EE 046746 - Technion - Computer Vision

Homework 3 - Segmentation and Homographies

Due Date: 08.01.2025

Submission Guidelines

READ THIS CAREFULLY

- Submission only in **pairs**.
- **No handwritten submissions**.
- You can choose your working environment:
  - You can work in a `Jupyter Notebook`, locally with Anaconda or online on Google Colab
  - **Important**: Colab also supports running code on GPU, so if you don't have one, Colab is the way to go. To enable GPU on Colab, in the menu: `Runtime` → `Change Runtime Type` → `GPU`.
    - You can work in a Python IDE such as PyCharm or Visual Studio Code.
      - Both also allow opening/editing Jupyter Notebooks.
- Make sure you submit your exercise according to the requirements in the "Homework submission guidelines" file that appears in the course website (Moodle).
- **The code should run both on CPU and GPU without manual modifications**, require no special preparation and run on every computer.
- Be precise, we expect on point answers.
- Submission on the course website (Moodle).
- Bonuses are up to 10 points total (together). Maximum grade for submission is 105.

Python Libraries

- `numpy`
- `matplotlib`
- `pytorch` (and `torchvision`)
- `opencv` (or `scikit-image`)
- `scikit-learn`
- Anything else you need (`PIL`, `os`, `pandas`, `csv`, `json`,...)

Quick note

in this task, you will be required to use Deep learning segmentation methods. for that, you can use any method that has been presented in class, including but not limited to SegmentAnything,Mask-RCNN, and more. if you choose SegmentAnything, use the following link to see an example for usage: link here. Note: if you use SAM, you must in order to use it, download pre-trained weights. please note in the report which model-type you chose, the link to the download. DO NOT include the pre-traiend weights

Installing segment-anything package:

In [ ]:
```
! pip install git+https://github.com/facebookresearch/segment-anything.git
```

Tasks

- In all tasks, you should document your process and results in a report file (which will be saved as `.pdf`).
- You can reference your code in the report file, but no need for actual code in this file, the code is submitted in a seprate folder as explained above.

Part 1 - Classic Vs. Deep Learning-based Semantic Segmentation - bonus

In this part you are going to compare classic methods for segmentation to deep learning-based methods.

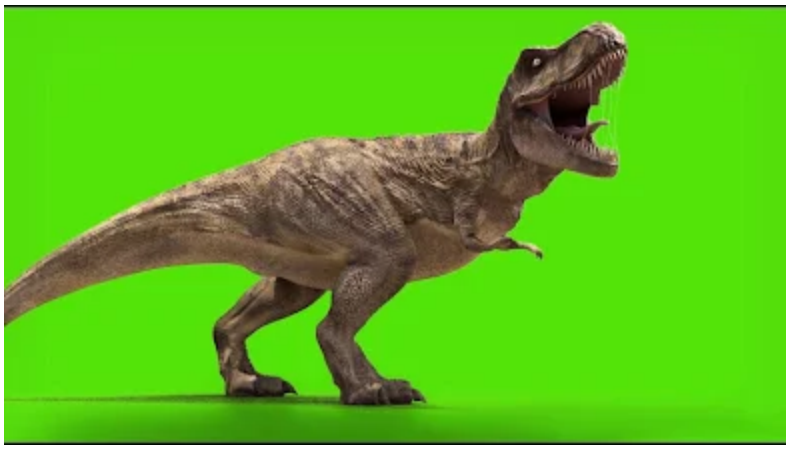
1. Load the images in the `./data/frogs` and `./data/horses` folders and display them.
2. Pick 1 classic method for segmentation and 1 deep learning-based method and segment the given images. Display the results.
    - **Briefly** summarize each method you picked and discuss the advantages and disadvantages of each method. In your answer, relate to the results you received in this section.

- You can use a ready implementation from the internet or OpenCV, no need to implement it yourselves.
- Note: the classic method **must not** use any neural network.

