Computer Vision 1

1. About

This is a summary or notes of the course ,,Computer Vision 1" taught by Professor Stefan Roth at Technical University of Darmstadt.

To shortly introduce myself: I am Ilyass, Mechanical Engineering student at Technical University of Darmstadt. I hope these notes help you somehow.

2. Course Plan

The following topics will be covered:

- Introduction and overview
- Image Formation
- Basics of 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. Lecture Objectives

Get intuitions about

- What is computer vision?
- Why is it useful?
- Why is it hard / interesting?
- How might we go about it?

But why not only use deep learning? Why learn the first principles of vision? Understanding first principles is essential because they offer concise explanations when deep learning falls short, provide insights when models fail, enable data synthesis when collection is impractical, and satisfy our innate curiosity about how things work.

2. 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.

3. Why is it useful?

Computer Vision has many applications for:

- Face detection
- Human pose estimation
- Earth viewers
- Photo browsing
- Optical character recognition (OCR)
- Driver assistance systems
- Special effects (Shape Capture, Motion Capture, Sports)
- Vision in space
- Robotics
- Medical imaging

4. Why is it hard & 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.

Computer Vision allows you to apply theoretical skills that you may otherwise only use rarely. It is quite rewarding since often visually intuitive and encouraging results.

It is a growing field: Cameras are becoming increasingly commonplace, there are several vision companies both here and abroad, even the big players are hiring computer vision experts, conferences are growing rapidly.

5. 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?

Let's first look at artistic cues to interpret what is an image. For example, edges, colour, texture, contrast, composition, and the use of light and shadow etc.

We as humans use many different cues to interpret what is in an image. For example:

- Stereo parallax (parallax means the observed displacement of an object caused by the change of the observer's point of view.)
- Motion parallax
- Shadows
- Convergence
- Context

However, 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

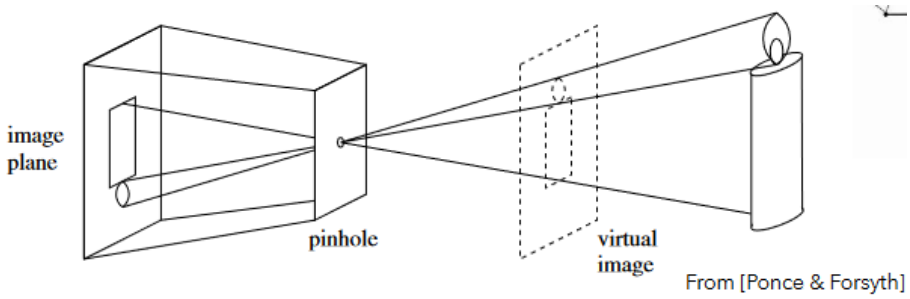
1. Introduction to Cameras

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 do 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".

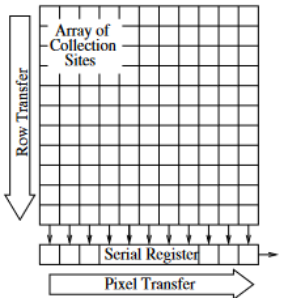
This describes exactly how the **pinhole camera** (idealized standard camera model) works as seen in the figure below:



From [Ponce & Forsyth]

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. Adding a barrier with an **opening** (pinhole, aperture*)* to block most of the rays **reduces blurring**.

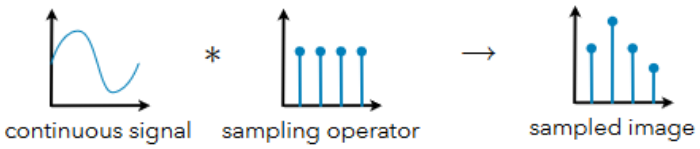
A pinhole camera is a model for many common sensors like human eyes, photographic cameras, X-ray machines etc. We usually work with the **upright virtual image**. For the **image plane** we use a CCD or CMOS Sensor:



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

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

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



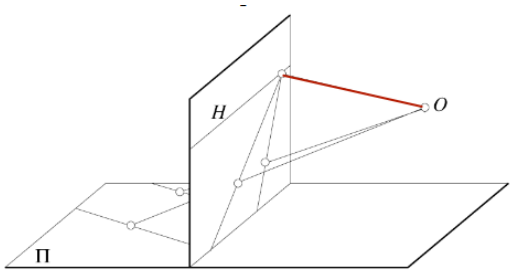
For our assumptions we will regard the image as **spatially discrete** as an array of pixels.

A pinhole camera is a dimensionality Reduction Machine (3D to 2D). What have we lost?

- **Angles**
- **Distances** (lengths)

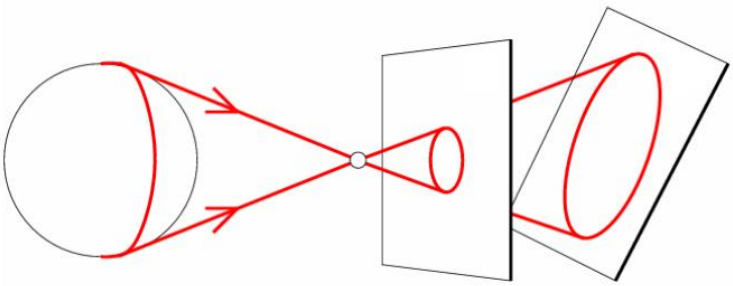
This leads us at some general projection properties of the pinhole camera:

- Many-to-none: All points along the same ray map to the same point in image
- Points → Points: The projection of points on **focal plane** (image plane) is undefined
- Lines → Lines (collinearity is preserved): **But** line 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?

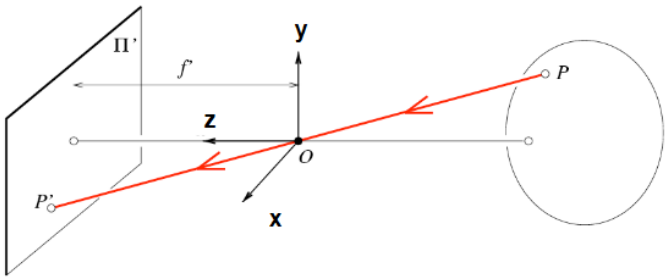
Perspective distortion is a common challenge for pinhole cameras. For example, let's look at the projection of a sphere. As one can see in the figure below if the **focal point is normal** to the image plane, we don't have a distortion otherwise the sphere turns into an oval shape.



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 (pointed out by Da Vinci).

2. Perspective projection

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:



Note that we are assuming a **virtual image** here! 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:

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

$f' = z'$ is the **projected depth**. But the division with z is **not a linear** transformation.

$$(x, y, z) \rightarrow (f' \frac{x}{z}, f' \frac{y}{z})$$

To make this a linear transformation 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** (augment a vector with a constant 1):

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

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

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

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

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

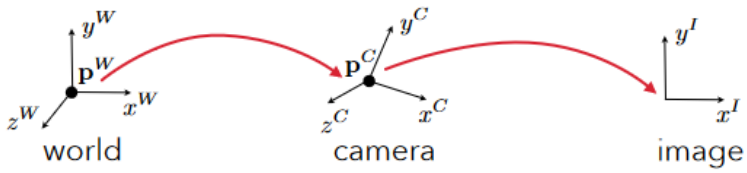
This leads us now to the **projection matrix** which is a matrix multiplication in homogeneous coordinates. After that we divide by the third coordinate to get Cartesian coordinates back!

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

3. Single View Geometry

We are considering 3 coordinate systems:

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

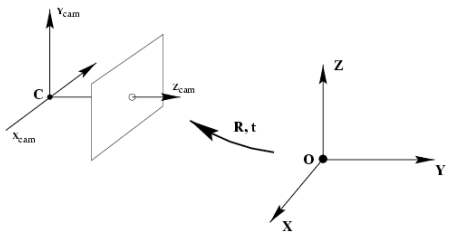


This leads us to two key transformations:

- **Extrinsic camera transformation**: Takes world into camera coordinates.
- **Intrinsic camera transformation:** Describes the **image formation process** and takes camera into image coordinates.

3.1. Extrinsic Camera Transformation

The **extrinsic camera transformation** is described as a **linear transformation** using homogeneous coordinates. The camera coordinate frame is related to the world coordinate frame by a **rotation** and a **translation**.



In **non-homogenous** coordinates we first obtain:

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

Where:

- $\tilde{\boldsymbol{c}}$: Coordinates of camera centre in the world frame (translation)
- $\boldsymbol{R}$: Rotation matrix
- $\tilde{\boldsymbol{x}}^W$**:** Coordinates of a point in the world frame
- $\tilde{\boldsymbol{x}}^C$: Coordinates of point in camera frame

Finally using **homogeneous coordinates** yields:

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

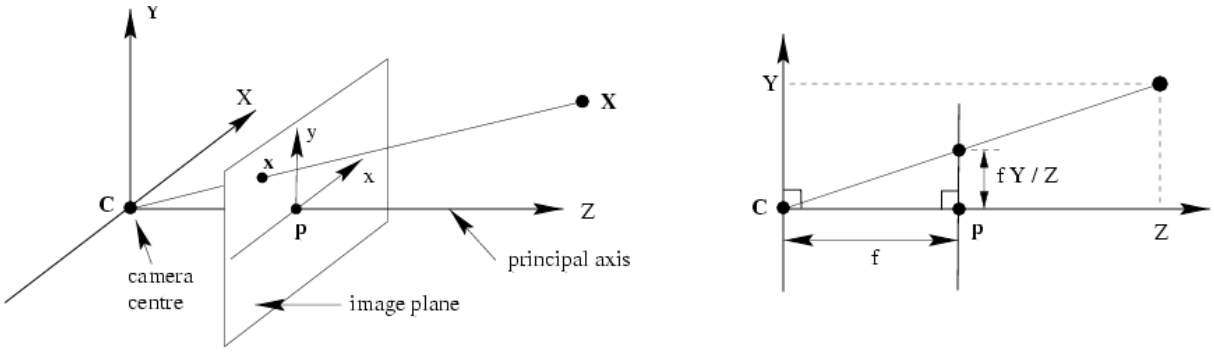
where the translation can be written as $\boldsymbol{t} = -\boldsymbol{R}\tilde{\boldsymbol{c}}$ and $\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ 0 & 1 \end{bmatrix} \in \mathbb{R}^{4x4}$.

3.2. Intrinsic Camera Transformation

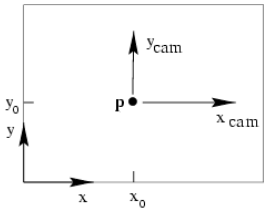
The **intrinsic camera transformation** is described as a linear transformation plus a perspective division. The pinhole camera model we are using is a special case, but the intrinsic camera transformation can deal with more general camera models as well.

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 next figure!



Let's recall the 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 $\boldsymbol{x}^I = \boldsymbol{K}[\boldsymbol{I}|0]\boldsymbol{x}^C = \boldsymbol{K}\tilde{\boldsymbol{x}}^C$, where $\boldsymbol{K} = diag\{f, f, 1\}$. 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:

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

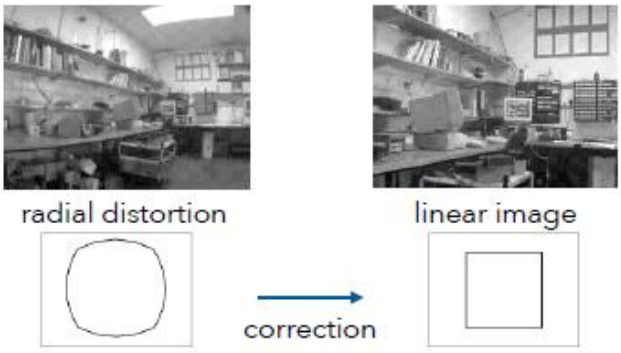
where $\boldsymbol{K} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix}$ is called the **calibration matrix.** However, we are not done yet. To finally get the pixel/image plane coordinates we need to do one more matrix multiplication with **pixel magnification factors**, which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

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

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 $\boldsymbol{K}$:

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

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

3.3. Perspective Projection Pipeline

Now we can concatenate the extrinsic and intrinsic camera transformations into a single projection matrix $\boldsymbol{P}$. In homogeneous coordinates we get:

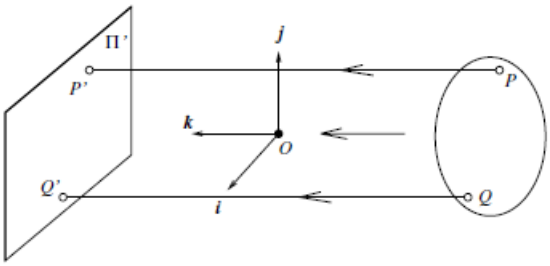
$$\boldsymbol{x}^I = \boldsymbol{K} \cdot [\boldsymbol{I}|\boldsymbol{0}] \cdot \boldsymbol{x}^C = \boldsymbol{K} \cdot [\boldsymbol{I}|\boldsymbol{0}] \cdot \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ 0 & 1 \end{bmatrix} \cdot \boldsymbol{x}^W = \boldsymbol{K} \cdot [\boldsymbol{R}|\boldsymbol{t}]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

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

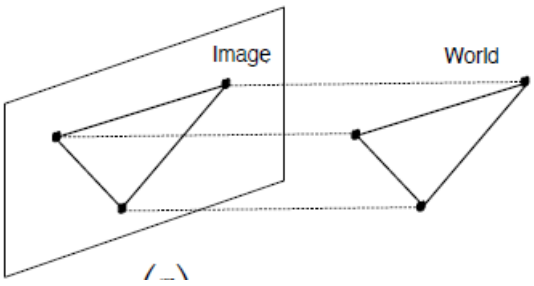
3.4. Orthographic Projection

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:

$$\begin{pmatrix} 1000 \\ 0100 \\ 0001 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \rightarrow (x, y)$$

3.5. 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** and again **homogenous coordinates***:*

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

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 $\boldsymbol{x_i}$ and $\boldsymbol{X_i}$ have the same direction and to compensate for not knowing the scale we just add a variable λ.

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

Rewriting the expanded cross-product to an equation system with **two linearly independent equations** 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 $\boldsymbol{A}$ 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** $\boldsymbol{A r} = 0$ (left nullspace would be $\boldsymbol{I^T A} = 0$) 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? See detailed derivation in chapter 3.7!

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:

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

2. Next, decompose $\boldsymbol{M}$ into upper triangular part $\boldsymbol{K}$ (calibration) and orthonormal part $\boldsymbol{R}$ (rotation) using RQ-decomposition.

3. If required, extract 3 rotation axis and angle from $\boldsymbol{R}$ in which its columns are the axes of the rotated coordinate system.

4. Finally, find $\boldsymbol{c}$ as the nullspace of $\boldsymbol{P}$ by means of SVD.

3.6. Camera calibration - Coplanar points

A set of points is coplanar if they all lie on the same geometric plane. For coplanar points that satisfy

$$\boldsymbol{\Pi}^{\boldsymbol{T}} X_i = 0$$

we will get degenerate solutions $(\boldsymbol{\Pi}, 0,0)$, $(0, \boldsymbol{\Pi}, 0)$ or $(0,0, \boldsymbol{\Pi})$.

$$\begin{bmatrix} 0 & -\boldsymbol{X_1^T} & y_1\boldsymbol{X_1^T} \\ \boldsymbol{X_1^T} & 0 & -x_1\boldsymbol{X_1^T} \\ \vdots & \vdots & \vdots \\ 0 & -\boldsymbol{X_n^T} & y_n\boldsymbol{X_n^T} \\ \boldsymbol{X_n^T} & 0 & -x_n\boldsymbol{X_n^T} \end{bmatrix} \begin{pmatrix} \boldsymbol{p_1} \\ \boldsymbol{p_2} \\ \boldsymbol{p_3} \end{pmatrix} = \boldsymbol{0} \rightarrow \boldsymbol{Ap} = \boldsymbol{0}$$

3.7. Homogeneous least squares: Introduction & Derivation

We would like to find the vector $\boldsymbol{p} \in \mathbb{R}^n$ that satisfies $\boldsymbol{Ap} = \boldsymbol{0}$, with $\boldsymbol{A} \in \mathbb{R}^{\mathrm{mxn}}$ matrix and $\boldsymbol{0} \in \mathbb{R}^{\mathrm{m}}$. We further assume that $m \geq n$ and that $rank(\boldsymbol{A}) = n$. Since the entries of $\boldsymbol{A}$ are measured and thus affected by noise, we rather minimize the (squared) error that we are making:

$$\|\boldsymbol{Ap}\|^2$$

To exclude the trivial solution $\boldsymbol{p} = \boldsymbol{0}$, we additionally constrain the norm of $\boldsymbol{p}$ as:

$$\|\boldsymbol{p}\| = 1$$

This leads us to a constrained optimization problem or **constrained least squares** such that we find $\boldsymbol{p}$ that minimizes $\|\boldsymbol{Ap}\|^2$ subject to $\|\boldsymbol{p}\| = 1$. Rewriting the constraint as $1 - \boldsymbol{p^T p} = 0$ allows us to write the problem using a **Lagrange multiplier** and derive the **necessary condition**:

$$\frac{\partial}{\partial \boldsymbol{p}}\left(\boldsymbol{p^T A^T Ap} + \lambda\left(1 - \boldsymbol{p^T p}\right)\right) = 0$$

$$\leftrightarrow 2\boldsymbol{A^T Ap} - 2\lambda\boldsymbol{p} = 0$$

The Eigenvalue problem $\left(\boldsymbol{A^T A}\right)\boldsymbol{p} = \lambda\boldsymbol{p}$ implies that $\boldsymbol{p}$ is an eigenvector of $\boldsymbol{A^T A}$ and λ an eigenvalue.

Which eigenvector/eigenvalue pair do we have to pick? Recall that we are minimizing $\|\boldsymbol{Ap}\|^2 = \boldsymbol{p^T A^T Ap}$ this means that we can rewrite the squared error as $\|\boldsymbol{Ap}\|^2 = \boldsymbol{p^T} \lambda \boldsymbol{p} = \lambda$, where the last step used the constraint $\|\boldsymbol{p}\| = 1$.

This leads us to **homogenous least squares**: To minimize $\|\boldsymbol{Ap}\|^2$ subject to $\|\boldsymbol{p}\| = 1$, we thus choose $\boldsymbol{p}$ as the eigenvector of $\boldsymbol{A^T A}$ with the **smallest** eigenvalue λ.

We need to perform **singular value decomposition** (SVD), i.e. $\boldsymbol{A} = \boldsymbol{USV^T}$, where $\boldsymbol{U} \in \mathbb{R}^{\mathrm{mxn}}$ and $\boldsymbol{V} \in \mathbb{R}^{\mathrm{nxn}}$ are orthonormal and $\boldsymbol{S} \in \mathbb{R}^{nxn}$ is a diagonal matrix with descending singular values along the diagonal. Now we have that

$$\boldsymbol{A^T A} = \left(\boldsymbol{USV^T}\right)^{\boldsymbol{T}} \boldsymbol{USV^T} = \boldsymbol{VS^T U^T USV^T} = \boldsymbol{VS^2 V^T}$$

This leads us to:

- The eigenvectors of $\boldsymbol{A^T A}$ are the right singular vectors of $\boldsymbol{A}$ (i.e. columns of $\boldsymbol{V}$).
- The eigenvalues of $\boldsymbol{A^T A}$ are the square of the singular values of $\boldsymbol{A}$

To find the eigenvector of $\boldsymbol{A^T A}$ with the smallest eigenvalue, we compute the **last right-singular vector** of $\boldsymbol{A}$**.**

3.8. Important Conclusions

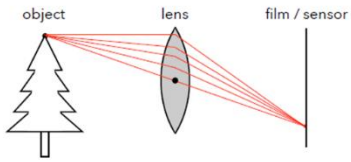
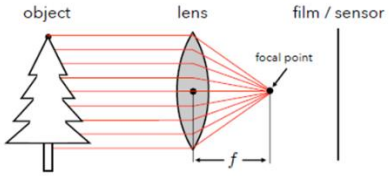
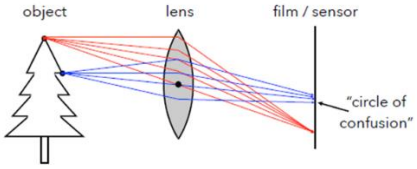
What have we got now? We have a projection matrix $\boldsymbol{P}$, which maps any point in the 3D world coordinate frame onto the (infinite) image plane of the camera. **Given a 3D point in the world, we can find out where its projection is in the image.**

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 $\boldsymbol{x}$**.** 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).

Adding a lens focuses light onto the film. The next table depicts some important properties.

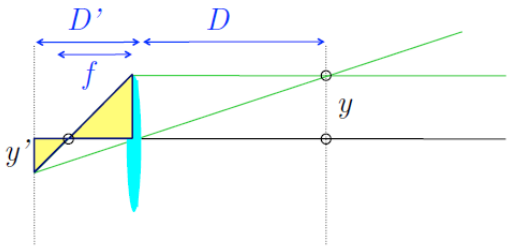
Rays passing through the center are not deviated. 	Rays converge to one point on a plane located at the focal length f. 	There is a specific distance at which objects are "in focus". Other points project to a "circle of confusion" in the image 

4.1. Thin Lens Formula

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

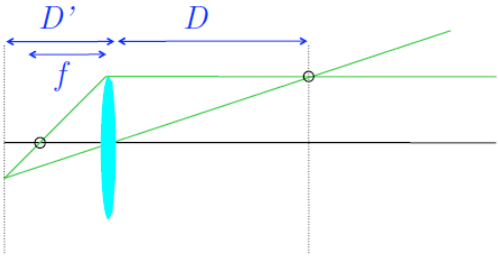
$$\frac{y'}{y} = \frac{D'}{D}$$

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



Any point satisfying the thin lens equation is in **focus:**

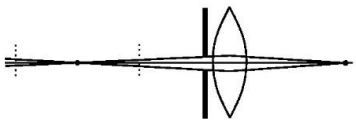
$$\frac{1}{D'} + \frac{1}{D} = \frac{1}{f}$$



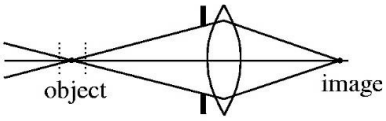
4.2. Depth of Field

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)!



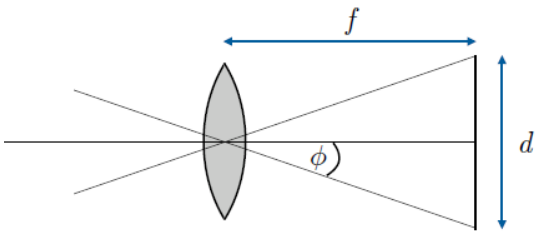
Increasing the aperture makes the depth of field smaller but requires lesser exposure (to the light). You see we have a **tradeoff**!



4.3. Field of View

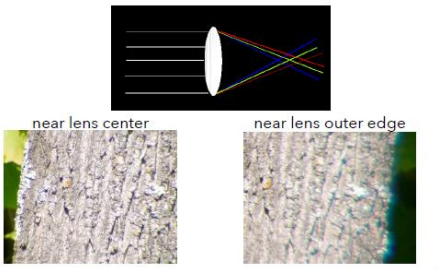
Field of View (FOV) is the angle that's visible and depends on **focal length** and the **size of the camera retina** (sensor). A **smaller FOV** equals a **larger focal** length (camera is far from object). A **larger FOV** equals a **smaller focal** length (camera is close to object).

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

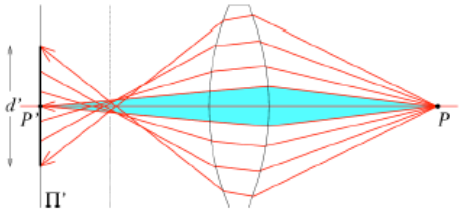


4.4. Lens Flaws

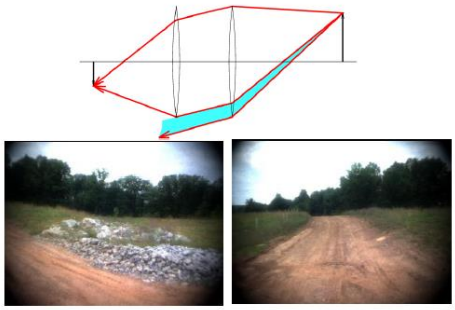
Chromatic aberration: Lens has different refractive indices for different wavelengths. This causes color fringing.



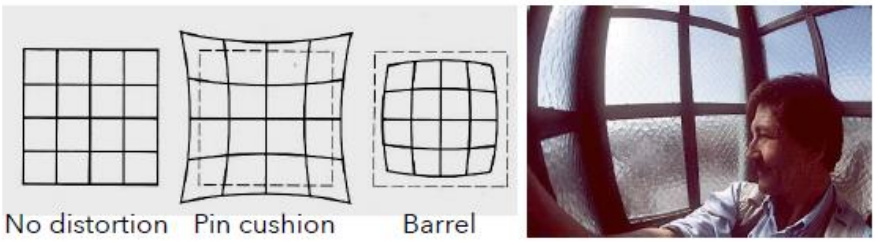
Spherical aberration: Spherical lenses don't focus light perfectly. Rays farther from the optical axis focus closer which causes blur away from the image center.



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



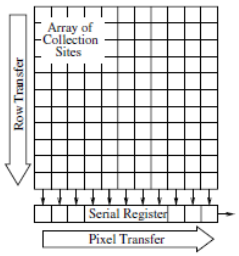
Radial Distortion: Caused by imperfect lenses. Deviations are most noticeable for rays that pass through the edge of the lens.



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).

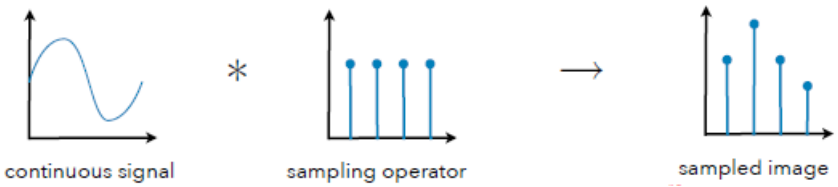
Images arriving at our CCD or CMOS sensor are spatially discrete with individual pixels.



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

$$I(x,y), I\colon \mathbb{R}\times\mathbb{R}\to\mathbb{R}$$

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:



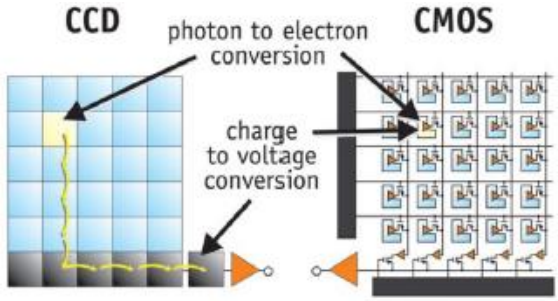
For more details investigate signal processing and Fourier theory. For our purposes here, we will regard the image as spatially discrete. So, an array of pixels.

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!



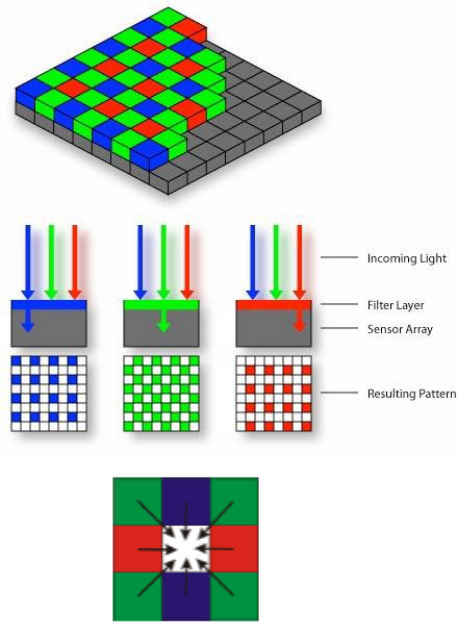
CCDs move photogenerated charge from pixel to pixel and convert it to voltage at an output node. CMOS imagers convert charge to voltage inside each pixel.

5.2. Color Sensing in Camera

A camera captures color using a Bayer grid or color filter array, where each individual pixel of the sensor is sensitive to a specific color: red, green, or blue. The distribution of these colors typically favors green (50% of the pixels), as the human eye is more sensitive to this color, with the remaining pixels evenly split between red and blue (25% each).

However, each pixel only captures one color component, not an entire RGB triplet. To obtain the full color information for each pixel, a process known as demosaicing is used. This process estimates the missing color components for each pixel based on the values of neighboring pixels.

