MIT EECS 6.815/6.865: Assignment 6:

Homographies and Manual Panoramas

Due Monday March 19 at 9pm

1 Summary

- Warp an image according to a homography
- Compute a homography given four pairs of points
- Create a fun composite
- 6.865 only: use SVD to compute the homography, for extra robustness
- 6.865 only: least square fit using more than 4 pairs
- Compute the bounding box of the resulting merged image

Be careful. You will need to reuse this code for pset 7 and 8 when we will implement fully automatic panorama stitching and blending.

Homogenous coordinates

We will use homogenous coordinates to deal with the perspective transforms involved in panorama stitching. We also use the y, x order rather than the more mathematically common x, y one so that image indexing is simpler. This is different from pset 2.

A Euclidean point $(y, x)^T$ is represented by the family of homogenous coordinates $(yw, xw, w)^T$ for any non-zero scalar w.

2 Warp by a homography

This section is a minor variation over problem set number 2. You should be able to use the same code structure with loops over the output pixels, with a different warp function and a minor twist for pixels outside the source image.

Write a function `applyhomography(source, out, H, bilinear=False)` that modifies an image `out` to composite the `source` image warped according to the 3×3 homography matrix `H`. If the Boolean `bilinear` is True, use bilinear reconstruction, otherwise use simple nearest neighbor.

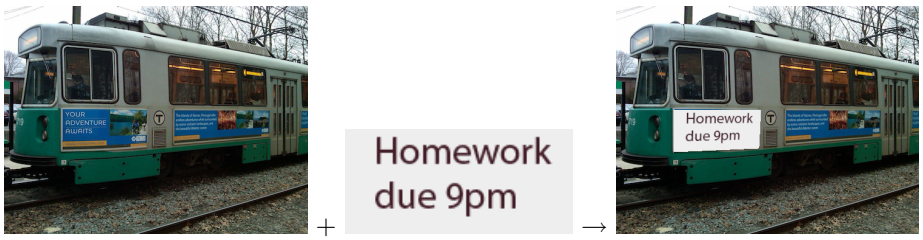
`H` is the matrix going from the output coordinates to the source coordinates. You apply a homography matrix to a 2D point y, x by computing the product

$H \begin{pmatrix} y \\ x \\ 1 \end{pmatrix} = \begin{pmatrix} y' \\ x' \\ w' \end{pmatrix}$ and dividing by the last coordinate to yield the Euclidean coordinates $\begin{pmatrix} y'/w' \\ x'/w' \end{pmatrix}$.

The function does not return anything, it just modifies `out`.

We will treat pixels coordinates outside the source image differently from previous problem sets. For compositing, we want to leave the corresponding output pixels untouched. For this, you need to test if the warped coordinates are outside the source image before updating the pixel value. This scheme might lead to stair artifacts at the boundary but we won't worry about it. In a later assignment, we will perform nice feathering at the boundaries between images.

Test your function on the provided `green.png` and `poster.png` images using the homography: `H=array([[ 0.8346, -0.0058, -141.3292], [ 0.0116, 0.8025, -78.2148], [ -0.0002, -0.0006, 1. ]] )`



3 Solve for a homography from four pairs of points

In this section, a user will provide correspondences between two images, by clicking using a patent-pending javascript interface, and your job is to infer the homography matrix that maps the points from the first image into those in the second one.

The correspondences will be provided as a list of pairs of points, where each point is encoded as an array of size 2 that contains the y and x coordinates of a point. The order of coordinates is y, x, unlike problem set 2, which should make your life easy. Indexing will be according to `listOfPairs[pair_index][image][coordinate]`. This means that `listOfPairs[2][0][0]` contains the y coordinate in the first image for the third pair. Its y coordinate in the second image is `listOfPairs[2][1][0]`.

Like for problem set 2, you need to click on points in the same order for the left and right images.

Write a function `computehomography(listOfPairs)` that takes a list of pairs of points and return a 3×3 homography matrix that maps the first point of each pair to the second one. That is, the homography maps from the first image to the second one.

A homography matrix is always defined up to scale. This means that H and kH represent the same geometric transformation for any non-zero scalar k. In this assignment, we will use a quick and dirty way to resolve the scale ambiguity and will assume that the bottom right coefficient of the homography is 1. This could create some problem if the true family of matrices H had a zero there but we will ignore this and leave the cleaner solution based on SVD as extra credit.

You need to create a matrix A and vector b so that the 8 remaining coefficients of the homography are encoded in an 8-dimensional vector x that satisfies $Ax = b$.

