Computer Vision and Image processing

IPCV

Luca Marini

Contents

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).

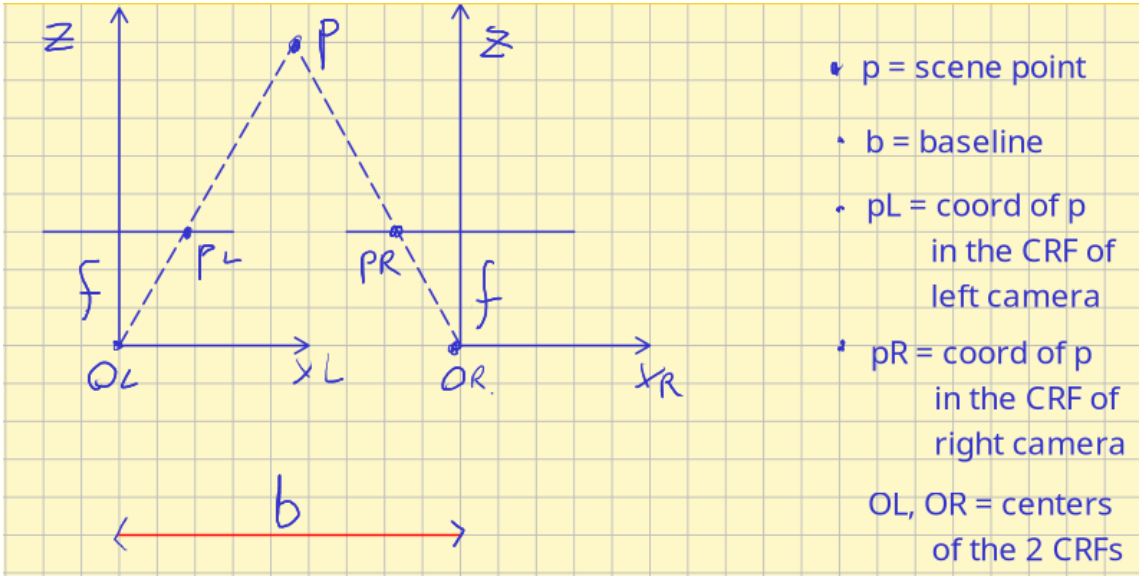


Figure 1.1

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

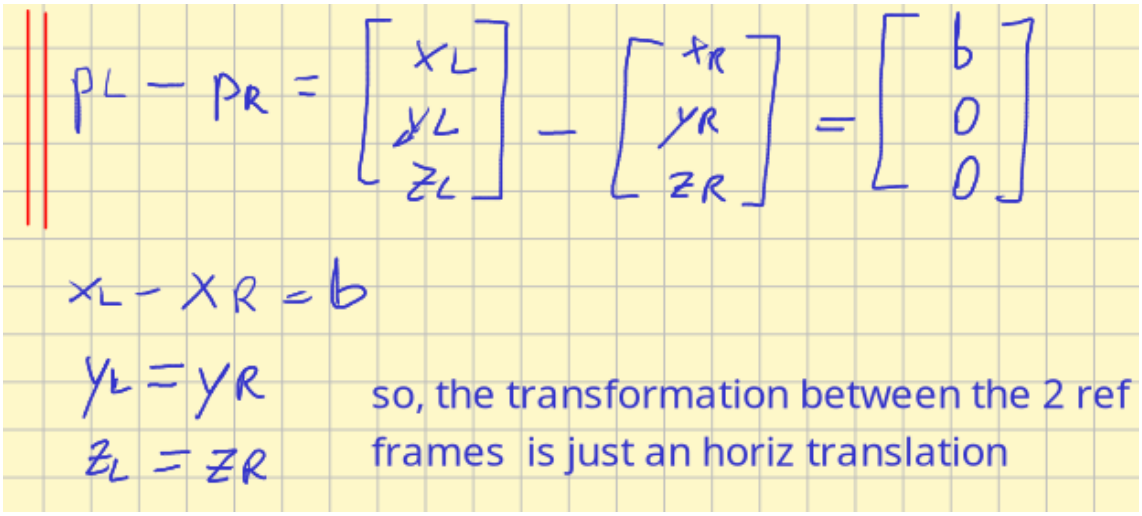


Figure 1.2

- apply Perspective Projection Equations to get image coords:

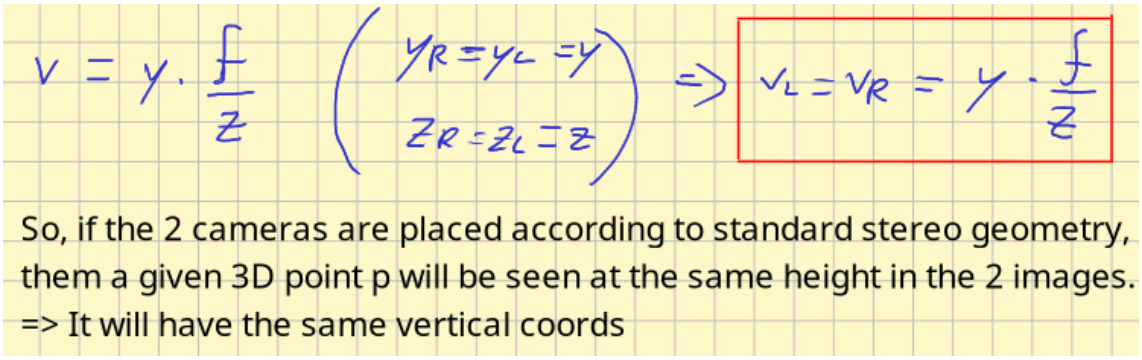


Figure 1.3

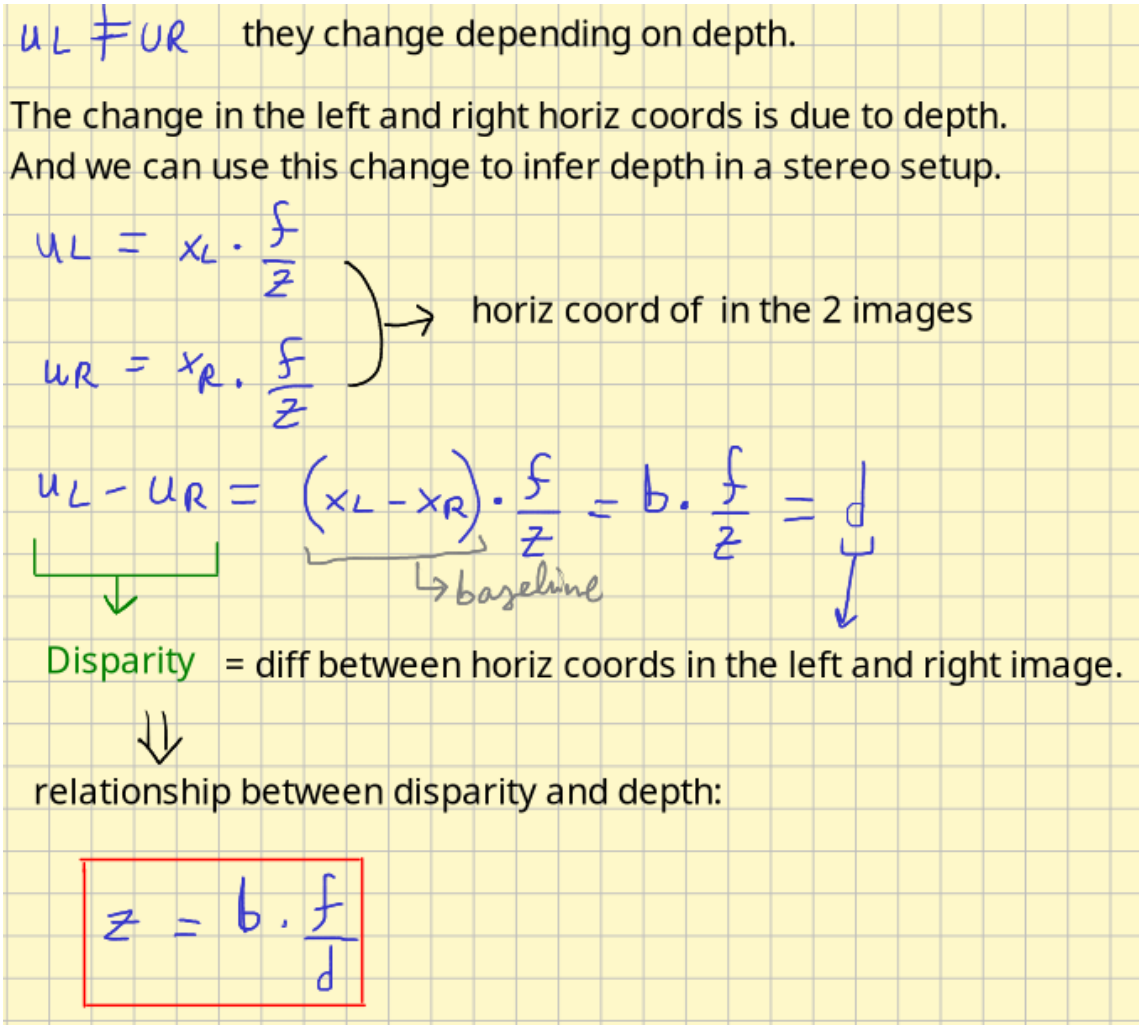


Figure 1.4

- 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 (uL, vL) and (uR, vR).
- And we can look for correspondences in 1D space because we know that **vL** = **vR**.
- So, if I have the point pL in the left image plane, then the point in the right image plane will be at the same height of the point pL.
- So, I don't search throughout the whole image, but just on the same height.

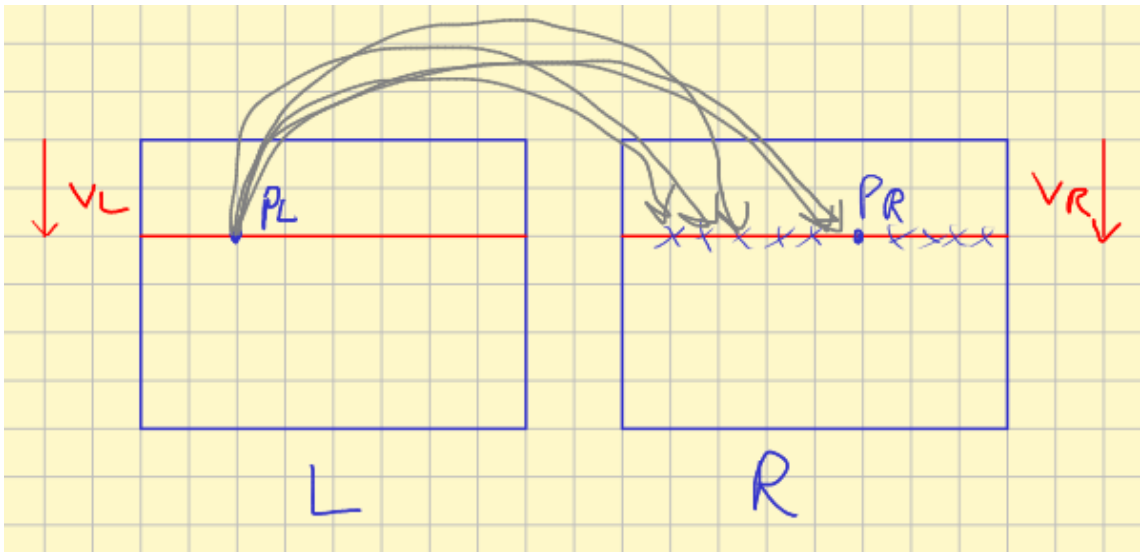


Figure 1.5

- Possible problem: maybe a point in the left image is not in the right image, due to occlusions, light chages, ecc.
- How to find correspondences? This is the key problem in Stereo Vision
- Take a neighboorhood (a window around) pL 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):

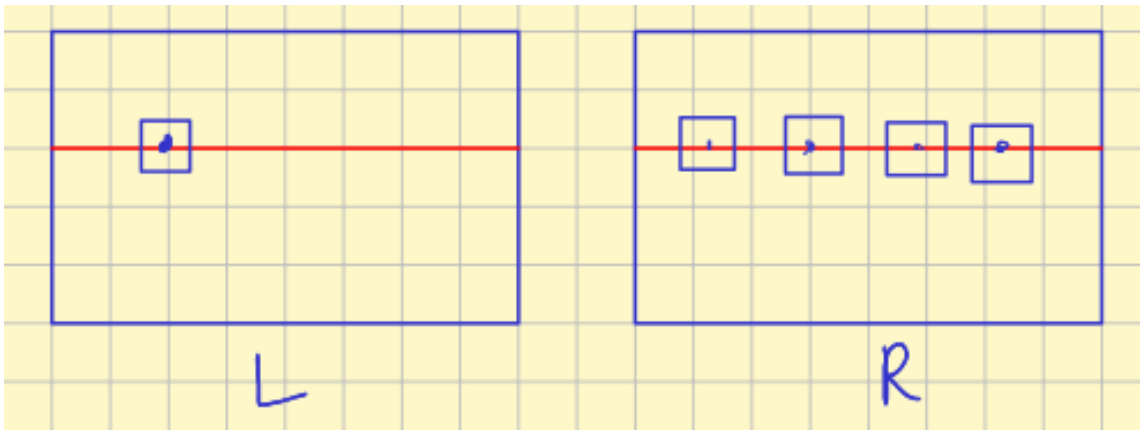


Figure 1.6

- What if the 2 cameras are arbitrarily arranged?

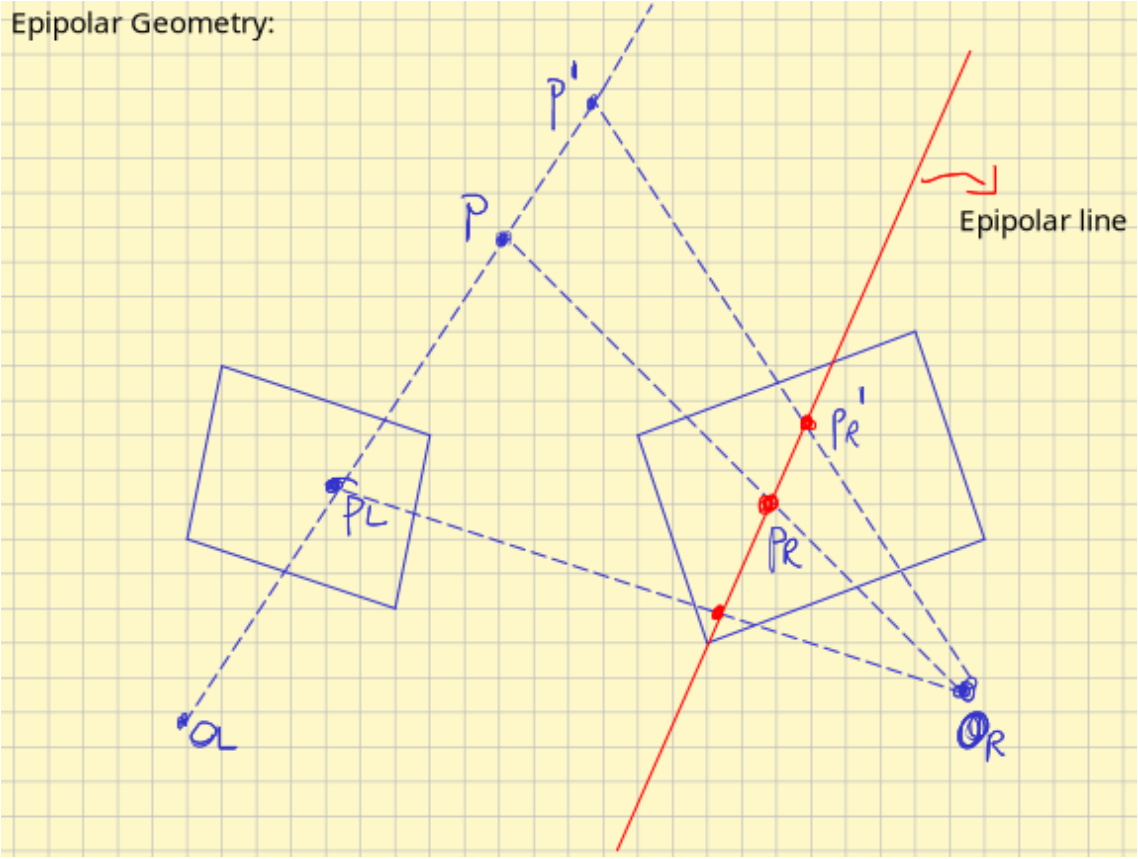


Figure 1.7

- I still search the correspondent point in a 1D space, but in an oblique line (less efficient)
- we know that, given pL, its corresponding 3D point has to be on the dashed line
- so, the corresponding point pR 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.

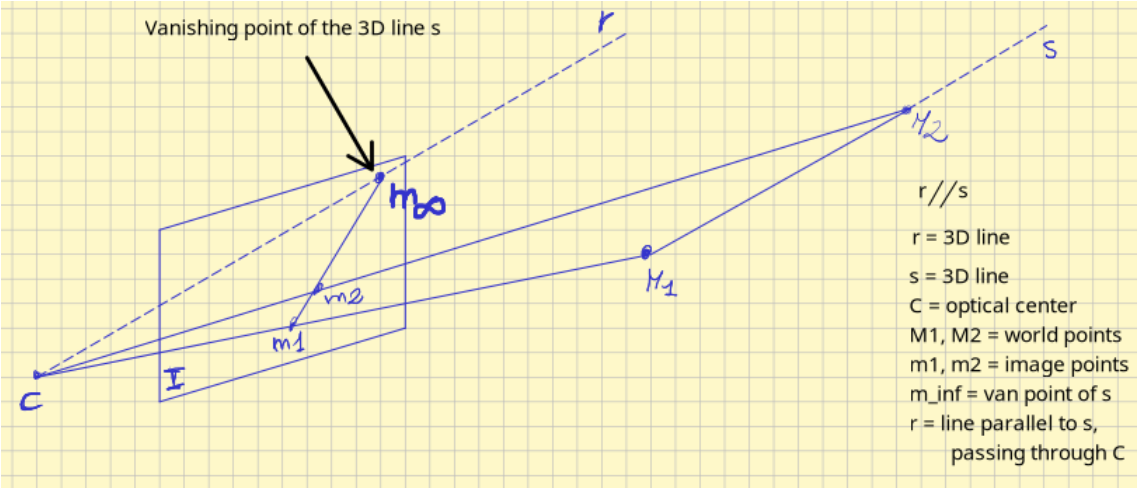


Figure 1.8

- Project world points into the image with persp proj (M1 -> m1) => by drawing line from M1 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:

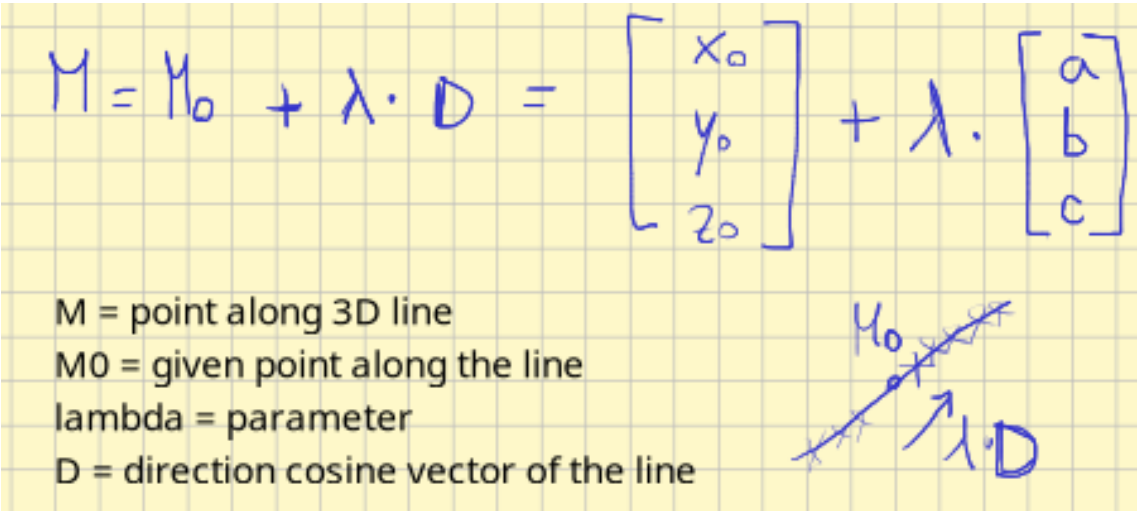


Figure 1.9

- 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:

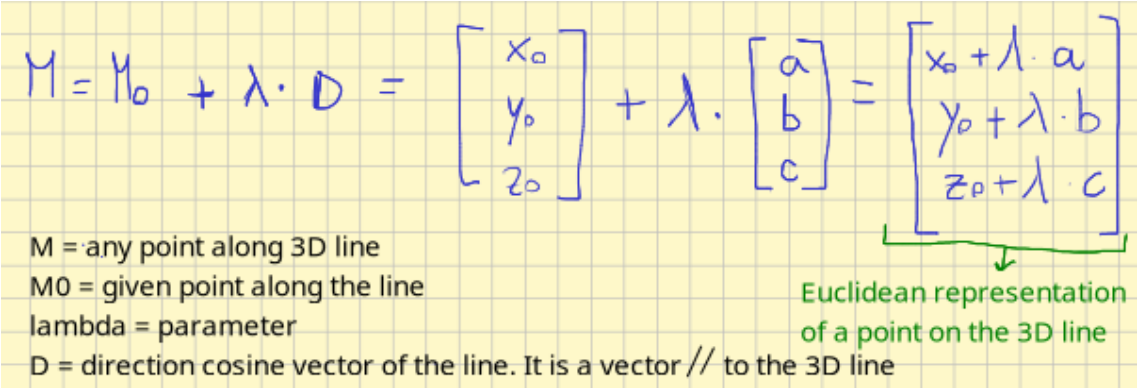


Figure 1.10

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

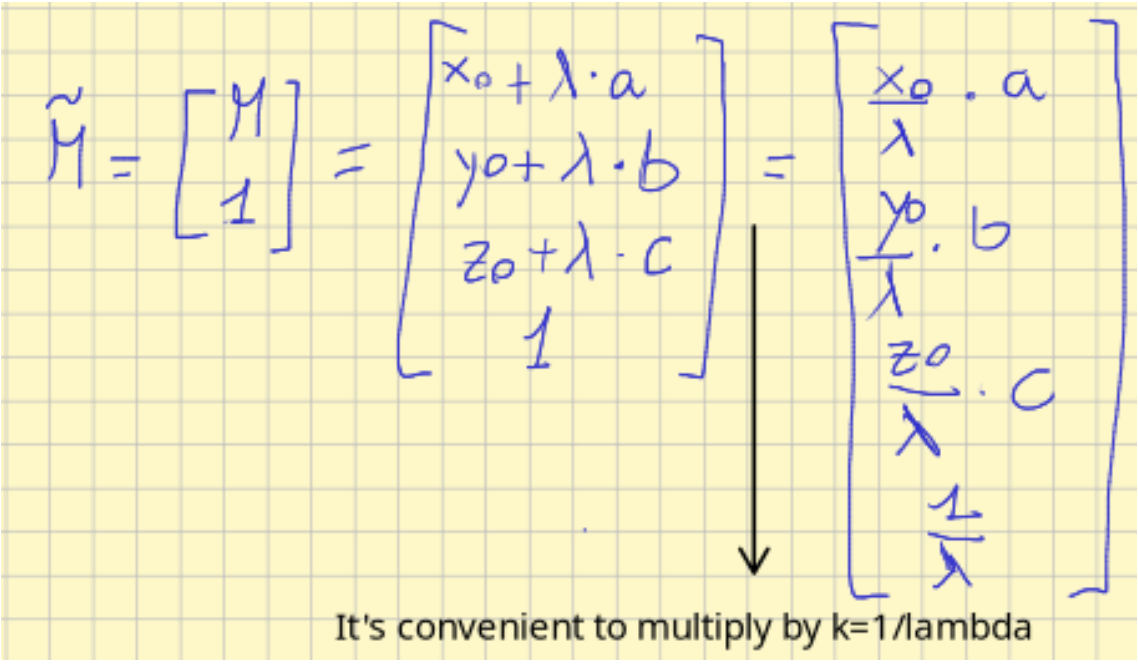


Figure 1.11

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

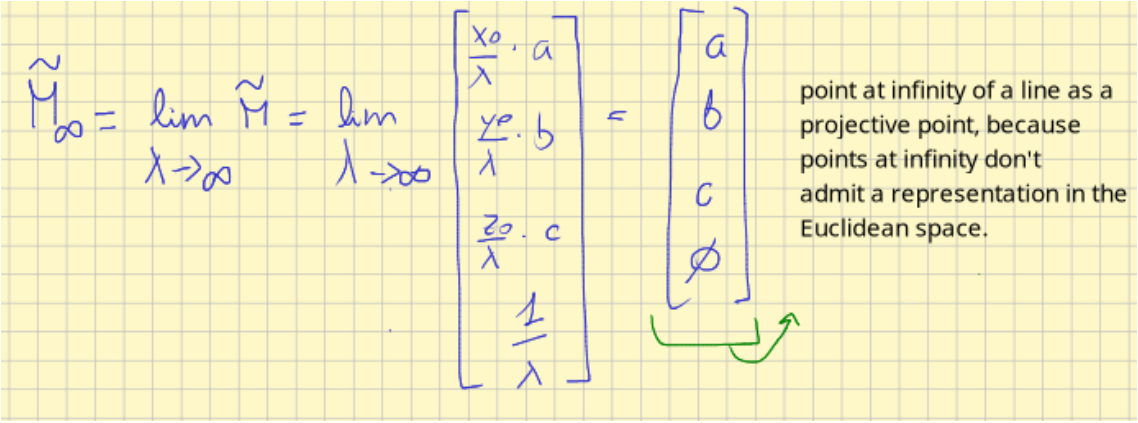


Figure 1.12

- 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.

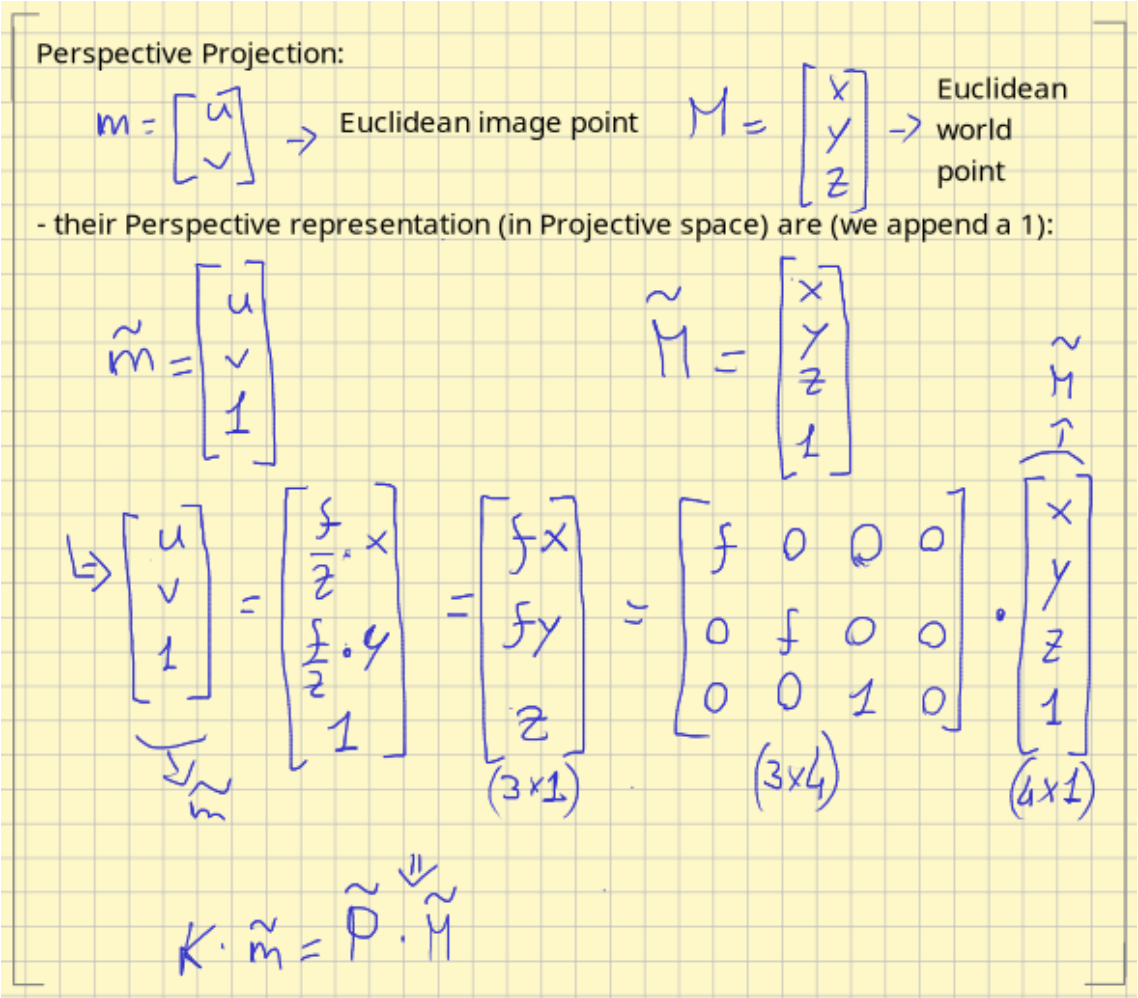


Figure 1.13

- 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:

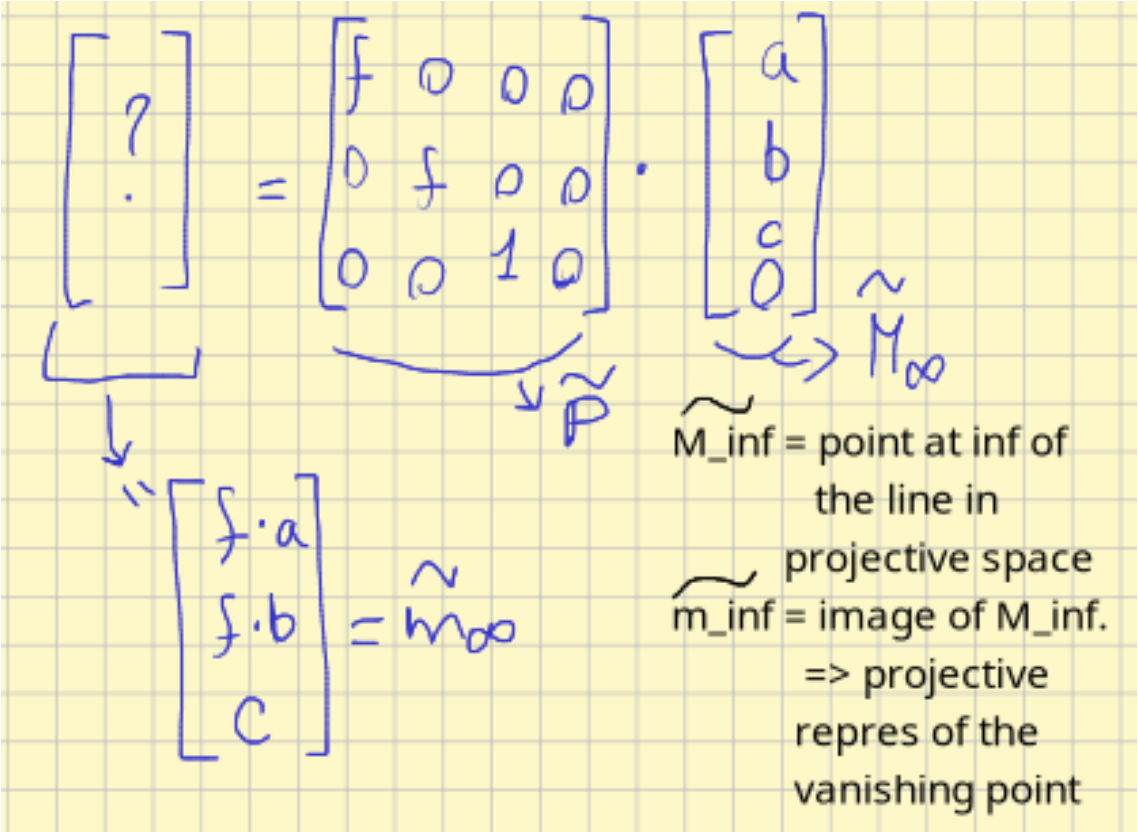


Figure 1.14

- Now, I go back to the corresponding Euclidean vanishing point:
  - I divide by the last coord and delete the last coord:

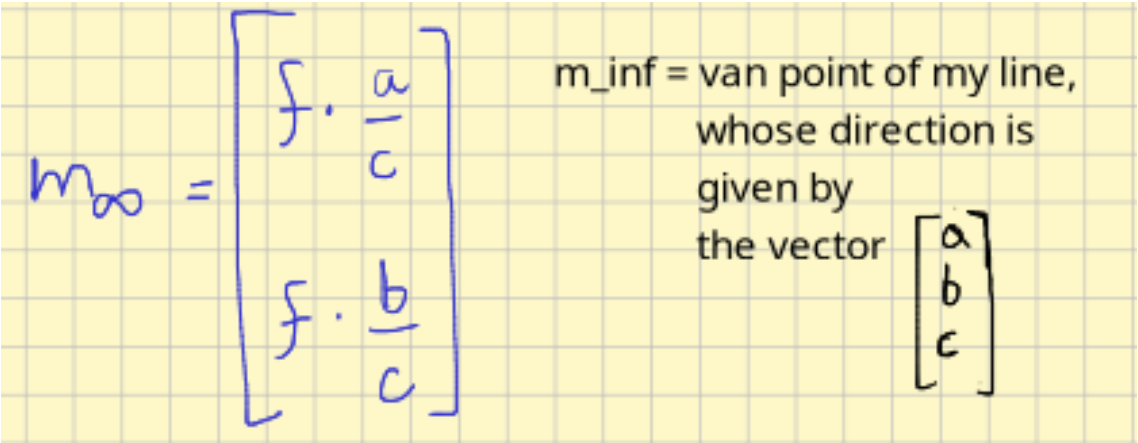


Figure 1.15

1.3 Camera parameters

1.3.1 First let's talk about Noise

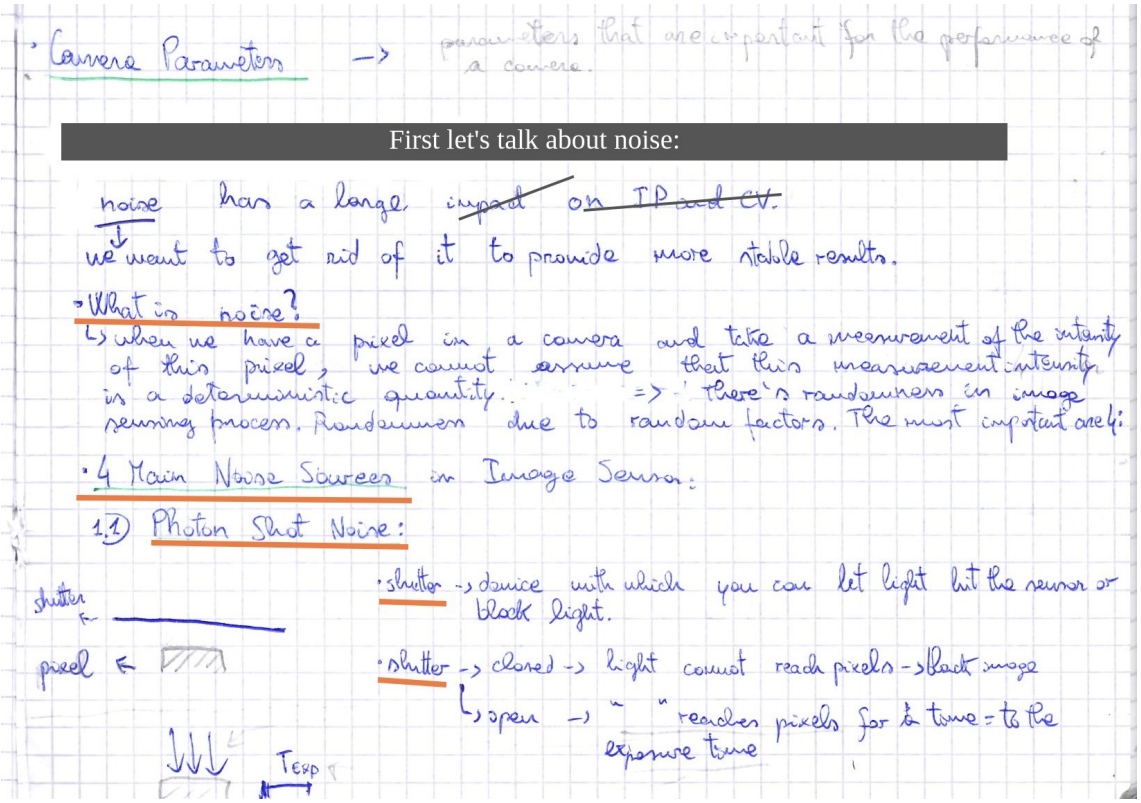


Figure 1.16

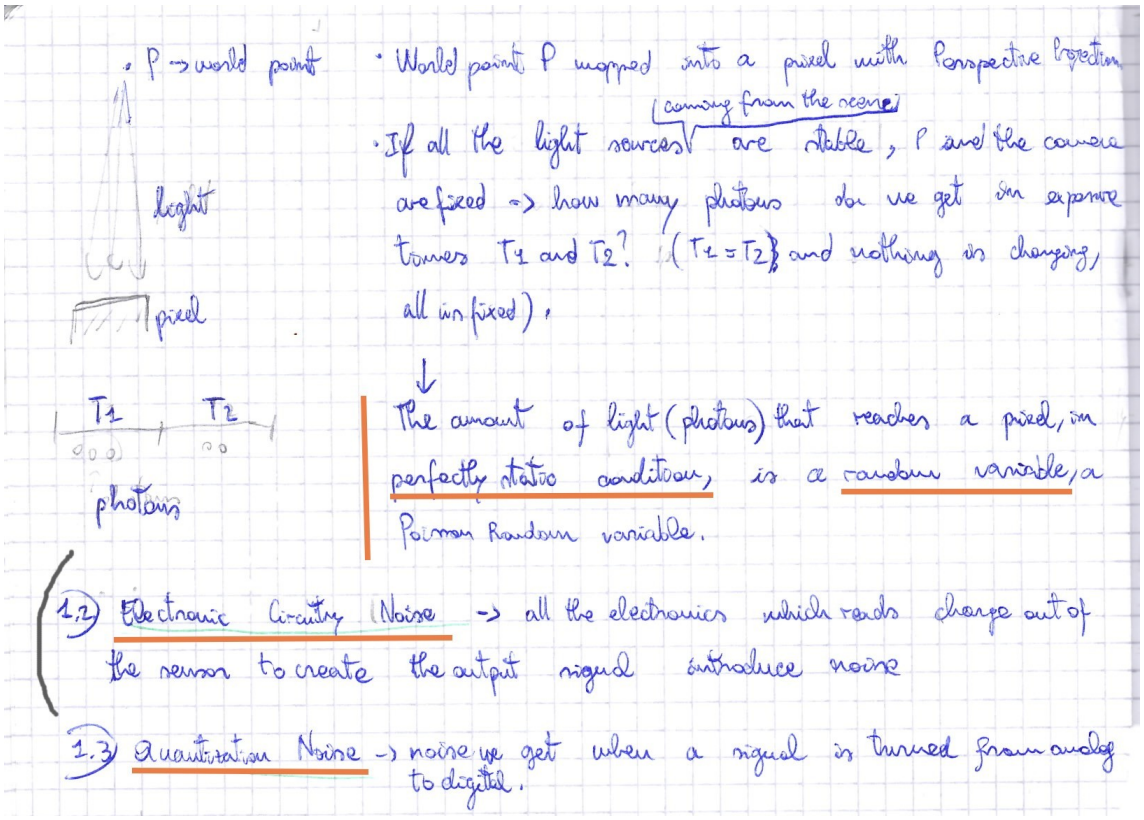


Figure 1.17

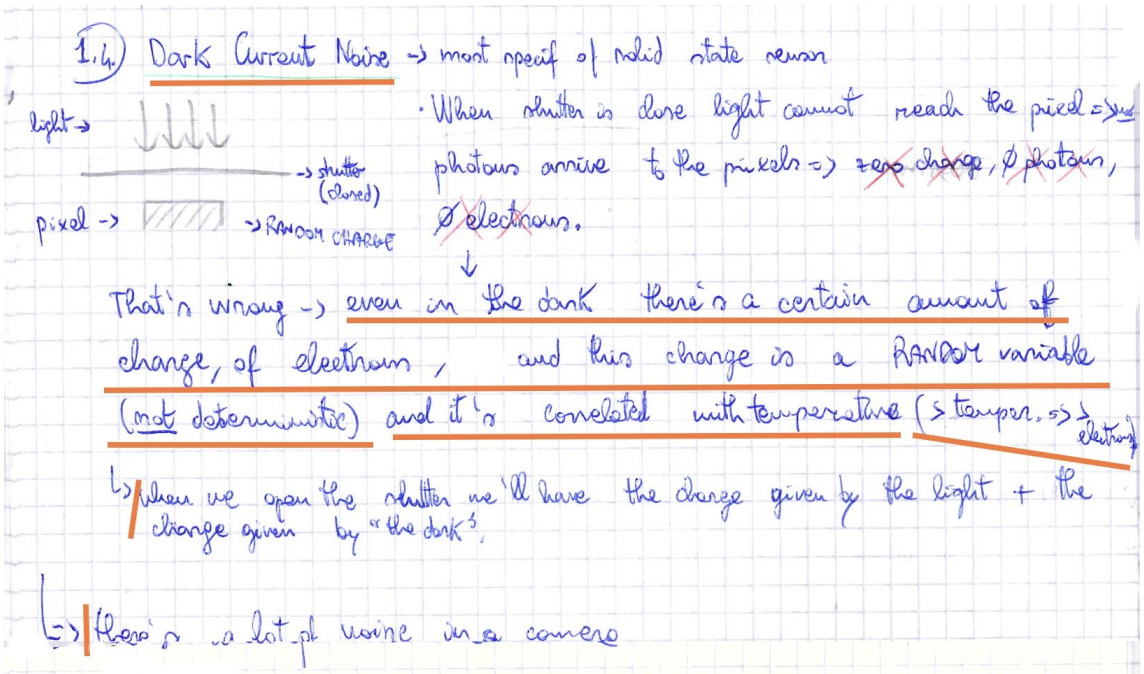


Figure 1.18

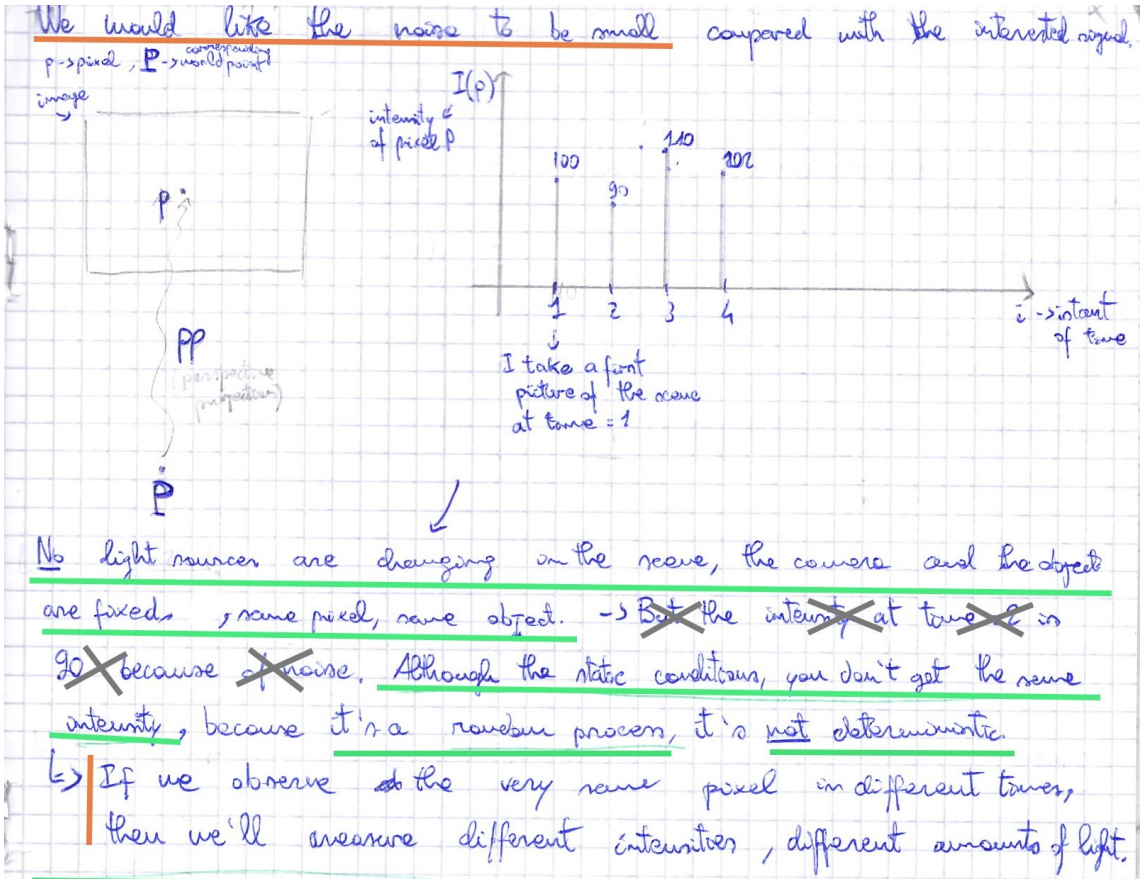

We would like the noise to be small compared with the interested signal.
p -> pixel, P -> corresponding world point
image
p
PP
perspective projection
P
I(p)
intensity of pixel p
100
90
110
102
1
2
3
4
i -> instant of time
I take a first picture of the scene at time = 1
No light sources are changing in the scene, the camera and the objects are fixed, same pixel, same object. -> But the intensity at time 2 is 90 because of noise. Although the static conditions, you don't get the same intensity, because it's a random process, it's not deterministic.
If we observe the very same pixel in different times, then we'll measure different intensities, different amounts of light.

