School of Computer Science and Communication, KTH

DD2423 Image Processing and Computer Vision:

Lab 3: Image matching and 3D reconstruction

You will in this lab explore different methods for robust image matching using local features and use two kinds of models to relate features between pairs of images, homographies when the world is assumed flat, and fundamental matrices otherwise. Once features have been matched and models estimated, you will attempt to make a 3D reconstruction of the world as viewed in the images.

The *goal* of this lab is for you to understand the possibilities and limitations of feature matching using models such as homographies and fundamental matrices to gain practical experience enough for you to approach new problems for which feature matching may be necessary.

As *prerequisites* to this lab you should have read the course material on local image features, model fitting and representation.

Reporting: When assessing this lab emphasis is placed on understanding how robust image matching with RANSAC works, how to do parameter estimation though constrained least square minimization, what remains to be done to determine the 3D positions of matched image points, and how sensitive the different steps are to different kinds of noise.

It is recommended to use the command `subplot` to assemble multiple results into figures, thus simplifying the interpretation of these. In the answer sheet write down summarizing conclusions and compile results for the explicitly stated exercises and questions.

Advice: Think more about what you are expected to learn, than what you are expected to do. Note that in many cases there are many possible correct answers to the same question. How would you characterize the methods you test? And when would you use them? **Before you present your results, make sure that you understand how each of the methods works.**

Python packages The labs rely on a couple of Python packages that you need to install before you can begin. Most important is `NumPy` which includes fundamental functions for linear algebra and representation of matrices (and images). Selected functions in `SciPy` might also be used. We will use `Matplotlib` to display images and graphs. Finally, `OpenCV` will be used for loading, storing and filtering RGB images. Additional functions in the lab necessary for drawing and generating data are kept in `plots.py` and `utils.py`.

We will study two different models for matching images of the same scene, one based on homographies and another based on the fundamental matrix. Homographies are typically used when two images are taken with cameras placed at the same position but with slightly different orientations. For the images to be matched there must be a bit of overlap, where the same parts of the scene are seen in both images. Once the two images have been matched, typically done using local image features, the images can be transformed into a new image that combines the information from both.

If the images are taken from two different vantage points, they can instead be matched using the epipolar geometry constraint with a model based on the fundamental matrix. Once the fundamental matrix has been recovered, a 3D representation of the points observed in the scene can be created and visualized. However, the quality of the 3D reconstruction depends heavily on how well the cameras have previously been calibration, and the degree of image noise.

1 Using OpenCV and NumPy

OpenCV is an open-source library for computer vision, the most widely used library for more than 20 years. While much of the development of new algorithms was originally done in Matlab, implementations on real hardware, such as robots, were usually based on OpenCV, which consists of functions written in highly optimised C++ code. To be as fast as possible, accuracy was sometimes sacrificed for speed, e.g. by representing pixels in terms of bytes instead of floating point values. Nowadays OpenCV has a Python binding and is built on top of NumPy, a very efficient library for numerical operations.

1.1 Images in OpenCV

With OpenCV, colour images are internally stored as three-dimensional NumPy arrays, with the last dimension representing the colour space. Convenient and frequently used image functions are `imread()` for reading images from disk, `imwrite()` for storing images and `resize()` for changing the size of the image. Please read the help text for the individual functions to understand how they are used in practice. The commands

```
import cv2
img = cv2.imread('filename.jpg', flags=0)
```

can for example be used to read a JPEG file from disk. The argument `flags=0` means the image is converted to a grey-level image when loaded, which can be convenient in many cases. Typically, images are stored with each pixel value represented by a byte. To keep the precision when applying sequential filtering operations, it is common to change the image format to floating points using either the types `np.float64` or `np.float32`.

```
import numpy as np
I = img.astype(np.float32)
```

One should keep in mind though, that floating points will decrease the speed of computation.

1.2 Matrices in NumPy

In this lab NumPy will be used for matrix manipulation and to solve systems of linear equations. The easiest way to create a matrix is through the function `np.array()`, such as in

```
M1 = np.array([[1,0,0,0], [0,1,0,0], [0,0,1,0]])
```

that creates a 3×4 matrix `M1`, but one can also use `np.zeros()` or `np.ones()` to create matrices of just zeros and ones. Another common function is `np.eye()` for creating an identity matrix. To stack multiple matrices on top of or next to each other, it is possible to use `np.vstack()` or `np.hstack()`. Matrix multiplication is written with an at sign, `C = A @ B`, which is often confused with `C = A * B`, which instead denoted an element-wise multiplication.

To solve systems of linear equations, when the number of equations is the same as the number of unknown parameters, one might use `np.linalg.inv()` for matrix inversion. In many other cases, you end up with the problem of finding eigenvectors and eigenvalues for which either `np.linalg.eig()` or `np.linalg.svd()` can be used. Often preferred is `np.linalg.svd()`, since it works on rectangular matrices, not just square matrices, and the results have the singular values sorted in descending order. For example, if `Q` is 9×9 matrix, then

```
U, S, Vh = np.linalg.svd(Q)
v = Vh[8,:]
```

will result in `v`, which is the eigenvector corresponding to the eigenvalue of `Q` closest to zero.

2 Recovering homographies

In this exercise, you will try to use local image features to match two images of the same scene, where the matching part of the scene is assumed to be flat. In some cases, that part is indeed flat, e.g. a wall, floor or ceiling, but in other cases, the part is simply located so far from the camera that its image looks flat in the projection. In Figure 1 we see two images of the same wall, with several image features matched between the images.