Start from the homography correspondence equation:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ h & g & 1 \end{pmatrix} \begin{pmatrix} y \\ x \\ 1 \end{pmatrix} = \begin{pmatrix} w'y' \\ w'x' \\ w' \end{pmatrix}$$

Where the unknowns are a, b, c, d, e, f, g and y, x and y', x' are known. w' is not known per say but can be deduced easily from other unknowns and known quantities. You have four such pairs of equation for each pair of corresponding points. To be clear, the x vector in $Ax = b$ is actually $(a, b, c, d, e, f, g)^T$. Do not be confused between the 3×3 homography matrix and the 8×8 matrix A for the linear system.

I recommend that you write a subroutine that fills in two rows of the matrix and vector for a given correspondence pair, since the pattern is the same for all pairs of points.

Use `linalg.inv` to compute the inverse of matrix A and apply it to b using `dot`.

One you have solved the system, you need to turn your vector x into a 3×3 matrix. See below for a little trick that might make it easier.

Alternative If you prefer, you can create a 9×9 system where the last line trivially says that the last unknown is equal to 1. It makes it slightly easier to turn the resulting `x` back into a matrix using `numpy.reshape`.

Debugging homographies and projective spaces are big scary things. Do not try directly with the provided pairs of points. Use simpler cases. Think about pairs of correspondences where you can compute the matrix easily by hand. Try to infer the identity or a translation.

Test Test your function on the same example as above, using the following pairs:

```
src=imread('pano/poster.png')
h, w = src.shape[0]-1, src.shape[1]-1
pointListPoster=[array([0, 0, 1]), array([0, w, 1]), array([h, w, 1]),
array([h, 0, 1])]
pointListT=[array([170, 95, 1]), array([171, 238, 1]), array([233,
```

```
235, 1]), array([239, 94, 1])]
listOfPairs==zip(pointListPoster, pointListT)
```

and on the pair of images from the Stata center:

```
im1=imread('pano/stata-1.png')
im2=imread('pano/stata-2.png')
pointList1=[array([209, 218, 1]), array([425, 300, 1]), array([209,
337, 1]), array([396, 336, 1])]
pointList2=[array([232, 4, 1]), array([465, 62, 1]), array([247, 125,
1]), array([433, 102, 1])]
listOfPairs=zip(pointList1, pointList2)
H=computehomography(listOfPairs)
out=im1*0.2
applyhomography(im2, out, H, True)
imwrite(out)
```

The multiplication by 0.2 is here to better show the transition. Note how well the two images match. However, we need to enlarge our canvas to make room for the extra portion of the scene in the second image. We'll do this in the next section.

3.1 Have fun!

Use your code to create a fun compositing with your own source and target in the spirit of the T example.

4 Bounding boxes

In the examples so far, the output image is the same size as one of the two inputs. But for panorama stitching, we want to create a bigger image, as shown by our Stata example. For this, we need to estimate the size of the output.

The output coordinate system will mostly be that of one of the images. But the final image might be extended towards the negative coordinates of the

reference, which will force us to translate the reference, as discussed below.

4.1 Transform a bounding box

To compute the required size for one image, we transform its four corners into the output space. Be careful: whereas we usually need to consider the inverse warp to go from output coordinates into source ones, here we need to do the opposite and figure out where the bounds of the source project into the output space. You might need to invert your homography matrix.

We will represent the resulting required size as a bounding box, encoded by its min and max y, x coordinates.

Write a function `computeTransformedBBox(im, H)` that takes an image and a homography as input and returns a 2D array storing [[ymin, xmin], [ymax, xmax]].

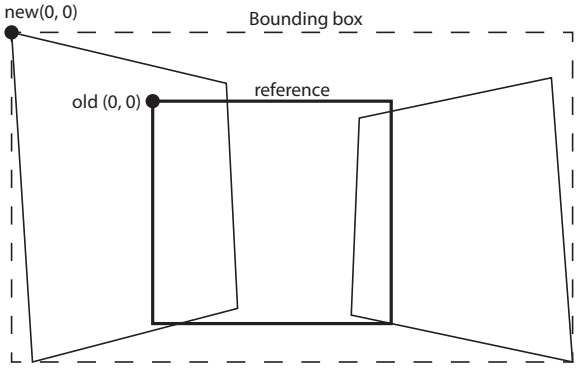
Try it on the second image of the Stata example and see if it qualitatively makes sense. Try it on simpler matrices as well.

