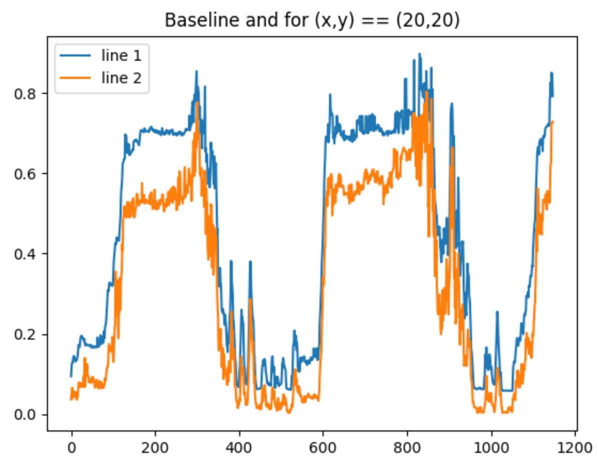
The graph obtained for (5,5) is:



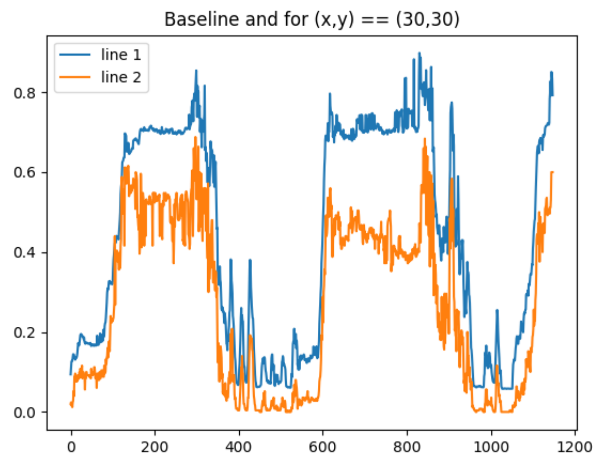
The graph obtained for (10,10) is:



The graph obtained for (20,20) is:



The graph obtained for (30,30) is:



2.2 Trade-off:

From the table above we see that the optimisation is not a good one because the time does not change significantly rather it goes up for higher values of the input as seen in the table. This could possibly be because the time taken in reducing the quality of the image also increases. The decrease in time is hence observed for some values but increases at x,y == 30, 30

We again define the notion of utility as in Method 1 as :