2.1 Mathematical background

If enough such features are successfully matched, we can find a projective transformation H, known as an *homography*, with which coordinates of points in one of the images can be transformed into those of the other image. In homogeneous coordinates, the projections of an individual point seen in both

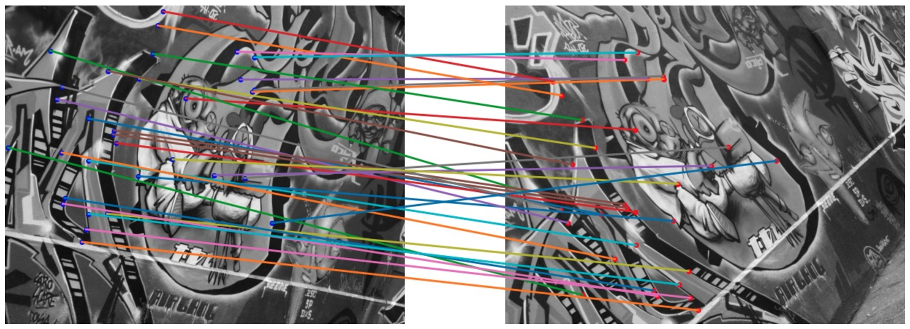

Figure 1: Feature matches between two images of a world that is quite flat.

images can be written as

$$\mathbf{x}_a = \begin{pmatrix} x_a \\ y_a \\ 1 \end{pmatrix}, \text{ and } \mathbf{x}_b = \begin{pmatrix} x_b \\ y_b \\ 1 \end{pmatrix}$$

and the homography

$$H = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} = \begin{pmatrix} \mathbf{h}_1^T \\ \mathbf{h}_2^T \\ \mathbf{h}_3^T \end{pmatrix}$$

Then the relationship between the two images can be written

$$\mathbf{x}_b \simeq H\mathbf{x}_a,$$

where $\simeq$ indicates equivalence up to scale, or $\mathbf{x}_b = \lambda H \mathbf{x}_a$ with λ being some unknown scale factor. Even if H contains 9 unknown parameters, it only has 8 degrees of freedom, since we use homogeneous coordinates and the scale does not matter. For example, we could have set $h_{33} = 1$ and treated the other 8 parameters as unknowns, at least if we can guarantee that h_{33} is not zero. An alternative is to only look for parameters for which the norm of the vector of all parameters

$$\mathbf{h} = (h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}, h_{32}, h_{33})^T$$

is equal to one, i.e. $||\mathbf{h}||^2 = \mathbf{h}^2 = 1$.

We can eliminate the unknown parameter λ by going back to Euclidean coordinates, which is done by simply dividing the first two elements of $H\mathbf{x}_a$ by the last. Instead of having a 3-vector in homogeneous coordinates, we get the two coordinates

$$x_b = \frac{h_{11}x_a + h_{12}y_a + h_{13}}{h_{31}x_a + h_{32}y_a + h_{33}} = \frac{\mathbf{h}_1^T\mathbf{x}_a}{\mathbf{h}_3^T\mathbf{x}_a}, \quad \text{and} \quad y_b = \frac{h_{21}x_a + h_{22}y_a + h_{23}}{h_{31}x_a + h_{32}y_a + h_{33}} = \frac{\mathbf{h}_2^T\mathbf{x}_a}{\mathbf{h}_3^T\mathbf{x}_a}.$$

We can simplify the expressions by multiplying with the denominator on both sides and then subtracting the right side from the left.

$$\begin{cases} h_{11}x_a + h_{12}y_a + h_{13} - h_{31}x_a x_b - h_{32}y_a x_b - h_{33}x_b = 0 \\ h_{21}x_a + h_{22}y_a + h_{23} - h_{31}x_a y_b - h_{32}y_a y_b - h_{33}y_b = 0 \end{cases}$$

Assuming a single matching pair of image points, we end up with a small linear system of equations, with 2 equations and 9 unknowns.

If we instead have multiple matching pairs of points between the two images, we can stack all the respective equations on top of each other. Since the equations are linear in terms of the parameters h_{ij}, we can write everything in matrix form

$$\underbrace{\begin{pmatrix} x_a^1 & y_a^1 & 1 & 0 & 0 & 0 & -x_a^1 x_b^1 & -y_a^1 x_b^1 & -x_b^1 \\ 0 & 0 & 0 & x_a^1 & y_a^1 & 1 & -x_a^1 y_b^1 & -y_a^1 y_b^1 & -y_b^1 \\ x_a^2 & y_a^2 & 1 & 0 & 0 & 0 & -x_a^2 x_b^2 & -y_a^2 x_b^2 & -x_b^2 \\ 0 & 0 & 0 & x_a^2 & y_a^2 & 1 & -x_a^2 y_b^2 & -y_a^2 y_b^2 & -y_b^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_a^n & y_a^n & 1 & 0 & 0 & 0 & -x_a^n x_b^n & -y_a^n x_b^n & -x_b^n \\ 0 & 0 & 0 & x_a^n & y_a^n & 1 & -x_a^n y_b^n & -y_a^n y_b^n & -y_b^n \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{pmatrix}}_{\mathbf{h}} = \mathbf{0},$$

where all the unknown parameters are stacked into the vector $\mathbf{h}$ and the large matrix A contains all the known coordinates of the points.

It is worth noting that all image points that originate from the positions of extracted image features are subject to noise, meaning that most of the elements in A can be expected to be noisy. With exactly 4 matching image points, we will get a perfect solution to $A\mathbf{h} = 0$, since we have 8 equations and 9 unknowns, but this solution will be affected by noise, in some cases heavily affected. This is a very typical case in computer vision. What seems to be an easy problem, may turn into something much more complicated, due to the existence of noise. In this case, it helps greatly if the 4 points are located in either corner of the images, but this is not something one can typically expect.

Instead of using a minimum set of point matches, it is preferable to exploit as many matches as possible to reduce the influence of noise, but then it becomes impossible to find a perfect solution. With more equations than unknown parameters, we cannot perfectly solve $A\mathbf{h} = 0$, except for the trivial solution $\mathbf{h} = 0$. What we instead do in practice is to find a set of parameters $\hat{\mathbf{h}}$ as close as possible to a perfect solution by minimizing the function

$$f(\mathbf{h}) = \|A\mathbf{h}\|^2$$

and then rearrange the optimum $\hat{\mathbf{h}} = \arg\min_{\mathbf{h}} \|A\mathbf{h}\|^2$ as a prediction of the homography H. To avoid the trivial solution $\mathbf{h} = 0$ we add the requirement that the constraint equation

$$g(\mathbf{h}) = \mathbf{h}^2 - 1 = 0$$

must be fulfilled, i.e. we only look for solutions $\mathbf{h}$ that have a length equal to 1.

