EE 046746 - Technion - Computer Vision

Homework 3 - Homographies and Panorama Stitching

Due Date: 04.01.2022

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
    - 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` $\rightarrow$ `Change Runtime Type` $\rightarrow$ `GPU`.
  - You can work in a Python IDE such as PyCharm or Visual Studio Code.
    - Both also allow opening/editing Jupyter Notebooks.
- You should submit two **separated** files:
  - A compressed `.zip` file, with the name: `ee046746_hw3_id1_id2.zip` which contains:
    - A folder named `code` with all the code files inside (`.py` or `.ipynb` ONLY!), and all the files required for the code to run (your own images/videos) inside `my_data` folder.
      - **The code should run both on CPU and GPU without manual modifications**, require no special preparation and run on every computer.
    - A folder named `output` with all the output files that are not required to run the code. This includes videos and visualizations that are not included in your PDF report.
  - A report file (visualizations, discussing the results and answering the questions) in a `.pdf` format, with the name `ee046746_hw3_id1_id2.pdf`.
    - Be precise, we expect on point answers. **But don't be afraid to explain you statements (actually, we expect you to).**
      - Even if the instructions says "Show...", you still need to explain what are you showing and what can be seen.
  - No other file-types (`.docx`, `.html`, ...) will be accepted.
- Submission on the course website (Moodle).

Python Libraries

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

In [ ]:

```python
import numpy as np
import matplotlib.pyplot as plt
import cv2
import pickle
import scipy
from matplotlib import pyplot as plt
%matplotlib inline
```

Important Tips

- You can resize the images to a lower resolution if the computation takes too long.
- You can add parameters or outputs to the functions if you need.

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.

Introduction

In this homework, we will explore the homography between images based on the locations of the matched features (remember hw1?). Specifically, we will look at the planar homographies. Why is this useful? In many robotics applications, robots must often deal with tabletops, ground, and walls among other flat planar surfaces. When two cameras observe a plane, there exists a relationship between the captured images. This relationship is defined by a 3×3 transformation matrix, called a planar homography. A planar homography allows us to compute how a planar scene would look from a second camera location, given only the first camera image. In fact, we can compute how images of the planes will look like from any camera at any location without knowing any internal camera parameters and without actually taking the pictures, all using the planar homography matrix.

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 $p \equiv (u_1, v_1, 1)^T$ on the first camera C_1 and $q \equiv (u_2, v_2, 1)^T$ 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 $p = \{p_1, p_2, \dots, p_N\}$ in an image taken by camera C_1 and corresponding points $q = \{q_1, q_2, \dots, q_N\}$ in an image taken by C_2. Suppose we know there exists an unknown homography H between corresponding points for all $i \in \{1, 2, \dots, N\}$. 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:

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

- Where each point pair contributes 2 equations and therefore we need at least 4 matches.

Part 1 - Planar Homographies: Practice

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).**
- Add all your implemented functions for this section into `my_homography.py` under the `code` folder.
- Make sure you answer/demostrate all the "bullet" questions in your report
- For each step add illustration images to your report.
- You can demonstrate your steps using `/code/data/incline_L.jpg` and `/code/data/incline_R.jpg` images, or any other relevant example images (unless specified otherwise).

1.1 - Manually finding corresponding points

Implement a function that enables manual matching of corresponding image points between two images. You will probably want to use the function `ginput()` from `matplotlib`. The success of the registration depends on the accuracy of your match, so mark it carefully. (It might be difficult to make `ginput()` work in a `Jupyter Notebook`. If you use `Pycharm` make sure your `Show plot in tool window` checkbox is off, you can find it under `Python Scientific` in the setting menu).

In [ ]:
```python
def getPoints(im1, im2, N):
    """
    Your code here
    """
    return p1,p2
```

Inputs: `im1` and `im2` are two 2D grayscale images. `N` is the number of corrosponding points you want to extract. Output: `p1` and `p2` should be $2 \times N$ matrices of corresponding $(x, y)^T$ coordinates between two images.