Figure 1.19

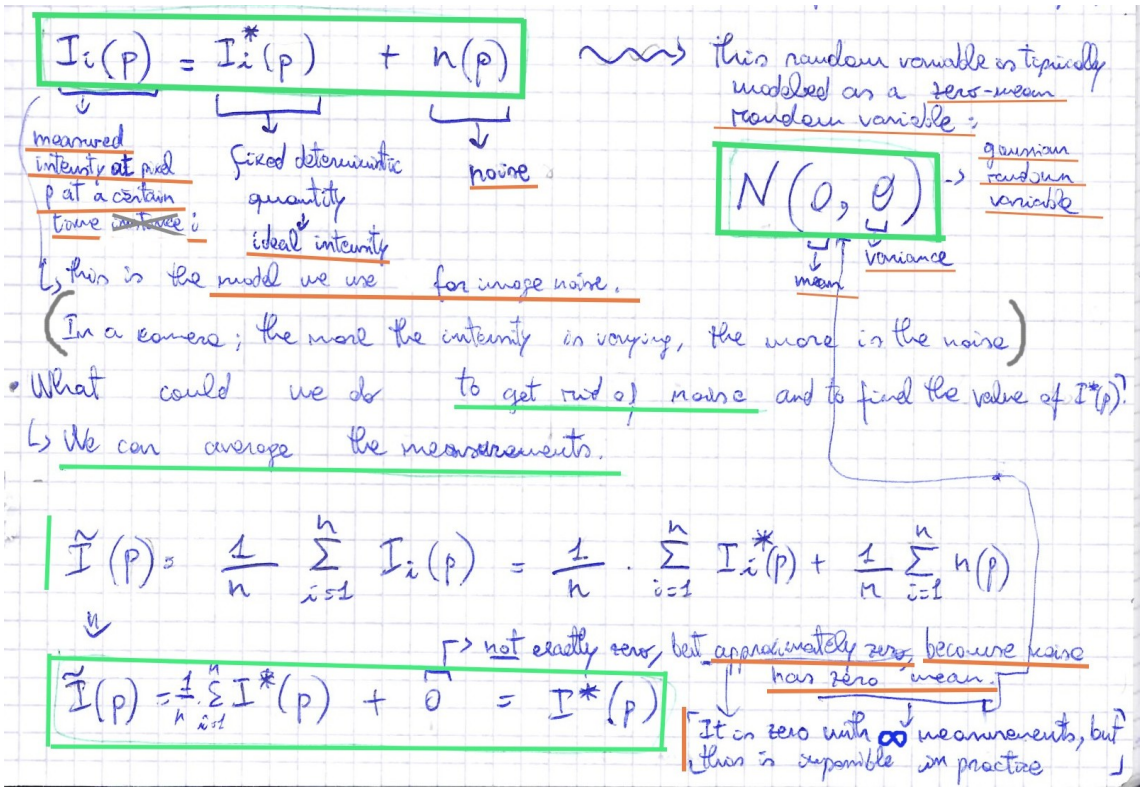


Figure 1.20

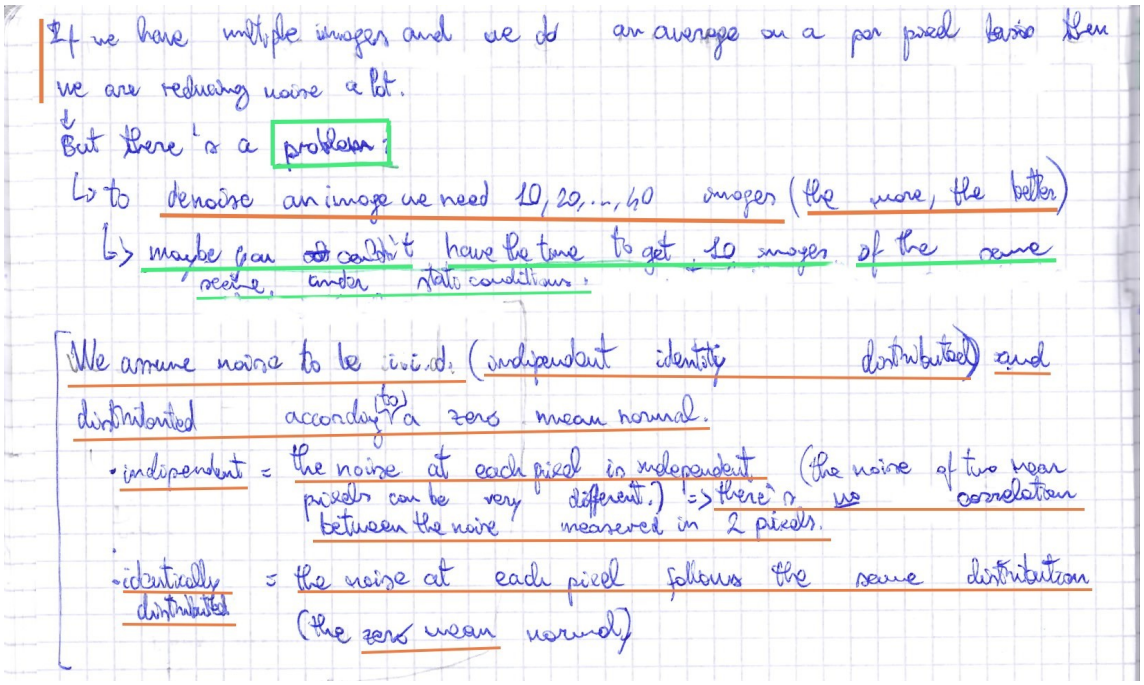


Figure 1.21

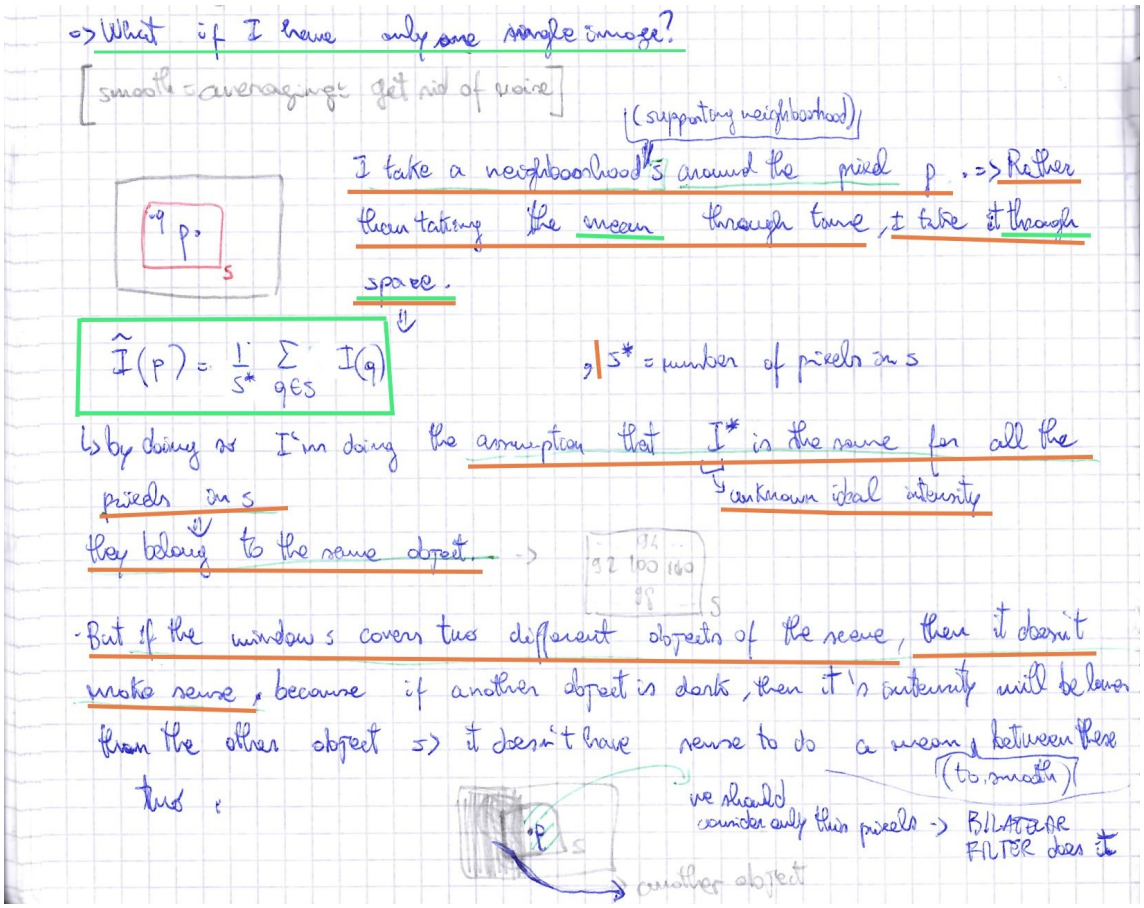


Figure 1.22

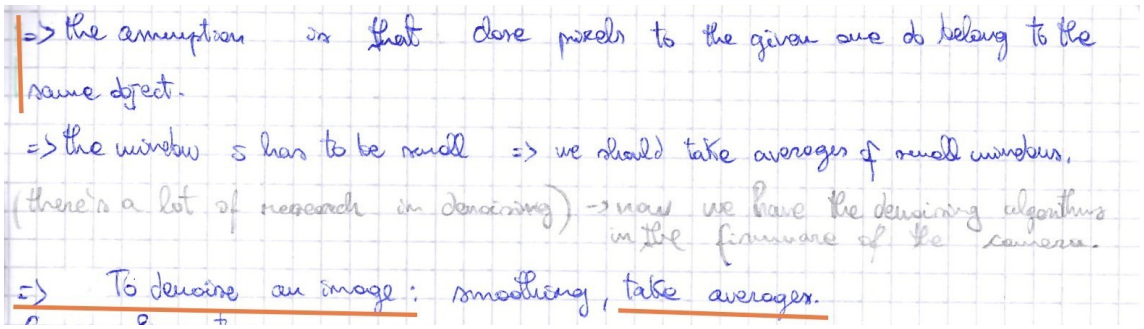


Figure 1.23

1.3.2 SNR (Signal to noise ratio)

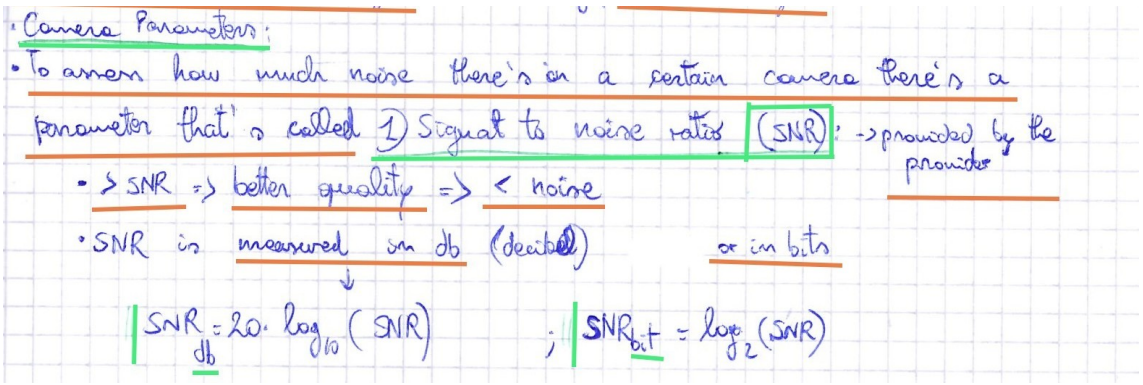


Figure 1.24

1.3.3 Dynamic Range (DR)

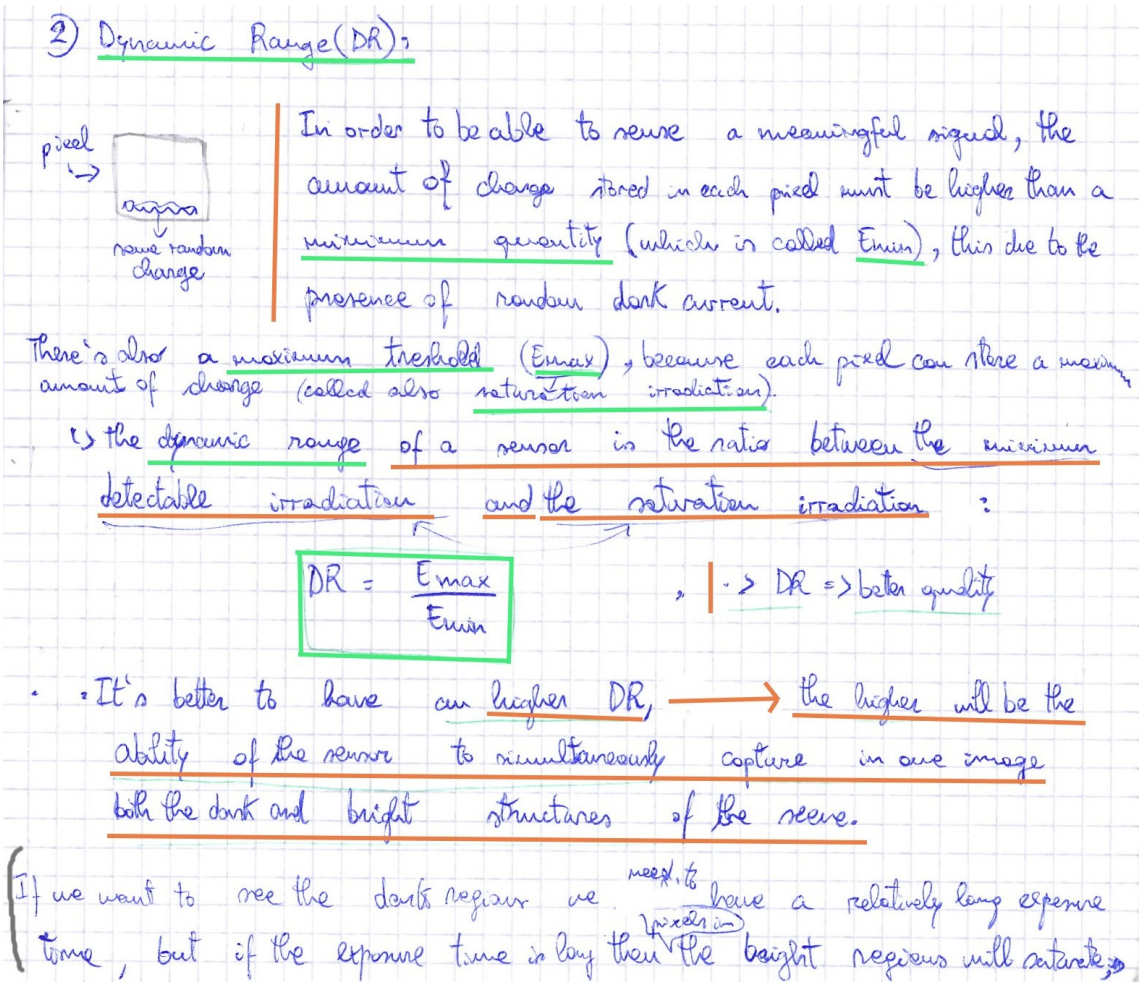


Figure 1.25

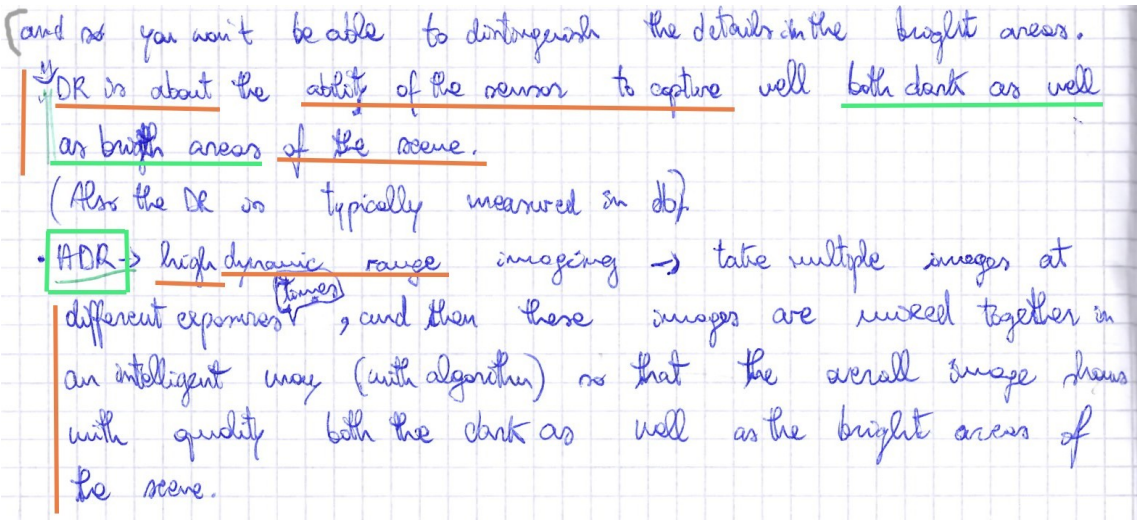


Figure 1.26

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

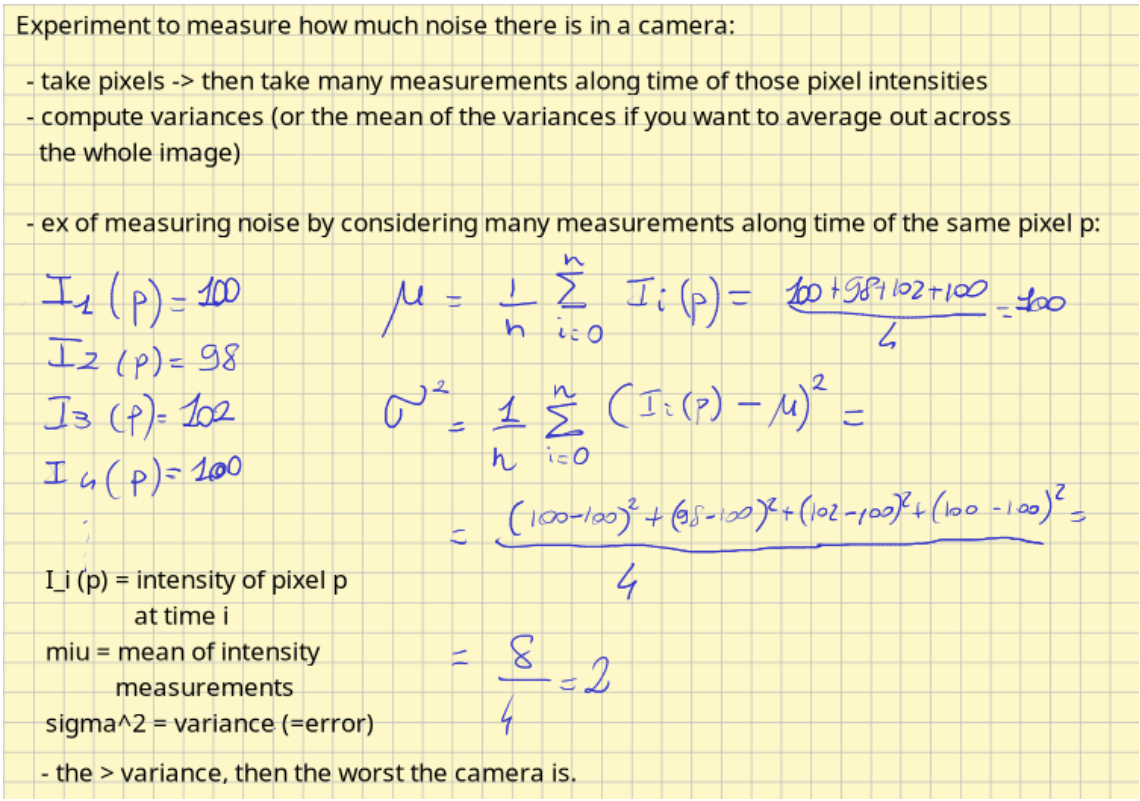


Figure 1.27

Chapter 2

Camera Calibration

2.1 Intrinsic parameters

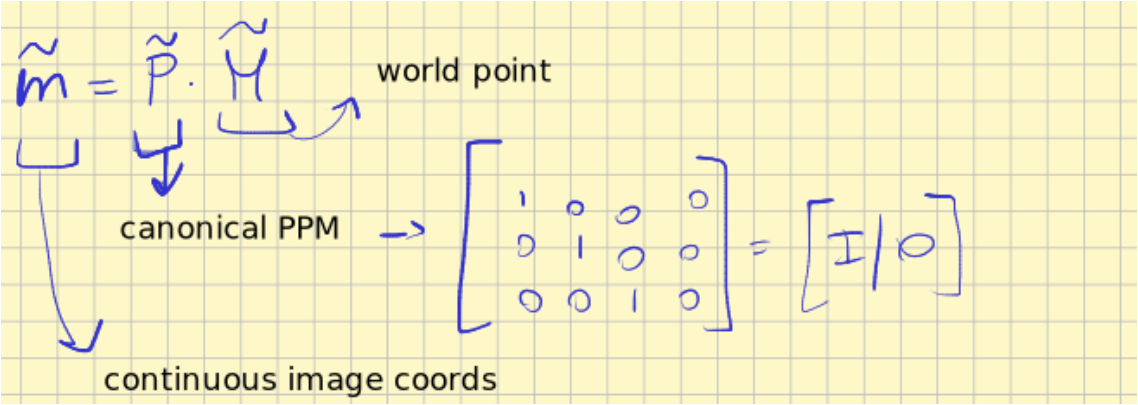


Figure 2.1

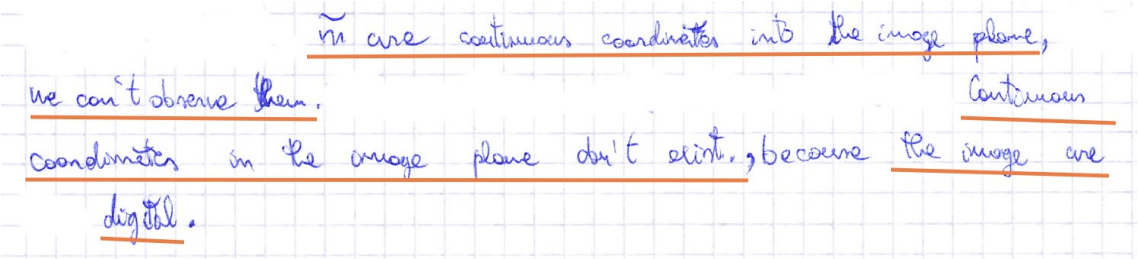


Figure 2.2

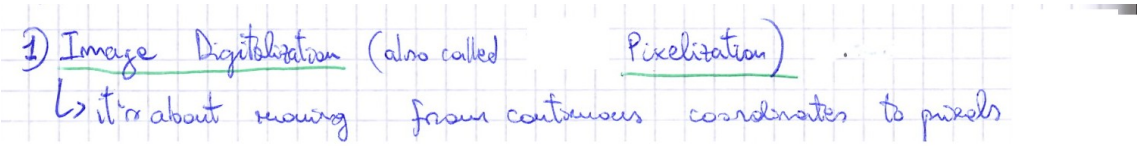


Figure 2.3

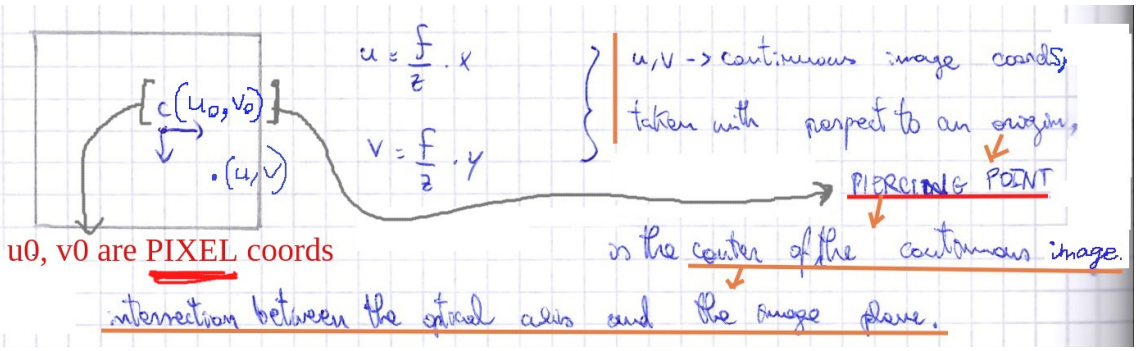


Figure 2.4

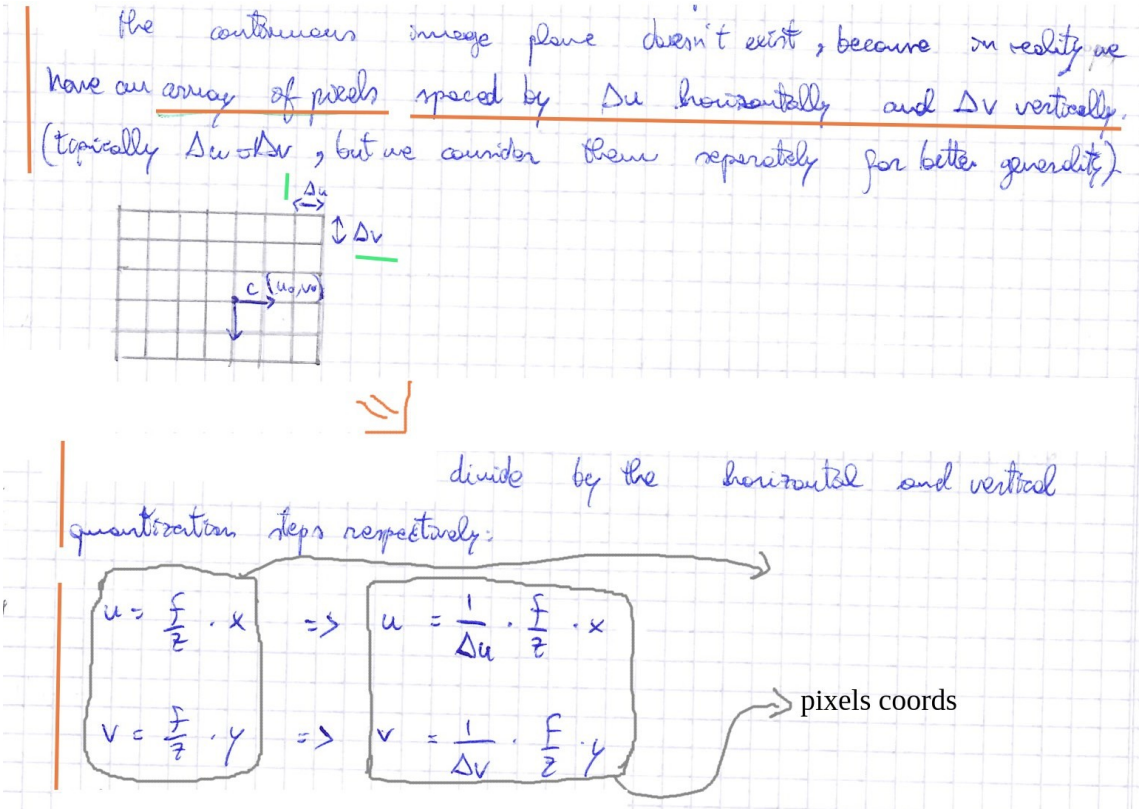


Figure 2.5

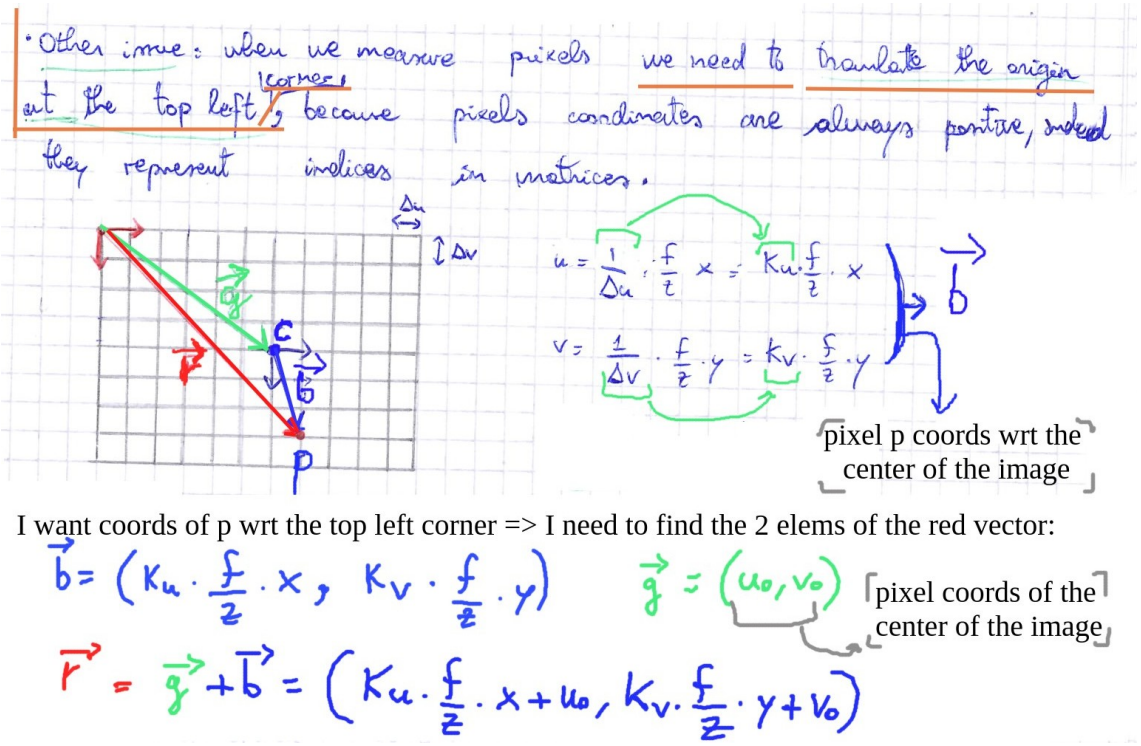


Figure 2.6

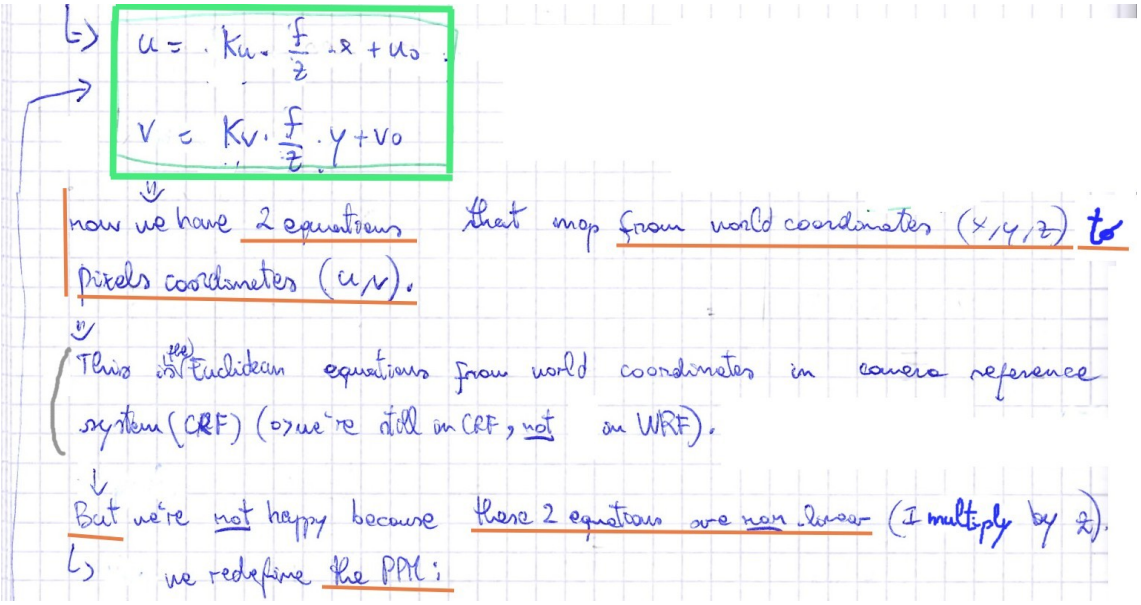


Figure 2.7

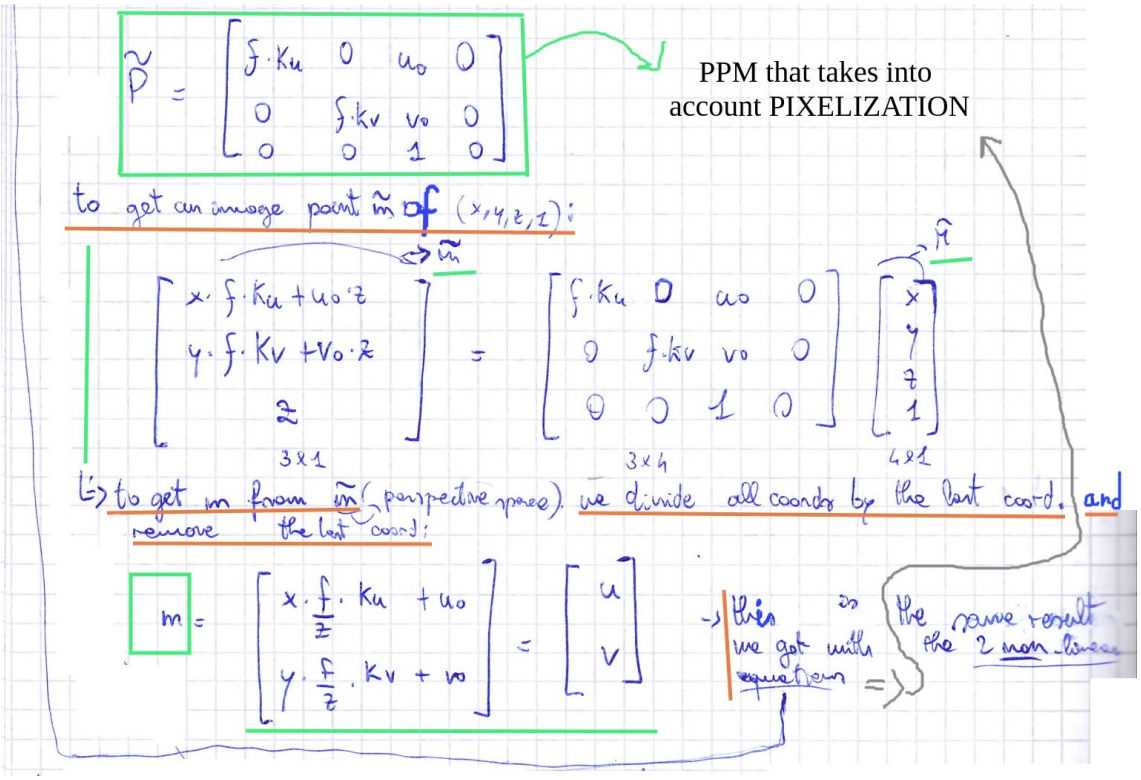


Figure 2.8

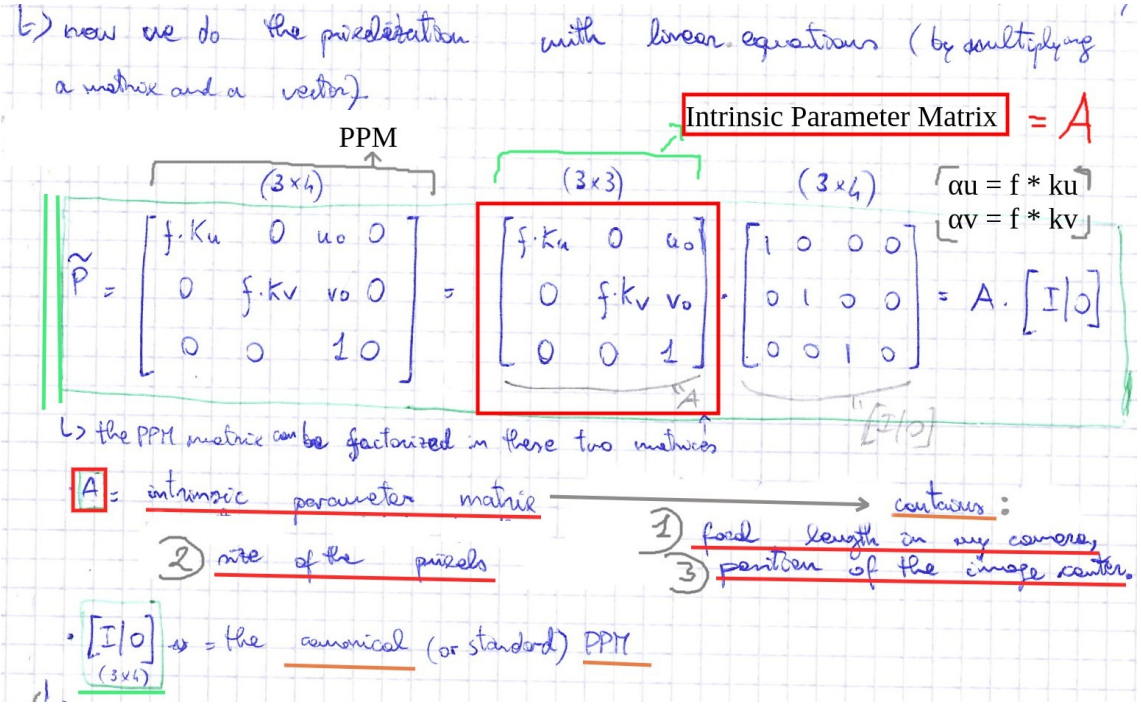


Figure 2.9

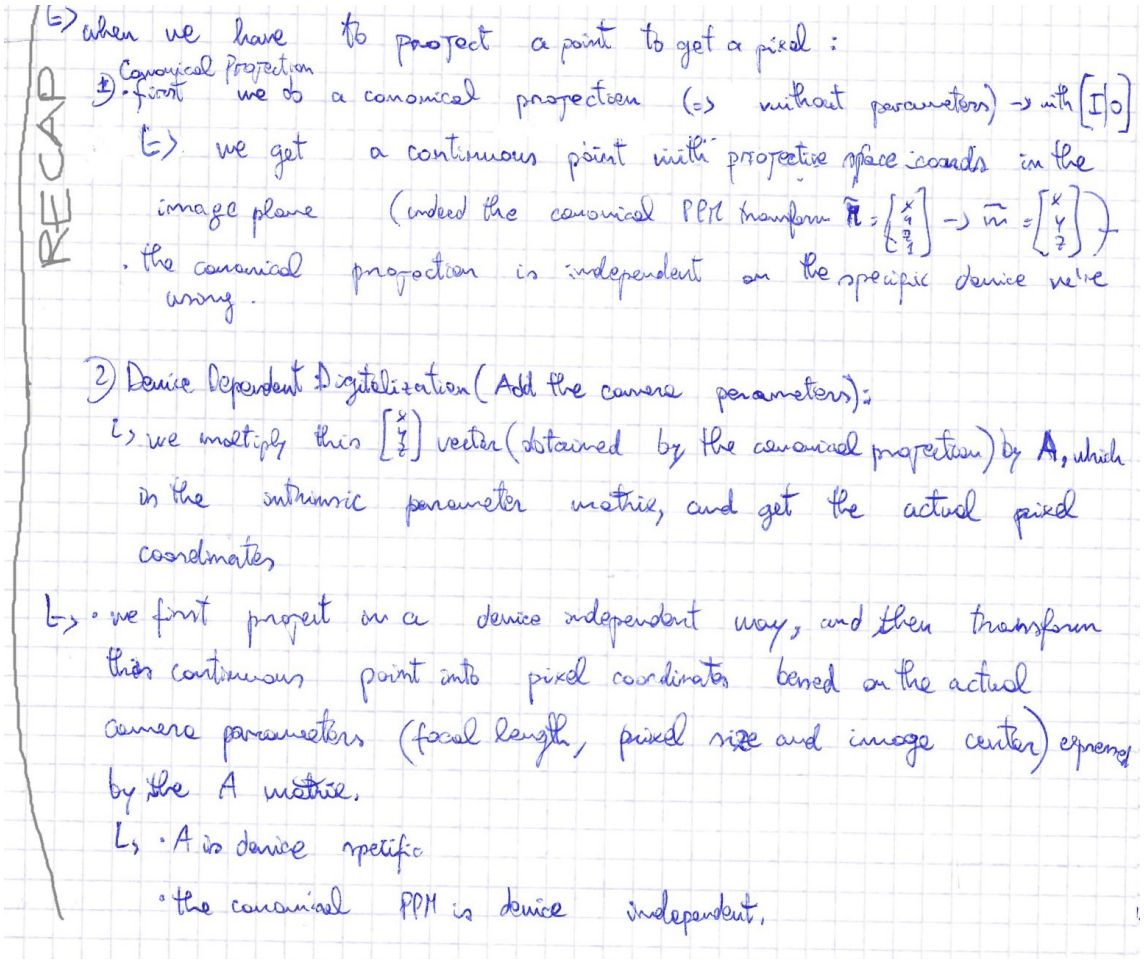


Figure 2.10

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 -> intrinsic parameters
  - extrinsic paremeters -> rigid motion between the WRF and the CRF:
    - R -> totation matrix
    - T -> translation vector
  - lens distortion parameters -> because lenses are not perfeectly thin => we have some distortion.
- How to estimate these parameters:
  - we have the model of image formation process:

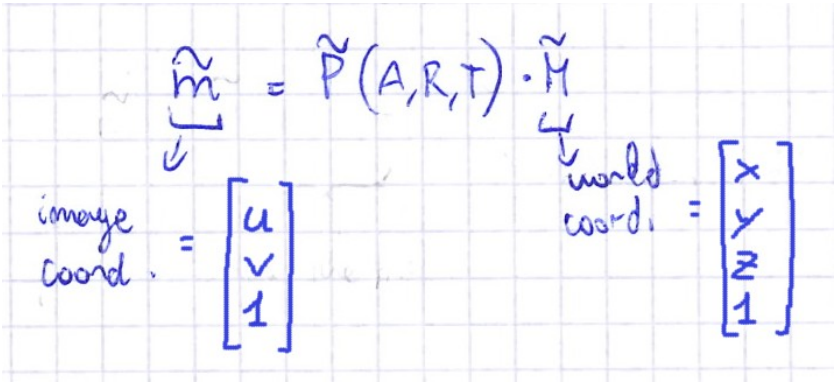


Figure 2.11

we have to solve 

=> to estimate camera parameters A, R, T we need to know some world (M_tilde) and pixel coordinates (m_tilde).

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

Figure 2.12

- How to find pairs of known world and pixel coords (M_tilde, m_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:

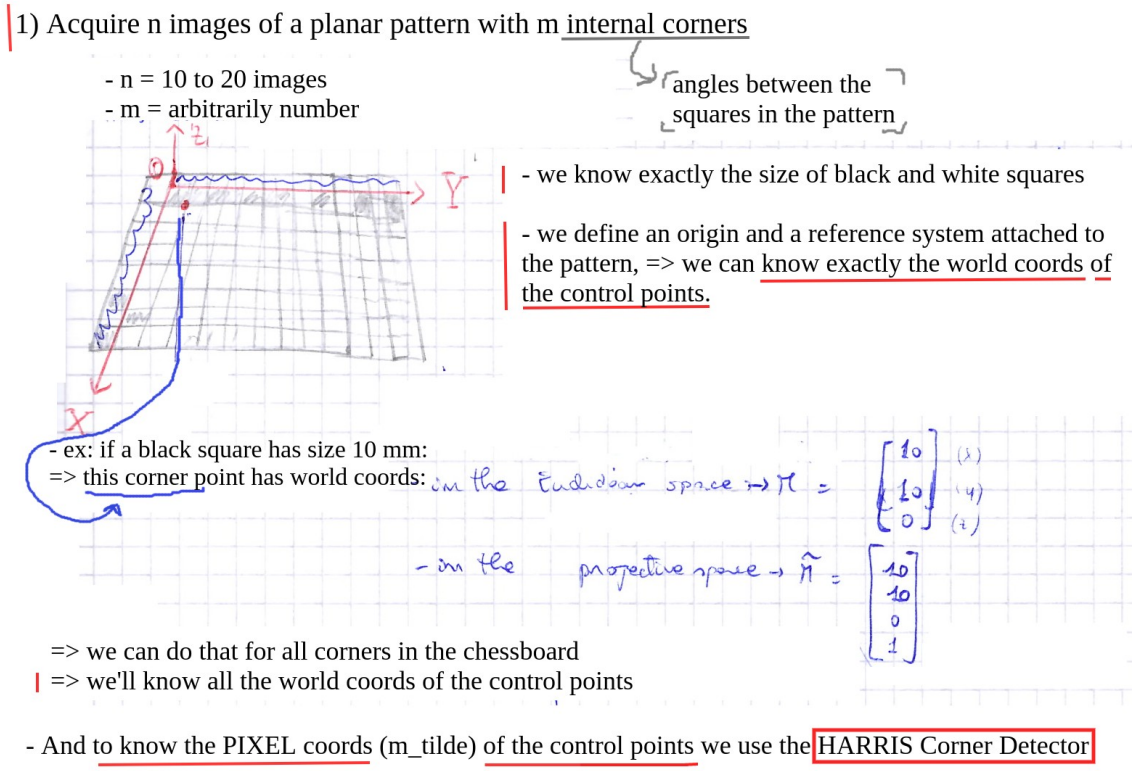


Figure 2.13

- 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.

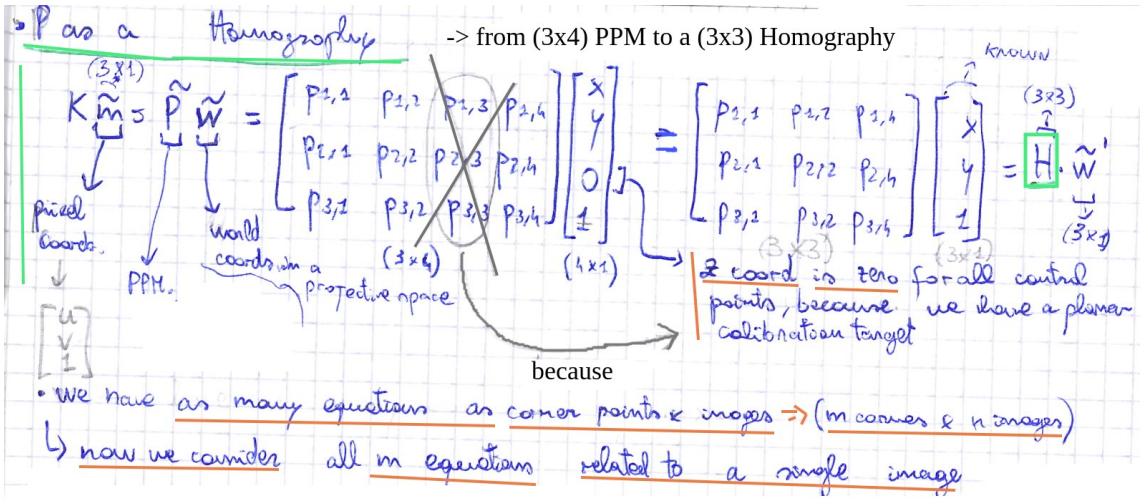


Figure 2.14

N.B. 1) If we have a planar scene (with z = 0), then we project that into an image by an Homography => 3x3 and NOT 3x4

2) Any two images of a planar scene are related by an Homography

Figure 2.15

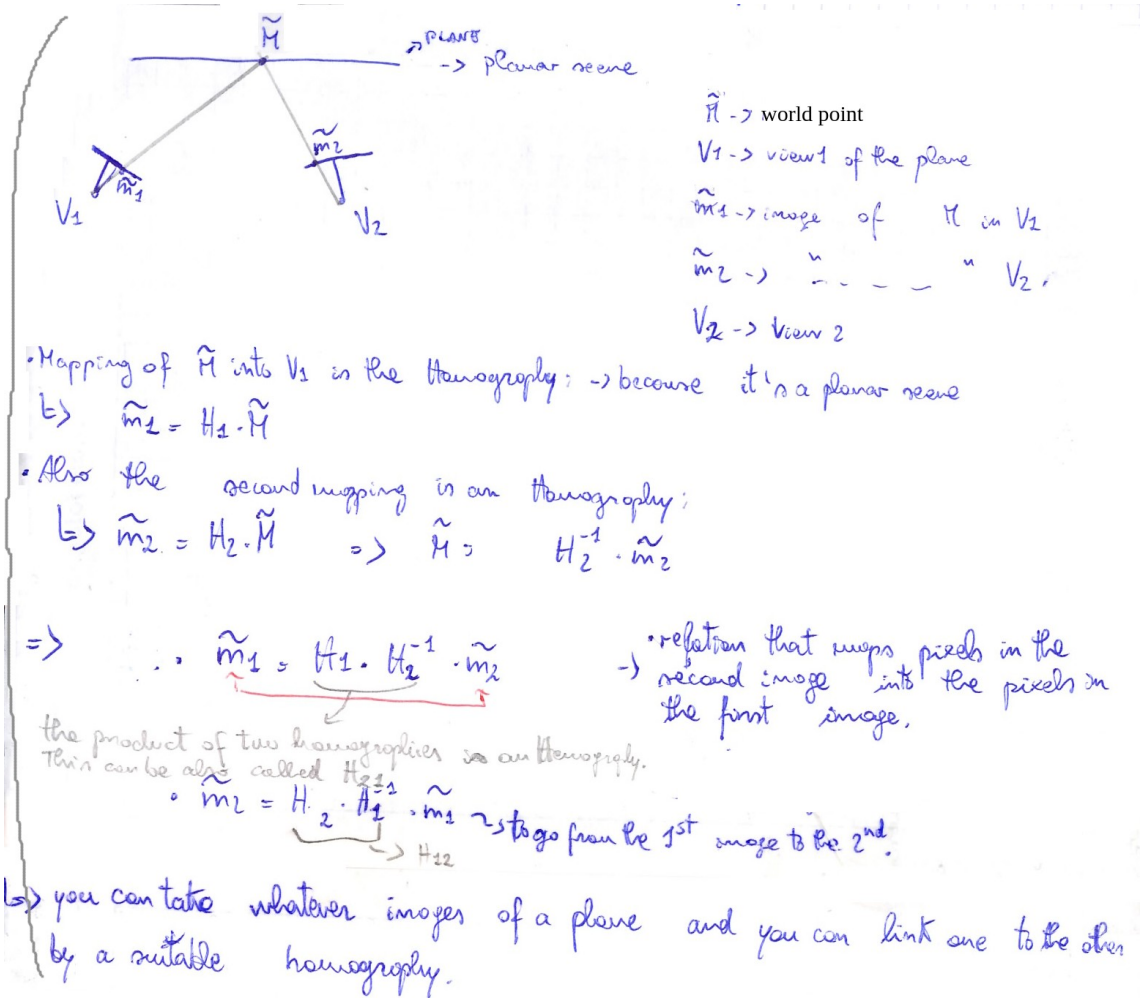


Figure 2.16

- 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

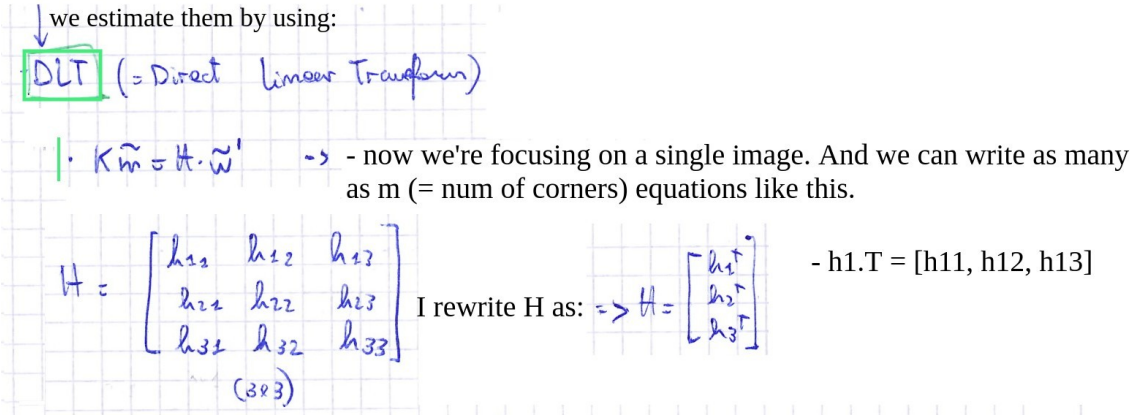


Figure 2.17

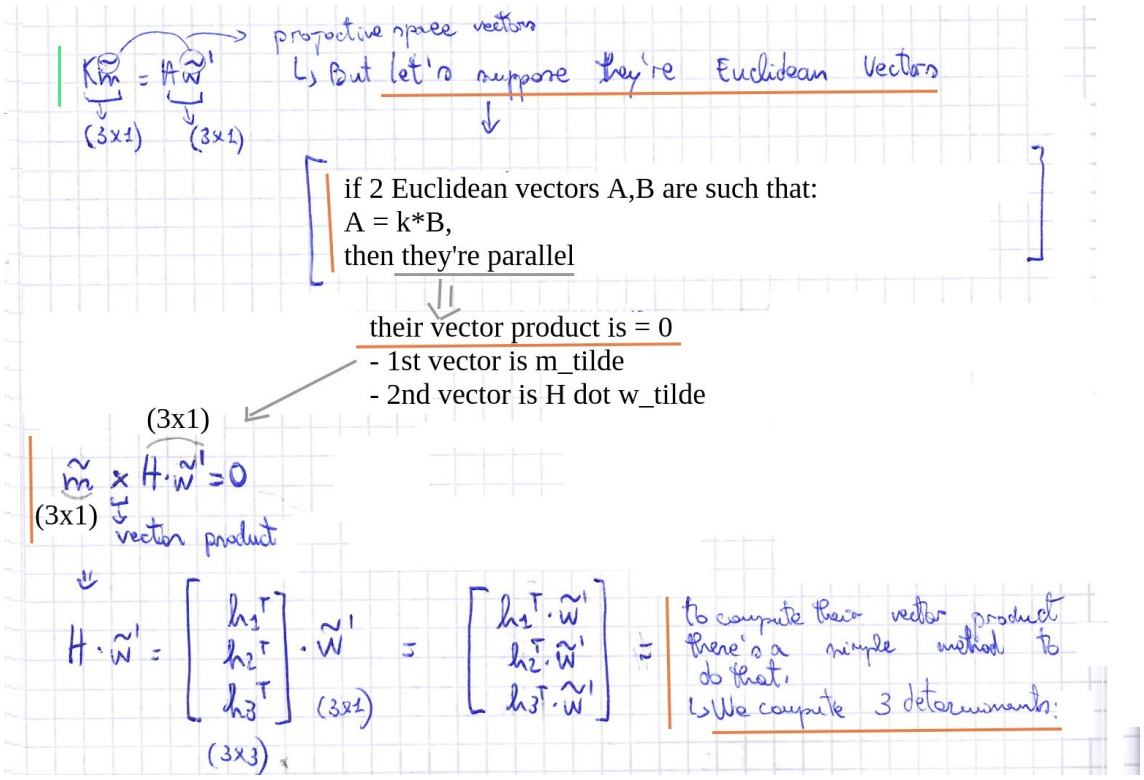

projective space vectors
But let's suppose they're Euclidean vectors
if 2 Euclidean vectors A,B are such that:
A = k*B,
then they're parallel
their vector product is = 0
- 1st vector is m_tilde
- 2nd vector is H dot w_tilde
(3x1)
(3x1)
vector product
(3x1)
(3x3)
To compute their vector product there's a simple method to do that.
We compute 3 determinants:

Figure 2.18

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

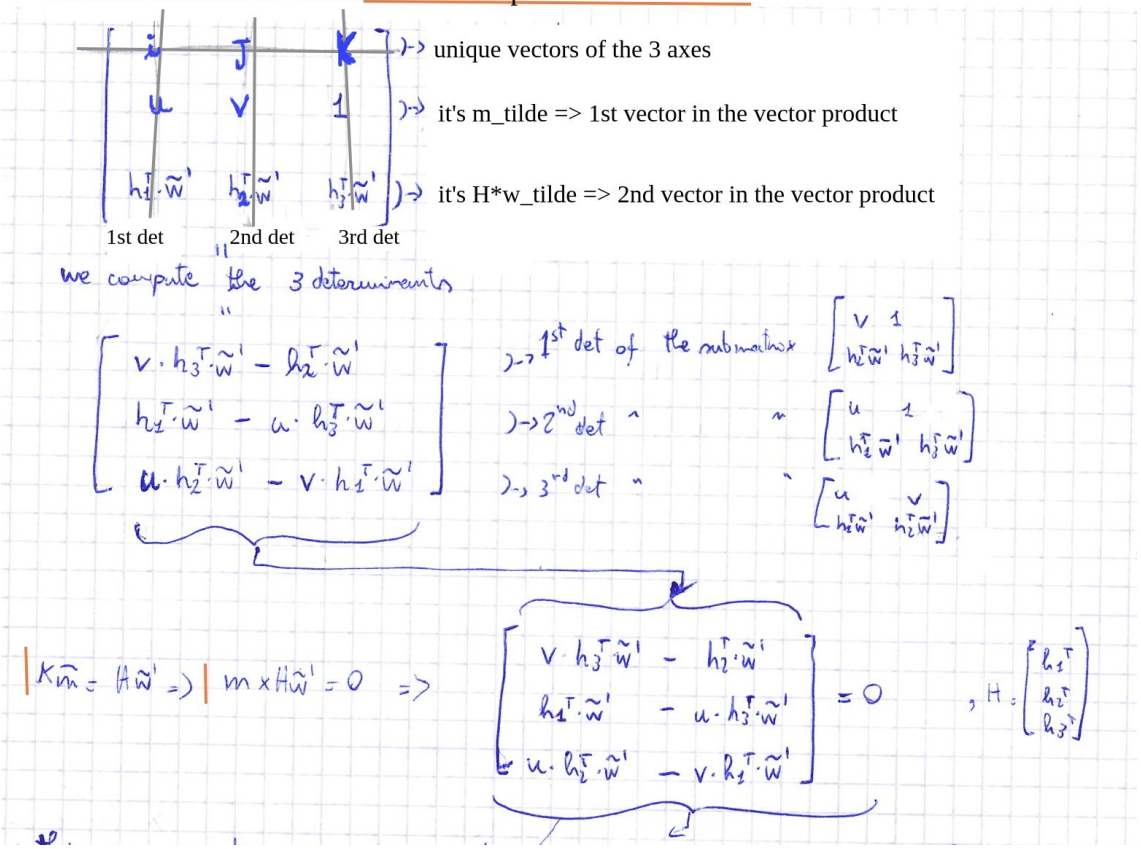


Figure 2.19

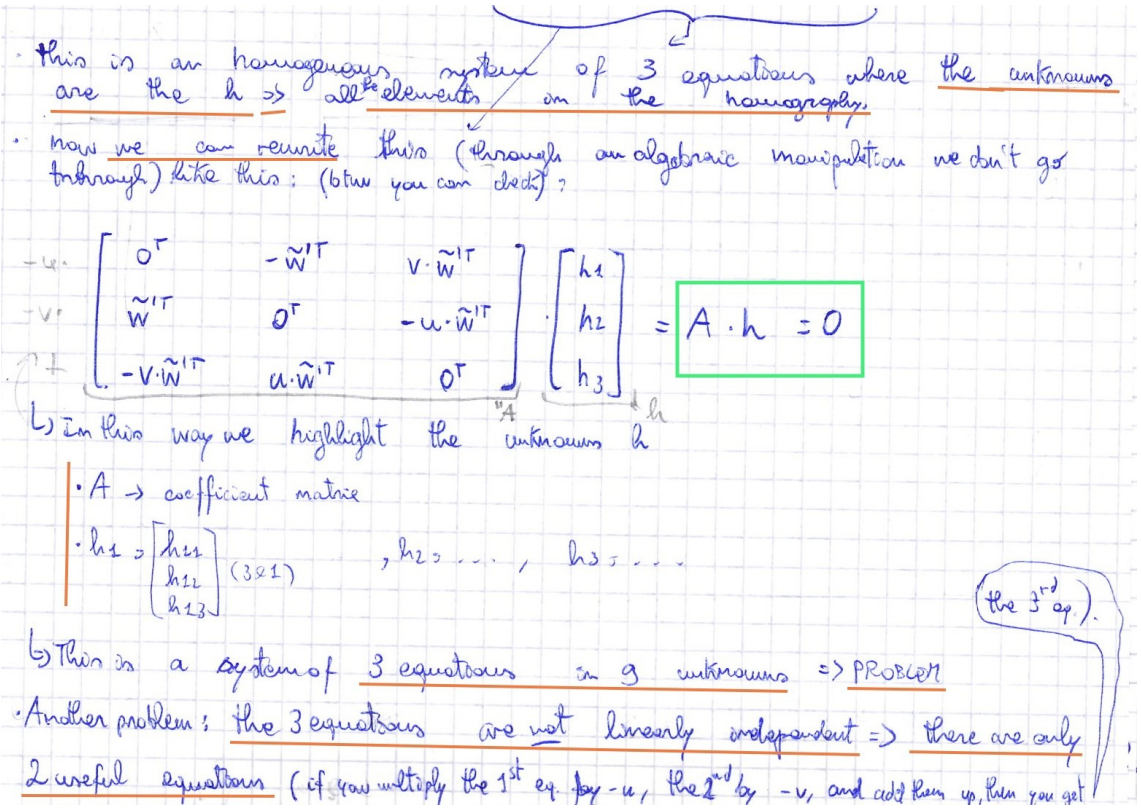

This is an homogeneous system of 3 equations where the unknowns are the h => all the elements on the homography.
now we can rewrite this (through an algebraic manipulation we don't go trhrough) like this: (btw you can check):
$\begin{bmatrix} 0^T & -\tilde{w}'^T & v\cdot\tilde{w}'^T \\ \tilde{w}'^T & 0^T & -u\cdot\tilde{w}'^T \\ -v\cdot\tilde{w}'^T & u\cdot\tilde{w}'^T & 0^T \end{bmatrix} \cdot \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = A\cdot h = 0$
"A"
h
b) In this way we highlight the unknowns h
• A -> coefficient matrix
• $h_1 = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \end{bmatrix}$ (3x1) , $h_2 = \dots$, $h_3 = \dots$
(the 3rd eq).
b) This is a system of 3 equations in 9 unknowns => PROBLEM
•Another problem: the 3 equations are not linearly independent => 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 = 3x3 elements of the homography).
=> Solution: there are m corners in each calibration image
=> we have 2 x 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 x 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):

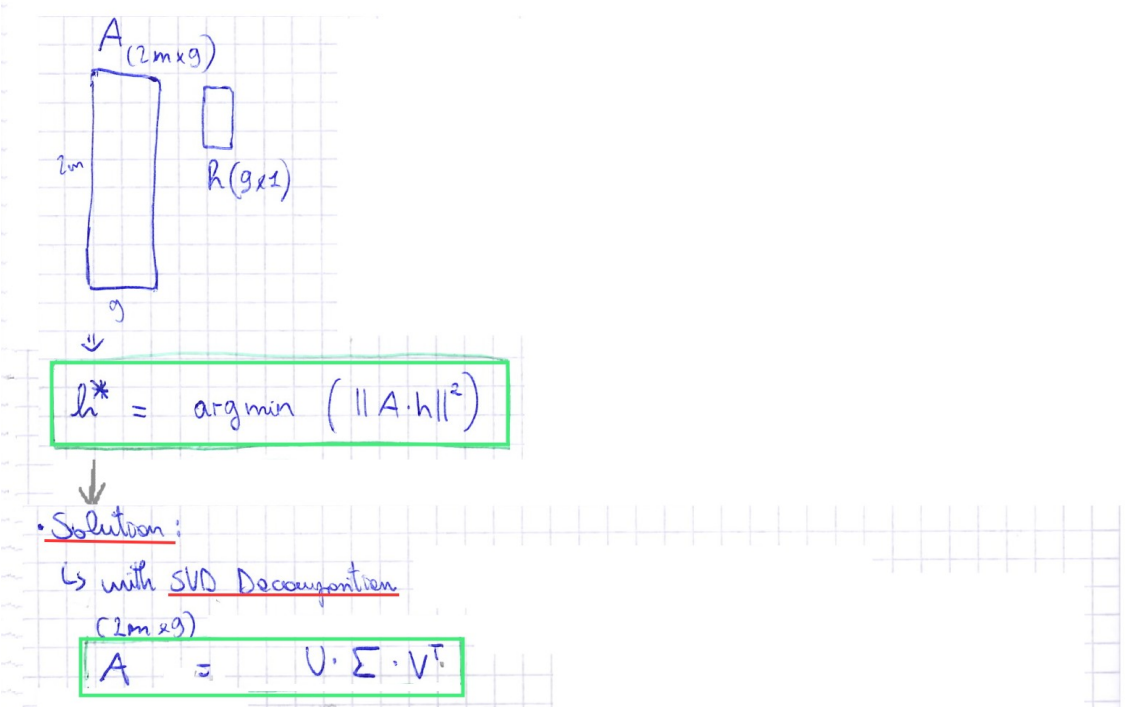


Figure 2.21

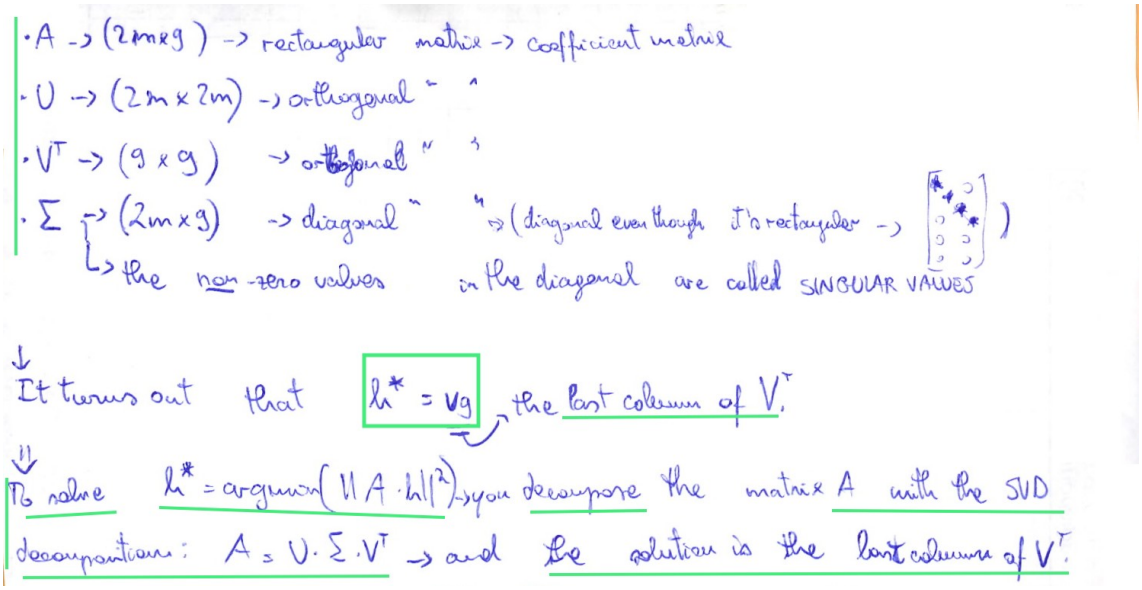


Figure 2.22

The homography could have been solved in 8 unknowns => we just need m = 4 corners => 4 x 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?

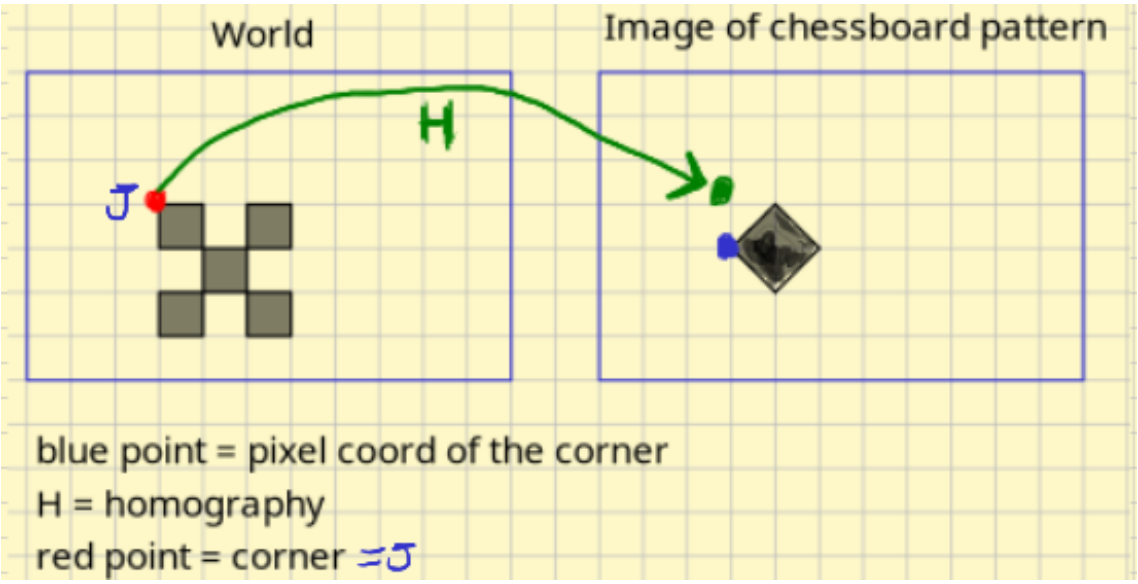
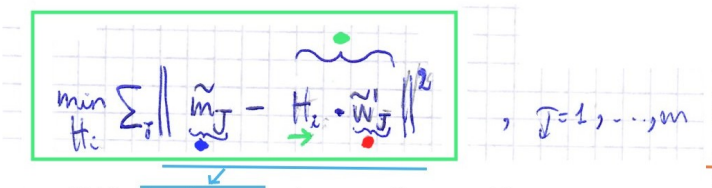


Figure 2.24

  - the link between these 2 planes is the homography
    - because the scene (world) is planar (z = 0).
  - If I multiply J (red pointpoint) 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).



- I want Hi * w_tilde_j (the green point estimated through the homography) to be as close as possible to m_tilde (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, ..., m -> I want the smallest distance for each corner point.

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

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

Figure 2.26

- Now we have n (n = num of calibration images) Homograpies:
  - all of the homographies have in common the Intrinsics.
  - => 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

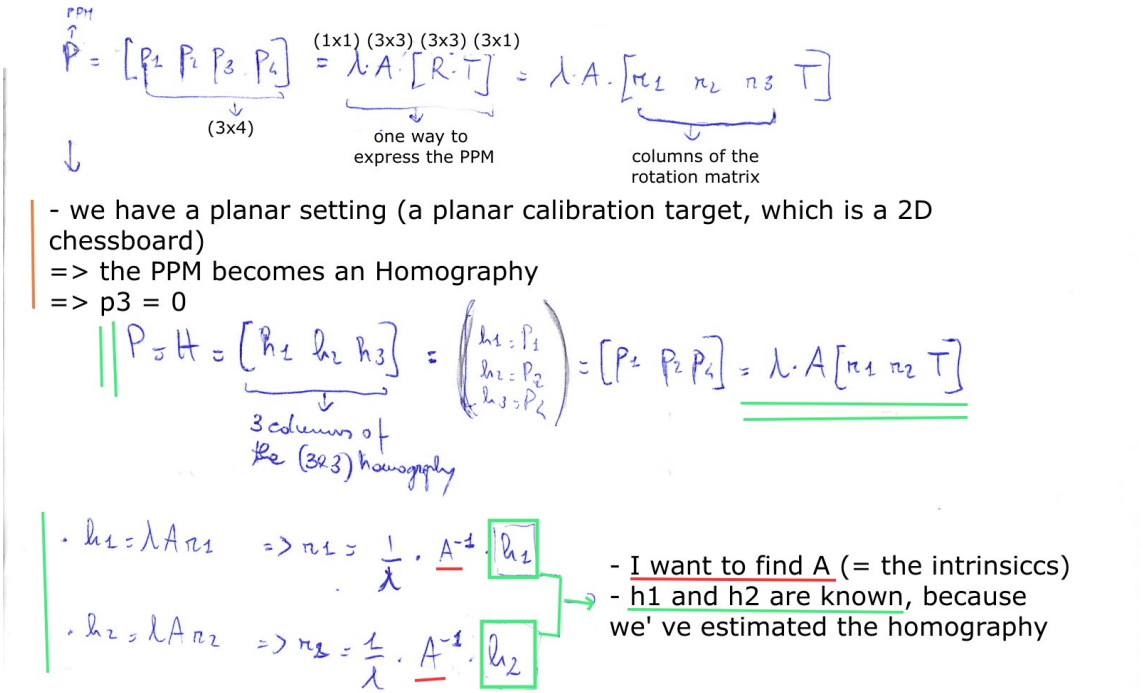


Figure 2.27

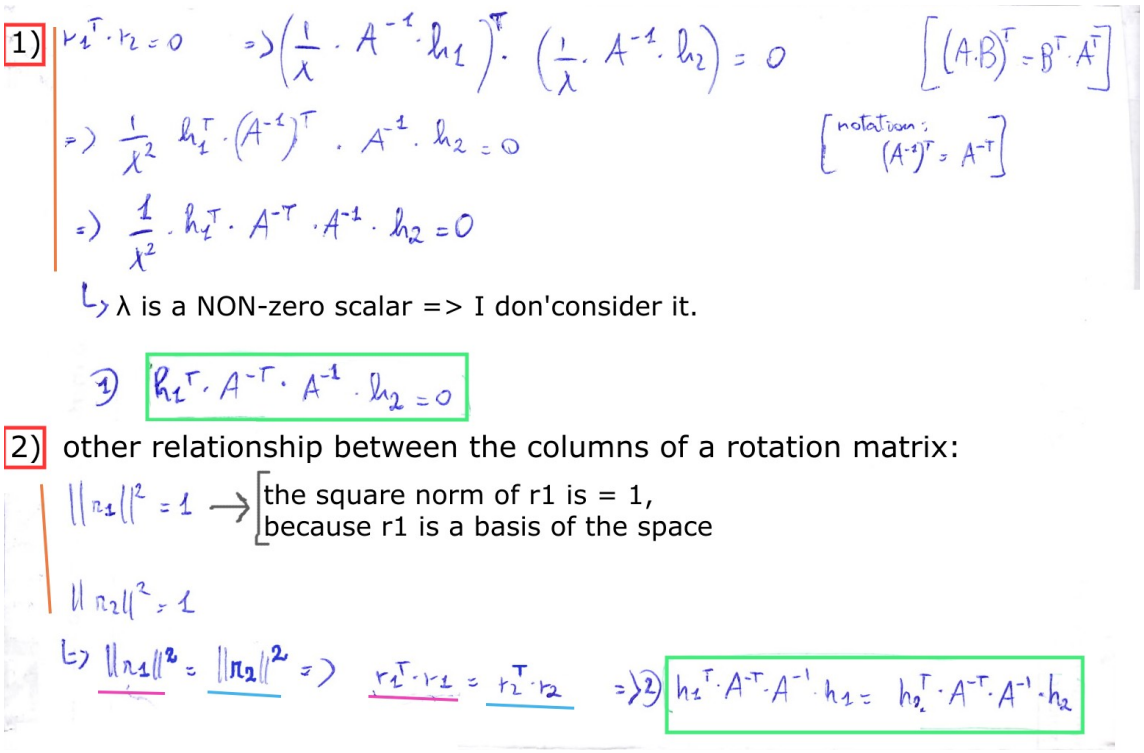


Figure 2.28

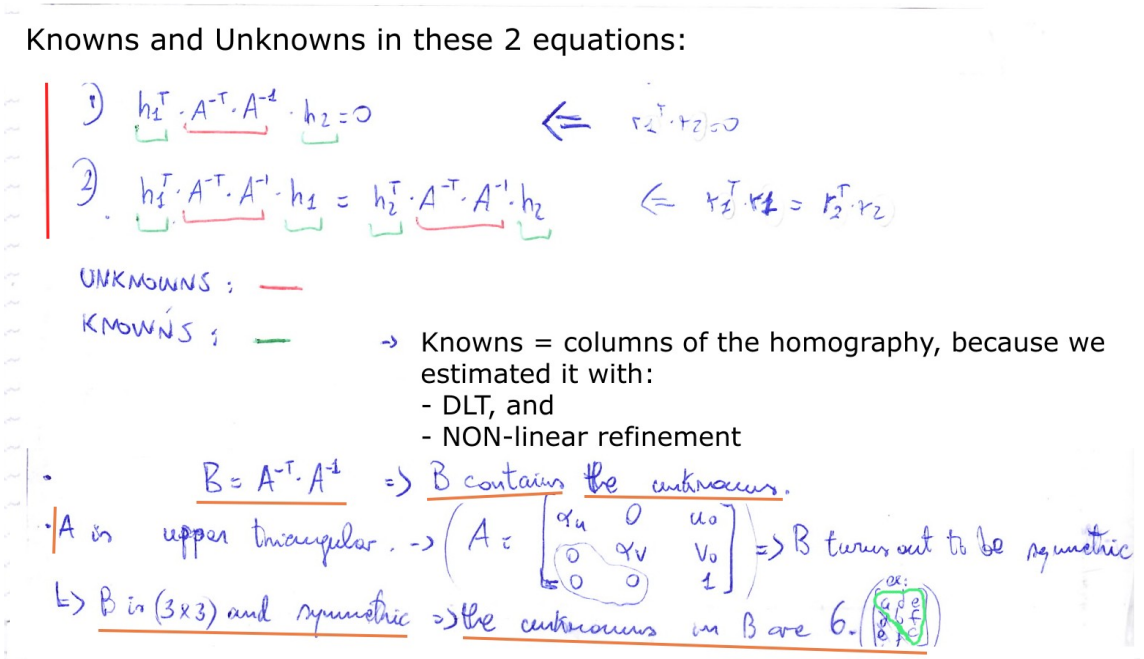


Figure 2.29

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

=> a 2n x 6 linear system, n = num of calibration images

- To solve this problem we could use only 3 calibration images (2*3 x 6), BUT we prefer an overconstrained problem (20-30 calibration images) (=> num equations >> num unknowns)

Figure 2.30

At the end we get the system:

$V \cdot b = 0$ -> $2n, 6$ (6 unknowns; $2n$ eq.)

- Unknowns = vector b = 6 unknowns
- this system is overconstrained (num equations >> num unknowns):

=> we solve it with the SVD decompostion:
- we decompose the coefficient matrix V and the solution (6 x 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 Extrinsics as the number of calibration imges.

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: h1, h2, h3

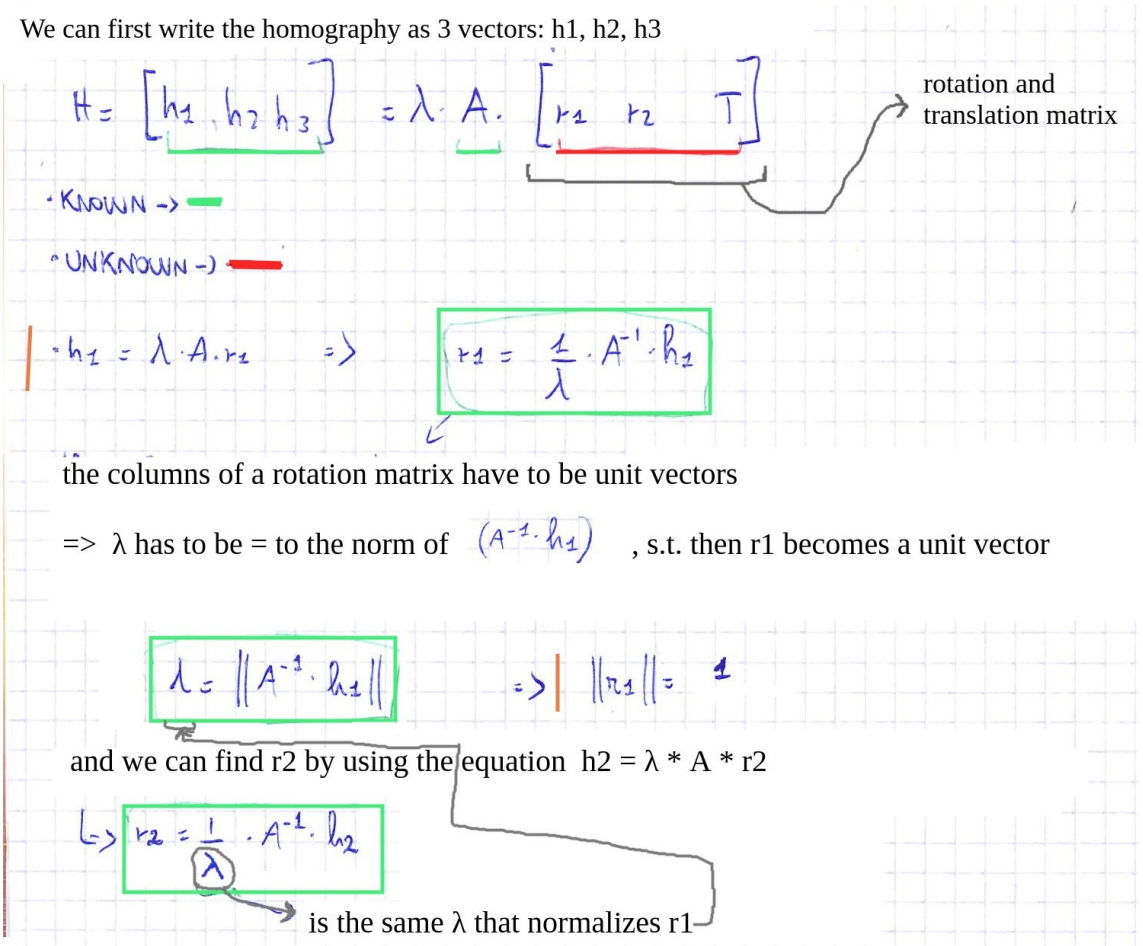
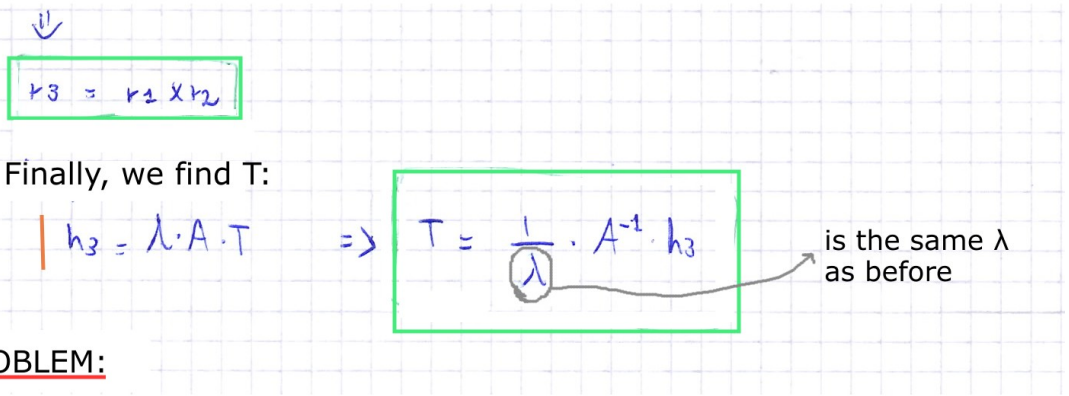


Figure 2.32

to find r3 we know that r1, r2 and r3 have to be a basis of the space => r3 is the vector product of the other two.

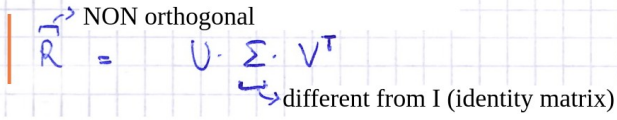


PROBLEM:

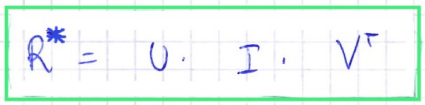
- λ normalizes r1 to a unit norm
=> λ is NOT going to rormalize r2
=> the R matrix is NOT orthogonal => is NOT a rotation matrix

Figure 2.33

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



To make R orthogonal:
=> I substitute Σ (sigma) with the identity matrix I



-> R* is now orthogonal

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

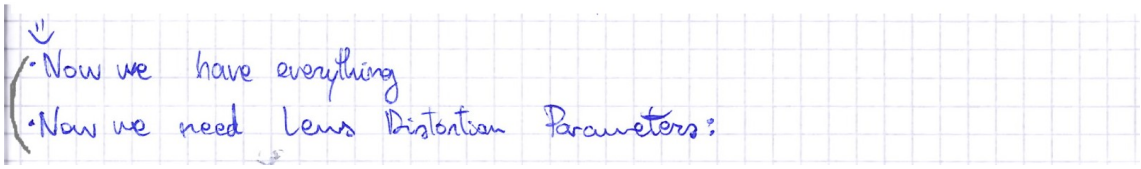


Figure 2.34

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

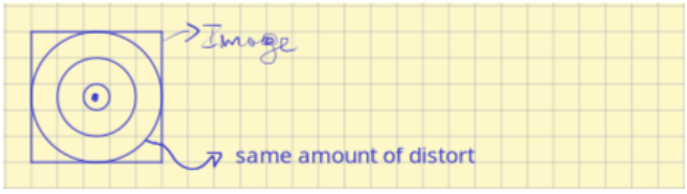


Figure 2.35

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:

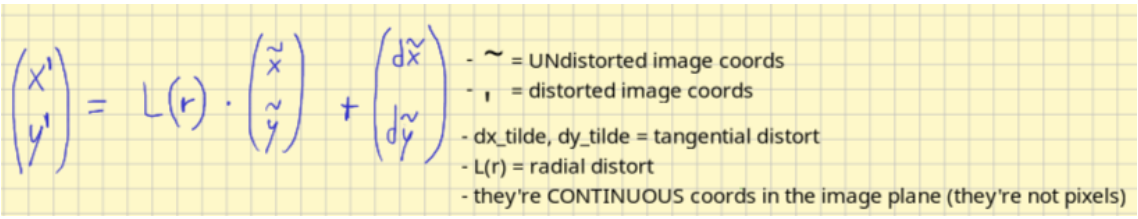


Figure 2.36

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

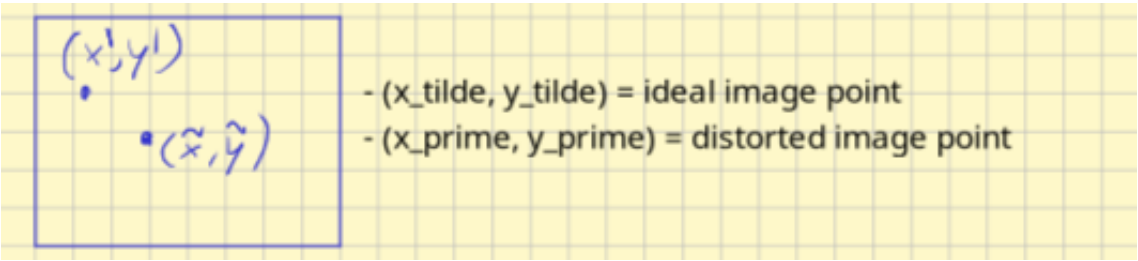


Figure 2.37

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

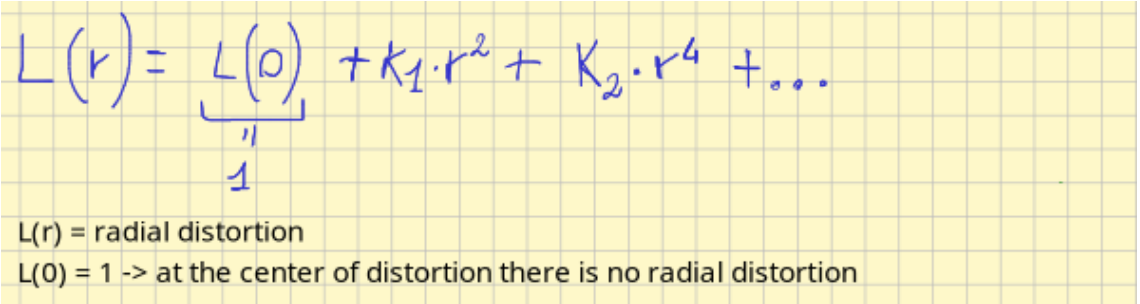


Figure 2.38

- Consider only **even terms** because the radial distortion (L(r)) is an even function (it's defined only for r >= 0). (L(-r) = L(r)).
- So, to model radial distort (L(r)) I have to model the 2 params:
  - k1
  - k2
- While, to model tangential distort I have other 2 params:
  - p1
  - p2

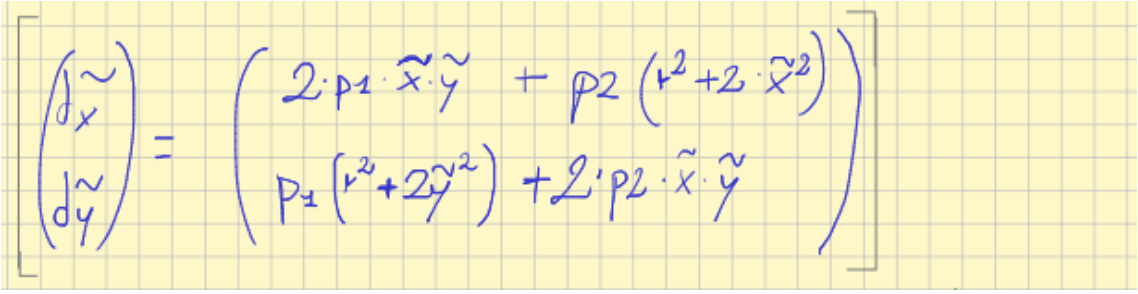


Figure 2.39

2.5.1 Estimation of radial lens distortion coefficients (k1, k2) with Zhang's

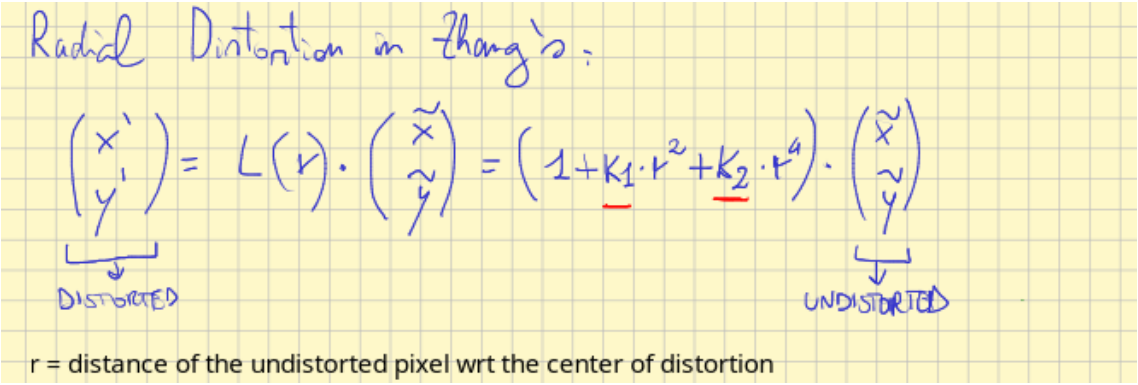


Figure 2.40

- To estimate k1, k2 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):

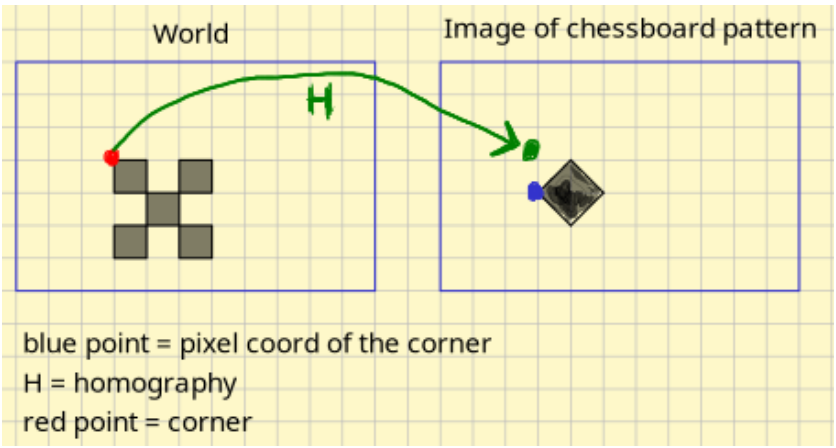


Figure 2.41

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

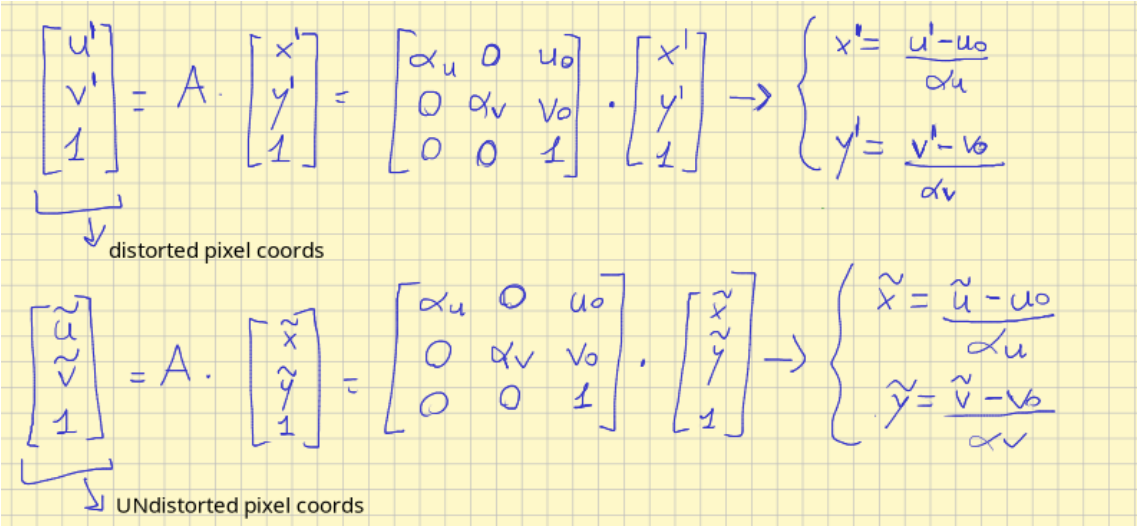


Figure 2.42

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

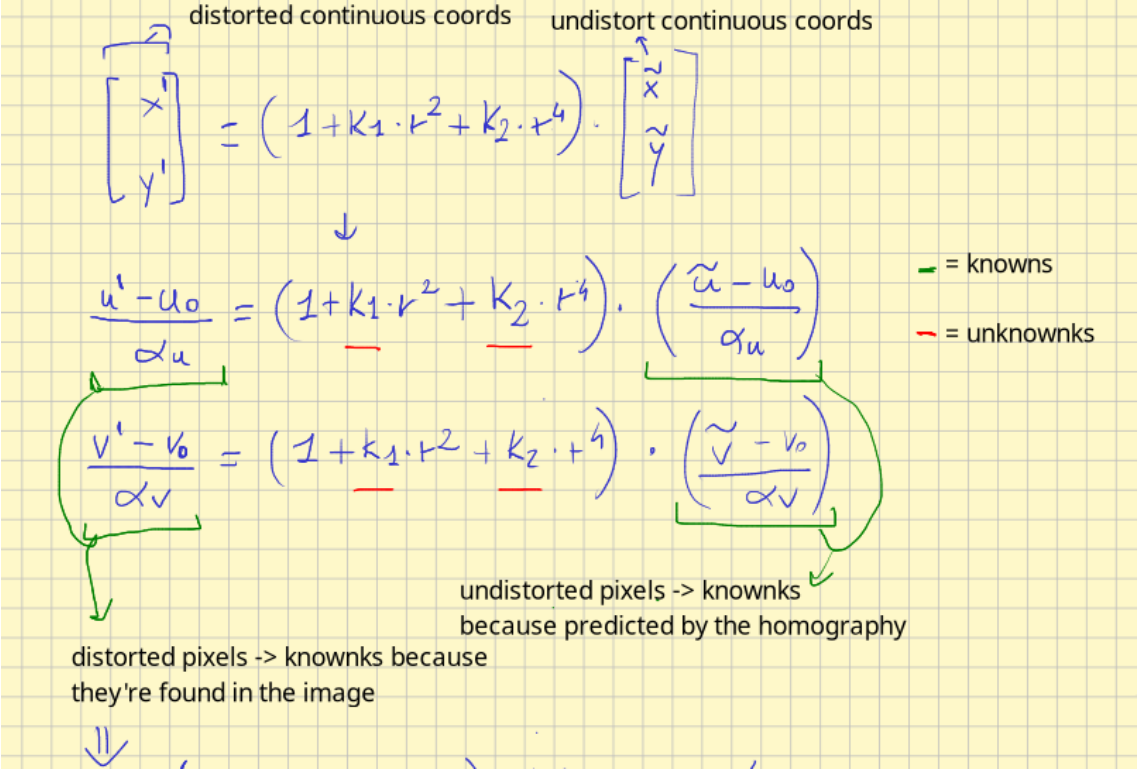


Figure 2.43

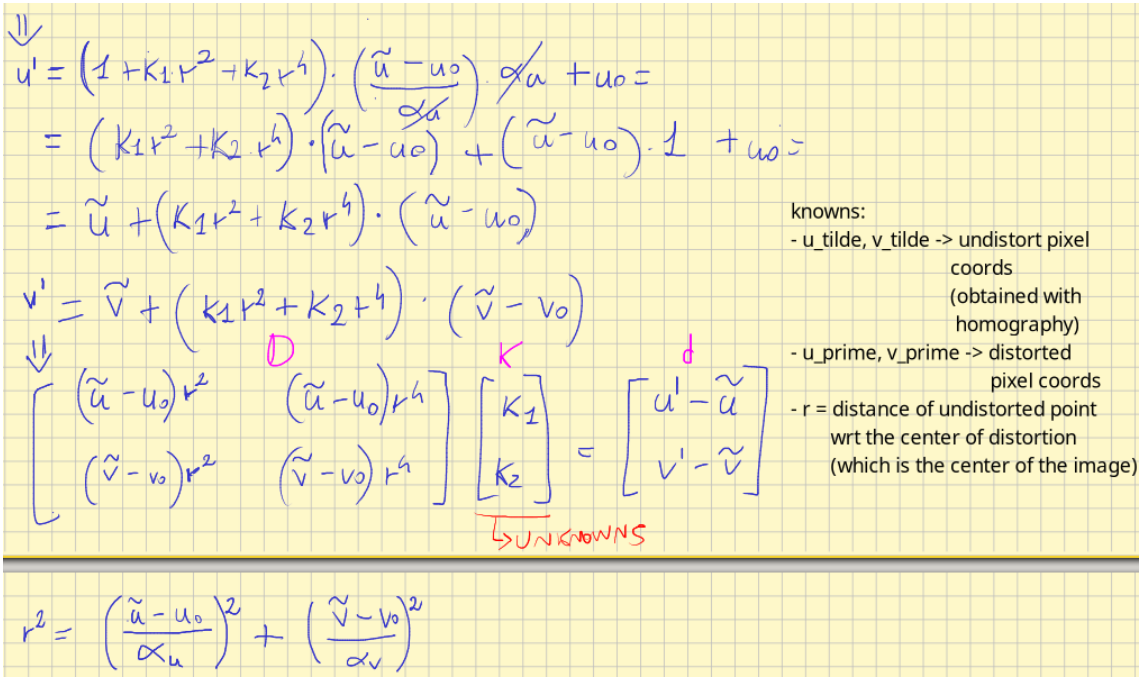


Figure 2.44

- for 1 control point (1 corner) I have 2 equations in 2 unknownks
  - 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 x 2)
    - k -> vector of the unknowns (2mn x 1)
    - d -> vector of knowns
  - this system (Dk = d) can be solved as a **least square problem**:

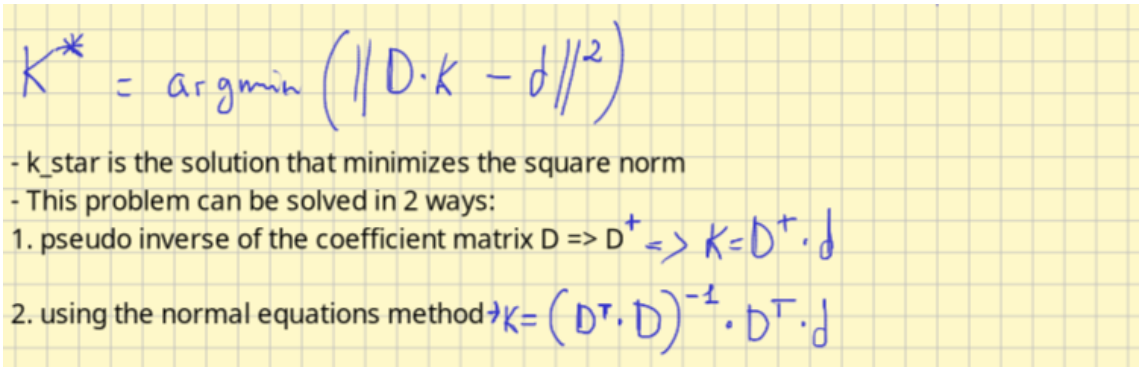


Figure 2.45

2.6 Refinement by non-linear optimization - Zhang's

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

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

- $m_{i,j}$ -> generic corner found by the Harris corner detector into the current image.
  - i index -> goes through the images
  - j index -> " " corners of the image.
  - => there are m·n corners.

- $\hat{m}$ -> pixel position predicted by our model
  - A = intrinsics
  - k = lens distortion parameters
  - Ri, Ti -> estrinsics for the current image i
  - wj = position of the world point

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

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

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

=> 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?