A constrained system of equations, such as the one we have here, can be solved by minimizing a so-called *Lagrangian* function $J(\mathbf{h}, \lambda) = f(\mathbf{h}) - \lambda g(\mathbf{h})$, which includes the original function $f(\mathbf{h})$ and the constraint $g(\mathbf{h})$ weighted by an unknown factor λ. In our case we have the following:

$$J(\mathbf{h}, \lambda) = f(\mathbf{h}) - \lambda g(\mathbf{h}) = \|A\mathbf{h}\|^2 - \lambda(\mathbf{h}^2 - 1) = \mathbf{h}^T A^T A\mathbf{h} - \lambda(\mathbf{h}^T\mathbf{h} - 1),$$

where both $\mathbf{h}$ and λ have to be solved for. We can do this by first computing the derivatives with respect to $\mathbf{h}$ and λ and then setting these to zero. For the derivative of $J(\mathbf{h}, \lambda)$ with respect to λ, we simply get $g(\mathbf{h}) = \mathbf{h}^2 - 1 = 0$, which does not add anything new. However, for the other derivative, we get

$$\frac{\delta J(\hat{\mathbf{h}}, \lambda)}{\delta \mathbf{h}} = 2(A^T A\hat{\mathbf{h}} - \lambda\hat{\mathbf{h}}) = 0 \Rightarrow (A^T A)\hat{\mathbf{h}} = \lambda\hat{\mathbf{h}},$$

where we can identify λ as the smallest eigenvalue of $A^T A$ for which the corresponding eigenvector is the optimum $\hat{\mathbf{h}}$ we are searching for.

2.2 Implementation

We now have a recipe for finding the homography between two images taken of the same scene, where the scene is assumed to be dominated by a flat surface.

```
Find homography H between two images:
%   Extract local image features in each image
%   Match features between images and identify matching pairs
%   Use image positions of matching pairs to create a matrix A
%   Compute square of A, C = transpose(A) * A
%   Find eigenvector h of smallest eigenvalue lambda of C
%   Rearrange 9-vector h into a 3x3 homography H
```

You will in this lab implement this in Python, or at least parts of it. For extracting and matching SIFT features there is already a prepared function

```
extract_and_match_SIFT(img1, img2, num)
```

which returns two matching lists of a maximum of `num` image features from the images `img1` and `img2`.
However, in most cases, it is better to start with a randomly generated set of points, for you to have full control over the distribution of points, while you implement and debug your code. There is another function for your convenience,

```
generate_2d_points(num, noutliers, noise = 0.5, vergence = 10.0, focal = 1000)
```

a function that generates points on a flat 3D surface facing the cameras and projects these onto the camera images. Here `noise` is the standard deviation of the noise added to the generated image points (in pixels), `vergence` is the vergence angle between the two cameras (in degrees) and `focal` is the focal length of the camera (in pixels). The final parameter, `noutliers`, is the number of *outliers* injected into the data set, which can be used to test the implementation's robustness to complete mismatches, something that often occurs in practice.
As can be seen in Figure 1, image points, either extracted from real images or randomly generated, can be visualized with the following command:

```
draw_matches(pts1, pts2, img1 = None, img2 = None)
```

The two arrays `pts1` and `pts2` contain N matching point pairs, which will be drawn on top of the images `img1` and `img2`, if these are assigned. Otherwise, the points are drawn without images, such as when points have been randomly generated.

Task 1: Given two matched sets of features, `pts1` and `pts2`, write a function in Python

```
find_homography(pts1, pts2)
```

that computes an estimate of a homography H, as described above in section 2.1. A skeleton of such a function can be found in file `homography.py` provided in the course library. The function should work for an arbitrary number of matching points, as long as there are at least 4. Try out your implemented function, initially with randomly generated feature matches and then afterward with real extracted and matched SIFT features, once the function works properly.

Ensure that no outliers, i.e. paired image features that do not come from the same point in 3D space, are added to your randomly generated image points by setting `noutliers = 0`. If you set `noise = 0.5`, you get a realistic noise level of half a pixel in standard deviation. Most modern feature extractors can do better than that, but not much better.

Question 1: To compute a homography, what are the computational steps involved? Why do you think we are interested in finding the smallest eigenvalue, but not in the other eigenvalues?

Using `num = 100` generated features, start with a noise level of `noise = 0.1`, then gradually increase the noise level and study the homography error reported by the function `homography_error()`. Make several different runs before drawing any conclusions to see the variation due to the randomization.

Question 2: How much image noise can you typically have before the error becomes significant? Is it preferable to use a few or many feature points when creating A?

Set the noise level to `noise = 0.5` and then gradually increase the number of outliers inserted into the dataset, `noutliers`, while studying the reported error. In a practical scenario using real data, completely avoiding outliers is almost impossible, simply because the world consists of a lot of repetitive structures that look very similar, especially in man-made environments. In fact, due to changes in viewing directions and varying lighting conditions, an incorrect match might well look more similar to a point than its real match.

Question 3: How many outliers can you typically have before the errors start to increase too much? Given what you try to optimize, how can the sensitivity to outliers be explained?

To minimize the risk of introducing an outlier in the computations, one might use as few features as possible, instead of the full set of features. In this case, we need at least 4 feature matches, given that each match provides two equations, and the unknown parameter vector $\mathbf{h}$ has 8 degrees of freedom, assuming that $\|\mathbf{h}\| = 1$. However, if we randomly sample 4 features from the full set of matched features, we might still run into trouble, if the features are poorly distributed. For example, if 3 points are collinear, the system of equations will collapse, A^TA will have multiple eigenvalues close to zero, and we will have many possible solutions. Then there is still the risk that the set includes an outlier.

To overcome this problem, we will use an algorithm called Random Sampling Consensus (RANSAC). The idea is to generate many different sets of 4 features each, compute an estimated parameter vector $\hat{\mathbf{h}}$ for each such set, and then test these estimates against the full set of feature matches. An algorithm can be summarized as follows:

```
Find parameters for a homography using RANSAC
%   Loop niter times:
%       Generate a minimum set of 4 random feature matches
%       Find homography H using these matches
%       Count the number of inliers among all features using homography H
%       Keep track of homography with the highest number of inliers
%   Return estimated homography with the highest number of inliers
```

A feature match $(\mathbf{x}_a, \mathbf{x}_b)$ is considered an *inlier* with respect to an estimated homography H if the error of that match is smaller than some threshold `thresh`. In this case, we measure the Euclidean distance between the real position in the other image and the position predicted using the homography, and compute the error

$$err = \sqrt{\left(x_b - \frac{\mathbf{h}_1^T\mathbf{x}_a}{\mathbf{h}_3^T\mathbf{x}_a}\right)^2 + \left(y_b - \frac{\mathbf{h}_2^T\mathbf{x}_a}{\mathbf{h}_3^T\mathbf{x}_a}\right)^2}$$

which is then compared to `thresh` to determine if $(\mathbf{x}_a, \mathbf{x}_b)$ is an inlier or not. There is a function

```
count_homography_inliers(H, pts1, pts2, thresh = 1.0)
```

in `homography.py` that does this for you. The function returns the number of inliers, `ninliers`, and a vector of errors, `errors`, one for each point in `pts1`. A threshold of `thresh = 1.0` is reasonable given typical image noise, but it might be changed depending on the quality of the images.

