

LABORATION 1: Geometri och Visualisering

Kurs: Datorgeometri och visualisering (2D1428)

Kursledare: Stefan Carlsson
Assistent: Oscar Danielsson
Numerical Analysis and Computing Science
KTH, Stockholm, Sweden
{stefanc,osda02}@nada.kth.se

Preface

The aim of this laboratory is to understand the relationship between images, i.e. the projection of the 3D world, and the 3D world itself. In particular you will create a panoramic image from a collection of images taken by a purely rotating camera. You will also build a 3D model using two images from a calibrated camera. The 3D model will be stored as a textured VRML-model which can be rendered on a web-page using a Java-applet. The last exercise addresses the same task of creating 3D objects. However, in contrast to the previous exercise the calibration of the camera is not needed.

All three exercises will be carried out with real images taken from a “normal” digital camera. This means that you can use the code to create your own panoramas or 3D models from your own digital images.

All exercises will use MATLAB.

Some *good advises* in advance: Read *all* the instructions carefully before you start implementing. This laboratory assumes that you are familiar with the theoretical concepts of the chapters 1 – 4 in the lecture notes. For the assignment, it is important that the code is written in a readable style. You have to do the exercises in their order, i.e. 1 to 3, since the code will be reused. The first and last exercise is “fairly” short. You might expect to spend most time on the second exercise.

GOOD LUCK!

1 Introduction

1.1 Basic Datastructures

1.1.1 Image points, 3D points and cameras

All the 2D and 3D data will be stored in homogeneous coordinates. This means that an image point p is a column vector with 3 elements, i.e. $p \sim (x, y, 1)^T$. A 3D point P is a column vector with 4 elements, i.e. $P \sim (x, y, z, 1)^T$. Note, if 2D or 3D vectors are written in homogeneous coordinates then “ \sim ” means equality with an unknown scale, i.e. $p \sim \lambda p$. The projection matrix of a camera is denoted as M which represents a 3×4 matrix. This means that a 3D point P_i which is visible in camera M_j is projected onto image point p_i^j in the way:

$$p_i^j \sim M_j P_i. \quad (1)$$

In the case of m cameras and n points you obtain $m * n$ image points, in case all 3D points are visible in all m images. The cameras, 3D points and image points are stored in a compact way: $data = cameras * model$. $model$ is a matrix of size $4 \times n$, where $model(:, i)$ (MATLAB notation) represents the 3D points i (column vector). $cameras$ is a matrix of size $3 * m \times 4$, where $cameras(j * 3 - 2 : j * 3, :)$ is the camera j (3×4 matrix). The image point $data(j * 3 - 2 : j * 3, i)$ is the projection of 3D point i into camera j (column vector). Furthermore, $data(j * 3 - 2 : j * 3, :)$ represents the projection of all 3D points into camera j and $data(:, i)$ the projection of a 3D point i into all cameras. If a 3D point P_i is not visible in camera M_j then the image point p_i^j is NaN , i.e. $data(j * 3 - 2 : j * 3, i)$ is the column vector $(NaN, NaN, NaN)^T$.

1.1.2 Images

In this lab you will only work with grey-scale images. All the images are stored in a cell array, i.e. $images = cell(amount\ cameras, 1)$. The pixel (x, y) in image j can be accessed with $images\{i\}(x, y)$. The grey-value in an image ranges from 0 to 255.

1.2 Get started

1. Create your own course directory, e.g. *kurs_datorgeometri*
2. Copy the subdirectories *images/ data/ functions/ vrml/* from: */info/NADA-kurser/datorgeometri-gk/* to your new course directory
3. Go into the subdirectory *functions/* and start MATLAB
4. In MATLAB, choose *Set Path...* from the *File* menu
5. Click *Add with subfolders...* and select your new course directory
6. Click *Save*. If you get a dialog box asking if you want to create a new *pathdef.m* file, go ahead and create it in the */functions* subdirectory. It will be automatically loaded whenever you start MATLAB from that directory

1.3 Content of the Directories

All the necessary data for this lab is in the four subdirectories that you just downloaded. They contain the following data:

functions/ All the given and incomplete functions in MATLAB

images/ All grey-level images in JPG format. The files *.txt list all the respective images for loading them into a file.

data/ You find here some simulated data.

vrml/ Here is the web-page and Java-applet in order to show your final VRML-models.

2 Exercise 1: Generating Panoramic Images

In this exercise you will stick together three images. This means that two images are mapped by a linear mapping to a third image (the reference view).

Let us consider two views a and b of a rotating camera. You know from the lecture that the image points p^a and p^b , which are the projection of a 3D point P into the images a and b are related by a linear mapping H (3×3 matrix (see eqn. (33) in the lecture notes). This gives:

$$p^b = H_{a,b} p^a \quad (2)$$

where $H_{a,b}$ relates the views a and b . This linear mapping H is called a *homography*.

Let us specify one of the images, e.g. 3, as the reference image (*ref*). The task of this exercise is to determine the two homographies $H_{1,ref}$ and $H_{2,ref}$ for the two views 1, 2 to the reference view. If you have derived these homographies, you can map the images 1 and 2 to the reference view. The result is one bigger image, which is build from these three images.

2.1 Incomplete Functions

In this exercise you have to complete the following functions/scripts:

- script: generate_panorama
- function H = det_homographies(points1, points2)
- function norm_mat = get_normalization_matrices(data);

2.2 Obtain Correspondences between Images

The script which generates panoramic image is called: *generate_panorama*. Study this file to understand the main steps of generating a panormaic image. In order to determine homographies you have to get point correspondences. If you run the script *generate_panorama* three windows with images will open and you have to specify point correspondences between them.

