3D Vision: Two-view Correspondence, 3D Reconstruction and Depth-based Segmentation (E5ADSB)

Janus Bo Andersen [1]

11th December 2020

[1]Student id: JA67494. Mail: ja67494@post.au.dk or janus@janusboandersen.dk

Contents

Summary

In this project, a 3D vision pipeline is developed that uses two-view correspondences to perform sparse reconstruction of a 3D scene and subsequent depth-based image segmentation. Samples from the pipeline are shown on the cover page.

Particular emphasis is placed on solving the correspondence problem of tracking feature points across two views. This is the first part of the project. Here, a full pipeline is developed and implemented to handle the correspondence problem: Feature detection by the Harris corner detector algorithm, feature description by the BRIEF-32 descriptor with a Gaussian test geometry, and brute-force feature tracking using the Hamming distance and Lowe's Ratio Test.

The second part of the project is an overview of image formation in a camera modelled using homogeneous coordinates and transformation groups. Here, 2D homographies are demonstrated, which directly leads to the calibration process for the author's camera.

In the third part, using the implemented algorithms and images from the calibrated camera, relative camera poses are estimated from two views. Using the correspondence points, the fundamental matrix from epipolar geometry is estimated, which enables recovery of the relative pose. 3D points in the world frame are then recovered by triangulating the correspondence points and camera positions.

The fourth and final part of the project demonstrates segmentation of an image using the recovered 3D points. Here, one of the images with established correspondences is thresholded using Otsu's method, and the resulting binary image is then morphologically closed. This gives large connected regions that can be labelled. The 3D scene is differentiated into near and far using the mean L_2 norm of the 3D vectors, and the entire connected region in the image corresponding to the "far" points is segmented out.

The project is concluded with an overview of next steps and possible improvements, as well as a conclusion.

1. Introduction and motivation

When 3-dimensional objects are imaged using a camera, they are projected onto the 2-dimensional surface of the film or the sensor. This gives images without depth information and with "distortions" due to linear perspective.

Relying on experience, a human can often (but not always) accurately distinguish distances, sizes and infer geometric properties of objects in images.

Computers, however, can not easily process multidimensional signals to infer 3D information. But it is very useful to have depth/ranging data, and to be able to perform measurements of geometric properties, so mathematical techniques and signal processing algorithms have been developed to computationally obtain or recover these from one or more images.

This project works with the tasks involved in recovering 3-dimensional information from images, including the rather involved pre-processing steps required to do so.

1.1 Background

The first key insight for solving this problem is perspective and projection.

Figure 1.1: Parallel lines along the train tracks, the trackbed and the tree tops all converge to one vanishing point at the horizon at infinity. The tracks and the sleepers are positioned at 90° angles to each other, but due to the perspective of the camera, they do not appear orthogonal.

When looking at things (or taking pictures), one will notice that objects appear smaller as they move further away and that angles between lines change as the viewpoint / perspective is changed. One can also notice that all parallel lines converge to a single point on the horizon (the vanishing point), and that all lines converge toward the same horizon, but not necessarily the same vanishing point. These observations are illustrated in fig. 1.1.

In the Renaissance, the mathematical method of linear geometric perspective was (re)discovered by the Florentine architect and engineer Brunelleschi. He devised a method for drawing buildings and objects with realistic perspective, like in the figure of the train tracks. With his method, visual depictions of the world could now accurately be shown in a perspective similar to how we experience it using our eyes.

A little later, in the 16th century, Leonardo da Vinci used a camera obscura (pinhole camera) to study optics and the human visual system[1]. It produces a similar projection effect, as rays of light reflected by objects in the outside world enter a central pinhole into a dark chamber (lat.: camera obscura). The rays are projected and focused onto a back wall or transluscent surface, revealing the inverted image. Fig. 1.2 shows a drawing of a pinhole camera.

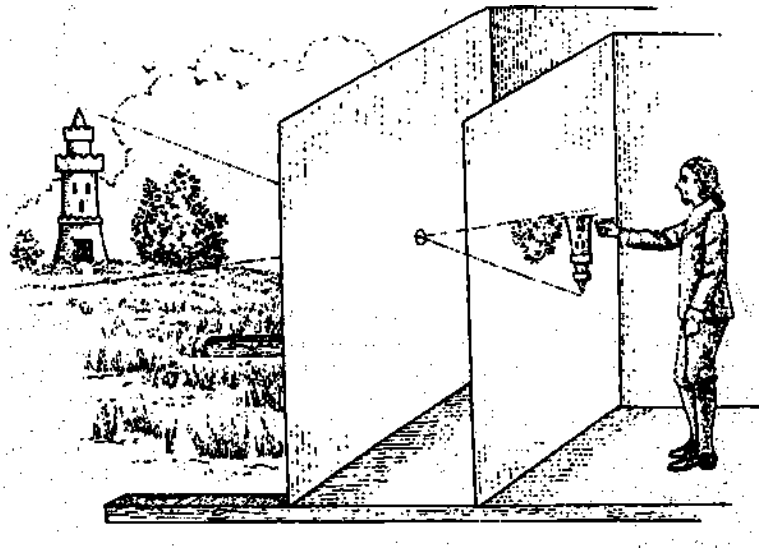

Figure 1.2: The pinhole camera is an important abstraction for modeling the image formation in a modern lens-based camera.

Today, the pinhole camera and Brunelleschi's work on linear perspective remain very important. The former as a simple model of image formation in camera-based imaging systems. The latter for understanding perspective projections, and as a motivation for projective geometry. Both are key background for solving the 3D problems in this project.

[1] This device has a long history, from early religious symbolism, to observing solar eclipses.

2. Problem statement

The requirements for this project are refined from the original proposal, and adds the segmentation-step explicitly. This project must

- Implement a prototype processing pipeline, which must
  - Take as input a set of 2D images of an object or scene viewed from different angles (two-views).
  - Output a partially reconstructed 3D model of the object or scene.
  - Perform depth-based segmentation of an image from the 2D image set.
- Implement pre-processing blocks, which must
  - Locate key features in the images, and
  - Track key features across the image set, in order to
  - Enable (fit into) the subsequent projective geometry / reconstruction steps.
- The processing must be automatic, and not rely on human input to identify features in images.

2.1 Ambition and delimitations

2.1.1 Ambitions in the project

The theoretical background required to master end-to-end calibrated 3D reconstruction seems to traditionally require a sequence of two advanced undergraduate level courses or graduate level courses in computer vision, see e.g. [1, p. vii-ix]. So the ambition in this project is to understand, implement and demonstrate *only some* of the key, foundational techniques and algorithms in the field.

In particular, the focus in this project is on implementing solutions to the correspondence problem and on using the reconstructed sparse 3D points to perform depth-based image segmentation. These topics are fully relevant within the course description of E5ADSB.

The projective geometry and linear algebra used for finding the F-matrix and performing the triangulation is also interesting, but a prioritisation is required, and the general applicability of the chosen elements seems higher. The relevance with respect to the course description is also not as clear.

Nonetheless, some work on projective geometry is performed by way of significant background research, reviewing the image formation process in part 2, and actually fitting 2D homography transformations.

The aim is to select, investigate/develop and implement *own* algorithms for the selected core functionality, rather than using built-ins from MATLABs Computer Vision Toolbox, OpenCV or other similar packages. The belief is that this will give deeper insights and familiarity with the field, and yield a good foundation to build on going forward.

2.1.2 Delimitations

As mentioned in the Ambitions section, projective geometry is not the main focus in this project, and there is consequently no emphasis on independently developing algorithms for that or documenting related theory.

So derivations of results from multiple view geometry, or the related optimisation problems, will not be given in this report. They are also too extensive, and anyway better explained in the standard texts of the field.

The basics from linear algebra will also not be covered in any significant detail in this report. The project and report relies on linear algebra equivalent to the 5th semester elective course ETALA at Aarhus University. This covers most of David Lay's text [2], including basic linear algebra, coordinate transformations, rotation matrices and generally orthogonal matrices, matrix decompositions and eigen-things, quadratic forms, and study cases in homogeneous coordinates for 3D to 2D image projections and similar.

3. A 3D vision pipeline

This chapter outlines the steps in building an image processing pipeline for 3D vision, with the purposes stated in the problem statement. Fig. 3.1 gives an overview.

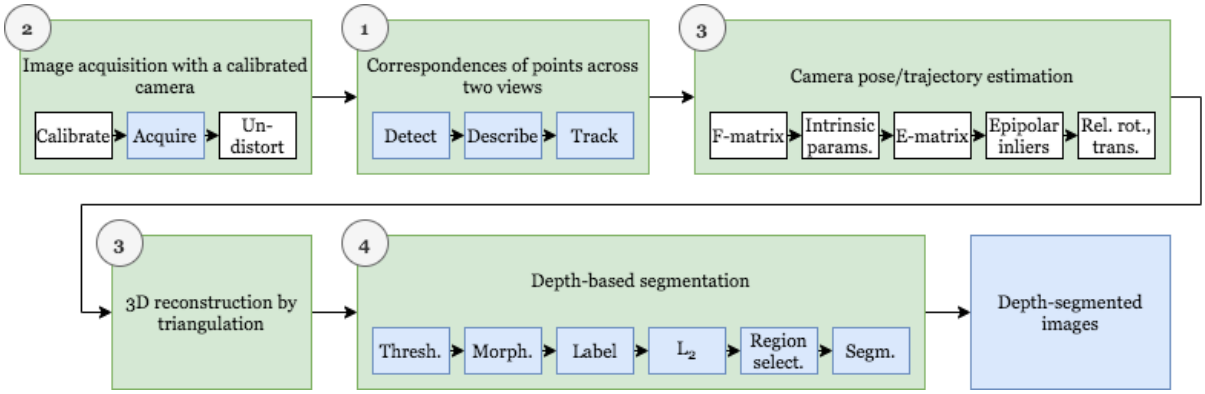

Figure 3.1: There are 5 key steps in the 3D vision pipeline. All 5 steps are included in this project, and the numbers correspond to the parts in the report. The blue boxes mark algorithms that are developed and implemented in this project.

Each challenge in the pipeline is discussed briefly in the following section.

3.1 Key ideas and key problems

The key idea for obtaining 3D information from images works like the depth perception of human stereo vision: Relations between points and lines seen from two different views can via triangulation reveal distances of points and differences in depth. Adding a parallax (like an owl moving its head from side to side) emphasizes this stereo effect. It is the focus of **projective geometry** to model this problem mathematically. The key ingredients are homogeneous coordinates, linear algebra and transformations. This is covered in part 2 of the project.

The **first challenge** in actually implementing these ideas algorithmically is to detect feature points in images, describe them, and to track them across views (= images of the same scene from different viewpoints). These are points in different images that *correspond* to the same point in the world. This is called the **correspondence problem**, and is the focus already in part 1 of the project. As correspondence is so foundational for much of (3D) computer vision, the author has chosen to place the main emphasis on this part of the project.

A **second challenge** is to model the image formation process, and the coordinate transformations in the projection that "brings" a point from the world into the sensor's coordinate system. In particular, it is important to calibrate the specific camera being used: A camera has intrinsic parameters due to its sensor and lens construction, such as the focal length and the geometry of the sensor. To relate images to 3D geometry, these parameters must be estimated. This problem is called **camera calibration**, and is also handled in part 2 of the project.

It turns out that the change in **camera pose** between images, *for a static scene*, can be estimated without knowledge of the absolute locations of points in the world. A number of constraints must hold for images of the same 3D point in two views. With these, it is possible to solve for the relative camera translation and rotation using enough point correspondences. These constraints are called the **eipolar constraints**, which are co-planarity constraints. They are discussed in part 3 of the report.

The relative translation and rotation are considered extrinsic camera parameters. Determining these is sometimes considered part of camera calibration, in particular in the case of a stereo camera rig with fixed relative pose.

A **third challenge** is then to triangulate to determine 3D point locations. Done for all point correspondences, one gets a sparse reconstruction of the 3D scene. The two relative positions of the camera were already estimated, and using this relative pose and the image point locations, 3D coordinates of world points can be fitted. This is however only up to a scaling factor, meaning that one could always imagine photographing an object twice as large from twice as far away, and then still get the same image. If the size of an object in the scene is known, Euclidian reconstruction can be performed to obtain correct scale, but that is not in focus in this project.

The **fourth challenge** is then to actually use the 3D information for a useful application. Tesla uses 3D vision for analysing a car's surroundings, drones andother UAVs use 3D vision for visual SLAM, and in the following section a number of further applications are listed.

In this project, the end-application is to segment an image based on depth information. Segmentation is a useful tool in itself for many analysis tasks. A depth-segmented image can also be used as input in another vision system, e.g. machine learning-based object recognition.

Multiple view geometry is usually the name of the subfield within computer vision, which focuses on solving this series of problems. In some areas of computer vision, problems 1-3 are called structure from motion (SfM).

3.2 Relevant applications of 3D vision techniques

Application of the mentioned concepts in image processing is typically under the umbrella of computer vision. A few relevant applications are mentioned in this section:

- In stereo vision, images can be segmented based on the estimated depth of different regions in a scene. That is also the ultimate aim in this project, but there is an entire field dedicated to emulating the capabilities of human vision, using extremely well-calibrated stereo rigs to solve interesting vision problems.
- In robotics and embedded vision, visual SLAM is the simultaneous estimation of the rigid body motion of the camera (and thus the robot), and partial reconstruction/mapping of 3D geometry of the environment. This allows a robot to navigate using inexpensive cameras.
- In photogrammetry, objects are measured precisely. E.g. by aerial photography or from closer range. Applications could be to obtain distance, size or area estimates for land surveying, for instance with orthophotos. Several photos are "registered" with keypoints (like correspondences) and corrected for projective transformations due to camera pose. Another application is to augment

maps with accurate dense 3D models, like on Google Earth. Quality control is yet another possible application.

- In gaming, visual effects and AR, images and video are augmented with objects transformed by correct projective mappings and placed automatically in a location in the frame that appears realistic.
- In photography, panoramic mosaics are stitched together from multiple overlapping individual images, each corrected for the projective distortion from the pose of the camera. The images are fused at feature points.
- In video, one way of stabilising recordings is to digitally undo camera motion between frames, e.g. by adjusting for optical flow (small changes in camera pose). Foundationally, this is quite similar to the correspondence problem, but also using a spatial derivative.

Cameras are often small, inexpensive and ubiquitous, being available in edge devices such as personal smartphones, cars, drones, etc. So there are wide-ranging opportunities to deploy algorithms like the ones developed in this project.

Methods like Lidar and SAR Radar also give ranging information to reconstruct 3-dimensional structure, but are typically more expensive and complex to deploy than cameras, and so, for now, perhaps have fewer economically feasible applications.

4. Literature review and notation

4.1 Literature and materials

The standard textbook in the field appears to be "Multiple View Geometry in Computer Vision" by Hartley and Zisserman [3]. It is referenced in many other works, and in several implementations in MATLAB and OpenCV. The background introduction to projective geometry and transformations in 2D and 3D is very thorough and useful. The book is available through the library at Aarhus University.

Another common and cited book is "An Invitation to 3-D Vision" by Ma, Soatto, Kosecka, and Sastry [1]. This book is denser and very to-the-point, like most maths books from Springer. The sections on fitting transformations are particularly helpful. The introductory material on group theory, and in particular the Lie group and Lie algebra structure of continuous rigid-body transformations and the use of exponential coordinates is very interesting. The appendices cover many important results in linear algebra. The book is not available through the library at Aarhus University, but is for sale as an e-book from Springer. For a preview, a non-pulished manuscript is available free of charge via the University of Delaware [4], with contents that appear very similar to the published version from Springer. This report will however cite and reference the book with page and formula numbers as in that purchased from Springer.

TU Munich has a 2014 lecture series based on this book. It was given by Prof. Daniel Cremers and was recorded in-class [5]. However, the approach is quite theoretical with few practical examples.

Both books are standard in the field, but they are from 2004, and there have been developments since. For instance in terms of feature descriptors. Particularly, the newer binary descriptors are interesting due to reduced computational complexity and thus suitability for real-time edge devices. Specifically, the BRIEF descriptor by Calonder et al. published in 2012 is interesting. The original article is very readable, and BRIEF is used in this project [6].

An interesting set of lectures is the "Photogrammetric Computer Vision" series, released in April 2020 and recorded specifically for an online audience by Prof. Cyrill Stachniss from the University of Bonn [7]. It was through this that the author became aware of among other the newer BRIEF feature descriptor. The lectures update and condense his older but more comprehensive and beginner-friendly series (45 1-2-hour lectures) recorded in-class in 2015/16 during the courses Photogrammetry I & II [8].

Séan Mullery from Institute of Technology Sligo has a lecture series titled "Multiple View Geometry in Computer Vision" from 2019 [9]. It is course material from the MEng programme in Autonomous Vehicles, and has 20 1-hour classes that give a good and broad overview of the field.

Finally, Richard Szeliski, currently a research scientist at Facebook, has two CV books freely available online for non-commercial use. One is a draft version of his to-be-published "Computer Vision: Algorithms and Applications, 2nd ed." [10]. This is being updated on an on-going basis and looks almost complete. The other book is the published "Computer Vision: Algorithms and Applications, 1st ed." [11].

4.2 Notation

This report tries to keep with a notation that is relatively consistent across the literature.

A general vector, such as a translation, will be written as $\bar{t}$. An image point in the 2D plane, the image plane, has its coordinate vector written as a bold, italicized lowercase symbol like $\boldsymbol{x} = [x, y]^\intercal$. $^\intercal$ denotes the transpose. So coordinate vectors are column vectors.

Matrices are written as A or $A_{m\times n}$ if the dimensions are not clear from context. For invertible matrices, $-^\intercal$ is the transpose of the inverse, as in $A^{-\intercal} = (A^{-1})^\intercal$.

Scalars are written like the indexed coordinates inside vectors, or just λ or Z.

A point in Euclidian three-space $\mathbb{E}^3$, or the corresponding projective space $\mathbb{P}^3$ is written as e.g. $p \in \mathbb{E}^3$. Its coordinates depend on the basis (coordinate frame) and a coordinate vector for Euclidian space is $\mathbf{X} = [X_1, X_2, X_3]^\intercal$.

In projective space (homogeneous coordinates) a 1 is appended to the coordinate vectors, and these are then given as $\mathbf{X} = [X_1, X_2, X_3, 1]^\intercal$. Whether a coordinate vector represents Euclidian or projective space should be clear from the context.

5. Part 1: The correspondence problem

In short, the correspondence problem is to track the locations of the *same* 3D world points across multiple *different* views (= images taken with different camera poses). The typical flow for this is shown in fig. 5.1.