While this method allows for accurate color capture, it does come with a high computational cost due to the complexity of the demosaicing process.



The problem with Bayer sensors and demosaicing is **color moire** which is a visual effect that happens when **two similar patterns** overlap, creating new, wavy, or unwanted stripes of color that go across a photo that wasn’t originally there like shown in the following picture:

The cause of **color moiré** is fine black and white detail in images misinterpreted as color information. We can avoid this using an **optical low-pass filter** (artificial blur).

Other options for color sensing in cameras are:

- **Prisms:** Requires three chips and precise alignment, more expensive, big & heavy
- **Foveon X3:** CMOS sensor which takes advantage of the fact that red, blue, and green light penetrate silicon to different depths. This results in better image quality.
- **X-Trans:** CMOS sensor that uses color filters as in Bayer sensor, but with pseudo-random spatial arrangement. It avoids some demosaicing artefacts that are due to the regular Bayer grid. It requires more complex demosaicing procedure.

5.3. Issues with digital cameras

- **Noise:** low light is where you most notice noise, light sensitivity (ISO) / noise tradeoff, stuck pixels
- **Resolution:** Are more megapixels better? Requires higher quality lens, noise issues
- **In-camera processing:** Oversharpening can produce halos
- **RAW vs. compressed:** file size vs. quality tradeoff
- **Blooming:** charge overflowing into neighboring pixels, white balance

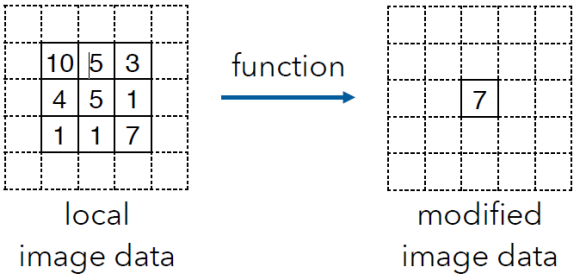
5.4. Historical Context

- Pinhole model: Mozi (470-390 BC), Aristoteles (384-322 BC)
- Principles of optics (including lenses): Alhazen (965-1039 AD)
- Camera obscura: Leonardo da Vinci (1452-1519), Johann Zahn (1631-1707)
- First photo: Joseph Nicephore Niepce (1822)
- Daguerréotype (1839)
- Photographic film (Eastman, 1889)
- Cinema (Lumière Brothers, 1895)
- Color photography (Lumière Brothers, 1908)
- Television (Baird, Farnsworth, Zworykin, 1920s)
- First consumer camera with CCD: Sony Mavica (1981)
- First fully digital camera: Kodak DCS100 (1990)

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:

$$I(x,y), I\colon \mathbb{R}\times\mathbb{R}\to\mathbb{R}$$

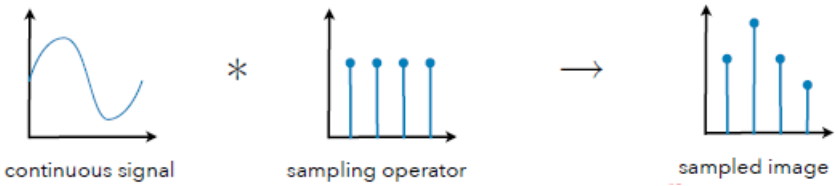
For every 2D point it tells us what the (light) intensity is. Realistically, the size of the sensor is limited and so is the range of brightness it can capture:

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

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 we 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

3.1. Linear Filtering

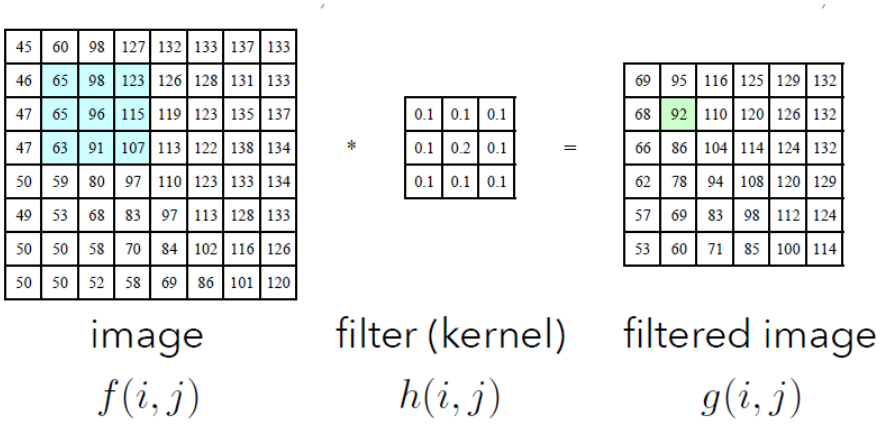
Let's start by defining basic properties of linear filtering, meaning linear operations or systems:

- 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.

3.1.1. 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 (Is the output smaller?):



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!

3.1.2. 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 ($\otimes$ 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 " $+$ ".

3.1.3. Why 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.

$$\begin{array}{|c|c|c|} \hline 2 & 3 & 3 \\ \hline 3 & 20 & 2 \\ \hline 3 & 2 & 3 \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|} \hline 2 & 3 & 3 \\ \hline 3 & 3 & 2 \\ \hline 3 & 2 & 3 \\ \hline \end{array}$$

There are two main types of linear filters which can be used to **remove noise**:

- Box Filters (a special case of the Average Filter)
- Gaussian Filter

3.1.4. Average and Box 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.

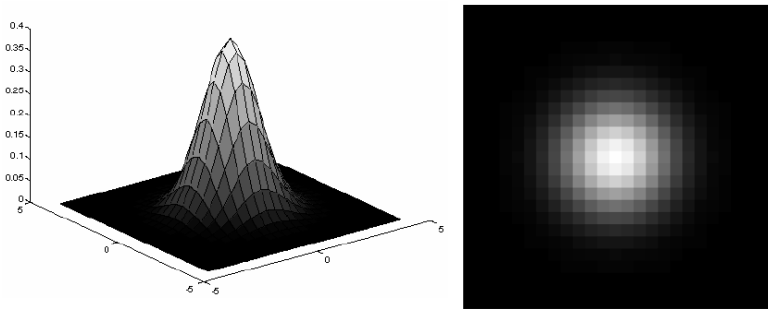
$$\frac{1}{9} \cdot \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

3.1.5. Gaussian Filter

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:

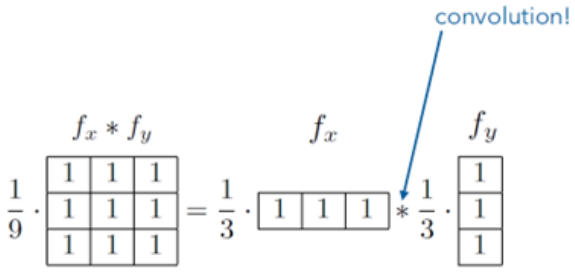


3.1.6. Efficient Implementation of Box and Gaussian filters

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!

3.1.7. 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 for example padded with 0 values wherever the convolution kernel extends beyond the original image boundaries! How can we compensate for this? Or what are good boundary handling strategies?

- **Zero:** Set all pixels outside the source image to 0
- **Wrap:** loop around 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.**

3.2.1. Median 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.

3.2.2. Morphological filters

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** $\theta(f * s, 1)$: Makes the original image thicker
- **Erosion** $\theta(f * s, S)$: Makes the original image thinner, 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

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

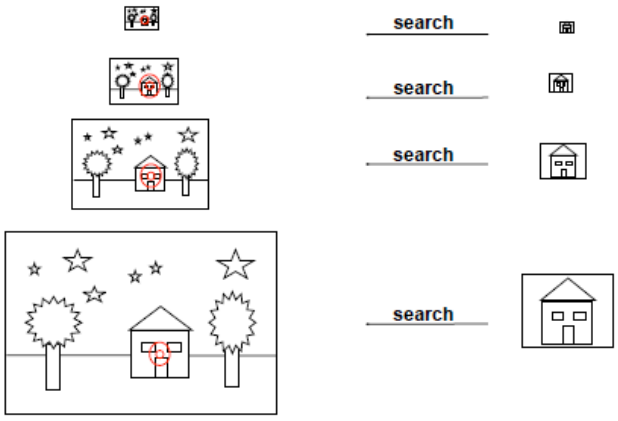
erosion

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

4. Multi-scale image representation

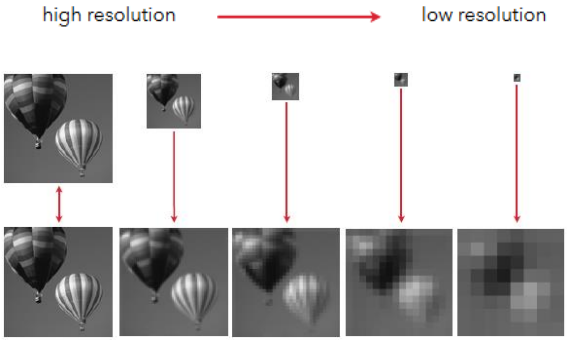
We often may wish to change the resolution of an image before proceeding. For example, to match a computer screen or the output of a printer. Alternatively, we also want to reduce the size of an image resolution to speed up the execution of an algorithm or to save on storage space. Sometimes we also don't know the right resolution an image should be.

For example, the task to find a house in an image. In this case we don't know the scale at which the house will appear, so we need to generate a whole pyramid of differently sized images and search each one for possible faces as illustrated in the next figure.



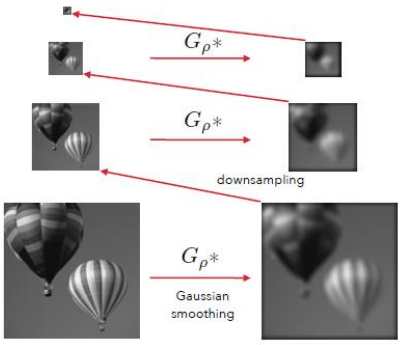
4.1. Gaussian pyramid

A Gaussian pyramid represents images at multiple scales (resolutions) stacked up like a pyramid. By resolution we mean the number of pixels of an image. Here is an example:



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)



There are two important questions to answer:

- Which information is preserved over scales?
- And which information is lost over scales?

4.1.1. Aliasing

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:

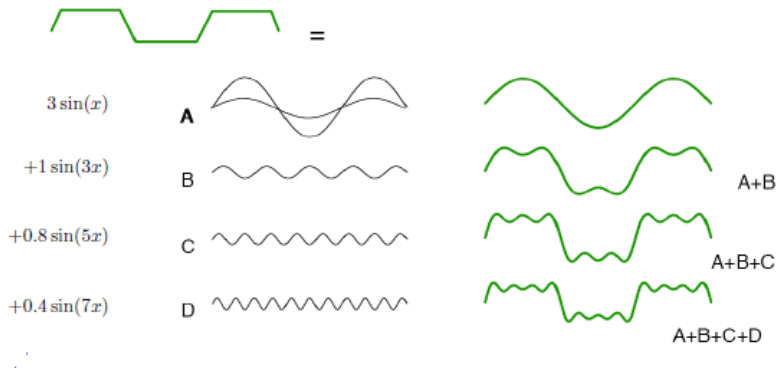
Spatial frequencies are misinterpreted. This is called **Aliasing!**

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.



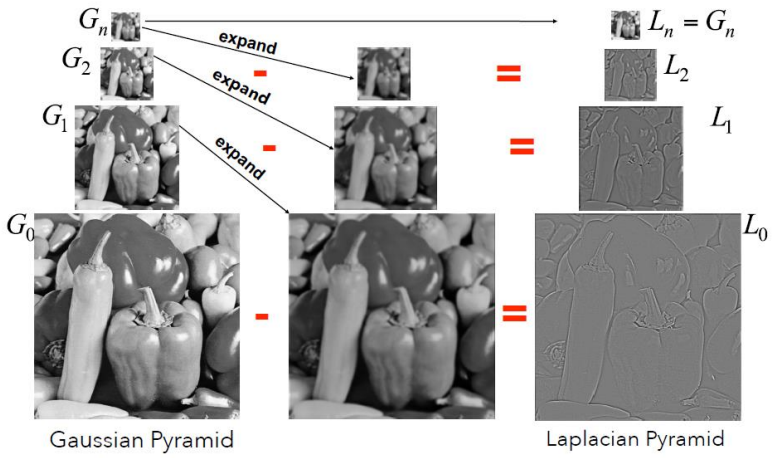
The **Laplacian pyramid** provides a simple **frequency decomposition into sub bands**. Each level/sub band contains only image structure of a particular range of spatial frequencies e.g., the finest level contains all the high-frequency detail. Laplacian Pyramids are formed from the Gaussian Pyramids. There is no exclusive function for that. Laplacian pyramid images are like edge images only. Most of its elements are zeros. They are used in image compression. A level in Laplacian Pyramid is formed by the difference between that level in Gaussian Pyramid and expanded version of its upper level in Gaussian Pyramid:

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

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.



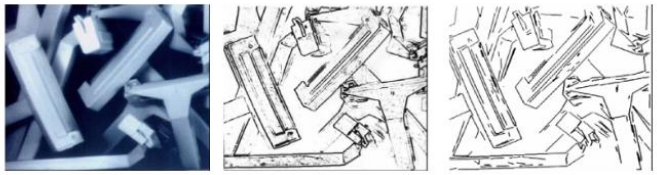
5. Edge detection

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

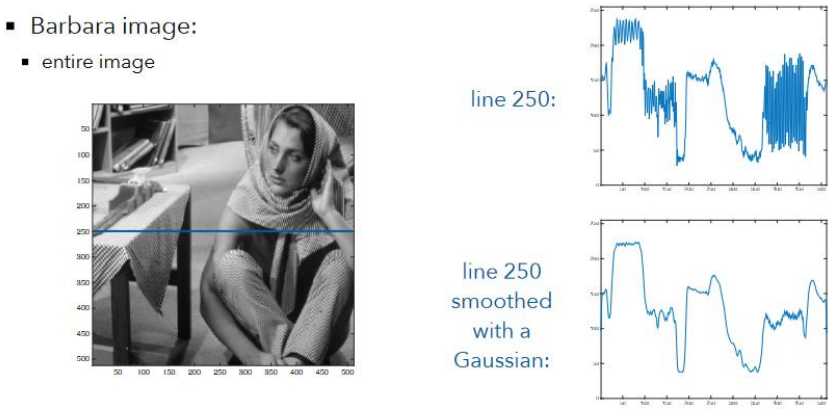


3. Project model into the image and match to lines (solving for 3D pose)

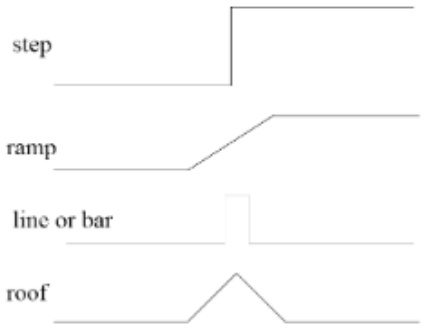
Another mostly historical approach is the matching of models (wireframe/geons/generalized cylinders...) to edges and lines. The only remaining problem to solve is to reliably extract lines & edges that can be matched to these models (wireframe, cylinders etc.).

5.2. Image Scanline & Edges

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.

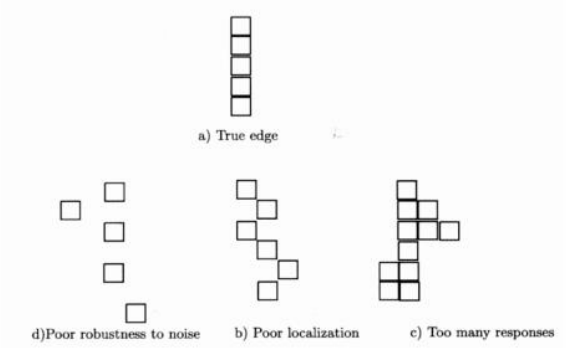


The question to ask is what are edges (1D) exactly or how do they look in the intensity function meaning how do the jumps look? Idealized edge types are steps, ramps, line or bar and roof as depicted in the next figure:

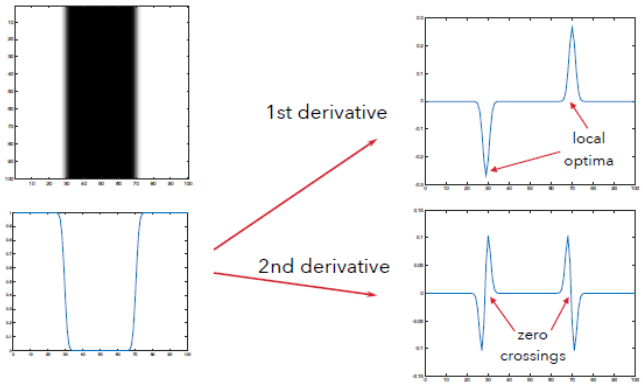


The **goals of edge detection** are:

- **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)



Edges correspond to fast changes in the intensity function where the magnitude of the derivative is large.



We differentiate between:

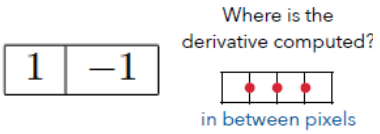
- **1D Edge Detection**: Find local optima in the 1st derivative and also use a threshold. But this is not enough it gets us thick edges but not thin edges but it's a good first idea!
- **2D-Edge Detection**: Calculate gradient (magnitude and direction), then do non-maximum suppression, then do hysteresis!

5.3. 1D Edge Detection

To compute the 1st derivative is pretty easy using **forward differences** (convolutional filter):

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

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!

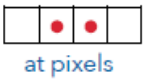


The 1st derivate can also be computed using **central differences**.

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

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:

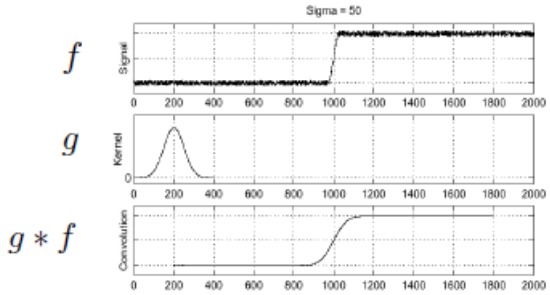
$$\frac{1}{2}\cdot\boxed{1}\boxed{0}\boxed{-1}$$



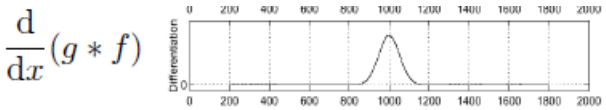
5.3.1. Implementing 1D Edge Detection

Algorithmically: Find peaks in the 1st derivative: $\frac{d}{dx}(g * f)$

1. Smooth with Gaussian



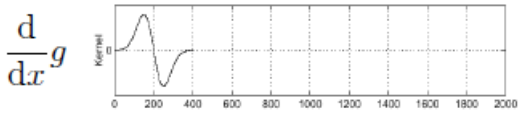
2. Calculate derivative



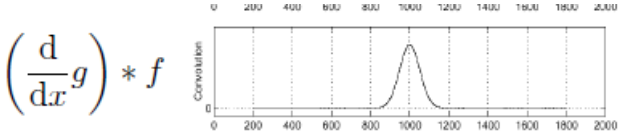
3. Finds its local optimum **and** it should be **sufficiently** large!

To **save one 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



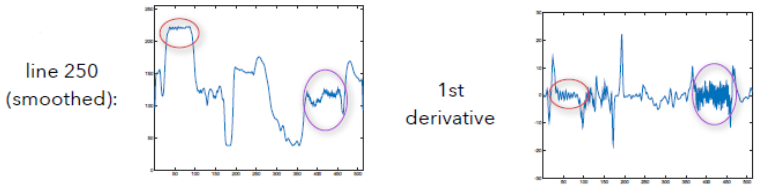
2. Smooth with Gaussian derivative



3. Finds its local optimum **and** it should be **sufficiently** large!

5.3.2. Problems

The 1st derivative **amplifies** small variations!



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):

- high threshold (of absolute value) to start edge curve
- low threshold to continue them.

This only makes sense in 2D! Hold on now this leads us to **2D edge detection**!

5.4. 2D Edge Detection

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

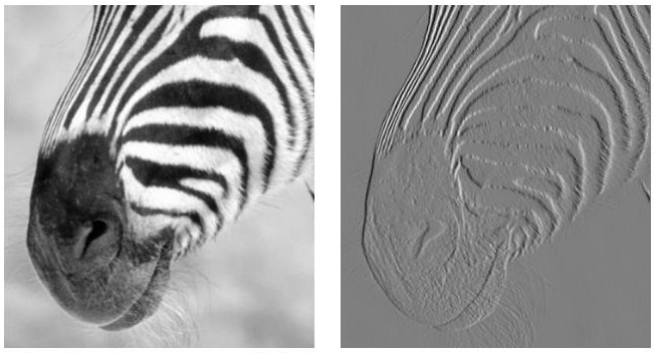
$$\frac{\partial}{\partial x} I(x,y) = I_x \approx D_x * I$$

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

Partial derivatives are often approximated with simple filters (finite differences):

Box Filter	**Sobel Filter** (Gaussian smoothing in opposite direction)
$D_x = \frac{1}{6} \cdot \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$	$D_x = \frac{1}{8} \cdot \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$
$D_y = \frac{1}{6} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$	$D_y = \frac{1}{8} \cdot \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$

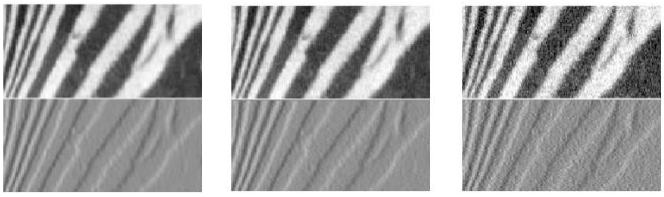
Question: In which direction did we perform the derivative in the image below? And did we convolution or correlation?



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)

5.4.1. The problem of noise (again!)

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):



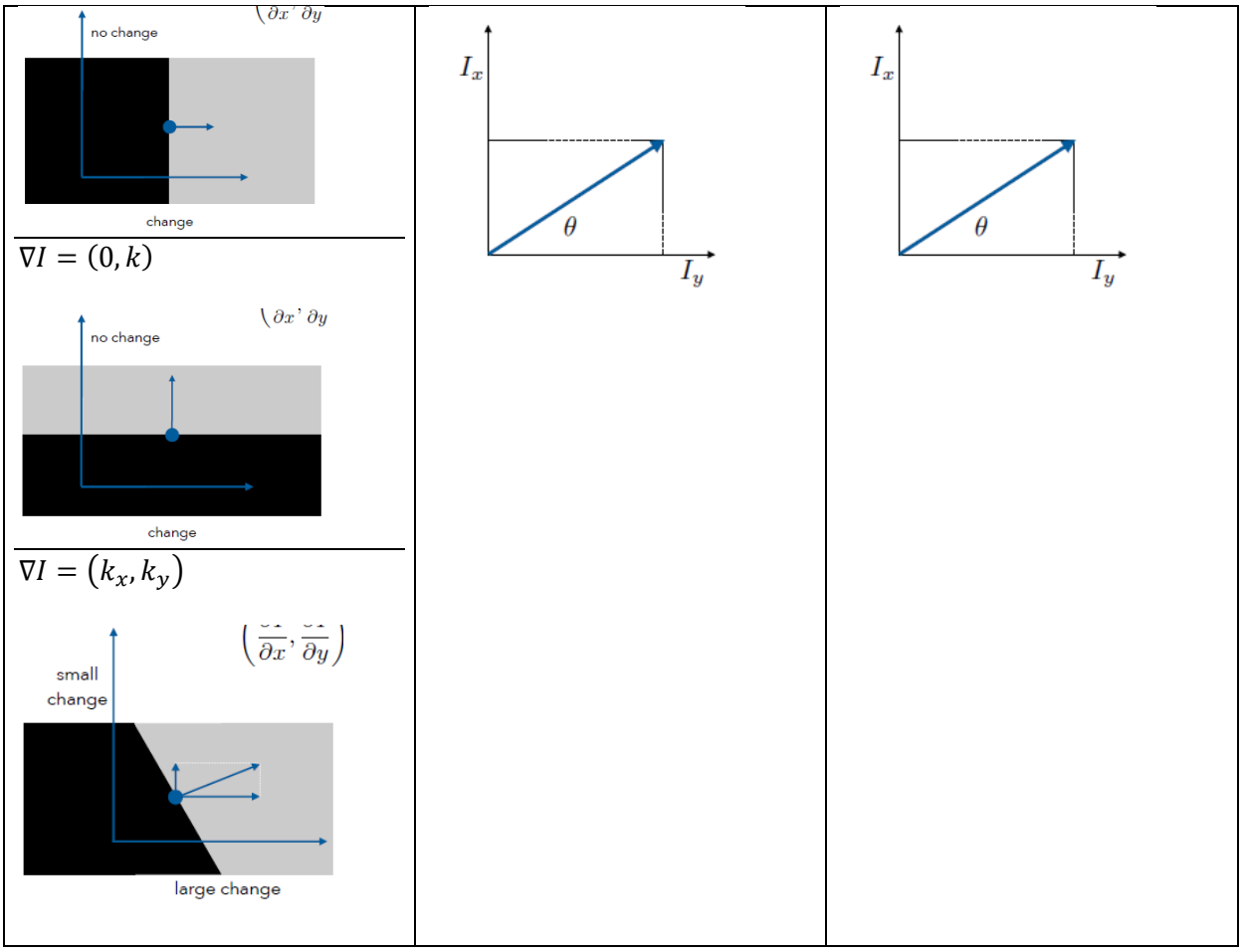
To solve this problem, we can use **gaussian smoothing**. This explains why gaussian smoothing is so important for edge detection not only in the direction where we don't compute the derivative but also in the direction of the derivative itself! Hence the partial derivate for example in x-direction would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

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

5.4.2. Gradient

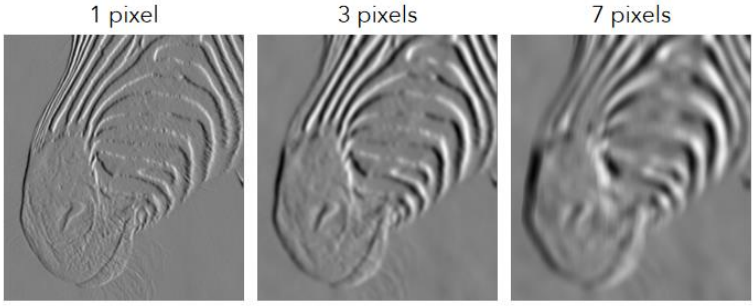
The **gradient vector** points in the direction of maximum change (steepest ascent) in the intensity function:

Gradient	**Magnitude** (measures edge strength)	**Direction** (perpendicular to edge)
$\nabla I = (I_x, I_y) = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right)$ $\nabla I = (k, 0)$	$\|\nabla I\| = \sqrt{I_x^2 + I_y^2}$	$\theta = \tan^{-1}\left(\frac{I_y}{I_x}\right)$



5.4.3. Problems

The **scale** of the smoothing filter affects derivative estimates and also the semantics of the edges recovered: **Strong edges persist across scales** as seen in the next figure!



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**!

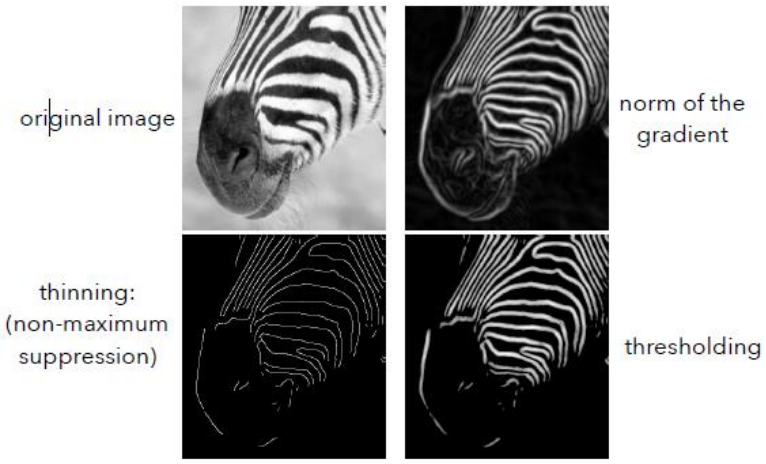
5.4.4. Optimal Edge Detection: Canny

Assuming we are only doing **linear filtering** of an image and we have **additive i.i.d. gaussian noise** in an image. Considering that edge detection as already mentioned earlier should have:

- **good detection**: filter responds to edge, not to noise
- **good localization**: detected edge near true edge
- **single response**: one per edge

then according to Canny (1986) the optimal edge detector is approximately 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:



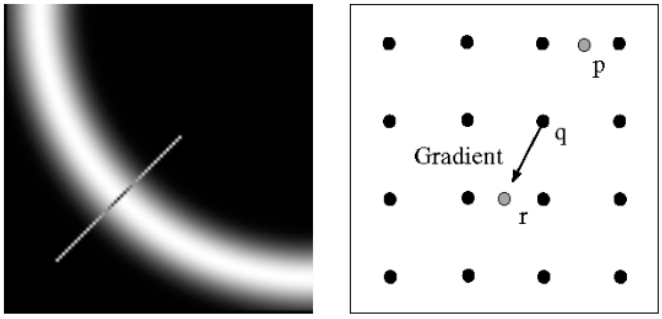
Using only the 1st derivative leads to thick edges. So how can we improve the single response! We can get thinner edges by using **non-maximum suppression**.

