Computer Vision 1

1. Course Plan

The following topics will be covered:

- Introduction and Overview
- Image Formation
- Basic Image Processing
- Template-Based Object Recognition
- Interest Points & Image Features
- Single & Two-View Geometry
- Stereo Vision
- Dense Motion Estimation
- Local & global image representations
- Object categorization & detection
- Deep Learning Approaches

Introduction and Overview

1. What is computer vision?

Computer Vision or machine vision is mainly about developing computational models and algorithms to interpret digital images and understand the visual world we live in. But what does it mean to "see" and "perceive"? To "see" means to become aware of something from observation or from a written or other visual source. To "perceive" means to become aware of something using one's senses, especially that of sight.

2. Why is it useful?

Computer Vision has many applications for:

- Face detection
- Human Pose estimation
- Earth Viewers
- Photo Browsing
- Optical character recognition
- Driver Assistance Systems
- Special Effects (Shape Capture, Motion Capture, Sports)
- Vision in space
- Robotics
- Medical Imaging

3. Why is it interesting?

It is a challenging problem that is far from being solved. It combines insights and tools from many fields and disciplines: Mathematics and statistics, Cognition and perception, Engineering (signal processing) and of course, computer science. It is a growing field: Cameras are becoming increasingly commonplace.

4. How might we go about it?

We need a formal model that describes our problem as well as an algorithm that realizes (i.e., implements) it. Neither the model alone, nor the algorithm alone suffices (in the long run). Both mathematically and computationally. Two main questions arise:

- What properties/cues of the visual world can we exploit or measure?
- What general (prior) knowledge of the world (not necessarily visual) should we exploit?

We as humans use many different cues to interpret what is in an image. They are not always right. We use information on what we regard as being a plausible and meaningful interpretation. The measurement alone does not suffice. This is necessary because an image is a 2D projection of a complex 3D world. A lot of information about the world is lost when we take a picture. Additionally, our image *data is always ambiguous*. Not only is the image data too little to fully recover and understand the "state of the visible world" (due to an image

being a 2D projection of a complex 3D world) but it may also be of poor quality because of low resolution, (sensor) noise etc.

The goal now is to devise models and algorithms to understand the visual world much like we do. This means we must use a lot of different cues, we have to deal with ambiguity, and we have to exploit what is a plausible and meaningful interpretation in order to extract information about the visual world from a small amount of data.

CV is an inverse problem and to make it work and turn around these cues we need:

- to understand the geometry and physics of the world (to some extent)
- a mathematical model of the cues that we want to exploit
- a mathematical model of our prior knowledge of the world
- a computational model and an algorithm to infer the state of the world from cues and prior knowledge.

Image Formation

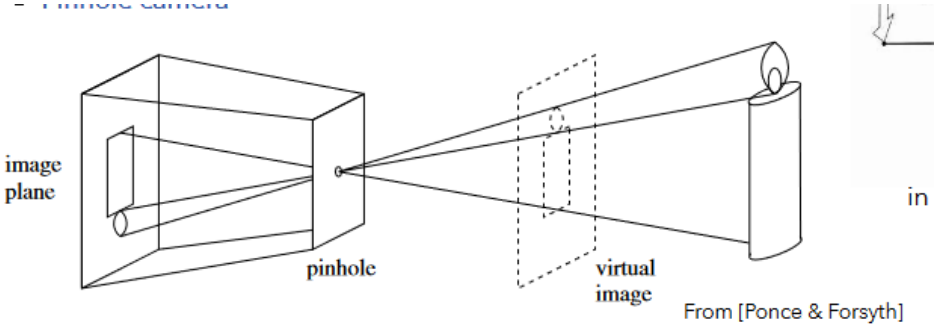
1. Introduction to Cameras

Cameras are everywhere these days. Not just your digital and video cameras. Essentially every new cell phone or computers has a camera built in (are there more cameras than people?). Let's look at the history of cameras:

- From "camera obscura" (dark room)
- Basic principle known to Mozi (470-391 BC) and Aristoteles (384-322 BC)
- Described by Ibn Al-Haytham (Alhazen) in his "Book of optics" (around 1000 AD).
- Drawing aid for artists: described by Leonardo da Vinci (1452-1519)
- 2015: International Year of Light (1000th anniversary of Al-Haytham's work)

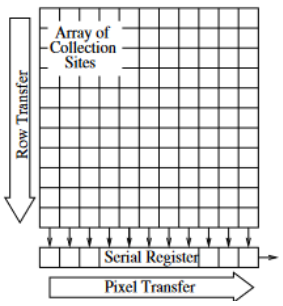
How so we design a camera? Da Vinci stated that "When images of illuminated objects ... penetrate through a small hole into a very dark room ... you will see [on the opposite wall] these objects in their proper form and colour, reduced in size ... in a reversed position, owing to the intersection of the rays".

This describes exactly how the Pinhole Camera (idealized standard camera model) works as seen in the figure below:



The pinhole camera is very simple but for many purposes sufficient. By putting only, a piece of film (or CCD) in front of an object the image would be blurry. A pinhole would reduce this. A pinhole camera is model for many common sensors like human eyes, photographic cameras, X-ray machines etc.

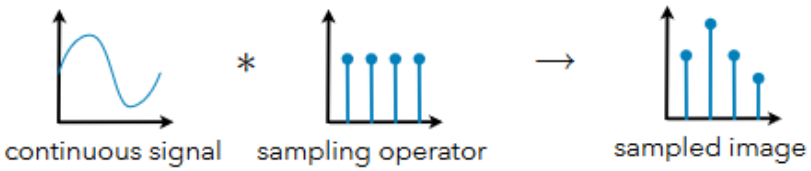
We usually work with the upright virtual image. For the image plane we use a CCD or CMOS Sensor:



Images arriving at CCD or CMOS sensor are spatially ("räumlich") discrete with individual pixels. But the visual world is not discrete. The light hits the sensor everywhere. The spatially (räumlich) **continuous** intensity function is defined as follows:

$$I(x, y), I: \mathbb{R}x\mathbb{R} \rightarrow \mathbb{R}$$

The image sensor performs a sampling of this function and turns it into discrete array of intensity values. The next figure illustrates the idealized spatial sampling (1D analogy):



For our assumptions we will regard the image as spatially discrete as an array of pixels.

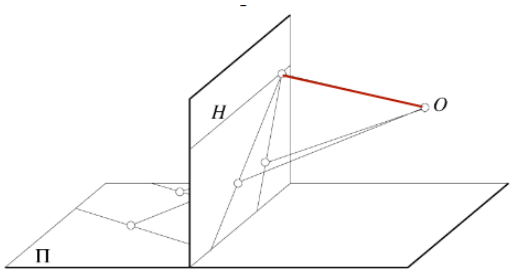
2. The Pinhole Projection Model

In this chapter we will cover the properties of pinhole cameras and look at the algebraic model of the perspective projection matrix.

2.1. Properties of pinhole cameras

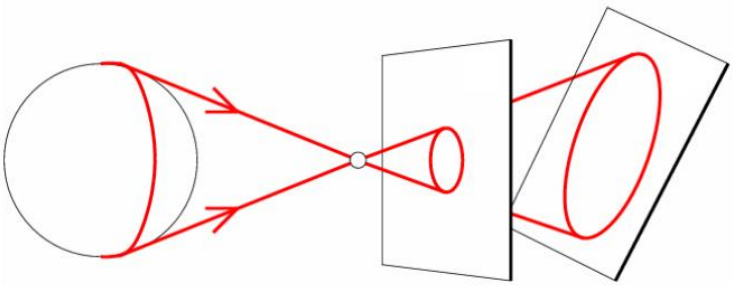
Let's look at some general projection properties of the pinhole camera:

- Many-to-none: all points along the same ray map to the same point in image
- Points → Points: undefined on focal plane
- Lines → Lines (collinearity is preserved): Lines through focal point projects to a point
- Plane → planes (or half-planes): but plane through focal point projects to line
- Parallel lines converge at a vanishing point: Each direction in space has its own vanishing point, but parallels also parallel to the image plane remain parallel. All directions in the same plane have vanishing points on the same line.



How do we construct the vanishing point/line?

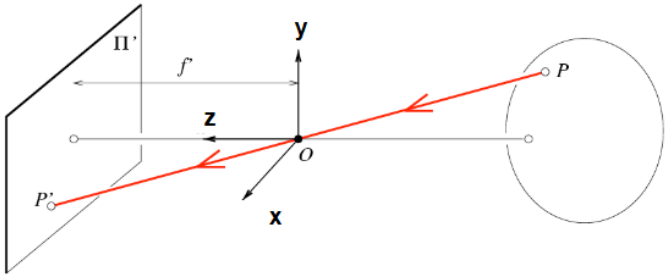
Perspective distortion is a common challenge for pinhole cameras. For example, let's look at the projection of a sphere. As one can see in the figure below if the focal point is normal to the image plane, we don't have a distortion otherwise the sphere turns into an oval shape.



We can summarize that perspective distortion does not happen due to lens flaws, but due to the orientation and position of the image plane regarding the focal point.

2.2. Algebraic Model: Perspective projection matrix

To model the projection, we will use the pinhole model as an approximation where the optical centre O is at the origin of the coordinate system and the image plane π' is in front of O as seen in the next figure:



To derive the perspective projection of a pinhole camera we first need to compute the intersection of a ray in this case from point $P = (x, y, z)$ with the image plane π'. Using similar triangles, we then obtain the following relations:

$$\frac{x}{x'} = \frac{z}{f'} \rightarrow x' = f' \frac{x}{z}$$
$$\frac{y}{y'} = \frac{z}{f'} \rightarrow y' = f' \frac{y}{z}$$

With $f' = z'$ being the project depth. But the division with z is not a linear transformation. Instead of using $(x, y, z) \rightarrow (f' \frac{x}{z}, f' \frac{y}{z})$ we can add one more coordinate (we are basically storing the division with z for a later time), which leads us to *homogeneous image and scene coordinates*:

$$(x, y) \rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$(x, y, z) \rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Converting back from homogenous to cartesian coordinates we just need to divide by the last coordinate (third for image, fourth for scene).

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} \rightarrow \left(\frac{x}{w}, \frac{y}{w}\right)$$

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

This leads us now to the *projection matrix* which is a matrix multiplication in homogeneous coordinates:

$$\begin{bmatrix} f' & 0 & 0 & 0 \\ 0 & f' & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} f'x \\ f'y \\ z \end{pmatrix} \rightarrow \left(f'\frac{x}{z}, f'\frac{y}{z}\right) \; divide\ by\ the\ third\ coordinate$$

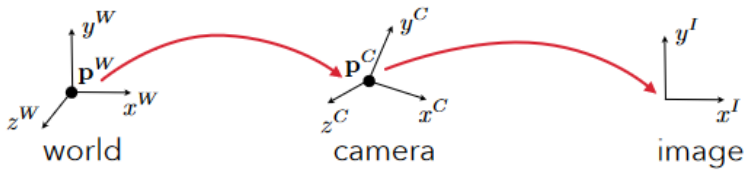
3. Single View Geometry

In this chapter we will cover coordinate transformations from world to image coordinates and about camera calibration.

3.1. Projection from world to image coordinates

We are considering 3 coordinate systems which are highlighted also in the next figure:

- World coordinate system: Objects
- Camera coordinate system: Camera system
- Image coordinate system: Image



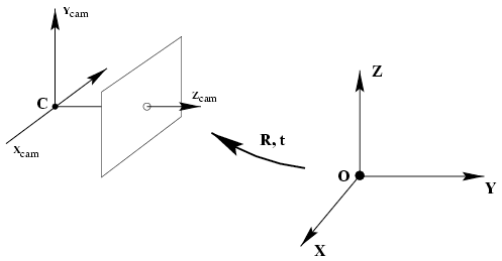
An extrinsic camera transformation takes world into camera coordinates. An intrinsic camera transformation describes the *image formation process* and takes camera into image coordinates.

Extrinsic Camera Transformation

The *extrinsic camera transformation* is described as a linear transformation using homogeneous coordinates (augment a vector with a constant 1). It is a combination of rotation and translation.

The camera coordinate frame is related to the world coordinate frame by a rotation and a translation. In non-homogenous coordinates we obtain:

$$\tilde{\boldsymbol{x}}^C = \boldsymbol{R}(\tilde{\boldsymbol{x}}^W - \tilde{\boldsymbol{c}})$$



where $\tilde{c}$ is the coordinate of the camera centre in the world frame. Using homogeneous coordinates yields:

$$\boldsymbol{x}^C = \begin{pmatrix} \boldsymbol{R}(\tilde{\boldsymbol{x}}^W - \tilde{\boldsymbol{c}}) \\ 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{R}\tilde{\boldsymbol{x}}^W - \boldsymbol{R}\tilde{\boldsymbol{c}} \\ 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{R}\cdot\tilde{\boldsymbol{x}}^W - \boldsymbol{R}\cdot\tilde{\boldsymbol{c}} \\ 0\cdot\tilde{\boldsymbol{x}}^W + 1\cdot 1 \end{pmatrix} = \begin{bmatrix} \boldsymbol{R} & -\boldsymbol{R}\tilde{\boldsymbol{c}} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{\boldsymbol{x}}^W \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R} & -\boldsymbol{R}\tilde{\boldsymbol{c}} \\ 0 & 1 \end{bmatrix} \boldsymbol{x}^W$$
$$= \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ 0 & 1 \end{bmatrix} \boldsymbol{x}^W$$

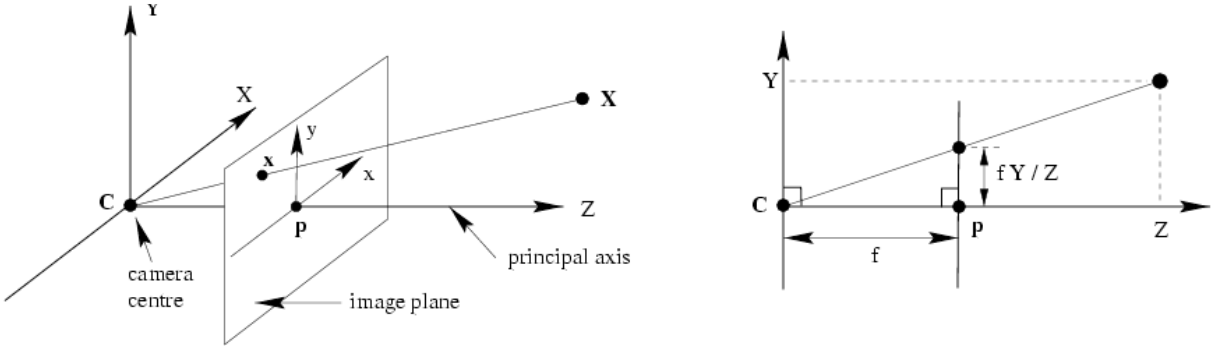
where the translation can be written as $\boldsymbol{t} = -\boldsymbol{R}\tilde{\boldsymbol{c}}$ and $\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ 0 & 1 \end{bmatrix} \in \mathbb{R}^{4x4}$.

Intrinsic Camera Transformation

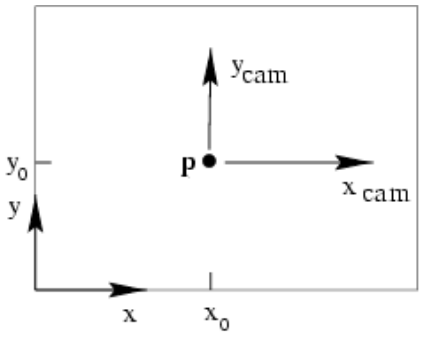
The *intrinsic camera transformation* is described as a linear transformation plus a perspective division. The pinhole camera model we are using is a special case, but the intrinsic camera transformation can deal with more general camera models as well.

We fist need to define new terminologies to describe intrinsic camera transformation.

- *Principal axis:* line from the camera center perpendicular to the image plane.
- *Normalized (camera) coordinate system:* camera center is at the origin and the principal axis is the z-axis
- *Principal point ($p = (p_x, p_y)$):* point where principal axis intersects the image plane (origin of normalized coordinate system)



The origin of the image coordinate system is in the corner and the origin of the camera coordinate system is at the principal point $p = (p_x, p_y)$ when looking at the image plane. This results in a *principal point offset* as seen in the figure!



Let's Recall Pinhole camera model $(x, y, z) \rightarrow \left(f\,\frac{x}{z}, f\,\frac{y}{z}\right)$. In homogenous coordinates the relationship is described using homogenous coordinates and the projection matrix:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} fx \\ fy \\ z \end{pmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} x \\ Y \\ Z \\ 1 \end{pmatrix}$$

More compactly $\boldsymbol{x}^I = \boldsymbol{K}[\boldsymbol{I}|\boldsymbol{0}]\boldsymbol{x}^C = \boldsymbol{K}\tilde{\boldsymbol{x}}^C$. The next step is to consider the principal point offset $(x + p_x, y + p_y, z) \rightarrow (f\,\frac{x}{z} + p_x, f\,\frac{y}{z} + p_y)$. Again, in homogenous coordinates we will get:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} fx + zp_x \\ fx + zp_y \\ z \end{pmatrix} = \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

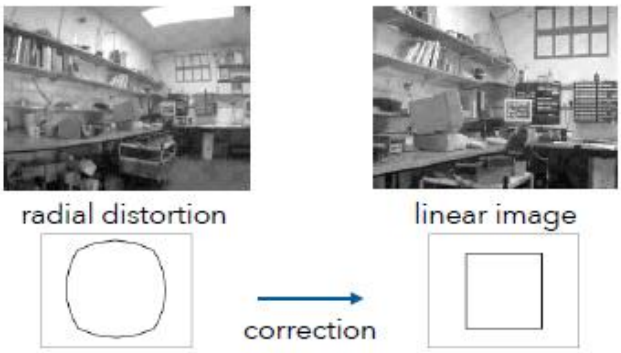
where $\boldsymbol{K} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix}$ is called the *calibration matrix*. However, we are not done yet. To finally get the pixel/image plane coordinates we need to do one more matrix multiplication with *pixel magnification factors*, which are the bridge between the digital image world and the physical world:

$$\mathrm{K} = \underbrace{\begin{bmatrix} m_x & & \\ & m_y & \\ & & 1 \end{bmatrix}}_{\text{pixels/unit}} \underbrace{\begin{bmatrix} f & & p_x \\ & f & p_y \\ & & 1 \end{bmatrix}}_{\text{units}} = \underbrace{\begin{bmatrix} \alpha_x & & \beta_x \\ & \alpha_y & \beta_y \\ & & 1 \end{bmatrix}}_{\text{pixels}}$$

where m_x are pixels per unit (m, mm, inch,...) in horizontal direction and m_y are pixels per unit in the vertical direction. The full pixel size is given by $\frac{1}{m_x} \times \frac{1}{m_y}$. To summarize the intrinsic parameters are the:

- Principal point coordinates
- Focal length
- Pixel magnification factors

Other factors like skew (non-rectangular pixels) are not modelled with linear calibration. This causes radial distortion, which must be corrected afterwards! Here is an example for radial distortion:



The final intrinsic camera transformation can be compactly written in (homogeneous coordinates) by using the full calibration matrix $\boldsymbol{K}$:

$$\boldsymbol{x}^I = \boldsymbol{K}[\boldsymbol{I}|\boldsymbol{0}]\boldsymbol{x}^C = \boldsymbol{K}\tilde{\boldsymbol{x}}^C$$

where $\boldsymbol{K} \in \mathbb{R}^{3x3}$.

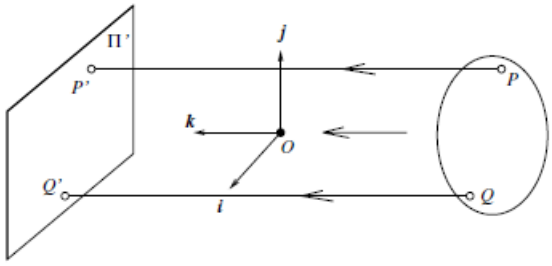
Perspective Projection Pipeline

Now we can concatenate the extrinsic and intrinsic camera transformations into a single *projection matrix* $\boldsymbol{P}$:

$$\boldsymbol{x}^I = \boldsymbol{K}[\boldsymbol{I}|\boldsymbol{0}]\boldsymbol{x}^C = \boldsymbol{K}[\boldsymbol{I}|\boldsymbol{0}]\begin{bmatrix}\boldsymbol{R} & \boldsymbol{t}\\ 0 & 1\end{bmatrix}\boldsymbol{x}^W = \boldsymbol{K}[\boldsymbol{R}|\boldsymbol{t}]\boldsymbol{x}^W = \boldsymbol{P}\boldsymbol{x}^W$$

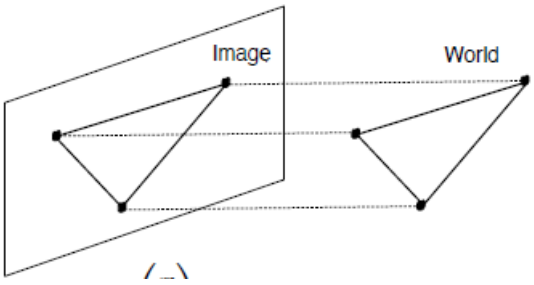
where $\boldsymbol{P} \in \mathbb{R}^{3x4}$.

A *special case* where the projection matrix can easily be obtained is the *orthographic camera*.



$$x' = x$$
$$y' = y$$

The distance from the center of the projection to the image plane is infinite. This is also called "parallel projection".



Hence the projection matrix is simple to obtain:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad \Rightarrow \quad (x, y)$$

3.2. Camera calibration

Given n points with known 3D coordinates $\boldsymbol{X_i}$ and known image projections $\boldsymbol{x}_i$, estimate the camera parameters. To solve this problem, we will use collinearity:

$$\lambda \boldsymbol{x_i} = \boldsymbol{P X_i} \rightarrow \lambda \begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} = \begin{bmatrix} \boldsymbol{p}_1^T \\ \boldsymbol{p}_2^T \\ \boldsymbol{p}_3^T \end{bmatrix} \boldsymbol{X_i}$$

where $\boldsymbol{p}_j^T$ are row vectors of $\boldsymbol{P}$. Then we will rewrite the current equation to a cross product (which will expand), because we know that $\boldsymbol{x_i}$ and $\boldsymbol{X_i}$ have the same direction and to compensate for not knowing the scale we just add a variable λ.

$$x_i \times \boldsymbol{P X_i} = \boldsymbol{0} \rightarrow \begin{pmatrix} y_i \boldsymbol{p}_3^T \boldsymbol{X}_i - \boldsymbol{p}_2^T \boldsymbol{X}_i \\ \boldsymbol{p}_2^T \boldsymbol{X}_i - x_i \boldsymbol{p}_3^T \boldsymbol{X}_i \\ x_i \boldsymbol{p}_2^T \boldsymbol{X}_i - y_i \boldsymbol{p}_1^T \boldsymbol{X}_i \end{pmatrix} = \boldsymbol{0}$$

Rewriting the expanded cross-product to an equation system yields:

$$\begin{bmatrix} 0 & -\boldsymbol{X}_i^T & y_i \boldsymbol{X}_i^T \\ \boldsymbol{X}_i^T & 0 & -x_i \boldsymbol{X}_i^T \\ -y_i \boldsymbol{X}_i^T & x_i \boldsymbol{X}_i^T & 0 \end{bmatrix} \begin{pmatrix} \boldsymbol{p_1} \\ \boldsymbol{p_2} \\ \boldsymbol{p_3} \end{pmatrix} = \boldsymbol{0} \rightarrow \boldsymbol{A p} = \boldsymbol{0}$$

where $\boldsymbol{p}$ is the *projection matrix* as a vector (entries of $\boldsymbol{P}$ are stacked up in a vector) and $\boldsymbol{A}$ is the (rectangular) *equation system matrix*. $\boldsymbol{P}$ has 11 degrees of freedom (12 parameters, but scale is arbitrary).One 2D/3D correspondence gives us two linearly independent equations. So, in total 6 correspondences are needed for a minimal solution. Using more points, a least-squares estimation is the way to go.

How do we solve this equation system? The trivial soliton $\boldsymbol{p} = 0$ is a problem. To solve this problem, we will add a constraint on norm of $\boldsymbol{p}$ (value can be arbitrarily):

$$\boldsymbol{A p} = 0 \; s.t. \; \|\boldsymbol{p}\| = 1$$

In practice we will use a method called *homogenous least squares*: Since the entries of A are measured and thus affected by noise, we rather minimize the (squared) error:

$$\|\boldsymbol{A p}\|^2 \; s.t. \; \|\boldsymbol{p}\| = 1$$

Then we find the right nullspace (right nullspace: $\boldsymbol{A r} = 0$, left nullspace: $\boldsymbol{I^T A} = 0$) of $\boldsymbol{A}$ using *singular value decomposition* (SVD):

1. Decompose equation system matrix $\boldsymbol{A} = \boldsymbol{U S V^T}$
2. Assume that singular values are sorted. i.e., $\boldsymbol{S} = diag(s_1, \dots, s_{12}), s_{i+1} \leq s_i$
3. Take last singular vector $\boldsymbol{p} = \boldsymbol{v_{12}}$

Why does this work? You can look at the slides 13-16. No need to know all the proof!

Once we've recovered the values of the camera matrix, we still must figure out the intrinsic and extrinsic parameters. Here is how to do it step by step:

1. First split the projection matrix into a 3x3 matrix and a 3x1 vector:

$$\mathbf{P} = [\mathbf{M}|\mathbf{m}] = [\mathbf{KR}| - \mathbf{KR}\tilde{\mathbf{c}}]$$

2. Next, decompose ***M*** into upper triangular part ***K*** (calibration) and orthonormal part ***R*** (rotation) using RQ-decomposition.

3. If required, extract 3 rotation axis and angle from ***R*** in which its columns are the axes of the rotated coordinate system.

4. Finally, find ***c*** as the nullspace of ***P*** by means of SVD.

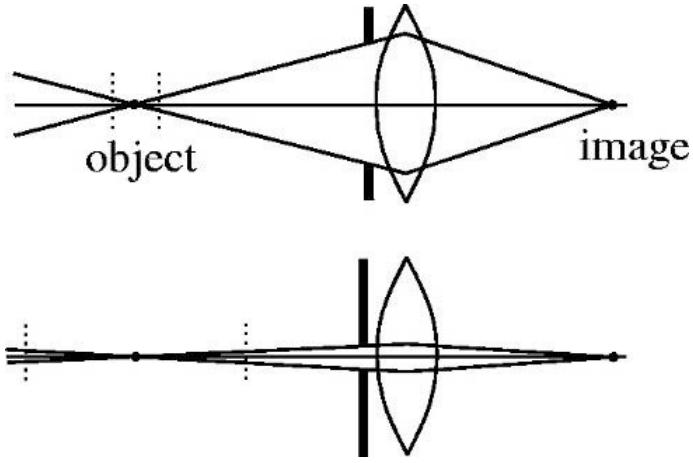
Important Conclusions

What have we got now? We have a projection matrix ***P***, which maps any point in the 3D world coordinate frame onto the (infinite) image plane of the camera. Given a 3D point in the world, we can find out where its projection is in the image.

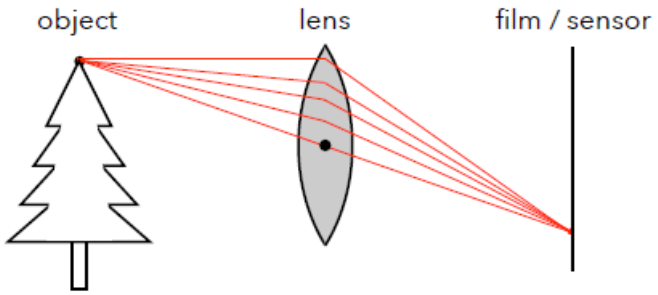
One goal of vision is that given a point in the images, find out where their pre-image is in the world (3D reconstruction). However, we cannot do this from one image, because the depth is lost. The projection matrix can only tell us about the ray through an image point ***x***. We will cover this topic in stereo vision later.

4. Cameras with Lenses

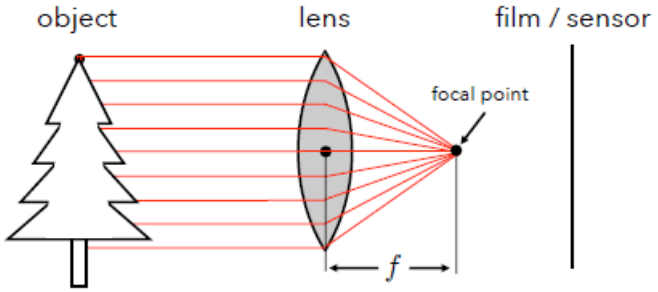
Let's build a real camera now. A home-made pinhole camera will always be blurry (due to size of the aperture too much light gets through) and even if we make the aperture as small as possible so that less light gets through, we will still get diffraction effects! The solution is to additionally use a lens behind the aperture. With that we can also control the depth of field depending on the size of the aperture (more later).



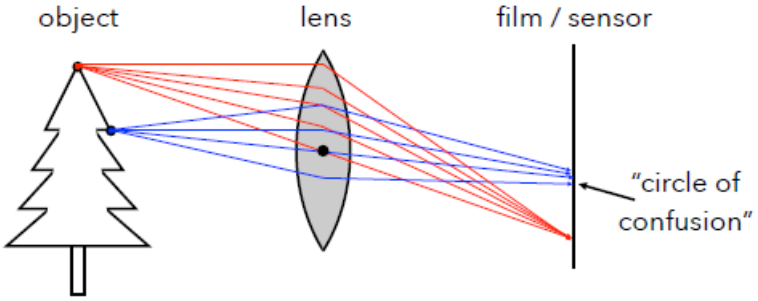
A lens focuses light onto the film. The rays passing through the center are not deviated.



All parallel rays converge to one point on a plane located at the focal length f.