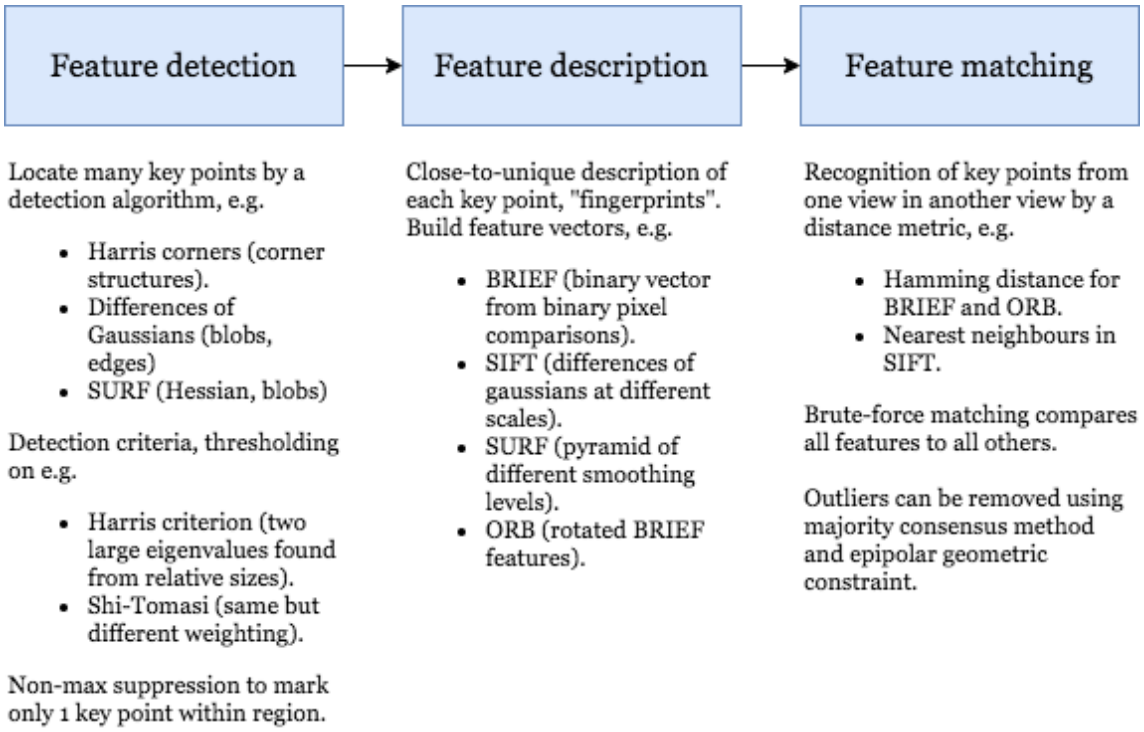

Figure 5.1: The 3 steps in the correspondence problem. Acronyms: BRIEF (Binary Robust Independent Elementary Features). SIFT (Scale-Invariant Feature Transform). SURF (Speeded Up Robust Features). ORB (Oriented Fast BRIEF).

5.1 Detecting feature points

In this section, a feature detector is developed to detect key points, or rather "key patches" which are small neighbourhoods. These are locally distinct features that will remain relatively stable across images and lighting conditions. A typical algorithm for this is the Harris corner detector (also called Harris-Stephens), see e.g. [1, pp. 90-92] or Gonzalez & Woods [12, pp. 869-875].

The rationale for detecting corners is that a corner is localisable in two directions. Corners are roughly orthogonal intersections of two edges. An edge should be understood as a sudden change in intensity (brightness). The idea is to search the image for points with significant intensity changes in two directions.

5.1.1 Image structure and eigenvalues

Derivates and gradients can be used to describe image structure. The idea is best illustrated with different image structures. Repeated from [13, p. 9], fig. 5.2 shows three cases:

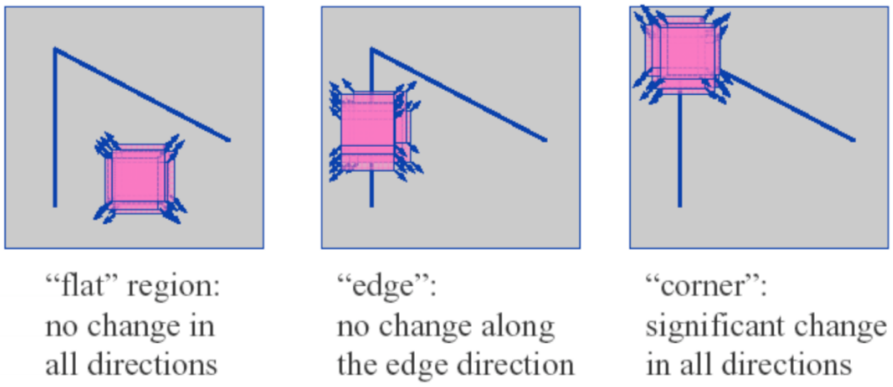

Figure 5.2: Flat regions, edges and corners will have different gradients.

The partial derivatives, I_x and I_y in an image patch will have different distributions depending on the structure in the patch. Fig. 5.3, also repeated from [13, pp. 17-18], illustrates this.

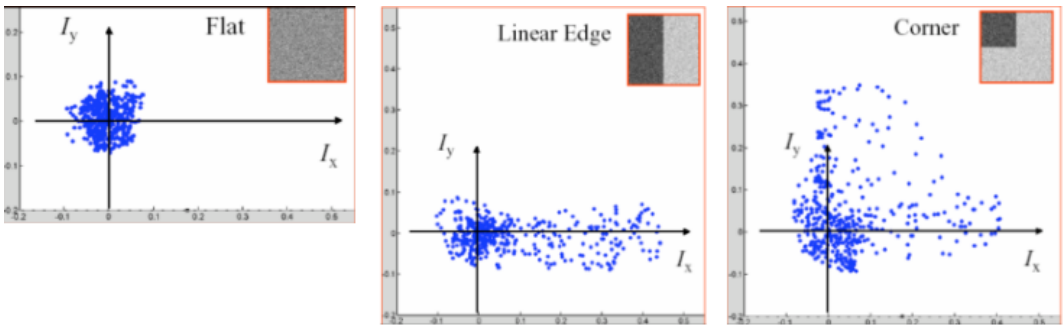

Figure 5.3: Flat regions, edges and corners have different distributions of partial derivatives for a patch in an image. Each dot is an observation of the gradient at a particular pixel-point in the patch.

Note in particular from the figure, that a pixel near a corner will have two large partial derivatives. Also note that a patch containing a corner structure will have many observations of large intensity change along the x and y directions (in this special case of a non-rotated corner). The next section presents a method to condense this gradient distribution in a patch into a single matrix that will summarize the structure in the patch.

5.1.2 The Harris Matrix

For a given point, (x, y), its surrounding neighbourhood $W_{x,y}$ is considered. The intensity (brightness) of a pixel in the neighbourhood is $I(u, v) : (u, v) \in W_{x,y}$. The intensity difference to another pixel under a shift $(\delta u, \delta v)$ is considered as single observation of the local intensity change. The sum of squared differences (SSD) for the entire neighbourhood is used to summarise all observations in the patch, yielding a single value for the point in the centre of the patch, $f(x, y)$:

$$f(x, y) = \sum_{(u,v) \in W_{x,y}} (I(u, v) - I(u + \delta u, v + \delta v))^2 \tag{5.1}$$

A first-order Taylor expansion around (u, v) is $I(u+\delta u, v+\delta v) \approx I(u, v)+\begin{bmatrix} J_x(u, v) & J_y(u, v) \end{bmatrix} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix}$, with $\begin{bmatrix} J_x(u, v) & J_y(u, v) \end{bmatrix}$ being the Jacobian of partial intensity derivatives at (u, v) in the x and y directions

respectively, e.g. $J_x(u,v) = \frac{\partial I(u,v)}{\partial x}$. In this case, the Jacobian has the same elements as the gradient.

Restating as a matrix expression, and simplifying the notation, the SSD value becomes [12, eqs. 11-59 to 11-61]:

$$f(x,y) \approx \begin{bmatrix} \delta u & \delta v \end{bmatrix} \begin{bmatrix} \sum_W J_x^2 & \sum_W J_x J_y \\ \sum_W J_x J_y & \sum_W J_y^2 \end{bmatrix} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \tag{5.2}$$

This shows that the partial derivatives are summed for all points across the neigbourhood instead of summing the quadratic form. This is computationally faster and can be done using a box filter. The 2×2 matrix is called the Harris matrix, or the Structure matrix, and is denoted M.

Given the discussion in the previous section, the Harris matrix is expected to have the following values in the different (nonrotated) cases:

$$\begin{aligned} M = \begin{bmatrix} \sim 0 & \sim 0 \\ \sim 0 & \sim 0 \end{bmatrix} &\rightarrow \text{Flat} \\ M = \begin{bmatrix} >> 1 & \sim 0 \\ \sim 0 & \sim 0 \end{bmatrix} \text{ or } M = \begin{bmatrix} \sim 0 & \sim 0 \\ \sim 0 & >> 1 \end{bmatrix} &\rightarrow \text{Edge} \\ M = \begin{bmatrix} >> 1 & \sim 0 \\ \sim 0 & >> 1 \end{bmatrix} &\rightarrow \text{Corner} \end{aligned} \tag{5.3}$$

If the structure is rotated, there will be some other combination of values, and for that reason it will make sense to look at the eigenvalues of M rather than the matrix itself. This is done later.

As noted in [12, p. 871], if the image is noisy, it will make sense to do Gaussian smoothing of the neighbourhood beforehand. This will reduce the detector's sensitivity to noise.

5.1.3 Computing partial derivatives

The image intensity derivatives can be computed on a grayscale image using 2D convolution. Here a Sobel kernel is chosen to compute derivatives. These are the same as in the Sobel edge detector. There could be many other choices.

$$D_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, D_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{5.4}$$

```
clear all; close all; clc;

chu_orig = im2double((imread('img/chu1.jpg')));
input_image = imresize(chu_orig, 0.4);           % 40% size, faster proces.

I2 = im2double(rgb2gray(input_image));           % Grayscale
I2 = conv2(I2, fspecial('gaussian', 9), 'same'); % Gaussian blur de-noising

% 3x3 sobel operators (kernels)
Dy = fspecial('sobel');
```

```
Dx = Dy';

% Compute partial derivatives by 2D convolution.
% The 'same' setting ensures the window size remains fixed.
Jx = conv2(I2, Dx, 'same');
Jy = conv2(I2, Dy, 'same');
```

Below, three patches with different structure are picked out to illustrate the concept.

```
nhood = 6;                  % 13x13 pixel neighbourhood (6+1+6)

% Pixel (x,y) locations
flat = [819, 705];
edge = [826, 684];
corn = [720, 694];
points = [flat; edge; corn];
names = {'Flat', 'Edge', 'Corner'};

% pixel limits
pmin = @(p) p - nhood;
pmax = @(p) p + nhood;

% Plot comparison
figure;
for m = 1:3
    % Pick out patch coordinates and order into dataset for scatterplot
    x = points(m, 1); y = points(m, 2);

    patch = input_image(pmin(y):pmax(y), pmin(x):pmax(x));

    grad_obs = [reshape(Jx(pmin(y):pmax(y), pmin(x):pmax(x)), [], 1), ...
                reshape(Jy(pmin(y):pmax(y), pmin(x):pmax(x)), [], 1)];

    subplot(2,3,m); imshow(patch);
    title([names(m), ' patch'], 'FontSize', 16);
    subplot(2,3,m+3); scatter(grad_obs(:,1),grad_obs(:,2));
    title([names(m), ' gradient distr.'], 'FontSize', 16); grid on;
    xlabel('$J_x$'); ylabel('$J_y$');
end
sgtitle('Comparison of patch structures and gradient distributions', ...
    'FontSize', 18)
```

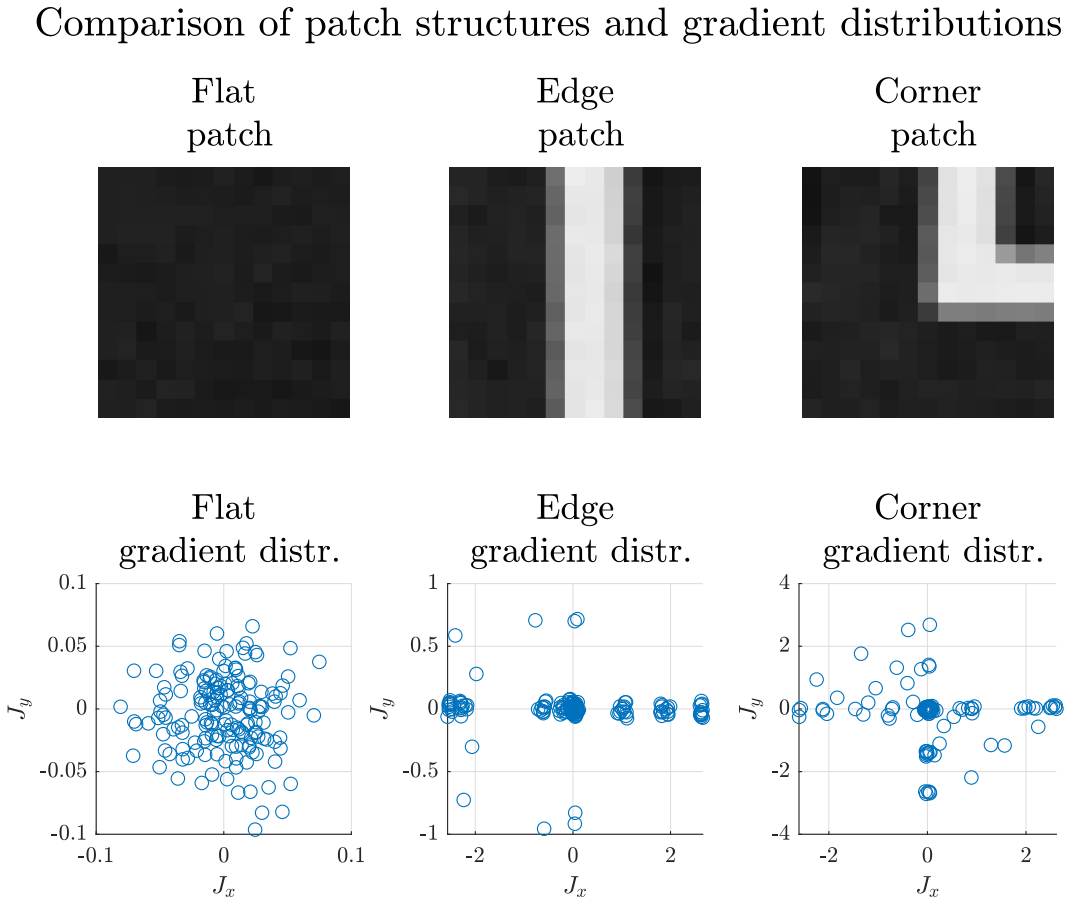

Figure 5.4:

Note from the figure in particular that the corner structure has a distribution of gradient observations with many large values for both J_x and J_y, whereas the edge structure is mostly along the J_x-axis, and the flat/noisy structure appears randomly scattered.

5.1.4 Computing elements for the Harris matrix

The elements for the Harris matrix are computed at each pixel, and the box filter (sum of the neighbourhood) for a 5×5 patch is then computed

$$\begin{aligned} J_x^2 &= (D_x * I)^2 \\ J_y^2 &= (D_y * I)^2 \\ J_x J_y &= J_y J_x = (D_x * I)(D_y * I) \end{aligned} \tag{5.5}$$

$$\sum_{(u,v)\in W_{x,y}} J_x(u,v)^2 = (B_5 * J_x^2), \text{ where } B_5 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{5.6}$$

And similar for the other two elements.

```
% Point values for the Harris Matrix
JxJx = Jx .^ 2;
JyJy = Jy .^ 2;
JxJy = Jx .* Jy;

% Box filter to sum across neighbourhood / patches
B = ones([5 5]);

% Compute sums for the matrix M at each pixel
sum_JxJx = conv2(JxJx, B, 'same');
sum_JxJy = conv2(JxJy, B, 'same');
sum_JyJy = conv2(JyJy, B, 'same');
```

The Harris Matrix can be constructed from these elements at each point. The task is then to actually detect corners using the eigenvalues of M.

5.1.5 The Harris response, corner detection criterion

The eigenvalues of M summarize the power of the partial derivatives at the given point. Considering the stretching of the (squared) gradient distribution as a point-vector transformation, the size of the eigenvalues of the transformation describe the "stretching" of vectors along two orthogonal principal axes[1]. If there are two dominant directions of change, like illustrated for the corner structure above, then there are two large eigenvalues. The advantage of using the eigenvalues and not the gradient itself is the rotational invariance of the eigenvalues (they remain stable even as the corner is rotated).

So, a Harris corner is detected where M has two large eigenvalues. The Harris response at each point is [12, eq. 11-63]:

$$\begin{aligned} R &= \det(M) - k\,\mathrm{trace}(M)^2 \\ &= \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2 \end{aligned} \tag{5.7}$$

The two eigenvalues of M are λ_1, λ_2. The parameter k is a sensitivity factor (more likely to detect corners when k is small), typically set default at $k = 0.04$ [12, p. 875]. A corner is detected where both eigenvalues are large, so

$$\begin{aligned} &R >> 0 \implies \lambda_1 \approx \lambda_2 >> 0 \rightarrow \text{corner} \\ &R < 0 \rightarrow \text{edge} \\ &|R| \approx 0 \rightarrow \text{flat region} \end{aligned} \tag{5.8}$$

Below, the Harris response is computed for the entire image.

```
% At each pixel, compute the response
k = 0.04;                % As proposed by Harris
```

[1] This can be seen seen from the eigenvalue decomposition of a symmetric matrix [2, p. 398]: A nonsingular symmetric matrix $A_{n\times n}$ is orthogonally diagonalisable as $A = PDP^\intercal$ where D has the n eigenvalues of A along the diagonal, and P is an orthogonal matrix. The eigenvalues $\lambda_1, \ldots, \lambda_n$ describe the stretching along the principal axes, and P and its inverse $P^\intercal$ perform the rotation of axes to/from the principal axes. The principal axes are the corresponding eigenvectors, see e.g. Lay [2, p. 404-405].

```
row_max = size(I2, 1);
col_max = size(I2, 2);

% Ignore border area
border_ignore = 20;

% Compute Harris response for each pixel
H = zeros(size(I2));
for col = border_ignore:col_max-border_ignore
    for row = border_ignore:row_max-border_ignore
        M = [sum_JxJx(row,col) sum_JxJy(row,col);
             sum_JxJy(row,col) sum_JyJy(row,col)];
        R = det(M) - k * trace(M)^2;
        H(row,col) = R;
    end
end
```

The idea is now to pick out corners using a threshold on the Harris response.

5.1.6 Why non-maximum suppression of Harris response is needed

There can be several values above a given threshold near the same corner. Ideally, the algorithm only picks out one corner per actual corner. The figure generated below illustrates this issue, and the reason why non-maximum suppression must be implemented.

