Assignment 1:

Homography & Panorama

Due Date: See in Moodle

The problem:

Given a pair of images (src.jpg, dst.jpg) and a file of matching points (matches.mat), we want to make a projective transformation of the source image in order to merge it with the destination image, and build a panorama image.
For motivation for the final result, you may view examples in the link:
http://www.cs.bath.ac.uk/brown/autostitch/autostitch.html

The exercise outline is as follows: We'll start with a system for calculating homography (2D projective transformation) from a list of matching points. Next, we'll add the ability to cope with outliers. Finally, we'll use this system to build a panorama in a (semi) automatic approach.

The exercise will be implemented in python 3.9.5 or higher. We recommend working with conda. We supplied an environment.yml file so you can create the conda environment from this file. You're free to use an IDE of your choice (some people prefer Pycharm over others), but your code needs to run from the terminal. That is, we'll run:

```
python main.py
```

This line should print all panormas as figures and additional data to the terminal.

It is highly recommended to use Linux, and specifically Ubuntu from an LTS version. We will test your code on an Ubuntu 16.04 machine.

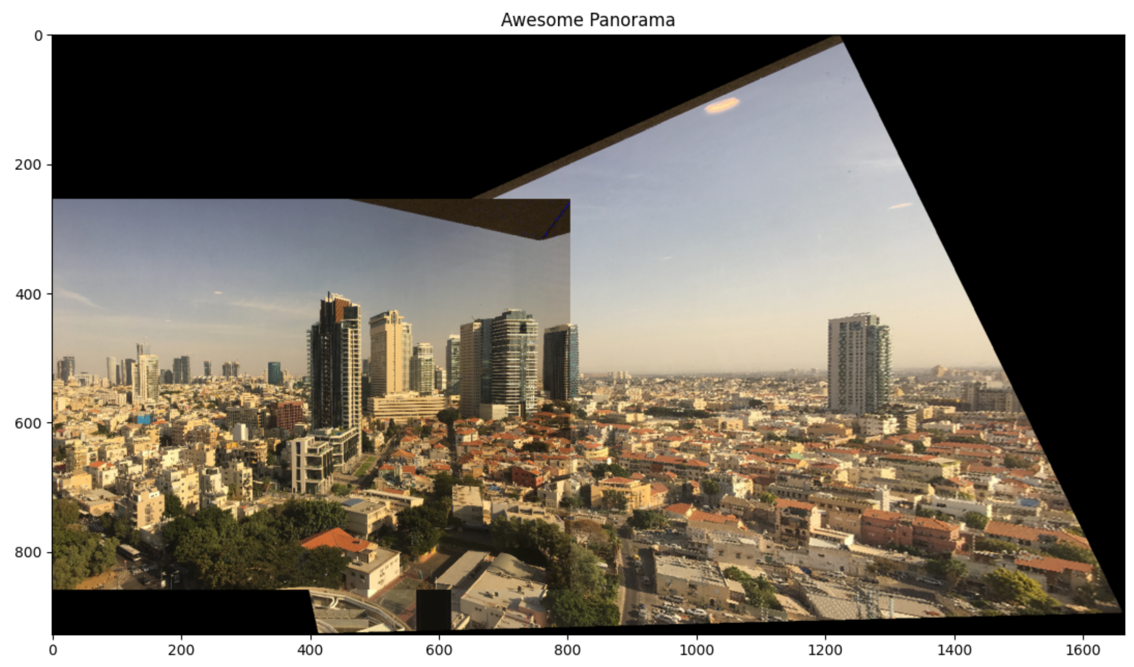
Python packages in this assignment: pillow, numpy, scipy, random, matplotlib, time and opencv. No other additional libraries are allowed.

A few preparatory steps:

- Load the images src.jpg, dst.jpg (appears at the beginning of test_script.py).
- Load the file matches_perfect.mat , notice that this is a dictionary containing two keys: match_p_src, match_p_dst.
- Display the matching points on both images and check if they are indeed a perfect match.
- Similarly, load the file matches.mat, this file contains in addition to correct match points, some mismatching pairs (outliers).
- Display the points on both images and notice the mismatched points.
- *Tip – you may use the function matplotlib.pyplot.scatter to draw points on an image.

