Image Analysis And Computer Vision

Laboratory

Christian Rossi

Academic Year 2024-2025

Abstract

The course begins with an introduction to camera sensors, including their transduction, optics, geometry, and distortion characteristics. It then covers the basics of projective geometry, focusing on modeling fundamental primitives such as points, lines, planes, conic sections, and quadric surfaces, as well as understanding projective spatial transformations and projections.

The course continues with an exploration of camera geometry and single-view analysis, addressing topics like calibration, image rectification, and the localization of 3D models. This is followed by a study of multi-view analysis techniques, which includes 3D shape reconstruction, self-calibration, and 3D scene understanding.

Students will also learn about linear filters and convolutions, including space-invariant filters, the Fourier Transform, and issues related to sampling and aliasing. Nonlinear filters are discussed as well, with a focus on image morphology and operations such as dilation, erosion, opening, and closing, as well as median filters.

The course further explores edge detection and feature detection techniques, along with feature matching and tracking in image sequences. It addresses methods for inferring parametric models from noisy data and outliers, including contour segmentation, clustering, the Hough Transform, and RANSAC (random sample consensus).

Finally, the course applies these concepts to practical problems such as object tracking, recognition, and classification.

Contents

CHAPTER 1

Introduction

1.1 MATLAB basics

To begin working in MATLAB, it's helpful to clear the workspace, close all figures, and clear the command window. This ensures a clean environment for running code.

```
close all % Close all figures
clear % Clear all variables from the workspace
clc % Clear the command window
```

1.1.1 Variables

In MATLAB, all variables are stored in matrix form, regardless of their type or dimensionality. Scalars are represented as 1×1 matrices. Variables are created through assignments, and their size reflects the dimensions of the matrix.

```
v = 12; % Define a scalar
c = 'c'; % Define a character
size(v) % Check the size of a variable
whos v % Display detailed information about a variable
```

The most commonly used variable types in MATLAB include:

- `double` (double-precision floating point),
- `uint8` (8-bit unsigned integer, ranges from 0 to 255),
- `logical` (boolean type, for true/false values).

1.1.2 Arrays and matrices

Arrays and matrices in MATLAB can be defined as either row or column vectors. MATLAB indices start at 1 (not 0 as in many other programming languages).

```
r = [1 2 3 4]; % Row vector
c = [1; 2; 3; 4]; % Column vector
```

Vectors can be defined with specific increments:

```
a = [1 : 2 : 10]; % Vector from 1 to 10 with a step of 2
```

Matrices can also be created directly:

```
v = [1 2; 3 4];
```

You can concatenate matrices or vectors, as long as their dimensions align. The apostrophe ' symbol is used for matrix transposition.

```
B = [v', v']; % Concatenate the transpose of v
num_rows = size(B,1); % Number of rows in B
num_cols = size(B,2); % Number of columns in B
```

To concatenate arrays along a specified dimension, use the `cat` function:

```
my_matrix = cat(1, my_vec1, my_vec2); % Concatenate along rows
```

o access elements in a matrix:

```
my_matrix(2,3); % Element in the 2nd row, 3rd column
my_matrix(:,2); % All elements in the 2nd column
my_matrix(1,:); % All elements in the 1st row
B = my_matrix(:, [1, 3]); % Elements in 1st and 3rd columns
```

MATLAB allows expanding matrices by assigning values outside the current bounds:

```
my_matrix(:,5) = 1; % Adds a 5th column with all values set to 1
```

The `end` keyword can be used to reference the last row or column:

```
my_matrix(1, [2 4]); % All rows and columns 2 and 4
my_matrix(:, end-1); % All rows, second-to-last column
```

Flatten a matrix into a single column vector:

```
my_vector = my_matrix(:);
```

1.1.3 Operations

MATLAB supports a variety of linear algebra operations on vectors and matrices. Note the difference between matrix and element-wise operations. Matrix operations:

```
v = [1 2 3];
v * v'; % Matrix multiplication
v' * v;
```

Element-wise operations:

```
[1 2 3] .* [4 5 6]; % Element-wise multiplication
[1 2 3] + 5; % Add 5 to each element
[1 2 3] ./ 2; % Divide each element by 2
[1 2 3] .^ 2; % Square each element
```

Common rounding functions:

```
ceil(10.56); % Round up
floor(10.56); % Round down
round(10.56); % Round to nearest integer
```

Basic arithmetic functions:

```
sum([1 2 3 4]); % Sum of vector elements
my_matrix = [1:4; 5:8];
sum(sum(my_matrix)); % Sum of all elements in a matrix
sum(my_matrix(:)); % Sum all elements after flattening
```

Using logical operators, you can create vectors of logical values:

```
b = v > 2; % Logical vector where each element of v > 2
```

1.2 Image processing

In MATLAB, images are represented as matrices, where pixel intensities correspond to matrix elements. Here, we'll cover how to load, display, modify, and process images.

