# Lecture Notes: Matching and Alignment

Subhransu Maji

April 1, 2025

## Contents



## 1 Overview

The next set of lectures focuses on using local features for image alignment. The process consists of two main steps:

1. Matching local descriptors to establish correspondences.
2. Estimating an image transformation based on the identified correspondences.

We will introduce the family of transformations that images undergo due to small changes in camera viewpoint. Additionally, we will cover RANSAC, a technique for robust model fitting, which helps handle outliers in the correspondence estimation process. Finally, we will warp one image based on the estimated transformation to form a seamless panoramic image.

## 2 Matching local descriptors

Concretely, given a set of local descriptors in each image, we can find descriptors that are similar to each other (e.g., based on the Euclidean distance between two SIFT vectors) to estimate correspondences. We can eliminate some poor matches by applying the ratio test, i.e., when the ratio of the distance between the best match and the second-best match is less than a threshold (typically 0.7-0.8 for SIFT features).

In the class we looked at a matching demo with and without the ratio test being applied. The implementation was in OpenCV and the code is linked from the course website.

Many of these matches may still be incorrect, so we need to find a transformation that is consistent with as many correspondences as possible. To do this, we must represent image transformations using parameters and then use RANSAC to efficiently find the optimal parameters.

## 2.1 Image Transformations

Before we delve into estimating transformation from correspondences lets understand the space of transformation that can happen to images when the camera viewpoint changes. Mathematically we would like to understand the space of functions that map the coordinates of a pixel in one image to another.

### 2.1.1 2D transformations

Lets start with transformations in two dimensions (Figure 1) and how to represent them mathematically:

- translation
- rotation
- scaling
- shearing

All of these can be implemented by multiplying a coordinate vector

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix},$$

by a 2x2 matrix $M$ and adding a 2x1 matrix $t$. This is written

$$\mathbf{x}' = M\mathbf{x} + t.$$

We now describe these basic transformations.

**Shifting (or translation)** Preserves orientation and all the properties of rigid transformations. Given a coordinate vector $\mathbf{x}$ that is being transformed to a q new position $\mathbf{x}'$, this can be written as

$$\mathbf{x}' = \mathbf{x} + \begin{bmatrix} t_x \\ t_y \end{bmatrix},$$

where $t_x$ and $t_y$ are the horizontal and vertical displacements. Using the general matrix notation this operation can be written as as

$$\mathbf{x}' = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{x} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}.$$

**Rotation** Preserves areas (or volumes if in 3-D) and all the properties of similarity transforms. This can be written as

$$\mathbf{x}' = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \mathbf{x},$$

where $\theta$ is the angle of rotation. Note that this operation will rotate the x around the point $(0, 0)$ rather than around the center of the image in a *counter-clockwise* manner.

**Scaling (Uniform)** A scaling operation can be written as

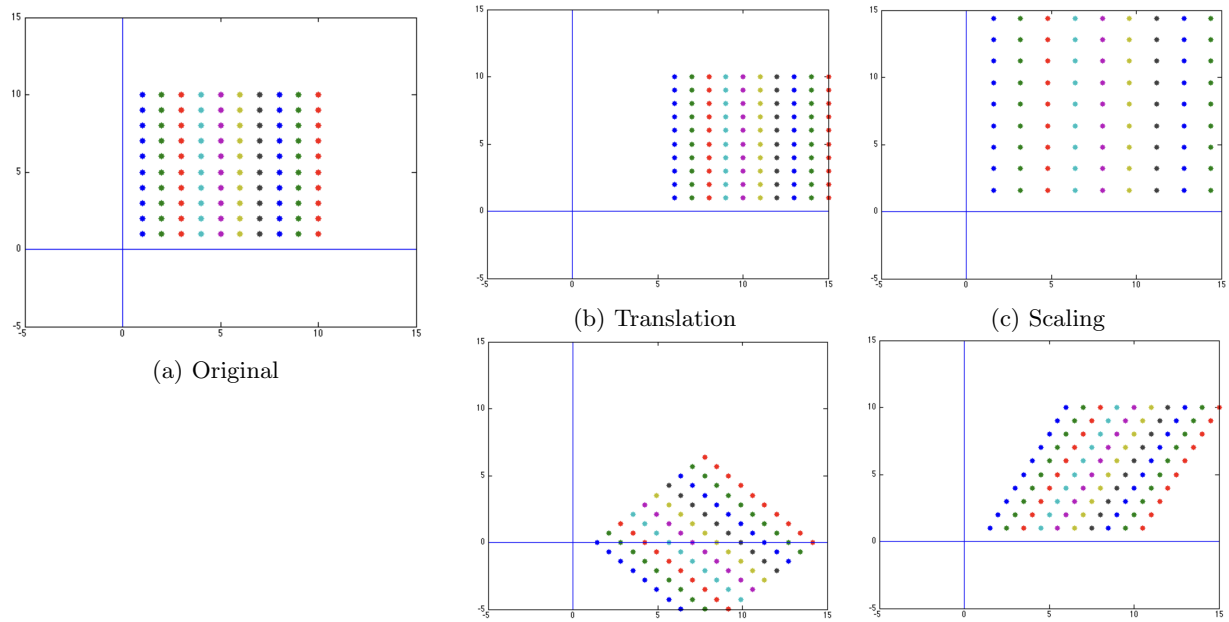
$$\mathbf{x}' = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \mathbf{x},$$