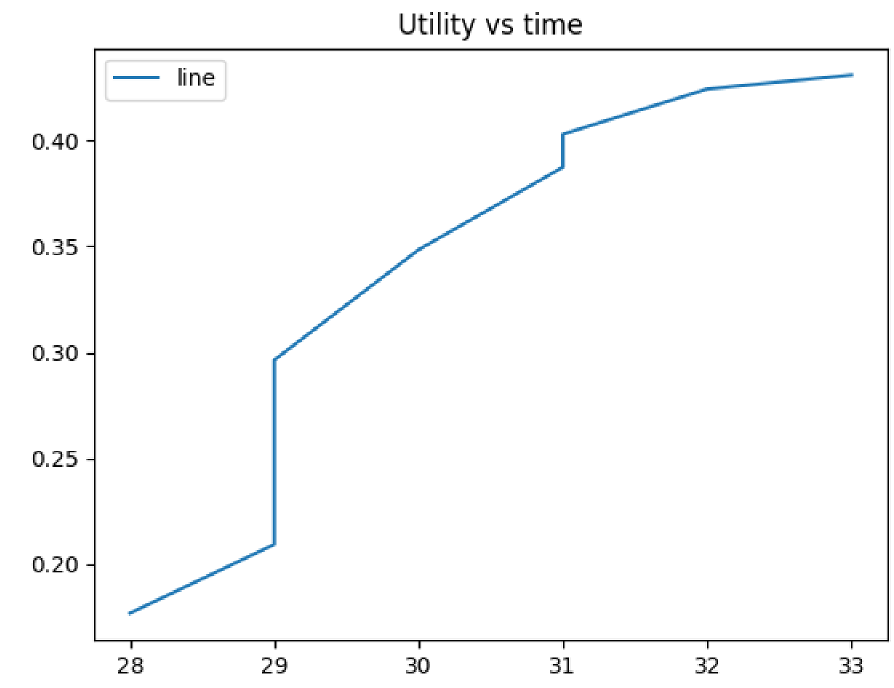
$$utility = 1/1 + e^{\alpha}$$

where

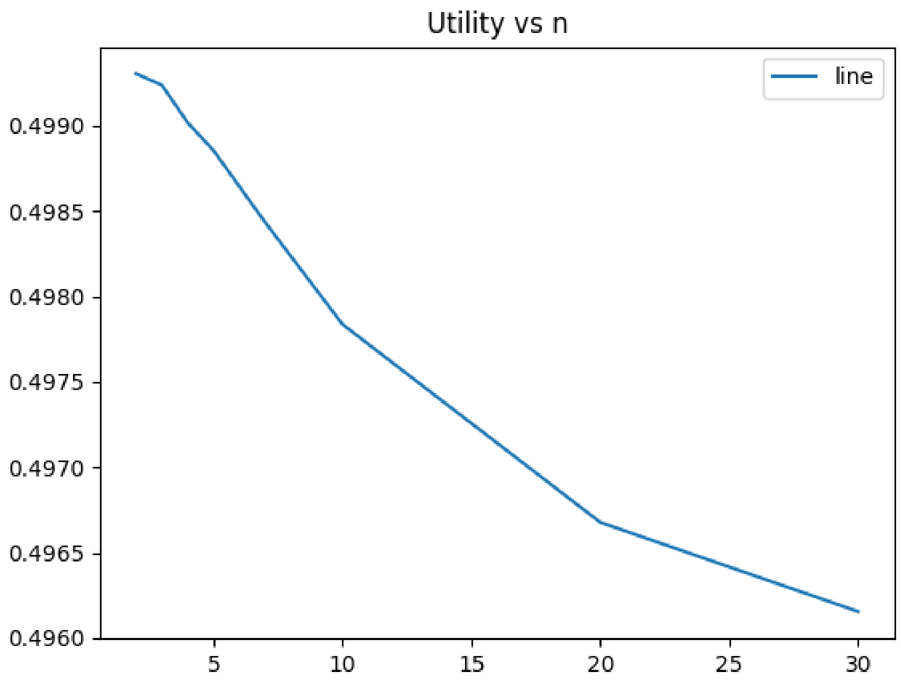
$$\alpha = \theta.error$$

We understand that if error increases the utility would decrease(given theta is positive). So we plot the utility versus time graph to understand the optimisation quality.

The graph is as shown below:



This graph accounts for the behaviour till x,y == 20, 20 after this the time starts increasing. theta here also was used as 10.



As the value of x_factor and y_factor increase the error increases which leads to a decrease in the utility. This is verified by the graph shown above.

3 Method 3(Spatial Division)

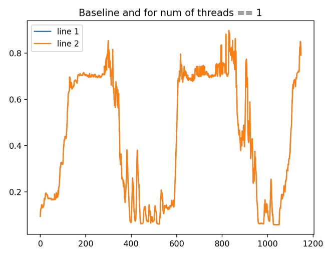
In this method we break the graph into a * b and then parallelly compute the density for the the given frame. This is a parallel optimisation, each thread works to compute the value corresponding to the block assigned to it. The total density will be equal to the sum of all densities (i.e. from each block). The threads do incur some overhead costs due to parrallisation of the process. The number of threads per image are a*b.