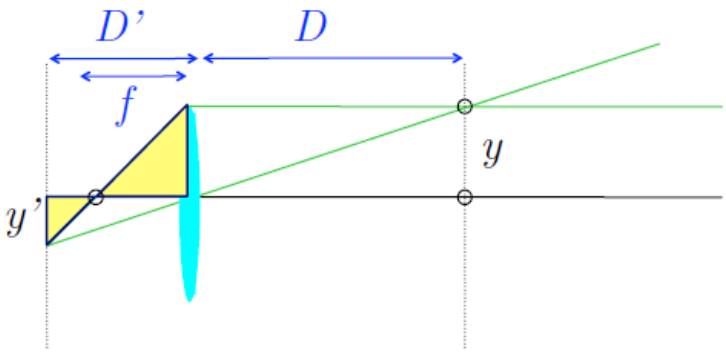
There is a specific distance at which objects are “in focus”. Other points project to a “circle of confusion” in the image:



4.1. Thin Lens Formula

We are applying similar triangle twice (see sketch) yield the thin lens formula:

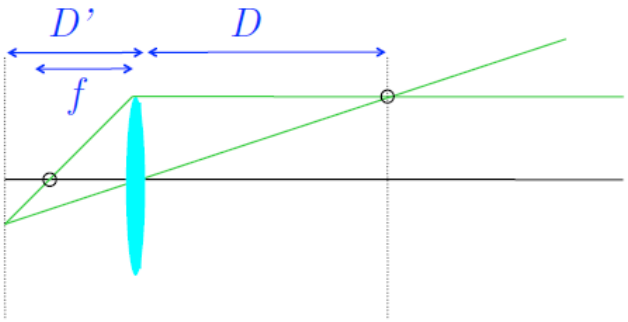
$$\frac{y'}{y} = \frac{D'}{D} \qquad \frac{y'}{y} = \frac{D' - f}{f}$$



$$\frac{1}{D'} + \frac{1}{D} = \frac{1}{f}$$

Any point satisfying the thin lens equation is in focus:

$$\frac{1}{D'} + \frac{1}{D} = \frac{1}{f}$$

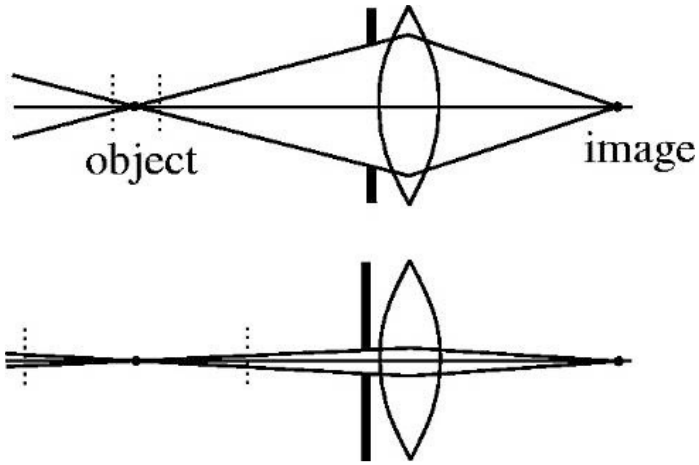


4.2. Depth of Field

How can we control the depth of field? The depth of field is the range in which the object is approximately in focus. Changing the aperture size affects the depth of field. Changing the depth of field means changing the distance of the lens to the sensor. The focal point of the lens can be also before or behind the sensor which will make the image blurry because the rays don't converge into a single point!

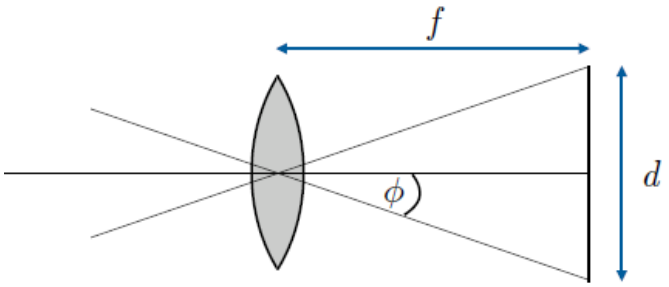
A smaller aperture increases the depth of field, but reduces the amount of light, so we need to increase the exposure (to the light)!

Increasing the aperture makes the depth of field smaller but requires lesser exposure (to the light). You see we have a tradeoff!



4.3. Field of View

Field of View (FOV)is the angle that's visible and depends on focal length and the size of the camera retina (sensor), A smaller FOV equals a larger focal length:



$$\phi = \tan^{-1}\left(\frac{d}{2f}\right)$$

Here is an example:

Large FOV, small f

Camera close to car

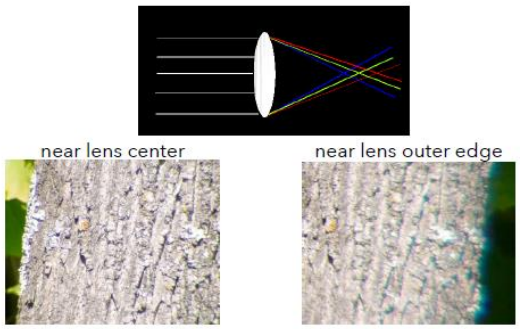
Small FOV, large f

Camera far from the car

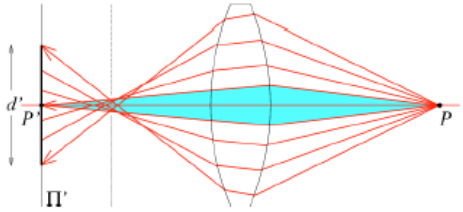
4.4. Lens Flaws

Let's look at some lens flaws:

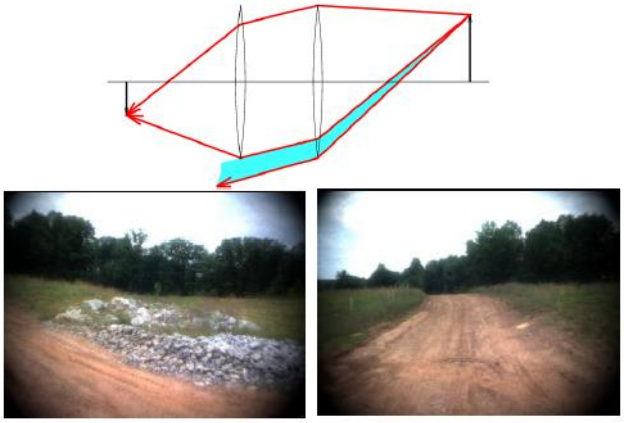
- *Chromatic aberration*: Lens has different refractive indices for different wavelengths. This causes color fringing.



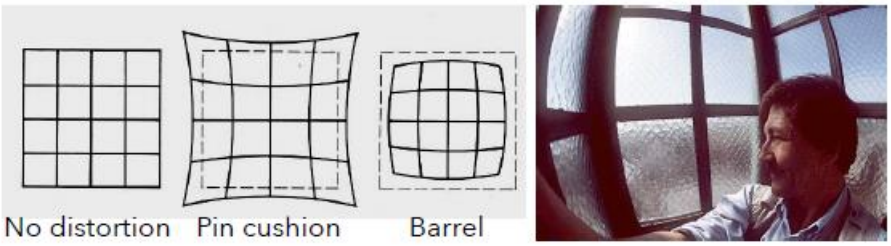
- *Spherical aberration*: Spherical lenses don't focus light perfectly. Rays farther from the optical axis focus closer which causes blur away from the image center.



- *Vignetting:* reduction of an image's brightness or saturation toward the periphery compared to the image center.



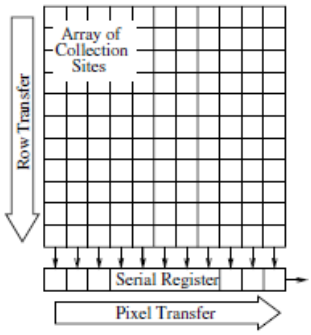
- *Radial Distortion:* Caused by imperfect lenses. Deviations are most noticeable for rays that pass through the edge of the lens.



5. Digital Cameras

A digital camera replaces film with a sensor array. Each cell in the array is light-sensitive diode that converts photons to electrons. Two common sensor types: Charge Coupled Device (CCD) and Complementary Metal Oxide Semiconductor (CMOS)

Images arriving at our CCD or CMOS sensor are spatially discrete with individual pixels.

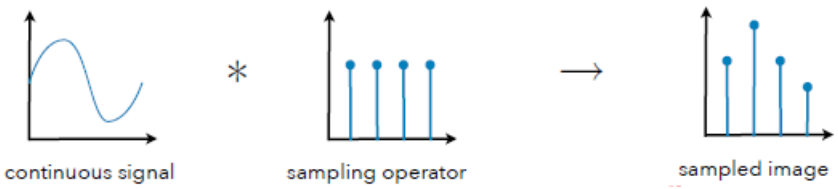


But is the visual world spatially discrete? No. Light hits the sensor everywhere. A spatially continuous intensity function can be defined as:

$$I(x,y), I: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

The image sensor performs a "sampling" of this function. It turns it into a discrete array of intensity values.

We usually do not work with spatially continuous functions since our cameras do not sense in this way. Instead use (spatially) discrete images by sampling the 2D domain on a regular grid. An idealized spatial sampling (1D analogy) is described here:



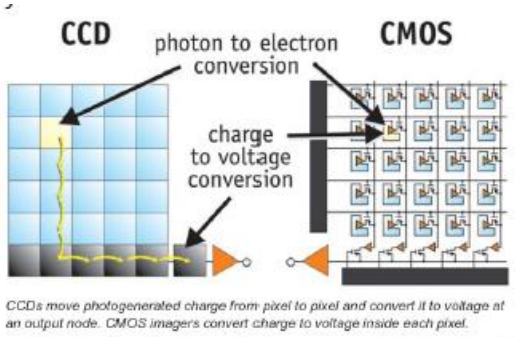
For more details investigate signal processing and Fourier theory. For our purposes here, we will regard the image as spatially discrete. So, an array of pixels.

5.1. CCD vs. CMOS

A CCD sensor transports the charge across the chip and reads it at one corner of the array. An analog-to-digital converter (ADC) then turns each pixel's value into a digital value by measuring the amount of charge at each photosite and converting that measurement to binary form.

A CMOS sensor uses several transistors at each pixel to amplify and move the charge using more traditional wires. The CMOS signal is digitized right away, so it needs no separate ADC. Also, often faster (for video applications).

These sensors don't get better with gathering more light but they are getting faster for getting out the electrical signals!

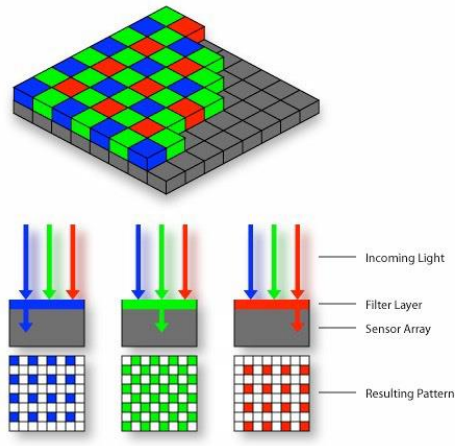


CCDs move photogenerated charge from pixel to pixel and convert it to voltage at an output node. CMOS imagers convert charge to voltage inside each pixel.

http://www.dalsa.com/shared/content/pdfs/CCD_vs_CMOS_Litwiller_2005.pdf

5.2. Color Sensing in Camera

A camera can only see RGB, so how does capture the color? We are using a Bayer grid or color filter array for each individual pixel of the sensor. For every pixel we get a RGB triplet as an approximation. We have more green pixels (50% better for the human eye to distinguish stuff) and usually 25% blue and 25% red.

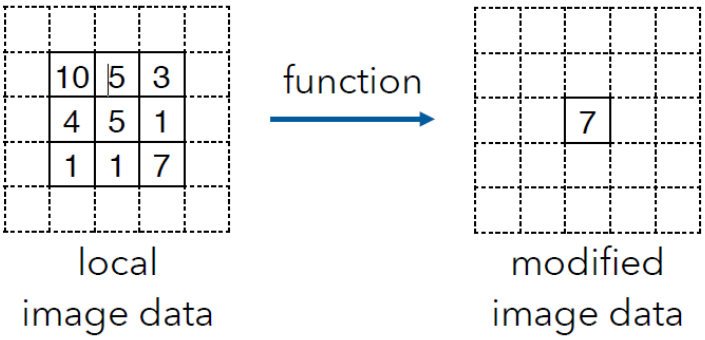


Estimate missing components from neighbouring values (demosaicing). However computational cost is high. Bayer sensors have problems which can be solved for example with an optical low-pass filter.

Basics of Digital Image Processing

1. Motivation

This chapter deals with some basics about (digital signal processing, FFT, …) and Image filtering. Image filtering can be used to reduce noise, to fill-in missing values/information and to extract image features (e.g., edges/corners).



2. Images

We can think of the image as a function:

$$I(x,y), I: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

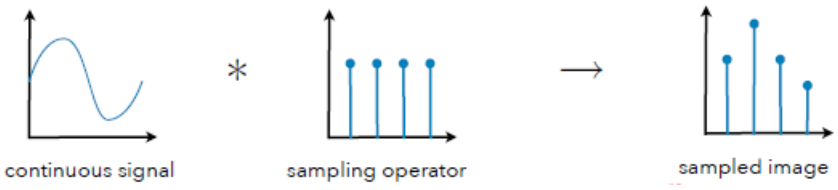
For every 2D point it tells us what the (light) intensity is. Realistically, the size of the sensor is limited and so is the range of brightness it can capture:

$$I(x,y), I: [a,b]x[c,d] \rightarrow [0,m]$$

Colour images can be thought of as vector-valued functions:

$$I(x,y) = \begin{pmatrix} I_R(x,y) \\ I_G(x,y) \\ I_B(x,y) \end{pmatrix}$$

We usually do not work with spatially continuous functions since our cameras do not sense in this way. Instead use (spatially) discrete images by sampling the 2D domain on a regular grid For 1D analogously we would get:



Intensity/colour values are usually also discrete. We quantize the values per channel (e.g., 8 bit).

3. Filtering

In this section we will cover linear and non-linear filtering approaches.

3.1. Linear Filtering

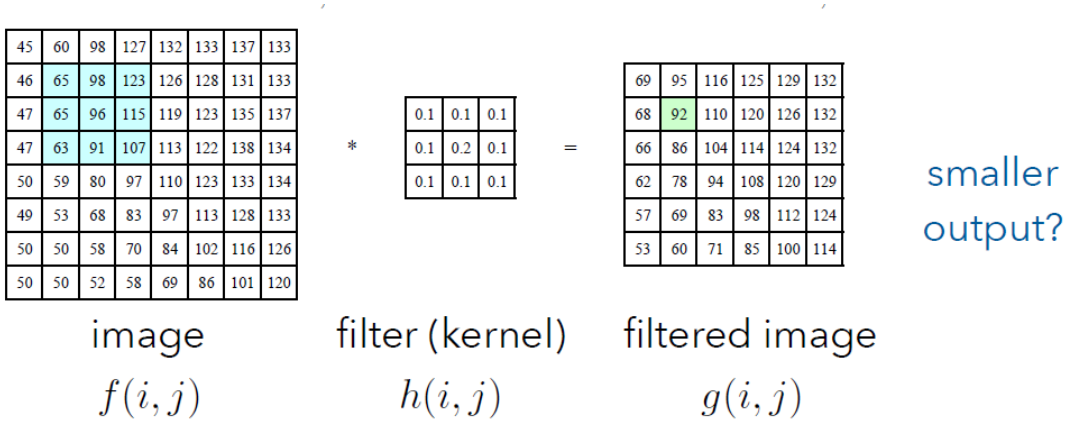
Let's start by defining basic properties of linear filtering meaning linear operations or systems:

- Homogeneity: $T[aX] = aT[X]$
- Additivity: $T[X + Y] = T[X] + T[Y]$
- Superposition: $T[aX + bY] = aT[X] + bT[Y]$

Examples we will use these properties are matrix-vector operations and convolutions.

Convolution

In a convolution *each pixel* will be replaced by a linear combination of its neighbours (and itself). Let's look at a *2D discrete convolution* in detail illustrated in the following picture:



When applying the filter for a pixel, we are multiplying the top left pixel of the image with bottom right of the filter (kernel) and sum everything up! This results in the following equation ($(*)$ is the convolution operator):

$$g = f * h \leftrightarrow g(i,j) = \sum_{k,l} f(i-k, j-l)h(k,l) = \sum_{k,l} f(k,l)h(i-k, j-l)$$

where i, j and k, l are the size of the filter which is (3x3) in this case. For the sum we have to start at $k = l = 0$.

A convolution further is:

- Linear: $h * (f_0 + f_1) = h * f_0 + h * f1$
- Associative: $(f * g) * h = f * (g * h)$
- Commutative: $f * h = h * f$
- Shift-invariant: $g(i,j) = f(i+k, j+l) \leftrightarrow (h * g)(i,j) = (h * f)(i+k, j+l)$
- Can be represented as matrix-vector product: $\boldsymbol{g = Hf}$

Continuous convolution versions do exist but will skip them!

Correlation

Note Correlation and Convolution are closely related but it does not mirror a filter. For symmetric filters convolution and correlation outputs are the same which makes sense if you look at it. In a correlation we are multiplying the top left pixel of the image with top left of the filter (kernel) and sum everything up! The following equation describes this ((⊗) is correlation operator):

$$g = f \otimes h \leftrightarrow g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l)$$

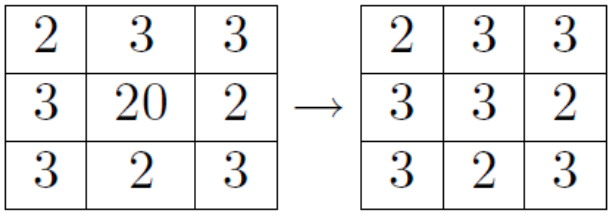
Notice this time we are using " + ".

Why use Filtering?

Why do we even use filtering? “Noise” is what are not interested in. We distinguish between two cases:

- low-level noise: light fluctuations, sensor noise, quantization effects, finite precision etc.
- Complex noise (not covered in this lecture): *s*hadows, extraneous objects.

Noise usually affects only 1 pixel and not its neighbors. Let’s assume we have 1 pixel with noise, and we want to remove it. How can we solve this? We are assuming the pixels neighborhood contains information about its intensity as explained in the next figure.



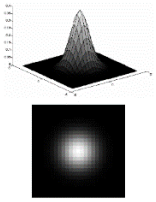
Box (Average) Filters and Gaussian Filters

There are two main types of linear filters which can be used to remove noise: *Box Filters* (a special case of the Average Filter) and the *Gaussian Filter*.

Using an *Average Filter* each pixel gets replaced with an average of its neighborhood. It’s basically a mask with positive entries that sum up to 1! If all weights are equal, it is called a *Box Filter* as seen in the next figure. Of course, we will lose details of the image.

$$\frac{1}{9} \cdot \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

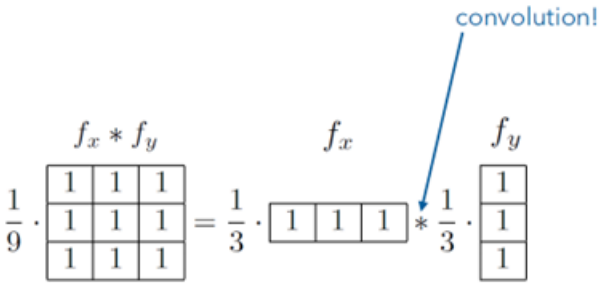
The *Gaussian Filter* is an isotropic Gaussian (rotationally symmetric) and weighs nearby pixels more than distant ones. The Smoothing kernel is proportional to $G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$ and represents a multivariate Gaussian distribution as highlighted in the next picture:



Both the Gaussian filter and box filter are *separable*! We first can convolve each row and then each column with a 1D filter:

$$(f_x * f_y) * I = f_x(f_y * I)$$

Remember that convolution is linear-associative and commutative! For example, let's look at the separable box filter:



Separable convolution filters can save a lot of computationally cost but only if the filter is big and also linear separable. Hence separable filters are very efficient!

The process of performing convolution requires K^2 operations per pixel where K is the size (height or width) of the convolution kernel e.g., the box filter in this case. With separable filtering this can be speed up to only $2K$ operations per pixel.

Boundary Issues

We need to be careful of the boundaries. Boundary issues arise due to darkening of the corner pixels, because the original image is being padded with 0 values wherever the convolution kernel extends beyond the original image boundaries! How can we compensate for this?

- *Zero:* Set all pixels outside the source image to 0
- *Wrap:* loop around the image the image in a toroidal configuration
- *Clamp:* repeat edge pixels indefinitely

- *Mirror:* reflect pixels across the image edge

3.2. Non-linear filtering & Morphology

There is a large variety of non-linear filters for noise removal. The problem with linear filtering is we are removing the noise and the signal, but we only want to remove noise and keep the signal. The solution is to use *non-linear filters*.

The simplest one is the *median filter*. First replace each pixel with the median in a neighborhood around it. Second sort the pixels in the local neighborhood and last take the middle value. But that's slow and computationally expensive.

There are also *morphological* filters which can only be applied to the special case *of binary images*. First some basics about morphological operations which apply only for binary images (1 = black and 0 = white). The convolution is performed with a structural element s. It's a binary mask (often circle or square). A thresholding to recover a binary image is performed:

$$\theta(f,t) = \begin{cases} 1 \ if \ f \geq t \\ 0 \ else \end{cases}$$

This can either result in *dilatation* $\theta(f * s, 1)$ which makes the original image thicker or in an *erosion* which makes the original image thinner $\theta(f * s, S)$ where S is the number of pixels. Morphological filters allow a generalization to grayscale images. How does it work?

1. Perform min/max-convolution with a structuring element s
2. Get previous behavior with flat structuring element:

$$s(x) = \begin{cases} 0 \ x \in B \\ -\infty \ otherwise \end{cases}$$

To achieve a dilation, we are using the *max operator* which makes the bright pixels thicker! To achieve erosion, we are using the *min operator* which makes the dark pixel thicker!

original

dilation

$\max_{\mathbf{y}}[f(\mathbf{x}) + s(\mathbf{x}-\mathbf{y})]$

erosion

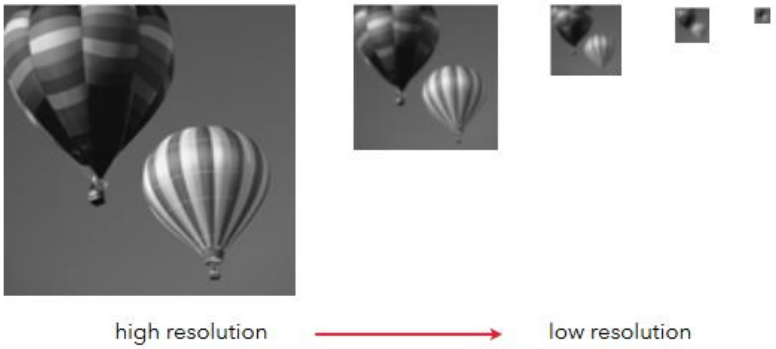
$\min_{\mathbf{y}}[f(\mathbf{x}) - s(\mathbf{y}-\mathbf{x})]$

4. Multi-scale image representation

We often may wish to change the resolution of an image before proceeding. For example, to match a computer screen or the output of a printer. Alternatively, we also want to reduce the size of an image resolution to speed up the execution of an algorithm or to save on storage space. Sometimes we also don't know the right resolution an image should be. For example, to the task to find a face in an image. In this case we don't know the scale at which the face will appear, so we need to generate a whole pyramid of differently sized images and scan each one for possible faces.

4.1. Gaussian pyramid

Gaussian pyramid represents images at multiple scales (resolutions) stacked up like a pyramid as seen in the next figure:



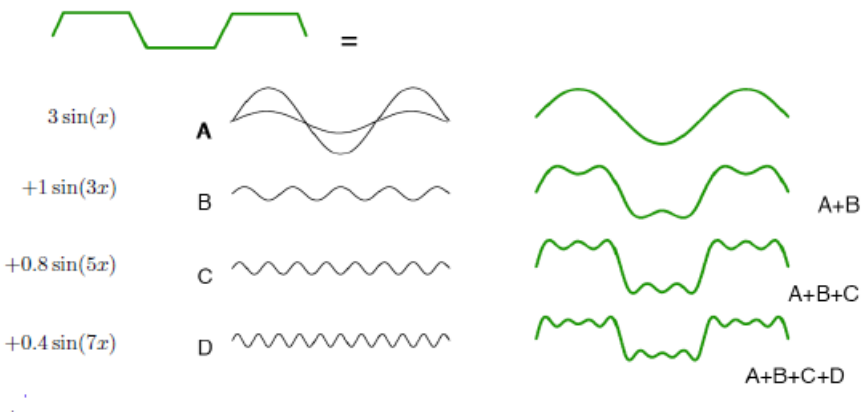
There are two important questions to answer: Which information is preserved over scales? And which information is lost over scales? But how do we even down sample in the first place?

1. Apply Gaussian smoothing G_{ρ^*}. This step is very important and crucial to avoid Aliasing!
2. Down sample (Usually take every second pixel or skip every second pixel)

But we can't shrink an image by taking every second pixel as explained above, because high frequencies cannot be represented anymore (high frequencies = details, see intensity plot). If we do that anyway, characteristic errors appear: Spatial frequencies are misinterpreted. This is called *Aliasing*! In addition, small phenomena look bigger and fast phenomena look slower. Some examples are wagon wheels rolling the wrong way in movies, checkerboards misinterpreted in ray tracing or striped shirts look strange on color television. So, constructing a pyramid by taking every second pixel leads to layers that badly misrepresent the top layer! To avoid Aliasing we are applying *gaussian smoothing*!

4.2. Laplacian pyramid

First let's recap a bit about the Fourier transform to talk about spatial frequencies.



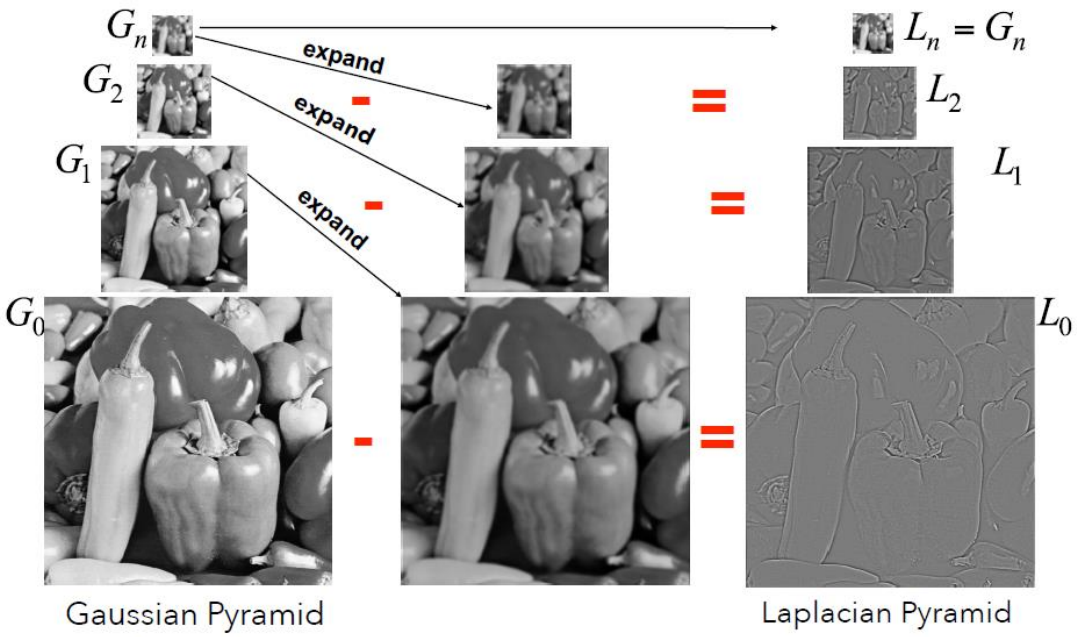
The *Laplacian pyramid* provides a simple *frequency decomposition into sub bands*. Each level/sub band contains only image structure of a particular range of spatial frequencies e.g., the finest level contains all the high-frequency detail.

$$L_i = G_i - expand(G_{i+1})$$

The best thing is we can get back the original image by *reversing the decomposition*.

$$G_i = L_i + expand(G_{i+1})$$

An example where this can be used is image sharpening by amplifying high-frequency components.



5. Edge detection

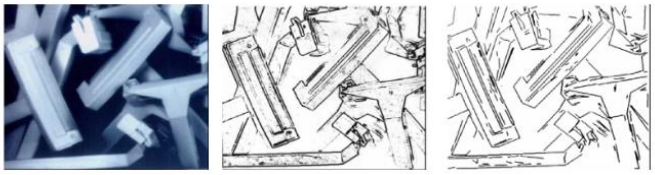
This section will handle edge detection. A very interesting and fun topic in computer vision.

5.1. Motivation: Recognition using line drawings

One of the big goals of this course is to learn how to recognize objects! We need to always first define the goal in computer vision! We humans can recognize an object through a line drawing. Are edges important for recognition then? YES!