3. Pick 3 images (download from the internet or take them yourself) that satisfy the following, and dispaly them:
   - One image of a living being (human, animal,...).
   - One image of commonly-used object (car, chair, smartphone, glasses,...).
   - One image of not-so-commonly-used object (fire extinguisher, satellite,... **BE CREATIVE**).
4. Apply each method (one classic and one deep learning-based) on the 3 images. Display the results (mask and segmented image).
   - Which method performed better on each image? Describe your thoughts on why one method is better than the other.
   - For the classic method you can change parameters per-image, document them in the report.
   - You can add manual post-processing to get a mask if needed. If you do that, document in your report "how hard" you had to work in the post-processing stage, as it's an indication of the quality of the method.
5. As you probably have noticed, segmentation can be rough around the edges, i.e., the mask is not perfect and may be noisy around the edges. What can be done to fix or at least alleviate this problem? Your suggestions can be in pre-processing, inside the segmentation algorithm or in post-processing.

Part 2 - Jurrasic Fishbach - bonus

In this part you are going to apply segmentation on a video, and integrate with other elements.



1. Film a short video of yourself (you can use your phone for that), but without too much camera movement. You on the other hand, can move however you want (we expect you to). Convert the video to frames and resize the images for a reasonable not too high

resolution (lower than 720p ~ 1280x720 pixles). You can use the function in `./code/frame_video_convert.py` to help you. Display 2 frames in the report.
2. Segment yourself out of the video (frame-by-frame) using one of the methods (classic or deep). Display 2 frames in the report.
3. Pick one of the objects in the supplied videos file (`./data/video_models`), convert it to images and segement it out using one of the methods from Part 1(classic or deep). Display 2 frames in the report. You can choose another object from: https://pixabay.com/videos/search/green%20screen/.
    - Explain how you performed the sementation for this specific type of video (i.e., green-screen videos). Did you use a simple/classic method? Deep method? Combined both?
4. Put it all together - pick a background, put yourself and the segemented object on the background. Stich it frame-by-frame (don't make the video too long or it will take a lot of time, 10secs maximum). Display 2 frames of the result in your report. Convert the frames back to video. You can use the function in `frame_video_convert.py` to help you.
    - Tip: To make it look good, you can resize the images, create a mapping from pixel locations in the original image to pixels locations in the new image.
    - You should submit the final video in the `./output` folder (**MANDATORY**).
    - We expect some creative results, this can benefit you a lot when you want to demonstrate your Computer Vision abilities.

Part 3 - Planar Homographies - mandatory :

After we saw how descriptors are implemented and performed, now we will see how to use them for homographis.

In this part you will implement an image stitching algorithm, and will learn how to stitch several images of the same scene into a panorama. First, we'll concentrate on the case of two images and then extend to several images.

For the following tasks:

- **You are not allowed to use OpenCV/Scipy or any other "ready to use" functions when you are asked to implement a function (you can still use the functions to save and load images).**
- For each step add illustration images to your report.
- You can demonstrate your steps using `incline_L.jpg` and `incline_R.jpg` images, or any other relevant example images (unless specified otherwise).

Planar Homographies: Theory review

Suppose we have two cameras C_1 and C_2 looking at a common plane Π in 3D space. Any 3D point P on Π generates a projected 2D point located at $p \equiv (x, y, 1)^T$ on the first camera C_1 and $q \equiv (u, v, 1)^T$ on the second camera C_2. Since P is confined to the plane Π, we expect that there is a relationship between p and q. In particular, there exists a common 3×3 matrix H, so that for any P, the following conditions holds:

$$(1)\ q \equiv Hp \tag{1}$$

We call this relationship '*planar homography*'. Recall that both p and q are in homogeneous coordinates and the equality $\equiv$ means p is proportional to Hq (recall homogeneous coordinates). It turns out this relationship is also true for cameras that are related by pure rotation without the planar constraint.

