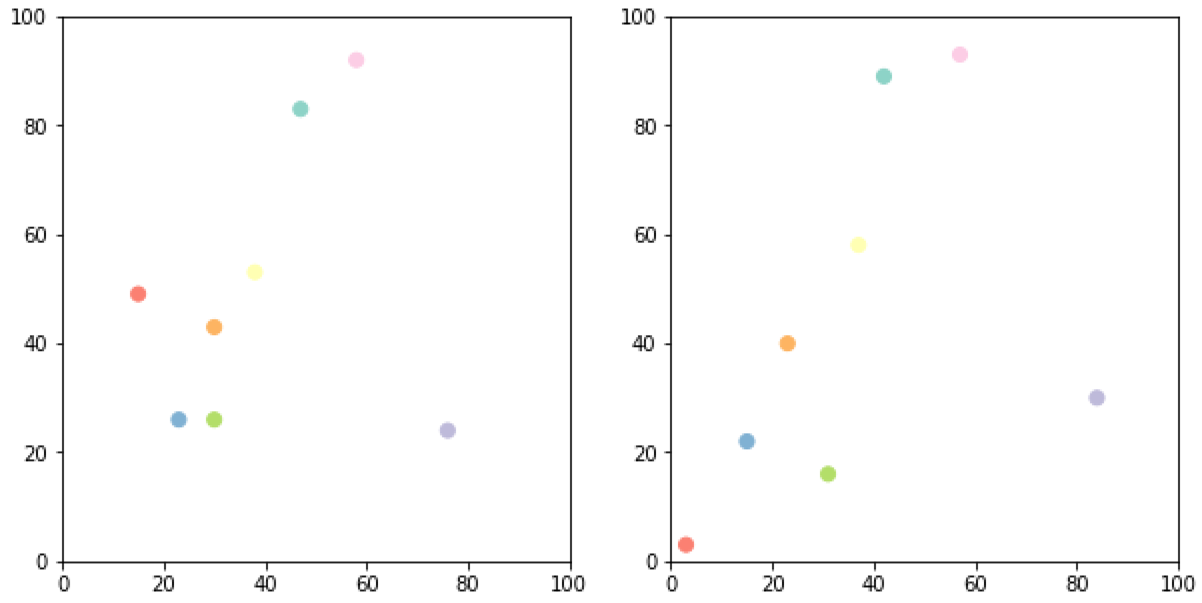
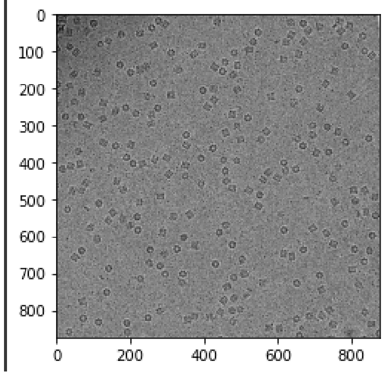
The implementation of the get cost function is provided in the code.
The matrix was constructed by primarily following the steps in the paper.
The first step in the code was to construct the top left matrix, which is essentially the coat matrix, across frame0 and frame1, which gave the L2 distance.
The second step was to take the minimum value from the cost value and construct the bottom right part of the whole matrix.
Then the standard deviation was calculated for the top right part of the matrix (calculating r = 3*standard deviation). The standard deviation was calculated row-wise in the cost matrix. If the value in the r value was greater than in the cost matrix, then the value is replaced with r in the cost matrix.
The value 'd' was calculated by taking the 80th of the values in the cost-matrix. The top right quadrant of the matrix comprises the values x, which were the cut-off values when compared with r in the cost matrix portion (the top left). The diagonal of the top right matrix was the d value. And since for this whole matrix, both b and d are similar and the frames are of equal length, the bottom left and top right are the same. The whole matrix resulted to be of size (16 by 16) and provided the following output.



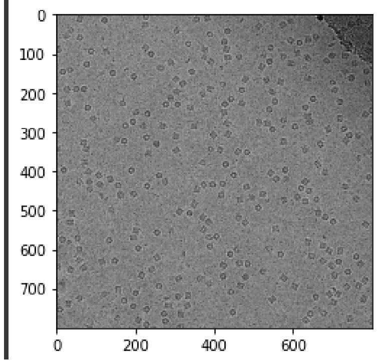
The lap function was implemented in the code, using the package linear_sum_assignment() method, that takes in the cost matrix (returned in the get_cost function), as an argument parameter, by deploying the scipy.optimize function.
The linear sum assignment method takes in the cost matrix as the input, where for each [i,j] value of the matrix, is described by the cost of i and j data. The objective is to find the complete assignment of one particular to the next in the second frame. And the algorithm then calculates the optimal cost and outputs an array.