1.2.1 Image loading

To import and display an image:

```
im = imread('E1_data/image_Lena512.png'); % Load an image
imshow(im); % Display the image
imagesc(im), colormap spring; % Display with arbitrary colormap
```

When converting an image to `double` format, rescale the pixel intensities to the [0, 1] range:

```
imshow(double(im) / 255); % Convert to double in the range [0, 1]
```

1.2.2 Image histogram

An image histogram shows the distribution of intensity values. You can compute and display it as follows:

```
h = hist(im(:), 0:255); % Histogram of intensity values
figure;
stairs(0:255, h, 'b', 'LineWidth', 3); % Plot histogram
title('Intensity Histogram');
xlabel('Intensity');
ylabel('Frequency');
axis tight;
```

Histograms represent the probability density function (PDF) of pixel intensities. Histogram equalization adjusts these intensities to make the distribution more uniform:

```
cdf_im = cumsum(h); % Cumulative distribution function
cdf_im = cdf_im / cdf_im(end); % Normalize to range [0, 1]
figure;
stairs(0:255, cdf_im, 'b', 'LineWidth', 3);
title('Cumulative Distribution Function');
xlabel('Intensity');
ylabel('CDF');
```

To map pixel intensities based on the CDF:

```
map = floor(255 * cdf_im); % Mapping based on CDF
im_eq = map(im + 1); % Equalized image
hist_im_eq = hist(im_eq(:), 0:255); % Histogram of equalized image
figure;
subplot(2, 2, 1);
imshow(im);
title('Original Image');
subplot(2, 2, 2);
imshow(im_eq / 255);
title('Equalized Image');
subplot(2, 2, 3);
bar(h);
title('Original Histogram');
subplot(2, 2, 4);
bar(hist_im_eq);
title('Equalized Histogram');
```

1.2.3 Image Modification

You can adjust image brightness by adding a fixed value to each pixel. For example:

```
figure;
imshow([im, im + 50]);
title('Brightness Increased by 50 Levels');
```

Contrast adjustments can also be controlled by setting intensity limits in `imshow`:

```
figure;
subplot(2,2,1);
imshow(im, [-100, 156]);
title('Brightened, Same Contrast');
subplot(2,2,2);
imshow(im, [0, 156]);
title('Brightened and Higher Contrast');
subplot(2,2,3);
imshow(im, [100, 256]);
title('Dimmed, Higher Contrast');
subplot(2,2,4);
imshow(im, [100, 356]);
title('Dimmed, Same Contrast');
```

1.2.4 RGB Channels

Color images are represented as 3D matrices, with three layers corresponding to the Red, Green, and Blue channels. You can isolate and display each channel:

```
imr = im(:, :, 1);
imshow(imr);
title('Red Channel');
img = im(:, :, 2);
imshow(img);
title('Green Channel');
imb = im(:, :, 3);
imshow(imb);
title('Blue Channel');
```

1.2.5 Logical Operations on images

Logical operations can be applied to images to create binary masks:

```
l = imb > (1.3 * imr); % Pixels where blue is 30% stronger than red
imshow(l);
title('Blue 30% Stronger than Red');
```

1.2.6 Gamam correction

Gamma correction adjusts brightness non-linearly. Here's a visualization of different gamma transformations:

```
x = 0:0.001:1;
figure;
hold on;
for gamma = [0.04, 0.1, 0.2, 0.4, 0.7, 1, 1.5, 2.5, 5, 10, 25]
    y = x .^ gamma;
    plot(x, y, 'DisplayName', sprintf('\gamma = %.2f', gamma), 'LineWidth', 3);
    text(x(round(end / 2)), y(round(end / 2)), sprintf('\gamma = %.2f', gamma));
    end
xlabel('Input Intensity');
ylabel('Output Intensity');
title('Gamma Correction Curves');
legend('Location', 'eastoutside');
grid on;
hold off;
```