A first simple approach was developed by David Lowe. His idea was to match a 3D model with parameters (3D position and orientation) with an image (which has multiple perspectives). This is how it works:

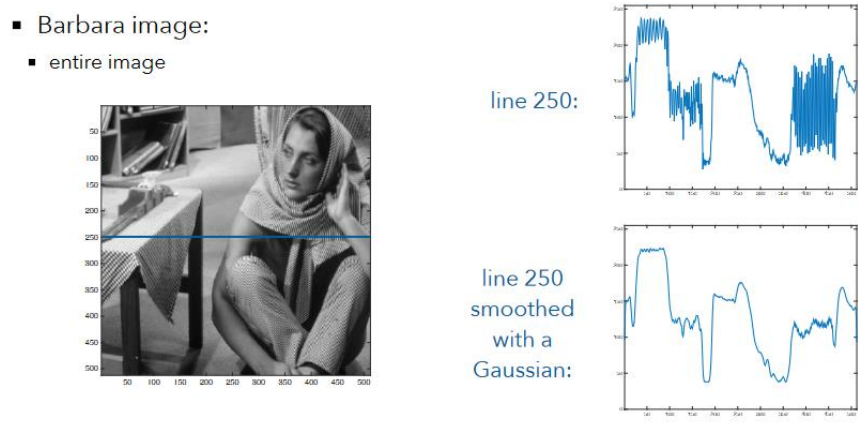
1. Filter image to find brightness changes.
2. Fit lines to the raw measurements



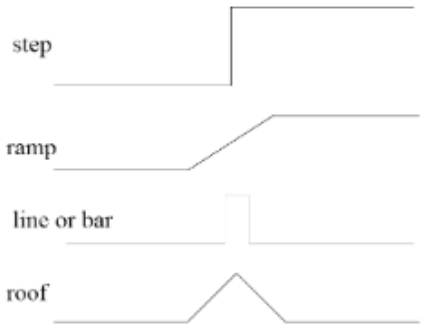
3. Project model into the image and match to lines (solving for 3D pose)

Another mostly historical approach is the matching of models (wireframe/geons/generalized cylinders...) to edges and lines. The only remaining problem to solve is to reliably extract lines & edges that can be matched to these models (wireframe, cylinders etc.).

The image scanline or intensity function plays a crucial role in edge detection! Remember images are functions and we need to find the jumps in the intensity function (local optima or minima). Edges occur at boundaries between regions of different color, intensity, or texture.

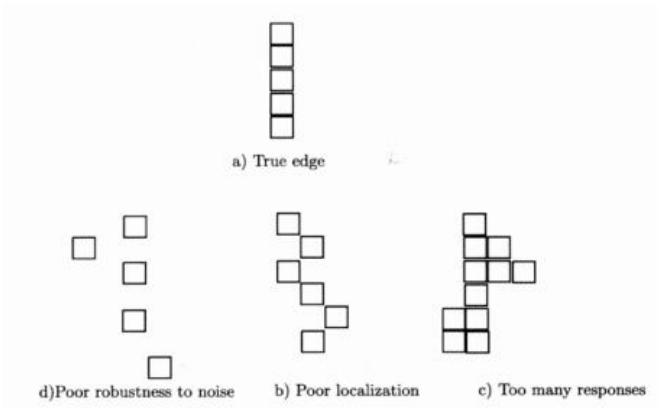


The question to ask is what are edges (1D) exactly or how do they look in the intensity function meaning how do the jumps look? Idealized edge types are steps, ramps, line or bar and roof as depicted in the next figure:



The goals of edge detection are as also visualized in the next figure:

- *Good detection*: filter responds to edge, not to noise (we find only edges and nothing else)
- *Good localization*: detected edge near true edge (the edge is detected where it is and not somewhere else, not shifted)
- *Single response*: one per edge (we want a thin edge and not a large bar)

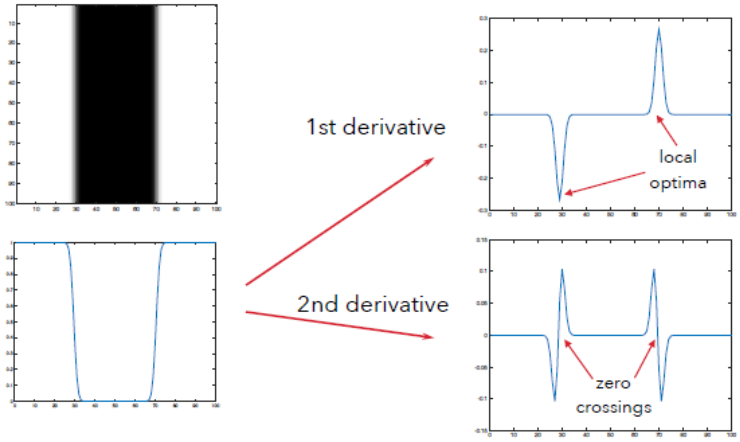


5.2. 1D Edge detection

We will talk about 1D edge detection to get a first intuition about this epic topic.

Image derivatives 1st and 2nd order

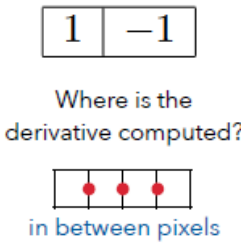
Edges correspond to fast changes in the intensity function where the magnitude of the derivative is large. The 1st derivative finds the local optima and also uses a threshold. But this is not enough it gets us thick edges but not thin edges but good first idea! The 2nd derivatives zero-crossings can also tell us the location of edges.



To compute the 1st derivative is pretty easy using *forward differences*:

$$\frac{d}{dx}f(x) = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \approx f(x+1)-f(x)$$

Note that images are discrete we can't have $h = 0$. So, the smallest step we can take is 1 pixel meaning $h = 1$. We can implement forward differences as a linear filter e.g., a *convolution* with the mask shown below (be careful depending on where we put -1 in the filter, we can have convolution or correlation). The derivative is computed in between pixels, or the derivative response is shifted half a pixel to the right!

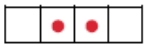


The 1st derivate can also be computed using *central differences*.

$$\frac{d}{dx}f(x) = \lim_{h\to 0}\frac{f(x+h)-f(x-h)}{2h} \approx \frac{f(x+1)-f(x-1)}{2}$$

The derivate is computed at the pixels. This solves the issue of the shifting because we want the correct location. This also can be implemented as a linear filter e.g., a *convolution* with the mask shown below:

$$\frac{1}{2} \cdot \boxed{1 \mid 0 \mid -1}$$



The 2nd derivate can also be computed using finite differences:

$$\frac{d^2}{dx^2} f(x) = \lim_{h \to 0} \frac{\frac{d}{dx} f(x) - \frac{d}{dx} f(x-h)}{h} \approx \frac{d}{dx} f(x) - \frac{d}{dx} f(x-1)$$

$$\approx f(x+1) - 2f(x) + f(x-1)$$

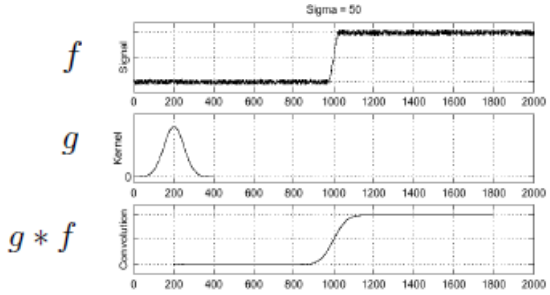
This also can be implemented as a linear filter e.g., a *convolution* with the mask shown below:

$$\boxed{1 \mid -2 \mid 1}$$

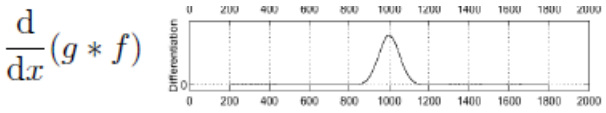
1D Edge Detection Procedure

1D edge detection is based only on the 1st derivative (finite differences): $\frac{d}{dx}(g * f)$

1. Smooth with Gaussian



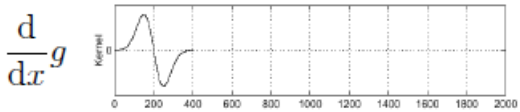
2. Calculate derivative



3. Finds its local optima

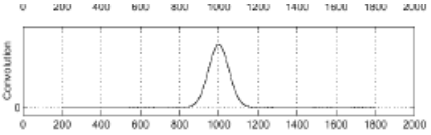
To save 1 operation we can simplify it using the association rule: $\frac{d}{dx}(g * f) = \left(\frac{d}{dx} g\right) * f$

1. Calculate derivative of Gaussian



2. Smooth with Gaussian derivative

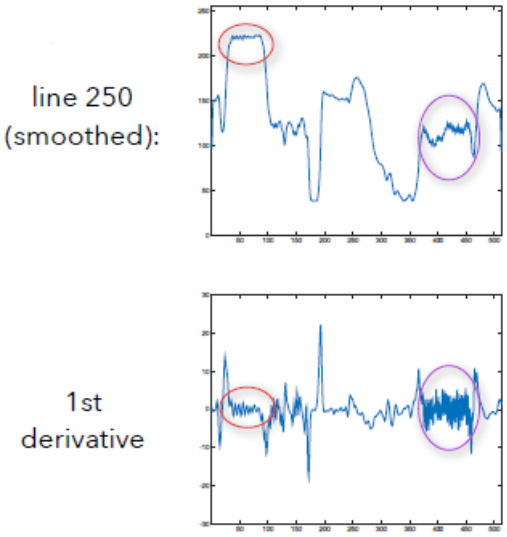
$$\left(\frac{\mathrm{d}}{\mathrm{d}x}g\right) * f$$



3. Finds its local optima

Problems

The 1st derivative amplifies small variations!



5.3. 2D Edge Detection

Algorithmically in 1D edge detection we find peaks in the 1st derivative which should be a local optimum and it should be 'sufficiently' large.

Instead, we can use 2 thresholds (called hysteresis): Use a high threshold (of absolute value) to start edge curve and a low threshold to continue them. This leads us to 2D edge detection!

Partial Derivatives

Partial derivatives in x and y direction can be expressed as:

$$\frac{\partial}{\partial x}I(x,y) = I_x \approx D_x * I \qquad \frac{\partial}{\partial y}I(x,y) = I_y \approx D_y * I$$

They are often approximated with simple filters (remember the 1D mask for finite differences):

- Box Filter:

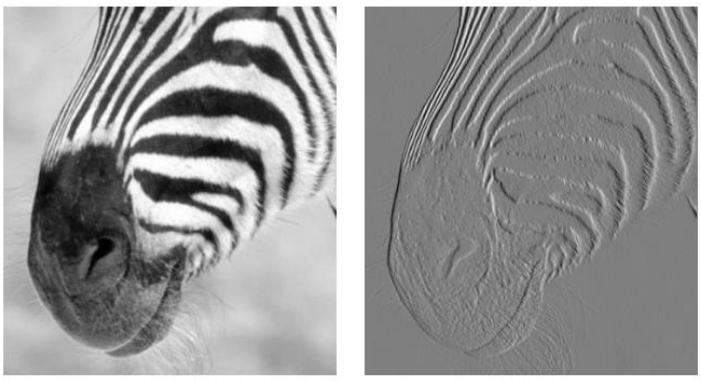
$$D_x = \frac{1}{6} \cdot \begin{array}{|c|c|c|}\hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline \end{array} \qquad D_y = \frac{1}{6} \cdot \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$$

- Sobel Filter (Gaussian Filter with gaussian smoothing in opposite direction):

$$D_x = \frac{1}{8} \cdot \begin{array}{|c|c|c|}\hline 1 & 0 & -1 \\ \hline 2 & 0 & -2 \\ \hline 1 & 0 & -1 \\ \hline \end{array} \qquad D_y = \frac{1}{8} \cdot \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array}$$

Partial Derivative Example

Question: In which direction did we perform the derivative in the image below? And did we convolution or correlation?

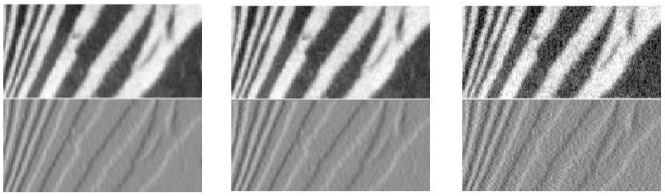


The answer is that we did a derivation in x direction because the lines in y direction are more "stronger" or “highlighted”. Fine details like the hair in x direction is not modeled so good.

We also used convolution and not correlation since we are going from light (high brightness) to dark (low brightness). Hence the derivative is negative. In a correlation the would be positive (low brightness to high brightness)

Noise, Derivatives and Gaussian Smoothing

Whenever we compute a derivative, we are basically finding changes in the intensity function. But noise also leads to changes in the signal, which isn't an edge. The derivative of the image will then be even noisier than the original image because the derivative operator in the frequency spectrum corresponds to a linear function which amplifies high frequencies. Here is an example with zero mean additive gaussian noise (increasing noise from left to right):



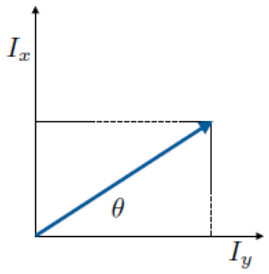
To solve this problem, we can use gaussian smoothing. This explains why gaussian smoothing is so important for edge detection not only in the direction where we don't compute the derivative but also in the direction of the derivative itself! Hence the partial derivate for example in x-direction would change to (again, we can save 1 operation by using the association rule):

$$D_x * (G * I) = (D_x * G) * I$$

Gradient

Gradient vector points in the direction of maximum change (steepest ascent) in the intensity function:

$$\nabla I = (I_x, I_y) = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right)$$



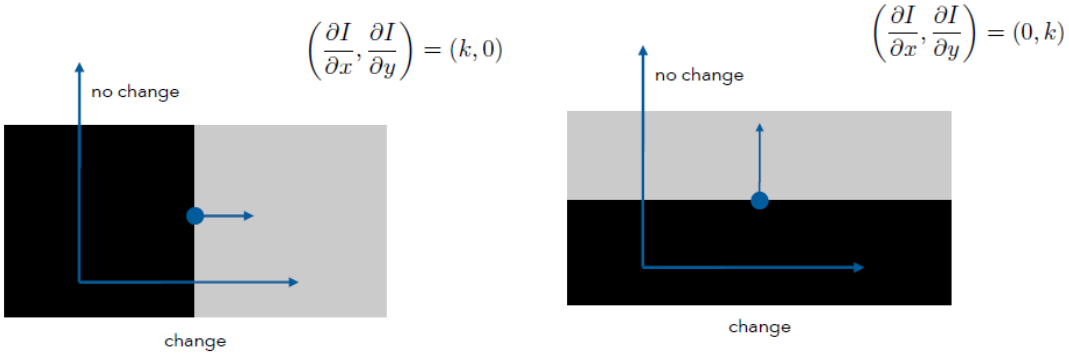
The *Magnitude of gradient* measures the edge strength:

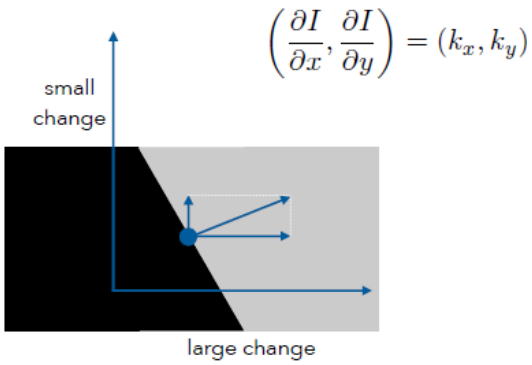
$$||\nabla I|| = \sqrt{I_x^2 + I_y^2}$$

The *direction (or orientation) of the gradient* is perpendicular to the edge:

$$\theta = \text{atan}(I_y, I_x)$$

Some examples:

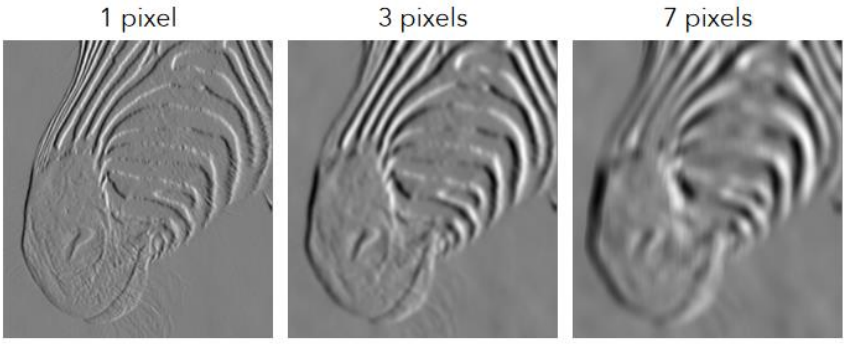




Problems with 2D edge detection

The scale of the smoothing filter affects derivative estimates and also the semantics of the edges recovered: Strong edges persist across scales!

A worst-case scenario would be if we want to detect fine structures (like hair) but at the same time also have a lot of noise in the image. If we smooth with a gaussian we will lose these fine details!



Canny's optimal edge detector

Assuming we are only doing linear filtering of an image and we have additive gaussian noise in an image then Cranny's optimal edge detector in the frame of good detection, good localization and single response is the *Derivative of Gaussian*.

But we have a tradeoff between detection and localization. More smoothing improves detection because we are removing the noise with it. But we are hurting the localization.

Anyway, Canny's optimal edge detector starts by first doing gaussian smoothing then computing the gradient using the Sobel filter. But that is not good enough, because we have thick edges as seen in the following figure:

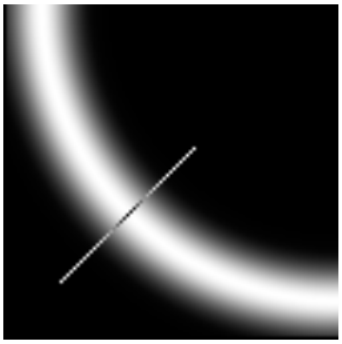
Non-Maximum Suppression

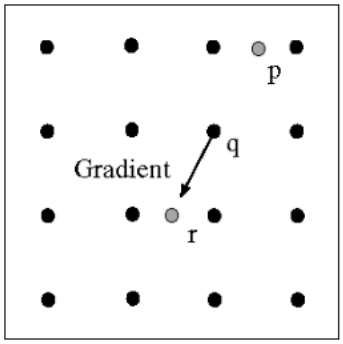
Using only the 1st derivative leads to thick edges. So how can we improve the single response! We can get thinner edges by using *non-maximum suppression*.

After getting gradient magnitude and direction, a full scan of image is done to remove any unwanted pixels which may not constitute the edge. For this, at every pixel, pixel is checked if it is a local maximum in its neighbourhood in the direction of gradient. Here is how it works:

1. Check if pixel is local maximum along gradient direction. Requires checking interpolated pixels *p* an *r*.
2. Choose the largest gradient magnitude along the gradient direction.

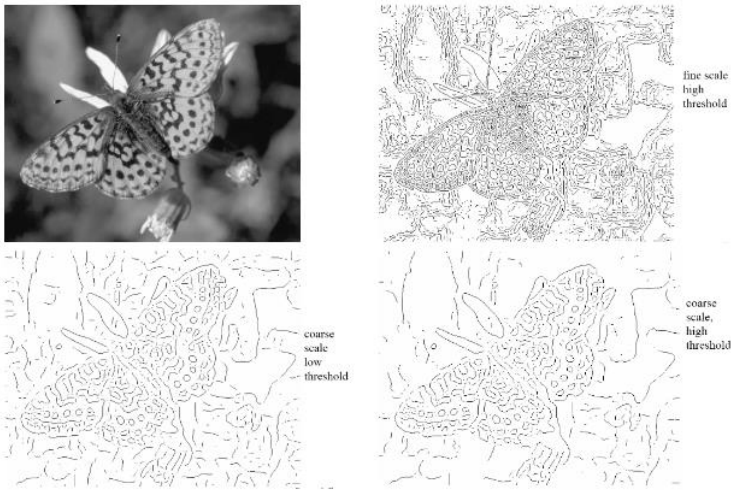
But where do we start the procedure? We *start with pixels above a certain threshold*. For each pixel repeat non maximum suppression procedure and we get the thinning!





Hysteresis

Looking at an image example below, the *edges are interrupted* at some image locations. We could lower the threshold of the gradient magnitude to detect these edges but then we would also get a lot of junk (lines that aren't edges)! So non-maximum suppression is not enough!



One way to solve this is to do *Hysteresis*: It decides which of all edges are really edges and which are not. For this, we need two threshold values, *minVal* and *maxVal*. Any edges with intensity gradient more than maxVal are sure to be edges and those below minVal are sure to

be non-edges, so discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to "sure-edge" pixels, they are part of edges. Otherwise, they are also discarded. Lets summarize this:

1. Start with high threshold to start edges. Noise will typically have lower gradient magnitudes thus not identified as an edge.

2. Label all pixels as an edge that:
   a. have a gradient magnitude above a second, lower threshold.
   b. are connected to a pixel above higher threshold (3x3 neighborhood typically)

The edge grows by 1 pixel only. Iterate step 1 and 2 for all pixels! The figure below highlights the benefits!

2D Edge Detection Full Procedure

1. Apply Gaussian Smoothing to reduce noise of imagen.
2. Calculate Gradient with the Sobel filter. Get direction and magnitude of it too.
3. Non-maximum suppression: remove any pixels that do not belong to the edge. This results in thin edges.
4. Hysteresis: It decides which of all edges are really edges and which are not.

5.4. Edge Detection using Laplacian

So far, we only used the 1st derivative for edge detection. However as already mentioned the 2nd derivative can also tell us the location of the edges (local optima) with the zero-crossing.

For that we will use the Laplacian which is defined as:

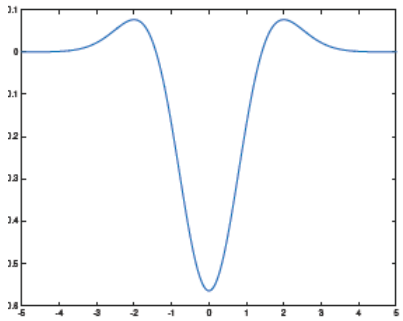
$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Again, we can apply this as a linear filter but not doing the second derivative of the image (gradient with Sobel filter) after applying gaussian smoothing but instead doing the second derivative of the gaussian first and then computing the gradient of the image using it. The Laplacian of Gaussian (LoG) is then:

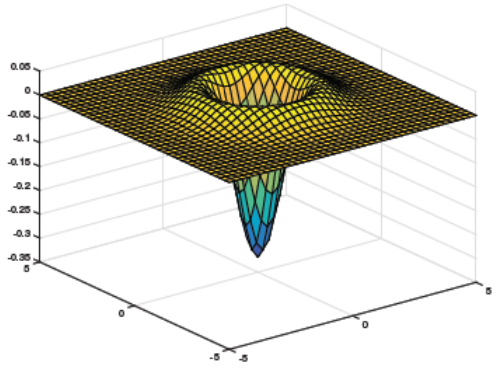
$$\nabla^2(G * I) = (\nabla^2 G) * I$$

This is how LoG looks:

- in 1D:



in 2D ("mexican hat function"):



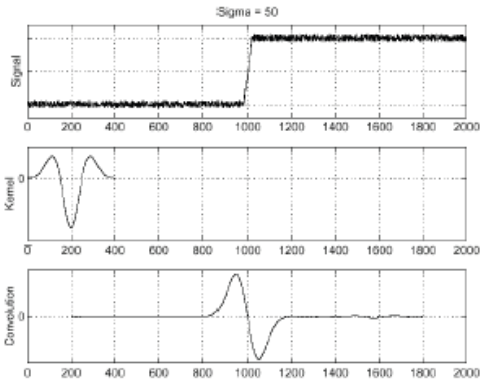
This is the procedure as explained for 1D Edeg detection with Laplacian:

LoG - Laplacian of Gaussian operator

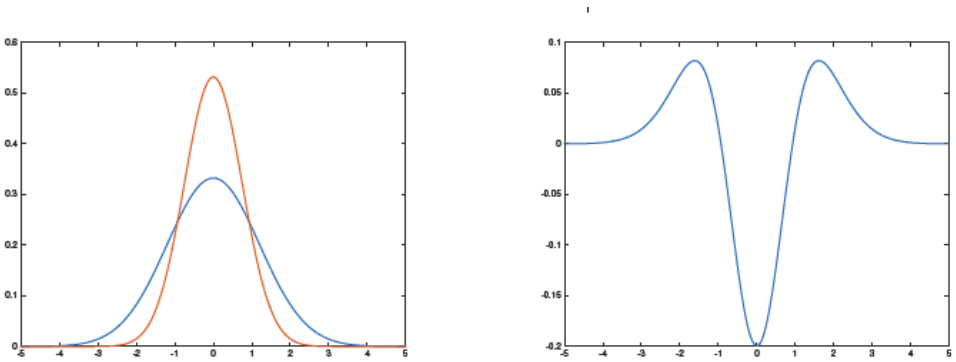
$$f$$

$$\frac{d^2}{dx^2} g$$

$$\left(\frac{d^2}{dx^2} g\right) \otimes f$$



We can approximate the LoG by a Difference of Gaussians (DoG). DoG at different scales:



$\sigma_2 = 1.6 \cdot \sigma_1$ (other choices possible)

6. Wrap-Up

You now know:

1. The properties of linear operations
2. What convolution is and its properties
3. What average and gaussian filters are
4. What separability of filters means
5. What morphological operations are
6. What different multi-scale pyramid representations are
7. What is aliasing
8. Why is edge detection important
9. How to compute derivates for edge detection

7. Questions

Is convolution a linear operator?

What is shift invariance property of convolution operation?

What is the difference between convolution and correlation?

Why do the coefficients sum to one in average filter?

How is Gaussian filter separable?

Why do Gaussian smoothing (besides down sampling) when creating a pyramid?

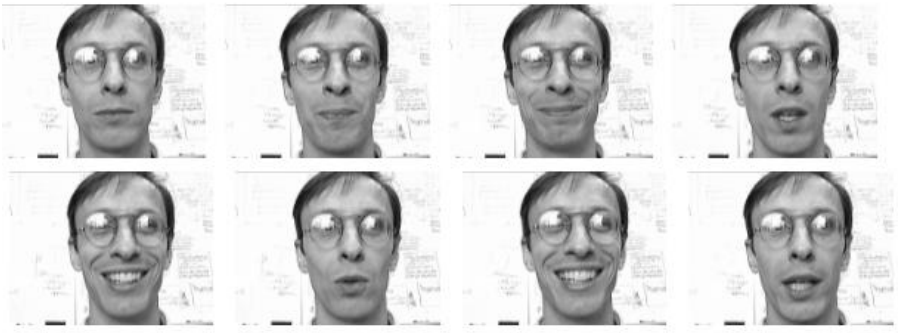
What are the goals of edge detection?

How to compute edge strength?

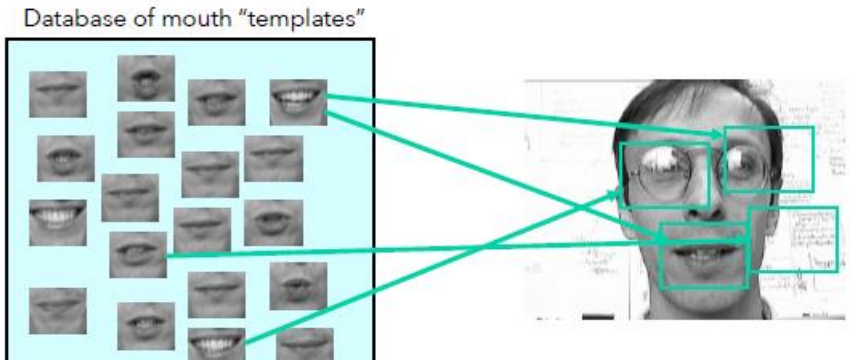
View-Based Recognition

1. Motivation

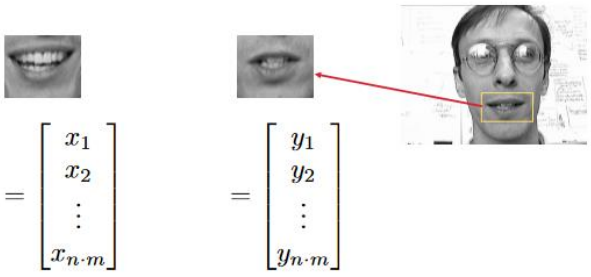
This chapter deals with view-based recognition using template methods. Simple line-based object models are not as easy as it seems. Let us look at a more constrained setting:



How can we find the mouth in an image? How can we recognize expression? The *Naive View-Based Approach* would solve this as follows: First collect templates meaning enough images of an (3D) object (e.g., different perspectives, shapes etc. of the mouth). Then search every image region (at every scale). Last but not least compare each template and choose the best match. But we need a lot of memory and computational cost für this approach!

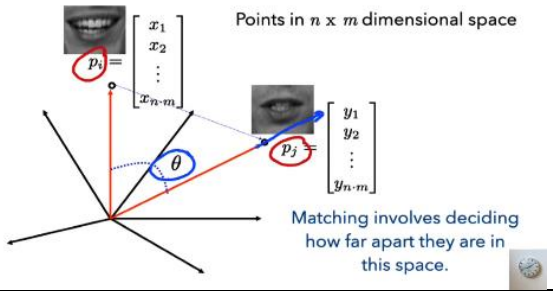


Images can be represented as vectors and points. The best way to represent pixels are vectors. From this point on we will think of images as arrays which we will reshape with standard lexicographic ordering to a vector:



e.g. standard lexicographic ordering

To define the similarity of two images we can calculate the distance or angle between the two image vectors. Matching involves deciding how far part they are in space, e.g. looking at the angle, because if we make the vector longer this just means the image will get brighter (magnitude of a vector)!



2. Template methods

Filters are Templates

Recall image filtering. Applying a filter at some pixel can be seen as taking a dot product between the image and some vector. Filtering the image is a *set* of dot products (remember a convolution is the same as dot product):

$$f[m,n] = (I * g)[m,n] = \sum_{k,l} I[m-k, n-l]g[k,l]$$

Intuition:

- *Filters look like effects they are intended to find*: Filters are designed to highlight or emphasize specific visual effects or features within an image. For example, an edge-detection filter. Though the filter itself doesn't physically resemble an edge, it's structured to identify such effects within an image.

- *Filters find effects they look like:* Filters are created to detect or uncover visual effects or features that are like their designed structure or pattern. For instance, a filter made to identify certain textures will 'find' or accentuate those textures in an image because it's constructed to respond strongly to such visual characteristics.

