

COMPUTER VISION AND IMAGE PROCESSING

IPCV

Luca Marini

Contents

1	Image Formation and Acquisition	1
1.1	Stereo vision	1
1.2	Vanishing points	5
1.2.1	Find the vanishing point in projective space	7
1.3	Camera parameters	11
1.3.1	First let's talk about Noise	11
1.3.2	SNR (Signal to noise ratio)	16
1.3.3	Dynamic Range (DR)	16
1.3.4	Experiment to measure how much noise there is in a camera .	17
2	Camera Calibration	18
2.1	Intrinsic parameters	18
2.2	Whole Camera Calibration and Zhang's	22
2.2.1	Homographies in Zhang's and their Estimation	24
2.3	Estimation of INTinsics in Zhang's	32
2.4	Estimation of EXTrinsics in Zhang's	34
2.5	Lens distortion	36
2.5.1	Estimation of radial lens distortion coefficients (k1, k2) with Zhang's	39
2.6	Refinement by non-linear optimization - Zhang's	42
2.7	Point Cloud having Depth Camera	43
2.7.1	What if you have a depth camera, you have a z, and you want to go back into the 3D world (CRF) knowing the intrinsics? .	43
2.7.2	Now if you want to find the corresponding pixels of another view of the same scene taken by the same depth camera? (You move the camera, camera motion being known)	46
2.7.3	case with 2 different cameras	47
2.8	Image Warping	47
2.8.1	Forward Mapping	49
2.8.2	Backward Mapping	49
3	Intensity Transformations	54
3.1	Histogram Equalization	54
4	Spatial Filtering	61
4.1	Convolution	61
4.2	Correlation	70
4.3	Bilateral Filter	72

4.4	Non-local Means Filter	76
5	Image Segmentation	79
5.1	Colour-based Segmentation	79
6	Binary Morphology	87
7	Blob Analysis	88
7.1	Connected Components	90
7.2	Labeling problem (two scan algorithm)	93
7.2.1	Implementation	95
8	Edge Detection	98
8.1	LoG Edge detector	98
8.2	Canny's Edge detector	106
8.2.1	Hysteresis Thresholding	108
9	Local Invariant Features	112
9.1	3 Main steps of local invariant features pipeline matching	112
9.2	Evaluate the performance of a feature MATCHING pipeline	114
9.3	Harris Corner Detector	116
9.4	DOG key points	122
9.4.1	In the local invariant feature pipeline how do we achieve rotation and scale invariance? Rotation and scale invariance in SIFT.	126
9.5	SIFT Descriptor	127
9.5.1	Validating matches in SIFT	128
10	Object Detection	129
10.1	Template matching	129
10.1.1	SSD	130
10.1.2	SAD	132
10.1.3	NCC	133
10.1.4	ZNCC	136
10.1.5	Fast Template Matching	138
10.2	Shape-based matching algorithm	140
10.3	GHT	148
10.3.1	GHT + SIFT	157

Chapter 1

Image Formation and Acquisition

1.1 Stereo vision

Stereo vision is a technology that allows to infer the **depth** of a point in the space using 2 cameras placed with a particular geometry, which is the stereo standard geometry.

In this geometry the 2 cameras are displayed like human eyes, and have:

- Parallel (x,y,z) axes
- Same focal lenght
- Coplanar image planes

The 2 CRF (camera reference frames) of the 2 cameras are horizontally displayed one to another by a quantity which is b (the baseline).

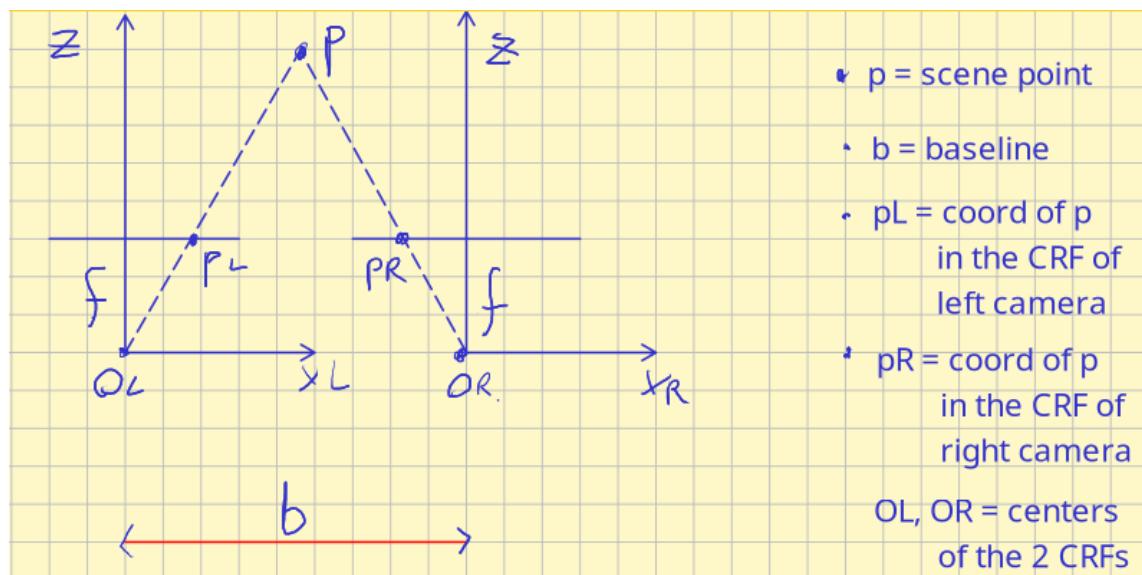


Figure 1.1

So, we want to place the 2 cameras in a way such that this relation is satisfied:

$$|| \quad p_L - p_R = \begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix} - \begin{bmatrix} x_R \\ y_R \\ z_R \end{bmatrix} = \begin{bmatrix} b \\ 0 \\ 0 \end{bmatrix}$$

$$x_L - x_R = b$$

$$y_L = y_R \quad \text{so, the transformation between the 2 ref frames is just an horiz translation}$$

$$z_L = z_R$$

Figure 1.2

- apply Perspective Projection Equations to get image coords:

$$v = y \cdot \frac{f}{z} \quad \left(\begin{array}{l} y_R = y_L = y \\ z_R = z_L = z \end{array} \right) \Rightarrow v_L = v_R = y \cdot \frac{f}{z}$$

So, if the 2 cameras are placed according to standard stereo geometry, them a given 3D point p will be seen at the same height in the 2 images.
 \Rightarrow It will have the same vertical coords

Figure 1.3

$u_L \neq u_R$ they change depending on depth.

The change in the left and right horiz coords is due to depth.
And we can use this change to infer depth in a stereo setup.

$$u_L = x_L \cdot \frac{f}{z}$$

$$u_R = x_R \cdot \frac{f}{z}$$

) → horiz coord of in the 2 images

$$u_L - u_R = (x_L - x_R) \cdot \frac{f}{z} = b \cdot \frac{f}{z} = d$$

\downarrow \downarrow baseline

Disparity = diff between horiz coords in the left and right image.



relationship between disparity and depth:

$$z = b \cdot \frac{f}{d}$$

Figure 1.4

- So, in standard stereo vision, is easy to compute the depth of point p by knowing the correspondences of that point in the 2 image planes. => By knowing the correspondences (u_L, v_L) and (u_R, v_R) .
- And we can look for correspondences in 1D space because we know that $v_L = v_R$.
- So, if I have the point p_L in the left image plane, then the point in the right image plane will be at the same height of the point p_L .
- So, I don't search throughout the whole image, but just on the same height.

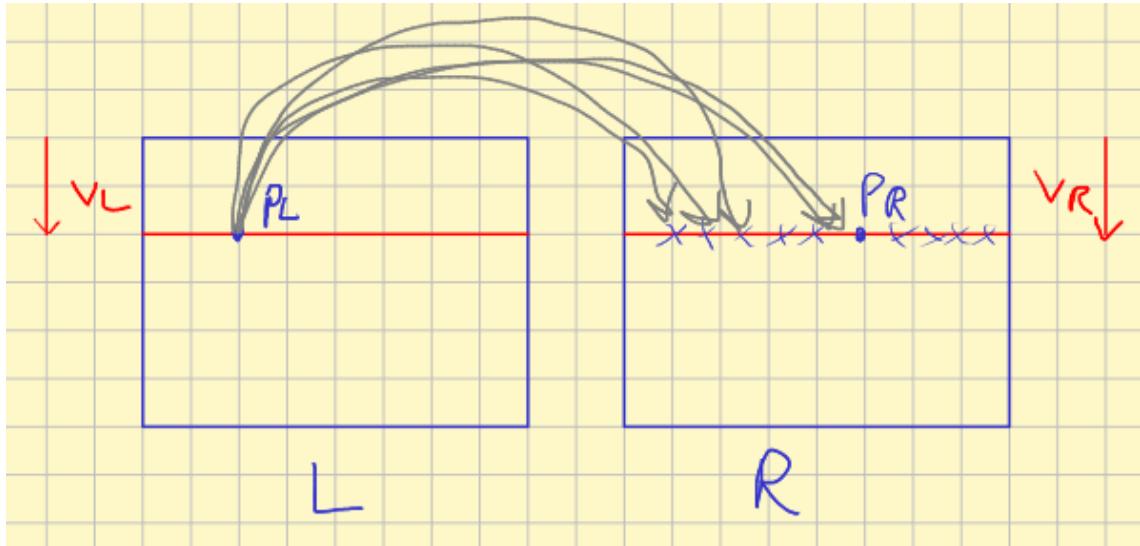


Figure 1.5

- Possible problem: maybe a point in the left image is not in the right image, due to occlusions, light changes, ecc.
- How to find correspondences? This is the key problem in Stereo Vision
- Take a neighbourhood (a window around) p_L and look for windows in the right image (in a 1D space) that f.e. maximizes a similarity function (f.e. using intensities of window pixels):

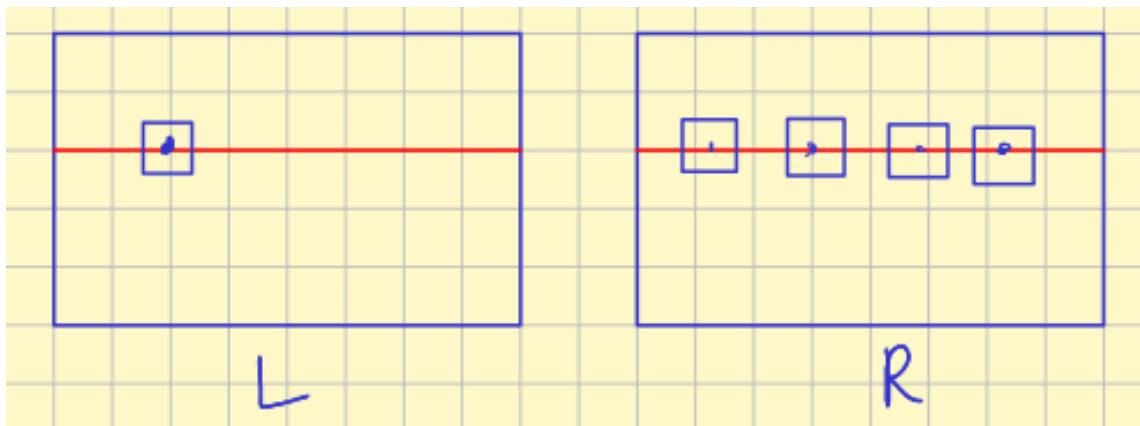


Figure 1.6

- What if the 2 cameras are arbitrarily arranged?

Epipolar Geometry:

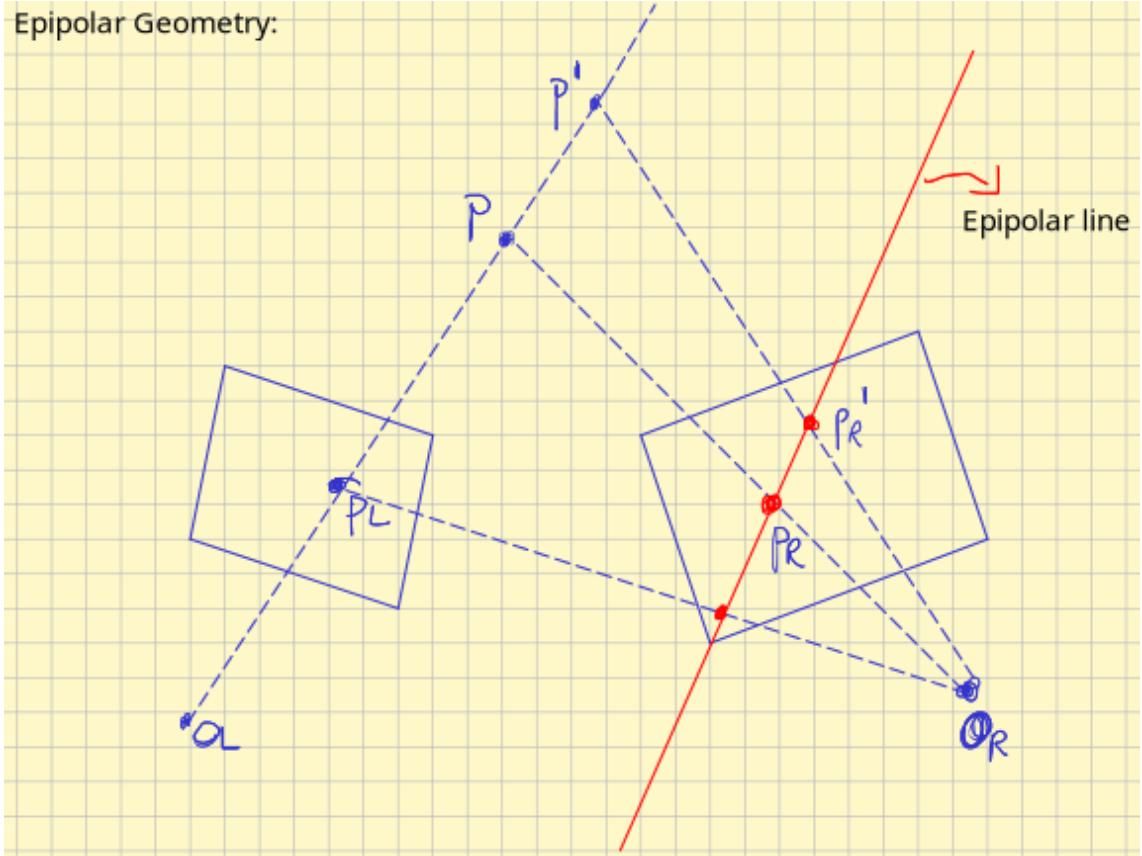


Figure 1.7

- I still search the correspondent point in a 1D space, but in an oblique line (less efficient)
- we know that, given p_L , its corresponding 3D point has to be on the dashed line
- so, the corresponding point p_R has to be on the projection of the dashed line into the right image (the Epipolar line)
- To compute the Epipolar line you need to know the focal lengths and the roto-translations between the 2 CRFs

1.2 Vanishing points

- The **vanishing point of a 3D line** is the image of the point at infinity of the line.

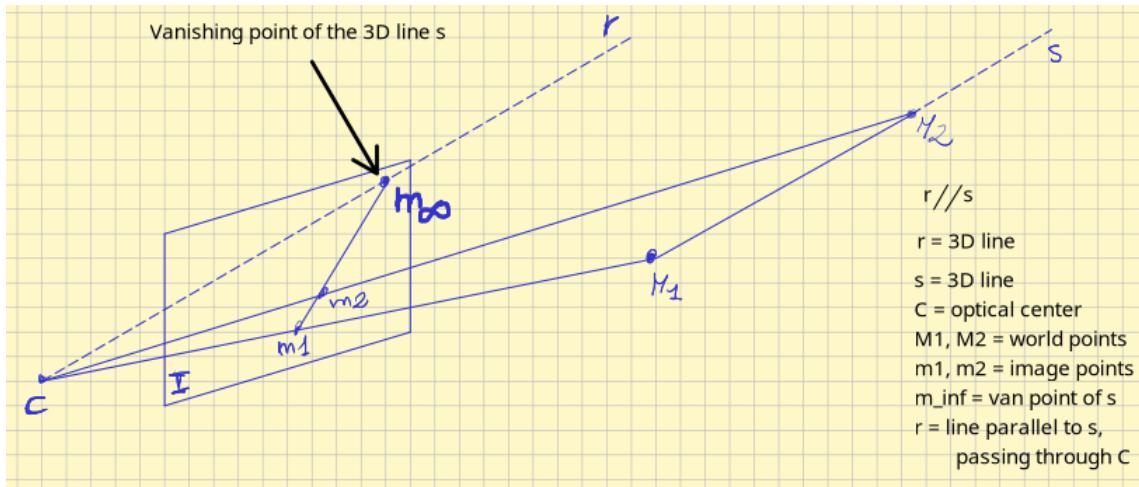


Figure 1.8

- Project world points into the image with persp proj ($M_1 \rightarrow m_1$) => by drawing line from M_1 to optical center C and taking the intersection with the image plane
- To find graphically the vanishing point of the 3D line s (which is m_{infinite}), I need to find the intersection between:
 - the line parallel to the 3D line and passing through the optical center (this line is r)
 - and the image plane
- All parallel 3D lines meet at the same vanishing point in the image.
- If the 3D line is parallel to the image plane => it will have no vanishing point (its parallel passing through the optical center will not intersect the image plane)
- Parametric equation of the 3D line:

$$M = M_0 + \lambda \cdot D = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + \lambda \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

M = point along 3D line
 M_0 = given point along the line
 lambda = parameter
 D = direction cosine vector of the line

Figure 1.9

- To find the vanishing point I need to take the point at infinity of this line and find its image.
 - but if I take the point at infinity, then its coords are inf, inf, inf => I should use Perspective Projection.

1.2.1 Find the vanishing point in projective space

- how can you find the vanishing point of a 3D line by modeling perspective projection through the PPM? (So, in homogeneous coords)
- Parametric equation of a 3D line:

$$M = M_0 + \lambda \cdot D = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + \lambda \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} x_0 + \lambda \cdot a \\ y_0 + \lambda \cdot b \\ z_0 + \lambda \cdot c \end{bmatrix}$$

M = any point along 3D line
 M₀ = given point along the line
 lambda = parameter
 D = direction cosine vector of the line. It is a vector // to the 3D line

↓
Euclidean representation of a point on the 3D line

Figure 1.10

- Projective representation of M (append a 1, and multiply by a constant k):

$$\tilde{M} = \begin{bmatrix} M \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 + \lambda \cdot a \\ y_0 + \lambda \cdot b \\ z_0 + \lambda \cdot c \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x_0 + \lambda \cdot a}{\lambda} \\ \frac{y_0 + \lambda \cdot b}{\lambda} \\ \frac{z_0 + \lambda \cdot c}{\lambda} \\ \frac{1}{\lambda} \end{bmatrix}$$

↓
It's convenient to multiply by k=1/lambda

Figure 1.11

- This generic point becomes the point at infinity of that line when lambda -> inf:

$$\tilde{M}_\infty = \lim_{\lambda \rightarrow \infty} \tilde{M} = \lim_{\lambda \rightarrow \infty} \begin{bmatrix} \frac{x_0}{\lambda} \cdot a \\ \frac{y_0}{\lambda} \cdot b \\ \frac{z_0}{\lambda} \cdot c \\ \frac{1}{\lambda} \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \\ \emptyset \end{bmatrix}$$

point at infinity of a line as a projective point, because points at infinity don't admit a representation in the Euclidean space.

Figure 1.12

- So, the projective coords of the point at infinity of a 3D line are obtained by taking an Euclidean vector parallel to the line (f.e. the direction cosine vector) and appending a 0 as fourth coord.

Perspective Projection:

$$m = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \rightarrow \text{Euclidean image point} \quad M = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{array}{l} \text{Euclidean} \\ \text{world} \\ \text{point} \end{array}$$

- their Perspective representation (in Projective space) are (we append a 1):

$$\tilde{m} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \quad \tilde{M} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \tilde{M} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\hookrightarrow \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f \cdot x \\ f \cdot y \\ f \cdot z \\ 1 \end{bmatrix} = \begin{bmatrix} fx \\ fy \\ fz \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$K \cdot \tilde{m} = \tilde{P} \cdot \tilde{M}$$

Figure 1.13

- To compute the vanishing point of a line (in Projective space) (so to get the image of the point at infinity in PS of the line):
 - I just multiply the PPM to the point at infinity of the line in PS:

$$\begin{bmatrix} ? \\ \cdot \\ \vdots \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ 0 \end{bmatrix} \xrightarrow{\sim} M_\infty$$

$\downarrow \sim P$

$$\sim \begin{bmatrix} f \cdot a \\ f \cdot b \\ c \end{bmatrix} = \tilde{m}_\infty$$

$\sim M_\infty$

$M_{\text{inf}} = \text{point at inf of the line in projective space}$

$\tilde{m}_{\text{inf}} = \text{image of } M_{\text{inf}}.$
 $\Rightarrow \text{projective repres of the vanishing point}$

Figure 1.14

- Now, I go back to the corresponding Euclidean vanishing point:

– I divide by the last coord and delete the last coord:

$$m_\infty = \begin{bmatrix} f \cdot \frac{a}{c} \\ f \cdot \frac{b}{c} \end{bmatrix}$$

$m_{\text{inf}} = \text{van point of my line, whose direction is given by the vector } \begin{bmatrix} a \\ b \\ c \end{bmatrix}$

Figure 1.15

1.3 Camera parameters

1.3.1 First let's talk about Noise

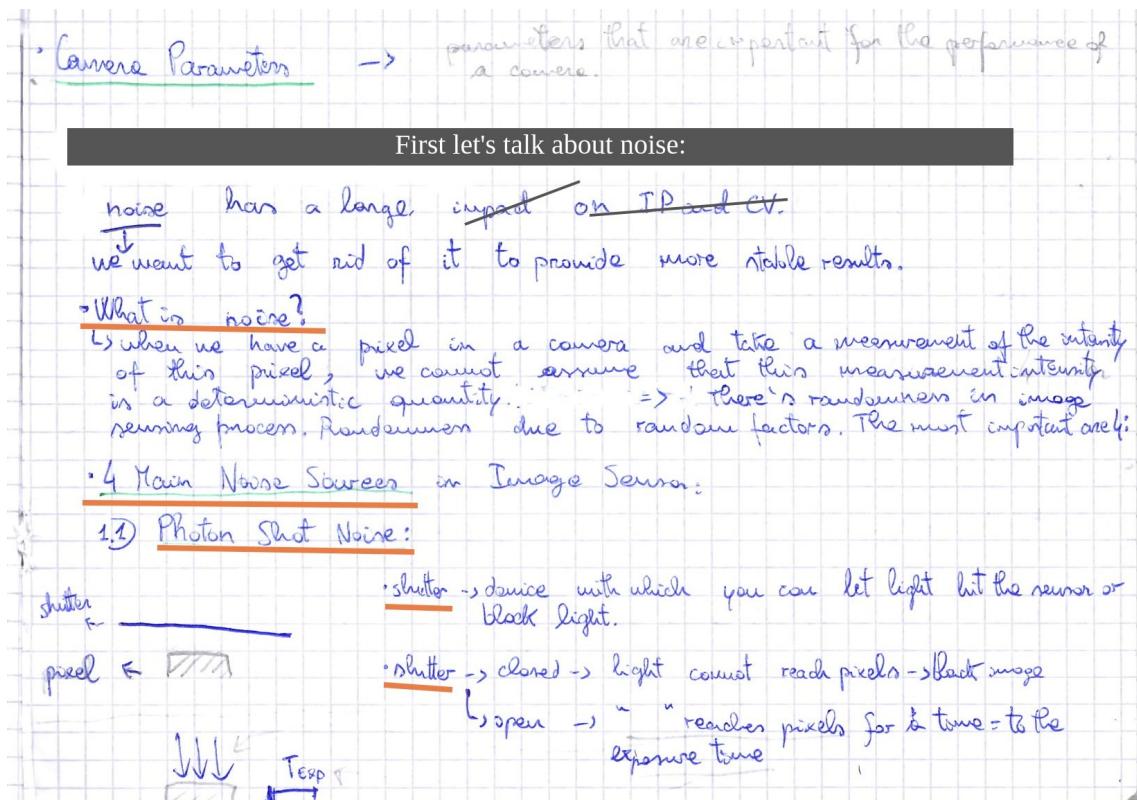


Figure 1.16

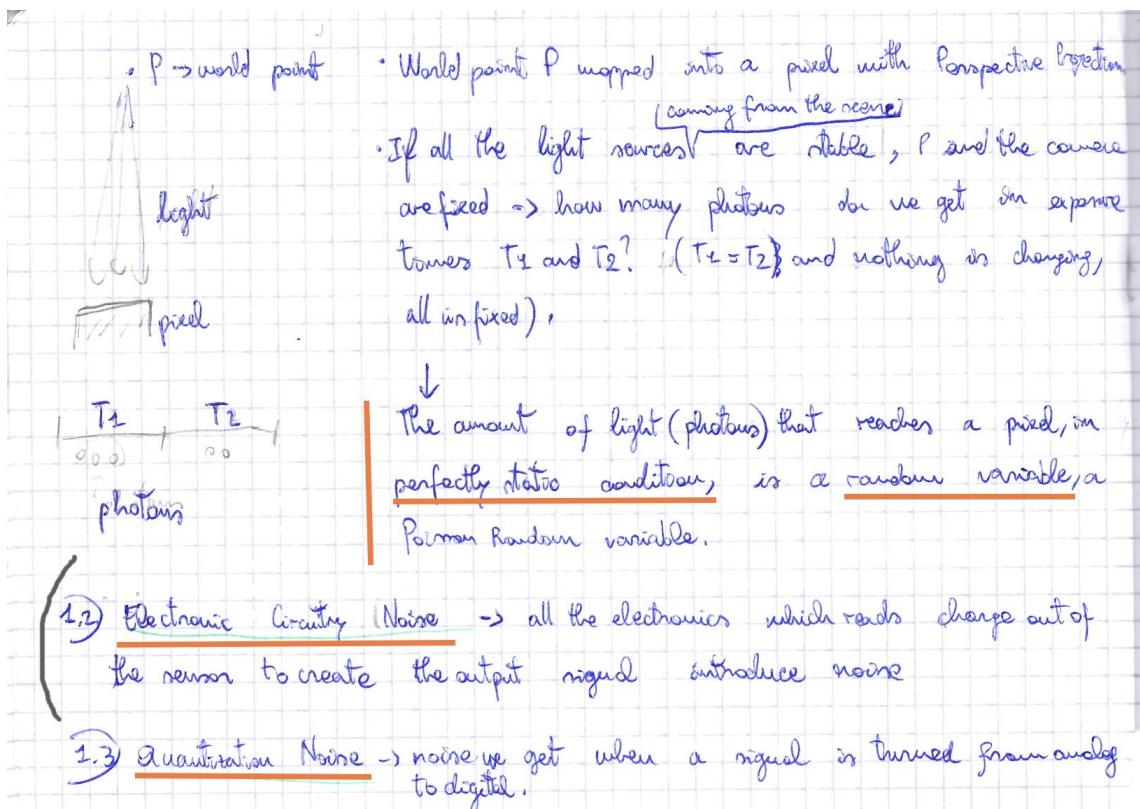


Figure 1.17

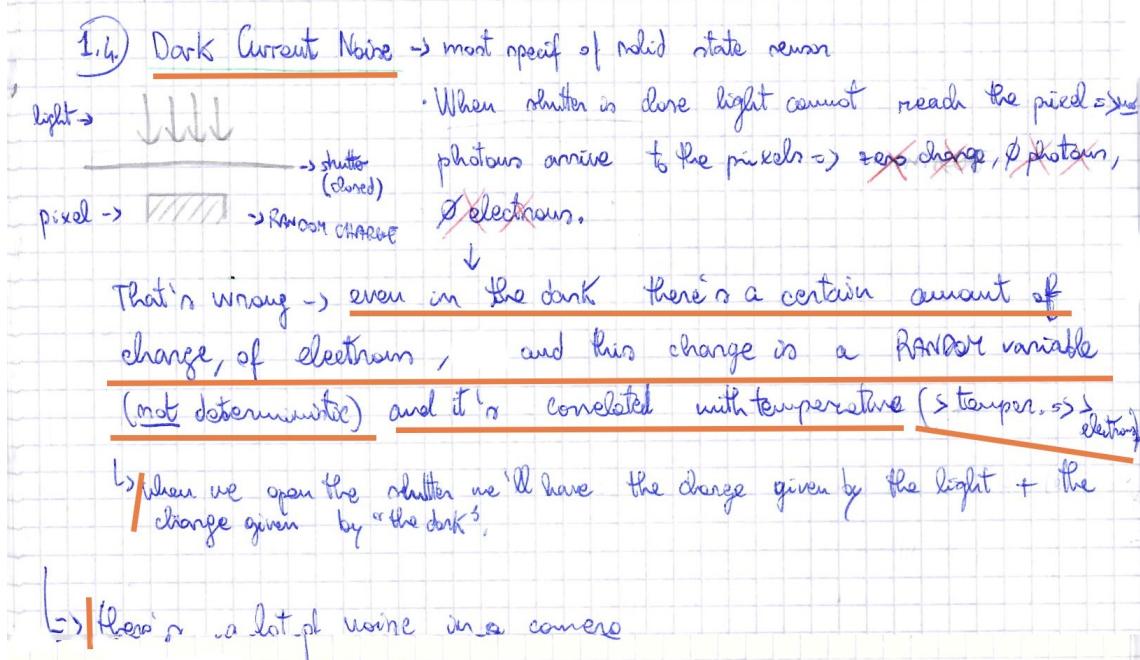


Figure 1.18

We would like the noise to be small compared with the interested signal.

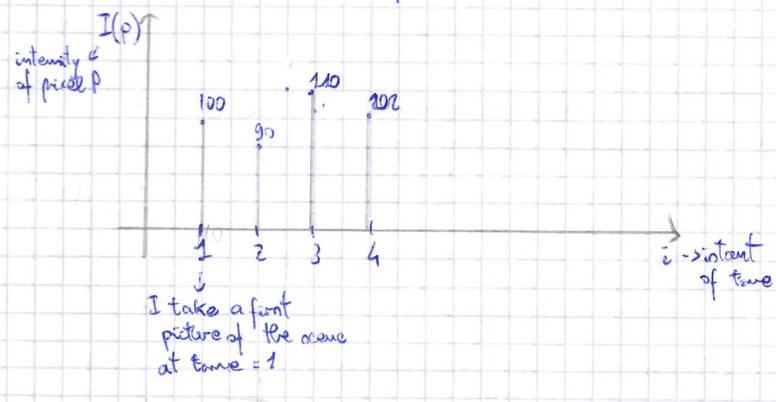
p → pixel, P → world point

image

p

p'

perspective projection



No light sources are changing in the scene, the camera and the object are fixed, same pixel, same object. \rightarrow But the intensity at time 2 is

95 because of noise. Although the static conditions, you don't get the same intensity, because it's a random process, it's not deterministic.

\hookrightarrow If we observe at the very same pixel in different times, then we'll measure different intensities, different amounts of light.

Figure 1.19

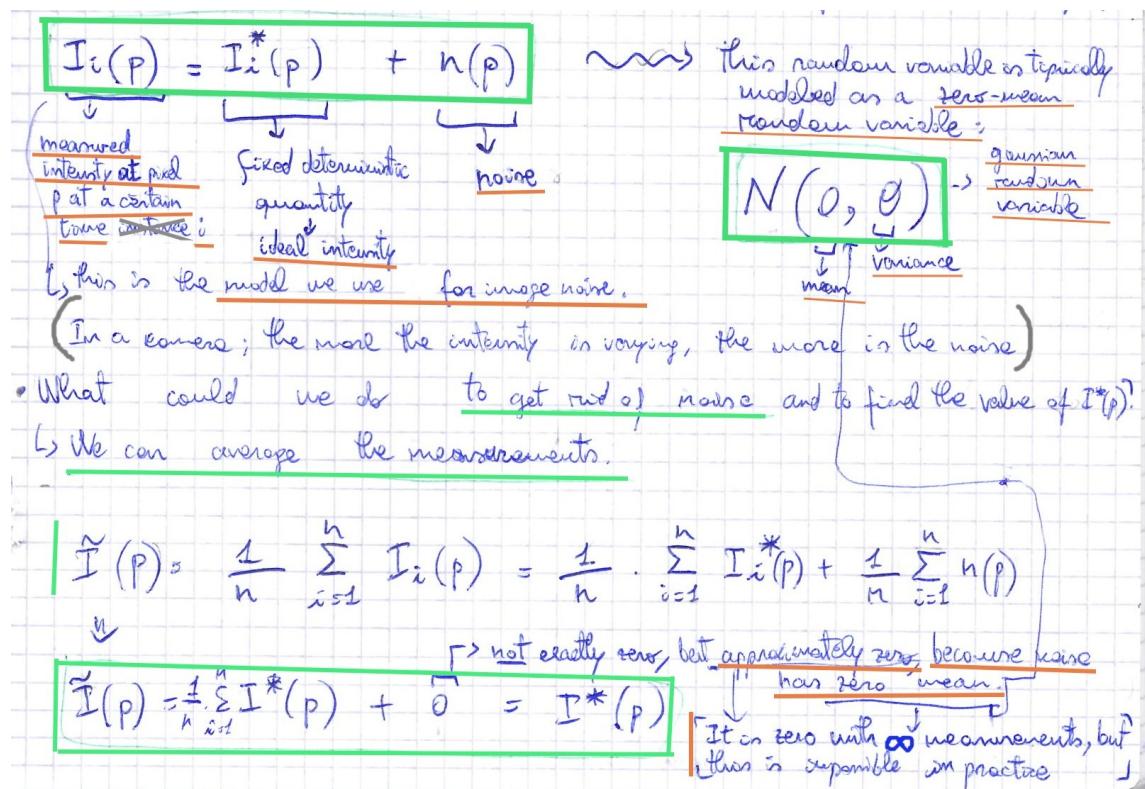


Figure 1.20

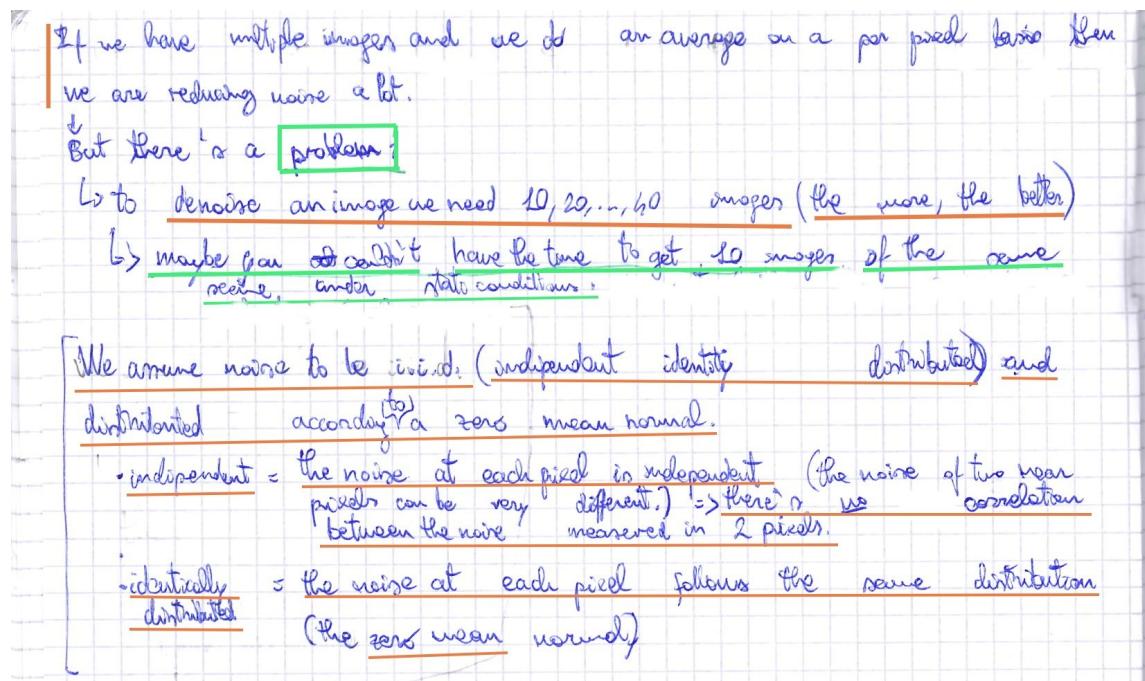
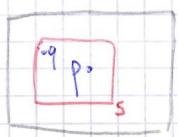


Figure 1.21

\Rightarrow What if I have only one image?

[smooth = averaging, get rid of noise]

(supporting neighborhood)



I take a neighborhood s around the pixel p . \Rightarrow Rather than taking the mean through time, I take it through space.

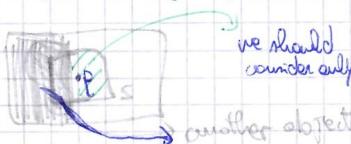
$$\hat{I}(p) = \frac{1}{s^*} \sum_{q \in s} I(q)$$

, s^* = number of pixels in s

\Leftrightarrow by doing so I'm doing the assumption that I^* is the same for all the pixels in s $\xrightarrow{\text{unknown ideal intensity}}$
they belong to the same object. \rightarrow

91	...
92	100
100	100
98	5

\Rightarrow But if the window s covers two different objects of the scene, then it doesn't make sense, because if another object is dark, then its intensity will be lower than the other object \Rightarrow it doesn't have sense to do a mean between these two:



we should consider only this pixels \rightarrow BILATERAL FILTER does it
(to smooth)

Figure 1.22

\Rightarrow the assumption is that close pixels to the given one do belong to the same object.

\Rightarrow the window s has to be small \Rightarrow we should take averages of small numbers, (there's a lot of research in denoising) \rightarrow now we have the denoising algorithms in the firmware of the cameras.

\Rightarrow To denoise an image: smoothing, take averages.

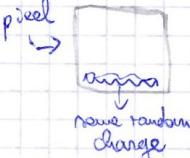
Figure 1.23

1.3.2 SNR (Signal to noise ratio)

- Camera Parameters:
 - To answer how much noise there's in a certain camera there's a parameter that's called ① Signal to noise ratio (SNR): \rightarrow provided by the provider
 - \rightarrow SNR \Rightarrow better quality \Rightarrow less noise
 - SNR is measured in dB (decibel) or in bits
- $$\text{SNR}_{\text{dB}} = 20 \cdot \log_{10}(\text{SNR})$$
- $$\text{SNR}_{\text{bit}} = \log_2(\text{SNR})$$

Figure 1.24

1.3.3 Dynamic Range (DR)

- ② Dynamic Range (DR):
- 

In order to be able to reuse a meaningful signal, the amount of charge stored in each pixel must be higher than a minimum quantity (which is called E_{\min}), this due to the presence of random dark current.

There's also a maximum threshold (E_{\max}), because each pixel can store a maximum amount of charge (called also saturation irradiation).

\hookrightarrow the dynamic range of a sensor is the ratio between the minimum detectable irradiation and the saturation irradiation:

$$DR = \frac{E_{\max}}{E_{\min}}$$

$\rightarrow DR \Rightarrow$ better quality

 - \rightarrow It's better to have an higher DR, \rightarrow the higher will be the ability of the sensor to simultaneously capture in one image both the dark and bright structures of the scene.

If we want to see the dark regions we need to have a relatively long exposure time, but if the exposure time is too long the bright regions will saturate,

Figure 1.25

and so you won't be able to distinguish the details in the bright areas.

DR is about the ability of the sensor to capture well both dark as well as bright areas of the scene.

(Also the DR is typically measured in dB).

- HDR → high dynamic range imaging → take multiple images at different exposures, and then these images are mixed together in an intelligent way (with algorithm) so that the overall image shows with quality both the dark as well as the bright areas of the scene.

Figure 1.26

1.3.4 Experiment to measure how much noise there is in a camera

Experiment to measure how much noise there is in a camera:

- take pixels → then take many measurements along time of those pixel intensities
- compute variances (or the mean of the variances if you want to average out across the whole image)
- ex of measuring noise by considering many measurements along time of the same pixel p:

$$\begin{aligned}
 I_1(p) &= 100 & \mu = \frac{1}{n} \sum_{i=0}^n I_i(p) &= \frac{100+98+102+100}{4} = 100 \\
 I_2(p) &= 98 \\
 I_3(p) &= 102 \\
 I_4(p) &= 100 \\
 &\vdots \\
 I_i(p) &= \text{intensity of pixel } p \quad \sigma^2 = \frac{1}{n} \sum_{i=0}^n (I_i(p) - \mu)^2 = \\
 &\quad \quad \quad = \frac{(100-100)^2 + (98-100)^2 + (102-100)^2 + (100-100)^2}{4} =
 \end{aligned}$$

$I_i(p)$ = intensity of pixel p
at time i
 μ = mean of intensity
measurements
 σ^2 = variance (=error)

- the $>$ variance, then the worst the camera is.

Figure 1.27

Chapter 2

Camera Calibration

2.1 Intrinsic parameters

$$\tilde{m} = \tilde{P} \cdot \tilde{H}$$

world point

canonical PPM

continuous image coords

$$\rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = [I | 0]$$

Figure 2.1

\tilde{m} are continuous coordinates into the image plane,
we can't observe them.

Continuous
coordinates in the image plane don't exist, because the image are
digital.

Figure 2.2

① Image Digitization (also called Pixelization) .
↳ It's about moving from continuous coordinates to pixels

Figure 2.3

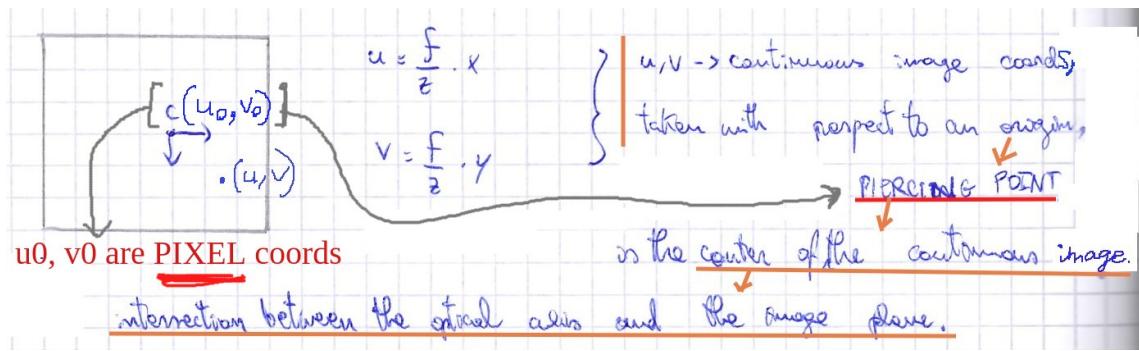
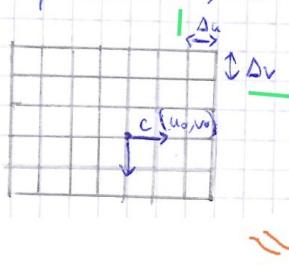


Figure 2.4

The continuous image plane doesn't exist, because in reality we have an array of pixels spaced by Δu horizontally and Δv vertically. (typically $\Delta u = \Delta v$, but we consider them separately for better generality)



divide by the horizontal and vertical quantization steps respectively:

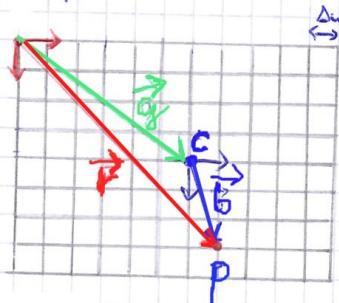
$$u = \frac{f}{z} \cdot x \Rightarrow u = \frac{1}{\Delta u} \cdot \frac{f}{z} \cdot x$$

$$v = \frac{f}{z} \cdot y \Rightarrow v = \frac{1}{\Delta v} \cdot \frac{f}{z} \cdot y$$

pixels coords

Figure 2.5

* Other issue: when we measure pixels we need to translate the origin at the top left corner, because pixels coordinates are always positive, indeed they represent indices in matrices.



$$u = \frac{1}{\Delta u} \cdot \frac{f}{z} \cdot x = Ku \cdot \frac{f}{z} \cdot x \quad \boxed{b}$$

$$v = \frac{1}{\Delta v} \cdot \frac{f}{z} \cdot y = Kv \cdot \frac{f}{z} \cdot y \quad \boxed{b}$$

pixel p coords wrt the
center of the image

I want coords of p wrt the top left corner => I need to find the 2 elems of the red vector:

$$\vec{b} = \left(Ku \cdot \frac{f}{z} \cdot x, Kv \cdot \frac{f}{z} \cdot y \right) \quad \vec{g} = (u_0, v_0) \quad \text{pixel coords of the center of the image}$$

$$\vec{r} = \vec{g} + \vec{b} = \left(Ku \cdot \frac{f}{z} \cdot x + u_0, Kv \cdot \frac{f}{z} \cdot y + v_0 \right)$$

Figure 2.6

↳ $u = Ku \cdot \frac{f}{z} \cdot x + u_0$

↳ $v = Kv \cdot \frac{f}{z} \cdot y + v_0$

now we have 2 equations that map from world coordinates (x, y, z) to pixels coordinates (u, v) .