Matched points:

Given a set of points $p = \{p_1, p_2, \ldots, p_N\}$ in an image taken by camera C_1 and corresponding points $q = \{q_1, q_2, \ldots, q_N\}$ in an image taken by C_2. Suppose we know there exists an unknown homography H between corresponding points for all $i \in \{1, 2, \ldots, N\}$. This formally means that $\exists H$ such that:

$$(2)\ q^i \equiv Hp^i$$

where $p^i = (x_i, y_i, 1)$ and $q^i = (u_i, v_i, 1)$ are homogeneous coordinates of image points each from an image taken with C_1 and C_2 respectively.

- Given N correspondences in p and q and using Equation 2, we derived a set of $2N$ independent linear equations in the form:

$$(3)\ Ah = 0$$

where h is a vector of the elements of H and A is a matrix composed of elements derived from the point coordinates:

$$\begin{bmatrix} & & & & & \cdots & & & \\ x_i & y_i & 1 & 0 & 0 & 0 & -x_iu_i & -y_iu_i & -u_i \\ 0 & 0 & 0 & x_i & y_i & 1 & -x_iv_i & -y_iv_i & -v_i \\ & & & & & \cdots & & & \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix} = \begin{bmatrix} \cdots \\ 0 \\ 0 \\ \cdots \end{bmatrix}$$

Each point pair contributes 2 equations and therefore we need at least 4 matches.

Planar Homographies: Practice

Quick detour- Feature Descriptor

In this part we are going to use **Scale-Invariant Feature Transform** (SIFT). We will use OpenCV [2] for the implementation.

3.0.1 SIFT Implementaion

Implement the following function:

In [ ]:
```
def SIFT_descriptor(img):
 # Returns the SIFT descriptor keypoints of an image,and draw its detected keypoint
 # INPUTS
 #      img                - An image read by cv2.imread()
 # OUTPUTS
 #      sift_descriptor    - The SIFT descrptor computed by OpenCV
 #      sift_keypoints     - The SIFT keypoints computed by OpenCV
  """
  Your code here
  """
  return sift_keypoints, sift_descriptor
```

The function get an image ,returns its SIFT descriptor and keypoints, and draw the detected keypoints over the image.

Plot the results obtained for the `model_chickenbroth` image and another `chickenbroth` image (Use the same image from previous section).

- Implementaton guidance:
  - Use `sift = cv2.xfeatures2d.SIFT_create()` to instantiate the SIFT detector.
  - Detect and compute SIFT keypoints and descriptors by `sift_keypoint, sift_descriptor = sift.detectAndCompute(img,None)`.
  - Draw the keypoints over the image by using `cv2.drawKeypoints()`.

3.1 - Finding corresponding points using SIFT:

Use the guidelines from 3.0.1 and implement the function `getPoints_SIFT()`, which gets two images and outputs `p1,p2` SIFT keypoints, where `p1[j],p2[j]` are pairs of cooresponding points between `im1` and `im2`.

In [ ]:
```
def getPoints_SIFT(im1, im2):
    """
    Your code here
```

```
    """
    return p1,p2
```

Inputs: `im1` and `im2` are two 2D grayscale images.

Output: `p1` and `p2` should be $2 \times N$ matrices of corresponding $(x, y)^T$ coordinates between two images (N is the number of corrosponding points you want to extract.).

- You can also use color images instead of grayscale images, just state it in the report.

3.2 - Calculate transformation:

Implement a function that gets a set of matching points between two images and calculates the transformation between them. The transformation should be 3×3 H homogenous matrix such that for each point in image $p \in C_1$, there would be a transformation in image C_2 such that $p = Hq$, $q \in C_2$.

In [ ]:
```
def computeH(p1, p2):
    """
    Your code here
    """
    return H2to1
```

Inputs: `p1` and `p2` should be $2 \times N$ matrices of corresponding $(x, y)^T$ coordinates between two images.

Outputs: `H2to1` should be a 3×3 matrix encoding the homography that best matches the linear equation derived above for Equation 2.

Hint: Remember that a homography is only determined up to scale. The `numpy`'s functions `eig()` or `svd()` will be useful. Note that this function can be written without an explicit for-loop over the data points.

Hint for debugging: A good test of your code is to check that the homography of an image with itself is an identity.

- Implement the computation function, describe and explain your implementation.
- Show that the transformation is correct by selecting arbitrary points in the first image and projecting them to the second image.