The functionality for doing this is:

- left(and middle) button - click a point in each view (green)
- right button - finish a correspondence of points (red)
- u - (up) change to next image (active)
- d - (down) change to previous image (active)
- e - (end) stop the clicking process

You should choose view 3 as reference view and click at least four points between view 1 and 3 and view 2 and 3. *Tip:* You should stop the process (with pause or error) and check how the data looks like. The data should look like

x	x	x	x	NaN	NaN	NaN	NaN
x	x	x	x	NaN	NaN	NaN	NaN
1	1	1	1	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	x	x	x	x
NaN	NaN	NaN	NaN	...	x	x	x
NaN	NaN	NaN	NaN	1	1	1	1
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x
1	1	1	1	1	1	1	1

where x is an arbitrary entry.

You should get correspondences for set of images specified in *names_images_kthsmall.txt*. These images are not of high quality (size 160×120). Because of the high execution time of the process you should use these small images first. If your algorithm runs correctly you can use the images *names_images_kth.txt*. *Tip:* The function `click_multi_view(images, am_cams, dataset, para)` gives you the option to display a given *dataset*. With the parameter *para* you have the ability to show the number of the points as well.

2.3 Determine the Homographies

Using the point correspondences you should be able to calculate the homographies between a reference view *ref* and all other views *j*. These homographies will be stored in *homographies{j}*, i.e. $p^{ref} \sim homographies\{j\} p^j$. Note, the homography *homographies{ref}* has to be set as well. In order to find a specific homography you have to write the function: $H = det_homographies(points1, points2)$ as well. This function should take all the clicked points of two images. This means that *points1* and *points2* are matrices of size $3 \times n$, i.e. the image point p_i^1 in view 1 is *points1(:, i)*. The homography connects these point sets by: $points1 = H points2$. You can determine the homography *H* as explained on the top of page 19 in the lecture notes.

A numerical sound way to determine the solution of a homogeneous linear system: $W h = 0$ is to apply a singular value decomposition (SVD) on this matrix: $W = U \Sigma V^T$ (see section 4.1 in the lecture note). In MATLAB a singular value decomposition is achieved by $[USV] = svd(W)$. *S* is a diagonal-matrix with the singular values on the diagonal (sorted in decreasing order). The columns of *V* represent the singular vectors of *W*. If *W* is symmetric and positive definite, i.e. $W = A^T A$, it is $U = V$ and the square root of the singular values are the eigenvalues of *W*. The singular vectors and eigenvectors of *W* are the same. The solution *h* (with the constraint $\|h\|^2 = 1$) is represented by the nullspace of *W*. The nullspace of *W* is the linear combination of all singular vectors which have a singular value 0. Therefore, a linear system *W* has a unique solution *h* (with the constraint

$\|h\|^2 = 1$) if the nullspace of W is one-dimensional. In this case, the vector $V(:, \text{size}(W, 2))$ gives the non-trivial solution for a homogeneous linear system: $W h = 0$.

Tip: You can check if the data is *NaN* with the command *isnan*.

The average error in the pixels should be less than 5.0 for the image set: *names_images_kthsmall.txt*. After correct implementation you should get a panoramic image. The final panoramic image is stored in the directory *images/* with the name *panorama_image.jpg*.

Tip: You might have trouble to calculate the final panoramic image with the function: *generate_warped_image*, e.g. memory exhausted. In this case you can use the alternative but slower function: *generate_warped_image_alt* (see line 58 in script: *generate_panorama*).

2.4 Normalizing the Data

The process of determining the homographies can be further improved by preconditioning the data. In the previous version, the x- and y-values of the image points have been in the range of [0,640] and [0,480]. This means that elements in the linear system of equation might have very different magnitudes. Since this is a drawback for numerical calculations, the data is normalized in the way that the x- and y-values are in average 1. This can be achieved by moving the centroid of the selected image points in each image to the origin of the image, i.e. $(0, 0, 1)^T$. The centroid of the selected image points in image j can be determined as:

$$\text{centroid} = 1/n \sum_{i=1}^n p_i^j \quad (3)$$

Furthermore, the average distance of the selected image points to the centroid should be $\sqrt{2}$. This distance can be determined for an image j as:

$$\text{distance} = 1/n \sum_{i=1}^n \|p_i - \text{centroid}\|_2, \quad (4)$$

where $\|\cdot\|_2$ represents the Euclidean distance, i.e. $\|a\|_2 = \sqrt{a_x^2 + a_y^2}$. We would like to determine for each image j a matrix N_j which normalizes the image point p_i^j to $p_{i(norm)}^j$. This can be achieved with:

$$N_j = \begin{pmatrix} \frac{\sqrt{2}}{\text{distance}} & 0 & -\frac{\sqrt{2} \text{centroid}_x}{\text{distance}} \\ 0 & \frac{\sqrt{2}}{\text{distance}} & -\frac{\sqrt{2} \text{centroid}_y}{\text{distance}} \\ 0 & 0 & 1 \end{pmatrix} \quad \text{where} \quad p_{i(norm)}^j = N_j p_i^j \quad (5)$$

The normalized points in an image j , i.e. p_{norm}^j , should have centroid $(0, 0, 1)^T$ and average distance $\sqrt{2}$.

Rewrite the function: *function norm_mat = get_normalization_matrices(data)* which takes the data (including NaN) and gives back a matrix *norm_mat* of the size $3 * m \times 3$. The matrix $N = \text{norm_mat}(3 * j - 2 : 3 * j, :)$ should represent the normalization matrix for image j . So the normalized points for camera j can be determined by:

$$\begin{aligned} & \text{data_norm}(j * 3 - 2 : j * 3, :) = \\ & \text{norm_mat}(j * 3 - 2 : j * 3, :) * \text{data}(j * 3 - 2 : j * 3, :) \end{aligned} \quad (6)$$

(see line 35 in the script: *generate_panorama*). Be aware that you don't use the *NaN* data for the normalization.