The exercise has changed from past years and it is now a “fill your code here” exercise. That means, that in addition to following the exercise as it is specified in this document, you can use the function documentation to your assistance.

We supplied a file named ex1_student_solution.py. This is the only file you can write code to. Don’t use other files.



Part A: Homography computation

1. Build a system of equations of the form $A\underline{x} = \underline{b}$, as learned in class, for **projective** transformation. Attach the formula development to your exercise solution. How do we get the conversion matrix from the equation system?

2. Write a function that estimates the transformation coefficients from source (src) to destination (dst), from the equation system in section 1.

   Use the following API:

def compute_homography_naive(mp_src, mp_dst):

Input:

match_p_src	-	A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the src image.
match_p_dst	-	A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the dst image.

Output:

homography	-	Projective transformation matrix from src to dst.

3. Load the matches_perfect.mat file and calculate the transformation coefficients using the compute_homography_naive function. Present the result.

Part A2: Forward mapping slow and fast:

Our goal here is to compute the src image transformed to the destination using **Forward Mapping**. We will implement a naive solution which will turn out as slow compared to a vectorized fast solution.

4. Implement the transformation function from the source to the destination using the **Forward Mapping** transform. Use the homography from (3) to display the source image after a projective transformation. The result image should be in the size of the dst image. See function documentation for specific implementation details.

   Use the following API:

def compute_forward_homography_slow(homography, src_image, dst_image_shape):

Input:

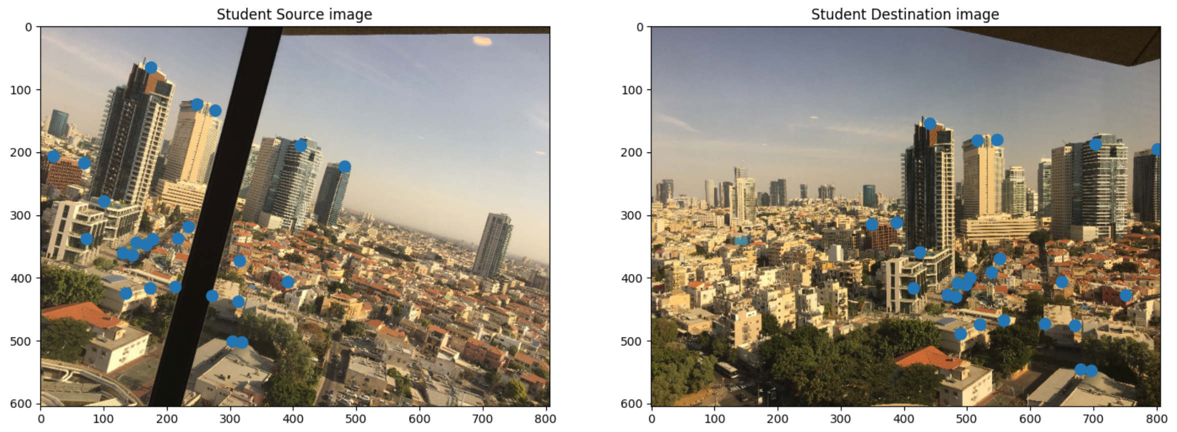
homography	-	Projective transformation matrix from src to dst.
src_image	-	Source image expected to undergo projective transformation.
dst_image_shape	-	Tuple of length 3 indicating the destination image height, width and color dimension. Namely: (H,W,3)

Output:

forward_map	-	The forward homography of the source image to its destination. It should be a numpy array of shape HxWx3.