5.4.5. Non-maximum suppression

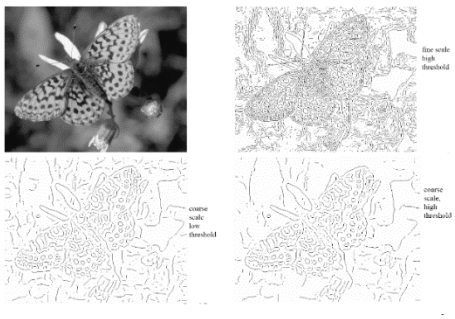
After getting **gradient** magnitude and direction, a **full scan** of image is done to remove any unwanted pixels which may not **constitute** to the edge. For this **every** pixel is checked if it is a local maximum in its neighbourhood in the direction of the gradient. Here is how it works:

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!



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!



5.4.6. Hysteresis

One way to solve the problems of non-maximum suppression 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 (minVal).
   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!

5.4.7. Full Procedure

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

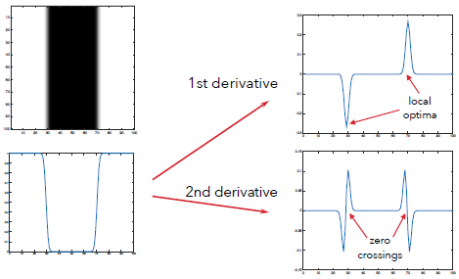
5.5. Edge Detection using Laplacian

So far, we only used the 1st derivative for edge detection (finite differences) with the respective mask:

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

1	−1

However as already mentioned the 2nd derivative can also tell us the location of the edges (local optima) with the **zero-crossing** as seen in the next figure again.



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:

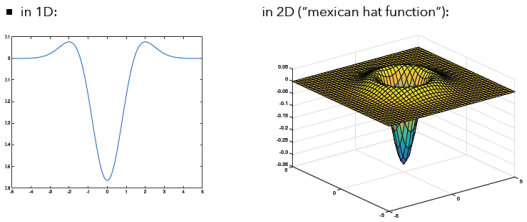
1	−2	1

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

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

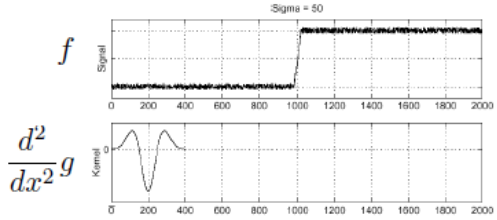
Again, we can apply this as a linear filter (Sobel filter). To save one operation we can use the association rule by deriving the gaussian first and then do a convolution with the image (instead of smoothing the image first with a gaussian and the deriving it). This is computationally cheaper! The **Laplacian of Gaussian** (LoG) is then the term in the brackets:

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

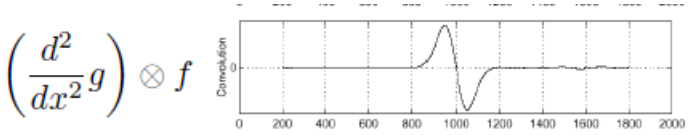


The procedure is as follows:

1. Calculate **LoG**

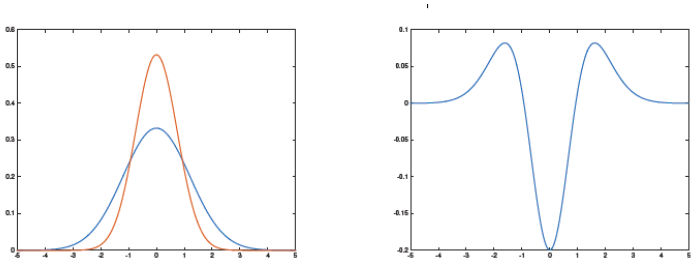


2. Convolve **LoG** with image signal



3. Find local optimum

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

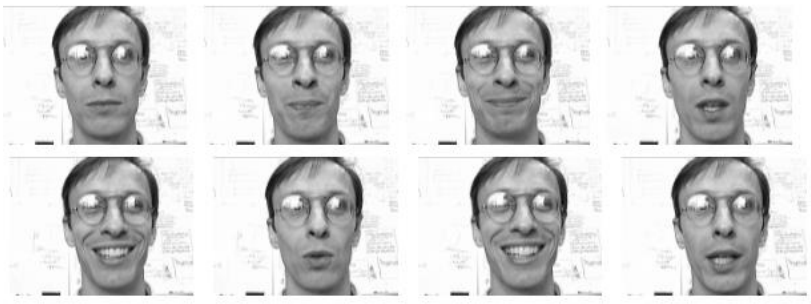


$\sigma_2 = 1.6 \cdot \sigma_1$ (other choices possible)

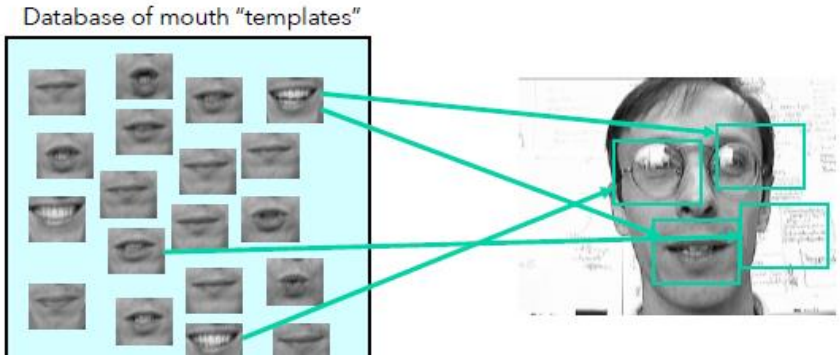
View-Based Recognition

1. Motivation

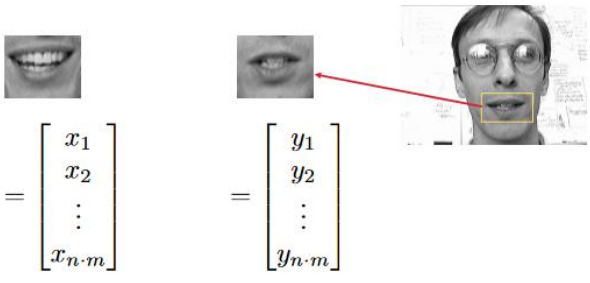
This chapter deals with view-based recognition using template methods. Simple line-based object models are not as easy as it seems. Let us look at a more constrained setting:



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 for this approach!

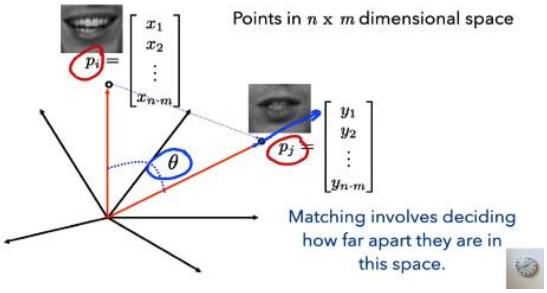


Images can be represented as vectors and points. The best way to represent pixels are vectors. From this point on we will think of images as arrays which we will reshape with standard lexicographic ordering to a vector:



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)!



1.1. Template methods (matching)

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 find effects they look like
- We can use filters to match a template against an image

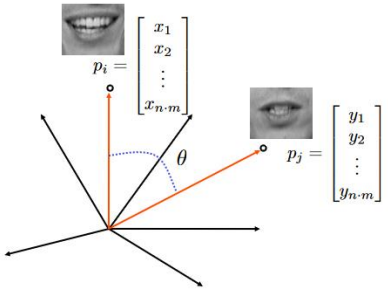
To obtain the angle of 2 vectors we can use the dot product e.g., normalized correlation, which then measures similarity between 2 vectors:

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

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:

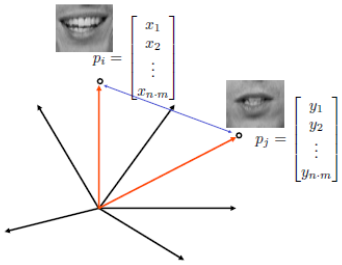
$$\cos(\theta) = \frac{\boldsymbol{a}^T\boldsymbol{b}}{|\boldsymbol{a}||\boldsymbol{b}|}$$

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:

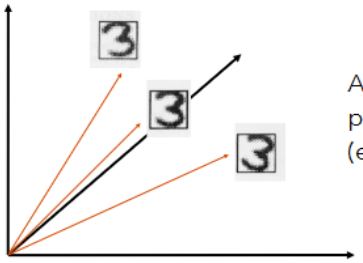
$$E(I,T) = \sum_{i,j} \big(I(i,j) - T(i,j)\big)^2$$



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.

2. Linear dimensionality reduction

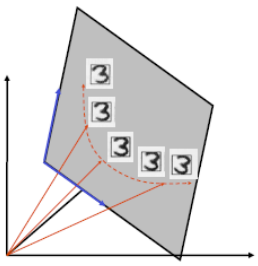
How can we find more efficient representations for the ensemble of views and more efficient methods for matching? The observation is that images are not random. Especially images off the same object (category) have similar appearance:



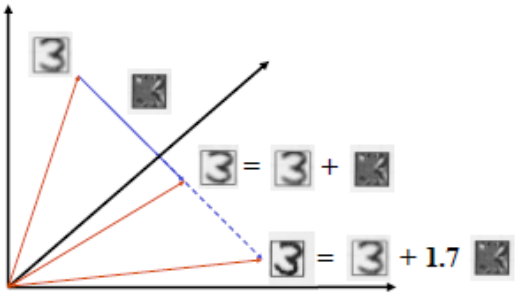
Assume images to be represented as points in a high-dimensional space (e.g., one dimension per pixel)

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.

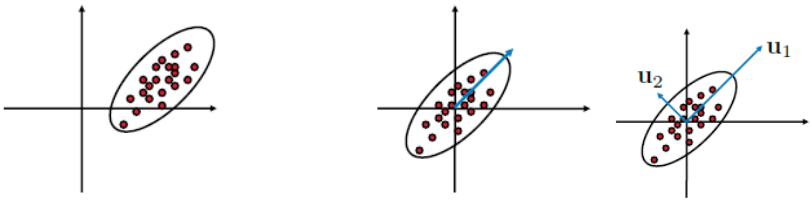
Question: What linear transformations of the images can be used to define **a lower-dimensional subspace** that captures most of the structure in the image ensemble?



Basic idea: Given that differences are structured, we can use **basis images** to transform images into other images in the **same space**. In this case we subtracted 2 images to get 1 basis image (blue vector). Look how scaling the basis image can construct a new picture! We need to find these basis images, but it's not that simple!



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.



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!

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:

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

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, $\sum_{j=D+1}^{M} b_j \boldsymbol{u_j}$ is the error, a_i describes the marginal variance and $\boldsymbol{u}_i$ are the basis vectors. We want to find the D bases vectors that minimize the mean squared error over the training data:

$$\arg\min E(\boldsymbol{u}_1, \dots, \boldsymbol{u_D}) = \sum_{n=1}^{N} \|\boldsymbol{x}^n - \tilde{\boldsymbol{x}}^n\|^2$$

The following rules are important to understand the next formulations:

$$(1)\ \|\boldsymbol{x}\|^2 = \boldsymbol{x}^T\boldsymbol{x}$$

$$(2)\ \boldsymbol{u}^T\boldsymbol{u} = 1 \text{ (to avoid trivial solutions!)}$$

$$(3)\ a_n = \boldsymbol{u}^T\boldsymbol{x}^n$$

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

$$\begin{aligned} E(\boldsymbol{u}) &= \sum_{n=1}^{N} \|\boldsymbol{x}^n - \tilde{\boldsymbol{x}}^n\|^2 \\ &= \sum_{n=1}^{N} \left\|\boldsymbol{x}^n - \left(\boldsymbol{u}^T\boldsymbol{x}^n\right)\boldsymbol{u}\right\|^2 \quad (3) \\ &= \sum_{n=1}^{N} \|\boldsymbol{x}^n\|^2 - 2\left(\boldsymbol{u}^T\boldsymbol{x}^n\right)^2 + \left(\boldsymbol{u}^T\boldsymbol{x}^n\right)^2 \cdot \boldsymbol{u}^T\boldsymbol{u} \quad (1), (2) \\ &= \sum_{n=1}^{N} \|\boldsymbol{x}^n\|^2 - \left(\boldsymbol{u}^T\boldsymbol{x}^n\right)^2 = \sum_{n=1}^{N} \|\boldsymbol{x}^n\|^2 - (a_n)^2 \quad (3) \end{aligned}$$

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:

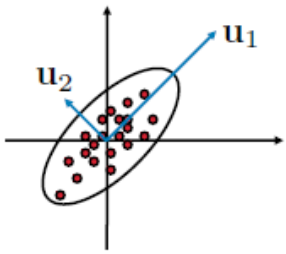
$$x^n - \bar{x}$$

Rewriting the prior decomposition of datapoint n:

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

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

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



But how do we find the axis of **largest variance**? With Principal Component Analysis (PCA)!

2.1. Principal Component Analysis (PCA)

The first principal direction $\boldsymbol{u}_1$ is the direction along which the variance of the data is maximal, i.e. it maximizes:

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

The second principal direction $\boldsymbol{u_2}$ maximizes the variance of the data in the orthogonal complement of the first principal direction etc. How do we get to this end goal? Let's derive this!

2.1.1. Derivation of PCA Objective

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:

$$\bar{a} = \frac{1}{N}\sum_{n=1}^{N} a_n = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{u^T x_n} = \frac{1}{N}\boldsymbol{u}^T \left[\sum_{n=1}^{N} \boldsymbol{x}^n\right] = \boldsymbol{u}^T \bar{\boldsymbol{x}}$$

The variance of the projection can be calculated as follows:

$$\begin{aligned}
\sigma^2 &= \frac{1}{N}\sum_{n=1}^{N} (a_n - \bar{a})^2 \\
&= \frac{1}{N}\sum_{n=1}^{N} \left(\boldsymbol{u^T x_n} - \boldsymbol{u}^T \bar{\boldsymbol{x}}\right)^2 \\
&= \frac{1}{N}\sum_{n=1}^{N} \left(\boldsymbol{u^T} (\boldsymbol{x_n} - \bar{\boldsymbol{x}})\right)^2 \\
&= \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{u^T} (\boldsymbol{x_n} - \bar{\boldsymbol{x}})(\boldsymbol{x_n} - \bar{\boldsymbol{x}})^T \boldsymbol{u} \\
&= \boldsymbol{u^T} \left[\frac{1}{N}\sum_{n=1}^{N} (\boldsymbol{x_n} - \bar{\boldsymbol{x}})(\boldsymbol{x_n} - \bar{\boldsymbol{x}})^T\right] \boldsymbol{u} \\
&= \boldsymbol{u}^T \boldsymbol{C} \boldsymbol{u}
\end{aligned}$$

$\boldsymbol{C}$ is the **Covariance** matrix. Covariance is a measure of how much two random variables vary together (like the height of a person and the weight of a person in a population).

To maximize σ^2 under the constraint $\|\boldsymbol{u}\| = 1$, we will use the Lagrangian function:

$$F(u; \lambda) = \boldsymbol{u^T C u} - \lambda(\|\boldsymbol{u}\|^2 - 1) = \boldsymbol{u^T C u} - \lambda(\boldsymbol{u^T u} - 1)$$

$$\frac{\partial F(u;\lambda)}{\partial u} = 2Cu - 2\lambda u = 0$$
$$\leftrightarrow Cu = \lambda u, \|u\| = 1$$

To solve this equation system, we compute eigenvectors and eigenvalues of C by using:

- Eigendecomposition
- Singular Value Decomposition

The largest **eigenvalue** is the maximal variance. The corresponding **eigenvector** is the direction with the maximal variance.

2.1.2. Eigendecomposition

Let the eigenvectors and eigenvalues of $\boldsymbol{C}$ be $\boldsymbol{u}_k$ and λ_k for $k \leq M$, i.e.,

$$\boldsymbol{C}\boldsymbol{u}_k = \lambda_k \boldsymbol{u}_k$$

with $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_M$. In matrix form we can write this as:

$$\boldsymbol{CU} = \boldsymbol{U\Lambda}$$

Where $\boldsymbol{\Lambda} = diag(\lambda_1, \dots, \lambda_M)$ and $\boldsymbol{U} = [\boldsymbol{u}_1, \dots, \boldsymbol{u}_M]$. Because $\boldsymbol{U}$ is orthonormal (we usually assume that the eigenvectors have unit norm) we know that

$$\boldsymbol{U}\boldsymbol{U}^{\boldsymbol{T}} = \boldsymbol{I}$$

Since $\boldsymbol{C}$ is also **real**, **symmetric**, and **positive-definite**, we can decompose it as:

$$\boldsymbol{C} = \boldsymbol{U\Lambda}\boldsymbol{U}^{\boldsymbol{T}}$$

$\boldsymbol{U}$ consists of orthonormal columns which are the eigenvectors of $\boldsymbol{C}$ and the diagonal elements of $\boldsymbol{\Lambda}$ are their respective eigenvalues!

2.1.3. Solving PCA

Now back to solving the PCA. If $\lambda_k \approx 0$ for $k > D$ for some $D \ll M$ then we can use the subset of the first D eigenvectors to define a basis for approximating the data vectors.

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

Where $a_i = \boldsymbol{u_i^T}(\boldsymbol{x^n} - \overline{\boldsymbol{x}})$. This representation has the **minimal mean squared error** (MSE) of all linear representations of dimension :

$$\min E(\boldsymbol{u}_1, \dots, \boldsymbol{u_D}) = \sum_{n=1}^{N} \|\boldsymbol{x}^n - \tilde{\boldsymbol{x}}^n\|^2$$

Note that axes of the projection are orthogonal and decorrelate the data, i.e. in the coordinate frame of these axes, the data is uncorrelated (side note: this only works for Gaussians).

Now we know how we can represent our data in a lower dimensional space in a principled fashion.

1. Compute the mean of the data and subtract it.
2. Compute the covariance matrix, decompose it, and choose the first D eigenvalues (Eigendecomposition)
3. This gives us an (eigen)basis for representing the data:

$$\boldsymbol{a}^n = \boldsymbol{B}^T(\boldsymbol{x}^n - \bar{\boldsymbol{x}}), \boldsymbol{B} = [\boldsymbol{u_1}, \dots, \boldsymbol{u_D}]$$

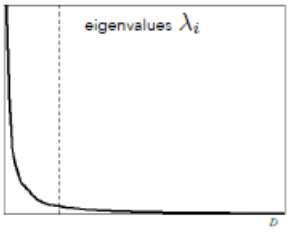
$$\tilde{\boldsymbol{x}}^n = \bar{\boldsymbol{x}} + \boldsymbol{B}\boldsymbol{a}^n$$

It is common to also normalize the variance of each dimension. But how do we choose D? A Larger D leads to better approximation. There are at least 2 good possibilities for choosing D:

- First, choose D based on application performance, i.e. choose the smallest D that makes the application work well enough.
- Second, choose D so that the eigenbasis captures some fraction of the variance, say

Remember: Eigenvalue λ_i describes the marginal variance captured by $\boldsymbol{u_i}$. Choose D using:

$$\sum_{i=1}^{D} \lambda_i \geq \eta \sum_{i=1}^{M} \lambda_i$$



2.1.4. PCA in Practice

Suppose we want to find a low-dimensional representation for mouth images. Let's assume the images are of moderate size $128x128$. How big is the covariance matrix?

The data vector has $2^7 \cdot 2^7 = 2^{14}$ dimensions. So, the covariance matrix has 2^{28} entries. This takes up 2^{31}Bytes. i.e. 2GB. This is way too large to handle data efficiently. What can we do to reduce this?

- **Observation 1**: While we may work in spaces that have 100000s of dimensions, we often only have 100s or 1000s of examples.
- **Observation 2:** We do not need to compute all eigenvectors and eigenvalues. Computing a few dozen of the largest ones usually suffices.

But how do we exploit this? Use Singular Value Decomposition (SVD)

2.1.5. Singular Value Decomposition (SVD)

We can rewrite PCA as follows. Given the data vectors in matrix form $\boldsymbol{X} = [\boldsymbol{x^1}, \dots, \boldsymbol{x^N}]$, we first subtract the mean:

$$\bar{\boldsymbol{x}} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{x}^n$$

$$\widehat{\boldsymbol{X}} = \boldsymbol{X} - [\bar{\boldsymbol{x}}, \dots, \bar{\boldsymbol{x}}]$$

Using that we can express the covariance matrix differently:

$$\widehat{\boldsymbol{C}} = \frac{1}{N}\sum_{n=1}^{N} (\boldsymbol{x_n} - \bar{\boldsymbol{x}})(\boldsymbol{x_n} - \bar{\boldsymbol{x}})^T = \frac{1}{N}\widehat{\boldsymbol{X}}\widehat{\boldsymbol{X}}^{\boldsymbol{T}}$$

Recall: SVD is a generalization of the eigendecomposition to rectangular matrices $\widehat{\boldsymbol{X}} \in R^{MxN}$:

$$\widehat{\boldsymbol{X}} = \boldsymbol{USV}^{\boldsymbol{T}}$$

$\boldsymbol{U}$ consists of left-singular vectors (orthonormal columns), the diagonal entries of $\mathbf{S}$ are singular values (non-negative) and $\boldsymbol{V}^T$ consists of right-singular vectors (orthonormal rows).

To use SVD for PCA we decompose the data matrix $\widehat{\boldsymbol{X}}$ with SVD:

$$\begin{aligned}
&\frac{1}{N}\widehat{\boldsymbol{X}}\widehat{\boldsymbol{X}}^{\boldsymbol{T}} = \frac{1}{N}\boldsymbol{USV}^{\boldsymbol{T}}\left(\boldsymbol{USV}^{\boldsymbol{T}}\right)^{\boldsymbol{T}} \\
&= \frac{1}{N}\boldsymbol{USV}^{\boldsymbol{T}}\boldsymbol{VS}^{\boldsymbol{T}}\boldsymbol{U}^{\boldsymbol{T}} \\
&= \frac{1}{N}\boldsymbol{USS}^{\boldsymbol{T}}\boldsymbol{U}^{\boldsymbol{T}} \\
&= \boldsymbol{U}\left(\frac{1}{N}\boldsymbol{S}^2\right)\boldsymbol{U}^{\boldsymbol{T}} \\
&= \boldsymbol{U\Lambda U}^{\boldsymbol{T}}
\end{aligned}$$

That the same thing from the homogenous least squares method! we perform SVD on the data matrix (after subtracting the mean) then:

- The **left-singular** vectors give us the **eigenvectors** of the covariance matrix.
- The **singular values** let us easily compute the **eigenvalues** of the covariance matrix:

The advantage is that we **never** have to explicitly build and store the covariance matrix!

$$\lambda_i = \frac{1}{N}s_i^2$$

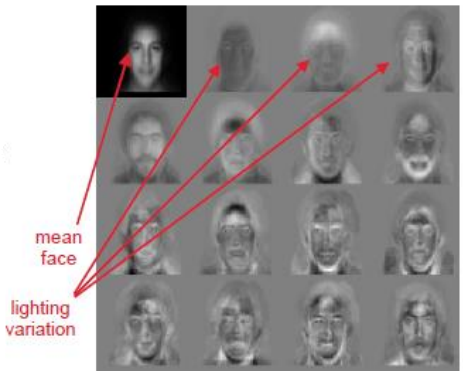
How does this help us now? $\boldsymbol{S} \in R^{MxN}$ is of manageable size now, and so is $\boldsymbol{V} \in R^{NxN}$. But $\boldsymbol{U} \in R^{MxM}$ is still much too large.

- **Observation 1:** We only need the first N left-singular vectors, because only N singular values can be non-zero. “Economy” decomposition, makes it all manageable memory-wise and computationally.
- **Observation 2:** We will typically need even fewer than N singular values and left-singular vectors. Special variants that only compute a partial decomposition

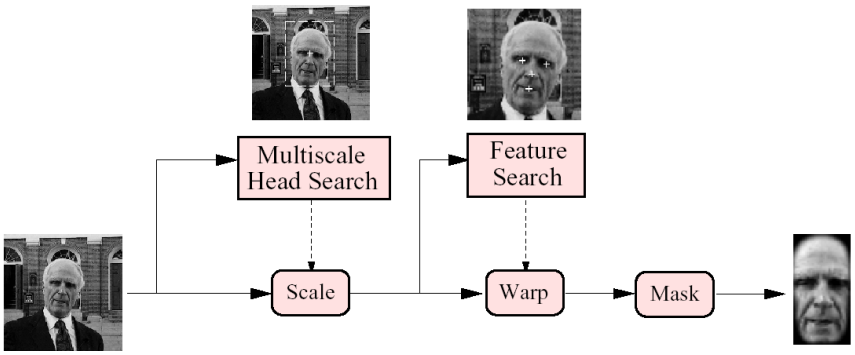
2.1.6. Eigenfaces

First popular use of PCA for object recognition was for the detection and recognition of faces [Turk and Pentland, 1991]:

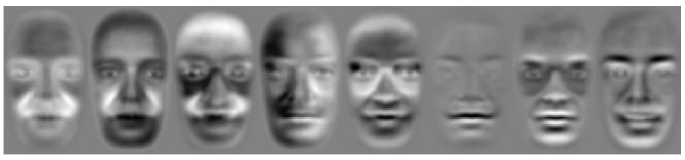
1. Collect a face ensemble
2. Normalize for contrast, scale, & orientation.
3. Remove backgrounds
4. Apply PCA & choose the first D eigen-images that account for most of the variance of the data.