This ⁽¹⁾ Euclidean equations from world coordinates in camera reference system (CRF) (so we're still in CRF, not in WRF).

But we're not happy because these 2 equations are non-linear (I multiply by $\frac{f}{z}$).

↳ we redefine the PPM;

Figure 2.7

$$\tilde{P} = \begin{bmatrix} f \cdot k_u & 0 & u_0 & 0 \\ 0 & f \cdot k_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

PPM that takes into account PIXELIZATION

to get an image point \tilde{m} of $(x, y, z, 1)$:

$$\begin{bmatrix} x \cdot f \cdot k_u + u_0 \cdot z \\ y \cdot f \cdot k_v + v_0 \cdot z \\ z \\ 1 \end{bmatrix}_{3 \times 1} = \begin{bmatrix} f \cdot k_u & 0 & u_0 & 0 \\ 0 & f \cdot k_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}_{3 \times 4} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}_{4 \times 1}$$

\Rightarrow to get m from \tilde{m} (perspective space), we divide all coords by the last coord. and remove the last coord:

$$m = \begin{bmatrix} x \cdot \frac{f \cdot k_u + u_0}{z} \\ y \cdot \frac{f \cdot k_v + v_0}{z} \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix}$$

This is the same result we got with the 2 main cases

Figure 2.8

b) now we do the pixelization with linear equations (by multiplying a matrix and a vector).

$$\tilde{P} = \begin{bmatrix} f \cdot k_u & 0 & u_0 & 0 \\ 0 & f \cdot k_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}_{(3 \times 4)} = \begin{bmatrix} f \cdot k_u & 0 & u_0 \\ 0 & f \cdot k_v & v_0 \\ 0 & 0 & 1 \end{bmatrix}_{(3 \times 3)} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}_{(3 \times 4)} = A \cdot \begin{bmatrix} I | 0 \end{bmatrix}_{(3 \times 4)}$$

Intrinsic Parameter Matrix = A

\Rightarrow the PPM matrix can be factorized in these two matrices

A contains:

- ① intrinsic parameter matrix
- ② size of the pixels
- ③ position of the image center.

$\cdot \begin{bmatrix} I | 0 \end{bmatrix}_{(3 \times 4)}$ = the canonical (or standard) PPM

Figure 2.9

- ↳ when we have to project a point to get a pixel :
 ↳ first we do a canonical projection (\Rightarrow without parameters) \rightarrow with $[I|0]$
 ↳ we get a continuous point with projective space coords in the
 image plane (indeed the canonical PPM transform $\tilde{m} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \tilde{m} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$)
 . the canonical projection is independent on the specific device we're using.
- ② Device Dependent Digitization (Add the camera parameters):
 ↳ we multiply this $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$ vector (obtained by the canonical projection) by A , which
 is the intrinsic parameter matrix, and get the actual pixel
 coordinates.
- ↳ we first project in a device independent way, and then transform
 this continuous point into pixel coordinates based on the actual
 camera parameters (focal length, pixel size and image center) expressed
 by the A matrix.
 ↳ A is device specific
 * the canonical PPM is device independent.

Figure 2.10

2.2 Whole Camera Calibration and Zhang's

- Goal = find (estimate) all the parameters of the image formation model of my camera, which are:
 - $A \rightarrow$ intrinsic parameters
 - extrinsic parameters \rightarrow rigid motion between the WRF and the CRF:
 - * $R \rightarrow$ rotation matrix
 - * $T \rightarrow$ translation vector
 - lens distortion parameters \rightarrow because lenses are not perfectly thin \Rightarrow we have some distortion.
- How to estimate these parameters:
 - we have the model of image formation process:

$$\tilde{m} = \tilde{P}(A, R, T) \cdot \tilde{M}$$

image coord. = $\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$

world coord. = $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$

Figure 2.11

we have to solve $\tilde{m} = \tilde{P}(A, R, T) \cdot \tilde{M}$ => to estimate camera parameters A, R, T we need to know some world (M_{tilde}) and pixel coordinates (m_{tilde}).
 ↓
 UNKNOWN → known

- once we solve that, we've calibrated our camera.

Figure 2.12

- How to find pairs of known world and pixel coords ($M_{\text{tilde}}, m_{\text{tilde}}$)? => ZHANG'S METHOD
 - we use Calibration Targets = image in which we know exactly world coords of relevant points (=> Easy to find world points are called Control Points)
 - f.e. we can use a 2D chessboard pattern:

1) Acquire n images of a planar pattern with m internal corners

- n = 10 to 20 images
 - m = arbitrarily number
-
- we know exactly the size of black and white squares
 - we define an origin and a reference system attached to the pattern, => we can know exactly the world coords of the control points.
- ex: if a black square has size 10 mm:
=> this corner point has world coords: *in the Euclidean space* $\rightarrow \pi = \begin{bmatrix} 10 \\ 10 \\ 0 \end{bmatrix} \begin{matrix} (x) \\ (y) \\ (z) \end{matrix}$
- in the projective space* $\rightarrow \tilde{\pi} = \begin{bmatrix} 10 \\ 10 \\ 0 \\ 1 \end{bmatrix}$
- => we can do that for all corners in the chessboard
=> we'll know all the world coords of the control points
- And to know the PIXEL coords (m_tilde) of the control points we use the **HARRIS Corner Detector**

Figure 2.13

- Zhang will estimate
 - * intrinsic parameters -> they're the same for all images, because the camera is the same.
 - * extrinsic parameters -> as many as the num of calibration images, because we take the images with different positions => in each image we have a different reference system
 - * lens distortion parameters -> same for all images, because the lens is the same.
- => we have a list of correspondences between world and pixel coords for each image.
 - * these correspondences will be used to estimate the calibration parameters.

2.2.1 Homographies in Zhang's and their Estimation

- Now we can reduce our PPM into an Homography:
 - because the calibration image is planar => z coords are = 0.

$\Rightarrow P$ as a Homography \rightarrow from (3×4) PPM to a (3×3) Homography

$$K \tilde{m} \tilde{s} \tilde{P} \tilde{W} = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} & p_{1,4} \\ p_{2,1} & p_{2,2} & p_{2,3} & p_{2,4} \\ p_{3,1} & p_{3,2} & p_{3,3} & p_{3,4} \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = H \cdot W'$$

Known \downarrow

because $\underline{\text{z coord is zero for all control points, because we have a planar calibration target}}$

- We have as many equations as corner points \times images $\Rightarrow (m \text{ corners} \times n \text{ images})$
- now we consider all m equations related to a single image

Figure 2.14

- N.B.**
- If we have a planar scene (with $z = 0$), then we project that into an image by an Homography
 $\Rightarrow 3 \times 3$ and NOT 3×4
 - Any two images of a planar scene are related by an Homography

Figure 2.15

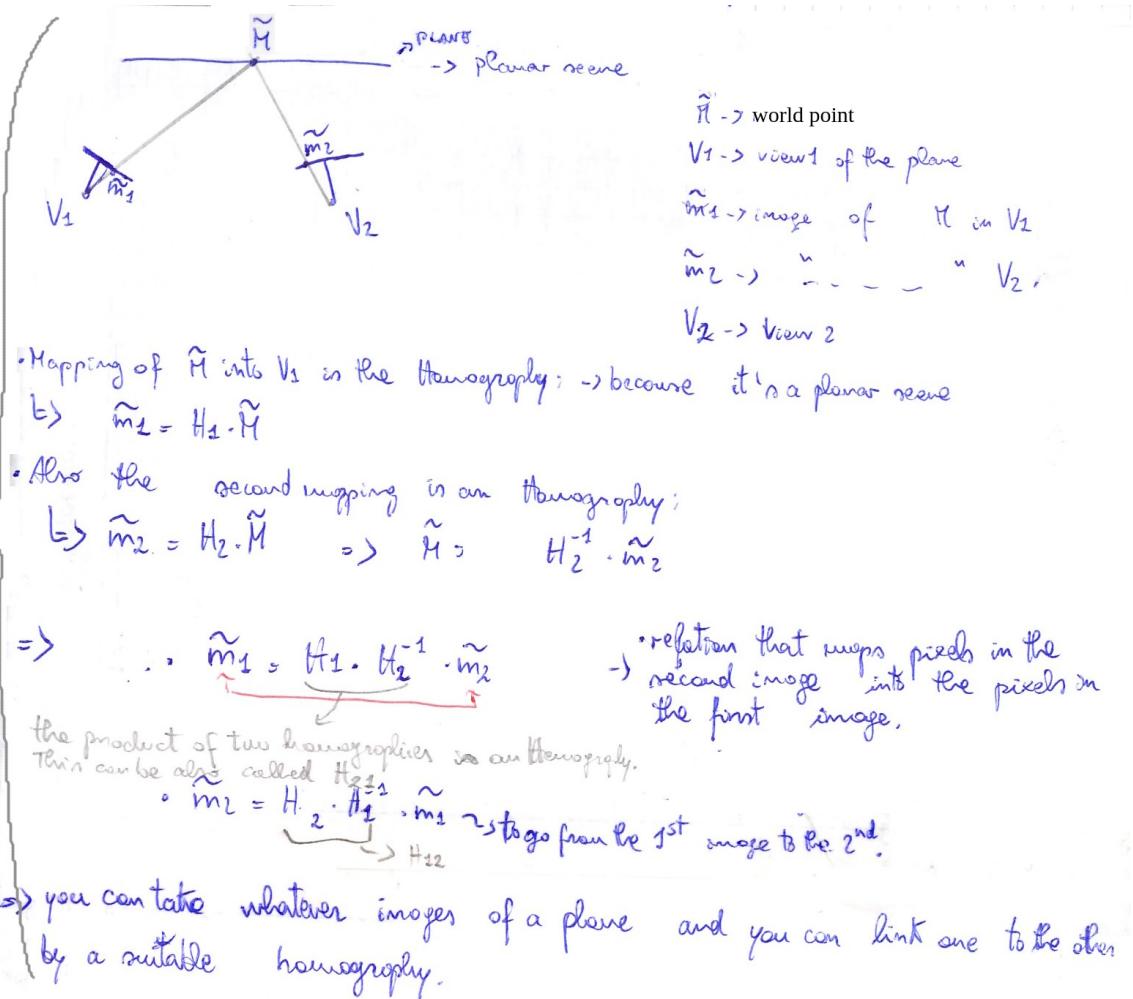


Figure 2.16

- How to estimate the Homographies in Zhang's: DLT:

How Homographies are estimated:

- if we find the homography, we will then be able to find A, R and T

we estimate them by using:

DLT (= Direct Linear Transform)

- $K\tilde{m} = H \cdot \tilde{w}'$ \rightarrow now we're focusing on a single image. And we can write as many as m (= num of corners) equations like this.

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \quad (3 \times 3)$$

I rewrite H as:

$$H = \begin{bmatrix} h_{11} \\ h_{21} \\ h_{31} \end{bmatrix}$$

$$- h1.T = [h_{11}, h_{12}, h_{13}]$$

Figure 2.17

projective space vectors

$$\underbrace{K \tilde{m}}_{(3 \times 1)} = \underbrace{H \tilde{w}^1}_{(3 \times 1)}$$

L, But let's suppose they're Euclidean Vectors

if 2 Euclidean vectors A, B are such that:
 $A = k * B$,
then they're parallel

their vector product is = 0
- 1st vector is \tilde{m}
- 2nd vector is $H \cdot \tilde{w}^1$

$$\underbrace{\tilde{m}}_{(3 \times 1)} \times \underbrace{H \cdot \tilde{w}^1}_{(3 \times 1)} = 0$$

vector product

$$H \cdot \tilde{w}^1 = \begin{bmatrix} h_1^T \\ h_2^T \\ h_3^T \end{bmatrix} \cdot \tilde{w}^1 = \begin{bmatrix} h_1^T \cdot \tilde{w}^1 \\ h_2^T \cdot \tilde{w}^1 \\ h_3^T \cdot \tilde{w}^1 \end{bmatrix} =$$

to compute these vector product
there's a simple method to
do that,
We compute 3 determinants:

Figure 2.18

We build a matrix in which we will compute 3 determinants:

$$\begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \mathbf{u} & \mathbf{v} & \mathbf{w} \\ h_1^T \tilde{\mathbf{w}}^T & h_2^T \tilde{\mathbf{w}}^T & h_3^T \tilde{\mathbf{w}}^T \end{bmatrix} \rightarrow \begin{array}{l} \text{unique vectors of the 3 axes} \\ \text{it's } m_{\text{tilde}} \Rightarrow \text{1st vector in the vector product} \\ \text{it's } H^* w_{\text{tilde}} \Rightarrow \text{2nd vector in the vector product} \end{array}$$

1st det 2nd det 3rd det

we compute the 3 determinants

$$\begin{bmatrix} v \cdot h_3^T \tilde{\mathbf{w}}^T - h_2^T \tilde{\mathbf{w}}^T \\ h_2^T \tilde{\mathbf{w}}^T - u \cdot h_3^T \tilde{\mathbf{w}}^T \\ u \cdot h_2^T \tilde{\mathbf{w}}^T - v \cdot h_1^T \tilde{\mathbf{w}}^T \end{bmatrix} \rightarrow \begin{array}{l} \text{1st det of the submatrix} \\ \rightarrow 2^{\text{nd}} \det \\ \rightarrow 3^{\text{rd}} \det \end{array} \begin{bmatrix} v & 1 \\ h_2^T \tilde{\mathbf{w}}^T & h_3^T \tilde{\mathbf{w}}^T \end{bmatrix} \quad \begin{bmatrix} u & 1 \\ h_2^T \tilde{\mathbf{w}}^T & h_3^T \tilde{\mathbf{w}}^T \end{bmatrix} \quad \begin{bmatrix} u & v \\ h_2^T \tilde{\mathbf{w}}^T & h_2^T \tilde{\mathbf{w}}^T \end{bmatrix}$$

$$K \tilde{\mathbf{w}} = H \tilde{\mathbf{w}} \Rightarrow m \times H \tilde{\mathbf{w}} = 0 \Rightarrow \begin{bmatrix} v \cdot h_3^T \tilde{\mathbf{w}}^T - h_2^T \tilde{\mathbf{w}}^T \\ h_2^T \tilde{\mathbf{w}}^T - u \cdot h_3^T \tilde{\mathbf{w}}^T \\ u \cdot h_2^T \tilde{\mathbf{w}}^T - v \cdot h_1^T \tilde{\mathbf{w}}^T \end{bmatrix} = 0, H = \begin{bmatrix} h_1^T \\ h_2^T \\ h_3^T \end{bmatrix}$$

Figure 2.19

- This is an homogeneous system of 3 equations where the unknowns are the $h \Rightarrow$ all elements on the homography.
- Now we can rewrite this (through an algebraic manipulation we don't go through) like this: (but you can check),

$$\begin{array}{c} \text{1st:} \\ \text{2nd:} \\ \text{3rd:} \end{array} \left[\begin{array}{ccc} 0^T & -\tilde{W}^T & V \cdot \tilde{W}^T \\ \tilde{W}^T & 0^T & -U \cdot \tilde{W}^T \\ -V \cdot \tilde{W}^T & U \cdot \tilde{W}^T & 0^T \end{array} \right] \cdot \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = A \cdot h = 0$$

b) In this way we highlight the "A" \downarrow unknown h

$A \rightarrow$ coefficient matrix

$$h_1 = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \end{bmatrix} \quad (3x1) \quad , \quad h_2 = \dots, \quad h_3 = \dots$$

(the 3rd eq.).

c) This is a system of 3 equations in 3 unknowns \Rightarrow PROBLEM

Another problem: the 3 equations are not linearly independent \Rightarrow there are only 2 useful equations (if you multiply the 1st eq. by $-u$, the 2nd by $-v$, and add them up, then you get

Figure 2.20

Problem: we have 2 equations in 9 unknowns ($9 = 3 \times 3$ elements of the homography).

=> Solution: there are m corners in each calibration image

=> we have $2 \times m$ equations (2 for each corner) in 9 unknowns

- And the unknowns become 8 if we divide all of them by one of them => 8 unknowns.

=> $2 \times m$ equations and 8 unknowns => > equations than unknowns

=> I can solve this in a Least Square Sense:

- we search for a solution h^* that minimizes A^*h (because we want $A^*h = 0$):

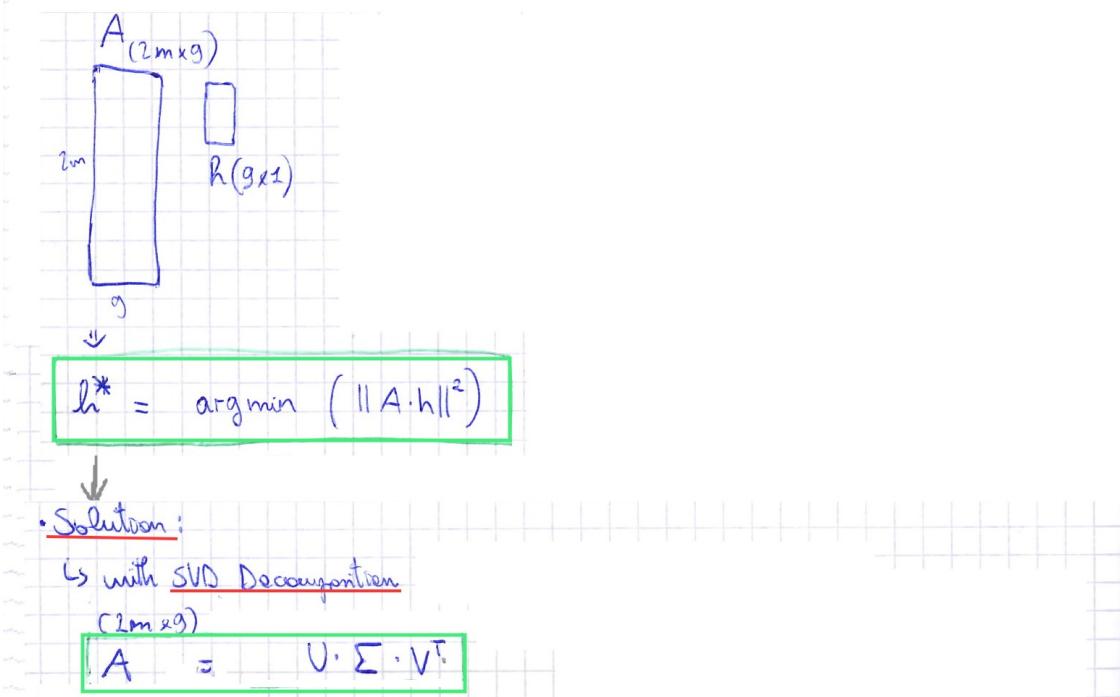


Figure 2.21

$\cdot A \rightarrow (2m \times 9) \rightarrow$ rectangular matrix \rightarrow coefficient matrix
 $\cdot U \rightarrow (2m \times 2m) \rightarrow$ orthogonal " T "
 $\cdot V^T \rightarrow (9 \times 9) \rightarrow$ orthogonal " T "
 $\cdot \Sigma \rightarrow (2m \times 9) \rightarrow$ diagonal " T " \rightarrow (diagonal even though it's rectangular \rightarrow $\begin{bmatrix} * & 0 & 0 \\ 0 & * & 0 \\ 0 & 0 & * \end{bmatrix}$)
 the non-zero values in the diagonal are called SINGULAR VALUES

↓ It turns out that $h^* = v_g$ the last column of V^T

↓ To solve $h^* = \arg \min (||A \cdot h||^2)$, you decompose the matrix A with the SVD decomposition: $A = U \cdot \Sigma \cdot V^T \rightarrow$ and the solution is the last column of V^T .

Figure 2.22

The homography could have been solved in 8 unknowns => we just need $m = 4$ corners => $4 \times 2 = 8$ equations in 8 unknowns.

BUT with 8 equations we satisfy exactly them, which means that we're fitting exactly the noise of our pixels, and we do NOT want this.

=> the Least Square Solution is more robust wrt noise => better.

Figure 2.23

- How can we say that an Homography is good?

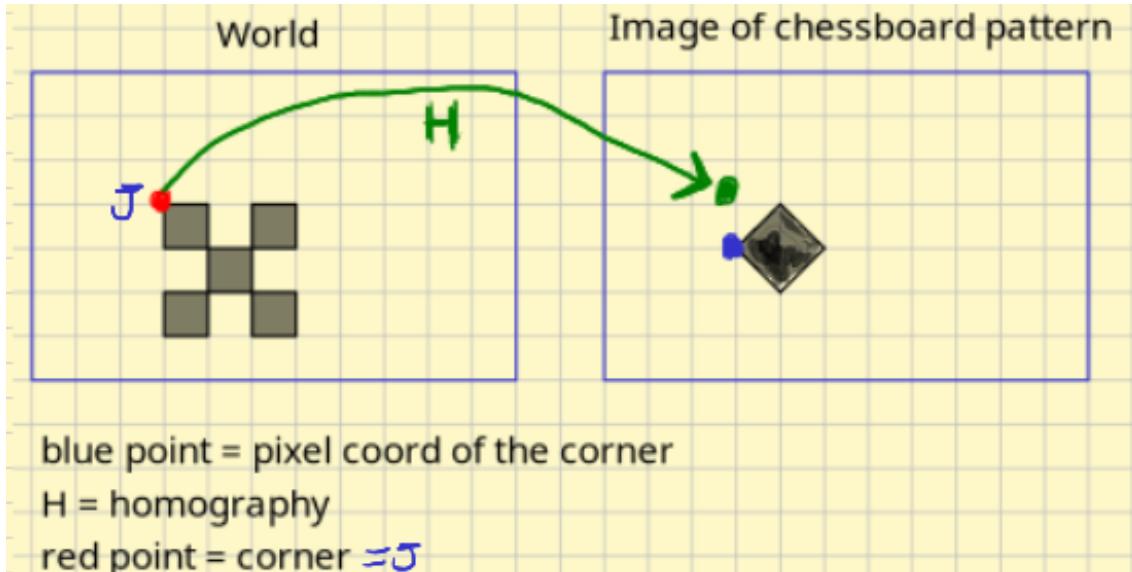


Figure 2.24

- the link between these 2 planes is the homography
 - * because the scene (world) is planar ($z = 0$).
- If I multiply J (red point) by the Homography I get the green pixel;
- and I want the green pixel to be as close as possible to the real corresponding pixel (the blue pixel).

$$\min_{H_i} \sum_j \| \tilde{m}_j - H_i \cdot \tilde{w}_j \|^2, \quad j=1, \dots, m$$

I would like this distance to be as small as possible

- I want $H_i \cdot w_{\text{tilde}}_j$ (the green point estimated through the homography) to be as close as possible to \tilde{m}_j (the blue point => the real corresponding pixel of the world point)

Figure 2.25

- distance = square of the norm of the difference between the real corresponding image point (blue) and the prediction given by the homography (green).

- $j=1, \dots, m \rightarrow$ I want the smallest distance for each corner point.

So:

- We find an initial estimation of the Homography by the DLT (\Rightarrow with SVD decomposition). And we have a starting point. (we minimize the algebraic error).

- Then, we use the NON-linear iterative optimization of $\min_{\mathbf{H}} \sum_j \|\tilde{\mathbf{m}}_j - \mathbf{H}_j \tilde{\mathbf{w}}'_j\|^2, j=1 \dots m$ with Levenberg-Marquardt algorithm to refine the Homography. (we minimize the distance, the geometric error).

Figure 2.26

- Now we have n ($n = \text{num of calibration images}$) Homographies:
 - all of the homographies have in common the Intrinsics.
 - \Rightarrow we can obtain the intrinsics through the homographies;
 - then, we can obtain the extrinsics through the intrinsics and the homographies.

2.3 Estimation of INTrinsics in Zhang's

$$\overset{\text{PPM}}{\mathbf{P}} = \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 & \mathbf{p}_4 \end{bmatrix}_{(3 \times 4)} = \lambda \cdot \mathbf{A} \cdot \begin{bmatrix} \mathbf{R} & \mathbf{T} \end{bmatrix} = \lambda \cdot \mathbf{A} \cdot \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 & \mathbf{T} \end{bmatrix}_{\text{columns of the rotation matrix}}$$

one way to express the PPM

- we have a planar setting (a planar calibration target, which is a 2D chessboard)

\Rightarrow the PPM becomes an Homography

$\Rightarrow p_3 = 0$

$$\mathbf{P} \underset{3 \text{ columns of the } (3 \times 3) \text{ homography}}{\underset{\downarrow}{=}} \mathbf{H} = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} = \begin{pmatrix} h_1 = \mathbf{p}_1 \\ h_2 = \mathbf{p}_2 \\ h_3 = \mathbf{p}_4 \end{pmatrix} = \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_4 \end{bmatrix} = \lambda \cdot \mathbf{A} \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{T} \end{bmatrix}$$

$$\begin{aligned} \cdot h_1 = \lambda A r_1 &\Rightarrow r_1 = \frac{1}{\lambda} \cdot A^{-1} \cdot h_1 \\ \cdot h_2 = \lambda A r_2 &\Rightarrow r_2 = \frac{1}{\lambda} \cdot A^{-1} \cdot h_2 \end{aligned}$$

- I want to find A (= the intrinsics)
- h_1 and h_2 are known, because we've estimated the homography

r1 and r2 are the columns of a rotation matrix (in a rotation matrix the columns and the rows are a basis of the space)

\Rightarrow r1 and r2 are perpendicular

\Rightarrow their SCALAR product is = 0 $\Rightarrow \mathbf{r}_1 \cdot \mathbf{r}_2 = 0$

Figure 2.27

1) $r_1^T \cdot r_2 = 0 \Rightarrow \left(\frac{1}{\lambda} \cdot A^{-T} \cdot h_1\right)^T \cdot \left(\frac{1}{\lambda} \cdot A^{-T} \cdot h_2\right) = 0 \quad [(A \cdot B)^T = B^T \cdot A^T]$

 $\Rightarrow \frac{1}{\lambda^2} h_1^T \cdot (A^{-T})^T \cdot A^{-T} \cdot h_2 = 0 \quad [\text{notation: } (A^{-T})^T = A^{-T}]$
 $\Rightarrow \frac{1}{\lambda^2} \cdot h_1^T \cdot A^{-T} \cdot A^{-T} \cdot h_2 = 0$

$\hookrightarrow \lambda$ is a NON-zero scalar \Rightarrow I don't consider it.

2) $r_2^T \cdot A^{-T} \cdot A^{-1} \cdot h_2 = 0$

2) other relationship between the columns of a rotation matrix:

$\|r_1\|^2 = 1 \rightarrow$ [the square norm of r_1 is = 1,
because r_1 is a basis of the space]

$\|r_2\|^2 = 1$

$\hookrightarrow \|r_1\|^2 = \|r_2\|^2 \Rightarrow r_1^T \cdot r_1 = r_2^T \cdot r_2 \Rightarrow 2) h_1^T \cdot A^{-T} \cdot A^{-1} \cdot h_1 = h_2^T \cdot A^{-T} \cdot A^{-1} \cdot h_2$

Figure 2.28

Knowns and Unknowns in these 2 equations:

1) $\underbrace{h_1^T \cdot A^{-T} \cdot A^{-1}}_{\text{Knowns}} \cdot \underbrace{h_2}_{\text{Unknowns}} = 0 \quad \Leftarrow r_2^T \cdot r_2 = 0$

2) $\underbrace{h_1^T \cdot A^{-T} \cdot A^{-1}}_{\text{Knowns}} \cdot \underbrace{h_1}_{\text{Unknowns}} = \underbrace{h_2^T \cdot A^{-T} \cdot A^{-1}}_{\text{Knowns}} \cdot \underbrace{h_2}_{\text{Unknowns}} \quad \Leftarrow r_1^T \cdot r_1 = r_2^T \cdot r_2$

UNKNOWNs : —

KNOWNS : —

- Knowns = columns of the homography, because we estimated it with:
 - DLT, and
 - NON-linear refinement

• $B = A^{-T} \cdot A^{-1} \Rightarrow B$ contains the unknowns.

• A is upper triangular. $\rightarrow A = \begin{pmatrix} a_u & 0 & u_0 \\ 0 & a_v & v_0 \\ 0 & 0 & 1 \end{pmatrix} \Rightarrow B$ turns out to be symmetric

$\hookrightarrow B$ is (3×3) and symmetric \Rightarrow the unknowns in B are 6. $\left(\begin{array}{ccc} a_u & a_v & u_0 \\ a_v & a_v & v_0 \\ 0 & 0 & 1 \end{array} \right)$

Figure 2.29

=> I have 2 equations (1) and 2) for each calibration image, because there are as many homographies as the calibration images.

=> a $2n \times 6$ linear system, $n = \text{num of calibration images}$

- To solve this problem we could use only 3 calibration images ($2*3 \times 6$), BUT we prefer an overconstrained problem (20-30 calibration images) ($\Rightarrow \text{num equations} >> \text{num unknowns}$)

Figure 2.30

At the end we get the system:

$$V \cdot b = 0$$

$$\rightarrow | \begin{matrix} 2n, 6 \\ \downarrow \\ 2n \text{ eq.} \end{matrix} | \xrightarrow{\text{6 unknowns}}$$

- Unknowns = vector b = 6 unknowns

- this system is overconstrained (num equations $>>$ num unknowns):

=> we solve it with the SVD decomposition:

- we decompose the coefficient matrix V and the solution (6×1) will be the last column of the right orthogonal matrix in the SVD decomposition.

=> Now we have got:

- the homographies
- the intrinsic

=> we have to find as many set of Extrinsic as the number of calibration images.

Figure 2.31

2.4 Estimation of EXTrinsics in Zhang's

- How to estimate the extrinsic parameters given the homography and the intrinsics in Zhang's?

We can first write the homography as 3 vectors: h_1, h_2, h_3

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} = \lambda \cdot A \cdot \begin{bmatrix} r_1 & r_2 & T \end{bmatrix}$$

rotation and translation matrix

- KNOWN → —
- UNKNOWN → —

$$h_1 = \lambda \cdot A \cdot r_1 \Rightarrow r_1 = \frac{1}{\lambda} \cdot A^{-1} \cdot h_1$$

the columns of a rotation matrix have to be unit vectors

$\Rightarrow \lambda$ has to be = to the norm of $(A^{-1} \cdot h_1)$, s.t. then r_1 becomes a unit vector

$$\lambda = \|A^{-1} \cdot h_1\| \Rightarrow \|r_1\| = 1$$

and we can find r_2 by using the equation $h_2 = \lambda * A * r_2$

$$\Rightarrow r_2 = \frac{1}{\lambda} \cdot A^{-1} \cdot h_2$$

is the same λ that normalizes r_1

Figure 2.32

to find r_3 we know that r_1, r_2 and r_3 have to be a basis of the space $\Rightarrow r_3$ is the vector product of the other two.

↓

$$r_3 = r_1 \times r_2$$

Finally, we find T :

$$h_3 = \lambda \cdot A \cdot T \Rightarrow T = \frac{1}{\lambda} \cdot A^{-1} \cdot h_3$$

is the same λ as before

PROBLEM:

- λ normalizes r_1 to a unit norm
- $\Rightarrow \lambda$ is NOT going to normalize r_2
- \Rightarrow the R matrix is NOT orthogonal \Rightarrow is NOT a rotation matrix

Figure 2.33

SOLUTION:

- => I turn R as orthogonal by the SVD decomposition by substituting the diagonal matrix Σ (sigma) with the identity matrix I

$$R = U \cdot \Sigma \cdot V^T$$

NON orthogonal
different from I (identity matrix)

To make R orthogonal:

=> I substitute Σ (sigma) with the identity matrix I

$$R^* = U \cdot I \cdot V^T$$

-> R^* is now orthogonal

=> we solved the problem of NON-perfect normalization of λ by turning R into an orthogonal matrix

Now we have everything
Now we need Lens Distortion Parameters:

Figure 2.34

2.5 Lens distortion

- Sometimes in images we can see curve lines instead of expected straight lines.
- this is due to lens
- we need lens to gather enough light in a small amount of time and keep the image on focus. The lens conveys all light into a point.
- we see curves because lens are not ideal, they are not thin.
- So, we need to model Lens Distortion (which is a task of camera calibration).
 - model a set of parameters -> to undistort the image
- 2 types of distortions:
 1. Radial distortion
 - more distortion as long as we move away from the center of distortion
 - > radius => > distort

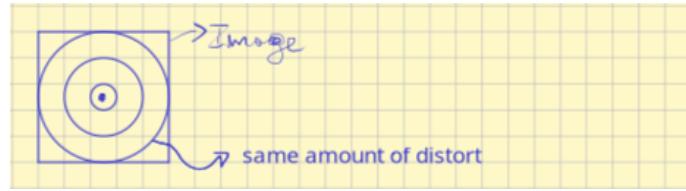


Figure 2.35

2. Tangential distortion

- due to the fact that ideally the lens should be perfectly parallel to the image plane, but it could be a little moved.
- Lens distort is modeled with a **non-linear** equation:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = L(r) \cdot \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} + \begin{pmatrix} dx_{\tilde{x}} \\ dy_{\tilde{y}} \end{pmatrix}$$

- \sim = UNdistorted image coords
 - $,$ = distorted image coords
 - $dx_{\tilde{x}}, dy_{\tilde{y}}$ = tangential distort
 - $L(r)$ = radial distort
 - they're CONTINUOUS coords in the image plane (they're not pixels)

Figure 2.36

- So, Lens Distortion is modeled **BEFORE PIXELIZATION**.

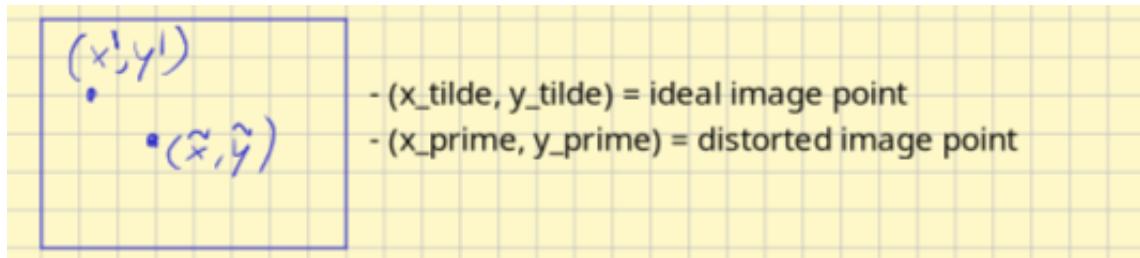


Figure 2.37

- $L(r) \rightarrow$ increasing function of the radius
 - **Radius** = distance of the undistorted pixel wrt the center of distortion:
 $r = \sqrt{(\tilde{x} - \tilde{x}_c)^2 + (\tilde{y} - \tilde{y}_c)^2}$
 - **Center of distortion:** $(\tilde{x}_c, \tilde{y}_c)$.
- $L(r)$ is non-linear and complex \Rightarrow approximate it through Taylor about the origin ($r=0$):

$$L(r) = \underbrace{L(0)}_1 + k_1 \cdot r^2 + k_2 \cdot r^4 + \dots$$

$L(r)$ = radial distortion

$L(0) = 1 \rightarrow$ at the center of distortion there is no radial distortion

Figure 2.38

- Consider only **even terms** because the radial distortion ($L(r)$) is an even function (it's defined only for $r \geq 0$). ($L(-r) = L(r)$).
- So, to model radial distort ($L(r)$) I have to model the 2 params:
 - k_1
 - k_2
- While, to model tangential distort I have other 2 params:
 - p_1
 - p_2

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \begin{pmatrix} 2 \cdot p_1 \cdot \tilde{x} \cdot \tilde{y} + p_2 (r^2 + 2 \cdot \tilde{x}^2) \\ p_1 (r^2 + 2 \cdot \tilde{y}^2) + 2 \cdot p_2 \cdot \tilde{x} \cdot \tilde{y} \end{pmatrix}$$

Figure 2.39

2.5.1 Estimation of radial lens distortion coefficients (k_1, k_2) with Zhang's

Radial Distortion in Zhang's:

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = L(r) \cdot \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = (1 + k_1 \cdot r^2 + k_2 \cdot r^4) \cdot \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix}$$

\downarrow
DISTORTED

\downarrow
UNDISTORTED

r = distance of the undistorted pixel wrt the center of distortion

Figure 2.40

- To estimate k_1, k_2 I need to know distorted and UNdistorted coords fo the same point.
 - distort coords = corners in the image (blue point)
 - UNdistort coords = projection of the world coord into the image (green point) (we get ideal coords because homographies don't model lens distortion):

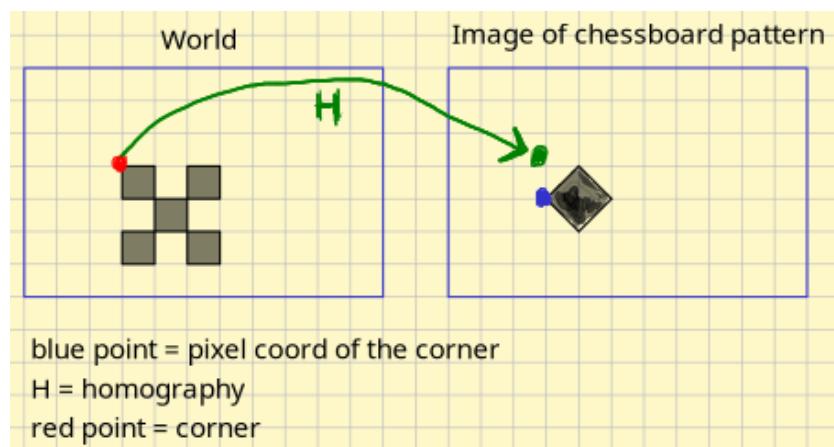


Figure 2.41

- but the distort and undistort coords are **continuous** coords in the image plane
 \Rightarrow not pixels.
 - so, to turn continuous coords into pixel coords we use **A = the intrinsic parameters**:

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = A \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_u & 0 & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \rightarrow \begin{cases} x' = \frac{u' - u_0}{\alpha_u} \\ y' = \frac{v' - v_0}{\alpha_v} \end{cases}$$

↓ distorted pixel coords

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ 1 \end{bmatrix} = A \cdot \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_u & 0 & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ 1 \end{bmatrix} \rightarrow \begin{cases} \tilde{x} = \frac{\tilde{u} - u_0}{\alpha_u} \\ \tilde{y} = \frac{\tilde{v} - v_0}{\alpha_v} \end{cases}$$

↓ UNdistorted pixel coords

Figure 2.42

- now I rewrite the lens distort model with pixel coords:

distorted continuous coords undistort continuous coords

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = (1 + k_1 \cdot r^2 + k_2 \cdot r^4) \cdot \begin{bmatrix} \tilde{x} \\ \tilde{y} \end{bmatrix}$$

↓

$$\frac{u' - u_0}{\alpha_u} = (1 + k_1 \cdot r^2 + k_2 \cdot r^4) \cdot \left(\frac{\tilde{u} - u_0}{\alpha_u} \right)$$

— = knowns
— = unknowns

$$\frac{v' - v_0}{\alpha_v} = (1 + k_1 \cdot r^2 + k_2 \cdot r^4) \cdot \left(\frac{\tilde{v} - v_0}{\alpha_v} \right)$$

undistorted pixels → knowns
because predicted by the homography

distorted pixels → knowns because
they're found in the image

↓

Figure 2.43

$$\begin{aligned}
 u' &= (1 + k_1 r^2 + k_2 r^4) \cdot \left(\frac{\tilde{u} - u_0}{\alpha_u} \right) + u_0 = \\
 &= (k_1 r^2 + k_2 r^4) \cdot (\tilde{u} - u_0) + (\tilde{u} - u_0) \cdot 1 + u_0 = \\
 &= \tilde{u} + (k_1 r^2 + k_2 r^4) \cdot (\tilde{u} - u_0)
 \end{aligned}$$

knowns:
 - \tilde{u} , \tilde{v} (undistorted pixel coords obtained with homography)
 - u' , v' (distorted pixel coords)
 - r = distance of undistorted point wrt the center of distortion (which is the center of the image)

$$\begin{aligned}
 v' &= \tilde{v} + (k_1 r^2 + k_2 r^4) \cdot (\tilde{v} - v_0) \\
 \downarrow &\quad \quad \quad D \quad \quad \quad K \quad \quad \quad d \\
 \begin{bmatrix} (\tilde{u} - u_0) r^2 & (\tilde{u} - u_0) r^4 \\ (\tilde{v} - v_0) r^2 & (\tilde{v} - v_0) r^4 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} &= \begin{bmatrix} u' - \tilde{u} \\ v' - \tilde{v} \end{bmatrix} \\
 &\quad \quad \quad \text{UNKNOWNs}
 \end{aligned}$$

$$r^2 = \left(\frac{\tilde{u} - u_0}{\alpha_u} \right)^2 + \left(\frac{\tilde{v} - v_0}{\alpha_v} \right)^2$$

Figure 2.44

- for 1 control point (1 corner) I have 2 equations in 2 unknowns
 - but it's better to consider $2*m*n$ equations (m = num of control points, n = num of calibration images) in 2 unknowns
 - $D k = d$
 - * D -> coefficient matrix ($2mn \times 2$)
 - * k -> vector of the unknowns ($2mn \times 1$)
 - * d -> vector of knowns
 - this system ($Dk = d$) can be solved as a **least square problem**:

$$k^* = \arg \min \left(\| D \cdot k - d \|^2 \right)$$

- k^* is the solution that minimizes the square norm
 - This problem can be solved in 2 ways:
 1. pseudo inverse of the coefficient matrix $D \Rightarrow D^+ \Rightarrow k = D^+ \cdot d$
 2. using the normal equations method $\Rightarrow k = (D^T \cdot D)^{-1} \cdot D^T \cdot d$

Figure 2.45

2.6 Refinement by non-linear optimization - Zhang's

We have found everything (intrinsic, extrinsics, and lens distortion parameters),
 BUT at the end we apply a NON-linear refinement:

$$\left\| \sum_{i=1}^n \sum_{j=1}^m \| m_{i,j} - \hat{m}(A, k, R_i, T_i, w_j) \|^2 \right.$$

$m_{i,j} \rightarrow$ generic corner found by the Harris corner detector into the current image.

i index \rightarrow goes through the images

j index \rightarrow m corners of the image.

\hookrightarrow there are m corners.

$\hat{m} \rightarrow$ pixel position predicted by our model

- A = intrinsic
- k = lens distortion parameters
- R_i \rightarrow extrinsics for the current image i
- T_i
- w_j = position of the world point

- we want to minimize the distance $\| m_{i,j} - \hat{m}(A, k, R_i, T_i, w_j) \|^2$
 \Rightarrow the reprojection error between the observed and the predicted pixel,

and we do it for each and every corner $(\sum_{j=1}^m)$,

and for each and every image $(\sum_{i=1}^n)$

\Rightarrow we minimize the cost function wrt the calibration parameters with Levenberg-Marquardt, up to a certain convergence level.

Figure 2.46

2.7 Point Cloud having Depth Camera

2.7.1 What if you have a depth camera, you have a z, and you want to go back into the 3D world (CRF) knowing the intrinsics?