Task 2: Using the same sets of features as before, `pts1` and `pts2`, write a Python function

```
find_homography_RANSAC(pts1, pts2, niter = 100, thresh = 1.0)
```

that uses RANSAC to compute an estimate of a homography, as described in the pseudocode above. The function should iterate `niter` times and compute a homography for each iteration. The number of inliers is computed for each such homography, with respect to a threshold `thresh` on the error per feature. The function should return (`Hbest, nlinliers, errors`), where `Hbest` is the homography with the highest number of inliers, `ninliers`, and `errors` is a list of errors for each feature match. Try out the function first with randomly generated and later with real feature matches.

With the list of errors in `errors`, it is possible to identify the outliers and recompute a better estimate of the homography by calling `find_homography()` with all features, but with the outliers excluded. If the homography is correctly computed for the features in Fig. 1, you should get a result such as the one in Fig. 2 using the function `draw_homography(img1, img2, H)`. As can be seen in the figure, the two images overlap, despite being taken from quite different orientations.

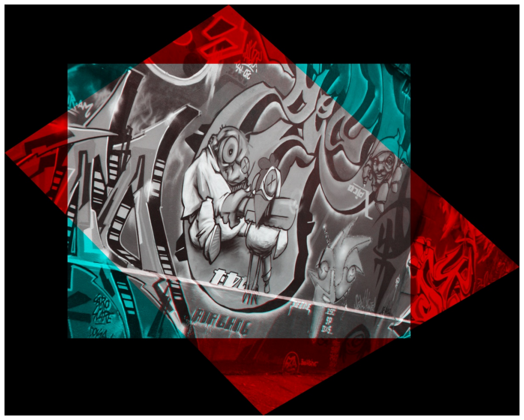

Figure 2: Two images combined, with one of the images warped using a homography.

Using random feature matches obtained with `generate_2d_points()`, gradually increase the number of outliers in the dataset, `noutliers`, and study how robust the prediction is compared to the original version without RANSAC. As the fraction of outliers increases, it becomes more and more difficult to sample a subset of features that contain no outliers. To overcome this difficulty, increase the number of RANSAC iterations by modifying `niter` accordingly.

Question 4: If you have `num = 100` points in total and `noutliers = 50` outliers, how many iterations do you typically have to do until you find a solution with a small enough homography error?

Question 5: Why does RANSAC make the homography estimation so much more robust compared to the original implementation? Try to come up with a short, but easily understood, explanation.

Try other images in the course library, which are not necessarily dominated by a large flat surface, such as those in `books1.jpg` and `books2.jpg`. To ensure that you make enough trials, use a relatively large number of iterations, e.g. by setting `niter = 10000`.

Question 6: What do you observe in the results? Do you get an overlap similar to what is shown in Fig. 2? Can you somehow exploit the results even in regions for which there is no overlap?

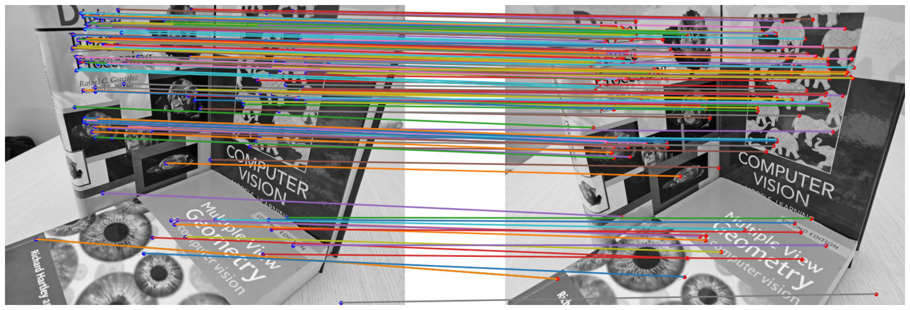

Figure 3: Feature matches between two images of a world that is not flat.

3 Recovering fundamental matrices

When the world is not flat, such as in Fig. 3, we can no longer use a homography to relate matching image points between the two images. An alternative is instead to use the so-called *epipilar constraint*, according to which an image feature in one image can find its corresponding match in the other image alone a particular line, known as the *epipolar line*. The constraint comes from the fact that the two image points, such as q and q' in Fig. 4 when viewed as 3D points, will be located on the same plane as the two optical centers, o and o'. Three points are enough to determine the plane, while the fourth point can be found on the same plane or along a line if it also has to be located on an image plane. In Fig. 4 that would be the epipolar line that passes through q' and the epipole e'.

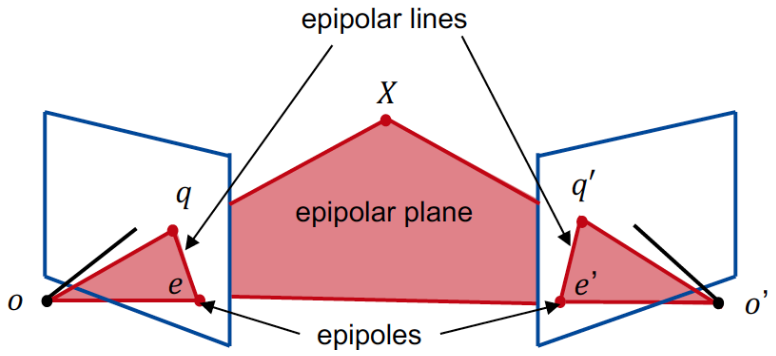

Figure 4: The epipolar geometry according to which q' can be found on the plane given by q, o, and o', which intersects the image plane along a line, the epipolar line.