Face Recognition [Moghaddam, Jebara & Pentland, 2000]:

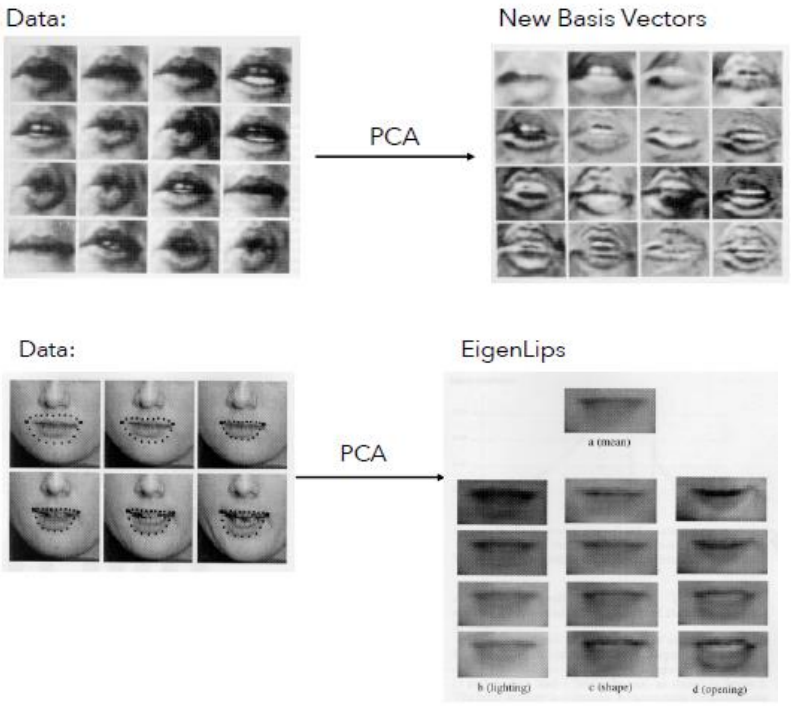


Intra-personal subspace (variations in expression)



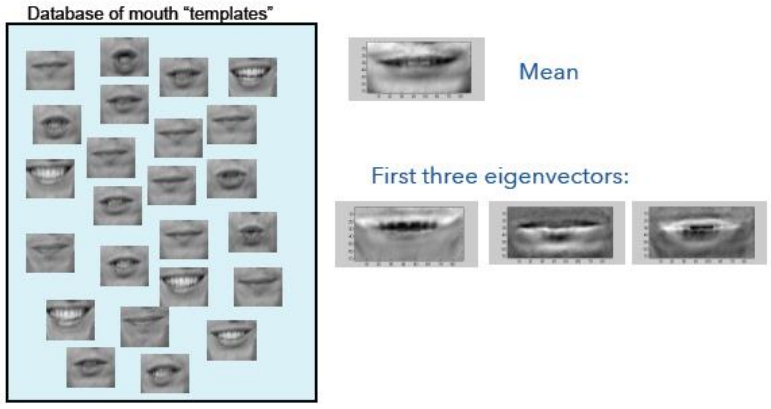
Extra-personal subspace (variation between people)

Eigen Features:



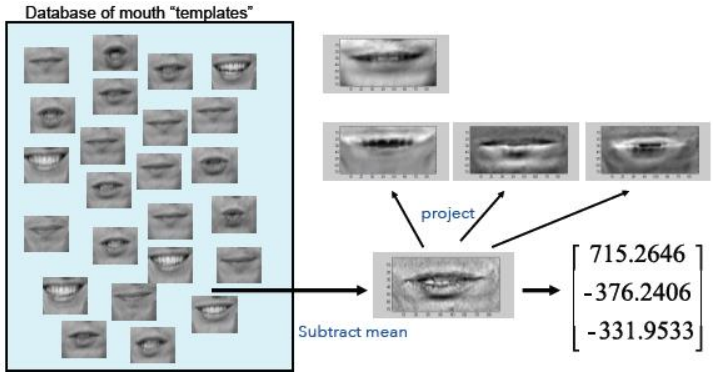
3. View-Based Recognition

We already saw that the Naïve based approach was not ideal due bad computational efficiency. Let's us now put PCA in practice! The database of mouth "templates" can now be represented by a mean and for example the first 3 eigenvectors:



3.1. Simple Search Strategy

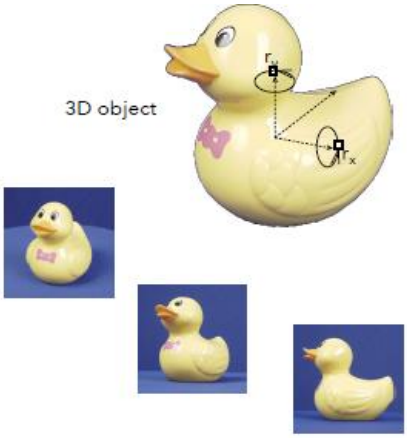
1. Project each training image onto the low-dimensional subspace and store the **vectors of coefficients**.



2. For each image region:
    a. Project it onto the low-dimensional subspace
    b. Compare this to each stored coefficient vector (cheap)
    c. If the smallest distance is less than some threshold, then it is a mouth

4. Appearance-Based Instance Recognition

Basic assumption: 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. There was a fundamental paradigm shift in the 90's!



However, there are some **challenges** with this:

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

Appearance manifolds: Many objects do not have convex subspaces when one considers different poses and lighting variations! Also, there are limitations of linear representations hence **PCA will not work!**

The idea now is to 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!

View-based Approaches…:

- are **severely challenged** by these common variations. To make them work, we would need an unmanageable amount of examples (training data).
- 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 have to 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.

Interest Points & Image Features

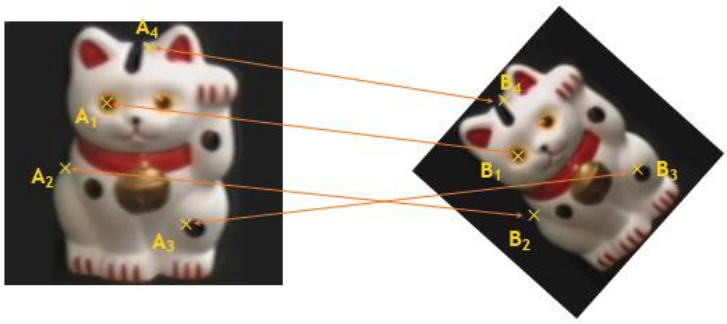
1. Motivation

Recognizing object categories is hard (more later). For now 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:

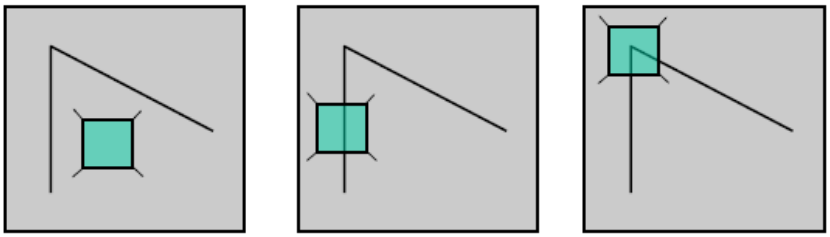
1. Find important. i.e., **"interesting" points** in the images
2. Find which interest points **correspond** to one another.

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



2. Derivation Interest Point Detection

We are starting the local measure of **interest point uniqueness** (def.: intersection of two or more edge segments or the point at which the direction of the object's border rapidly changes) by looking at how a window **changes** within an image when we **shift** it. Shifting the window in any direction should cause a **big change** e.g. a corner to be considered a unique interest point! The next figure illustrates this:

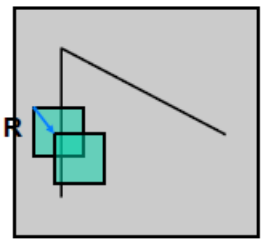


"flat" region: no change in all directions

"edge": no change along the edge direction

"corner": significant change in all directions

Consider shifting the window R by a vector $\begin{pmatrix} u \\ v \end{pmatrix}$:



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 the shifted window using a first order Taylor Series approximation:

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

where $\epsilon(u^2,v^2)$ is the approximation error. Hence using the approximation in the energy, we get:

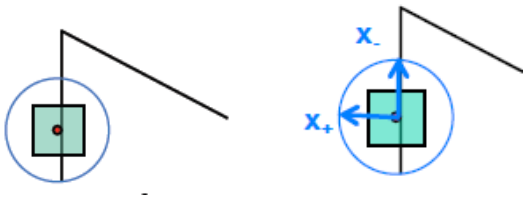
$$\begin{aligned}
E(u,v) &= \sum_{(x,y)\in R} \big(I(x+u,y+v) - I(x,y)\big)^2 \\
&\approx \sum_{(x,y)\in R} \left(I(x,y) + u\frac{\partial}{\partial x}I(x,y) + v\frac{\partial}{\partial y}I(x,y) - I(x,y)\right)^2 \\
&= \sum_{(x,y)\in R} \left(u\frac{\partial}{\partial x}I(x,y) + v\frac{\partial}{\partial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

$$= (u, v)\left[\sum_{(x,y)\in R} \nabla \boldsymbol{I}(x, y)\nabla \boldsymbol{I}(x, y)^T\right]\begin{pmatrix} u \\ v \end{pmatrix}$$
$$= (u, v)\boldsymbol{H}\begin{pmatrix} u \\ v \end{pmatrix}$$

where $\nabla \boldsymbol{I} = \begin{pmatrix} I_x \\ I_y \end{pmatrix}$. The most important part of this derivation is the **structure tensor $\boldsymbol{H}$:**

$$\boldsymbol{H} = \sum_{(x,y)\in R} \nabla \boldsymbol{I}(x, y)\nabla \boldsymbol{I}(x, y)^T$$

Suppose you can move the red 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):

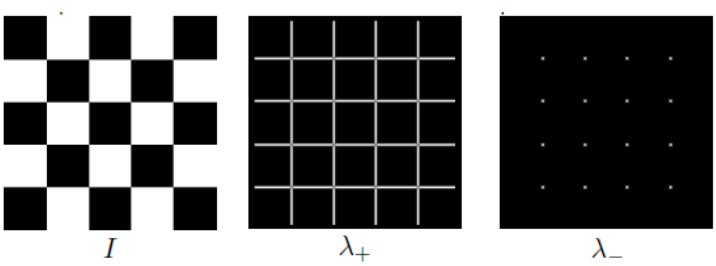
$$\boldsymbol{H}\boldsymbol{x}_+ = \lambda_+\boldsymbol{x}_+$$
$$\boldsymbol{H}\boldsymbol{x}_- = \lambda_-\boldsymbol{x}_-$$

Where:

- $\boldsymbol{x}_+$ = direction of largest increase in E
- λ_+ = amount of increase in direction $\boldsymbol{x}_+$
- $\boldsymbol{x}_-$ = direction of smallest increase in E
- λ_- = amount of decrease in direction $\boldsymbol{x}_-$

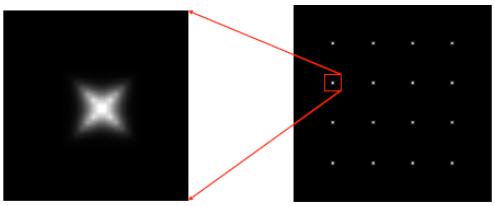
But how are $x_+, \lambda_+, x_-, \lambda_-$ relevant for interest point detection? What's our feature scoring function?

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 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 $\lambda\_$ is a local maximum as interest points:



2.1. The Harris Operator

$\lambda\_$ 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 $\lambda\_$ 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.

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. Calculate f value
5. $f > threshold$
6. Find local maxima of f

2.2. Hessian Determinant

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

$$\boldsymbol{H}(I) = \begin{pmatrix} I_{xx} & I_{xy} \\ I_{yx} & I_{yy} \end{pmatrix}$$

$$\det(\boldsymbol{H}) = I_{xx}I_{yy} - I_{xy}^2$$

2.3. Problems

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 at the next image where: (1) original, (2) similarity transformation (translation, image plane rotation, scaling), (3) projective transformation.

(1) (2) (3)

2.4. Rotation Invariance Detectors

The Harris interest point detector is **rotation invariant** because rotating the image leads to rotation of the gradient!

∇I $\mathrm{R}\nabla I$

The structure tensor then is:

$$\boldsymbol{H} = \sum_{(x,y)} \boldsymbol{R}\nabla I(x,y)(\boldsymbol{R}\nabla I(x,y))^T = \sum_{(x,y)} \boldsymbol{R}\nabla I(x,y)\nabla I(x,y)^T\boldsymbol{R}^T = \boldsymbol{R}\boldsymbol{H}\boldsymbol{R}^T$$

Rotation invariance follows from:

$$\det(\boldsymbol{R}\boldsymbol{H}\boldsymbol{R}^T) = \det(\boldsymbol{R})\det(\boldsymbol{H})\det(\boldsymbol{R}^T) = \det(\boldsymbol{H})$$

$$trace(\boldsymbol{R}\boldsymbol{H}\boldsymbol{R}^T) = trace(\boldsymbol{H})$$

2.5. Scale Invariance Detectors

We will look at 3 different Scale-invariant interest point detection methods:

- Matching images of different scales
- Automatic scale selection
- Scale invariant methods for feature extraction: **Harris-Laplace**

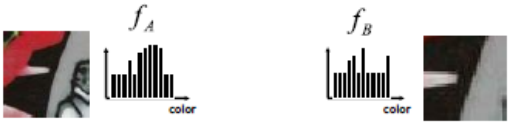
2.5.1. Matching images patches of different scales

To match image patches of different scales we:

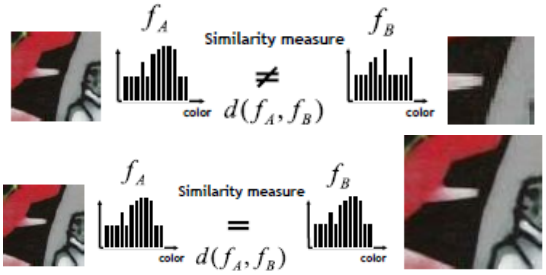
1. Detect interest points

2. Extract patches

3. Compute **feature descriptors** (def.: A feature descriptor (for example a historgram) is a method that extracts the feature descriptions for an interest point. Kind of numerical "fingerprint")



4. Comparing feature descriptors



However, in the last step, it's impossible to match different histograms due to different patch content. The patch should contain the same part of the image. How do we find the correct size? We can find the correct size by comparing feature descriptors while varying the patch size, e.g. repeat steps 2,3 and 4 repeatedly! This is computationally inefficient but possible for matching and computationally prohibitive for retrieval in large databases as well as recognition.

2.5.2. Automatic scale selection

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.

2.5.3. Harris Laplace

Recall the Harris corner detector

$$f = \det(\boldsymbol{H}) - \alpha \cdot trace(\boldsymbol{H})^2$$

$$\boldsymbol{H} = \sum_{(x,y)\in R} \mathrm{w}(\mathrm{x},\mathrm{y}) \nabla I(x,y) \nabla I(x,y)^T$$

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:

1. Compute Harris function Detect Harris points for multiple scales $\sigma_1, \sigma_2, \sigma_3, \sigma_4, \ldots$
2. This results in thousands of interest points which is why we need a criterion.
3. Select points which maximize the Laplacian. Automatic Scale detection: 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.

2.5.4. SIFT

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

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

$$\boldsymbol{H} = \begin{pmatrix} I_{xx} & I_{xy} \\ I_{yx} & I_{yy} \end{pmatrix}$$

SIFT is a blob detector rather than corner detector. We will cover more stuff about SIFT later!

2.5.5. 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 detector rather than a corner detector!

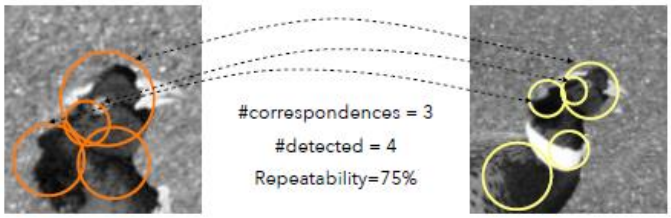
2.6. Evaluation of Interest Point Detections

This is what we did so far:

1. Interest Point Detection: Identify unique points in the image.
2. Feature Descriptor Extraction: Describe these points for matching.
3. Matching Interest Points: Compare descriptors to find corresponding points.

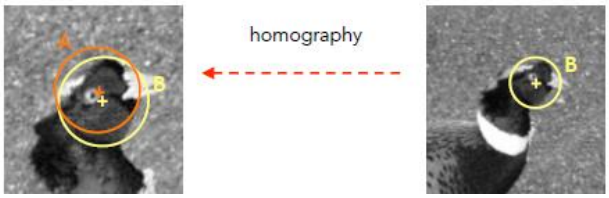
Which interest points work when? We can us the **repeatability rate** which is a percentage of corresponding points:

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



But how do we find which interest points correspond to one another?

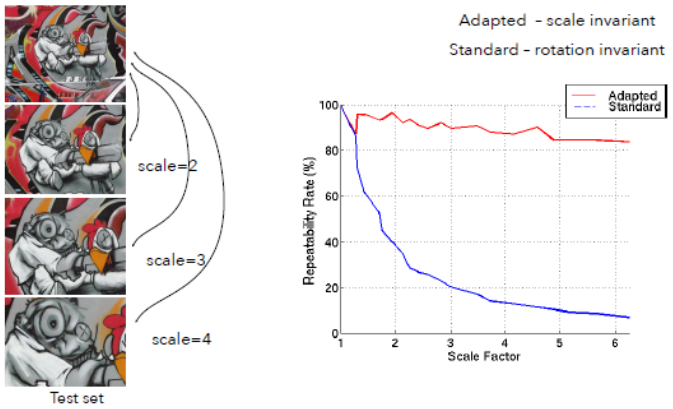
4. Homography Estimation: Compute transformation between images.



5. Ground Truth Generation: Manually transform images using known homography. Evaluation: Compare detected points with ground truth. Calculate intersection over union error (Jaccard index). 2 points are corresponding if

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

Adapted **scale invariant** detectors have a way higher repeatability rate over different scales than standard **rotation invariant** detectors:



We will have a more detailed look in the next chapter on how to find which interest points correspond to one another.

3. Local Descriptors

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)

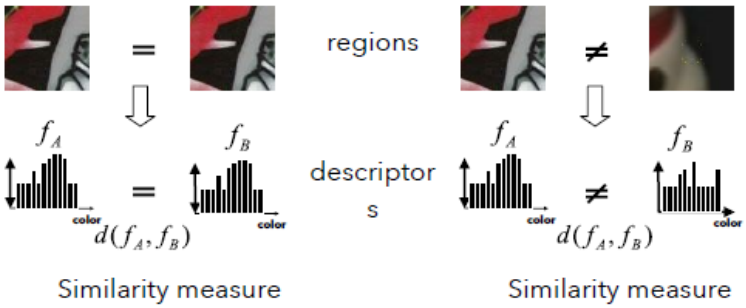
Note we already introduced this for the Harris Laplace Detector if you paid attention.

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 e.g. distinctiveness, invariance, robustness, and dimensionality.

3.1.1. Distinctiveness

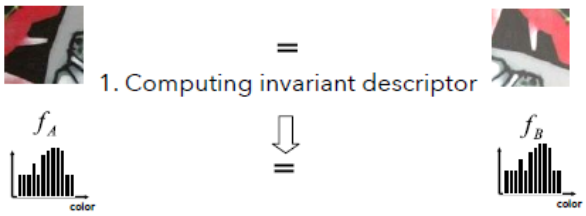
Visually similar regions should have similar descriptors and different regions should have different descriptors.



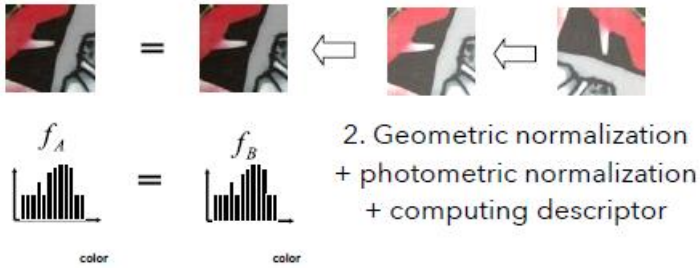
3.1.2. Invariance

Visually similar regions should have similar descriptors despite the transformation (geometric, photometric) i.e., rotation, brightness. There are two ways to obtain invariance:

a) Computing invariant descriptor directly

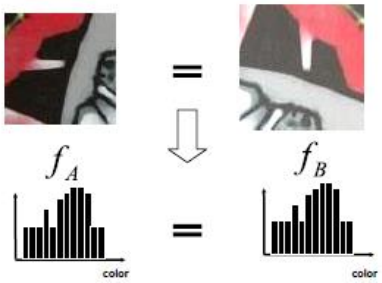


b) First doing geometric normalization then photometric normalization and at the end computing the descriptor



3.1.3. Robustness

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



3.1.4. 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.

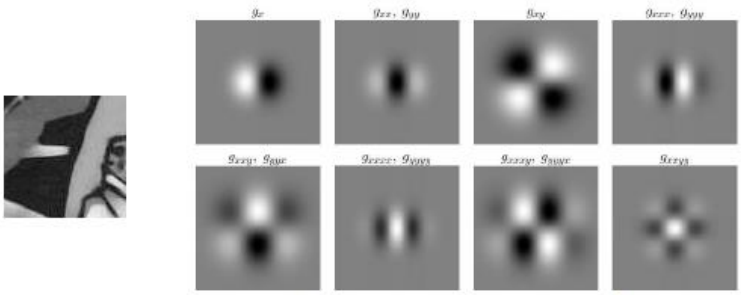
3.2. Local Descriptors

3.2.1. 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.

3.2.2. Bank of Filter responses (jet)

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



$[g_x, g_y, g_{xx}, g_{xy}, g_{yy}, \ldots]$

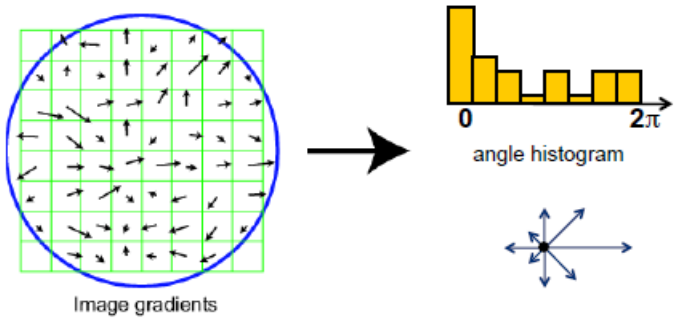
Responses of Gaussian derivatives

3.2.3. Scale Invariant Feature Transform (SIFT)

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

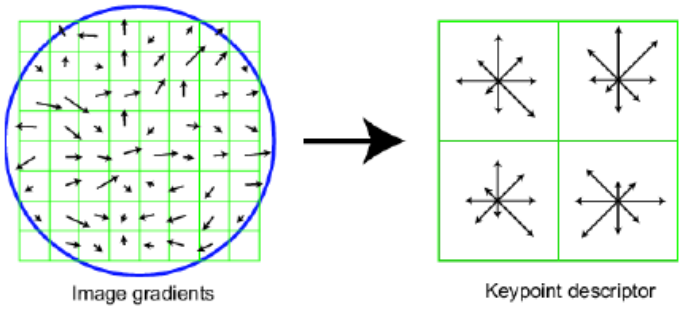
1. Take 16x16 square window around detected interest point
2. Compute edge orientation (angle of the gradient - 90°) for each pixel
3. Throw out weak edges (threshold gradient magnitude)

4. Create histogram of surviving edge orientations



The full version works as follows:

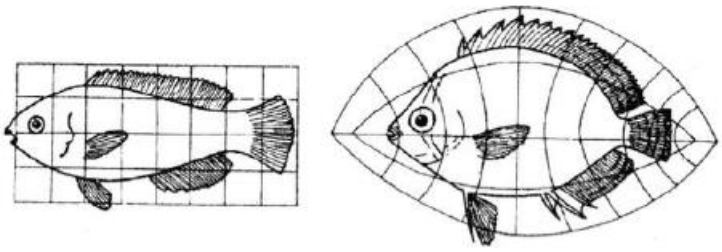
1. Divide the 16x16 window into a 4x4 grid of cells (2x2 case shown below)
2. Compute an orientation histogram for each cell
3. 16 cells * 8 orientations = 128-dimensional descriptor



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.

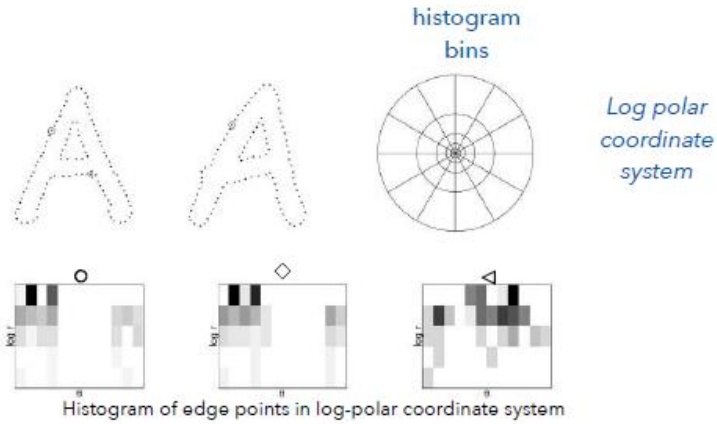
3.2.4. Shape Context

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



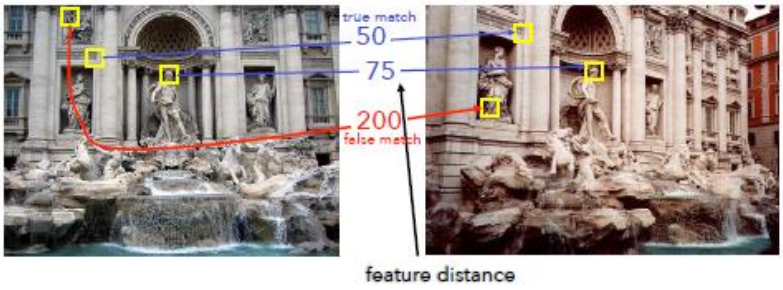
Represent histogram of edge locations:

1. Run edge detector
2. Make histogram insensitive to minor shape variations: Use log polar coordinate systems



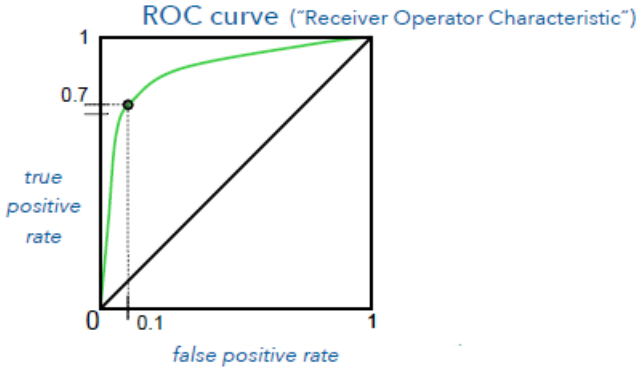
3.2.5. Evaluation

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



The goal is to **maximize true positives** (number of detected matches that are correct) and **minimize false positives** (number of detected matches that are incorrect).

How should we choose the threshold in both settings? For that we can use **Receiver Operator Characteristic** (ROC) 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).



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

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

Single & Two View Geometry

1. Homography

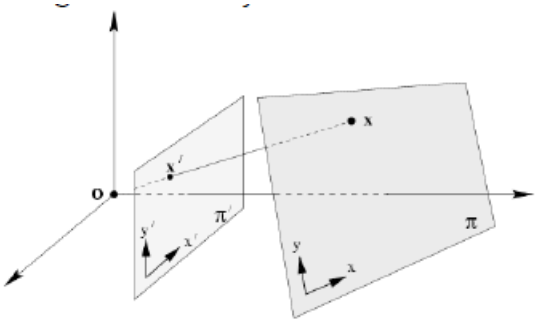
Homography is a transformation that maps the points in one image to the corresponding points in another image. This concept is used when the images are taken from different perspectives of the **same planar surface**.

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.

1.1. Perspective image of a plane

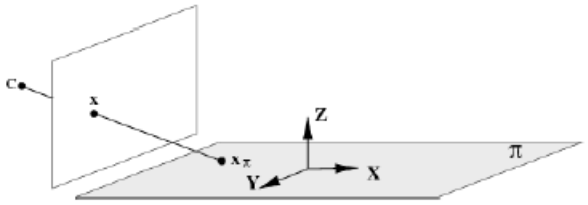
Let's start with the geometric intuition first. Projection with a straight ray from a 2D plane π (scene) to another 2D plane π' (image plane of the camera). There is **no need** for dimensionality reduction because the ray intersects each plane exactly once. The **inverse projection** along the same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



Remember a pinhole camera is represented by a (3x4) homogenous projection matrix $\boldsymbol{P}$**:**

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

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



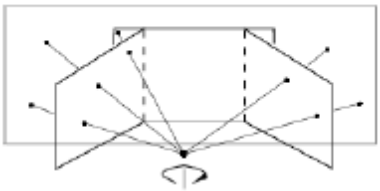
$$\begin{aligned}\boldsymbol{x} &= \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\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

The mapping from a 2D plane (scene) to the image plane is given by (3x3) matrix $\boldsymbol{H}$, which is called a **Homography** (or projectivity):

- Most general projective 2D transformation
- Maps a square to a general quadrangle.
- Has **8 degrees of freedom**
- $\boldsymbol{H}$ 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 $x = Hx_\pi \leftrightarrow x_\pi = H^{-1}x$

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):

$$\boldsymbol{x = K[I|0] = K\widetilde{X} \leftrightarrow \widetilde{X} = K^{-1}x}$$

$$\boldsymbol{x' = K'[R|0]X = K'R\widetilde{X} \leftrightarrow x' = K'RK^{-1}x = Hx}$$

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):

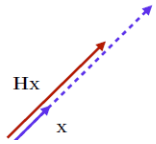
$$\lambda x' = Hx$$

$$x' \propto Hx$$

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 (homogenous coordinates):

$$x' \propto Hx \rightarrow Hx \times x' = 0$$



Writing this out leads to:

$$x = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, x' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

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

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

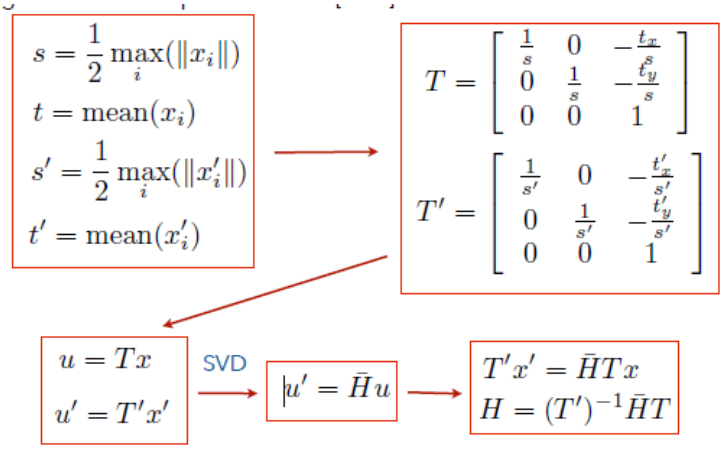
$$Ah = 0, s.t. \|\boldsymbol{h}\| = 1$$

$$A = USV^T$$

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.