The Harris responses are viewed as a surface over an image region. A custom function has been implemented to perform this analysis. The two highest responses in the selected region are highlighted, but depending on the choice of global threshold, there could be several positive responses.

```
% pixel ranges for a 25x13 image
corner = [720, 694];
nhood = [6, 12];

% View surface of Harris responses
harris_surface(H, input_image, corner, nhood);
```

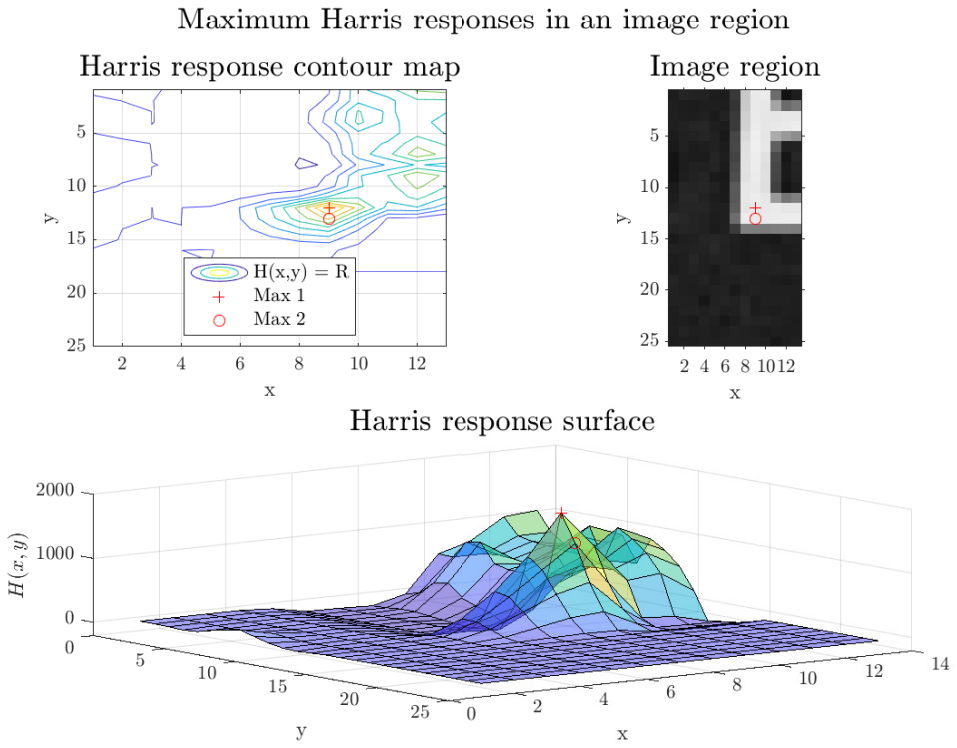

Figure 5.5:

The figure shows two strong responses near the same corner. If these are in the same region of interest (however that will be defined), only one of these maxima should trigger a corner detection. So non-maximum suppression must be implemented.

5.1.7 Non-maximum suppression

This section implements non-maximum suppression in neighbourhood using a maximum filter and subsequent logic suppression. The goal is to find the local maximum in a neighbourhood, and then suppress all other values that are not the maximum. The implemented method uses ordered filtering. The approach is to build a map of maximum values for neighbourhoods and use this map for subsequent logic suppression.

- Find the maximum Harris response in a neighbourhood of size $d \times d$:
- Use a structuring element to define which neighbours are included.
- Run the filter over the Harris response matrix built for the entire image, H.
- The `ordfilt2` filter sorts all included neighbours, from lowest to highest.
- `ordfilt2` builds a H_{max} matrix where all neigbourhood values are replaced by the d^2th value in the sorted list, i.e. the maximum value of the neighbourhood.
- Perform logic suppression by constructing a H_{localmax} matrix of the Harris response matrix, where only points that are the neighbourhood maximum are retained. Others are zero'ed.

```
1 % Ordfilt2 finds the max value in the neighbourhood
2 % This max value is used for logic filter
```

```
3
4  % neighbourhood is d x d (square structure elem.)
5  d = 9;
6
7  % Max. filt.
8  % Replace with max value in nbourhood (d^2 th of d^2 sorted values)
9  Hmax = ordfilt2(H, d*d, ones([d d]));
10
11 % Suppress non-maxima (only retain the local maximum)
12 H_localmax = H .* (H == Hmax);
13
14 % View the non-max suppressed Harris surface
15 harris_surface(H_localmax, input_image, corner, nhood);
```

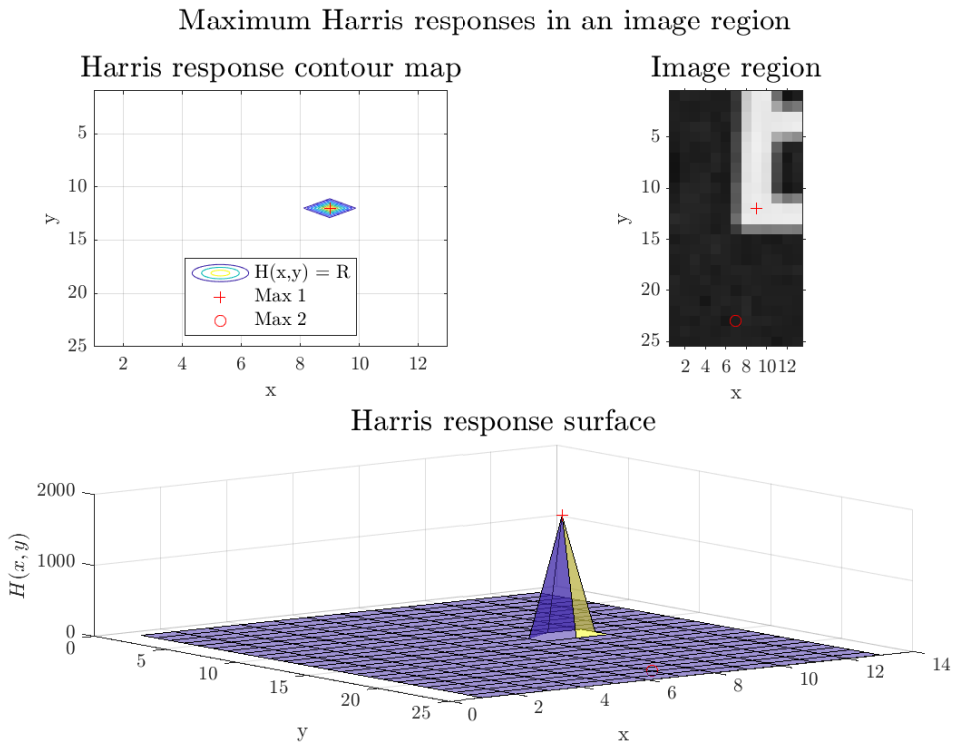

Figure 5.6:

As can be seen from the figure, all non-maxima have been suppressed. So any non-zero threshold will be sure to pick out true local maxima without neighbouring points. A threshold can then be set to control the strength of the selected corners.

5.1.8 Thresholding and corner detection

One option is to set an absolute threshold based on the knowledge of the image or based on experimentation. Another approach, chosen here, is to choose how many Harris corners are desired:

- Set threshold so only N most significant corners are included:

- All localmax corners are sorted.
- The threshold is set based on the Nth value
- Harris corners are detected where the localmax is greater than the threshold.

One could consider doing local sorting in subsets of the image, in order to get better spatial distribution. This would amount to a kind of adaptive thresholding.

After thresholding, the detected corners are shown on the original image.

```
num_corners = 250;
sorted_resp = sort(reshape(H_localmax, [], 1), 'descend');  % global sort
threshold = sorted_resp(num_corners);     % Nth highest value is threshold

% Corners are the local maxima with values higher than threshold
corners = (H_localmax >= threshold);

% Get locations where we've found a corner (binary image is True)
[y, x] = find(corners);
X = [x, y];     % reverse order to get pairs as (xi, yi)

% See the detected corners
figure;
imshow(input_image); hold on;
plot(X(:,1), X(:,2), 'r+');
legend({'Corner locations'}, 'FontSize', 16);
hold off;
title(['Harris corner detection (N = ', num2str(num_corners), ')'], ...
    'FontSize', 20);
```

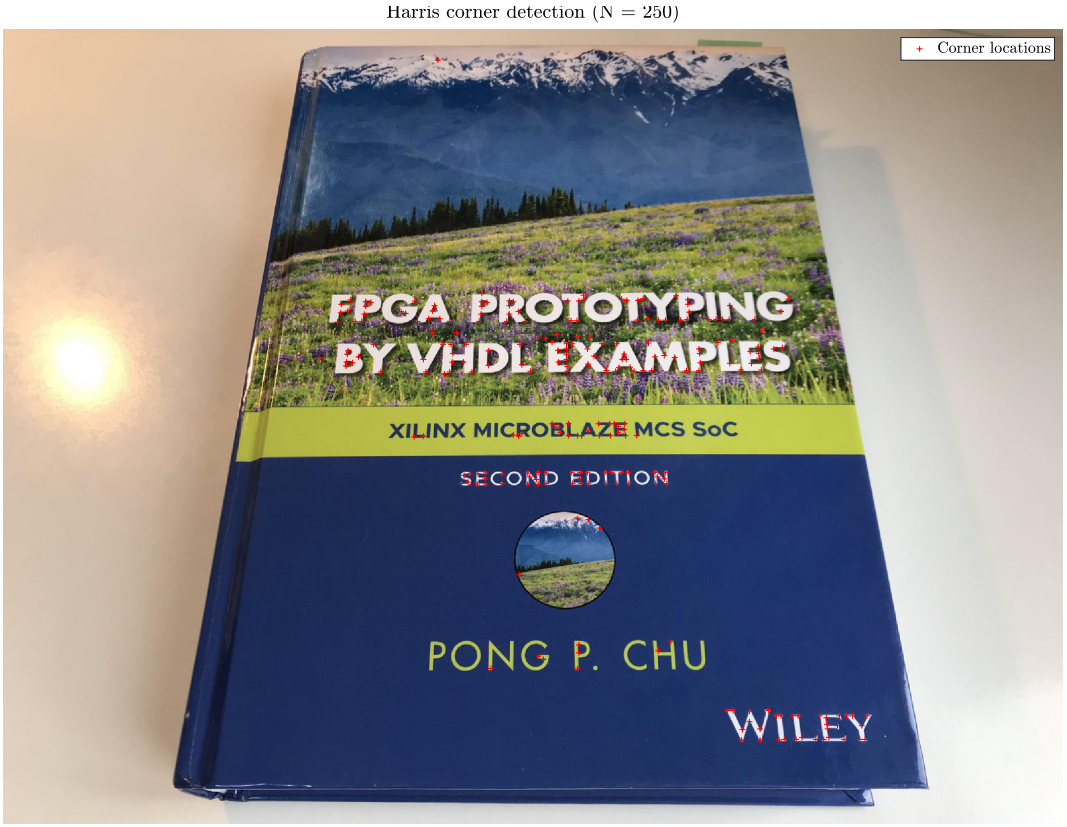

Figure 5.7:

As the figure shows, the detector has found mostly what we would consider corner structures, apart from some corner-like features in the "grass". It is not important that the features are actually corners, only that they are stable across multiple views.

The white/green, white/blue intensity change results in larger grayscale gradients than green/blue, so the detected corners are mostly on white text. One possible improvement to the algorithm would be forcing better spatial distribution of corners.

The developed algorithm has been implemented in a function for use later in the project: `harris_corners(input_image, num_corners, k, border_ignore)`.

5.2 Describing feature points

In this section, an algorithm for feature descriptors is implemented. It has been chosen to implement BRIEF (Binary Robust Independent Elementary Features). A BRIEF feature is described by a feature vector that is simply a bitstring of optional length. Comparison of two descriptors is by the Hamming distance. These descriptors are very interesting, because they are *relatively* simple to compute (much simpler than SURF and SIFT) and *very* simple to compare. This makes them suitable for real-time systems. The sources for this chapter is primarily the original article by Calonder et al. [6] with background info from [7, lec.: visual features part 2].

5.2.1 Test images

Two images have been takes of the same scene from two different views to develop and test the BRIEF descriptor. They are shown below with harris corners overlay.

```
% load images and resize to speed up processing
imleft = im2double(imread('img/IMG_L3.jpg'));
imright = im2double(imread('img/IMG_R3.jpg'));
im1 = imresize(imleft, 0.5);
im2 = imresize(imright, 0.5);

target_points = 200;    % Force finding this many points in each im
border_skip = 20;       % Pixels in border are ignored
k = 0.04;               % Weighting parameter, 0.04 default by Harris

X1 = harris_corners(im1, target_points, k, border_skip);
X2 = harris_corners(im2, target_points, k, border_skip);

% Display test images
show_test_images_with_features(im1, im2, X1, X2, target_points);
```

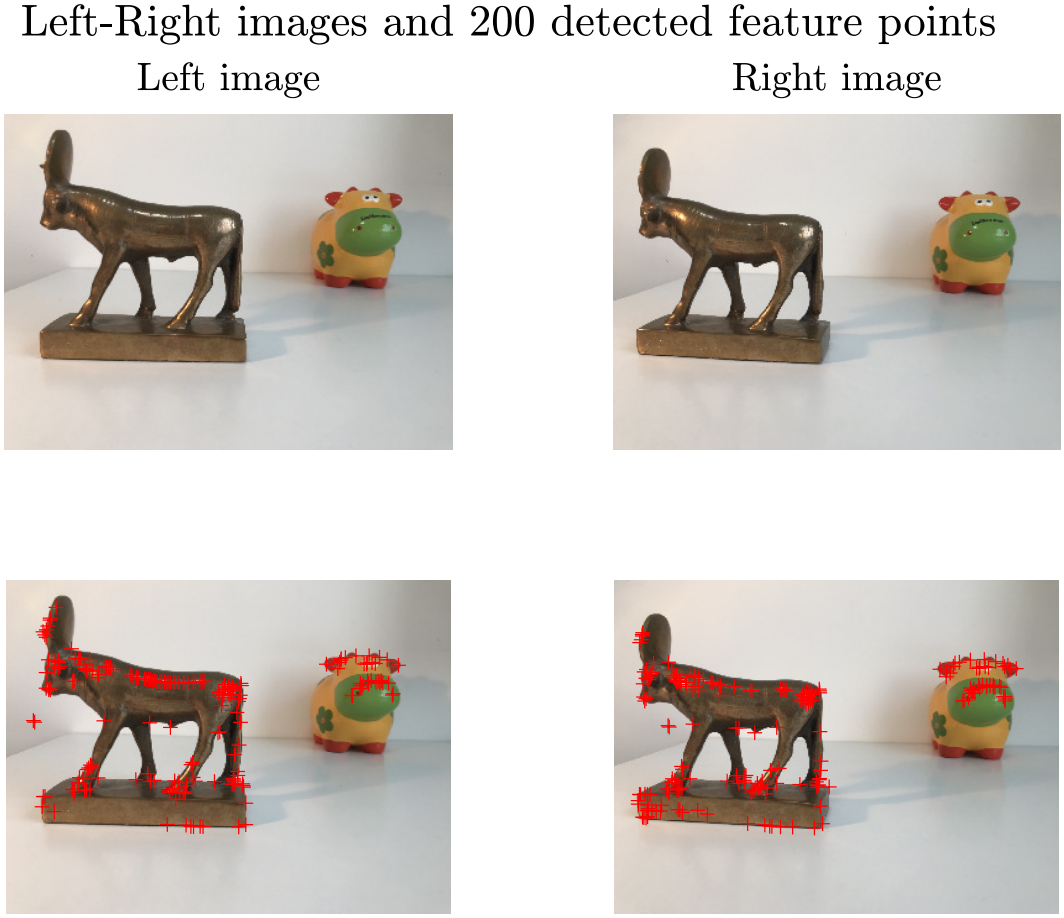

Figure 5.8:

The goal is now to implement feature descriptors for each detected Harris corner and later to track the features across images.

5.2.2 Descriptor and design for describing a single feature point

The BRIEF features are built from pair-wise binary intensity comparisons in a neighbourhood around a key point. The binary test τ on an image patch $\boldsymbol{p}$ of size $S \times S$ between point $\boldsymbol{x}$ and $\boldsymbol{y}$ is [6, eq. 1]:

$$\tau(\boldsymbol{p}, \boldsymbol{x}, \boldsymbol{y}) := \begin{cases} 1 & \text{if } I(\boldsymbol{p}, \boldsymbol{x}) < I(\boldsymbol{p}, \boldsymbol{y}) \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

In an embedded system, each descriptor would be stored as a 32-bit integer, whose value would be encoded as [6, eq. 2]:

$$\sum_{1 \le i \le n_d} 2^{i-1} \tau(\boldsymbol{p}, \boldsymbol{x}, \boldsymbol{y}) \tag{5.10}$$

In MATLAB, the feature descriptors can be stored as binary/logic vectors and matrices.

There are four key design decisions for the BRIEF descriptor:

- How many pairwise comparisons to make?
- How to perform smoothing before comparisons?
- Which spatial arrangement / geometry to use when sampling comparison points?

- How large a patch to sample for the comparisons?

The first is partly a matter of speed and storage space. Given $n_d = 128$ comparisons, $\frac{128}{8} = 16$ bytes are required to store the descriptor. This is called BRIEF-16, and so on.

The original authors have analysed different choices for the other design decisions, and it appears that:

- Box smoothing with a 7×7 kernel performs about as well as Gaussian smoothing.
- A sampling geometry based on an isotropic bivariate Gaussian distribution (method `G II`) outperforms the other methods in terms of recognition rate.
- The patch size is not discussed extensively, but enters in the variance of the bivariate Gaussian distribution. From figures in the paper, it appears that their patch size is about 30×30 pixels [6, p. 1284].

In the following, a BRIEF-32 is implemented using sampling method `G II` with a patch size of 33×33 and Gaussian smoothing with a 7×7 kernel.

```
nd = 256;        % BRIEF-32
S = 33;          % S x S patches
```

5.2.3 Implementation of sampling geometry

As in [6, p. 1282], the `G II` sampling geometry is

$$(\mathbf{X}, \mathbf{Y}) \sim \text{i.i.d. Gaussian}(0, \frac{1}{25}S^2) \tag{5.11}$$

So each sampling pair $\boldsymbol{x}, \boldsymbol{y}$ is drawn from the bivariate normal distribution, where the distribution of each point is independent and identically distributed (i.i.d) to the other. The coordinate locations are centered on the key point (zero mean).