(e.g Kinect)

depth camera: it's a camera in which, for every pixel in the image, you don't get an intensity or a colour, but you get a depth \Rightarrow an estimation of how far from the camera reference frame is a world point.

These devices output a $^{(3D)}$ data structure, called point cloud, which is a list of points, and for each point we have x, y, z coordinates.

↳ we have a point cloud in which the 3D coordinates of the 3D points projected into each pixels are expressed into the camera reference frame.

Figure 2.47

$A[R \cdot T]$

\downarrow

one factorization of the FPM

\downarrow

point in 3D

$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$ = we get a pixel

$\begin{bmatrix} R & T \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow$

they represent the rotation and translation of the reference system in which coordinates x are expressed with respect to the camera reference system

$\hookrightarrow R, T$ are the change of coordinates from the world reference system to the camera reference system.

But what if $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$ are already measured in the camera reference system?

↳ then R will become a 3×3 identity matrix, and T will become a zeros vector, because in this case the world reference system is already the camera reference system.

Figure 2.48

$$[I \ \phi] \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = [I \ \phi] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

The goal here is to find $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$

Now if we want to move to pixels we have to multiply it by A : $\begin{bmatrix} u_0 & 0 & v_0 \\ 0 & u_v & v_v \\ 0 & 0 & 1 \end{bmatrix}$:

$$\begin{bmatrix} u_u \cdot x + v_u \cdot z \\ u_v \cdot y + v_v \cdot z \\ z \end{bmatrix} = \underbrace{\begin{bmatrix} u_u & 0 & v_u \\ 0 & u_v & v_v \\ 0 & 0 & 1 \end{bmatrix}}_A \underbrace{\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}_P$$

\downarrow

$P^* = A \cdot P$

I am interested in finding P (=the 3D coords), because I want my sensor to output a point cloud.

\Downarrow

$P = A^{-1} \cdot P^*$ \Rightarrow I can find 3D coords as a function of intrinsic parameters and pixel coordinates.

Figure 2.49

From this ($P = A^{-1} \cdot p^*$) I am saying that I can find 3D coords from pixel coords, but it's impossible, because there's ambiguity along depth.

↳ Indeed p^* are homogeneous pixel coordinates in which the 3rd coord is z .

↳ Now I can just multiply and divide by z :

$$P = z \cdot A^{-1} \cdot \frac{p^*}{z}$$

$$A^{-1} = \begin{bmatrix} \frac{1}{\alpha_u} & 0 & -\frac{u_0}{\alpha_u} \\ 0 & \frac{1}{\alpha_v} & -\frac{v_0}{\alpha_v} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\frac{p^*}{z} = \begin{bmatrix} \alpha_u \frac{x}{z} + u_0 \\ \alpha_v \frac{y}{z} + v_0 \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = p$$

actual pixel coords
that we know

✳️ $P = z \cdot A^{-1} \cdot p$

3D coordinates of the world points which projects into the pixel p , in the CRF.

I know z (depth), because I have a depth sensor.

✳️ Formula to obtain, in a calibrated depth camera, a point cloud defined in the CRF

Figure 2.50

2.7.2 Now if you want to find the corresponding pixels of another view of the same scene taken by the same depth camera? (You move the camera, camera motion being known)

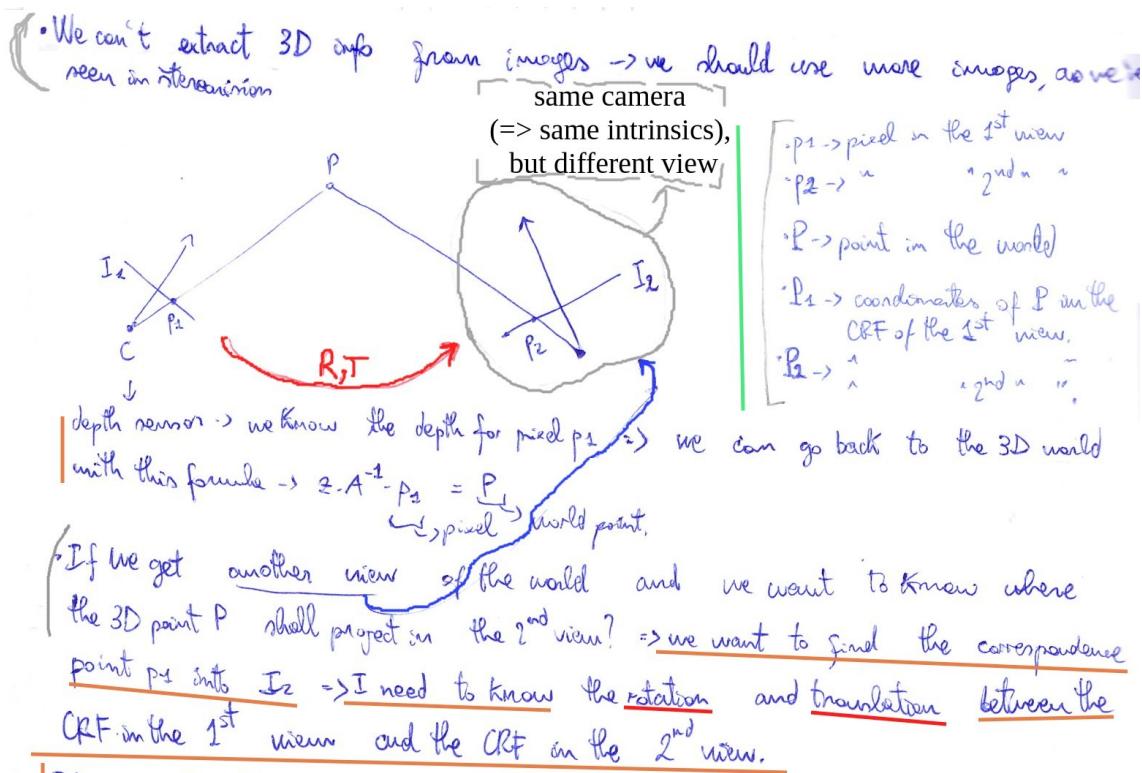


Figure 2.51

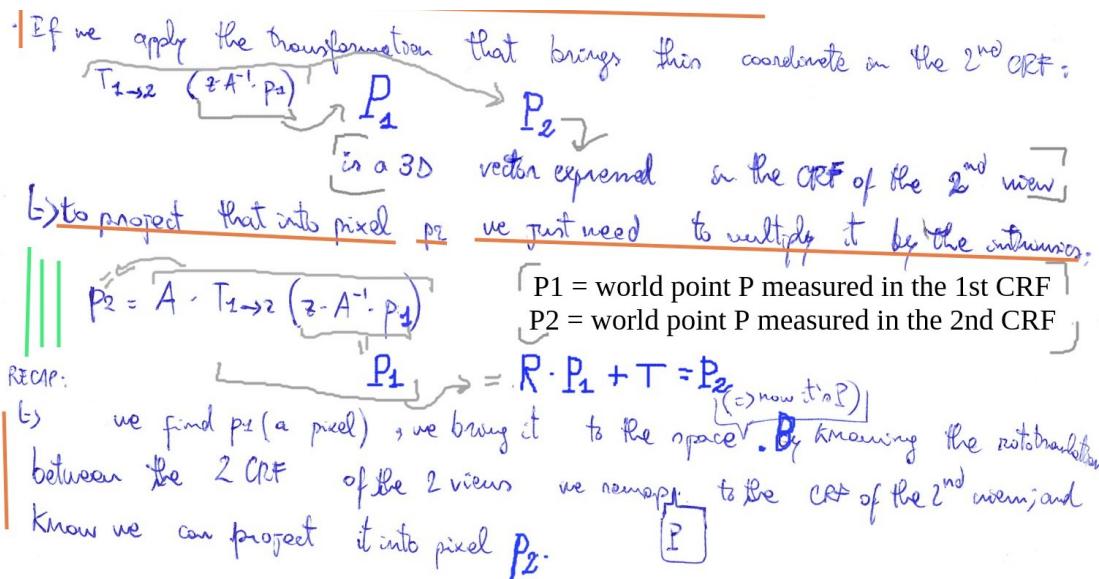


Figure 2.52

(b) we can remap pixels into different views of the same camera

Figure 2.53

2.7.3 case with 2 different cameras

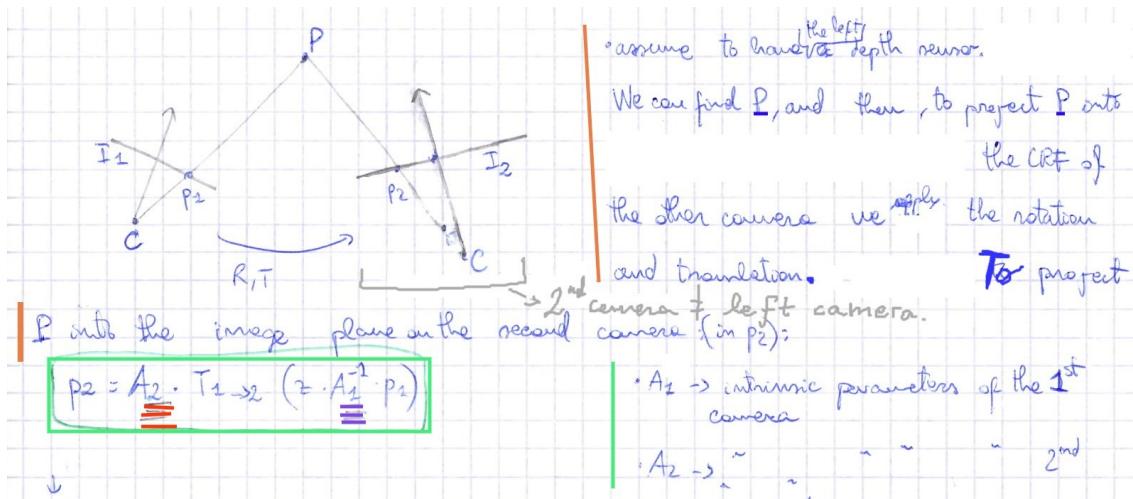


Figure 2.54

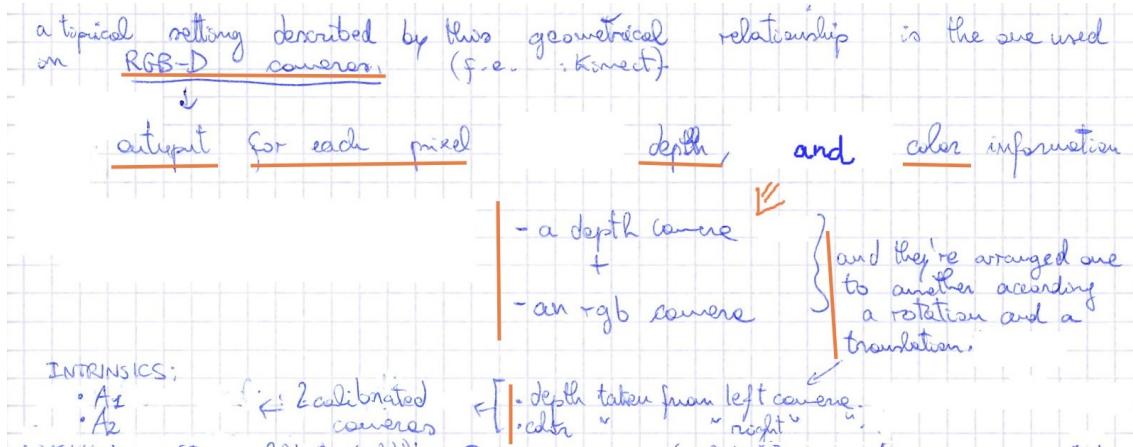


Figure 2.55

2.8 Image Warping

We have:

- 2 images I and I' , with coords in I denoted by (u, v) and in I' denoted by (u', v') .
- a vector value function $f(u, v)$, which takes the coords in image I and computes corresponding coords in image I' .

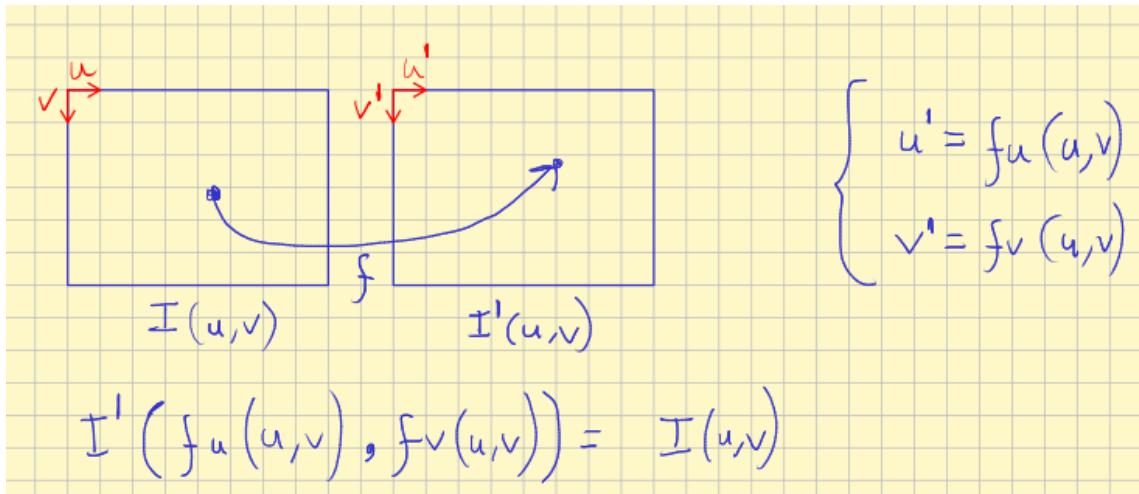


Figure 2.56

- **Def warping:** warping is defined through these 2 functions:
 1. f_u gives the new u coord (u') given the old u, v coords
 2. f_v gives the new v coord (v') given the old u, v coords
 3. and you need to apply these to remap pixel coords from one image to another
- warping is some kind of transformation between pixel coords in 1 image and pixel coords in another image.
- intensity at position (u, v) in $I \rightarrow$ is transferred at pos (u', v') (or $f_u(u, v)$, $f_v(u, v)$) in target image I'
- examples of image warping:
 - rotation
 - remove deformations (license plate)
- How to perform warping?

2.8.1 Forward Mapping

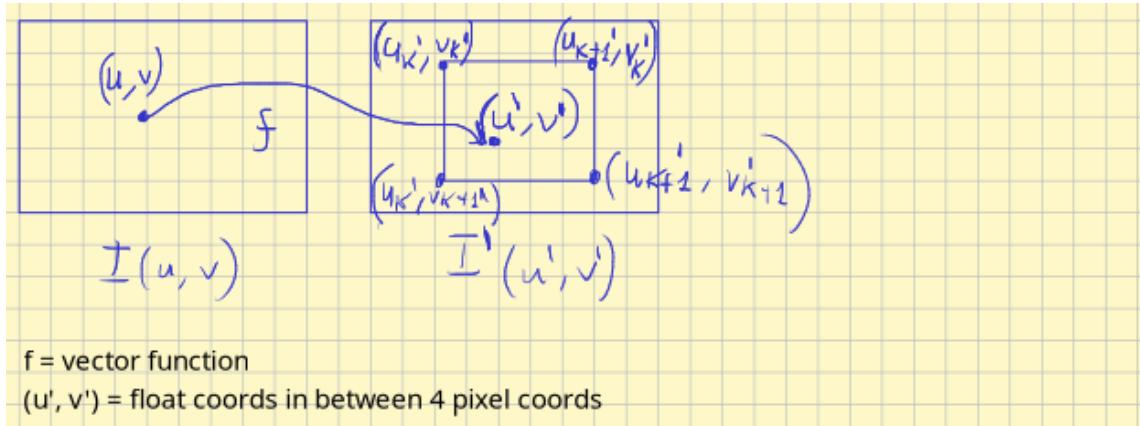


Figure 2.57

- Problem: from pixel coords (**integer**) we can obtain (through warping) **float** coords.
 - so, I obtain something in between 4 pixel pos
- I can assign the intensity of the closest of the 4 pixels:

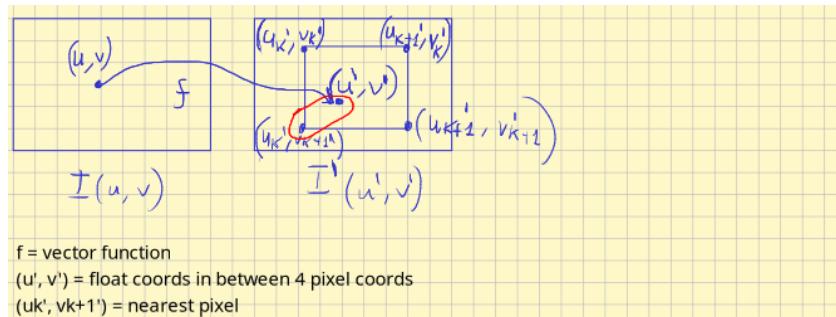


Figure 2.58

- but **problems**:
 1. no guarantee that all the pixels in the output image will be hit.
 2. due to rounding errors, some pixels in the output image can be hit multiple times => multiple intensities
- So, Forward Mapping is **not** good

2.8.2 Backward Mapping

- I can apply it to guarantee that all the pixels in the output image will have an intensity
- warping function defined backward: map output coords -> into input coords:

$$\begin{aligned}
 u &= g_u(u', v') & u, v &= \text{pixels of the input image} \\
 v &= g_v(u', v') & g_u, g_v &= \text{mapping functions} \\
 && u', v' &= \text{pixels of the output image} \\
 \Downarrow && \text{intensity of the pixel } u', v' \text{ in the output image} \\
 \forall (u', v') : \quad I'(u', v') &= I(g_u(u', v'), g_v(u', v'))
 \end{aligned}$$

Figure 2.59

- No problems now (no holes and folds):
 - I assign a value to each output pixel
 - if more output pixels map into one input value, then it is **not** a problem

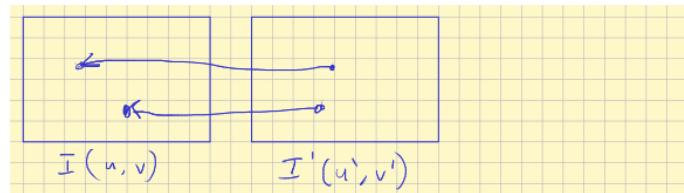


Figure 2.60

Mapping Strategies

- Other problem: when applying backward mapping we still output a real value (float):
 - Mapping Strategies:
 1. floor function -> always top left pixel -> **not** so good
 2. map to closest point (nearest neighbour mapping)
 3. **Bilinear Interpolation:**
 - compute value to assign to the output pixel as a function of all the intensities of the 4 closest pixels in the input image
 - * (this couldn't be done with forward mapping because there we go to the output and the output doesn't have intensity values)
- Bilinear Interpolation:

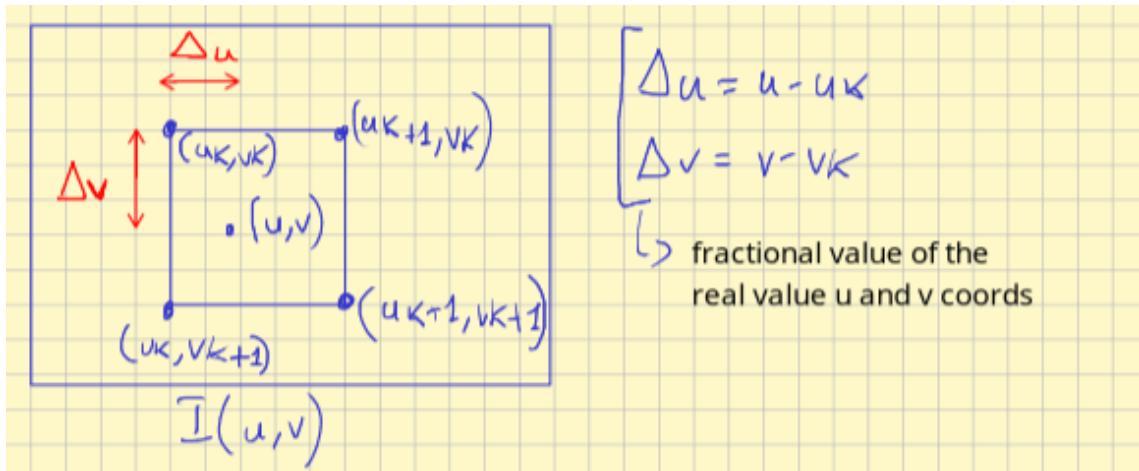


Figure 2.61

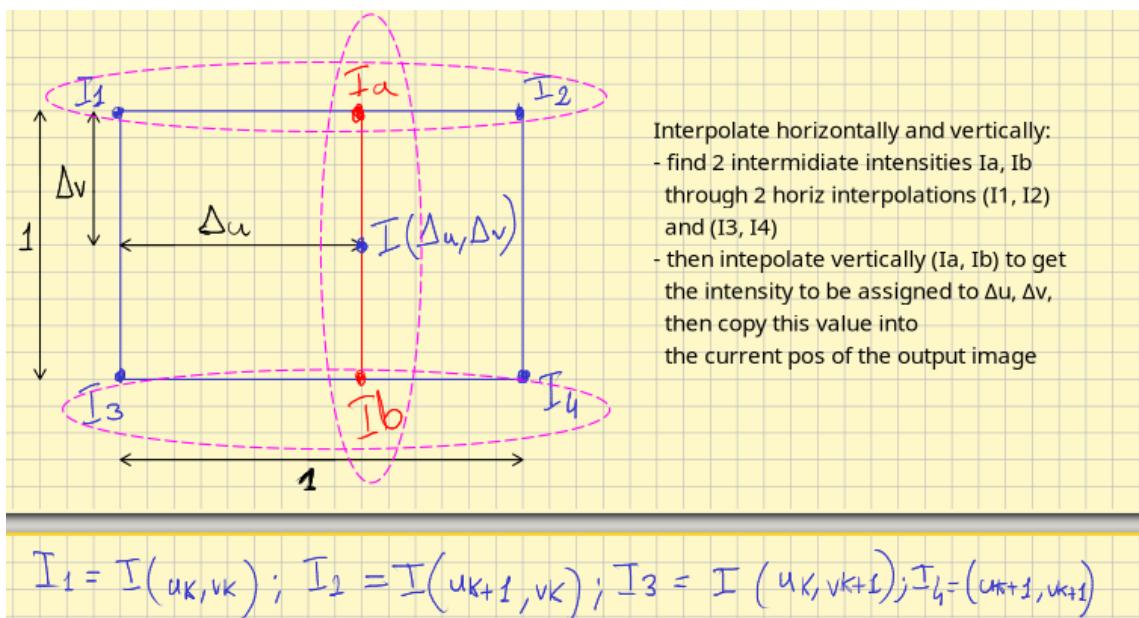


Figure 2.62

- How to compute a linear interpolation:

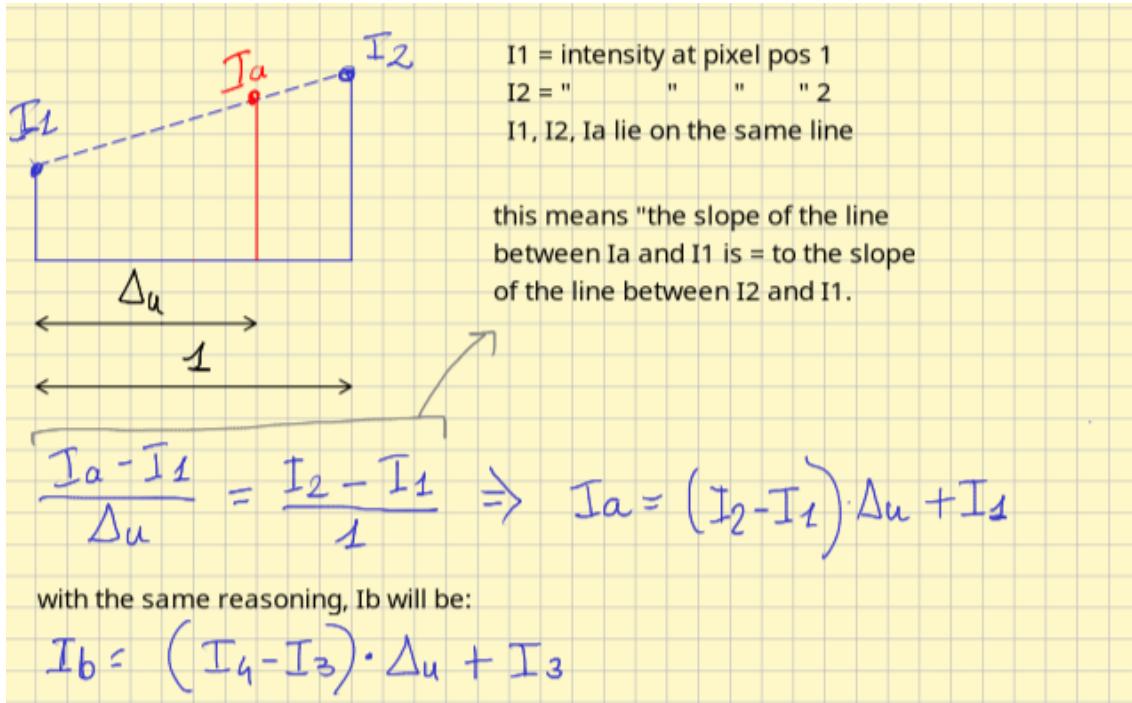


Figure 2.63

now we have the 2 horiz interpolations => we compute the vertical interpolation with $I_a, I_b, \Delta v$:

$$\begin{aligned}
 I(\Delta u, \Delta v) &= (I_b - I_a) \cdot \Delta v + I_a = \\
 &= \underbrace{\left((I_4 - I_3) \cdot \Delta u + I_3 \right)}_{I_b} - \underbrace{\left((I_2 - I_1) \cdot \Delta u + I_1 \right)}_{I_a} \cdot \Delta u + \underbrace{\left((I_2 - I_1) \cdot \Delta u + I_1 \right)}_{I_a}
 \end{aligned}$$

so, you plugin $\Delta u, \Delta v$, then the 4 intensities at the 4 pixels of the input image and get $I(\Delta u, \Delta v)$, which is the interpolated value to be assigned in the current pixel position of the output image I' .

with some manipulations you get:

$$I'(u', v') = (1 - \Delta u) \cdot (1 - \Delta v) \cdot I_1 + \Delta u \cdot (1 - \Delta v) \cdot I_2 + (1 - \Delta u) \cdot \Delta v \cdot I_3 + \Delta u \cdot \Delta v \cdot I_4$$

Figure 2.64

- so the interpolated intensity $I'(u', v')$ is a weighted sum of the 4 input intensities of the 4 nearest pixels.
 - and the weight of an intensity is higher as its coords are closer to $\Delta u, \Delta v$. And if:
 1. $\Delta u, \Delta v = 0 \Rightarrow I'(u', v') = I_1$ (I' is on the top left corner).
 2. $\Delta u, \Delta v = 1 \Rightarrow I'(u', v') = I_4$ (I' is on the bottom right corner).
 3. ecc.
 - so, more closer to a given pixel => more weight to its intensity.
- Example of backward warping: lens distortion

- I have the radial lens distortion params (k_1, k_2)
- backward warp from undistorted to distorted image based on the lens distort model

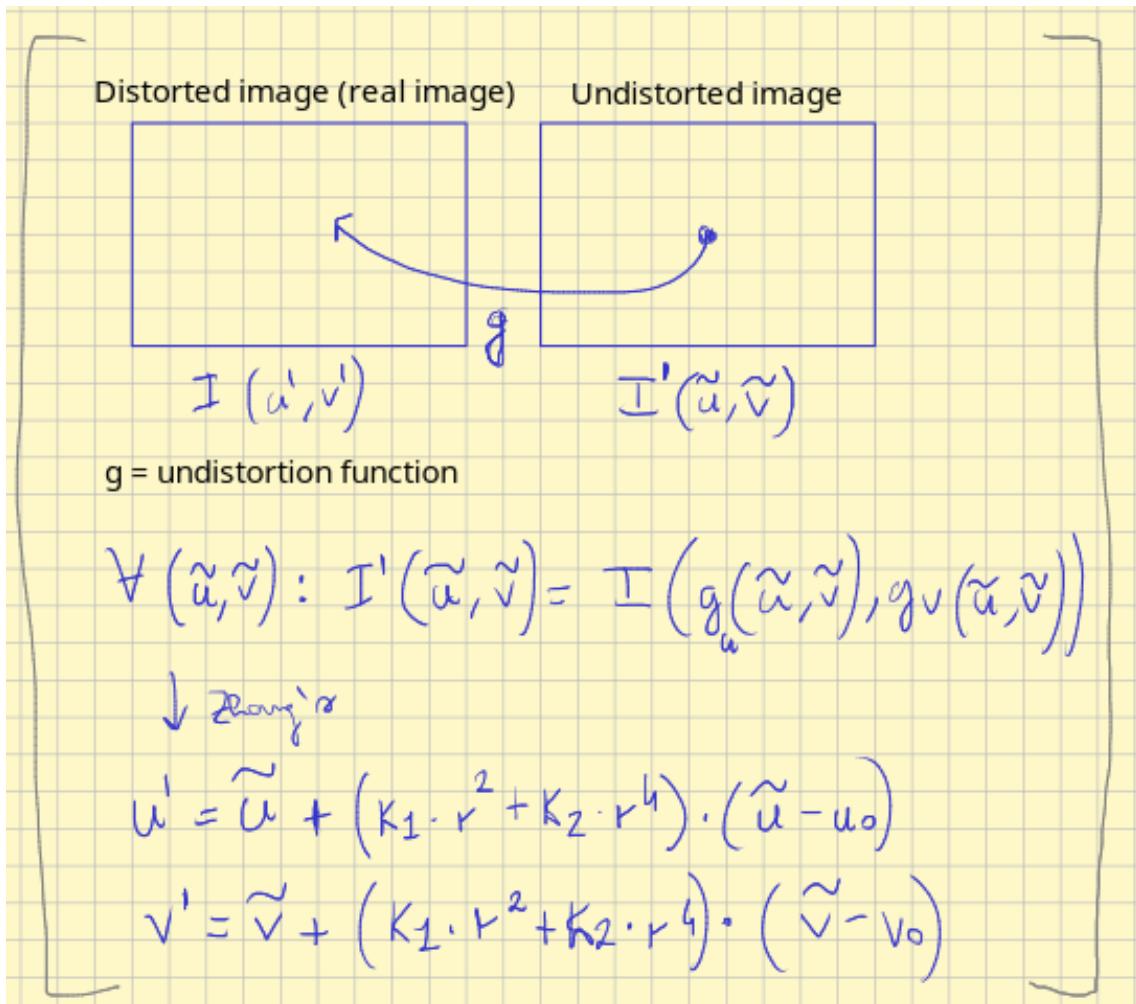


Figure 2.65

Chapter 3

Intensity Transformations

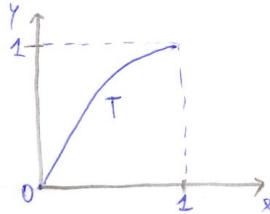
3.1 Histogram Equalization

- is an Intensity Transformation Operator,
 - and it's fully automatic.
- Goal: IMPROVING THE CONTRAST of an image.
 - this is achieved through an equalizing function that TRIES to spread gray levels uniformly across the whole range.
 - * We don't get a uniform histogram in practice due to the discrete settings of images.
 - * And the image won't be better if we have the same number of pixels in the gray levels,
 - but the image will look better (have a better contrast), because we use the whole gray level range.
- The concept of Equalization comes from Probability theory:

consider:

- $x \rightarrow$ continuous random variable
- $T \rightarrow$ a strictly monotonically increasing function \rightarrow it will give us another random variable y

$$x \in [0,1] \rightarrow y = T(x) \in [0,1]$$

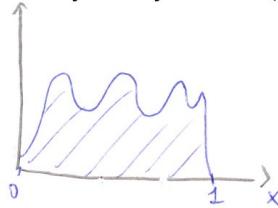


\Rightarrow we want to find a T function:

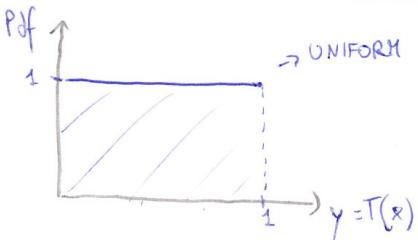
- that maps x into a UNIFORM random variable y

random variable whose probability distribution is flat

probability density function (Pdf) of x



probability density function of $y \rightarrow$ flat



(N.B.: Whatever x , it's always possible to turn x into a uniform random variable with a flat Pdf)

\Rightarrow we want to EQUALIZE the random variable x

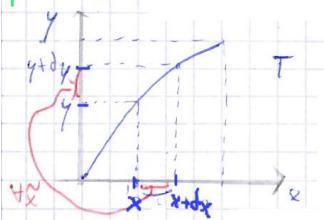
Figure 3.1

So, let's consider an infinitesimal interval $[x, x+dx]$, but first we introduce:

$P_x(x)$ = Pdf of x = probability density function of the input variable x

$P_y(y)$ = Pdf of $y =$ " " " output " " y

T = monotonically increasing function



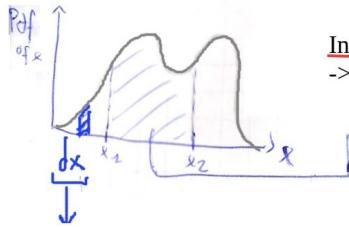
- $[x, x+dx] \rightarrow$ infinitesimal interval
- $T(x) = y$
- $T(x+dx) = y+dy$

$$\forall \tilde{x} \in [x, x+dx] \rightarrow \tilde{y} = T(\tilde{x}) \in [y, y+dy]$$

\Rightarrow Whatever x_{tilde} I take (\Rightarrow whatever x in $[x, x+dx]$), I end up with a which is in the interval $[y, y+dy]$ due to the transform function T being monotonically increasing

Figure 3.2

\Rightarrow The probability of the random variable x to lie in $[x, x+dx]$ is the same as the probability of random variable y to lie in $[y, y+dy]$, because whatever x in $[x, x+dx]$ takes me into $[y, y+dy]$.



In general, to compute the probability of a point x to be in an interval $[x_1, x_2]$:
 -> we should compute the integral of the Pdf of x between x_1 and x_2 :

$$\Rightarrow P(x \in [x_1, x_2]) = \int_{x_1}^{x_2} p_x(\xi) d\xi$$

Instead, if I take an infinitesimal interval dx , then the probability of x to belong to dx is equal to:
 $p_x * dx$

- p_x = Pdf of x
 - p_y = Pdf of y

$$p_x \cdot dx = p_y \cdot dy \Rightarrow p_y(y) = p_x(x) \cdot \frac{dx}{dy}$$

it's the derivative
of the inverse
function T

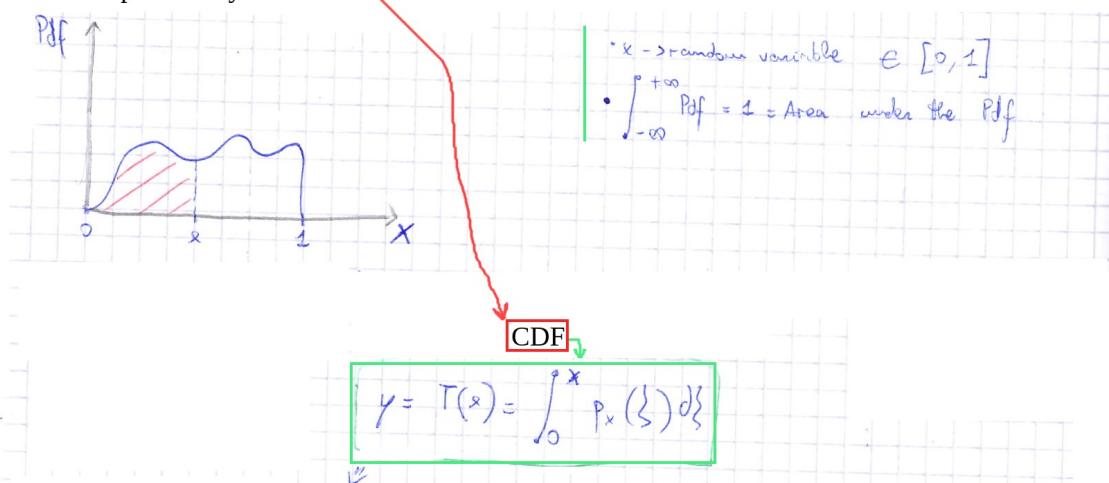
because $y = T(x) \Rightarrow \frac{dy}{dx} = T'(x) \Rightarrow \frac{dx}{dy} = \frac{1}{T'(x)}$

Figure 3.3

- So, to turn x into a UNIFORM random variable y we use the cdf (CUMULATIVE DISTRIBUTION FUNCTION), which:

- is a NON-linear function
- and it maps values into $[0,1]$
- and it's monotonically increasing

The cdf at value x is the area of the Pdf up to x
 => it's the probability that $x \leq x$



- => cdf = probability density function up to x.
- The cdf:
- is a function of x (because x is in the integral)
- maps values to $[0,1]$
- is monotonically increasing, because the area increases if we increase x

We assume it to be STRICTLY monotonically increasing,
 BUT in some cases the cdf is NOT STRICTLY monotonically increasing:

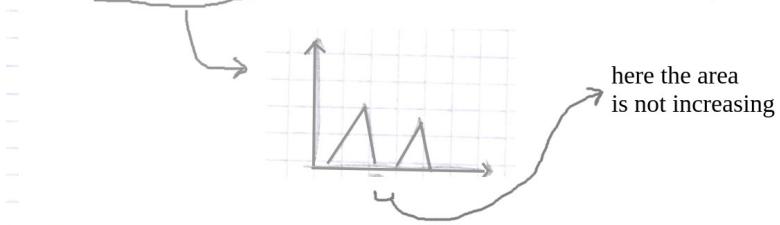


Figure 3.4

$$F(x) = \int_0^x p_x(s) ds \Rightarrow F'(x) = p_x(x)$$

the derivative of an integral function is the function which is integrated $\Rightarrow F' \circ p_x(x)$

So, the previous relation becomes:

$$P_Y(y) = p_x(x) \cdot \frac{dx}{dy} = p_x(x) \cdot \frac{1}{\frac{dy}{dx}} = p_x(x) \cdot \frac{1}{F'(x)} = \frac{p_x(x)}{p_x(x)} = 1$$

∴

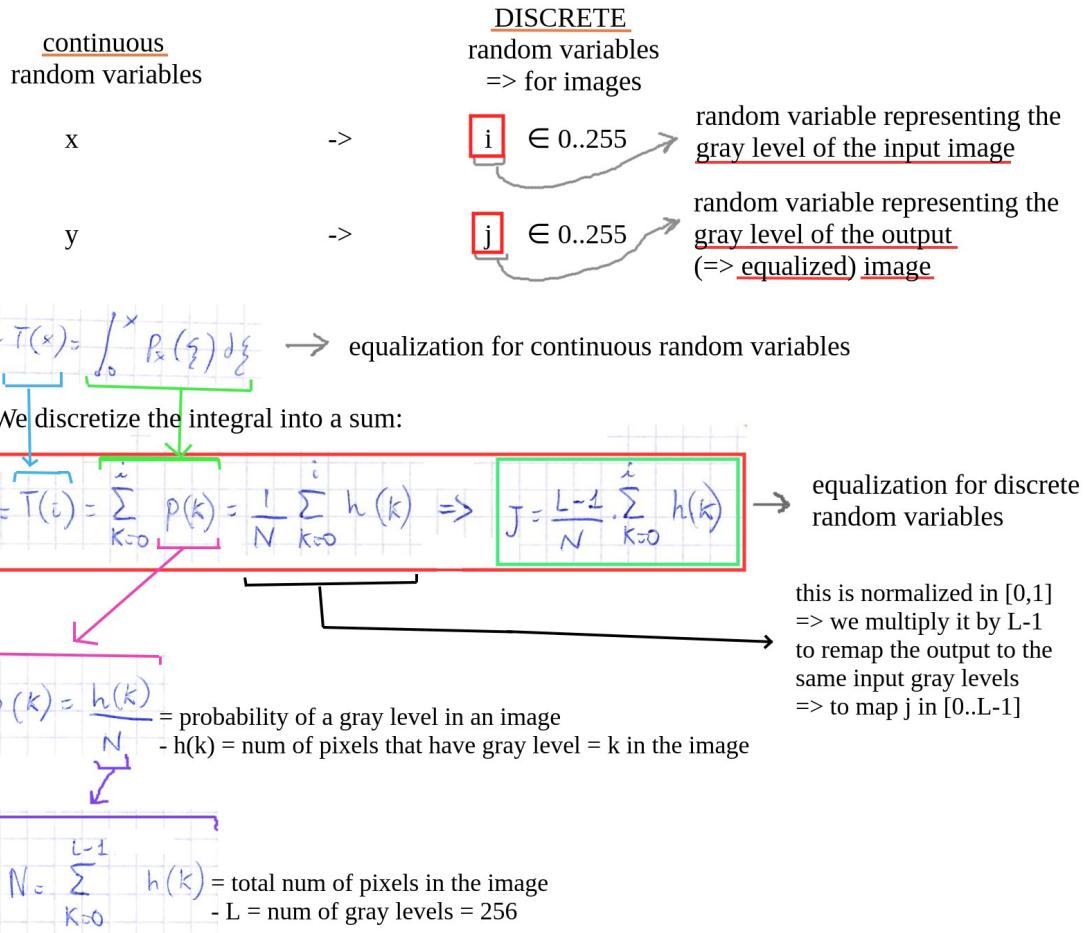
=> we have turned x (whatever is its Pdf \Rightarrow its $p_x(x)$) into a random variable y, whose Pdf ($p_y(y)$) is always = 1.

=> we have turned x into a UNIFORM random variable.

=> We can EQUALIZE a continuous random variable through the CDF.

Figure 3.5

- => Now we want to turn the CDF into a DISCRETE formulation:



this is normalized in $[0,1]$
 \Rightarrow we multiply it by $L-1$
 to remap the output to the same input gray levels
 \Rightarrow to map j in $[0..L-1]$

$$p(k) = \frac{h(k)}{N} = \text{probability of a gray level in an image}$$

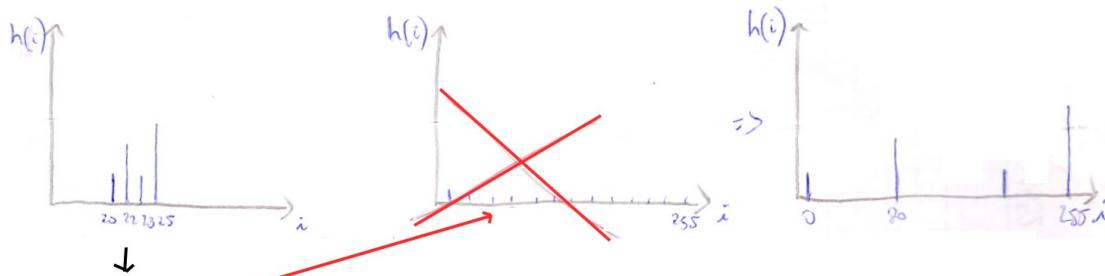
- $h(k)$ = num of pixels that have gray level = k in the image

$$N = \sum_{k=0}^{L-1} h(k) = \text{total num of pixels in the image}$$

- L = num of gray levels = 256

Figure 3.6

- Ex of equalization. And why you can't spread uniformly gray levels in discrete settings:



It's NOT possible to spread all pixels into a uniform histogram in discrete settings because we apply $j = T(i)$ (\Rightarrow a function of i)

\Rightarrow when f.e. $i = 20$, then all pixels whose gray level is $= 20$ will end up in the same position j in the output image.

- all pixels with gray level $= 22$ ($i=22$) will f.e. remapped in the gray level $j=80 \Rightarrow$ you remap in larger range, but you did NOT spread the pixels equally across gray levels.

- one thing that could happen instead is the following:

- you can collapse f.e. pixels with $i=20$ and $i=22$ into the same output gray level $j=80$, because of rounding

\Rightarrow at least you can reduce the number of gray levels.