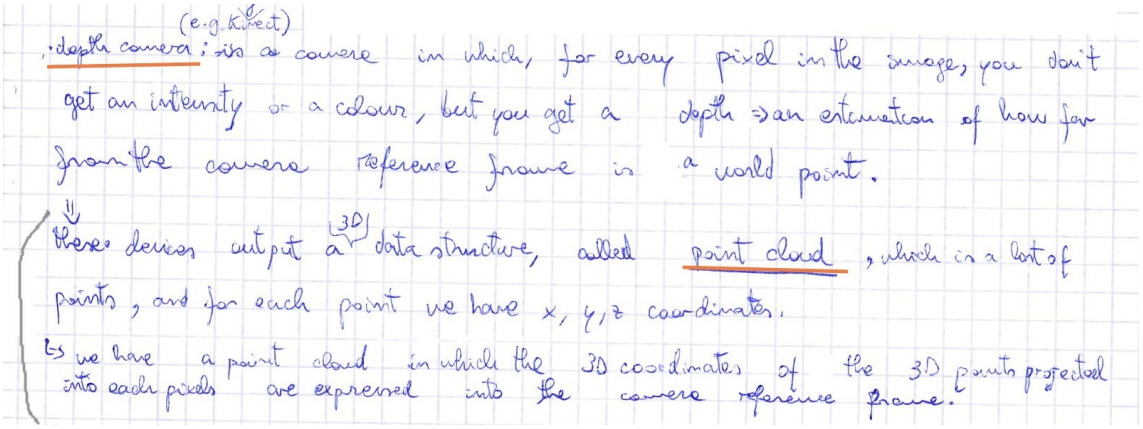


Figure 2.47

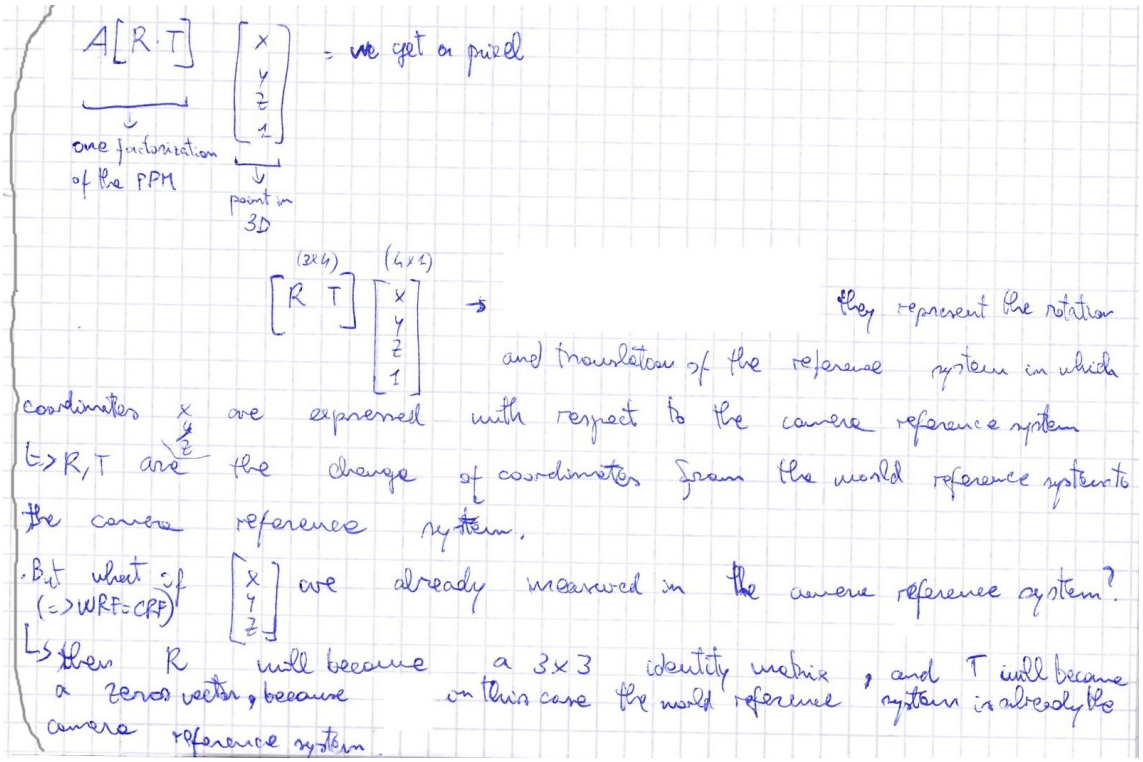


Figure 2.48

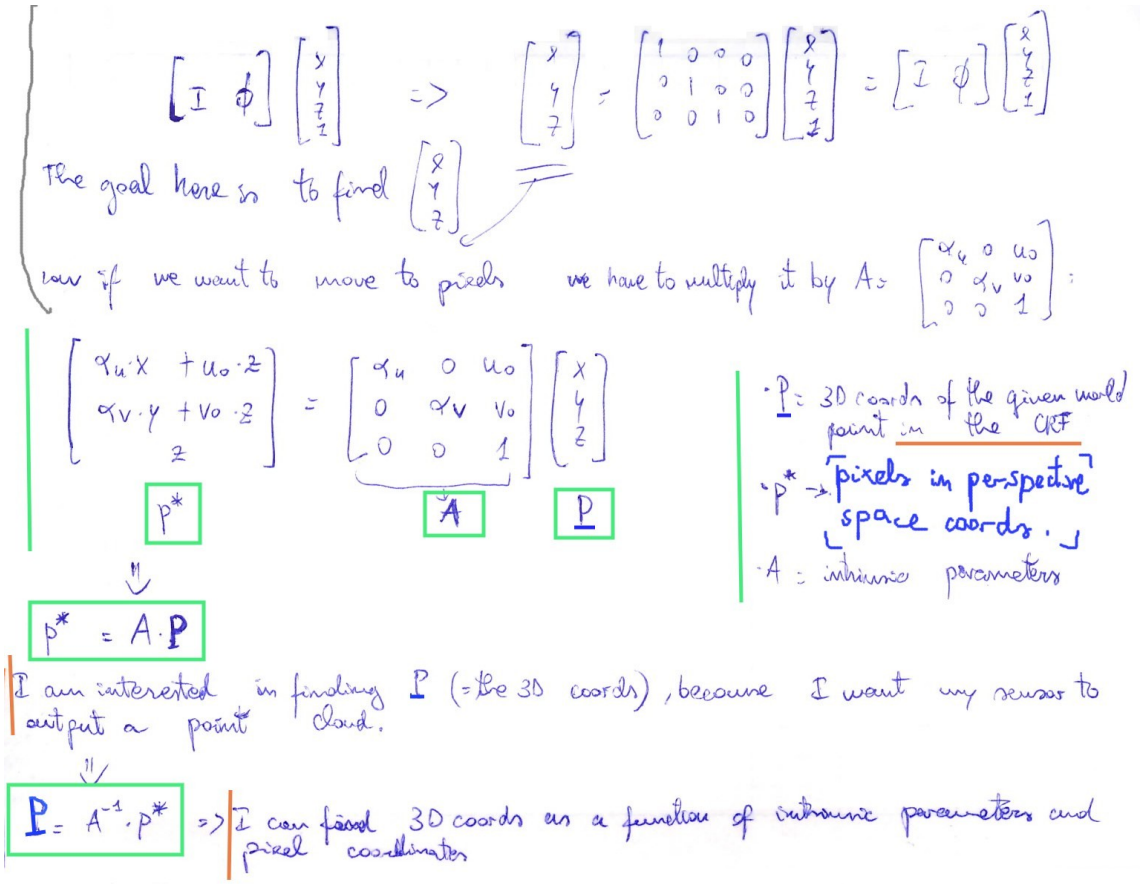

The goal here is to find
raw if we want to move to pixels we have to multiply it by A:
P: 3D coords of the given world point in the CRF
p* → pixels in perspective space coords.
A: intrinsic parameters
I am interested in finding P (=the 3D coords), because I want my sensor to output a point cloud.
=> I can found 3D coords as a function of intrinsic parameters and pixel coordinates

Figure 2.49

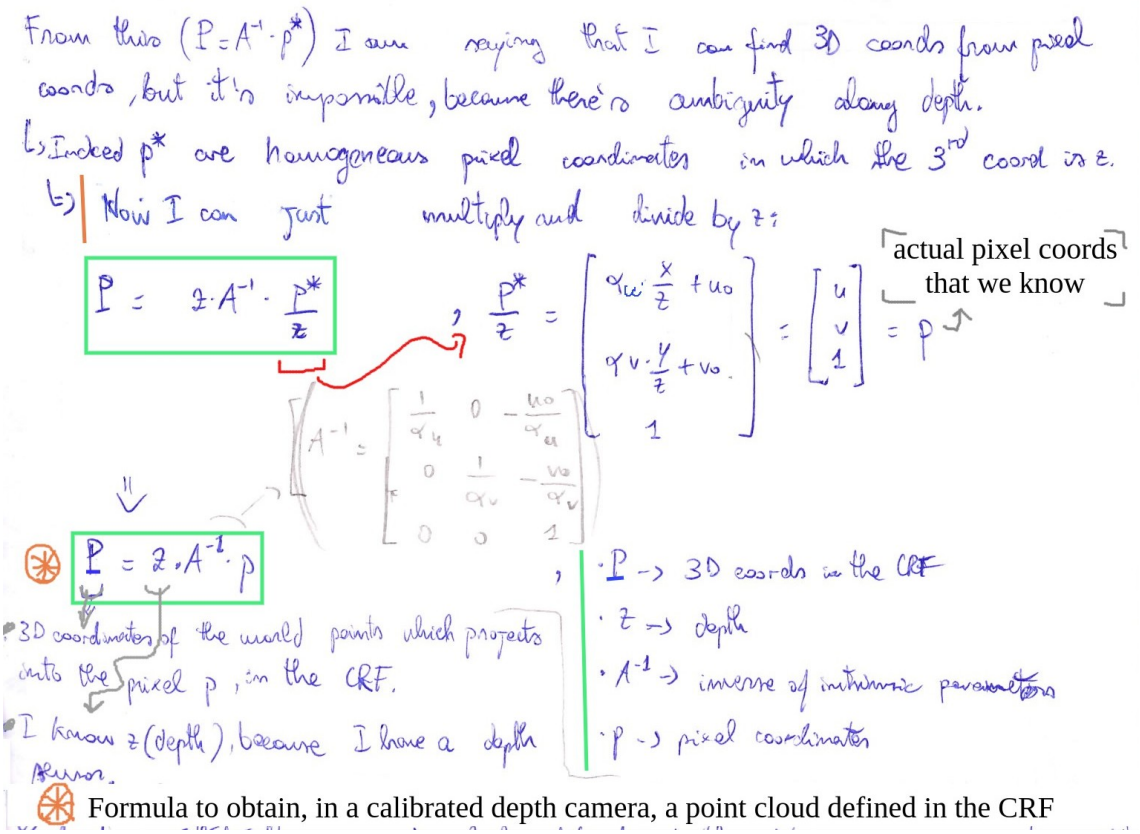

From this ($P = A^{-1} \cdot p^*$) I am saying that I can find 3D coords from pixel coords, but it's impossible, because there's ambiguity along depth.
↳ Indeed p^* are homogeneous pixel coordinates in which the 3rd coord is z.
↳ Now I can just multiply and divide by z:
$P = z \cdot A^{-1} \cdot \frac{p^*}{z}$
$\frac{p^*}{z} = \begin{bmatrix} \alpha_u \cdot \frac{x}{z} + u_0 \\ \alpha_v \cdot \frac{y}{z} + v_0 \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = p$
actual pixel coords that we know
$A^{-1} = \begin{bmatrix} \frac{1}{\alpha_u} & 0 & -\frac{u_0}{\alpha_u} \\ 0 & \frac{1}{\alpha_v} & -\frac{v_0}{\alpha_v} \\ 0 & 0 & 1 \end{bmatrix}$
$P = z \cdot A^{-1} \cdot p$
· P -> 3D coords in the CRF
· z -> depth
· A^{-1} -> inverse of intrinsic parameters
· p -> pixel coordinates
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)

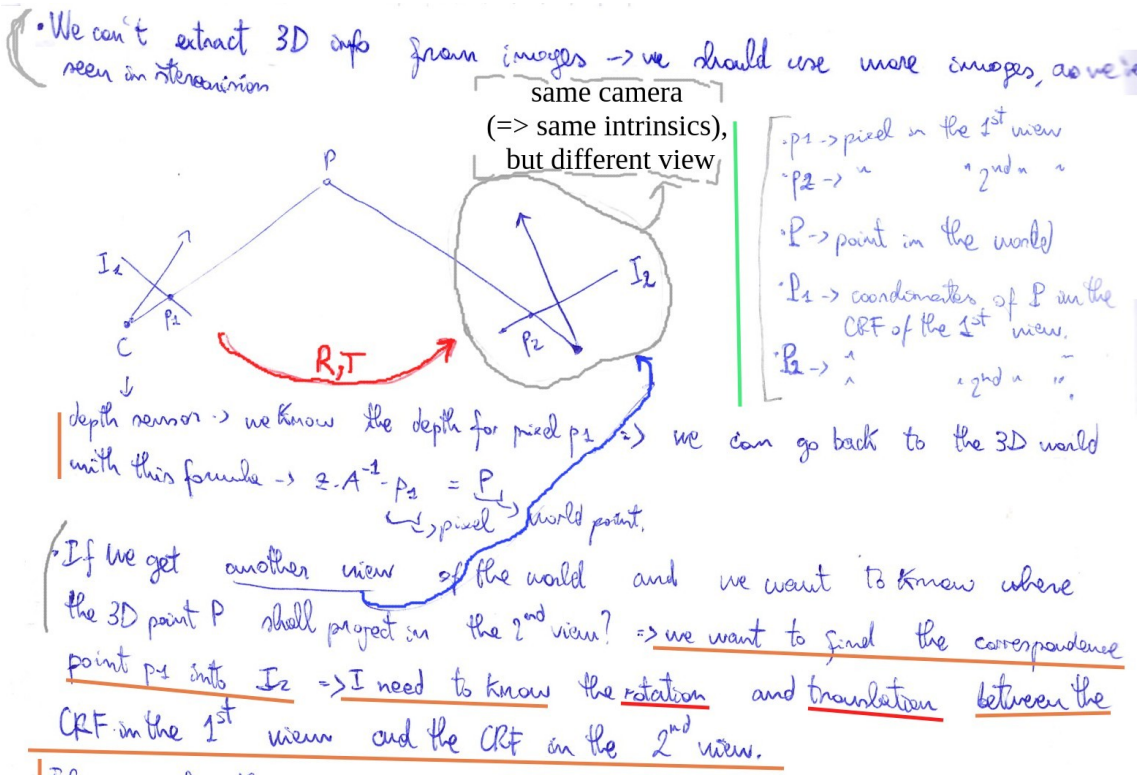


Figure 2.51

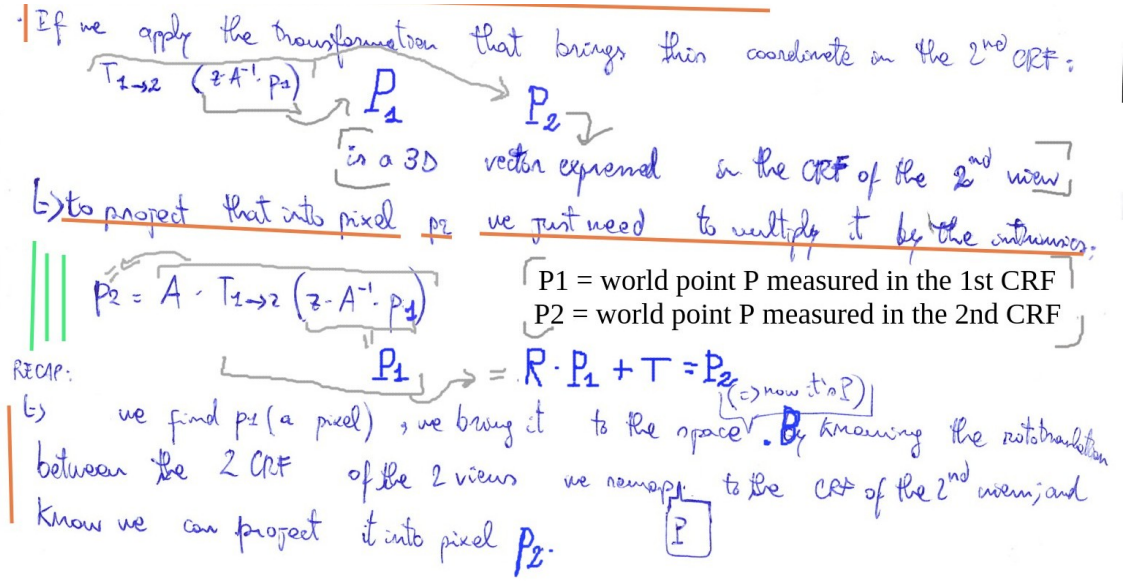


Figure 2.52

Figure 2.53

2.7.3 case with 2 different cameras

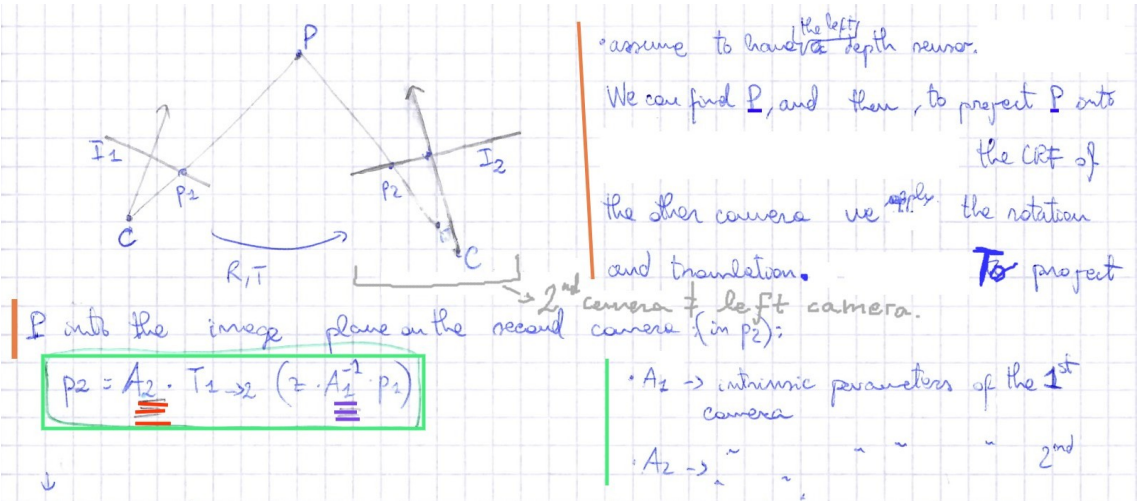


Figure 2.54

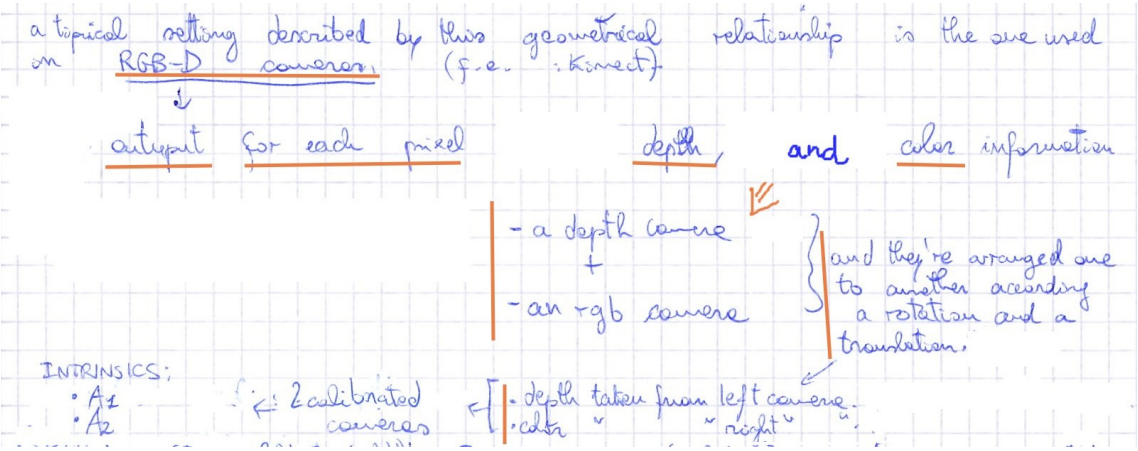


Figure 2.55

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 vale function f (fu, fv), which takes the coords in image I and computes corresponding coords in image I'.

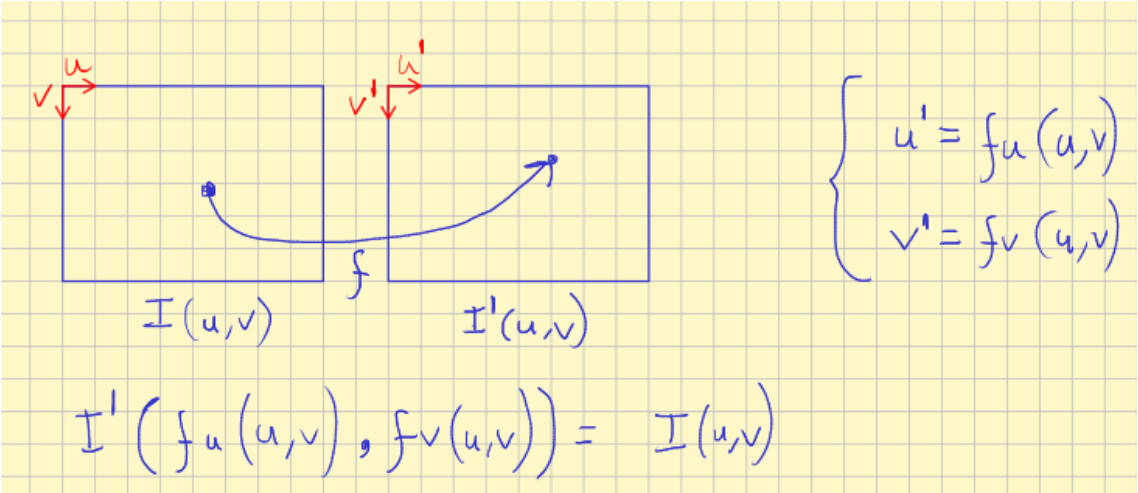


Figure 2.56

- **Def warping:** warping is defined through these 2 functions:
  1. fu gives the new u coord (u') given the old u,v coords
  2. fv 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 -> is transferred at pos (u',v') (or fu(u,v), fv(u,v)) in target image I'
- examples of image warping:
  - rotation
  - remove deformations (license plate)
- How to perform warping?

2.8.1 Forward Mapping

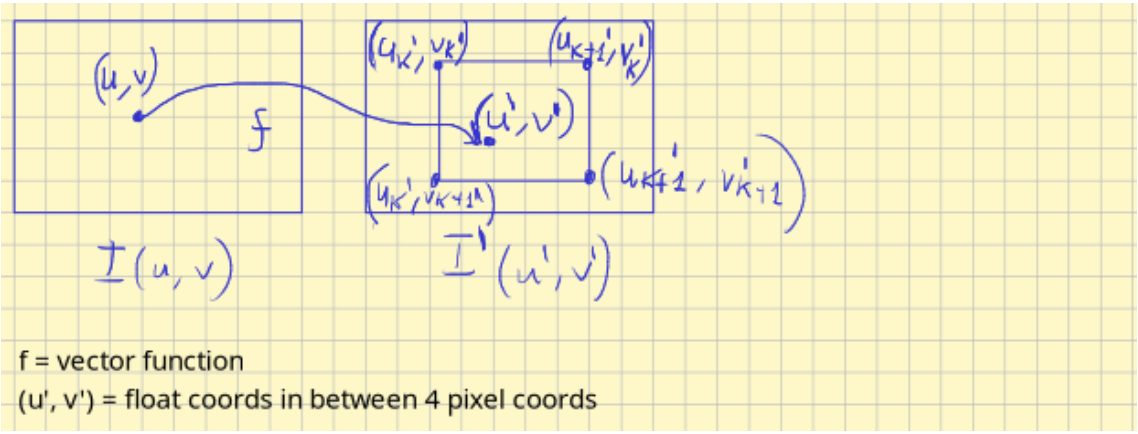


Figure 2.57

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

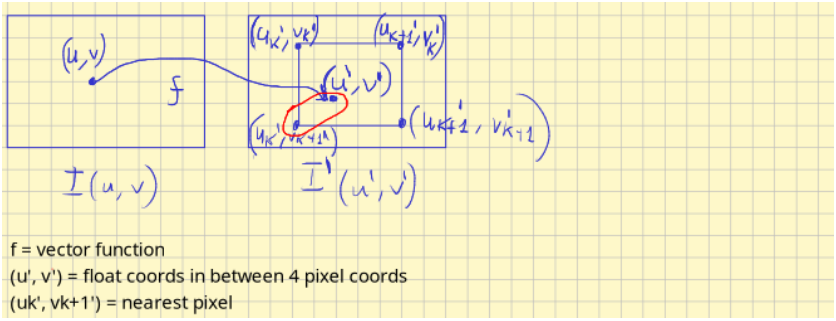


Figure 2.58

  - 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:

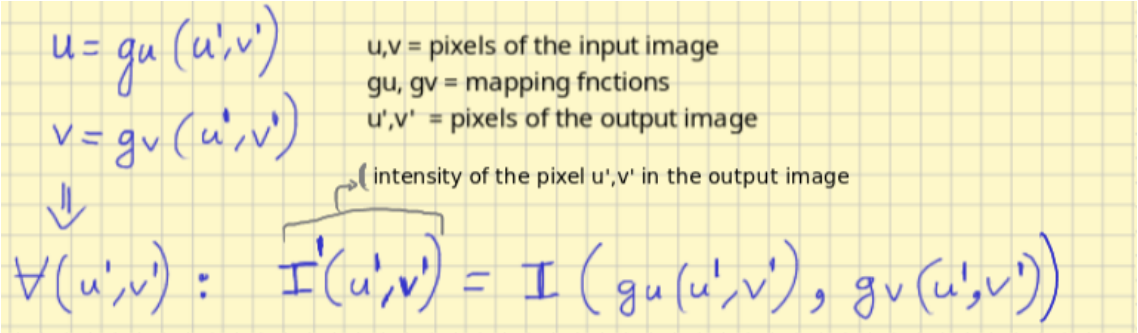


Figure 2.59

- **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

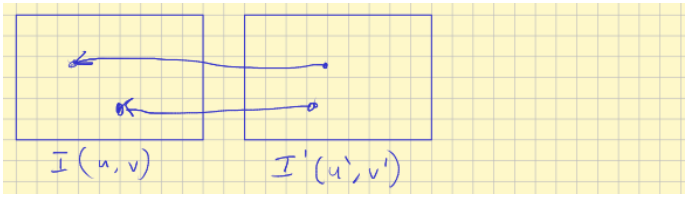


Figure 2.60

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:

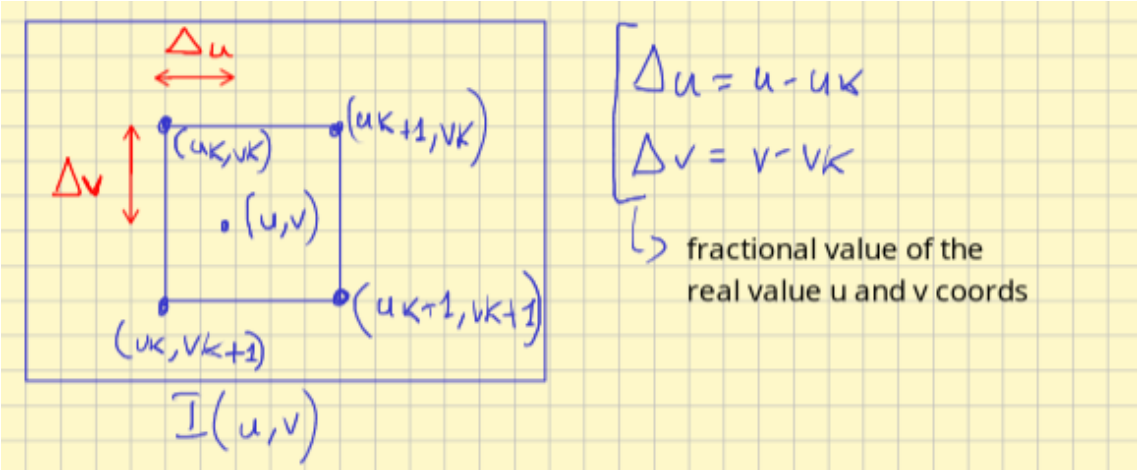


Figure 2.61

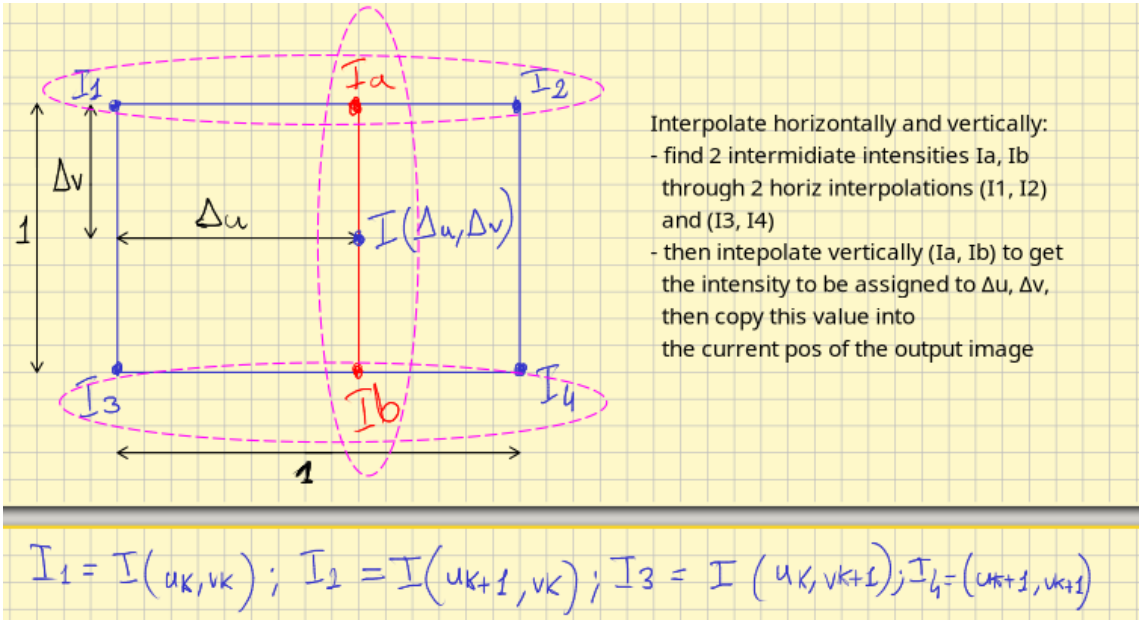


Figure 2.62

- How to compute a linear interpolation:

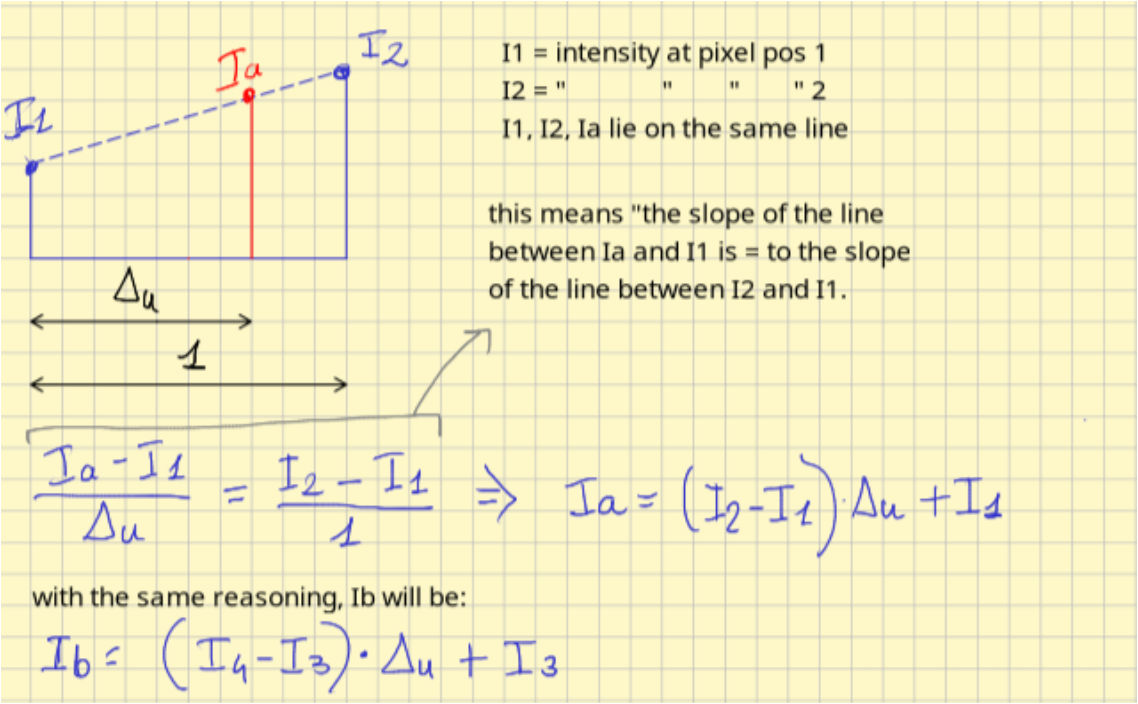


Figure 2.63

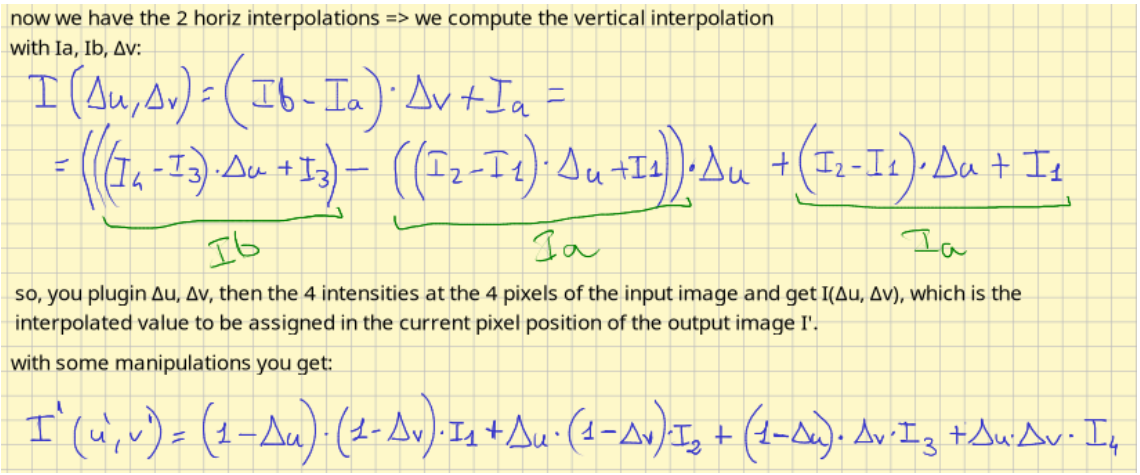


Figure 2.64

- 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. $\Delta u, \Delta v = 0$ => I'(u',v') = I1 (I'm on the top left corner).
    2. $\Delta u, \Delta v = 1$ => I'(u',v') = I4 (I'm 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 (k1, k2)

– backward warp from undistorted to distorted image based on the lens distort model

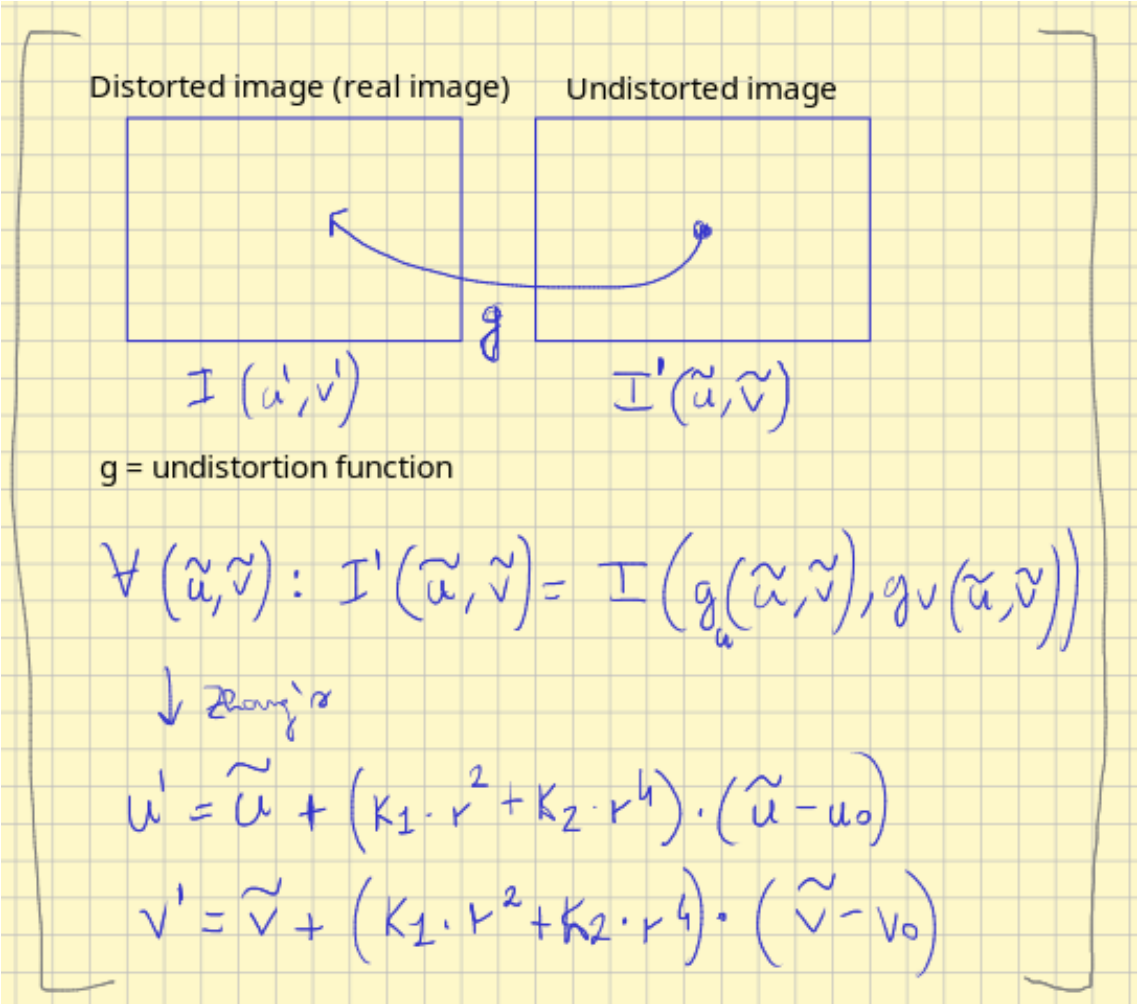


Figure 2.65

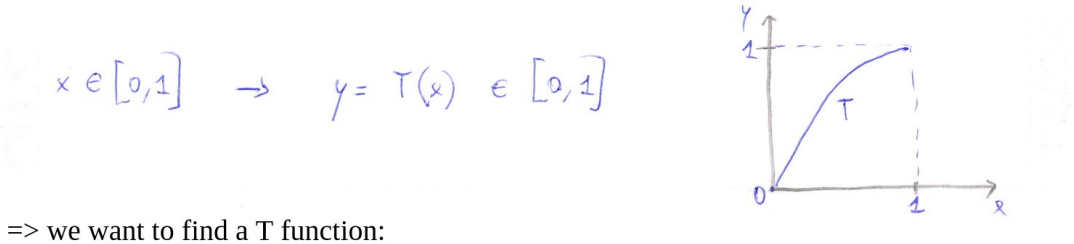
Chapter 3

Intensity Transformations

3.1 Histogram Equalization

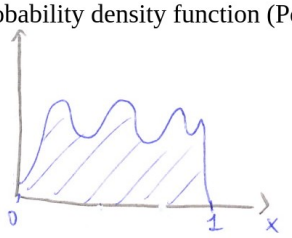
- is an Intensity Transformation Operator,
  - and it's fully automatic.
- Goal: IMPROVING THE CONTRAST of an image.
  - this is achieved through an equalizing function that TRIES to spread gray levels uniformly across the whole range.
    - We don't get a uniform histogram in practice due to the discrete 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 -> continuous random variable
- T -> a strictly monotonically increasing function -> it will give us another random variable y

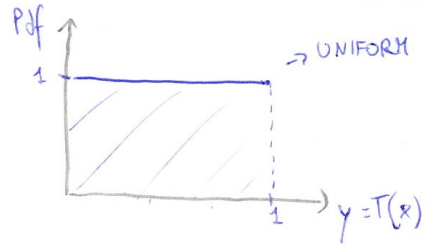


=> we want to find a T function:
- that maps x into a UNIFORM random variable y
  (random variable whose probability distribution is flat)

probability density function (Pdf) of x



probability density function of y -> flat

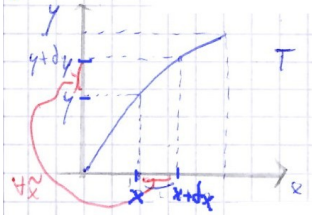


N.B.: Whatever x, it's always possible to turn x into a uniform random variable with a flat Pdf

=> 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] -> infinitesimal interval
- T(x) = y
- T(x+dx) = y+dy

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

=> Whatever x_tilde I take (=> 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

=> 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].

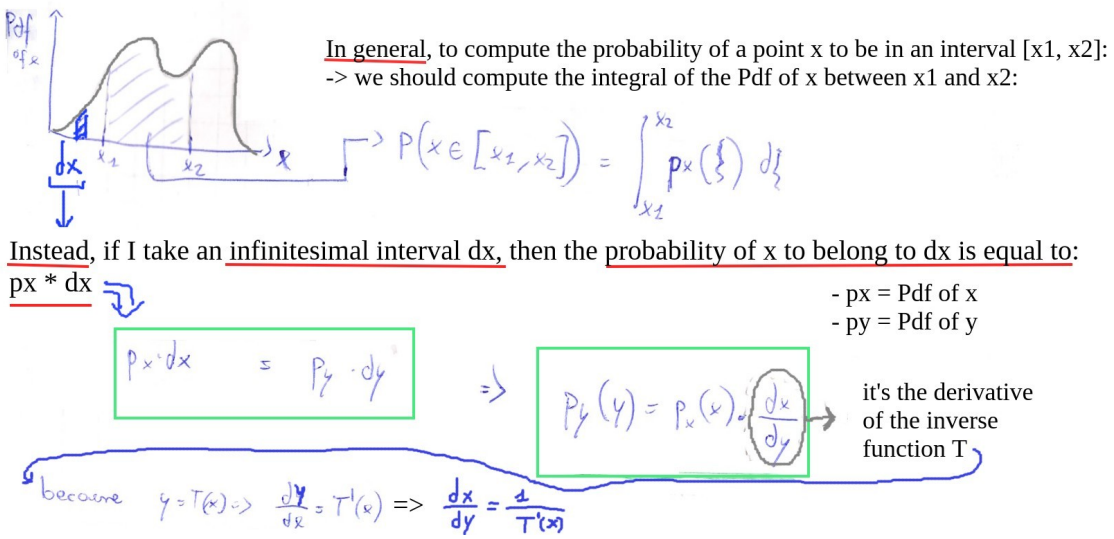
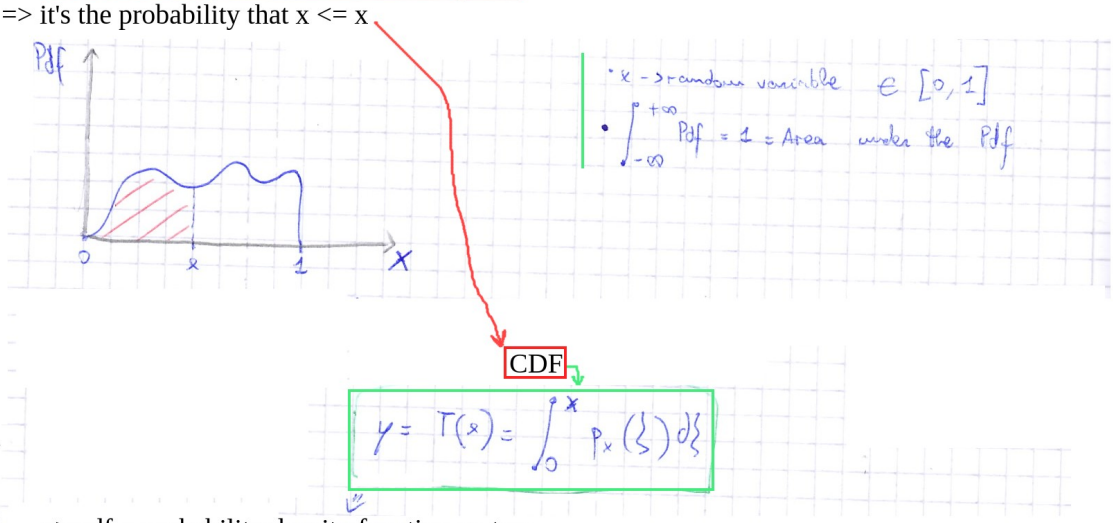


Figure 3.3

- 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 <= 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:

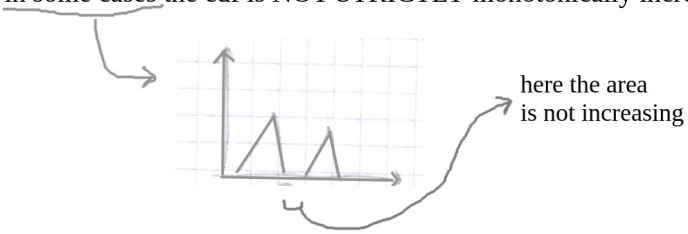
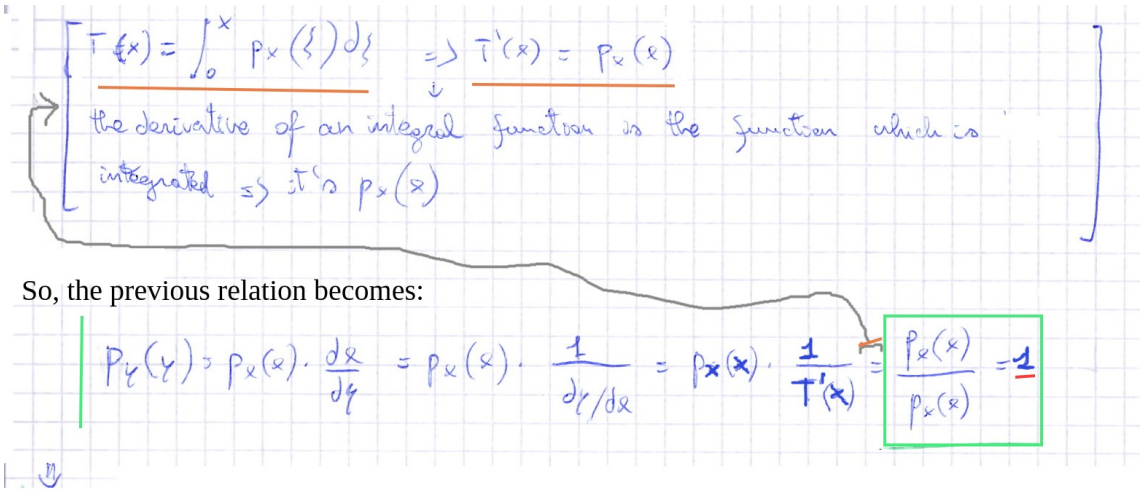


Figure 3.4



=> we have turned x (whatever is its Pdf => its px(x)) into a random variable y, whose Pdf (py(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:

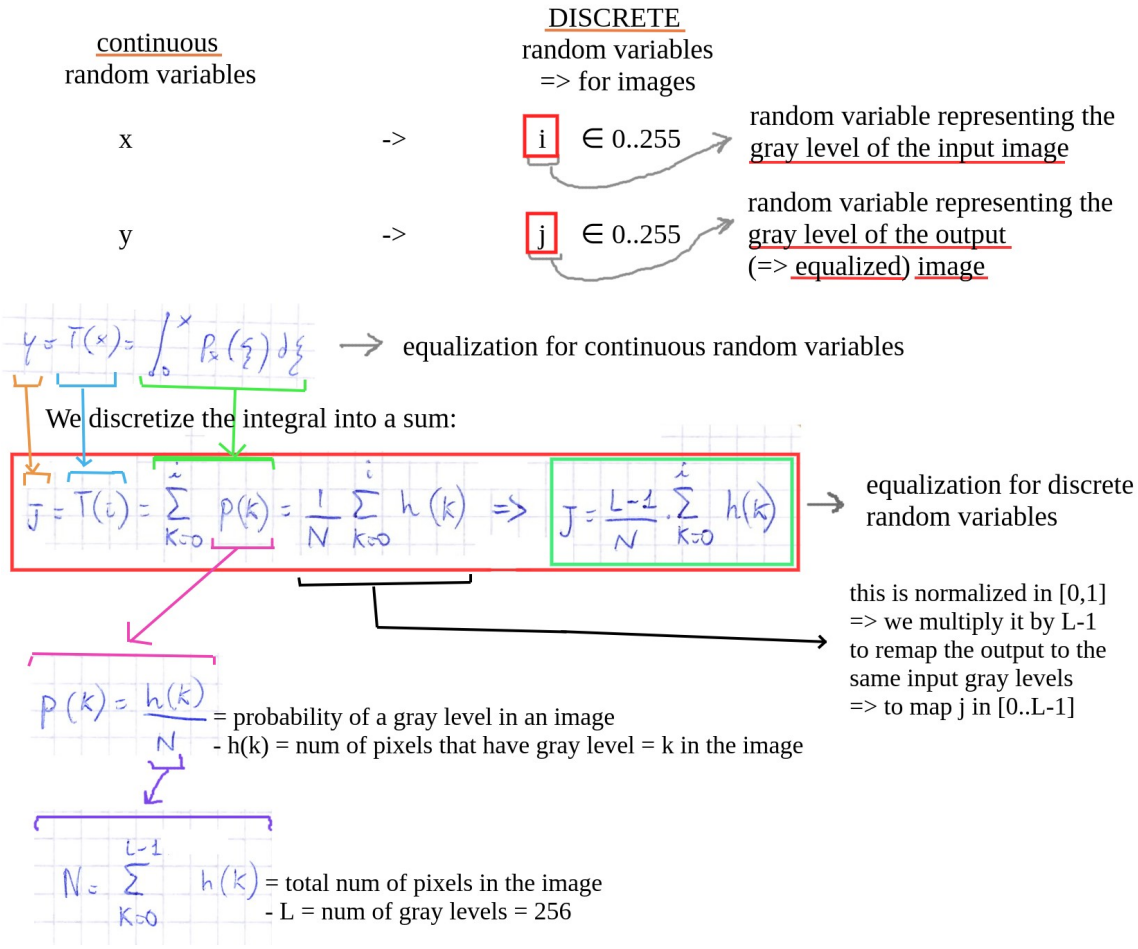
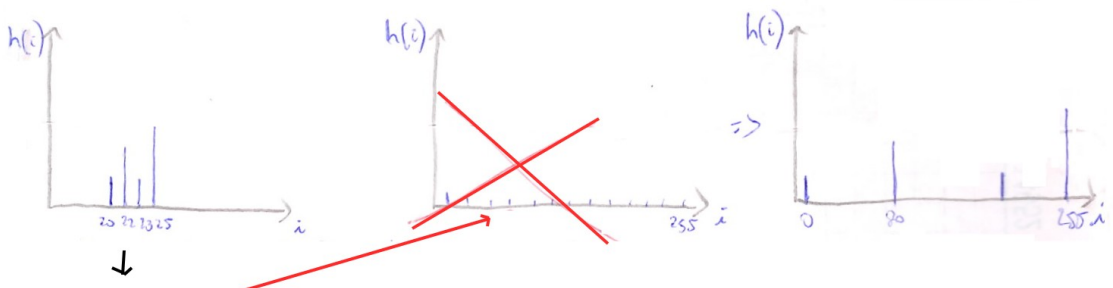


Figure 3.6

- 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) (=> a function of i)

=> 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 => 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
=> at least you can reduce the number of gray levels.

