Algorithms in Computer Vision

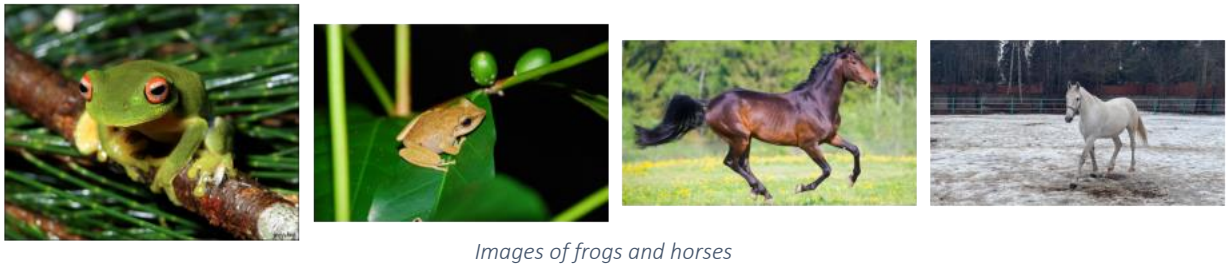
HW 3

Submitted by:
Nir Sassy: 316491737

Part 1 – Classic vs Deep Learning-Based Segmentation

In this section we will compare and discuss the results of segmenting objects out of images using classic methods and deep-learning methods.

1. First, we loaded some images:



Images of frogs and horses

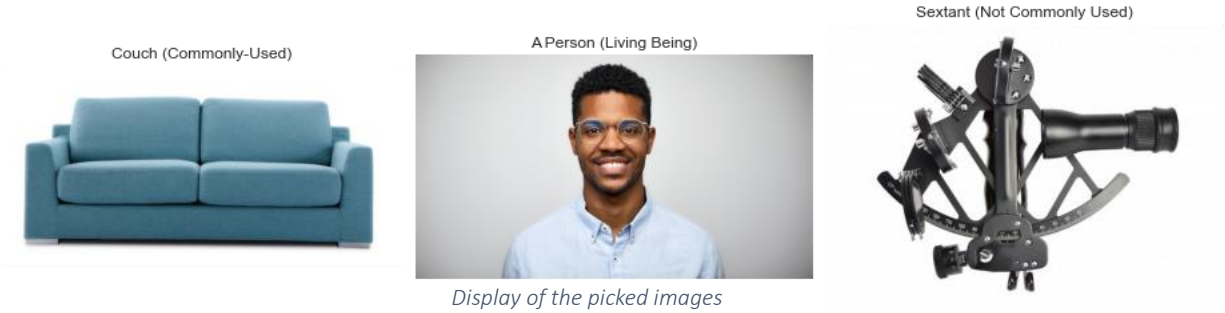
2. For the classic method, we used OpenCV's 'grabCut' algorithm. The algorithm is based on graphs cuts, while the user needs to input the bounding box for the object to be segmented. The algorithm creates a graph of the image where each pixel is a node and assigns weights to edges between nodes based on proximity and similarity in color. It uses this initial bounding box to separate the pixels to foreground and background nodes and computes a minimum cut to separate the foreground and background (and thus segmenting the object) and iterates until convergence.
   The main advantage of this method is that it doesn't need to be trained on specific labels. Also, this technique is better in segmenting 'inner edges' such as holes and supposed to fix the edges segmentation along the way. However, in practice, it's not always the case.
   The main con for this method is the need for stating the bound box in advance. It can be problematic for moving objects and just make for hard work compared to NN-methods. Also, in areas where it's hard to distinct between the object and the background, this method usually gives bad results (For example, a white horse on a white background)
   For the deep-learning method, we used 'deeplabv3' algorithm. This algorithm is using a pretrained 'ResNet101' network as its CNN backbone which extracts features and uses an architecture of parallel dilated convolutions to capture contextual information at multiple scales which called 'Atrous Spatial Pyramid Pooling'. Finally, the algorithm uses a 1x1 convolution that gives the final classification for each pixel, therefore creating a mask for the desired object.
   This method works better especially on objects that the network can classify and was trained on. It can also make multiple segmentation on one image, and there is no need to specify a bounding box. However, it requires training a neural network model which is computationally expensive, edges are more likely to be rough, misclassifications widely affect segmentation and having problem segmenting two connected objects at once (Like a man holding a towel) because of the labeling.