3.1 Mathematical background

Two matching image points $\mathbf{x}_a$ and $\mathbf{x}_b$ can be related through the epipolar constraint

$$\mathbf{x}_b^T F \mathbf{x}_a = 0,$$

where F is a fundamental matrix that can be written in terms of four components

$$F = K_b^{-T} R [\mathbf{t}]_\times K_a^{-1}.$$

In this equation K_a and K_b are the respective camera matrices, R is the relative rotation expressed as a rotation matrix between the coordinate systems of the two cameras, and $\mathbf{t}$ is the relative translation; in Fig.4 this would be the difference in position between o and o'. Here $[\mathbf{t}]_\times$ denotes a skew-symmetric matrix that corresponds to what happens when you do a cross product with $\mathbf{t}$, i.e. $[\mathbf{t}]_\times \mathbf{x} = \mathbf{t} \times \mathbf{x}$, but written in a more convenient matrix form.

Fully recovering all four components K_a, K_b, R, and $[\mathbf{t}]_\times$, is very complicated and beyond the scope of this lab. We will instead assume that the camera matrices have an easier form

$$K_a = K_b = K = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

where only the focal length f needs to be considered. However, since the true camera matrices are likely to differ from K, we should expect considerable errors, if we later attempt to produce a 3D reconstruction using the recovered relative rotation and translation. For better 3D reconstruction, the cameras must be properly calibrated, which is difficult if you no longer have access to the original camera taking the images.

If we write the fundamental matrix in terms of its elements,

$$F = \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix}$$

the epipolar constraint that relates the two matching points $\mathbf{x}_a$ and $\mathbf{x}_b$ can be written as

$$\mathbf{x}_b^T F \mathbf{x}_a = \begin{pmatrix} x_b & y_b & 1 \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} \begin{pmatrix} x_a \\ y_a \\ 1 \end{pmatrix} =$$

$$= f_{11}x_bx_a + f_{12}x_by_a + f_{13}x_b + f_{21}y_bx_a + f_{22}y_by_a + f_{23}y_b + f_{31}x_a + f_{32}y_a + f_{33} = 0.$$

If $\mathbf{x}_a$ is known, the point $\mathbf{x}_b$ in the other image can be found along the epipolar line $\mathbf{x}_b^T \mathbf{l}_b = 0$ for which the coordinates are given by $\mathbf{l}_b = F\mathbf{x}_a$. As we saw earlier, each point match resulted in two equations when we used homographies, while we only had one equation for fundamental matrices. Similarly to the case of homographies, we can write the unknown elements of F as a vector of 9 elements,

$$\mathbf{f} = (f_{11}, f_{12}, f_{13}, f_{21}, f_{22}, f_{23}, f_{31}, f_{32}, f_{33})^T$$

and stack up the equations corresponding to multiple point matches in a linear system of equations

$$\underbrace{\begin{pmatrix} x_b^1x_a^1 & x_b^1y_a^1 & x_b^1 & y_b^1x_a^1 & y_b^1y_a^1 & y_b^1 & x_a^1 & y_a^1 & 1 \\ x_b^2x_a^2 & x_b^2y_a^2 & x_b^2 & y_b^2x_a^2 & y_b^2y_a^2 & y_b^2 & x_a^2 & y_a^2 & 1 \\ x_b^3x_a^3 & x_b^3y_a^3 & x_b^3 & y_b^3x_a^3 & y_b^3y_a^3 & y_b^3 & x_a^3 & y_a^3 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_b^nx_a^n & x_b^ny_a^n & x_b^n & y_b^nx_a^n & y_b^ny_a^n & y_b^n & x_a^n & y_a^n & 1 \end{pmatrix}}_{B} \underbrace{\begin{pmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{pmatrix}}_{\mathbf{f}} = \mathbf{0}.$$

With one equation per match and 9 unknown parameters, we need at least 8 matches to determine F, given that we are using homogeneous coordinates and may use the additional constraint

$$g(\mathbf{f}) = \mathbf{f}^2 - 1 = 0.$$

With possibly more than 8 matching points, there might be no solution to $B\mathbf{f} = 0$. We may instead search for a parameter set $\hat{\mathbf{f}}$ that minimizes

$$f(\mathbf{f}) = \|B\mathbf{f}\|^2$$

while satisfying $g(\mathbf{f}) = 0$, which is equivalent to minimizing the Lagrangian function

$$J(\mathbf{f}, \lambda) = f(\mathbf{f}) - \lambda g(\mathbf{f}) = \|B\mathbf{f}\|^2 - \lambda(\mathbf{f}^2 - 1) = \mathbf{f}^T B^T B\mathbf{f} - \lambda(\mathbf{f}^T\mathbf{f} - 1).$$

Just like in the case of homographies before, it boils down to the problem of finding the eigenvector $\hat{\mathbf{f}}$ that corresponds to the smallest eigenvalue of square matrix $B^T B$. Thus, even if we had two quite different relations between two images, both relations were found in very similar manners.

3.2 Implementation

In this exercise, you will apply the approach described above to develop a method for estimating a fundamental matrix with image features from two cameras covering the same scene from two different orientations, much similar to what you earlier did for homographies. It is again recommended that you use randomly generated points during the development, before you later try the method on real image features extracted and matched with `extract_and_match_SIFT()`. Previously, we used the function `generate_2d_points()`, with the assumption that 3D points are located on a plane, to randomly generate feature matches containing a controllable amount of errors, either errors due to image noise or mismatches.

In this case, however, we do not have a planar assumption but generate 3D points in a cube located in front of both cameras using the function

```
generate_3d_points(num, noutliers, noise, vergence, focal, spherical = False)
```

with the same kinds of controllable errors, when the points are projected to the respective images. This function returns two sets of image points in the respective cameras and the true fundamental matrix F between the cameras. If you set the parameter `spherical = True`, the points will be generated on a 3D sphere, which ensures that points are well spread and easier to visualize, which can be seen if you call `draw_matches()`.