Figure 3.7

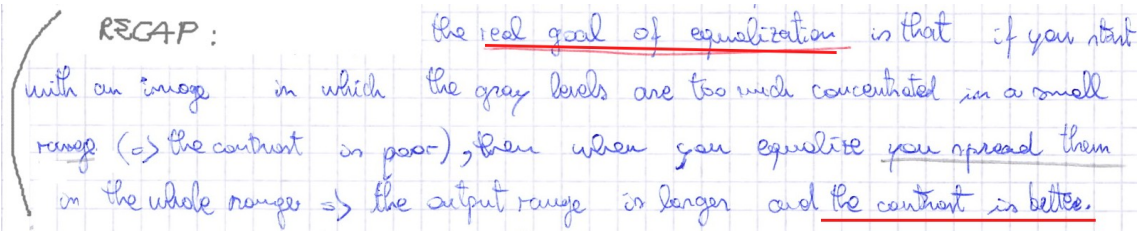


Figure 3.8

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):

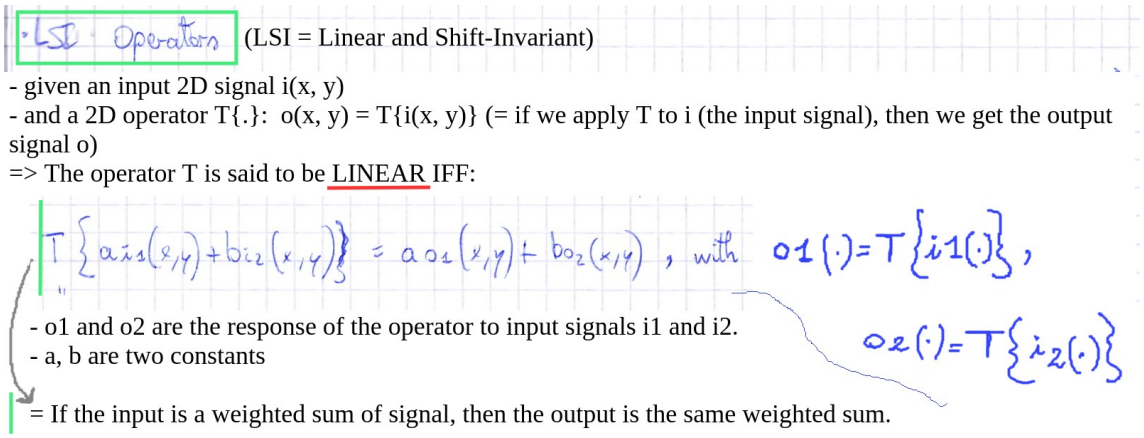


Figure 4.1

The operator T is SHIFT-INVARIANT IFF:

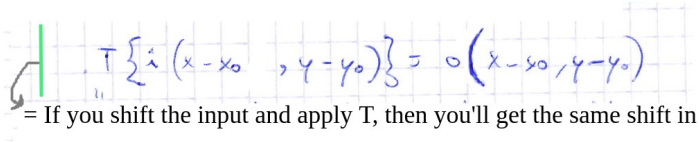


Figure 4.2

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

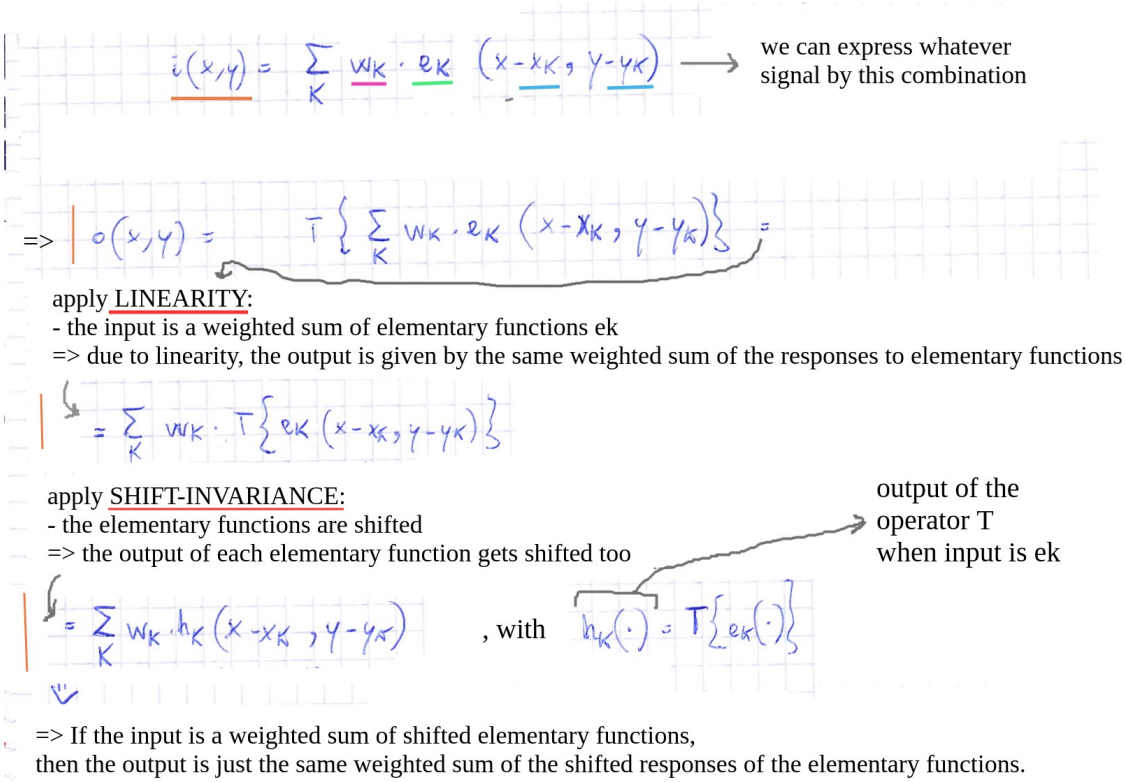


Figure 4.3

- Convolution:

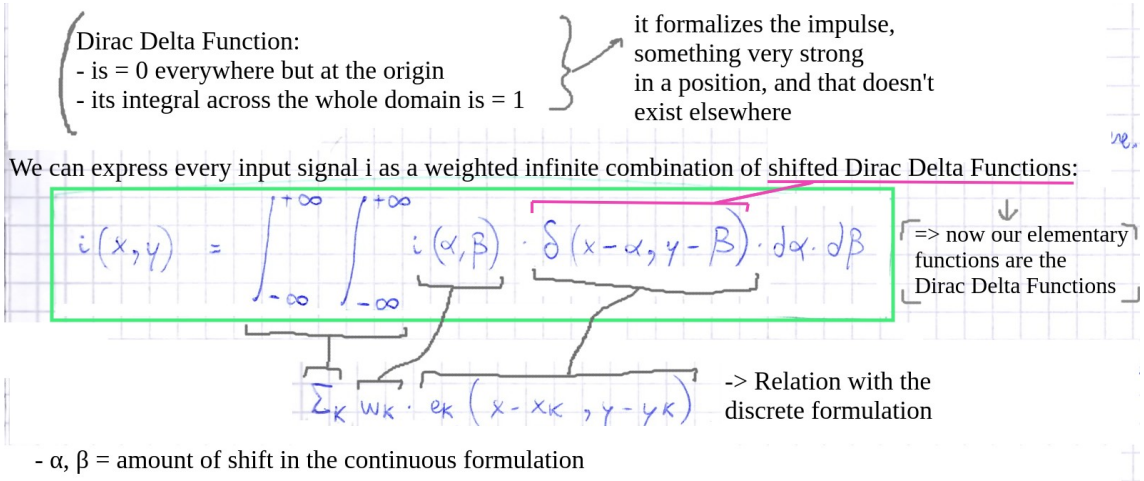


Figure 4.4

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( ) is the output I get if I feed my operator T with the elementary function

$$h = T\{\delta(x,y)\}$$

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

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

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

Figure 4.5

- Ex of Convolution calculus:

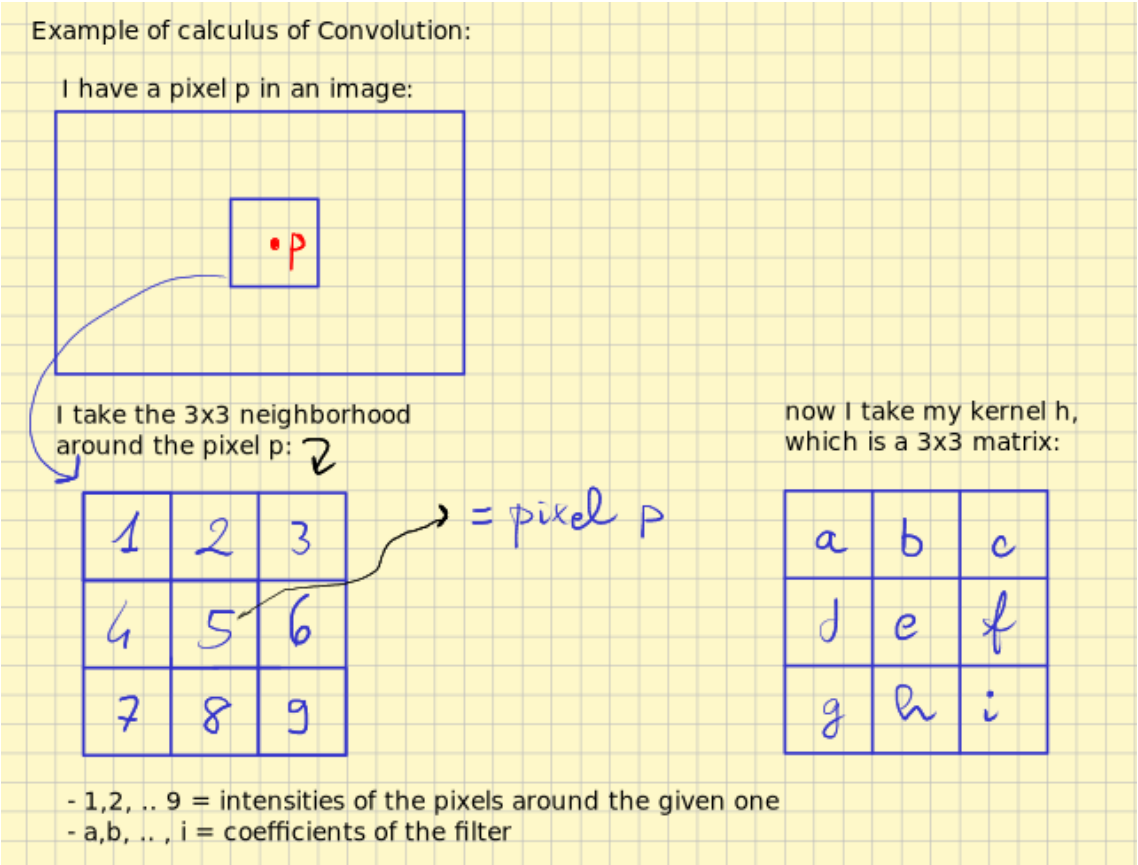


Figure 4.6

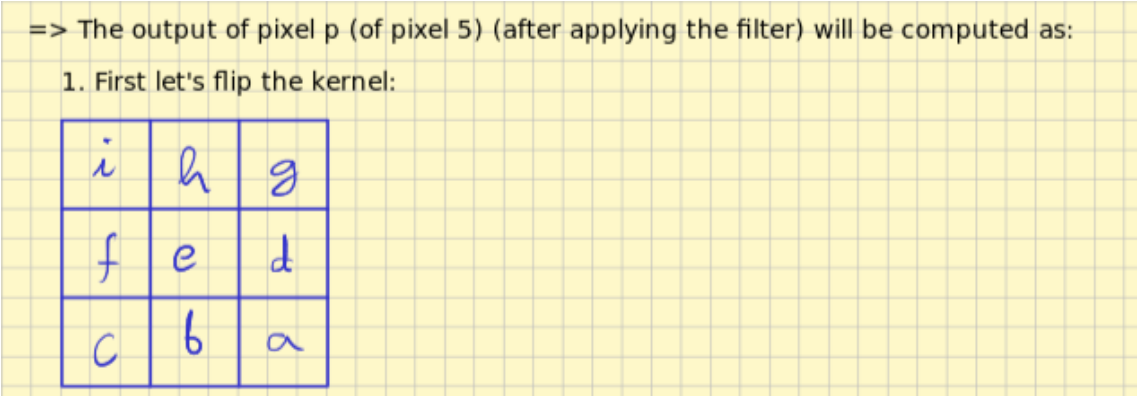


Figure 4.7

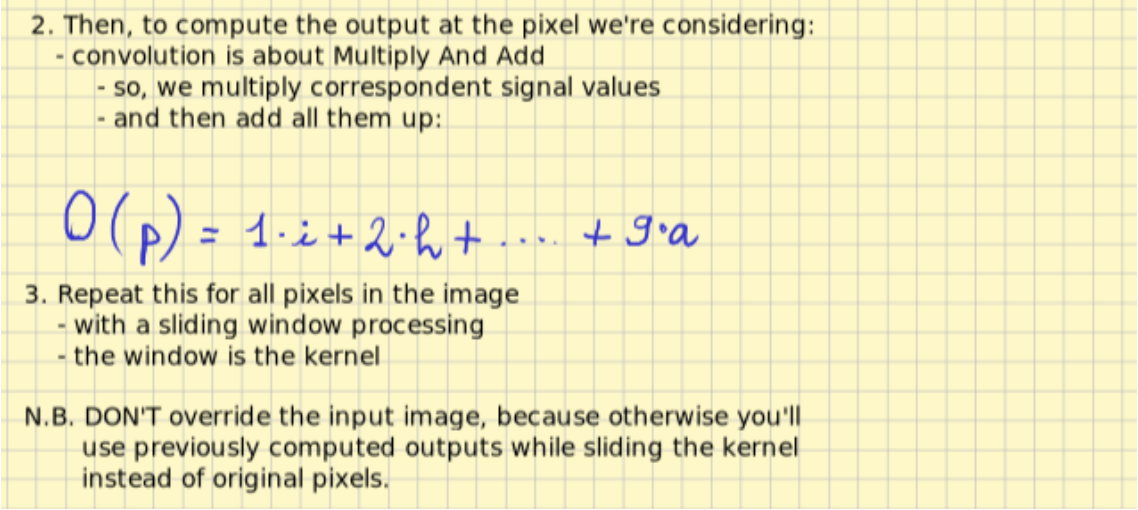


Figure 4.8

- Problems with Practical Convolution:

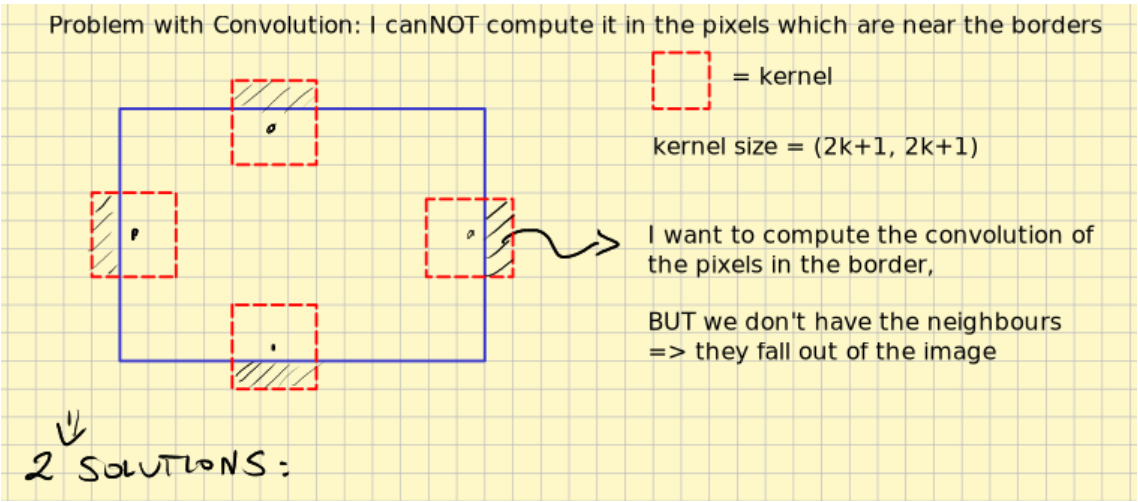


Figure 4.9

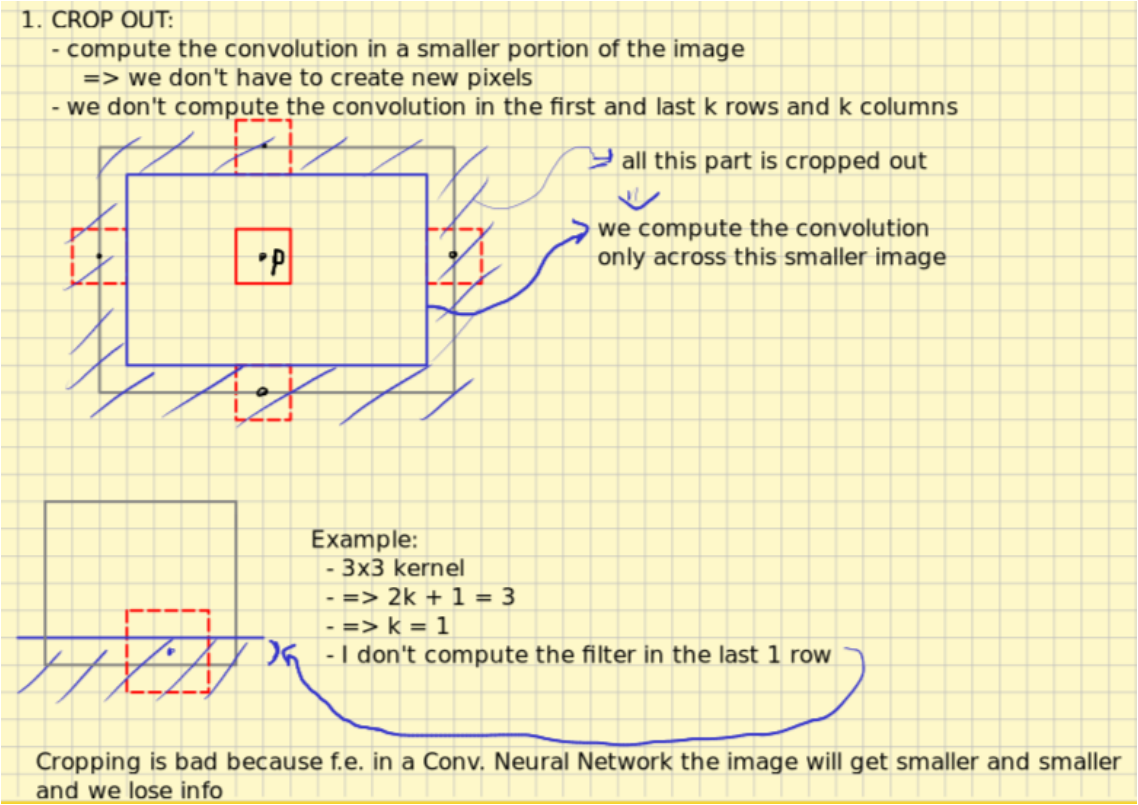


Figure 4.10

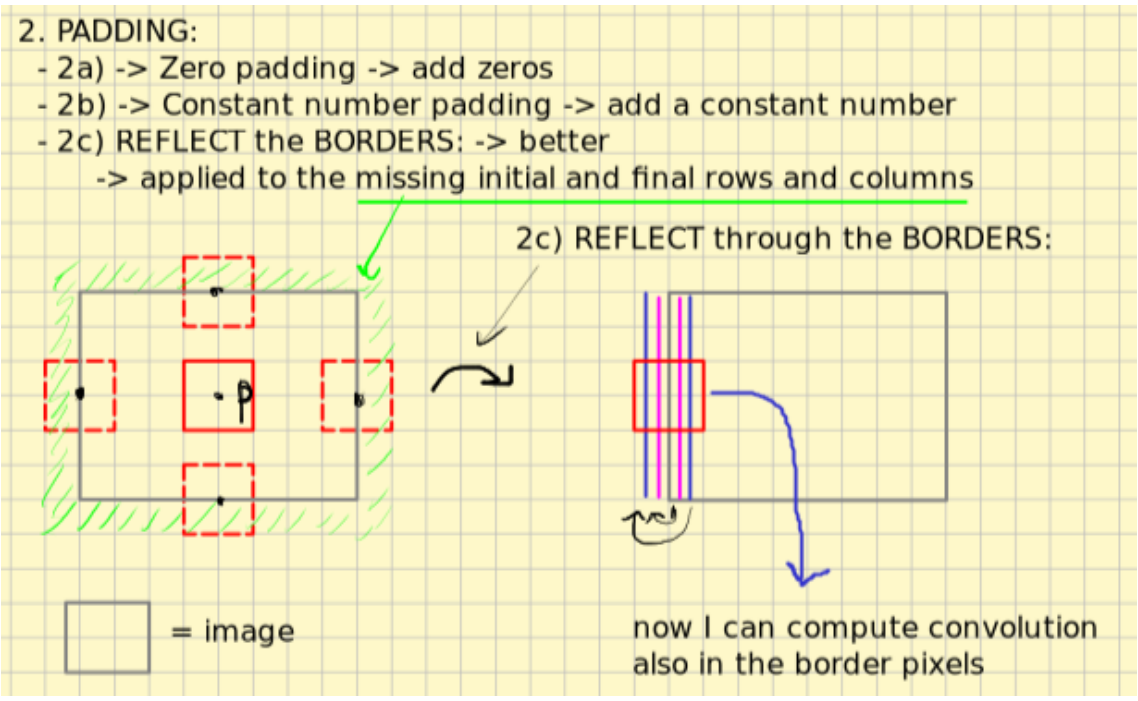


Figure 4.11

- Properties of Convolution:

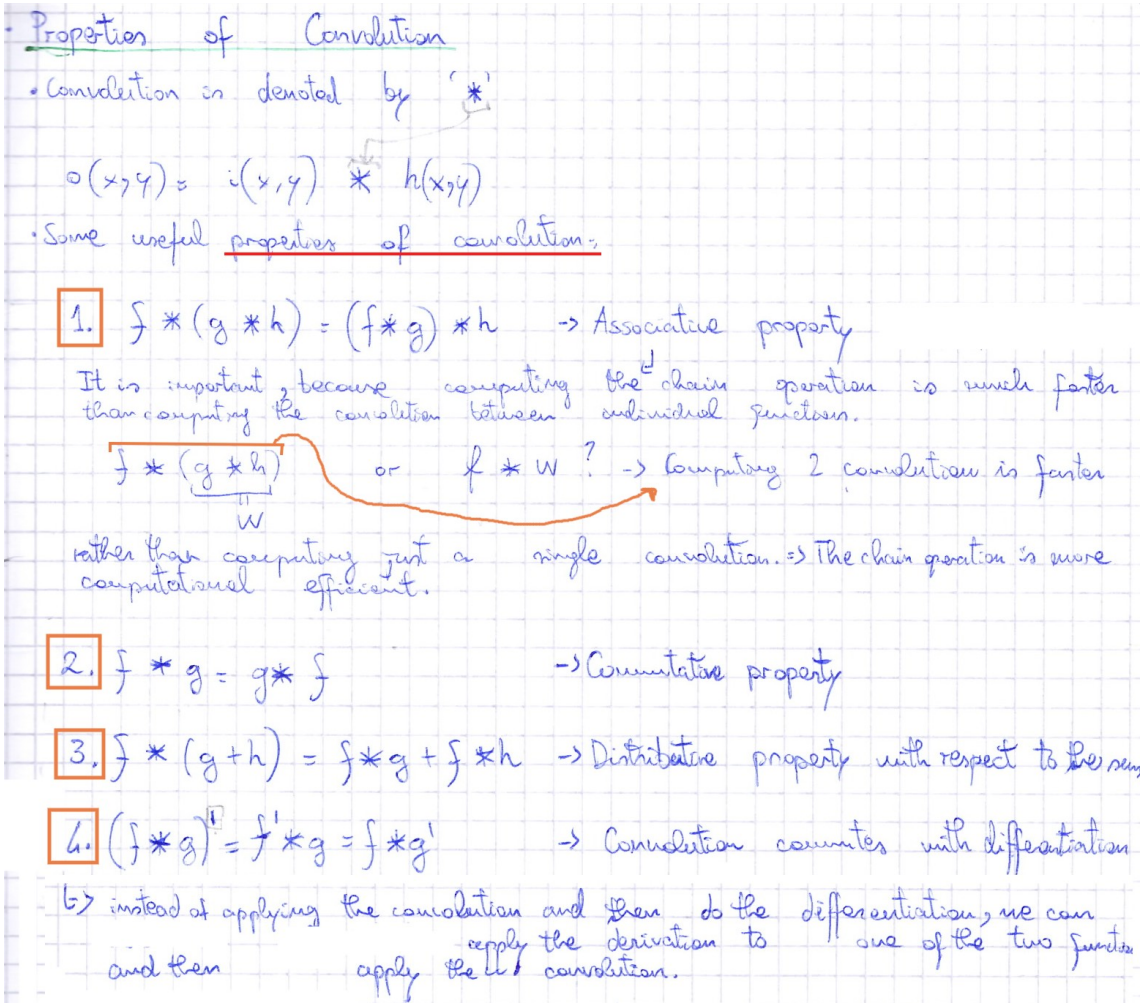


Figure 4.12

- Graphical view of Convolution:

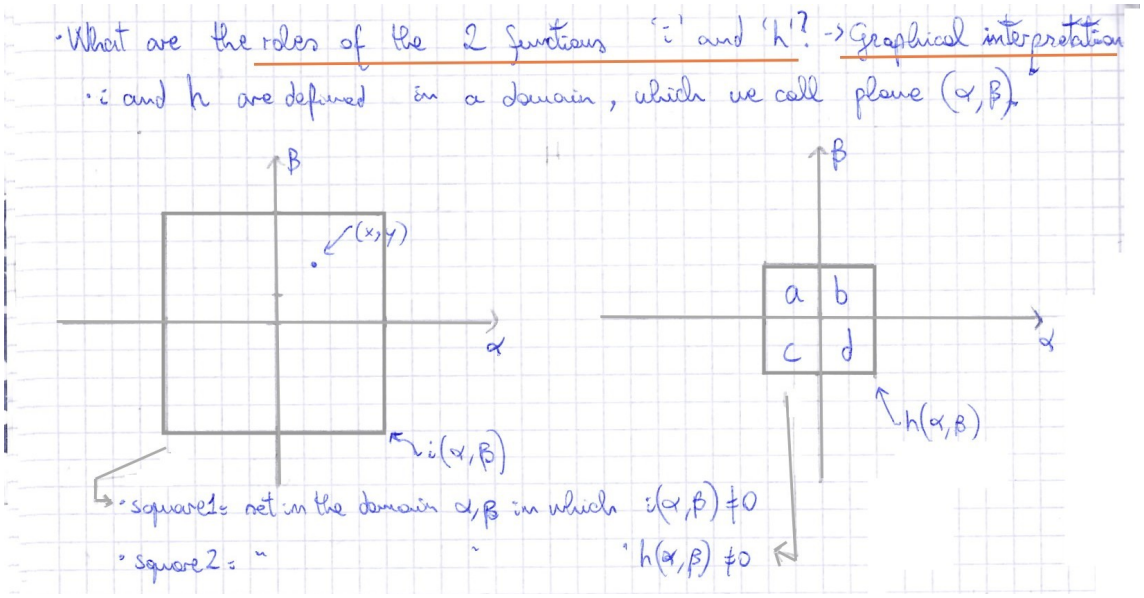


Figure 4.13

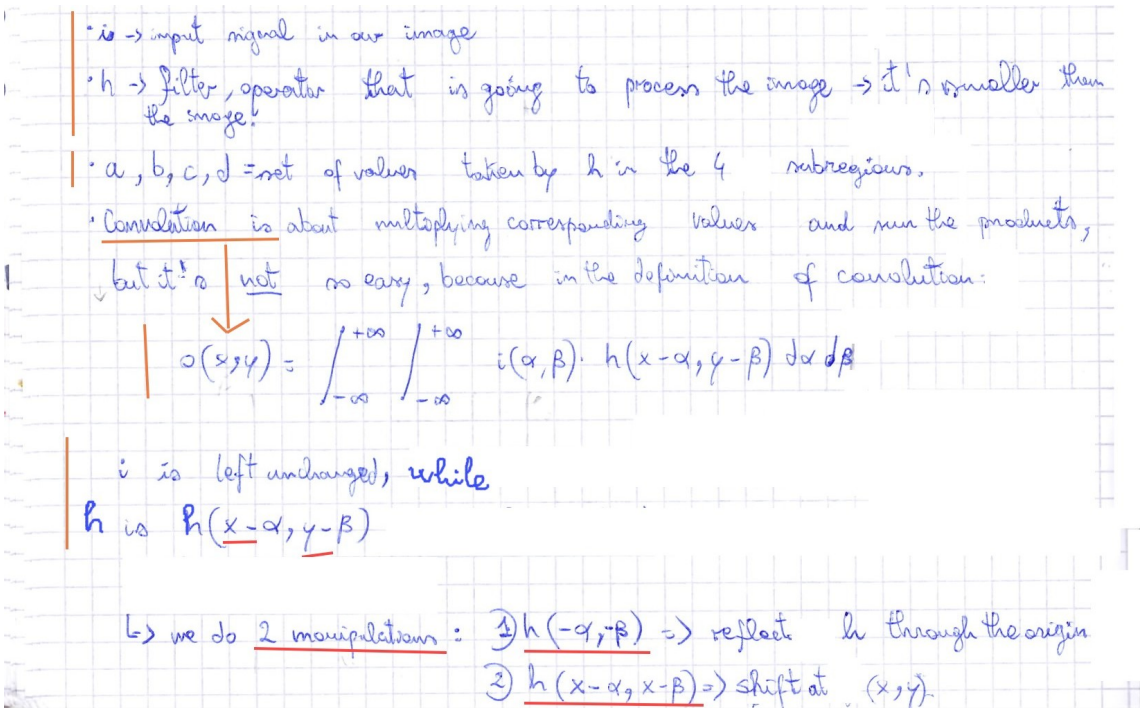

• i -> input signal in our image
• h -> filter, operator that is going to process the image -> it's smaller than the image!
• a, b, c, d = set of values taken by h in the 4 subregions.
• Convolution is about multiplying corresponding values and sum the products, but it's not so easy, because in the definition of convolution:
$o(x,y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} i(\alpha,\beta) \cdot h(x-\alpha, y-\beta)\, d\alpha\, d\beta$
i is left unchanged, while
h is $h(x-\alpha, y-\beta)$
-> we do 2 manipulations: 1) $h(-\alpha,-\beta)$ => reflect h through the origin
2) $h(x-\alpha, x-\beta)$ => shift at (x,y).

Figure 4.14

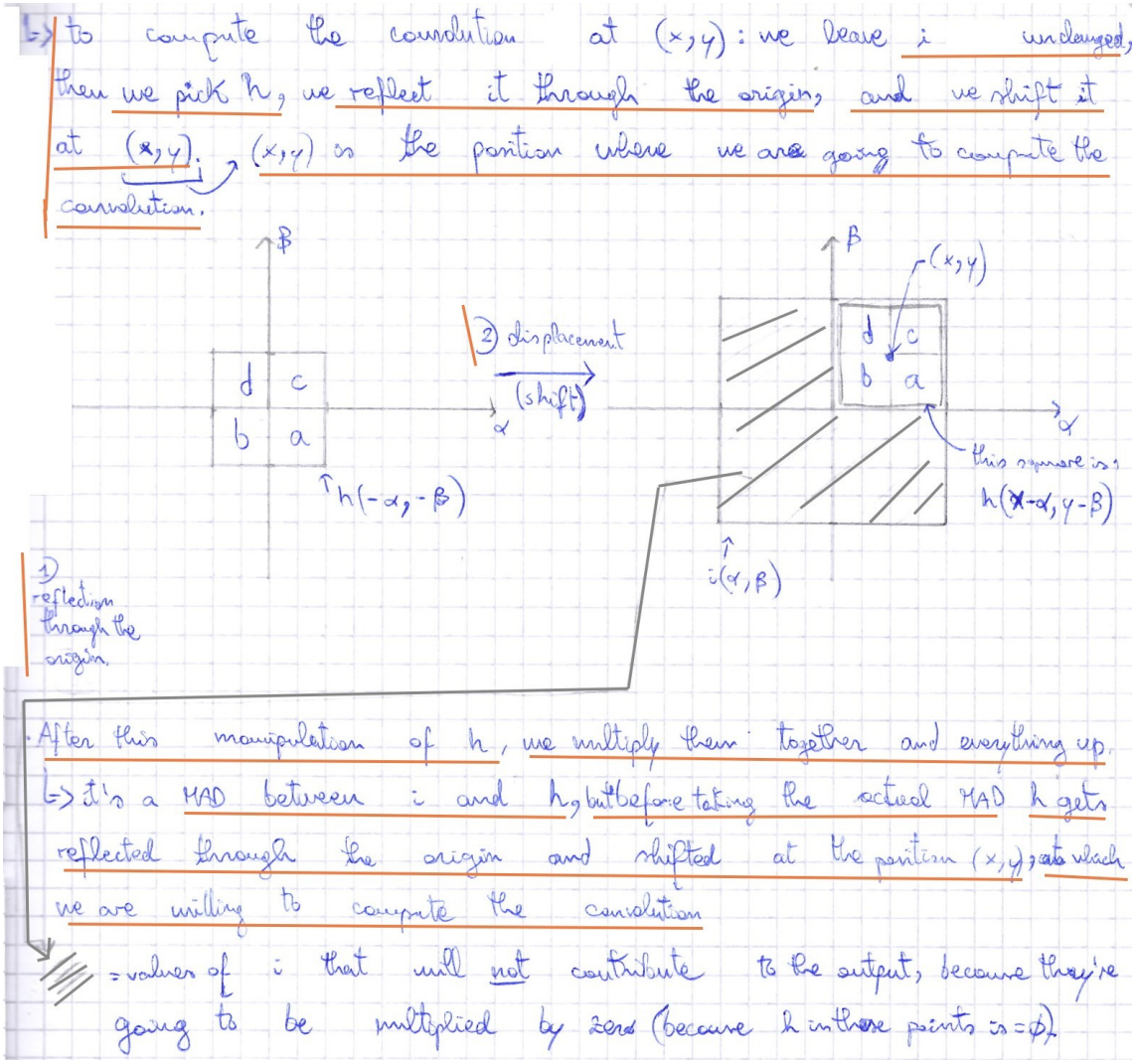


Figure 4.15

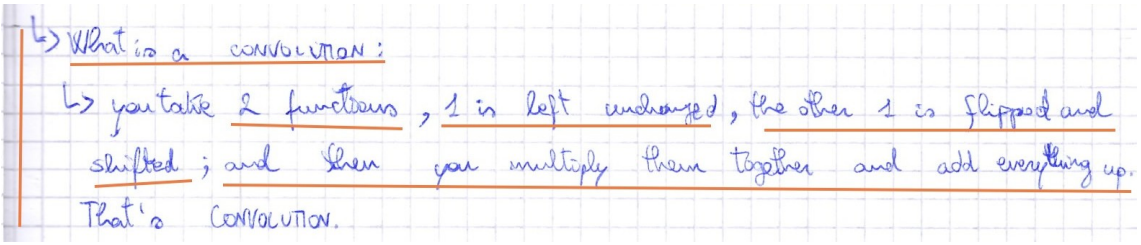


Figure 4.16

- Discrete Convolution:

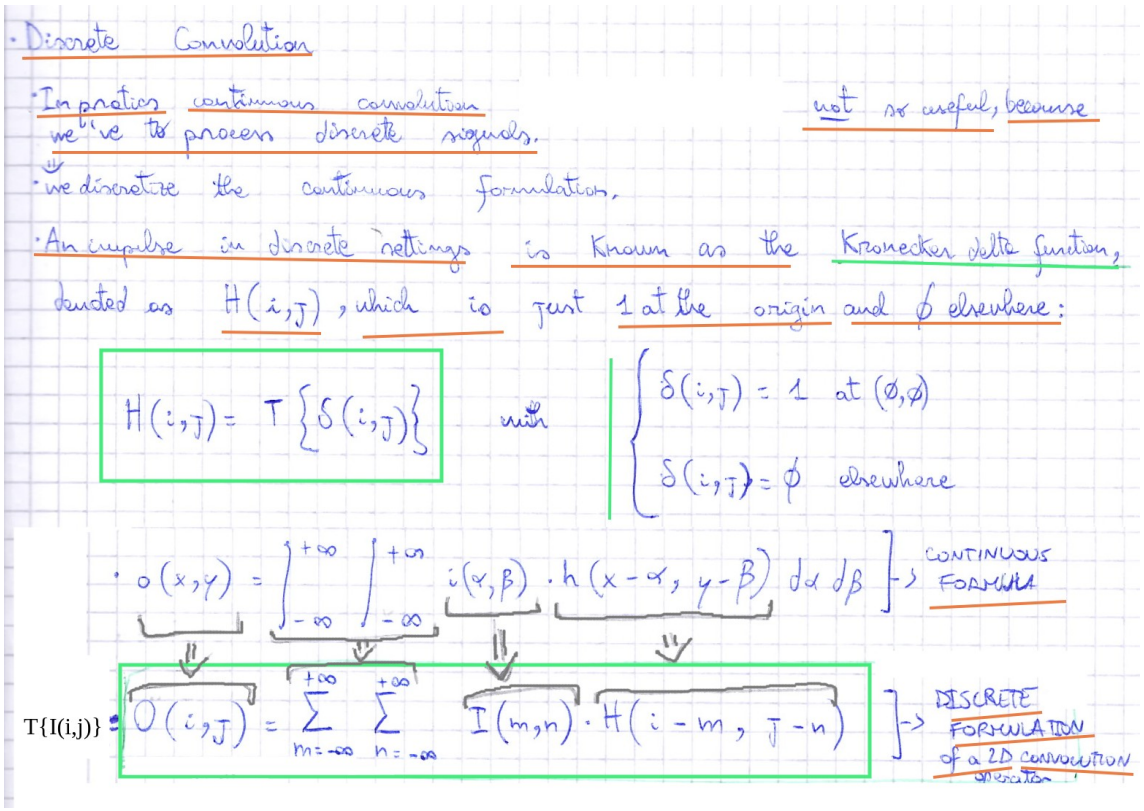


Figure 4.17

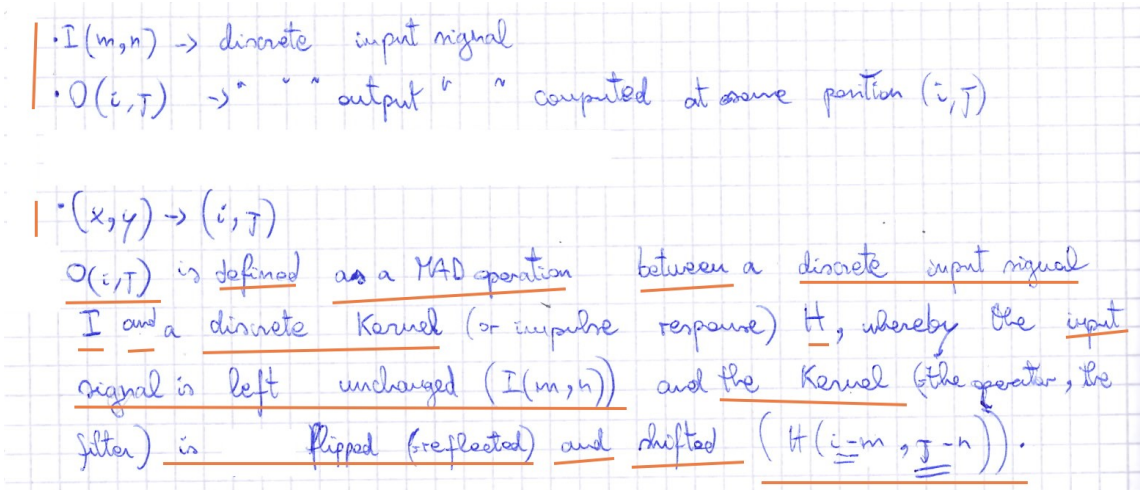


Figure 4.18

4.2 Correlation

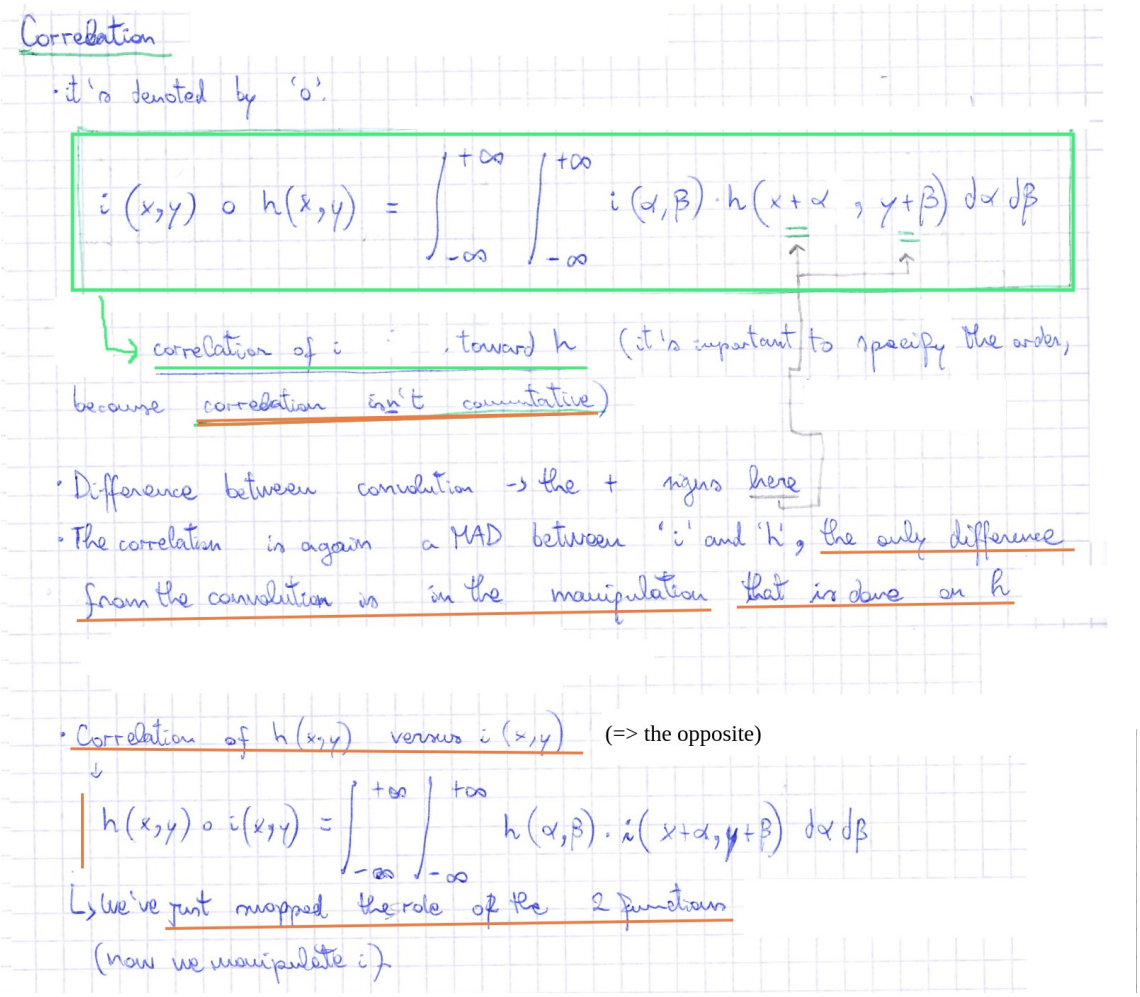


Figure 4.19

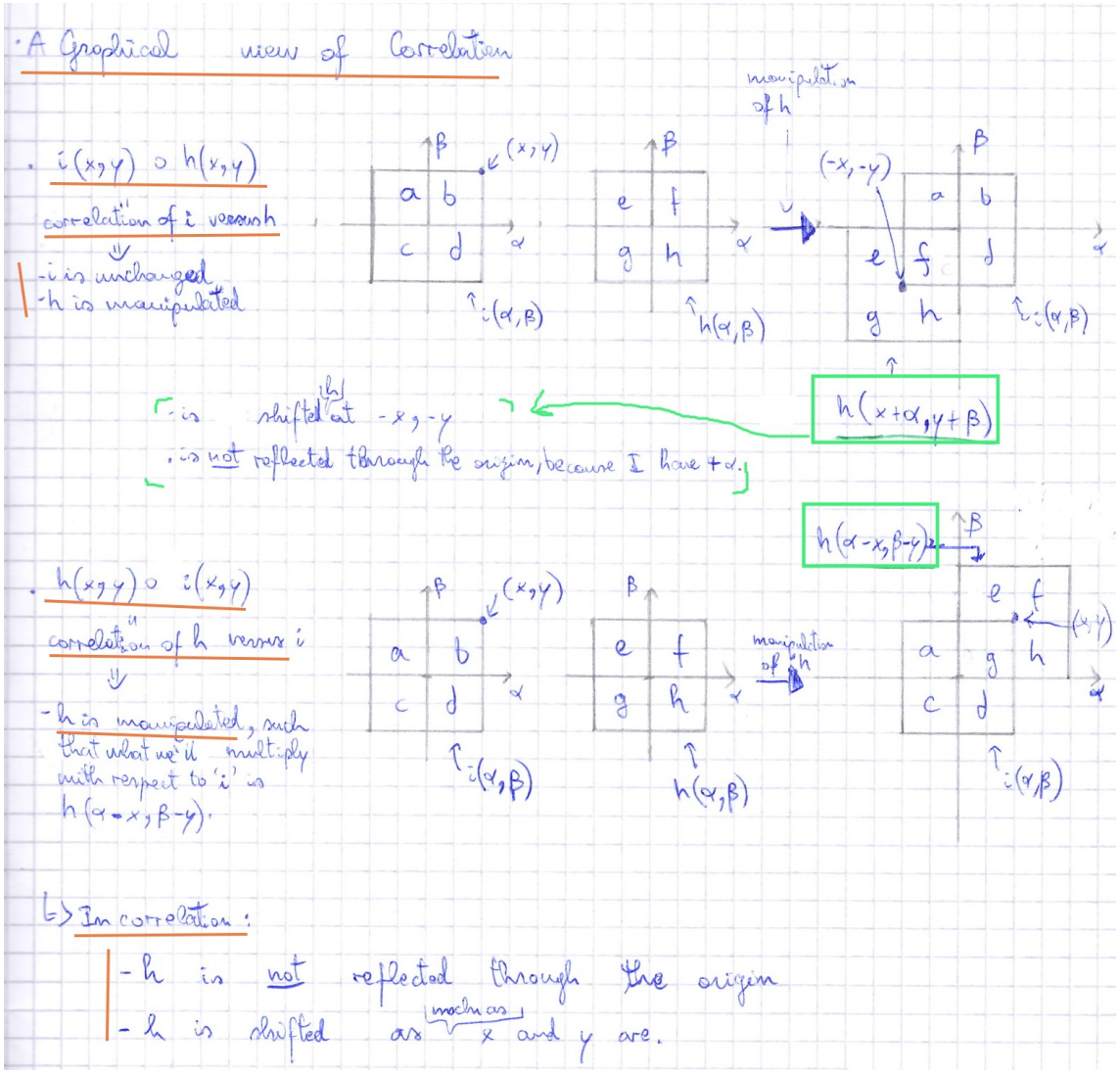


Figure 4.20

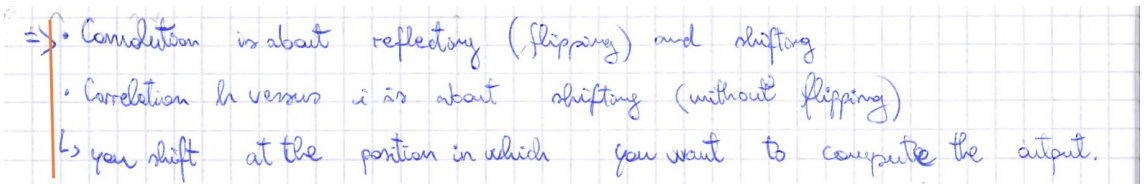


Figure 4.21

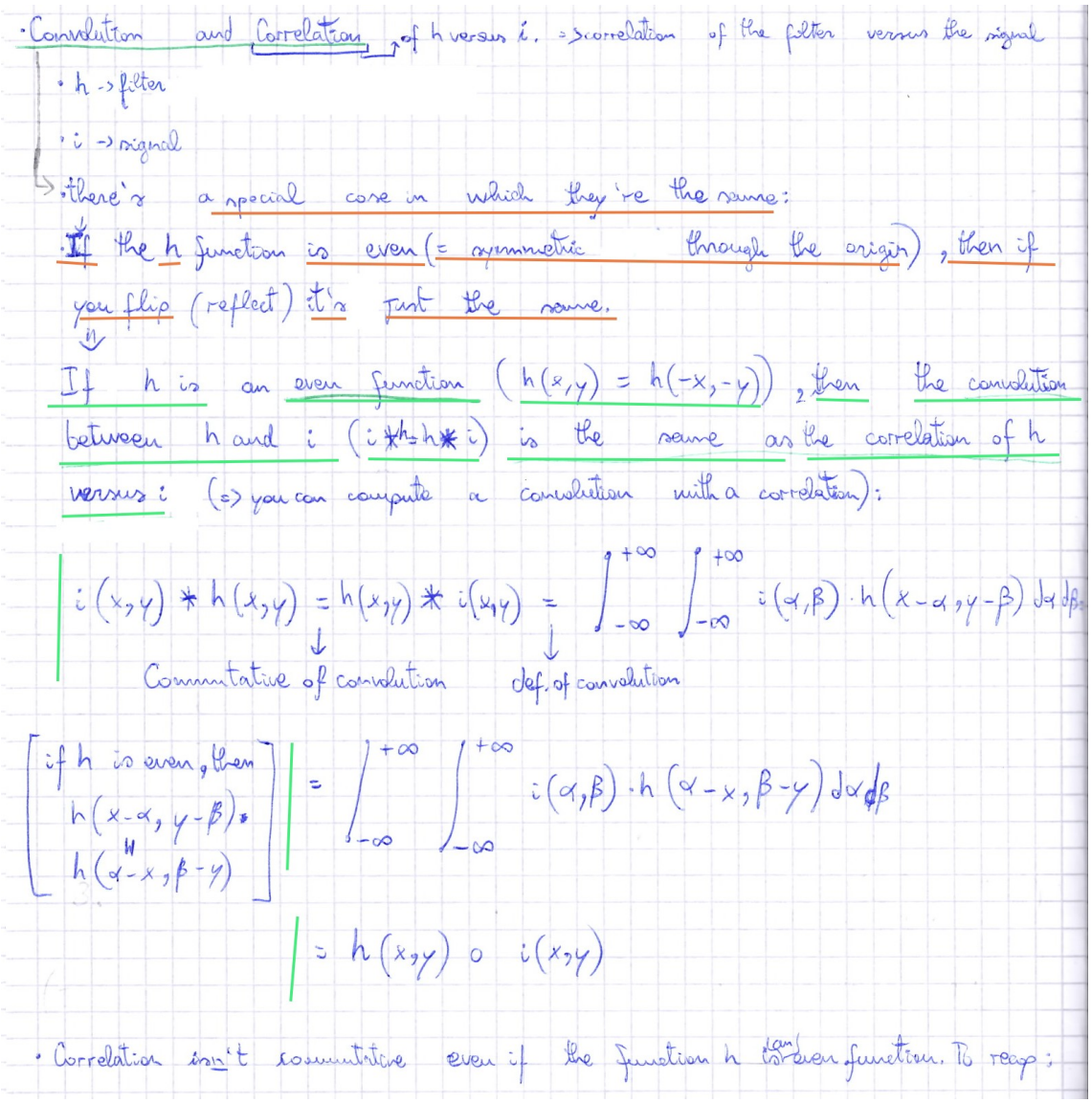


Figure 4.22

4.3 Bilateral Filter

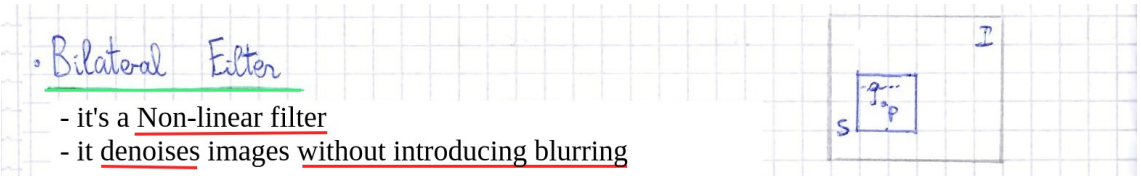
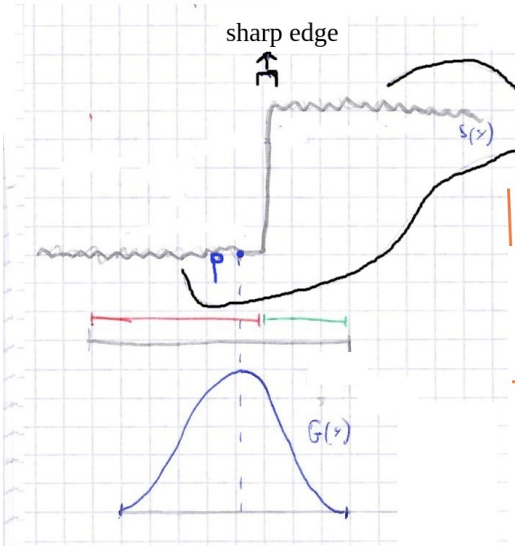


Figure 4.23

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

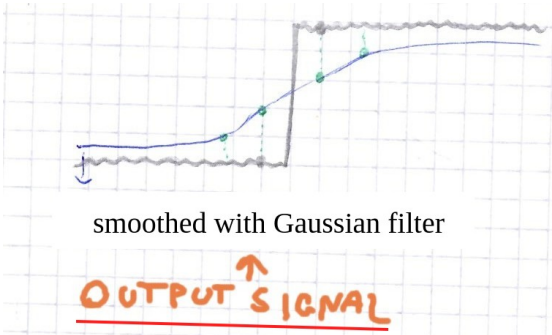


- s(x) -> 1D input signal
- p = pixel on the dark side of the edge

- we want to filter p
- But if we filter a point p in proximity of an edge with the Gaussian kernel, then we weight equally:
  - pixels on the same side of the edge of the given pixel
  - and pixels on the other side of the edge.