4.2 Bounding box union

When stitching N images, the output is the union of the N bounding boxes. Write a function `bboxUnion(B1, B2)` that takes two bounding boxes represented like above as an array and returns the union of the bounding boxes, with the same representation. Bounding box algebra is pretty simple and you should figure it out.

4.3 Translation

After computing and merging the bounding boxes of the N images (including that of the reference), we have the following situation:



If the upper left corner of the bounding box has negative coordinates, we need to translate the coordinate system of the reference image to set this corner to coordinates $(0, 0)$. The translation vector is simply the negative of this corner's coordinates. You can then obtain the translation homography matrix

for translation by $(t_y t_x)^T$ as:

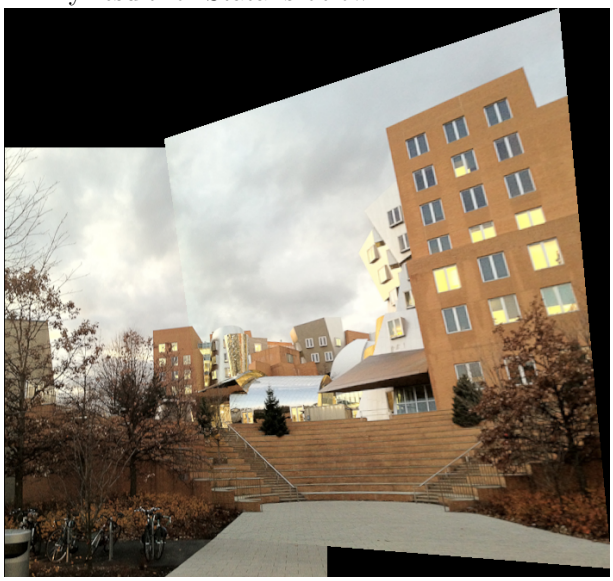
$$\begin{pmatrix} 1 & 0 & t_y \\ 0 & 1 & t_x \\ 0 & 0 & 1 \end{pmatrix}$$

write a function `translate(bbox)` that takes a bounding box as input and returns a 3×3 matrix corresponding to a translation that moves the upper left corner of the bounding box to $(0, 0)$.

4.4 Putting it all together

Write a function `stitch(im1, im2, listOfPairs)` that takes two images and their point correspondences and outputs a stitched panorama in the coordinate system of the first (reference) image. You first need to compute the homography between the two images. Then compute the resulting bounding box and corresponding translation matrix. Create a new black image of the size of the bounding box. Then use a combination of your translation and homography matrix to composite both images into the output. Be careful that you also need to transform the first (reference) image to take the translation into account and that you need to combine the translation and the homography for the second image. Use the right matrix product!

My result for Stata is below:



The seam is not perfect but we will fix this in a later assignment.

Compute another panorama from one of the provided files or from photos that you took. When using the javascript UI, you need to select points that are as well spread as possible for the computation to be well conditioned. In particular, avoid collinear points.

4.5 6.865 only: N images

Write a version of stitch that takes N images and N pairs of correspondences, as well as the index of the reference image to output a full panorama. Make sure you chain your pairwise homographies correctly.

The function should be `stitchN(listOfImages, listOfListOfPairs, refIndex)`.

4.6 6.865 only: speed up warping using Bounding Boxes

Accelerate the warping function by restricting the warping loop using the bounding box of each image. Implement `applyhomography(source, out, H, bbox, bilinear=False)` and update your function `stitchN(listOfImages, listOfListOfPairs, refIndex)` to use it. Report the speedup you obtain.

5 Extra credits

Easy: Implement a least square estimation of the homography for more than four pairs of points. Use `pinv` instead of inv.

Accelerate your warp using scipy's `ndimage.map_coordinates`, which resamples an array according to coordinates stored in a second array. In your case, the first array will be the source image. You will also need to compute a mask to avoid updating pixels outside the source footprint.

Use SVD to alleviate the assumption that the bottom right coefficient of the homography matrix is 1.

6 Submission

Turn in your images, python files, and make sure all your functions can be called from a module `a5.py`. Put everything into a zip file and upload to Stellar.

Include a `README.txt` containing the answer to the following questions:

- How long did the assignment take?
- Potential issues with your solution and explanation of partial completion (for partial credit)
- Any extra credit you may have implemented
- Collaboration acknowledgement (but again, you must write your own code)
- What was most unclear/difficult?
- What was most exciting?
- 6.865: what was the speed-up from using bounding boxes?

Include images for the T example, your own composite, the first version of Stata (without bbox enlargement), the final version of Stata, and a second 2-image pano. For 6.865, add at least one pano with more than 2 images.