Task 3: Given two matched sets of features, `pts1` and `pts2`, write a function in Python

```
find_fmatrix(pts1, pts2, normalize = False)
```

that estimates a fundamental matrix F between two unknown cameras, using the approach explained in Section 3.1. A basic outline of this function can be found in the file `fmatrix.py` in the course library. The function should support an arbitrary number of matching points, with a minimum of 8. Test your implementation initially with randomly generated feature matches, and once it works correctly, try it with real SIFT features that have been extracted and matched.

Unlike the earlier case of homographies, the linear system of equations can too easily become ill-conditioned, if the selected features are not properly spread in image space. A common approach to overcome this is to first normalize the points by centering them around the origin with unit variance, then estimate the fundamental matrix, and finally adjust the matrix to reverse the normalization. There is already code available in the template for doing this, which can be selected by setting the option `normalize = True`.

Question 7: What are the similarities and differences between the code you wrote for estimating a fundamental matrix and a homography?

Repeat the experiments you did previously by setting the number of features to `num = 100` and noise level to `noise = 0.5`, and then gradually increase the number of outliers, `noutliers`, until you start getting too large errors in the estimated fundamental matrix as reported by the function `fmatrix_error()`.

Question 8: If you have 100 generated feature matches, how many outliers can you typically have before you get significant errors in the estimated fundamental matrices? Can you see a difference with normalization versus without?

To get reliable enough results that could later be used to extract the 3D structure of the scene we are looking at, we need to identify mismatches (outliers) and exclude these from further consideration. RANSAC with random sampling of minimum sets of feature matches is yet again the method of choice to do this. With 9 unknown parameters in $\mathbf{f}$ to be determined, one equation per feature match, including a constraint equation $g(\mathbf{f}) = 0$, a minimum set of features includes 8 feature matches.

In order for outliers to be identified and excluded from the dataset, we need an error measure. For the positions of a matching pair of feature points to be perfect, the corresponding points have to lie along their respective epipolar lines in both images, which is true only if

$$\mathbf{l}_a^T \mathbf{x}_a = 0 \quad \text{and} \quad \mathbf{l}_b^T \mathbf{x}_b = 0,$$

where the line coefficients are given by $\mathbf{l}_a = F^T \mathbf{x}_b = (l_{a,x}, l_{a,y}, l_{a,w})^T$ and $\mathbf{l}_b = F\mathbf{x}_a = (l_{b,x}, l_{b,y}, l_{b,w})^T$. Howewer, due to noise $\mathbf{l}_a^T \mathbf{x}_a$ and $\mathbf{l}_b^T \mathbf{x}_b$ are typically not zero, but contain errors. The points do not lie exactly on the lines, but there is some distance between them. A commonly used measure to quantify this distance is the Symmetric Epipolar Distance,

$$err^2 = \frac{(\mathbf{l}_a^T \mathbf{x}_a)^2}{l_{a,x}^2 + l_{a,y}^2} + \frac{(\mathbf{l}_b^T \mathbf{x}_b)^2}{l_{b,x}^2 + l_{b,y}^2},$$

which treats the errors in the respective images in a symmetric manner. For distances to be measured in pixels to allow for thresholding, each squared error has been scaled by the magnitude of the first two line coefficients. In the file `fmatrix.py` there is a function

```
count_fmatrix_inliers(F, pts1, pts2, thresh = 1.0)
```

that uses this measure to classify feature matches into inliers and outliers. The function returns the number of inliers, `ninliers`, and a vector of errors, `errors`, one for each point in `pts1`.

Task 4: Similarly to what you earlier did for homographies, write a Python function

```
find_fmatrix_RANSAC(pts1, pts2, niter = 100, thresh = 1.0)
```

that uses RANSAC to compute an estimate of a fundamental matrix given two matched feature sets, `pts1` and `pts2`. The function should return `(Fbest, ninliers, errors)`, where `Fbest` is the fundamental matrix with the most inliers, `ninliers` is the count of these inliers, and `errors` is a list of errors for each feature match. Test the function first with randomly generated feature matches, and then with real matches. In each iteration, use `count_fmatrix_inliers()` to count the number of inliers for each generated hypothesis.

Question 9: If you have `num = 100` points in total and `noutliers = 50` outliers, how many iterations do you need until you find a solution with a small enough error in the estimated fundamental matrix? What is the difference compared to when you tried to find homographies?

Try some real images in the course library, such as `desk1.jpg` and `desk2.jpg` and use `draw_matches()` to visualize the results. Ensure you let RANSAC run with enough iterations for the maximum number of inliers to stabilize.

Question 10: How large a fraction of all original feature pairs seems to be classified as inliers? Do you see any obvious mismatch being classified as inlier and thus not removed? How much do the results depend on which pair of images you tried?

4 Extracting projection matrices

Our ultimate goal is not just to find a fundamental matrix F, but to recover the projection matrices of the respective cameras and finally a 3D reconstruction of the observed points in the world. With no other world coordinate system given, we could use the first camera to define a coordinate system in the world, resulting in a fairly trivial projection matrix for this camera,

$$M_a = K_a(I \mid 0) = K_a \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

where K is the camera matrix that we assume is given, but in many cases is not. The corresponding projection matrix for the other camera, assuming a relative rotation R and translation $\mathbf{t} = (t_x, t_y, t_z)^T$ between the two cameras, would then be given by

$$M_b = K_b R(I \mid \mathbf{t}) = K_b R \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \end{pmatrix}.$$

We thus need to recover the rotation R and translation $\mathbf{t}$ through a decomposition of the estimated fundamental matrix $\hat{F}$, which is not easy since there are multiple possible solutions and $\hat{F}$ might contain severe errors due to noise compared to the correct fundamental matrix F. You typically do this through a singular value decomposition (SVD) of

$$\hat{F}' = K_b^T \hat{F} K_a.$$

Here we have multiplied $\hat{F}$ by the camera matrices from both sides to get a fundamental matrix $\hat{F}'$ in terms of image coordinates when F was originally expressed in pixel coordinates. In image coordinates the camera has a focal length $f = 1$ and the center of projection is exactly in the center of the image, simplifying further computations.