- *We can use filters to match a template against an image:* Filters help to highlight key features or characteristics within both the template and the larger image. By emphasizing these features, filters make it easier to compare the template with different parts of the image during matching.

Template Matching

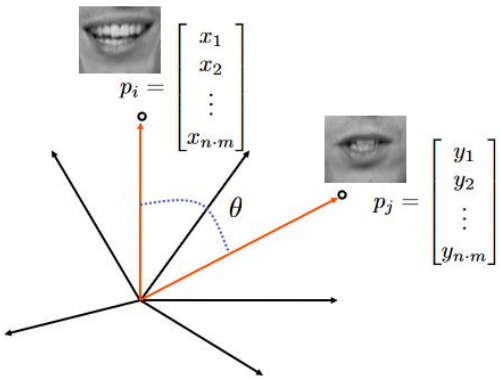
To obtain the angle of the angle of 2 vectors we can use the dot product e.g., normalized (cross) correlation, which then measures similarity between 2 vectors:

$$\boldsymbol{a}^T\boldsymbol{b} = |\boldsymbol{a}||\boldsymbol{b}|\cos(\theta)$$

where $\boldsymbol{a}^T$ is a template or filter vector, $\boldsymbol{b}$ is an image patch as vector, $\boldsymbol{a}^T\boldsymbol{b}$ is a correlation (the sum of product of "signals") and θ is the angle between the vectors. The normalized correlation is given as:

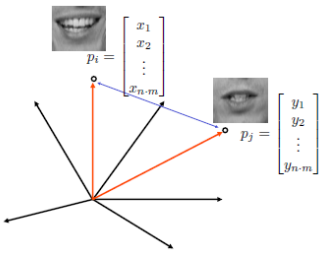
$$\cos(\theta) = \frac{\boldsymbol{a}^T \boldsymbol{b}}{|\boldsymbol{a}||\boldsymbol{b}|}$$

When we filter, we measure the angle (cosine of it, really) between the filter template and the image patch, *however scaled by the length of the vectors*.



An alternative is to minimize the Sum of Squared Differences (SSD), meaning the distance of the vectors:

$$E(I,T) = \sum_{i,j} \big(I(i,j) - T(i,j)\big)^2$$

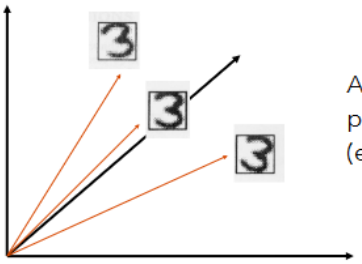


However, normalized cross correlation is better because it is brightness invariant. Image templates are the simplest view-based method. We keep an image of every object from different viewing directions, lightning conditions, etc. Then we perform nearest neighbor cross-correlation matching with images in the model database (or robust matching for clutter & occlusion). However, this has the problem of storage and computation costs become unreasonable as the number of objects increases. Additionally, it may require very large ensemble of training images. The solution would be Neural Networks (last chapter) and linear dimensionality reduction, which will cover in this chapter.

3. Linear dimensionality reduction

3.1. Why are we doing this?

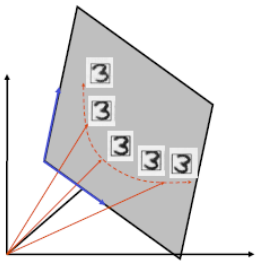
How can we find more efficient representations for the ensemble of views and more efficient methods for matching? The observation is that images are not random. Especially images off the same object (category) have similar appearance:



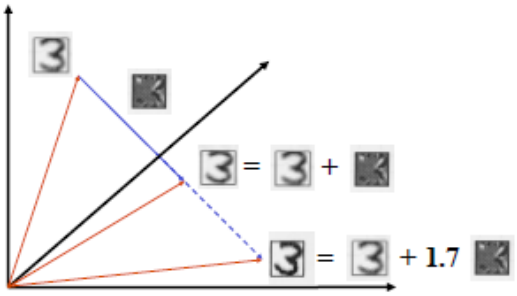
Assume images to be represented as points in a high-dimensional space (e.g., one dimension per pixel)

Our approach now is to find a lower dimensional representation that captures the *variability* in the data. Then we are *searching* using this low-dimensional model.

Question: What linear transformations of the images can be used to define a lower-dimensional subspace that captures most of the structure in the image ensemble?



Basic idea: Given that differences are structured, we can use *basis images* to transform images into other images in the same space.

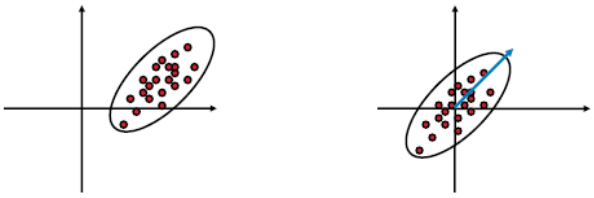


Goal: Given a datapoint (image vector) n, $\boldsymbol{x}^n \in \mathbb{R}^{\mathrm{M}}$, find a low-dimensional representation, $\boldsymbol{a}^n \in \mathbb{R}^{\mathrm{D}}, D \ll M$. We restrict this mapping $\boldsymbol{x}^n \rightarrow \boldsymbol{a}^n$ to be a linear function:

$$\boldsymbol{a}^n = \boldsymbol{B}\boldsymbol{x}^n, \boldsymbol{B} \in \mathbb{R}^{\mathrm{DxM}}$$

3.2. Intuition of how it works

If I give you the mean and one vector to represent the data, what vector would you choose? I would choose the blue vector because it captures the most data.



A datapoint n can be decomposed as follows:

$$\boldsymbol{x}^n = \sum_{i=1}^{D} a_i \boldsymbol{u}_i + \sum_{j=D+1}^{M} b_j \boldsymbol{u}_j = \tilde{\boldsymbol{x}}^n + \sum_{j=D+1}^{M} b_j \boldsymbol{u}_j$$

Where $\tilde{\boldsymbol{x}}^n$ is an approximation and $\sum_{j=D+1}^{M} b_j \boldsymbol{u}_j$ is the error. We want the D bases that minimize the mean squared error over the training data:

$$\arg\min E(u_1, \dots, u_D) = \sum_{n=1}^{N} \|\boldsymbol{x}^n - \tilde{\boldsymbol{x}}^n\|^2$$

The error can be rewritten (assuming a single basis vector for now):

$$\begin{aligned} E(\boldsymbol{u}) &= \sum_{n=1}^{N} \|\boldsymbol{x}^n - \tilde{\boldsymbol{x}}^n\|^2 \\ &= \sum_{n=1}^{N} \left\|\boldsymbol{x}^n - (\boldsymbol{u}^T \boldsymbol{x}^n)\boldsymbol{u}\right\|^2 \\ &= \sum_{n=1}^{N} \|\boldsymbol{x}^n\|^2 - 2\left(\boldsymbol{u}^T \boldsymbol{x}^n\right)^2 + \left(\boldsymbol{u}^T \boldsymbol{x}^n\right)^2 \cdot \boldsymbol{u}^T \boldsymbol{u} \\ &= \sum_{n=1}^{N} \|\boldsymbol{x}^n\|^2 - \left(\boldsymbol{u}^T \boldsymbol{x}^n\right)^2 = \sum_{n=1}^{N} \|\boldsymbol{x}^n\|^2 - (a_n)^2 \end{aligned}$$

Minimizing the error is equivalent to maximizing the variance of the projection. Assuming a mean 0 we get:

$$\frac{1}{N} \sum_{n=1}^{N} a_n^2$$

We can ensure a zero-mean projection by subtracting the mean from every data point:

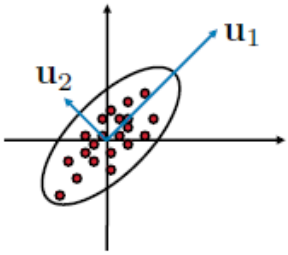
$$x^n - \bar{x}$$

Rewriting the prior decomposition of datapoint n:

$$x^n - \bar{x} = \sum_{i=1}^{D} a_i \mathbf{u}_i + \sum_{j=D+1}^{M} b_j \mathbf{u_j}$$

$$\tilde{\mathbf{x}}^n = \sum_{i=1}^{D} a_i \mathbf{u}_i + \bar{x}$$

Projecting onto $\mathbf{u}_1$ captures most of the variance and hence projecting onto it minimizes the error.



Question: Why is $\boldsymbol{u_1}$ the direction of greatest variance? Aren't the data points further away from $\boldsymbol{u_2}$ than from $\boldsymbol{u_1}$?

Answer: Your confusion here comes from misunderstanding of how Cartesian coordinates work. Remember the orthogonal distances of the points from the axis labelled $\boldsymbol{u_2}$ *are the* $\boldsymbol{u_1}$ coordinates. That is, they measure the distance parallel to the vector $\boldsymbol{u_1}$ from the origin. The variability of these distances away from the vector $\boldsymbol{u_2}$ is greater than the corresponding variance of the distances from the vector $\boldsymbol{u_1}$, but these distances are the $\boldsymbol{u_1}$ coordinates, not the $\boldsymbol{u_2}$ coordinates!

3.3. Principal component analysis

How do we find the axis of largest variance exactly now? The first principal direction $\boldsymbol{u}_1$ is the direction along which the variance of the data is maximal, i.e. it maximizes:

$$\boldsymbol{u}_1 = \arg\max_{\boldsymbol{u}} \boldsymbol{u^T C u} \;\; s.t. \boldsymbol{u^T u} = 1$$

The second principal direction $\boldsymbol{u_2}$ maximizes the variance of the data in the orthogonal complement of the first principal direction etc. Let us formalize this.

Formulation of problem

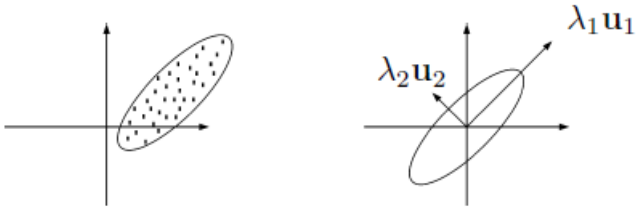
Let $\boldsymbol{X} = [\boldsymbol{x}^1, \dots, \boldsymbol{x}^N] \in \mathbb{R}^{MxN}$ be a matrix of N vectors in M-dimensional input space. Let $\boldsymbol{u} \in \mathbb{R}^M$ be a direction (a vector of length 1) in the input space. The projection of the n-th vector $\boldsymbol{x^n}$ onto the vector $\boldsymbol{u}$ can be calculated as:

$$a_n = \boldsymbol{u}^T \boldsymbol{x}^n = \sum_{i=1}^{M} x_i^n u_i$$

The goal is to find a direction $\boldsymbol{u}$ that maximizes the variance of the projections of all input vectors $\boldsymbol{x^n}, n = 1 \dots N$.

The mean of the projection is the projection of the mean:

Goal: Find the so-called principal directions, and the variance of the data along each principal direction. λ_i is the marginal variance along the principal direction u_i



My Basic understanding

Image we only have two images (we can't visualize this if we have more than three images!)

Each axis represents one of the two images. The datapoints are the pixel values of the images.

First step is to calculate the average pixel value of image 1 and average pixel value of image 2. This gives us the center of the data which we will plot. From this point we don't need to look at the pixel data anymore and we will focus only on the graph!

Next shift data so that the average pixel value is at the origin of the graph (0,0). Note this has no effect on the data!

Next try to fit a line to the data that goes through the origin. The best line found has the direction (slope) of the principal direction u_1. u_2 is then just perpendicular to it!

Interest Points & Image Features

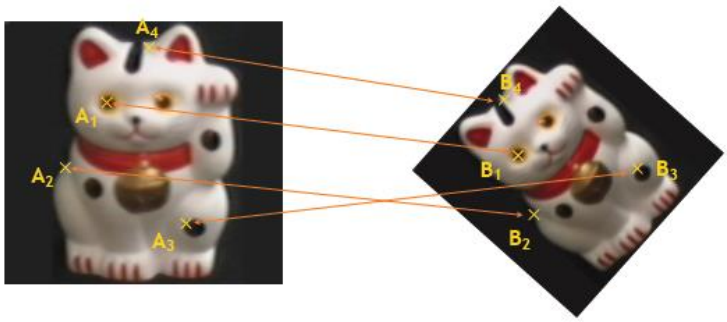
1. Motivation

Recognizing object categories is hard (more later). For now, let's look at a easier problem: Let's look at two images depicting the very same object/scene. How do we tell they are the same? How do we find corresponding points? Which points should we even look at? This gets even harder if one of the images has a different viewpoint.

The general idea of detecting corresponding points is simple:

1. Find important. i.e., "interesting" points in the images
2. Find which interest points correspond to one another.

The next figure illustrates this approach. We will cover the second point later for now let's focus on interest points.

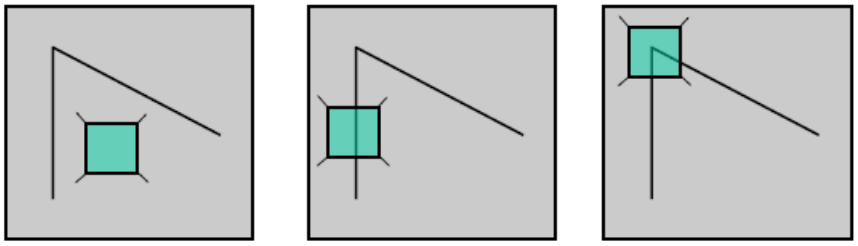


2. Interest Point Detection

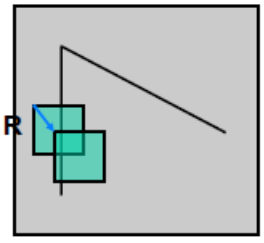
In this section we will derive interest point detection and cover the Harris Operator as well as other interest point detectors.

2.1. Derivation

We are starting the local measure of unique interest points (def.: intersection of two or more edge segments or the point at which the direction of the object's border rapidly changes) by looking how a window within an image change when we shift it. Shifting the window in any direction causes a big change



Consider shifting the window by a vector $R = \binom{u}{v}$.



To obtain how the pixels in R change we can compare each pixel before and after by summing up the squared differences (SSD), which defines an SSD "error" or also called energy of $E(u,v)$:

$$E(u,v) = \sum_{(x,y)\in R} \big(I(x+u, y+v) - I(x,y)\big)^2$$

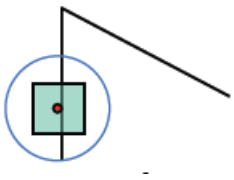
We can approximate this using a Taylor Series approximation:

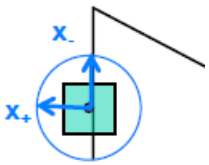
$$I(x,y) + u\frac{\partial}{\partial x}I(x,y) + v\frac{\partial}{\partial y}I(x,y) + \in (u^2, v^2)$$

Where $\in (u^2, v^2)$ is an approximation error. Hence, we get:

$$\begin{aligned}
E(u,v)E(u,v) &= \sum_{(x,y)\in R} \big(I(x+u, y+v) - I(x,y)\big)^2 \\
&\approx \sum_{(x,y)\in R} \left(\frac{\partial}{\partial x}I(x,y) + v\frac{\partial}{\partial y}I(x,y)\right)^2 \\
&= \sum_{(x,y)\in R} \big(u \cdot I_x(x,y) + v \cdot I_y(x,y)\big)^2 \\
&= \sum_{(x,y)\in R} (u,v)\nabla I(x,y)\nabla I(x,y)^T \begin{pmatrix} u \\ v \end{pmatrix} \\
&= (u,v)\left[\sum_{(x,y)\in R} \nabla I(x,y)\nabla I(x,y)^T\right]\begin{pmatrix} u \\ v \end{pmatrix} \\
&= (u,v)\boldsymbol{H}\begin{pmatrix} u \\ v \end{pmatrix}
\end{aligned}$$

Where $\nabla I = \begin{pmatrix} I_x \\ I_y \end{pmatrix}$ and $\boldsymbol{H}$ is called the structure tensor. Now suppose you can move the green center of the green window to anywhere on the blue unit circle.

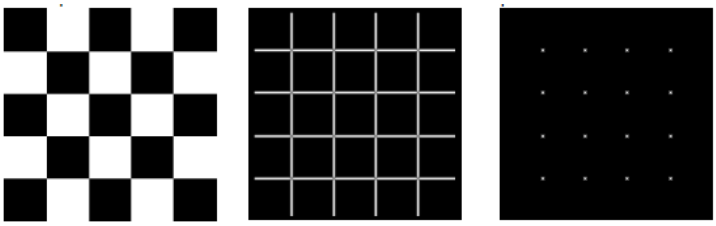




Which directions will result in the largest and smallest energy values E? We can find these directions by looking at the eigenvectors of $\boldsymbol{H}$. Recall the homogeneous least squares! The Eigenvalues of $\boldsymbol{H}$ define shifts with smallest and largest change (E value):

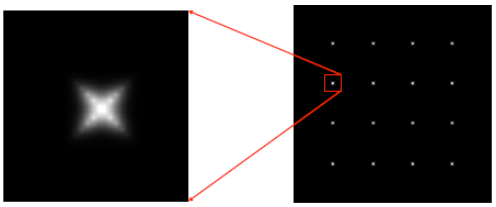
$$\begin{aligned}
Hx_+ &= \lambda_+ x_+ \\
Hx_- &= \lambda_- x_-
\end{aligned}$$

where x_+ is the direction of largest increase in E, λ_+ is the amount of increase in direction x_+, x_- is the direction of smallest increase in E and λ_- is the amount of decrease in direction x_-. But how are $x_+, \lambda_+, x_-, \lambda_-$ relevant for interest point detection? We want $E(u,v)$ to be large for small shifts in all directions. The minimum of $E(u,v)$ should be large over all unit vectors $\begin{pmatrix} u \\ v \end{pmatrix}$ and is given by the smaller eigenvalue λ_- of $\boldsymbol{H}$. The next figure illustrates this.



Here is now how interest point detection works:

1. Compute the gradient at each point in the image
2. Create the structure Tensor $\boldsymbol{H}$ from entries in the gradient
3. Compute the eigenvalues
4. Find points with large response: $\lambda_- > threshhold$
5. Choose those points where λ_- is a local maximum as interest points:



Harris Operator/Harris Corner Detector

λ_- is a variant of the Harris operator for interest point detection:

$$f = \lambda_1\lambda_2 - \alpha(\lambda_1 + \lambda_2)^2 = \det(\boldsymbol{H}) - \alpha \cdot trace(\boldsymbol{H})^2, \alpha \in [0.04, 0.15]$$

where α is a parameter that controls the trade-off between the determinant (product of the eigenvalues) and the trace (sum of the eigenvalues). The trace is the sum of the diagonals, i.e., $trace(H) = h_{11} + h_{22}$. It's very similar to λ_- but less expensive (no square root). It is called the "Harris Corner Detector" or "Harris Operator". There are lots of other detectors, but this is one of the most popular.

Hessian Determinant

We can also use the hessian determinant to find interest points:

$$\boldsymbol{H}(I) = \begin{pmatrix} I_{xx} & I_{xy} \\ I_{yx} & I_{yy} \end{pmatrix} \rightarrow \det(\boldsymbol{H}) = I_{xx}I_{yy} - I_{xy}^2$$

Problems of Harris and Hessian detectors and solutions

Interest point detection so far is based on Harris or Hessian detectors. It finds interesting, i.e., discriminative points (Harris detector was the "de-facto" standard for a long time) used for recognition, correspondence for stereo, sparse optical flow/motion, etc. We still need to clarify the goals of interest point detection. This boils down to distinctiveness vs. invariance to transformation. Harris & Hessian find distinctive points, but they are **not** invariant to *scale*, *affine* and *projective* transformations!

Let's look at examples of different geometric transformations: (1) original, (2) similarity transformation (translation, image plane rotation, scaling), (3) projective transformation.

(1) (2) (3)

At least the Harris interest point detector is rotation invariant because rotating the image leads to rotation of the gradient.

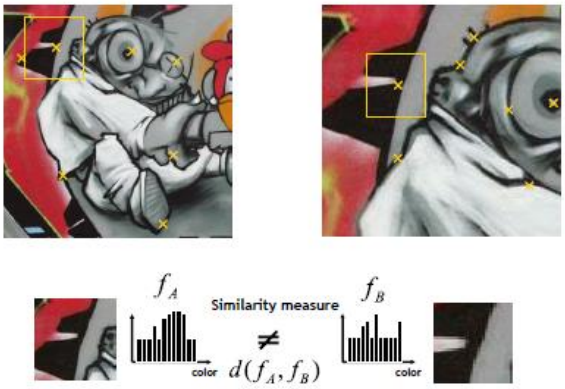
∇I R ∇I

The structure tensor then is : $\boldsymbol{H} = \sum_{(x,y)} \boldsymbol{R}\nabla I(x,y)(\boldsymbol{R}\nabla I(x,y))^T = \sum_{(x,y)} \boldsymbol{R}\nabla I(x,y)\nabla I(x,y)^T\boldsymbol{R}^T = \boldsymbol{RHR}^T$. Rotation invariance follows from: $\det(\boldsymbol{RHR}^T) = \det(\boldsymbol{R})\det(\boldsymbol{H})\det(\boldsymbol{R}^T) = \det(\boldsymbol{H})$ and $trace(\boldsymbol{RHR}^T) = trace(\boldsymbol{H})$.

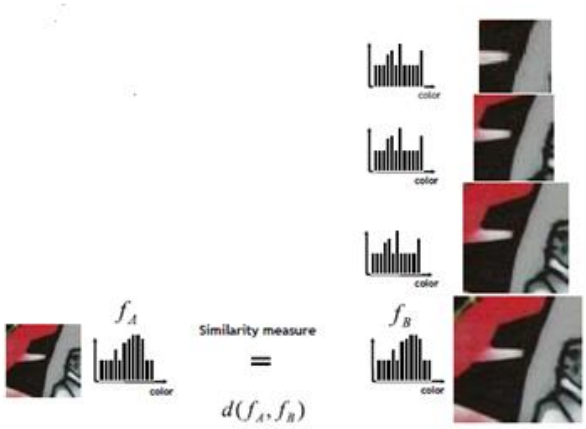
One scale invariant method for feature extraction is for example the Harris Laplace Detector. This allows matching images of different scales and automatic scale selection. Other popular interest point detectors are SIFT and SURF.

2.2. Scale Invariant Interest Point Detection

To match image patches of different scales we first detect interest points, extract patches, and then compute feature descriptors (def.: A feature descriptor is a method that extracts the feature descriptions for an interest point) as illustrated in the next figure.



However, it's impossible to match different histograms due to different patch content. SO the patch should contain the same part of the image. We can find the correct size by comparing feature descriptors while varying the patch size:



This computationally inefficient but possible for matching and computationally prohibitive for retrieval in large databases as well as recognition. An alternative approach is that the detector finds *location* and *scale* of interest points. This allows independent automatic scale detection by finding the characteristic scale of an interest point.

The key idea is to find a scale that gives local maximum of some criterion. But what criterion should we use? We will answer that soon. Note that the level of details decreases monotonically as the scale of Gaussian smoothing is increased. One approach to solve this is Harris Laplace.

Harris Laplace

Recall the Harris corner detector $f = \det(\boldsymbol{H}) - \alpha \cdot trace(\boldsymbol{H})^2$ with $\boldsymbol{H} = \sum_{(x,y)\in R} \mathrm{w(x,y)} \nabla I(x,y) \nabla I(x,y)^T$. The $w(x,y)$ is a spatially varying weight, which weighs the contribution of the pixels within the region. In the simple base definition of the structure tensor (lecture 6, slide 72), the weight is equal across all pixels (and hence $w(x,y)$ is left out). Typical implementations of the structure tensor use a Gaussian weighting function that gives a higher weight to central pixels.

Harris Laplace works like this: First detect Harris points for multiple scales $\sigma_1, \sigma_2, \sigma_3, \sigma_4, \ldots$ This results in thousands of interest points which is why we need a criterion. We are selecting

points which maximize the Laplacian. Given a point (x, y, σ_n), we are keeping the point if the following conditions both are valid:

$$\Big(L_{xx}(\sigma_n) + L_{yy}(\sigma_n)\Big) > \Big(L_{xx}(\sigma_{n-1}) + L_{yy}(\sigma_{n-1})\Big)$$
$$\Big(L_{xx}(\sigma_n) + L_{yy}(\sigma_n)\Big) > \Big(L_{xx}(\sigma_{n+1}) + L_{yy}(\sigma_{n+1})\Big)$$

otherwise, we reject the point.

SIFT

SIFT (Scale Invariant Feature Transform) detects local maxima (in space and scale) in Laplacian pyramid (similar to HarLap). This leads to many interest points. We then "verify" interest points with a Hessian-based criterion, which ensured the eigenvalues are not too dissimilar:

$$\frac{\det(\boldsymbol{H})}{trace^2(\boldsymbol{H})} \geq \frac{r}{(r+1)^2}$$

Where $\boldsymbol{H} = \begin{pmatrix} I_{xx} & I_{xy} \\ I_{yx} & I_{yy} \end{pmatrix}$. SIFT is a blob rather than corner detector. We will cover more stuff about SIFT later!

SURF

SURF (speeded up robust features) uses the local scale space maxima of Hessian determinants.

$$\det\big(H(x, y, s\sigma)\big) = s^2(I_{xx}I_{yy} - I_{xy}^2)$$

SURF too is a blob rather than a corner detector!

2.3. Evaluation of Interest Points

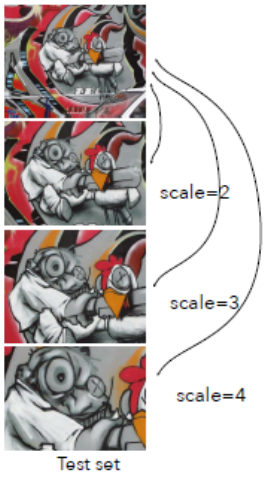
Which interest points work when? We can us the repeatability rate which is a percentage of corresponding points:

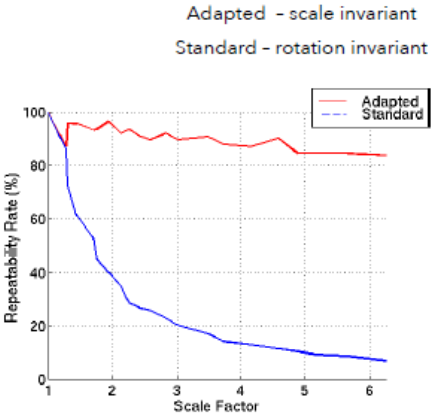
$$repeatability = \frac{\#correspondences}{\#detected} \cdot 100\%$$

Two points are corresponding if:

$$\frac{A \cap B}{A \cup B} > T = 60\%$$

Where we look at the intersection over union error a.k.a. the Jaccard index. Here is an example where we compare adapted scale invariant and standard rotation invariant detectors:





3. Local Descriptors/Features

Recap the motivation chapter. The general idea of detecting corresponding points is simple:

1. Find important. i.e., “interesting” points in the images (Done)
2. Find which interest points correspond to one another. (Now)

How do we decide if two image regions match? The general idea is:

1. Compute local descriptors/features (describing the region around interest point
2. Compare them (using some distance)

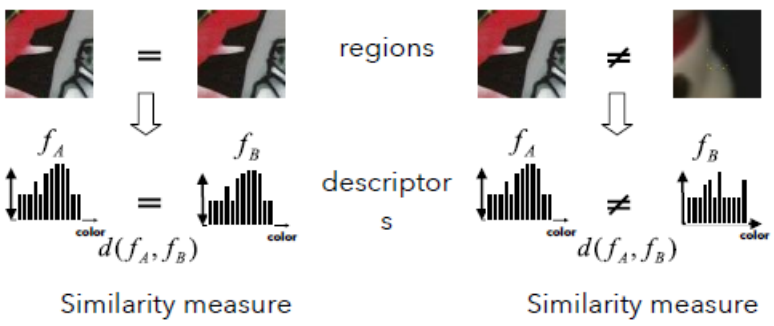
Note we already introduced this for the Harris Laplace Detector if you paid attention.

3.1. Important properties of local descriptors

In general, the detector finds location, scale, and shape of interest regions. The local descriptors are computed for interest regions. Let’s look at some important properties of local descriptors such as distinctiveness, invariance, robustness, dimensionality etc.

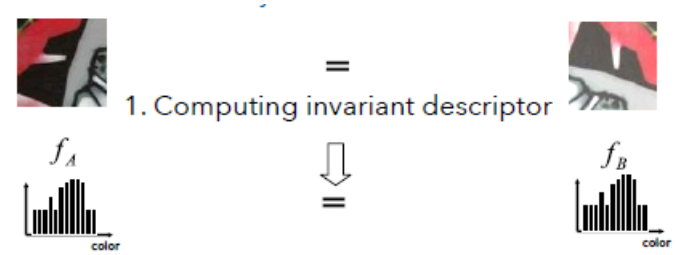
Distinctiveness

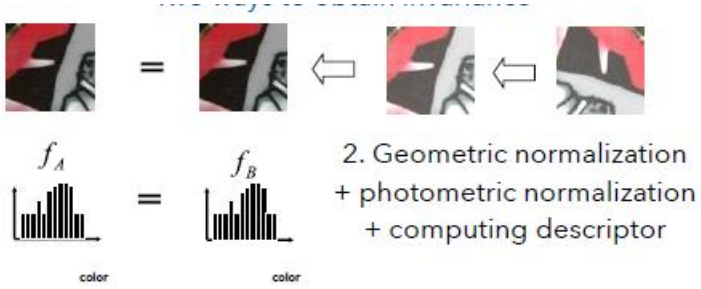
Visually similar regions should have similar descriptors and different regions should have different descriptors.