However, we **cannot try** all possible samples meaning **doing a lot 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	$k = \frac{\log(1-z)}{\log(1-w^d)}$

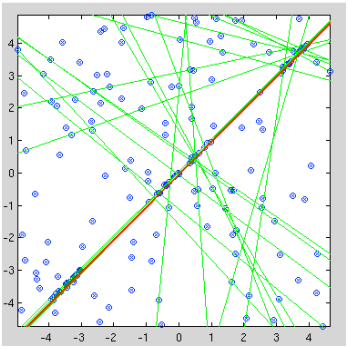
Depending on the dimensionality of the data and how big the fraction of uncontaminated samples is the number of iterations changes as displayed in the next table:

	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

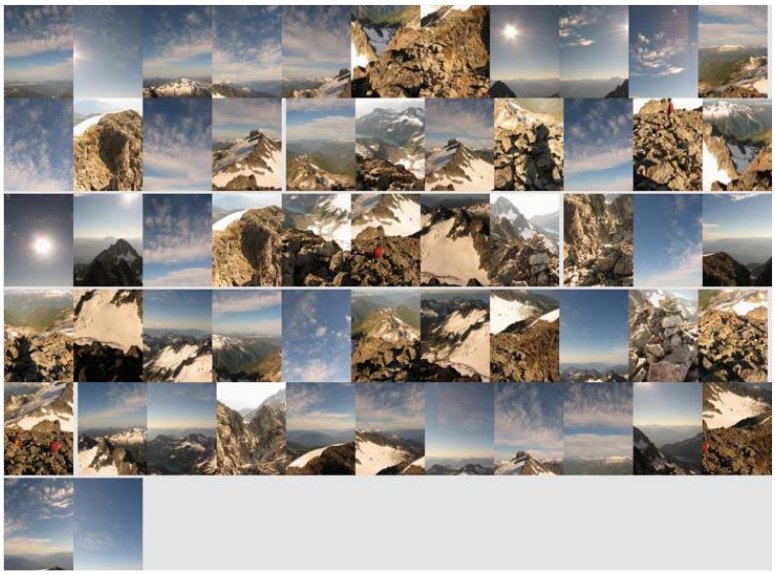


Algorithm 2: Estimating the Homography using RANSAC

Input: 2 images from a rotating camera (same viewpoint)
Output: Homography $\boldsymbol{H}$
1: **condition** all images
2: **iterate**
3: pick 4 correspondences
4: build equations, estimate homography
5: transform all points x
6: measure distance to all points x' (in non-homogeneous coordinates!)
7: count inliers (scale down threshold too!)
8: re-estimate final homography
9: **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?

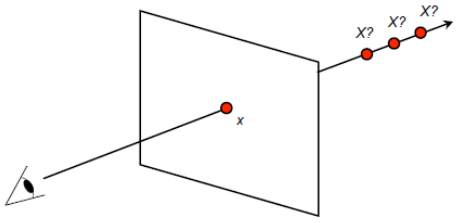
What condition do you use for estimating the Homography matrix H ?

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

Stereo Vision

1. Motivation

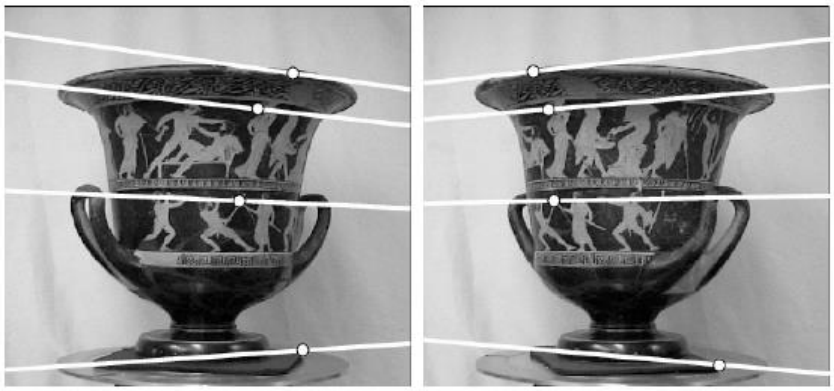
The next goal is the **recovery of the 3D structure**. Thats only possible if we have **2 images** taken from two different viewpoints. The reason for that is that the recovery of depth from one image is inherently **ambiguous**.



We will cover now geometric methods that rely on two views.

1.1. What is Stereo Vision?

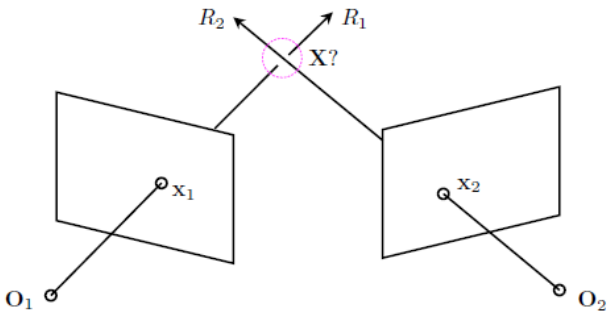
Stereo vision, stereopsis, or short stereo is the perception or **measurement of depth** from two projections. It was first described in technical detail by Charles Wheatstone in the 1830s with the invention of the stereoscope. The human visual system heavily relies on stereo vision: Our eyes give two 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** to construct the 3D rays, which we will discuss later in the **epipolar geometry** section.

Given projections of a 3D point $\boldsymbol{X}$ in two or more images (with known camera matrices), find the coordinates of the point.



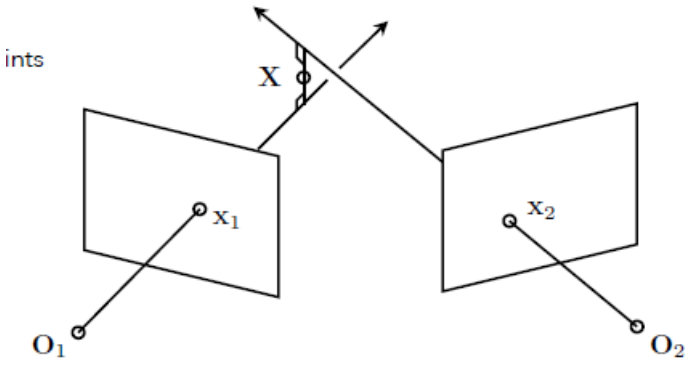
We want to intersect the two viewing rays corresponding to x_1 and x_2, but because of noise and numerical errors, they do not meet exactly.

To solve this problem there are 3 main methods:

- Geometric Midpoint
- Linear Method
- Non-linear Method

2.1. Geometric Midpoint

The goal is to find the shortest segment connecting the two viewing rays and let $\boldsymbol{X}$ be the geometric midpoint of that segment:

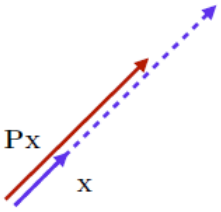


The workflow is as follows:

1. Find the segment direction (normal to both rays)
2. Construct 2 planes each containing the segment and one ray
3. Intersect planes with “other” rays to yield segment endpoints
4. Average points

2.2. Linear method

To find the coordinates of the 3D point $\boldsymbol{X}$ we need to solve **two** independent equations per image point for three unknown entries of $\boldsymbol{X}$. 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.



$$\lambda_1 \boldsymbol{x_1} = \boldsymbol{P_1 X} \rightarrow \boldsymbol{x_1} \times \boldsymbol{P_1 X} = 0$$

$$\lambda_2 \boldsymbol{x_2} = \boldsymbol{P_2 X} \rightarrow \boldsymbol{x_2} \times \boldsymbol{P_2 X} = 0$$

2.2.1. 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:

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

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

Hence, we can write:

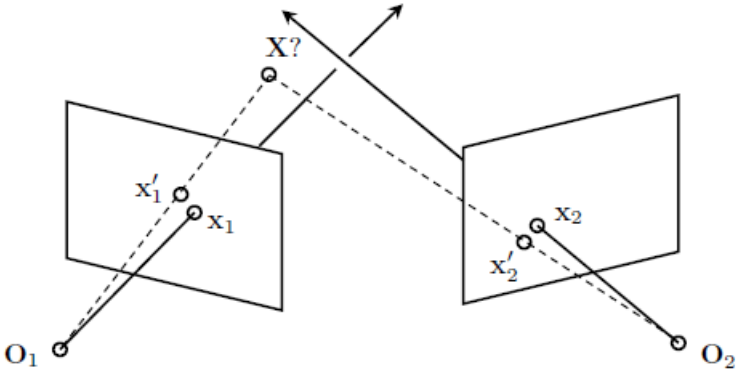
$$[\boldsymbol{x}]_\times \boldsymbol{P X} = 0$$

where $[\boldsymbol{x}]_\times \boldsymbol{P}$ is a matrix.

2.3. Non-Linear Method

Find X that minimizes the (squared) reprojection error:

$$d^2(\boldsymbol{x_1}, \boldsymbol{P_1 X}) + d^2(\boldsymbol{x_2}, \boldsymbol{P_2 X})$$



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)

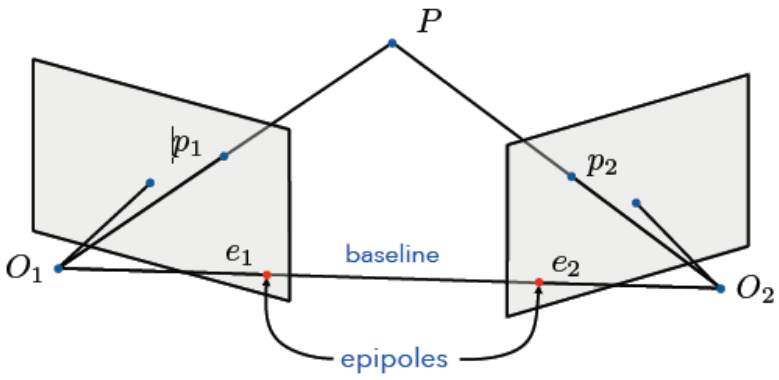
Well, the important question is now how to efficiently find 2 point correspondences of the same scene? We can use epipolar geometry for that!

3. Epipolar Geometry

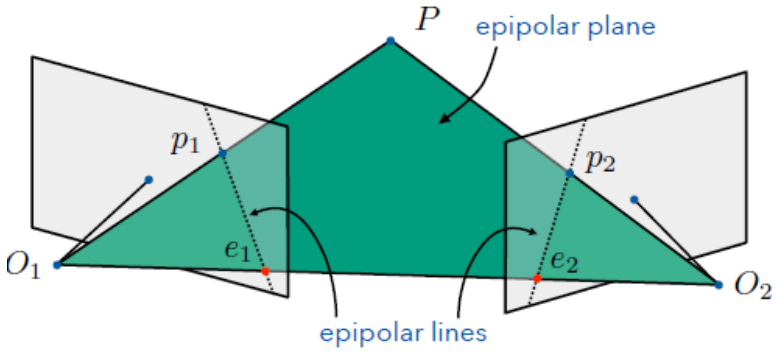
The next goal is to find the relation between two different views of the same scene. In detail we will recover **camera placement**, we will do matching and triangulation for **multiple views** and at end scene modelling will be possible only from images.

3.1. Geometrical Relations

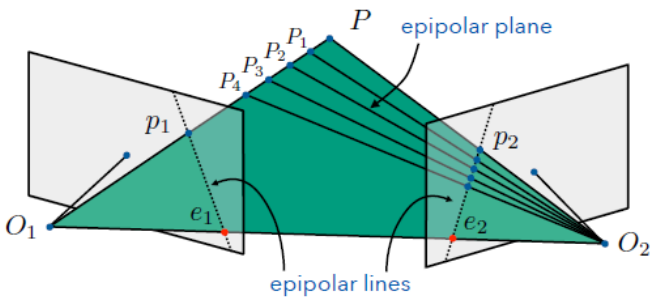
Given a 3D point $\boldsymbol{P}$ in the world and two images taken by two cameras with the **same** calibration matrix (later more) from two **different** viewpoints then the **epipole** is defined as the image location of the optical center of the other camera:



The **epipole** can be outside of the visible area (imagine expanding the images infinitely). The **epipolar plane** is defined as a plane through both camera centers and the world point as seen in the next figure.



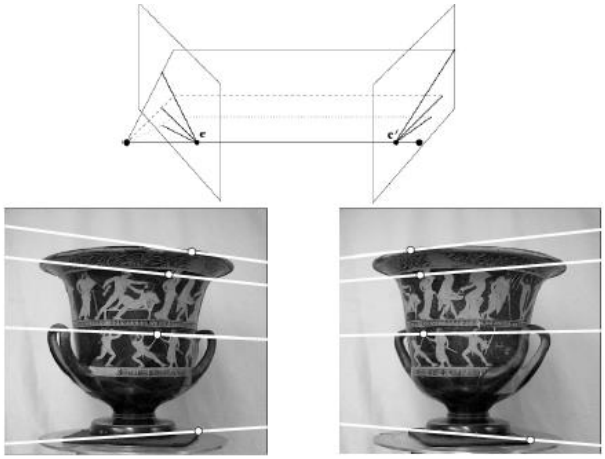
The epipolar line **constrains** the location where a particular feature from one view can be found in the other! In the figure below possible matches for p_1 are displayed.



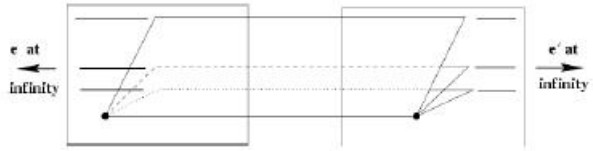
Epipolar lines intersect at the epipoles and are in general not parallel. In other words, we only need to search along the epipolar line!

3.1.1. Examples

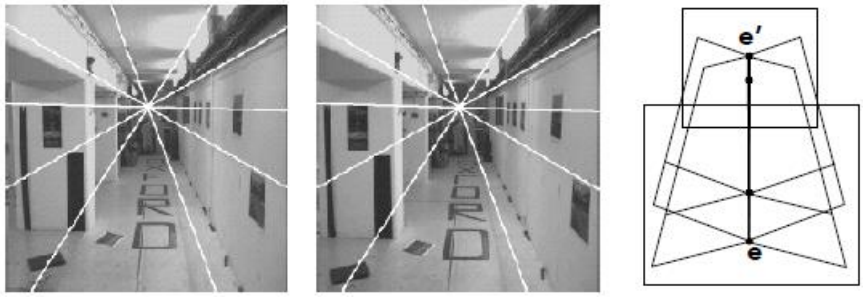
- Converging cameras



- When the motion is parallel to the image plane, it results in a special case for the location of the epipoles. In this scenario, the **epipoles** are located at **infinity.**

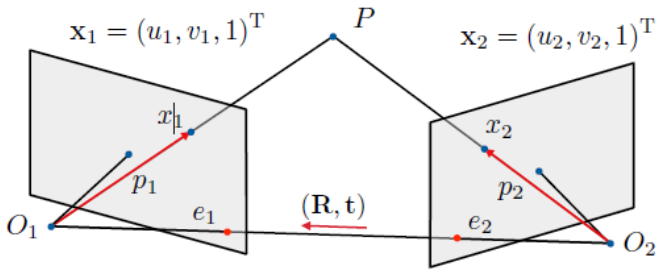


- Another example is forward motion. The epipole has the same coordinate in both images. The point moves along lines radiating from epipole resulting in a “focus of expansion”.



3.2. Algebraic relations

Let's describe epipolar geometry mathematically. Again the 2 camera centers differ by a translation and rotation!



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** (remember epipolar lines!). 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** $\boldsymbol{E}$**:**

$$\boldsymbol{E} = [\boldsymbol{t}]_x \boldsymbol{R}$$

which captures the **relation of two views**:

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

Hence two views of the same 3D point must satisfy $\boldsymbol{p}_1^T \boldsymbol{E} \boldsymbol{p}_2 = 0$. Other important properties 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 $\boldsymbol{E}$: $\boldsymbol{e_1^T E} = \boldsymbol{E}^T\boldsymbol{e_1} = 0$, $\boldsymbol{E e_2} = 0$
- 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).

3.2.1. 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:

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

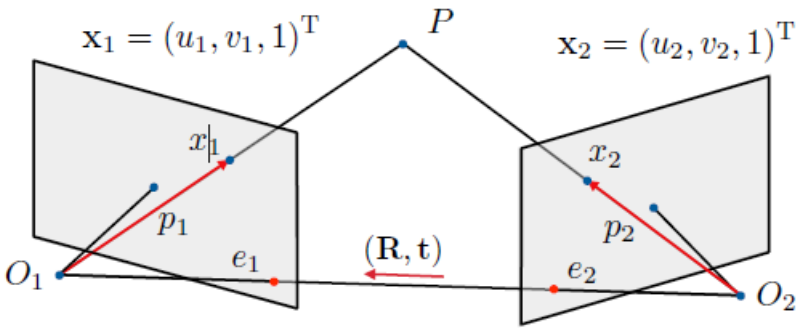
Plugging this into the epipolar constraint yields:

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

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

3.2.2. Uncalibrated Cameras

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

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

where $\boldsymbol{F} = \boldsymbol{K_1^{-1} E K_2^{-1}}$ 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 the contraint:

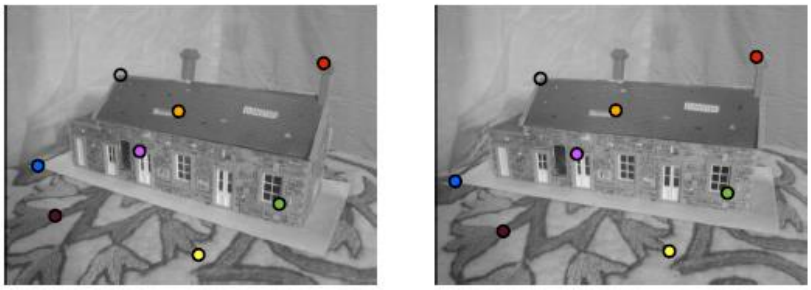
$$\mathrm{x}_1^{\mathrm{T}} \mathrm{F} \mathrm{x}_2 = 0$$

Other important properties of are:

- epipolar lines: $\boldsymbol{l_1 = F x_2}$ and $\boldsymbol{l_2 = F^T x_1}$
- epipoles are the (left/right) null-space of $\boldsymbol{F}$: $\boldsymbol{e_1^T F = F^T e_1 = 0}$, $\boldsymbol{F e_2 = 0}$
- Further the essential matrix is singular and has rank 2
- The two remaining eigenvalues are equal
- **7 degrees of freedom**

3.2.3. Estimating the fundamental matrix

The fundamental matrix has 7 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.



Here is how it works:

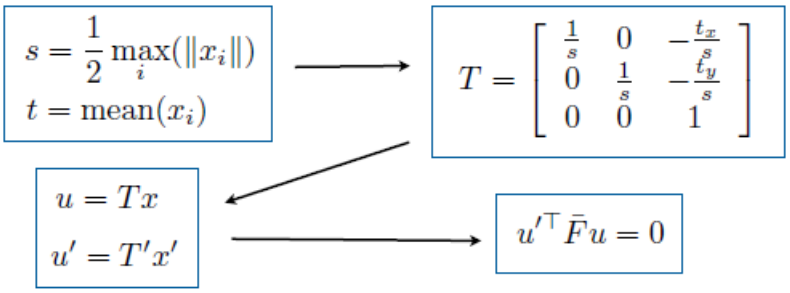
1. Stack equations from at least 8 correspondences:

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

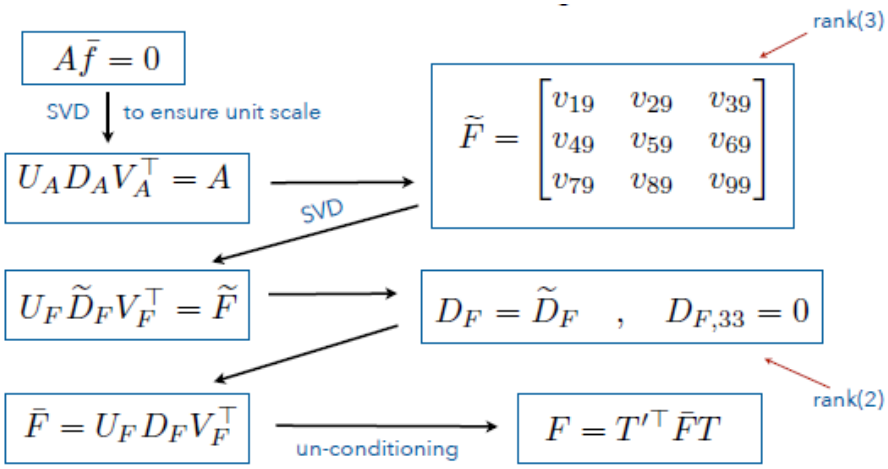
$$\begin{bmatrix} xx' & yx' & x' & xy' & yy' & y' & x & y & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0 \longrightarrow \boxed{\mathrm{Af} = 0}$$

2. Solve homogeneous linear equation system: $\|\boldsymbol{Af}\|^2 \; s.t. \|\boldsymbol{f}\| = 1$ with SVD

We have again problem with **numerical stability**, just like in the Homography chapter 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:



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



3.2.4. Important Conclusions and Cookbook Recipe

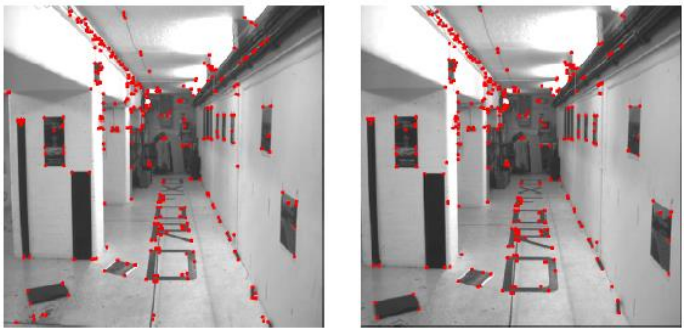
Calibration is not required, hence we can deal with archival images, photos not taken for reconstruction purposes and varying intrinsics.

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

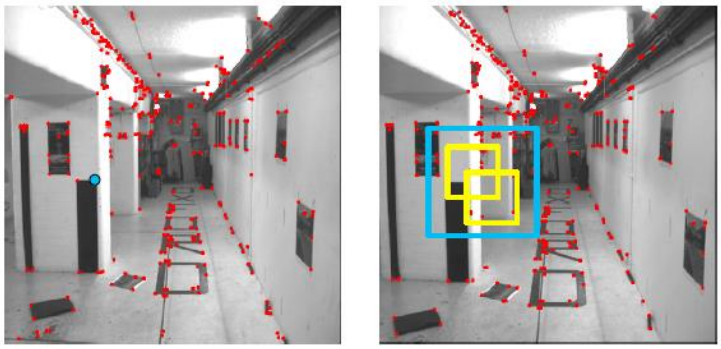
1. Find interest points in both images
2. Match them without epipolar constraint
3. Compute epipolar geometry
4. Refine

Let's look at an example:

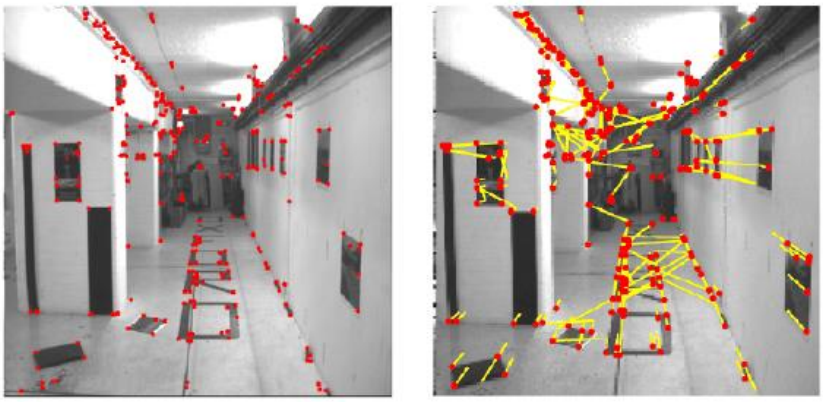
1. Interest points (here: Harris corner detector)



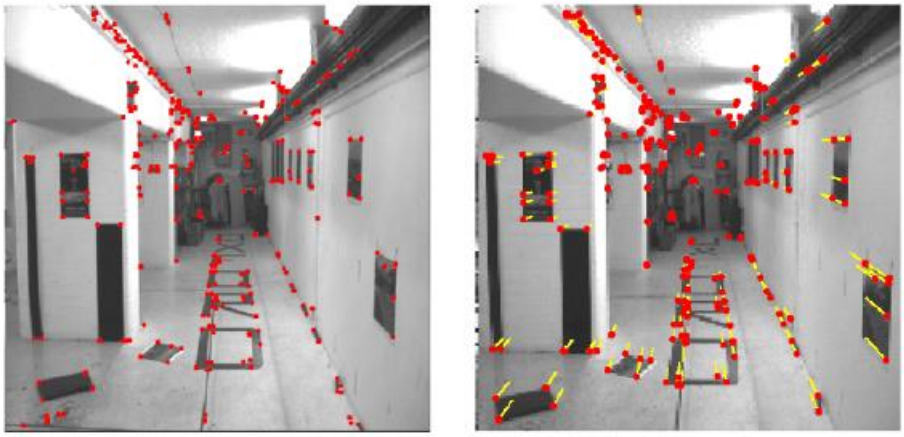
2. Match them without epipolar constraint using only weak constraints (here: proximity)



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



4. Correspondences consistent with epipolar geometry:



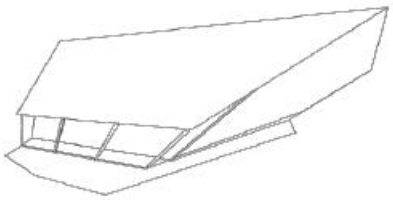
5. Resulting epipolar geometry:

6. Refine!

3.3. Discussion

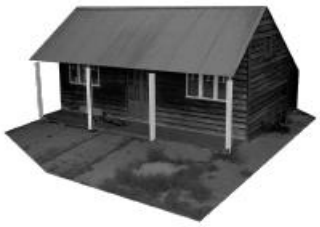
What can we do with 2 views? Given two views of a static scene, and a small number of correspondences. If the views were recorded with any pinhole camera, we can:

- 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

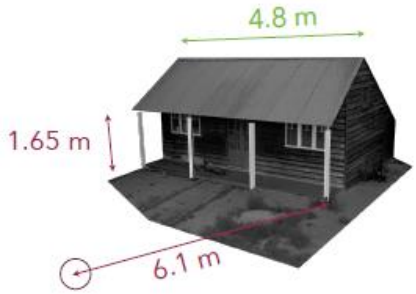
- reconstruct the scene up to scale
- determine angles, relative distances
- allows 3D modelling.



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 Estimation

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!

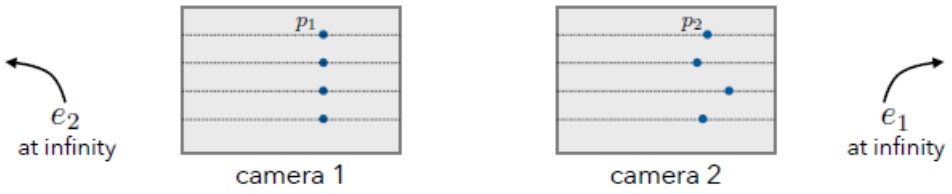
Narrower formulation: Given two views with similar (but not identical!) viewpoint, fuse them to obtain a depth image. Way easier!