=> we mix pixels of different intensities (coming from dark and bright sides).

Figure 4.24



- the input pixels in the dark side will have a higher intensity in the output image

- while the input pixels in the bright side will have a lower intensity in the output image

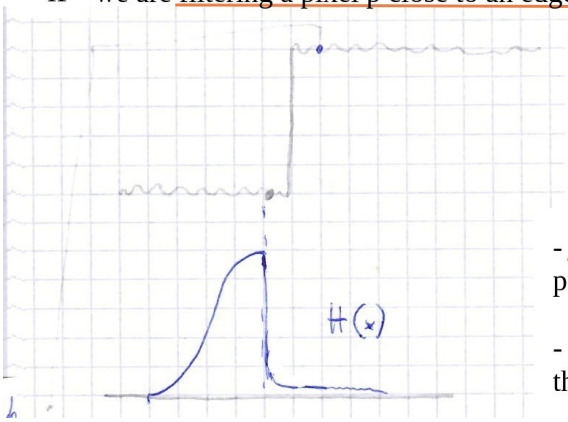
- the edge that was sharp is now smoothed => we see BLUR
- with Gaussian filter (and with linear filters) we consider equally pixels on both sides of the edge => we have blur

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:



- on the same side of the pixel we weigh proportionally to the Gaussian

- on the other side of the edge we just ignore the pixels

Figure 4.26

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

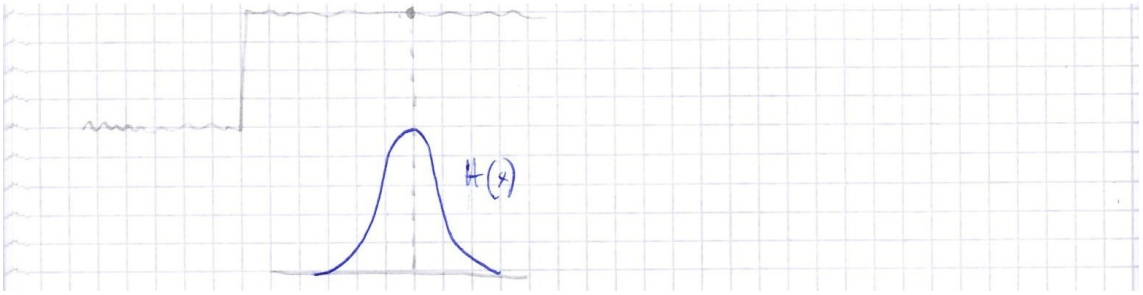
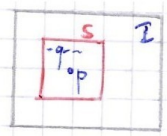


Figure 4.27

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 _> we want to compute the 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

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

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

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

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

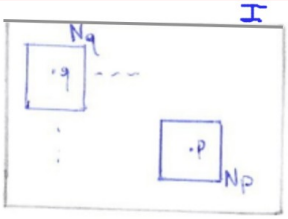
Non-local Means Filter

- 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
- Np = neighborhood of p
- Nq = " " " q

- we will weight q (when smoothing p) according to how similar Np and Nq 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

$$Z(p) = \sum_{q \in I} e^{\frac{\|N_p - N_q\|_2^2}{h^2}}$$

- 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

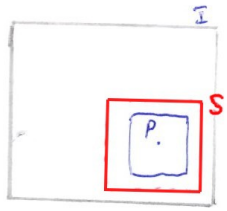
Figure 4.32

Computational complexity of this filter -> O(n^2),
- because for each pixel we take a weighted sum of all the other pixels in the image

=> 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 windoiw S (smaller wrt the entire image)



N_p, N_q -> typically 7x7

S -> 21x21

we still go far away from the local pixel p
=> we still don't consider just a local neighborhood
But we do NOT consider the whole image.

- > size of S => 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:

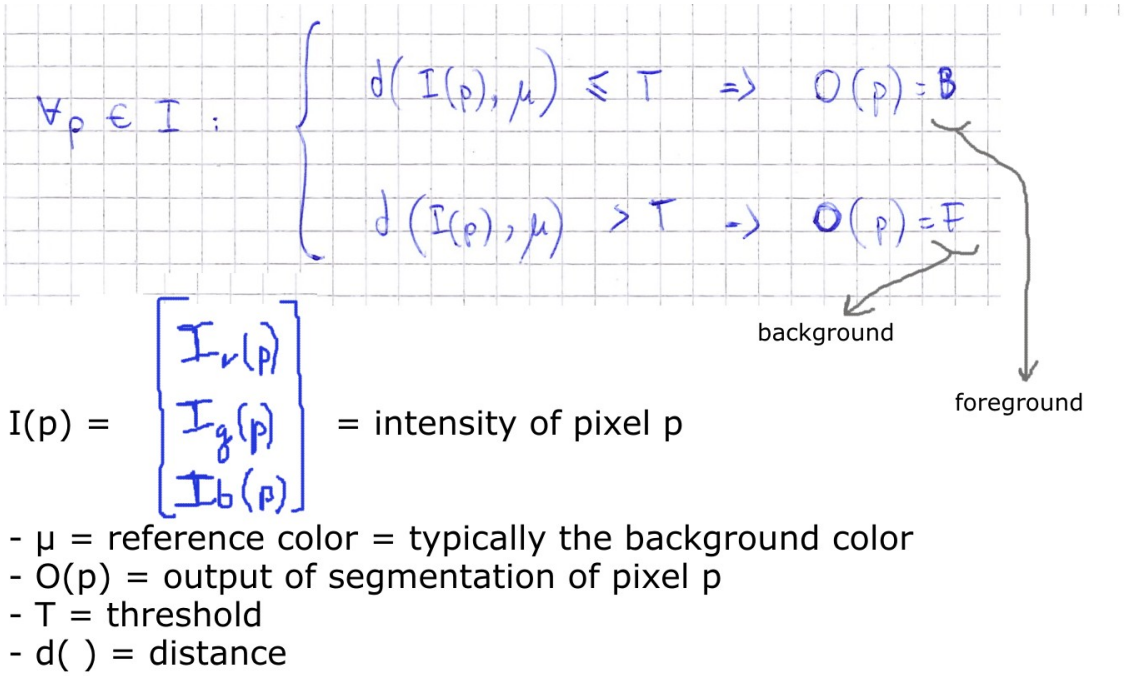


Figure 5.1

- 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:

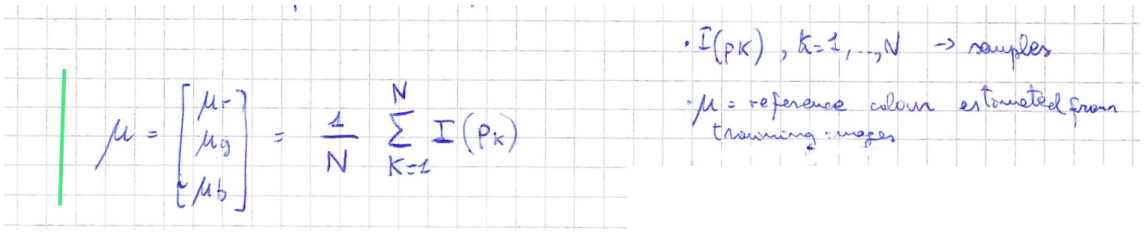


Figure 5.2

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

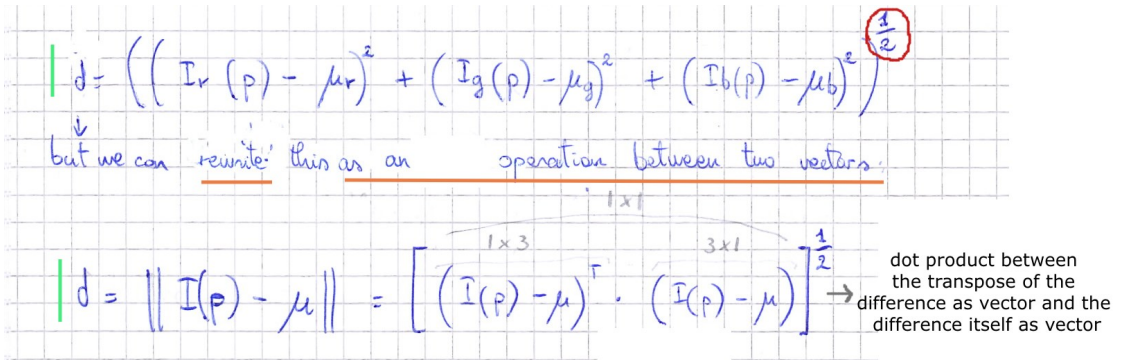
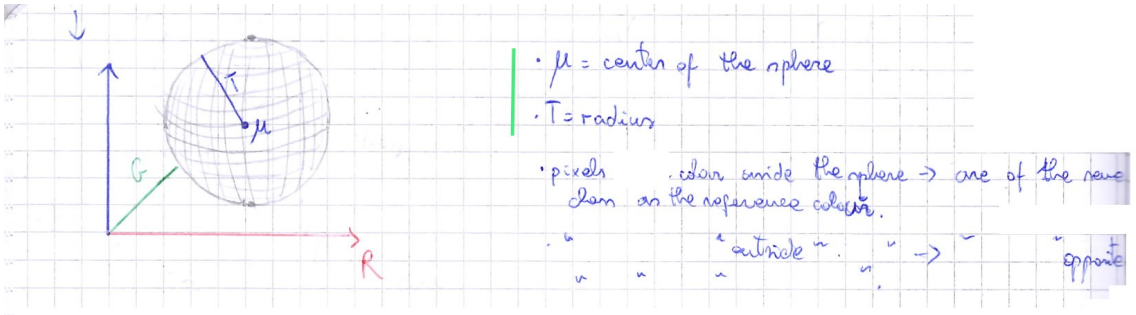


Figure 5.3

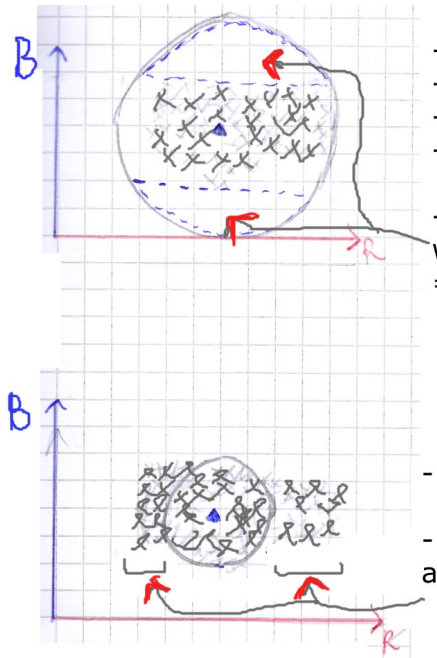
- With Euclidean distance we define an isotropic decision surface -> a SPHERE:



at test time (if μ (= reference color) is BACKGROUND):
- if Euclidean_dist(I(p), μ) <= T -> p classified as BACKGROUND
- if Euclidean_dist(I(p), μ) > T -> p classified as FOREGROUND

Figure 5.4

- Problem with Euclidean distance:
  - If the samples are NOT distributed isotropically => If I have more variation in 1 axis than the other:



- more variation along Red axis
- ▲ = mean = reference colour
- x = training samples
- radius = threshold

- in the top and bottom areas we'll classify wrongly foreground pixels as background pixels => False Negatives

- smaller radius => smaller threshold

- in this case I classify wrongly background pixels as foreground pixels => False Positives

=> 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:

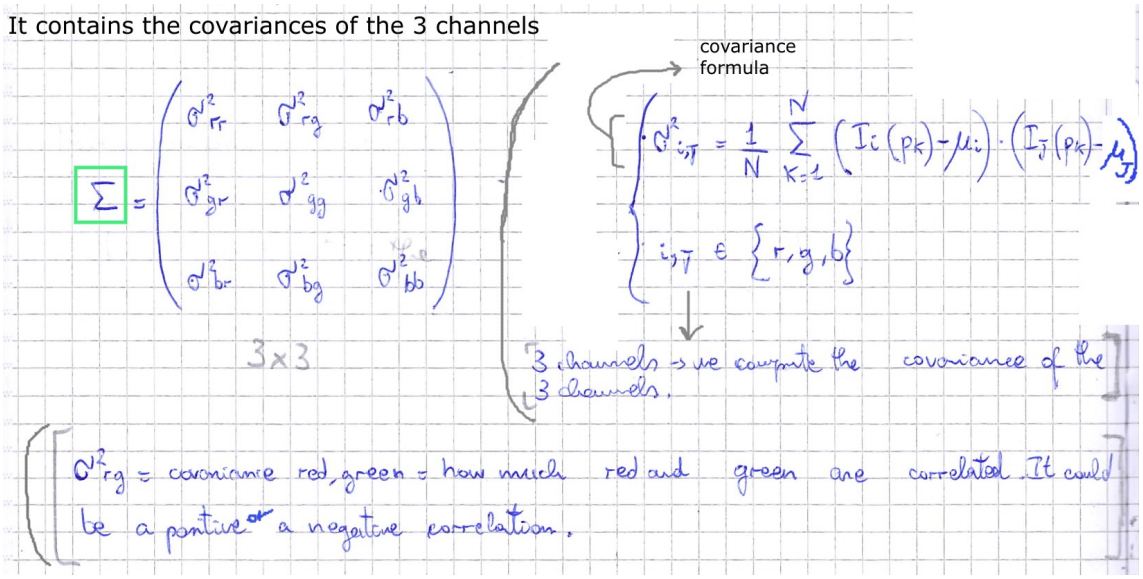


Figure 5.6

- => the MAHALANOBIS Distance is:

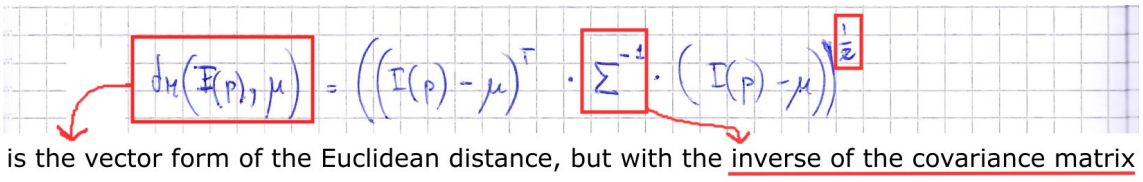
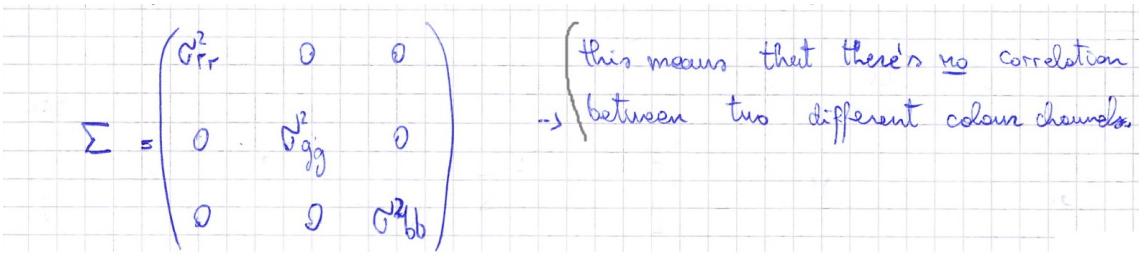


Figure 5.7

- To understand the difference between Eucl dist and Mahal dist:
  - assume the covariance matrix is diagonal (we won't lose generality):



Then, let's take its inverse:

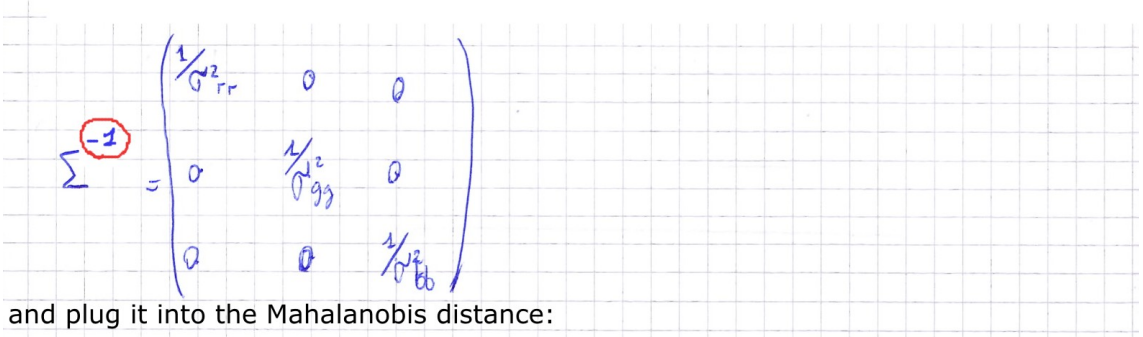


Figure 5.8

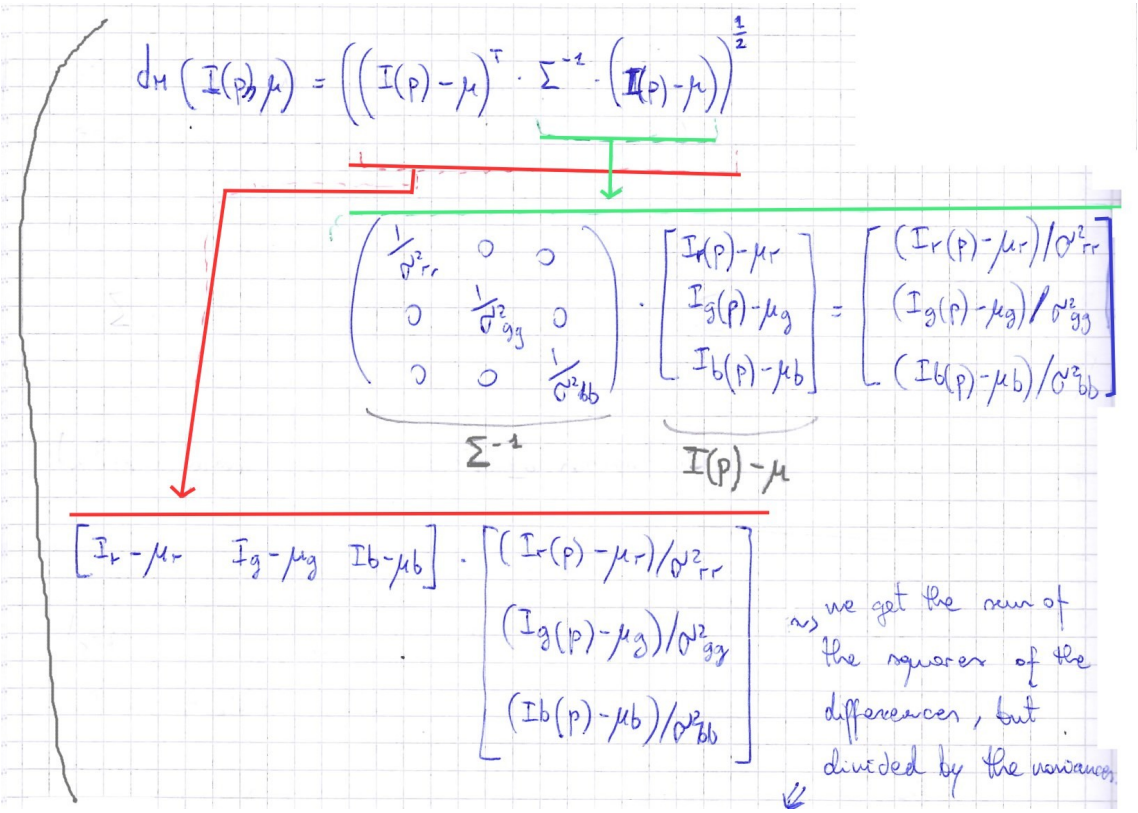


Figure 5.9

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

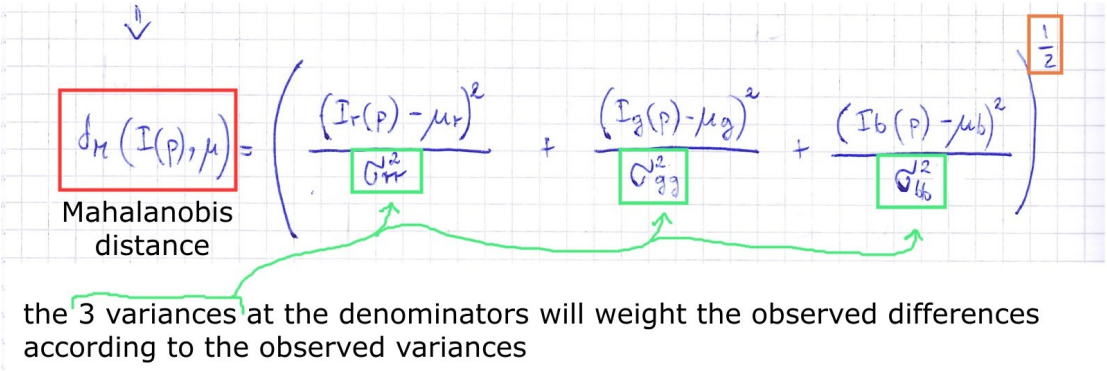


Figure 5.10

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

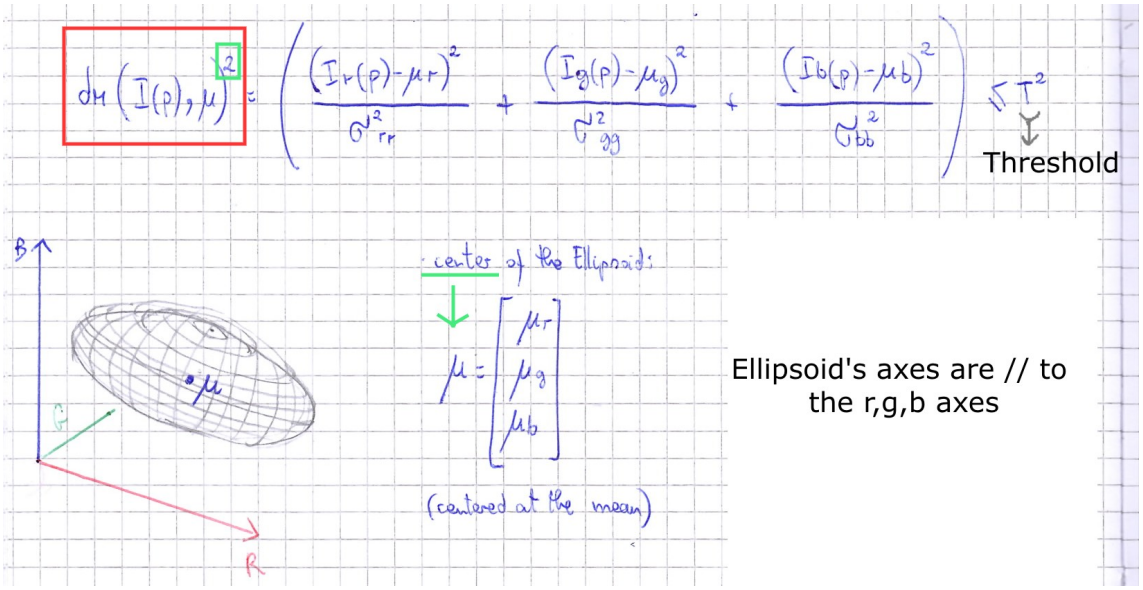


Figure 5.11

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

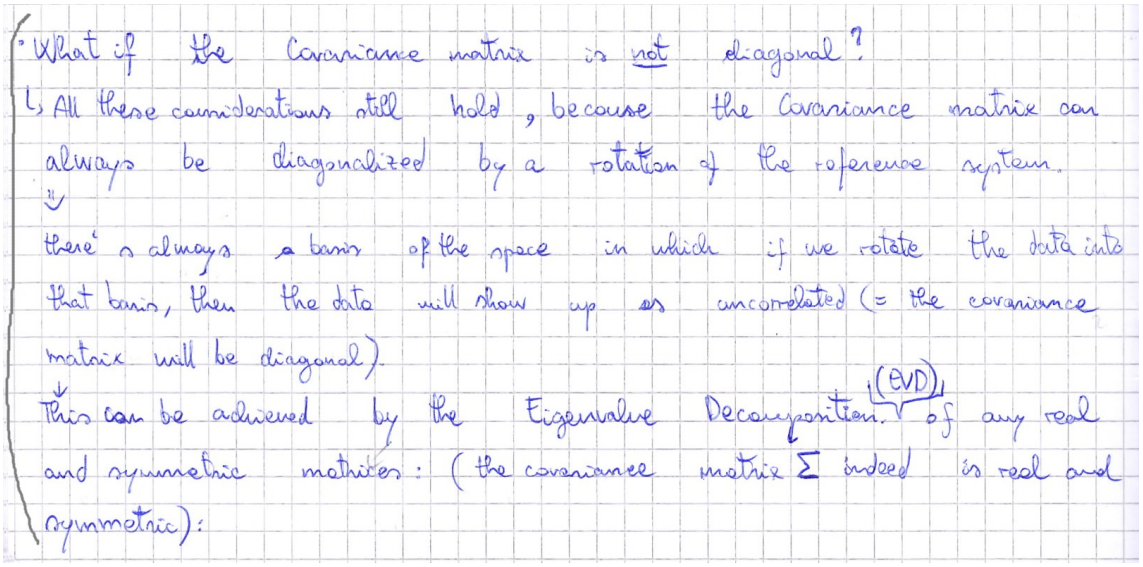


Figure 5.12

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

↳ 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 } elements along the diagonal = eigenvalues of the matrix Σ.

  ↳ 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

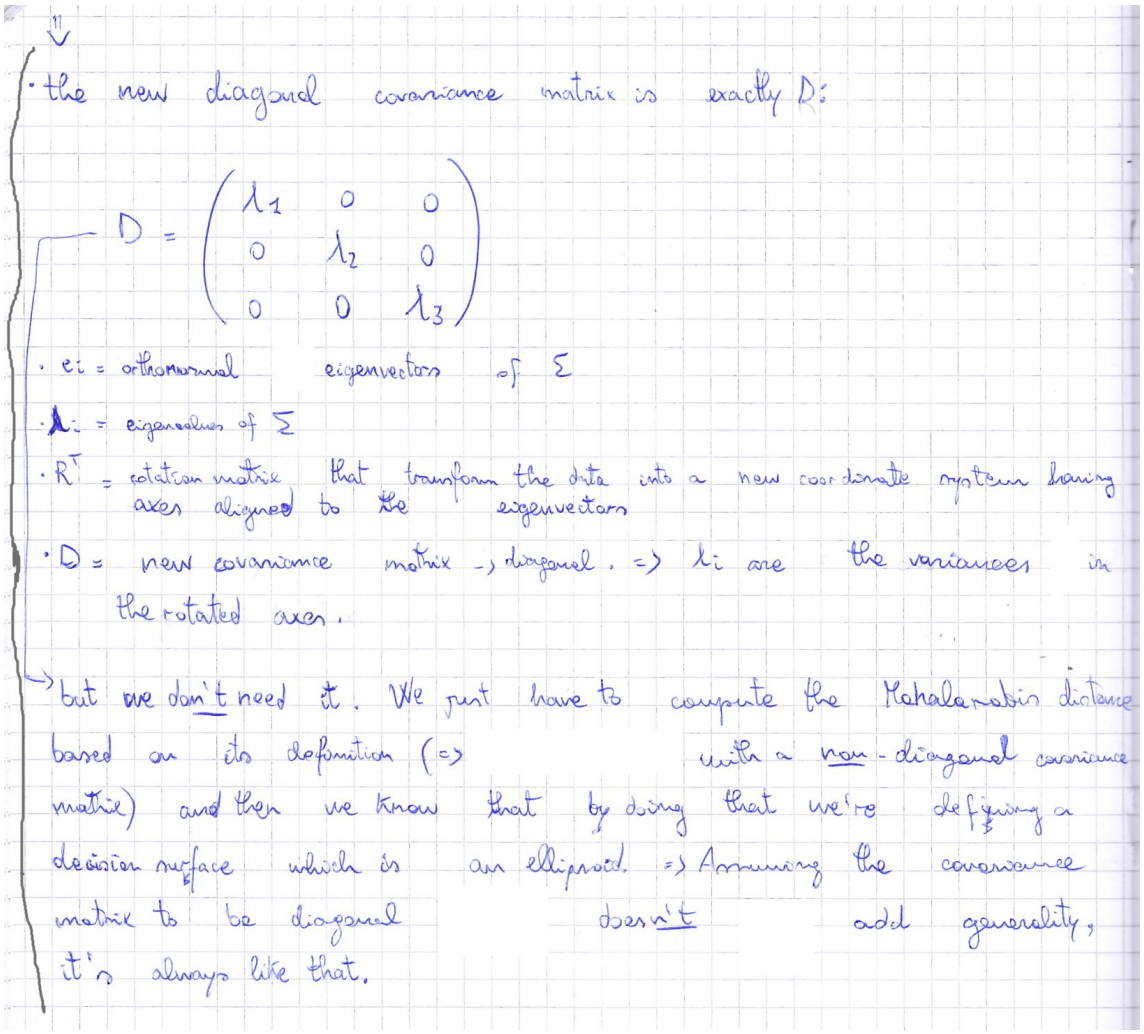

11
• the new diagonal covariance matrix is exactly D:
$D = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix}$
• e_i = orthonormal eigenvectors of Σ
• λ_i = eigenvalues of Σ
• R^T = rotation matrix that transform the data into a new coordinate system having axes aligned to the eigenvectors
• D = new covariance matrix → diagonal. ⇒ λ_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 (⇒ with a non-diagonal covariance matrix) and then we know that by doing that we're defining a decision surface which is an ellipsoid. ⇒ 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 pipelin is the following:

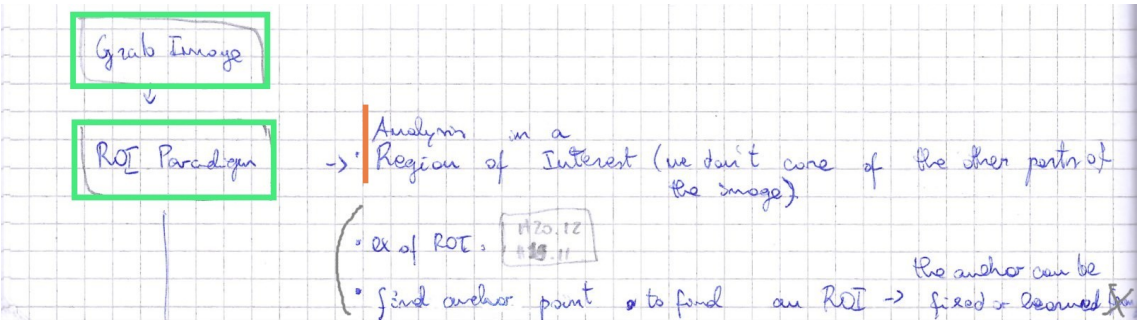


Figure 7.1

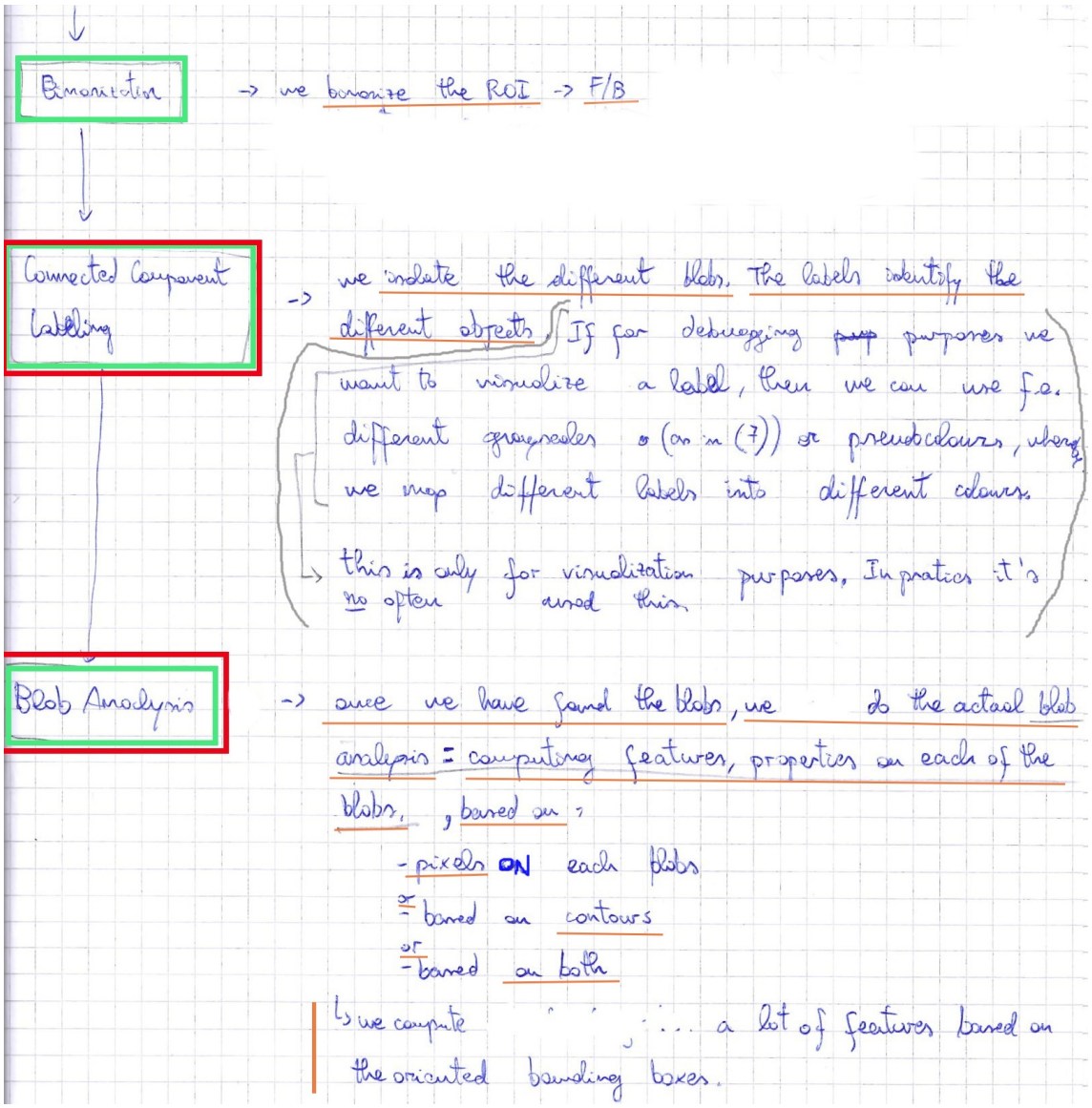

Binarization
-> we binarize the ROI -> F/B
Connected Component Labelling
-> we isolate the different blobs. The labels identify the different objects
If for debugging purposes we want to visualize a label, then we can use f.e. different grayscales (as in (7)) or pseudocolours, where we map different labels into different colours.
this is only for visualization purposes, In pratice it's no often used this.
Blob Analysis
-> once we have found the blobs, we do the actual blob analysis = computing features, properties on each of the blobs, based on:
- pixels ON each blobs
or = based on contours
or = based on both
Is we compute ... a lot of features based on the oriented bounding boxes.

Figure 7.2

7.1 Connected Components

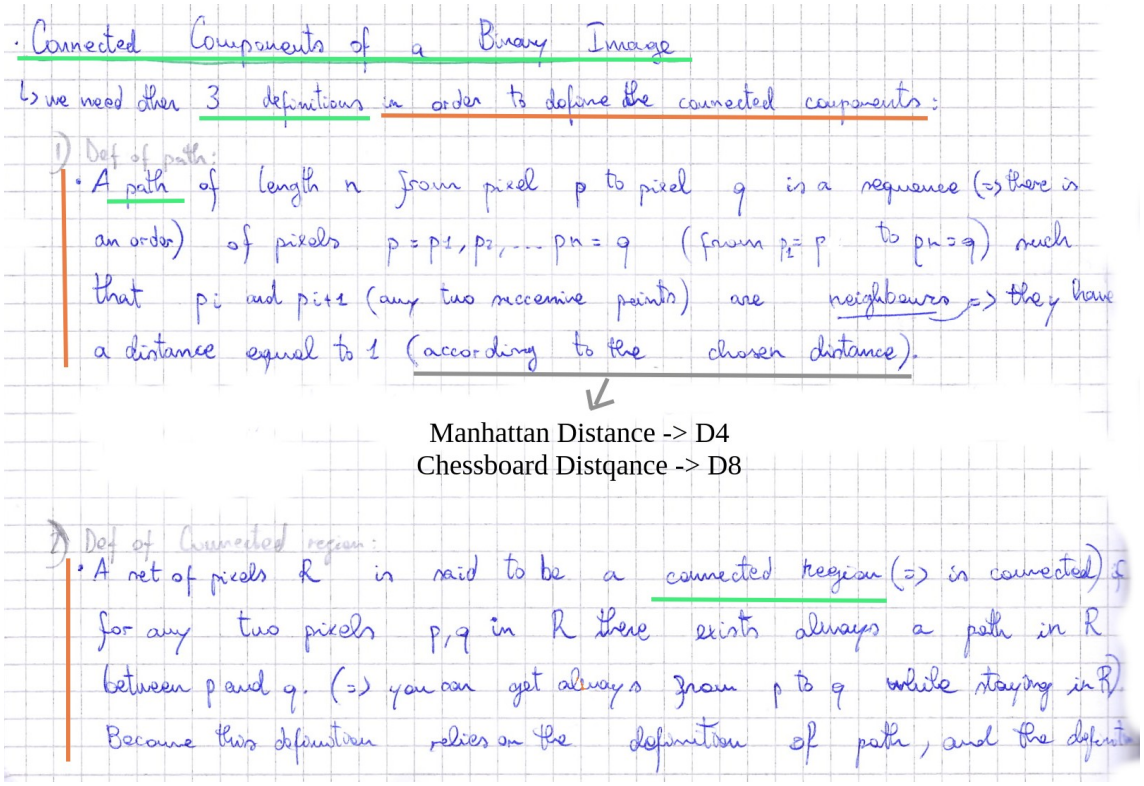


Figure 7.3

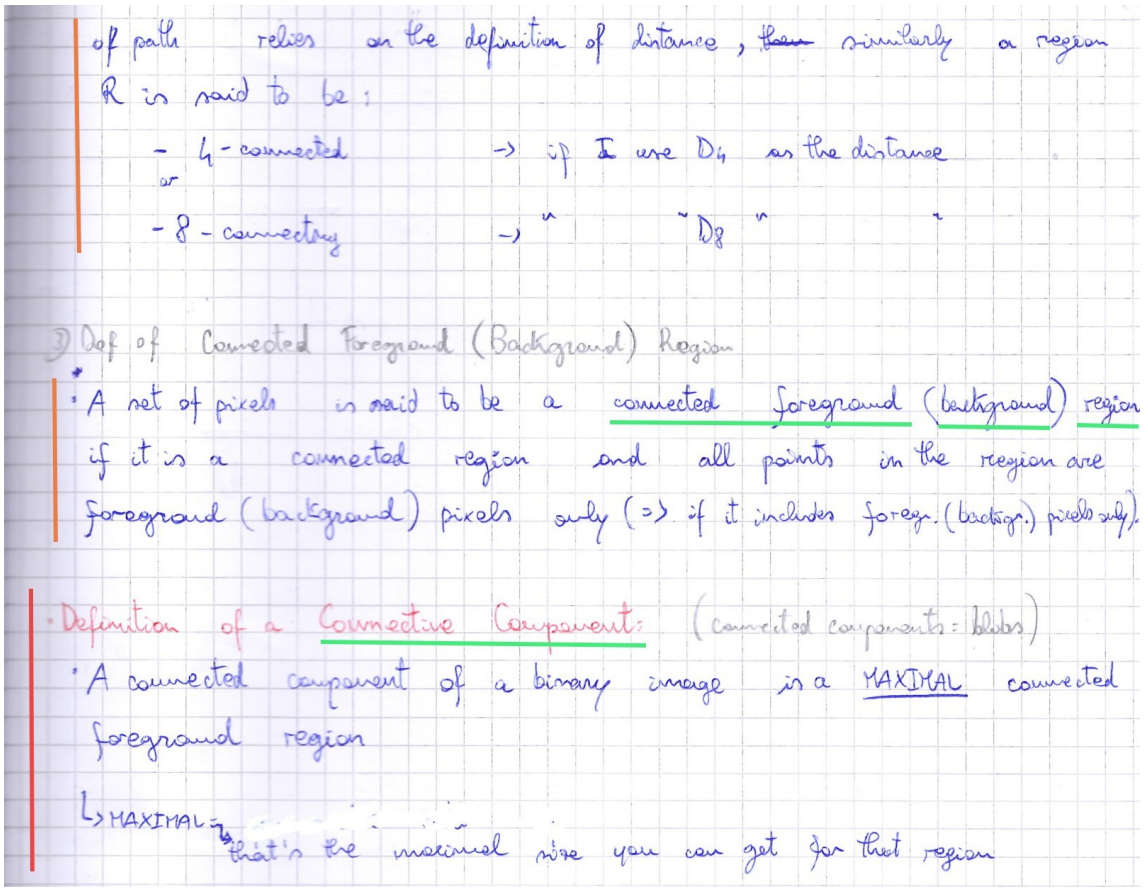


Figure 7.4

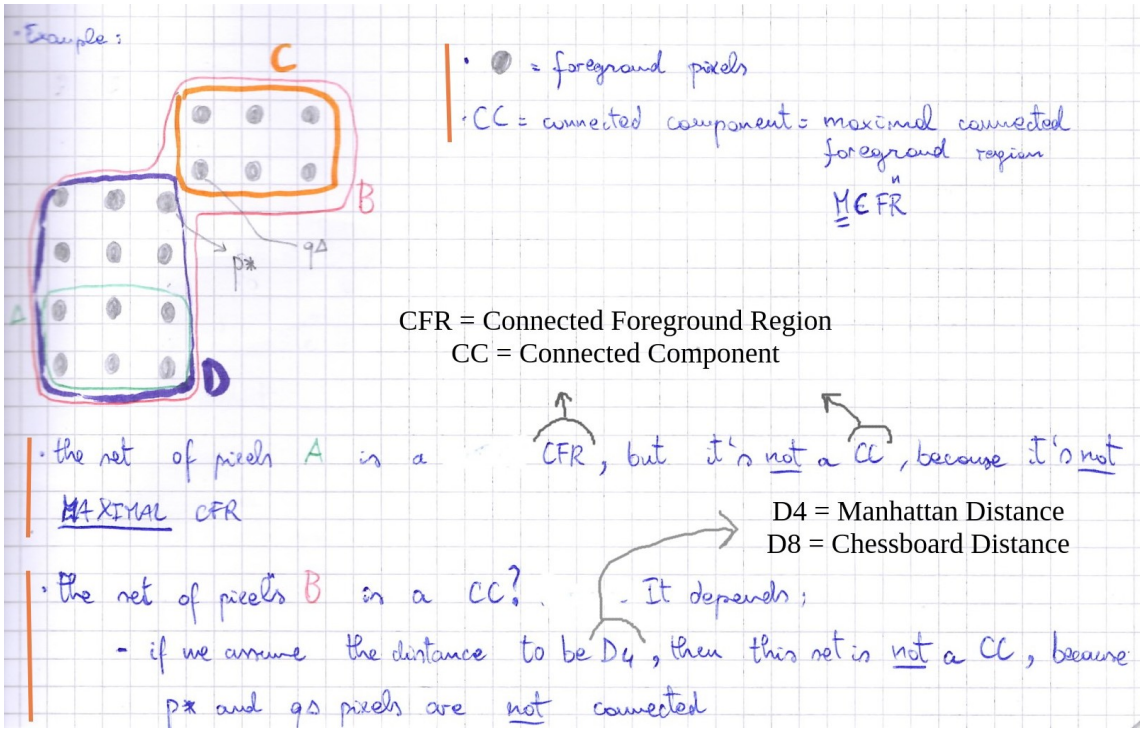


Figure 7.5

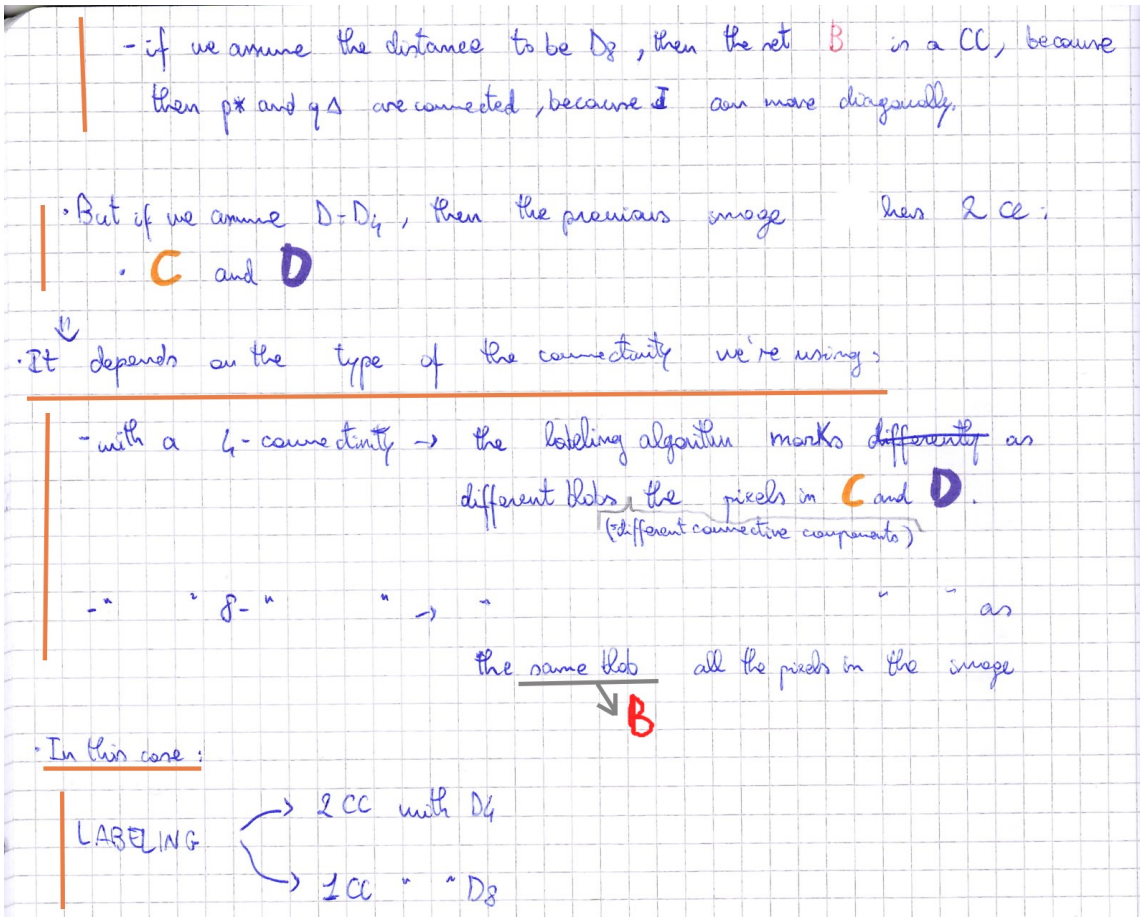


Figure 7.6

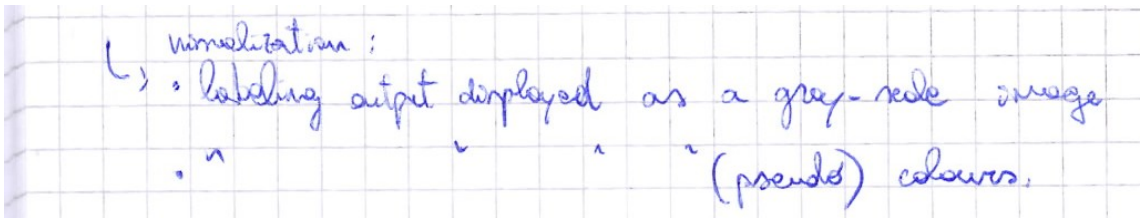
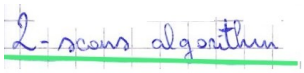