Invariance

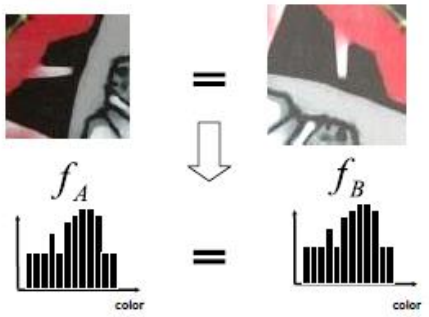
Visually similar regions should have similar descriptors despite the transformation (geometric, photometric) i.e., rotation, brightness. There are two ways to obtain invariance. Computing invariant descriptor directly or first doing geometric normalization then photometric normalization and at the end computing the descriptor.





Robustness

Visually similar regions should have similar descriptors despite noise (geometric, photometric).



Dimensionality

Descriptors should be low dimensional, i.e., small number of histogram bins. Additionally, they should aim for efficiency (large databases) and have a generalization property.

3.2. Local Descriptors

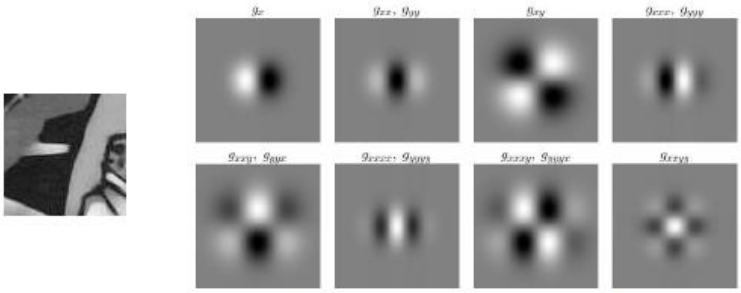
Local descriptors are e.g., image patches (Vector of pixels), Filter bank, SIFT and shape context.

Image Patches

A vector of pixels which is only invariant when computed on normalized patches. It is distinctive and easy to implement but is high dimensional and not very robust. To match descriptors the Euclidian distance or cross correlation is used.

Bank of Filter responses

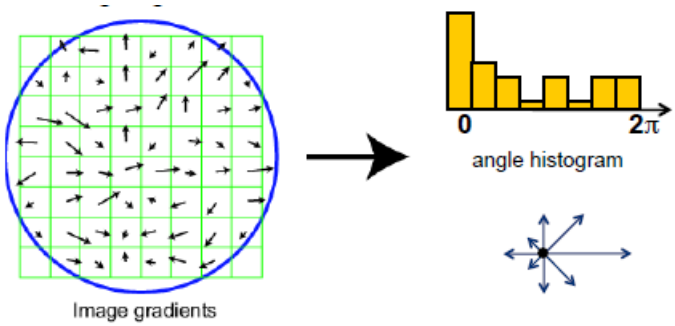
Invariant only when computed on normalized patch. We are basically comparing responses of gaussian derivatives.



Scale Invariant Feature Transform (SIFT)

Uses an orientation histogram. The basic idea is as follows:

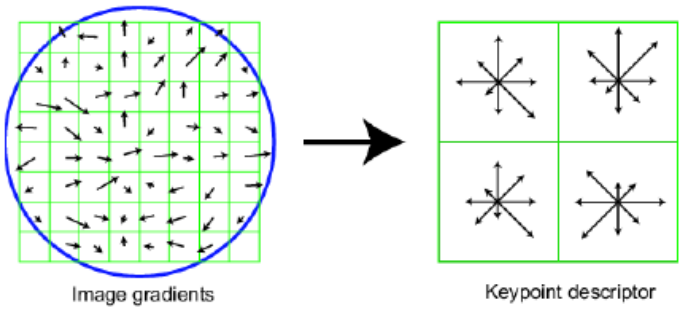
1. Take 16x16 square window around detected interest point
2. Compute edge orientation (angle of the gradient - 90°) for each pixel
3. Throw out weak edges (threshold gradient magnitude)
4. Create histogram of surviving edge orientations



The full version works as follows:

1. Divide the 16x16 window into a 4x4 grid of cells (2x2 case shown below)
2. Compute an orientation histogram for each cell

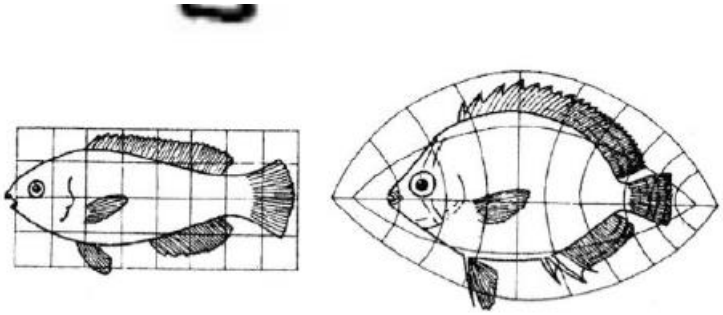
3. 16 cells * 8 orientations = 128-dimensional descriptor



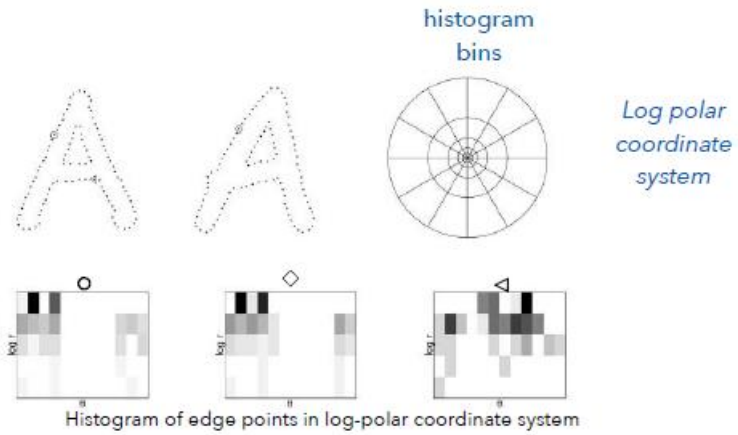
SIFT is an extraordinarily robust matching technique and can handle changes in viewpoint (up to about 60 degrees out of plane rotation), significant changes in illumination (sometimes even day vs. night; below) and is fast and efficient as well as can run in real time.

Shape Context

The motivation or goal is to cope with complex geometric transformations where we cannot compare pixel to pixel. For example:

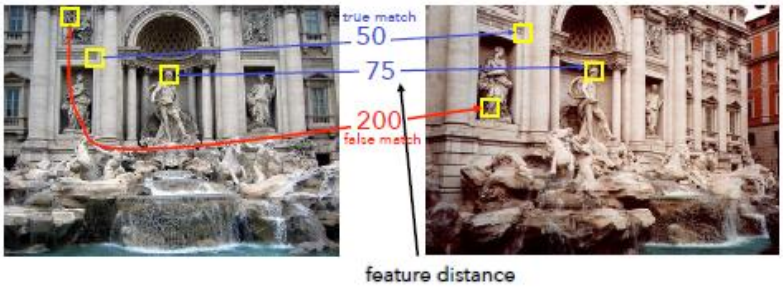


Represent histogram of edge locations: Run edge detector, make histogram insensitive to minor shape variations. Main idea:

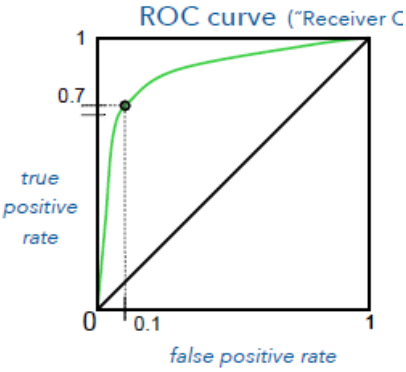


3.3. Evaluation

How can we measure the performance of a feature matcher (descriptor and a distance/classifier)? We have to look at the feature distance. The distance threshold affect performance:



The goal is to maximize true positives which is the number of detected matches that are correct and minimize false positives which is the number of detected matches that are incorrect. How should we choose the threshold in both settings? For that we can use ROC (Receiver Operator Characteristic) curves which are generated by counting the number of current/incorrect matches for different thresholds. We want to maximize the area under the curve (AUC).



$$true\ positive\ rate = \frac{\#true\ positives}{\#matching\ features\ (positives)}$$

$$false\ positive\ rate = \frac{\#false\ positives}{\#unmatched\ features\ negatives)}$$

4. Wrap up

You know now:

- Different methods for identifying interest points, including those that are invariant to scale and affine transformations
- How to evaluate different methods for interest point detection
- Properties of local descriptors
- How local descriptors like SIFT and Shape context work
- How to evaluate the performance of local descriptors

5. Questions

Which interest points detection algorithm have a better repeatability rate?

How do you compute the canonical orientation of image patches ?

Why does the SIFT descriptor divide the 16x16 window into a grid of 4x4 cells?

Why does log-polar coordinate system work better than the Euclidean coordinate system for shape context descriptors?

How do we evaluate the performance of different feature matchers?

Single & Two-View Geometry

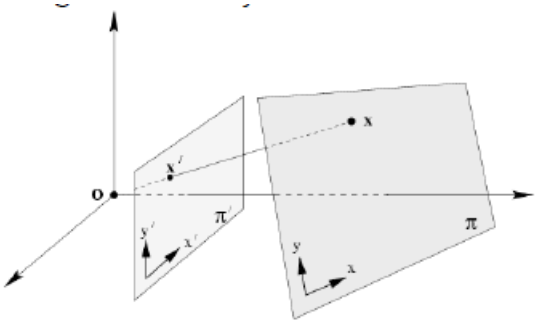
1. Homography

Why is projecting a planer object useful? To learn how undistort perspective images of planer structures and to map from one plane to another.

Geometrically it means how does a pinhole camera map a plane. Algebraically this means show how a projection matrix acts on co-planar points. To give you one example think about taking picture of a building from a side perspective. The question to solve would be how

1.1. Perspective image of a plane

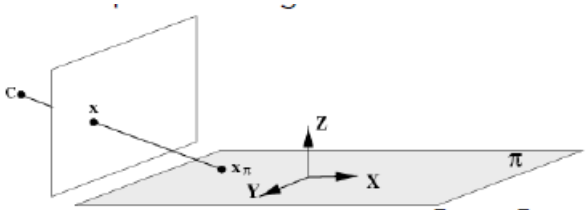
Let's start with the geometric intuition first. Projection with a straight ray from a 2D plane (in the real world) π to another 2D plane (image plane of the camera) π'. There is no need for dimensionality reduction because the ray intersects each plane exactly once. The inverse projection along the same ray bundle is possible now, but not before for the 3D to 2D mapping.



Remember a pinhole camera is represented by a (3x4) homogenous projection matrix P:

$$x = PX = K[R|-R\tilde{C}]X$$

To project a plane, we will set the z-axis of its coordinate system to zero: $Z = 0$. This trick allows us to simplify the projection matrix by removing the third column:



$$x = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ 0 \\ 1 \end{bmatrix} =$$

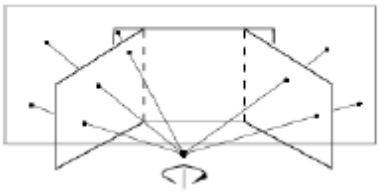
$$= \begin{bmatrix} p_{11} & p_{12} & p_{14} \\ p_{21} & p_{22} & p_{24} \\ p_{31} & p_{32} & p_{34} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ 1 \end{bmatrix} = Hx_\pi$$

The mapping from a 2D plane to the image plane is given by (3x3) matrix $\boldsymbol{H}$, which is called a *homography* (or projectivity). $\boldsymbol{H}$ is the most general projective 2D transformation and maps a square to a general quadrangle. The homography has 8 degrees of freedom and is a one-to-one mapping, hence *invertible*! The projection matrix for the 3D-2D mapping was not invertible. The (inverse) relationship is then given by as already mentioned above:

$$x = Hx_\pi \leftrightarrow x_\pi = H^{-1}x$$

1.2. Views from same camera centre

The relation between two images, if the camera only rotates (baseline=0) are related by a homography under the conditions that the projection rays are identical and the ray maps directly from one image plane to the other. This basically means two images are taken from the same viewpoint and are a plane. Consider x and x' to be two points of two images that satisfy these conditions then the relationship is (in homogenous coordinates):



$$x = K[I|0]X = K\tilde{X} \qquad \tilde{X} = K^{-1}x$$
$$x' = K'[R|0]X = K'R\tilde{X} \qquad x' = \boxed{K'RK^{-1}}x$$
$$x' = Hx$$

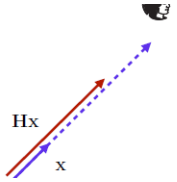
This is used for example in panorama stitching in which we estimate the homography between two overlapping images, transform one into the image plane of the other.

2. (Homography) Estimation

The homography has 8 unknown parameters. Per point we have two linearly independent equations (homogenous coordinates):

$$\lambda x' = Hx, x' \propto Hx$$

we know both vectors are pointing in the same direction, but we don't know the scale of these vectors which is why we are using λ as a scale variable. To solve this system of equations we need at least 4-point correspondences! The more the better. Writing the relationship to a cross product (because we know both vectors are pointing in the same direction) leads to:



$$\mathbf{x}' \propto \mathbf{H}\mathbf{x} \quad \rightarrow \quad \mathbf{H}\mathbf{x} \times \mathbf{x}' = 0$$

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \mathbf{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & x & y & 1 & -xy' & -yy' & -y' \\ -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = 0$$

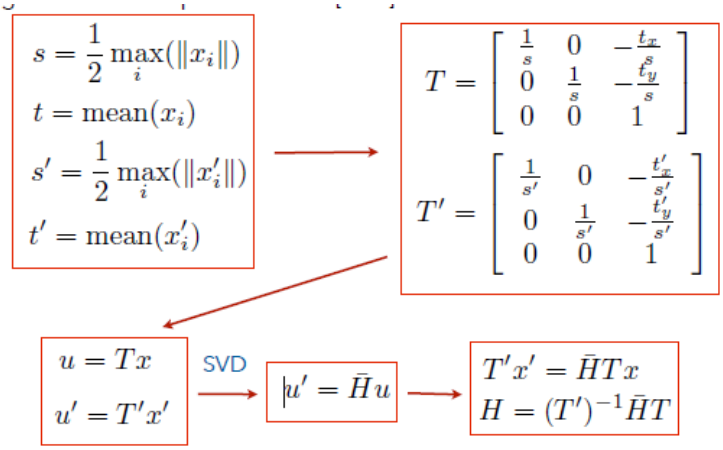
This homogenous linear equation system can be solved by enforcing a norm constraint using SVD (Singular Value Decomposition):

$$Ah = 0, s.t. \|\boldsymbol{h}\| = 1$$
$$SVD: A = USV^T$$

The solution is given by the last right singular vector $h = \begin{bmatrix} v_{19} \\ v_{29} \\ \vdots \\ v_{99} \end{bmatrix}$.

2.1. Numerical conditioning of coordinates

We have a problem with *numerical stability* now. The coefficients of an equation system should be in the same order of magnitude so as not to lose significant digits in pixels: $xx' \sim 10^6$. We can solve this with *conditioning* by *scaling and shifting* points to be in in the range $[-1,1]$. We can exploit the inverse transform of the estimated homography. This is also a general recommendation, not only for homography. This is how this would work:



where T and T' are matrices that are used to scale the points of two different image planes.

2.2. Robust fitting with RANSAC

Some correspondences may be wrong. Using false correspondences will corrupt the estimate. To solve this problem, we need a method to get rid of matching errors. Again, this can be applied generally, not only for homography. This is where RANSAC (random sample consensus) comes into play.

First randomly pick a minimal set of points, construct the model (a line) and measure the support (count number of points with small error). Repeat the procedure for a number of iterations and keep the best hypothesis. Algorithm 1 summarizes this.

Algorithm 1 explains it. However, we cannot try all possible samples meaning doing a of iterations due to computationally cost etc. Assume we have a conservative guess of the fraction of "inliers" (correct points): Say 20% of the correspondences are wrong then how many random samples do we have to pick, to "almost certainly" find the right homography? Assume we have a conservative guess of the fraction w of "inliers" (correct points) then:

probability of picking an uncontaminated sample (requiring d data points)	w^d
probability of seeing no uncontaminated sample in k trials	$1 - z = (1 - w^d)^k$
required k to be z (say, 99%) sure there is an uncontaminated sample	$k = \frac{\log(1-z)}{\log(1-w^d)}$

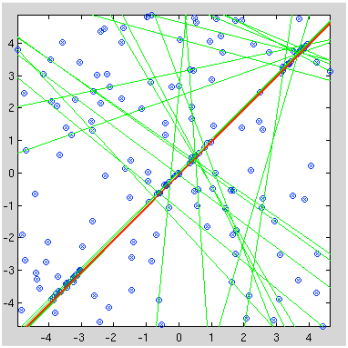
Depending on the dimensionality of the data and big the fraction of uncontaminated samples the number of iterations is:

	proportion of inliers (w)						
d	95 %	90 %	80 %	75 %	70 %	60 %	50 %
2	2	3	5	6	7	11	17
3	3	4	7	9	11	19	35
4	3	5	9	13	17	34	72
5	4	6	12	17	26	57	146
6	4	7	16	24	37	97	293
7	4	8	20	33	54	163	588
8	5	9	26	44	78	272	1177

The more dimensions there are the more iterations we need to do. This is an immensely successful brute-force method. But even the best hypothesis may be a rather bad fit. The solution would be to always refit to all inliers. Algorithm 2 summarizes the homography estimation using RANSAC.

Algorithm 1: RANSAC

Input: datapoints (e.g., matching interest points)
Output: mathematical model
1: **iterate**
2: randomly pick a minimal set of points
3: construct the model (a line, a homography, ...) from the points
4: measure the support (count number of points with small error)
5: **return** mathematical model

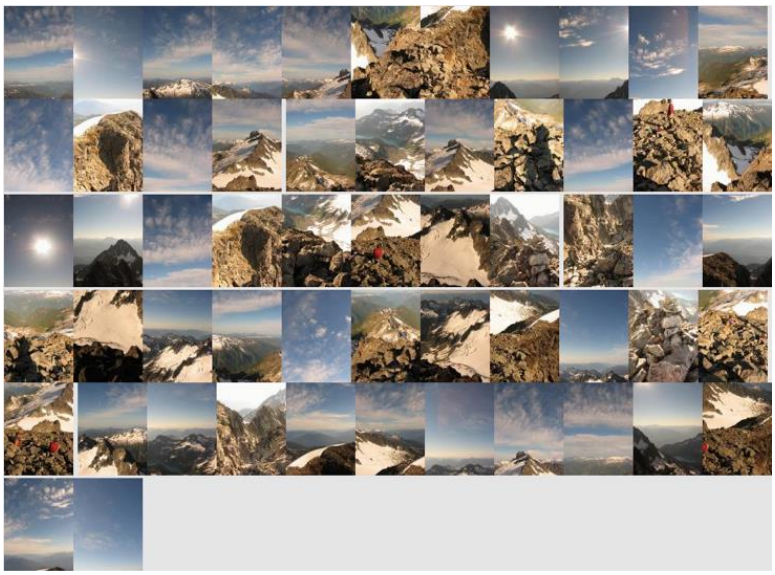


Algorithm 2: Estimating the Homography using RANSAC

Input: 2 images from a rotating camera (same viewpoint)
Output: Homography ***H***
1: **condition** all images
2: **iterate**
3: pick 4 correspondences
4: build equations, estimate homography
5: transform all points x measure distance to all points x' (in non-homogeneous coordinates!)
6: count inliers (scale down threshold too!)
7: re-estimate final homography
8: **undo** conditioning

3. Application: Panorama or Auto Stitching

Given a collection of unordered photographs



Automatically create a panorama by:

1. Find images related by a homography
2. Extract relative rotation
3. Remap to a sphere
4. Blend colours along seams

4. Wrap-Up

You know now:

- What Homography is and how it is helpful for projection from one 2D plane to another (or helpful for relating images of planes)
- How to estimate the Homography matrix H
- How to solve numerical issues when estimating H
- Robust Estimation using RANSAC algorithm
- RANSAC for Homography Estimation
- Different methods for estimation 3D geometry

5. Questions

How can we obtain the homography matrix H from the projection matrix P?

What are some applications of a homography?

Why are 4-point correspondences required for estimation of a homography?

How do we scale the equation system matrix so that all values lie between -1 and +1?

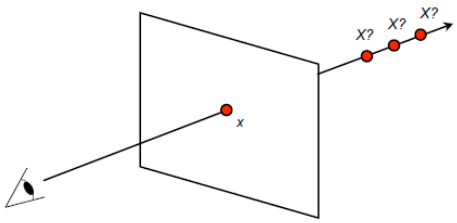
What condition do you use for estimating the Homography matrix H ?

How many numbers of trials are required for estimating matrix H with 30% inliers and z=99%?

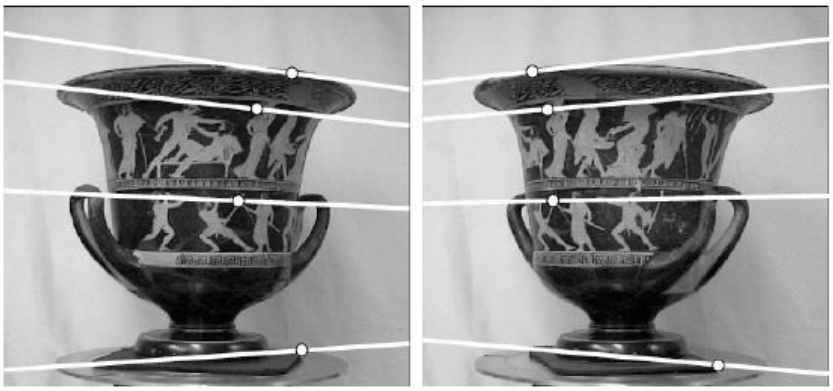
Stereo Vision

1. Motivation

The next goal is the recovery of the 3D structure. Thats only possible if we have 2 images taken from two different viewpoints. The reason for that is that the recovery of depth from one image is inherently ambiguous.



We will cover now geometric methods that rely on two views. *Stereo vision*, stereopsis, or short stereo is the perception or measurement of depth from two projections. It was first described in technical detail by Charles Wheatstone in the 1830s with the invention of the stereoscope. The human visual system heavily relies on stereo vision: Our eyes give two slightly shifted projections of the scene. But only relative depth can be judged accurately by humans.



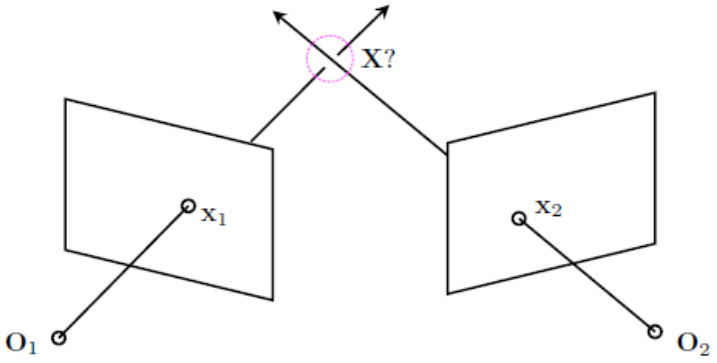
2. Triangulation

One method to get the depth from stereo and do 3D point reconstruction is triangulation. This enables 3D point reconstruction by ray intersection.

Since we will intersect two rays originating from the same point in the scene it requires correspondences (knowledge which pixels are images of “the same point”), like in the last lecture. We also need to know the camera pose (in order to construct the 3D rays), which we will discuss later in the epipolar geometry section.

3D point reconstruction by ray intersection

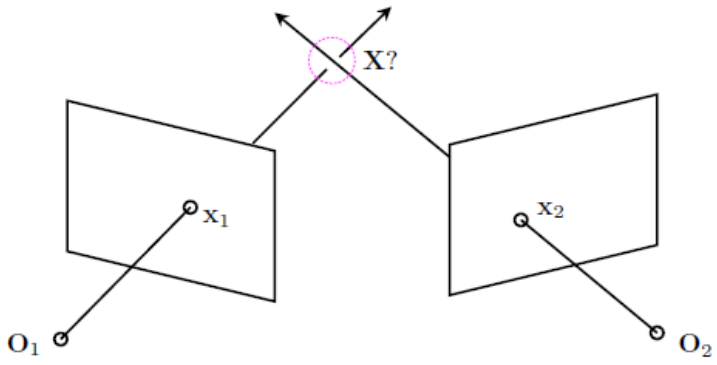
Given projections of a 3D point in two or more images (with known camera matrices), find the coordinates of the point.



We want to intersect the two viewing rays corresponding to x_1 and x_2, but because of noise and numerical errors, they do not meet exactly. To solve this problem there are 3 main methods: Geometric Midpoint, Linear Method, and Non-linear Method.

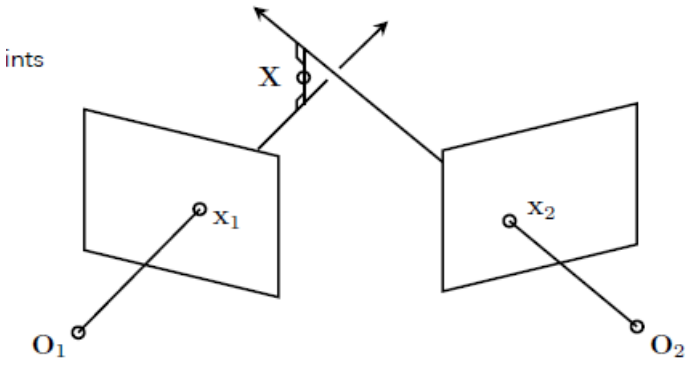
2.1. Geometric Midpoint

Given projections of a 3D point in two or more images (with known camera matrices), find the coordinates of the point.



We want to intersect the two viewing rays corresponding to x_1 and x_2, but because of noise and numerical errors, they do not meet exactly.

The idea is to find the shortest segment connecting the two viewing rays and let X be the geometric midpoint of that segment:

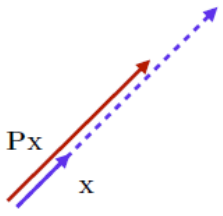


We first find the segment direction (normal to both rays), then construct 2 planes each containing the segment and one ray. At last, intersect planes with “other” rays to yield segment endpoints and average these points

2.2. Linear method

To find the coordinates of the 3D point X we need to solve two independent equations per image point for three unknown entries of X. Again, this yields a homogeneous linear equation system and again, we can solve it with SVD. Note that this directly generalizes to more than 2 views, since we only need to just stack more equations.

$$\lambda_1 \mathbf{x}_1 = \mathbf{P}_1 \mathbf{X} \qquad \mathbf{x}_1 \times \mathbf{P}_1 \mathbf{X} = 0$$
$$\lambda_2 \mathbf{x}_2 = \mathbf{P}_2 \mathbf{X} \qquad \mathbf{x}_2 \times \mathbf{P}_2 \mathbf{X} = 0$$



Algebraic Trick

On a side note, an algebraic trick would be useful here and will also be later in the epipolar geometry section. It is about how to get there without expanding the cross-product and re-ordering. The cross-product can be written as a matrix-vector multiplication:

$$\mathbf{x} \times \mathbf{PX} = 0\,?$$

$$\mathbf{a} \times \mathbf{b} = 0$$

$$[a]_\times \mathbf{b} = 0 \quad \text{with} \quad [a]_\times = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}$$

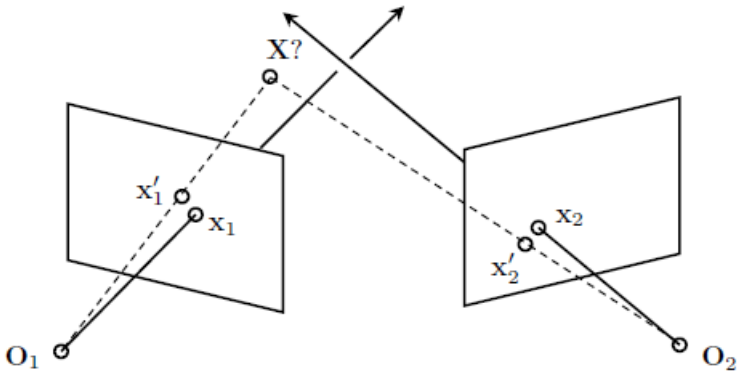
$$[\mathbf{x}]_\times \mathbf{PX} = 0$$

a matrix

2.3. Non-Linear Method

Find *X* that minimizes the (squared) reprojection error:

$$d^2(\mathbf{x}_1, \mathbf{P}_1\mathbf{X}) + d^2(\mathbf{x}_2, \mathbf{P}_2\mathbf{X})$$



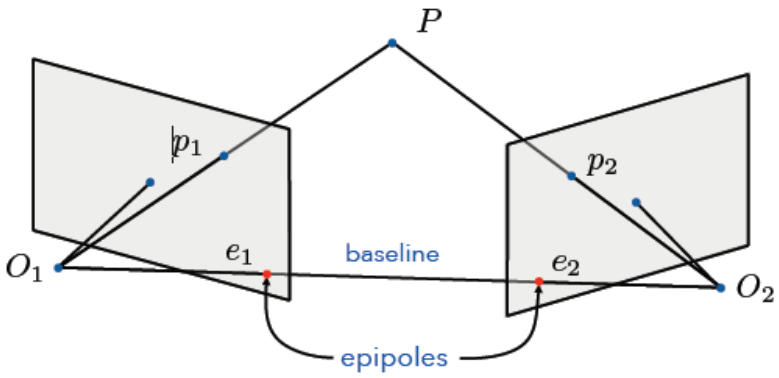
This the most accurate method but is more complex than the other two. The idea is to find roots of a 6th degree polynomial with 2 cameras (see: Hartley & Zisserman, chapter 12). If we have more than 2 cameras first initialize with a linear estimate and then optimize with iterative methods (Gauss-Newton, Levenberg-Marquardt - HZ, appendix 6)

3. Epipolar Geometry

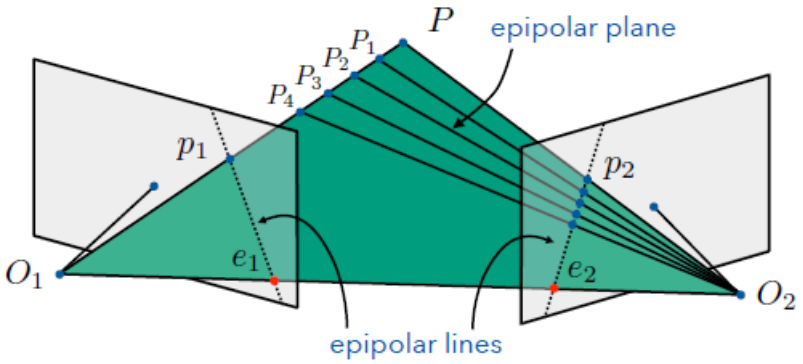
The next goal is to find the relation between two different views of the same scene. In detail we will recover camera placement, we will do matching and triangulation for multiple views and at end scene modelling will be possible only from images.