4.2. Binocular Stereo

Binocular stereo is a special case of epipolar geometry for stereo cameras with a standard binocular setup:

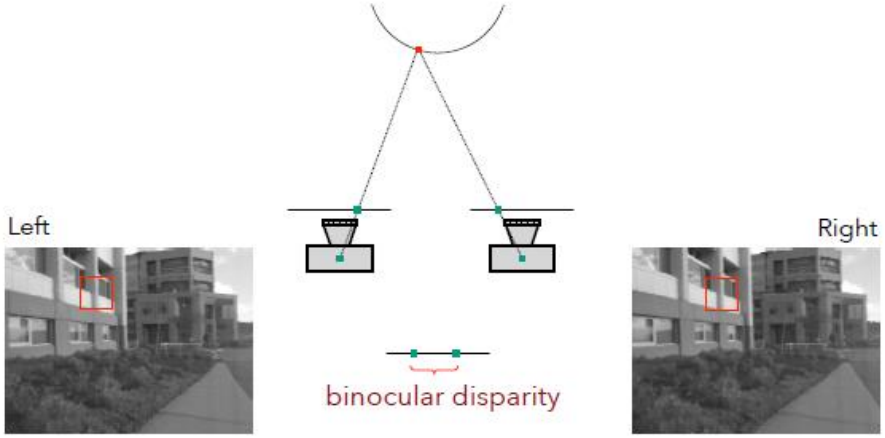
- Epipoles are at **infinity**
- Epipolar lines are parallel
- Points correspond along the scanlines of the image

In a binocular setup the rotation matrix of the cameras is zero. So, we only have a difference in translation! Just like human eyes!

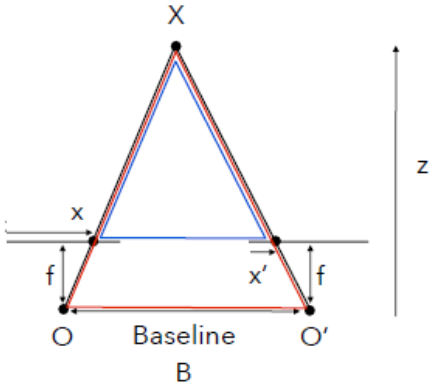


4.2.1. Disparity

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 **for each single pixel separately** 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 can estimate the depth then with:

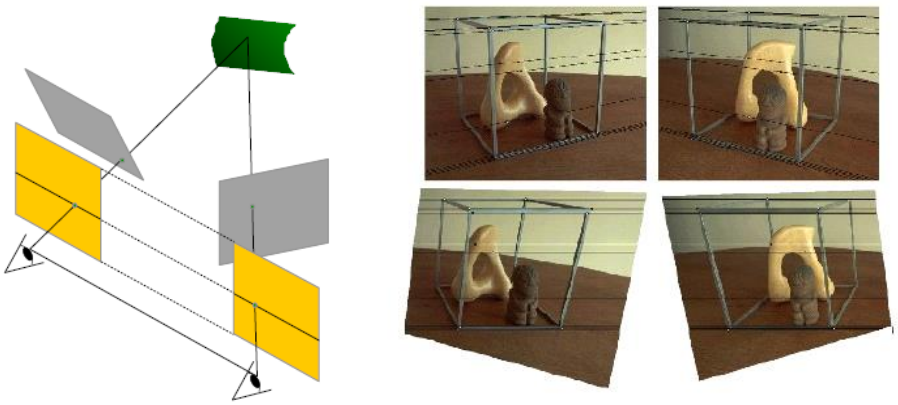
$$Z(x, y) = \frac{fB}{d(x, y)}$$

Now we only need to find the disparity by **searching for the best match along** the epipolar line! Don't need to consider any other region than the epipolar line!

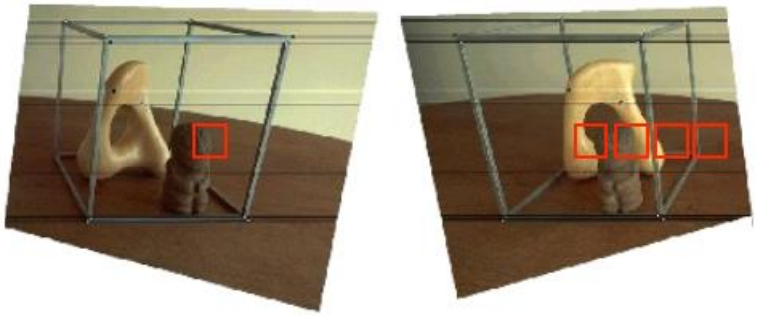
4.2.2. Stereo Rectification

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

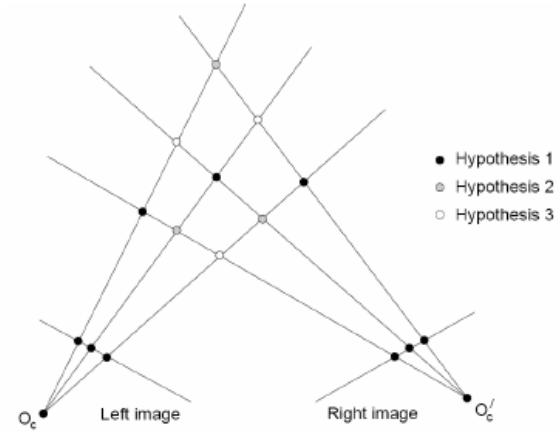


Again matching only must occur along epipolar lines. Now in the simpler binocular case where the cameras are pointing forward.



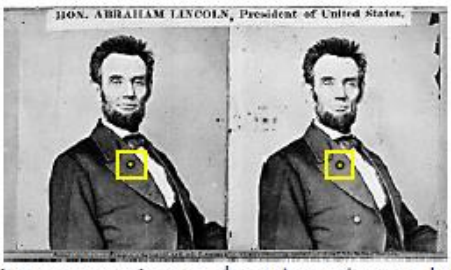
4.2.3. Stereo Correspondence Problem

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?



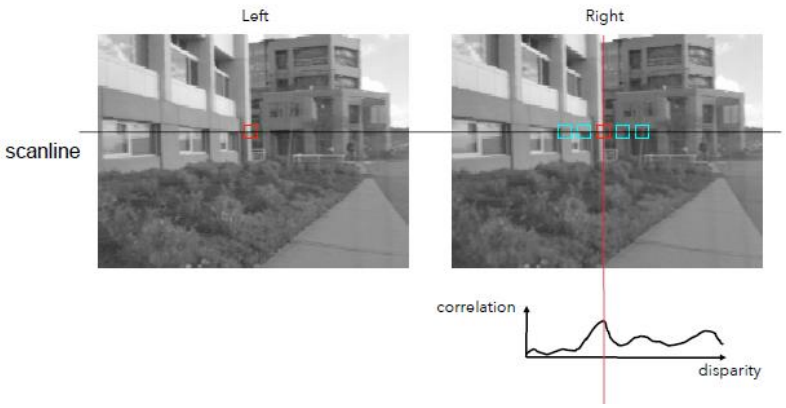
Let us make some assumptions to simplify the matching problem:

- 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 as seen in the next figure.



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.

4.2.4. 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

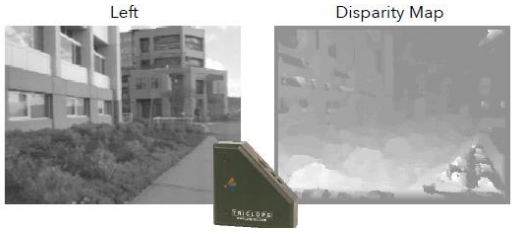
Even a simpler method is to use **sum of squared (Pixel) differences**. The SSD cost measures the intensity difference as a function of disparity (What we are saying is that the brightness doesn't change for the patch we are searching for in the second image even if it moved with the disparity d. It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

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



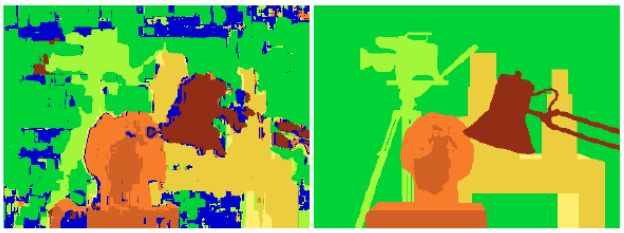
4.2.5. Problems of Window-based matching

We have 2 main problems:

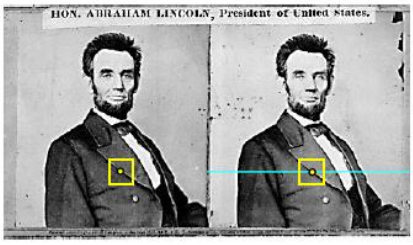
- How do we choose the right **window size** m? Use adaptive window
- Mismatches often lead to relatively **poor results quality**! Use similarity constraint

No window size fits every condition. We get better results with **adaptive window**:

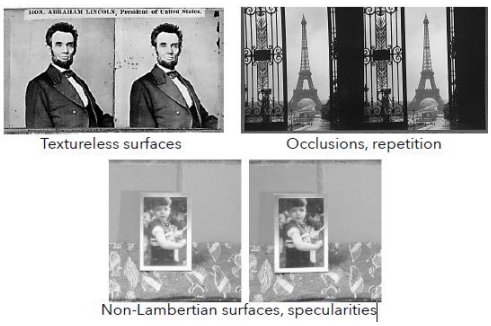
How can we improve window-based matching? The next picture shows us the result we get with window correlation:



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?



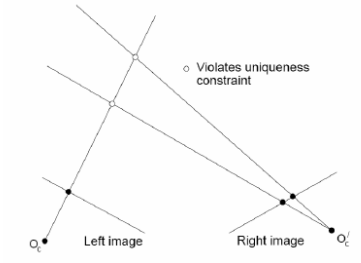
It fails for textureless surfaces, occlusions, repetition, non-Lambertian surfaces (glossy surfaces) and specularities as illustrated in the next figure.



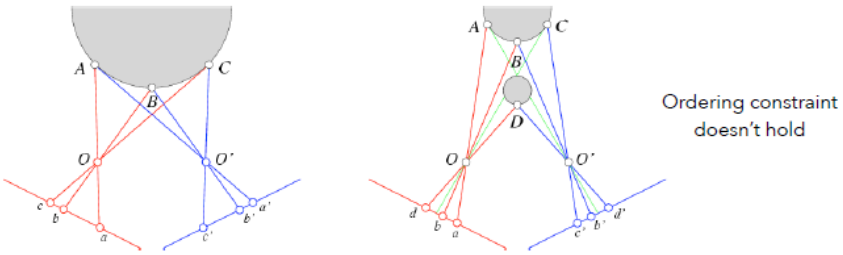
The **similarity constraint** is local. Each reference window is matched independently, and other points do not influence the result. Another method to improve window-based matching is to enforce **non-local correspondence constraints.** This requires **spatial regularity** (Computer Vision II)!

Non-local constraints ensure:

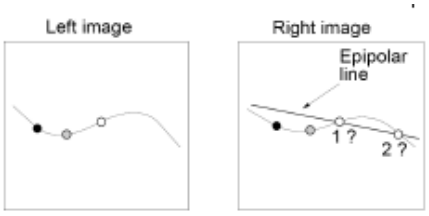
- **Uniqueness**: For any point in one image, there should be at most one matching point in the other image.



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



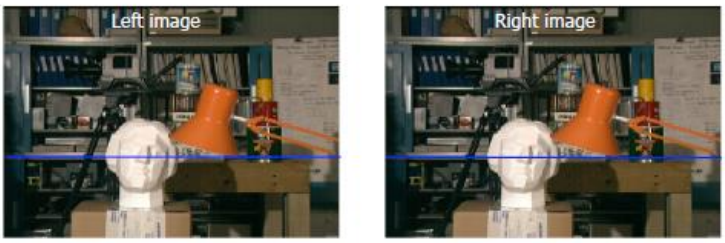
- **Smoothness:** We expect disparity values to change slowly (for the most part).



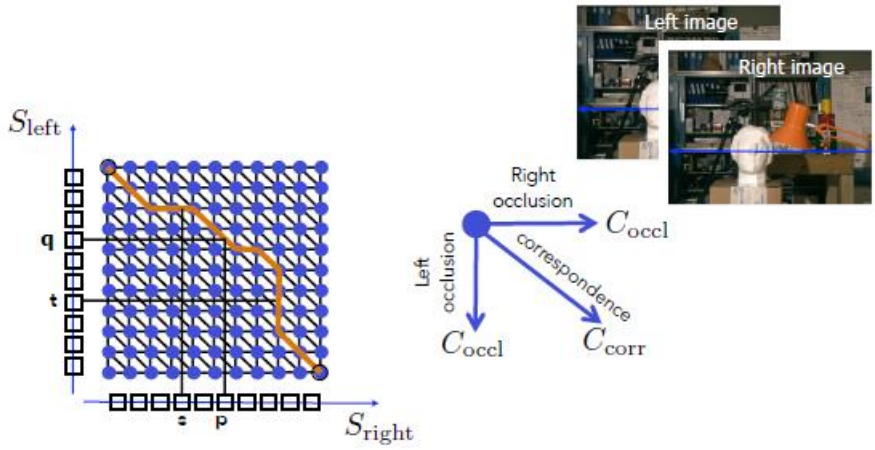
Given matches ● and ○, point ○ in the left image must match point 1 in the right image. Point 2 would exceed the disparity gradient limit.

4.3. Scan line stereo

Try to coherently match pixels on the entire scanline. Different scanlines are still optimized independently:

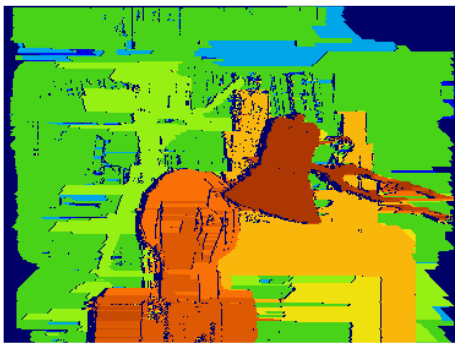


The **shortest path** for scan-line stereo can be found using **dynamic programming**:



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

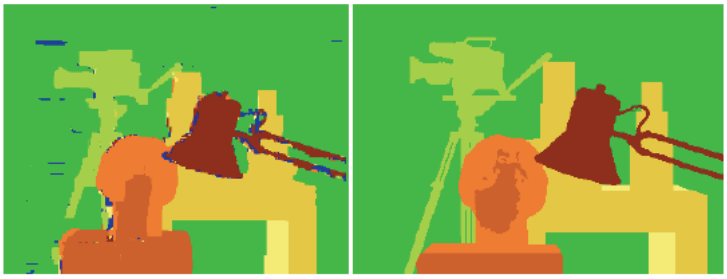
Scanline stereo generates **streaking** artifacts:



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

4.4. What's next?

How can be **improve** the results (see figure below)? We need to impose **spatial regularity** to improve results! Stay tuned for Computer Vision 2!



Graph cuts Ground truth

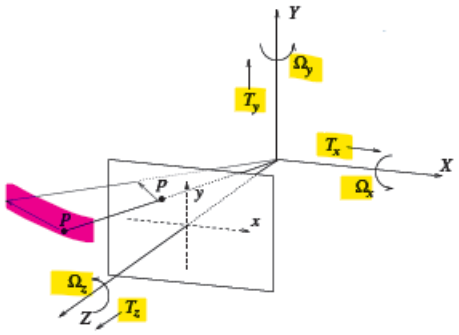
Optical Flow: Dense Motion Estimation

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.

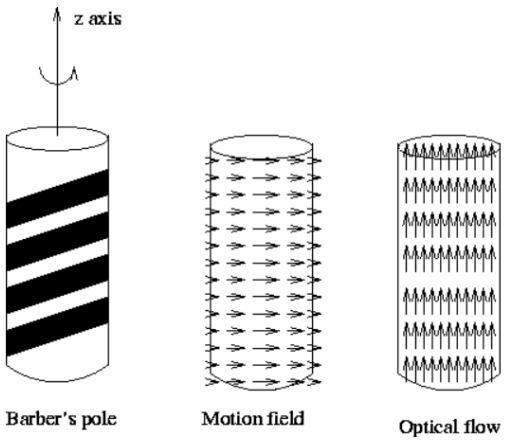
1.1. Motion Field

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**!

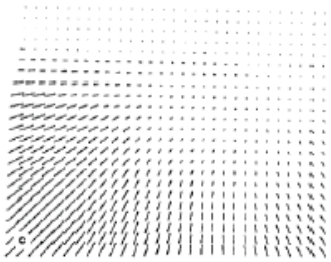


1.2. Optical Flow

Optical flow is the 2D velocity field **describing** the **apparent motion** in the images. Optical flow focuses on local pixel motion within the image plane (How do the pixels move after a timestep t?). Optical Flow and Motion Field are **not** the same! . For example, the motion field and optical flow of a rotating barber's pole are different, as illustrated in the figure below.

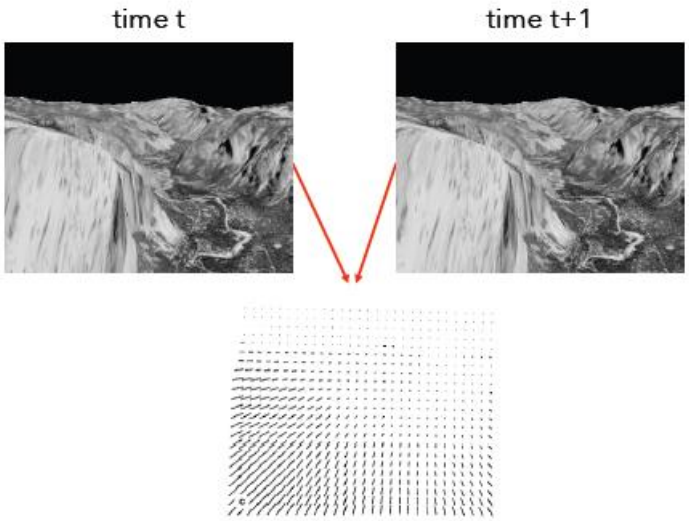


For **every pixel** we want to obtain a **vector** that tells us how it moved. This vector has a horizontal component $u(x, y)$ and vertical component $v(x, y)$. With that we can create an **optical flow field** as displayed in the next figure:



The intensity function now gets an additional variable t! Hence $I(x, y, t)$ 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. Here is an example:



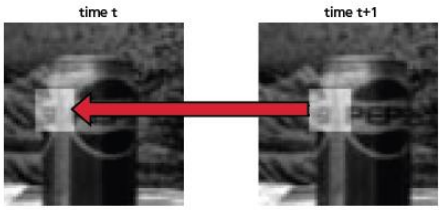
2. Computing Flow

2.1. Two important Assumptions

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

2.1.1. Brightness Constancy

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



Assumption 1:

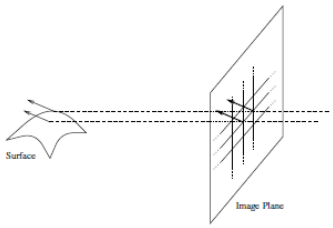
$$I(x + u(x,y), y + v(x,y), t+1) = I(x,y,t)$$

shifted by horizontal flow

shifted by vertical flow

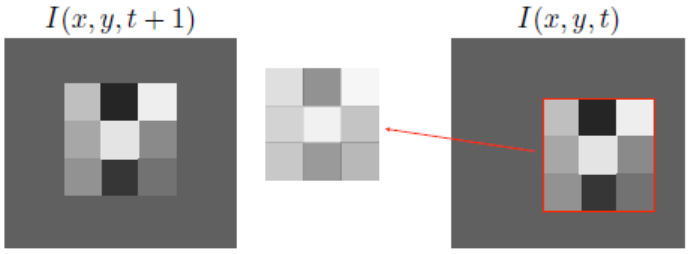
2.1.2. Spatial Coherence

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.



2.2. 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:

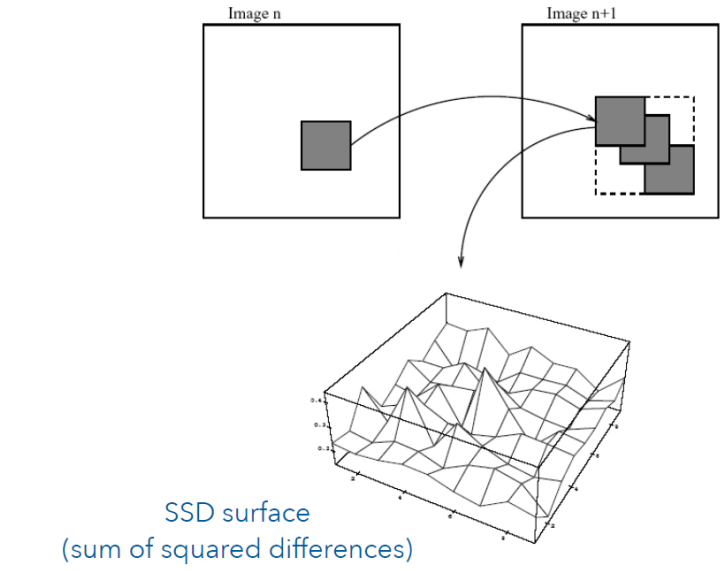


$$E_{SSD}(u,v) = \sum_{(x,y)\in R} \left(I(x+u, y+v, t+1) - I(x,y,t)\right)^2$$

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 (see next figure)?



2.3. Optical Flow Constraint Equation

The main problem why $E_{SSD}(u, v)$ 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** and the brightness **varies smoothly** 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), also called linearized brightness constancy constraint:

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

Another better notation is:

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

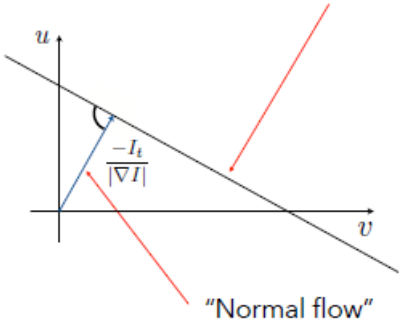
$$\downarrow$$

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

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

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

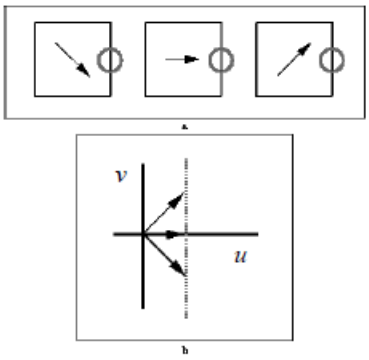
$$u \cdot I_x + v \cdot I_y + I_t = 0$$



2.4. 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. No matter in which direction you would move according to the flow (all 3 arrows possible), you would only see the movement perpendicular to v. Therefore, the **OFCE** is insufficient!

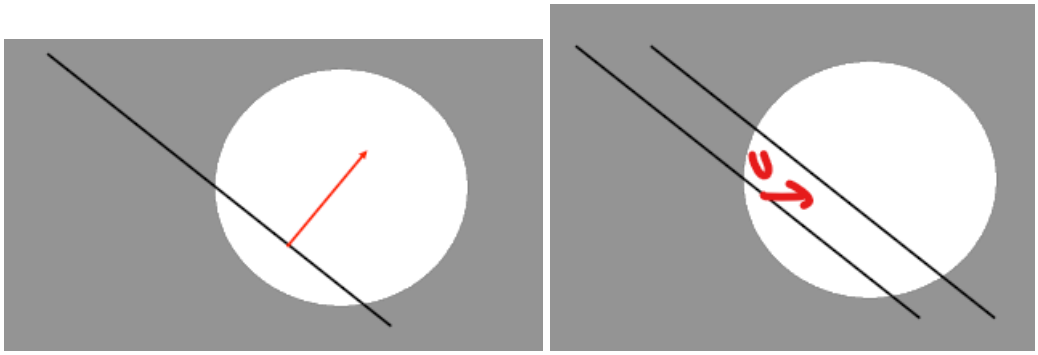
$$\begin{aligned} u \cdot I_x + v \cdot I_y &= -I_t \\ u \cdot I_x + 0 \cdot I_y &= -I_t \\ u \cdot I_x &= -I_t \\ u &= -\frac{I_t}{I_x} \end{aligned}$$



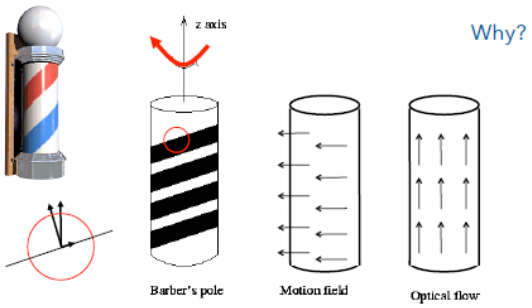
v could be anything!

2.4.1. Examples

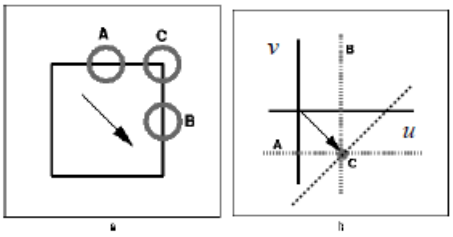
We can only measure **normal velocity** (perpendicular to the edge), because we are looking at a small region for example this pole below might not move diagonally (in direction of the normal velocity) but to the right.



Another popular example is the barber pole illusion. Here the optical flow and motion field differ.



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



3. Lukas Kanade

3.1. 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?

3.2. Structure Tensor

We differentiate $E_{SSD}(u,v)$ with respect to u and v and set this to zero

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

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

We can write this cleaner to:

$$\frac{\partial E_{SSD}}{\partial u} \approx 2\sum_R (u \cdot I_x + v \cdot I_y + I_t) I_x = 0$$
$$\frac{\partial E_{SSD}}{\partial v} \approx 2\sum_R (u \cdot I_x + v \cdot I_y + I_t) I_y = 0$$

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**:

$$\left[\sum_R I_x^2\right] u + \left[\sum_R I_x I_y\right] v = -\left[\sum_R I_x I_t\right]$$
$$\left[\sum_R I_x I_y\right] u + \left[\sum_R I_y^2\right] v = -\left[\sum_R I_y I_t\right]$$

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

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!

We can 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 called **Lucas-Kanade**: An iterative image registration technique with an application to stereo vision.

Very important questions:

- **What happens if the region is homogeneous?** Left term **is not** invertible hence we can't use Lukas-Kanade anymore!
- **What happens if there is a single edge?** Left term **is not** invertible hence we can't use Lukas-Kanade anymore!

- **What happens if there is a corner?** Left term **is** invertible

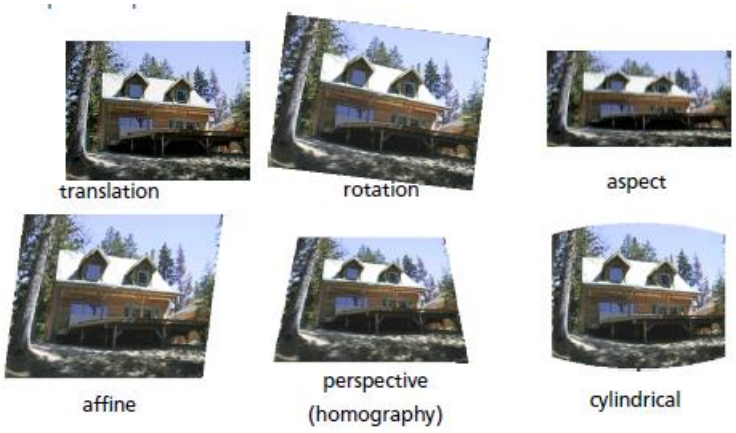
3.3. Image Registration

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:

1. Compute the flow with the entire images.
2. Shift the second image toward the first based on the flow
3. Iterate until convergence.

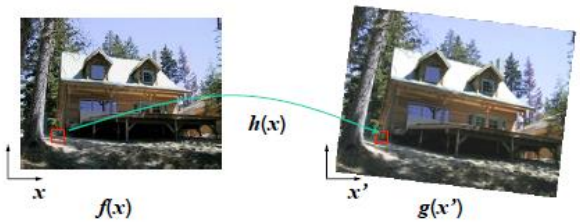
But how do we “shift” an image? We can use **Forward** or **Inverse** Warping!

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:

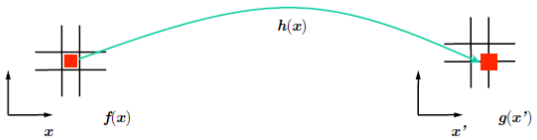


3.3.1. Forward Warping

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

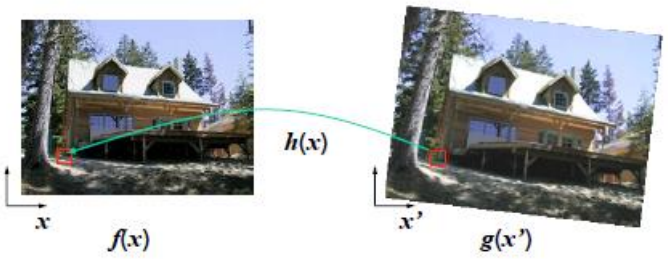


What if pixel lands “between” two pixels? Answer: Add “contribution” to several pixels, normalize later (splatting).