A function `brief_geom(nd, S)` has been implemented to create the geometry, which can be seen in the appendix. The return value is an $n_d \times 4$ matrix, with each column holding the 256 sampled coordinate locations in the order (u, v, q, r) for $\boldsymbol{x} = (u, v)$ and $\boldsymbol{y} = (q, r)$.

```
rng(1)                          % seed to control randomness
geom = brief_geom(nd, S);       % compute sampling geometry
brief_display_geom(geom);       % show the sampling geometry
```

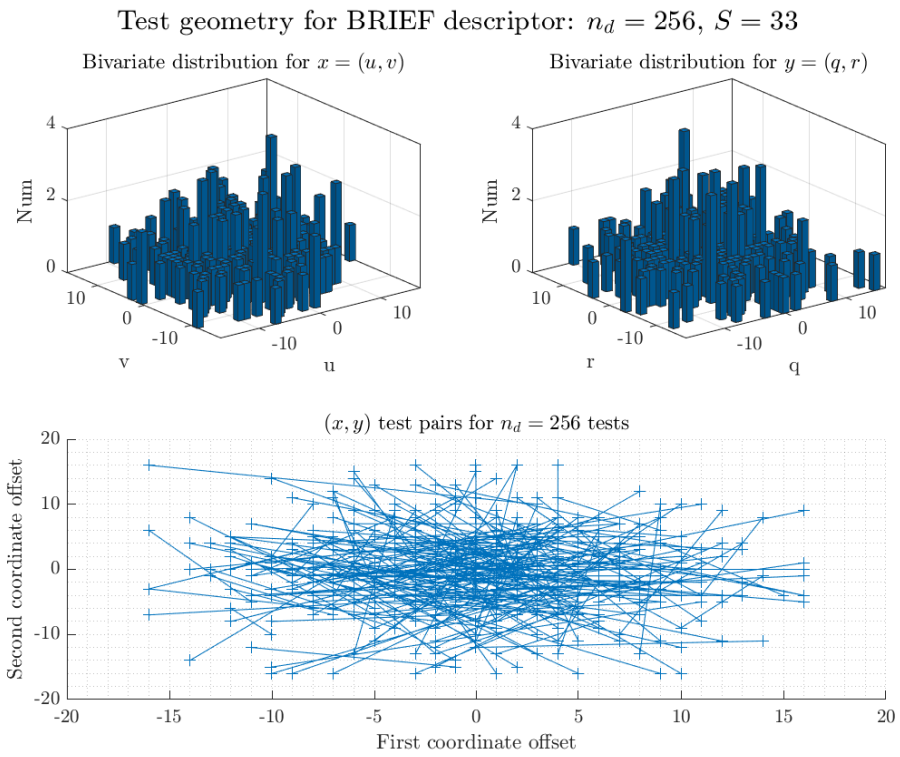

Figure 5.9:

The upper two plots in the figure show that the coordinates of each point in the sampling pair approximately follow a Gaussian distribution. The lower plot shows the pairwise comparisons, which also display a clear tendency to be centered around the feature point, *"motivated by the fact that pixels at the patch center tend to be more stable under perspective distortion than those near the edges."* [6, p. 1283].

5.2.4 Computing the feature descriptors

A function `brief_fd(image, feature_points, S, geom)` has been implemented to generate a matrix of feature descriptors for an image. The implementation can be seen in the appendix.

The return value is a logic/binary matrix of size $(N_{\text{featurepoints}} \times n_d)$, where each row represents a feature point (Harris corner) and the n_d columns are the bits in the binary vector (bitstring). Each bit is the result of a binary comparison by eq. 5.9.

For the test images with 200 Harris corners, each feature matrix is 200×256. The feature matrices for the two test images are computed below based on the previously created test geometry.

```
1 im1_fds = brief_fd(im1, X1, S, geom);
2 im2_fds = brief_fd(im2, X2, S, geom);
```

5.3 Tracking feature points

In this section, feature points are matched using the developed BRIEF feature descriptor. The purpose is to find the correspondences across views.

This is the final step in solving the correspondence problem, and will output a set of "putative" matches. The matches are "putative" in that they are *inferred* by way of algorithms, and thus *assumed* to be true, but however still without proof from e.g. epipolar geometric matching.

5.3.1 Method

The method used in this section is brute-force across two views: Each feature in the first view is compared to every feature in the second view. The comparison is done using the Hamming distance, which counts how many bits are different between two bitstrings. It is a very simple distance metric, which can be efficiently implemented by way of `XOR` and bit-count operations [6, p. 1281].

An additional step, borrowed from David Lowe, the inventor of SIFT, is Lowe's Ratio Test, a relative method to determine if a match is "good enough". It is described in [7, lec.: visual features part 2, 19:50] and has three steps:

- For descriptor q in view 1, find closest two descriptors p_1 and p_2 from view 2.
- Test if distance to best match is smaller than threshold: $d(q, p_1) < T$
- Accept match if and only if the best match is substantially better than the second best match: $\frac{d(q,p_1)}{d(q,p_2)} < \frac{1}{2}$

An optional statistical test to eliminate outlier matches has been homebrewed during the project:

- It relies on the distribution of the L_2 norm of distances between the matches.
- If any matches are spatially further away than 2.5 standard deviations from the mean of all the matches, then such a match is improbable and is tagged as an outlier.

5.3.2 Implementation

A function `brief_matches(X1, X2, im1_fds, im2_fds, T, stat_outliers)` has been implemented to generate two $(m \times 2)$ sets of coordinates of matched feature points in image 1 and image 2 respectively. The implementation can be seen in the appendix.

Another function, utilising the `showMatchedFeatures` plotting tool from the Computer Vision Toolbox displays a montage of the correspondence matches.

```
T = 50;     % compare to max Hamming distance n_d=256 if all bits flipped
stat_outliers = true;   % do statistical removal of outliers

% Match feature points
[M1, M2, match_info] = brief_matches(X1, X2, im1_fds, im2_fds, ...
                                     T, stat_outliers);
```

```
7
8
9  % Display the matches
10 display_type = 'false';
11 correspondence_show_matches(im1, im2, M1, M2, match_info, display_type);
```

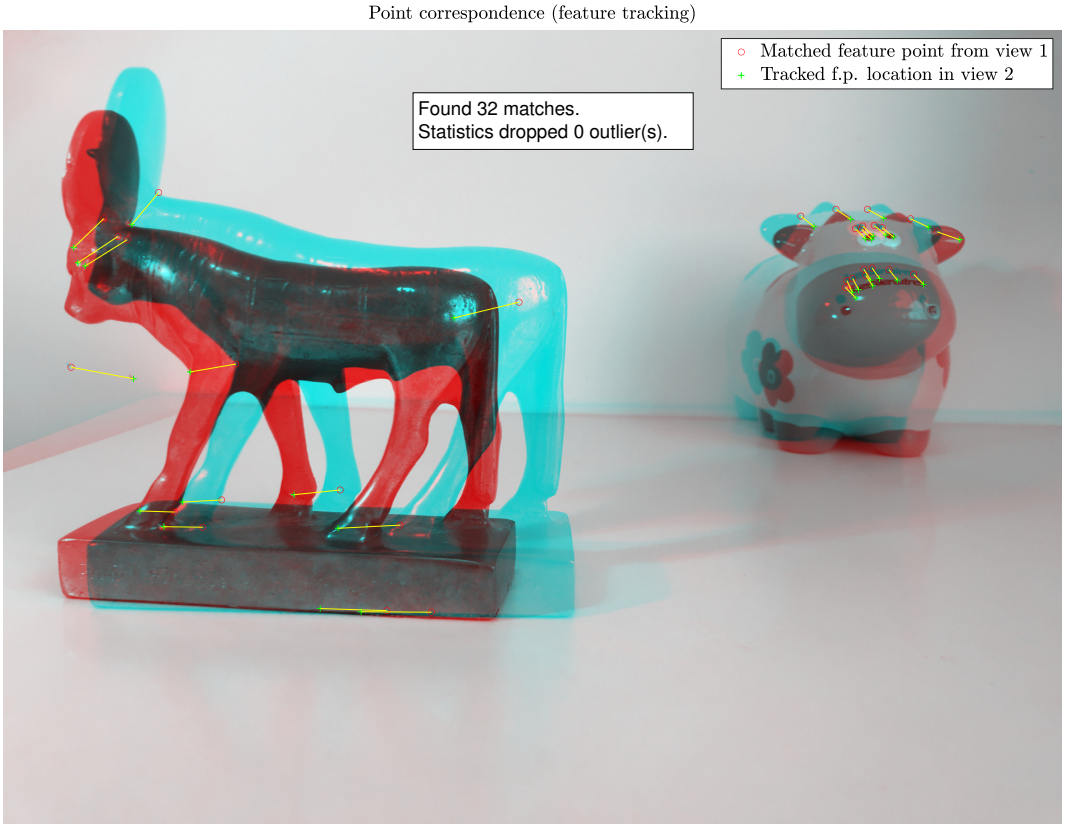

Figure 5.10:

The figure shows that 32 matches were made out of the 200 feature points that were initially detected. No outliers were removed from the matched set. The quality of the points in the scene is not high, as there is not a lot of clear structure on neither the "Apis" nor the "cow". Yet, 32 points would be enough to do camera pose estimation and some sparse 3D reconstruction.

Note also the lines of movement:

- View 2 is taken by moving the camera to the left in relation to the scene, closer to the "cow", and adding a slight rotation towards it.
- The lines of movement are long on the "Apis" and short on the "cow" due to the camera's trajectory.
- All lines of movement on the "Apis" are in the same direction. The same is true for the lines on the "cow".
- This makes sense as the objects are not deforming, and the camera remains relatively far away from the objects.

5.3.3 Evaluation of the algorithms and parameters

The matches seem good, and the algorithm appears to work well without ambiguous matches. Obviously, if there was a lot of repeated structure / periodic patterns in the scene, the algorithm would likely have made many ambiguous matches.

Notes on parameter settings:

- Requesting a higher number of corners from the algorithm generally gives more matches. So it is *not* the case, that corners that are "strongest" in one view are the only ones that can be recognized in another view.
- Generally, a moderate patch size such as 33x33 or 41x41 performs much better than a smaller patch (say, 13x13):
  - There are far fewer bad matches (ambiguous, spurious/outliers).
  - Requires less computation than a big patch (a de-noising convolution is performed on each patch).
  - Allows more independent binary comparisons to be made.
- An n_d around 256 seems to be a sweet spot:
  - It is a trade off between robustness and (perhaps) overconstraining the feature descriptor.
  - A small number does not sufficiently discrimate between points, a large number (perhaps) overconstrains the feature space and has higher computational requirement.
  - Two-way comparisons are allowed, and this might be inefficient, and disproportionaly reduce performance for small values of n_d (i.e., checking both p(x) < p(y) and p(y) < p(x), as these are clearly not independent and could add either 0 or 2 to the Hamming distance).
- There is a degree of randomness in the performance, as the test geometry is based on bivariate gaussian samples, and a different seed will give different performance.

6. Part 2: Image formation, projective geometry and camera calibration

Exhaustive derivations of projective geometry and image formation are given in [1, ch. 3] and [3, ch. 2-3]. The first is mostly focused on the topic as it relates to the special case of image formation in the pinhole camera model, whereas the second is a very in-depth treatment of projective and algebraic geometry in 2D and 3D. In the following, only bare essentials are repeated.

6.1 Pinhole camera model

The pinhole camera model[1] assumes that all rays of light pass through the optical centre of the camera, o, as shown in fig. 6.1.

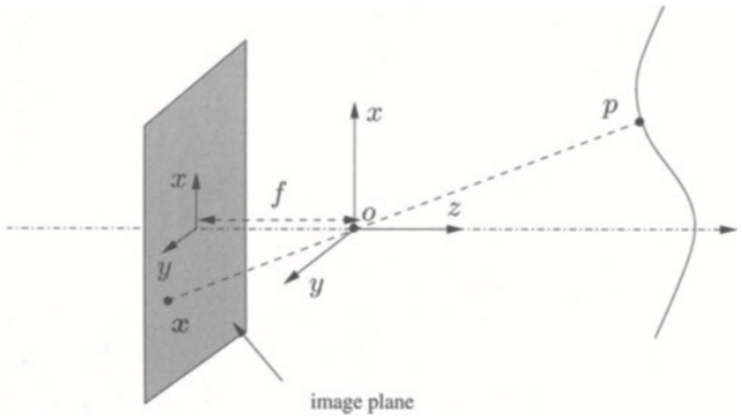

Figure 6.1: The pinhole camera model [1, p. 50].

The world point p in 3D space is projected onto a 2D image plane (sensor or film), placed behind o, at a distance f along the optical axis z. The focal length f is an intrinsic camera parameter. An inverted image of p appears at $\boldsymbol{x}$ on the plane. It is normal to adjust the model and place the image plane in front of the optical centre, so the image is not inverted. This model is in fig. 6.2.

[1] This is the simplest camera model, and approximates a well-focused imaging system [1, p. 50]. It is obviously not physically fully realistic, as the energy through the pinhole decreases to zero as the opening decreases toward a single point, and because many effects are not accounted for.

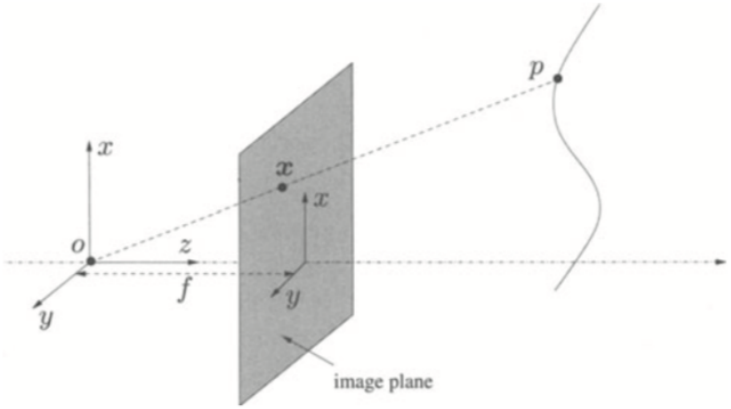

Figure 6.2: The frontal pinhole camera model [1, p. 51].

Based on the image $\boldsymbol{x}$, it is not possible to know where in space p is, only that it is somewhere along the ray (line) through o and p. So depth information is lost.

The projection of p is found by similar right triangles. Let p have the coordinates $\mathbf{X} = [X, Y, Z]^\intercal$ and $\boldsymbol{x} = [x, y]^\intercal$. One triangle is formed between points o, $\boldsymbol{x}$ and a line from the optical axis up to $\boldsymbol{x}$, perpendicular to the axis and at a distance f from o. The other is similarly between o, p and the optical axis at a distance Z. The x and y coordinates of the image are then given by the ratio equalities $\frac{x}{f} = \frac{X}{Z}$ and $\frac{y}{f} = \frac{Y}{Z}$.

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \end{bmatrix} = \frac{f}{Z} \begin{bmatrix} X \\ Y \end{bmatrix} \tag{6.1}$$

This is the **ideal perspective projection** [1, pp. 49-52].

6.2 Homogeneous coordinates

Homogeneous coordinates (h.c.) are motivated by the fact that any point along the ray through o and p will give the same image $\boldsymbol{x}$. So h.c. are an equivalence relation that $\mathbf{X} \sim \lambda\mathbf{X}$, saying that a scaling by $\lambda \in \mathbb{R}_+$ still represents the same point $\mathbf{X}$.

This is encoded by appending 1 to the coordinate vector, so a point in the world in h.c. is $\mathbf{X} = [X, Y, Z, 1]^\intercal \in \mathbb{R}^4$. With h.c. a point in the image plane is $\boldsymbol{x} \in \mathbb{R}^3$.

To convert back to to inhomogeneous coordinates, normalise by the last coordinate as $[\lambda X, \lambda Y, \lambda Z, \lambda]^\intercal \mapsto [\frac{X}{\lambda}, \frac{Y}{\lambda}, \frac{Z}{\lambda}]^\intercal$.

With h.c., the ideal perspective projection is then rewritten from 6.1 [1, p. 52]:

$$Z \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{6.2}$$

With h.c. in 3D, coordinates are vectors in $\mathbb{R}^4 - [0, 0, 0, 0]^T$, which means that the vector $[0, 0, 0, 0]^\intercal$ is not a valid h.c. vector. This space is also called projective space, $\mathbb{P}^3$.

6.3 Transformations

The different groups of linear transformations of $\mathbb{P}^3$ with h.c. vectors are important[2] The transformations in table 6.1 below are explained in detail in [1, ch. 2] and summarised in [3, pp. 78]. They are included here for reference. The matrix $I_{3\times3}$ is the identity matrix, $A_{3\times3}$ is an invertible matrix, $R_{3\times3}$

Transformations in homogeneous coordinates			
Group	Matrix	Effect	Invariants
Translation	$\begin{bmatrix} I & \bar{t} \\ \bar{0}^\intercal & 1 \end{bmatrix}$	Moves points	Everything but location
Rotation	$\begin{bmatrix} R & \bar{0} \\ \bar{0}^\intercal & 1 \end{bmatrix}$	Rotates vectors about an axis	Location, orientation (no mirroring)
Rigid-body motion	$\begin{bmatrix} R & \bar{t} \\ \bar{0}^\intercal & 1 \end{bmatrix}$	Changes pose of rigid bodies	Distances, angles, volumes
Similarity	$\begin{bmatrix} sR & \bar{t} \\ \bar{0}^\intercal & 1 \end{bmatrix}$	Includes scaling on top of rigid-body motion	Angles
Affine	$\begin{bmatrix} A & \bar{t} \\ \bar{0}^\intercal & 1 \end{bmatrix}$	Includes skews and shears	Parallelism of planes and lines
Projective	$\begin{bmatrix} A & \bar{t} \\ \bar{v}^\intercal & 1 \end{bmatrix}$	Includes linear perspective, vanishing points and horizon (infinity)	Straightness of planes and lines

Table 6.1: Overview of linear transformation groups.

is a rotation matrix so $R^\intercal R = RR^\intercal = I$ and $\det(R) = +1$, $\bar{t}_{3\times1}$ is a translation vector from $\mathbb{R}^3$, $\bar{v}_{3\times1}$ is scaling vector from $\mathbb{R}^3$, and $\bar{0}_{3\times1} = [0,0,0]^\intercal$ is the zero vector.

The groups of transformations stated in table 6.1 are also illustrated in fig. 6.3. In particular, note that the Euclidian (rigid-body) contains both rotation and translation. Also note that projective transformation has the same effect as that seen from the train tracks.

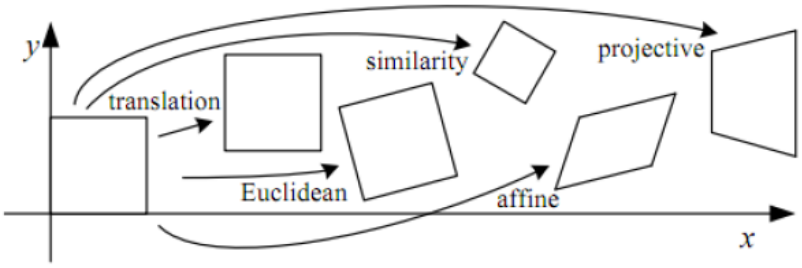

Figure 6.3: The effects of different types of transformations relevant for image formation are represented in a stylized way.