3.1. Geometrical and Algebraical Relations

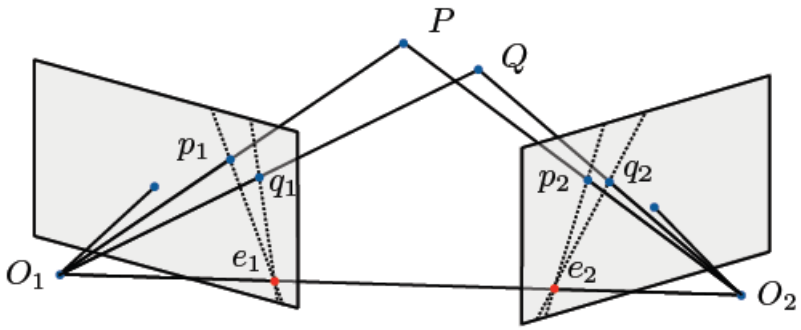
Given a 3D point *P* in the world and two images taken by two cameras with the same calibration matrix (later more) from two different viewpoints (baseline is not zero) then the *epipole* is defined as the image location of the optical center of the other camera:



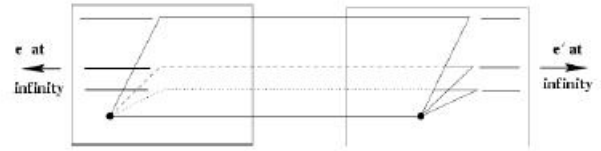
The epipole can be outside of the visible area (image expanding the images infinitely). The *epipolar plane* is defined as a plane through both camera centers and the world point. The *epipolar line* constrains the location where a particular feature from one view can be found in the other. In the figure below possible matches for p_1 are displayed.



Epipolar lines intersect at the epipoles are in general not parallel.

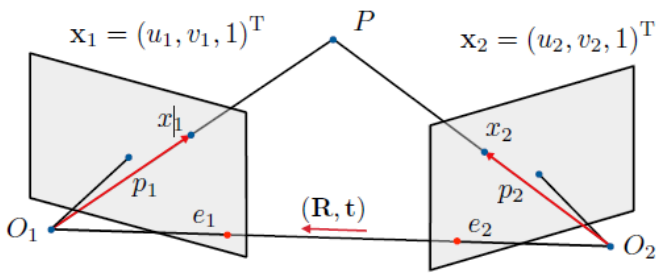


Some special cases for the location of the epipole are if we have a motion parallel to the image plane. The epipoles would be at infinity:



Another example is forward motion. The epipole has the same coordinate in both images. The point moves along lines radiating from epipole resulting in a “focus of expansion”.

Let’s describe epipolar geometry mathematically:



Mathematically: $\overrightarrow{O_1P}, \overrightarrow{O_2P}, \overrightarrow{O_1O_2}$ are coplanar.

Equivalently: $\overrightarrow{O_1p_1}, \overrightarrow{O_2p_2}, \overrightarrow{O_1O_2}$ are coplanar.

In other terms: $\overrightarrow{O_1p_1} \cdot \left[\overrightarrow{O_1O_2} \times \overrightarrow{O_2p_2}\right] = 0$

The next step is to rewrite the resulting equation in the camera coordinate system of camera 1 (for example):

$$\begin{aligned} \overrightarrow{O_1p_1} \cdot \left[\overrightarrow{O_1O_2} \times \overrightarrow{O_2p_2}\right] &= 0 \\ \Leftrightarrow \quad \mathbf{p}_1^{\mathrm{T}} \left[\mathbf{t} \times (\mathbf{R}\mathbf{p}_2)\right] &= 0 \end{aligned}$$

where $\boldsymbol{p}_1^T[\boldsymbol{t} \times (\boldsymbol{R}\boldsymbol{p}_2)] = 0$ is called the *epipolar constraint*. The epipolar constraint is very important in 3D reconstruction! Expressing the cross-product as a matrix (remember the trick we introduced earlier) yields the *epipolar constraint with essential matrix* $\boldsymbol{E} = [\boldsymbol{t}]_x\boldsymbol{R}$, which captures the relation of two views:

$$\begin{aligned} 0 &= \mathbf{p}_1^{\mathrm{T}} \left[\mathbf{t} \times (\mathbf{R}\mathbf{p}_2)\right] \\ &= \mathbf{p}_1^{\mathrm{T}} \left[([\mathbf{t}]_\times \mathbf{R})\mathbf{p}_2\right] \\ &= \mathbf{p}_1^{\mathrm{T}} \mathbf{E} \mathbf{p}_2 \end{aligned}$$

Hence two views of the same 3D point must satisfy $\boldsymbol{p}_1^T \boldsymbol{E} \boldsymbol{p}_2 = 0$. Other important properties of are:

- epipolar lines: $\boldsymbol{l_1} = \boldsymbol{E}\boldsymbol{p_2}$ and $\boldsymbol{l_2} = \boldsymbol{E}^T\boldsymbol{p_1}$
- epipoles are the (left/right) null-space of $\boldsymbol{E}$: $\boldsymbol{e}_1^T\boldsymbol{E} = \boldsymbol{E}^T\boldsymbol{e_1} = 0$, $\boldsymbol{E}\boldsymbol{e_2} = 0$
- Further the essential matrix is singular and has rank 2
- The two remaining eigenvalues are equal
- 5 degrees of freedom (translation + rotation have 6, but scale is arbitrary).

Special Case: Binocular Stereo

In this case the cameras are parallel and differ only by a translation t the rotation matrix equals the identity matrix: $R = I, t = \begin{bmatrix} -b \\ 0 \\ 0 \end{bmatrix}$. Thus, the essential matrix is:

$$\mathbf{E} = [\mathbf{t}]_\times \mathbf{R} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & b \\ 0 & -b & 0 \end{bmatrix}$$

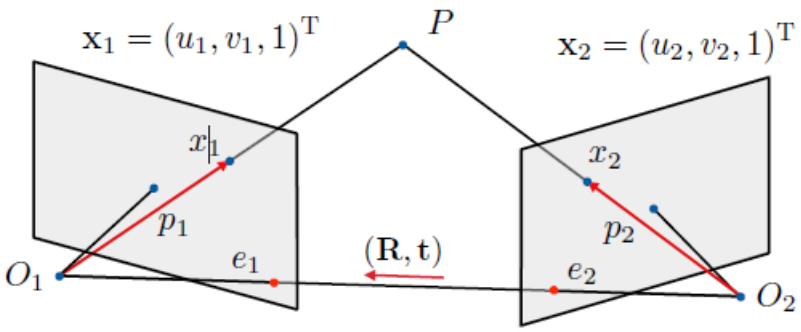
Plugging this into the epipolar constraint yields:

$$\begin{aligned} 0 &= \mathbf{p}_1^{\mathrm{T}} \mathbf{E} \mathbf{p}_2 \\ &= [x_1, y_1, 1] \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & b \\ 0 & -b & 0 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \\ &= [x_1, y_1, 1] \begin{bmatrix} 0 \\ b \\ -by_2 \end{bmatrix} = by_1 - by_2 \end{aligned} \quad \rightarrow \quad \begin{aligned} &y_1 = y_2 \\ &x_2 \text{ arbitrary} \end{aligned}$$

As you can see the y-coordinates of both images are the same! They only differ in x-coordinates! The good thing is we know we only need to search along the epipolar line and compare windows (image patches) to find the corresponding location!

3.2. Uncalibrated Cameras

Observation: the epipolar constraint is derived by intersecting lines and planes these lines and planes exist for any two pinhole cameras if we do not know the calibration, the constraint must still hold.



The transformation from rays $\boldsymbol{p}$ to image points $\boldsymbol{x}$ is the calibration $\boldsymbol{K}$. The transformation exists and is invertible (intrinsic camera transformation is always invertible), even if it is unknown

$$\mathbf{x}_1 = \mathbf{K}_1\mathbf{p}_1 \qquad \mathbf{x}_2 = \mathbf{K}_2\mathbf{p}_2$$
$$\mathbf{p}_1 = \mathbf{K}_1^{-1}\mathbf{x}_1 \qquad \mathbf{p}_2 = \mathbf{K}_2^{-1}\mathbf{x}_2$$

We silently assumed that everything was done in world coordinates. We can simply plug in the pixel coordinates

$$0 = \mathbf{p}_1^{\mathrm{T}}\mathbf{E}\mathbf{p}_2 = \mathbf{x}_1^{\mathrm{T}}\mathbf{K}_1^{-\mathrm{T}}\mathbf{E}\mathbf{K}_2^{-1}\mathbf{x}_2$$
$$0 = \mathbf{x}_1^{\mathrm{T}}\mathbf{F}\mathbf{x}_2$$

where $\boldsymbol{F} = \boldsymbol{K_1^{-1}EK_2^{-1}}$ is called the *fundamental matrix* and encapsulates the relative geometry of the two uncalibrated views. It holds for calibrated and uncalibrated cameras. Geometrically two rays must be coplanar, independent of any specific $\boldsymbol{K}, \boldsymbol{R}. \boldsymbol{t}$.

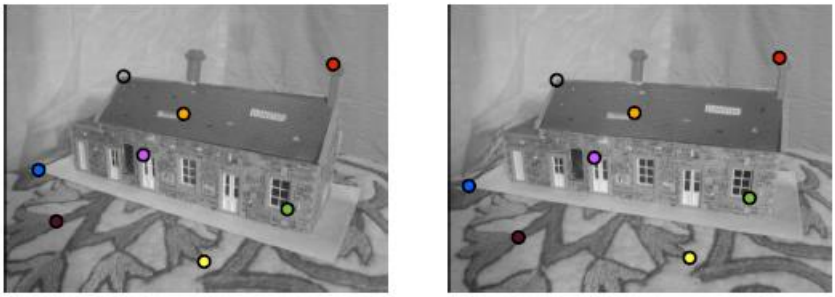
Two image coordinates of the same 3D point must satisfy (same as epipolar constraint):

$$\mathbf{x}_1^{\mathrm{T}}\mathbf{F}\mathbf{x}_2 = 0$$

Other important properties of are:

- epipolar lines: $\boldsymbol{l_1} = \boldsymbol{Fx_2}$ and $\boldsymbol{l_2} = \boldsymbol{F^T x_1}$
- epipoles are the (left/right) null-space of $\boldsymbol{F}$: $\boldsymbol{e_1^T F} = \boldsymbol{F^T e_1} = 0$, $\boldsymbol{Fe_2} = 0$
- Further the essential matrix is singular and has rank 2
- The two remaining eigenvalues are equal
- 7 degrees of freedom

How do we estimate the fundamental matrix? Seven degrees of freedom. One equation per correspondence. To estimate $\boldsymbol{F}$, we need at least 7 pairs of corresponding points, but the solution is non-linear. Linear solution with at least 8-point pairs yields the normalized eight-point algorithm.



Eight-point Algorithm

Stack equations from at least 8 correspondences. Remember homogeneous linear equation system and solve it with constraint s.t., $\|\boldsymbol{f}\| = 1$ using SVD:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0$$

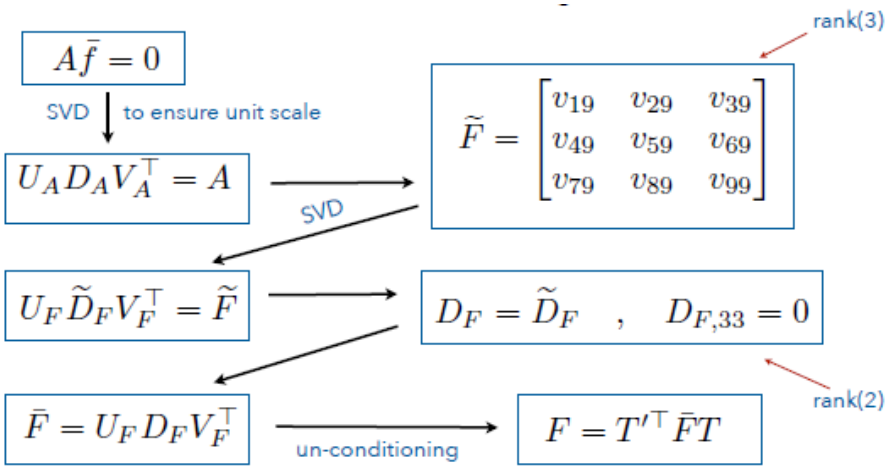
$$\begin{bmatrix} xx' & yx' & x' & xy' & yy' & y' & x & y & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0 \longrightarrow \mathbf{Af} = 0$$

We have a problem with *numerical stability* now. The coefficients of an equation system should be in the same order of magnitude so as not to lose significant digits in pixels: $xx' \sim 10^6$. We can solve this with *conditioning* by *scaling and shifting* points to be in in the range $[-1,1]$. We can exploit the inverse transform of the estimated fundamental matrix. This is also a general recommendation, not only for this case. This is how this would work:

$$s = \frac{1}{2} \max_i(\|x_i\|) \qquad t = \text{mean}(x_i) \longrightarrow T = \begin{bmatrix} \frac{1}{s} & 0 & -\frac{t_x}{s} \\ 0 & \frac{1}{s} & -\frac{t_y}{s} \\ 0 & 0 & 1 \end{bmatrix}$$

$$u = Tx \qquad u' = T'x' \longrightarrow u'^{\top} \bar{F} u = 0$$

Problem: the result should be of rank(2). Find the most similar rank(2)-matrix, i.e., force the smallest eigenvalue to be 0!



Important Conclusions and Cookbook Recipe

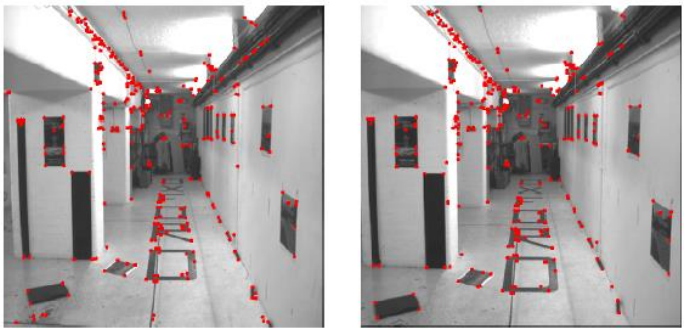
Calibration is not required, hence we can deal with archival images, photos not taken for reconstruction purposes and varying intrinsics.

So, where do we even start? The cookbook recipe is:

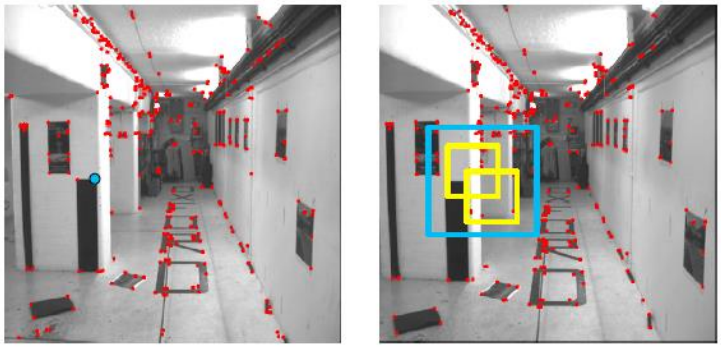
1. Find interest points in both images
2. Match them without epipolar constraint
3. Compute epipolar geometry
4. Refine

Let's look at an example:

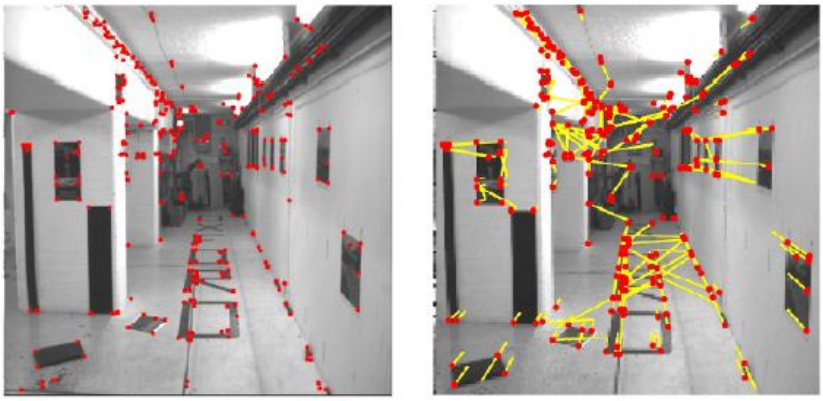
1. Interest points (here: Harris corner detector)



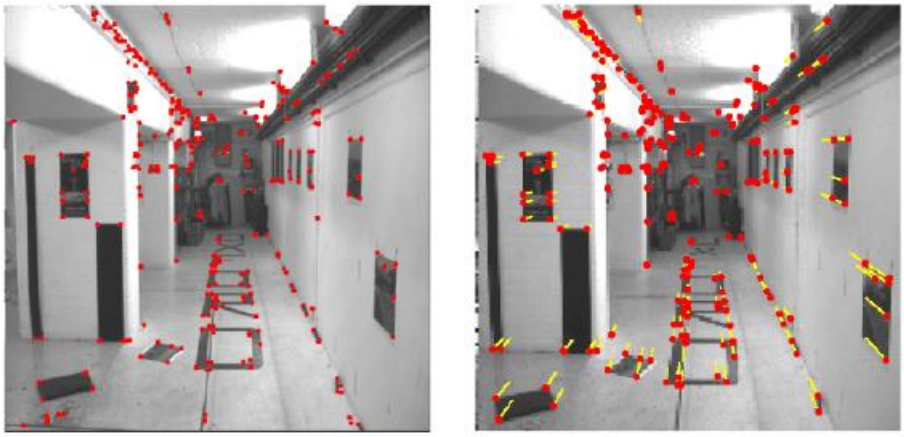
2. Match using only weak constraints (here: proximity)



3. Many wrong matches, but enough to estimate $\boldsymbol{F}$ (RANSAC)



4. Correspondences consistent with epipolar geometry:

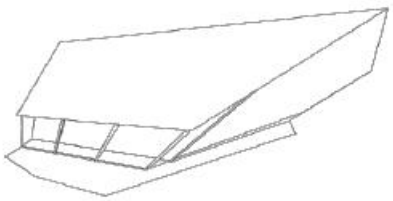


5. Resulting epipolar geometry:

What can we do with 2 views? Given two views of a static scene, and a small number of correspondences. If the views were recorded with any pinhole camera, we can:

- reconstruct the scene up to a projective ambiguity
- determine line intersections, coplanarity

- need at least a 3rd view to find the calibration

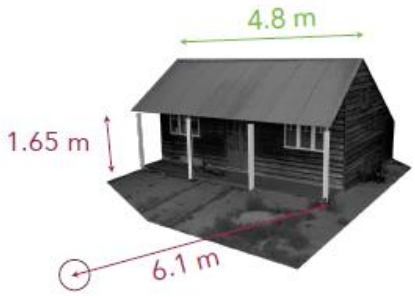


If the internal camera parameters have been calibrated, we can

- reconstruct the scene up to scale
- determine angles, relative distances
- allows 3D modelling.

If the baseline length (or any other distance in the scene) is known, we can

- determine the global scale
- measure absolute distances
- allows navigation



4. Dense Geometry

4.1. Motivation

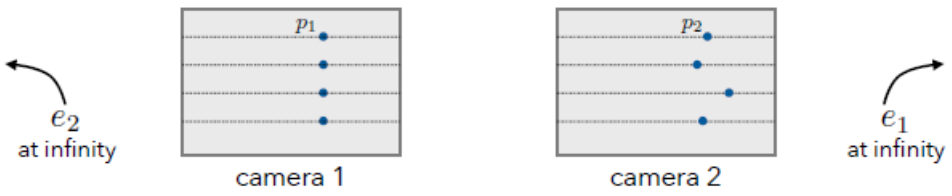
Goal: Given two views of the same scene, estimate its 3D geometry densely, not just at interest points! This is way too hard!

Narrower formulation: Given two views with similar (but not identical!) viewpoint, fuse them to obtain a depth image. Way easier!

4.2. Binocular Stereo

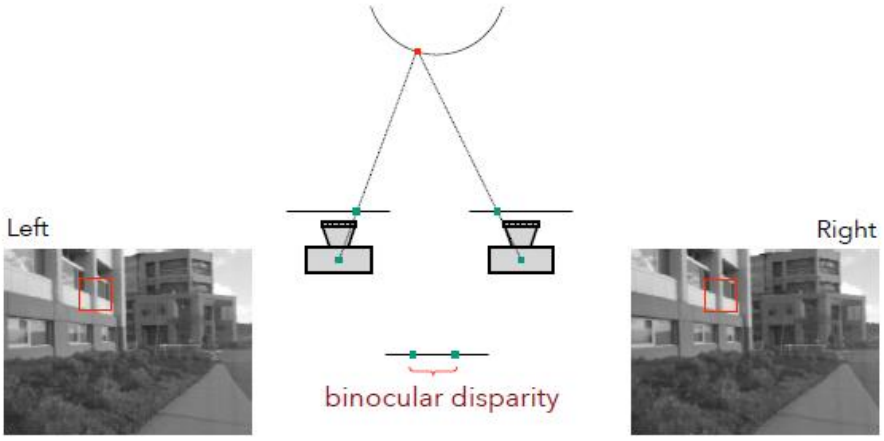
We will start to tackle the goal with binocular stereo. As already known, this is a special case of epipolar geometry for stereo cameras with a standard binocular setup:

- Epipoles are at infinity
- Epipolar lines are parallel
- Points correspond along the scanlines of the image.

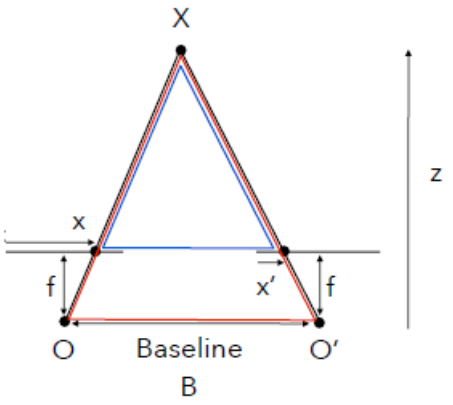


(Binocular) Disparity

From known geometry of the cameras and estimated *disparity*, recover depth in the scene. Disparity refers to the difference in image location (x-coordinate) of an object seen by the left and right image. The y-coordinates are the same though!



From known geometry of the cameras and estimated disparity, recover depth in the scene. For Parallel image planes like here we can get the disparity from equal triangles:



$$\frac{Z-f}{B-(x-x')} = \frac{Z}{B}$$
$$B \cdot Z - B \cdot f = B \cdot Z - Z \cdot (x-x')$$
$$Z = \frac{B \cdot f}{x-x'}$$

The *disparity* is then given by:

$$d = x - x' = \frac{B \cdot f}{Z}$$

where $Z(x,y)$ is the depth *at pixel* (x,y) and $d(x,y)$ is the disparity. We ca estimate the depth then with:

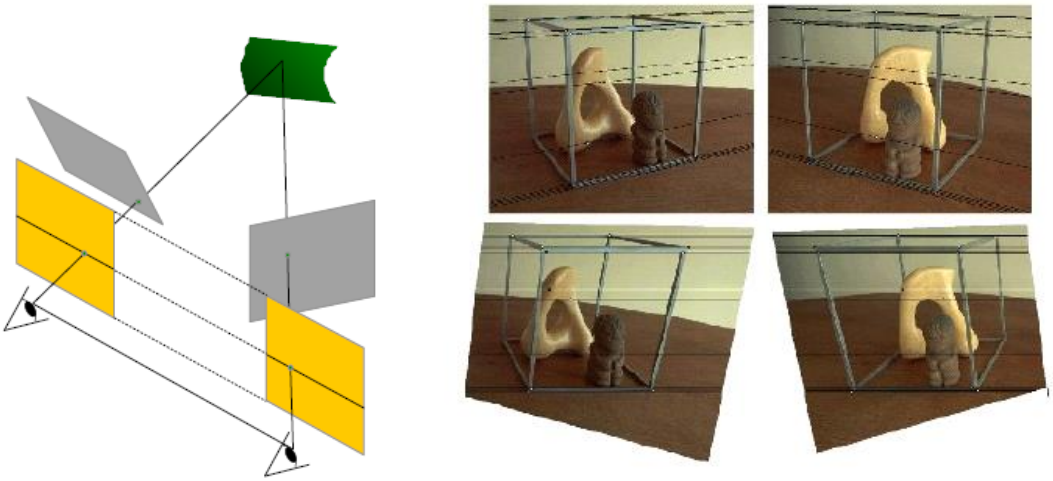
$$Z(x,y) = \frac{fB}{d(x,y)}$$

Now we only need to find the disparity by searching for the best match along the epipolar line!

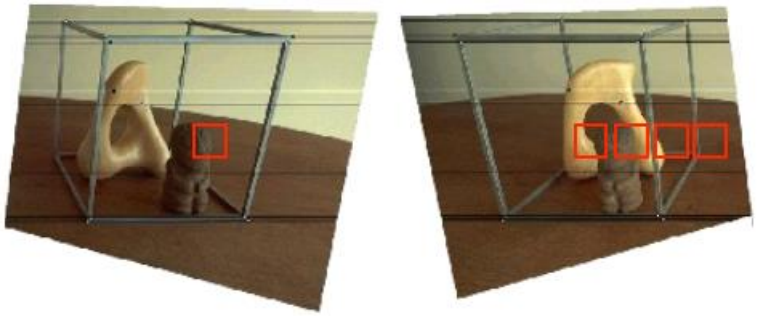
Stereo Rectification

What if we don't have a binocular setup? Use *stereo rectification*! Rewrap the images so they are a binocular setup! Here is how to do it:

1. Map both image planes to a common plane parallel to the baseline
2. Requires a homography for each image
3. After the transform, pixel motion is horizontal again

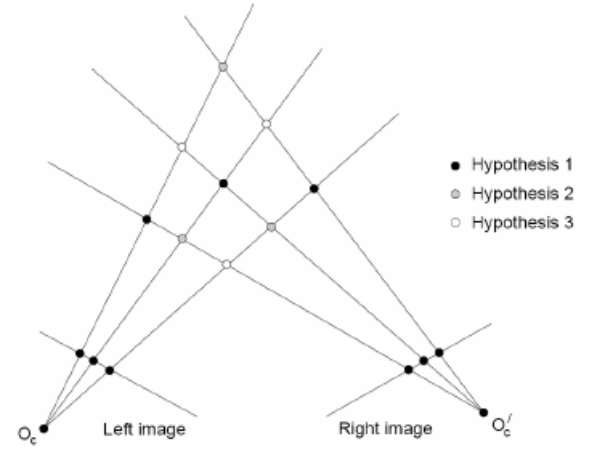


Matching only must occur along epipolar lines. Now in the simpler binocular case where the cameras are pointing forward.



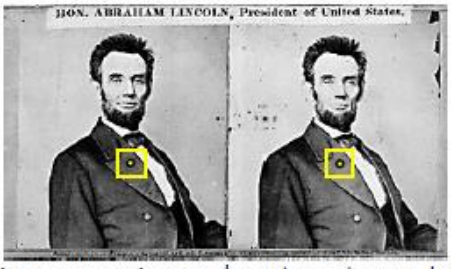
Problems of Stereo Correspondence and its solution

Search over disparity to find correspondences. The range of disparities to search over can change dramatically within a single image pair. We also have a correspondence problem. Even when constrained to 1D (by the epipolar constraints), there are multiple matching hypotheses. So which one is correct?



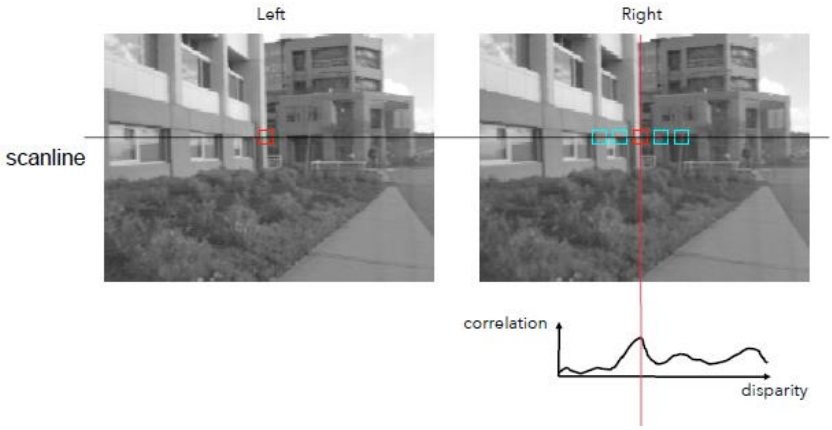
Let us make some assumptions to simplify the matching problem:

- The baseline is relatively small (compared to the depth of scene points). We want to limit the disparity. Also called “narrow-baseline stereo”
- Matching regions are similar in appearance
- Most scene points are visible in both views



note: the 1st assumption makes the others a lot more likely

We will find correspondences by using correlation.