where $s$ is the amount of scaling. If $s$ is greater than one, the image will be magnified. If $0 < s < 1$, the image will be shrunken.

**Non-uniform Scaling** In non-uniform scaling, the horizontal and vertical directions are "stretched" by different amounts. The operation can be written as

$$\mathbf{x}' = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \mathbf{x},$$

where $s_x$ is the amount of scaling in the horizontal direction and $s_y$ is the amount of scaling in the vertical direction.



(a) Original (b) Translation (c) Scaling (d) Rotation (e) Shearing

Figure 1: 2D transformations as change of coordinates

**Shearing** A shearing can be written as either

$$\mathbf{x}' = \begin{bmatrix} 1 & s_x \\ 0 & 1 \end{bmatrix} \mathbf{x},$$

for shearing horizontally, or

$$\mathbf{x}' = \begin{bmatrix} 1 & 0 \\ s_y & 1 \end{bmatrix} \mathbf{x},$$

for shearing vertically. Shearing in the $x$ direction "tilts" the image to the right if the shearing value is positive and to the left if the shearing value is negative.

### 2.1.2 Larger families of transformations

In this subsection, we discuss the following families of transformations:

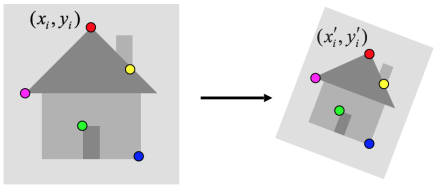
- rigid transformations
- similarity transformations
- affine transformations

The transformations in each family are a subset of the transformations below them. For example, the family of rigid transformations is a subset of the family of similarity transformations, and also a subset of affine transformations.

**Rigid (translation + rotation)** Rigid transformations are any combination of translations and rotations. They can be done to a "rigid" object without deforming it or stretching it. They are also known as **Euclidean** transformations. Rigid transformations in 2 dimensions preserve areas (or volumes if in 3-D) and all the properties of similarity transforms.

**Similarity (rigid + scale)** Similarity transformations are any combination of translations, uniform scaling, and rotations. Hence, they can also be described as any combination of a rigid transformation and a uniform scaling. They preserve angles, straightness of lines, and parallelness of lines.

**Affine (similarity + shear)** Affine transformations include any combination of translations, rotations, scaling (including non-uniform) and shearing. Any combination of 2x2 matrix for $M$ and 2x1 for $T$ and represents an affine transformation.




$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \end{bmatrix}$$

$$\mathbf{x}_i' = \mathbf{M}\mathbf{x}_i + \mathbf{t}$$

Want to find **M**, **t** to minimize

$$\sum_{i=1}^{n} \| \mathbf{x}_i' - \mathbf{M}\mathbf{x}_i - \mathbf{t} \|^2$$

Figure 2: Estimating a transformation given correspondences. The objective function quantifies the difference between a pixel correspondence and its estimated correspondence calculated using $M$ and $t$.

### 2.2 General image transformations

A 2D affine transformation approximates viewpoint changes for planar objects under small perspective distortions or orthographic projection.

A broader class of transformations, known as homographies, models the transformation of planar surfaces under full perspective projection. However, real-world scenes are three-dimensional and generally require more complex transformations. Affine and homographic transformations are often used as initialization steps for more advanced models in feature matching and image alignment such as 3D rigid transformations and bundle adjustment.

More generally, in the presence of non-rigid objects or moving objects, a single affine or homography is not sufficient to describe transformations. Matching then requires an even broader set of transformations such as thin-plate splines.