Tip: You can check if the normalization is correct in the following way. Apply `get_normalization_matrices` on the normalized image data `data_norm`. The matrices `norm_mat(j*3-2:j*3,:)` should be the identity matrix, i.e. `eye(3)`. A deviation of less than e^{-10} is acceptable.

Calculate now the homography H_{norm} for the normalized points, i.e.

$$p_{norm}^{ref} \sim H_{norm} p_{norm}^j . \quad (7)$$

Be aware, the homography you have to store in `homographies{j}` is still the one between the non-normalized points. If we insert $p_{norm}^j = N_j p^j$ into eqn. (7) we obtain

$$p^{ref} \sim N_{ref}^{-1} H_{norm} N_j p^j . \quad (8)$$

This means that the homographies H and H_{norm} are related by

$$H = N_{ref}^{-1} H_{norm} N_j . \quad (9)$$

2.5 Use Larger Images (optional)

If your algorithm runs correctly you can produce a nicer panoramic image from the image set: `names_images_kth.txt`. In order to do this you have to load these images (see line 16 in the script: `generate_panorama`). These images are of size 640×480 . You can click the corresponding points once more or use the data from `names_images_kthsmall.txt`. In the latter case you have to multiply the x - and y -values of the data by 4. Since your algorithm might take a while to produce a nice result, this part of the exercise is optional. *Tip:* Use the function `generate_warped_image_alt` instead of `generate_warped_image` if necessary.

2.6 Questions:

- What is the minimum number of point-correspondances needed to determine the homography between 2 images?
- How much improvement did the normalization yield in terms of average and maximum error?
- Given that you have the minimum number of point-correspondances to determine a homography, the homography might still not be uniquely determined. How can this happen and how can you check if it has happened?
- Look at the function `generate_warped_image`. Why does the function use the inverse of the homographies (line 79)?
- Can you extend the method used in this exercise to generate a panorama for images from a camera rotating about 360° ?

3 Exercise 2: Calibrated Reconstruction

In this exercise you will reconstruct a 3D object from a pair of images (also called stereo reconstruction). This is achieved by assuming that the internal calibration of the camera is known. The 3D model and external camera parameters are unknown. The reconstruction will be done on the basis of point correspondences between the images. The main steps of this procedure are explained in section 2.7 in the lecture notes. The final step of this procedure is to create a textured 3D model from the reconstructed point cloud.

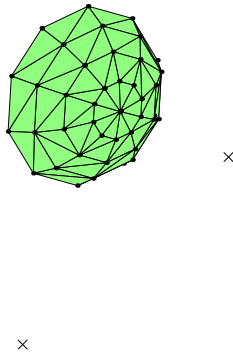


Figure 1: Visualization of the reconstruction with the synthetic data: *sphere_data.mat*. The crosses indicate the two camera centers, the half-sphere is represented by a triangulation of 55 points.

3.1 Incomplete Functions

In this exercise you have to complete the following functions/scripts:

- script: `calibrated_reconstruction`
- function `E = det_E_matrix(points1, points2, K1, K2)`
- function `[cams, cam_centers] = det_stereo_cameras(E, K1, K2, data)`
- function `model = det_model(cam, data)`
- function `[error_average, error_max] = check_reprojection_error(data, cam, model)`
- function `triang = get_delaunay_triang(data, num_ref)`
- function `save_vrml(data, model, triang, filename, name_image, image_size, para1, para2)`

3.2 Main Function & Data & Camera calibration

The main script is called: *calibrated_reconstruction*. This script includes the steps for a calibrated reconstruction. Before your algorithm works correctly you should use the synthetic (simulated) data: *sphere_data.mat*. If you run this script it loads the data (*load sphere_data.mat*) in the beginning. You can investigate (with *whos*) that it loads the data: *cam_sim* (6×4), *model_sim* (4×55) and *data_sim* (6×55). The synthetic data is a half-sphere of 55 points seen by two cameras (see figure 1). This data is error free, i.e $data_sim = cam_sim * model_sim$. When your code works correctly with the synthetic data you should reconstruct a teapot from two real images: *names_images_teapot.txt*.

The internal calibrations of the cameras are given. For the synthetic data, the calibration of the first and the second camera is:

$$K1 = \begin{pmatrix} 5 & 0 & 3 \\ 0 & 4 & 2 \\ 0 & 0 & 1 \end{pmatrix}, \quad K2 = \begin{pmatrix} 7 & 0 & 1 \\ 0 & 10 & 5 \\ 0 & 0 & 1 \end{pmatrix}. \quad (10)$$

In case of real images, the two calibration matrices are as follows:

$$K1 = K2 = \begin{pmatrix} 2250 & 0 & 400 \\ 0 & 2250 & 300 \\ 0 & 0 & 1 \end{pmatrix}. \quad (11)$$

Note, the principle point $(400, 300, 1)^T$ of the camera correspond to the middle of the image, which has size 800×600 . The focal length is 2250 pixel.

3.3 Determine the Essential matrix

The first step of a calibrated reconstruction is to determine the Essential matrix from two sets of points between the two views. The essential matrix E connects two corresponding points p_{cam}^a, p_{cam}^b (which have the same point in 3D) in the images a and b by the epipolar constraint (see equation (89) and middle of page 29 in the lecture notes):

$$(x^b \ y^b \ 1) E \begin{pmatrix} x^a \\ y^a \\ 1 \end{pmatrix} = 0 \quad \text{with} \quad (12)$$

$$p_{cam}^a = \begin{pmatrix} x^a \\ y^a \\ 1 \end{pmatrix}, \quad p_{cam}^b = \begin{pmatrix} x^b \\ y^b \\ 1 \end{pmatrix}, \quad E = \begin{pmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{pmatrix}.$$

This can be written as:

$$e_{11}x^bx^a + e_{12}x^by^a + e_{13}x^b + e_{21}y^bx^a + e_{22}y^by^a + e_{23}y^b + e_{31}x^a + e_{32}y^a + e_{33} = 0. \quad (13)$$

The points p_{cam}^a and p_{cam}^b are points in a normalized camera coordinate system, which has the internal calibration matrix $I_{3 \times 3}$. This means that the original image points p^a, p^b and the normalized points p_{cam}^a, p_{cam}^b are related by:

$$p_{cam}^a = K_a^{-1} p^a, \quad p_{cam}^b = K_b^{-1} p^b, \quad (14)$$

where K_a, K_b are the internal calibration matrices of camera a and b respectively.

