Homework 3: Image Mosaicking

CS 639, Fall 2020

Due on October 27

Total points: 13

Please follow the homework guidelines in your submission.

`runHw3.m` will be your main interface for running your code. Parameters for the different programs or unit tests can also be set in that file. Before submission, make sure that you can run all your programs with the command `runHw3("all")` with no errors.

Vectorization

MATLAB is optimized for operations involving matrices and vectors. Avoid using loops (e.g., `for`, `while`) in MATLAB whenever possible – looping can result in long-running code. Instead, you should vectorize loops to optimize your code for performance. In many cases, vectorization also results in more compact code (fewer lines to write!). If you are new to MATLAB, refer to this article for some ideas on how to optimize MATLAB code.

ImageMosaickingApp (13 points)

Your task is to develop an "Image Mosaicking App" that stitches a collection of photos into a mosaic. Creating image mosaics and panoramas has become one of the most popular features on smartphones. Outside the realm of smartphones, similar applications have also been developed to create stunning panoramas seen on GigaPan. With the concepts and algorithms presented in the Image Alignment lecture, you too can create an image mosaic.

Before we create the mosaicking app, we will go over the individual tools required to build it. After that, you will create these tools as separate programs which you write and submit.

We will take one photo (any image is okay) from the collection as the *reference image* for the mosaicking process. Given this reference image, the steps to construct the mosaic are:

1. registration: estimate the geometric transformation relating each image in the collection to the reference image,
2. warping: for each image in the collection, we "undo" the transform estimated from the step above, so that each image is in the same "frame of reference". Still, the different photos will cover different regions of the field of view, for which we perform...
3. blending: combine the warped images into a single panoramic view.

The registration method is prone to errors, so we usually apply the RANSAC (RANdom Sampling And Consensus) technique to make it more reliable. **In this homework, the basic registration is already implemented for you, but you will need to implement RANSAC.**

Each step will be built as its own program, and we will test them individually on some simple inputs so that we are sure that they work correctly. Once we do this, we will put them together to implement the entire stitching pipeline. **This homework has five parts: Challenge 1a through 1e.**

Registration (already implemented)

NOTE: We strongly recommend checking out the `debug1` section in the `runHw3.m` script. It is a test of the registration implementation, and will be important to understand when you get to RANSAC.

The registration program calculates the homography between a pair of images. In our case, the homography is estimated as a transformation relating individual *keypoints* across images.

Assume that you have 2 sets of n corresponding pixel locations – the first set are the n pixel in the reference image, and the second are the corresponding pixels in the other image (the one we want to register to the reference). We use the equation in the Image Alignment lecture to estimate the homography between these sets of points, which you can find in the `computeHomography.m` file.

`computeHomography` calculates the homography between the two sets of corresponding pixels in two images:

```
H = computeHomography(src_pts, dest_pts)
```

It uses the MATLAB function `eig` to compute the homography matrix.

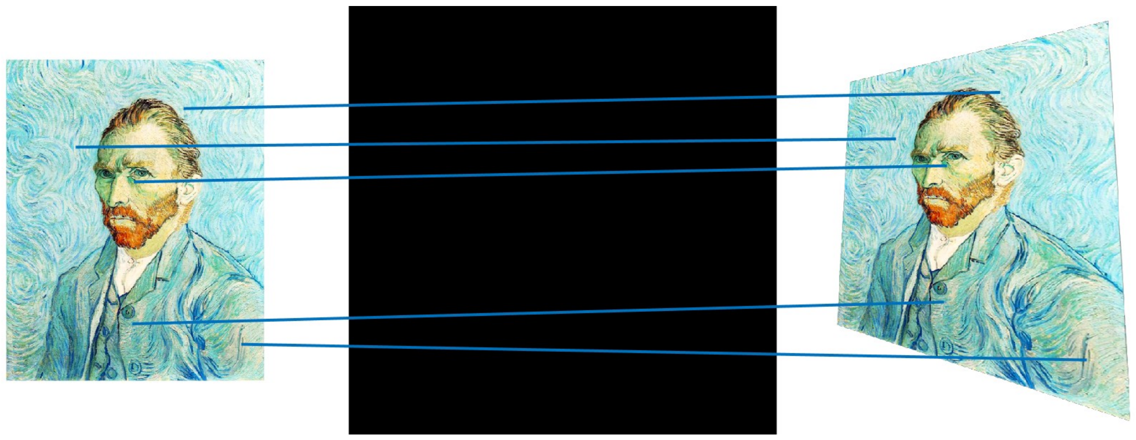
A second program named `applyHomography` applies a homography to a set of points:

```
dest_pts = applyHomography(H, test_pts)
```