We are looking at a patch or window! This is better than only looking at 1 pixel! Because we have then more information! Now we are comparing image patches along the scanline/disparity and plot the correlation of them. Then find the disparity with the maximum correlation value! This leads us to window-based matching.

Window-based matching

In practice we compute the *normalized correlation*:

w_L and w_R are corresponding m x m windows of pixels. We also write them as vectors $\boldsymbol{w_L}$ and $\boldsymbol{w_R}$. The normalized correlation computes the cosine of the angle between the patches:

$$\mathrm{NC}(x,y,d) = \frac{(\mathbf{w}_L(x,y) - \bar{\mathbf{w}}_L(x,y))^{\mathrm{T}}(\mathbf{w}_R(x-d,y) - \bar{\mathbf{w}}_R(x-d,y))}{|\mathbf{w}_L(x,y) - \bar{\mathbf{w}}_L(x,y)|\,|\mathbf{w}_R(x-d,y) - \bar{\mathbf{w}}_R(x-d,y)|}$$

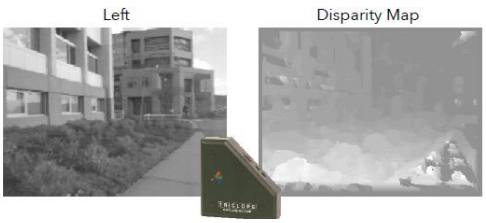
patch mean

Even a simpler method is to use sum of squared (Pixel) differences. The SSD cost measures the intensity difference as a function of disparity (What we are saying is that the brightness doesn't change for the patch we are searching for in the second image even if it moved with the disparity d. It stays the same!):

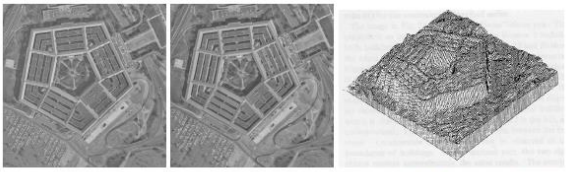
$$\mathrm{SSD}_R(x,y,d) = \sum_{(x',y') \in w_L(x,y)} (I_L(x',y') - I_R(x'-d,y'))^2$$

In practice:

1. Choose some disparity range $[0, d_{max}]$
2. For all pixels (x, y) try all disparities and choose the one that maximizes the normalized correlation or minimizes the SSD.



3. Optional: Of course, we can convert the disparity back to depth:

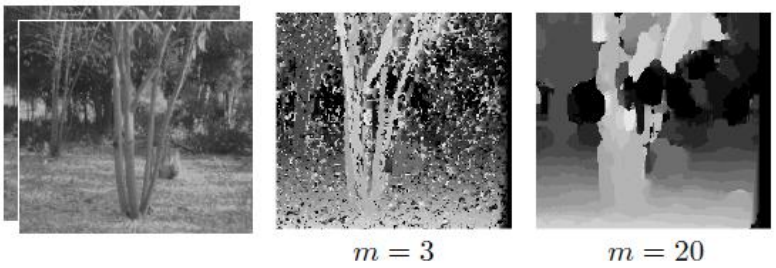


From [Ohta & Kanade, 1985]

Challenges and Problems:

- How do we choose the right window size m?
- Mismatches often lead to relatively poor results quality.

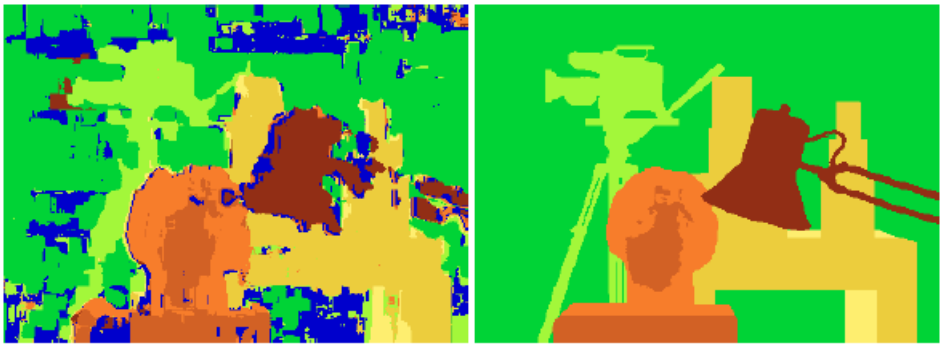
No window size fits every condition. Better results with adaptive window:



$m = 3$ $m = 20$

Improving window-based matching

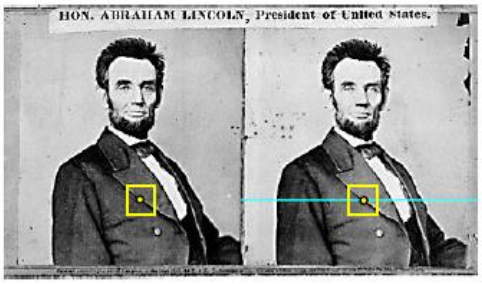
How can we improve window-based matching? Because this is the result with window correlation:



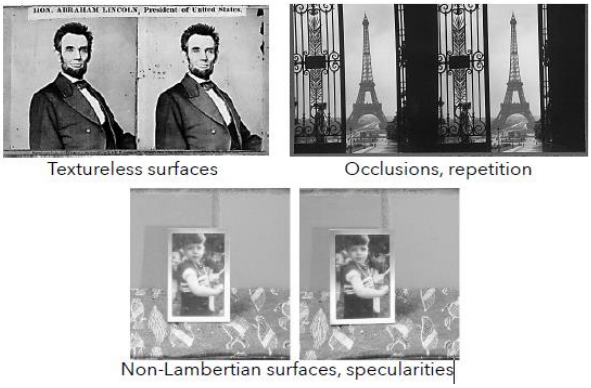
Window-based matching (best window size)

Ground truth

It doesn't look good. We can use the *similarity constraint*. Corresponding regions in two images should be similar in appearance and non-corresponding regions should be different. When will the similarity constraint fail or where are its limitations?



It fails for textureless surfaces, occlusions, repetition, non-Lambertian surfaces (glossy surfaces) and specularities:



Even when the cameras are identical models, there can be differences in gain and sensitivity. The cameras do not see exactly the same surfaces, so their overall light levels can differ. Plus, there may be other "artifacts". The *normalized correlation* is invariant to some of these (e.g., additive changes of light level). Other possibility is to use a matching criterion that is less sensitive to mismatches:

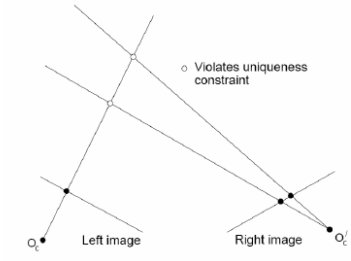
$$E_R(x, y, d) = \sum_{(x', y') \in w_L(x, y)} \rho\left(I_L(x', y') - I_R(x' - d, y')\right)$$

Robust matching function

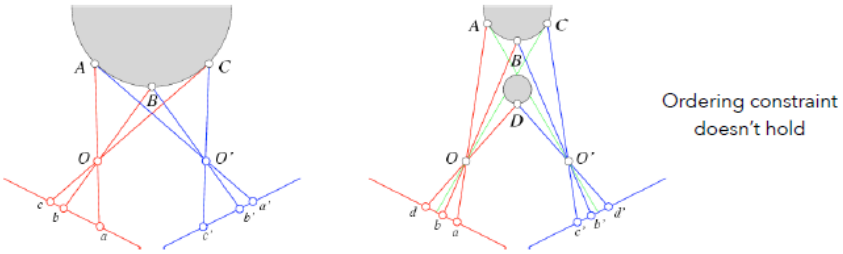
The similarity constraint is local. Each reference window is matched independently, and other points do not influence the result. Another method to improve window-based matching is to enforce *non-local correspondence constraints*. This requires spatial regularity (Computer Vision II)!

Non-local constraints ensure:

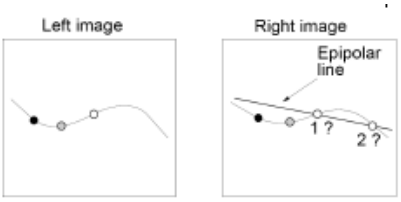
- Uniqueness: For any point in one image, there should be at most one matching point in the other image.



- *Ordering*: Corresponding points should be in the same order in both views



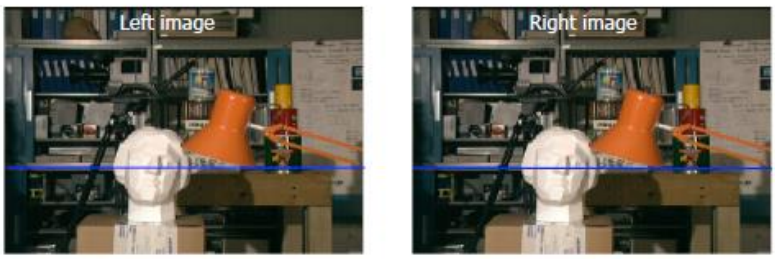
- *Smoothness:* We expect disparity values to change slowly (for the most part).



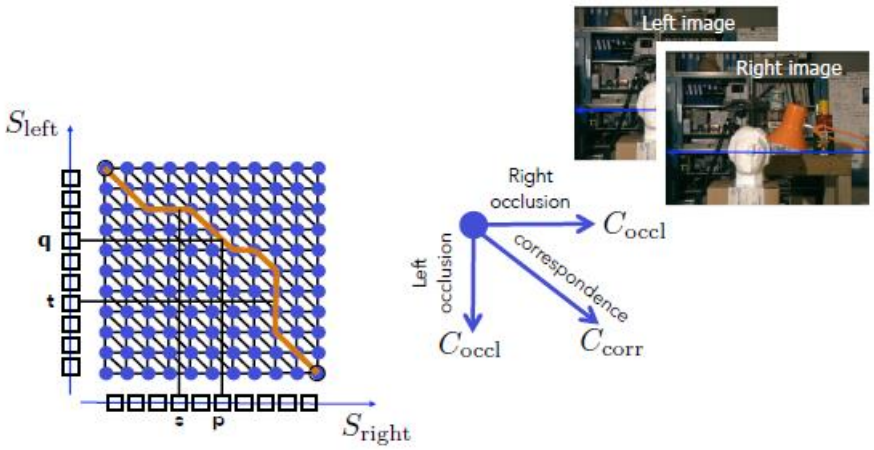
Given matches ● and ○, point ○ in the left image must match point 1 in the right image. Point 2 would exceed the disparity gradient limit.

Scan line stereo

Try to coherently match pixels on the entire scanline and different scanlines are still optimized independently:

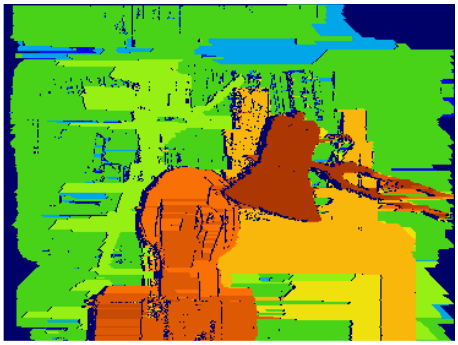


The shortest path for scan-line stereo can be found using dynamic programming:



Can be implemented with dynamic programming
[Ohta & Kanade '85], [Cox et al. '96]

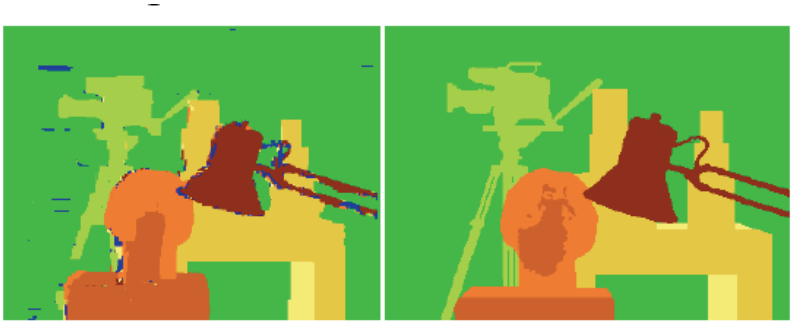
Scanline stereo generates streaking artifacts:



Can't use dynamic programming to find spatially coherent disparities/ correspondences on a 2D grid!

What's next?

Well anyway this is the best result we can achieve:



Graph cuts Ground truth

How can be improve the results? We need to impose spatial regularity to improve results! Stay tuned for Computer Vision 2!

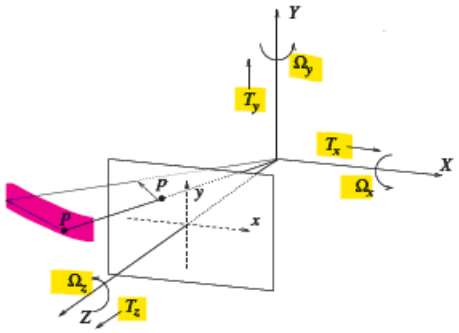
Dense Motion Estimation

1. Introduction

What is image motion? Image motion refers to the apparent movement of objects or patterns within an image or video. It occurs when there is a change in the position or orientation of objects within the frame, or when the camera itself moves.

1.1. Motion Field

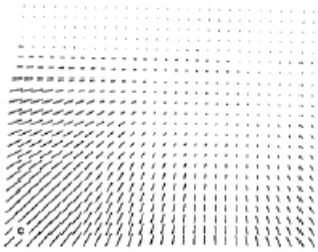
Motion field is a 2D motion field representing the projection of the 3D motion of points in the scene onto the image plane. This can be the result of camera motion (yellow colour) or object motion (pink colour) (or both)!



1.2. Optical Flow

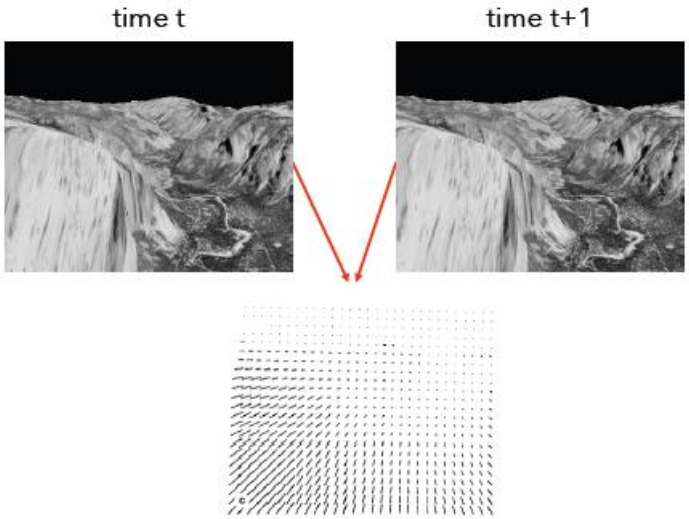
Optical flow is the 2D velocity field describing the apparent motion in the images. Optical Flow and Motion Field are not the same.

For every pixel we want to obtain a vector that tells us how it moved. This vector has a horizontal component $u(x, y)$ and vertical component $v(x, y)$.



The intensity function now gets an additional variable t: $I(x, y, t)$ is the image brightness at time t and location $\boldsymbol{x} = (x, y)$.

We will now only work with 2 images taken at different time steps t and $t + 1$. The reference coordinate system is at the image taken at time t:

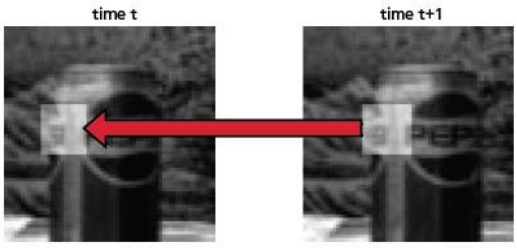


2. Computing Flow

How do we compute flow? We will make 2 important assumptions first. Then we will exploit those computationally.

Assumption 1 - Brightness Constancy

Image measurements (e.g., brightness) in *a small region* remain the same although their location may change.



Assumption 1:

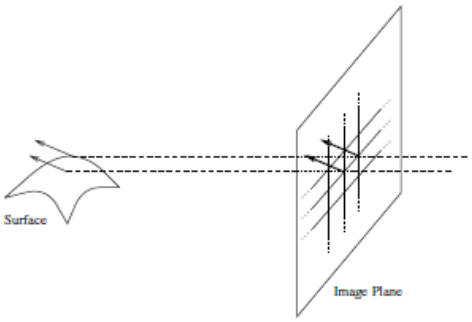
$$I(x + u(x, y), y + v(x, y), t + 1) = I(x, y, t)$$

shifted by horizontal flow

shifted by vertical flow

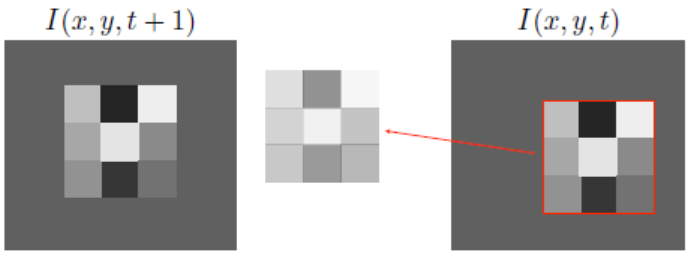
Assumption 2 – Spatial Coherence

Neighbouring points in the scene typically belong to the same surface and hence typically have similar 3D motions. Since they also project to nearby points in the image, we expect *spatial coherence* in the image flow.



Simple Flow Estimation Algorithm

The goal is to *minimize the brightness difference* or optimize the *sum of squared difference's function* (SSD). As already mentioned, image measurements (e.g., brightness) in *a small region* remain the same although their location may change. This is exploited here:

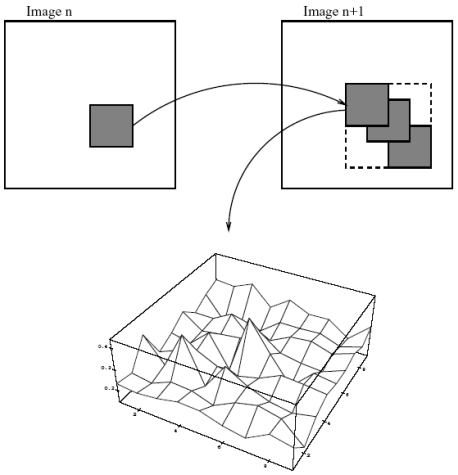


$$E_{SSD}(u, v) = \sum_{(x,y)\in R} \big(I(x+u, y+v, t+1) - I(x, y, t)\big)^2$$

The simple way of doing flow estimation is:

1. Discretize the space of possible motions, just like in dense stereo.
2. For each pixel, (take its neighbourhood region and) try all possible motions.
3. Take the one that minimizes the SSD.

The problem with this approach is that it is very inefficient. The motions are discrete, which is usually not true in practice. So how can we optimize this nonlinear and non-convex function?



Optical Flow Constraint Equation

The main problem because $E_{SSD}(u, v)$ is difficult to optimize (normally quadratic functions should be easy to optimize) lies in the term $I(x + u, y + v, t + 1)$. We can approximate this term if we assume *small motions* with a first order Taylor Series approximation. After some mathematical operations we will get:

$$E_{SSD}(u, v) \approx \sum_{(x,y)\in R} \left(u \cdot I_x(x, y, t) + v \cdot I_y(x, y, t) + I_t(x, y, t)\right)^2$$

This approximated SSD objective is a *convex* function of the motion u and v. It becomes much easier to optimize. We will see that shortly. But this only holds for small motions!

By minimizing this Taylor series approximation to the SSD, we are trying to enforce the so-called optical flow constraint equation (OFCE):

$$u \cdot I_x + v \cdot I_y + I_t = 0$$

This is also called the linearized brightness constancy constraint. Another better notation is:

$$u \cdot I_x + v \cdot I_y + I_t = 0$$

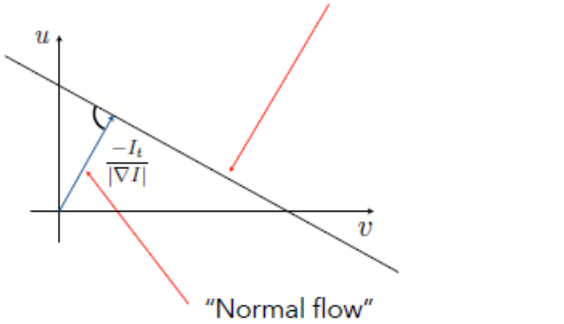
$$\downarrow$$

$$\nabla I^{\mathrm{T}} \mathbf{u} = -I_t$$

$$\mathbf{u} = \begin{pmatrix} u \\ v \end{pmatrix} \qquad \nabla I = \begin{pmatrix} I_x \\ I_y \end{pmatrix}$$

If we take a single image pixel, we get a constraint line:

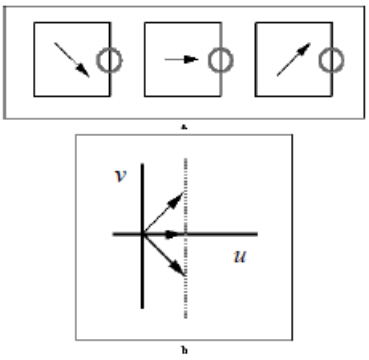
$$u \cdot I_x + v \cdot I_y + I_t = 0$$



Aperture Problem

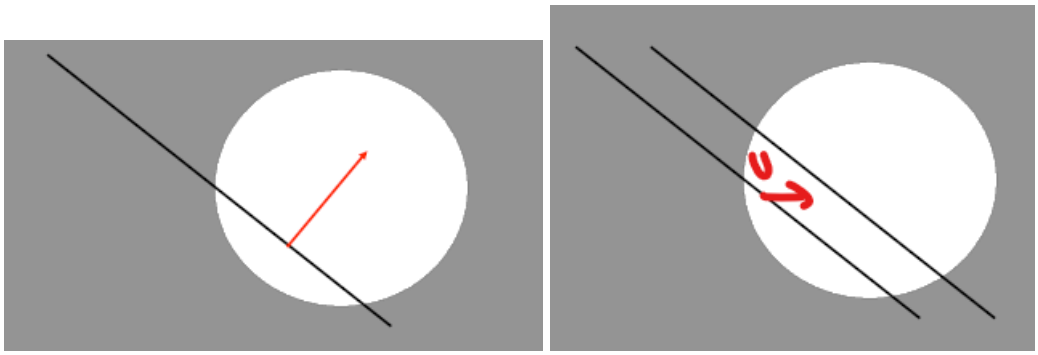
Limited view through a *small window/region* creates ambiguity in perceiving the true motion direction of objects. Contextual information is needed to resolve this ambiguity.

$$
\begin{aligned}
u \cdot I_x + v \cdot I_y &= -I_t \\
u \cdot I_x + 0 \cdot I_y &= -I_t \\
u \cdot I_x &= -I_t \\
u &= -\frac{I_t}{I_x}
\end{aligned}
$$

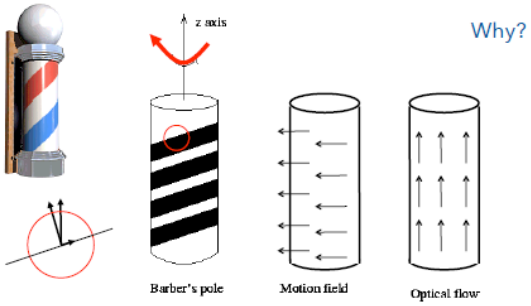


v could be anything!

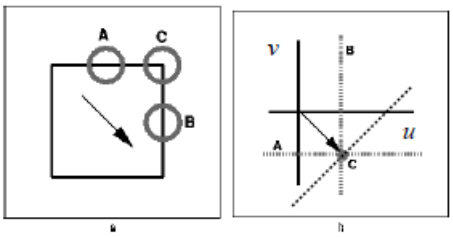
We can only measure normal velocity (perpendicular to the edge), because we are looking at a small region for example this pole below might not move diagonally (in direction of the normal velocity) but to the right.



Another popular example is the barber pole illusion:



To solve this is easy: Combine multiple constraints to get an estimate of the velocity (not only the normal component).



3. Lukas Kanade Optical Flow Method

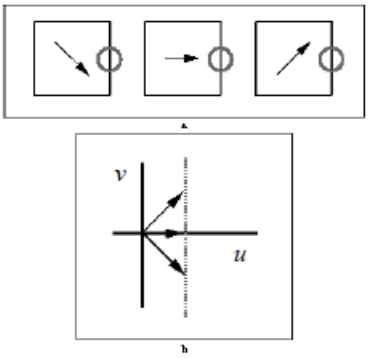
3.1. Derivation

We will start with the Aperture Problem.

Aperture Problem

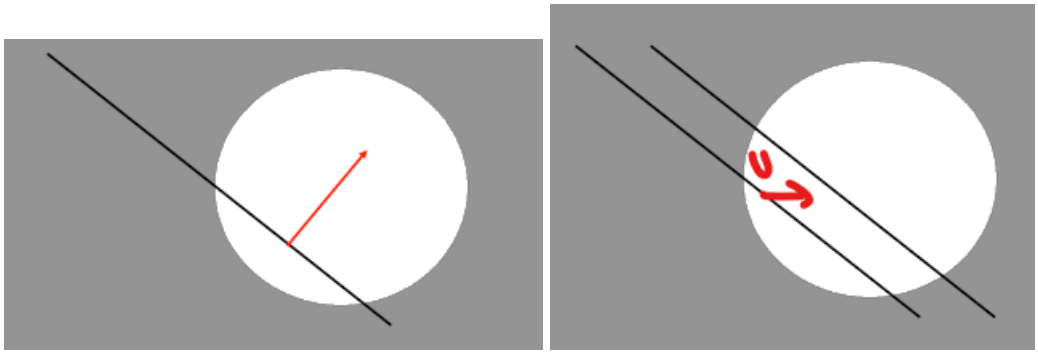
Limited view through a *small window/region* creates ambiguity in perceiving the true motion direction of objects. Contextual information is needed to resolve this ambiguity.

$$
\begin{aligned}
u \cdot I_x + v \cdot I_y &= -I_t \\
u \cdot I_x + 0 \cdot I_y &= -I_t \\
u \cdot I_x &= -I_t \\
u &= -\frac{I_t}{I_x}
\end{aligned}
$$

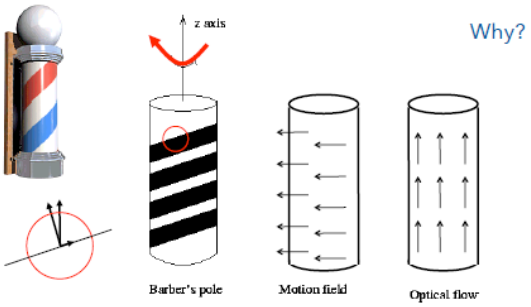


v could be anything!

We can only measure normal velocity (perpendicular to the edge), because we are looking at a small region for example this pole below might not move diagonally (in direction of the normal velocity) but to the right.



Another popular example is the barber pole illusion:



To solve this is easy: Combine multiple constraints to get an estimate of the velocity (not only the normal component).

Area-Based Flow and Optimization

How do we combine multiple constraints? Remember our assumptions. We will assume spatial smoothness (coherence) of the flow. More specifically: Assume that the flow is constant in a region.

$$E_{SSD}(u,v) \approx \sum_{(x,y)\in R} \big(u \cdot I_x(x,y,t) + v \cdot I_y(x,y,t) + I_t(x,y,t)\big)^2$$

This is what we have been doing (sliding window). But how do we solve for the motion? We differentiate $E_{SSD}(u,v)$ with respect to u and v and set this to zero-

$$\begin{aligned}
\frac{\partial}{\partial u} E_{SSD}(u,v) &\approx 2 \sum_{(x,y)\in R} \big(u \cdot I_x(x,y,t) + v \cdot I_y(x,y,t) + I_t(x,y,t)\big) I_x(x,y,t) = 0 \\
\frac{\partial}{\partial v} E_{SSD}(u,v) &\approx 2 \sum_{(x,y)\in R} \big(u \cdot I_x(x,y,t) + v \cdot I_y(x,y,t) + I_t(x,y,t)\big) I_y(x,y,t) = 0
\end{aligned}$$

Rearrange the terms and drop the constant. This yields a system of 2 equations in 2 unknows where the structure tensor (left matrix) is positive definite and hence invertible:

$$\begin{aligned}
\left[\sum_R I_x^2\right] u + \left[\sum_R I_x I_y\right] v &= -\left[\sum_R I_x I_t\right] \\
\left[\sum_R I_x I_y\right] u + \left[\sum_R I_y^2\right] v &= -\left[\sum_R I_y I_t\right]
\end{aligned}$$

$$\begin{bmatrix} \sum_R I_x^2 & \sum_R I_x I_y \\ \sum_R I_x I_y & \sum_R I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\sum_R I_x I_t \\ -\sum_R I_y I_t \end{bmatrix}$$

Structure tensor

The structure tensor is an important tool in vision. The eigenvectors give the principal directions of the local image variation. The eigenvalues indicate their strength. We've use it before to find interest points.

Rewrite using the abbreviations from before:

$$\left(\sum_R \nabla I \nabla I^{\mathrm{T}}\right) \mathbf{u} = -\sum_R I_t \nabla I$$

Invert structure tensor to obtain flow for example for u (do this respectively for v):

$$\mathbf{u} = -\left(\sum_R \nabla I \nabla I^{\mathrm{T}}\right)^{-1} \left(\sum_R I_t \nabla I\right)$$

This is a classical flow technique what we just derived: B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision.