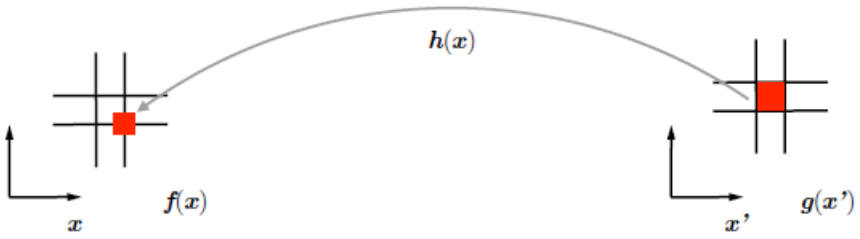


3.3.2. Inverse Warping

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.

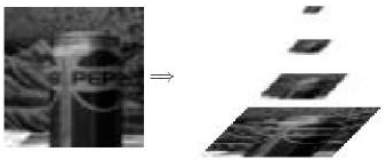


Possible interpolation filters: nearest neighbor, **bilinear**, bicubic (interpolating). This very important and needed to prevent "jaggies". When iteratively warping, always warp the **original** image!

3.4. Discussion

How do we "shift" an image? We use image warping to shift the image. But there is still one problem: 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!)

3.5. Dense LK Flow

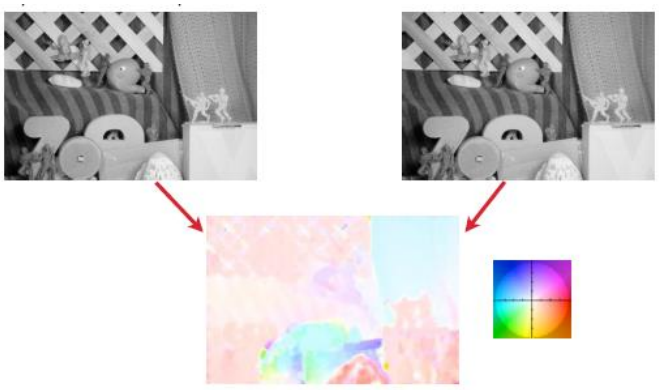
What if we now want to estimate **dense flow**? We can just take the region R to be a small region **around every pixel** and compute a flow vector for every pixel. But we face the same problems as before: The LK method only works for **small motions**. Two workarounds (use both):

- Iterative estimation
- Coarse-to-fine estimation

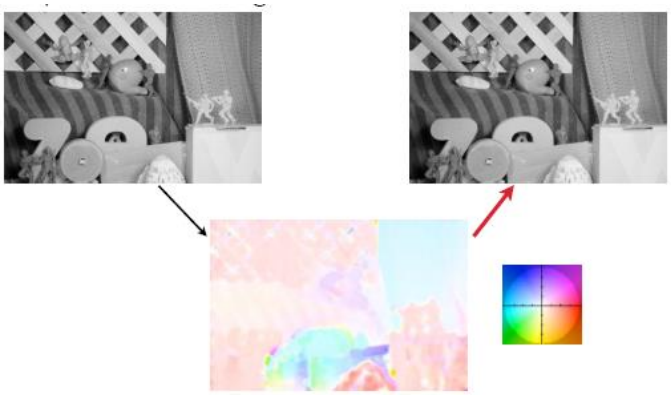
3.5.1. Iterative estimation

We will cover this using an example:

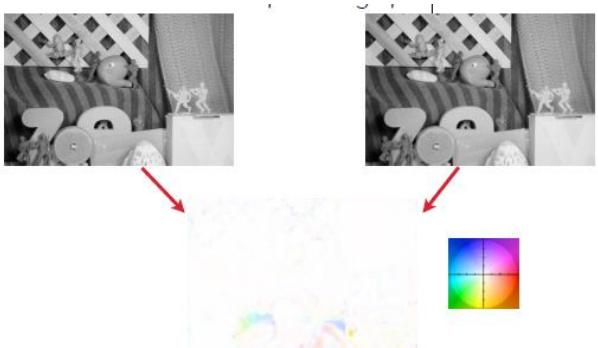
1. Take image pair and compute LK flow (21x21 window). The window is so big because we need to make sure its invertible later (white colour means we have no motion)



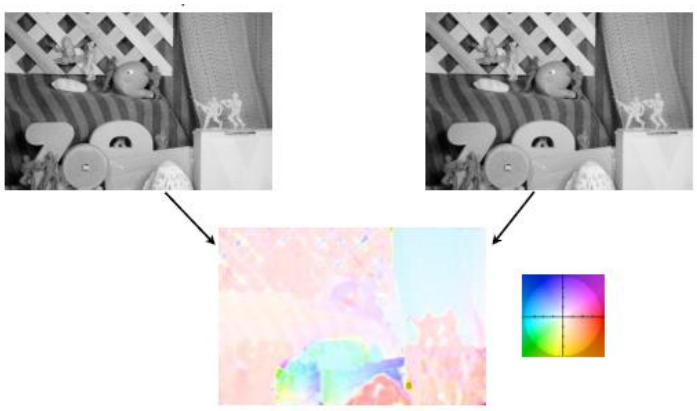
2. Backward warp the second image towards the first image



3. Estimate incremental flow from warped image pair



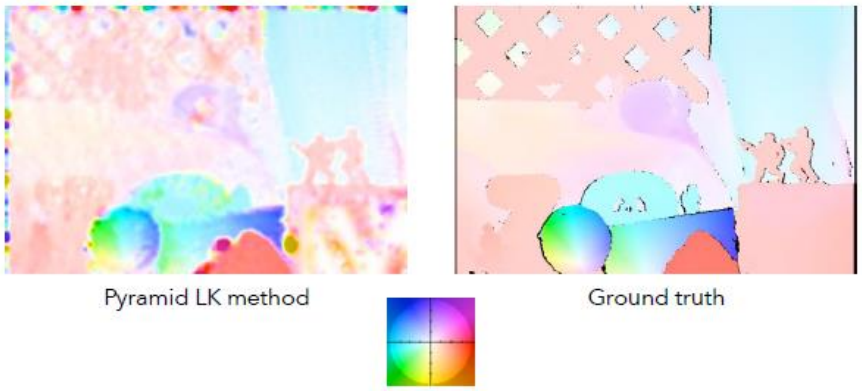
4. Add incremental flow to previous estimate



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

3.5.2. Coarse-to-fine Estimation

Of course, 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):



3.5.3. How do we get the ground truth?

For many years, there were only very few image sequences with ground truth flow available. All of them were synthetic and furthermore very simple. Middlebury optical flow benchmark [Baker et al., 07]: Contains a number of complex synthetic and real sequences with ground truth. Real sequences are captured using normal + UV light.

3.5.4. What is the problem?

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. Introduction

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!

1.1. Naïve View-Based Approach

Simple line-based object model is not as easy as it seems. More constrained setting:

How can we find the mouth? How can we recognize the “expression”?

1. Create database of mouth templates
2. Search every image region (at every scale)
3. Compare each template with it
4. Choose the best match

1.2. Appearance-Based Object Recognition

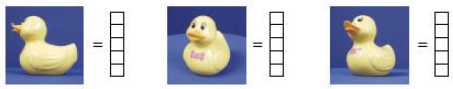
Basic assumptions:

- 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

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!

1.3. 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

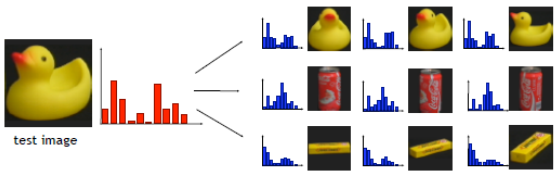
Feature representation can be divided in two main parts:

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

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

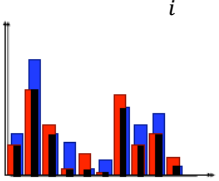
2.1. Colour Histogram Approach

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.



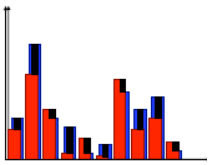
How do we even measure the histogram comparisons? There are quite a few methods:

- **Histogram intersection:** Measures the common part of both histograms. The range is mostly $[0,1]$.

Normalized Histograms [0,1]	**Unnormalized histograms**
$\cap(Q,V) = \sum_i \min(q_i, v_i)$ 	$\cap(Q,V) = \frac{1}{2}\left(\frac{\sum_i \min(q_i,v_i)}{\sum_i q_i} + \frac{\sum_i \min(q_i,v_i)}{\sum_i v_i}\right)$

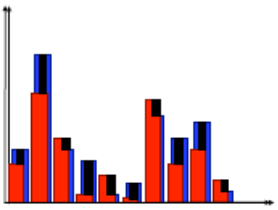
- **Euclidean Distance:** Focuses on the differences between the histograms. The Range is $[0, \infty]$. All cells are weighted equally. Not very discriminative.

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



- **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. χ^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.

2.1.1. Recognition using Histograms

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.

2.1.2. Discussion

Color histograms for recognition work surprisingly well in favourable conditions.

Advantages:

- Invariant to object translations
- Invariant to image rotations
- Slowly changing for out-of-plane rotations

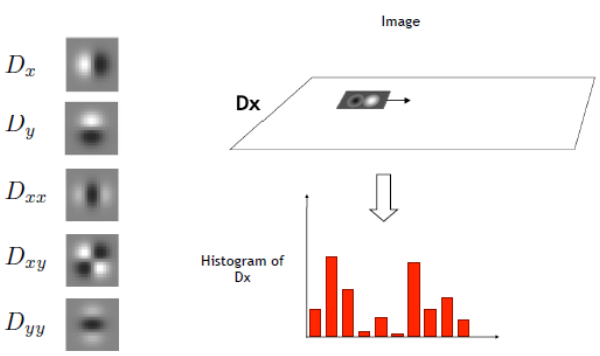
- No perfect segmentation necessary
- Histograms change gradually when part of the object is occluded
- Possible to recognize deformable objects

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.

2.2. Multidimensional receptive field histograms

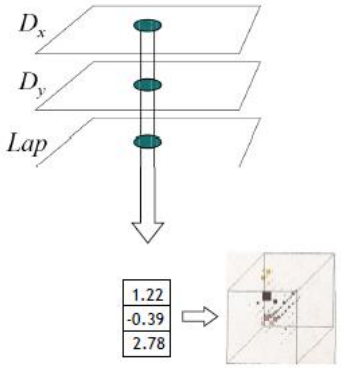
Histograms can be made out of derivatives. General receptive field histograms can be build using any local descriptors (e.g., filter, filter combination).



Some examples are:

- Gradient magnitude: $Mag = \sqrt{D_x^2 + D_y^2}$
- Gradient direction: $Dir = \arctan \frac{D_y}{D_x}$
- Laplacian: $Lap = D_{xx} + D_{yy}$

Multidimensional Histograms are a 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.



2.3. Summary

In Appearance-based object recognition we are using global representations with full and without any spatial information.

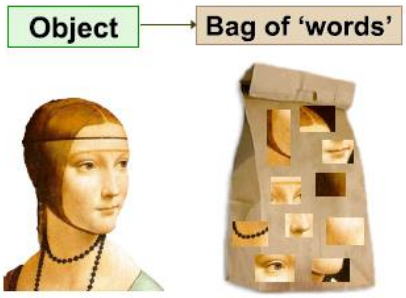
We looked at Histograms: Color histograms, Histogram comparison measures and Multidimensional histograms.

So far only object identification / instance recognition was the goal. The histogram idea can be also used for object categorization... stay tuned.

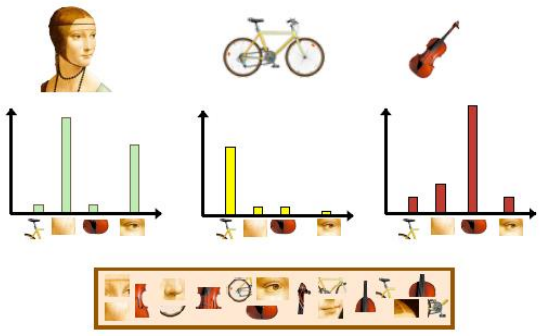
3. Bag-of-Words Model

3.1. Overview

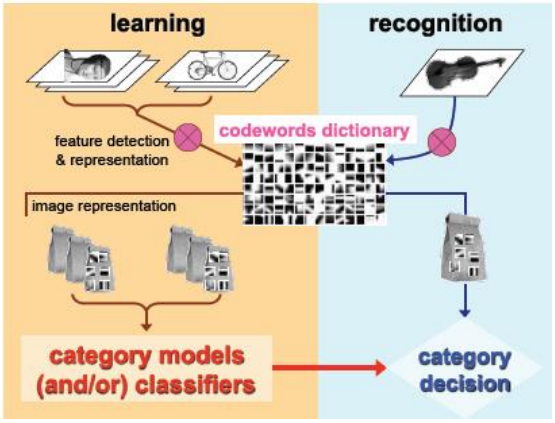
Assume we want to recognize an object like e.g., a face. We know a face has specific features like 1 nose, 2 eyes etc. The idea is to build a dictionary of visual words called **"Bag of words"** which contain these specific features of the face (nose, eyes etc.).



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**.



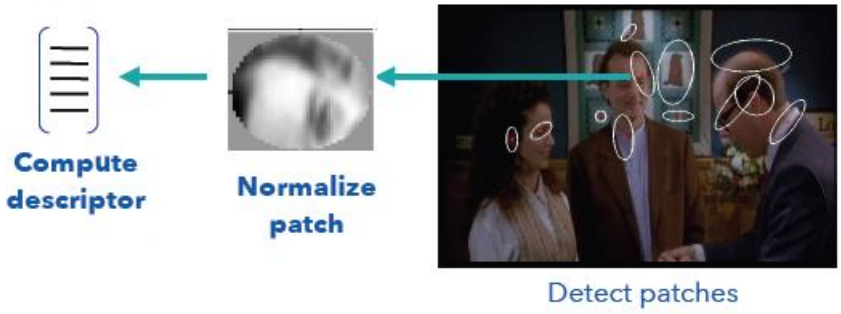
3.2. Object Representation & Learning

The learning phase is done in 3 steps:

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

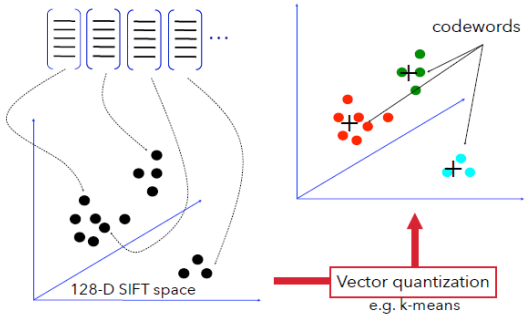
3.2.1. Feature detection and representation

This is where the sparse feature representation comes in. As already covered there are 2 main steps: 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.



3.2.2. 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.

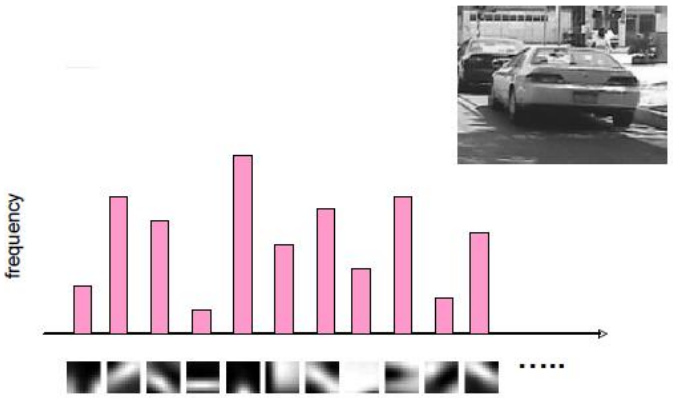


This is how K-means clustering works:

1: Specify the number k of clusters to assign
2: Randomly initialize k centroids (cluster centers)
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

3.2.3. 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.



Then do the same for dogs, cats, or any other images we use!

4. Excursion into Machine Learning

In Machine learning we develop a machine/an algorithm that learns to perform a task from experience. To put it more abstractly:

- Our task is to learn a mapping from input to output.
- Put differently, we want to predict the output from the input.

Mathematically we can describe this as follows:

$$f: I \rightarrow O$$
$$y = f(x; \theta), x \in I, y \in O, \theta \in \Theta$$

where:

- $x \in I$ is the input e.g. images, text, other measurements

- $y \in O$ is the output
- $\theta \in \Theta$ are the parameter(s) that is/are being learned

4.1. Classification vs Regression

In **classification** we learn a mapping into a discrete space, e.g.:

- $O = \{0,1,2,3, \dots\}$
- $O = \{verb, noun, nounphrase, \dots\}$

Some examples: Spam / not spam, sea bass vs. salmon, parsing a sentence, recognizing digits, etc.

In **regression** we learn a mapping into a continuous space, e.g.:

- $O = R$
- $O = R^3$

Some examples are “Curve fitting”, financial analysis,etc.

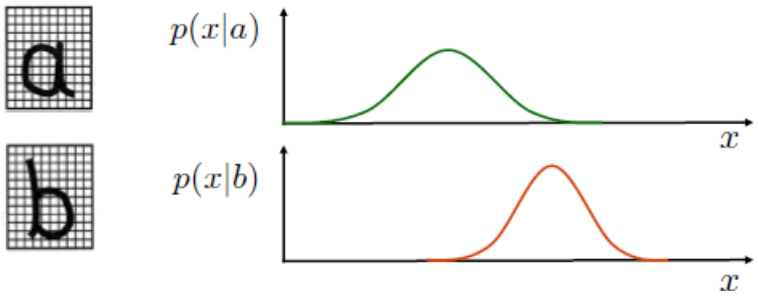
4.2. General Paradigm

1. Split the dataset into **training** and **test set**.
2. **Training set:** Learn the model (learned parameters θ)
3. **Test set:** Is different form training data and I used to predict the output with the learned model

4.3. Bayesian Decision Theory

1st concept: Class conditional probabilities $p(x|C_k)$

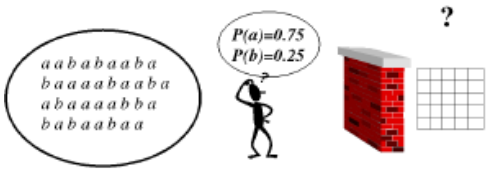
- Probability of making an observation x knowing that it comes from some class C_k
- Here x is a feature (vector)
- x measures / describes properties of the data



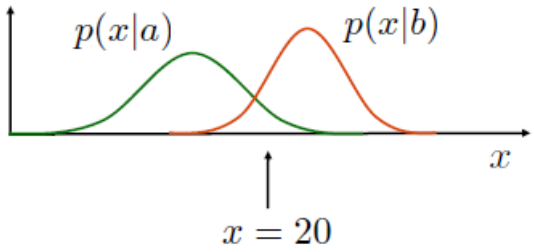
2nd concept: Class priors $p(C_k)$

- A priori (past) probability of a data point belonging to a particular class)
- Example:

$$C_1 = a \quad p(C_1) = 0.75$$
$$C_2 = b \quad p(C_2) = 0.25$$
$$\sum_k p(C_k) = 1$$



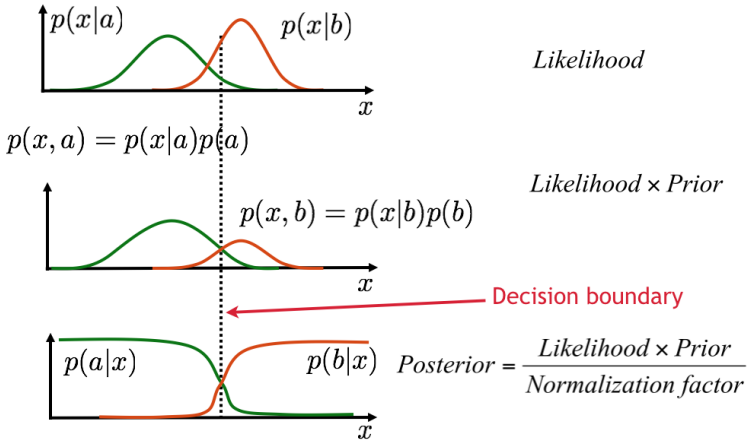
Question: How do we decide which class the data point belongs to in the next figure?



Here $p(x|a) = p(x|b)$ so what now? Since $p(a) > p(b)$ we should decide for class a! We can formalize this using **Bayes' theorem**!

$$p(C_k|x) = \frac{p(x|C_k)p(C_k)}{p(x)} = \frac{p(x|C_k)p(C_k)}{\sum_j p(x|C_j)p(C_j)}$$

$$Posterior = \frac{Likelihood\ xPrior}{Normalization\ Factor}$$



The **optimal decision rule** is to decide for C_1 if

$$\begin{aligned} & p(C_1|x) > p(C_2|x) \\ \leftrightarrow & \frac{p(x|C_1)p(C_1)}{p(x)} > \frac{p(x|C_2)p(C_2)}{p(x)} \\ \leftrightarrow & p(x|C_1)p(C_1) > p(x|C_2)p(C_2) \\ \leftrightarrow & \frac{p(x|C_1)}{p(x|C_2)} > \frac{p(C_2)}{p(C_1)}\ (LikelihoodRatio\ Test) \end{aligned}$$

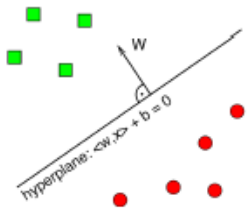
A classifier obeying this rule is called **Bayes Optimal Classifier**. The *decision boundary* is the point where $\frac{p(x|C_1)}{p(x|C_2)} = \frac{p(C_2)}{p(C_1)}$. This line (or curve) can then be drawn into some graph and is the point where the classifier "switches" to the other class. This is most of the time only used for understanding what the classifier does than for real application (however, understanding what happens is important).

4.4. Discriminative vs Generative Models

	Discriminative model	**Generative model**
Goal	Directly estimate $p(C_k\|x)$	Estimate $p(x\|C_k)$ to then deduce $p(C_k\|x)$
What's learned?	Decision boundary	Probability distributions of the data
Examples	SVM, Regression	Naïve Bayes

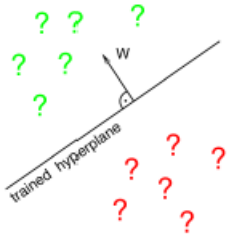
4.5. Discriminative Functions

A Linear hyperplane $y(\boldsymbol{x}) = w^T\boldsymbol{x} + \boldsymbol{b}$ is trained on samples of both classes C_1(■) and C_2(∗).

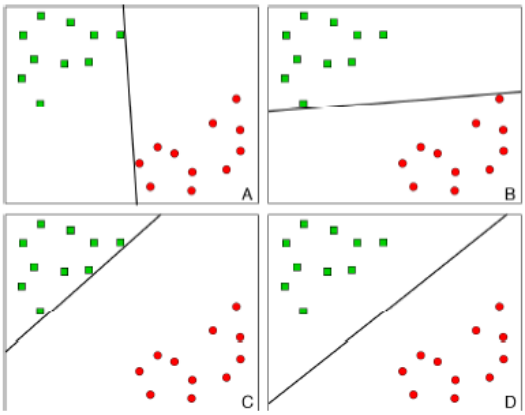


Classification with trained hyperplane:

- If $y(x) > 0$ decide for class C_1(?).
- If $y(x) < 0$ decide for class C_2(?)



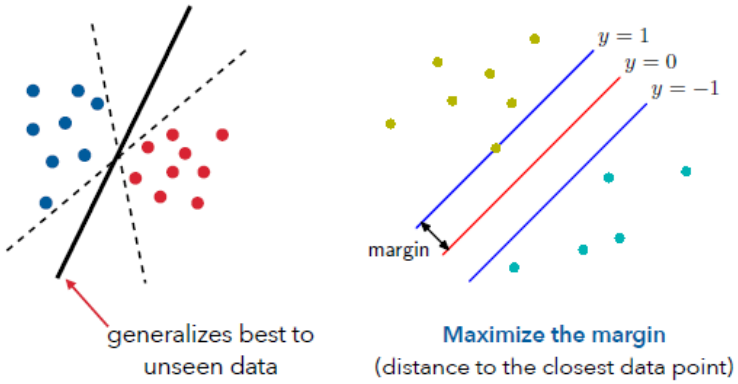
Question: Which Hyperplane is best and why?



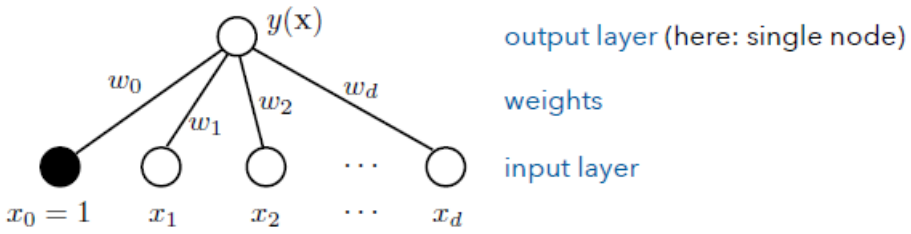
Answer: The best hyperplane is the one in image D. This is because it provides the maximum margin between the two classes of data points (green squares and red circles). This maximum margin ensures better model generalization and performance, reducing the risk of misclassification.

4.6. Support Vector Machines

Intuitively: We should find the hyperplane that maximizes the margin (distance to the closest data point) between two or more classes. This ensures the best generalization to unseen data.

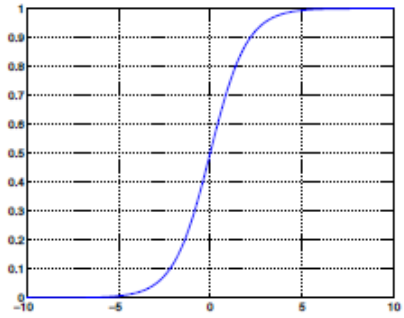


4.7. Single layer network



- **Linear Outputs:** $y(x) = \boldsymbol{w}^T\boldsymbol{x} + w_0 = \sum_{i=1}^{d} w_i x_i + w_0$
- **Non-linear** (e.g. logistic) **outputs**, e.g. using the sigmoid ("S-shaped") function which squashes real numbers into the interval $[-1,1]$:

$$y(x) = \sigma\left(\boldsymbol{w}^T\boldsymbol{x} + w_0\right)$$
$$\sigma(a) = \frac{1}{1 + e^{-a}}$$



4.8. Multi-Class Network

Can be used to do multidimensional linear regression. But also, multi-class linear classification. Nonlinear extension is straightforward.

- **Linear Outputs:** $y_k(x) = \sum_{i=0}^{d} W_{ki} x_i$
- **Non-linear** (e.g. logistic) **outputs:** $y_k(x) = \sigma\left(\sum_{i=0}^{d} W_{ki} x_i\right)$

In classification there isa problem of using sigmoid because the class proabailites do not sum to 1 hence SoftMax nonlinearity is used:

$$y_k(x) = \frac{e^{a_k(x)}}{\sum_{l=1}^{c} e^{a_l(x)}} = \frac{e^{\sum_{i=0}^{d} W_{ki} x_i}}{\sum_{l=1}^{c} e^{\sum_{j=0}^{d} W_{lj} x_j}}$$

4.9. Learning

How does supervised learning of the weights W work? We are given

- N training data points: $X = [x^1, \dots, x^N]$
- C target values for each data point: $T_k = \left[t_k^1, \dots, t_k^n\right]$

First Compute c outputs of the network $y_k(x^n; W)$ then minimize loss function:

$$E(W) = \frac{1}{N} \sum_{n=1}^{N} E^n(W) = \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{c} \mathcal{L}(y_k(x^n; W), t_k^n)$$

$$E^n(W) = \sum_{k=1}^{c} \mathcal{L}(y_k(x^n; W), t_k^n)$$

For classification we often use log-loss (cross entropy):

$$\mathcal{L}(y, t) = -t \log y - (1 - t) \log(1 - y)$$

To minimize the loss function we are using **gradient descent.**

4.10. Gradient Descent: Single-layer neural net with linear activation

Training a single-layer neural net with linear activation results in the following gradient:

$$\frac{\partial E^n(W)}{\partial W_{l,j}} = \sum_{k=1}^{c} \frac{\partial \mathcal{L}(y_k(x^n; W), t_k^n)}{\partial y_k} \frac{\partial y_k(x^n; W)}{\partial W_{l,j}}$$

$$= \frac{\partial \mathcal{L}(y_l(x^n; W), t_k^n)}{\partial y_l} x_j$$
$$= \mathcal{L}'(y_l(x^n; W), t_l^n) x_j$$

4.10.1.Batch Learning

The gradient is computed using **all** training data points. However, this is **computationally expensive**!

$$W_{lj}^{t+1} = -\eta \frac{\partial E(W)}{\partial W_{lj}}|_{W^{(t)}}$$

$$\frac{\partial E(W)}{\partial W_{lj}} = \sum_{n=1}^{N} \frac{\partial E(W)}{\partial W_{lj}}$$

where η is the learning rate.