The `showCorrespondence` function helps to view (and debug) the mapping between corresponding points in the two images:

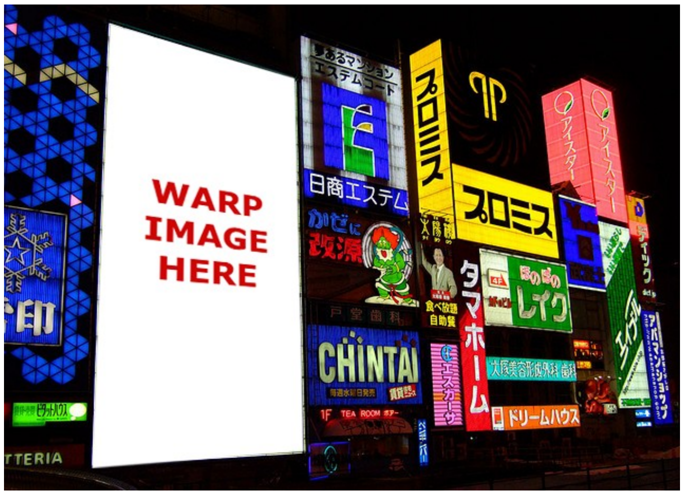
```
result_img = showCorrespondence(src_img, dest_img, src_pts, dest_pts)
```

Here, `src_img` and `dest_img` are two images related by a homography, and `result_img` is an image showing `src_img` and `dest_img` side-by-side, with lines connecting `src_pts` and `dest_pts`. The following image shows an example output of `showCorrespondence`.



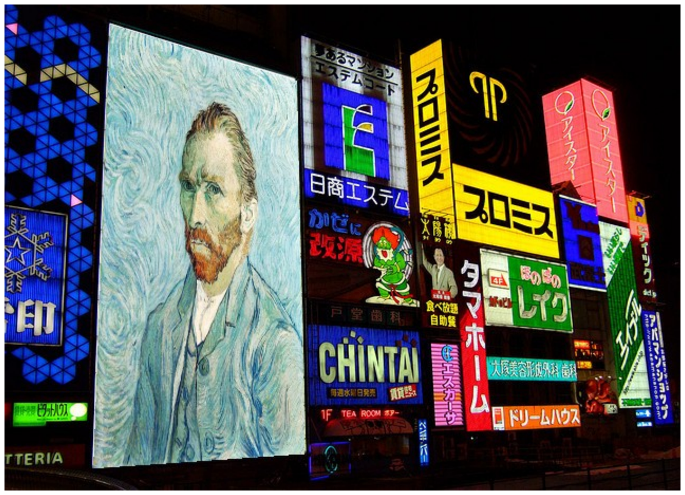
Challenge 1a: Warping (4 points)

Here, we will develop a program to warp an image, and use it to warp and paste a portrait of Vincent (`portrait_small.png`) to the empty billboard in the image shown below (`osaka.png`). To help you get started, a skeleton code has been provided in `challenge1a`.



Before

(source)



After

- First, you need to estimate the homography from the portrait to the target billboard frame. Fill in the missing parts in `challenge1a` where appropriate using `computeHomography`. **(1 point)**

- Second, we need to replace the "Warp Image Here" part of the target image with the portrait image. This will be achieved using *backward warping*: we will create a new image, such that only the area corresponding to the "Warp Image Here" region in the billboard image, gets exactly filled with the portrait, while the rest is left "blank" (pixel value = zero).