Figure 3.7

RECAP : the real goal of equalization is that if you start with an image in which the gray levels are too much concentrated in a small range (\Rightarrow the contrast is poor), then when you equalize you spread them in the whole range \Rightarrow the output range is larger and the contrast is better.

Figure 3.8

Chapter 4

Spatial Filtering

4.1 Convolution

- it's an LSI operator that we use to apply a filter (f.e. Gaussian Filter)
- First, let's define LSI Operators (Linear and Shift-Invariant Operators):

• LSI Operators (LSI = Linear and Shift-Invariant)

- given an input 2D signal $i(x, y)$
- and a 2D operator $T\{\cdot\}$: $o(x, y) = T\{i(x, y)\}$ (= if we apply T to i (the input signal), then we get the output signal o)

=> The operator T is said to be LINEAR IFF:

$$T\{a_i i_1(x, y) + b_i i_2(x, y)\} = a_i o_1(x, y) + b_i o_2(x, y), \text{ with } o_1(\cdot) = T\{i_1(\cdot)\}, \\ o_2(\cdot) = T\{i_2(\cdot)\}$$

- o_1 and o_2 are the response of the operator to input signals i_1 and i_2 .
- a, b are two constants
= If the input is a weighted sum of signal, then the output is the same weighted sum.

Figure 4.1

The operator T is SHIFT-INVARIANT IFF:

$$T\{i(x - x_0, y - y_0)\} = o(x - x_0, y - y_0)$$

= If you shift the input and apply T , then you'll get the same shift in the output

Figure 4.2

Now we assume that the input signal can be expressed as a weighted sum of shifted elementary functions e (each shifted by a quantity (x_k, y_k))

$$i(x, y) = \sum_k w_k \cdot e_k(x - x_k, y - y_k) \rightarrow \text{we can express whatever signal by this combination}$$

$$\Rightarrow o(x, y) = T \left\{ \sum_k w_k \cdot e_k(x - x_k, y - y_k) \right\} =$$

apply LINEARITY:

- the input is a weighted sum of elementary functions e_k
- \Rightarrow due to linearity, the output is given by the same weighted sum of the responses to elementary functions

$$= \sum_k w_k \cdot T \left\{ e_k(x - x_k, y - y_k) \right\}$$

apply SHIFT-INVARIANCE:

- the elementary functions are shifted
- \Rightarrow the output of each elementary function gets shifted too

$$= \sum_k w_k \cdot h_k(x - x_k, y - y_k), \text{ with } h_k(\cdot) = T \left\{ e_k(\cdot) \right\}$$

\Rightarrow If the input is a weighted sum of shifted elementary functions, then the output is just the same weighted sum of the shifted responses of the elementary functions.

Figure 4.3

- Convolution:

Dirac Delta Function:

- is = 0 everywhere but at the origin
- its integral across the whole domain is = 1

it formalizes the impulse, something very strong in a position, and that doesn't exist elsewhere

We can express every input signal i as a weighted infinite combination of shifted Dirac Delta Functions:

$$i(x, y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta) \cdot \delta(x - \alpha, y - \beta) \cdot d\alpha \cdot d\beta$$

\Rightarrow now our elementary functions are the Dirac Delta Functions

$\sum_k w_k \cdot e_k(x - x_k, y - y_k)$ \rightarrow Relation with the discrete formulation

- α, β = amount of shift in the continuous formulation

Figure 4.4

We can get the output of an LSI operator by applying Linearity and Shift-Invariance and we obtain:

The formula of a 2D continuous CONVOLUTION:

$$o(x, y) = T\{i(x, y)\} = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta) \cdot h(x - \alpha, y - \beta) d\alpha d\beta$$

- $h(\cdot)$ is the output I get if I feed my operator T with the elementary function

- elementary functions = Dirac Delta functions = impulses
- h = impulse response = kernel

recap:

- => we know that whatever input signal $i(x, y)$ can be expressed as
- => we can get the output of an LSI operator by the formula of a 2D convolution
(to compute the output we should know its impulse response function h)

=> applying an LSI operator consists in computing a CONVOLUTION

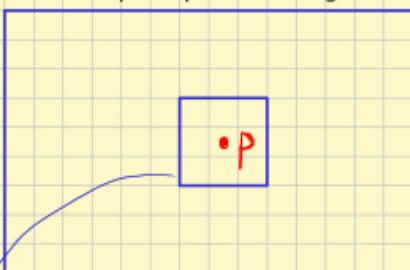
$$i(x, y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta) \cdot \delta(x - \alpha, y - \beta) d\alpha d\beta$$

Figure 4.5

- Ex of Convolution calculus:

Example of calculus of Convolution:

I have a pixel p in an image:



I take the 3x3 neighborhood around the pixel p :

1	2	3
4	5	6
7	8	9

= pixel p

now I take my kernel h , which is a 3x3 matrix:

a	b	c
d	e	f
g	h	i

- 1, 2, ... 9 = intensities of the pixels around the given one
- a, b, ..., i = coefficients of the filter

Figure 4.6

=> The output of pixel p (of pixel 5) (after applying the filter) will be computed as:

1. First let's flip the kernel:

i	h	g
f	e	d
c	b	a

Figure 4.7

2. Then, to compute the output at the pixel we're considering:

- convolution is about Multiply And Add
- so, we multiply correspondent signal values
- and then add all them up:

$$O(p) = 1 \cdot i + 2 \cdot h + \dots + 9 \cdot a$$

3. Repeat this for all pixels in the image

- with a sliding window processing
- the window is the kernel

N.B. DON'T override the input image, because otherwise you'll use previously computed outputs while sliding the kernel instead of original pixels.

Figure 4.8

- Problems with Practical Convolution:

Problem with Convolution: I canNOT compute it in the pixels which are near the borders

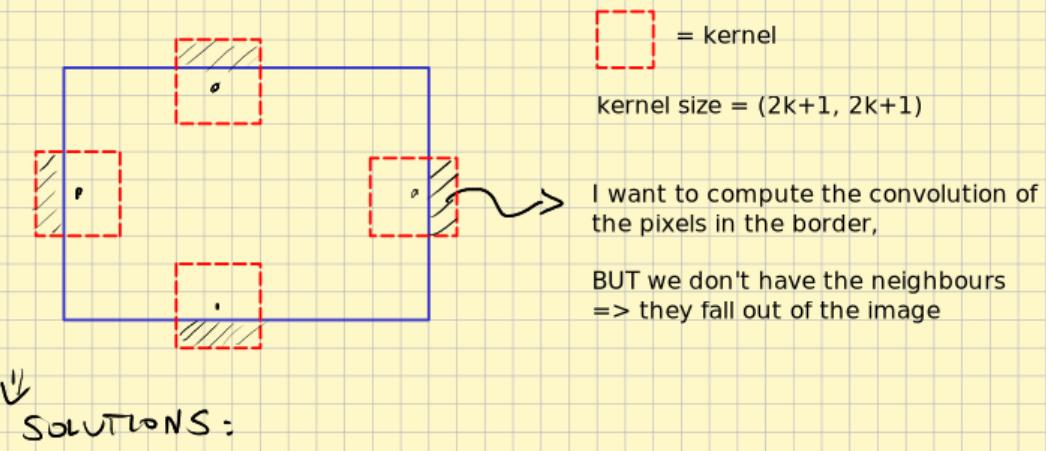
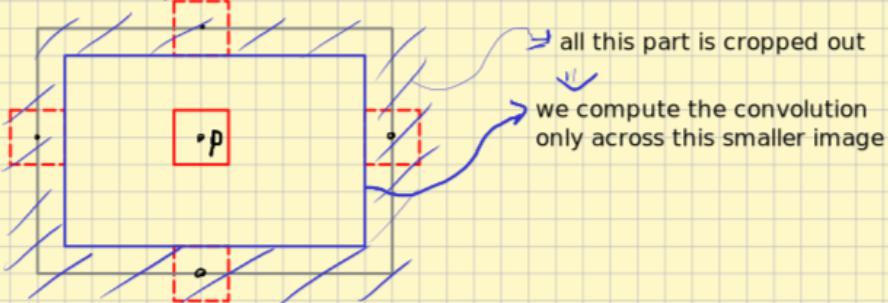


Figure 4.9

1. CROP OUT:

- compute the convolution in a smaller portion of the image
=> we don't have to create new pixels
- we don't compute the convolution in the first and last k rows and k columns



Example:

- 3x3 kernel
- => $2k + 1 = 3$
- => $k = 1$
- I don't compute the filter in the last 1 row

Cropping is bad because f.e. in a Conv. Neural Network the image will get smaller and smaller and we lose info

Figure 4.10

2. PADDING:

- 2a) -> Zero padding -> add zeros
- 2b) -> Constant number padding -> add a constant number
- 2c) REFLECT the BORDERS: -> better
-> applied to the missing initial and final rows and columns

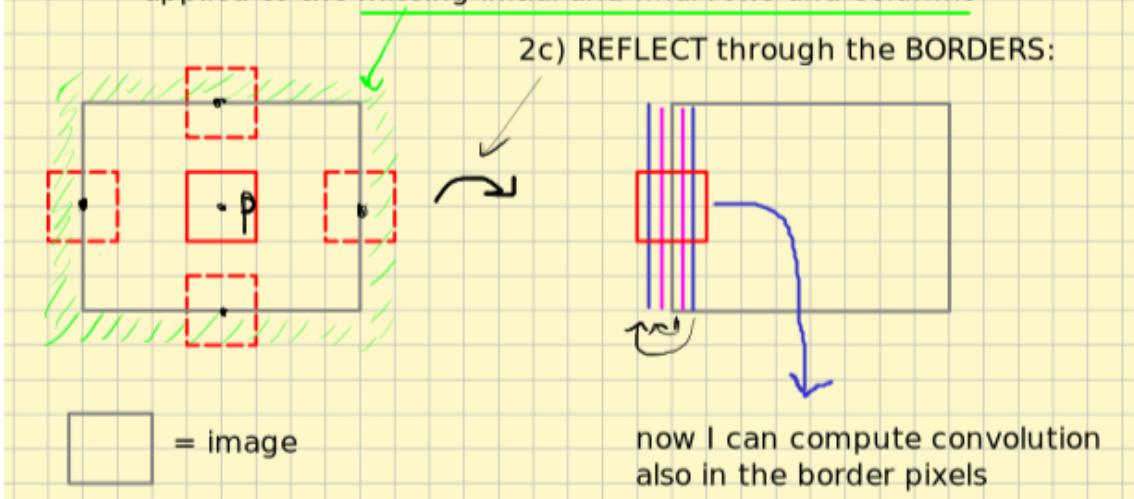


Figure 4.11

- Properties of Convolution:

Properties of Convolution

- Convolution is denoted by ' $*$ '

$$o(x,y) = i(x,y) * h(x,y)$$

- Some useful properties of convolution.

1. $f * (g * h) = (f * g) * h \rightarrow$ Associative property

It is important, because computing the chain operation is much faster than computing the convolution between individual functions.

$f * (g * h)$ or $f * W ? \rightarrow$ Computing 2 convolution is faster

rather than computing just a single convolution. \Rightarrow The chain operation is more computationally efficient.

2. $f * g = g * f \rightarrow$ Commutative property

3. $f * (g+h) = f * g + f * h \rightarrow$ Distributive property with respect to the sum

4. $(f * g)' = f' * g = f * g' \rightarrow$ Convolution commutes with differentiation

\hookrightarrow instead of applying the convolution and then do the differentiation, we can apply the derivation to one of the two functions and then

Figure 4.12

- Graphical view of Convolution:

• What are the roles of the 2 functions ' i ' and ' h '? \rightarrow Graphical interpretation

i and h are defined in a domain, which we call plane (α, β) .

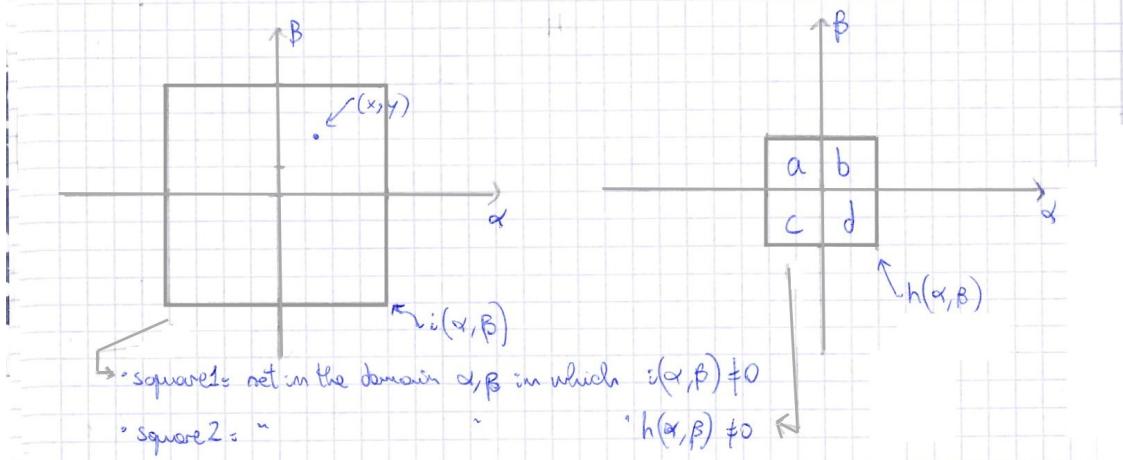


Figure 4.13

• $i \rightarrow$ input signal in our image

• $h \rightarrow$ filter, operator that is going to process the image $\rightarrow h$ is smaller than the image!

• $a, b, c, d =$ set of values taken by h in the 4 subregions.

• Convolution is about multiplying corresponding values and sum the products,

but it's not as easy, because in the definition of convolution:

$$o(s,y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta) \cdot h(x-\alpha, y-\beta) d\alpha d\beta$$

i is left unchanged, while

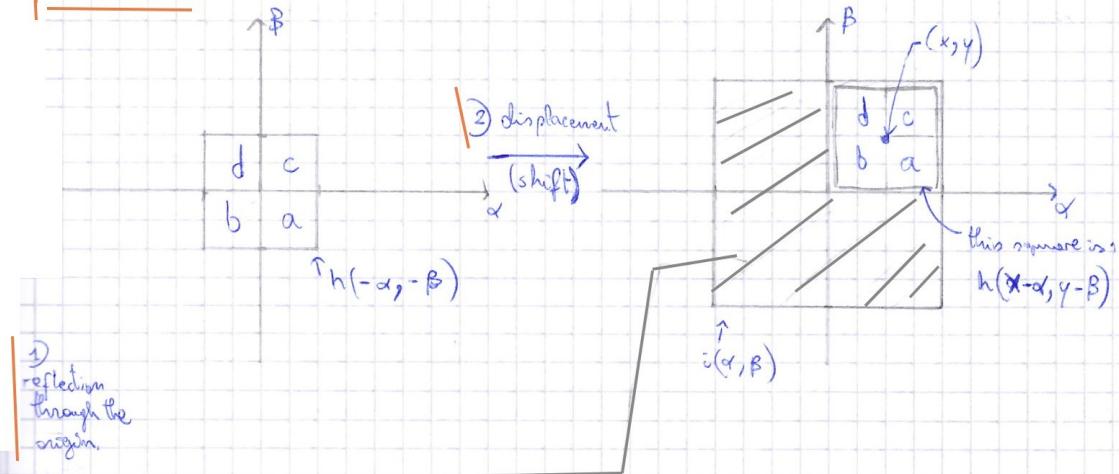
h is $h(x-\alpha, y-\beta)$

\hookrightarrow we do 2 manipulations: ① $h(-\alpha, -\beta) \rightarrow$ reflect h through the origin

② $h(x-\alpha, y-\beta) \rightarrow$ shift at (x, y) .

Figure 4.14

\Rightarrow To compute the convolution at (x, y) : we leave i unchanged, then we pick h , we reflect it through the origin, and we shift it at (x, y) . (x, y) is the position where we are going to compute the convolution.



- After this manipulation of h , we multiply them together and everything up.
- \Rightarrow It's a MAD between i and h , but before taking the actual MAD h gets reflected through the origin and shifted at the position (x, y) , at which we are willing to compute the convolution
- \Rightarrow = values of i that will not contribute to the output, because they're going to be multiplied by zero (because h in those points is $= 0$)

Figure 4.15

- \Rightarrow What is a convolution:
- \Rightarrow you take 2 functions, 1 is left unchanged, the other 1 is flipped and shifted; and then you multiply them together and add everything up. That's convolution.

Figure 4.16

- Discrete Convolution:

Discrete Convolution

- In practice continuous convolution we've to process discrete signals.
- We discretize the continuous formulation.
- An impulse in discrete settings is known as the Kronecker delta function, denoted as $\delta(i,j)$, which is just 1 at the origin and 0 elsewhere:

$$H(i,j) = T\{S(i,j)\} \quad \text{with} \quad \begin{cases} \delta(i,j) = 1 \text{ at } (0,0) \\ \delta(i,j) = 0 \text{ elsewhere} \end{cases}$$

$\bullet o(x,y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\gamma, \beta) \cdot h(x-\gamma, y-\beta) d\gamma d\beta$ CONTINUOUS FORMULATION
 \Downarrow \Downarrow \Downarrow
 $T\{I(i,j)\} : O(i,j) = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} I(m,n) \cdot H(i-m, j-n)$ DISCRETE FORMULATION
of a 2D convolution operation!

Figure 4.17

$\bullet I(m,n) \rightarrow$ discrete input signal
 $\bullet O(i,j) \rightarrow$ "output" computed at some position (i,j)

$\bullet (x,y) \rightarrow (i,j)$
 $O(i,j)$ is defined as a MAD operation between a discrete input signal I and a discrete Kernel (or impulse response) H , whereby the input signal is left unchanged ($I(m,n)$) and the Kernel (the greater, the filter) is flipped (reflected) and shifted ($H(i-m, j-n)$).

Figure 4.18

4.2 Correlation

Correlation

- it's denoted by ' \circ '.

$$i(x, y) \circ h(x, y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta) \cdot h(x + \alpha, y + \beta) d\alpha d\beta$$

↳ correlation of i toward h (it's important to specify the order, because correlation isn't commutative)

- Difference between convolution \rightarrow the '+' signs here
- The correlation is again a MAD between ' i ' and ' h ', the only difference from the convolution is in the manipulation that is done on h

- Correlation of $h(x, y)$ versus $i(x, y)$ (\Rightarrow the opposite)

$$h(x, y) \circ i(x, y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} h(\alpha, \beta) \cdot i(x + \alpha, y + \beta) d\alpha d\beta$$

↳ we've just swapped the role of the 2 functions
(now we manipulate i).

Figure 4.19

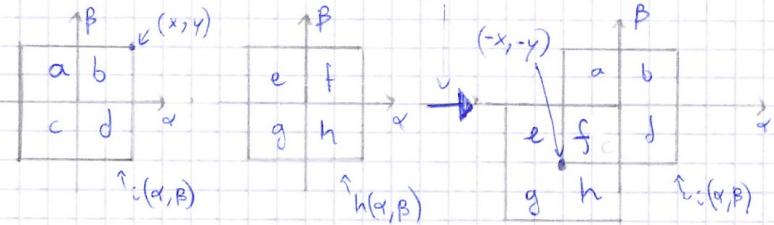
A Graphical view of Correlation

- $i(x,y) \circ h(x,y)$

correlation of i versus h

\Downarrow

- i is unchanged
- h is manipulated



i is shifted at (α, β)

i is not reflected through the origin, because I have $+\alpha$.

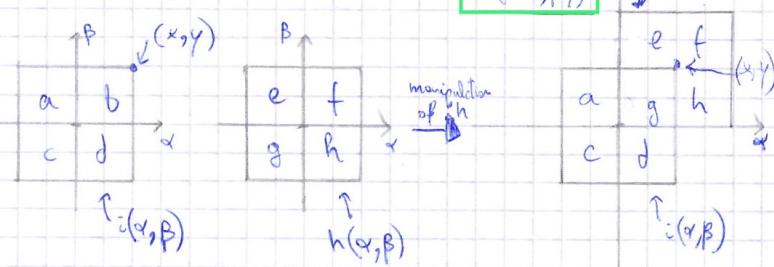
$$h(x+\alpha, y+\beta)$$

- $h(x,y) \circ i(x,y)$

correlation of h versus i

\Downarrow

- h is manipulated, such that what we'll multiply with respect to ' i ' is $h(\alpha-x, \beta-y)$.



\hookrightarrow In correlation:

- h is not reflected through the origin

- h is shifted as x and y are.

Figure 4.20

\Leftrightarrow • Correlation is about reflecting (flipping) and shifting

• Correlation h versus i is about shifting (without flipping)

\hookrightarrow you shift at the position in which you want to compute the output.

Figure 4.21

Convolution and Correlation of h versus i . \Rightarrow correlation of the filter versus the signal

• $h \rightarrow$ filter

• $i \rightarrow$ signal

→ there's a special case in which they're the same:

If the h function is even (= symmetric through the origin), then if you flip (reflect) it's just the same.

If h is an even function ($h(x,y) = h(-x,-y)$), then the convolution between h and i ($i * h * i$) is the same as the correlation of h versus i (\Leftrightarrow you can compute a convolution with a correlation):

$$i(x,y) * h(x,y) = h(x,y) * i(x,y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta) \cdot h(x-\alpha, y-\beta) d\alpha d\beta$$

↓ ↓

Commutative of convolution def. of convolution

$$\left[\begin{array}{l} \text{if } h \text{ is even, then} \\ h(x-\alpha, y-\beta) * \\ h(x-\alpha, \beta-y) \end{array} \right] = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha, \beta) \cdot h(x-\alpha, y-\beta) d\alpha d\beta$$

$$= h(x,y) \circ i(x,y)$$

• Correlation isn't commutative even if the function h is even function. To resp:

Figure 4.22

4.3 Bilateral Filter

Bilateral Filter

- it's a Non-linear filter

- it denoises images without introducing blurring

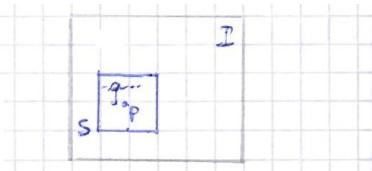


Figure 4.23

1D example of Gaussian Filter: -> it introduces noise when smoothing an edge:

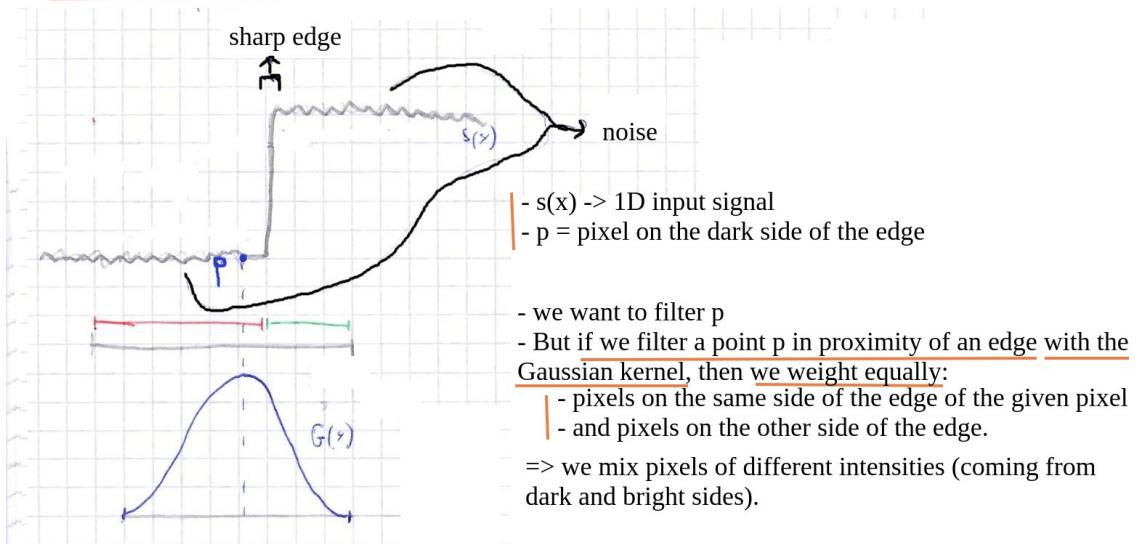
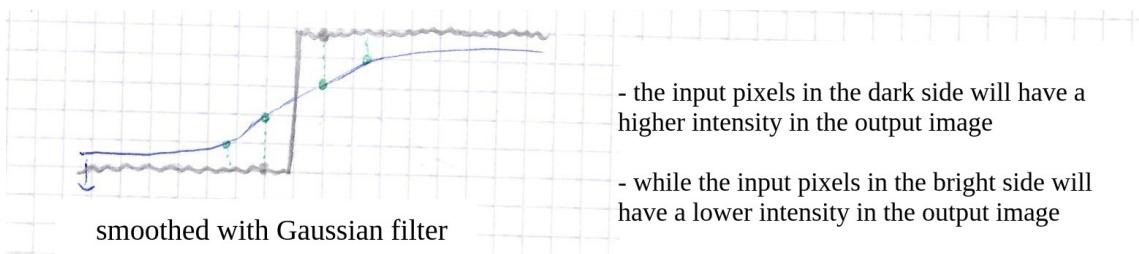


Figure 4.24



- OUTPUT SIGNAL
- the edge that was sharp is now smoothed => we see BLUR
 - with Gaussian filter (and with linear filters) we consider equally pixels on both sides of the edge => we have blur

Figure 4.25

Bilateral Filter:

- uses a different kernel at each pixel
- when we smooth a pixel p close to an edge:
 - it gives a high weight to pixels that are on the same side of the considered pixel p
 - and gives much less weight to pixels that are on the other side of the edge.
- IF we are filtering a pixel p close to an edge:

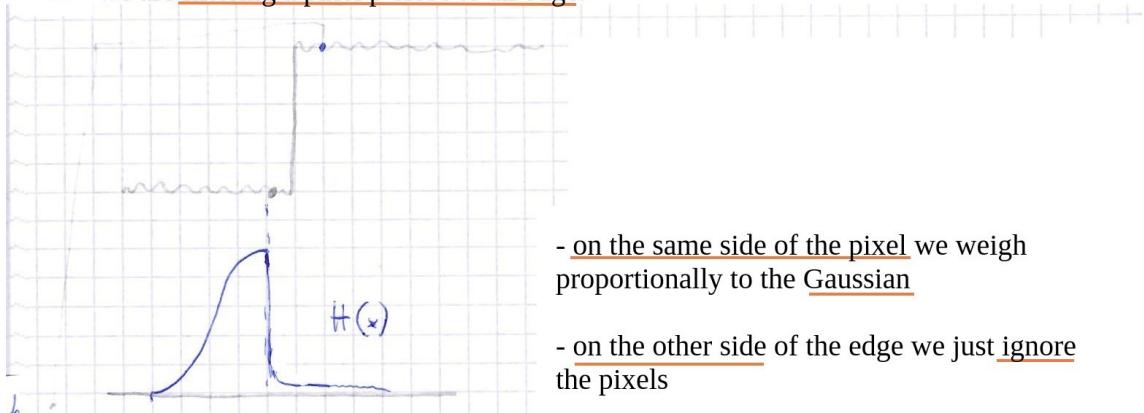


Figure 4.26

- IF we are filtering a pixel in a UNIFORM area:
 - then the filter will behave like a normal Gaussian

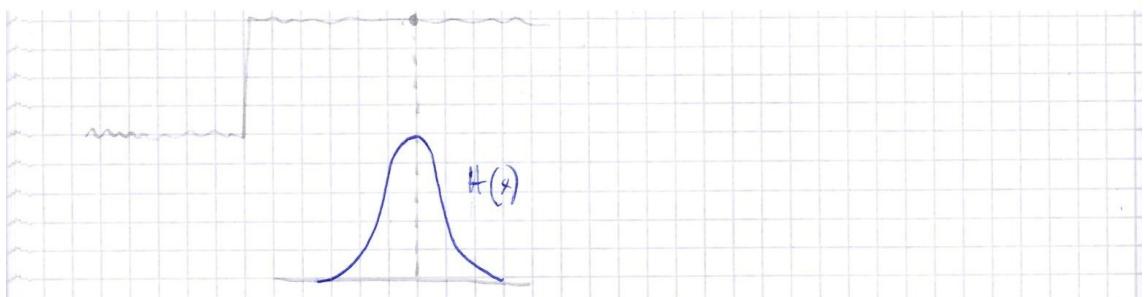


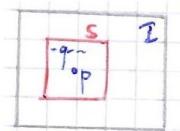
Figure 4.27

Implementation of the Bilateral Filter:

- we should weight pixels based on:

- the distance from the considered pixel p (< distance => > weight)
- and also based on the SIMILARITY with the considered pixel p (> similarity => > weight) (the similarity is computed as $|I(p) - I(q)|$)

=> a pixel will receive an high weight iff it has an high similarity AND is closer to the given one.



- p = considered pixel \rightarrow we want to compute its output
- q = running pixel in the supporting window s

- The output at pixel p is computed by weighting the intensities of pixels in the supporting window s, and summing them:

$$O(p) = \sum_{q \in S} H(p, q) \cdot I_q$$

weighting function H

$$H(p, q) = \frac{1}{W(p, q)} \cdot G_{\sigma_s}(d_s(p, q)) \cdot G_{\sigma_r}(d_r(I_p, I_q))$$

normalization factor
→ to have a unity gain (= all the coeff.s of the filter sum up to 1)

$$W(p, q) = \sum_{q \in S} G_{\sigma_s}(d_s(p, q)) \cdot G_{\sigma_r}(d_r(I_p, I_q))$$

Figure 4.28

- The weighting function H is the product of 2 Gaussian functions.

- the 1st Gauss has as argument the spatial distance between p (given pixel) and q (running pixel)

$$d_s(p, q) = \|p - q\|_2 = \sqrt{(u_p - u_q)^2 + (v_p - v_q)^2}$$

spatial distance = Euclidean dist between p and q

- the 2nd Gauss has as argument the similarity distance between p (given pixel) and q (running pixel)

$$d_r(I_p, I_q) = |I_p - I_q|$$

similarity distance = absolute value of the difference of the 2 intensities

- we know that a Gaussian function is high if its argument is small

=> the product of them (=the weighting function) is high if both arguments are small => if the pixel q is both similar and close to pixel p.

Figure 4.29

- => within a uniform area -> Bilateral filter behaves like a Gaussian:
 - indeed the similarity distance will be small => $G(\sigma_r)$ will be high
 - => it all depends on the 1st Gaussian $G(\sigma_s)$ => the one based on distance
 - while near an edge we have the NON-linear behaviour -> 2 Gaussians
 - Drawback of Bilateral filter -> is quite Slow, because we have to compute the weighting function H for all points.
- (pro of Bilateral filter wrt a Gaussian filter -> we can keep small details, while with the Gaussian filter we lose them if we set a scale of interest that is too small)

Figure 4.30

4.4 Non-local Means Filter

Non-local Mean Filter

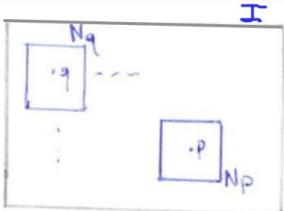
- it's a NON-linear filter that does smoothing without blurring the image.
- it has a very different idea from other filters:
 - other filters -> based on Locality -> smooth considering a local neighbourhood
 - Non-local Mean filter -> idea of looking at ALL pixels in the WHOLE image to smooth a given one.

How does it work?

- look across the whole image
- then give a weight to any other pixels based on:
 - the similarity of the region around the considered pixel and the regions around any other pixels in the image.

Figure 4.31

Formulation of Non-local Mean filter:



- p = considered pixel that we want to smooth
- q = any pixel in the image I
- N_p = neighborhood of p
- N_q = " " " q

- we will weight q (when smoothing p) according to how similar N_p and N_q are.
- the output at pixel p is: (the smoothed pixel p is)

$$O(p) = \sum_{q \in I} w(p, q) \cdot I(q)$$

we consider all pixels q
in the image

- the weighting function weights the intensity of pixels q based on the similarity of their neighborhoods with the neighborhood of p:

$$w(p, q) = \frac{1}{Z(p)} \cdot e^{-\frac{\|N_p - N_q\|_2^2}{h^2}}$$

↳ - this norm is small when
the 2 neighborhood are similar

↳ - if the norm is small
=> the value of w is large

↳ - h = bandwidth (it's a parameter)
↳ - Z(p) = normalization factor
↳ - Z(p) is useful to have a
unity gain = to normalize
all the coefficients in the
filter such that they sum up to 1.

=> we consider all pixels q in the image, each weighted by:
how similar its neighborhood is to the neighborhood of the given pixel p

Figure 4.32

Computational complexity of this filter $\rightarrow O(n^2)$,

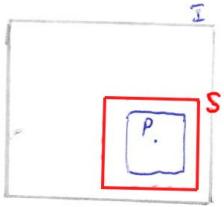
- because for each pixel we take a weighted sum of all the other pixels in the image

\Rightarrow Non-local Mean filter is very Slow,

- because the kernel is as large as the whole image.

Practical Implementation of Non-local Mean filter:

- In practice, when we smooth a pixel, we do NOT consider all pixels in the whole image
- but we consider a smaller window S (smaller wrt the entire image)



$N_p, N_q \rightarrow$ typically 7×7

$S \rightarrow 21 \times 21$

we still go far away from the local pixel p

\Rightarrow we still don't consider just a local neighborhood
But we do NOT consider the whole image.

- > size of S \Rightarrow better result, But > computational cost

- the reason to use S is just for a sake of efficiency

- it's still a NON-local approach because, while smoothing, you consider pixels that are far away from the pixel that is being smoothed.

Figure 4.33

Chapter 5

Image Segmentation

5.1 Colour-based Segmentation

- Goal = a way to perform background/foreground segmentation based on color.
 - Used f.e. when the background has a very different color from the objects (f.e. green/blue background -> food industry)
 - perform segmentation just by computing a distance between the given color and the reference color:

$$\forall p \in I : \left\{ \begin{array}{l} d(I(p), \mu) \leq T \Rightarrow O(p) = B \\ d(I(p), \mu) > T \Rightarrow O(p) = F \end{array} \right.$$

background

foreground

- μ = reference color = typically the background color
 - $O(p)$ = output of segmentation of pixel p
 - T = threshold
 - $d(\cdot)$ = distance

Figure 5.1

- How to ESTIMATE the REFERENCE COLOR:
 - take pixels from a set of background images (train images)
 - and estimate the reference color as the mean of these samples:

$$\mu = \begin{bmatrix} \mu_r \\ \mu_g \\ \mu_b \end{bmatrix} = \frac{1}{N} \sum_{k=1}^N I(p_k)$$

• $I(p_k), k=1, \dots, N \rightarrow$ samples
 • μ = reference colour estimated from training images

Figure 5.2

- We can take the EUCLIDEAN distance to perform segmentation (but not the best):

$$d = \sqrt{(I_r(p) - \mu_r)^2 + (I_g(p) - \mu_g)^2 + (I_b(p) - \mu_b)^2}$$

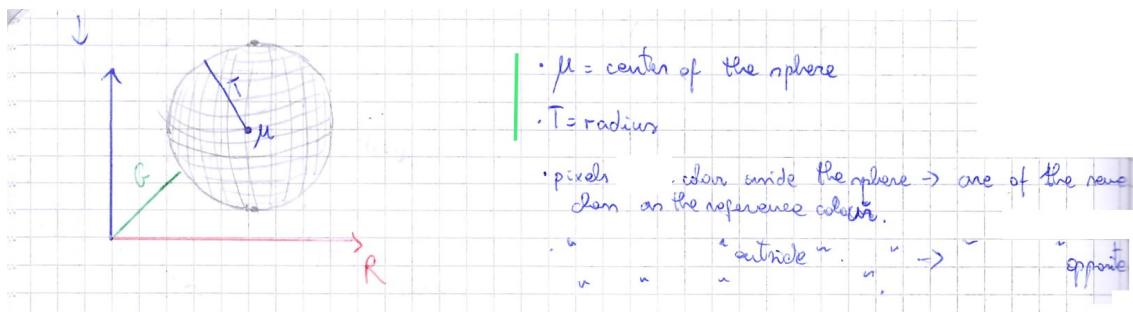
but we can rewrite this as an operation between two vectors.

$$d = \|I(p) - \mu\| = \left[(I(p) - \mu)^T \cdot (I(p) - \mu) \right]^{\frac{1}{2}}$$

dot product between the transpose of the difference as vector and the difference itself as vector

Figure 5.3

- With Euclidean distance we define an isotropic decision surface \rightarrow a SPHERE:



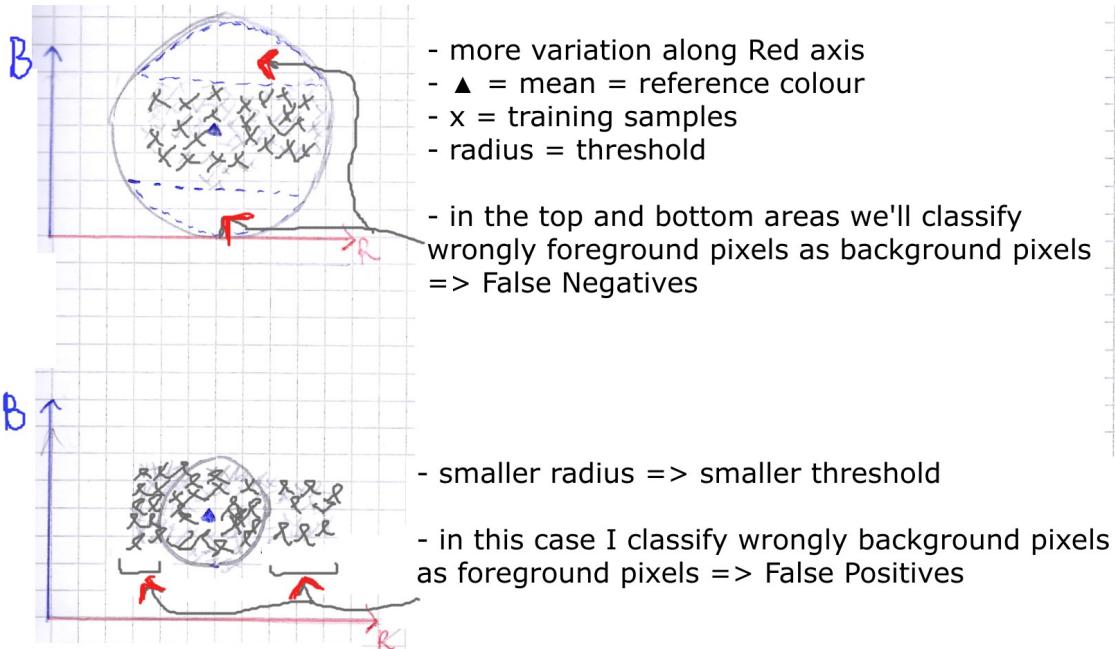
at test time (if μ (= reference color) is BACKGROUND):

- if $\text{Euclidean_dist}(I(p), \mu) \leq T \rightarrow p$ classified as BACKGROUND
- if $\text{Euclidean_dist}(I(p), \mu) > T \rightarrow p$ classified as FOREGROUND

Figure 5.4

- Problem with Euclidean distance:

- If the samples are NOT distributed isotropically \Rightarrow If I have more variation in 1 axis than the other:



=> Due to asymmetrical distribution of the variations in color samples => the Euclidean distance is NOT so precise

Figure 5.5

- => we use the MAHALANOBIS DISTANCE:

- We take into account:
 - * we compute the COVARIANCE matrix:
 - it captures the correlations between the color channels:

It contains the covariances of the 3 channels

$$\Sigma = \begin{pmatrix} \sigma_{rr}^2 & \sigma_{rg}^2 & \sigma_{rb}^2 \\ \sigma_{gr}^2 & \sigma_{gg}^2 & \sigma_{gb}^2 \\ \sigma_{br}^2 & \sigma_{bg}^2 & \sigma_{bb}^2 \end{pmatrix}$$

3×3

covariance formula

$$\cdot \sigma_{ij,\bar{i}}^2 = \frac{1}{N} \sum_{k=1}^N (I_i(p_k) - \mu_i) \cdot (I_{\bar{j}}(p_k) - \mu_{\bar{j}})$$

$i, \bar{i} \in \{r, g, b\}$

3 channels \rightarrow we compute the covariance of the 3 channels.

σ_{rg}^2 = covariance red, green = how much red and green are correlated. It could be a positive or a negative correlation.

Figure 5.6

- => the MAHALANOBIS Distance is:

$$d_M(\mathbf{I}(p), \mu) = ((\mathbf{I}(p) - \mu)^T \cdot \Sigma^{-1} \cdot (\mathbf{I}(p) - \mu))^{\frac{1}{2}}$$

is the vector form of the Euclidean distance, but with the inverse of the covariance matrix

Figure 5.7

- To understand the difference between Eucl dist and Mahal dist:

– assume the covariance matrix is diagonal (we won't lose generality):

$$\Sigma = \begin{pmatrix} \sigma_{rr}^2 & 0 & 0 \\ 0 & \sigma_{gg}^2 & 0 \\ 0 & 0 & \sigma_{bb}^2 \end{pmatrix}$$

this means that there's no correlation
between two different colour channels.

Then, let's take its inverse:

$$\Sigma^{-1} = \begin{pmatrix} \frac{1}{\sigma_{rr}^2} & 0 & 0 \\ 0 & \frac{1}{\sigma_{gg}^2} & 0 \\ 0 & 0 & \frac{1}{\sigma_{bb}^2} \end{pmatrix}$$

and plug it into the Mahalanobis distance:

Figure 5.8

$$d_M(I(p), \mu) = \left((I(p) - \mu)^T \cdot \Sigma^{-1} \cdot (I(p) - \mu) \right)^{\frac{1}{2}}$$

Σ^{-1}

$$\begin{pmatrix} \frac{1}{\sigma^2_{rr}} & 0 & 0 \\ 0 & \frac{1}{\sigma^2_{gg}} & 0 \\ 0 & 0 & \frac{1}{\sigma^2_{bb}} \end{pmatrix} \cdot \begin{bmatrix} I_r(p) - \mu_r \\ I_g(p) - \mu_g \\ I_b(p) - \mu_b \end{bmatrix} = \begin{bmatrix} (I_r(p) - \mu_r)/\sigma^2_{rr} \\ (I_g(p) - \mu_g)/\sigma^2_{gg} \\ (I_b(p) - \mu_b)/\sigma^2_{bb} \end{bmatrix}$$

$I(p) - \mu$

$$\begin{bmatrix} I_r - \mu_r & I_g - \mu_g & I_b - \mu_b \end{bmatrix} \cdot \begin{bmatrix} (I_r(p) - \mu_r)/\sigma^2_{rr} \\ (I_g(p) - \mu_g)/\sigma^2_{gg} \\ (I_b(p) - \mu_b)/\sigma^2_{bb} \end{bmatrix}$$

we get the sum of
the squares of the
differences, but
divided by the variance.