[2] Transformations that can be represented by a single matrix are linear transformations. The affine transformation of $\bar{x} \in \mathbb{R}^3$ combining a linear map $A \in \mathbb{R}^{3\times3}$ with a translation $\bar{t} \in \mathbb{R}^3$ is given by $A\bar{x} + \bar{t}$ and can not be represented by a single matrix in $\mathbb{R}^{3\times3}$. In homogeneous coordinates, the transformation can be represented with a single matrix from $\mathbb{R}^{4\times4}$: $\begin{bmatrix} A & \bar{t} \\ \bar{0}^T & 1 \end{bmatrix}_{4\times4} \mathbf{X}_{4\times1}$.

6.4 Projection from world coordinates to sensor coordinates

The transformation from a 3D point in world coordinates to a 2D coordinate in the sensor's coordinate system is via a sequence of different reference frames. This sequence is shown in fig. 6.4.

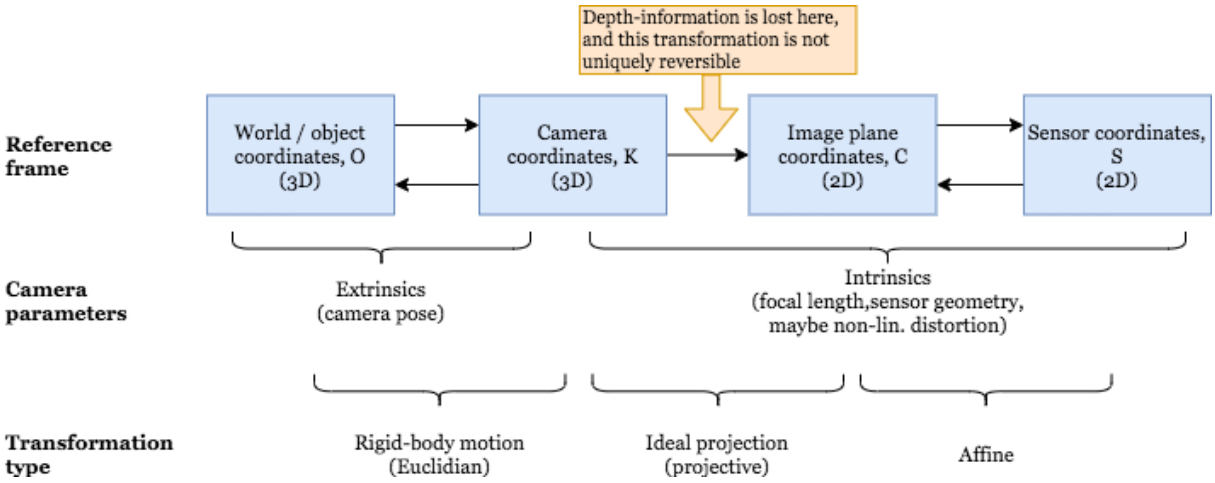

Figure 6.4: The projection of a point in the 3D world to a point on the sensor and in the digital image is composed of a sequence of transformations.

The sequence of transformations can be summarised in one projection equation in homogeneous coordinates

$$\boldsymbol{x} = P\mathbf{X} \tag{6.3}$$

This says that the 3D world point $\mathbf{X}_{4\times1}$ is projected onto the sensor/pixel coordinate $\boldsymbol{x}_{3\times1}$ via the projection matrix $P_{3\times4}$. The transformation is better understood as the composition as in fig. 6.4.

$$\boldsymbol{x} = {}^{S}H_{C}\,{}^{C}P_{K}\,{}^{K}H_{O}\mathbf{X} \tag{6.4}$$

Where e.g. ${}^{K}H_{O}$ means a transformation *from* coordinate frame O *to* coordinate frame K. Each of these transformations is briefly summarised in the following sections.

6.4.1 World frame to camera frame, 3D $\rightarrow$ 3D

This is a Euclidian transformation into the camera's 3D coordinate frame. If the vector from the origin of the world system to the camera system is $\mathbf{X}_O$ in Euclidian coordinates, and the rotation is reprensented by the matrix R, then the transform is

$${}^{K}H_{O} = \begin{bmatrix} R & -R\mathbf{X}_O \\ \bar{0}^{\intercal} & 1 \end{bmatrix} \tag{6.5}$$

There are 6 degrees of freedom in this transformation, 3 from the rotation and 3 from the translation.

6.4.2 Camera frame to image plane, 3D → 2D

For an ideal camera, this is a projective transformation using the ideal perspective projection (pinhole camera model). This projection was already discussed in eq. 6.2.

$$ {}^{C}P_K = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{6.6} $$

6.4.3 Image plane to sensor coordinates, 2D → 2D

The principal point in the image sensor is typically the top-left pixel. So points must be shifted so that pixel gets coordinate $[0, 0]^\intercal$. Further, there can be a scale difference m in x and y, and a shear factor s due to shear of the axes in the sensor. The transformation due to these factors is

$$ {}^{S}H_C = \begin{bmatrix} 1 & s & c_x \\ 0 & 1+m & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{6.7} $$

6.4.4 Calibration matrix

The first three columns of the ideal projection contain the camera focal length. Combine with ${}^{S}H_C$ to summarise intrinsic camera parameters:

$$ K = \begin{bmatrix} 1 & s & c_x \\ 0 & 1+m & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} f & fs & c_x \\ 0 & f(1+m) & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} f_x & \hat{s} & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{6.8} $$

This matrix K represents an affine transformation with 5 parameters, which must be estimated during camera calibration. According to MATLAB, the $\hat{s}$ parameter is normally assumed zero for modern cameras.

6.5 Moving the camera, a 2D → 2D planar homography

An example that will be useful later for understanding camera calibration is a 2D → 2D, or 3×3, planar homography[3]. This is a projective transformation that changes the perspective by "moving the camera" (simulating the effect on the projection of having moved the camera). It is also generally relevant for perspective correction, as shown here.

An acquired image shows linear perspective distortion. A point in the original image plane is in homogeneous coordinates as $\boldsymbol{x} = [x, y, 1]^\intercal$. The target image is a virtual or "desired" image without the distortion, with homogeneous coordinates $\boldsymbol{x'} = [x', y', z']^\intercal$. The homography is a projective transformation $H_{3\times3} : \boldsymbol{x} \mapsto \boldsymbol{x'}$ so [3, p. 33]

$$ \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.9} $$

[3] A homography is a relation between two planes in space. Two images of the same planar surface will be related by a homography.

The homography H must be solved, which requires 4 points[4]. This is the same as performing a rotation and translation of the camera to change the pose of the camera, and then reacquiring the image from the new pose.

Transforming the LHS in eq. 6.9 to inhomogeneous image coordinates eliminates the scale factor, so [3, p. 35]

$$[x', y', z']^\intercal \mapsto [u, v]^\intercal = \left[\frac{x'}{z'}, \frac{y'}{z'}\right]^\intercal \tag{6.10}$$

$$u = \frac{x'}{z'} = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}} \tag{6.11}$$

$$v = \frac{y'}{z'} = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \tag{6.12}$$

Multiplying both sides by the denominators, distributing u and v, and setting the scale factor $h_{33} = 1$, yields two equations per image point:

$$u = h_{11}x + h_{12}y + h_{13} - h_{31}xu - h_{32}yu \tag{6.13}$$

$$v = h_{21}x + h_{22}y + h_{23} - h_{31}xv - h_{32}yv \tag{6.14}$$

The image to be changed is again of the book from our intro FPGA course, shown in fig. 6.5. The figure clearly shows the distortion from projection, as the vertical sides are not parallel.

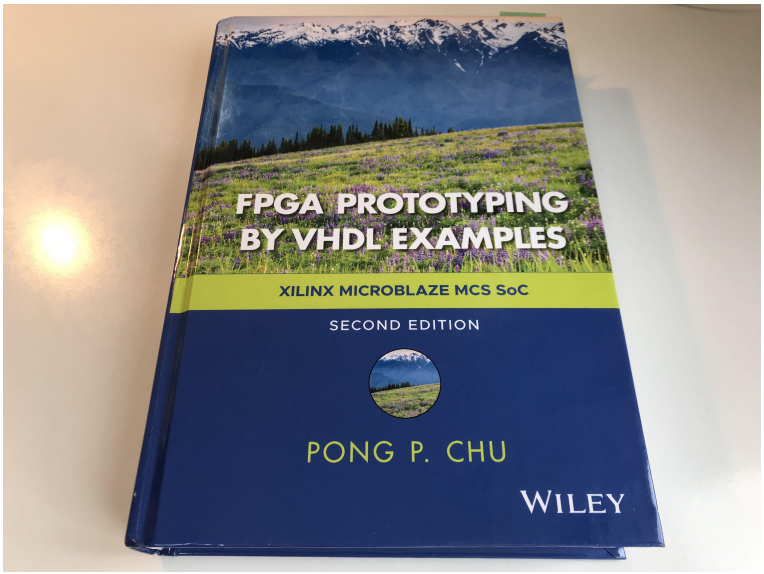

Figure 6.5: The book is imaged from an angle that gives a clear distortion due to the perspective.

Using Matlab, the coordinates of the 4 corner points of the book are found (manually). An alternative could be to find long lines in the image using the Hough transform and the locate the intersections, which must be book corners. Another alternative is to estimate the vanishing point in the image (where lines parallel to the vertical sides of the book would intersect). A transformation could then move this infinity point directly under the plane of the book.

[4] 4 points give 8 equations for the 8 degrees of freedom, that is the 9 parameters of the matrix minus 1 due to free scaling.

Then rectified (perspective corrected) coordinates are set up to get a rectangular outline of the book instead of a trapezoidal.

```
% Pick up the 4 corner points
% [x,y] = getpts
x = round(1.0e+03 * [1.1625, 2.9705, 0.6785, 3.5185]');
y = round(1.0e+03 * [0.0930, 0.0770, 2.8850, 2.8970]');

% Make rectangular
u = [x(1), x(2), x(1), x(2)]';
v = [y(1), y(1), y(3), y(3)]';

% Show the image and the planned homography
I = imread('img/chu1.jpg');
setlatexstuff('Latex');

figure;
imshow(I); hold on;
plot(x,y, 'r+');
line([u(1) u(2)], [v(1) v(2)], 'Color', 'green');
line([u(2) u(4)], [v(2) v(4)], 'Color', 'green');
line([u(1) u(3)], [v(1) v(3)], 'Color', 'green');
line([u(3) u(4)], [v(3) v(4)], 'Color', 'green');
hold off;
legend({'Selected corners', 'Targeted perspective'}, 'FontSize', 14);
title('Perspective-distorted image of book and targeted homography.', ...
        'FontSize', 18)
```

Perspective-distorted image of book and targeted homography.

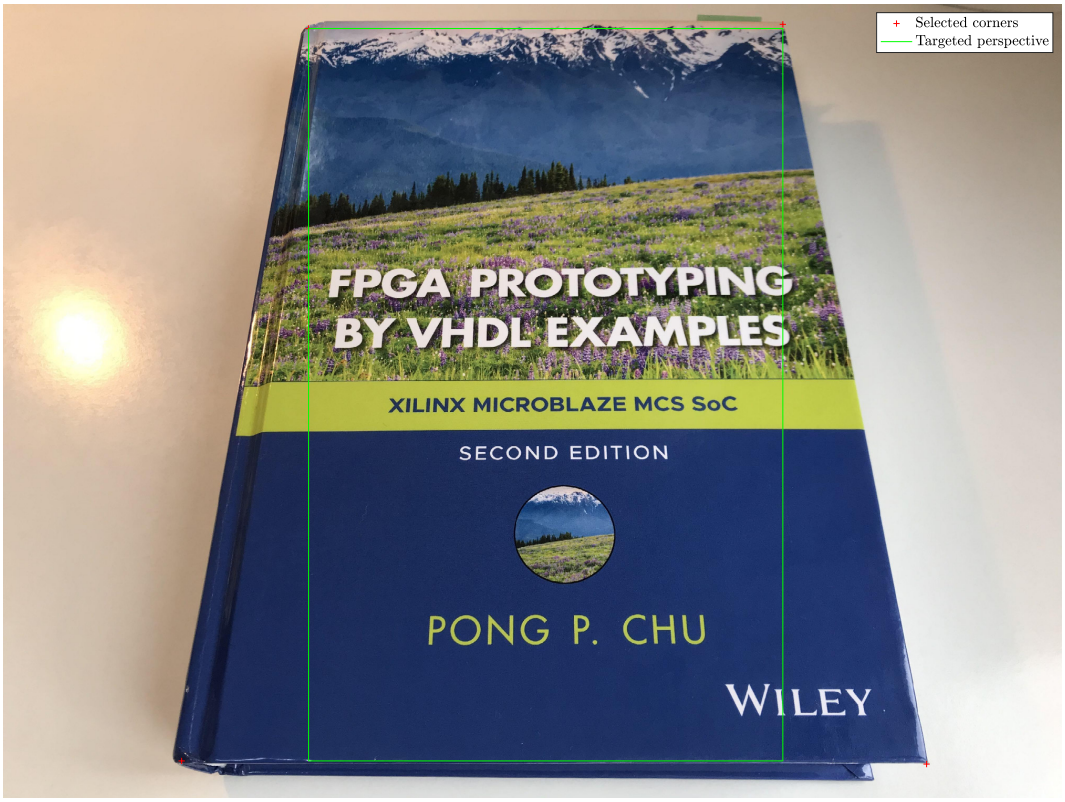

Figure 6.6:

The system of equations for the selected 4 points (8 equations) must be built. We are solving for the elements in H, and $h_{33} = 1$, so the 8 first elements of H are stacked into a vector $\bar{h}$. The point mappings from eqs. 6.11 and 6.12 give the system $A_{8\times8}\bar{h}_{8\times1} = \bar{b}_{8\times1}$.

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1u_1 & -y_1u_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2u_2 & -y_2u_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3u_3 & -y_3u_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4u_4 & -y_4u_4 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1v_1 & -y_1v_1 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2v_2 & -y_2v_2 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3v_3 & -y_3v_3 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4v_4 & -y_4v_4 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} \tag{6.15}$$

```
A = [ x(1) y(1)  1  0    0     0  -x(1)*u(1) -y(1)*u(1);
      x(2) y(2)  1  0    0     0  -x(2)*u(2) -y(2)*u(2);
      x(3) y(3)  1  0    0     0  -x(3)*u(3) -y(3)*u(3);
      x(4) y(4)  1  0    0     0  -x(4)*u(4) -y(4)*u(4);
      0    0     0  x(1) y(1)  1  -x(1)*v(1) -y(1)*v(1);
      0    0     0  x(2) y(2)  1  -x(2)*v(2) -y(2)*v(2);
      0    0     0  x(3) y(3)  1  -x(3)*v(3) -y(3)*v(3);
      0    0     0  x(4) y(4)  1  -x(4)*v(4) -y(4)*v(4);];
b = [u(1) u(2) u(3) u(4) v(1) v(2) v(3) v(4)]';
```

```
10
11 h = A\b;                  % Solve the linear system Ah=b
12 h = [h; 1];               % Append h33=1 for scaling
13 H = reshape(h, [3,3]);
```

The image is finally warped to perform the transformation, i.e., move coordinates to their new locations.

```
1 tform = projective2d(H);
2 I2 = imwarp(I,tform);
3 figure
4 montage({I, I2})
5 title('Perspective-corrected image of book.', 'FontSize', 18)
```

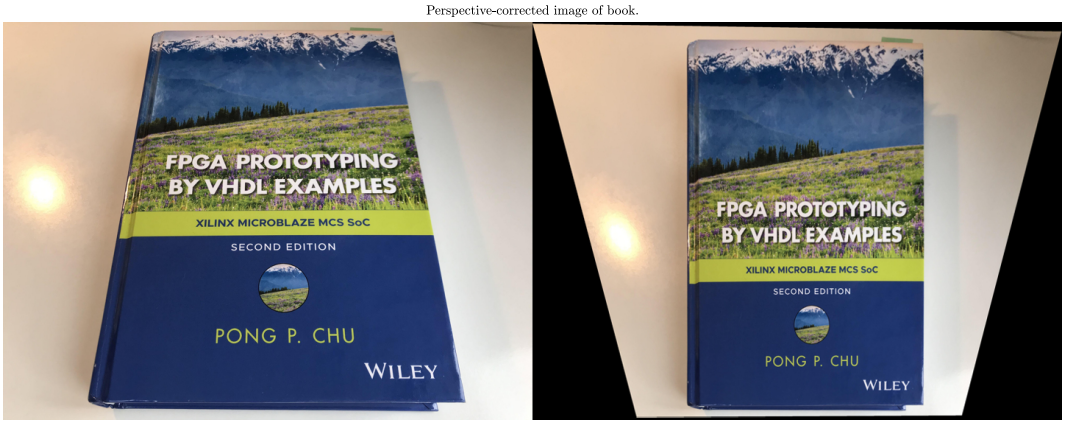

Figure 6.7:

The right-side image shows the new perspective, corrected as desired. A similar approach can be used for camera calibration. Then there are multiple images of a calibration plane with known dimensions, and the homography between calibration images is computed to back-out a calibration matrix for the camera.

6.6 Camera calibration

The setup to do camera calibration is seen in fig. 6.8, showing the planar checkerboard calibration target.

Figure 6.8: A checkerboard with 22×22 mm squares placed on a planar surface is used to calibrate the camera. It is available through the MATLAB command `open checkerboardPattern.pdf`. The camera is a Nikon D90 with a 35mm prime lens.

The MATLAB command `cameraCalibrator` opens the calibration app, and there is a good guide availabe at [14]. The camera is mounted with a fixed focal length (prime) lens, so the f camera parameter remains constant (further, avoid using autofocus). The aperture is set to a high F-stop to avoid a shallow depth of field.

Images are taken from different angles, and then loaded into the app, which will detect the checkerboard pattern, as shown in fig. 6.9. The side length of a square is 22 mm, this is entered into the app to give it scene knowledge.