We obtain E by an algorithm which is called the “normalized 8-point algorithm”. Each pair of corresponding points p_{cam}^a, p_{cam}^b give one linear constraint (13). The nine, unknown elements of the E-matrix can be put into a vector:

$$h = (e_{11}, e_{12}, e_{13}, e_{21}, e_{22}, e_{23}, e_{31}, e_{32}, e_{33})^T. \quad (15)$$

By using the linear constraint in (13), we obtain a linear system $W h = 0$, which can be solved as in the previous exercise by using SVD. Since the E matrix has only 8 degrees of freedom (but 9 entries), 8 corresponding points are sufficient.

Complete the function `det_E_matrix(points1, points2, K1, K2)` which determines the Essential matrix. $K1$ and $K2$ represent the internal camera matrices K_a and K_b . In order to evaluate if the E matrix is correct you should check the epipolar constraint (12) for each point. For the synthetic data you should get: $p_{cam}^b T E p_{cam}^a < e^{-10}$.

As you probably discovered in the previous exercise, the normalization of the image data is quite essential for the numerical stability of the process. You can use the function of the previous exercise: `norm_mat = get_normalization_matrices(data)` to normalize the points p_{cam}^a, p_{cam}^b in the images a and b . You get the E matrix from the matrix E_{norm} , which uses the normalized points p_{cam_norm} , as:

$$E = N_b^T E_{norm} N_a \quad \text{where} \quad p_{cam_norm}^j = N_j p_{cam}^j$$

$$\text{and} \quad p_{cam_norm}^b T E_{norm} p_{cam_norm}^a = 0. \quad (16)$$

Note, the function `det_E_matrix(points1, points2, K1, K2)` should always return the matrix E and not E_{norm} .

A property of the E-matrix is that two singular values are equal (and larger than 0) and one singular value is exactly 0. If the image data is corrupted by noise this property might not be fulfilled. However, we would like to determine an E matrix which fulfills this property exactly. If E has the singular value decomposition $E = USV^T$, then the correct Essential matrix $E_{correct}$ can be defined as:

$$E_{correct} = U \begin{pmatrix} (S(1,1) + S(2,2))/2 & 0 & 0 \\ 0 & (S(1,1) + S(2,2))/2 & 0 \\ 0 & 0 & 0 \end{pmatrix} V^T. \quad (17)$$

Include this property and the normalization of the points into your code. If you run the script: `calibrated_reconstruction` you should obtain an E-matrix which fulfills the epipolar constraint (12) and has the above properties.

3.4 Obtain the Cameras (Motion)

The next step is to determine the two camera matrices from the Essential matrix. The two cameras M_a, M_b , which are each 3×4 matrices, are defined as:

$$M_a = K_a(I | \mathbf{0}), \quad M_b = K_b R(I | \mathbf{t}). \quad (18)$$

Note, M_a and M_b are only unique up to scale, i.e. $M_a = \lambda M_a$. The remaining unknowns in equation (18) are the rotation R and translation t . The vector t can be seen as well as the camera center of the second camera. As you know from the lecture these unknowns can be determined from the E-matrix.

We will determine the translation t first. Let us assume that E has the SVD: $E = USV^T$. In section 2.5.1 of the lecture notes, we saw that t is the null vector of E , i.e. $E t = 0$. Therefore, the translation t is given by $t = V(:, 3)$. Since the scale of t can not be determined, you can normalize t to one: $\|t\|^2 = 1$. You can now store t as the second camera center, i.e. `cam_centers(:, 2) = t`.

Let us now compute the rotation R . We will present here an alternative way as in section 2.5.2 in the lecture notes. The rotation matrix R has 2 possible solutions:

$$R_1 = U W V^T; \quad R_2 = U W^T V^T \quad \text{with} \quad W = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \quad (19)$$