  Write a program named `backwardWarpImg` that warps an image based on a homography:

```
[mask, dest_img] = backwardWarpImg(src_img, dest_to_src_homography,
                                   dest_canvas_width_height)
```

The input argument `src_img` is the source image (the portrait in this case). Because you will implement backward warping, `dest_to_src_homography` is the inverse of the homography that maps source to destination. `dest_canvas_width_height` specifies width and height of the destination image.

Now the output arguments. `dest_img` is the output image. In this case, `dest_img` is a warped portrait on a black canvas. `mask` is a binary mask indicating the area of the warped image on the canvas. `mask` is needed because the warped image is often not the final result, but used as an input to other post-processing tasks (in this case, you will post the warped image to another image). **(3 points)**

You are allowed (and expected) to use the MATLAB built-in function `interp2`.

After generating a warped portrait image and its associated mask, superimpose the image on the given billboard image. **The code to perform this task is already included in `challenge1a`.**

Functions not allowed: ALL `im*`, `cp*`, `tf*` functions.

Challenge 1b: RANSAC (3 points)

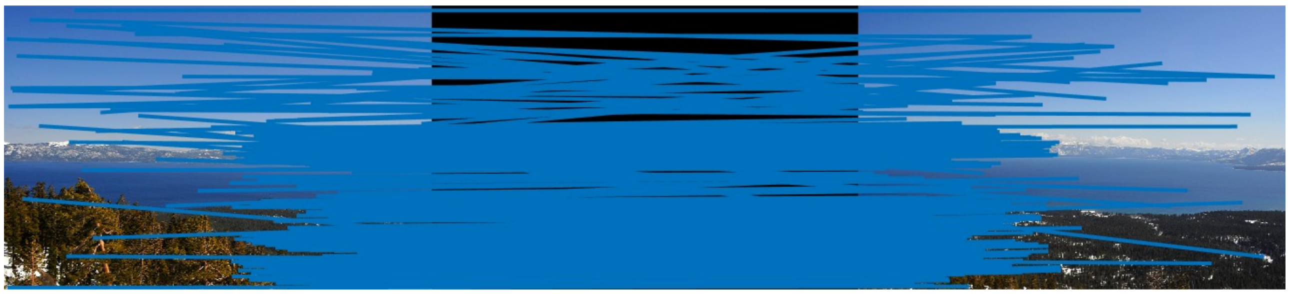
When stitching together real images, we will not select keypoints manually as you did in Challenge 1a. We will use automatic keypoint extraction methods such as SIFT. Here too, a function `genSIFTMatches` has been provided to compute a set of points (`src_pt`) in the source image and a set of corresponding points (`dest_pt`) in the destination image. Typically, these matches will include many outliers, that is, the matches are not correct (they correspond to different scene points). We have discussed the outlier problem in the Image Alignment lecture.

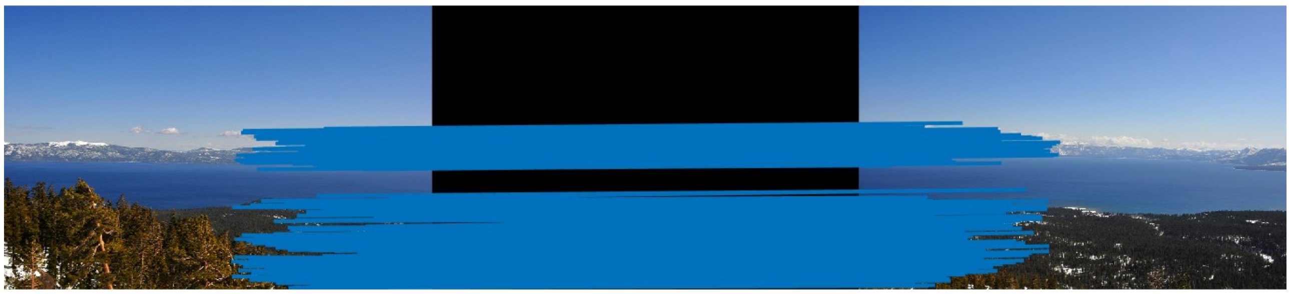
RANSAC (RANdom Sampling And Consensus) is a popular method to robustly estimate quantities in the presence of outliers. **Write a function named `runRANSAC` to robustly compute homography using RANSAC to address the outlier problem.**

```
[inliers_id, src_to_dest_H] = runRANSAC(src_pt, dest_pt, ransac_n,
                                        ransac_eps)
```