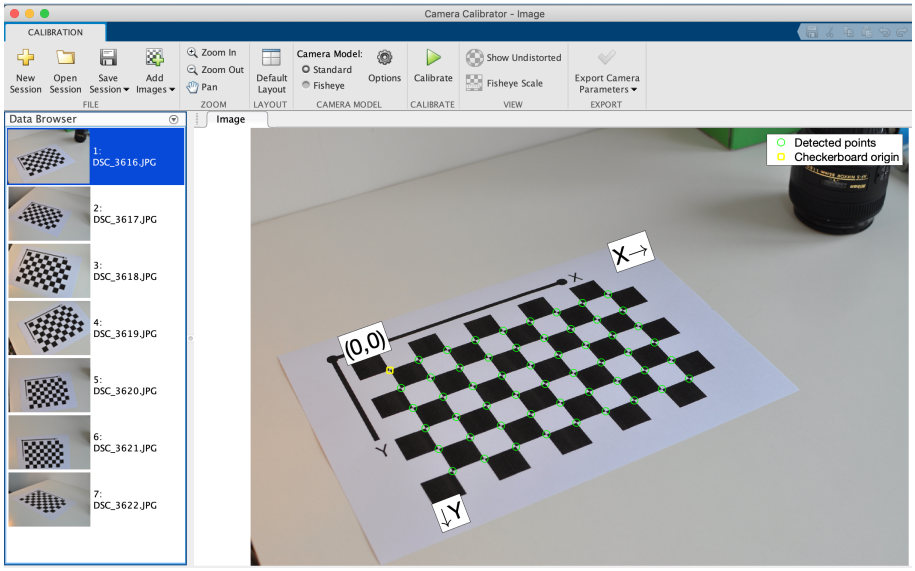

Figure 6.9: The calibrator app automatically detects the checkerboard. Seven images were loaded to make a calibration.

The relation to the 2D homography presented earlier is that the app must solve a series, here 7,

simultaneous 3D homographies: Each image must obey certain constraints in relation to the other images. A 3D homography has 15 parameters (matrix is $4 \times 4 - 1$ for homogeneous scaling). The intrinsic matrix has 6 parameters, but these are the same for each image. Further constraints arise to fit the size of the checkerboard squares. Each image will have 54 correspondences (corners on the checkerboard), so the system is clearly overdetermined, so it would be fitted using e.g. SVD. It would be interesting to implement this algorithm, and there appear to be two dominant methods, the Direct Linear Transform (DLT) and Zhang's Method [7, vid. 9 and vid. 10].

The finished calibration is shown in fig. 6.10. The reprojection errors show the quality of the calibration, and only image 7 which was taken from a difficult angle had significant errors.

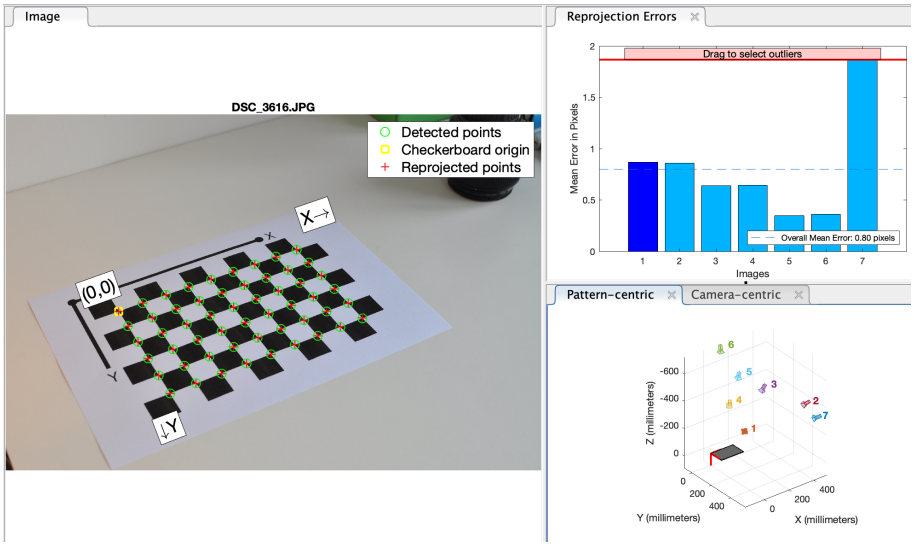

Figure 6.10: The finished calibration shows the fitted corners, the camera locations and the quality of the fit.

Finally, a `cameraParams` object is exported to the workspace. It contains the intrinsic and extrinsic camera parameters, as well as coefficients for non-linear adjustments, used to undistort images for radial and tangential distortion. The estimated intrinsics are

$$K = \begin{bmatrix} f_x & \hat{s} & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4.92e+03 & 0 & 1.61e+03 \\ 0 & 4.93e+03 & 1.05e+03 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.16}$$

The c_x and c_y values correspond well to being half of the pixel dimensions of images from the camera, 2136×3216. If images are resized, or the lens changed, the camera calibration is of course no longer valid.

7. Part 3: Sparse 3D scene reconstruction

This part of the pipeline uses two-view correspondences from the calibrated camera to perform a sparse 3D reconstruction of the scene in the image. Sparse meaning that only the tracked points are placed in 3D space.

7.1 Epipolar constraints

There is a constraint between two views of the same 3D point. A point in one view must map to a line in the other (an epipolar line), and vice versa.

The constraint is modeled by a matrix. Depending on the coordinates one has available for point correspondences, this is either the Essential matrix (E-matrix) for image plane coordinates, or the Fundamental matrix (F-matrix) for sensor coordinates. Sensor coordinates are exactly what is available in digital images, like in MATLAB, so the F-matrix is used in this report. There is an affine coordinate transformation between the image plane and sensor coordinates, described by the calibration matrix K, as was outlined in part 2. So F can be seen to generalize E.

The epipolar constraint is shown in terms of F in fig. 7.1.

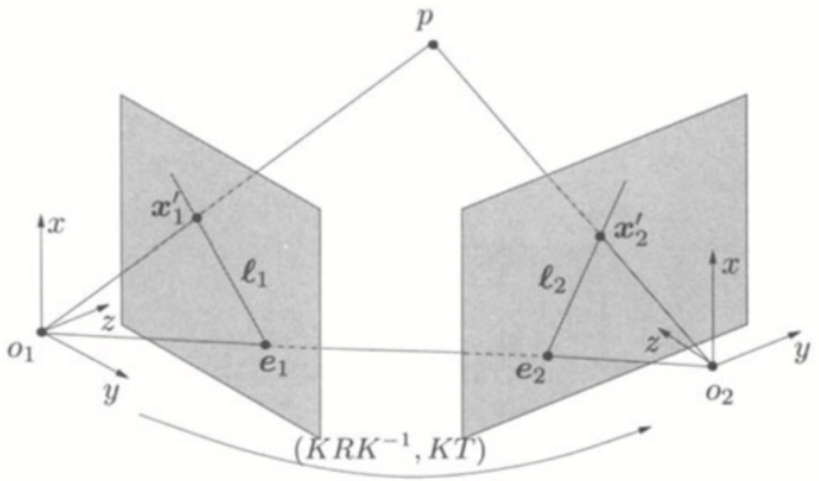

Figure 7.1: Two images of the point p, $\boldsymbol{x'_1}$ and $\boldsymbol{x'_2}$, arise from projection via the the two camera centres o_1 and o_2, respectively. The relative rotation between the projection centres is KRK^{-1} with K being the calibration matrix. The relative translation is KT, with T a translation vector. [1, p. 179].

The epipolar constraint says that the points o_1, p, o_2 must form a plane, and the projection points $\boldsymbol{x'_1}$ and $\boldsymbol{x'_2}$ must lie in this plane. So it is a co-planarity constraint. Also, any point along the ray o_1p is imaged as line ℓ_2 in view 2. And any point along the ray o_2p is imaged as line ℓ_1 in view 1. The lines ℓ_1 and ℓ_2 are the epipolar lines. The epipoles are the points e_1 and e_2, which is the where the baseline between o_1 and o_2 intersects the two planes.

If a correspondence was found between $\boldsymbol{x'_1}$ and $\boldsymbol{x'_2}$, but, say, $\boldsymbol{x'_2}$ does not lie on (near) ℓ_2, then an

epipolar outlier is identified.

The epipolar constraint is [1, eq. 6.9, p. 177]

$$\boldsymbol{x_2'}^\intercal F \boldsymbol{x_1'} = 0 \tag{7.1}$$

The matrix F contains the transformation between points, and is given by [1, eq. 6.12, p. 178]

$$F = \widehat{KT} K R K^{-1} \in \mathbb{R}^3 \tag{7.2}$$

Where the hat operator makes a skew symmetric matrix from a vector[1]. There is an epipolar constraint for each two-view correspondence, giving a set of simultaneous equations.

With noise and camera lens distortion, points do not line up exactly as required, and as the equation system should also be overdetermined (more point correspondences than constraints), the F-matrix must be fitted using a robust estimation technique. Using the iterative RANSAC method, one will obtain a consensus estimate of which correspondences are outliers (epipolar outliers), and probably spurious.

7.2 Point correspondences in a richer scene

A new scene has been imaged, with objects that are richer in features than the ones analysed in previous chapters. The scene is shown below.

The calibration parameters are used again here to undistort images before they are processed. This reduces the risk of epipolar outliers later for estimation of the Fundamental matrix.

```
clc; close all; clear;
load('data/camera_calib.mat');      % loads cameraParams into workspace

scene_1 = im2double(imread('img/scene/DSC_3637.jpg'));
scene_2 = im2double(imread('img/scene/DSC_3638.jpg'));

% Undistort images using the calibrated camera parameters
im1 = undistortImage(scene_1, cameraParams);
im2 = undistortImage(scene_2, cameraParams);

figure; imshowpair(im1,im2,'montage');
title(['Two-view scene, left and right views. ', ...
    'Acquired with calibrated camera and undistorted.'], 'FontSize', 18);
```

[1] It is not crucial here, but for completeness, the cross product between vectors u and v from $\mathbb{R}^3$, $u \times v$ can be encoded as $\hat{u}v$, where $\hat{u} = \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix}$ is a skew symmetric matrix.

Two-view scene, left and right views. Acquired with calibrated camera and undistorted.

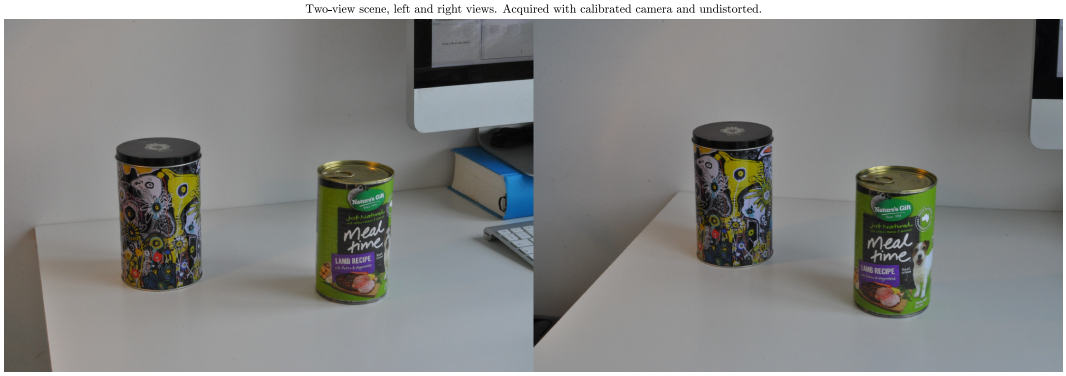

Figure 7.2:

7.2.1 Running the correspondence pipeline

The feature tracking functions are run with the same settings as before, except that more target points are sought out.

```
% Settings for the processing pipeline
target_points = 400;        % Force finding this many points in each im
border_skip = 20;           % Pixels in border are ignored
k = 0.04;                   % Weighting parameter, 0.04 default by Harris
nd = 256;                   % BRIEF-32
S = 33;                     % S x S patches
T = 50;                     % compare to max Hamming distance n_d=256
display_type = 'false';     % Display the matches
stat_outliers = true;       % Do statistical removal of outliers

% Implemented correspondence processing pipeline
X1 = harris_corners(im1, target_points, k, border_skip);
X2 = harris_corners(im2, target_points, k, border_skip);
rng(1); geom = brief_geom(nd, S);
im1_fds = brief_fd(im1, X1, S, geom);
im2_fds = brief_fd(im2, X2, S, geom);
[M1, M2, match_info] = brief_matches(X1, X2, im1_fds, im2_fds, ...
                                     T, stat_outliers);

% Display test images
show_test_images_with_features(im1, im2, X1, X2, target_points);
```

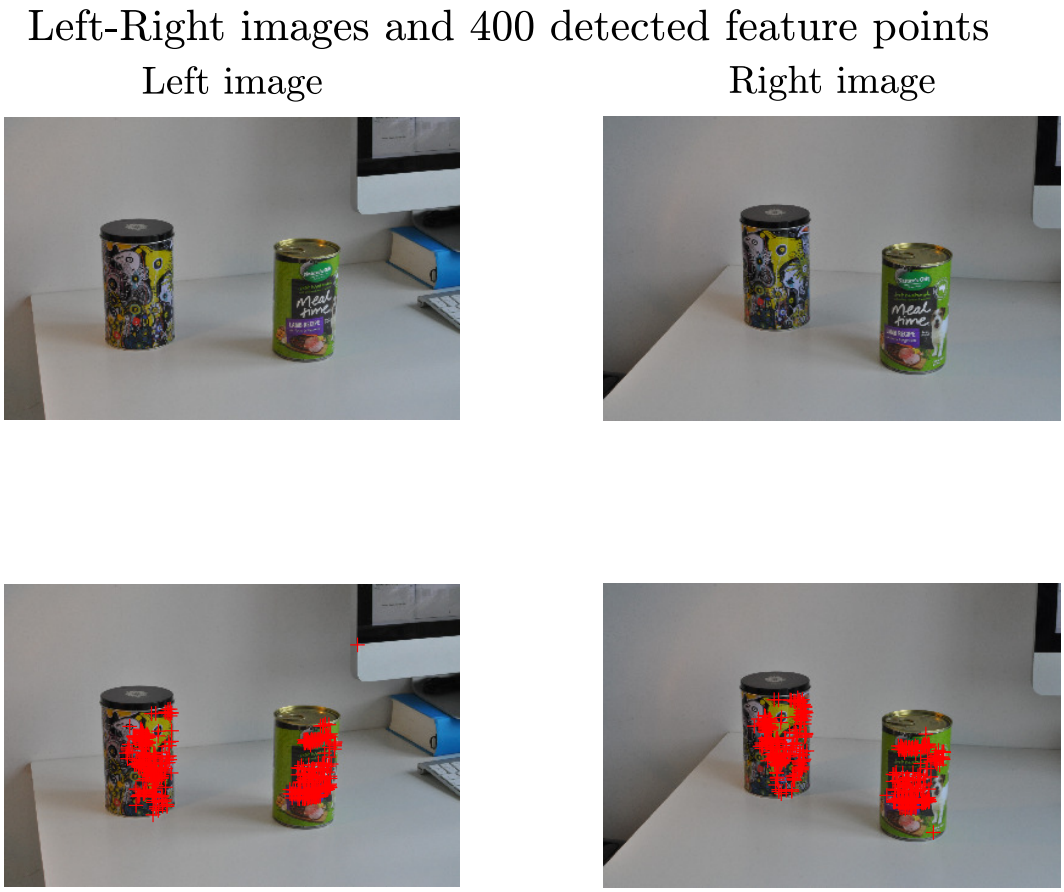

Figure 7.3:

Show correspondences

```
correspondence_show_matches(im1, im2, M1, M2, match_info, display_type);
```

Figure 7.4:

The figures show that out of 400 Harris corners, 87 can be tracked across the two views. This should be sufficient to get convergence in the estimation of the fundamental matrix.

7.3 Estimating the Fundamental matrix and relative pose

The Computer Vision toolbox is used to estimate F and get the inliers. Outlier correspondences are not used further.

```
% Compute fundamental matrix for matched points
[fundamental_matrix, epipolar_inliers] = estimateFundamentalMatrix(...
  M1, M2, 'Method', 'MSAC', 'NumTrials', 100000);

% Find epipolar inliers, i.e. those that fulfill the epipolar constraint
inlier_points1 = M1(epipolar_inliers, :);
inlier_points2 = M2(epipolar_inliers, :);

% Display inlier matches
figure
showMatchedFeatures(im1, im2, inlier_points1, inlier_points2);
title(['Matched points after removal of epipolar outliers, $m=$ ', ...
    num2str(length(inlier_points1))]);
```

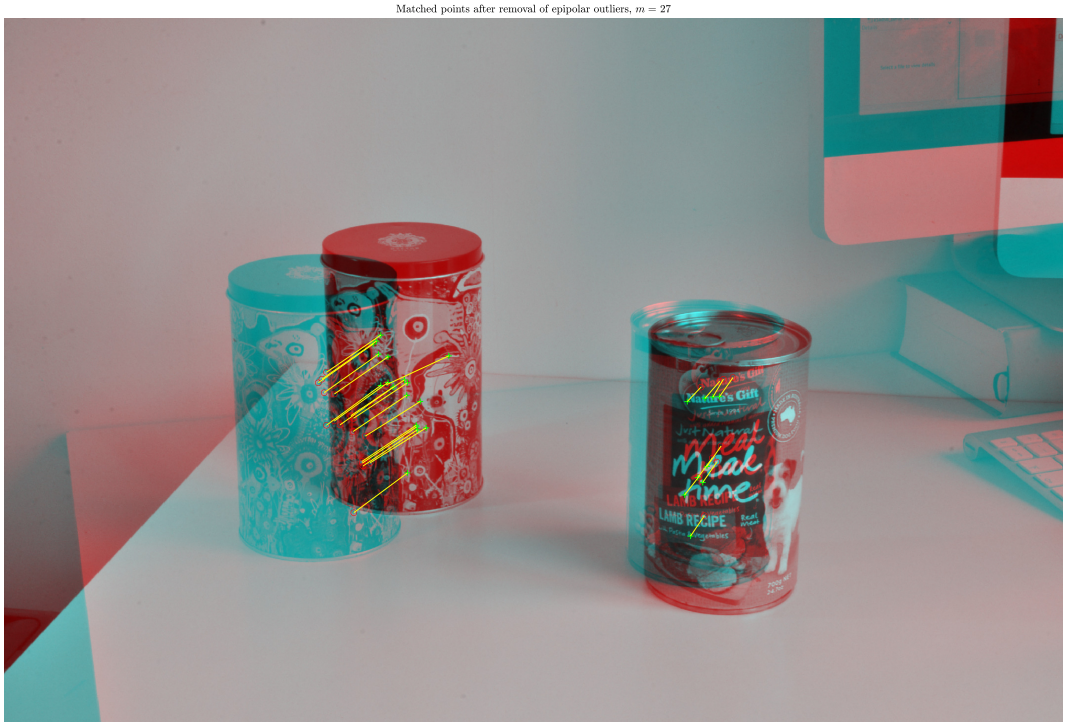

Figure 7.5:

The figure shows that there are fewer inlier points than correspondences. 27 matches remain, so 60 correspondences have been removed. As most of the 87 matches seemed OK by visual inspection, this indicates either errors in the acquisition process (probably used autofocus) or a camera calibration that should be improved. There are however still enough points to continue.

The relative pose from view 1 to view 2 is estimated using another built-in from the Computer Vision Toolbox. The estimated quantities are the R and T from eq. 7.2.

```
1 [R, t] = relativeCameraPose(fundamental_matrix, cameraParams, ...
2     inlier_points1, inlier_points2);
```

R and T could be used directly as an estimate of relative motion in a SLAM system.