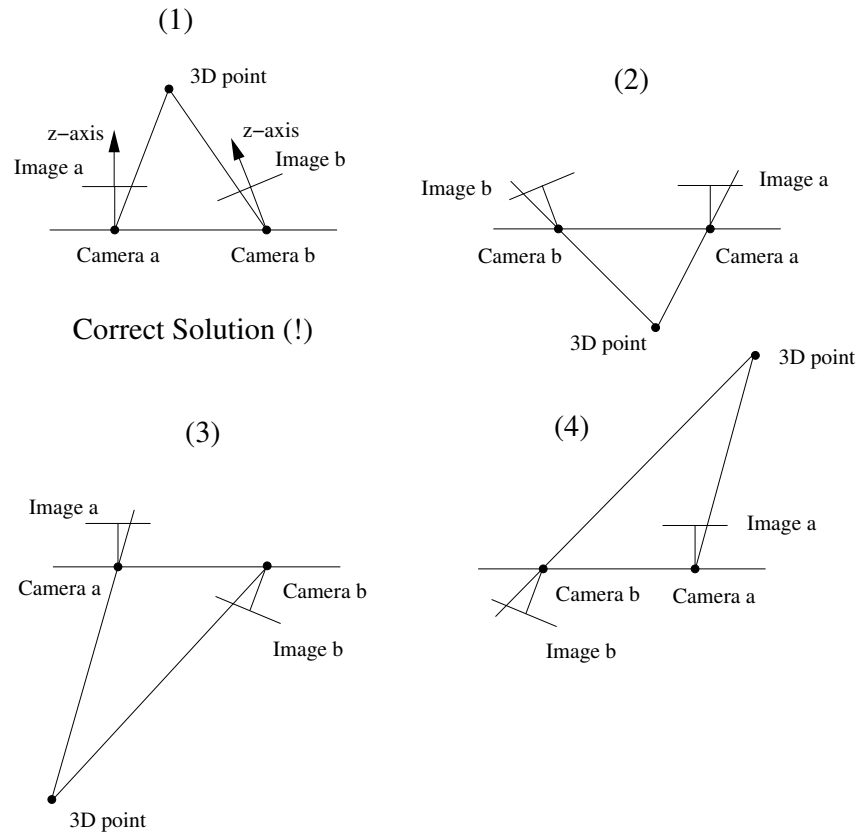


Figure 2: The 4 different solutions for the cameras M_a and M_b which reconstruct one 3D point. However, only one solution satisfies the property that the reconstructed 3D point is in front of both cameras.

The matrix W is a rotation matrix. Note, the ambiguity in R is briefly mentioned in the lecture notes. A rotation matrix must have determinant 1 (determinant -1 means that the coordinate system is rotated and some coordinate axis are swapped). Therefore, you have to multiply R_1 or R_2 with -1 if the respective determinant is -1 . The second camera matrix M_b has 4 different solutions:

$$M_b = K_b R_1 (I | \mathbf{t}) \quad \text{or} \quad K_b R_1 (I | -\mathbf{t}) \quad \text{or} \quad K_b R_2 (I | \mathbf{t}) \quad \text{or} \quad K_b R_2 (I | -\mathbf{t})$$

The 4 different solutions are displayed in figure 2. The camera M_a is in all 4 cases the same. The difference between the 1 and 2 case is that the translation of the camera M_b is in the opposite direction, i.e. t and $-t$. The same applies to the third and fourth case. The difference between the first and third case, i.e. R_1 and R_2 , is that the camera M_b is rotated about 180° around the axis which connects the two camera centers. The figure shows that only one solution fulfills an essential property that a reconstructed 3D point is in front of both cameras. In the figure this is case 1. In order to check this property for all 4 solutions of pairs of cameras, you have to reconstruct one point for all 4 possible pairs of cameras. How to do this is described in section (3.5).

You have to write the function $model = det_model(cam, data)$ first (see section (3.5)). After that you have to complete the function: $[cams, cam_centers] = det_stereo_cameras(E, K1, K1, data)$

which gives back the two cameras $cams(1 : 3, :)$ and $cams(4 : 6, :)$. The two camera centers will be stored in $cam_centers(1 : 4, 1)$ and $cam_centers(1 : 4, 2)$. $K1$ and $K2$ represent the internal camera matrices K_a and K_b . With the function det_model you are able to reconstruct a 3D point, e.g. $data(:,1)$, for all 4 possible pairs of cameras. Determine which of the 4 pairs of cameras is the right one, by checking the property that the reconstructed 3D point is in front of both cameras. Think of a way to decide if a point is in front of a certain camera or not. *Tip*: Use the z-axis of the cameras, which is the last column of the rotation matrix, i.e. $R(:, 3)$.

Verify in the end that the following equation holds (see lecture notes eqn. (90)):

$$E = R \begin{pmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{pmatrix} \quad \text{with } t = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}. \quad (20)$$

Remember that E is only determined up to some scale factor.

3.5 Obtain the 3D Points (Structure)

If the cameras are known, the 3D structure, i.e. 3D points, can be determined as described in section 2.1 in the lecture notes. This process is called “intersection” or “triangulation”. We have n unknown 3D points, which means that the image data is of size $6 \times n$. Each unknown 3D point $(X, Y, Z, 1)^T$ has 3 degrees of freedom (but 4 entries). Each camera provides two linear equations of the form (67) in the lecture notes. Therefore 4 linear equations (from the two cameras) give a linear system (see middle of page 23):

$$W \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = 0, \quad (21)$$

where W is of size 4×4 . This homogeneous system of equations can again be solved by using SVD.

Implemented the function $model = det_model(cam, data)$ which returns all 3D points, i.e. the $model$ of size $4 \times n$.

3.6 Check the Re-projection Error

If you have reconstructed the 3D points and the cameras you can check if the constraint: $data = cams * model$ is fulfilled. You should complete the function: $[error_average, error_max] = check_reprojection_error(data, cam, model)$. The average and maximum error represents the average and maximum deviation between a re-projected point $((cams * model)(3 * j - 2 : 3 * j, i))$ and the point in the image $(data(3 * j - 2 : 3 * j, i))$.

Write the function $check_reprojection_error$. If you run the script $calibrated_reconstruction$ with the synthetic data, you should get $error_average = error_max = 0.0$. The function $norm_points_to_one$, which is called before $check_reprojection_error$, makes sure that the 4th coordinate of the model and the camera centers is 1.

3.7 Create and Visualize the 3D Model with VRML

The final steps of the procedure is to triangulate the model and to create a textured VRML-model.