## 3 Estimating transformations from matches

Given matches between a set of points and a transformation family the transformation can be obtained by solving a system of linear equations (Figure 2). Each match provides two constraints, one for the x and y coordinate. Thus, depending on the number of parameters to be estimated, different numbers of matches are necessary. A translation has two parameters $t_x$ and $t_y$, hence a single correspondence is sufficient, while a affine transformation has six parameters and needs three matches.

However, a challenge is we might have many outliers or incorrect correspondences in the previous step. Simply fitting the transformation using least squares optimization using all the correspondences will likely lead to a poor result. Thus we need an approach for model fitting that is robust to outliers. This is where RANSAC comes in.

### 3.1 Random Sample Consensus (RANSAC) algorithm

**Random Sample Consensus (RANSAC)** is a robust method for model fitting in the presence of outliers. The basic procedure is as follows:

1. Randomly select a small subset of data points uniformly from the dataset.
2. Fit a model to this subset.
3. Determine which of the remaining points are consistent with the model—i.e., which lie within a specified threshold—and treat the rest as outliers.
4. Repeat this process many times, and select the model that has the highest number of inliers.

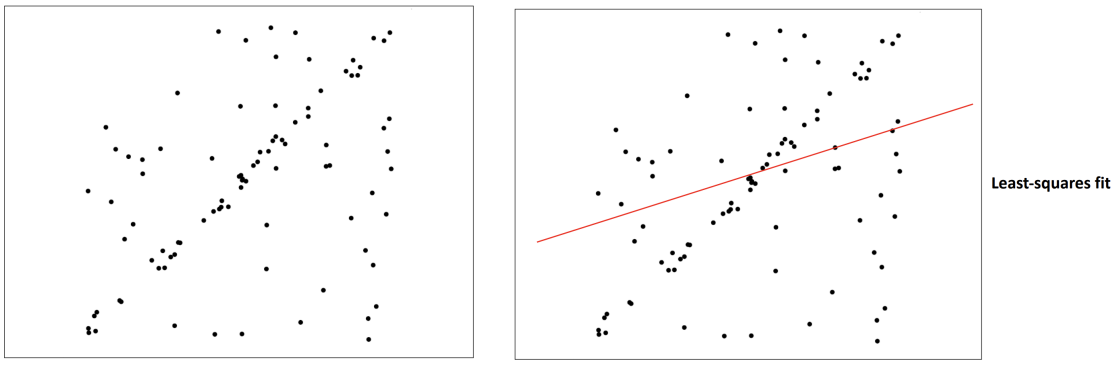
For example, when estimating a *rigid transformation* from feature correspondences, we can recover parameters such as rotation, scaling, and translation. However, the RANSAC algorithm is general and can be applied to a wide range of model fitting problems. Let's now consider a simple example to illustrate how it works.

### 3.2 Fitting a Line in the Presence of Outliers Using RANSAC

Suppose you are given a set of 2D points (Figure 3a: $(x_1, y_1), \ldots, (x_n, y_n)$, and the goal is to find a line that passes through the largest number of points. A common approach is to fit a line using least-squares minimization:

$$\min_{a,b} \sum_i (ax_i + b - y_i)^2, \tag{1}$$

which finds the line $y = ax + b$ that minimizes the squared error between the predicted and observed values. However, as shown in Fig. 3b, this method can be heavily influenced by outliers, resulting in a poor fit that does not represent the majority of the data.




(a) Original (b) Fitting with Least Squares

Figure 3: Fitting a line using least squares minimization can be heavily influenced by outliers

RANSAC provides a robust alternative by explicitly seeking the line that best fits the largest set of inliers. The algorithm proceeds as follows:

1. Repeat for $N$ iterations:
   (a) Randomly select 2 points from the dataset.
   (b) Fit a line to these 2 points.
   (c) Identify inliers among the remaining points—i.e., those whose perpendicular distance to the line is less than a predefined threshold $t$.
   (d) If the number of inliers is greater than $d$ (a minimum threshold) and also higher than the current best model:
      i. Accept the new line as the best model so far.
      ii. Optionally refit the line using all identified inliers.

This approach is robust to outliers and often yields better results in practice than standard least-squares fitting when outliers are present.