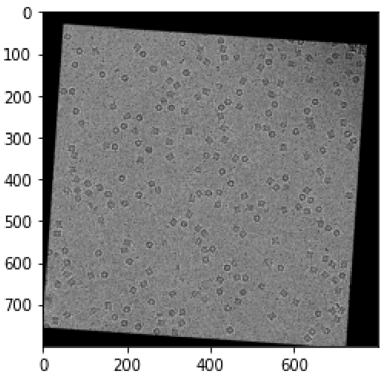
2. Image registration is basically to pixel wise map the input image into the reference image to align different images of the same scene. The first basic step was to read the input and reference images as separate arrays using common image reading tools such as open cv2.

```
Input Image
AxesImage(54,36;334.8x217.44)
```



```
Reference Image:
AxesImage(54,36;334.8x217.44)
```

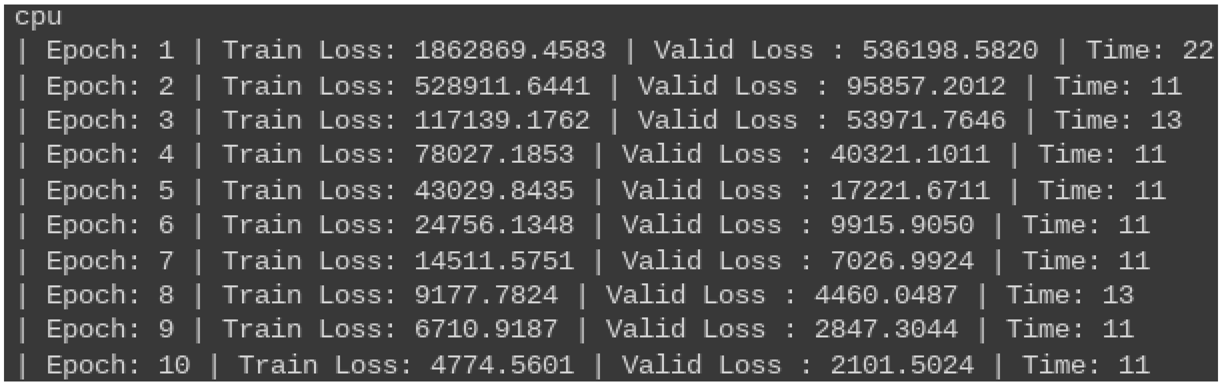




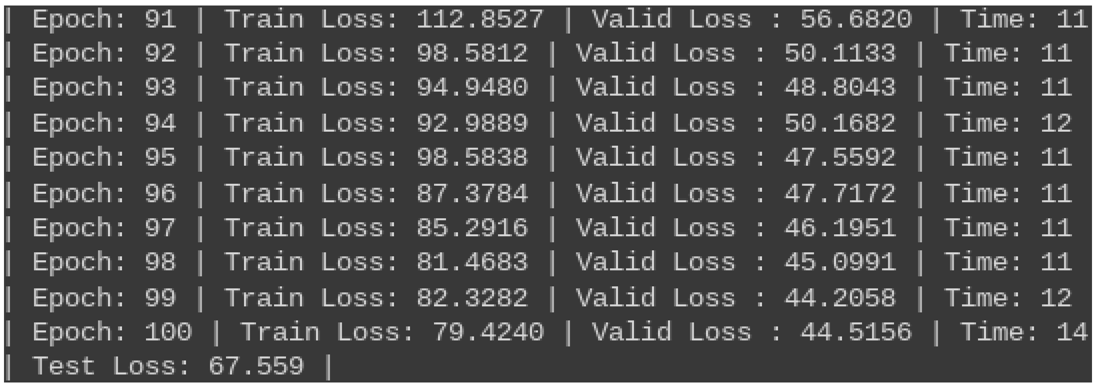
Output Image

For convenience the images were converted to grayscale images, as reducing the number of channels makes a lot of image tool application easier. Then to implement the ORB detector, using cv2’s orb creator function of 5000 features was used and applied on both the input image and the reference image. Then these features were being matched from the image to be aligned, to the reference image by storing the coordinates of the associated key points. And these key points (points/pixels that stand out) were chosen to compute the transform. ORB orient and rotation brief package of open CV was used for this transform mapping. This was chosen as it provided both the key points in the image with the associated image descriptor, which helped to match the key points amongst the two images. Additionally, a brute force matcher BFMatcher was deployed, as it seemingly retrieved the best match. by deploying ‘cv2.NORM_HAMMING’ was the mode used to Brute Force matcher with Hamming distance as measurement mode.This provided a good alignment result. Then the two sets of descriptors were matched. And was sorted based on the hamming distance. After sorting, BFMatcher.match() was used to fetch the top k key points and remove the noisy ones. The value of k was set to 0.9, which is the top 90% best matches were picked. Finally, the homography matrix for the images were calculated, using cv2.findHomography, with cv2.RANSAC (which is a method to estimate the model parameters in the presence of outliers to find data points/ pixels that are noisy and wrong). And then by deploying cv2.warpPerspective, with its associated required parameters, the original unaligned image was transformed. The cv2.warpPerspective basically takes the input points and outputs the transformation matrix for that image (based on the previous calculations).