Figure 5.9

In this formulation (wrt the Euclidean dist) we introduce a DIVISION, a SCALING given by the VARIANCES:

↓

$$d_M(I(p), \mu) = \sqrt{\frac{(I_r(p) - \mu_r)^2}{\sigma^2_{rr}} + \frac{(I_g(p) - \mu_g)^2}{\sigma^2_{gg}} + \frac{(I_b(p) - \mu_b)^2}{\sigma^2_{bb}}}$$

Mahalanobis distance

$\frac{1}{2}$

the 3 variances at the denominators will weight the observed differences according to the observed variances

Figure 5.10

- $\Rightarrow >$ variance $\Rightarrow <$ weight to an observed change
- Decision surface with Mahalanobis dist:
 - Ellipsoid with axes proportional to the variances:

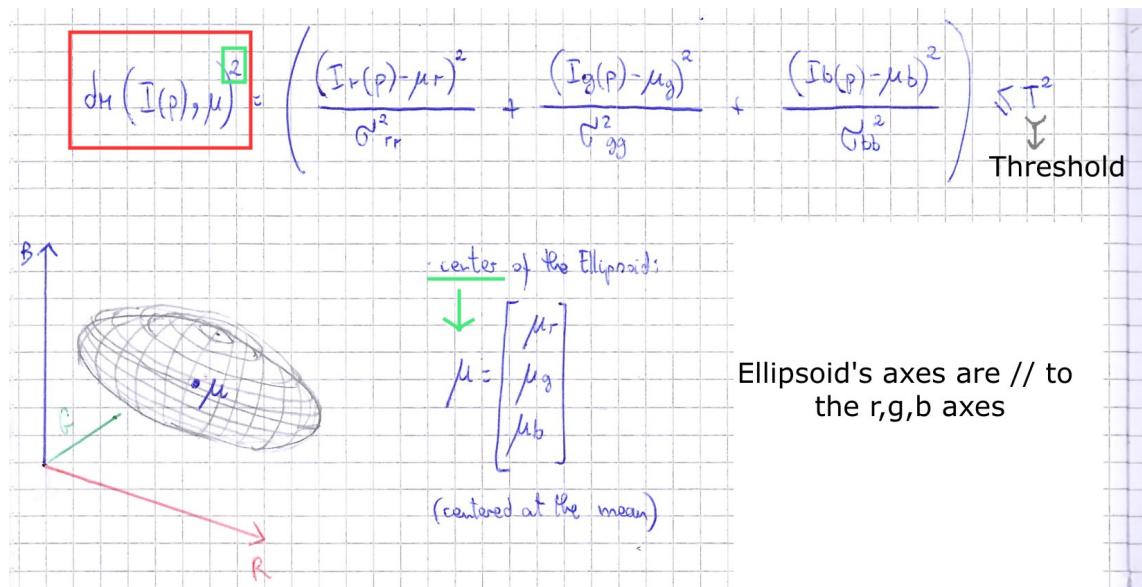


Figure 5.11

- What if the covariance matrix is NOT diagonal? (only if he asks)

What if the covariance matrix is not diagonal?

↳ All these considerations still hold, because the covariance matrix can always be diagonalized by a rotation of the reference system.

↳ there's always a basis of the space in which if we rotate the data into that basis, then the data will show up as uncorrelated (= the covariance matrix will be diagonal).

↳ This can be achieved by the Eigenvalue Decomposition (EVD) of any real and symmetric matrices: (the covariance matrix Σ indeed is real and symmetric):

Figure 5.12

$$\Sigma = R \cdot D \cdot R^T : R = (e_1 \ e_2 \ e_3) , \quad D = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix}$$

↳ This is like the SVD (singular value decomposition), but the SVD applies also to non-square matrices, while EVD applies to square matrices.

• EVD:

↳ the covariance matrix (Σ) can always be expressed as the product of 3 matrices:

- R → orthogonal matrix → it's a rotation matrix → columns = eigenvectors of the matrix Σ .
- D → diagonal matrix

$\left. \begin{matrix} \text{elements along the diagonal} \\ \text{of the diagonal} \end{matrix} \right\} = \left[\begin{matrix} \text{eigenvalues of the matrix} \\ \Sigma \end{matrix} \right]$

↳ given the eigenvalues and eigenvectors of Σ , then we can decompose it as:

$$\Sigma = R \cdot D \cdot R^T.$$

• And there's always a transformation (which is a rotation) which can be applied to the original covariance matrix and that turns the covariance matrix into a diagonal covariance matrix. This transformation is a rotation through R^T .

Figure 5.13

↳ We can always turn the original data samples into new data samples such that the covariance matrix will be diagonal.

↓
The Mahalanobis distance doesn't change with a rotation ⇒ the distance is the same also with the diagonal covariance matrix.

Figure 5.14

11

- the new diagonal covariance matrix is exactly D:

$$D = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix}$$

- e_i = orthonormal eigenvectors of Σ

- λ_i = eigenvalues of Σ

- R^T = rotation matrix that transform the data into a new coordinate system having axes aligned to the eigenvectors

- D = new covariance matrix \rightarrow diagonal. $\Rightarrow \lambda_i$ are the variances in the rotated axes.

but we don't need it. We just have to compute the Mahalanobis distance based on its definition (\Rightarrow with a non-diagonal covariance matrix) and then we know that by doing that we're defining a decision surface which is an ellipsoid. \Rightarrow Assuming the covariance matrix to be diagonal doesn't add generality, it's always like that.

Figure 5.15

Chapter 6

Binary Morphology

Chapter 7

Blob Analysis

Usually we have different steps before doing Blob Analysis. An example of pipeline is the following:

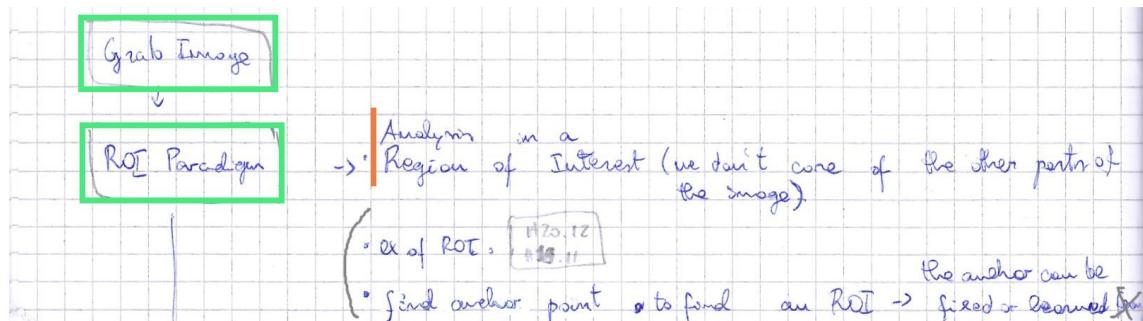


Figure 7.1

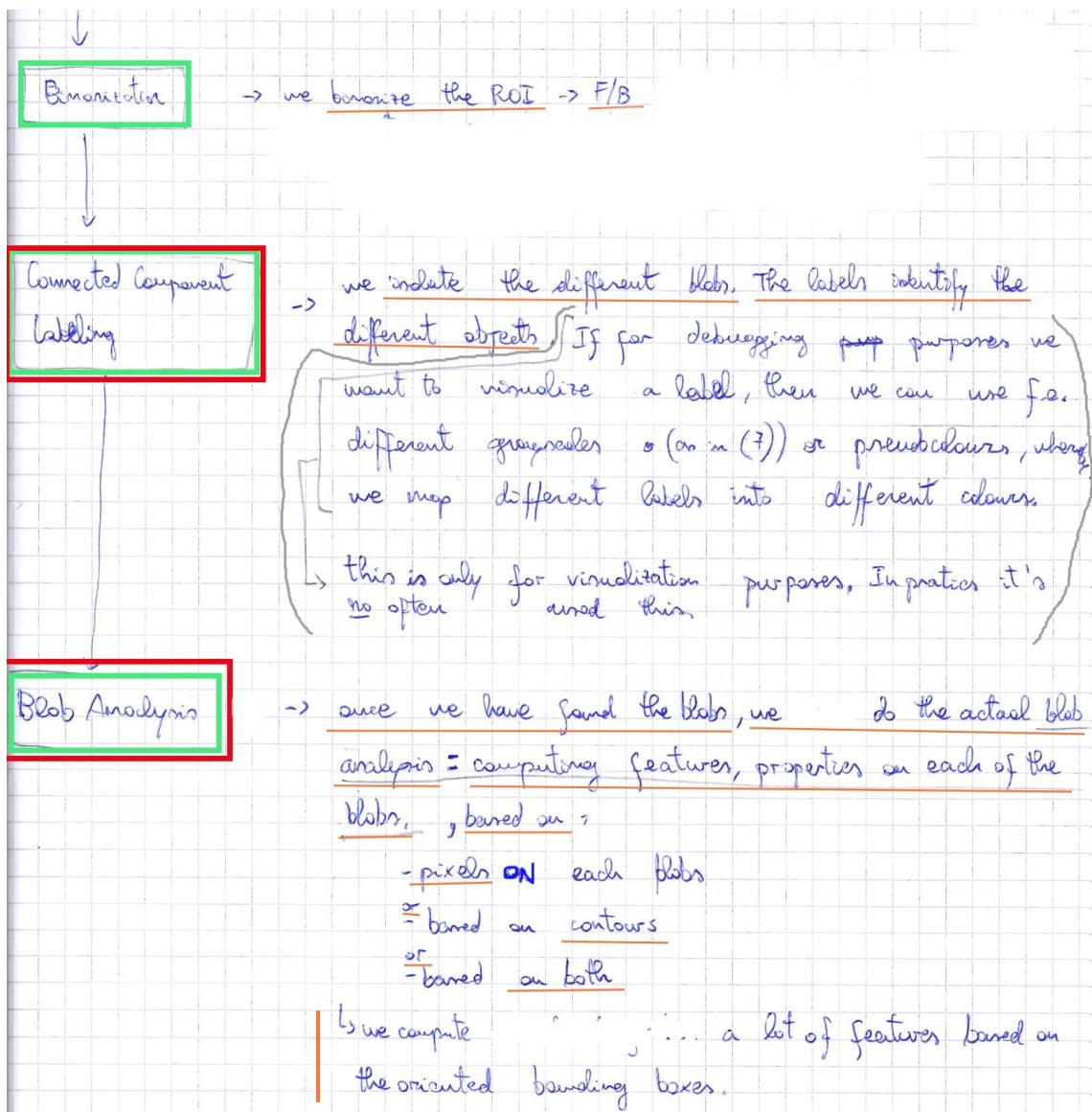


Figure 7.2

7.1 Connected Components

Connected Components of a Binary Image

↳ we need other 3 definitions in order to define the connected components:

1) Def of path:

- A path of length n from pixel p to pixel q is a sequence (\Rightarrow there is an order) of pixels $p = p_1, p_2, \dots, p_n = q$ (from $p_1 = p$ to $p_n = q$) such that p_i and p_{i+1} (any two successive points) are neighbours, \Rightarrow they have a distance equal to 1 (according to the chosen distance).



Manhattan Distance \rightarrow D4

Chessboard Distance \rightarrow D8

2) Def of Connected region:

- A set of pixels R is said to be a connected region (\Rightarrow is connected) if for any two pixels p, q in R there exists always a path in R between p and q . (\Rightarrow you can get always from p to q while staying in R). Because this definition relies on the definition of path, and the defint.

Figure 7.3

of path relies on the definition of distance, then similarly a region R is said to be:

- 4-connected \rightarrow if we use D_4 as the distance
or
- 8-connected \rightarrow " " D_8 "

③ Def of Connected Foreground (Background) Region

* A set of pixels is said to be a connected foreground (background) region if it is a connected region and all points in the region are foreground (background) pixels only (\Rightarrow if it includes foreg. (backgr.) pixels only).

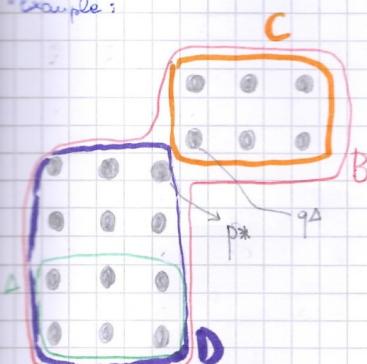
• Definition of a Connected Component: (connected components = blobs)

* A connected component of a binary image is a MAXIMAL connected foreground region

\hookrightarrow MAXIMAL \rightarrow that's the maximal size you can get for that region

Figure 7.4

- Example:



- \bullet = foreground pixels
- CC = connected component = maximal connected foreground region
MEFR

CFR = Connected Foreground Region

CC = Connected Component

- the net of pixels A is a MAXIMAL CFR
- the net of pixels B is a CC?
- if we assume the distance to be D_4 , then this net is not a CC, because p^* and q_5 pixels are not connected
- CFR , but it's not a CC, because it's not D_4 = Manhattan Distance
 D_8 = Chessboard Distance
- It depends;

Figure 7.5

- if we assume the distance to be D_8 , then the set B is a CC, because then p^* and q^* are connected, because it can move diagonally.
- But if we assume $D=D_4$, then the previous image has 2 CC:
 - **C** and **D**
 - It depends on the type of the connectivity we're using:
 - with a 4-connectivity \rightarrow the labeling algorithm marks differently as different blobs the pixels in **C** and **D**.
(different connective components)
 - " 8- " " \rightarrow " " as
the same blob all the pixels in the image
 $\searrow \text{B}$
- In this case:
 - LABELING
 - \rightarrow 2 CC with D_4
 - \rightarrow 1 CC " D_8

Figure 7.6

- ↳ normalization:
- ↳ labeling output displayed as a grey-scale image
 - " " " (pseudo) colours.

Figure 7.7

7.2 Labeling problem (two scan algorithm)

2-scan algorithm

- Goal: labeling the blobs found with image segmentation
=> label foreground pixels
- we scan the image 2 times
- After the 1st scan:
 - foreground pixels will take temporary labels based on the labels given to already visited neighbours
 - already visited neighbours depend on:
 - the chosen distance (Manhattan or Chessboard)
 - the scan order (f.e. top-down; left-right)

Figure 7.8

=> after the 1st scan I can end up with a single blob labeled with different labels.
Example:

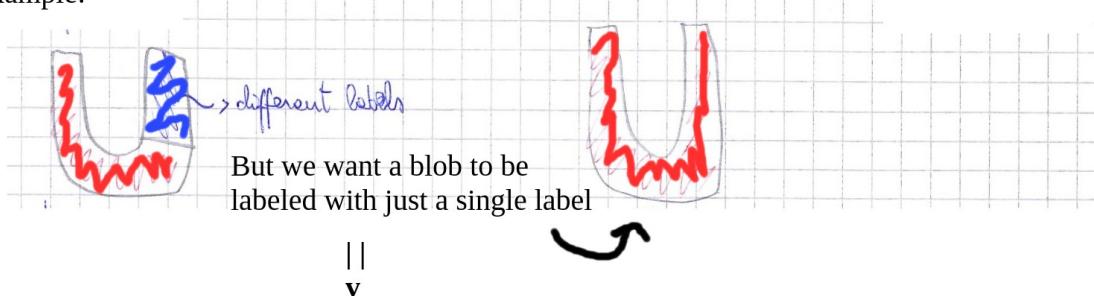


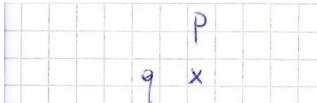
Figure 7.9

- we record all equivalences
- then, we find all equivalent classes (= sets of equivalent labels)
- then, we relabel each blob (each connected component) with just a single label taken from the equivalent class of labels

Figure 7.10

• First scan:

- if I use:
 - distance = Manhattan Distance (D4)
 - scan order = top-down; left-right
- I have a foreground pixel x and its already visited neighbors p and q



=> I will label x based on the labels of already visited neighbors p and q:
=> 5 cases:

$$1. l_q = l_p = B \rightarrow l_x = \text{Newlabel}$$

Figure 7.11

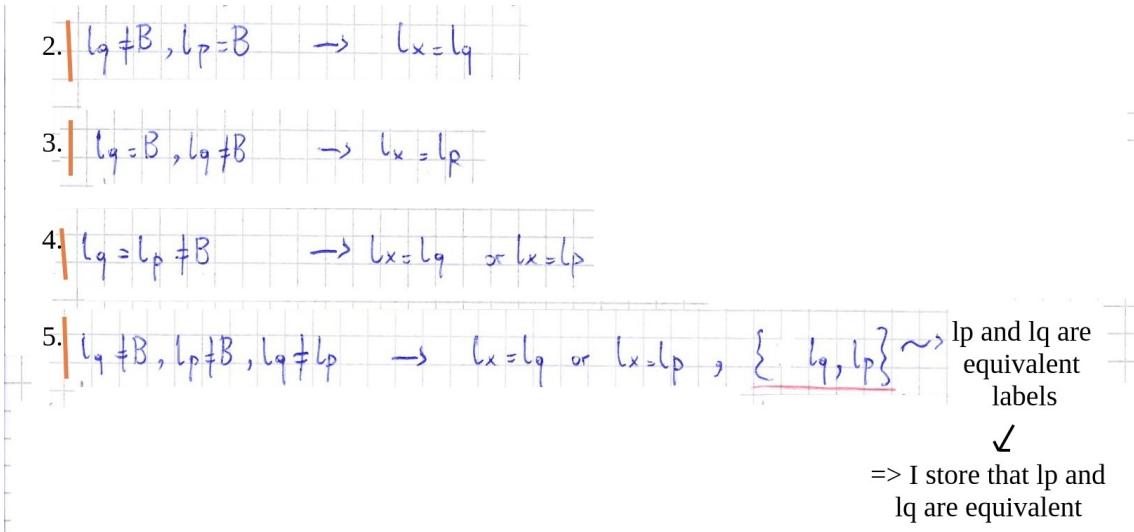


Figure 7.12

Example:

1	2	3				
1	2	3				
1	2	3				
1	2	3				
1	1	1	1	1	1	1

↓ ↓

conflict conflict

=> store => store

equivalence equivalence

$\{1, 2\}$ $\{1, 3\}$

Figure 7.13

=> After the 1st scan I have these temporary equivalences:

$$\{1, 2\}, \{1, 3\}$$

Then, I store the equivalences in equivalent classes (\Rightarrow in set of equivalent labels):
(in this ex. I get 1 equiv. class)

$$\{1, 2, 3\}$$

Then, in the 2nd scan:

- we pick a label from each equivalent class and relabel all blobs
- \Rightarrow we substitute temporary labels with equivalent labels

=> Final result of the example:

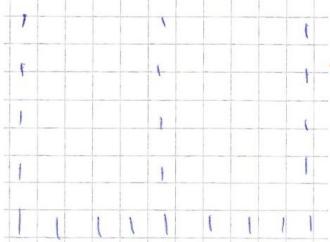


Figure 7.14

7.2.1 Implementation

Handling Equivalences:

We store the temporary equivalences in a NxN matrix
(N = num of temporary labels)

if i, j are
equivalent labels

$$B = \begin{bmatrix} 1 & \dots & i & \dots & N \\ i & \dots & \dots & \dots & \dots \\ \vdots & & \ddots & \ddots & \vdots \\ \vdots & & \vdots & \ddots & \vdots \\ N & & \vdots & \vdots & 1 \end{bmatrix} \Rightarrow B[i, j] = \begin{cases} 1, & \text{if } \{i, j\} \\ \emptyset, & \text{otherwise} \end{cases}$$

Figure 7.15

The B matrix is symmetric (because if i is equivalent to $j \Rightarrow j$ is equivalent to i):

$$B[i, j] = B[j, i] \quad \forall i, j = 1 \dots N \quad , N = \text{num of temporary labels}$$

Figure 7.16

=> After the 1st scan:

- we want to turn temporary labels into equivalent classes

=> we scan the matrix column by column and:

- if we find an equivalence (a 1) between i and j, then we take the OR between row j and row i and assign it to row i ($i = i \text{ OR } j$):

$$B[i, k] = B[i, k] \text{ OR } B[j, k] \quad \forall k=1, \dots, N$$

(N = num of temporary labels)

Figure 7.17

ex of row i equivalent to row j

$\Rightarrow i = i \text{ OR } j$

i	1	1	1	1	1
:	↑				
j	1	1	1	1	0

Figure 7.18

Example:

- assume we found the equivalences $\{1,2\}$ and $\{2,3\}$ during 1st scan:

- $N = 4 \Rightarrow$ num of temporary labels

	1	2	3	4
1	1	1	1	
2	1	1	1	
3		1	1	
4			1	

B

I scan the matrix column by column, and when I find an equivalence $\{i, j\}$, then i do:
 $i = i \text{ OR } j$

\Rightarrow I find equivalent classes.

row3 = row3 or row2

row1 = row1 or row2

	1	2	3	4
1	1	1	1	
2	1	1	1	
3	1	1	1	
4			1	

B

Figure 7.19

Now we replace each equivalence class (I have just 1 equiv. class) with a single label:

- we can use an LUT, with num of entries = N (N = num of temporary labels)
- we initialize the LUT:

$$\text{LUT} = \begin{bmatrix} 1 \\ \vdots \\ i \\ \vdots \\ N \end{bmatrix} \xrightarrow{\quad} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

then I scan the B matrix and:
- if I find $B[i, j] = 1$; and $i \neq j$:
 \Rightarrow I can pick one of the two (i or j) and do: $\Rightarrow \Rightarrow$

$$\text{LUT} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \xrightarrow{\quad} \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \end{bmatrix}$$

$\text{LUT}[j] = i \Rightarrow$ I do not consider j anymore

Finally, I relabel all blobs (all temporary labels) by accessing this LUT
 \Rightarrow F.e. in the 2nd scan when I find a 2 or a 3 \Rightarrow I will relabel them as 1

Figure 7.20

Chapter 8

Edge Detection

8.1 LoG Edge detector

Zero-Crossings of the 2nd derivative:

To find edges in a 1D signal:

- I locate them in min or max of the 1st derivative
- or I can also look for zero-crossings of the 2nd derivative of the signal

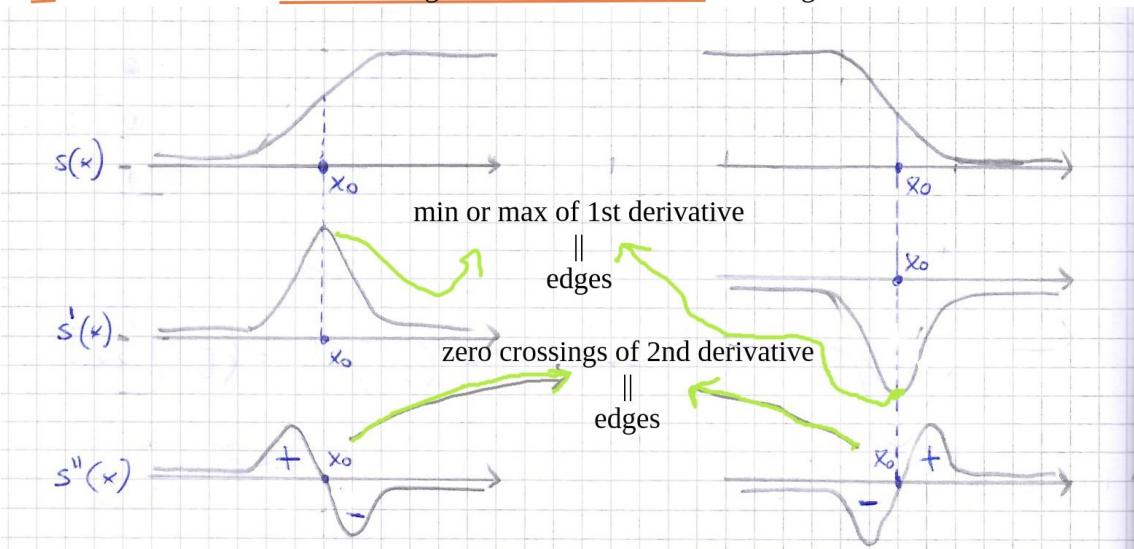


Figure 8.1

From 1D signals to 2D signals (images):

In a 2D signal you can compute derivatives along all directions:

- => I should be finding zero-crossings of 1 directional derivative
- => I choose the direction of the gradient
- => To find edges -> I should find the zero-crossings of the 2nd derivative along the gradient's direction

Figure 8.2

- How do we compute the 2nd derivative along the gradient?

- assume n is a unit vector that represents the gradient's direction
- n is the gradient normalized by its magnitude:

$$\vec{n} = \frac{\nabla I}{|\nabla I|}$$

- now we need to compute the 2nd derivative of the image along n (along the gradient's direction):

$$\frac{\delta^2 I}{\delta n^2}$$

- the 2nd derivative is just the derivative of the 1st derivative
- and we know that the 1st derivative along gradient's direction is the magnitude of the gradient
- => the 2nd derivative becomes:

- the derivative along n (along the gradient's direction) of the magnitude of the gradient:

$$\frac{\delta}{\delta n} (|\nabla I|)$$

Figure 8.3

- we know that the derivative of a function f along a direction n is equal to:

- the gradient of that function f dot product the unit vector of the direction:

$$\frac{\delta f}{\delta n} = \nabla f \cdot \vec{n}$$

- we consider the function f as the magnitude of the gradient ($|\nabla I|$) and we apply this rule:

$$\frac{\delta}{\delta n} (|\nabla I|) = \nabla (|\nabla I|) \cdot \vec{n}$$

- => to take the derivative of the magnitude of the gradient along the gradient direction:

1. we just take the gradient of the magnitude of the gradient

2. the gradient direction n

Figure 8.4

=> the 2nd derivative along the gradient is:

$$\nabla(|\nabla I|) \cdot \vec{n} = \frac{I_x^2 \cdot I_{xx} + 2 I_x \cdot I_y \cdot I_{xy} + I_y^2 \cdot I_{yy}}{I_x^2 + I_y^2}$$

- that's a very complicated function that requires a lot of computations:

- I need to compute 1st derivatives (I_x, I_y)
- and also 2nd derivatives (I_{xx}, I_{yy})

Figure 8.5

The LAPLACIAN

=> instead, we prefer to use the Laplacian:

- => this Laplacian substitutes, in a more efficient way, the 2nd derivative along the gradient
- the Laplacian is not a directional operator, you don't compute the Laplacian along a certain direction
- it is a 2nd order differential operator, which is the sum of the 2nd derivatives along the 2 axes

$$\nabla^2 I(x, y) = \frac{\delta^2 I(x, y)}{\delta x^2} + \frac{\delta^2 I(x, y)}{\delta y^2} = I_{xx} + I_{yy}$$

- we get a scalar

Figure 8.6

How to compute the Laplacian?

- we want to compute it in a discrete way
- => we compute 2nd derivatives through differences:
- => we use:
 - Forward differences -> to approximate 1st derivatives
 - Backward differences -> to approximate 2nd derivatives

Figure 8.7

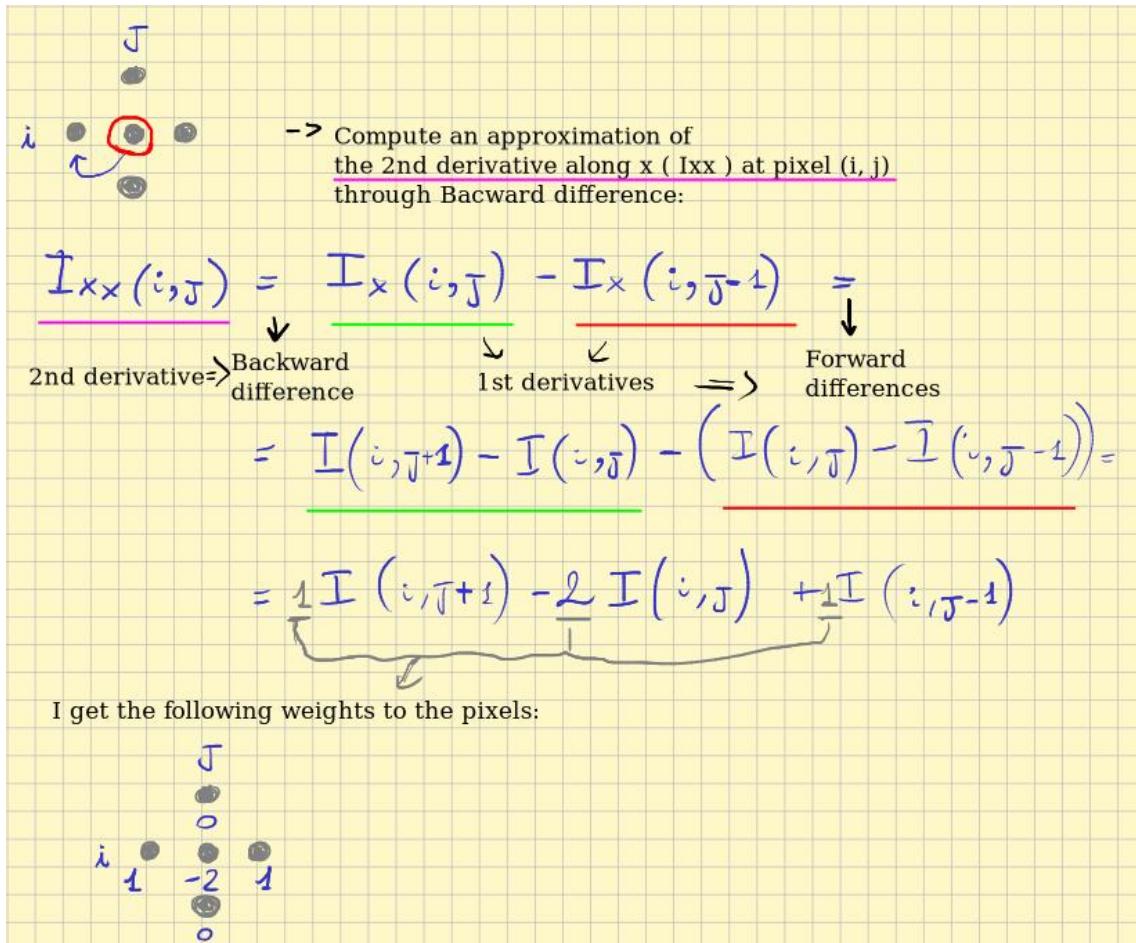


Figure 8.8

Then I compute the approximation of the 2nd derivative along y (Iyy):

$$I_{yy} = I_y(i, \bar{J}) - I_y(i-1, \bar{J}) = I(i-1, \bar{J}) - 2I(i, \bar{J}) + I(i+1, \bar{J})$$

For the definition of the Laplacian I sum the two 2nd derivatives together:

$$\nabla^2 I(x,y) = I_{xx} + I_{yy}$$

=>I sum the 2 kernels and the final kernel I get is:

$$\nabla^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

=> To detect edges:

- we need to correlate the image with this kernel
 - and then find the zero-crossings (edges) in the image

Figure 8.9

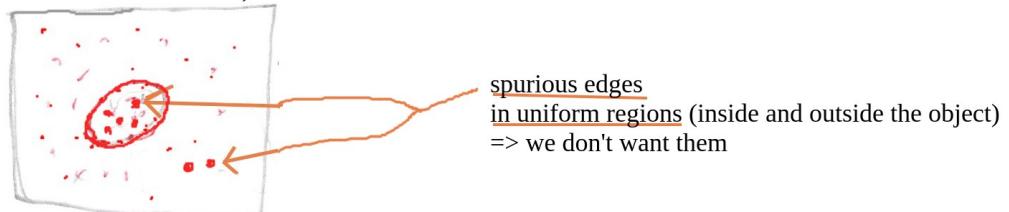
PROBLEM with NOISE:

- a 1st derivative amplifies noise
 - and a 2nd derivative does it even worse

=> we get more and more noise

- If I don't smooth noise, then I'll get:

- edge points of the contour of my object (good)
 - BUT I will have also edge points in the background and inside the object (NOT good, I do NOT want them)



=> to avoid these spurious edge points:

- I need to smooth the image, and in the LoG I smooth it with the Gaussian Filter. =>:

1. First I apply a Gaussian Filter to the image
 2. Then apply the Laplacian
 3. Extraction of zero-crossings (edges)

LOG (Laplacian of Gaussian)

Figure 8.10

1. How to perform Gaussian smoothing? Which sigma?

- σ^2 \Rightarrow to extract edges related to main scene structures
 - $< \sigma^2$ \Rightarrow to capture also small size details

Setting G is as nothing the scale - of - interest.

Figure 8.11

3) How to perform the Zero-Crossing search?

- we have computed the LoG in the image
 - => we now look for the zero-crossings
 - (we don't look for $\text{LoG} = 0$, but we look for zero-crossings, because $\text{LoG} = 0$ means that we're in a uniform area, where the 1st and 2nd derivatives are = to 0)
 - and in a digitalized image we can't find exactly zero-crossing pixels, due to discretization

Figure 8.12

Solution:

=> you search for **SIGN CHANGES** rather than searching for zero-crossings

- we look for sign changes along rows and find the 1st set of edges
- and then we look for sign changes along columns

- what do we do to localize sign changes?

- F.e. if I have a pixel with $\text{LoG} = -10$ and next pixel with $\text{LoG} = 20$

=> I don't pick both because I don't want thick edges

=> I just pick the pixel whose LoG is closer to zero

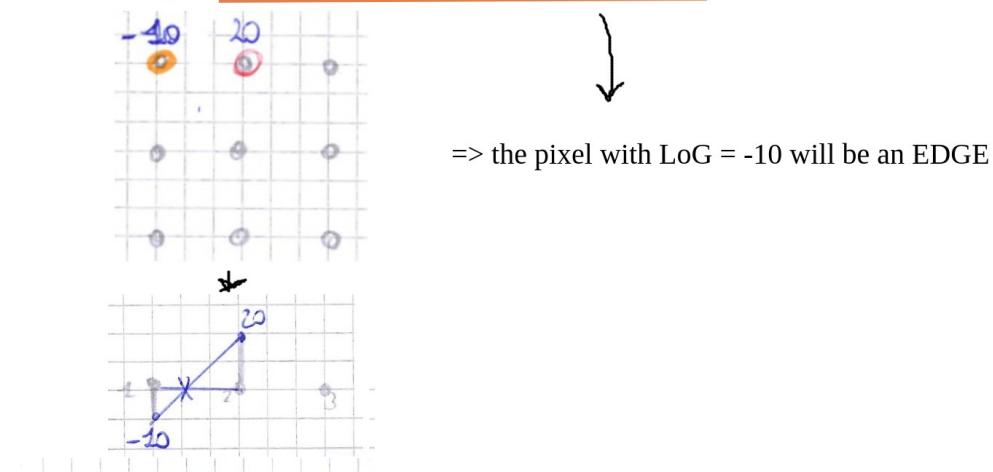


Figure 8.13

Localizing edges with sub pixel accuracy

you can split the interval between 2 pixels in f.e. in 4 steps and return the num of steps after the first pixel

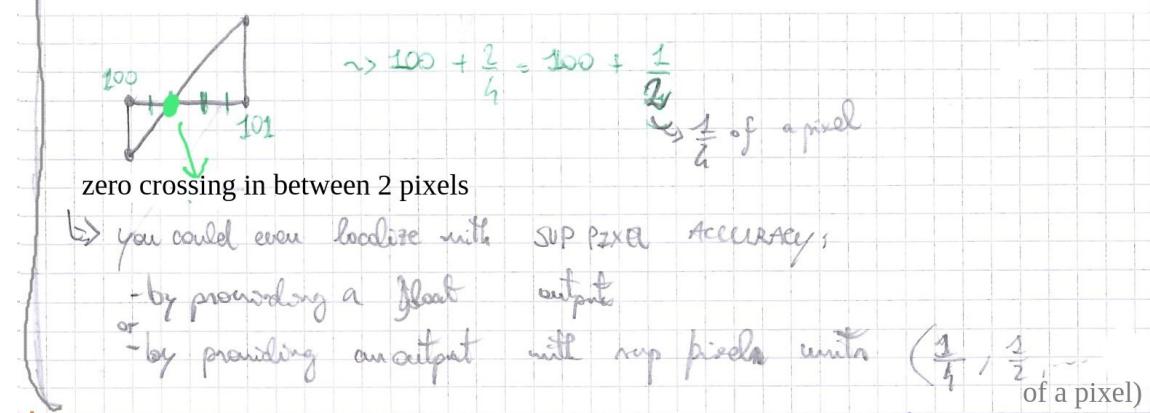


Figure 8.14

Finally, if we still get noisy edges, we can apply a final thresholding based on:

- the SLOPE of the Laplacian, which is given by:
 - the sum of the absolute values of the LoG in the two pixels



↳ > the Slope of the Laplacian => stronger is the edge

Example of use of this thresholding step:

- when smooth a lot to avoid noisy edges -> we lose also small details
=> instead:
 - I smooth less (to keep small details)
 - and then I try to get rid of noisy edges by applying this final thresholding step based on the slope of the Laplacian

Figure 8.15

2

How to actually compute the LoG?

$$\nabla^2 \tilde{I}(x, y) = \nabla^2 (I(x, y) * G(x, y)) = I(x, y) * \nabla^2 G(x, y) =$$

the Laplacian of a Gaussian filtered image

↓ differentiation commutes with Convolution
 => equivalently we can compute
 the convolution
 with the Laplacian of the Gaussian

$$= I(x, y) * (G''(x) \cdot G(y) + G''(y) \cdot G(x)) =$$

I apply Separability:

=> a 2D Gaussian Function is the product of a 1D Gauss Function along x and a 1D Gauss function along y:

Figure 8.16

Applying Separability:

$G(x,y) = G(x) \cdot G(y)$ -> a 2D Gauss function is the product of 2 1D Gauss functions

$$\frac{\delta^2 G(x,y)}{\delta x^2} = G''(x) \cdot G(y); \quad ; \quad \frac{\delta^2 G(x,y)}{\delta y^2} = G(x) \cdot G''(y)$$

2nd derivative of $G(x,y)$
wrt to x axis

2nd derivative pf $G(x)$
times $G(y)$

and the LoG is equal
to the sum of these two

Figure 8.17

Distributive property of Convolution wrt the sum:

=> The convolution of $I(x, y)$ wrt the sum of the 2 functions is the sum of the convolutions

$$= I(x,y) * (G^u(x) \cdot G(y)) + I(x,y) * (G^u(y) \cdot G(x)) =$$

Figure 8.18

$$= \left(I(x, y) * G^u(x) \right) * G(y) + \left(I(x, y) * G^u(y) \right) * G(x)$$

Distributivity:

- convolution between a product of 2 functions is the CHAIN of the convolutions

Now we compute the LoG with 4 1D Convolutions, rather than 1 2D Convolution

=> much faster

=> complexity:

- from d^2 to $4d$ (\Rightarrow better), with d = size of a kernel

Figure 8.19

8.2 Canny's Edge detector



Canny defined 3 criteria to assess if an edge detector is good or not:

1. Good Detection -> - the filter should be robust to noise => it should extract correctly edges in noisy images.
- Canny proved that the optimal denoising filter for edge detection is the Gaussian filter.
2. Good Localization -> - the edge detector should be precise
- it should detect edge points very precisely wrt the "true" edge points
- the "true" edge points are the steps in the noisy step signal.
3. One response to One edge -> - we want only one response at a single step
=> If there's a single step edge, then a single point must be found.

Figure 8.20

Canny discovered that if we want to find optimally edges according to these 3 criteria, then:

1. we need to compute a convolution between the noisy step and the 1st derivative of the Gaussian filter => a so called First Order Gaussian Derivative
2. And then locate the extrema of this convolution

How to apply this in 2D settings?

- I want to compute the 1st derivative of a filtered image by a Gaussian filter.
- in 2D signals I can compute derivatives along all directions
=> I should choose 1 directional derivative
- I choose the direction of the gradient, because it's the direction of steepest increase of the function
=> to find 1st derivative in 2D settings I need to compute the magnitude of the gradient.

Figure 8.21

The process of Canny is:

1 Compute smooth (filtered) derivatives:

$$\begin{aligned} \tilde{I}_x(x, y) &= \frac{\partial}{\partial x} (I(x, y) * G(x, y)) = I(x, y) * \frac{\partial G(x, y)}{\partial x} = \Delta \\ \tilde{I}_y(x, y) &= \frac{\partial}{\partial y} (I(x, y) * G(x, y)) = I(x, y) * \frac{\partial G(x, y)}{\partial y} = \star \end{aligned}$$

differentiation commutes
with convolution
=> I take the image outside
the derivation

I compute both horizontal and vertical smooth derivatives:

by taking the derivative of the convolution between the image and a Gaussian filter, and I do it along x and y directions.

$$\begin{aligned} \Delta &= I(x, y) * (G'(x) \cdot G(y)) = ((I(x, y) * G'(x)) * G(y)) \quad \text{(smooth horizontally and then vertically)} \\ \star &= I(x, y) * (G'(y) \cdot G(x)) = ((I(x, y) * G'(y)) * G(x)) \end{aligned}$$

I apply Separability of a 2D Gaussian function (a 2D Gaussian function is equal to the product of 2 1D Gaussian function along x and y.)

$$G(x, y) = G(x) \cdot G(y)$$

I do the chain of convolutions:
=> two 1D convolutions are more faster and computationally efficient than a single 2D convolution

Figure 8.22

=> I got the two components of the gradient, now:

2 Find peaks (local maxima) of the gradient magnitude along the gradient's direction by Non-Maxima Suppression

- we search only for maxima because the gradient magnitude is always a positive quantity

3 Then, we apply a final pruning step -> a thresholding step

- rather than keeping all the peaks of the gradient magnitude along the gradient direction, we threshold to prune out some of them because they might still due to noise.
=> we threshold the magnitude of the gradient.

- and we apply a particular thresholding called Hysteresis Thresholding.

Figure 8.23

8.2.1 Hysteresis Thresholding

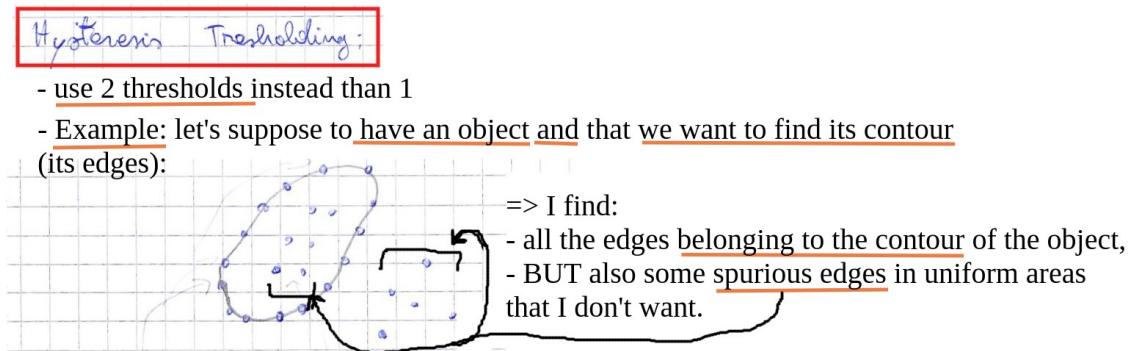


Figure 8.24

If I apply just 1 threshold Th:

- then I discard all noise (all spurious edges)
 - BUT I also discard some of the contour points of my object
 - I discard those that have a gradient magnitude < than Th
- => I have a broken contour.

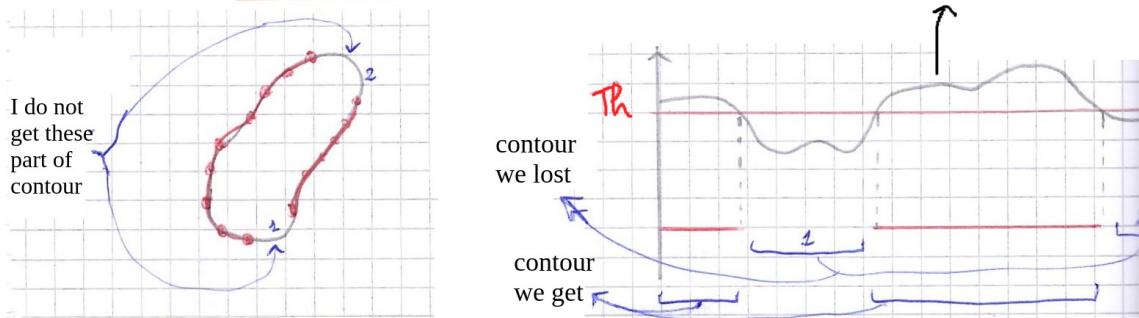


Figure 8.25

=> If I want to recover my lost contour, then I should have a smaller threshold => to include contour points (edges) that have a smaller gradient magnitude,

BUT if I use just a single lower threshold, then the drawback is that I will allow also noise points.
 => a single threshold is NOT enough.