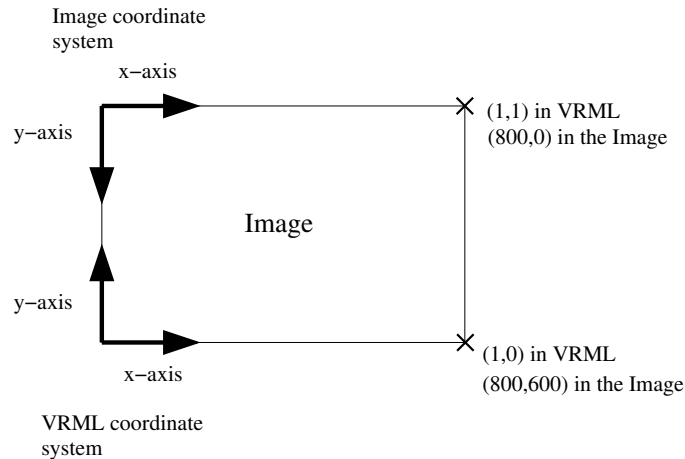


Figure 3: The coordinate system VRML uses and the original coordinate system of the image, which is of size 800×600 .

Let us consider the triangulation process first. MATLAB offers a command: `triang = delaunay(points_x, points_y)`. You can read more about it with `help delaunay`. It basically triangulates a set of n points, where `points_x` and `points_y` represent the x and y coordinate of a point. `triang` is of size $k \times 3$, where k is the number of triangles. Each entry in `triang` is an index (between 1 and n) to the corresponding 2D point. You should complete the function: `triang = get_delaunay_triang(data, num_ref)`. The parameter `num_ref` determines which image (that is, which set of image coordinates) to use for triangulation.

If you run the script: `calibrated_reconstruction` you should get an extra window which shows the triangulated 3D point cloud in green. The two red crosses represent the camera center. With the synthetic data `sphere_data.mat` you should get the result as in figure 1. This visualization was created with the script: `visualize(model, cam_centers, triang, version)`. If you choose the parameter `version` in different ways you can visualize different things.

Finally, we would like to create a textured VRML-model. Before doing this you have to copy (all) the files: `examine.jar`, `show_model.html`, `teapot1.jpg`, `toyhouse1.jpg` from `$DGGKHOME/vrml/` into your `vrml/` directory. The file `examine.jar` is a Java-applet. If you have created a VRML-file, you have to store it in your directory `vrml/` with the name `vrml_model.wrl`. A VRML-file has always the ending `.wrl`. If you load the page `show_model.html` with any browser which supports Java 1.1, e.g. `netscape` or `explorer` (later versions), you will see the textured VRML-model (you can call: `netscape show_model.html`). The Java-applet allows you to spin the object around and to zoom in and out. Alternatively, you can display the Java-applet alone with the command: `appletviewer show_model.html`.

You will create the VRML-model from real images of a teapot placed on a box. Therefore you have to change the flag: `flag_synthetic_data` in the beginning of the script: `calibrated_reconstruction` to 0. You will now load the real images with the name: `names_images_teapot.txt`. Furthermore, the correct internal camera parameters (see equation (11)) are chosen. Additionally you have to click some (at least 8) corresponding points between the two views. For a nice result you

should click at least 20 points which vary in depth as well, e.g. the corners of the cube. Be aware that not to put all points on a single plane since this would not yield a unique E -matrix. In order to create a VRML-model, you have to complete the function: `save_vrml(data, model, triang, filename, name_image, image_size, para1, para2)`.

You have learned about VRML in the course. A web page with the complete specification, tutorials and examples can be found at: <http://www.vrml.org/VRML2.0/>. Note, VRML 2.0 and VRML 97 is the same. The first VRML version was called VRML 1.0.

The parts which you have to fill in into the VRML-file could look in the end like this:

```

        coordIndex [
            0 5 19 -1,
            19 0 23 -1,
            .
            .
            .
            4 3 58 -1 ]
    texCoord TextureCoordinate { point [
        0.039796 0.882899,
        0.869289 0.889038,
        .
        .
        .
        0.440717 0.901318 ] }

```

The field `coordIndex` shows the list of all triangles. An entry of this list is an index to the 3D point specified in the field `coord`. Note, *VRML* indexes the 3D points between 0 and $n - 1$. The `-1` indicates the end of a patch (triangle). The field `texCoord` specifies the 2D point in the reference image (x-, y-coordinate). The i th entry in this list defines the i th 3D point. Therefore, the number of 2D positions has to be equal to the number of 3D points. Note, the coordinate system of the image has to be adapted to the coordinate system VRML uses. Figure 3 shows the two coordinate systems, where the image is of size 800×600 . The reference image is specified in the field: `texture`, e.g. `texture ImageTexture url "teapot1.jpg"`.

Complete the function `save_vrml` (most of the code is already given). Look at the VRML-model with e.g. `netscape show_model.html`.

3.8 Questions

- What is the rank of the essential matrix E and $E_{correct}$ if (a) the selected image points correspond perfectly and (b) if the selected image points do not correspond perfectly, i.e. they do not represent a unique 3D point in space?
- It might happen that all the 3D points lie on a certain surface, called a *critical configuration*, so that the Essential matrix is not unique. How could you detect such a critical configuration?
- Assume that you have m cameras looking at a scene which consists of 3D points. The camera matrices are known and the 3D points unknown. Some of the 3D points are visible in all m views, other points just in a subset of views. How would you obtain the unknown structure in an optimal way?

- The final VRML-model might not look perfect, e.g. the object looks “flat”. What would you suggest to get a nice reconstruction with texture of the whole(!) teapot ?

4 Exercise 3: Uncalibrated Reconstruction

In the previous exercise you have seen how to obtain a metric reconstruction from two calibrated cameras. However, in practice the calibration of a camera might not be known. In this case it is still possible to reconstruct an object. In this exercise you will reconstruct an object from two views of an uncalibrated camera. The reconstruction is called a *projective reconstruction*. Since a projective reconstruction does not look very “nice”, you have to rectify it. This rectification process transforms the projective reconstruction into a *metric reconstruction*. In this exercise the rectification process uses metric properties about the world. These metric properties are distances between 3D points. Alternatively, the rectification process could exploit the fact that the internal calibration of the camera is constant for several views. This process is then called *self-calibration* and will not be considered here.