3.3 - Image warping:

Implement a function that gets an input image and a transformation matrix H and returns the warped image. Please note that after the warping, there will be coordinates that won't be integers (e.g. sub-pixels). Therefore you will need to interpolate between neighboring pixels.

For color images, warp the image for each color channel and then connect them together. In order to avoid holes, use inverse warping.

Implement the wrapping function using numpy and SciPy interp2d() or RegularGridInterpolator() function . Discuss the influences of different interpolations kinds {'linear', 'cubic'}. Note: When performing a multi-step algorithm, you need to demonstrate and explain each of those additional improvments.

In [ ]:
```
def warpH(im1, H, out_size):
    """
    Your code here
    """
    return warp_im1
```

Inputs: `im1` is a colored image. `H` is a matrix encoding the homography between im1 and im2. `out_size` is the size of the wanted output (new_imH,new_imW).

Output: `warp_im1` is the warped image im1 including empty background (zeros).

3.4 - Panorama stitching:

Implement a function that gets two images after axis alignment (using OpenCV's `warpPerspective`) and returns a union of the two. The union should be a simple overlay of one image on the other. Leave empty pixels painted black.

In [ ]:
```
def imageStitching(img1, warp_img2):
    """
    Your code here
    """
    return panoImg
```

Inputs: `im1` , `warp_img2` are two colored images.

Output: `panoImg` is the gathered output panorama.

- Use all the above functions to create a panorama image. Demonstrate and explain you results on the `./data/incline` images.

3.5 - Several Images stitching:

- Show the results of the panoramas on the attached images of the beach (`./data/beach`) and Pena National Sintra Palace (`./data/sintra`) for the entire set of images.
- Note: When using SIFT without RANSAC (next section), take the top K matches for estimating the homography.

- What happens if you don't do so? Why is that?

3.6 - RANSAC:

Added bellow is an implementation of the RANSAC (**Ran**dom **Sa**mple **C**onsensus) algorithm.

- Explain when it is needed and why.
- Copmare between using RANSAC vs. not using it for creating the panoran images of the beach and SINTRA. Explain.
- What could have been done to get better results?

In [ ]:
```python
def ransacH(p1, p2, nIter=..., tol=...):
    N = p1.shape[1]
    stacked_p2 = np.vstack((p2, np.ones(N)))

    best_inliers_n = 0
    best_inliers = []

    for iter in range(nIter):
        rand_idxs = np.random.choice(np.arange(N), 4, replace=False)
        chosen_p1 = p1[:, rand_idxs]
        chosen_p2 = p2[:, rand_idxs]
        H2to1 = computeH(chosen_p1, chosen_p2)
        p2in1 = H2to1 @ stacked_p2
        p2in1 = p2in1 / p2in1[2, :]
        p2in1 = p2in1[0:2, :]
        L2dists = np.sqrt(np.sum((p2in1 - p1) ** 2, 0))
        inliers = (p1[:, L2dists < tol], p2[:, L2dists < tol])
        n_inliers = np.sum(L2dists < tol)
        if n_inliers > best_inliers_n:
            best_inliers_n = n_inliers
            best_inliers = inliers

    bestH = computeH(best_inliers[0], best_inliers[1])
    return bestH
```

Inputs:

- `p1` and `p2` are matrices specifying point locations in each of the images and `p1[j]` , `p2[j]` are matched points between two images.
- `nIter` is the number of iterations to run RANSAC
- `tol` is the tolerance value for considering a point to be an inlier.
- Define your function so that these `nIter` and `tol` have reasonable default values.

Outputs:

- `bestH` is the homography model with the most inliers found during RANSAC

3.7 - Be Creative:

- Go out and take at least 3 pictures of a far distance object (e.g. a building), and use what you have learned to create a new excellent Panorama image.
    - Add the resulted image to your report and to the `output` folder.

Credits

- Images from Imagenet
- Videos from Pixabay
    - Dinosaur video from Modern Media
- Icons from Icon8.com - https://icons8.com