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

1.2 - 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 [ ]:
```python
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 H 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.

1.3 - Image warping:

Read about OpenCV's `warpPerspective`.

- Does it implement a forward or inverse warping? Why?
- Warp the incline image with your computed H using different interpolations kinds {`‘linear’`, `‘cubic’`}, using `cv2.warpPerspective`. Leave the background empty (Zeros). Discuss the differences.
- Make sure that the entire image is being shown after the warp (and it isn't being cut).
  - Tip: You should take notice of the output size of the image.
  - Since you're going to warp more pictures afterwards, you should probably arrange all of the supporting code to the warping in functions.

1.4 - 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 output gathered panorama.

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

1.5 - Autonomous panorama stitching using SIFT:

In this section, we are going to use SIFT to stitch images. Read https://docs.opencv.org/3.4/da/df5/tutorial_py_sift_intro.html to learn how to use OpenCV to

extract SIFT keypoints and descriptors. To match the resulting points and get correspondences you can use the following tutorial:
https://www.docs.opencv.org/master/dc/dc3/tutorial_py_matcher.html.

- Implement the `getPoints_SIFT()` function, 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
```

- Choose a descriptors matching algorithm, and explain how it works (e.g. KNN).
- Use the SIFT and the matching algorithm to perform image stitching over the `code/data/incline` images.
- If you're using an openCV version where `SIFT_create()` is located differently than in the tutorial, state it in the report.

1.6 - Compare SIFT and Manuale image selection:

- Show the results of *manual* and *SIFT* panoramas on the attached images of the beach and Pena National Palace (Portugal) for the entire set of images.
  - Compare the two methods (pros/cons), and explain phenomenas and irregularities you see in the panoramas.
  - For the manual method, we've prepared 1 set of manual points for each photo in the file `code/data/points.pkl`. You can load the file using the **pickle** library (`pickle.load()`). The object is built as follows:
```
obj = {'sintra': {'1-2': [np.array of points from sintra1.jpg,
                          np.array of the corresponding points pairs
from sintra2.jpg],
                  '2-3': [np.array of points from sintra2.jpg,
                          np.array of the corresponding points pairs
from sintra3.jpg],
                  ...},
       'beach': {'1-2': ...}
      }
```
    You can also of course manually click on the photos yourselves with the function you created in 1.1 if you choose to do so.
- 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?

1.7 - 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 both manual and SIFT features. 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.

Algorithm Input Parameters: `nIter` is the number of iterations to run RANSAC for, `tol` is the tolerance value for considering a point to be an inlier. Define your function so that these two parameters have reasonable default values.

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

1.8 - 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.
- If you're super excited about the panorama you created, want to show off, and have spare time - you might also want to look at tutorial 7 to see how to blend images, but that's just for fun.

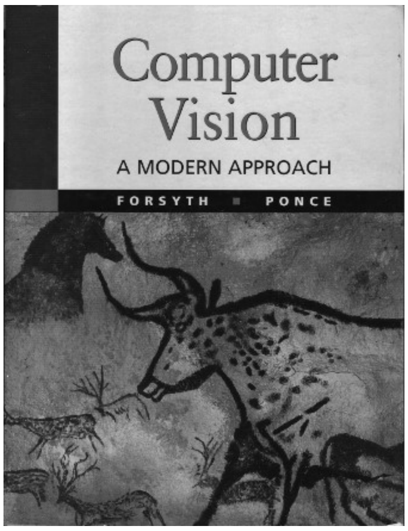
Part 2 - Creating your Augmented Reality application

Now with the code you have, you can create your own Augmented Reality application.

- Add all your implemented functions for this section into `my_ar.py` under the `code` folder.

2.1 Create reference model

Take a picture of your favourite book front cover (place it within `/code/my_data` folder). Make sure the book is not occluded, and the image resolution is reasonable. We want to convert your book to a model template. Use the functions you have created in the last section to warp the front cover into the XY plane. Use your functions to get the book's corners and warp them to a reasonable rectangle. The results should look like the following example:



In [ ]:
```python
def create_ref(im_path):
    """
       Your code here
    """
    return ref_image
```

`create_ref` gets the original image path and outputs a ref image.

- Add the resulted image to your report and to `code/my_data` folder.

2.2 Implant an image inside another image

Take another picture of the same book, this time plant the book somewhere in the scene. Now, take an image of another book (or something else) and stitch it within the first image, instead of the book, using planar homography.

- Add your function to `my_ar.py` file as `im2im()` (declare yourself the input and outputs).
- Add at least 3 examples to your `output` folder as `im2imx.jpg` where `x` is the example number.
    - i.e. Plant book A in scene K, plant book B in scene K, plant book C in scene K.
- Add one of the examples to your report.
- Explain all your steps.

References & Credits

- Carnegie Mellon University - CMU
- Icons from Icon8.com - https://icons8.com