

## Lab 3

# Image registration

An affine transformation is written

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} = A \begin{pmatrix} x \\ y \end{pmatrix} + t. \quad (3.1)$$

Apart from rotation, translation and scaling it also allows stretching the image in an arbitrary dimension.

**Ex 3.1** What is the minimal number of point correspondences,  $K$ , required in order to estimate an affine transformation between two images?

Once you have found a proper coordinate transformation between two images, you can use the provided function `affine_warp` to warp the source image and create a warped image. Let's try it.

**Ex 3.2** Load the image `mona.png` to the variable `img`. Try running

```
img = read_image('examples/mona.png');
A = [0.88 -0.48; 0.48 0.88];
t = [100;-100];
target_size = size(img);
warped = affine_warp(target_size, img, A, t);
imagesc(warped);
axis image;
```

Change the values in `A` and `t` to see what happens. Swap `A` for a `eye(2,2)` to try a pure translation. Does it behave as you expect?

For any estimation task it is a good idea to have at least one test case where you know what the answer should be. In the next exercise you should make such a test case for Ransac. Start by generating random points, `pts`, and a random transformation. Then transform these points to create a `pts_tilde`. If you want to make it more realistic, add random noise to the points. You now have two sets of points related by a known affine transformation as in (3.1). In the following exercises you will try to estimate this transformation. As you know the correct answer it is easy to detect if you make a mistake.

**Ex 3.3** Make a function

```
[pts, pts_tilde, A_true, t_true] = affine_test_case
```

that generates a test case for estimating an affine transformation. The transformation should map `pts` onto the `pts_tilde`. Don't add any outliers right now. Outputs `pts` and `pts_tilde` should be  $2 \times N$ -arrays. Also output the *true* transformation, so you know what to expect from your code.

**Ex 3.4** Make a minimal solver for the case of affine transformation estimation. In other words, make a function

```
[A, t] = estimate_affine(pts, pts_tilde)
```

that estimates an affine transformation mapping `pts` to `pts_tilde`, where `pts` and `pts_tilde` are  $2 \times K$ -arrays and  $K$  is the number you found in Ex. 3.1. Try your function on points from the test case in Ex 3.3.

**Ex 3.5** Make a function

```
residual_lgths = residual_lgths(A, t, pts, pts_tilde)
```

that computes the lengths of the 2D residual vectors. The function should return an array with the  $N$  values.

*Matlab hint:* Given a  $2 \times N$  matrix, `M`, the column-wise sum of the squared elements can be computed as `sum(M.^2, 1)`.

**Ex 3.6** Modify your function `affine_test_case` so it takes a parameter `outlier_rate` and produces a percentage of outliers among the output points. For example, the outliers could be spread randomly over the image.

**Ex 3.7** Make a function

```
[A,t] = ransac_fit_affine(pts, pts_tilde, threshold)
```

that uses Ransac to find an affine transformation between two sets of points. (Like before the transformation should map `pts` onto `pts_tilde`.) Test your function on test cases generated with your function `affine_test_case`. Try different outlier rates. Make sure that you get the right transformation and a reasonable number of outliers.

For the next exercise you should use the function

```
[pts, descs] = extractSIFT(img);
```

from Lab 1 to extract SIFT features. Note that it only works for grayscale images, so if you have a colour image you need to convert it using, e.g.,

```
[pts, descs] = extractSIFT(mean(img,3));
```

To match features you can use the built-in function `matchFeatures`. To use the Lowe criterion (with threshold 0.8) you should use the following options:

```
corrs = matchFeatures(descs1', descs2', 'MaxRatio', 0.8, 'MatchThreshold', 100);
```

**Ex 3.8** Write a function

```
warped = align_images(source, target)
```

that uses SIFT and Ransac to align the source image to the target image. To perform the actual warping, use

```
warped = affine_warp(target_size, source, A, t);
```

Be very careful about the order in which you send the points to Ransac!

**Ex 3.9** Align `vermeer_source.png` to `vermeer_target.png`. A primitive function

```
switch_plot(warped,target)
```

is provided for viewing. Every time you press a key on the keyboard it will switch between the warped image and the target. (After ten times it stops.)

Medical images often have less local structure, making SIFT matching more difficult. It often works better if we drop the rotation invariance. The provided `extractSIFT` function has an option for this

```
[points, descriptors] = extractSIFT(img, true);
```

assumes that the image has a default orientation.

**Ex 3.10** Modify your `align_images` so it takes the outlier threshold, `threshold` as an input and a boolean, `upright` stating whether the images have the same orientation, i.e.,

```
warped = align_images(source, target, threshold, upright)
```

Try aligning the images `CT_1.jpg` and `CT_2.jpg`. Try with and without rotation invariance and try different outlier thresholds.

**Ex 3.11** Try aligning `tissue_fluorescent.tif` and `tissue_brightfield.tif`. In the fluorescent image, the intensities are basically inverted, so you need to invert one of the images before computing descriptors. (Otherwise you won't get any good matches.) If you used `read_as_grayscale` to load the images, they should have values between 0 and 1 so you can invert it by taking

```
inverted_img = 1 - img;
```

## Warping

So far you have used Matlab's function for warping. The reason is that it is difficult to write a Matlab function for warping that is not painfully slow. Now you will get to write one anyway, but we will only use it for very small images.

**Ex 3.12** Make a function

```
value = sample_image_at(img, position)
```

that gives you the pixel value at `position`. If the elements of `position` are not integers, select the value at the closest pixel. If it is outside the image, return 1 (=white). Try your function on a simple image to make sure it works.

Next, you will do a warping function that warps a  $16 \times 16$  image according to the coordinate transformation provided in `transform_coordinates.m`.

**Ex 3.13** Make a function

```
warped = warp_16x16(source)
```

that warps `source` according to `transform_coordinates` and forms an output  $16 \times 16$  image `warped`. Use your function `sample_image_at` to extract pixel values. Try the function on `source_16x16.tif` and plot the answer using `imagesc`. You will know if you get it right.

## Least squares

**Ex 3.14** Write a function

```
[A, t] = least_squares_affine(pts, pts_tilde)
```

that estimates an affine transformation mapping `pts` to `pts_tilde` in a least square sense, i.e., all points in `pts` and `pts_tilde` are used to compute the transformation. (Depending on how you wrote you `estimate_affine.m`, this might be very easy.) Use it on the inliers from Ransac to refine the estimate. Add this to `align_images` and test it on the Vermeer images for different outlier thresholds. Do you notice an improvement?

## Report

For the report, we want you to submit some images as well as all your code. To save an image `img` in Matlab you write something like

```
imwrite(warped, 'CT_warped.png')
```

Submit images `vermeer_warped.png`, `CT_warped.jpg` and `tissue_warped.tif`. It doesn't matter which direction you chose to warp in, but rename the images you used as target to `CT_target.jpg` and `tissue_target.tif` and submit them as well.