Frogs and Horses Images – Segmentation

Frog, for example, isn't one of the deep network trained labels, and we can see it's poor performance on the first frog image. On the classic method, we can see that overall, the object segmented better, but introduced a new artifact in

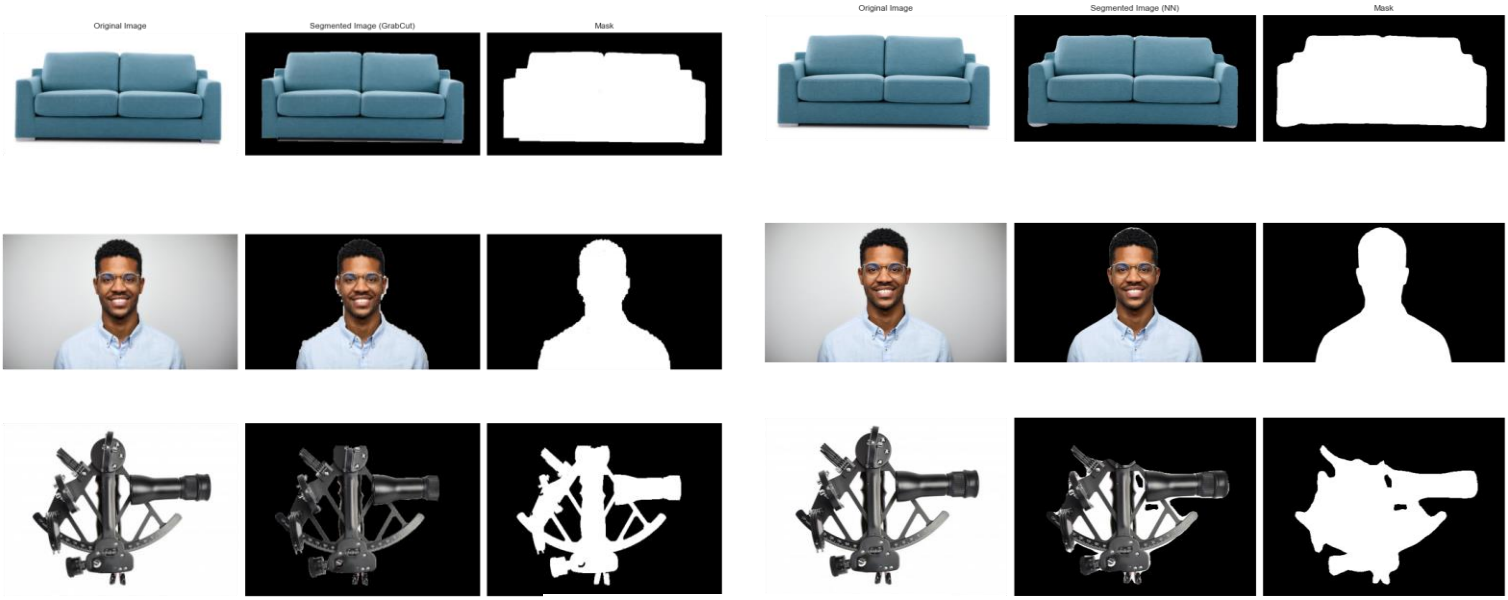
its eyes, where it considered most of it as part of the background. The second frog image segmented better, but with some details from the background. This was expected as this frog color is much more 'sticking out' from the background compared to the first. The horses were segmented almost perfectly on the NN-method (as horse is one of the trained labels) but with somewhat rough edges. The classic method did a good job on the first horse image in which the horse color is very different from the background, but on the second image, where the horse color is almost like the background color, it considered a very big part of the image's background (near the horse legs) as foreground.

3. We picked 3 images from the internet of a living being (A person), commonly used object (A couch) and a not-so commonly used object (A sextant):



Display of the picked images

4. We performed segmentation with the two methods on the picked images:



Picked Images segmentation comparison.

Overall, the segmentation went well on these images. However, since the person is close to the top edge of the image, picking the right rectangle coordinates can be tricky as we needed to change this parameter a lot of times until we got good results. Finally, we were able to fully segment him from the image, but the edges are very rough hinting that the algorithm were unable to converge to an optimal solution. The NN-method did a better job on the person image (which is expected as it's a very common class in its training data) except for the sextant which is the not commonly used item that it probably almost didn't on. The upper part of the object is classified as background and the inner 'holes', which are background, classified as part of the object. As discussed earlier, this is where the classic method performs better as it's not dependent on labels and training data. For the couch image, both algorithms worked well without any tuning, however the classic method did consider the shadow of the couch on the floor as part of the foreground, resulting in slightly worse segmentation.