7.4 Inverting projections and triangulation

The camera projection matrices for view 1 and view 2 must be computed. These are the composition sequences from world coordinates to sensor coordinates, as in eq. 6.4. Subsequently, world points can be found using triangulation. Both steps are done using built-ins from the Computer Vision Toolbox.

A note on the use of the `cameraMatrix` function: View 2's position relative to world coordinates must be given using coordinates in the *view 2 reference frame*. As view 1 can be taken as the world coordinates, the inverse of the transformation (R, T) from $1 \to 2$ is needed. For $2 \to 1$ the rotation is $R^{-1} = R^\intercal$. The translation vector T is in view 1 coordinates, so it must be converted to view 2 coordinates, which is by rotation to match view 2 orientation, and a negative sign to point the vector back to o_1. MATLAB has translation vectors as row vectors, so these are transposed versus the formulas.

```
% Compute the camera matrices for each position of the camera
% The view 1 camera is at the origin looking along the X-axis. So, its
% rotation matrix is identity, and its translation vector is 0.

cam_matrix1 = cameraMatrix(cameraParams, eye(3), [0 0 0]);
cam_matrix2 = cameraMatrix(cameraParams, R', -(R*t')');

% Compute location of world points by triangulation
world_points = triangulate(inlier_points1, inlier_points2, ...
    cam_matrix1, cam_matrix2);

% Show partially reconstructed 3D scene with camera positions
reconstruction_plot(world_points, R, t);
```

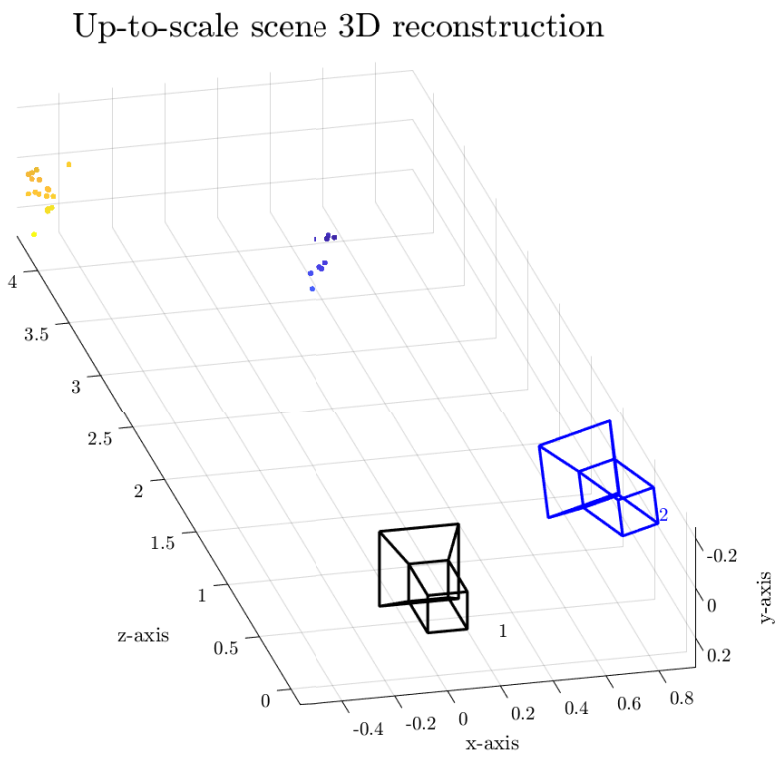

Figure 7.6:

The reconstruction appears similar to the scene. It is not to scale, of course, but there is clearly separation between the point clouds from the two objects. The camera locations are similar to the motion done when acquiring the images. The world points can now be used to segment the scene. One could also imagine picking out cubes from the 3D space (if no occlusions), but in this project, the L_2 norm from the world origin will just be used.

8. Part 4: Depth-based segmentation

The final part of the project and the 3D vision pipeline is to segment the scene based on the 3D points. This is the "easiest" part of the pipeline to implement. It is probably also the one that will require the most scene knowledge to work well.

8.1 Selecting distance to segment

The method implemented here is to segment based on the average distance, μ, of objects in the scene, measured from camera position 1 (world origin). Given the world points, the L_2 norm (Euclidian distance) is computed. Discrimination is done as e.g. $\| (X)_i \| < \mu$. The scene is only up-to-a-scale, so no absolute world distance can be used. The matrix of world points is $m \times 3$ where m is the number of epipolar matches. Each row is an $[x, y, z]$ vector for $(X)_i$.

```
L2 = vecnorm(world_points');     % transpose so (x,y,z) vector in each col
mean_dist = mean(L2);            % Find the norm of each point

% Pick out the indexes of points far/near
segm_far_idx = world_points(:,3) > mean_dist;     % Discriminate far
segm_near_idx = world_points(:,3) < mean_dist;    % Discriminate near

% Get the actual pixel positions for inliers in image 1
segm_far_pixels = inlier_points1(segm_far_idx, :);
segm_near_pixels = inlier_points1(segm_near_idx, :);

% Get the linear index for the (x,y) coordinate points for easy lookup
im_far_idx = sub2ind(size(im1), ...
                         segm_far_pixels(:,2), segm_far_pixels(:,1));
im_near_idx = sub2ind(size(im1), ...
                         segm_near_pixels(:,2), segm_near_pixels(:,1));
```

8.2 Pre-processing the image into labelled regions

Because the 3D reconstruction is sparse, the image needs to be seperated into contiguous regions that can be picked out. The method is to

- Threshold the image to get binary a binary image. It can be done automatically using Otsu's Method [12, p. 747].
- It however requires knowing that the background is white, so that the objects of interest have intensity below the threshold.
- The surfaces of objects will be broken up by the thresholding, so regions are not necessarily completely contiguous or may contain holes.

- Morphological closing (erosion of the dilation), will close gaps in surfaces.
- The structuring element size is set relative to the image size, 1 pct. of the smallest dimension (height) is used. A disk is used to get round-ish edge smoothing (less jagged edges).
- Any enclosed holes are filled to get a better surface. A hole is defined as a region of background pixels that cannot be reached in a fill operation starting from an image edge pixel.
- Contiguous regions are finally labelled using 8-connected neigbours (default), making larger regions than 4-connected only.
- Find label ids for the pixels that are categorised as far and near.
- The majority vote is used in case a few should have been wrongly categorised.

```
im1_gray = rgb2gray(im1);           % Grayscale image
T_otsu = graythresh(im1_gray);      % Using Otsu 1979 method for threshold
im1_thresh = im1_gray < T_otsu;     % Pick out the non-white parts

% Close the image morphologically to make larger connected regions
disk_size = round(0.01 * size(im1,1));  % Disk size of about 1% of pxheight
se = strel('disk', disk_size);
close_bw = imclose(im1_thresh,se);      %Morphological closing

close_bw = imfill(close_bw,'holes');    % Fill holes

% label the different connected regions in the image
bwconn = bwlabel(close_bw);

% Extract the connected region labels for all these pixels
far_labels = bwconn(im_far_idx);
near_labels = bwconn(im_near_idx);

% Perform majority voting
% Find most frequent value, that is majority voting decides
far_lbl = mode(far_labels);
near_lbl = mode(near_labels);

figure; imshow(label2rgb(bwconn));
hold on;
plot(segm_far_pixels(:,1), segm_far_pixels(:,2), 'r+')
plot(segm_near_pixels(:,1), segm_near_pixels(:,2), 'g+')
hold off;
legend({'Far points', 'Near points'}, 'FontSize', 18);
title('Connected regions with near/far dist. information in the scene', ...
    'FontSize', 22)
```

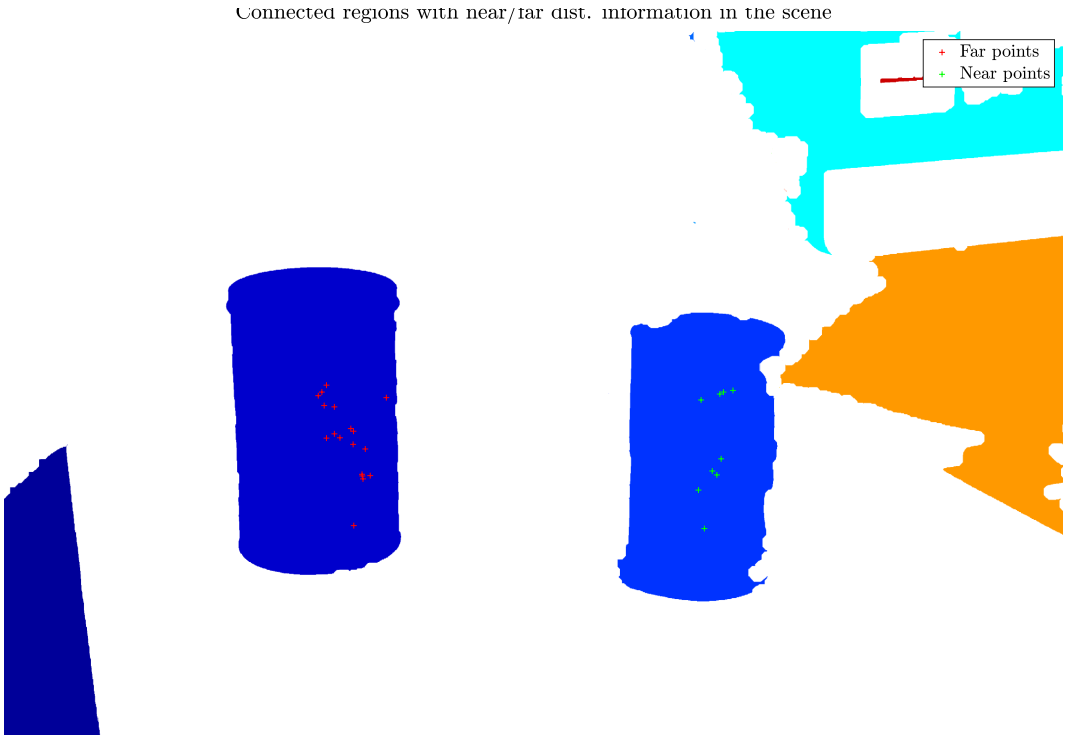

Figure 8.1:

8.3 Segmentation of the image and selection of objects

Finally, the image is segmented using regions based on the majority vote. A white background is forced for prettier printing.

```
% Process image, so unselected area is white
im_far = im1.*(bwconn == far_lbl);   % Pick out the far region
for rgb = 1:3
    im_far(:,:,rgb) = im_far(:,:,rgb) + 1.0*(im_far(:,:,rgb) == 0);
end

im_near = im1.*(bwconn == near_lbl); % Pick out the near region
for rgb = 1:3
    im_near(:,:,rgb) = im_near(:,:,rgb) + 1.0*(im_near(:,:,rgb) == 0);
end
```

The segmented image shows far and near regions.

```
figure;
imshowpair(im_far, im_near, 'false');
title('Depth-based scene segmentation (left:far, right:near)', ...
    'FontSize', 20);
```

Depth-based scene segmentation (left:far, right:near)

Figure 8.2:

Particular objects can be picked out with an almost complete surface.

```
figure;
imshow(im_far);
title('Depth-based scene segmentation, far object selection', ...
    'FontSize', 20);
```

Figure 8.3:

The above two figures show that the segmentation algorithm works as intended.

8.4 Evaluation

The scene segmentation works as desired. The algorithm can segment the scene and pick out far and near objects.

There are only two objects in this scene that are present in both views, and the background is mostly a flat white, so it is not a complex scene. For a more populated and complex scene, the algorithm should:

- Be expanded to discriminate into more groups. Either still using distances or other 3D based discrimination (pick out cubic slices of the world?).
- Use more points to get better 3D to 2D correspondence (better spatial distribution of correspondences, fewer epipolar outliers), which will require improving the image acquisition process.

The surface of the near object is not as nice as could have been hoped. The main reason is that the dog on the dogfood can is white, so it is removed by the thresholding. The lighting during acquisition was not ideal either. Further image processing would probably have made it possible to connect it into its region better, but that was not really the main focus here.

9. Next steps and future work

This section outlines ideas for improvements and future work.

- A first step would be to investigate in-depth how to reduce the number of epipolar outliers in the vision system. Perhaps a different lens, completely manual focus or similar could improve the acquisition. Potentially, more images could be used for camera calibration, also using the *exact same* setup as will be used to acquire images.
- The discrimination algorithm can be expanded to segment and pick out objects from a more complex scene.
- The Harris detector in its current form is somewhat slow. It takes a few seconds to process a 2136×3216 image. Smaller images could be used, but there is probably many ways to optimise the time performance. Either algorithmically in MATLAB, or by using a more performant programming language. Ideally, the algorithm should be OK to run on an embedded system, e.g. a drone.
- It would be useful to implement the ORB descriptor to get rotationally more robust feature tracking, while maintaining the strong performance of a binary descriptor.
- A benchmarking against other feature trackers would be useful and educational.
- Independent development of the algorithms for estimating the fundamental matrix and performing triangulation would be useful, in particular with the aim of implementing algorithms on embedded systems that will not run MATLAB, and perhaps not even OpenCV.
- Denser point cloud estimation would be useful for richer applications that can leverage more processing power, like self-driving cars.

10. Conclusion

This project has developed and demonstrated a prototype 3D vision pipeline. The system uses correspondences between two views acquired with one calibrated camera to perform partial 3D scene reconstruction and do depth-based image segmentation.

Special emphasis has been on investigating and implementing algorithms to solve the correspondence problem. Here, the Harris corner detector, the BRIEF-n feature descriptor and brute-force search with the Hamming distance as metric combined with Lowe's Ratio test have all been implemented and form the backbone of the solution.

Secondary emphasis has been placed on using the 3D scene reconstruction to do image segmentation by using depth information from the scene. Several methods from the E5ADSB course are used here,

among other thresholding, binary image labelling, and morphological operations.

Key concepts related to image formation have been reviewed. An overview of some of the literature and materials in the field has also been developed as part of the research for the project.

Overall, it has been a very interesting and somewhat challenging project. The scope of the requirements and project is also underlined by a rather lengthy report. It is hoped that the reader will look at the material that is most interesting and relevant, and not necessarily read from cover-to-cover.

Nonetheless, this work forms a good foundation to dive deeper into computer vision and its many interesting application areas.

11. References

[1] Y. Ma et al. *An Invitation to 3-D Vision*. First Edition. Springer-Verlag New York, ISBN: 978-0-387-00893-6, 2004. URL: https://www.springer.com/gp/book/9780387008936.

[2] David C. Lay, Steven R. Lay, and Judi J. McDonald. *Linear Algebra and Its Applications (5th Edition)*. Pearson, 2015. ISBN: 978-0321982384.

[3] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Second Edition. Cambridge University Press, ISBN: 0521540518, 2004.

[4] Y. Ma et al. *An Invitation to 3-D Vision*. Manuscript. 2001. URL: https://www.eecis.udel.edu/~cer/arv/readings/old_mkss.pdf.

[5] Daniel Cremers. *Computer Vision 2: Multiple View Geometry (2014/2015)*. Accessed: 2020-09-28. URL: https://vision.in.tum.de/teaching/online/mvg.

[6] M. Calonder et al. "BRIEF: Computing a Local Binary Descriptor Very Fast". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34.7 (2012), pp. 1281–1298. DOI: 10.1109/TPAMI.2011.222.

[7] Cyrill Stachniss. *Photogrammetric Computer Vision (2020)*. Accessed: 2020-11-20. URL: https://www.youtube.com/watch?v=X7AghhqMUm8&list=PLgnQpQtFTOGTPQhKBOGgjTgX-mzpsOGOX.

[8] Cyrill Stachniss. *Photogrammetry I and II (2015/2016)*. Accessed: 2020-11-20. URL: https://www.youtube.com/watch?v=_mOG_lpPnpY&list=PLgnQpQtFTOGRsi5vzy9PiQpNWHjq-bKN1.

[9] Séan Mullery. *Multiple View Geometry in Computer Vision, MEng in Autonomous Vehicles (2019)*. Accessed: 2020-11-20. URL: https://www.youtube.com/playlist?list=PLyH-5mHPFffFvCCZcbdWXAb_cTy4ZG3Dj.

[10] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Second Edition. Accessed: 2020-09-20. 2020. URL: https://szeliski.org/Book/.

[11] Richard Szeliski. *Computer Vision: Algorithms and Applications*. First Edition. Springer London, 2010. ISBN: 978-1-84882-935-0. URL: http://szeliski.org/Book/1stEdition.htm.

[12] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. 4th Edition. Pearson, 2018. ISBN: 978-1-292-22304-9.

[13] Robert Collins. *Lecture 06: Harris Corner Detector (slides)*. Accessed: 2020-11-20. URL: http://www.cse.psu.edu/~rtc12/CSE486/lecture06.pdf.

[14] Mathworks. *Single Camera Calibrator App*. Accessed: 2020-11-20. URL: https://au.mathworks.com/help/vision/ug/single-camera-calibrator-app.html.

12. Appendix: Implemented functions

During the project, a number of functions were implemented. These are included in the following sections.