The following are the observations and the calculated values

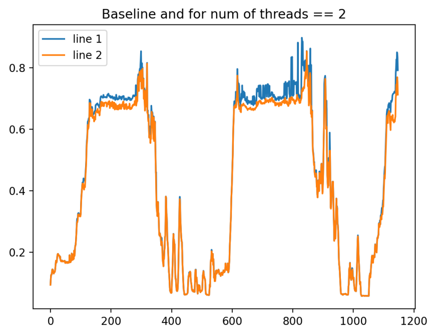
num of threads	runtime	error	utility
8	27	0.0674	0.338
7	28	0.0638	0.346
6	30	0.0578	0.359
5	29	0.0495	0.379
4	28	0.0516	0.374
3	29	0.0355	0.412
2	32	0.0286	0.423
1	33	0.0	0.5

3.1 Parameters

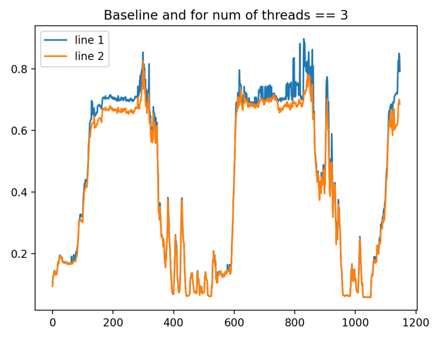
This method takes a single integer valued parameter(n= number of threads). The following are some of the observations taken by varying n:
for n == 1:



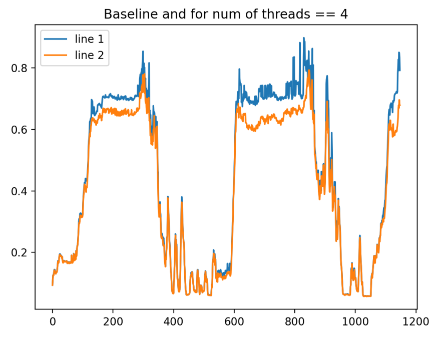
for n == 2:



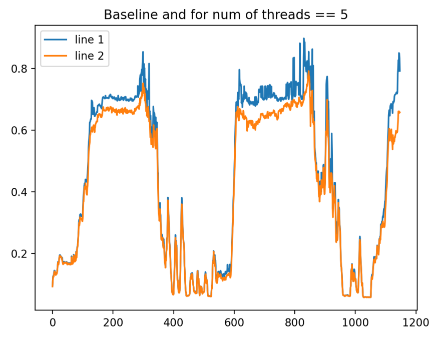
for n == 3:



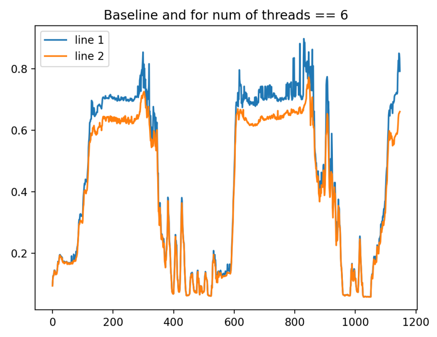
for n == 4:



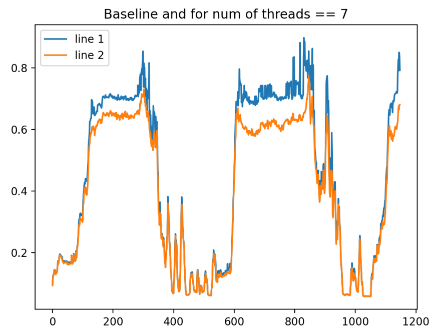
for n == 5:



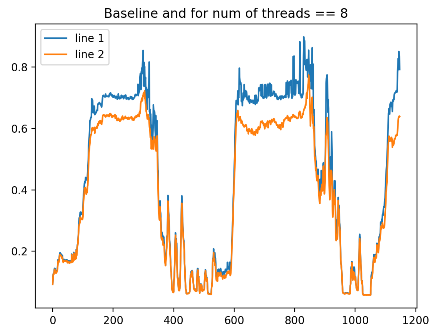
for n == 6:



for n == 7:



for n == 8:



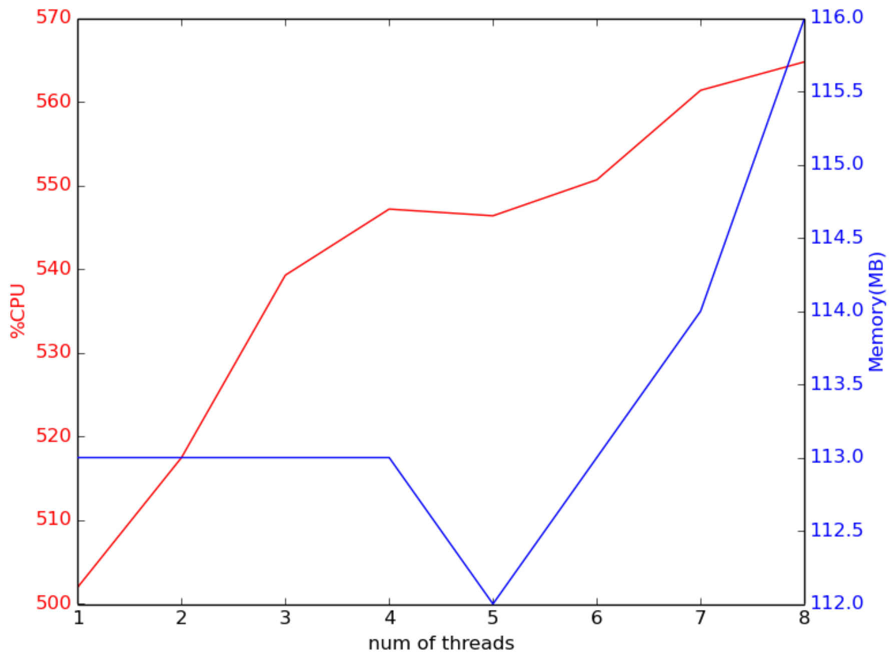
3.2 CPU Usage:

We now study the CPU utility as we change the number of threads in this method. The CPU usage trends show that the range of variation of %CPU is quite small, which is explained by the large amount of overhead processing in our program. The benefits of actual parallelisation that occurs in the form of queue density computation is neutralised by the extra work done in creating those number of threads. However, we do observe slight variations in CPU usage that suggest that using more number of threads implies more computation in the CPU.
The memory usage shows a similar trend but it saturates, this observation can be explained by the fact that the system allocates a fixed amount of memory to the program and the threads in our case do not add significantly to the threshold minimum memory(this threshold appears to be around 110 MB in the following table). Another noteworthy observation is that the memory usage increases continuously as the program executes, this is understood from the growth of stack memory in the course of program execution.
(all the values are maximum values observed through the course of runtime, the observations were made manually using 'top' at the terminal)
Although the peak %CPU doesn't show much variation, a real time manual observation showed that these peak values are somewhat misleading and mean values will have more significant variation.

The following is the CPU usage analysis for method 3:

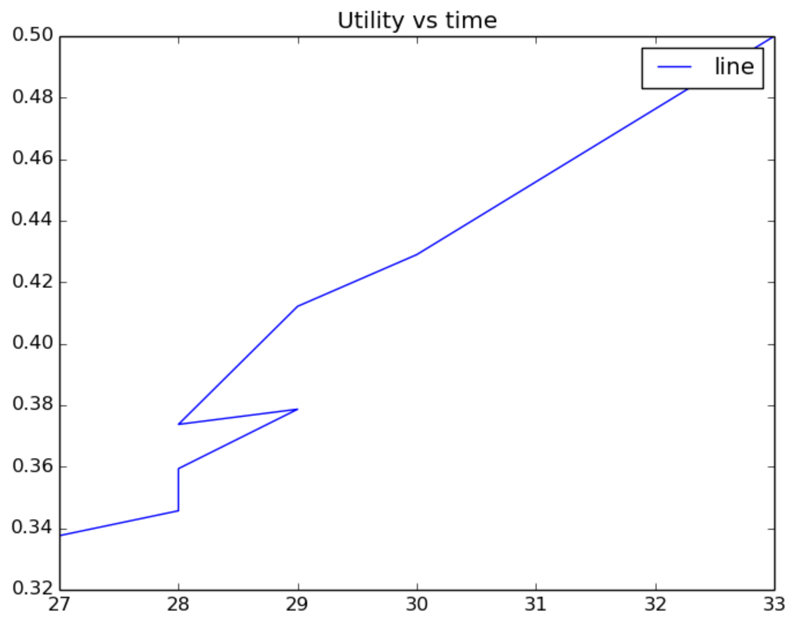
num of threads	%CPU	Memory(MB)
8	564.8	116
7	561.4	114
6	550.7	113
5	546.4	112
4	547.2	113
3	539.3	113
2	517.5	113.1
1	502.0	113