With U and V being two orthogonal matrices and Σ a diagonal matrix, the singular value decomposition of $\hat{F}'$ is given by

$$\hat{F}' = U \Sigma V^T.$$

From this, we can decompose $\hat{F}'$ into an estimated rotation

$$\hat{R} = U W V^T$$

and translation, expressed in terms of a skew-symmetric matrix,

$$[\hat{\mathbf{t}}]_\times = V Z V^T,$$

with the auxiliary matrices

$$W = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \text{and } Z = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Suppose there were no errors in the computations. Then we should be able to express the estimated fundamental matrix as $\hat{F}' \simeq \hat{R}[\hat{\mathbf{t}}]_\times$, similarly to what we already had at the beginning of Section 3.1, where we wrote the fundamental matrix as $F = K_b^{-T} R[\mathbf{t}]_\times K_a^{-1}$, which becomes the essential matrix

$$F' = R[\mathbf{t}]_\times$$

in image coordinates. If we write the product of the estimated rotation and translation, we get

$$\hat{R}[\hat{\mathbf{t}}]_\times = UWV^T VZV^T = UWZV^T = U \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} V^T,$$

which would the same as $\hat{F}' = U\Sigma V^T$, if $\Sigma \simeq WZ$, i.e. equivalence up to scale. Thus for a correct fundamental matrix, one of the singular values (the elements in the diagonal of Σ) should be 0 and the other two should be equal, but not necessarily 1, because the scales of matrices do not matter with homogeneous coordinates. In practice though, the singular values can be quite different due to noise. However, as a sanity check, one might check whether the two largest singular values are similar in scale and significantly larger than the smallest.

However, there is another problem. If we transpose either W or Z, the product $UWZV^T$ will be the same, except possibly for a change in sign, which does not matter with homogeneous coordinates. Thus we have two alternative rotations

$$\hat{R}_1 = UWV^T \quad \text{or } \hat{R}_2 = UW^TV^T$$

and two alternative translations

$$[\hat{\mathbf{t}}_1]_\times = VZV^T \text{ or } [\hat{\mathbf{t}}_2]_\times = VZ^TV^T,$$

four combinations in total. It is not easy to see which of these combinations is the correct one, without assuming information about the conditions under which the images were taken, information that you might not have. What you typically do instead is to create four different projection matrices

$$M_b^{ij} = K_b R_i (I \mid \mathbf{t}_j), \quad i, j \in \{1, 2\}$$

and then go the whole way to 3D reconstruction and finally check whether most reconstructed 3D points lie in front of both cameras. For only one of the M_b^{ij} matrices this is true, whereas in the other three cases, the points will lie behind at least one of the two cameras, which is not physically possible. Due to noise and mismatches, however, some reconstructed points might still be placed behind one of the cameras. This can never be avoided completely.

4.1 Triangulation

Assume we have a camera with projection matrix M, expressed in terms of its rows $\mathbf{m}_1^T$, $\mathbf{m}_2^T$, and $\mathbf{m}_3^T$, and want to reconstruct a 3D point $\mathbf{X}$. The image projection of the point can then be written as

$$\mathbf{x} \simeq M\mathbf{X} = \begin{pmatrix} \mathbf{m}_1^T \\ \mathbf{m}_2^T \\ \mathbf{m}_3^T \end{pmatrix} \mathbf{X} = \begin{pmatrix} \mathbf{m}_1^T \\ \mathbf{m}_2^T \\ \mathbf{m}_3^T \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

or

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \mathbf{m}_1^T\mathbf{X}/\mathbf{m}_3^T\mathbf{X} \\ \mathbf{m}_2^T\mathbf{X}/\mathbf{m}_3^T\mathbf{X} \end{pmatrix}$$

when expressed in the known Euclidean coordinates of $\mathbf{x}$. If we multiply both sides with $\mathbf{m}_3^T\mathbf{X}$ and move everything to the left side, we get a homogeneous linear system with two equations

$$\begin{pmatrix} x\mathbf{m}_3^T\mathbf{X} \\ y\mathbf{m}_3^T\mathbf{X} \end{pmatrix} = \begin{pmatrix} \mathbf{m}_1^T\mathbf{X} \\ \mathbf{m}_2^T\mathbf{X} \end{pmatrix} \Rightarrow \begin{pmatrix} x\mathbf{m}_3^T - \mathbf{m}_1^T \\ y\mathbf{m}_3^T - \mathbf{m}_2^T \end{pmatrix} \mathbf{X} = \mathbf{0}.$$

The two equations define a line passing through the camera center and the image point $\mathbf{x}$. Along this line $\mathbf{X}$ should be located. However, it could be anywhere on the line and we would need another camera and corresponding image point to determine exactly where the point is.

Let us thus assume that we have two different cameras with projection matrices M_a and M_b, and in these cameras, we have detected two corresponding projection points $\mathbf{x}_a$ and $\mathbf{x}_b$. We now have four linear equations for which the known constants can be stacked into a 4×4 matrix G,

$$\underbrace{\begin{pmatrix} x_a\mathbf{m}_{a,3}^T - \mathbf{m}_{a,1}^T \\ y_a\mathbf{m}_{a,3}^T - \mathbf{m}_{a,2}^T \\ x_b\mathbf{m}_{b,3}^T - \mathbf{m}_{b,1}^T \\ y_b\mathbf{m}_{b,3}^T - \mathbf{m}_{b,2}^T \end{pmatrix}}_{G} \mathbf{X} = \mathbf{0}.$$

Unfortunately, there might be no solution to this system, since that would imply that G has an eigenvalue equal to zero with $\mathbf{X}$ being the eigenvector. Due to noise, however, this is very unlikely to happen. In effect, we are trying to find the intersection point of two projection lines, one for each camera, but the two lines might not at all intersect. The alternative is to find a 3D point that lies as close as possible to the two lines.

Once again, we use constrained least square minimization to estimate the 3D point position $\mathbf{X}$. We do this by minimizing the objective

$$f(\mathbf{X}) = \|G\mathbf{X}\|^2,$$

with the constraint equation

$$g(\mathbf{X}) = \mathbf{X}^2 - 1 = 0,$$

which in turn leads to the Lagrangian function

$$J(\mathbf{X}, \lambda) = \|G\mathbf{X}\|^2 - \lambda(\mathbf{X}^2 - 1) = \mathbf{X}^T G^T G\mathbf{X} - \lambda(\mathbf{X}^T\mathbf{X} - 1).$$

Similarly to the other cases we have seen so far when we tried to estimate homographies and fundamental matrices, the estimated 3D point $\hat{\mathbf{X}}$ is equal to the eigenvector corresponding to the smallest eigenvalue of the square matrix G^TG. Once we have the estimated 3D point $\hat{\mathbf{X}}$, we can express it in Euclidean coordinates $(X, Y, Z)^T$. If $Z > 0$, the point lies in front of the camera. This can be used to test which of the four possible projection matrices M_b^{ij} is the correct one when M_a is the projection matrix of the other camera.