Figure 7.7

7.2 Labeling problem (two 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; lef-right)

Figure 7.8

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

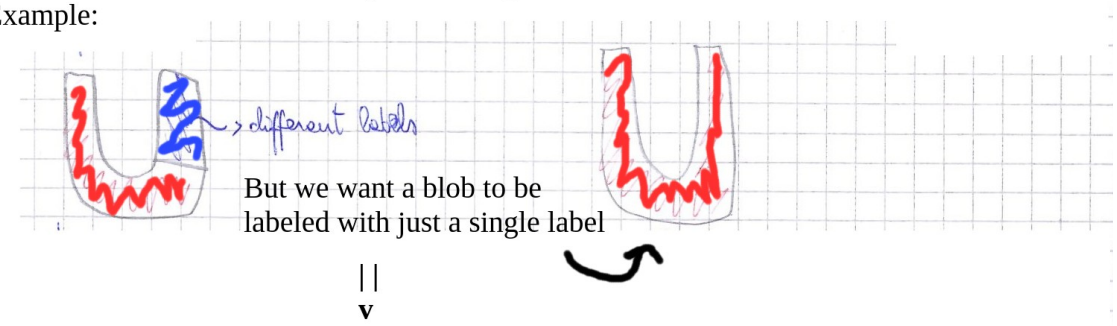


Figure 7.9

- 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

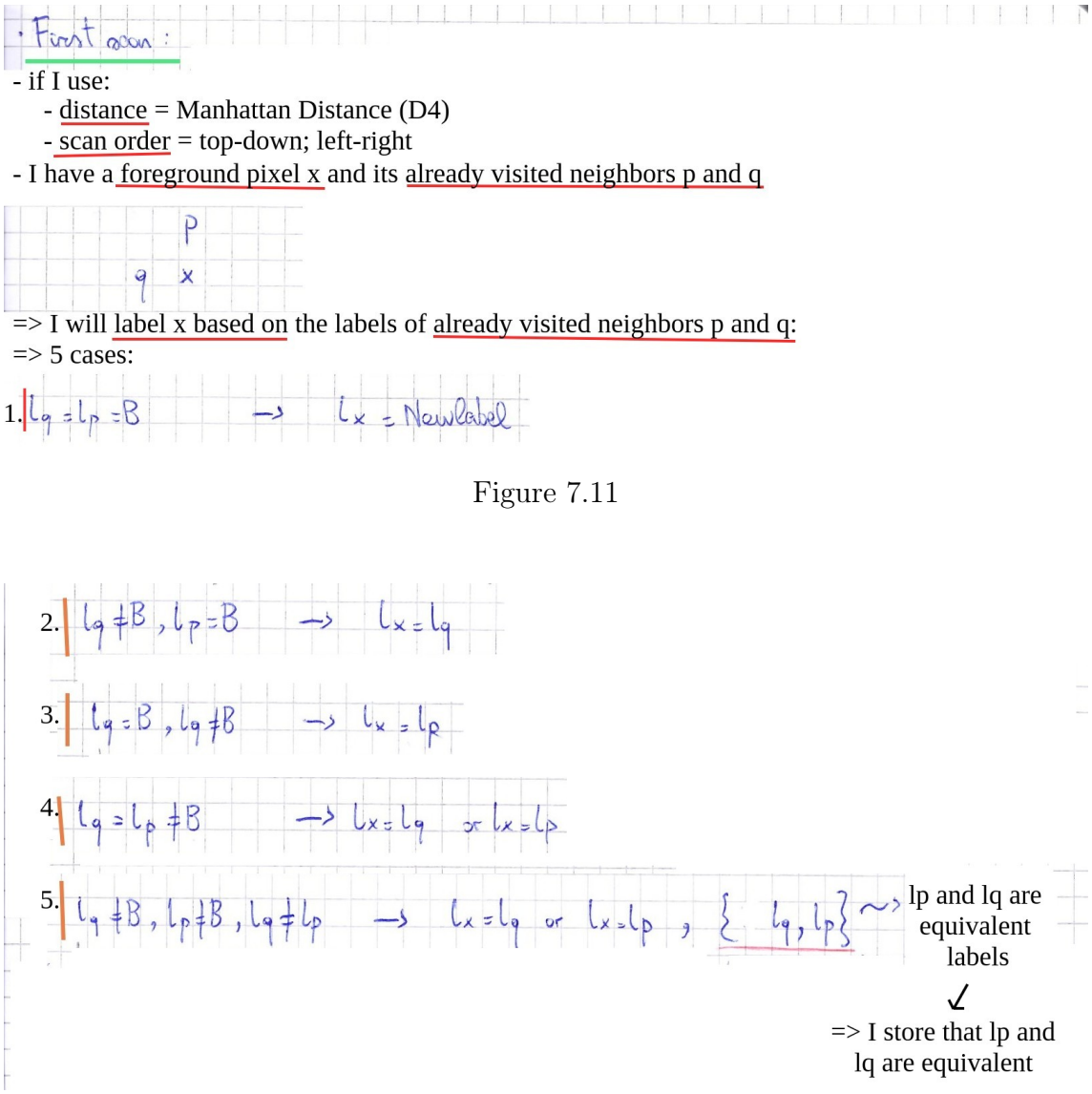


Figure 7.11

Figure 7.12

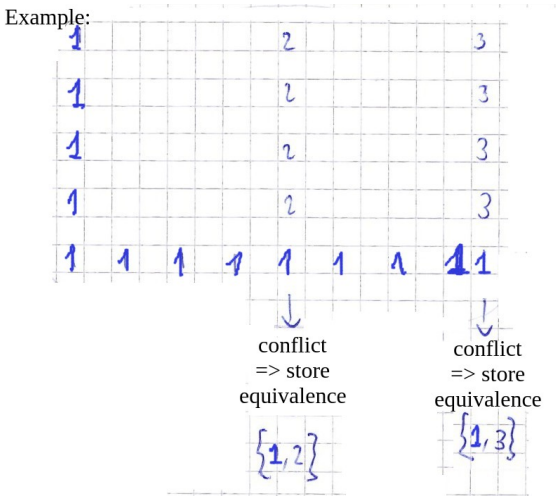
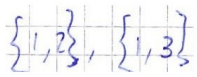
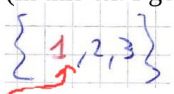


Figure 7.13

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



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



Then, in the 2nd scan:
- we pick a label from each equivalent class and relabel all blobs
=> we substitute temporary labels with equivalent labels

=> Final result of the example:

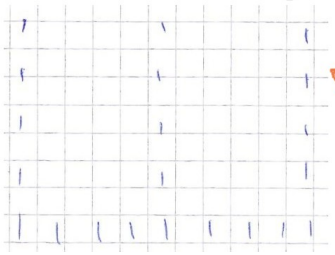


Figure 7.14

7.2.1 Implementation

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

if i,j are
equivalent labels

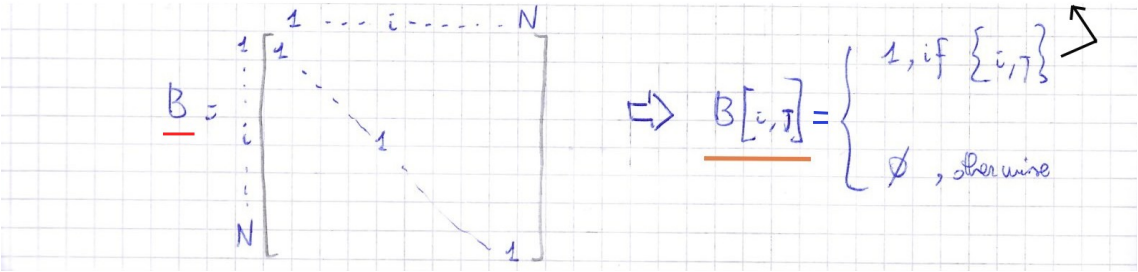


Figure 7.15

The B matrix is symmetric (because if i is equivalent to j => j is equivalent to i):

, N = 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 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

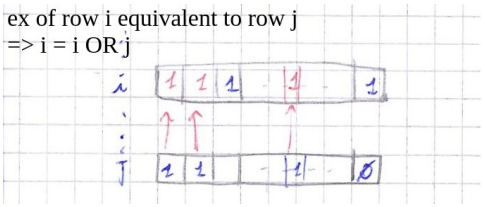


Figure 7.18

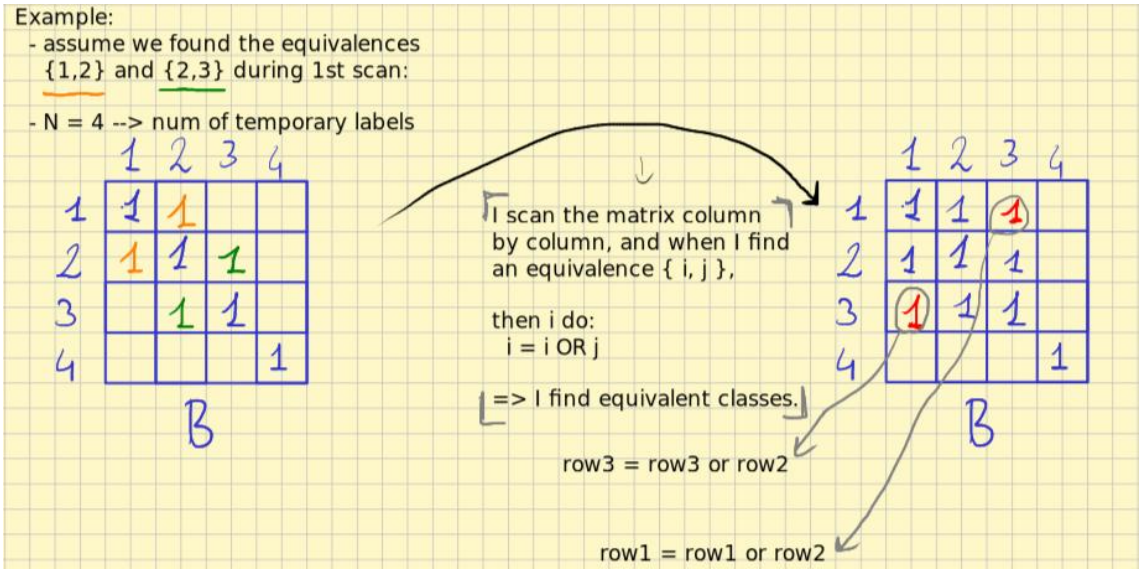


Figure 7.19

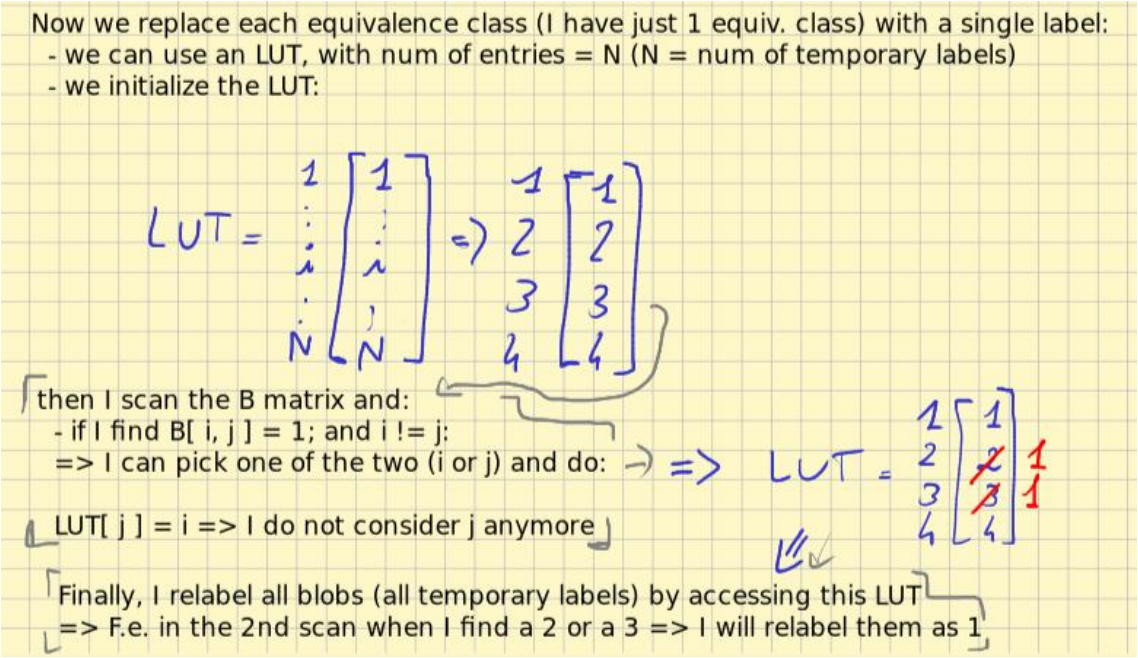

Now we replace each equivalence class (I have just 1 equiv. class) with a single label:
- we can use an LUT, with num of entries = N (N = num of temporary labels)
- we initialize the LUT:
LUT =
1
i
N
=>
then I scan the B matrix and:
- if I find B[i, j] = 1; and i != j:
=> I can pick one of the two (i or j) and do:
=>
LUT =
LUT[j] = i => I do not consider j anymore
Finally, I relabel all blobs (all temporary labels) by accessing this LUT
=> F.e. in the 2nd scan when I find a 2 or a 3 => 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

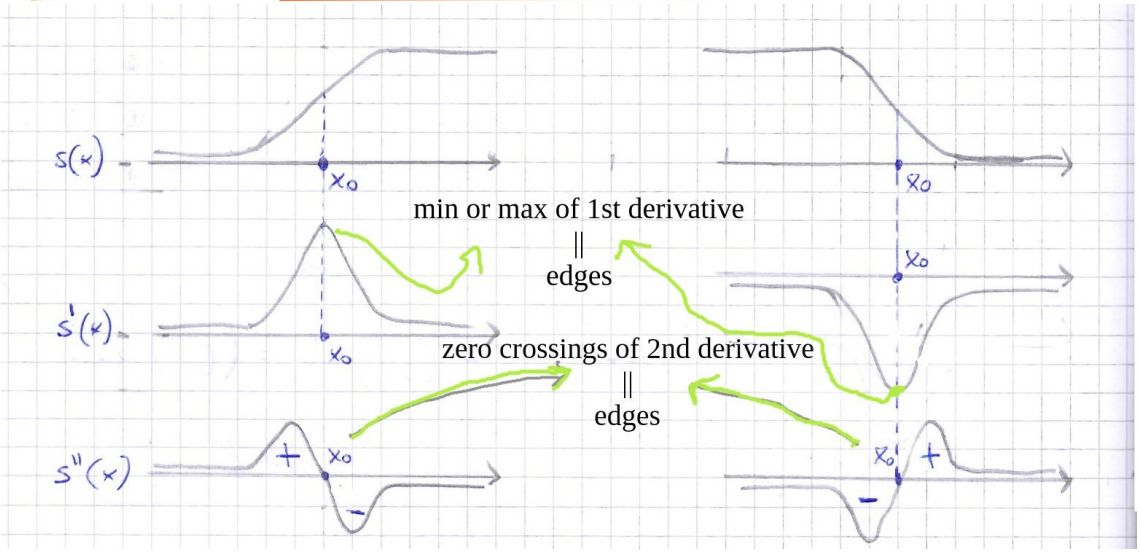


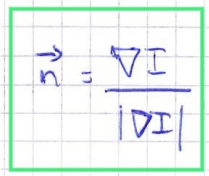
Figure 8.1

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:



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

  - 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:

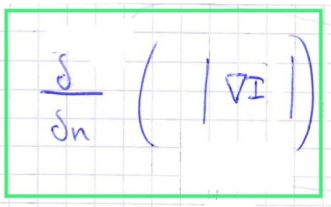
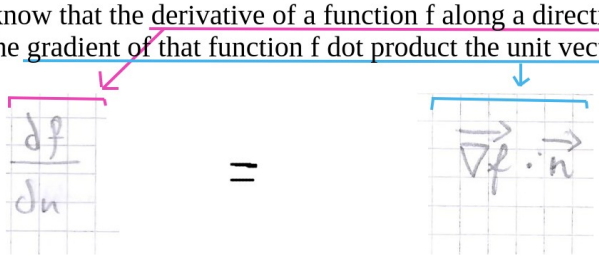


Figure 8.3

- 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:



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

- => 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. dot product
  3. 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 (Ix, Iy)
  - and also 2nd derivatives (Ixx, Iyy)

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

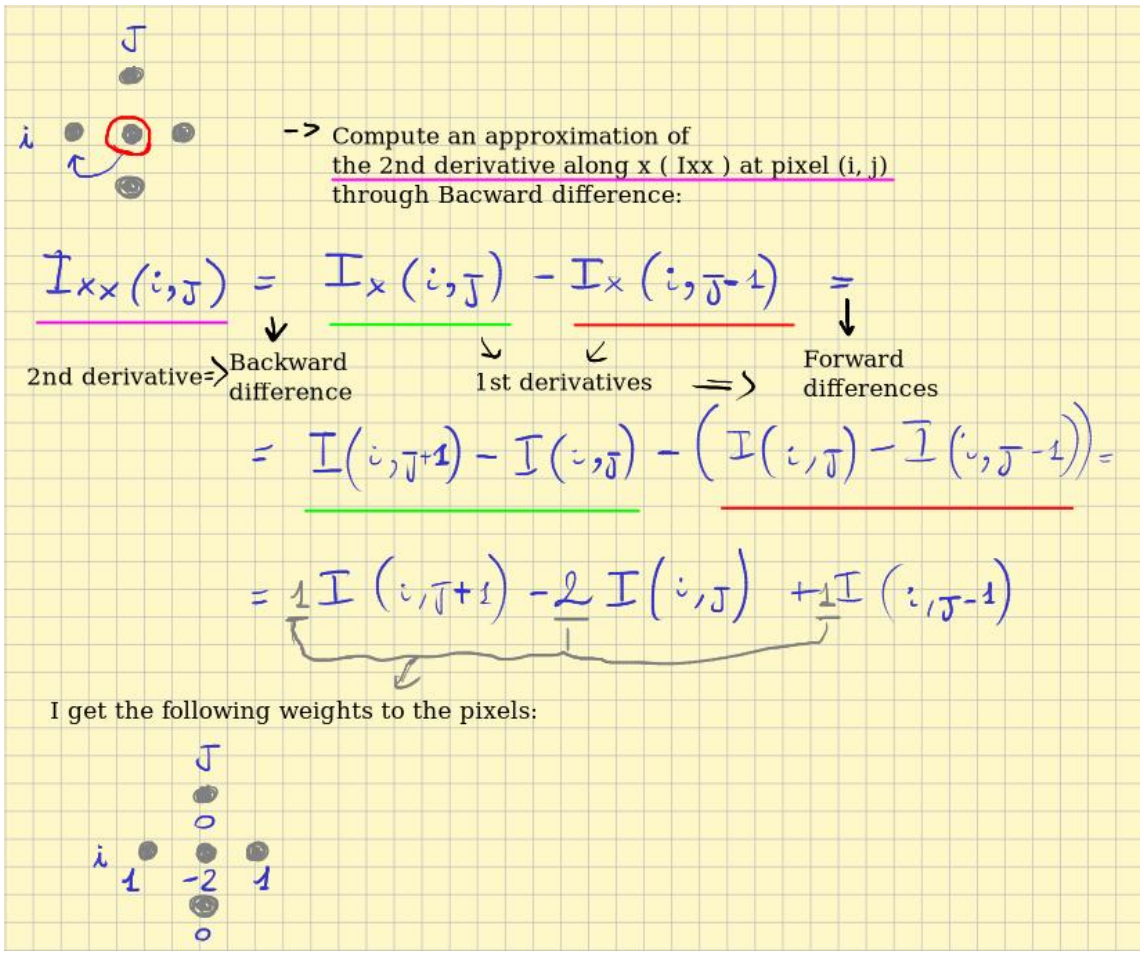


Figure 8.8

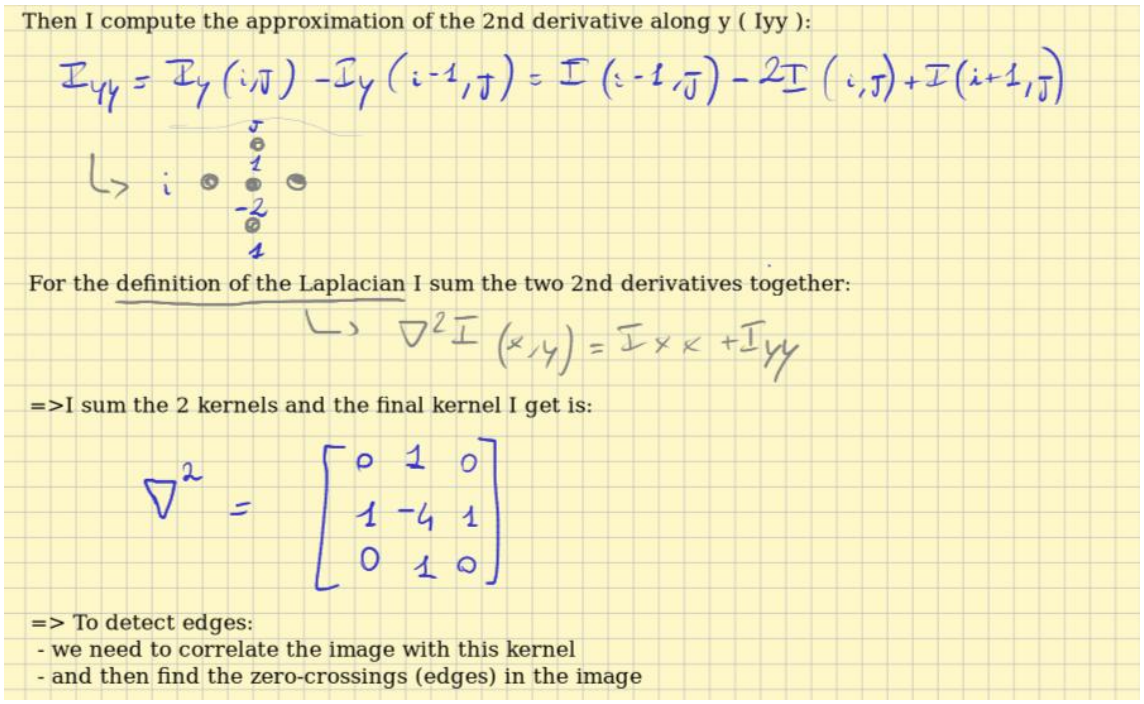


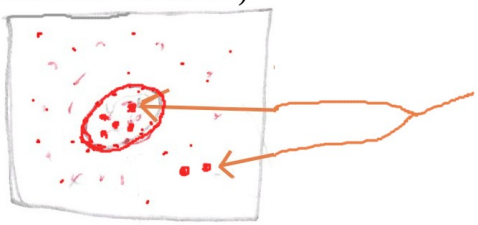
Figure 8.9

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)



spurious edges
in uniform regions (inside and outside the object)
=> we don't 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. =>:

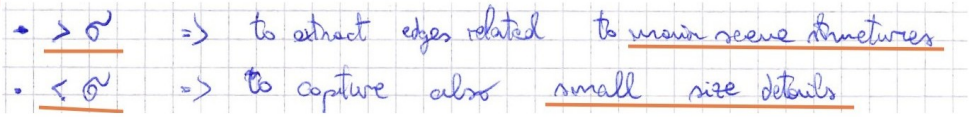
1. First I apply a Gaussian Filter to the image
2. Then apply the Laplacian
3. Extraction of zero-crossings (edges)

\\

LOG (Laplacian of Gaussian)

Figure 8.10

1. How to perform Gaussian smoothing? Which sigma?



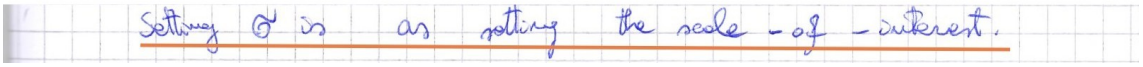


Figure 8.11

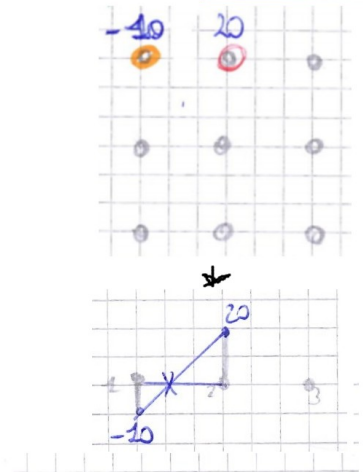
3 How to perform the Zero-Crossing search?

- we have computed the LoG in the image
  => we now look for the zero-crossings
  - (we don't look for LoG = 0, but we look for zero-crossings, because 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 Log = - 10 and next pixel with 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



=> the pixel with LoG = -10 will be an EDGE

Figure 8.13

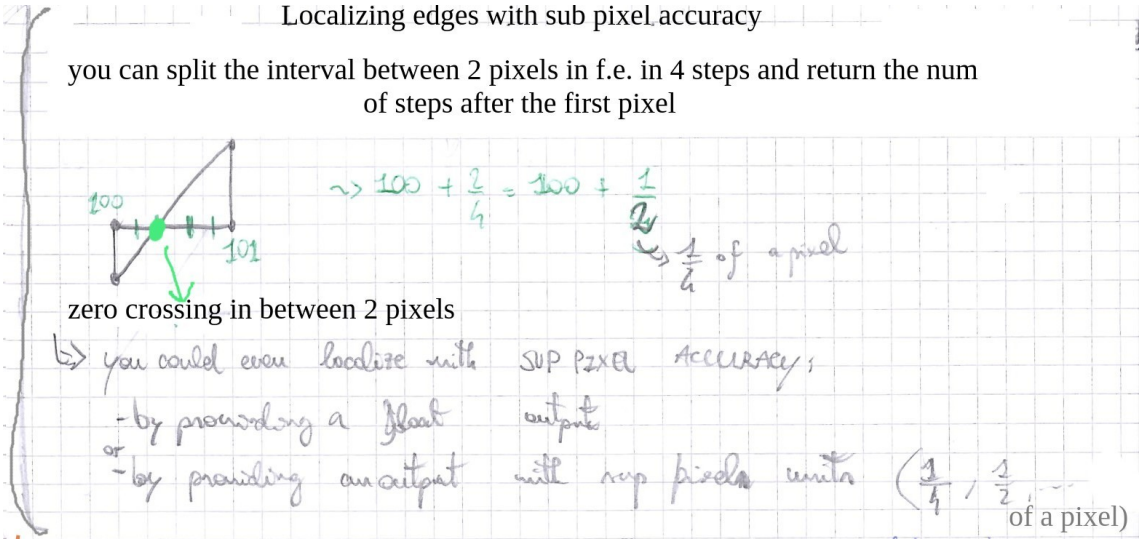
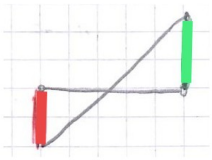


Figure 8.14

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



Slope of Laplacian = | + |

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

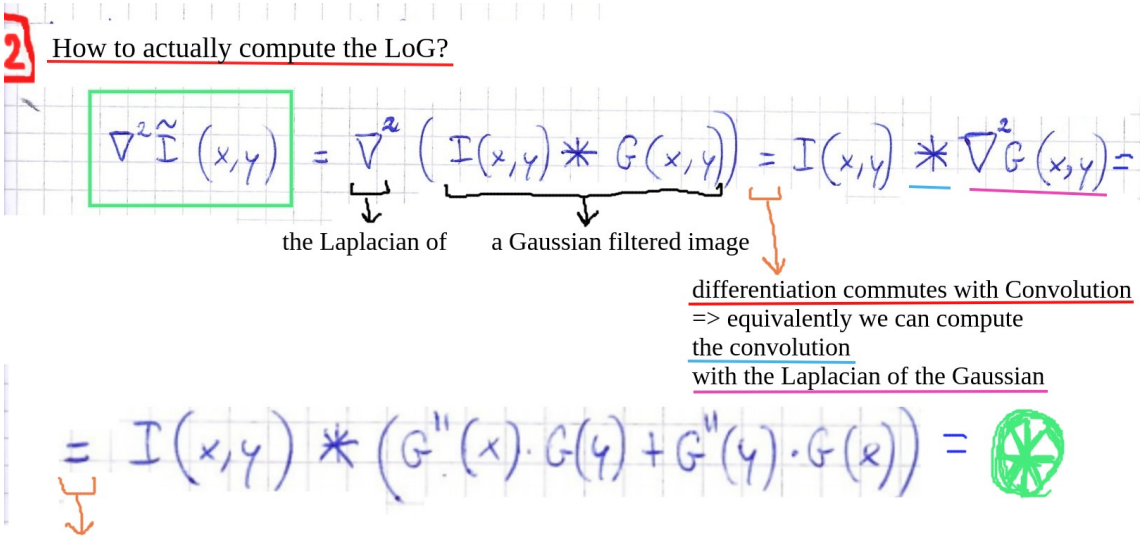
Example of use of this thresholding step:

- 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



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

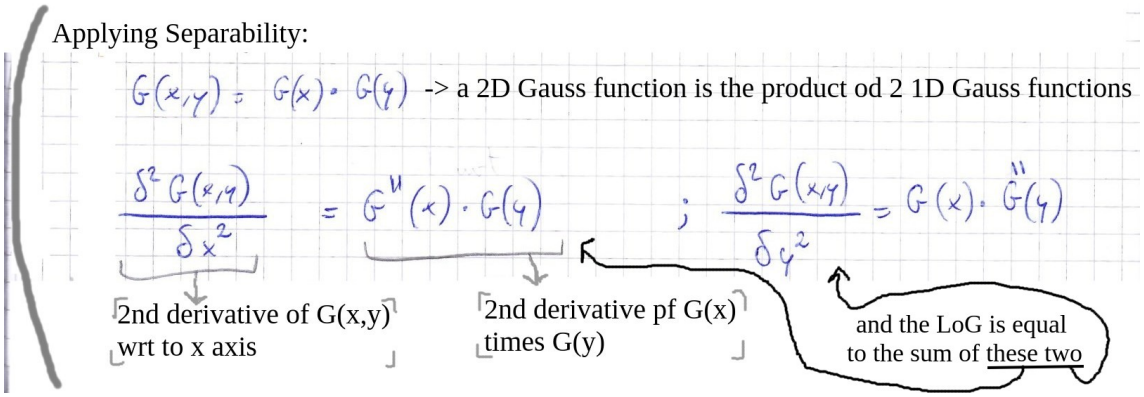


Figure 8.17

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

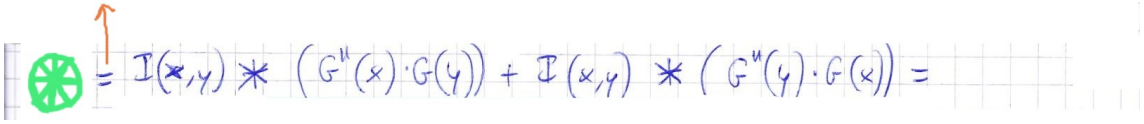
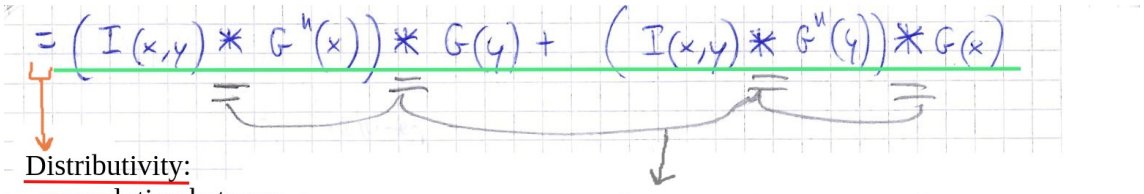


Figure 8.18



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 (=> 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 optimly 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:

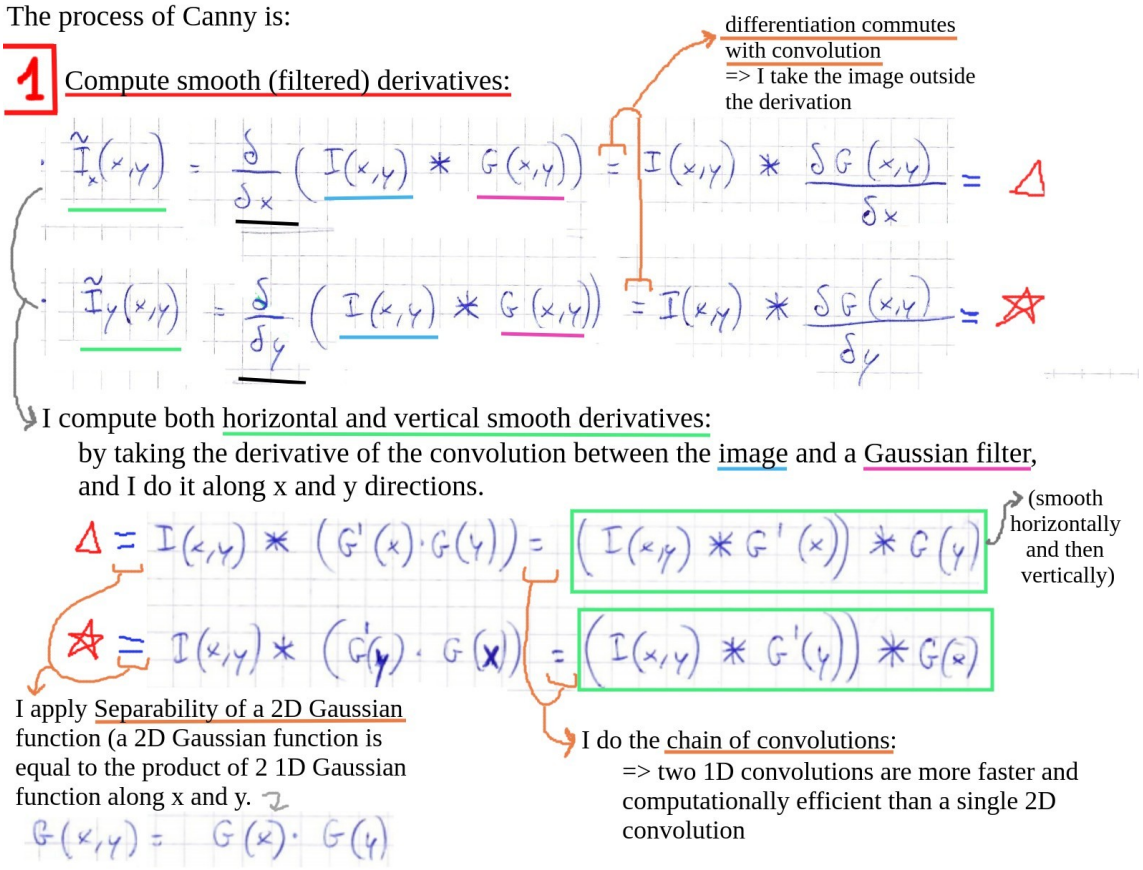


Figure 8.22

=> 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

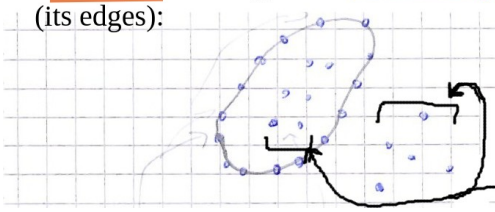
- rather than keeping all the peaks of the gradient magnitude along the gradient direction, we threshold to prune out some of them because they might still due to noise.
=> we threshold the magnitude of the gradient.

- and we apply a particular thresholding called Hysteresis Thresholding.

Figure 8.23

8.2.1 Hysteresis Thresholding

- use 2 thresholds instead than 1
- Example: let's suppose to have an object and that we want to find its contour (its edges):

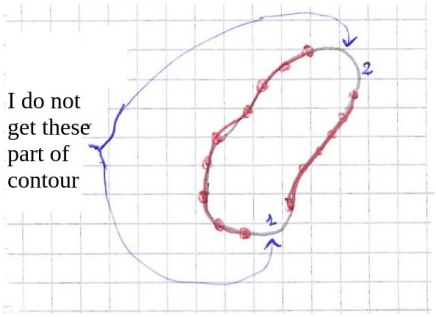


=> I find:
- all the edges belonging to the contour of the object,
- BUT also some spurious edges in uniform areas that I don't want.

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.



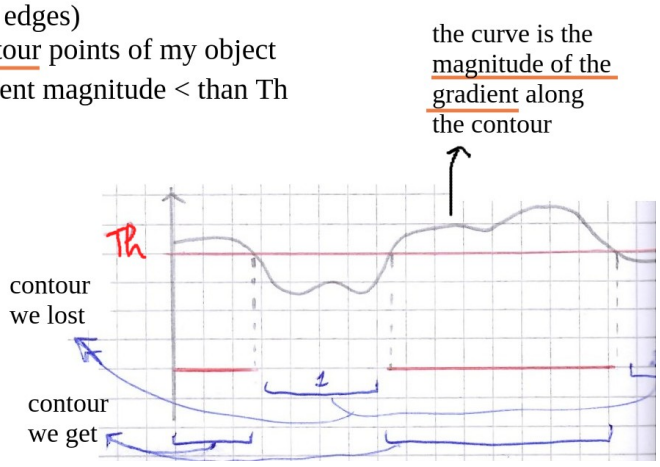


Figure 8.25

=> 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

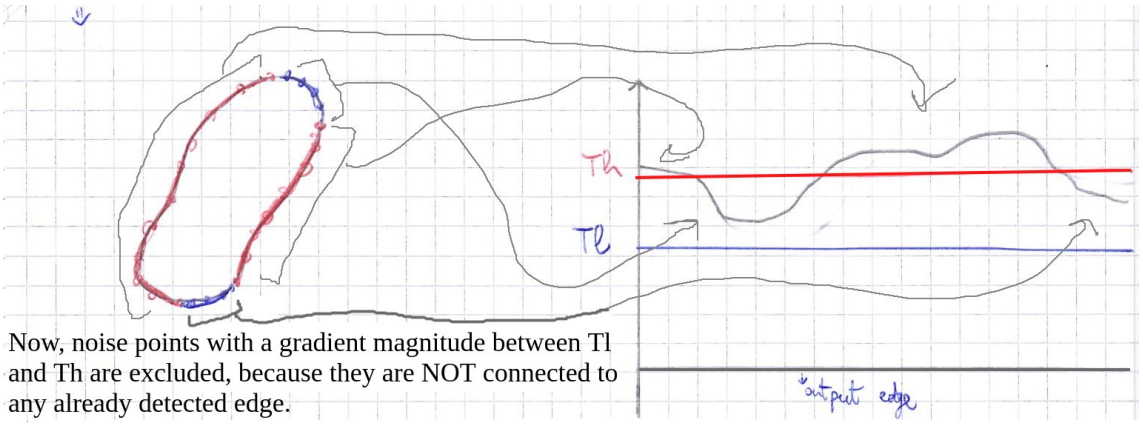
=> a pixel is taken as an EDGE if:
- its gradient magnitude is > than Th

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

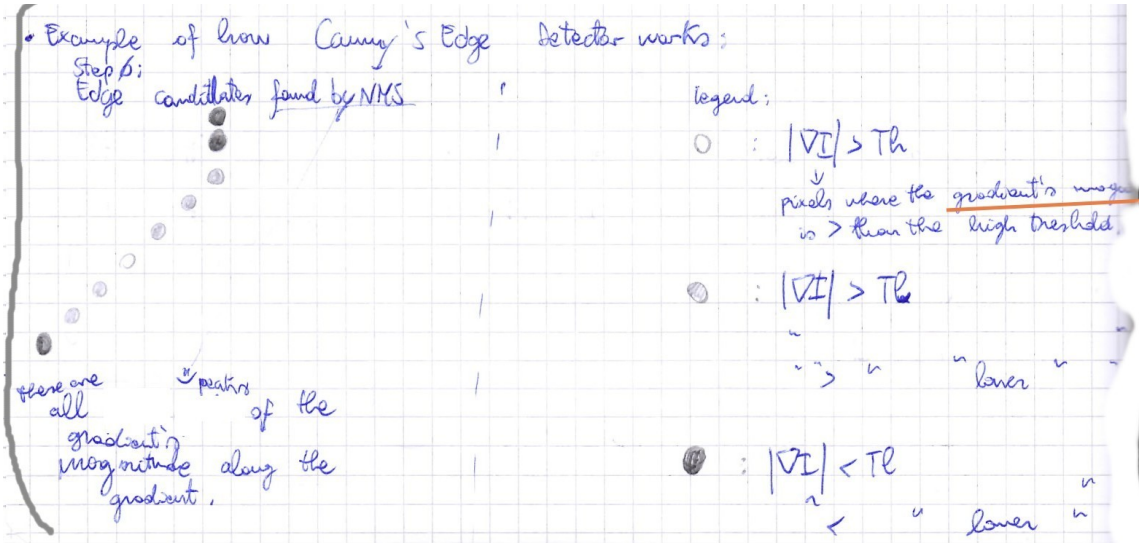
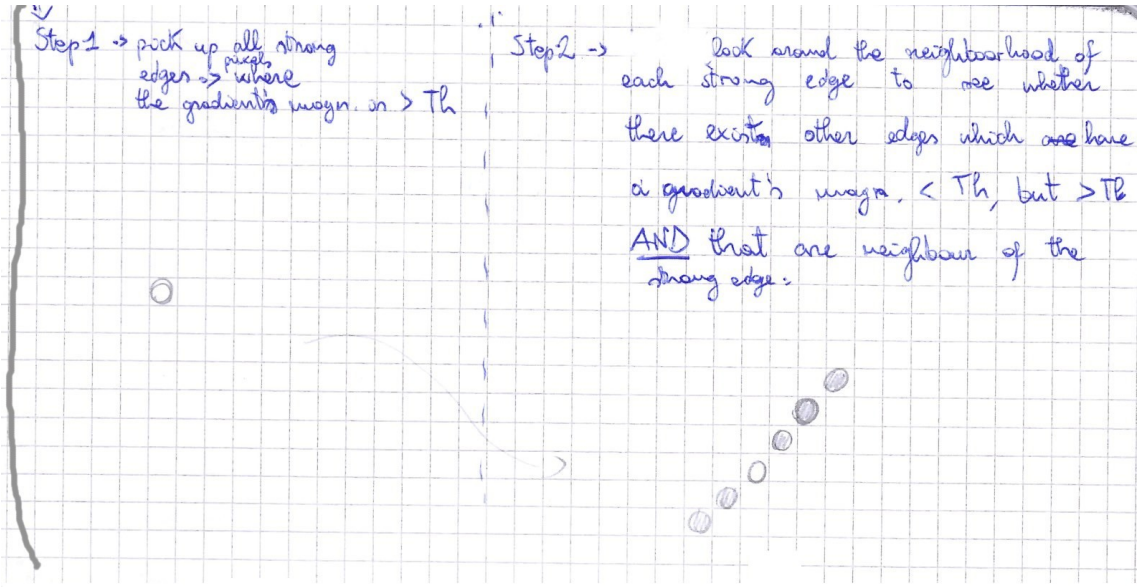


Figure 8.29



=> 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

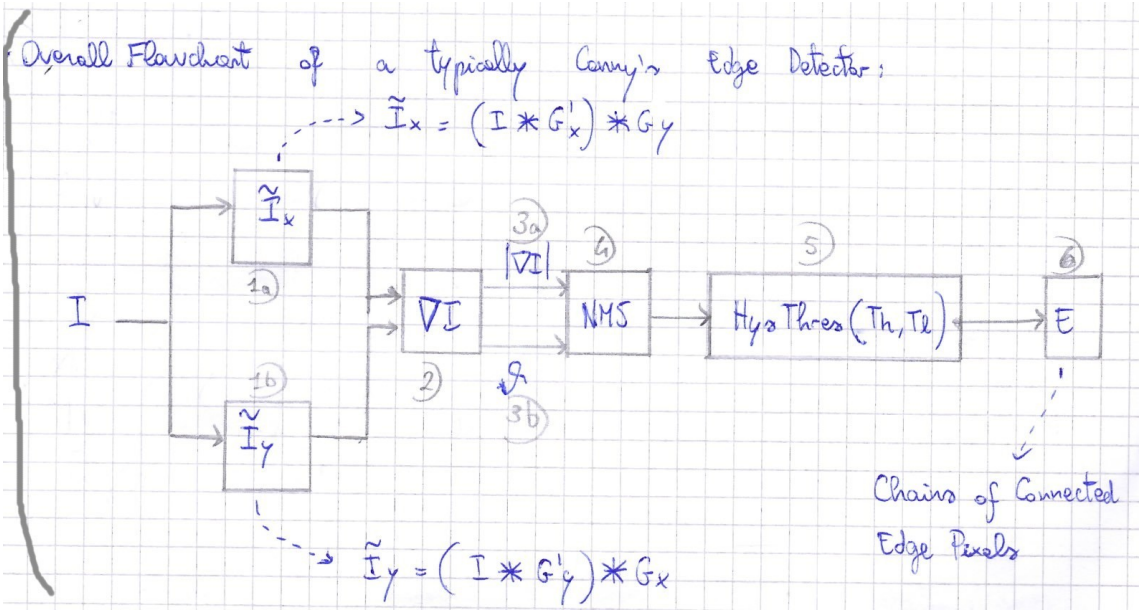


Figure 8.31

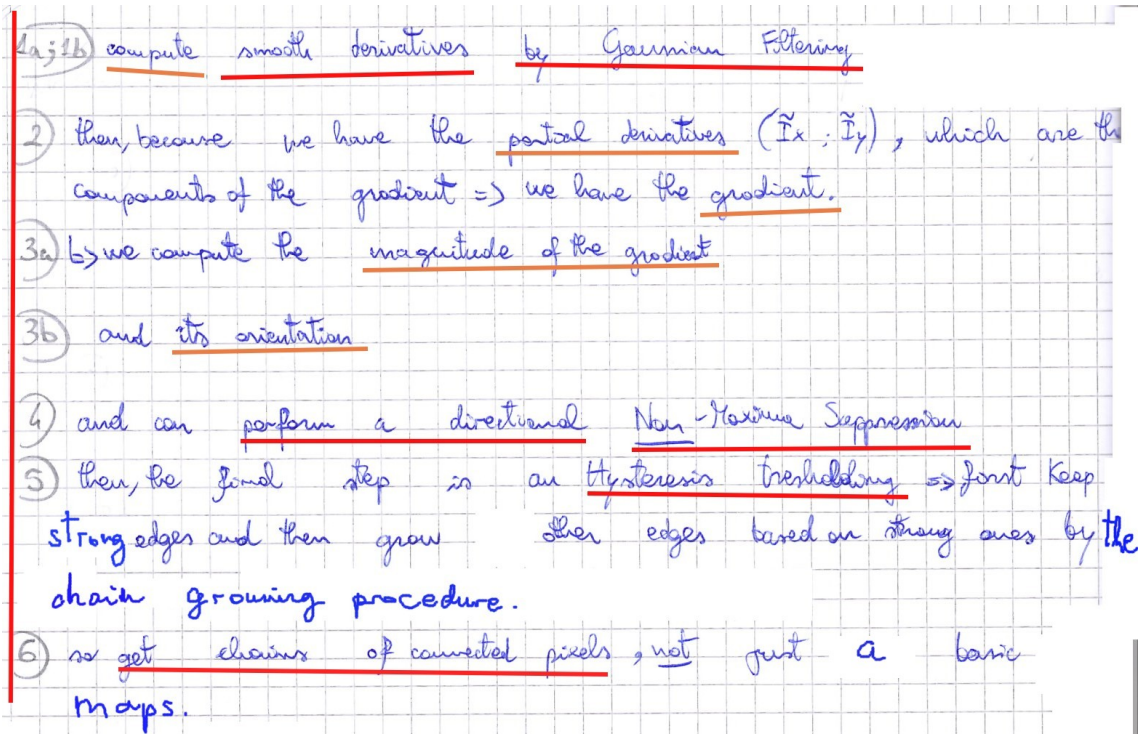

1a ; 1b) compute smooth derivatives by Gaussian Filtering
2) then, because we have the partial derivatives ($\tilde{I}_x$; $\tilde{I}_y$), which are the components of the gradient => we have the gradient.
3a) b) we compute the magnitude of the gradient
3b) and its orientation
4) and can perform a directional Non-Maxima Suppression
5) then, the final step is an Hysteresis Thresholding => 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