`ransac_n` specifies the maximum number of RANSAC iterations, and `ransac_eps` specifies the acceptable alignment error in pixels. Use Euclidean distance to measure the error. Your program should return a homography `src_to_dest_H` that relates the `src_pt` to the `dest_pt`. In addition, return a vector `inliers_id` that lists the indices of inliers, i.e., the indices of the rows in `src_pt` and `dest_pt`. The following two images show the matches between two images, before and after running RANSAC.

Functions not allowed: ALL `im*`, `cp*`, `tf*` functions


Before RANSAC


After RANSAC

Hint: We recommend you to initially try the values `ransac_n` = 100 and `ransac_eps` = 3, before experimenting more with them.

Implementation tip: The RANSAC loop can be implemented in nine lines of MATLAB, including the '`for`' and '`end`' lines. You may find the following snippets helpful, though it is not necessary to use them to receive full credit for this homework.

- `A = H * B; % A and B are 3 x n matrices and H is a 3 x 3 matrix`

-
  ```
  dist = sqrt(sum((A-B).^2)); % A and B are nx3 matrices and dist is a 1xn
                              % matrix
  ```

-
  ```
  inds = randperm(n, 4); % inds is a vector of 4 random unique integers
                         % in [1, n]
  ```

Challenge 1c: Blending (3 points)

Write a program named `blendImagePair` that blends two images into one:

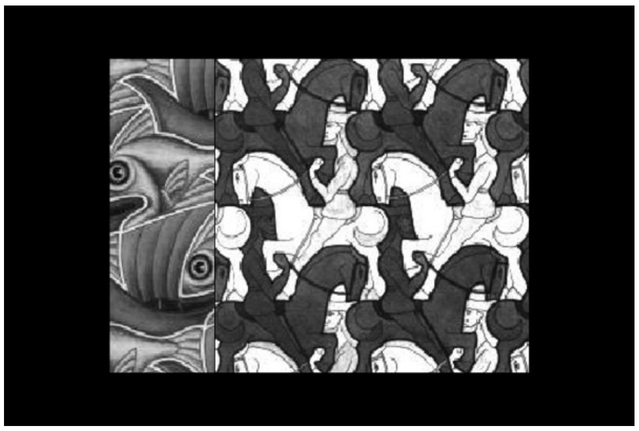
```
result = blendImagePair(img1, mask1, img2, mask2, blending_mode)
```

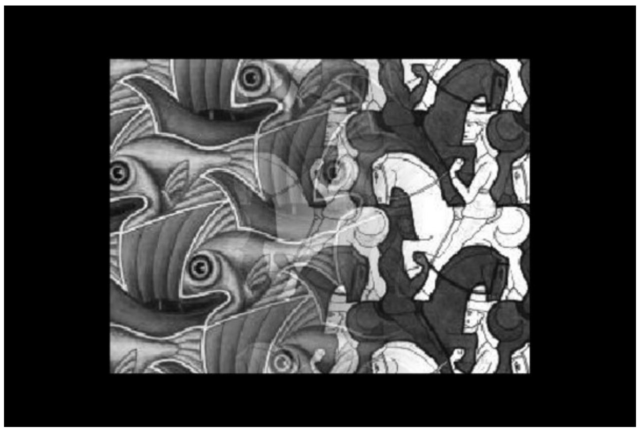
`img1` and `img2` are two input images. `mask1` and `mask2` are the corresponding binary masks indicating the regions of interest. `blending_mode`, a string, is either "`overlay`" or "`blend`".