3. The loss value for the model has significantly decreased over the epoch values, and seemingly would have produced better results if the epoch number was over 500 (which was computationally taking time and crashing colab).
The loss started off extremely high. And after 100 epochs, it was still high but very less, demonstrating that the model is seemingly working.

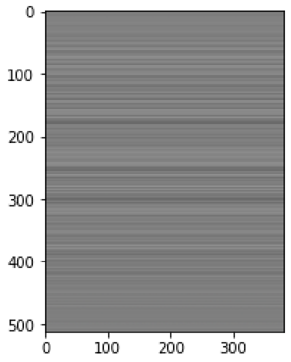


```
cpu
| Epoch: 1 | Train Loss: 1862869.4583 | Valid Loss : 536198.5820 | Time: 22
| Epoch: 2 | Train Loss: 528911.6441 | Valid Loss : 95857.2012 | Time: 11
| Epoch: 3 | Train Loss: 117139.1762 | Valid Loss : 53971.7646 | Time: 13
| Epoch: 4 | Train Loss: 78027.1853 | Valid Loss : 40321.1011 | Time: 11
| Epoch: 5 | Train Loss: 43029.8435 | Valid Loss : 17221.6711 | Time: 11
| Epoch: 6 | Train Loss: 24756.1348 | Valid Loss : 9915.9050 | Time: 11
| Epoch: 7 | Train Loss: 14511.5751 | Valid Loss : 7026.9924 | Time: 11
| Epoch: 8 | Train Loss: 9177.7824 | Valid Loss : 4460.0487 | Time: 13
| Epoch: 9 | Train Loss: 6710.9187 | Valid Loss : 2847.3044 | Time: 11
| Epoch: 10 | Train Loss: 4774.5601 | Valid Loss : 2101.5024 | Time: 11
```



```
| Epoch: 91 | Train Loss: 112.8527 | Valid Loss : 56.6820 | Time: 11
| Epoch: 92 | Train Loss: 98.5812 | Valid Loss : 50.1133 | Time: 11
| Epoch: 93 | Train Loss: 94.9480 | Valid Loss : 48.8043 | Time: 11
| Epoch: 94 | Train Loss: 92.9889 | Valid Loss : 50.1682 | Time: 12
| Epoch: 95 | Train Loss: 98.5838 | Valid Loss : 47.5592 | Time: 11
| Epoch: 96 | Train Loss: 87.3784 | Valid Loss : 47.7172 | Time: 11
| Epoch: 97 | Train Loss: 85.2916 | Valid Loss : 46.1951 | Time: 11
| Epoch: 98 | Train Loss: 81.4683 | Valid Loss : 45.0991 | Time: 11
| Epoch: 99 | Train Loss: 82.3282 | Valid Loss : 44.2058 | Time: 12
| Epoch: 100 | Train Loss: 79.4240 | Valid Loss : 44.5156 | Time: 14
| Test Loss: 67.559 |
```

More or less of the generated image looked alike and was obscure and not clear due to the high loss value. The depth of the neural encoders and decoders could have significantly helped reduce the problem, however it was causing the colab to crash (for technical reasons) and was taking too much cpu time (as gpu was not available).



Explanation of the implementation of the new image:

A Variational Autoencoder (VA) class was coded that structured the encoder and decoder of the VA model. The encoder returned the forward final pass of the input and ‘mu’ was the preceding value to the sigma. And in the forward method of the VA, the sample from latent distribution from the encoder was calculated (using torch.randn_like(sigma)). And then the output from the forward method was put together by operating mu + sigma*epsilon (where epsilon is the sample from the latent distribution). This is a core difference between VA and autoencoders in general. For calculating the loss, in addition to the loss function generated by the model, the kale divergence loss was also calculated and summed with the total loss. The learning rate for the model was 0.01 since the loss was too high to begin with and it would require really high epochs to reach a significantly low loss to reconstruct a quality image. So, the standard learning rate of 0.0001 was not used here. The number of epochs was set 100, and yet it seemed too less depending on the loss values generated by the model.
The main architecture of the model consisted of three linear layers in the encoder and decoder (following standard norm). A convolution could have been used to test, but just required more computation on colab (can be seen as being commented out in the code documented).
Standard Adam optimizer was used and for loss function cross entropy loss function was used (which was tested against other loss functions such as mean squared error loss, and seemed to give the faster decreased loss values).
The implementation was significant when sampling the latent distribution in the forward function and when calculating the loss differently by incorporating the kale divergence loss.
Then finally, the model was evaluated using a validation data set (model output shown above), which also showed promise in decreased loss value and if continued with a sufficient epoch number, could have helped reach a good model that properly reconstructed images as expected. Conclusively, the variational autoencoder (VA) is reconstructing the image, in a probabilistic manner for describing an observation in latent space. Thus, rather than building an encoder that outputs a single value to describe each latent state attribute, here we formulated the encoder to describe a probability distribution for each latent attribute.