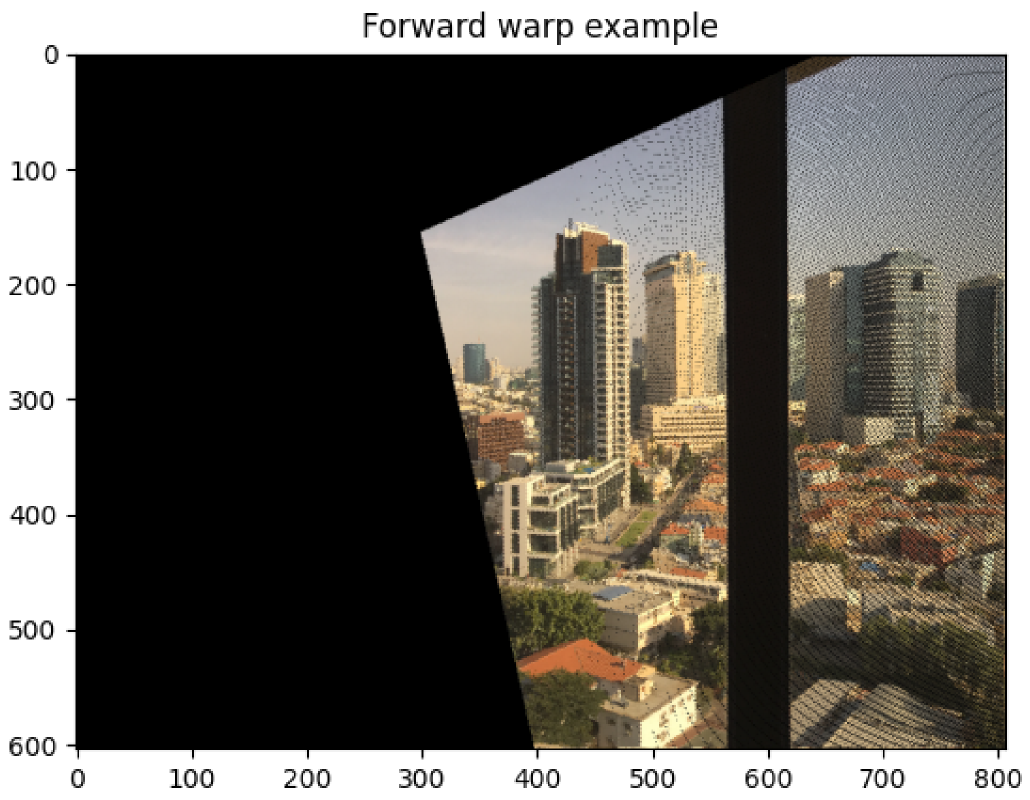
See expected reference solution below.

If the two source images are:



Blue dots are the matching points.

The expected solution here should look something like:



A faster way to compute the forward mapping is to define a meshgrid of the pixel rows and columns indices and use a single matrix multiplication to calculate all the transformed pixel locations at once.

5. Re-Implement the transformation function from the source to the destination using the **Forward Mapping** transform. Don't use loops.

def compute_forward_homography_fast(homography, src_image, dst_image_shape):

Input:

homography	-	Projective transformation matrix from src to dst.
src_image	-	Source image expected to undergo projective transformation.
dst_image_shape	-	Tuple of length 3 indicating the destination image height, width and color dimension. Namely: (H,W,3)

Output:

forward_map	-	The forward homography of the source image to its destination. It should be a numpy array of shape HxWx3.

6. What are the problems with Forward Mapping and how are they reflected in the image you received?

7. Now load the matches.mat file, and repeat items 5 and 6. Did you get a different result? Explain. The image may be too large to display, specify. (You can shrink it and then present it)

Part B: Dealing with outliers

8. Implement a function that calculates the quality of the projective transformation model.

   Use the following API:

def test_homography(homography, match_p_src, match_p_dst, max_err):

Input:

homography	- Projective transformation matrix from src to dst.
match_p_src	- A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the src image.
match_p_dst	- A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the dst image.
max_err	- A scalar that represents the maximum distance (in pixels) between the mapped src point to its corresponding dst point, in order to be considered as valid inlier.