- In the case of "`overlay`", copy `img2` over `img1` wherever the `mask2` applies. **This case is already implemented for you.**

- In the case of "`blend`", perform weighted blending as discussed in class. **You may use the MATLAB function `bwdist` to compute a new weighted mask for blending – use the '`euclidean`' method if you do so**. The figure below shows example outputs (left: "`overlay`", right: "`blend`"). Also see the figure on the next page, which gives a hint on how to use `bwdist` to compute the weighted mask. After computing the distance transform, you also need to make sure that the mask weights sum to one in the valid region of the output image.

  Functions not allowed: ALL `im*`, `cp*`, `tf*` functions


overlay


blend

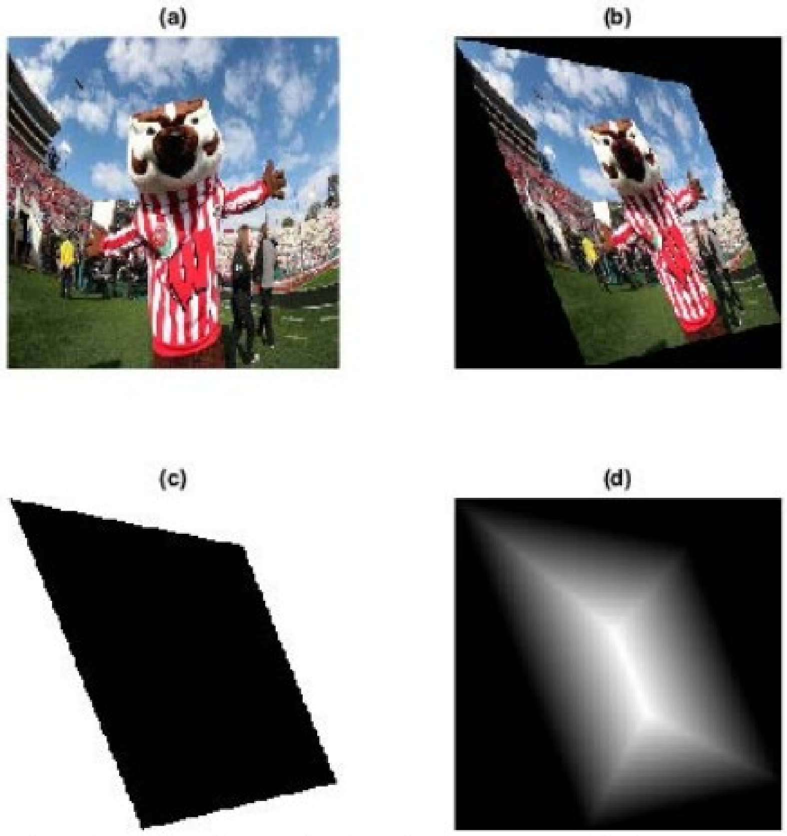


Illustration of computing the alpha channel of an input image. (a) Input image before warping. (b) Input image after warping. (c) Binarization of input - 1s outside the boundary of the warped image and 0s inside. (d) Distance transform of (c). The alpha channel should be the distance transform normalized to the interval (0, 1].

Challenge 1d: Stitching (2 points)

Now you have all the tools to build the mosaicking app! Write a program `stitchImg` that stitches the input images into one mosaic.

```
    stitched_img = stitchImg(img1, img2, ..., imgN)
```

Your program should accept an arbitrary number of images (you can use the MATLAB function `varargin` to take variable number of inputs). You can assume the order of the input images matches the order you wish to stitch the images. Also, in this assignment we will only stitch images of a single row/column. Use "`blend`" mode to blend the image when you call `blendImagePair`.

Functions not allowed: ALL `im*`, `cp*`, `tf*` functions

Challenge 1e: Your own photos (1 point)

Capture images using your own (cell phone) camera and stitch them to create a mosaic. Note that the mosaic need not be a horizontal panorama. Submit both the captured and stitched images.