4.2 Implementation

The sections above describe a scheme for finding the projection matrix M_b. The fundamental matrix F is first decomposed into two possible rotation matrices and two possible translations. By composing a projection matrix and reconstructing the 3D points for each combination of rotation and translation, and checking which of these leads to most 3D points being placed in front of both cameras, the correct combination can be identified. In the course library in `triangulation.py`, there is a function called

```
    projection_from_fmatrix(F, pts1, pts2, focal = 1000)
```

that computes the projection matrix M_b and a set of reconstructed 3D points `pts3d` using this scheme, with the assumption that M_a defines the world coordinate system and is thus fixed to $K_a(I \mid 0)$. As inputs, the function expects an estimated fundamental matrix `F`, two matching point sets preferably without outliers, `pts1` and `pts2`, and a focal length. The focal length is required to convert points from pixel to image coordinates, but without proper calibration camera calibration, this conversion may not be entirely accurate.

Task 5: The function `projection_from_fmatrix()` is still incomplete, since it calls the function

```
triangulate(M1, M2, pts1, pts2)
```

which has unfortunately not yet been implemented. It is your task to do this, using the procedure described in Section 4.1. Given the two projection matrices `M1` and `M2`, the function should loop over all the matching features in `pts1` and `pts2` and for each feature pair, find the closest 3D point between the respective projection lines, by solving a system of equations. The output should be an array of 3D points `pts3d` in Euclidean coordinates. Note that since points will originally be in homogeneous coordinates after solving the systems of equations, they have to be converted to Euclidean coordinates by dividing by the last coordinates.

In the main function `run_triangulation()`, the image features are centered before being used to estimate the fundamental matrix, create the projection matrices and finally triangulate the 3D points, even if the original image positions are still used for visualization. The reason for centering the points is to increase the robustness by reducing the bias due to the fact that the mean point position is not the origin. For the sake of the exercise this does not really matter, but it explains the existence of two kinds of points in the code.

Visualizing the final reconstructed 3D points can be very difficult, especially if you want to make a judgment on the quality of the reconstructions. In the course library, there are a couple of functions that can be used for visualization. With the function `draw_2d_points(pts1, pts2, colors, img1, img2)`, the extracted 2D points are drawn on top of the two images in colors that can be based on the 3D coordinates, e.g. the Z-value of points normalized to a range $[0, 1]$, such as to the left in Figure 5.

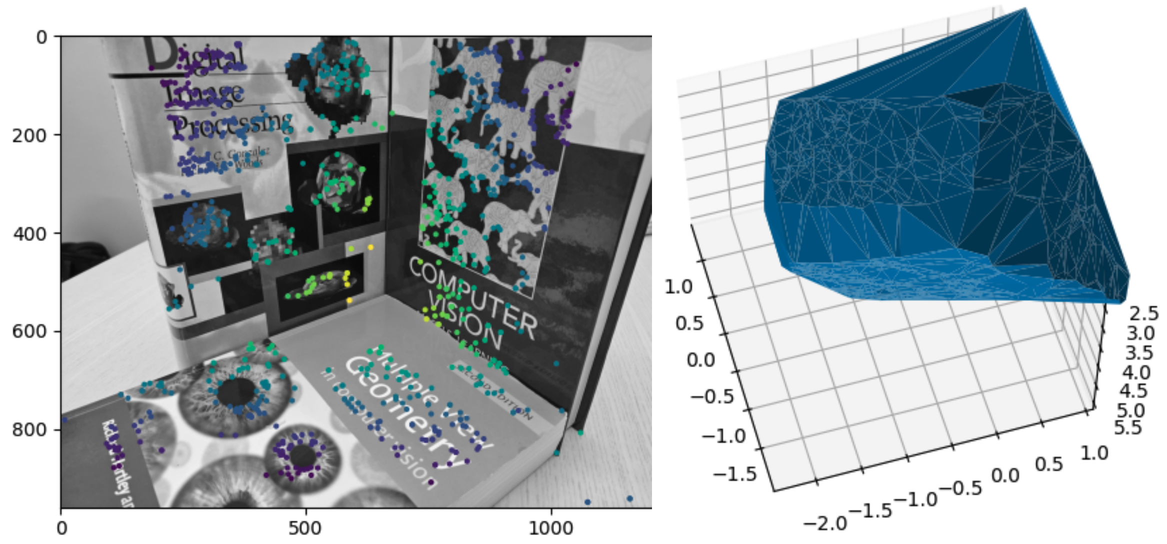

Figure 5: Examples of reconstructed 3D points from matched image features visualized using the functions `draw_2d_points()` and `draw_triangles()`.

The function `draw_3d_points(pts3d, color)` simply draws a cloud of 3D points in a given color, but with no reference to the original images, it may be difficult to draw any conclusions from this.

Another function is `draw_triangles(pts2d, pts3d)` which uses a set of 2D points `pts2d` to create a triangulation which is drawn in 3D space using the respective 3D coordinates in `pts3d`, resulting in illustrations such as the one to the right in Figure 5.

Experiment with different image pairs and examine the visualizations of the reconstructed 3D points. Since the cameras are uncalibrated, errors are expected due to incorrect camera parameters, particularly the focal length. To assess its impact on the reconstruction, adjust the focal length to achieve the best possible results.

Question 11: If we assume the angle between two books in `books1.jpg` is 90°, is it possible to tell approximately how large the real focal length is by varying the focal length used for reconstruction?

Question 12: How sensitive is the reconstruction to the choice of images and the characteristics of the 3D scene? Can it handle scenes primarily consisting of a flat surface, such as the scenes for which we earlier used homographies?

Question 13: What do you suggest one could do to further improve the reconstructions? Could we somehow build on what has been done so far in the lab to get considerably better results?

Laboration 3 in DD2423 Image Analysis and Computer Vision

. . . .
Student's personal number and name (filled in by student)

. . . .
Approved on (date) Course assistant