Figure 8.26

Hysteresis thresholding:

- 2 thresholds:

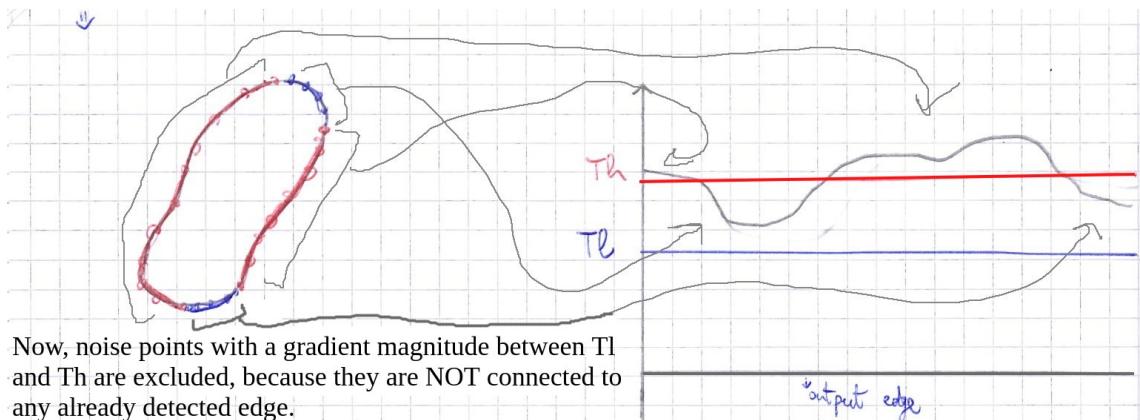
- an high threshold Th
- and a low threshold Tl

=> a pixel is taken as an EDGE if:

- its gradient magnitude is > than Th
- OR**
- its gradient magnitude is > than Tl **AND** the pixel is a neighbour of an already detected edge.

with this passage I will be able to accept weaker edges without introducing noise

Figure 8.27



Now, noise points with a gradient magnitude between Tl and Th are excluded, because they are NOT connected to any already detected edge.

Figure 8.28

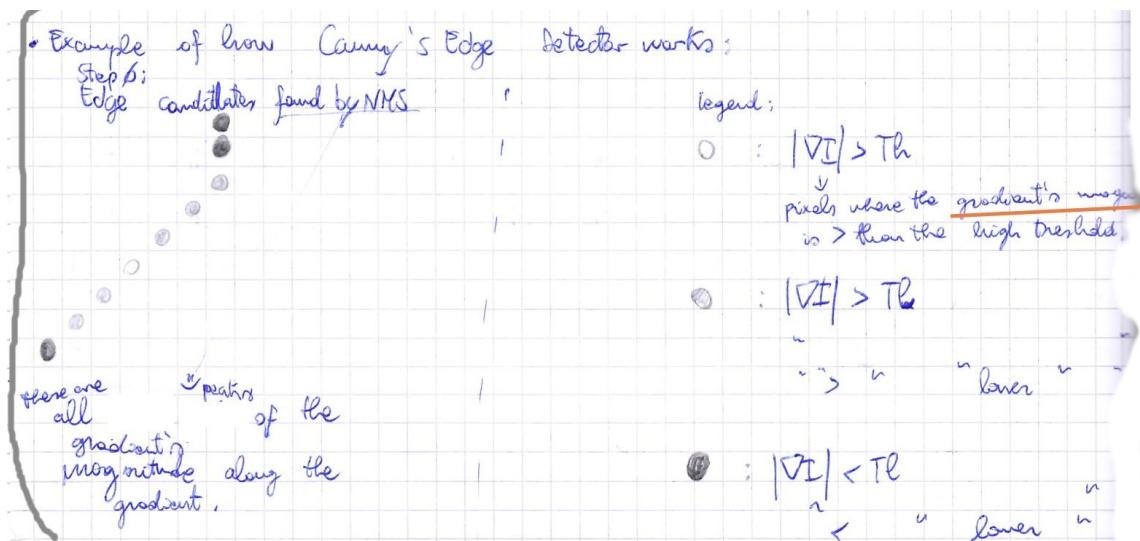
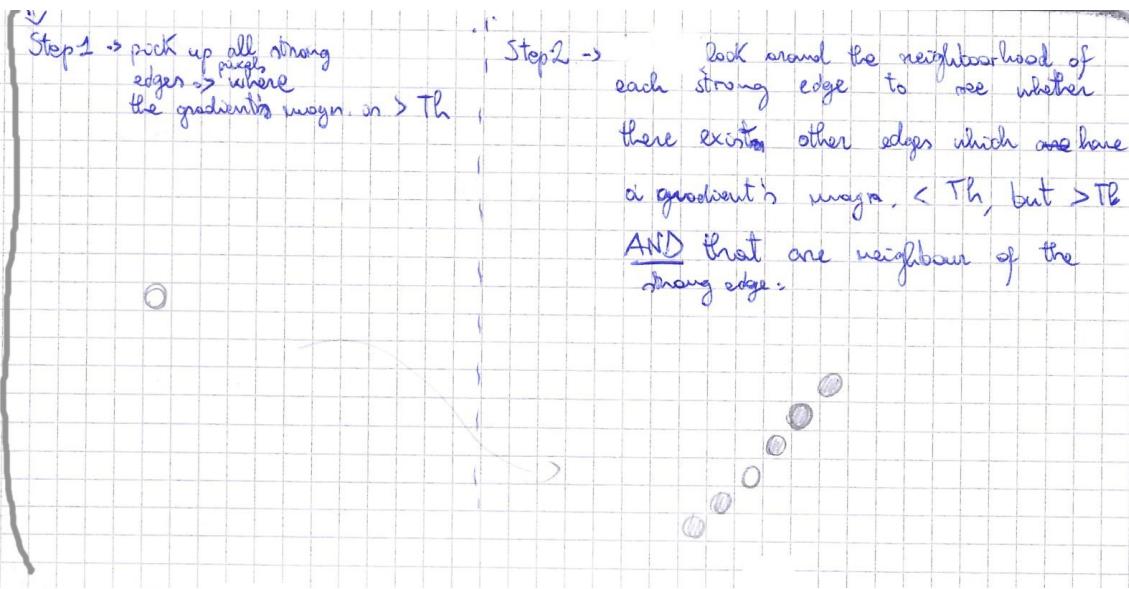


Figure 8.29



=> with Canny we get as output:

- not just a simple edge map,
 - but we get a set of Connected Contours
- => we do not need to label them

Figure 8.30

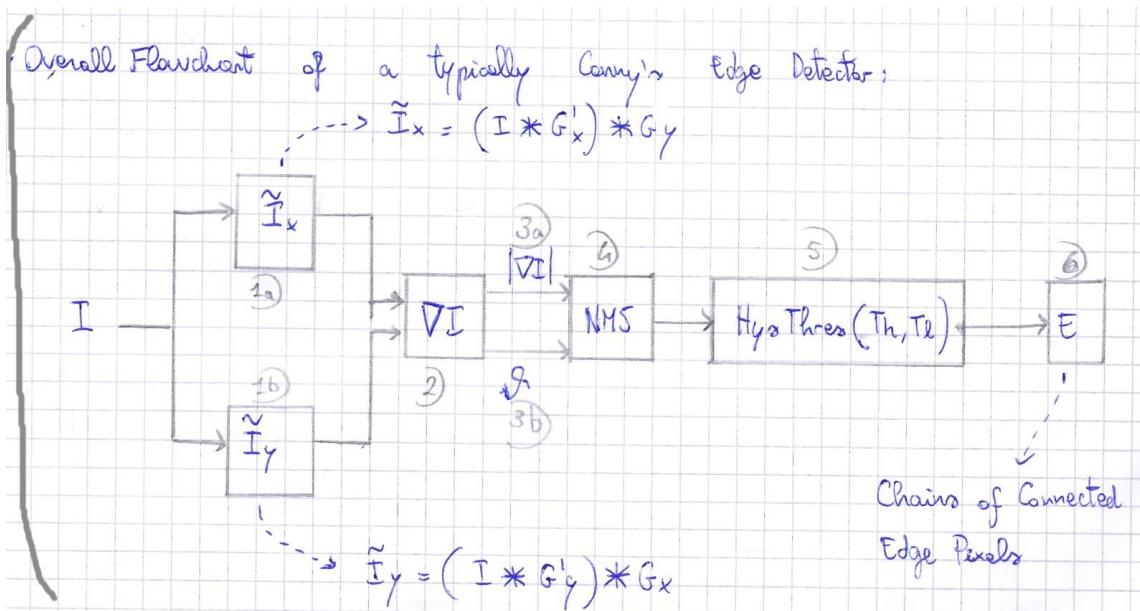


Figure 8.31

2a; 3b) compute smooth derivatives by Gaussian Filtering

- 2) then, because we have the partial derivatives (\tilde{I}_x ; \tilde{I}_y), which are the components of the gradient \Rightarrow we have the gradient.
- 3a) b) we compute the magnitude of the gradient
- 3b) and its orientation
- 4) and can perform a directional Non-Maximum Suppression
- 5) then, the final step is an Hysteresis thresholding \Rightarrow first keep strong edges and then grow other edges based on strong ones by the chain growing procedure.
- 6) so get chains of connected pixels, not just a basic maps.

Figure 8.32

Chapter 9

Local Invariant Features

9.1 3 Main steps of local invariant features pipeline matching

↳ Goal = establishing correspondencies between images
↓
• 3 successive steps to obtain it:

Figure 9.1



Figure 9.2

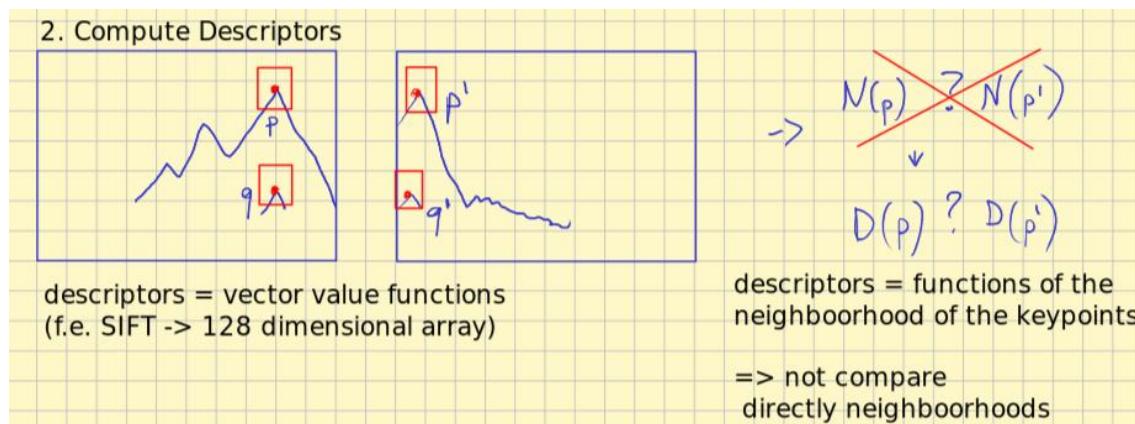


Figure 9.3

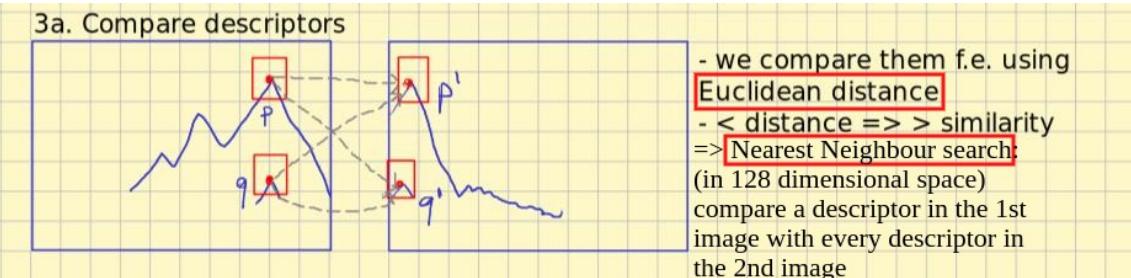


Figure 9.4

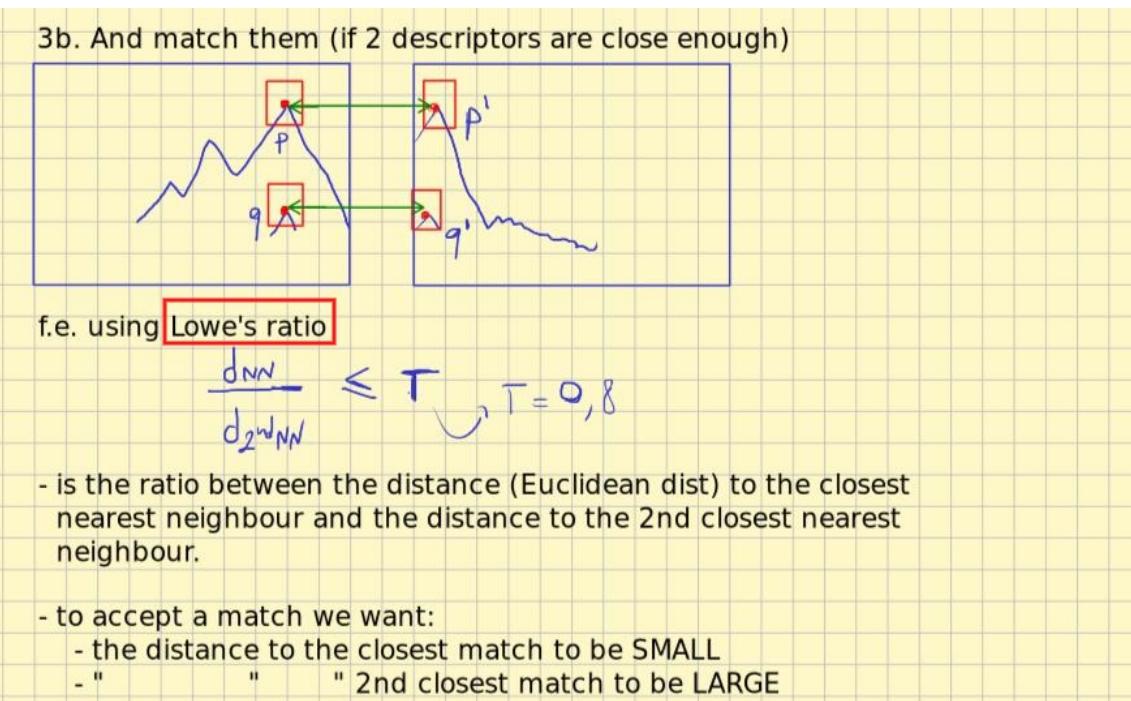


Figure 9.5

Why should we use a DESCRIPTOR rather than comparing directly the patches? (why don't we compare directly the grayscale of the patches?)

- ↳ because we want to establish a MATCH which is **INVARIANT** to all these possible changes. → (f.e. lighting changes, noise, different viewpoint, ecc...)
- ↳ The idea of descriptor is all about finding a function that factors out changes and extract the important informations from the patch.
- ↳
 - Matching 2 descriptors does work much better (rather than matching directly the windows) because it is invariant (more robust) to main transformations that might relate the 2 patches
 - The invariance to some properties depends on the chosen function (on the descriptor).

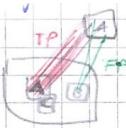
Figure 9.6

9.2 Evaluate the performance of a feature MATCHING pipeline

- How do we measure the performance of a whole Feature Matching Pipeline?
 - Suppose we have 2 pipelines, then they would differ in either or both detection and description, because matching is quite standardly deployed through an Euclidean distance.

Figure 9.7

- Trade-off between finding TP and FP:
 - ↳ assume you have not enough TP \Rightarrow we can modify the threshold on the Euclidean distance to accept also larger distances.
 - ↳ if we increase the MIN dist to allow a match \Rightarrow we gather more TP, but drawback: we will find more easily FP.



• How do we set the ideal threshold?

↳ Use Recall and Precision.

- Recall = number of true positives over the number of positives = $\frac{TP}{P} = \frac{TP}{TP+FN}$

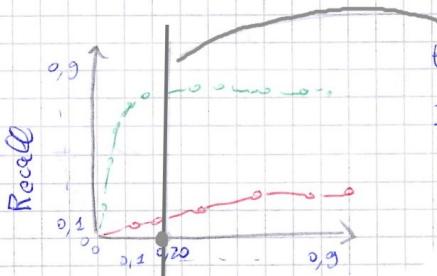
- Precision = $\frac{\text{all positives}}{\text{TP} + \text{FP}}$
- ↳ over the number of all matches we found
- e.g.: precision = 90% \Rightarrow 90% of matches are TP
- 10% are FP

↳ we cannot optimize both

Figure 9.8

Precision-Recall curves

↳ show both precision and recall as a function of the parameter used to establish a match (f.e. the threshold of the Euclidean distance).



if 1-1
this line means "I accept 20% Precision" (which mean
I can live with 20% wrong matches)

Ridge : the best place there?

Recall
It is that, for that level of precision,
gives me the higher recall

• Precision is about MAKING MISTAKES

Recall is about getting all the things you can detect.

Precision = \rightarrow Specificity

- Recall = ↳ specificity ↳ sensitivity

↑
but you could miss punctures) ← You'll never say that
if it is not. → never get a
fresh → you'll be very precise → many
errors.

Figure 9.9

9.3 Harris Corner Detector

- Harris Corner Detector:
 - based on the Cornerness function
 - * is a function which tries to find the points inside an image which have a max variation in all directions => tries to find corners
- Harris Corner Detector -> based on a continuous formulation of the Moravec Corner Detector
 - Moravec -> discrete formulation
 - Moravec -> you consider the error function from the given patch and a set of patches, each shifted by a discrete quantity

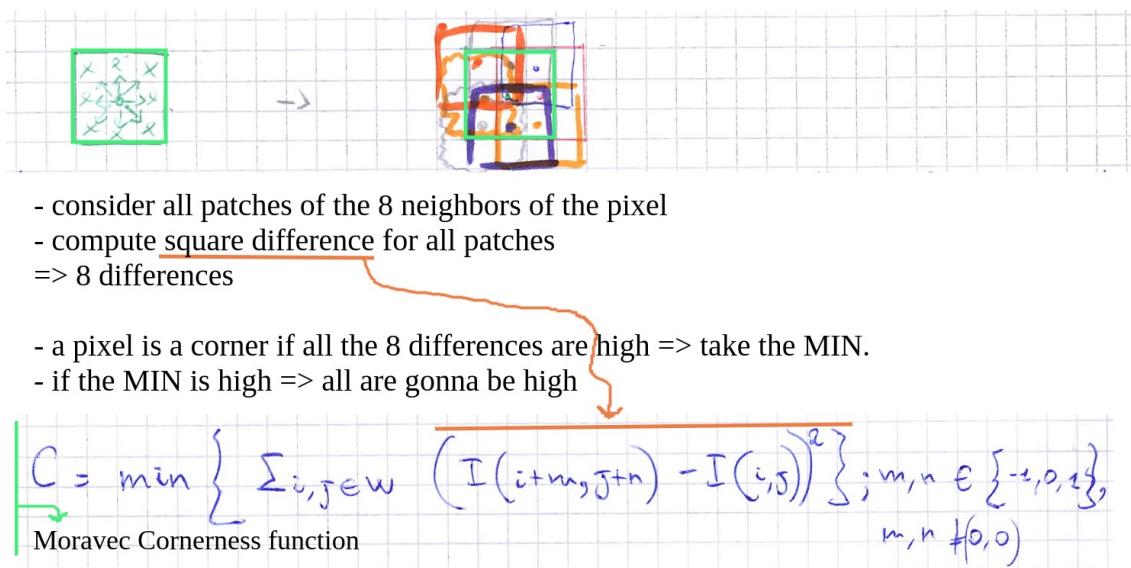


Figure 9.10

- In HARRIS -> consider the error function wrt the patch shifted by an infinitesimal quantity
 - => now we consider an infinitesimal shift

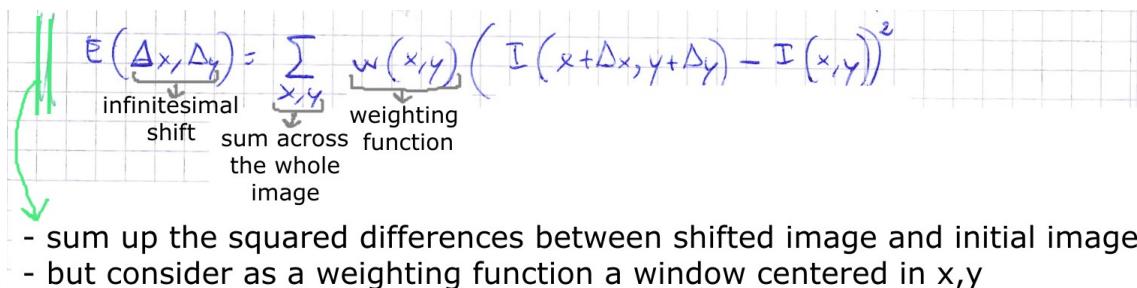


Figure 9.11

- purpose of the WEIGHTING FUNCTION $w(x,y)$:

- it is centered at the point which we're considering s.t. it would not longer be a global difference, but it would be a local difference
- => we consider only the differences of the patches (not of the entire images)

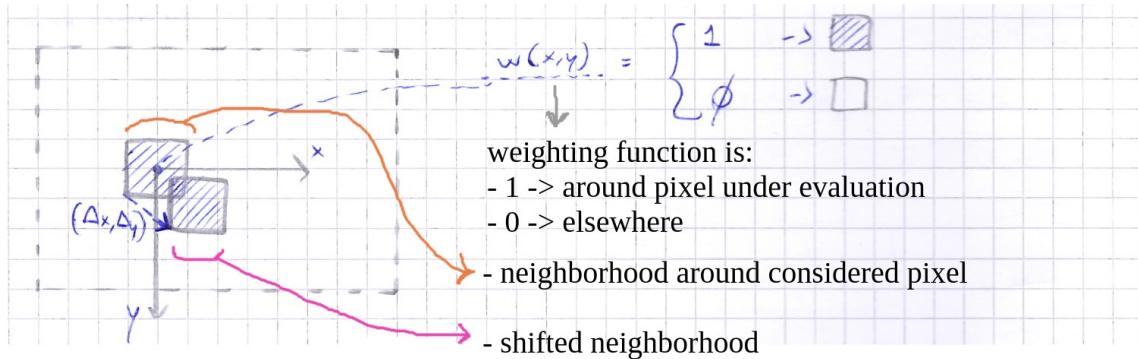


Figure 9.12

- the weighting function can be:

- * a step function
- * or, more often, a Gaussian function:
 - => so we can give more weight to the closer neighbours of the considered point



Figure 9.13

- with an infinitesimal shift => we can deploy TAYLOR'S EXPANSION to approximate our function:

Taylor's expansion of function $I(x+\Delta x, y+\Delta y)$ in a neighborhood of $I(x,y)$:

$$I(x+\Delta x, y+\Delta y) = I(x,y) + \underbrace{\frac{\delta I(x,y)}{\delta x} \cdot \Delta x}_{\text{derivative of } I(x,y) \text{ along } x} + \underbrace{\frac{\delta I(x,y)}{\delta y} \cdot \Delta y}_{\text{derivative of } I(x,y) \text{ along } y}$$

I approximate this function through a Taylor's expansion

Figure 9.14

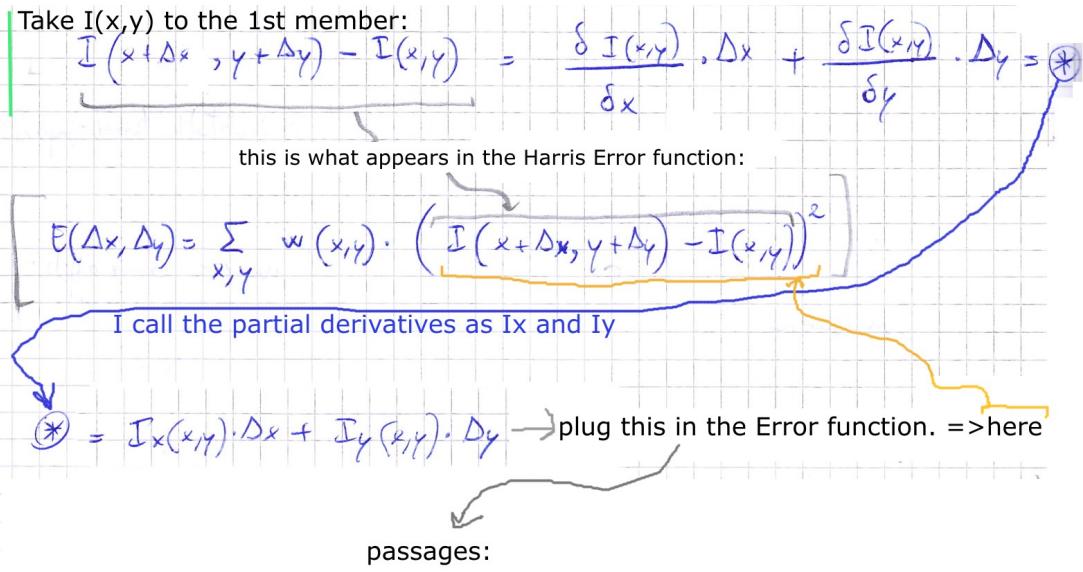


Figure 9.15

$$\begin{aligned}
 E(\Delta x, \Delta y) &= \sum_{x, y} w(x, y) \cdot (I(x + \Delta x, y + \Delta y) - I(x, y))^2 = \\
 &= \sum_{x, y} w(x, y) \cdot (I_x(x, y) \cdot \Delta x + I_y(x, y) \cdot \Delta y)^2 = \text{expand the square} \\
 &= \sum_{x, y} w(x, y) \cdot (I_x(x, y)^2 \cdot \Delta x^2 + I_y(x, y)^2 \cdot \Delta y^2 + 2I_x(x, y) \cdot I_y(x, y) \cdot \Delta x \cdot \Delta y) = \\
 &\text{rewrite the relationship in a matrix form} \\
 &= \sum_{x, y} w(x, y) \begin{pmatrix} \Delta x & \Delta y \end{pmatrix} \begin{pmatrix} I_x(x, y)^2 & I_x(x, y) \cdot I_y(x, y) \\ I_x(x, y) \cdot I_y(x, y) & I_y(x, y)^2 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \\
 &\text{put the shift vectors out of the sum} \\
 &\downarrow \\
 &\stackrel{1 \times 1}{=} \begin{bmatrix} \Delta x & \Delta y \end{bmatrix} \cdot \begin{pmatrix} \sum_{x, y} w(x, y) \cdot I_x(x, y)^2 & \sum_{x, y} w(x, y) \cdot I_x(x, y) \cdot I_y(x, y) \\ \sum_{x, y} w(x, y) \cdot I_x(x, y) \cdot I_y(x, y) & \sum_{x, y} w(x, y) \cdot I_y(x, y)^2 \end{pmatrix} \stackrel{2 \times 1}{=} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}
 \end{aligned}$$

STRUCTURE MATRIX M

Figure 9.16

- we obtain the Error as function of the structure matrix M:

$$E(\Delta x, \Delta y) = [\Delta x \ \Delta y] \cdot M \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

M = structure matrix

Figure 9.17

- the matrix M can be diagonalized:
 - because all real and symmetric matrices can be diagonalized.
- Error function with diagonal structure matrix:

error function:

$$E(\Delta x, \Delta y) = [\Delta x \ \Delta y] \cdot \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \cdot \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \lambda_1 \cdot \Delta x^2 + \lambda_2 \cdot \Delta y^2$$

Figure 9.18

- Error as a function of the eigenvalues \rightarrow 3 cases:
 1. IF both λ_1 and λ_2 (=the eigenvalues) are **SMALL**:

IF $\lambda_1, \lambda_2 \approx 0$ (= both λ_1 and λ_2 are **SMALL**)

THEN, whatever the direction, you'll always have a small error
 - because you multiply Δx^2 and Δy^2 by λ_1 and λ_2

\Rightarrow we are in a **UNIFORM AREA**

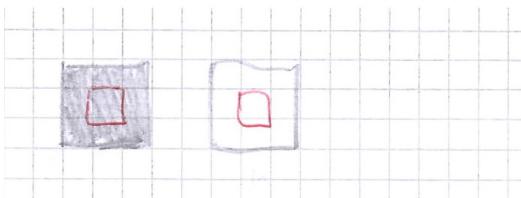


Figure 9.19

2. IF $\lambda_1 \gg \lambda_2$:

IF $\lambda_1 \gg \lambda_2$ ($\Rightarrow \lambda_1$ is large and λ_2 is small)

- ↳ - if the shift is equal to $[\Delta x, 0]$ ($\Rightarrow \Delta x \neq 0, \Delta y = 0$), and if λ_1 is large \Rightarrow the error is large
- if the shift is equal to $[0, \Delta y]$ ($\Rightarrow \Delta x = 0, \Delta y \neq 0$) (\Rightarrow the perpendicular shift to Δx), and if λ_1 is large \Rightarrow the error is small
- high variation only along 1 axis
and no variation in the perpendicular axis
 \Rightarrow we have an EDGE.

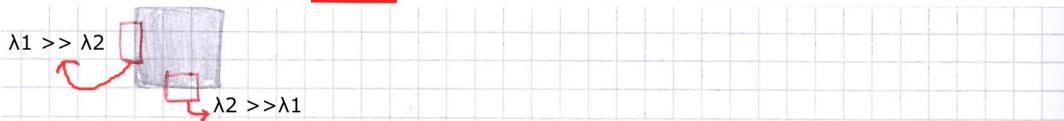


Figure 9.20

3. IF both λ_1 and λ_2 are LARGE:

IF both λ_1 and λ_2 are large:
 \Rightarrow THEN the error is high in whatever direction
 \Rightarrow I have a CORNER



Figure 9.21

- How is the Harris criterion formulated in practice?

- In practice we don't calculate the eigenvalues
 - * because it's costly from a computational viewpoint
- So, we compute the Cornerness function based on:
 - * the DETERMINANT of the structure matrix M
 - * and the TRACE of the structure matrix M

M

$$C = \det(M) - K \cdot \text{tr}(M)^2 = \lambda_1 \cdot \lambda_2 - K (\lambda_1 + \lambda_2)^2$$

- M = structure matrix
- k = parameter
- tr = trace = sum of elems on the main diag
- det = determinant

\Rightarrow we compute the Cornerness without calculating the eigenvalues

Figure 9.22

C = Cornerness function

- when $C \geq 0$ -> UNIFORM AREA

- when $C > 0$ -> CORNER

- when $C < 0$ -> EDGE

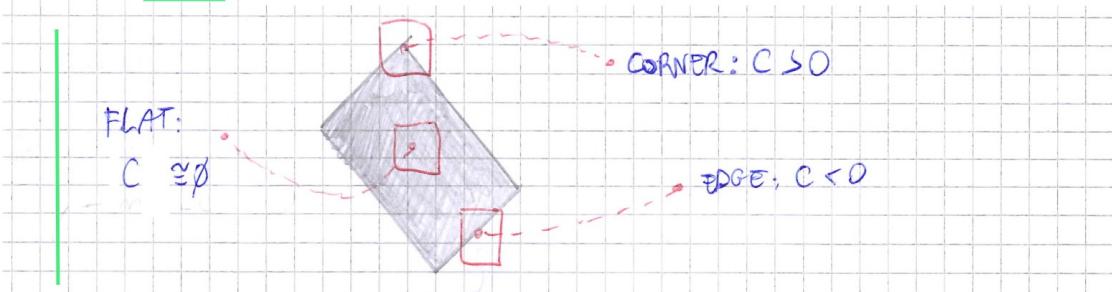


Figure 9.23

- Summary of HARRIS Corner Detector:

1. compute the Cornerness function C for all pixels in the image
 - using the determinant and trace of the structure matrix M
2. we select all pixels with $C >$ than a chosen threshold T
3. perform NMS (Non-Maxima Suppression) to just get the peaks of this cornerness function
 - => we have found our corners

- Invariance Properties of Harris Corner Detector:

- To Rotation Invariance -> YES
 - * because the determinant and the trace of a matrix don't change under rotations
- To Intensity Changes -> only with an additive bias.
- To Scale Changes? -> NO
 - * because we use a fixed sized window

- Why all real and symmetric matrices can be diagonalized:

The structure matrix M , as a real and symmetric matrix, can always be diagonalized through a rotation of the image coordinate system:

$$M = R \cdot \begin{bmatrix} \lambda_1 & \phi \\ \phi & \lambda_2 \end{bmatrix} \cdot R^T, \quad R = \text{rotation matrix.}$$

↳ and the element along the diagonal would be the eigenvalues of M .

↳ through the eigenvector decomposition, you can always diagonalize M and so, find a basis such that M is diagonal.

Figure 9.24

↳ You don't need to do the diagonalization, you just need to be focused on the eigenvalues of M . M is just computed by derivatives. M is given \Rightarrow we study its eigenvalues, which can always be computed. And then we establish a point being a **FEAT**, an **EDGE**, or a **COMER** based on the eigenvalues.

Figure 9.25

9.4 DOG key points

- Topic = Local Invariant Feature detection.
- Goal = find a way to detect keypoints.
- Original idea by Lindberg = Scale Normalized Derivatives (in which LoG is used)
- New idea = Lowe preferred to use a stack of increasingly smoothed pictures and to detect keypoints from their differences.
- Octave = range of scales from sigma to $2 * \sigma$

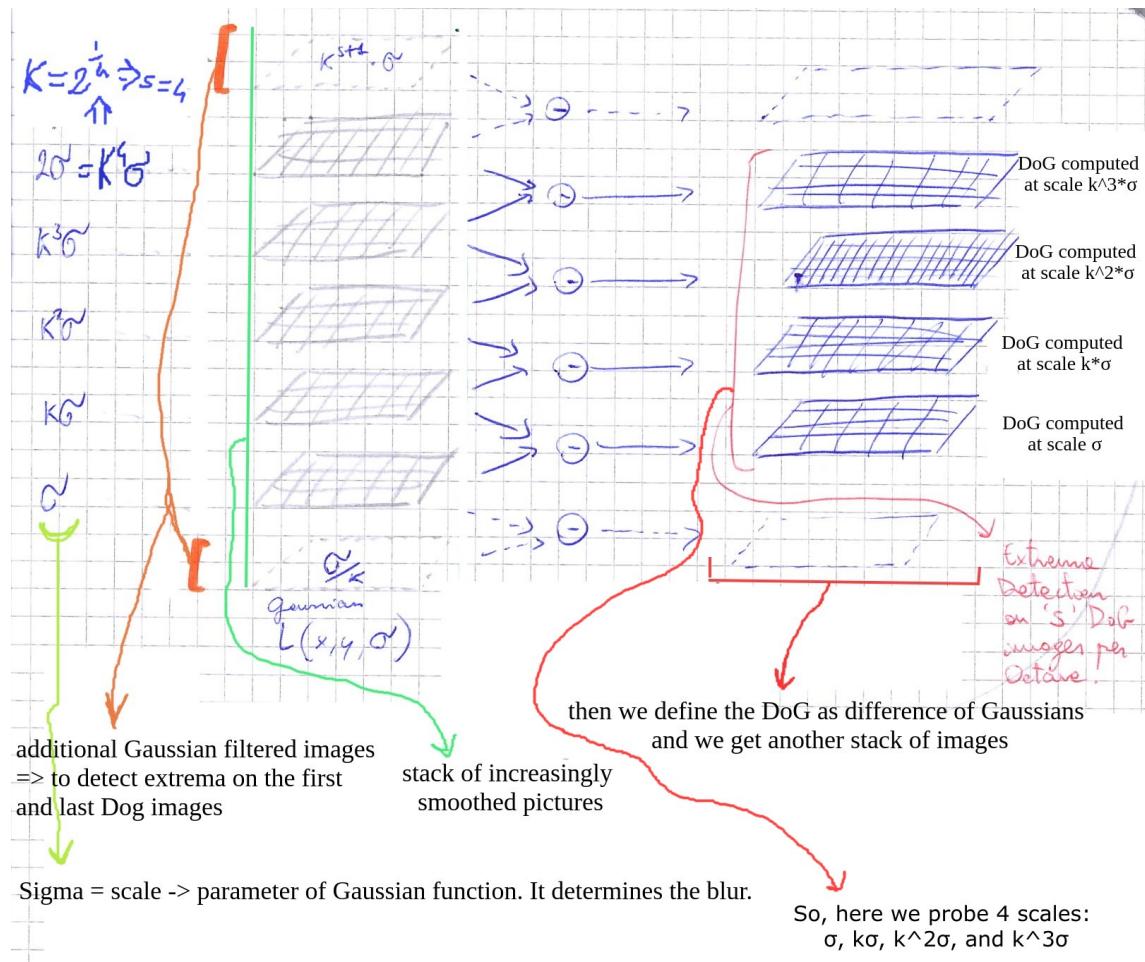
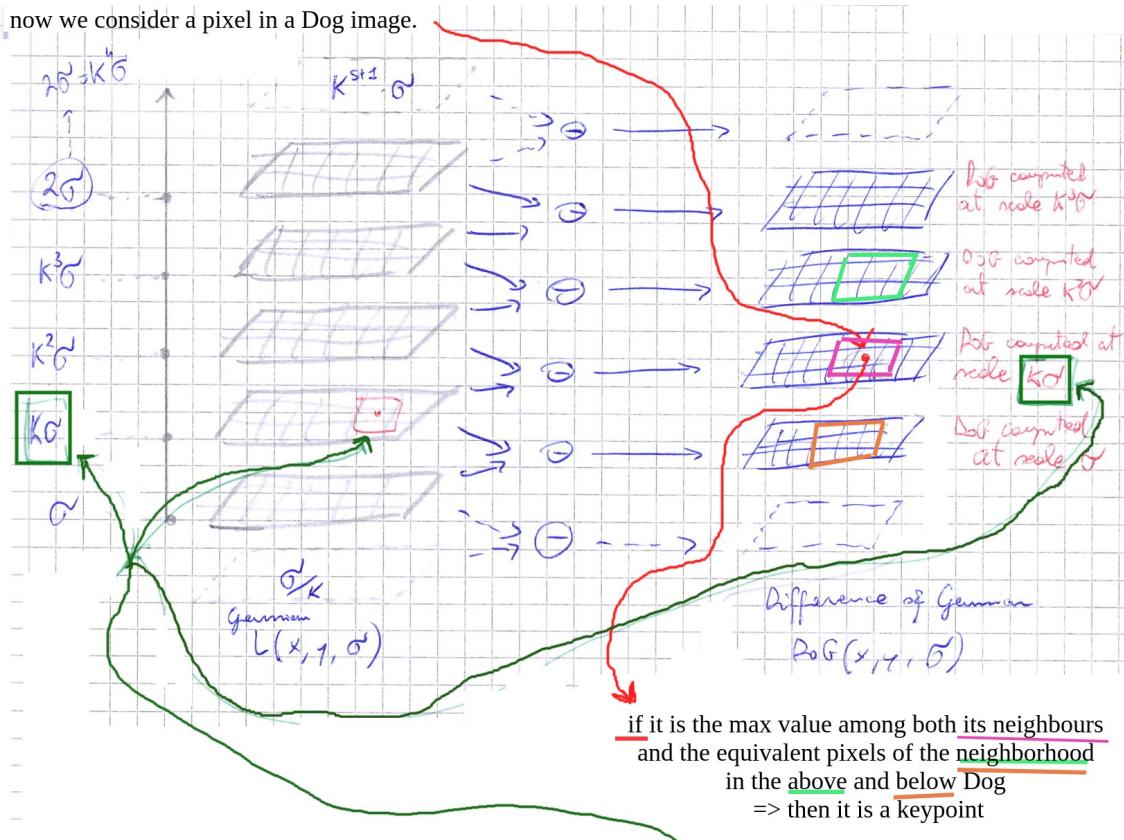


Figure 9.26

now we consider a pixel in a Dog image.



Then, we compute its descriptor at the scale at which it was found.
 => the patch to compute the descriptor has to be taken from the Gaussian filtered image at which that point was found to be a keypoint, which is $k\sigma$

Figure 9.27

Just a Single Octave? -> No, we want to sample more scales

- No -> multiple Octaves, but rather than doubling the size of the filters (= bigger kernels \Rightarrow more computations)
 - \Rightarrow shrink the images (divide by 2 height and width) and apply the same filters sizes

Pro of DoG wrt LoG (computational advantage of DoG):

- We already have this Gaussian filtered image.
 - Rather, if we computed the LoG, I would need to smooth the image with the given σ to compute the descriptor.

- Why do we compute the descriptor in a smoothed image?

- features can be present in multiple scales
 - we smooth in order to not keep details in large scale features such that then the large scale features can be matched when they appear at smaller scales (ex. with top of the mountain -> we smooth away flowers)

Figure 9.28

Formula and intuition of DoG from scale normalized Log Gaussian Scale Space

$$\text{DoG}(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma)$$

↓

DoG is a filter defined as a Gaussian $(x, y, k\sigma)$ minus another Gaussian function of (x, y, σ) \Rightarrow the ratio between these 2 σ is k . Then this 'difference of Gaussians' is the filter whereby we convolve the image.

Given our previous definition of a Gaussian Scale-Space

$(L(x, y, \sigma) = G(x, y, \sigma) * I(x, y))$, then this all this can be seen as the Gaussian Scale-Space at scale $k\sigma$ minus the Gaussian Scale-Space at scale σ .

Figure 9.29

Lowe proved that a DoG function is just a scaled version of Lindberg's scale normalized log.

$$G(x, y, k\sigma) - G(x, y, \sigma) \approx (k-1) \cdot \sigma^2 \cdot \Delta^2 G(x, y, \sigma)$$

there's just this multiplicative factor between a DoG and a scale normalized log

this multiplicative factor $(k-1)$ is irrelevant as far as extreme detection is concerned. Indeed, we're not interested in the exact values of the DoG or LoG functions.

And because these 2 functions differ just by this $(k-1)$ constant multiplicative term, then the two extremes are just found at the same position.

- = DoG
- = Laplacian

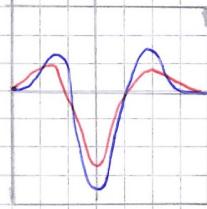


Figure 9.30

9.4.1 In the local invariant feature pipeline how do we achieve rotation and scale invariance? Rotation and scale invariance in SIFT.

The descriptor is invariant wrt scale, because:

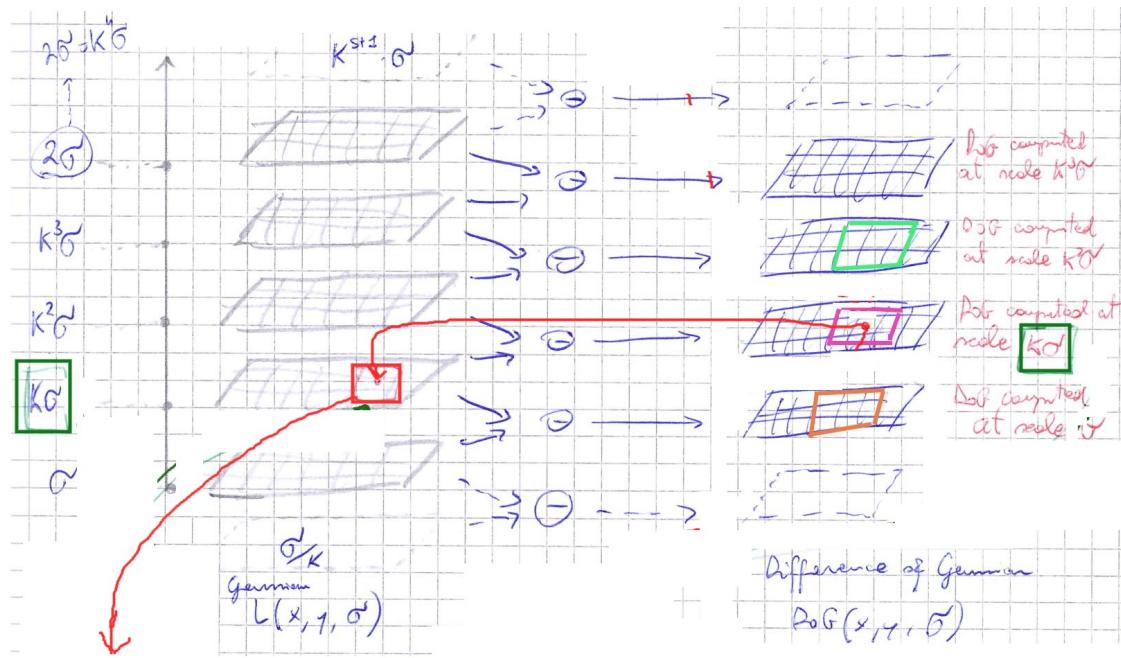
- I compute the descriptor at a right scale => it has the right level of blurring => the descriptor is invariant wrt scale.