The plot of CPU utility for spatial division is as follows:

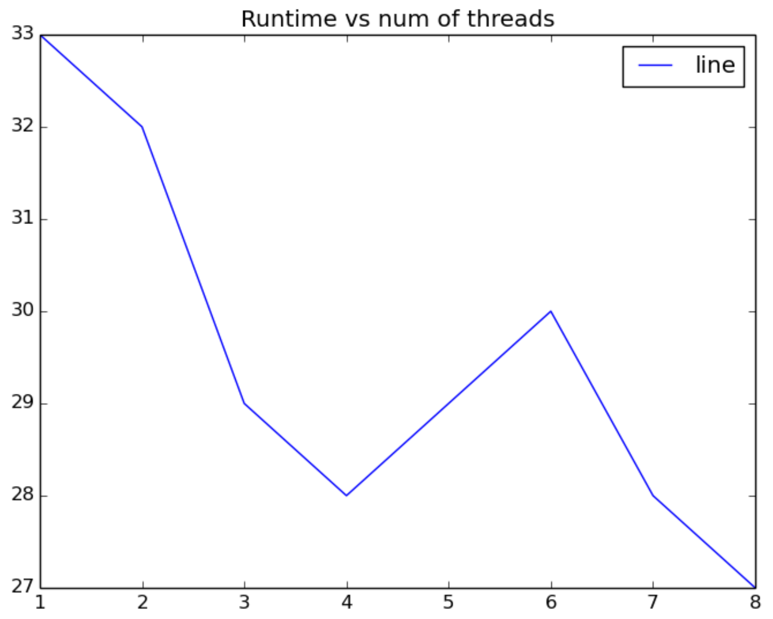


3.3 Trade-off:

The error is expected to increase and the utility is expected to decrease as we divide our frames into larger number of pieces(and hence larger number of threads). The utility vs time variation can be observed in the following graph:



The runtime vs number of threads plot is as follows:



4 Method 4(Temporal Division)

This method also uses parallel computation to optimise the program. In this some frames are taken are are parrallely assigned to threads to compute. After each thread completes the operation then next set of frames are computed.

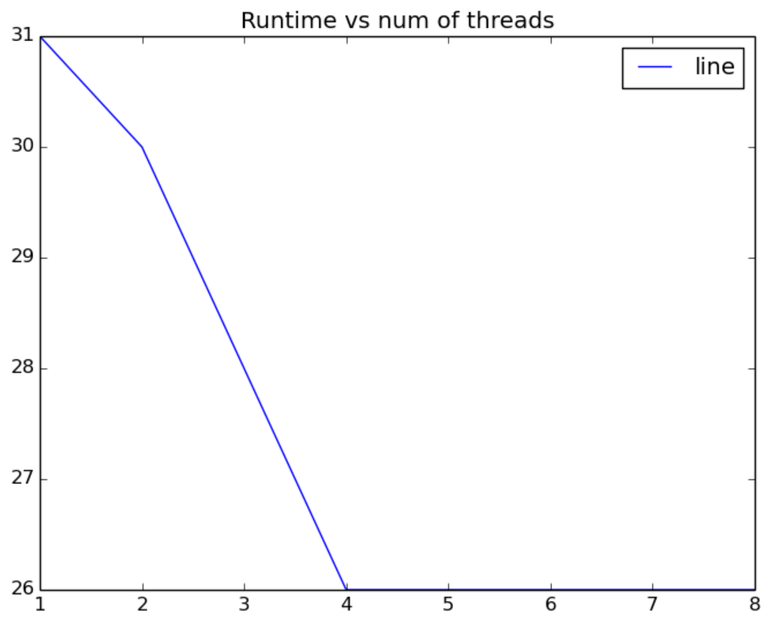
Functionally, the work performed is exactly the same as baseline and hence there should be 0 error which is observed in the table shown below.

