From images to localization

Contents

In this exercise, we will combine what we have learned from previous exercises with RANSAC to localize, directly from images.

1 Preliminaries

1.1 Outline of the exercise

In exercise 2 we have seen how to localize the camera given correct correspondences between image points and points in 3D. Then, in exercise 3 we have seen how to detect keypoints and match them between frames, even if with outliers. In this exercise, we connect the dots and localize from images, from scratch. 3D points (and correspondences in the first image) are still given in this exercise, but this can e.g. be obtained from stereo matching or the eight-point algorithm. A filtering scheme is necessary to cope with the outliers from the keypoint matching step. A powerful algorithm to perform this is RANSAC, which was presented during the lecture. We will start with implementing RANSAC on a simple curve fitting example, to build some intuition about how it works. Then, we will see how RANSAC can be used to both reject outlier keypoint correspondences and determine the pose of the camera from the same correspondences, in one fell swoop (see Fig. 1 and the video at https://youtu.be/n-_EZXwGLhA).

Figure 1: Outlier rejection and localization using PNP and RANSAC. Left: Inliers with displacement (green) and outliers (red). Right: 3D points (blue) and recovered camera poses.

1.2 Provided code

As usual, we provide you with skeletal Matlab code (`main.m`) with a section for each part of the exercise. Your job will be to implement the code that does the actual logic. We also provide the functions stubs with some comments about the input and output formats, so if these are not clear from this PDF, they should be clear from the function stubs. *Again, you do not need to reproduce the reference outputs exactly*, especially since RANSAC is not deterministic (though we encourage you to manually seed the random number generator using the `rng` command - this ensures that you get reproducible results, which makes debugging much easier). Note that you will be using code from previous exercises - you may use your own code, but it's probably less hassle if you use our reference implementations (1, 2, 3).

1.3 Conventions

Because all (square) patches need to be odd-sized, i.e. have a center pixel, we specify their size with a `patch_radius`, such that the patch has dimensions (`patch_radius` $\cdot 2+1)^2$. Pose transformations between frames A and B are denoted with rotation matrix and translation vector R_{AB} and t_{AB} such that the origin of B expressed in A is at t_{AB} and the (x,y,z) unit vectors of frame B expressed in frame A are the columns of R_{AB}. With this, a point p_B expressed in B can be expressed in A as follows:

$$p_A = R_{AB} \cdot p_B + t_{AB}. \tag{1}$$

The inverse transformation is given by $R_{BA} = R_{AB}^T$ and $t_{BA} = -R_{AB}^T \cdot t_{AB}$. W denotes the world or global frame and C the camera frame. The camera looks in positive z direction, x points to the right and y down in the image. In the following, we make the 0th camera frame C_0 coincide with the world frame W.

2 Part 1: RANSAC with a simple model

RANSAC (RANdom Sample Consensus) can be used whenever you'd like to recover a model from noisy and outlier-contaminated data. Given a method to calculate a model from s data points and another method to verify whether a given data point fits a given model RANSAC works as follows:

1. From the given data, choose s data points randomly.
2. Calculate a model guess from these data points.
3. Count how many other data points fit this guess (inlier count).
4. If the given guess has more inliers than previous guesses, save as best guess.
5. Repeat.

How often this procedure is repeated can be chosen depending on the desired outcome. Typically, a minimum inlier count is sought out, and often a maxiumum iteration count is imposed lest RANSAC loops forever. In this exercise, we will simply run RANSAC for 100 iterations, a number which has been arbitrarily chosen by the TAs. You will later compare this to a formula seen in class that can be used to derive that number.

To familiarize ourselves with RANSAC, we will first apply it to a simple problem: The given data consists of 20 inlier points (x_i, y_i) sampled at random from a given parabola $p(x) = ax^2 + bx + c$ (and perturbed in y-direction with noise with known maximum extent η_{max}), and 10 random outlier points. The problem is to recover the original parabola from the given points.

To solve this problem, one could simply fit the parabola from all the samples. However, the outliers would deteriorate the quality of the solution. Instead, RANSAC makes use of the known maximum noise and is thus able to identify outliers, and provide a better model of the parabola. Implement the function `parabolaRansac`, which returns the best guess and corresponding inlier count for each iteration. A data point (x_i, y_i) is considered an inlier if $|y_i - m(x_i)| \leq \eta_{max}$, where $m(x)$ is the model of the parabola. You should achieve something similar to Fig. 2. You should also obtain a lower root-mean-square (RMS) error than with a full data fit. Some hints:

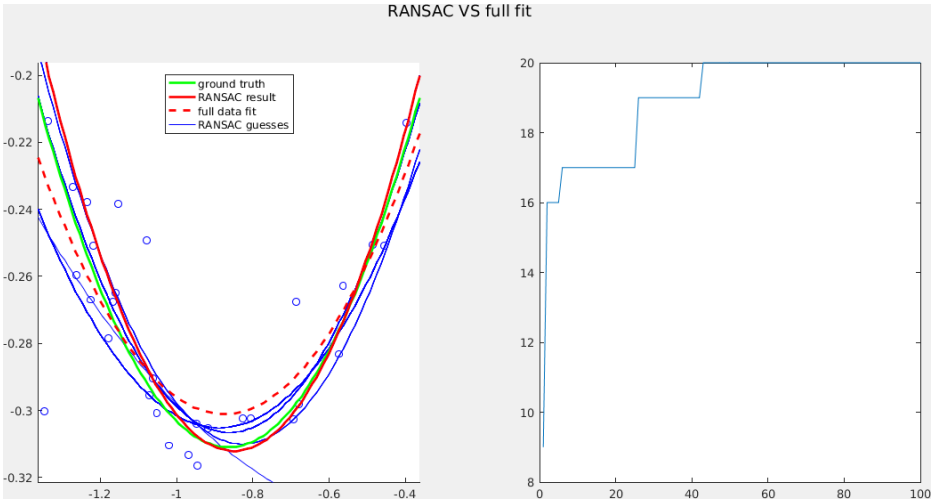

Figure 2: Left: Data points and outliers given in part 1 (blue circles), different models and ground truth (labeled lines). Right: Inlier count of best guess over the iterations.

- Consider the `datasample` function in Matlab, and pay attention to the `'Replace'` parameter.
- RANSAC involves randomness, which is provided by the Matlab random number generator (RNG). If you are debugging and would like to have the same random behaviour across multiple runs, you can set the RNG seed using the `rng` function at the beginning of your procedure. Different seeds will result in different behaviours. Once your code is debugged, however, remove the `rng` command to see how RANSAC behaves in different instances. The plot in Fig. 2 has been obtained with rng(2).
- Use $s = 3$, `polyfit` and `polyval`.

Once you have determined the best guess at a given time (step 4), you can improve your best guess further by fitting the model from all inliers. Do this in `parabolaRansac` and observe that the RMS error becomes even lower. Run this code a couple of times (remember to remove the call to `rng`). Roughly how many iterations does it usually take to find the correct solution? In class you have seen that given s, the outlier fraction ϵ and a desired success rate p, the required amount of iterations k can be calculated as follows:

$$k = \frac{\log(1-p)}{\log(1-(1-\epsilon)^s)} \tag{2}$$

Given the problem, what should k be? Why doesn't it correspond to what you observe running the code? What does this mean in practice?

3 Part 2: Localizing with RANSAC

Now that you have validated your RANSAC implementation on a simple example, it's time to apply it to localization! Together with what you have learned in exercises 1, 2 and 3, you now have all the tools to localize new frames from scratch, given 3D points and correspondences in a reference frame. Implement the function `ransacLocalization`, which takes a query image (the image whose camera pose you try to recover), the database image (the image in which keypoints are already given), keypoints in the database image, the corresponding 3D points and the camera matrix K, and recovers from them the pose (R_{CW}, t_{CW}) of the query image camera (and some other data we will use for visualization). The function should roughly do the following:

1. Find keypoints in the query image (use code and parameters from exercise 3). We recommend to extract 1000 keypoints.

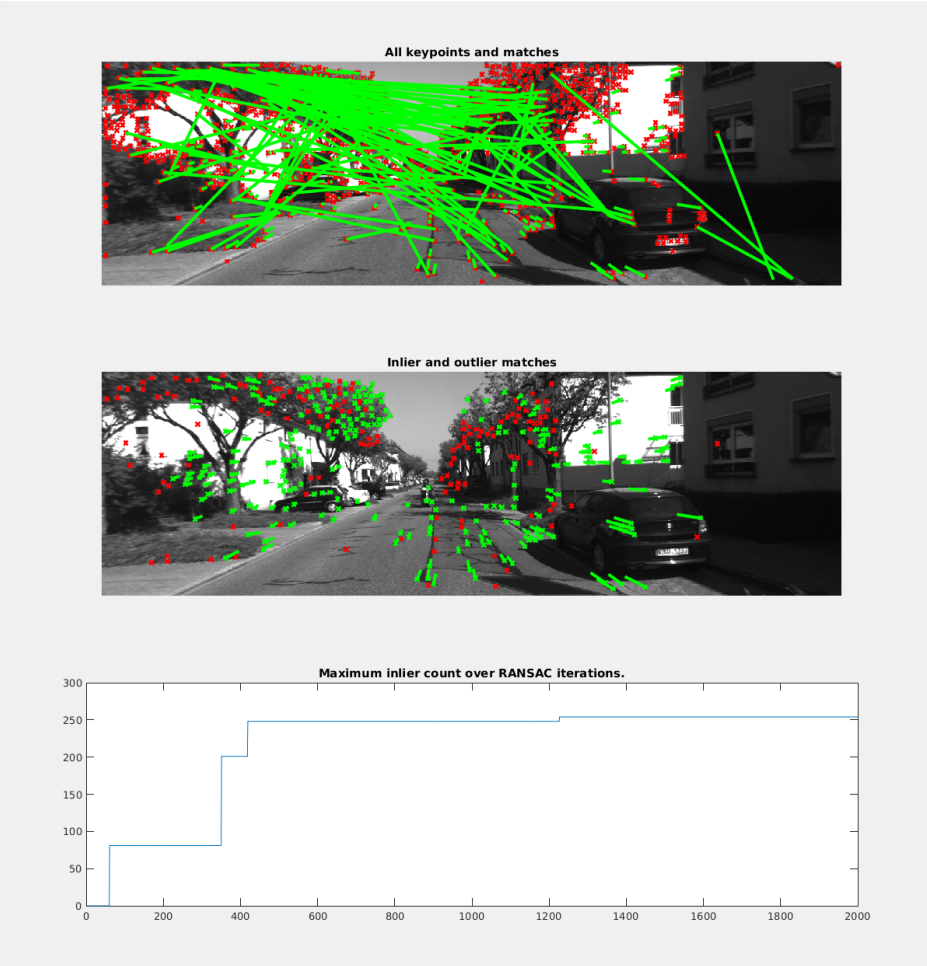

Figure 3: PNP and RANSAC applied to the first frame. RANSAC is able to reject the many outliers from descriptor matching seen in the top picture.

2. Match them with the keypoints given in the database image (use code and parameters from exercise 3, except match_lambda= 5). This will result in matches with outliers.

3. Apply RANSAC with DLT (exercise 2) to recover (R_{CW}, t_{CW}) in spite of the outliers obtained in the matching. Count as inliers all matches where the 3D landmark matched to a keypoint projects at most 10 pixels (tuned for your convenience) away from the keypoint (use the 3D to 2D projection code from exercise 1). Make sure to re-run DLT with all inliers on the best match, for added precision.

Note that the iteration count is up to you. Given what we just learned in Part 1, we recommend starting with a high number, say 2000.

You should obtain the result shown in Fig. 3. The recovered pose should correspond to a forward motion of the car of roughly 70cm, and the inlier ratio should be around 63%. With this, compare again the predicted required iteration count from (2) with what you get in practice.

4 Part 3: Less iterations with P3P

In class we have seen that (2) can be used to estimate the amount of iterations needed for RANSAC. In parts 1 and 2 of this exercise we have seen that with noisy data this formula can't be used reliably to choose the amount of iterations. This because strictly speaking (2) does not express the amount of iterations such that RANSAC succeeds with probability p, but the amount of iterations such that

RANSAC at some point selects s inliers with probability p. Depending on noise in the data and on how robust the model estimation is to such noise, a model built with s inliers can in some cases still be undistinguishable from a model built with outliers. This is much harder to model and definitely outside of the scope of this exercise.

Still, since having s inliers is typically necessary to obtain a correct model, the main idea seen in class holds: Requiring a lower s for building the model will result in RANSAC finding the correct model faster. We will validate this in this part. We have so far used DLT to solve the PNP problem using 6 data points. P3P is an algorithm which can do this using 3 data points, with the caveat that it provides two possible solutions. This is of course not a problem in RANSAC, since we can simply count inliers with both solutions, and pick the solution with more inliers. Modify `ransacLocalization` to build the model using P3P. The implementation of P3P is provided in `p3p.m` (if you are not using Matlab, you may skip this part. Otherwise, a C++ implementation is provided at http://www.laurentkneip.com/software->P3P . OpenGV, on the same website, apparently provides a Python wrapper, but might be cumbersome to install). Some caveats and hints:

- Given noisy data, P3P may return complex values. In this case you should retain only the real part.
- Note that P3P returns T_{WC}, and not T_{CW} like our DLT implementation.
- P3P only takes 3 points, so use DLT for the final refinement that you do once the inliers are determined.
- Note that P3P takes bearing vectors that have unit length! This also means you will need to make use of K.

Your RANSAC should now reliably find the correct solution with about 200 iterations.

5 Part 4: Localizing subsequent frames

You may now run part 4 in `main.m`. Your initial solution will not perform quite the same as https://youtu.be/n-_EZXwGLhA. We have achieved this behavior by increasing the number of iterations in RANSAC and requiring a minimum inlier count. Can you tweak these parameters to obtain similar behavior? Note that the amount of inliers decreases over the frames, even with increasing the RANSAC iteration count. For the last couple of frames, RANSAC often even fails completely. Why? Could you think of a way to make localization work across more frames (the one way we think of would involve changing the function interface)?