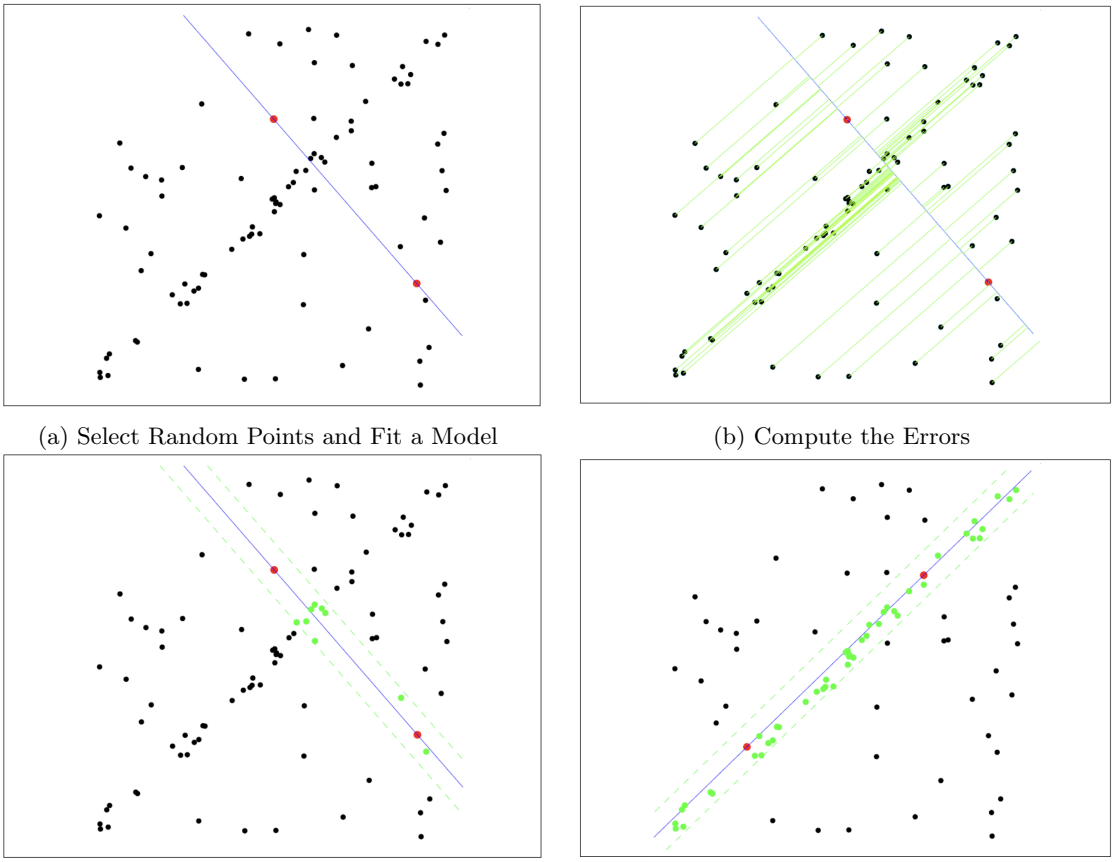
### 3.3 Robustness of RANSAC

RANSAC is a randomized algorithm, and there is always a chance that it may terminate after $N$ iterations without finding a correct model. However, this probability can be made arbitrarily small by increasing the number of iterations. Here, we analyze the probability of failure and show how quickly it decreases with more iterations.

Suppose a fraction $p = 0.5$ of the points are outliers—i.e., 50% of the points do not lie on the line that fits the inlier data. The probability that RANSAC fails in a single iteration is:

$$q = 1 - (1 - p)^2$$

This is because the only way to succeed in one iteration is to randomly choose two inlier points (since we need two points to define a line). The probability of choosing two inliers is $(1 - p)^2$, and thus the probability of failure is $q = 1 - (1 - p)^2$.



(a) Select Random Points and Fit a Model

(b) Compute the Errors

(c) Determine and count the Inliers

(d) Best fit accepted after N iterations

Figure 4: The RANSAC algorithm repeats steps (a), (b), and (c) for N iterations and accepts the model with most inliers

For $p = 0.5$, this gives:

$$q = 1 - (1 - 0.5)^2 = 1 - 0.25 = 0.75.$$

So there is a 75% chance of failure in a single iteration.