Output:

fit_percent	- The probability (between 0 and 1) validly mapped src points (inliers).
dist_mse	- Mean square error of the distances between validly mapped src points, to their corresponding dst points (only for inliers).

9. Before implementing the function which calculates the source-to-target coefficients that deal with outliers using **RANSAC**, we will implement a helper function which will tell us, given a homography hypothesis, matching point, and a threshold - which points meet the model. That is, which point-pairs are considered as inliers for the given homography under max_err distance.

def meet_the_model_points(homography, match_p_src, match_p_dst, max_err):

Input:

homography	- Projective transformation matrix from src to dst.
match_p_src	- A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the src image.
match_p_dst	- A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the dst image.
max_err	- A scalar that represents the maximum distance (in pixels) between the mapped src point to its corresponding dst point, in order to be considered as valid inlier.

Output:

mp_src_meets_model	- A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the src image, of points which meet the model. That is, when transformed using the homography, you obtain the corresponding point in match_p_dst.
mp_dst_meets_model	- A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the dst image. The dst points correspond to the mp_src_meets_model.

10. Suppose there are 30 match points and it is known that 80% of them are correct. What is the number of randomizations needed in this case to guarantee 90% confidence? Of 99%? How many iterations must be done to cover all options?

11. Implement a function that calculates the source-to-target coefficients that deal with outliers by using **RANSAC** (use the functions you built in previous sections).

    Use the following API:

def compute_homography(match_p_src, match_p_dst, inliers_percent, max_err):

Input:

match_p_src	-	A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the src image.
match_p_dst	-	A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the dst image.
inliers_percent	-	The expected probability (between 0 and 1) of correct match points from the entire list of match points.
max_err	-	A scalar that represents the maximum distance (in pixels) between the mapped src point to its corresponding dst point, in order to be considered as valid inlier.

Output:

homography	-	Projective transformation matrix from src to dst.

12. Load the matches.mat file and calculate the transformation coefficients using the compute_homography function. Present the obtained coefficients, as well as the source image after projective transform using forward mapping. Compare the results you got to the results in sections 5 and 7.

Part C: Panorama creation

13. Implement the transformation function from the source to the destination using the **Backward Mapping** transform, which uses Bi-linear interpolation, and display the source image after a projective transformation, according to the coefficients obtained in section 12. Compare to the image obtained in section 12.

    Use the following API:

def compute_backward_mapping(backward_projective_homography, src_image, dst_image_shape):

Input:

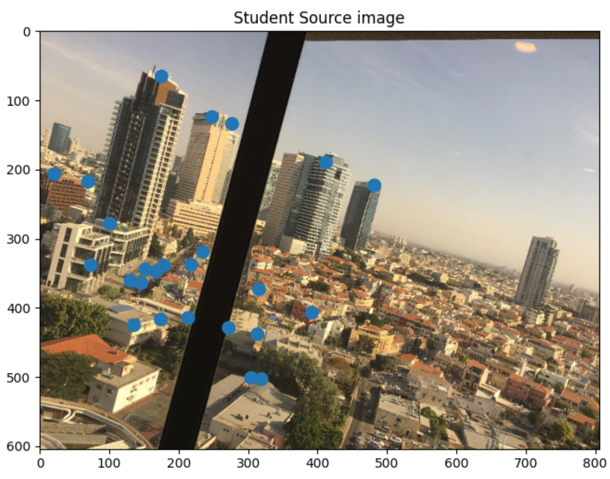
backward_projective_homography	-	Backward Projective transformation matrix. That is the transformation matrix from dst to src.
src_image	-	Source image expected to undergo projective transformation.
dst_image_shape	-	Tuple of length 3 indicating the destination image height, width and color dimension. Namely: (H,W,3)