To obtain a rotation invariant descriptor (=> Next step after keypoints have been found:)

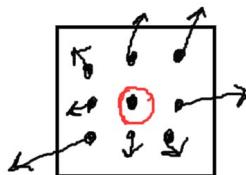
- we first want to find a canonical orientation of the patch of the keypoint => to have a descriptor which is rotation invariant => because we want to match features invariantly to rotation

To compute the canonical orientation we create an orientation histogram:

- divide angular range into bins (in SIFT $10^\circ \Rightarrow 36$ bins)
- for each pixel in the patch of the keypoint: compute magnitude and orientation (=the angle) of the gradient



we compute the canonical orientation in a patch of the keypoint:



$\text{O} = \text{found keypoint}$

Figure 9.31

- increment the corresponding bin (corresponding to the orientation of the gradient) by as much as the magnitude of the gradient for that pixel
- gradient magnitude is also weighted by a Gaussian function -> to give more weight to pixels closer to the given keypoint
- a peak will be the canonical orientation of my keypoint

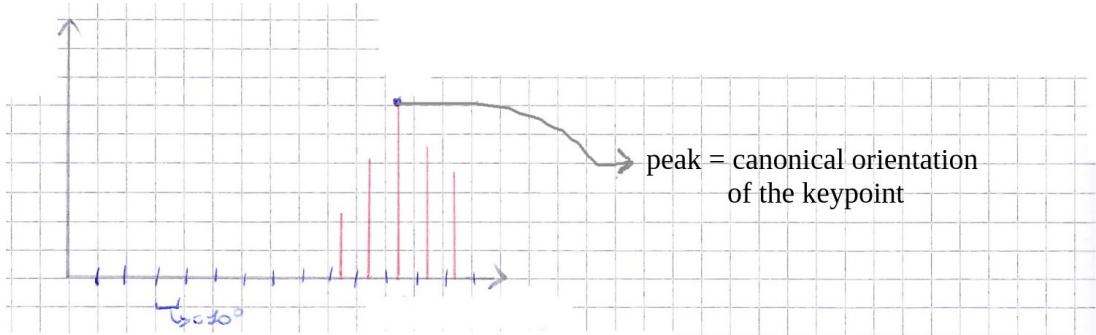


Figure 9.32

- if there's a 2nd peak higher than 80% of the main one => pick both
 - => we will compute 2 descriptors for that keypoint

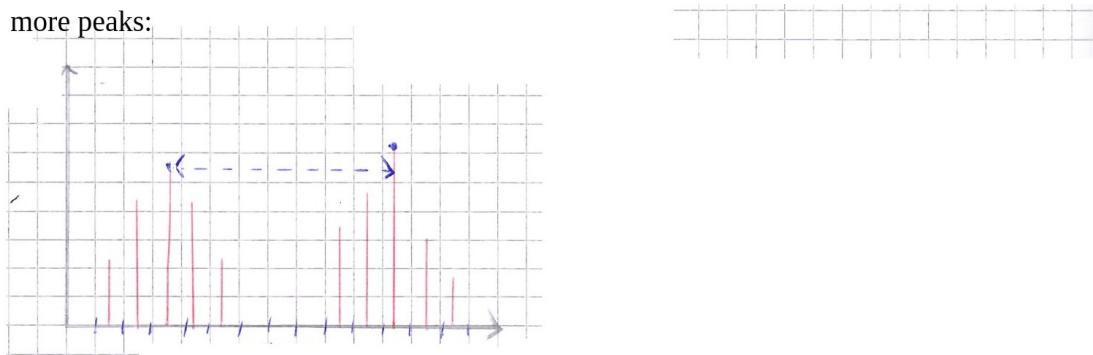


Figure 9.33

9.5 SIFT Descriptor

SIFT defines a grid of 16 sub-regions:

- each sub-region is a 4x4 pixel square

The descriptor is canonically oriented (\Rightarrow invariant to rotation thanks to the canonical orientation given by the orientation histogram).

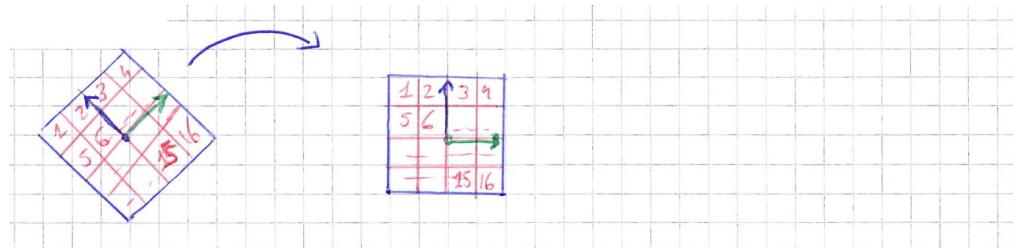


Figure 9.34

- for each sub-region compute an orientation histogram (different from the orientation histogram of the descriptor)
 - each histogram has 8 BINS (angular step = 45°)
 - all 16 pixels in each sub-region vote for their corresponding BIN (corresponding to the gradient orientation)
 - * a pixel votes by accumulating a quantity as large as its gradient magnitude into that BIN

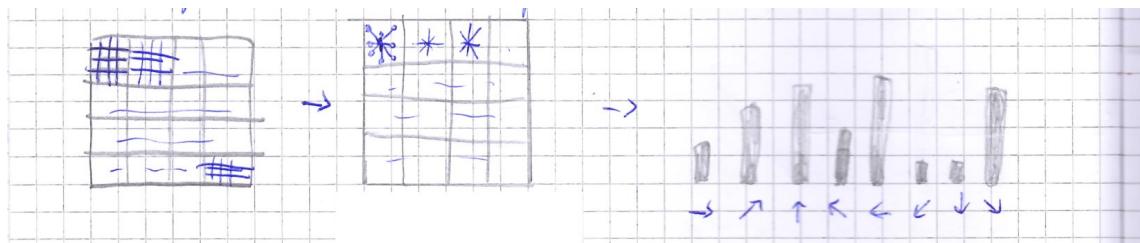


Figure 9.35

\Rightarrow 8 bins per histogram \times 16 sub-regions = 128 dimensional vector (= SIFT descriptor)

- arrows = 8 directions (separated by 45°) of quantized gradient.
- > large arrow \Rightarrow > magnitude of gradient in that BIN.

9.5.1 Validating matches in SIFT

Lowe's ratio \rightarrow Figure 9.4.

Chapter 10

Object Detection

10.1 Template matching

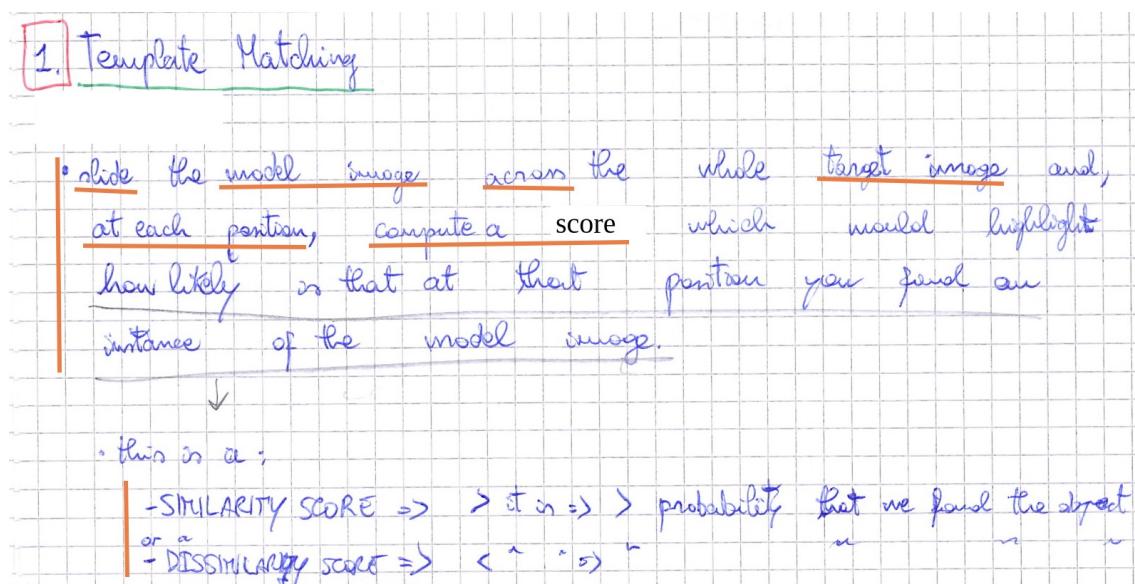


Figure 10.1

10.1.1 SSD

What are the (dis)similarity functions?

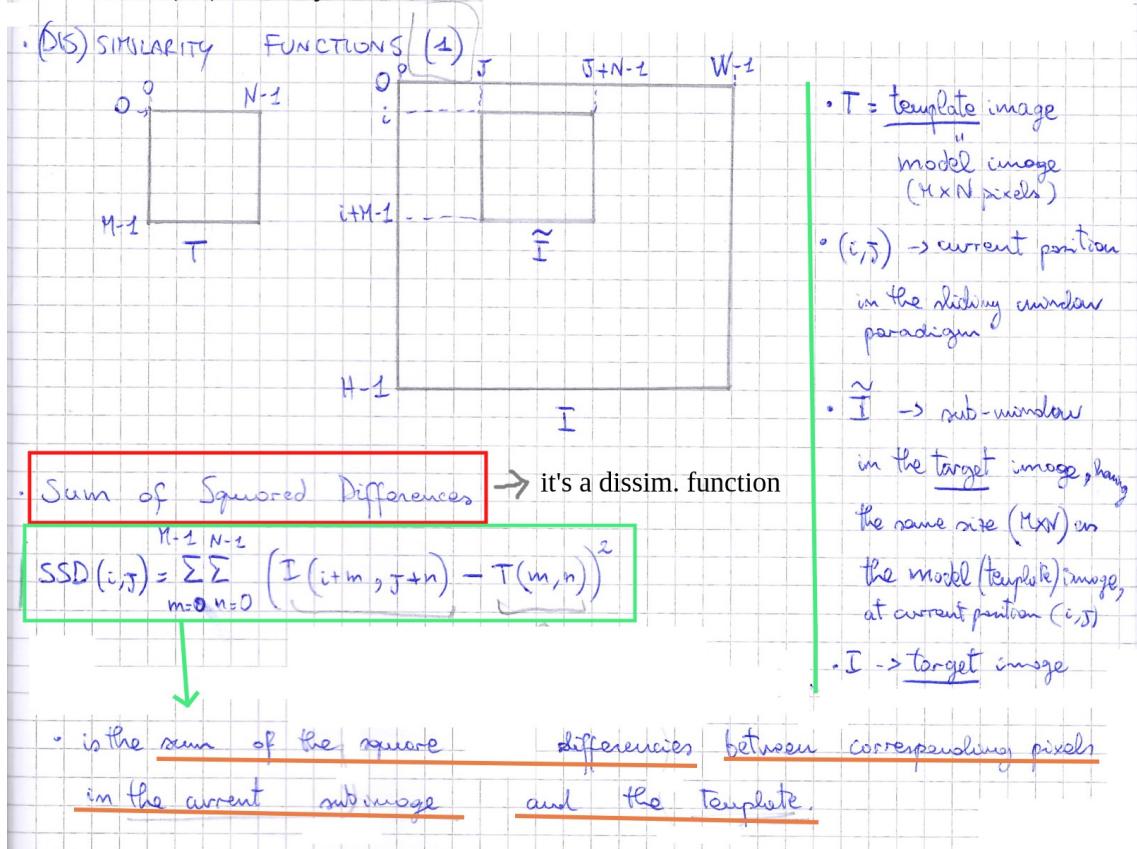


Figure 10.2

\hookrightarrow we consider each pixel of the template (= model) and compare it with the corresponding pixel in the current sub image \tilde{I} .

$\cdot \text{SSD} \Rightarrow \text{similar are the template } T \text{ and the current subimage } \tilde{I} \text{ at position } (i, j).$

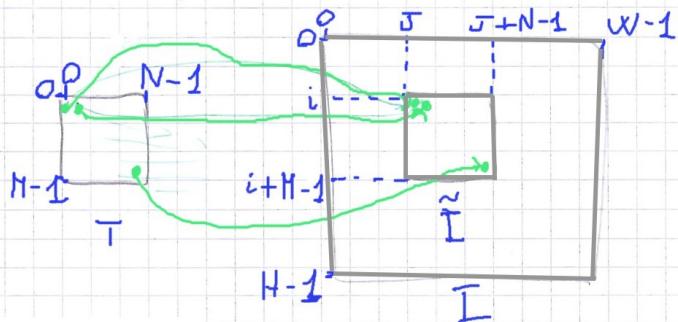
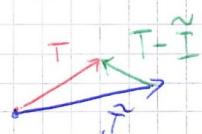


Figure 10.3

vector interpretation \rightarrow (it helps understanding)
 the invariances that the different similarity or dissimilarity functions
 do actually enjoy.

L, T (the template = the model) and the current subimage \tilde{I} as 2 vectors.



$\cdot T \rightarrow$ vector in $M \times N$ dimensional space
 \downarrow viewing all pixels as if they form a single vector.
 it's not in 2-Dimensional space.

Figure 10.4

b) $SSD = \text{square Euclidean Norm}^{(L2Norm)} \text{ of the green vector} \text{ (of the difference between } T - \tilde{T} \text{ represented as vector)}$

↓
it has the square root
↓
the square root goes away.

Basically it's the magnitude of the difference between the 2 vectors associated with T and \tilde{T} .

Figure 10.5

10.1.2 SAD

• DIS(SIMILARITY) FUNCTIONS [2]

DISSIMILARITY FUNCTION:

- Sum of Absolute Differences,

$$SAD(i, j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} |I(i+m, j+n) - T(m, n)|$$

here we take the sum of the absolute values of the differences between corresp. pixels in the current sub-image and the template.

Figure 10.6

↳ Why could we be interested on using SSD or SAD?

↳ Reasons:

- If our application settings ensure that there's no light changes between the conditions in which the model image is acquired and those in which the target will be search
 - ↳ Then using SAD or SSD is a very good choice because SAD and SSD are VERY FAST and EFFICIENCY is 1 key requirement in object detection.

• In particular:

- SSD → is better when the noise is higher. It is more robust to noise if the noise is i.i.d (independent and Gaussian)
- SAD → is definitely faster, because we compute the absolute difference instead of the square differences

↳ i.i.d = independent identically distributed at each pixel

↳ Gaussian = distribution is a zero-mean Gaussian distribution.

Figure 10.7

10.1.3 NCC

③ Normalised Cross-Correlation → it's a SIMilarity function

$$NCC(i, j) = \frac{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(i+m, j+n) \cdot T(m, n)}{\sqrt{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(i+m, j+n)^2} \cdot \sqrt{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} T(m, n)^2}}$$

Figure 10.8

- The NCC multiply all the corresponding intensities between the current sub-image (\tilde{I}) and the template image, and add all them together.

normalization factor → in the
product of the square roots of the square intensities of the
current subimage and the template

Figure 10.9

Vector Interpretation of NCC:

↳ the numerator → in the dot product between \tilde{I} and T .

the denominator → in the product of the norms (Euclidean Norms)

↓

scalar product

sub-image at pos (i,j)

$$\begin{aligned}
 \boxed{\text{NCC}(i,j)} &= \frac{\tilde{I}(i,j) \cdot T}{\|\tilde{I}(i,j)\| \cdot \|T\|} = \frac{\|\tilde{I}(i,j)\| \cdot \|T\| \cdot \cos \theta}{\|\tilde{I}(i,j)\| \cdot \|T\|} \\
 &= \cos \theta
 \end{aligned}$$

← cosine of the angle between the 2 vectors.

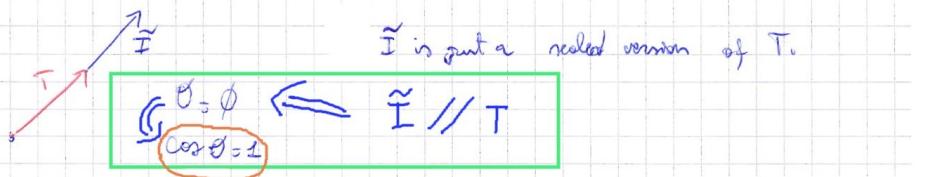
we can rewrite the dot product between 2 vectors by multiplying their norms and the cosine of the angle in between them.
 (we have already their norms in the denominator → they cancel out.)

Figure 10.10

• what if we assume that at the best matching position

$$\tilde{I}(i, j) = \boxed{\alpha} \cdot T ?$$

↳ there's an intensity change between the shape of the template and the current sub-image.



↳ $\theta = \text{the angle between } T \text{ and } I$

\Rightarrow maximum similarity between the template and the current sub-image, although the image has undergone a linear intensity change.

(If we have this linear intensity change with the SSD or SAD it would have been the norm of the vector difference \rightarrow with SSD or SAD we sense a difference).

While with NCC we have sensitivity to linear intensity change.

Figure 10.11

↳ NCC is preferable in all the applications in which one cannot guarantee stable lighting conditions.

Figure 10.12

• Drawbacks of NCC wrt SSD or SAD:

↳ NCC is slower

• Trade-off between:

Slower and robust / or

FASTER and less robust

\rightarrow it depends on the application settings.

Figure 10.13

10.1.4 ZNCC

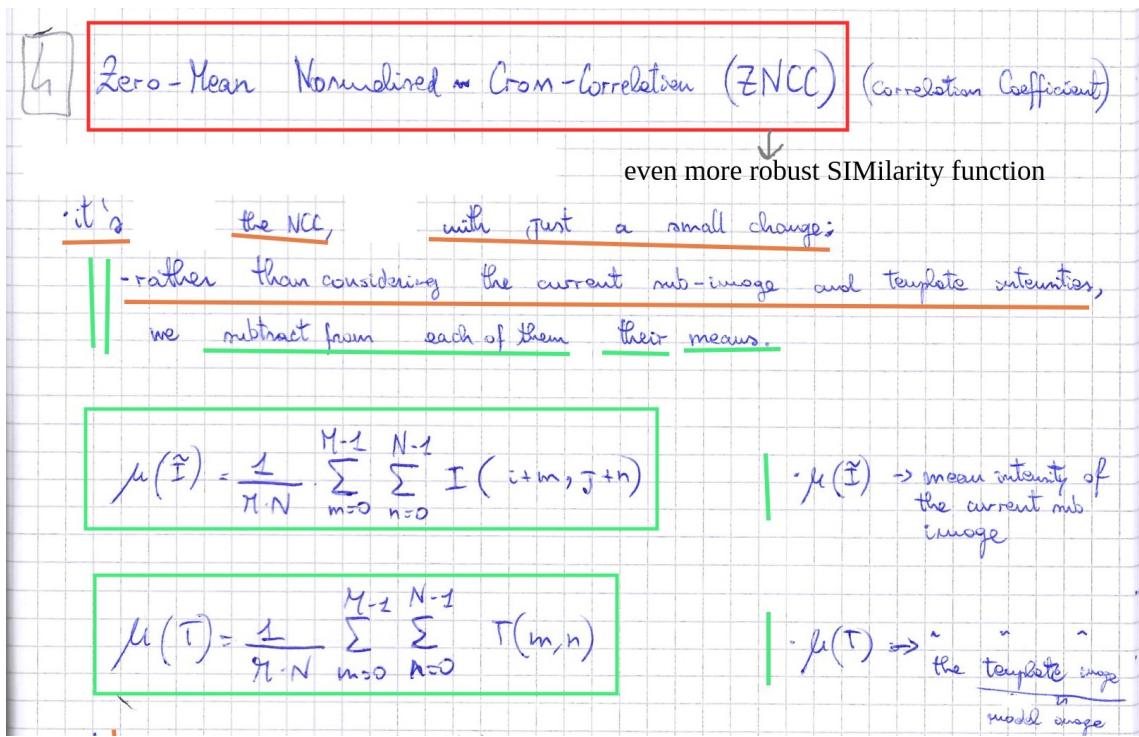


Figure 10.14

b) ZNCC \rightarrow we compute the NCC between the zero-mean versions of the current sub image and the template image.

$I(i+m, j+n) \rightarrow \underline{\underline{(I(i+m, j+n) - \mu(\tilde{I}))}}$

$T(m, n) \rightarrow \underline{\underline{(T(m, n) - \mu(T))}}$

\rightarrow rather than considering the current sub image, we subtract its mean

\rightarrow " " " template image, we subtract its mean.

$$ZNCC(i, j) = \frac{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} (\underline{\underline{(I(i+m, j+n) - \mu(\tilde{I}))}}) \cdot (\underline{\underline{(T(m, n) - \mu(T))}})}{\sqrt{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} (\underline{\underline{(I(i+m, j+n) - \mu(\tilde{I}))}})^2} \cdot \sqrt{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} (\underline{\underline{(T(m, n) - \mu(T))}})^2}}$$

Figure 10.15

Unlike NCC, the ZNCC is invariant not just to a linear change, but also to linear change + a bias

$$I(i, j) = \alpha \cdot T + \beta$$

b) ZNCC enjoys more invariance than the NCC.

Drawback:

L) ZNCC requires more computations because we have also to compute and subtract the 2 means:

- the mean for the template image is computed once because the template (= model) image doesn't change

BUT,

- the mean for the current subimage has to be computed as many times as the number of sub-images \Rightarrow as the number of different positions (i, j) .

more computation

Figure 10.16

L) However, computing this running mean can be accelerated by (as we did in box filtering \rightarrow the incremental computation of the mean we've seen when studying mean filter) using an incremental calculation scheme as the --- box filtering.



• Nonetheless, ZNCC is far more clever than NCC.



| ZNCC is SLOWER,
but MORE ROBUST.

Figure 10.17

b) If you think that in your application there will be significant light changes, then you would better go for the ZNCC because it's more invariant. Although, you cannot really know mathematically which model is the most suited tool for your application \Rightarrow you have to try.

Figure 10.18

10.1.5 Fast Template Matching

Fast Template Matching

- template matching can be very slow because you have to search the template across the whole image
- b) if either the template or the image, or both are large \Rightarrow that takes a lot of time
- c) to speed up, you can shrink target and template images and perform a search on smaller images.
- d) **Pyramid** to do that.
 - image structure in which the original image is shrunk more and more
 - typically, everytime you shrink by 2 both the number of rows as well as the number of columns.
 - It's good practice to perform **smoothing** and then **SUBSAMPLING** rather than just subsampling to avoid aliasing of the signal.

Figure 10.19

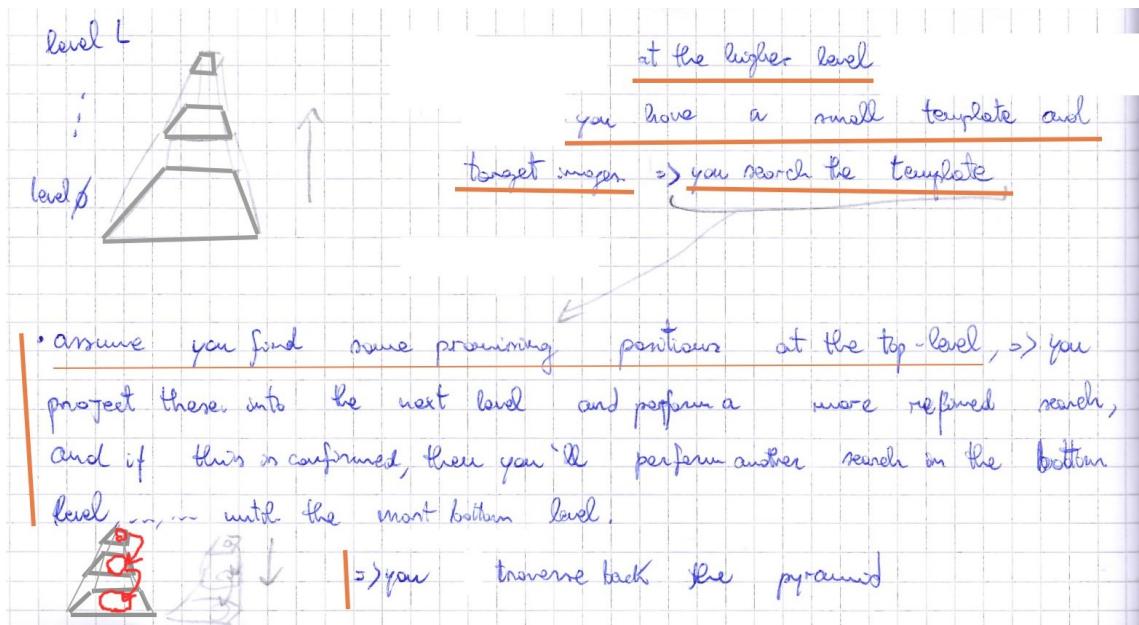


Figure 10.20

- PRO of **Fast Template Matching**
 - ↳ it saves a lot of computation => MORE SPEED, **FASTER**
- CONS:
 - ↳ it is not guaranteed to work because there's no guarantee that the overall search process would turn out equivalent to a full search, because details might be lost in the pyramid building process. without "destroying" the images

Figure 10.21

10.2 Shape-based matching algorithm

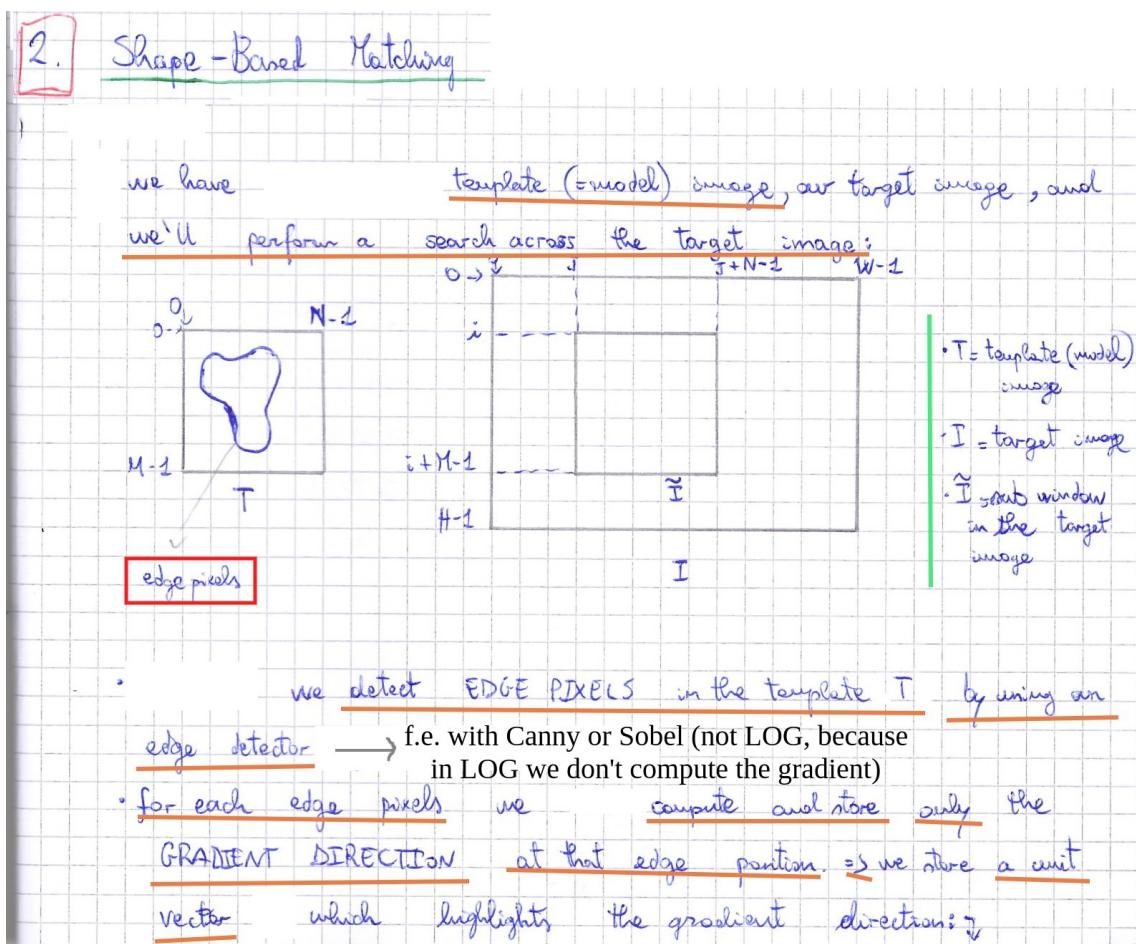
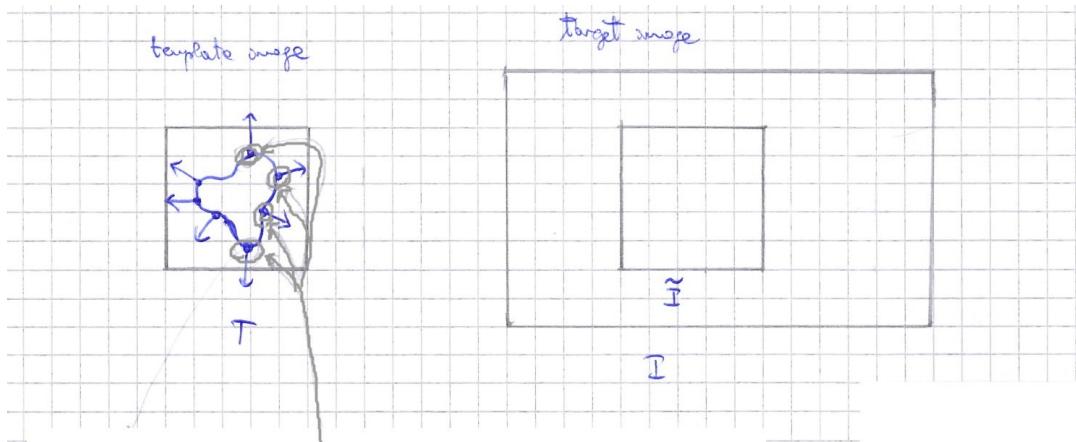


Figure 10.22



The template is summarized into a net of:

net of
CONTROL POINTS → which are offset wrt to the origin of the template
 and
 (the edge points)

CORRESPONDING UNIT → which represent gradient directions
VECTORS
 (corresponding to control points)

b) this template is then compared at each and every portion across the target
 image with the current subimage \tilde{I} .

Figure 10.23

To do the comparison:

- We consider as many points in the current subimage I as the number of edge pixels in the template.
- and then, from the origin of the current subimage I we apply the same offsets as in the template T \Rightarrow we consider the very same points in the current subimage as the edges extracted into the template.

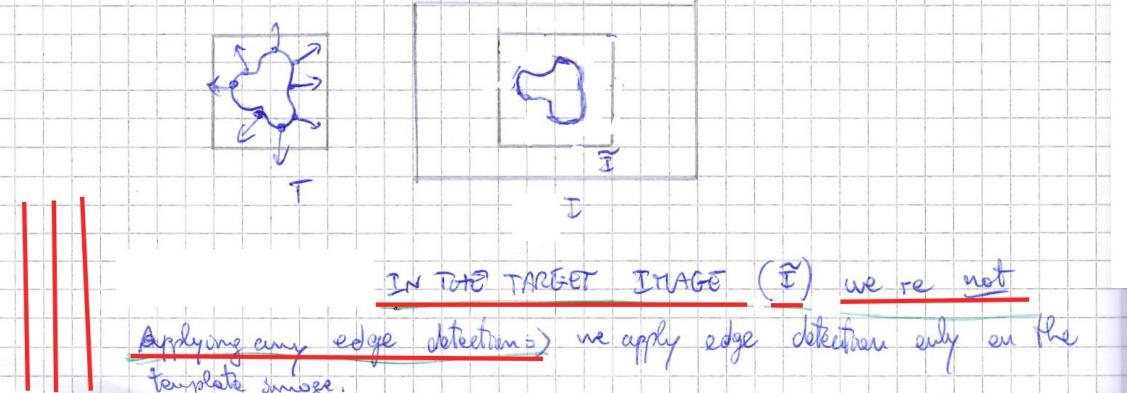


Figure 10.24

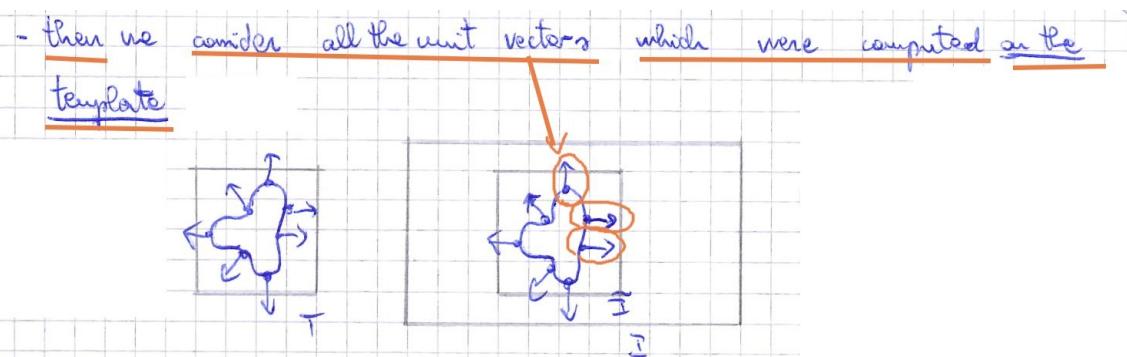


Figure 10.25

- and compare all the unit vectors which were computed on the template with the unit vectors that we compute on the target (=train) image;

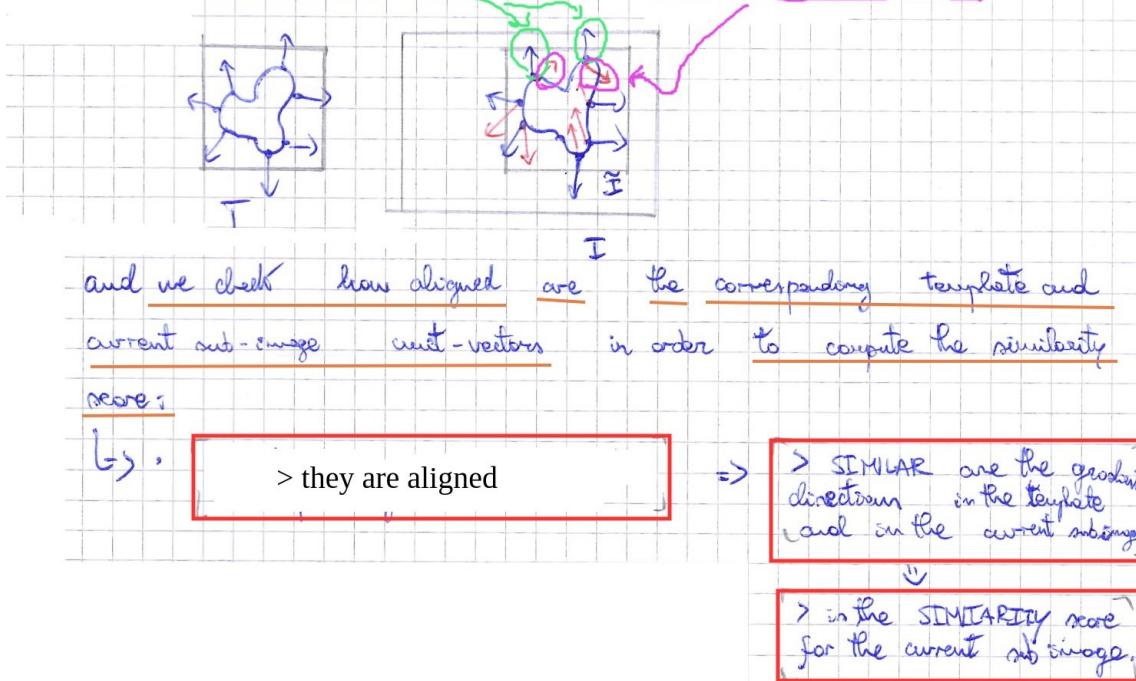


Figure 10.26

• Similarity Function let's first define the gradients and unit vectors at control points P_k

$$\textcircled{a} \quad G_k(P_k) = \begin{bmatrix} I_x(P_k) \\ I_y(P_k) \end{bmatrix}, \quad u_k(P_k) = \frac{1}{\|G_k(P_k)\|} \cdot \begin{bmatrix} I_x(P_k) \\ I_y(P_k) \end{bmatrix}, \quad k=1, \dots, n$$

$G_k(P_k)$ = gradient at the control point P_k in the template (in the model)

$u_k(P_k)$ = the unit vector associated with the control point in the template

b) this is the model that we store and search

b) control points associated with associated unit vectors.

$$\textcircled{b} \quad \tilde{G}_k(\tilde{P}_k) = \begin{bmatrix} I_x(\tilde{P}_k) \\ I_y(\tilde{P}_k) \end{bmatrix}, \quad \tilde{u}_k(\tilde{P}_k) = \frac{1}{\|\tilde{G}_k(\tilde{P}_k)\|} \cdot \begin{bmatrix} I_x(\tilde{P}_k) \\ I_y(\tilde{P}_k) \end{bmatrix}, \quad k=1, \dots, n$$

↳ then we do the same for the control points in the current sub-image which are denoted as \tilde{P}_k .

↳ we take their gradients and unit vectors

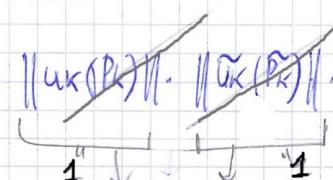
Figure 10.27

• Similarity Function

$$S(i, j) = \frac{1}{n} \sum_{k=1}^n u_k(P_k) \cdot \tilde{u}_k(\tilde{P}_k) = \frac{1}{n} \sum_{k=1}^n \cos \theta_k$$

↳ the similarity function is the sum of all the dot products between the unit vectors across the control points, divided by the number of control points.

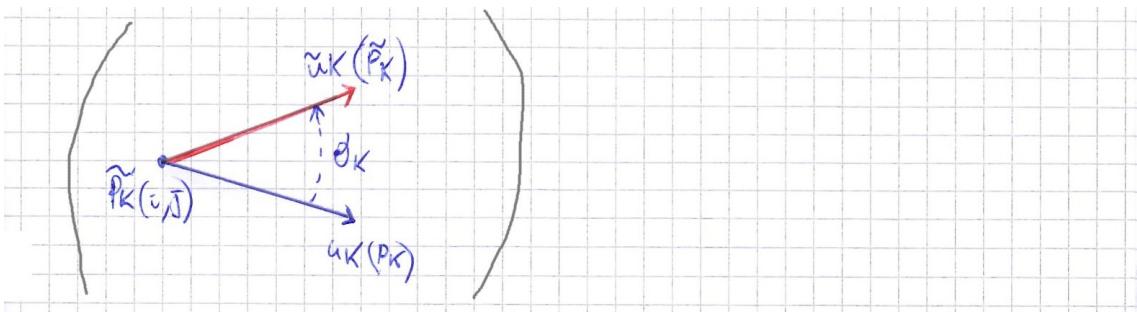
Figure 10.28

$\hookrightarrow \frac{1}{n} \sum_{k=1}^n u_k(p_k) \cdot \tilde{u}_k(\tilde{p}_k) =$
 $= \|u_k(p_k)\| \cdot \|\tilde{u}_k(\tilde{p}_k)\| \cdot \cos \theta_k = \frac{1}{n} \sum_{k=1}^n \cos \theta_k$


 Here 2 norms are ± 1 because, they are unit vectors.
 the similarity is the mean of the cosines between corresponding gradient directions.

b) range of this similarity $\rightarrow [-1; 1]$, because
 MAX VALUE:
 - if all the cosines are going to be ± 1 , which means that all gradients are perfectly aligned \Rightarrow the sum will be n , divided by $n \Rightarrow \frac{1}{n} \cdot n = 1$
 - MIN VALUE: \rightarrow when all the cosines $= -1 \Rightarrow \frac{1}{n} \cdot (-n) = -1$
 \Downarrow
 $+1 \rightarrow$ HIGHEST SIMILARITY
 $-1 \rightarrow$ SMALLEST SIMILARITY

Figure 10.29



What's good of this similarity function?

- it's INVARIANT to INTENSITY CHANGES, because we rely on gradients, gradients rely on differences, and differences cancel out biases \Rightarrow we're invariant to a bias. And we are invariant to a linear change.

Figure 10.30

• Why don't we extract edges on the target image?

↳ because:

- it's faster
- this method is a bit invariant to intensity changes \Rightarrow it makes sense to consider it when we have light changes
 \Rightarrow if there are light changes, how can you set the edge detection threshold on the target image?

↳ Whatever the algorithm you use for edge detection (Sobel,anny,...) at the end you have a threshold. And it is easy to set this threshold in the template image, but then on the target image we have changes \Rightarrow the threshold won't be good enough.

↳ we'll miss edges

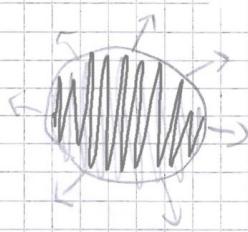
Figure 10.31

• What's good with Shape-based Matching?

1. It's invariant to intensity changes and
2. you don't need to extract edges on the target image.

A couple of simple variations:

↳ We can use a MORF Robust Similarity Function, which takes the absolute value rather than just the sum of the cosines, because this solves the following problem:



- a) at training time we could have our object which is darker than the background
- b) the gradients just point outward

Figure 10.32



but then, we could search our object in the target image with a darker background. And if the background is darker, then the gradients will point inward.

→ Taking the absolute value makes the similarity score invariant to a so called Global contrast polarity inversion.

$$S(i, j) = \frac{1}{n} \left| \sum_{k=1}^n u_k(p_k) \cdot \tilde{u}_k(\tilde{p}_k) \right| = \frac{1}{n} \left| \sum_{k=1}^n \cos \theta_k \right|$$

↳ we can do even better, if, rather than taking the absolute value of the overall function, we instead apply the absolute value to each individual term.

$$S(i, j) = \frac{1}{n} \sum_{k=1}^n |u_k(p_k) \cdot \tilde{u}_k(\tilde{p}_k)| = \frac{1}{n} \sum_{k=1}^n |\cos \theta_k|$$

This similarity function is invariant to a Local Contrast Polarity inversion.

Figure 10.33

10.3 GHT

- GHT; Generalized Hough Transform

↳ How the HT can be generalized to whatever shapes (non-analytical shapes):

- we have object which cannot be defined through equations.

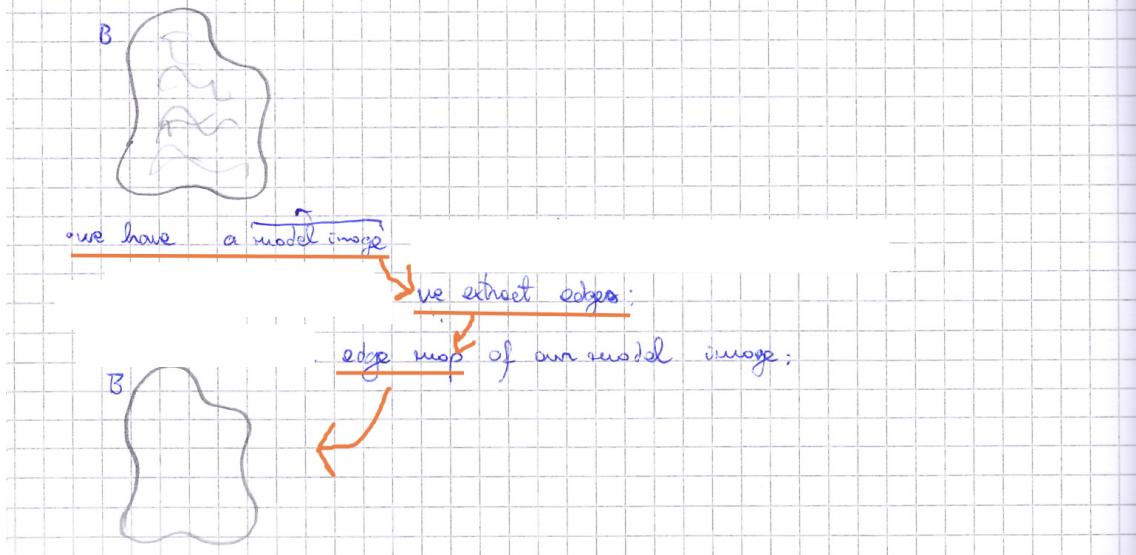
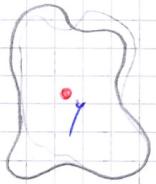


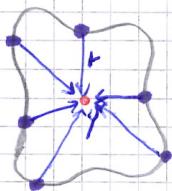
Figure 10.34

Now we found a reference point for this shape \rightarrow THE **BARYCENTER** of the shape (in this case the barycenter across all edge points):



$\cdot \gamma = \text{barycenter}$

Then (and this is the clever idea), for each of all edge points we compute a vector from the edge point to the barycenter (this vector is denoted by r)



r -vectors are also known as **JOINING VECTORS**.