Parameter: This method takes a single integer valued parameter(n = number of threads).

num of threads	runtime	error	utility
1	31	0.0	0.5
2	30	0.0	0.5
3	28	0.0	0.5
4	26	0.0	0.5
5	26	0.0	0.5
6	26	0.0	0.5
7	26	0.0	0.5
8	26	0.0	0.5
12	26	0.0	0.5
64	25	0.0	0.5

4.1 Trade-off:

We observe a decrease in runtime as the number of threads is increased, this is an expected behaviour as more threads means more parallelisation and hence faster computation. However, the range of variation is not too large which can be explained by a large amount of overhead computation as compared to queue Density computation. The number of threads vs runtime plot for temporal division is as follows:



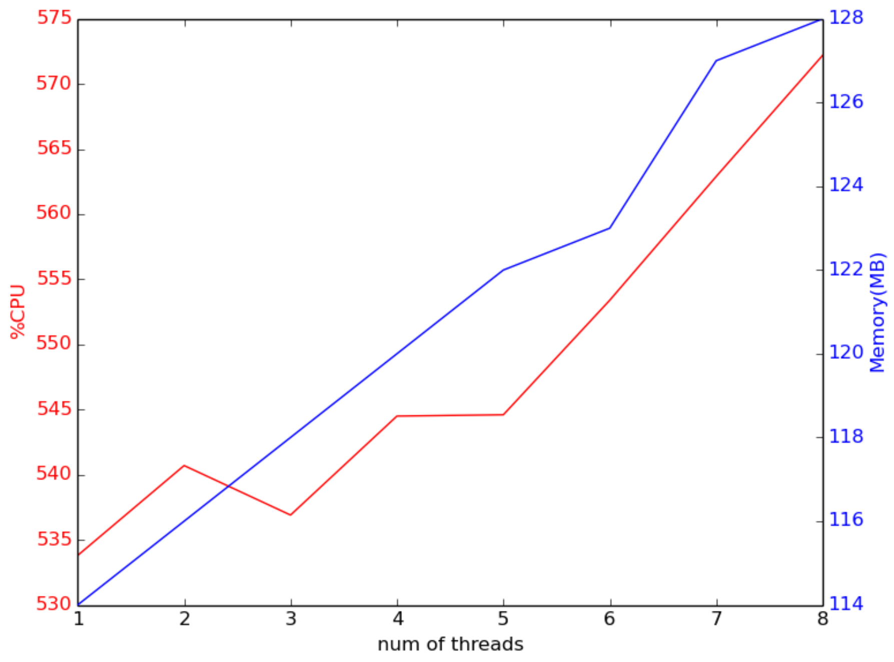
4.2 CPU Usage:

All the observations of memory and CPU usage made in method 3 hold here as well. Apart from that, we observe that the peak %CPU usage in this method is more, this follows from the fact that the computation that occurs in each thread is more heavy in as we do temporal division of work among threads passing entire frame to each thread. This also explains why the memory usage is significantly larger than method 3. (In spatial division, we only passed parts of frame to each thread)
(all the values are maximum values observed through the course of runtime, the observations were made manually using 'top' at the terminal)
Although the peak %CPU doesn't show much variation, a real time manual observation showed that these peak values are somewhat misleading and mean values will have more significant variation.

The following is th e CPU usage analysis for method 4:

num of threads	%CPU	Memory(MB)
8	572.2	128
7	562.9	127
6	553.4	123
5	544.6	122
4	544.5	120
3	536.9	118
2	540.7	116
1	533.8	114

The plot of CPU utility for temporal division is as follows:


%CPU
Memory(MB)
num of threads