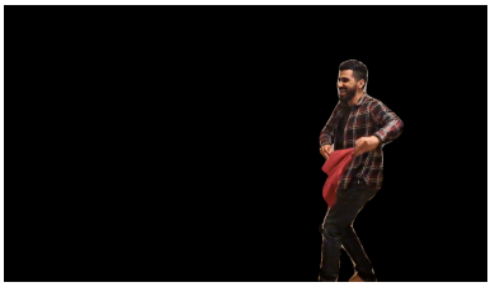
Part 2 – Jurassic Fishbach

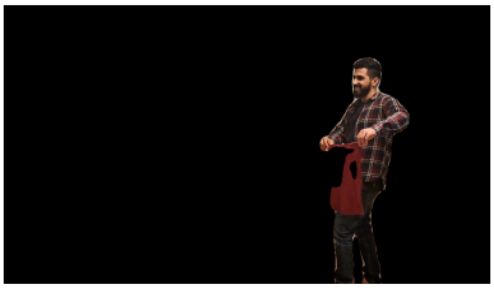
In this slightly embarrassing part, we created a video using segmentation. The idea for the video it is to play as a Spanish Bullfighter trying to anger a bull-looking dinosaur which roars back.

1. First, we filmed a video of Nir in his living room holding a piece of red cloth:


Two frames from the video filmed.

2. We segmented Nir from the frames using the NN-method:



Nir segmented from the frame (Right)
The semantic mask from the network (Left)

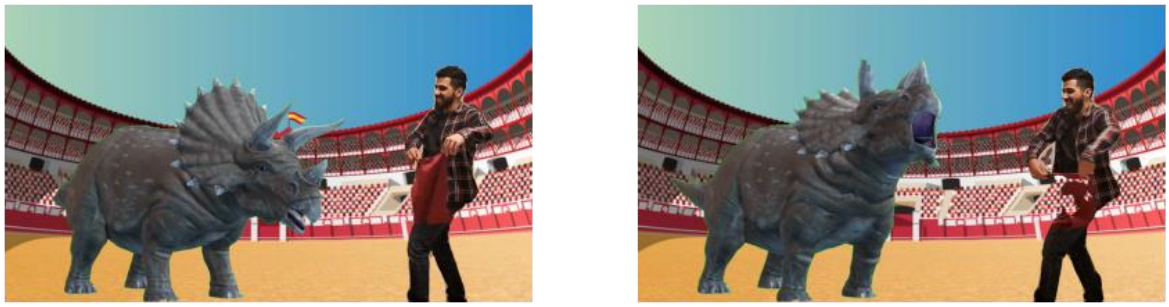
We can see that the piece of cloth is classified as background for some frames. This is one of the setbacks of the NN method (which still performed much better than the classic method on this task and did it in a reasonable time) as discussed earlier.

3. We picked a different model (from PixaBay) from the one given in the data folder. We performed segmentation using the classic method this time as it worked better, especially around the edges, due to the clear distinction between the object and the background.

Dinosaur model segmented (Left). Segmentation Mask (Right)

One of the major setbacks of the classic method is defining the bounding box in advance. Here, luckily, the object doesn't move much except for its head when it roars. This makes his horns go higher and sometime get out of the bounding box, which results in having them as background. We tried to change the rectangle parameters, but wider rectangles would result in a bad segmentation due to un-convergence of the algorithm. The NN-method, on the other hand, classified some of the background around the left as foreground, had rough edges, and overall did a worse job.

4. Putting it together for a video. Here are some frames from the video:



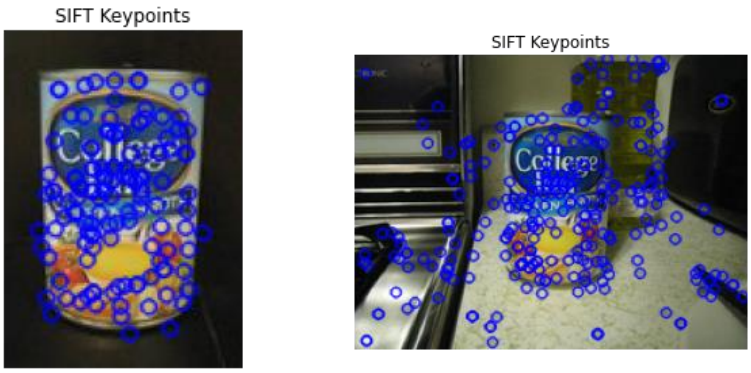
Frames from the final video

The background is an empty bullfighting arena. The objects (Nir and the dino) were adjusted in space using affine transformation such that it will look slightly better. The final video is in the output directory and here (YouTube).