4.1 Incomplete Functions

In this exercise you have to complete the following functions/scripts:

- script: `uncalibrated_reconstruction`
- function `F = det_F_matrix(points1, points2)`
- function `[cams, cam_centers] = det_uncalib_stereo_cameras(F)`
- function `H = det_rectification_matrix(points1, points2)`

Copy these functions from: `$DGGKHOME/functions/` into your subdirectory `functions/`.

4.2 Main Function & Data

The main script for uncalibrated reconstruction is called: *uncalibrated_reconstruction*. As in the previous exercise you should test your algorithms first with the synthetic data: *sphere_data.mat*, i.e. choose `flag_synthetic_data = 1`. When everything runs fine you will reconstruct a toyhouse from the real images: *names_images_toyhouse.txt*.

4.3 Determine the Fundamental matrix

The Fundamental matrix plays the same role for uncalibrated cameras as the Essential matrix does in the calibrated case. Two image points p^a, p^b are related with the F-matrix (size 3×3) as (see lecture notes eqn. (119)):

$$(x^b \ y^b \ 1) F \begin{pmatrix} x^a \\ y^a \\ 1 \end{pmatrix} = 0 \quad \text{with } p^a = \begin{pmatrix} x^a \\ y^a \\ 1 \end{pmatrix}, \quad p^b = \begin{pmatrix} x^b \\ y^b \\ 1 \end{pmatrix}. \quad (22)$$

Note, the image points p^a, p^b are *not* normalized with respect to the camera coordinate system, i.e. p_{cam}^a, p_{cam}^b .

Complete the function `det_F_matrix(points1, points2)` which determines the Fundamental matrix (you can use most of the code from `det_E_matrix(points1, points2, K1, K2)`). The computation of the F -matrix should be based on normalized image points: $p_{norm}^j = N_j p^j$.

The F -matrix has rank 2. Adapt the F -matrix to $F_{correct}$ so that this property is fulfilled.

4.4 Obtain the Cameras & Model

The determination of the two cameras M_a, M_b is in the uncalibrated case much easier compared to the calibrated one. The two cameras may be defined as (see lecture notes eqn. (135)):

$$M_a = (I \mid \mathbf{0}), \quad M_b = (SF \mid \mathbf{h}). \quad (23)$$

The vector h is the epipole in the second camera, i.e. $F^T h = 0$. The matrix S is an arbitrary 3×3 antisymmetric matrix. You may choose:

$$S = [v]_{\times} \quad \text{with } v = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}. \quad (24)$$

Write the function: `[cams, cam_centers] = det_uncalib_stereo_cameras(F)` which gives back the two cameras `cams(1 : 3, :)` and `cams(4 : 6, :)` and the two camera centers `cam_centers(1 : 4, 1)` and `cam_centers(1 : 4, 2)`.

The camera centers t_a and t_b (4×1 vectors) of cameras M_a and M_b are projected onto the center of projection, i.e. $(0, 0, 0)^T$. This can be written as

$$M_a t_a = (0, 0, 0)^T \quad \text{and} \quad M_b t_b = (0, 0, 0)^T. \quad (25)$$

This means that the camera centers t_a and t_b can be obtained from the SVD of the matrix M_a and M_b respectively.

The 3D model is created as in the previous exercise by triangulation. You implemented the necessary function `model = det_model(cam, data)` already.

If you run the script `uncalibrated_reconstruction` with the synthetic data, i.e. `flag_synthetic_data = 1`, you should obtain `error_average = error_max = 0.0`.

4.5 Rectify the Structure & Motion

The reconstruction of 3D points and cameras is correct in the sense that a 3D point P_i is projected via camera M_j onto the image point p_i^j , i.e. $p_i^j \sim M_j P_i$. However, we can apply an arbitrary linear transformation H (4×4 matrix) to both the 3D points and the cameras without changing this relationship. The new 3D points P'_i and cameras M'_j are defined as $P'_i = H P_i$ and $M'_j = M_j H^{-1}$ since it is:

$$p_i^j \sim M'_j P'_i = M_j H^{-1} H P_i = M_j P_i. \quad (26)$$

Such a reconstruction is called *projective reconstruction* since H is a general projective transformation. A projective reconstruction does not fulfill metric properties. Some metric properties are: the angle between two lines is correct, the ratio of two lengths is correct, parallel lines are parallel (intersect at infinity). Since we are familiar with these properties, i.e. we live in a Euclidean space, we would like to find a transformation H which rectifies the structure P to P' . The new structure P' should have these metric properties. In order to determine H , we need some metric information

about the scene or some information about the camera. The process which exploits only information about the camera is called *self-calibration*.

One simple way to determine H is to specify the 3D coordinates of 5 (or more) 3D points, i.e. exploit metric information about the scene. A point $P = (x, y, z, w)^T$ and the rectified point $P' = (x', y', z', 1)^T$ are related with H as:

$$P' \sim H P \quad \text{which is} \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} \sim \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{14} \\ h_{21} & h_{22} & h_{23} & h_{24} \\ h_{31} & h_{32} & h_{33} & h_{34} \\ h_{41} & h_{42} & h_{43} & h_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}. \quad (27)$$

Note, in a projective reconstruction the last coordinate w of a point P might be 0. Therefore, we can not normalize w to 1. You have seen such a mapping H already in the first exercise (see eqn. (2)). In that case H connected two 2D homogeneous points. You can think of H in eqn. (27) as a 4D homography which connects two 3D homogeneous points. The matrix H in eqn. (27) has 16 entries and 15 degrees of freedom (H is unique up to scale). Two corresponding points P, P' give three linear equations of the form:

$$\begin{aligned} x' &= \frac{h_{11}x + h_{12}y + h_{13}z + h_{14}w}{h_{41}x + h_{42}y + h_{43}z + h_{44}w} \\ y' &= \frac{h_{21}x + h_{22}y + h_{23}z + h_{24}w}{h_{41}x + h_{42}y + h_{43}z + h_{44}w} \\ z' &= \frac{h_{31}x + h_{32}y + h_{33}z + h_{34}w}{h_{41}x + h_{42}y + h_{43}z + h_{44}w}. \end{aligned} \quad (28)$$