To reduce the failure probability, we repeat the process $T$ times. The probability that RANSAC fails in all $T$ iterations is:

$$q^T = \left(1 - (1 - p)^2\right)^T.$$

If we want the success probability to be greater than 99%, i.e., $1 - q^T > 0.99$, we can solve for $T$. For $p = 0.5$, the required number of iterations is approximately:

- $T \approx 17$ for success probability $> 0.99$
- $T \approx 24$ for success probability $> 0.999$
- $T \approx 32$ for success probability $> 0.9999$

Thus, by increasing the number of iterations, we can significantly improve the probability of success, making RANSAC a highly robust algorithm in practice.

## 3.4 Fitting a Transformation Using RANSAC

RANSAC can be used to robustly estimate geometric transformations even in the presence of outlier correspondences. The algorithm proceeds by repeating the following steps:

- **Select a random subset of matches.** The number of matches required depends on the type of transformation being estimated:
  - At least 3 matches are needed to estimate a general *affine transformation* (6 parameters).

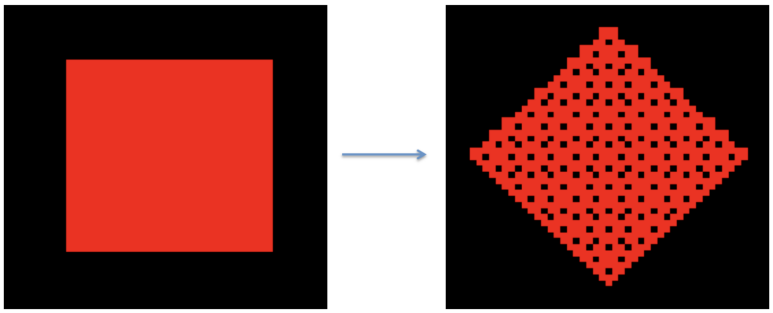


Figure 5: Forward warping may result in missing pixels in the destination image due to the discrete nature of the image grid.

  - At least 2 matches are needed to estimate a *similarity transformation* (scale, rotation, and translation — 4 parameters).
  - Only 1 match is needed to estimate a *pure translation* (2 parameters).
- **Fit the transformation** using the selected matches, typically via a least-squares formulation.
- **Compute the number of inliers**—i.e., matches that are consistent with the estimated transformation, within a predefined error threshold.
- Keep the transformation that yields the largest number of inliers across all iterations.

# 4 Warping Images

The final step in creating a panoramic image is to warp one image so that it aligns with the other. In the previous step, we estimated the transformation matrix $T$ that maps points from one image to the other. This transformation can now be used to align the images through a warping operation. There are two common approaches to image warping: **forward warping** and **inverse warping**.

**Forward warping:** In this approach, we iterate over all pixels in the *source image*, apply the transformation $T$ to compute the corresponding location in the *destination image*, and copy the pixel to that location. However, due to the discrete nature of the image grid, this process can result in missing pixels in the destination image, since not every destination pixel is guaranteed to be filled (Fig. 5).

**Inverse warping:** A more robust alternative is inverse warping. Here, we loop over every pixel in the *destination image*, apply the inverse of the transformation $T^{-1}$ to compute the corresponding location in the *source image*, and then sample the pixel value from that location. This method avoids holes in the destination image and is generally preferred in practice. Interpolation (e.g., bilinear or bicubic) is often used when the computed source coordinates are non-integer.

# 5 Beyond SIFT

The SIFT algorithm[1] was introduced by David Lowe in 1999 and quickly became a standard method for feature matching in computer vision tasks. Since then, many improvements and variations of SIFT have been proposed to make it faster, more accurate, and more robust to changes in viewpoint and geometry.

In 2005, researchers Mikolajczyk and Schmid[2] conducted a systematic evaluation of feature detectors and found that SIFT consistently ranked among the best for matching performance.

Today, feature matching has advanced significantly with the rise of deep learning. Modern methods use convolutional networks and transformers to extract richer features that can match parts of images even when there's very little texture—something traditional techniques like SIFT struggle with. This remains an active area of research, with many real-world applications like 3D reconstruction.

Competitions such as the Image Matching Challenge[3] are held each year to evaluate the best-performing algorithms. In 2024, top methods included SuperGlue[4] and LightGlue[5], which represent the state-of-the-art in deep learning–based feature matching.

[1] `https://en.wikipedia.org/wiki/Scale-invariant_feature_transform`
[2] `https://ieeexplore.ieee.org/abstract/document/1498756/`
[3] https://www.kaggle.com/competitions/image-matching-challenge-2024/overview
[4] https://github.com/magicleap/SuperGluePretrainedNetwork
[5] https://github.com/cvg/LightGlue