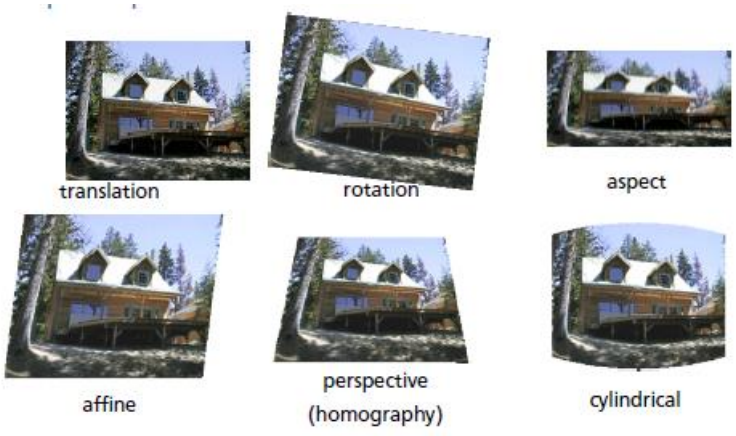
What happens if:

- The region is homogeneous?: Left term is not invertible
- There is a single edge?: Left term is not invertible
- There is a corner?: Left term is invertible

3.2. Image Registration

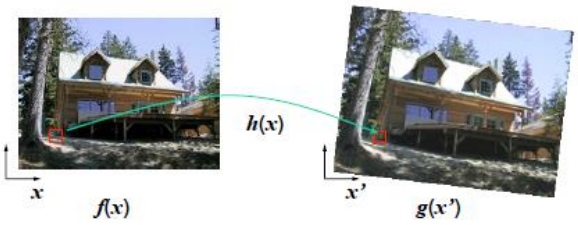
Remember the title of the Lucas-Kanade paper? "An iterative image registration technique with an application to stereo vision." What does that have to do with optical flow? We can use this to register (i.e., align) images: Compute the flow with the entire images. Shift the second image toward the first based on the flow. Iterate until convergence. How do we "shift" an image? We can use Forward od Inverse Warping!

Given a coordinate transform $x' = h(x)$ and a source image $f(x)$, how do we compute a transformed image $g(x) = f(h(x))$. Note that we only need translations for now, but it's good to know the general case. Here are some examples anyway:

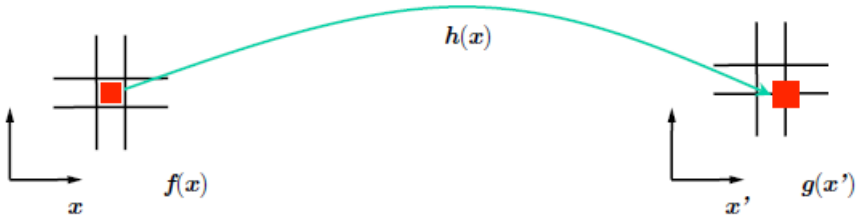


Forward Warping

Send each pixel $f(x)$ to its corresponding location $x' = h(x)$ in $g(x')$.

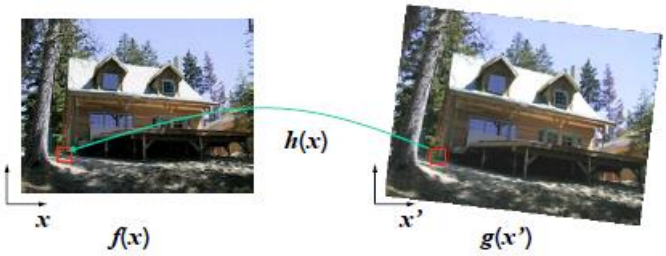


What if pixel lands “between” two pixels? Answer: Add “contribution” to several pixels, normalize later (splatting).

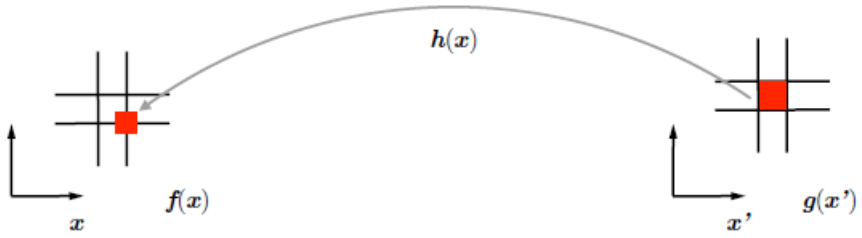


Inverse Warping

Get each pixel $f(x)$ from its corresponding location $x' = h(x)$ in $g(x')$.



What if pixel comes from “between” two pixels? Answer: Resample pixel value from interpolated source image.

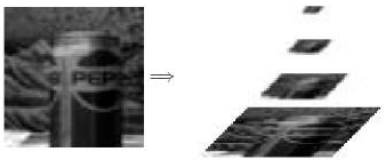


Possible interpolation filters: nearest neighbor, bilinear, bicubic (interpolating). Needed to prevent “jaggies”. When iteratively warping, always warp the original image!

Coarse-to-fine estimation

How do we “shift” an image? We use image warping to shift the image. But there is still one problem which is that the LK-model only holds for small motions. The solution is *Coarse-to-fine estimation*.

1. Build a Gaussian pyramid



2. Start with the lowest resolution (motion is small!)
3. Use this motion to pre-warp the next finer scale
4. Only compute motion increments (small!)

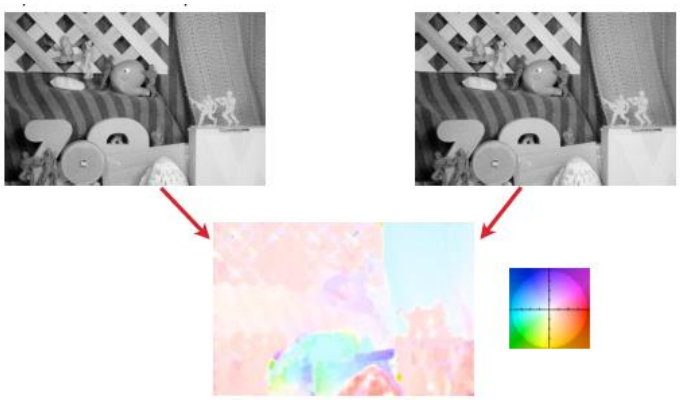
Dense LK Flow

What if we now want to estimate dense flow? We can just take the region R to be a small region around every pixel and compute a flow vector for every pixel. But we face the same problems as before: The LK method only works for small motions. Two workarounds (use both): Iterative estimation, Coarse-to-fine estimation.

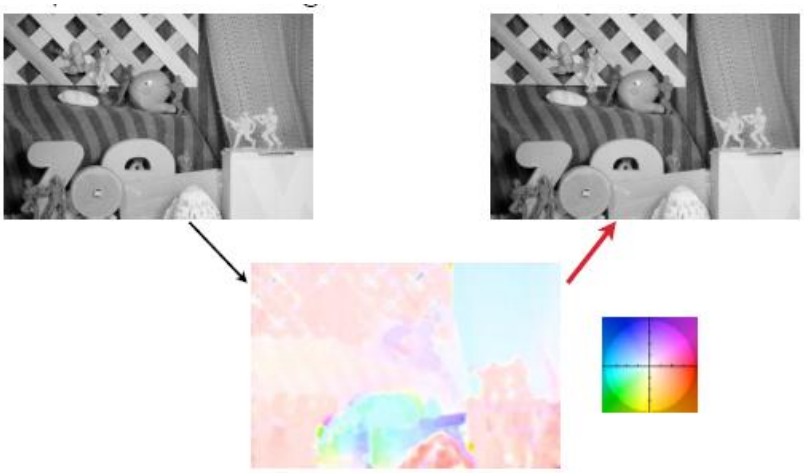
LK Full Example

1. Take image pair and compute LK flow (21x21 window). The window is so big because we need to make sure its invertible later. The white colour means we ha we no

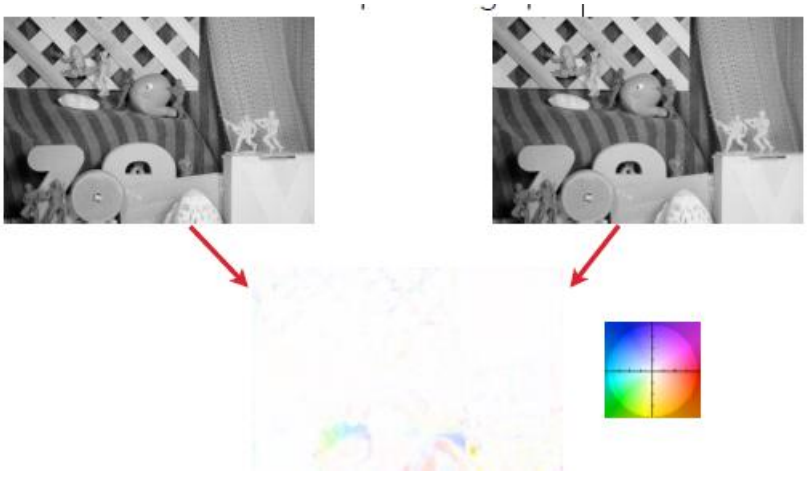
motion:



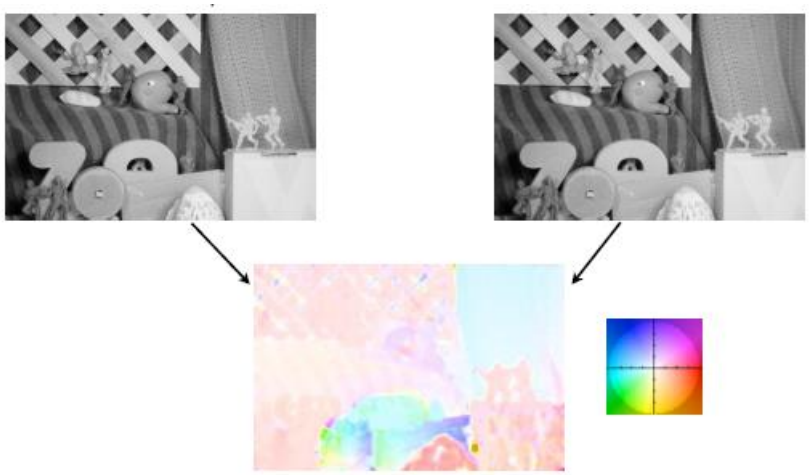
2. Inverse Warp the second image towards the first image



3. Estimate incremental flow from warped image pair



4. Add incremental flow to previous estimate



5. Warp again and so on... until convergence

Of coarse, we also apply our previous trick and use a Gaussian pyramid: Initialize with the flow from a coarser level. If we do this on the previous image pair, we get this (still not good):

What is the problem now?

The window is too big! All the discontinuities in the flow are smoothed over. But discontinuities do exist, e.g., at motion boundaries. But: The window is also too small! In some areas, the flow estimate is poor, because there is not enough image information in the window. This happens in areas with little texture. LK is a local optical flow method. Global flow methods to the rescue (CV II)!

Object Recognition

1. Motivation

Object recognition is a *classification* problem. Choose one class from a list of possible candidates! The following question all belong to classification problems: *What is it?* (Object and scene recognition), *Who is it?* (Identity recognition), *Where is it?* (Object detection), *What are they doing?* (Activity recognition). So far, we only looked at view-based recognition!

Naïve View-Based Approach

We have a database of mouth templates. Search every image region (at every scale) and compare each template with it. Choose the best match.

Appearance-Based Instance Recognition

Appearance-Based Instance Recognition basic assumptions were:

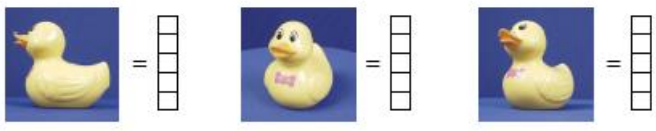
- Objects can be represented by a set of images (“appearances”).
- For recognition, it is sufficient to just compare the 2D appearances.
- No 3D model is needed.

But this has some challenges:

- Viewpoint changes: Translation, Image-plane rotation, Scale changes, Out-of-plane rotation
- Illumination
- Clutter
- Occlusion
- Noise

Global Representation

The idea now is to use a *global representation*. Represent each object (view) by a *global feature descriptor*.



For recognizing objects, just match the (global) descriptors. Some modes of variation are built into the descriptor, others have to be incorporated in the training data or the recognition process. In view-based representations, pixels (or projections onto global basis vectors) are the descriptor. Very limited amount of invariance!

Problems of view-based recognition

They are severely challenged by these common variations. To make them work, we would need an unmanageable number of examples (training data). They do not generalize well! Almost any variation that hasn't been captured in the training data will not be handled gracefully. Training data is expensive: Humans must gather and label it. What else can we do? Move away from representing the object simply by its pixels. Images as representation are too rigid.

2. Feature Representations

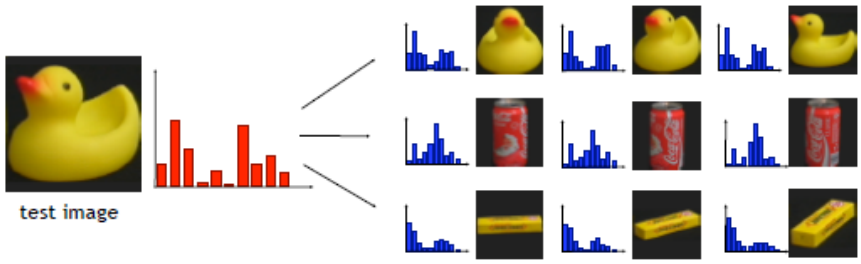
Feature representation can be divided in two main parts:

- *Dense representation* (regular grid):
  - Colour histogram approach
  - Multidimensional receptive field histograms

- *Sparse feature representations*:
  1. Find key points with interest point detector e.g., scale- or affine-invariant
  2. Represent key points with feature vectors e.g., SIFT, ShapeContext

2.1. Dense representation

Recognition using histograms is easy. We are doing a histogram comparison between the database of known objects and a test image of an unknown object. Database has multiple training views per object.

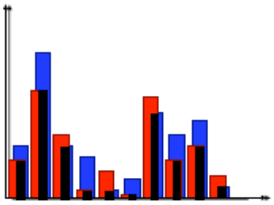


How do we even measure the histogram comparisons? There are quite a few methods: *Histogram intersection*, *Euclidian distance,* and *Chi-square*.

Histogram intersection

Measures the common part of both histograms. The range is [0,1].

$$\cap(Q,V) = \sum_i \min(q_i, v_i)$$



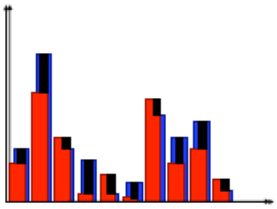
For unnormalized histograms, use:

$$\cap(Q,V) = \frac{1}{2}\left(\frac{\sum_i \min(q_i, v_i)}{\sum_i q_i} + \frac{\sum_i \min(q_i, v_i)}{\sum_i v_i}\right)$$

Euclidean Distance

Focuses on the differences between the histograms. The Range is $[0, \infty]$. All cells are weighted equally. Not very discriminative.

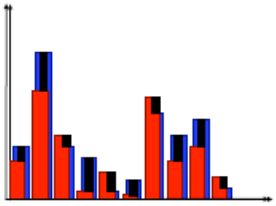
$$d(Q,V) = \sum_i (q_i - v_i)^2$$



Chi-Square

First the statistical background is to test if two distributions are different. It's possible to compute a significance score. The is range is $[0, \infty]$. Cells are not weighted equally! Therefore this is more discriminative and may have problems with outliers (therefore assume that each cell contains at least a minimum of samples).

$$\chi^2(Q,V) = \sum_i \frac{(q_i - v_i)^2}{q_i + v_i}$$



Which measure is best?

This depends on the application. Both intersection and χ^2 give good performance. Intersection is a bit more robust. $\chi 2$ is a bit more discriminative. Euclidean distance is not robust enough. There exist many other measures: Bhattacharyya distance or the information theoretic Kullback-Leibler divergence.

Recognition using Histogram Algorithm

It's a simple algorithm and follows the nearest neighbour strategy:

1. Build a set of histograms $H = \{M_1, M_2, M_3, \dots\}$ for each known object. More precisely, for each view of each object.

2. Build a histogram T for the test image.

3. Compare T to each $M_k \in H$, using a suitable comparison measure.

4. Select the object with the best matching score or reject the test image if no object is similar enough.

Colour Histograms

They work surprisingly well in favourable conditions.

Advantages:

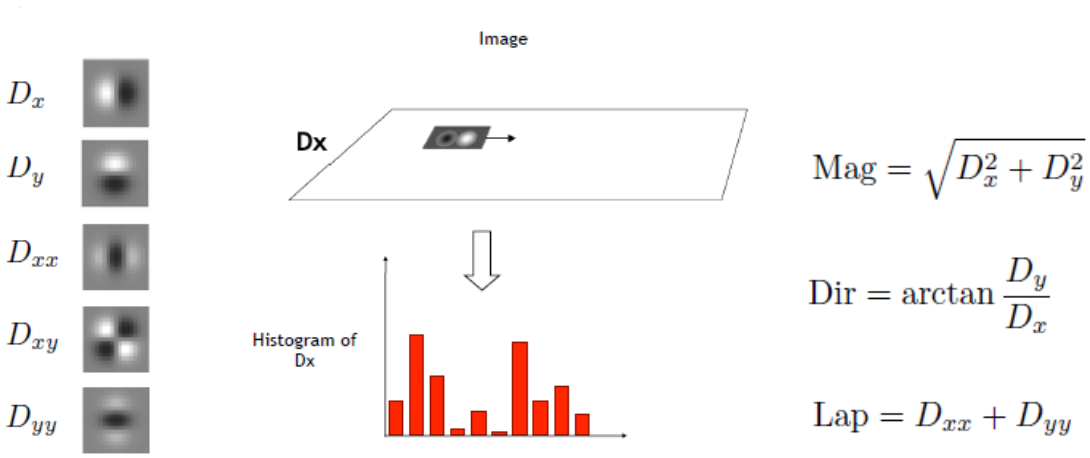
- Invariant to object translations
- Invariant to image rotations
- Slowly changing for out-of-plane rotations
- No perfect segmentation necessary
- Histograms change gradually when part of the object is occluded
- Possible to recognize deformable objects

Problems:

- The pixel colours change with the illumination („colour constancy problem") meaning the intensity and spectral composition (illumination color)
- Not all objects can be identified by their color distribution.

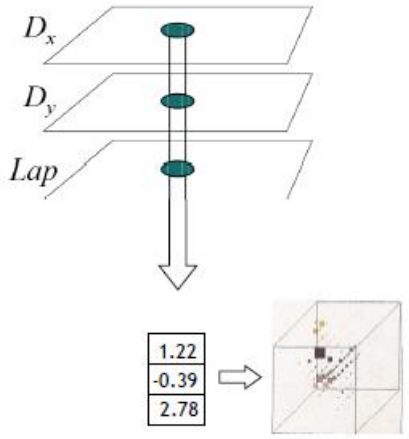
Generalization of the idea

We can build histograms using derivatives or any general receptive field histograms like local descriptors (e.g., filter, filter combination). Such examples are the gradient magnitude, gradient direction, or the Laplacian.



Multidimensional Histograms

Combination of several descriptors. Each descriptor is applied to the whole image. Corresponding pixel values are combined into one feature vector. Feature vectors are collected in a multidimensional histogram.



2.2. Sparse feature representations

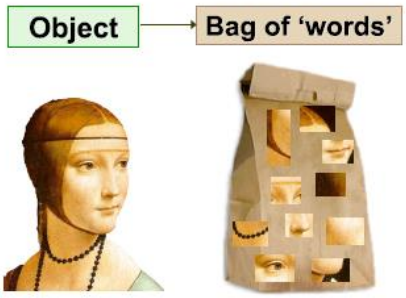
This is how this works:

1. Find key points with interest point detector e.g., scale- or affine-invariant
2. Represent key points with feature vectors e.g., SIFT, ShapeContext

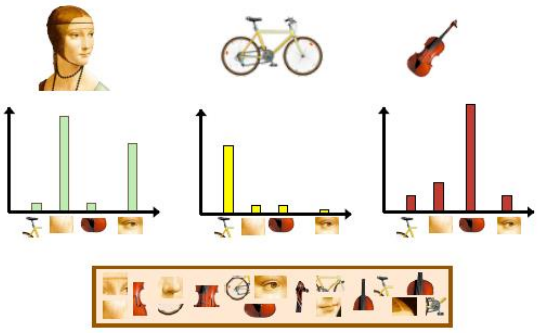
3. Bag-of-Words Model

3.1. Overview

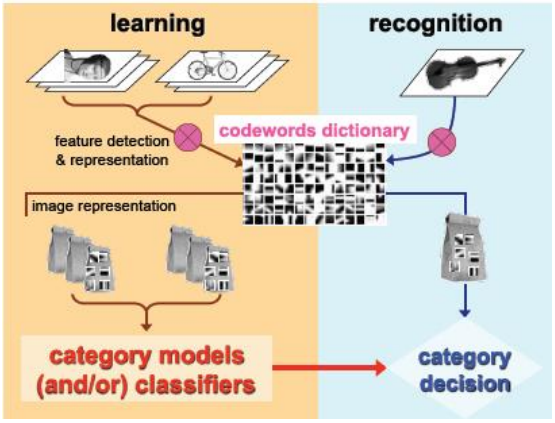
Assume we want to recognize an object like e.g., a face. We know a face has specific features like 1 nose, 2 eyes etc. The idea is to build a dictionary of visual words called "*Bag of words*" which contain these specific features of the face (nose, eyes etc.).



To recognize the object as a face we count how often each of these visual words appear in the object we want to recognize.



However, to construct a dictionary, we need example images first! The bag of words model consists of two phases: *Learning Phase* and *Recognition Phase*.



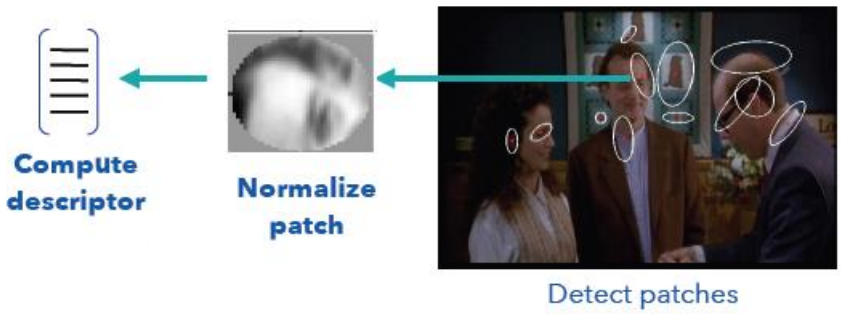
3.2. Object Representation & Learning

The learning phase is done in 3 steps:

1. Feature detection and representation
2. Codeword dictionary formation
3. Image representation

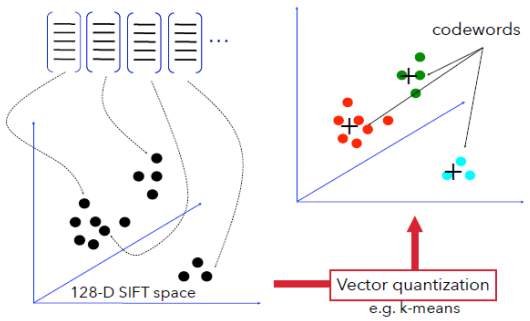
Feature detection and representation

This is where the sparse feature representation comes in. Detect patches with local interest operator (e.g., Harris-Laplace) or regular grid, normalize the patch and compute the feature descriptor e.g., SIFT, shape context, etc.



Codeword dictionary formation

The next step is Vector quantization. We must cluster all feature descriptors of all training images into codewords. For that we can use e.g., k-means.



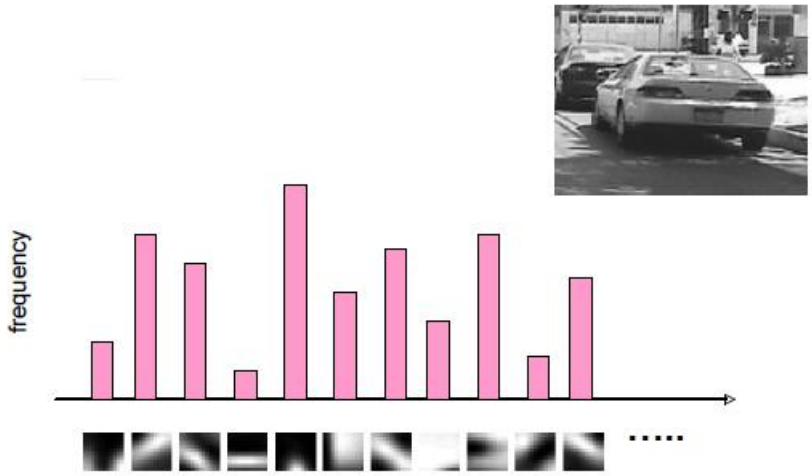
This is how K-means clustering works:

Algorithm 1 k-means algorithm

1: Specify the number k of clusters to assign.
2: Randomly initialize k centroids.
3: **repeat**
4: **expectation:** Assign each point to its closest centroid.
5: **maximization:** Compute the new centroid (mean) of each cluster.
6: **until** The centroid positions do not change.

Image representation

But we still don't know what these codewords or clusters represent. The image representation is basically a histogram how frequent visual words occur! But we are not doing it for all categories but only for the images of a certain category. Suppose we want to recognize car images. We take all these images and compute the visual words for it by detecting interest points then computing feature descriptors and then assigning these features to each cluster/bag of words by comparing them with each other.



The I do the same for dogs, cats or any other images we use! Now I know what the

Weiter:

Now new image same thinfinterst points feature descriptois but we dotn build a new dictionry we already have it form training

4. Machine Learning

Exam

Computervision 1 (Total 90 points) ~ 120min

Part 1. Explain concepts (4 general questions)

1. What's the main idea of non-maximum suppression?
2. ..
3. Why use lens instead of pinhole?
4. What is the use of Dimensionality Reduction in Object Recognition? (or something like that)

Part 2. Image formation

1. 2 characteristics of camera impact Field of View
2. Given distance from object to lens l1 2.5 cm, distance between lens and image plane is 1 cm, focal length is 0.5cm. Is the point in focus? -> apply Thin Lens Formula
3. Moiré effect: basically explain what it is and how to prevent it
4. Bayer grid and draw it on a sensor
5.

Part 3. Filter

1. Given a matrix ⅛*[[1,0,-1],[1,0,-1],[1,0,-1]], what's this filter name and explain 2 operations of it. <- Pretty sure that was the sobel filter instead of the differences
2. Name a pyramid, and why we use it
3. Name a non-linear filter and explain what can it be used for?
4. Given a matrix, apply 2 padding methods on the matrix.
   e.g.

		1	3		
		2	4		

Part 4. PCA

1. Draw a 2 dimensional dataset where PCA doesn't work
2. Which formula PCA optimises? And given a dataset X, how PCA represents it.
3. List 2 advantages of using low-dimensional data
4. Assuming the mean of data is subtracted, how to compute PCA via SVD?

Part 5. Interest points

1. How to make harris detector more generalised ? - (3d harris for mri images)
2. What's the pros and cons of Harris detector?
3. How to determine if two interest points are in different matches?
4. How does harris detector work?
5.

Part 6. Single/two view geometry

1. [Something about why should we use conditioning and how.]
2. How does RANSAC work?

Part 7. Stereo

1. Why do we use image rectification?
2. How many correspondences does the essential matrix need? Explain
3. How are the fundamental matrix and the essential matrix related? (+what do you still need to go e.g. from fundamental to essentia matrices)
4.

Part 8. Motion

1. What's the differences between motion field and optical flow field.
2. How to solve aperture problem [under some condition] ?

Part 9. BoW model

1. Explain how Bayers theorem is applied for BoW representation in classification task
2. What's pros and cons of colour histogram?

Part 10 Deep learning

1. Assuming you are designing a 5 layers neural network. Which 2 layers would you like to stack to the top of the network (first two layers), 2 fully-connected layers or 2 convolutional layers? Why?
2. Draw the network (2 input, 3 hidden 1 output) and give a formula for hidden layers(h1,h2,h3) and y for the forward pass using the weights w11, w12,..., v1, v2, v3, the activation function is sigmoid (4points)
3. Define machine learning and classification/regression. <- i think this was task 9 (but the question is valid)
4. Briefly explain deep learning. What's advantages of using DL compared to traditional computer vision methods? + 2 Applications of DL for CV

Wise20/21

Exam Questions (90min, 67 points):

* What is the receptive field? How to compute its size (depending on number of layers n, and mxm window size)?

* What is the architectural novelty of ResNet ? Which problem does it solve ?

* Homography with DL: How to obtain training data ? Which Loss function do you use ?

* Why is CV an inverse problem ?

* Draw binocular setup, explain important definitions of epipoles

* How are Depth estimation and optical flow related ?

* How many correspondences are necessary for estimating the fundamental matrix ? Pixel coordinates are a bad choice for this estimation. How to obtain world coordinates ?

* Explain the degrees of freedom of the homography

* State the OFCE and name the assumptions

* When does window based matching fail. Name to examples

* Name two advantages of PCA

* PCA: Derive relationship between SVD and eigendecomposition

* What is camera calibration ? Explain the important steps of Camera calibration

* What is homogeneous least squares? How did we solve it in class?

* How are gaussian and laplacian pyramids conceptually related?

* What is the problem with the Harris Operator? How does HarLap solve it ?

* What is the problem with window based depth estimation? How to solve it ?

* Explain the steps of how BoW features are created

* Given K codewords in the dictionary, N classes and C images/feature vectors with dimension d, how many dictionaries are required to separate the N classes?

* Name two histogram similarity measures and explain how they are more robust than euclidean distance

* Explain how Lucas-Kanade can be applied to fast motions ?

* Explain perspective distortion

Last lecture

Assume that we change the position of a camera from one exposure to another. In order to describe the new imaging process, do we need to change the extrinsic or intrinsic matrix? What action can cause the other matrix to change?

Sketch a two-dimensional dataset that cannot be represented well by a PCA projection and explain why?

What is the Harris operator and how does it relate to the structure tensor?

What is the difference between the essential matrix and the fundamental matrix? 1 Which one has fewer degrees of freedom and why?

Problems aim at checking the understanding of important concepts and methods. No extensive computation/derivation but the key formulas are important you should know what they say and which assumptions were made to arrive there No Python programming There will be some open questions