Part 3 – Planar Homographies

In this section we will demonstrate the uses of planar homographies.

1. First, we implemented 'SIFT_discriptors' function on applied it on the 'chickenbroth' images:



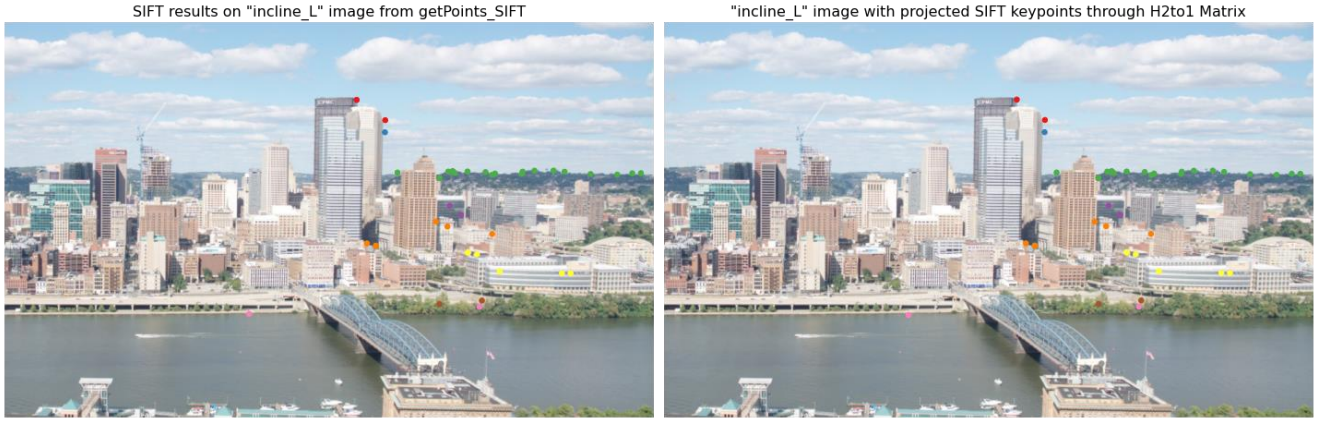
SIFT Keypoints on 'chickenbroth' images.

2. Next, we implemented 'getPoints_SIFT' function using cv2.BFMatcher with an L1 norm criteria. The keypoints are sorted in an ascending order by their distance. The function also receives parameter 'K' that determines how many 'top' keypoints the function should return. The function returns the points in a $2xN$ array for each image.
3. Next, we implemented the 'computeH' function which computes the homography matrix which is the transformation matrix between the images. The implementation followed the mathematical explanation for the section, where each set of corresponding points $p^i = (x_i, y_i, 1), q^i = (u_i, v_i, 1)$ yields two rows in the matrix A that determines the set of linear equations $Ah = 0$. The H matrix is only determined up to scale, such that we have 8 degrees of freedom and therefore need 4 corresponding points. The function is estimating the solution using SVD, by taking the right singular vector of the smallest singular value. The solution vector is reshaped to 3x3 and normalized by the last element (the scale factor) to get the homography matrix H.
   As hinted, we checked if the function returns the identity matrix for an image and itself:

```
Checking if H=I for two identical images ("Chickenbroth3"). The resulted H is:
[[ 1.00000000e+00 -3.00462920e-17 -1.29250301e-13]
 [ 6.69281154e-16  1.00000000e+00  2.77944632e-14]
 [-1.12673595e-18  4.50694380e-18  1.00000000e+00]]
```

computeH 'sanity check'

   To demonstrate the transformation is correct we applied the function on the two 'incline' images.
   First, we'll show the top 40 keypoints for the 'incline_L' image, using getPoint_SIFT and the transformed keypoints from the 'incline_R' image by our H matrix:



The transformed keypoints from the two perspectives:

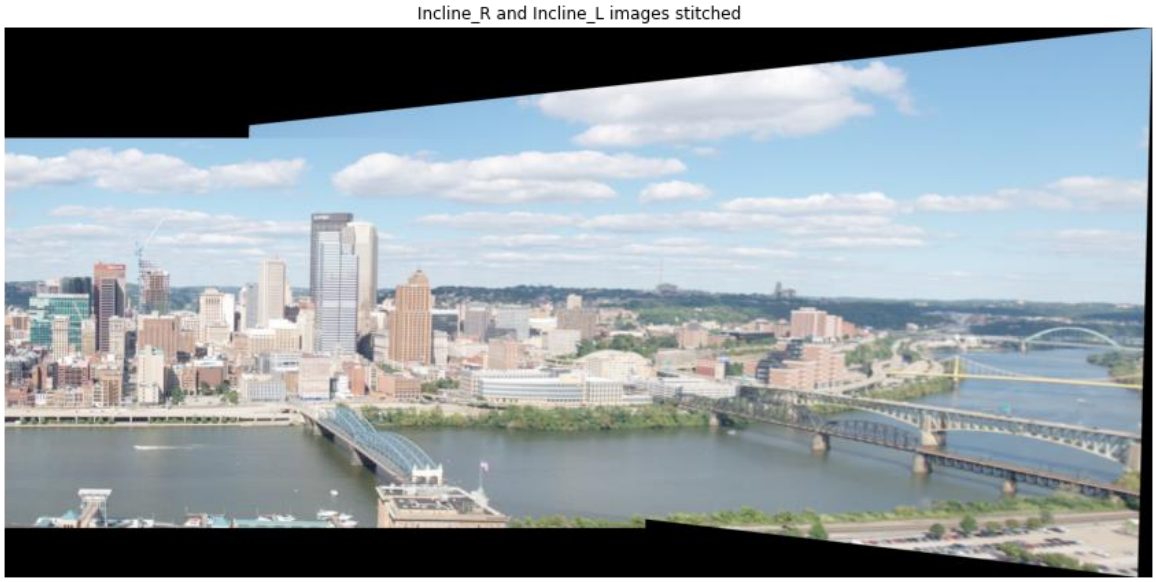
Keypoints transformation check for the incline images.

4. For the image warping part, we implemented another function called 'warp_info' that gets an image and an homography matrix H, and returns the warped output size, the bounding box of the warped image and the shifted homography matrix H such that the bounding box is at the origin, to ensure that the warped image does not have negative coordinates. The implementation is quite simple; only the coordinates of the four corners of the image are transformed by the H matrix, and the new coordinates are used to determine the shape of the size of the warped image and the bounding box. The shifted H matrix is calculated by multiplying it by a translation matrix with the bounding box parameters.
   The warpH function was implemented using inverse warping (to avoid holes) and scipy's interp2d method. The inverse H was calculated using numpy's linalg.inv and the 'input' coordinates were calculated by multiplying the out coordinates by the inverse of H. The values of the image in each 'out' coordinates were calculated using interpolation on the original image in the 'input' coordinates.
   The input 'out_size' and 'H' for warpH function is the output of 'warp_info' function.
   Here are the results of the function and a comparison of the two interpolation kinds (linear and cubic):

Comparison of two interpolation kinds for the 'incline_R' warped image

The interpolation kind determines how the value of a pixel is calculated with respect to its neighbor pixels. Both images are of high quality, the edges preserved well, and the colors are accurate.
Generally, cubic interpolation provides smoother and more accurate results than linear interpolation, but it depends on the nature of the data, and in this case, it hardly makes any difference.

5. In the image stitching part, we made some functions to help with the process. The ‘resize_for_stitching’ function receives the two images to be stitched and creates a blank image with the size of the stitched-to-be image by taking the maximum sum of the relevant dimension sizes of the images. It is then pasting every input image on a different created blank image and returns the result.
   The ‘imageStitching’ function receives the outputs of the ‘resize_for_stitching’ function and takes creates an image which is the maximum value at each point between the two images. That way, the pixels in the intersections between the images will be the maximum value between them, as the maximum value is supposed to represent more relevant information (a similar concept as Max Pooling). In areas where the images don’t intersect, it would be the maximum between zero and the image value which would result in the image value.
   The ’panorama’ function puts it all together – receives two images and computes everything for the image stitching part- Keypoints with SIFT, H matrix, the warped image (using OpenCV’s ‘warpPerspective’ function as requested), the scaled images for the stitching part and finally performs the stitching and returns the stitched image which is the panorama. Here is the result on the incline images:



Incline images panorama

6. We used our ‘panorama’ function to stitch several images. Here is the result on the ‘sintra’ images:

Sintra images panorama

Here is the result on the ‘beach’ images:

Beach images panorama (without RANSAC)

This result isn’t very good (and took a lot of tries to get) due to a lack of ‘good keypoints’ in the images. The whole operation is based on the keypoints extracted from SIFT- since most of the images contain just sand or clear blue sky (with barely one cloud), the algorithm struggles to find good and accurate corresponding points between the images (and instead finds a lot of inaccurate points which we’ll call outliers) and thus the H matrix is inaccurate and so on.