4.10.2.Sequential or pattern-based update

Computation of the gradient based on a **single t**raining data point! More **efficient**, but the gradient can be **noisy** (Intermediate solution: Use **small** training **batches)**

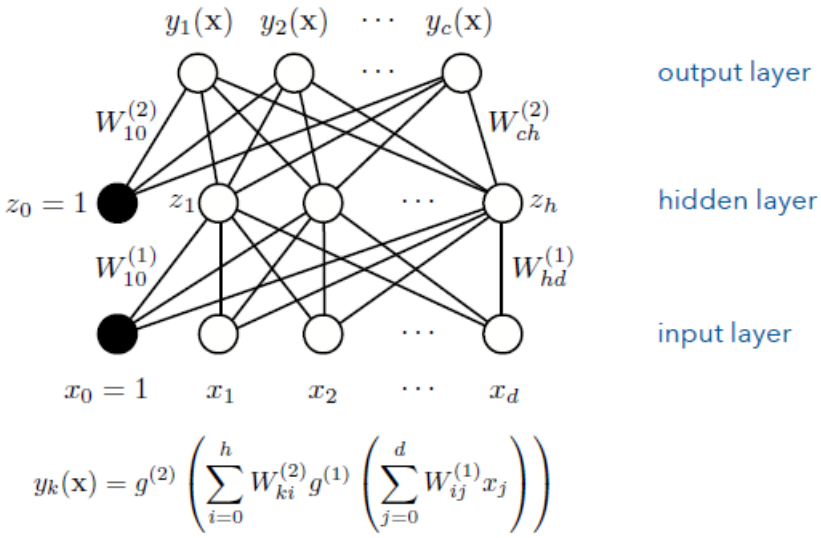
4.11. Gradient Descent: Single-layer neural net with non-linear, differentiable activation function

$$y_k(x^n) = g\big(a_k(x^n)\big) = g\left(\sum_{i=1}^{d} W_{ki} x_i^n\right)$$

$$\frac{\partial E^n(W)}{\partial W_{lj}} = \mathcal{L}'(y_l(x^n; W), t_l^n) \cdot x_j \cdot g'(a_l(x^n))$$

For example, the sigmoid network: $\sigma'(a) = \sigma(a)(1 - \sigma(a))$

4.12. Multi-Layer Perceptron



$$y_k(\mathbf{x}) = g^{(2)}\left(\sum_{i=0}^{h} W_{ki}^{(2)} g^{(1)}\left(\sum_{j=0}^{d} W_{ij}^{(1)} x_j\right)\right)$$

4.12.1.Learning

1. Forward pass/propagation:
    a. Compute output unit activations: $y_k(x^n)$
    b. Compute hidden unit activations: $z_i(x^n)$
2. Backward pass/propagation
    a. Compute output error: δ_k^n
    b. Compute hidden error: δ_i^n

- **Activation functions** $g^{(k)}$: $g^{(2)}(a) = \sigma(a), g^{(1)}(a) = a$
- The hidden layer can have an arbitrary number of nodes. There can also be multiple hidden layers.
- **Universal approximators:** A 2-layer network (1 hidden layer) can approximate any continuous function of a compact domain arbitrarily well! (assuming sufficient hidden nodes)

4.12.2.Gradient Descent

Loss of network

$$E(W) = \frac{1}{N}\sum_{n=1}^{N} E^n(W) = \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{c} \mathcal{L}(y_k(\mathbf{x}^n; W), t_k^n)$$

$$y_k(\mathbf{x}^n) = g^{(2)}\left(\sum_{i=0}^{h} W_{ki}^{(2)} g^{(1)}\left(\sum_{j=0}^{d} W_{ij}^{(1)} x_j^n\right)\right)$$

$$= g^{(2)}\left(\sum_{i=0}^{h} W_{ki}^{(2)} z_i(\mathbf{x}^n)\right)$$

with $$z_i(\mathbf{x}^n) = g^{(1)}\left(\sum_{j=0}^{d} W_{ij}^{(1)} x_j^n\right)$$

Compute output error (Weights between output and hidden layer)

$$\frac{\partial E^n(W)}{\partial y_k} = \mathcal{L}'(y_k(\mathbf{x}^n; W), t_k^n) =: \delta_k^n$$

Now we can compute the weight derivatives (Weights between output and hidden layer):

$$\frac{\partial E^n(W)}{\partial W_{lj}^{(2)}} = \frac{\partial E^n(W)}{\partial y_k}\frac{\partial y_k(\mathbf{x}^n; W)}{\partial W_{lj}^{(2)}}$$

$$= \mathcal{L}'(y_l(\mathbf{x}^n; W), t_l^n) \cdot g^{(2)'}\left(\sum_{i=0}^{h} W_{li}^{(2)} z_i(\mathbf{x}^n)\right) \cdot z_j(\mathbf{x}^n)$$

$$= \delta_l^n \cdot g^{(2)'}\left(\sum_{i=0}^{h} W_{li}^{(2)} z_i(\mathbf{x}^n)\right) \cdot z_j(\mathbf{x}^n)$$

Again, compute error (Weights between input and hidden layer):

$$\begin{aligned}\frac{\partial E^n(W)}{\partial z_l} &= \sum_{k=1}^{c} \mathcal{L}'(y_k(\mathbf{x}^n; W), t_k^n) \frac{\partial y_k(\mathbf{x}^n; W)}{z_l} \\ &= \sum_{k=1}^{c} \mathcal{L}'(y_k(\mathbf{x}^n; W), t_l^n) \cdot g^{(2)'} \left(\sum_{i=0}^{h} W_{ki}^{(2)} z_i(\mathbf{x}^n) \right) \cdot W_{kl}^{(2)} \\ &= \sum_{k=1}^{c} \delta_k^n \cdot g^{(2)'} \left(\sum_{i=0}^{h} W_{ki}^{(2)} z_i(\mathbf{x}^n) \right) \cdot W_{kl}^{(2)} =: \hat{\delta}_l^n\end{aligned}$$

Again, compute the weight derivatives (Weights between input and hidden layer:

$$\begin{aligned}\frac{\partial E^n(W)}{\partial W_{ji}^{(1)}} &= \frac{\partial E^n(W)}{\partial z_j} \cdot g^{(1)'} \left(\sum_{i=0}^{d} W_{ji}^{(1)} x_i \right) \cdot x_i \\ &= \hat{\delta}_j^n \cdot g^{(1)'} \left(\sum_{i=0}^{d} W_{ji}^{(1)} x_i \right) \cdot x_i\end{aligned}$$

It is important to note that the weight derivative form $W^{(1)}$has the same basic form as for $W^{(2)}$. We can derive **general rules for backpropagation**. All we need is the **chain rule**!

4.12.3.Discussion

Multi-layer perceptrons are usually trained using backpropagation:

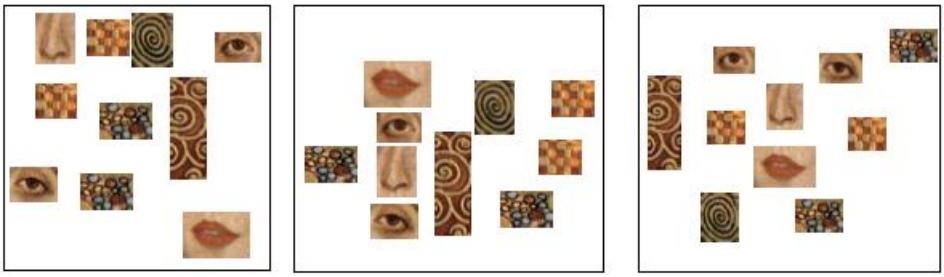
- Non-convex, many local optima.
- Can get stuck in poor local optima.
- The design of a working backprop algorithm is somewhat of a “black art”
- Because of that, they were largely out of fashion until ~10 years ago
- Derivatives are not done done by hand (anymore), but with autodiff techniques
- When trained properly, these models work very well

4.13. Summary & Discussion

The benefits of Bag-of-words representation are many:

- Sparse representation of object category
- Many machine learning methods are directly applicable
- Robust to occlusions
- Allows sharing of representation between multiple classes

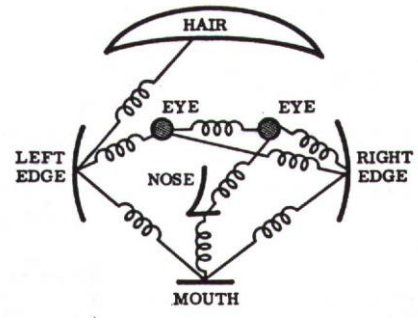
However, there are also problems e.g. what about **spatial information**? For example, let’s look at the following example:



All have equal probability for bag-of-words methods! Location information is important!

5. Part-based Representation

We see the object as a set of parts, e.g. a generative representation like pictorial structures.



Model:

- Relative locations between parts
- Appearance of part

Issues:

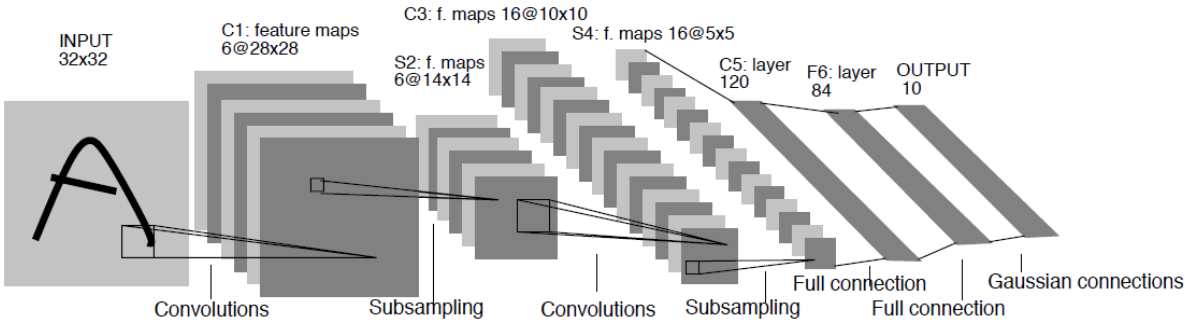
- How to model location
- How to represent appearance
- How to handle occlusion/clutter

There is also a **correspondence problem**. Assume we have a model with P parts, an image with N possible assignments for each part and consider the mapping is 1-1. Then we would have to try N^P combinations. This is computationally not feasible!

Deep Neural Networks

1. Convolutional Neural Networks

Convolutional Nurla Networks are Neural networks with a specialized, local connectivity structure.



The CNN exploits two properties of images:

1. Dependencies are local: No need to have each unit connect to every pixel
2. Spatially stationary statistics: Translation invariant dependencies, Only approximately true

Important consequences: Convolutions with local filters require many fewer parameters than a standard MLP (fully connected) layer. It can deal with entire images!

1.1. History

History: Multistage Hubel-Wiesel Architecture

- Stacks multiple stages of simple cells / complex cells layers.
- Higher stages compute more global, more invariant features
- Classification layer on top

History:

- Neocognitron [Fukushima 1971-1982]
- Convolutional Networks [LeCun 1988-2007]
- HMAX [Poggio 2002-2006]
- Many others….

Early ConvNet Successes:

- Handwritten text/digits: MNIST, Arabic & Chinese
- Simpler recognition benchmarks: CIFAR-10, Traffic sign recognition
- But less good at more complex datasets: Caltech-101/256 (few training examples)

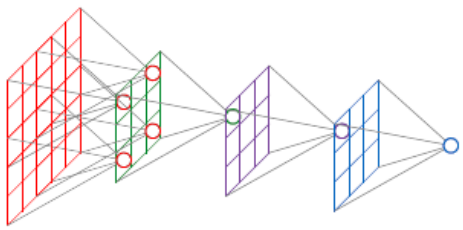
1.2. How it works

1. **Input:** The input to a CNN is typically an image, which is a matrix of pixel intensities.

2. **Convolution:** The first step in a CNN is the convolution of the input image with a set of learnable filters or kernels. The result are feature maps!
   - **The filters are learned during training**: Train convolutional filters by back-propagating classification error (Supervised Learning). The learned filters are tilted and local!
   - Each filter is applied at **every location** on the image
   - **Translation equivariance**: If an object in an image moves (translation), the corresponding features in the CNN's output (feature maps) move in the same way!
   - Layer in a CNN takes as input the output from the layer directly above it.
   - **Look at Local Window:** The filter is slid across the input image or feature map, covering a small window of pixels at each step.
   - **No Weight Tying:** In a convolutional layer, the same set of weights (the filter) is used for every neuron in the layer.

3. **Non-Linearity:** After convolution, a non-linearity is applied in the form of the **Rectified Linear Unit (ReLU)** function.
   - Applied per pixel
   - $output = max(0, input)$: This means that all negative values (black=negative, white=positive) in the output feature map are set to zero.
   - Other activation functions can also be used such as Tanh, Sigmoid, or Parameterized ReLU.

4. **Spatial Pooling**: The next step is spatial pooling, which reduces the dimensionality of the feature map while maintaining the most important information.
   - Pooling can be done over non-overlapping or overlapping regions
   - Can be a sum or max operation.
   - Role of Pooling: invariance to small transformations and larger receptive fields (see more of output)
   - **Receptive field**: Receptive field of the first layer is the filter size. The Receptive field (w.r.t. input image) of a deeper layer depends on all previous layers' filter size and strides Correspondence between a feature map pixel and an image pixel is not unique



5. **Normalization**: This is an optional step where normalization is performed across the data or features.
6. **Output:** The final output is a set of high-level features which are then typically fed into a fully connected layer for the task at hand (like classification or regression).

This process (steps 2-4) can be repeated multiple times, with the output feature map of one layer serving as the input to the next layer. This allows the network to learn increasingly complex features at each layer.

1.3. Training CNNs

Batch Normalization: Normalize the activations in each minibatch. This process speeds up training, allows for the use of higher learning rates, and provides regularization, avoiding overfitting

Loss function: For classification, it is typical to use the **cross entropy**:

$$H(y,p) = -\sum_{i,c} y_{i,c} \log p_{i,c}$$

Where y are the true labels (one hot coding) and p are the SoftMax probabilities. However, there are many other possibilities like **regularization** (e.g. Dropout) or **regression losses.**

Initialization: Initializing weights with 0 does not work. Recall gradient:

$$\frac{\partial E^n(W)}{\partial W_{l,j}} = \mathcal{L}'(y_l(x^n;W), t_l^n) \cdot g'\big(a_l(x^n)\big) \cdot x_j$$

Instead, we use random weights. It is very common to use **Xavier** initialization were

$$mean = 0$$

$variance = \frac{1}{n_{in}}$ **or** $variance = \frac{2}{n_{in}+n_{out}}$ (dependent on the in- and out-degree of a node)

1.4. Architecture

Big issue: how to select the right network? Manual tuning of features → manual tuning of architectures. E.g. this involves manually choosing:

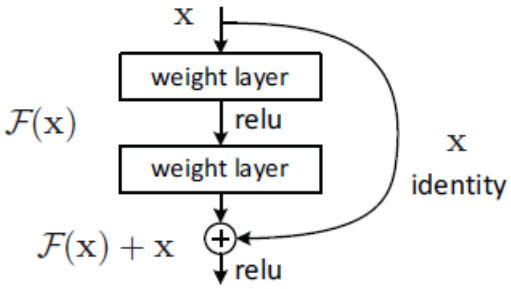
- Depth of network
- Width (i.e. number of feature maps)
- Parameter count (filter size)

There are many hyper-parameters like the **number of layers** or the **number of feature maps**. E.g. to demine the best architecture, we could use several methods like:

- **Cross-validation**
- **Grid search** (needs lots of GPUs)
- Smarter strategies:
    - **Random**
    - **Bayesian optimization** and related approaches
    - (Differentiable) **architecture search**

1.5. Training Big ConvNets

Very Deep Models can be hard to train! Really, really deep convnets don't train well! Key idea: introduce "pass through" into each layer. Thus only residual now needs to be learned! Such networks are called **Residual networks**:



We can also use **Stochastic Gradient Descent**. We compute (noisy estimate of) gradient on small batch of data and make a step. Note that we are taking as many steps as possible (even if they are noisy). There are different approaches on choosing the learning rate

- **Large initial learning rate**
- **Anneal learning rate**: Start large, slowly reduce. Explore different scales of energy surface

Another idea is to use **Momentum** in particular **Variants**!

Exam 1 (Total 90 points) ~ 120min

1. Explain concepts (4 general questions)

What's the main idea of non-maximum suppression?

Non-maximum suppression enables to improve the single response of edge detection which leads to thinner edges since using only the 1st derivative gives us thick edges. The idea can be explained in 2 steps and is about **removing** any pixels that do not belong to the edge!

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

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

Why use lens instead of pinhole?

A 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!

What is the use of Dimensionality Reduction in Object Recognition? (or something like that)

The goal is to find more (computationally) efficient **representations of views** and more (computationally) efficient **methods for image matching**! We are doing that by finding **lower dimensional** representation that captures the variability in the data. Then we are **searching** using this low-dimensional model! This also reduces the storage costs!

2. Image formation

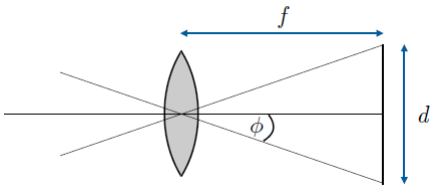
Characteristics of camera impact Depth of View?

The depth of field is the range where an object is approximately in focus. The depth of field is dependent on the aperture size! A small aperture size increases the depth of field, but we need more exposure to light. A larger aperture decreases the depth of filed but we don't need much exposure to light! So, there is a tradeoff.

Characteristics of camera impact Field of View?

The field of view depends on the **focal length** and the **size** of the camera sensor. It is given as:

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



A larger field of view equals a smaller focal length (camera is close to object). A smaller field of view equals a larger focal length (camera is close to object).

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

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

Plugging in the values gives us:

$$2 = 1.4$$

The thin lens formular is not satisfied hence the point is not in focus!

Moiré effect: Basically, explain what it is and how to prevent it!

The Moire effect is a visual effect that occurs when 2 similar patterns overlap, creating new, wavy unwanted stripes that go across the image which weren't originally there. The cause of the Moire effect are fine black and white details in images that are **misinterpreted as color information**. It can be prevented by using an **optical low-pass filter**, e.g. **artificial blur.**

Bayer Grid and Draw it on a sensor.

A camera captures color (RGB) using the Bayer grid **for each** individual pixel of the sensor. For every pixel we get an RGB triplet as an approximation. Usually, we have more green pixels around 50% (better for the human eye to distinguish stuff) and usually 25% red pixels and 25% blue pixels.

This is the resulting pattern.

B	G	B	G
G	R	G	R
B	G	B	G
G	R	G	R

Why is CV an inverse problem?

When we take an image a lot of information about the 3D world is lost. But that's exactly what we are interest in! to understand and reconstruct the 3d world based on images! For example, using stereo vison to calculate the depth to an object etc.

3. Filter

Given a matrix 1/8*[[1,0,-1],[2,0,-2],[1,0,-1]], what's this filter name and explain 2 operations of it.

This is Sobel filter! Used for edge detection, gradient calculation!

Name a pyramid, and why we use it? What is Aliasing and how can we avoid it?

Gaussian pyramids represent images at multiple scales (resolutions). We first apply Gaussian smoothing to avoid aliasing then down sample by taking every second pixel. These images are stacked like a pyramid. Gaussian pyramids can be used for example when we want to find an object in an image that is only clearly visible at a certain resolution or scale.

If we down sample an image by taking every second pixel we can't represent the details (high frequencies) cannot be represented anymore. Hence spatial frequencies will be misinterpreted (Aliasing)! So, the down sampled layers badly represent their top layers!

To avoid aliasing we can use gaussian smoothing!

Name a non-linear filter and explain what can it be used for?

When we use a linear filter to remove noise, we also remove the signal. That's not good the goal is to remove only the noise Here is where non-linear filters come in.

Median Filter: replace each pixel with median in a **neighborhood**, sort the pixels in local neighborhood, take the middle value -> computationally expensive.

Morphological filter: perform min/max-convolution with a structuring element (erosion (thickening): min operation, dilatation (thinning): max operation), get previous behavior with flat structuring element.

Given a matrix, apply 2 padding methods on the matrix.

Zero padding:

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	**2**	**3**	**8**	0	0
0	0	**4**	**6**	**7**	0	0
0	0	**5**	**1**	**9**	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Mirror padding:

9	7	5	1	9	4	8
1	6	4	6	7	6	1
8	3	**2**	**3**	**8**	3	2
7	6	**4**	**6**	**7**	6	4
9	1	**5**	**1**	**9**	1	5
3	6	4	6	7	6	3
8	7	5	3	8	4	2

4. PCA

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

PCA is a linear dimensionality reduction method! Hence it doesn't work for non-linear relationships! We also assume e.g. for PCA for images, that images of the same objects are similar and not random.

Which formula PCA optimizes? And given a dataset X, how PCA represents it.

PCA finds the direction u_1 which maximizes the variance of the data:

$$u_1 = \arg\max_u u^T C u \;\; s.t. u^T u = 1$$

List 2 advantages of using low-dimensional data!

Computationally efficient e.g. for matching templates with images

Lower storage: instead of a huge number of templates we can use smaller sub space to represent the variability in the data!

Assuming the mean of data is subtracted, how to compute PCA via SVD?

5. Interest points

How to make Harris detector more generalized?

E.g. we can add a weight to the structure tensor which they will turn it to the Harris Laplace Detector which is scale invariant!

What's the pros and cons of Harris detector?

Pro: Rotation invariant
Cons: Not scale Invariant

How to determine if two interest points are in different matches?

Use local descriptors such as Image patches, SIFT, Bank of filter responses (jet), Shape context.

How does Harris detector work?

1. Compute Gradient
2. Compute H
3. Compute Eigenvalues of H
4. Calc f
5. F > threshold
6. Find local maximum of f

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

Harris is not scale invariant! Harris Laplace solves it by using a criterion and automatic scale selection.

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

The Harris operator is an interest point detector and is given as:

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

$$\boldsymbol{H} = \sum_{(x,y)\in R} \nabla \boldsymbol{I}(x,y) \nabla \boldsymbol{I}(x,y)^T$$

H is the structure tensor!

6. Single/two view geometry

Something about why we should use conditioning and how?

Because we would 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.

How does RANSAC work?

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

How many iterations do we need? Use this

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	$k = \frac{\log(1-z)}{\log(1-w^d)}$

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

Training data: Input image pairs, output their known Homography e.g. known from estimating it with the 8-point algorithm. Don't forget numerical conditioning!

To handle outliers, we use RANSAC since correspondences can be wrong!

Loss function: MSE, Huber Loss, Homography based loss.

Explain the degrees of freedom of the homography.

The Homography is a (3x3) matrix and has 8 degrees of freedom. Scale is arbitrary that's why 8 and not 9 degrees of freedom. E.g. we deleted the **third** column of the projection matrix by using a simple trick setting the z-axis of the coordinate system of projected plane to 0.

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

To project a plane, we can set the z-axis of its coordinate system to 0. Hence, we can remove the third column of the projection matrix which results in the homography matrix!

What are some applications of a homography?

Panorama stitching

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

1-point correspondences give us 2 linear independent equations. The homography has 8 unknown parameters hence we need 4-point correspondences!

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

We are using numerical conditioning by shifting and scaling the points to be in the range of [-1,1]. For that we exploit the **inverse transform of the estimated** homography!

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

$$Ah = 0, s.t. \|\boldsymbol{h}\| = 1$$

$$A = USV^T$$

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

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

$$w = 0.3, z = 0.99, d = 4$$

$$k = \frac{\log(1-z)}{\log(1-w^d)} = 566{,}2$$

We need at least 567 trials!

7. Stereo

Why do we use image rectification?

If we don't have a binocular setup, we can **rewarp** the images, so they are a binocular setup. E.g. we map both image planes to a common plane parallel to the baseline. For that we require a homography for each image. After that the pixel transform is horizontal again!

How many correspondences does the essential matrix need? Explain

The essential matrix has 5 unknown parameters (scale is arbitrary). For each point correspondence we get 1 equation hence we need at least 5-poitn correspondences!

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

The fundamental matrix has 7 unknow parameters. We get 1 equation per 1-point correspondence hence we need at least 7-point correspondences for a nonlinear and 8-point correspondences for linear solution!

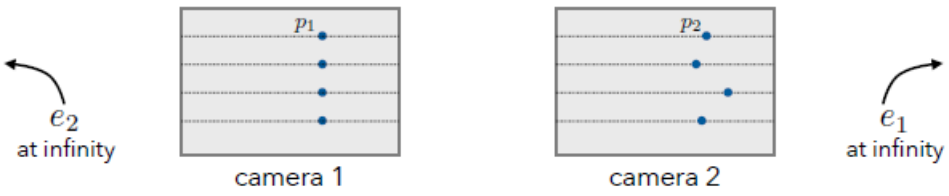
How are the fundamental matrix and the essential matrix related? (+what do you still need to go e.g. from fundamental to essential matrices)

$$\boldsymbol{F = K_1^{-1} E K_2^{-1}}$$

We need to transform form pixel coordinates back to the camera's coordinates!

Draw binocular setup, explain important definitions of epipoles!

- Epipoles are at **infinity**
- Epipolar lines are parallel



When does window based matching fail. Name two examples.

- textureless surfaces
- occlusions
- repetition
- non-Lambertian surfaces (glossy surfaces)
- specularities

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

- How do we choose the right **window size** m? Solve with **adaptive window**
- Mismatches often lead to relatively **poor results quality**!
    - Solve with similarity constraint: Corresponding regions in two images should be similar in appearance and non-corresponding regions should be different.
    - Another method is to use non-local correspondence constraints! But we need spatial regularity for that! Ensure uniqueness, ordereing, smoothness

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

Main difference is the essential matrix captures the relation of 2 views for calibrated cameras the fundamental matrix captures it for uncalibrated cameras too! The essential has fewer degrees of freedom!

8. Motion

What's the differences between motion field and optical flow 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 or object motion (pink or both!

Optical flow is the 2D velocity field **describing** the **apparent motion** in the images. Optical flow focuses on local pixel motion within the image plane (How do the pixels move after a timestep t?)

Optical Flow and Motion Field are **not** the same! For example, the motion field and optical flow of a rotating barber's pole are different, as illustrated in the figure below

How to solve aperture problem [under some condition]?

We are combining multiple constraints to get an estimate of the velocity not only the normal component! How? For that we will assume spatial coherence of the flow. E.g. the flow is constant in a region.

How are Depth estimation and optical flow related?

Both involve a disparity. In binocular stereo this is due to images taken from different viewpoints in this case the pixels have a disparity in x direction. In optical flow we want to obtain the apparent motion between images taken at 2 different time steps. The motion can be defined by a vector which is also kind of a disparity!

So, they are related by a disparity!

State the OFCE and name the assumptions!

- Small motions (since we used Taylor)
- Spatial Coherence in the image flow: Neighboring points belong to the same surface hence have similar 3D motions!
- Brightness constancy: Image measurements remain the same even when their location change!

Explain how Lucas-Kanade can be applied to fast motions? (LK only holds for small motions!)

With 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!)

9. BoW model

Explain how Bayers theorem is applied for BoW representation in classification task!

$$p(C_k|x) = \frac{p(x|C_k)p(C_k)}{p(x)}$$

What's pros and cons of colour histogram?

Explain the steps of how BoW features are created

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

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?

1 dictionary

10. Deep Learning

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?

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)

Define machine learning and classification/regression. <- i think this was task 9 (but the question is valid)

Briefly explain deep learning. What's advantages of using DL compared to traditional computer vision methods? + 2 Applications of DL for CV

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?

We inntrouitn a passthrough! Big Deep Leanrig mdoels are hard to train that's whx we using a pass through

* 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?

* Explain perspective distortion

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?