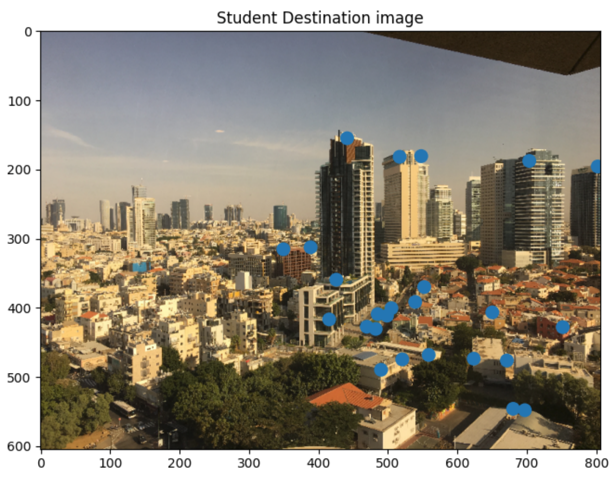
Output:

backward_warp	-	The source image backward warped to the destination coordinates.

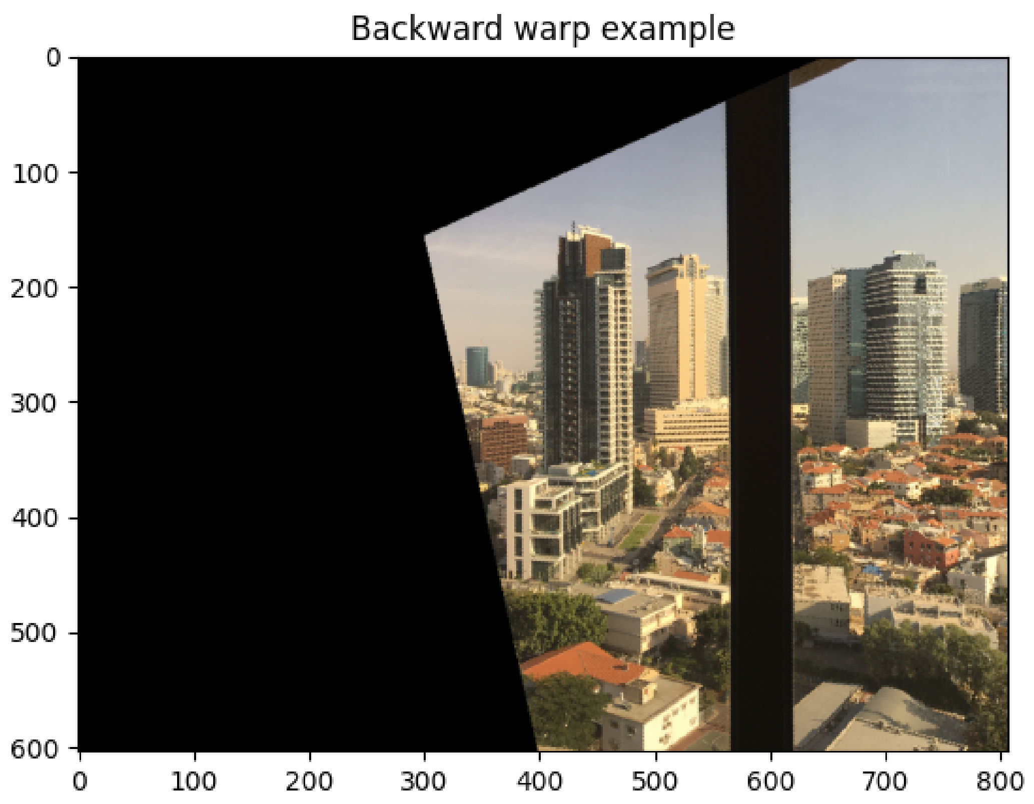
Feel free to use scipy's griddata function. An alternative is to implement the bi-cubic interpolation yourself. It's up to you to choose whatever alternative you want. Please don't use out-of-the-box cv2's function that simply implements the backward warp with some magic. Code your own solution.

A baseline example, to make the result clear:
These two images are the source and destination images:





Then the backward warp looks like that:



Note that a big part of the source image is missing. But hopefully, the artifacts you reported in section 6 are missing. We will take care of the full image alignment next.