This can be written as:

$$\begin{aligned} h_{11}x + h_{12}y + h_{13}z + h_{14}w - h_{41}xx' - h_{42}yx' - h_{43}zx' - h_{44}wx' &= 0 \\ h_{21}x + h_{22}y + h_{23}z + h_{24}w - h_{41}xy' - h_{42}yy' - h_{43}zy' - h_{44}wy' &= 0 \\ h_{31}x + h_{32}y + h_{33}z + h_{34}w - h_{41}xz' - h_{42}yz' - h_{43}zz' - h_{44}wz' &= 0. \end{aligned} \quad (29)$$

Therefore, 5 corresponding 3D points give 15 equations and define the matrix H uniquely up to scale. These 15 equations can be put into one linear system W of equations:

$$\begin{aligned} W h &= 0, \quad \text{where} \\ h &= (h_{11}, h_{12}, h_{13}, h_{14}, h_{21}, h_{22}, h_{23}, h_{24}, h_{31}, h_{32}, h_{33}, h_{34}, h_{41}, h_{42}, h_{43}, h_{44})^T. \end{aligned} \quad (30)$$

This linear system can be solved by applying a SVD on the matrix W . Alternatively, you can calculate H in a similar way as in the first exercise (see section 1.8 in the lecture notes).

Write the function $H = \text{det_rectification_matrix}(\text{points1}, \text{points2})$ which returns the mapping H . The mapping H should satisfy: $\text{points2} \sim H \text{points1}$. The data points1 and points2 is of size $4 \times n$ (n is in this case 5). You do *not* have to normalize the 3D points.

Furthermore, you have to determine the rectified *model*, *cameras* and *camera centers* with the use of H . Complete the part of the script *uncalibrated_reconstruction* which starts with the comment: `% rectify the cameras and the model`. Denote the rectified data as: *model_rec*, *cams_rec*, *cam_centers_rec*. *Tip*: The camera centers are rectified in the same way as a 3D point.

If you run the script *uncalibrated_reconstruction* with the synthetic data, i.e. *flag_synthetic_data* = 1, you should obtain the half-sphere and the two camera centers as in the previous exercise (see figure 1). The first and second camera center should be: $t_a = (0, 0, 0, 1)^T$, $t_b = (0, 3, 0, 1)^T$.

4.6 Create and Visualize the 3D Model with VRML

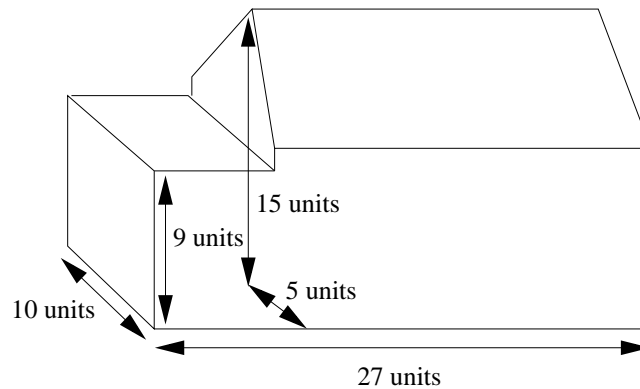


Figure 4: The dimensions of the toyhouse.

You wrote all functions for creating a textured VRML-model already for the previous exercise. Set the flag: *flag_synthetic_data* in the beginning of the script: *uncalibrated_reconstruction* to 0. You load now the two images: *names_images_toyhouse.txt*. Click some (at least 8) corresponding points between the two views. You have to specify *model_synthetic_points* and *model_synthetic* for the case that *flag_synthetic_data* = 0. In order to do this, the following lengths of the building are given (see figure 4). Make sure that 3D points you choose for the rectification process define the rectification matrix H uniquely. *Tip*: The function *click_multi_view(images, am_cams, data, 2)* shows you the numbers of all image points. When you create the VRML-model, use the first image as reference view, i.e. *image_ref* = 1. You can look at the VRML-model with either *netscape show_model.html* or *appletviewer show_model.html*. Make sure that you copied the image *toyhouse1.jpg* into your *vrml/* directory.

4.7 Questions

- How do you determine the rectification matrix H in an optimal way if you have more than 5 3D points given?
- What is the condition that two set of points define a unique rectification matrix H ?
- Assume that you do not rectify the model, i.e. *rec_matrix* = *eye(4)*. What kind of error can occur if you normalize the model with the function *norm_points_to_one*?
- How many degrees of freedom has the rectification matrix H ? And how many degrees of freedom of H do you have to fix in order to obtain a 3D model in the Euclidean space? Note, a model in the Euclidean space can only be rotated and translated.
- Assume that you have 3 cameras and $n(\geq 8)$ 3D points which are visible in all views. The cameras and the 3D points are unknown. How do you obtain 3 correct camera matrices? If you have achieved this, how would you obtain all the 3D points?

- Assume that you have extended this method to multiple (> 2) views. What are the main differences between this method and the factorization method for multiple parallel projection cameras described in section 4 of the lecture notes?

**Assignment for the Course:
Datorgeometri i bildanalys och visualisering
(2D1424)**

Students Namn:

Personnummer:

Exercise 1:

Datum

Kursledare/kursassistent

Exercise 2:

Datum

Kursledare/kursassistent

Exercise 3:

Datum

Kursledare/kursassistent