\bullet \circlearrowleft = edge point
 \nearrow = joining vector

Figure 10.35

and we store all these joining vectors into a structure called **R-Table**

KEY ELEMENT:

a) The R-Table is indexed through the gradient's direction at the given point.

b) for each edge

point we record:

-the r -vector

-and ALSO the GRADIENT DIRECTION as an angle (\because we don't take into account the magnitude of the gradient, but just its direction as an angle).

then we quantize this angle into steps (e.g. 10°) and we store all the r -vectors into the R-Table.

Figure 10.36

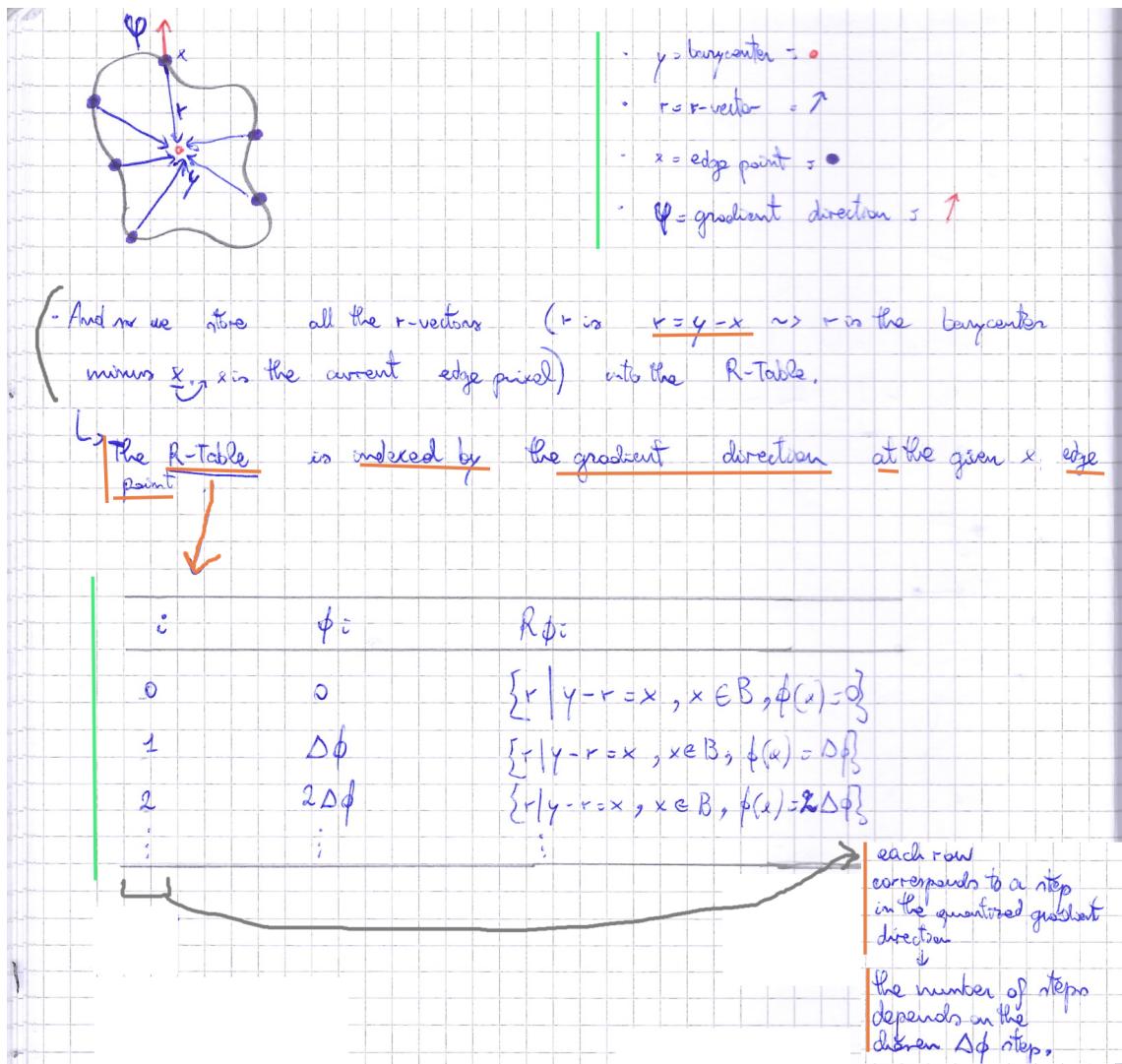


Figure 10.37

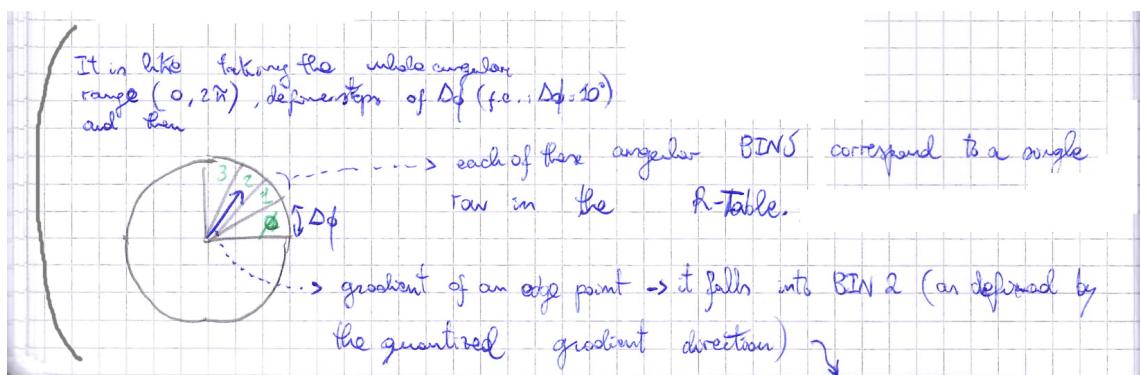


Figure 10.38

(b) we store the corresponding r-vector into the 2nd row (because of 2nd BIN) of the R-Table

b) All r-vectors are vectors stored into the row of the R-Table that corresponds to the quantized gradient direction at the given edge point.

b) in a certain row ^{case} we store more than 1 r-vectors if we have more edge points that show the same gradient direction (given by quantization step $\Delta\theta$)

Figure 10.39

→ have built, the model of the shape we seek to find in the image
↳ the model of the object in the R-Table

Figure 10.40

• Online Phase of the OTI → the phase in which we perform detection:
↳ we do edge detection first

Figure 10.41

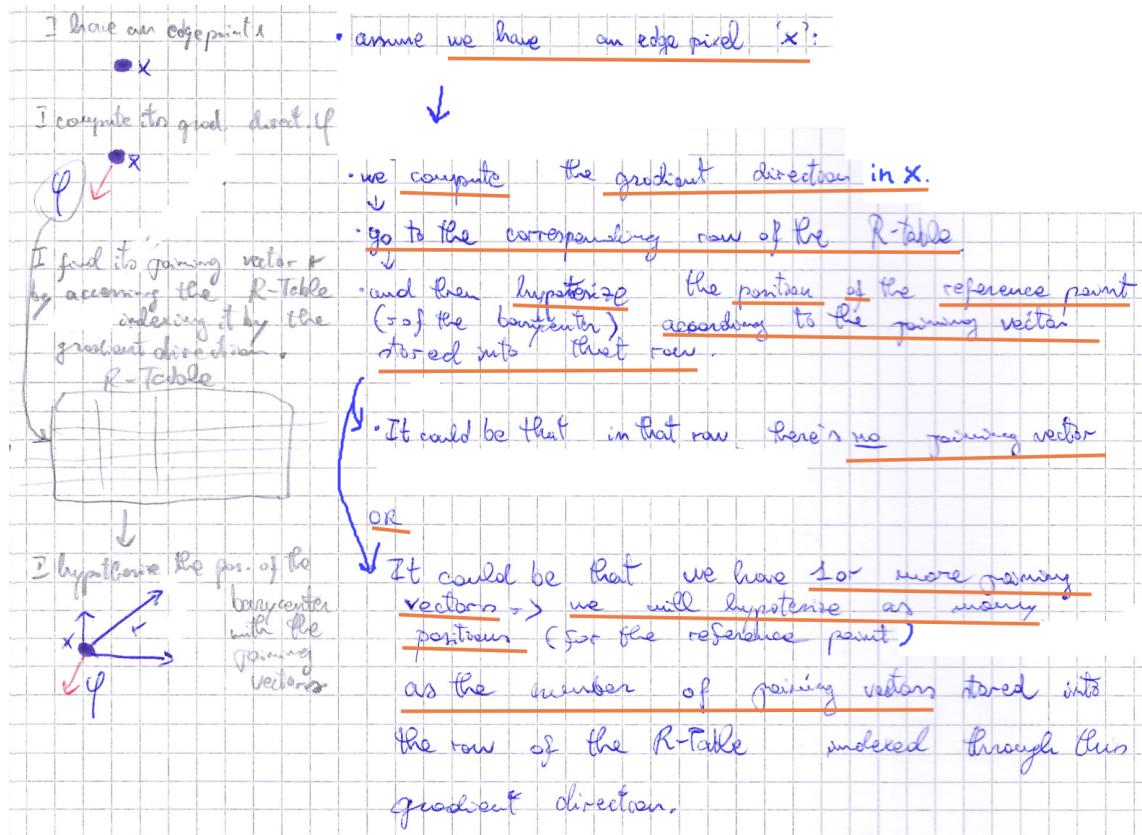


Figure 10.42

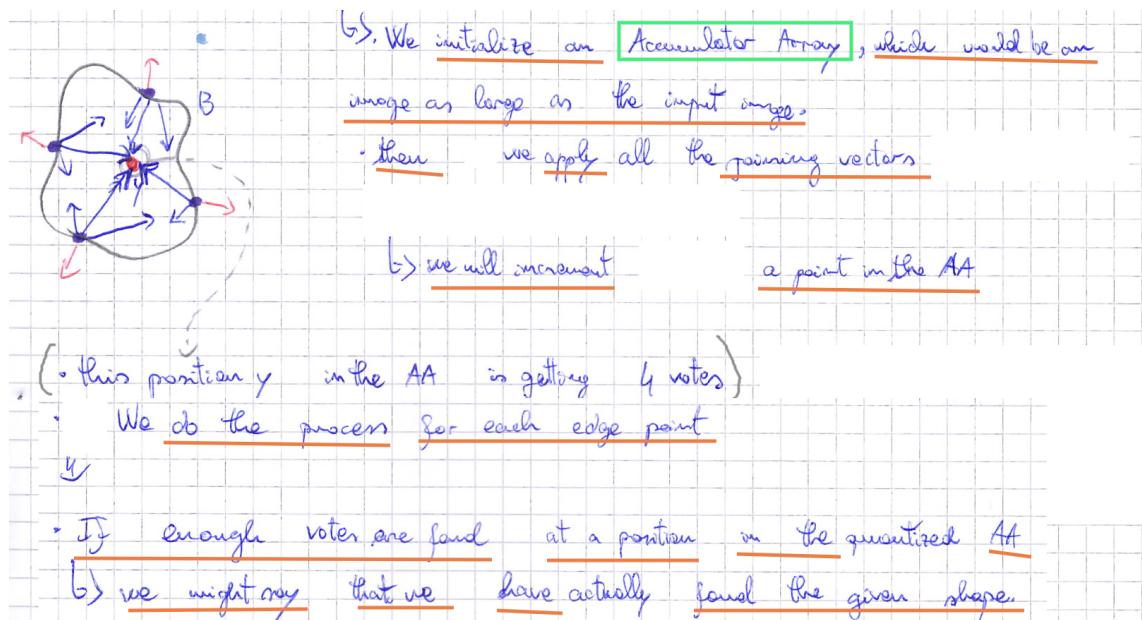


Figure 10.43

Difference with the HT:

- Here the voting process takes place through the guiding vectors and the R-table rather than an equation (as it is the case of the analytical HT). And this is because we don't have any equations.

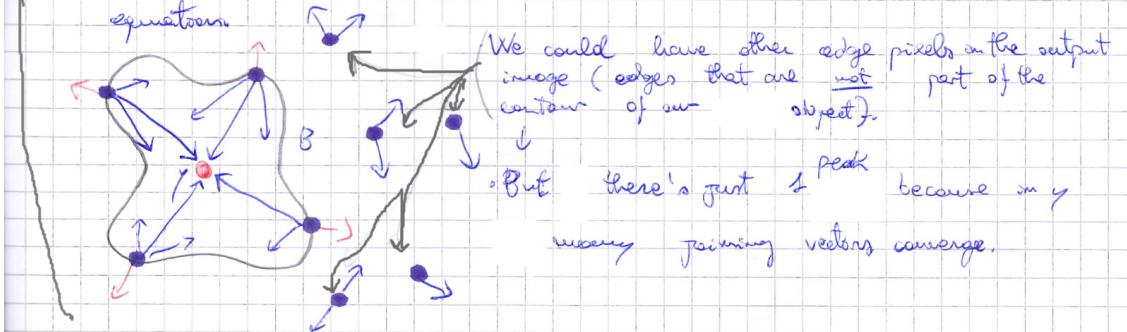


Figure 10.44

Once all edge pixels voted we've just the AA and we need to find the peaks in the AA:

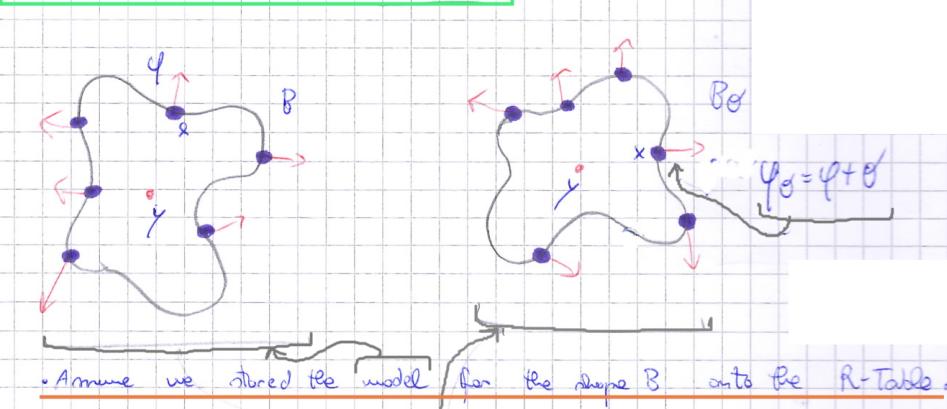
- if we have 1 peak \Rightarrow there's just 1 instance of the shape
- " many peaks \Rightarrow there are many instances"

NMS after voting = finding peaks

thresholding \rightarrow to prune out small peaks and keep only high peaks

Figure 10.45

• What is the shape in rotated?



• Assume we stored the model for the shape B onto the R-Table.

• And assume that the shape is present in the target image, but it is rotated by θ .

↳ the gradient of the edge point x will have a direction equal to $\phi + \theta$ instead of ϕ .

Figure 10.46

↳ if we index this gradient in the R-Table we wouldn't find the correct row because $\phi + \theta \neq \phi$

ϕ_θ

↳ we should index the R-Table in position $\phi - \theta$ to find the correct feature vector.

Figure 10.47

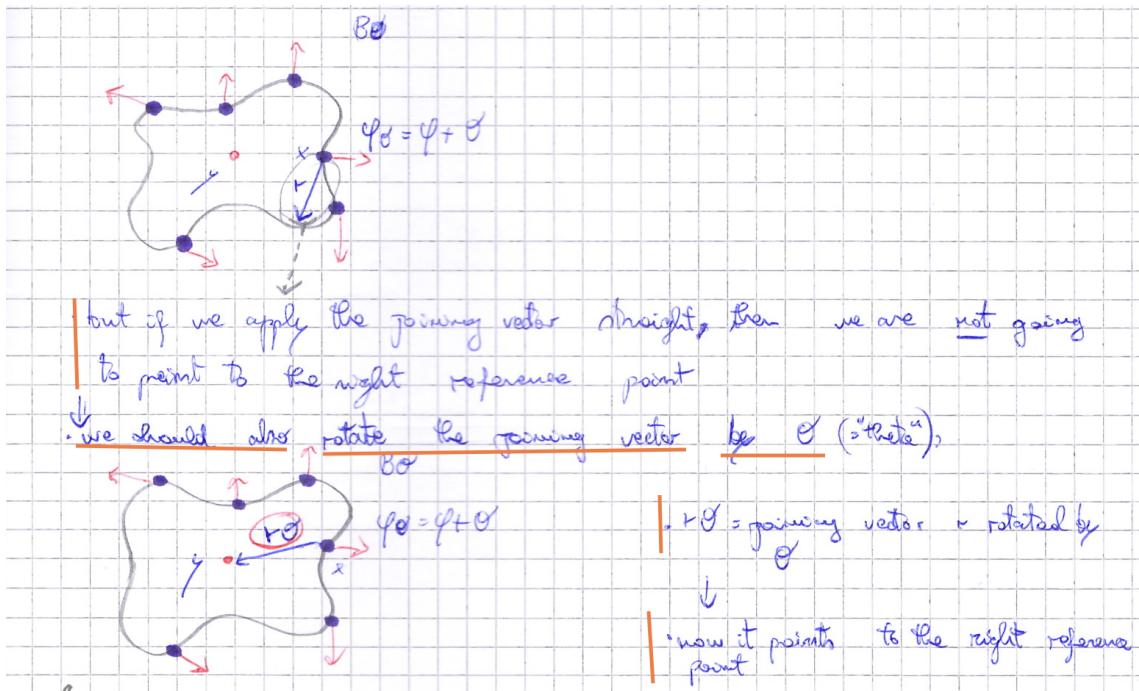


Figure 10.48

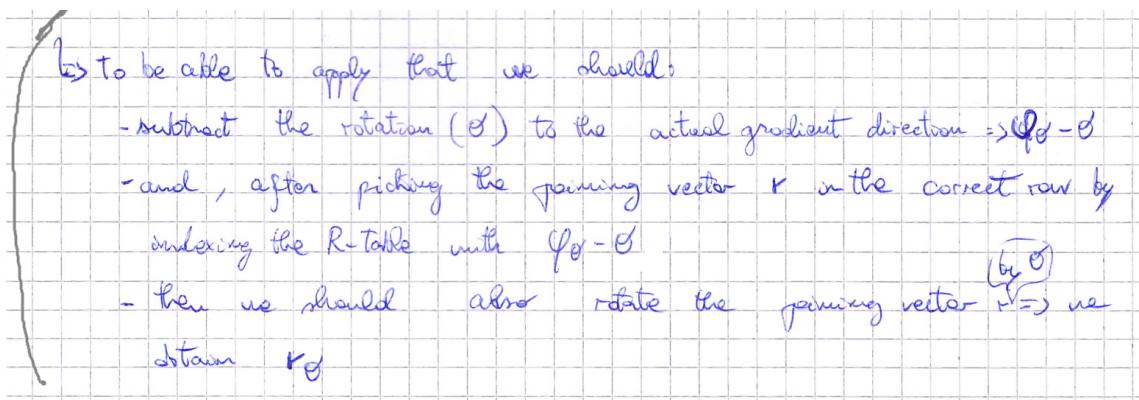


Figure 10.49

• Problem:

↳ we don't know this unknown rotation θ .

• we have to do all the previous process for all hypothesis of rotations \Rightarrow we should quantize the rotation range and try all rotations

• cast votes under rotation hypothesis θ_1 , and this creates a layer in AA; then cast votes under rotation hypothesis θ_2 and this creates another layer in the AA; and so on and so forth across the whole range of quantized rotation.

→ This is typically done when the unknown rotation range is small (e.g., -10° to 10°).
It is done only when the vote can find the object under small rotations, because otherwise it gets too slow.

Figure 10.50

• What if we want also to handle scale change?

we would have to

pick the growing vectors ← and scale them in order to note to the correct reference point.

• But scale how much?
↓

• If the scale change is unknown, then we should quantize the scale range and try everything.

(↳ we have a stack of AA layers for each of the scale hypothesis)

but only under a small scale change range.

• And if we have both Rotation and Scale Changes?

↳ We would have to explore the unknown rotation and scale change, ⇒ too much.

Figure 10.51

10.3.1 GHT + SIFT

• In practice, the GHT can't be used when:

- you have rotation and scale changes together because it would explode computationally



GHT based →

- NOT on edge pixels (⇒ NOT on edge features)

- BUT ON LOCAL INvariant FEATURES (SIFT features f.e.)

↳ then we can pursue object detection invariantly to any rotation, translation and scale change.

Figure 10.52

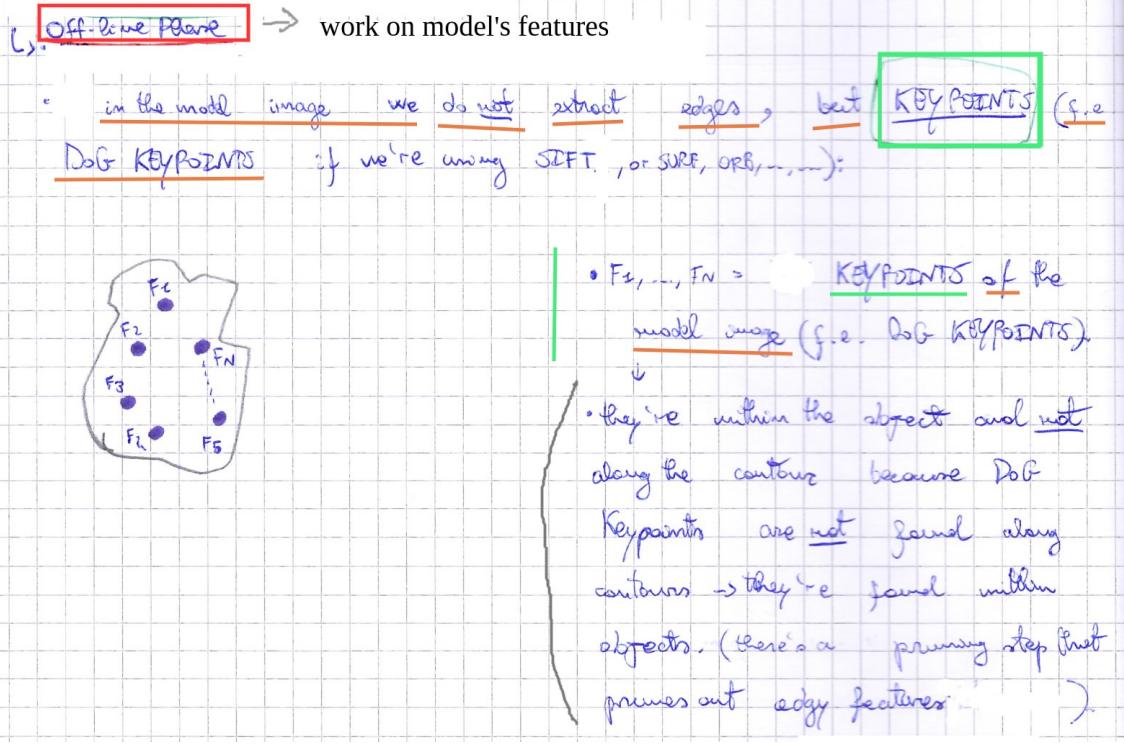


Figure 10.53

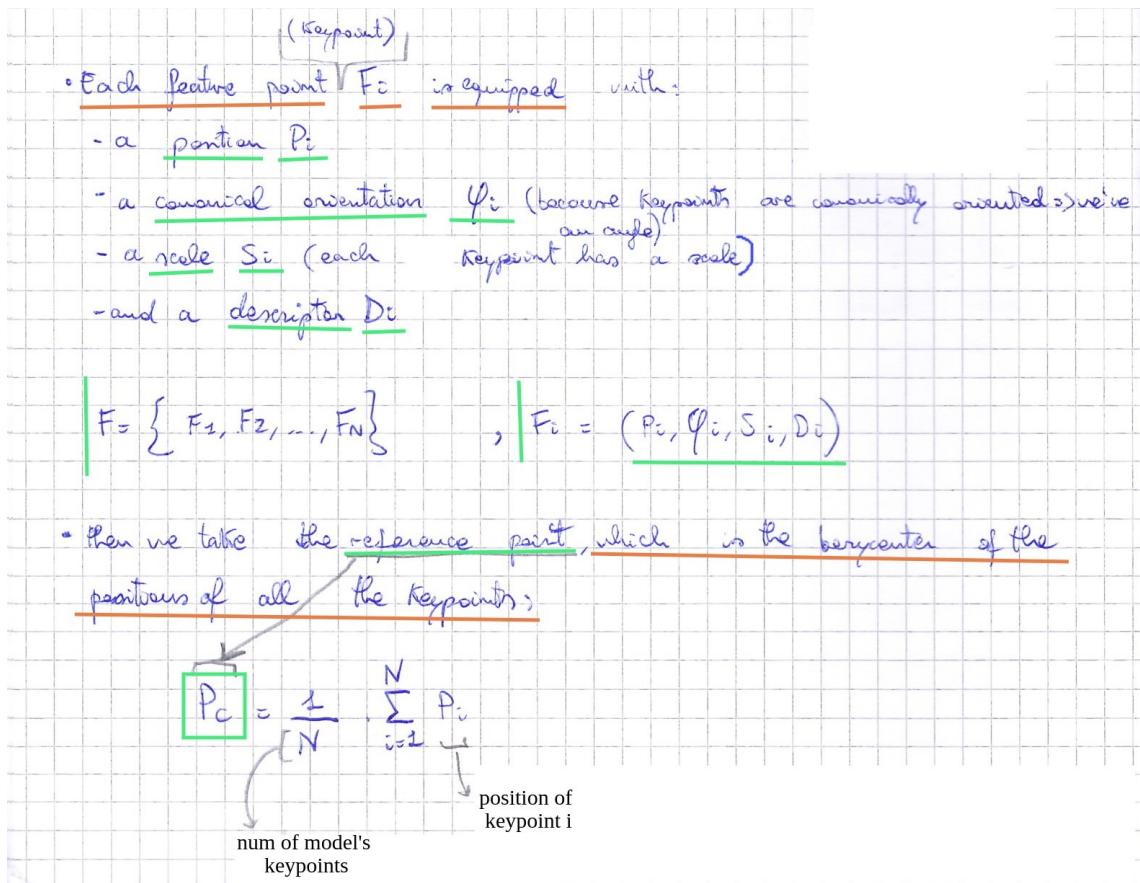
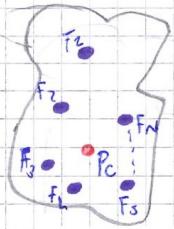


Figure 10.54



- and

↳ and we compute the joining vectors for all Keypoints to the barycenter

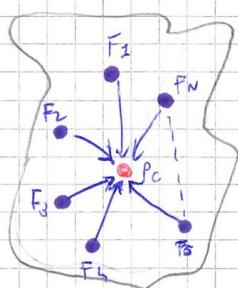
$$P_c = \frac{1}{N} \cdot \sum_{i=1}^N P_i$$

barycenter

$$\Rightarrow V_i = P_c - P_i$$

position of
the keypoint i

joining vector from keypoint i
to barycenter P_c



V_i = joining vector V_i

Figure 10.55

Now we can equip each feature (= each Keypoint) in our model with the corresponding joining vector:

$$F_i = (P_i, Q_i, S_i, D_i, V_i)$$

↳ Here we don't have any R-Table. Our model consists of all the Keypoints ↴

Figure 10.56



Figure 10.57

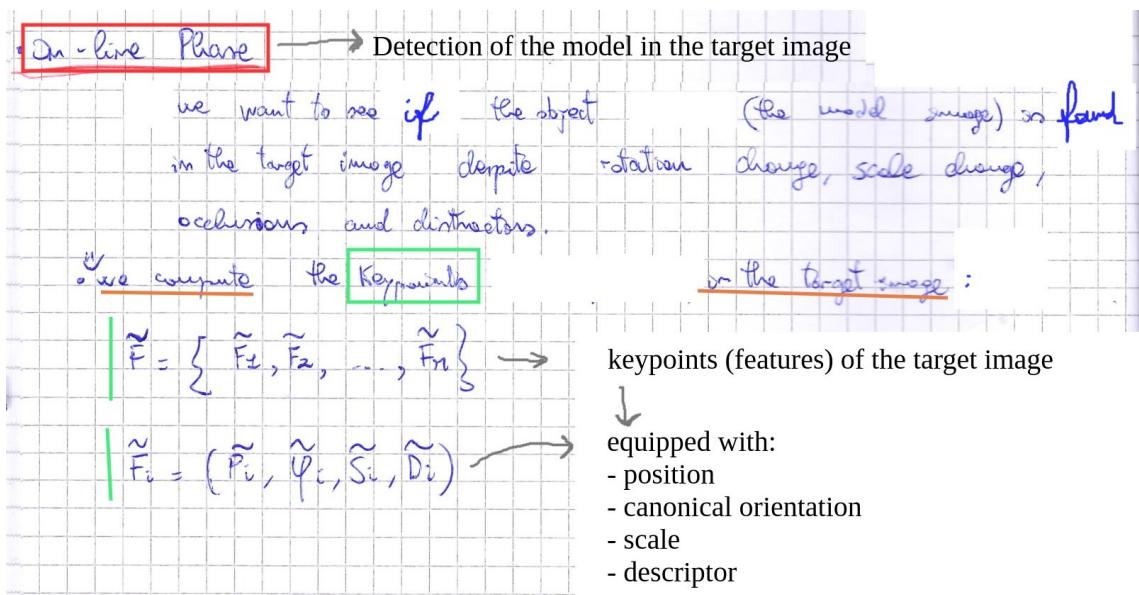


Figure 10.58

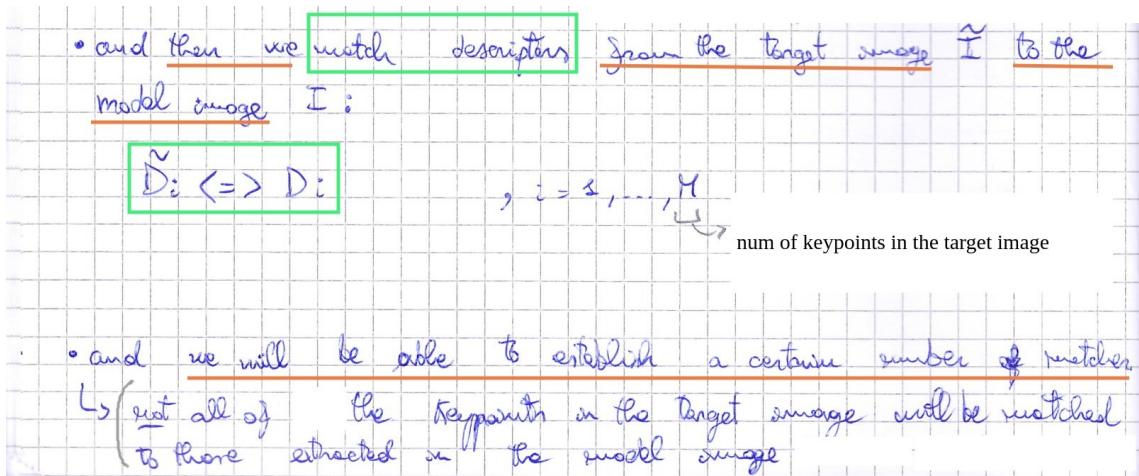


Figure 10.59

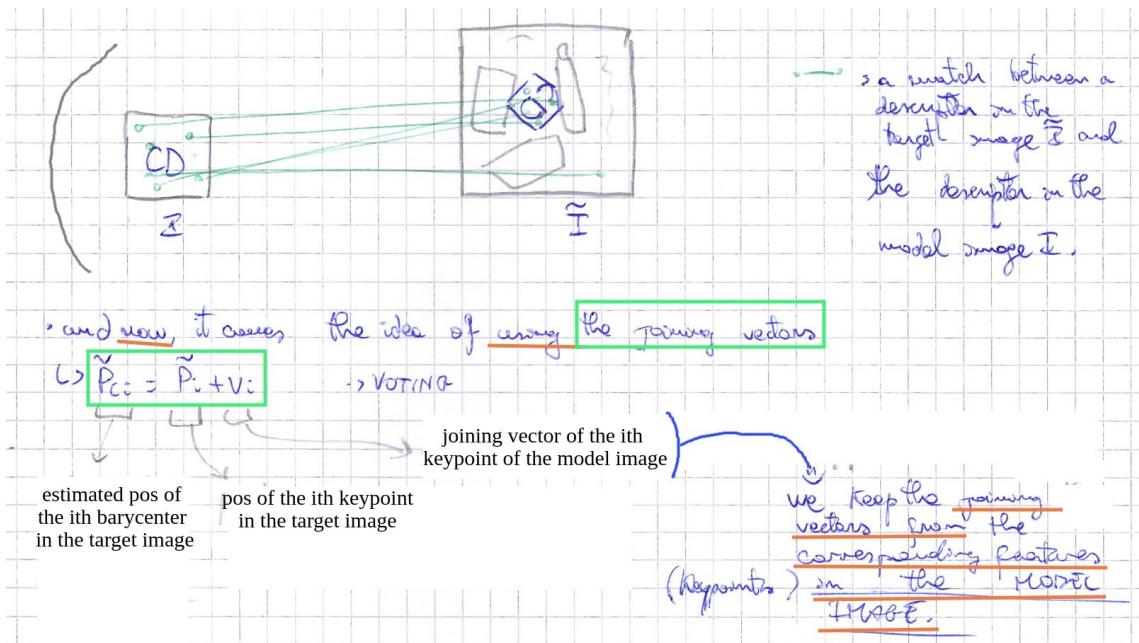


Figure 10.60

- ↳ we do that for each and every keypoint of the target image
 - that gets a match with a keypoint of the model image.
 - ↳ then you can't vote in the target image for the position of the barycenter
 - ↳ then you find points in the AT.
- Difference with the usual GHT:
- || the difference with the usual GHT is that now the votes are cast based on MATCHING DESCRIPTORS and not based on accompanying edge directions
 - ↳ And Matching Descriptors is a much stronger criterion.
 - ↳ If 2 descriptors match, then it's very likely that the features (the keypoints) are really corresponding

Figure 10.61

↳ So we can pick more robustly (not more robustly than we would have done with edges) the joining vectors.

Figure 10.62

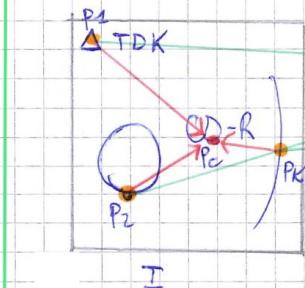
• Similarity Invariant Voting → Invariance to rotation and scale changes

L,

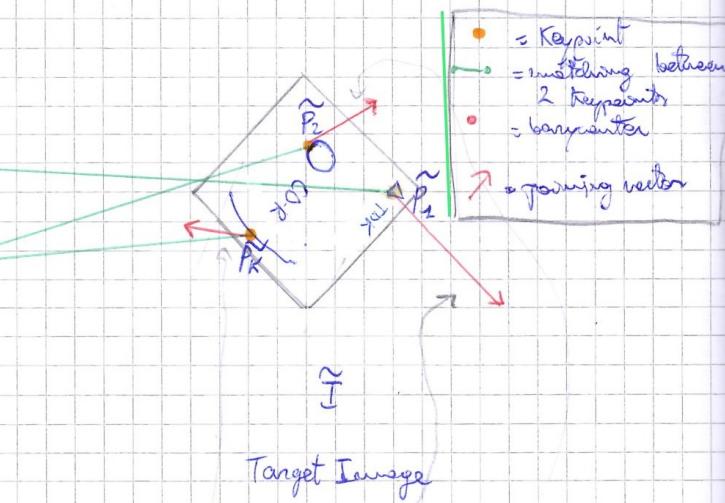
orientations of

the information between the scales and
the keypoints are stored alongside
the Keypoints

• Example:



Model Image



Target Image

a) L, assume we matched:

- P_1 with \tilde{P}_1
- P_2 " " \tilde{P}_2
- P_K " " \tilde{P}_K

b) P_1 matches with \tilde{P}_1 \Rightarrow we need to pick the joining vector for P_1 and apply it to \tilde{P}_1 \Rightarrow I obtain this.

- the same for $P_2 - \tilde{P}_2$ and $P_K - \tilde{P}_K$

Figure 10.63

b) the matches are good, but the train image is rotated w.r.t the model image \Rightarrow the journey vectors applied to $\tilde{P}_1, \tilde{P}_2, \tilde{P}_k$ are not pointing coherently toward the reference point (= toward the barycenter)

\downarrow
we should rotate all journey vectors by the amount of rotation that exists between the model image and the target image.

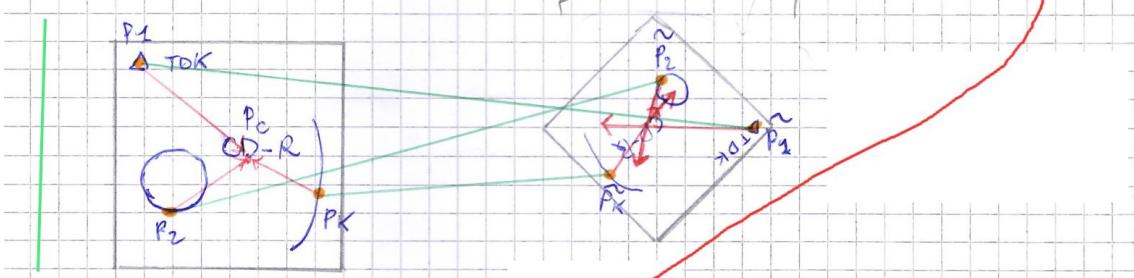


Figure 10.64

↳ we take the difference of the canonical orientation between two Keypoints that are matching:

$$\tilde{F}_i = (\tilde{P}_i, \tilde{q}_i, \tilde{s}_i, \tilde{d}_i) \quad \langle \approx \rangle, \quad F_i = (P_i, q_i, s_i, d_i, v_i) \rightarrow \text{matching keypoints}$$

Let me have

- $\hat{F}_1 \rightarrow$ (Keypoint) on the train sample
 - $F_2 \rightarrow$ - - - model " "
 - $\hat{F}_2 (\Leftrightarrow F_1) \rightarrow \hat{F}_2$ is matching with F_1

$$\Delta \varphi_i = \tilde{\varphi}_i - \varphi_i$$

\rightarrow if 2 features

are watching

It is exactly the rotation undergone by the object.

diff. between their canonical orientation
in the AMOUNT of ROTATION.

Figure 10.65

now the viewing vectors \vec{v}_i to the keypoints of the target shape are well rotated, but we have still a problem:
 \hookrightarrow the scale change
 \hookrightarrow we need to scale all these viewing vectors properly.
 \hookrightarrow we scale the viewing vectors by the ratio between the scales of the features (of the keypoints).
 D_i this is the amount of scaling

Figure 10.66

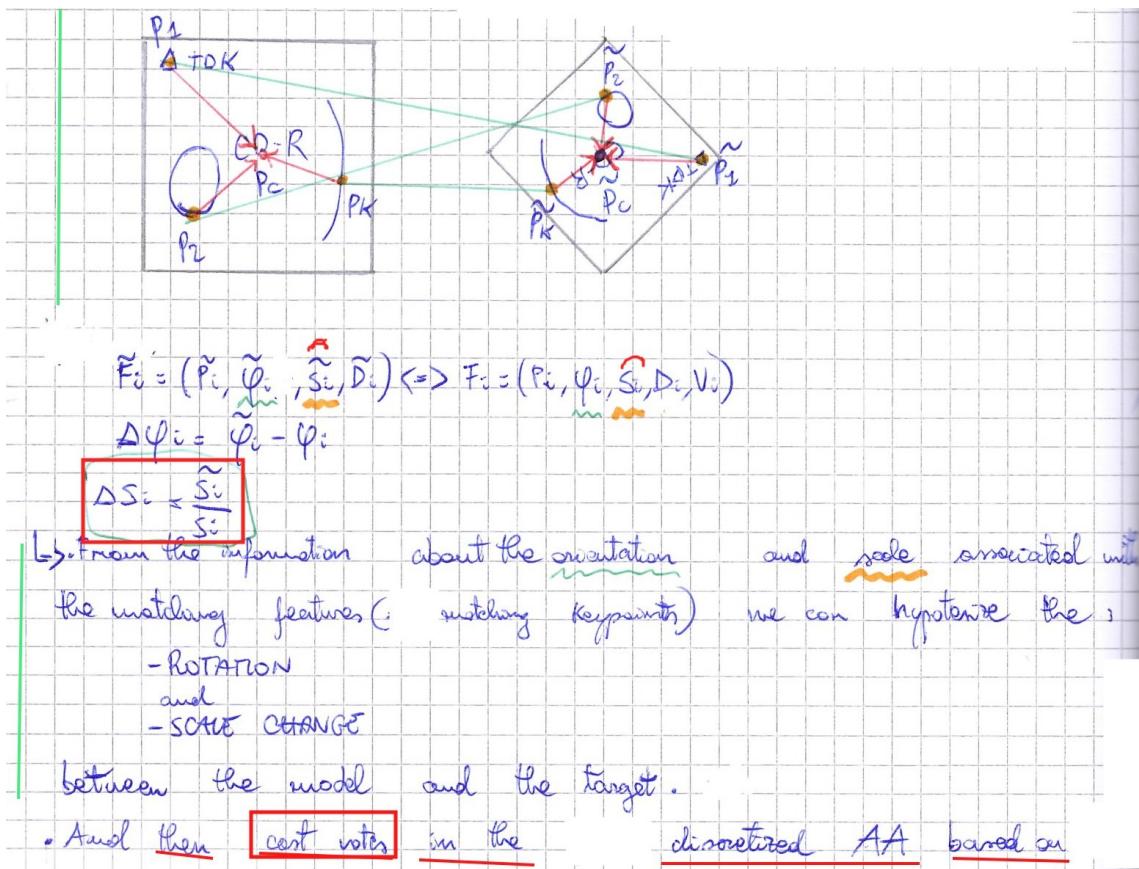


Figure 10.67

The Rotation and Scale hypothesis.

- We can cast votes **INVARIANTLY** wrt any similarity transformations.
- And this is just for free. We don't have to do anything in particular, because as long as we match features we have the rotation and scale informations for matching features, and just apply that to the joining vector associated to the model feature,

$$\tilde{P}_{ci} = \tilde{P}_i + \Delta s_i \cdot R(\Delta\Phi_i) \cdot V_i$$

P_{ci_tilde} = position of barycenter
 of an instance of model in target image
 P_i_tilde = pos of ith keypoint in target image
 Δs_i = amount of scale
 $\Delta\Phi_i$ = amount of rotation
 V_i = joining vector of ith model keypoint

- very robust
- and fast
- and effective

object detection approach.

Figure 10.68

b)

The GHT with SIFT:

- we cast votes after matching Keypoints (features)
- ~ ~ ~ INVARIANTLY wrt a similarity due to the scale and rotation informations already embedded onto the Keypoints
- once we've created the discretized AA \rightarrow we need to find the peaks.

Figure 10.69