14. You are given the method which calculates the panorama's shape. Feel free to debug it, if it misses +-1 row and columns. It returns the number of rows and columns of the final panorama, and another struct which holds the padding (in each axis) done to the target image to achieve the panorama's shape. Use this struct to add a translation component to the homography such that the final homography contains both the translation that the source pixels need to undergo before the homography is applied to them.

    Use the following API:

def add_translation_to_backward_homography(
backward_projective_homography, pad_left, pad_up):

Input:

backward_homography	- Backward Projective transformation matrix. That is the transformation matrix from dst to src.
pad_left	- number of pixels that pad the destination image with zeros from left.
pad_up	- number of pixels that pad the destination image with zeros from the top.

Output:

final_homography	- A new homography which includes the backward homography and the translation.

15. Implement a function that produces a panorama image from two images, and two lists of matching points, that deal with outliers using RANSAC (use the functions from previous sections).

    Use the following API:

def panorama(src_image, dst_image, match_p_src, match_p_dst, inliers_percent, max_err):

Input:

src_image	-	Source image expected to undergo projective transformation.
dst_image	-	Destination image to which the source image is being mapped to.
match_p_src	-	A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the src image.
match_p_dst	-	A variable containing 2 rows and N columns, where the i column represents coordinates of match point i in the dst image.
inliers_percent		The expected probability (between 0 and 1) of correct match points from the entire list of match points.
max_err		A scalar that represents the maximum distance (in pixels) between the mapped src point to its corresponding dst point, in order to be considered as valid inlier.

Output:

img_pan	-	Panorama image built from two input images.

Guidelines:

• Use the Forward Mapping transform on the source image corners to create a bounding rectangle for the output image.

• Use Backward Mapping to perform the transformation.

• For overlapping areas, select pixel values of the destination image.

16. Run the panorama function for the src.jpg and dst.jpg images, using the points from the matches.mat file. Set 80% inliers and a maximum error of 25 pixels. Present the output panorama.

17. Use a pair of your own images to build a panorama (using the panorama function). Images should be called src_test.jpg and dst_test.jpg. Note that it is preferable to use images of the same size.
Run the create_matching_points.py file to produce the matches_test.mat points file with 25 matching points. Make sure there are at least 10% incorrect matching points in the list (outliers). Present the input images, along with the marked matching points, and present the output panorama.

Tips:

- During development and debugging, you can use a down-scaled version of the image. Don't forget to scale the matching points accordingly.
    - This is not mandatory, but it can speed your debugging. Moreover, for too large images, the interpolation might be endless. Work with decent sized images (dimensions such as those of the given images).
    - We've supplied the code to do that for you. You can just change the DECIMATION_FACTOR to say 5.0.
- Images are displayed after conversion to uint8. So for numpy arrays, you can use the astype(numpy.uint8) method:

  numpy_image.astype(numpy.uint8)

- For the matching points script, when you're done marking the matching points. Don't close the figures windows - press any keyboard key on one of the images, and that should close the script gracefully.

<u>Submission instructions:</u>

- A document containing reference to all sections of the exercise must be submitted, showing all the results and answering all questions (no code is required in the document).
- All API-defined python functions of the assignment must be submitted, as well as any associated functions that you wrote (all functions should be in ex1_functions.py). The functions will be automatically run and tested.
- Attach your pair of images (src_test.jpg and dst_test.jpg) as well as your matching points file (matches_test.mat).
- Check that you are able to run the attached **main.py** without making any changes to it. This will only be possible if all the functions you need to write in the exercise appear in ex1_functions.py.
- **The solution with all relevant files must be submitted in the submission box in the Moodle within a zip file named:**

  assignment1_ID1_ <id_1> _ ID2_ <id_2>

  If the zip file exceeds 50MB, you may email it to: amirjevn@mail.tau.ac.il
  The subject of the email should be stated as follows:
  Assignment 1 ID1: <your_id_number> ID2: <your_id_number>
- Late submission will result in grade reduction.

Good Luck!