12.1 setlatexstuff

```
function [] = setlatexstuff(intpr)
% Settings for LaTeX layout in figures
% intpr (interpreter): 'Latex' or 'none'
% Janus Bo Andersen, 2019

    set(groot, 'defaultAxesTickLabelInterpreter',intpr);
    set(groot, 'defaultLegendInterpreter',intpr);
    set(groot, 'defaultTextInterpreter',intpr);
    set(groot, 'defaultGraphplotInterpreter',intpr);

end
```

12.2 Feature detection: Harris surface plot

```
function [] = harris_surface(H, image, point, nhood)
% Plots contour, surface and image for a neighbourhood
% H: Harris response matrix (m x n)
% image: (m x n x r) image
% point: (1 x 2) vector of (x,y) central point
% nhood: (1 x 2) vector of surrounding points
%
% Janus Bo Andersen, November 2020

    x_range = point(1) - nhood(1):point(1) + nhood(1);
    y_range = point(2) - nhood(2):point(2) + nhood(2);

    image_nhood = image(y_range, x_range);
    harris_resp = H(y_range, x_range);

    Z = harris_resp;
    dimZ = size(Z);
    [Y,X] = meshgrid(1:dimZ(2),1:dimZ(1));
    top_z = sort(reshape(Z, [],1), 'descend');
    [top1_row, top1_col] = find(Z == top_z(1));
    [top2_row, top2_col] = find(Z == top_z(2));

```

```
    figure;
    subplot(2,2,1);
    contour(Y, X, Z);
    xlabel('x'); ylabel('y'); grid on; hold on;
    plot3(top1_col, top1_row, top_z(1), 'r+'); % plots normal (x,y)-order
    plot3(top2_col, top2_row, top_z(2), 'ro');
    set(gca,'Ydir','reverse')
    hold off;
    title('Harris response contour map', 'FontSize', 16)
    legend({'H(x,y) = R', 'Max 1', 'Max 2'}, 'FontSize', 10, ...
        'Location', 'South')

    subplot(2,2,2);
    imshow(image_nhood); hold on;
    plot(top1_col, top1_row, 'r+'); % plots like (x,y)
    plot(top2_col, top2_row, 'ro');
    hold off;
    title('Image region', 'FontSize', 16)
    xlabel('x'); ylabel('y'); axis on;

    subplot(2,2,3:4)
    surf(Y, X, Z, 'FaceAlpha',0.5);
    xlabel('x'); ylabel('y'); zlabel('$H(x,y)$'); hold on;
    plot3(top1_col, top1_row, top_z(1), 'r+'); % plots normal (x,y)-order
    plot3(top2_col, top2_row, top_z(2), 'ro');
    set(gca,'Ydir','reverse')
    hold off;
    title('Harris response surface', 'FontSize', 16)

    sgtitle('Maximum Harris responses in an image region', 'FontSize', 16)

end
```

12.3 Feature detection: Harris Corner detector

```
function [X] = harris_corners(input_image, num_corners, k, border_ignore)
% Harris feature point detector
% Recommended that image is smoothed first to de-noise
%   Janus Bo Andersen, Nov 2020

    % 3x3 sobel operators (kernels)
    Dy = fspecial('sobel');
    Dx = Dy';


    I2 = im2double(rgb2gray(input_image));
```

```
    I2 = conv2(I2, fspecial('gaussian', 9), 'same'); % de-noising

    % Compute elements before structure matrix
    Jx = conv2(I2, Dx, 'same');
    Jy = conv2(I2, Dy, 'same');

    JxJx = Jx .^ 2;
    JyJy = Jy .^ 2;
    JxJy = Jx .* Jy;

    % Box filter to sum regions / patches
    % consider using a Gaussian instead
    B = ones([5 5]);

    % Compute sums, so there is a structure matrix M at each pixel
    sum_JxJx = conv2(JxJx, B, 'same');
    sum_JxJy = conv2(JxJy, B, 'same');
    sum_JyJy = conv2(JyJy, B, 'same');

    % At each pixel, compute the response
    %k = 0.04;               % As proposed by Harris
    x_max = size(I2, 1);
    y_max = size(I2, 2);

    % Compute Harris response for each pixel
    % Ignore border area
    H = zeros(size(I2));
    for x = border_ignore:x_max-border_ignore
        for y = border_ignore:y_max-border_ignore
            M = [sum_JxJx(x,y) sum_JxJy(x,y);
                 sum_JxJy(x,y) sum_JyJy(x,y)];
            R = det(M) - k * trace(M)^2;
            H(x,y) = R;
        end
    end

    % Non-maximum suppression in neighbourhood
    % Find local maxima, then suppress all other values
    % Ordfilt2 finds the max value in the neighbourhood
    % This max value is used for logic filter

    % Possible other methods: ordfilt2, imdilate, watershed, findpeaks (2d
       ?)
    d = 9;                                      % neighbourhood is d x d (
       square structure elem.)
    Hmax = ordfilt2(H, d*d, ones([d d]));       % Replace with max value in
        nborhood (d^2 th of d^2 sorted values)
```

```
56
57      % Suppress non-maxima (only retain the local maxima)
58      Hlocalmax = H .* (H == Hmax);
59
60      % Thresholding
61      % Set threshold so only N most significant corners are included
62      % Consider doing local sorting to get better spatial distribution
63      % Would amount to adaptive thresholding
64      % num_corners = 1000;
65      sorted_responses = sort(reshape(Hlocalmax, [], 1), 'descend');  %
            global sorting
66      threshold = sorted_responses(num_corners);      % Nth highest value
            becomes the threshold
67
68      % Corners are the local maxima with values higher than threshold
69      corners = (Hlocalmax >= threshold);
70
71      % Get locations where we've found a corner (binary image is True)
72      [y, x] = find(corners);
73      X = [x, y];      % reverse order to get pairs as (xi, yi)
74
75  end
```

12.4 Feature description: Show test images and feature points

```
1   function [] = show_test_images_with_features(im1,im2, X1,X2, points)
2   % Displays two L-R test images and their feature points.
3   %
4   % Janus Bo Andersen, Nov 2020
5
6       figure;
7       subplot(2,2,1); imshow(im1); title('Left image', 'FontSize', 16);
8       subplot(2,2,2); imshow(im2); title('Right image', 'FontSize', 16);
9       subplot(2,2,3); imshow(im1); hold on;
10      plot(X1(:,1), X1(:,2), 'r+'); hold off;
11      subplot(2,2,4); imshow(im2); hold on;
12      plot(X2(:,1), X2(:,2), 'r+'); hold off;
13      sgtitle(['Left-Right images and ', num2str(points), ...
14          ' detected feature points'], 'FontSize', 18);
15
16      drawnow;
17
18  end
```

12.5 Feature description: BRIEF - Create Spatial Sampling Geometry

```
function [geom] = brief_geom(nd, S)
% Creates a test geometry for nd intensity comparisons.
% Terminology is also BRIEF-k, where k=nd/8 is the number of bytes required
% to store a descriptor.
% Spatial geometry is method G2 from Calonder et al. (2012), using a using
% a bivariate i.i.d. Gaussian.
%
% Janus Bo Andersen, Nov 2020.
%
% Input arguments:
%   nd (int): Number of tests.
%   S (int): Patch size, S x S pixels centered around a feature point.
% Returns:
%   geom (matrix): Matrix with (rows, cols) = (nd x 4)
%                  Cols are [u v q r] corresp. to coord. pairs
%                  x=(u,v)' and y=(q,r)'.

    % Draw from bivariate normal distribution as described in e.g.
    % https://en.wikipedia.org/wiki/Multivariate_normal_distribution#...
    % Drawing_values_from_the_distribution
    mu = [0 0]';          % Mean vector giving central location in patch
    var = S^2/25;         % Gaussian variance for method 2
    C = diag([var var]); % Covariance matrix: isotropic, bivariate i.i.d.
    A = chol(C);          % Cholesky decomposition such that A'*A = A

    % Draw (4 x nd) samples of the bivariate distribution in one go
    % we are sampling location geometries in vectors of pairs
    % x=(u,v)' and y=(q,r)', so we can compare the image as I(x) < I(y).
    z1 = randn([2 nd]);                 % Sample std. normal
    z2 = randn([2 nd]);
    x = repmat(mu, [1 nd]) + A'*z1;
    y = repmat(mu, [1 nd]) + A'*z2;
    XY = [x ; y];                       % stack for easier manipulation

    % Round and clamp to stay within coordinates and patch dimensions
    idx_lower = XY < ceil(-S/2);  % clamp at lowest negative limit
    idx_upper = XY > floor(S/2);  % clamp at highest positive limit
    idx_round = ~(idx_lower | idx_upper);     % all others rounded

    % Perfom clamping and rounding
    XY(idx_lower) = ceil(-S/2);
    XY(idx_upper) = floor(S/2);
    XY(idx_round) = round(XY(idx_round), 0);
```

```
44
45      % Columns are [u v q r] corresp. to coord. pairs x=(u,v)' and y=(q,r)'.
46      geom = XY';
47
48  end
```

12.6 Feature description: BRIEF - Show the sampling geometry

```
1   function [] = brief_display_geom(geom)
2   % Display the geometry generated by brief_geom
3   % Janus Bo Andersen, Nov 2020
4
5
6   % Image patch at coordinate offsets (u,v) is compared to (q,r)
7   u = geom(:,1);
8   v = geom(:,2);
9   q = geom(:,3);
10  r = geom(:,4);
11
12      % see distributions and geometry
13      figure;
14      subplot(2,2,1)
15      histogram2(u, v)
16      title('Bivariate distribution for $x=(u,v)$');
17      xlabel('u'); ylabel('v');
18      zlabel('Num')
19
20      subplot(2,2,2)
21      histogram2(q, r)
22      title('Bivariate distribution for $y=(q,r)$');
23      xlabel('q'); ylabel('r');
24      zlabel('Num')
25
26      subplot(2,2,3:4)
27      line([u(1) v(1)], [q(1) r(1)], 'Marker', '+'); hold on;
28      for n = 2:length(u)
29          line([u(n) v(n)], [q(n) r(n)], 'Marker', '+');
30      end
31      grid minor;
32      title('$(x,y)$ test pairs for $n_d=256$ tests')
33      xlabel('First coordinate offset'); ylabel('Second coordinate offset');
34      hold off;
35      sgtitle('Test geometry for BRIEF descriptor: $n_d=256$, $S=33$')
36
37  end
```

12.7 Feature description: BRIEF - Compute Feature Descriptor

```
function [fds] = brief_fd(image, feature_points, S, geom)
% Computes a BRIEF feature descriptor at a feature point.
% The patch is smoothed with a gaussian kernel before descriptor.
% This reduces the noise sensitivity.
% Based on Calonder et al. (2012).
%
% Janus Bo Andersen, Nov. 2020.
%
%   Input arguments
%       image: image with feature points
%       feature_points: (Nx2) matrix of the N feature points, i.e. coords.
%            in image. With N feature points, and coords as
%            (x,y) = col, row in image.
%       S: Patch size, e.g. S=33 => patch: 33 x 33 pixels
%       geom: nd-Spatial geometry for test comparisons, columns [u,v,q,r]
%   Returns:
%       fds: (N x nd) feature descrips. Each descrip. is a row of nd-length

    ima = image;
    fps = feature_points;

    N = length(fps);
    nd = length(geom);

    fds = boolean(zeros([N nd]));   % Set up return matrix

    % Set up gaussian de-noising, as outlined in paper
    gaussian_kernel_variance = 2;   % as paper p. 1283
    gaussian_kernel_window = 7;     % 7x7 kernel, ibid.
    kernel = fspecial('gaussian', ...
        gaussian_kernel_window, ...
        gaussian_kernel_variance);


    % Compute feature descriptor for a single feature point per loop
    for fp  = 1:N

        % Get central coordinates for feature point
        xc = fps(fp, 1);
        yc = fps(fp, 2);

        % Get neighbourhood
        % grayscale image of patch centered on (xc, yc)
        h = (S-1)/2;    % half side
```

```
        nhood = rgb2gray(imcrop(ima, [xc-h yc-h S-1 S-1]));

        p = imfilter(nhood, kernel);    % pixel intensity function p(x)

        % Perform the K comparisons on nd randomly sampled pairs of points
        % tau(p, x, y) = { 1 if p(x) < p(y), else 0

        m = (S+1)/2;            % midpoint, e.g. (17,17) for a 33x33 patch

        % Perform intensity test, all test points are offset from mid
        % Make linear index to easily pull out pixel values vectorized
        % Example: x_idx = sub2ind(size(p), [1 1 33], [1 2 33])
        %  -> [1 34 1089] for a 33x33 image
        x_idx = sub2ind(size(p), m+geom(:, 2), m+geom(:, 1));
        y_idx = sub2ind(size(p), m+geom(:, 4), m+geom(:, 3));

        % Perform the vectorized test and get a logical bit-vector
        fds(fp, :) = (p(x_idx) < p(y_idx))';  % -> BRIEF descriptor, p.
            1282

    end

end
```

12.8 Feature tracking: BRIEF - Hamming Distance

```
function [hd] = brief_hd(lv1, lv2)
% Compute the Hamming distance between two logic vectors.
% The hamming distance is the number of bits that are different.
%
% Janus Bo Andersen, Nov 2020

    hd = sum(xor(lv1, lv2));

end
```

12.9 Feature tracking: BRIEF - Matching

```
function [M1, M2, match_info] = brief_matches(X1, X2, im1_fds, im2_fds, ...
                                               T, stat_outl)
% This function does brute-force matching of feature descriptors across two
% views.
%  Input parameters:
%   X1: (N1 x 2) matrix of (x,y) coordinates of feature points in image 1
```

```
%   X1: (N2 x 2) matrix of (x,y) coordinates of feature points in image 2
%   im1_fds: (N1 x n_d) matrix of BRIEF feature descriptors for image 1
%   im2_fds: (N1 x n_d) matrix of BRIEF feature descriptors for image 2
%   T: Threshold, maximum bits that can be flipped to accept match
%   stat_outl: true/false, do statistical removal of outliers
%  Output:
%   M1: (m x 2) matrix of coordinates of matched points for image 1
%   M2: (m x 2) matrix of coordinates of matched points for image 2
%
% The function also implements Lowe's Ratio Test to determine if shortest
% distance is "good enough" to be a match.
%
% Janus Bo Andersen, November 2020

    N1 = length(X1); % number of feature points in im1
    N2 = length(X2); % number of feature points in im2

    % Build matrix of Hamming dist between j-th feature point in image1
    % and k-th feature point in image 2
    hds = zeros([N1 N2]);

    % Matches for points in im1 are stored in an N1-length vector,
    % % where zeros mean no match, and a number > 0 corresponds
    % to the row index in the im2 feature points.
    match = zeros([N1 1]);

    for j=1:N1

        % Compute Hamming distances to all points in im2
        for k=1:N2
            hds(j,k) = brief_hd(im1_fds(j,:), im2_fds(k,:));
        end

        % Find the two best fits
        fits = sort(hds(j,:));
        fits = fits(1:2);

        % Threshold and Lowe's Ratio Test "Good-enough" criteria
        % Use a relative threshold to determine if a point q in the
        % original image is fit well enough by a point, p1, in the second
        % image. The fit has to be significantly better than the second
        % best point, p2.
        % This idea is presented by Cyrill Stachniss in as criteria by
        % Lowe (1999).
        % Here, use d(q, p1)/d(q, p2) < 0.5, where d(., .) is the distance.
        if (fits(1) <= T) & (fits(1)/fits(2) < 0.5)
            match(j) = find(hds(j,:) == fits(1)); % Register fit
```

```
        end

    end

    % Set up coordinate correspondences.
    % M1 and M2 are Mx2 matrices of the M correspondence points with
    % coordinates [x1,y1] and [x2,y2] in im1 and im2 respectively.

    M = sum(match > 0);
    M1 = zeros([M 2]);
    M2 = zeros([M 2]);

    m = 1;
    for j = 1:N1
        if match(j) > 0
            % Matched points
            M1(m, :) = X1(j, :);
            M2(m, :) = X2(match(j), :);
            m = m + 1;
        end
    end

    drop_idx = 0;

    % If turned on, check the L2 norm for the vectors between coordinates,
    % and remove outliers
    if stat_outl == true
        % Rough homebrewed algorithm to remove erroneous matches:
        % Assume a normal distribution for the
        % direct translation vectors between image coordinates.
        % Errors are outliers vectors beyond 2.5 standard devs (99.9%).
        % Obviously, this is dangerous for large planar rotations,
        % but BRIEF is not too robust to these transformations anyway.

        L2 = (M1 - M2).^2;
        L2 = sum(L2, 2).^0.5;               % Sum axis for col1 and col2

        outlier_max_lim = mean(L2) + 2.5*std(L2);
        outlier_min_lim = mean(L2) - 2.5*std(L2);
        drop_idx = (L2 > outlier_max_lim) | (L2 < outlier_min_lim);

        % Drop outlier rows
        M1(drop_idx, :) = [];
        M2(drop_idx, :) = [];
    end

    % Generate info on the matching
```

```
101
102     info_str1 = ['Found ', ...
103         num2str(length(M1) - sum(drop_idx)), ' matches.'];
104     info_str2 = ['Statistics dropped ', ...
105         num2str(sum(drop_idx)), ' outlier(s).'];
106
107     match_info = {info_str1, info_str2};
108
109 end
```

12.10 Feature tracking: Show matches between two views

```
1  function [] = correspondence_show_matches(im1, im2, M1, M2, ...
2                                             match_info, display_type)
3  % This functions shows the correspondence matches between two views.
4  % Input parameters:
5  %   im1: Image 1
6  %   im2: Image 2
7  %   M1: (m x 2) coordinates of matched points in image 1
8  %   M2: (m x 2) coordinates of matched points in image 2
9  %   - The order of points in M1 and M2 must correspond.
10 %   match_info: cell array with info strings
11 %   display_type: 'blend', 'false', 'montage'
12 %
13 % Janus Bo Andersen, Nov 2020
14
15     figure;
16     ax = axes;
17
18     % Image options are: blend, false, montage
19     showMatchedFeatures(im1, im2, M1, M2, display_type, 'Parent', ax);
20     title(ax, 'Point correspondence (feature tracking)', 'FontSize', 18);
21
22     % Place the annotation here
23     dim = [.4 .6 .3 .3];
24     annotation('textbox', dim, 'String', match_info, ...
25         'FitBoxToText', 'on', 'BackgroundColor', 'white', 'FontSize', 18);
26
27     legend(ax, {'Matched feature point from view 1', ...
28         'Tracked f.p. location in view 2'}, 'FontSize', 18);
29
30     drawnow;
31
32 end
```

12.11 Reconstruction: Plot point cloud of world points

```
function [] = reconstruction_plot(world_points, R, t)
% Creates a point cloud plot from world points
% Places two cameras in the plot. One is placed as the origin of the
% world coordinate frame, the other is at transformation g(R,t).
%
% Janus Bo Andersen, Dec 2020

    % Create a point cloud object from the world points
    point_cloud = pointCloud(world_points);

    % Make the partially reconstructed 3D scene with camera positions
    % Camera icon size
    cameraSize = 0.15;
    figure

    % Place the first camera as a black icon at the world reference frame
    plotCamera('Size', cameraSize, 'Color', 'k', 'Label', '1', ...
        'Opacity', 0);
    hold on
    grid on

    % Place the translated and rotated camera as a blue icon
    plotCamera('Location', t, 'Orientation', R, 'Size', cameraSize, ...
        'Color', 'b', 'Label', '2', 'Opacity', 0);

    % Visualize the point cloud
    pcshow(point_cloud, 'VerticalAxis', 'y', 'VerticalAxisDir', 'down', ...
        'MarkerSize', 45);

    % White background and black axes
    % https://au.mathworks.com/matlabcentral/answers/
    %   452951-how-make-the-background-of-pcshow-white-instead-of-black
    % https://au.mathworks.com/matlabcentral/answers/
    %   7966-how-do-i-change-color-of-the-y-axes-made-by-plotyy
    set(gcf,'color','w');
    set(gca,'color','w');
    set(gca,'xcolor','k');
    set(gca,'ycolor','k');
    set(gca,'zcolor','k');

    % Rotate and set zoom
    % dtheta is the horizontal rotation and dphi is the vertical rotation.
    camorbit(0,-20);
    camzoom(1);
```

```
45
46      xlabel('x-axis'); ylabel('y-axis'); zlabel('z-axis')
47
48      sgtitle('Up-to-scale scene 3D reconstruction', 'Interpreter', ...
49          'Latex', 'FontSize', 18);
50
51      drawnow;
52
53  end
```