For that matter, the RANSAC algorithm comes into play. This algorithm is used for estimating a robust model in a presence of outliers, by random sampling a set of points, fitting a model to it (in our case - computing H matrix) and counting the number of inliers in the model fitted (In our case - an inlier would be an L2 norm between from image 2 to 1 and an original point in image 1 that is smaller than some tolerance). The algorithm does this over and over and saves the model with the most inliers over all iterations.

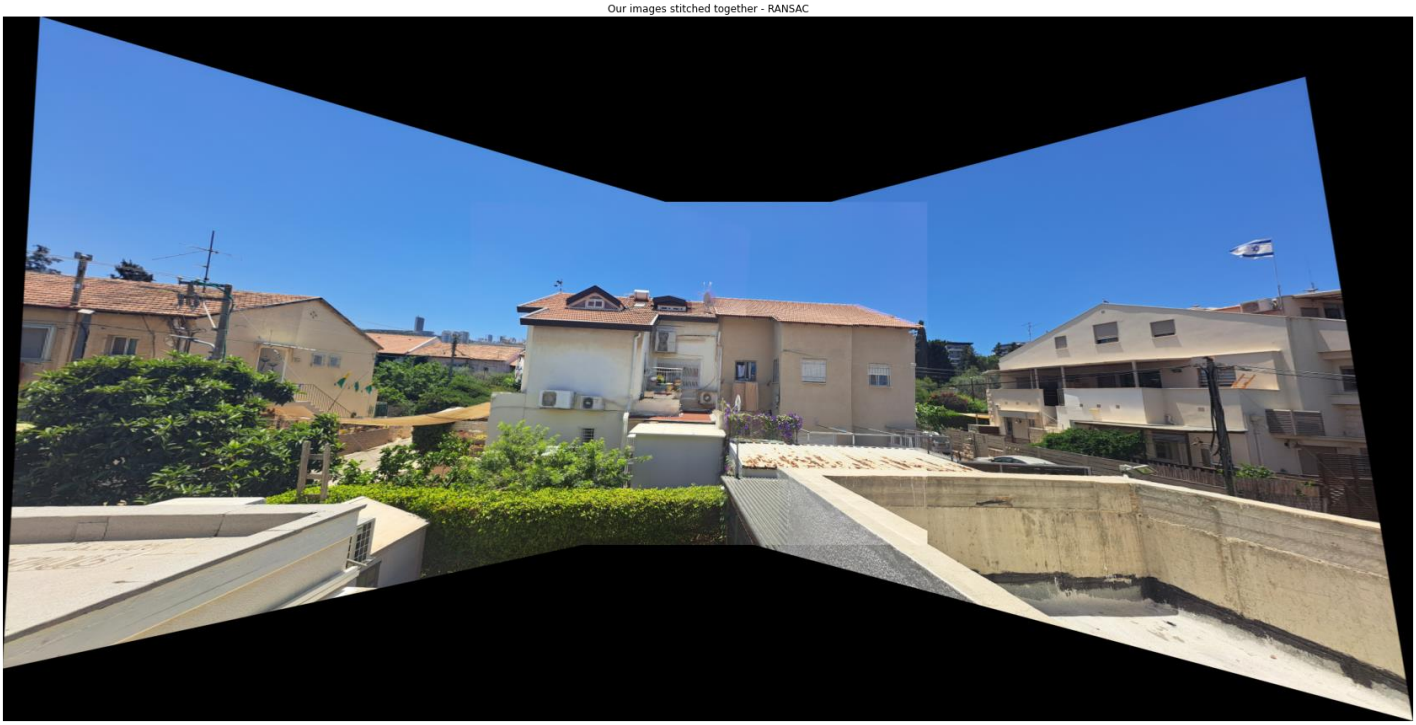
Using RANSAC, the beach images stitching went better (on the first try nonetheless):

Beach images panorama using RANSAC

7. Finally, we took pictures of our own and stitched them together. Here are the pictures:

Nir images from his balcony

And the panorama image (using RANSAC):



Balcony images panorama

Thank You!