1.3 Homogeneous coordinates in MATLAB

To illustrate homogeneous coordinates, we begin by analyzing a 5x5 checkerboard image and selecting points manually. Let's start by loading the image:

```
% Load the checkerboard image
```

```
I = imread('E1_data/checkerboard.png');
figure(1), imshow(I);
hold on;
[x, y] = getpts();
```

After selecting four points on the checkerboard, we store them as homogeneous coordinates by setting the third component to 1:

```
% Define points in homogeneous coordinates
a = [x(1); y(1); 1];
b = [x(2); y(2); 1];
c = [x(3); y(3); 1];
d = [x(4); y(4); 1];
```

To visualize the selected points, we add labels with different colors at each point's location:

```
% Display points on the image
text(a(1), a(2), 'a', 'FontSize', 12, 'Color', 'r');
text(b(1), b(2), 'b', 'FontSize', 12, 'Color', 'b');
text(c(1), c(2), 'c', 'FontSize', 12, 'Color', 'g');
text(d(1), d(2), 'd', 'FontSize', 12, 'Color', 'c');
```

Lines in homogeneous coordinates are represented by 3D vectors, where each vector contains the coefficients of the line equation. Using the cross product, we compute several lines based on the selected points:

```
% Compute lines from pairs of points
lab = cross(a, b); % Line through points a and b (reference line)
lad = cross(a, d); % Line orthogonal to lab
lac = cross(a, c); % Line at 45 degrees with lab
lcd = cross(c, d); % Line parallel to lab
```

To verify that points lie on certain lines, we can compute the incidence relations. For instance, checking if a point a lies on the line lab:

```
% Check incidence relations
a' * lab; % Should be close to zero if a in lab
c' * lcd; % Should also be zero if c in lcd
```

We define the borders of the image as lines in homogeneous coordinates and find intersections with our computed lines.

```
% Define the image border lines
c1 = [1; 0; -1];   % Left-most column
c500 = [1; 0; -500]; % Right-most column
r1 = [0; 1; -1];    % Top-most row
r500 = [0; 1; -500]; % Bottom-most row

% Compute the intersection between lab and the left-most column
x1 = cross(c1, lab);
x1 = x1 / x1(3); % Convert to Cartesian coordinates for plotting
text(x1(1), x1(2), 'x1', 'FontSize', 12, 'Color', 'b');
```

```
% Similarly, for the right-most column
x500 = cross(c500, lab);
x500 = x500 / x500(3);
text(x500(1), x500(2), 'x500', 'FontSize', 12, 'Color', 'b');

% Plot line segment between the intersections
plot([x1(1), x500(1)], [x1(2), x500(2)], 'LineWidth', 3);
```

To find the angle between two lines, we use the dot product and calculate the angle ϑ:

```
% Calculate the angle between lab and lac
l = lab;
m = lac;
cosTheta = (l(1) * m(1) + l(2) * m(2)) / sqrt(sum(l(1:2).^2) * sum(m(1:2).^2));
theta = acosd(cosTheta);
```

We verify that any linear combination of points a and b lies on lab:

```
% Linear combination of points
lambda = rand(1);
mu = 1 - lambda;
p = lambda * a + mu * b;
p' * lab; % Should be zero if p in lab
p = p / p(3); % Convert to Cartesian coordinates
text(p(1), p(2), 'p', 'FontSize', 12, 'Color', 'b');
```

Similarly, any linear combination of two lines passes through their intersection:

```
% Intersection of lines lab and lad
x0 = cross(lab, lad);
lambda = rand(1);
mu = 1 - lambda;
l = lambda * lab + mu * lad;
x0' * l; % Should be zero if x0 in l
```

When intersecting parallel lines, the result is a point at infinity. Here, we check intersections of lines that are theoretically parallel in homogeneous coordinates:

```
% Intersections of parallel lines lab and lcd
vac = cross(lab, lcd);
vac = vac / vac(3); % Point at infinity

% Intersections of top and bottom borders
vab = cross(r1, r500);
vad = cross(c1, c500);
```