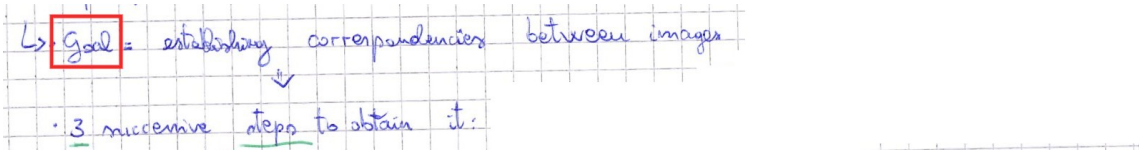


Figure 9.1

Figure 9.2

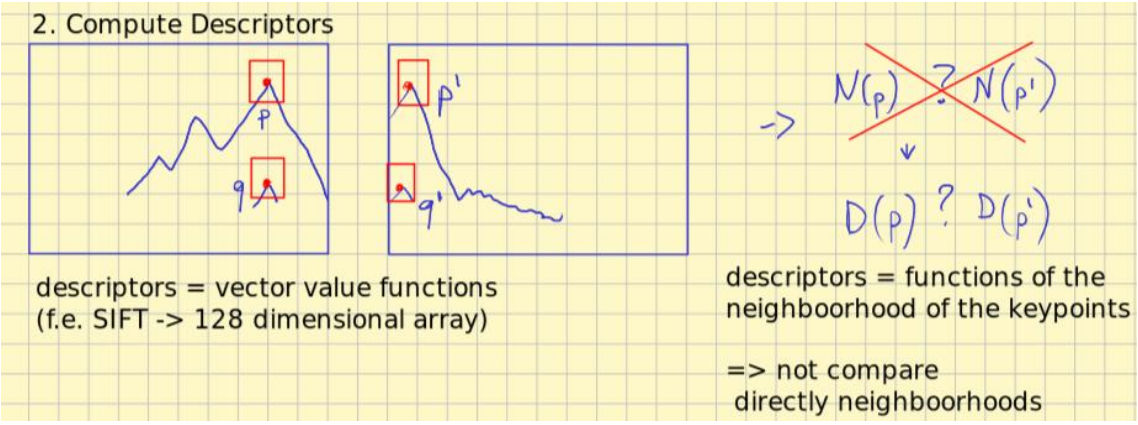


Figure 9.3

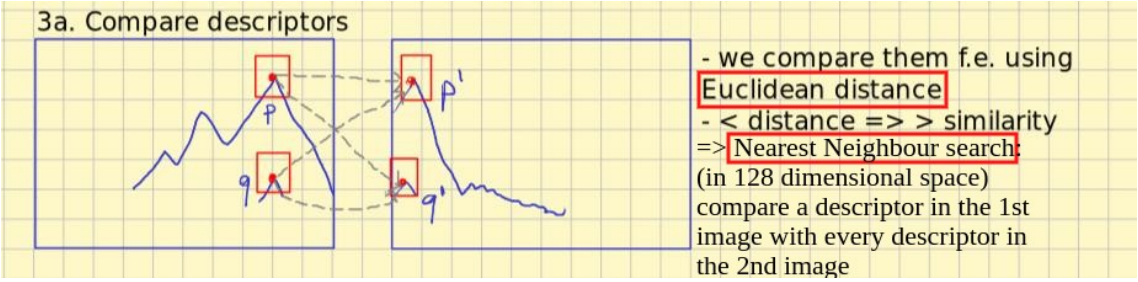


Figure 9.4

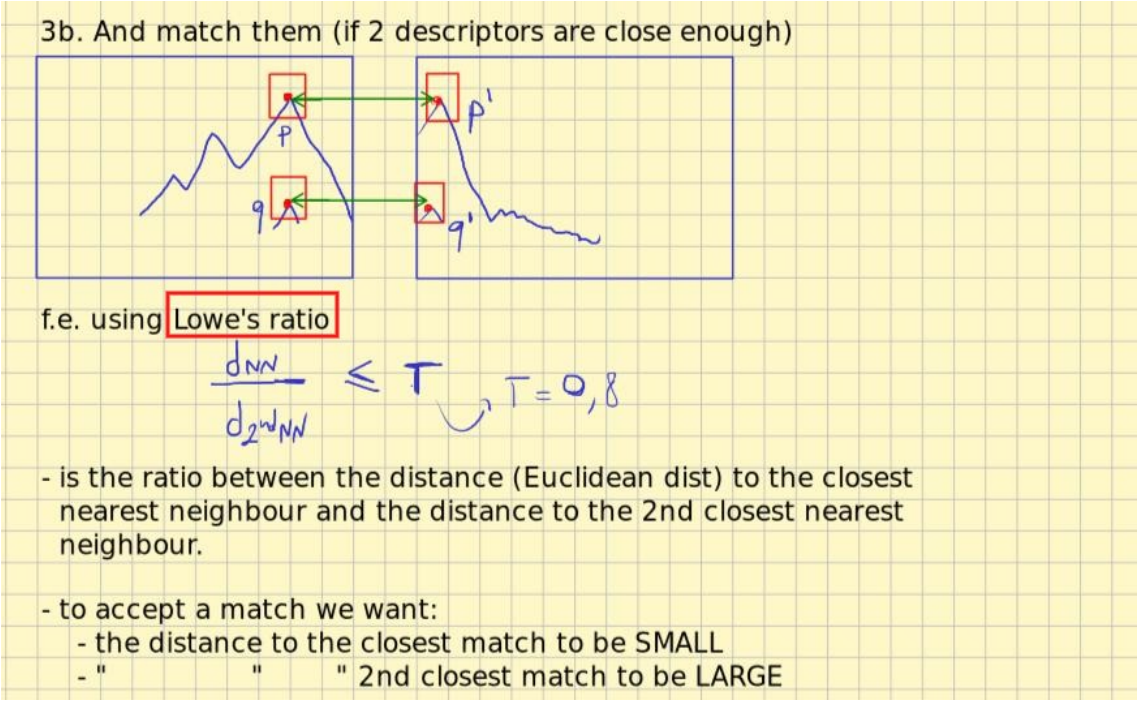


Figure 9.5

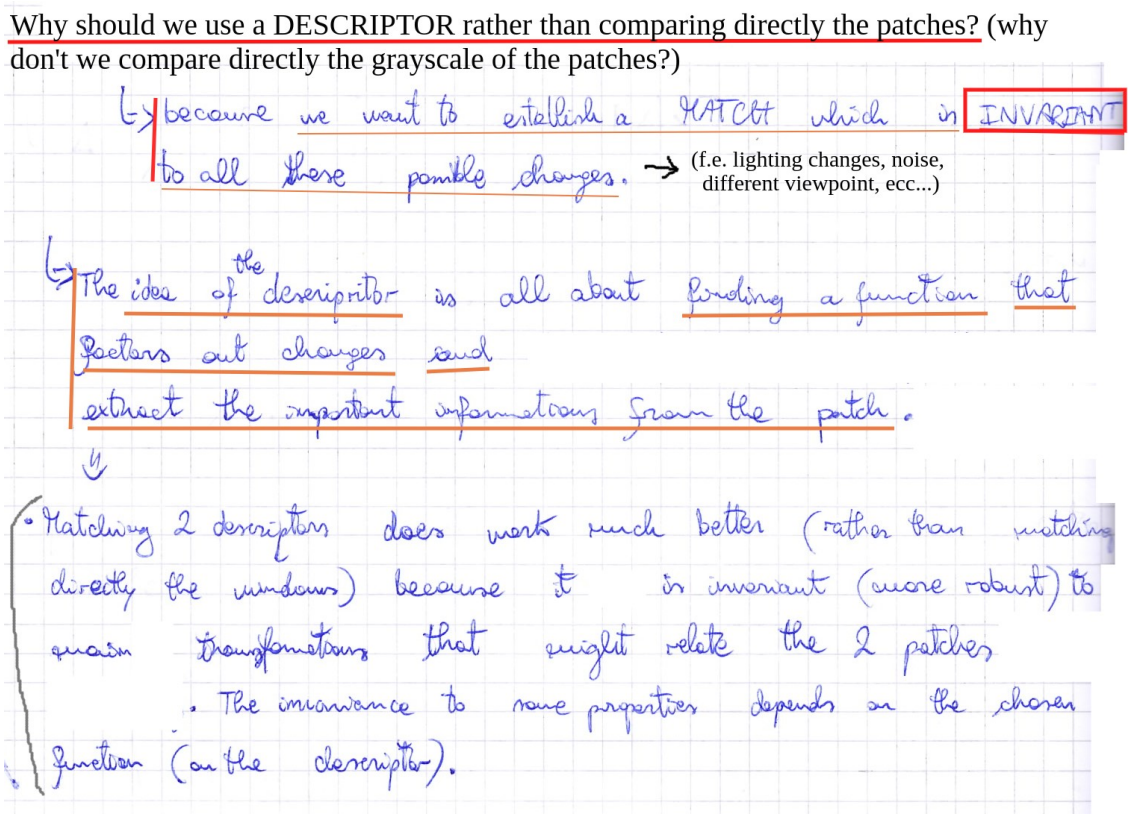


Figure 9.6

9.2 Evaluate the performance of a feature MATCHING pipeline

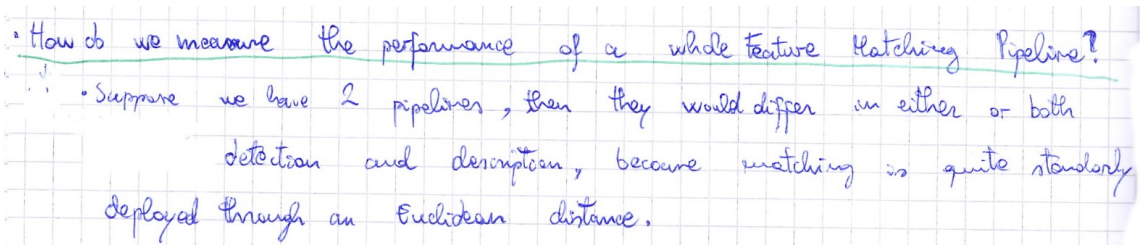


Figure 9.7

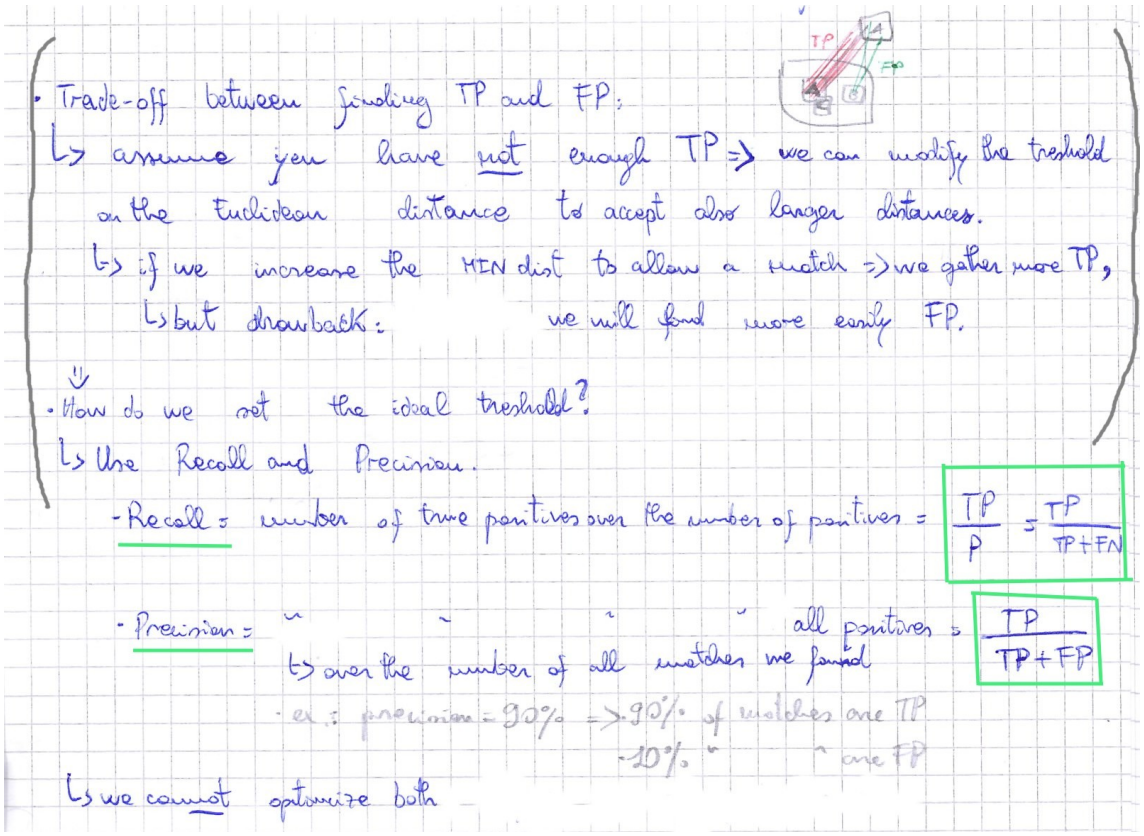


Figure 9.8

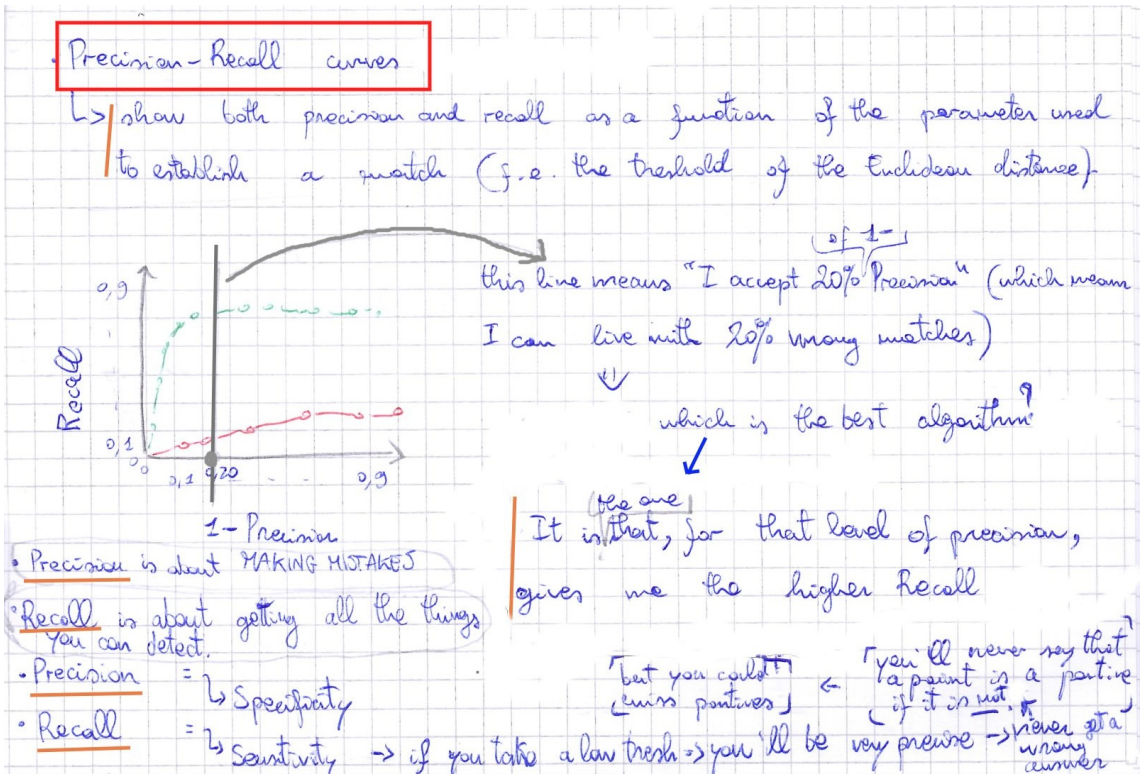


Figure 9.9

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

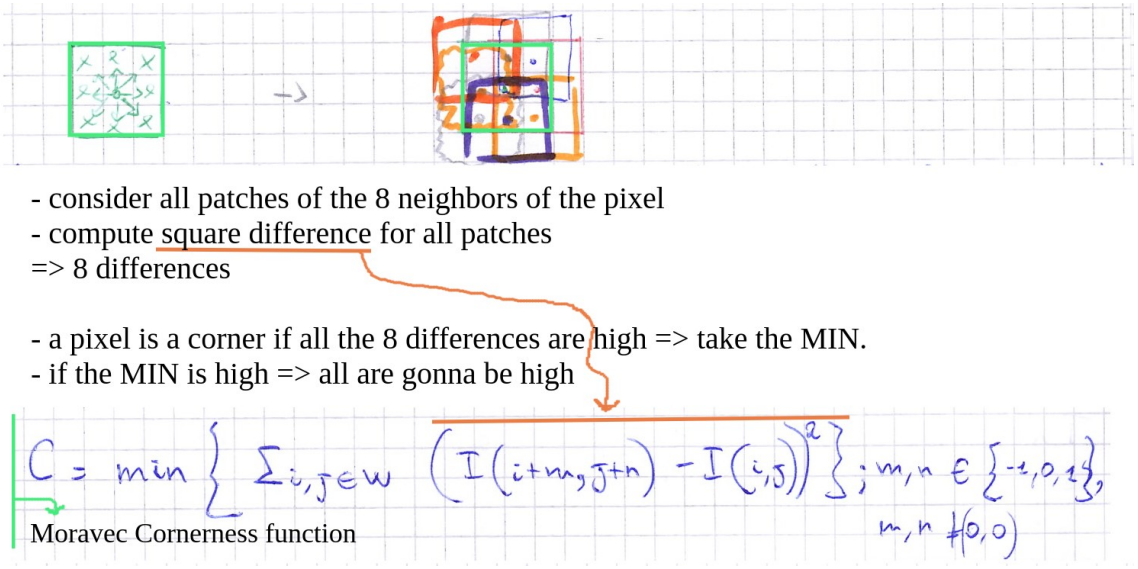


Figure 9.10

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

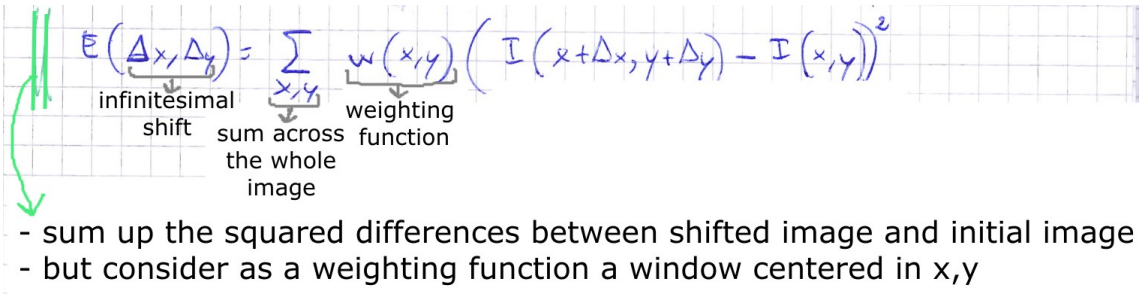


Figure 9.11

- 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)

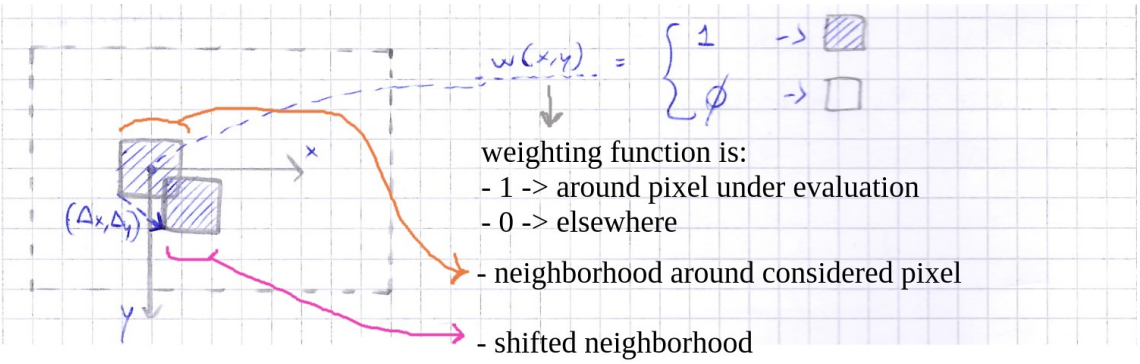


Figure 9.12

  - 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 TAYOLR'S EXPANSION to approximate our function:

Taylor's expansion of function I(x+Δ, y+Δy) in a neighborhood of I(x,y):

$$I(x+\Delta x, y+\Delta y) = I(x,y) + \frac{\delta I(x,y)}{\delta x}\cdot\Delta x + \frac{\delta I(x,y)}{\delta y}\cdot\Delta y$$

I approximate this function through a Taylor's expansion

derivative of I(x,y) along x

derivative of I(x,y) along y

Figure 9.14

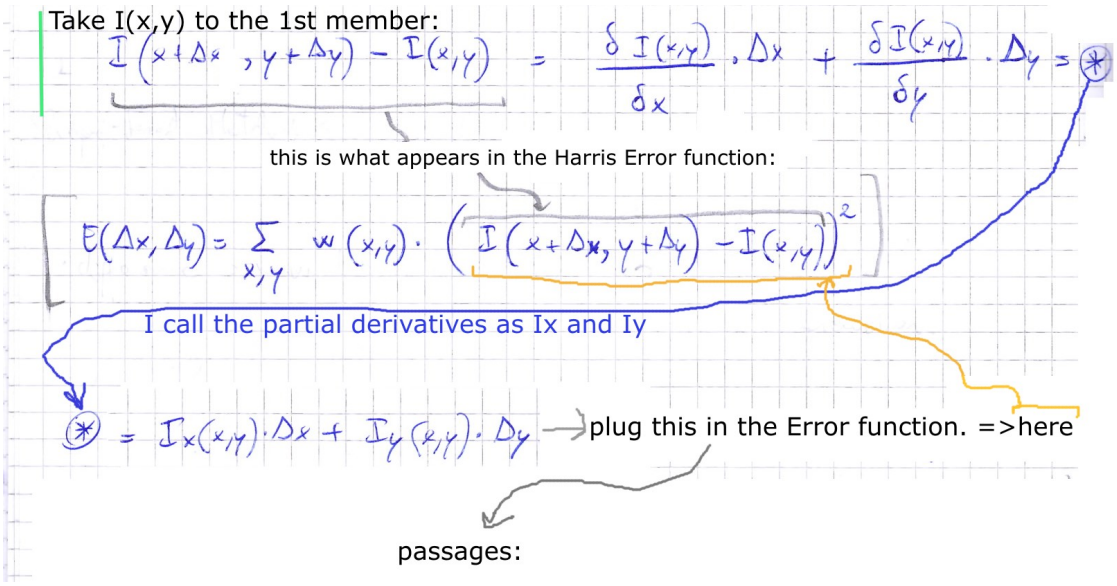


Figure 9.15

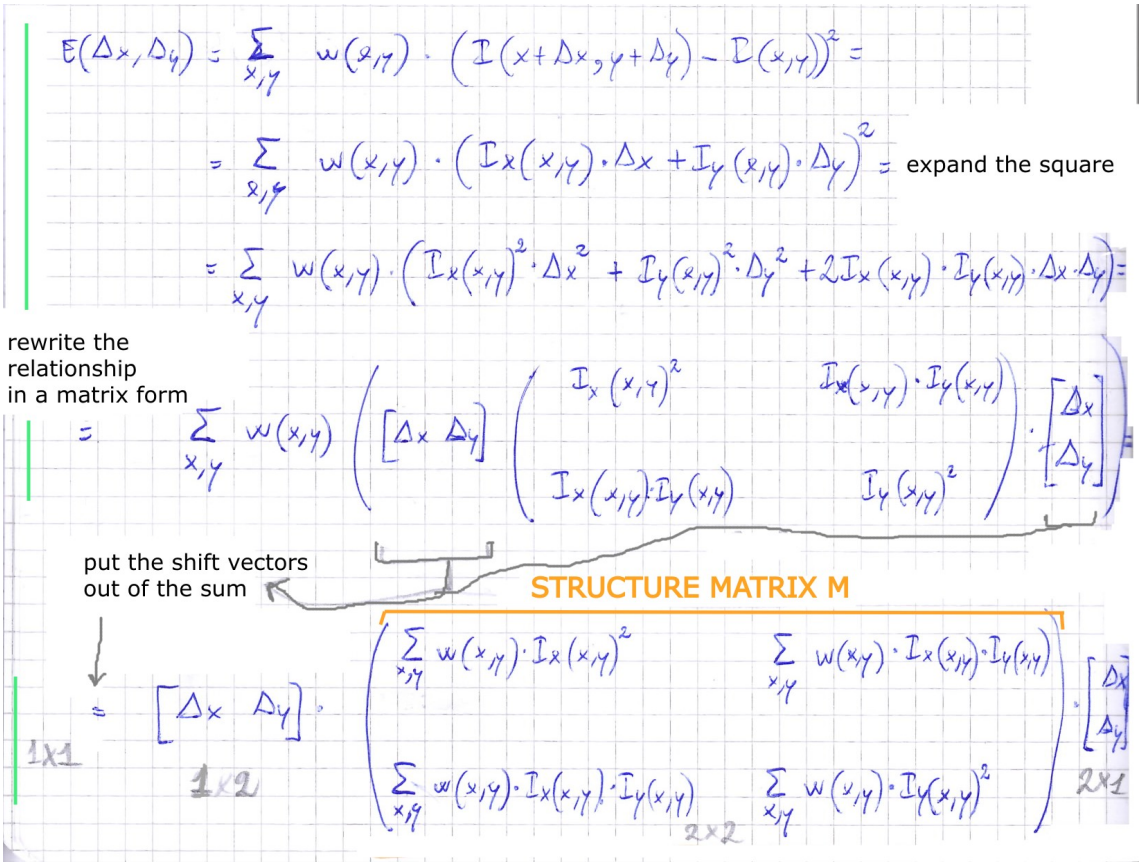


Figure 9.16

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

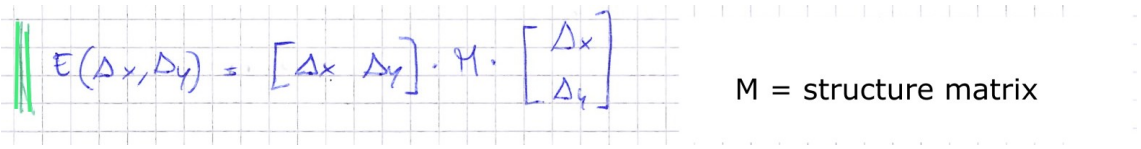


Figure 9.17

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

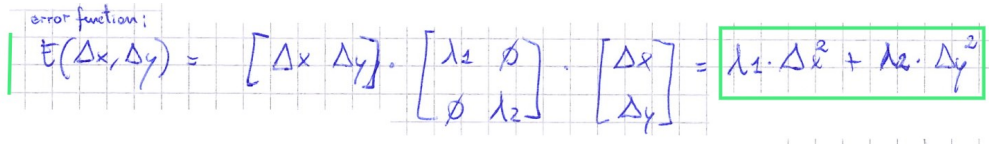


Figure 9.18

- Error as a function of the eigenvalues -> 3 cases:
  1. IF both lambda1 and lambda2 (=the eigenvalues) are SMALL:

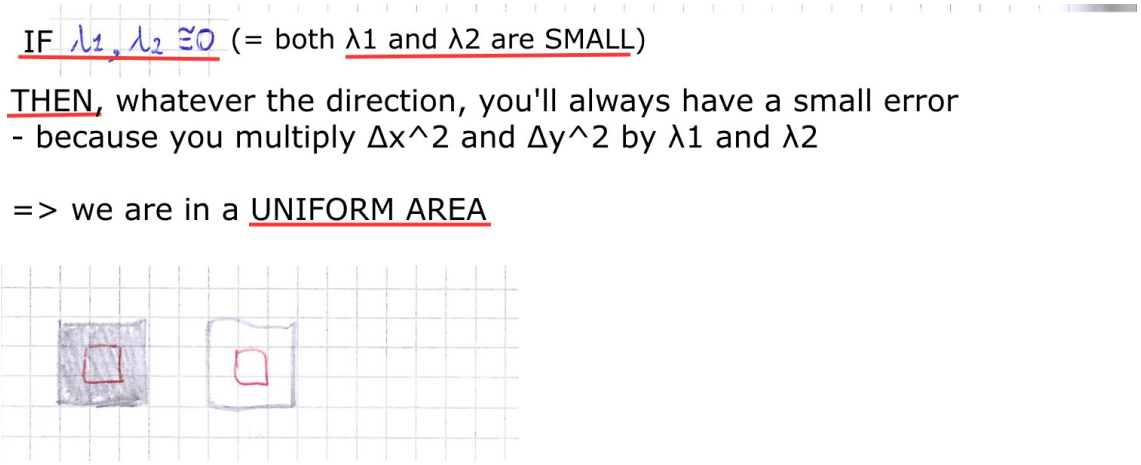


Figure 9.19

  2. IF lambda1 » lambda2:

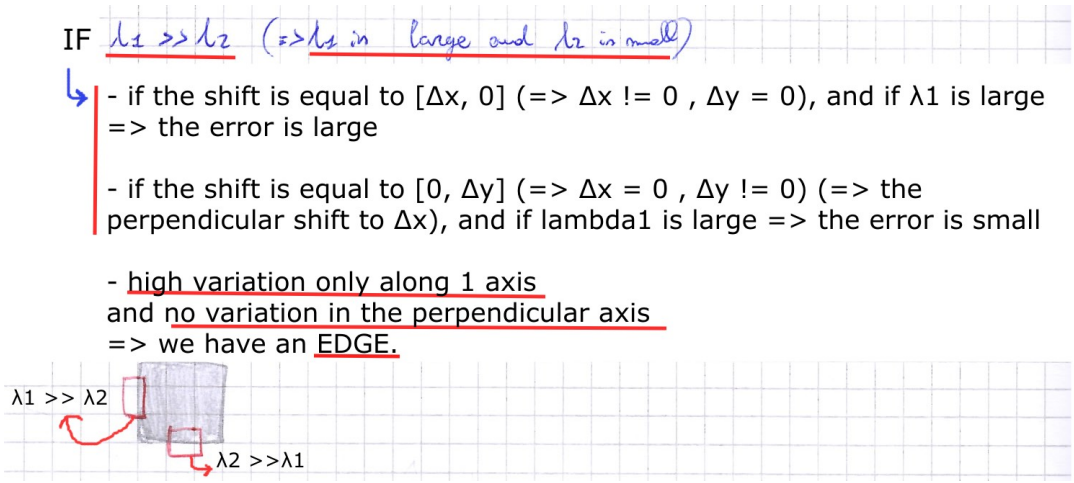


Figure 9.20

3. IF both lambda1 and lambda2 are LARGE:

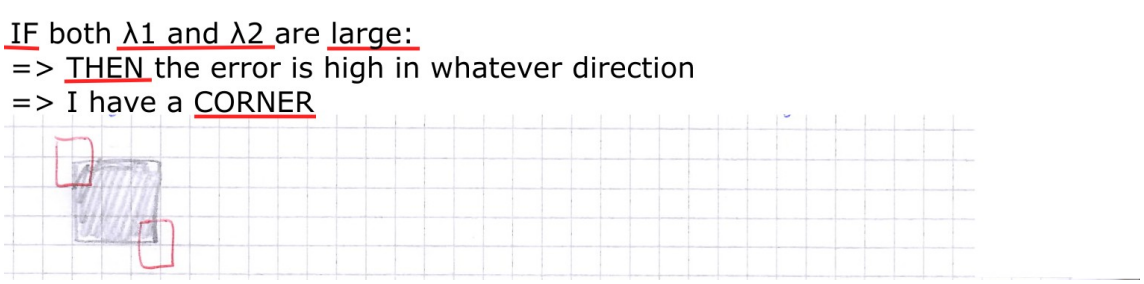


Figure 9.21

- 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

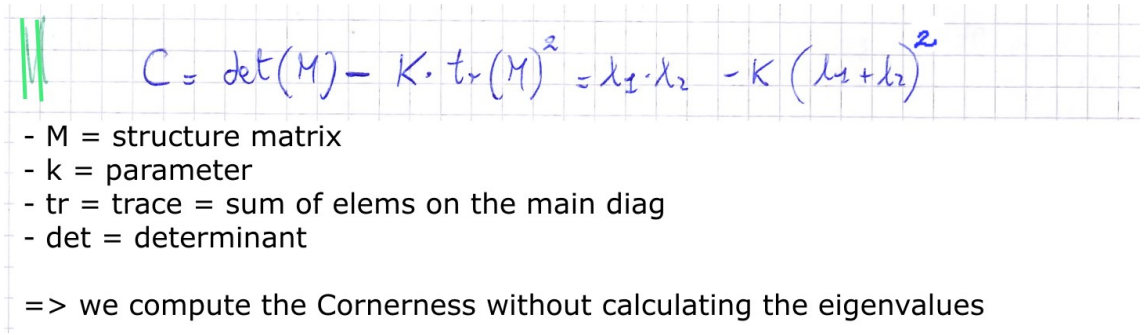


Figure 9.22

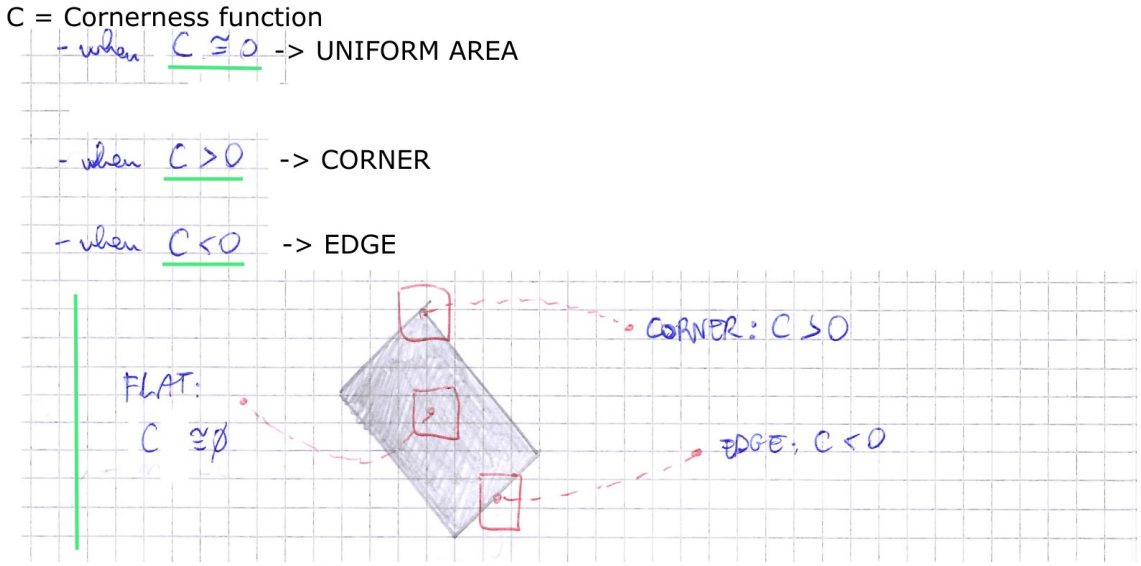


Figure 9.23

- 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
- Ivariance 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:

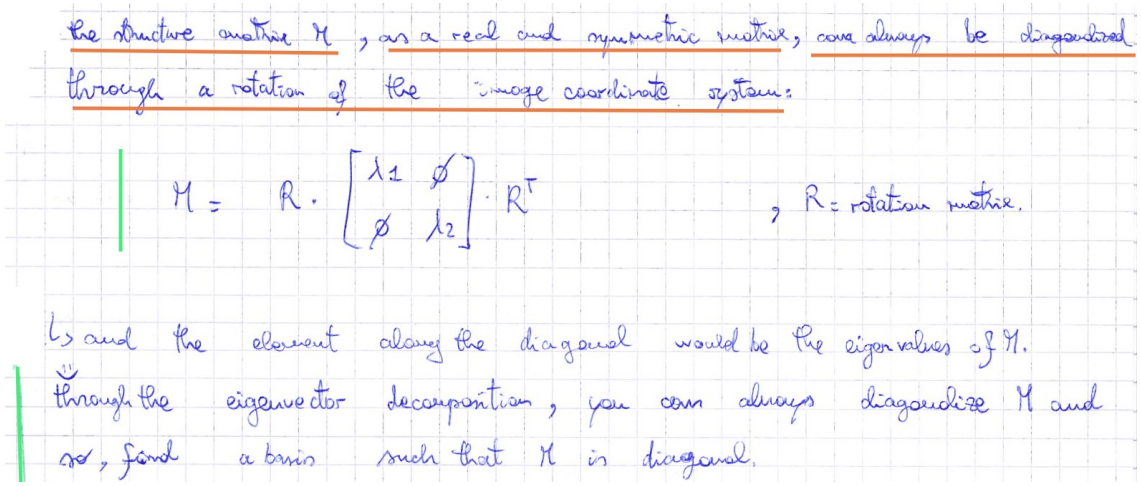


Figure 9.24

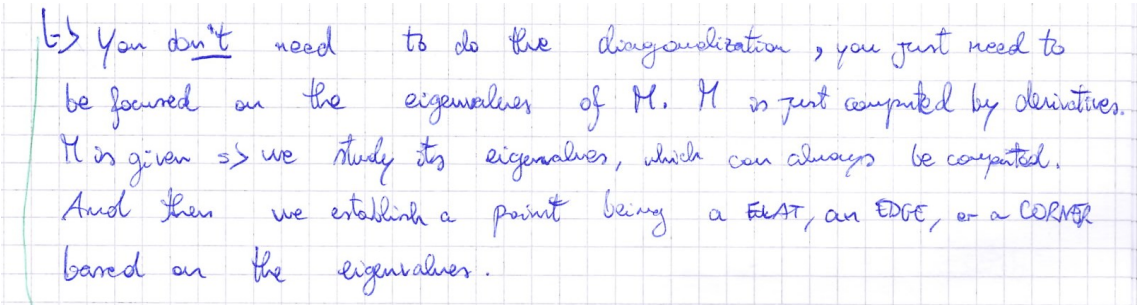


Figure 9.25

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

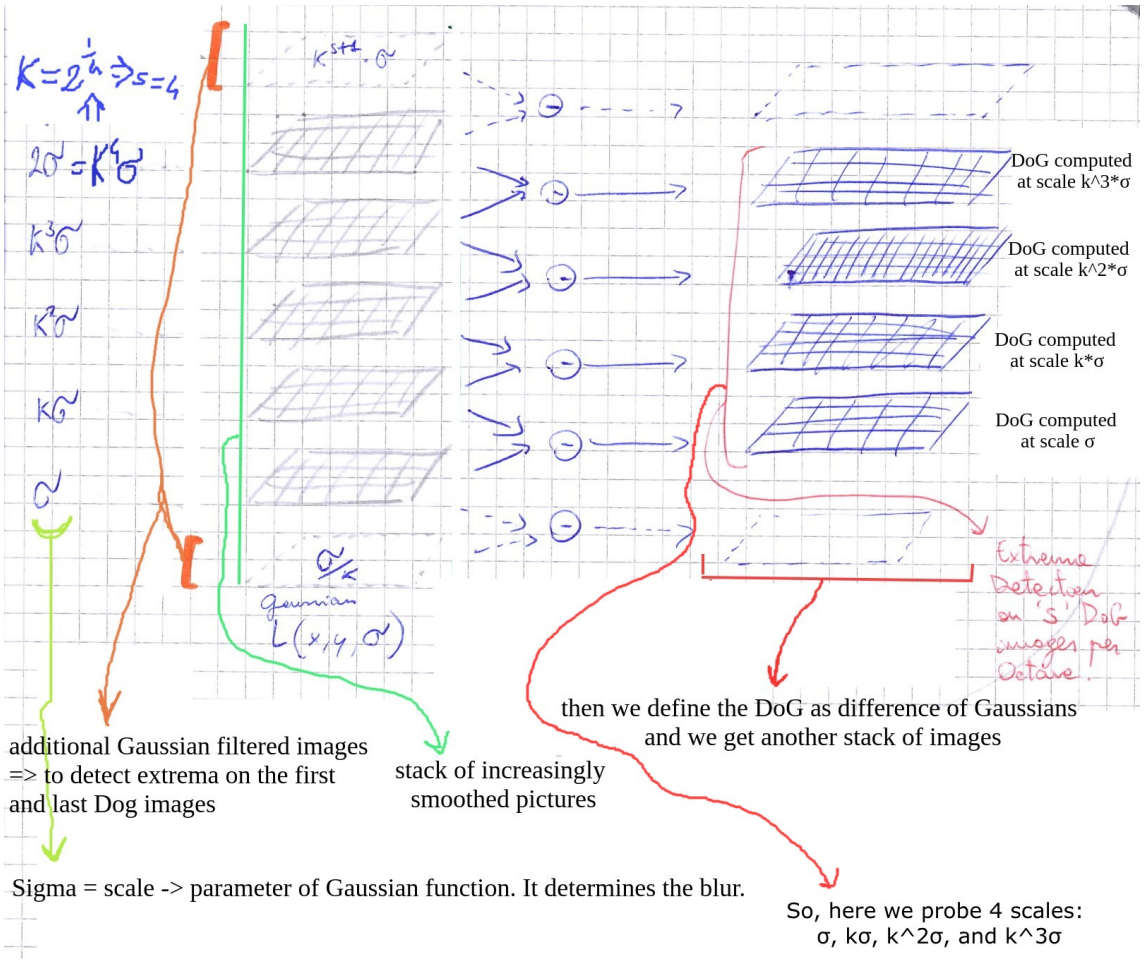

DoG computed at scale k^3*σ
DoG computed at scale k^2*σ
DoG computed at scale k*σ
DoG computed at scale σ
Extreme Detection on 'S' DoG images per Octave!
then we define the DoG as difference of Gaussians and we get another stack of images
additional Gaussian filtered images => to detect extrema on the first and last Dog images
stack of increasingly smoothed pictures
Sigma = scale -> parameter of Gaussian function. It determines the blur.
So, here we probe 4 scales: σ, kσ, k^2σ, and k^3σ

Figure 9.26

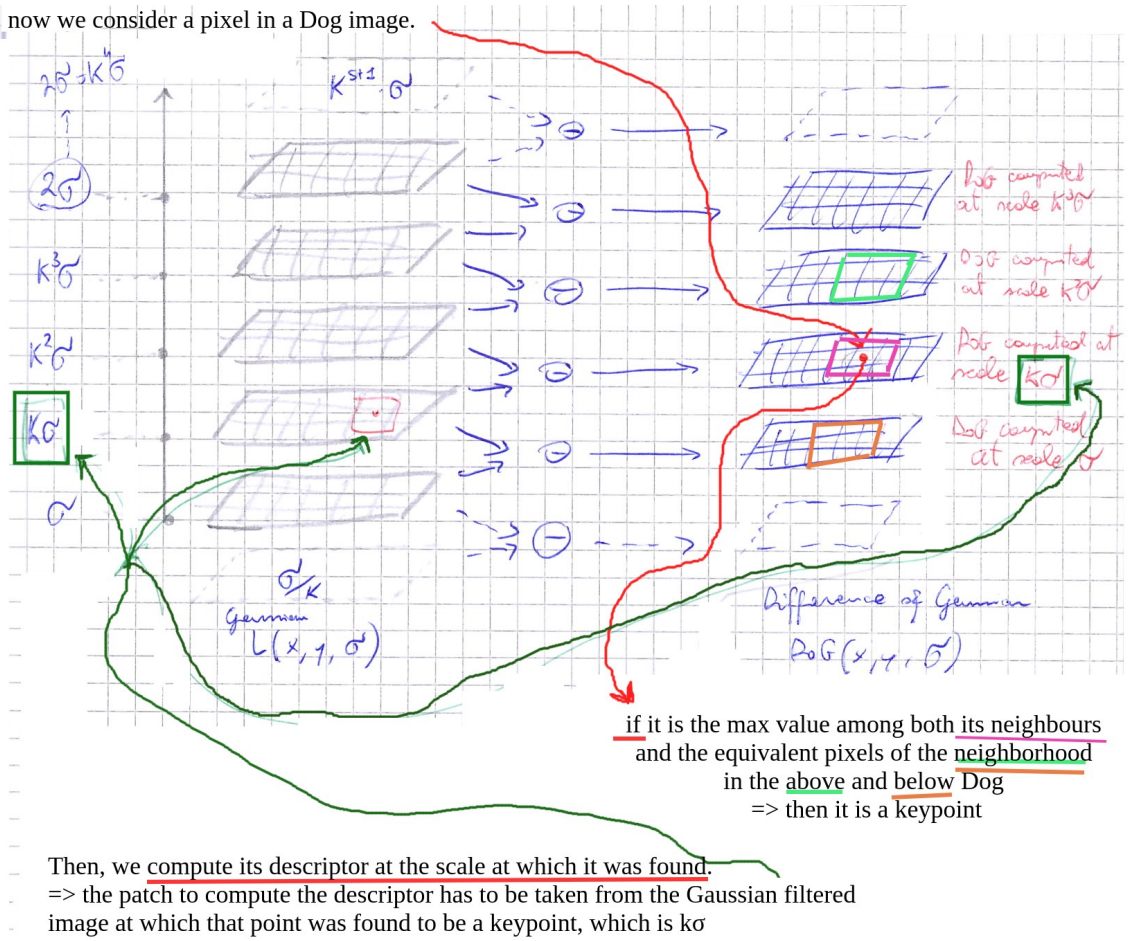


Figure 9.27

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 0> more computations)
- => shrink the images (divide by 2 heigh 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

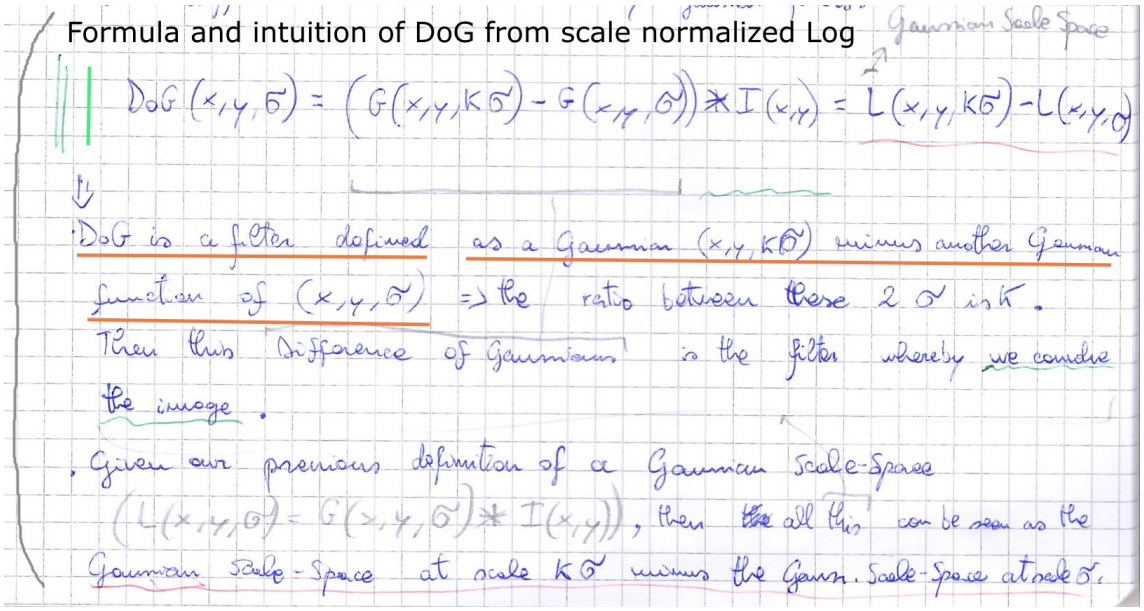


Figure 9.29

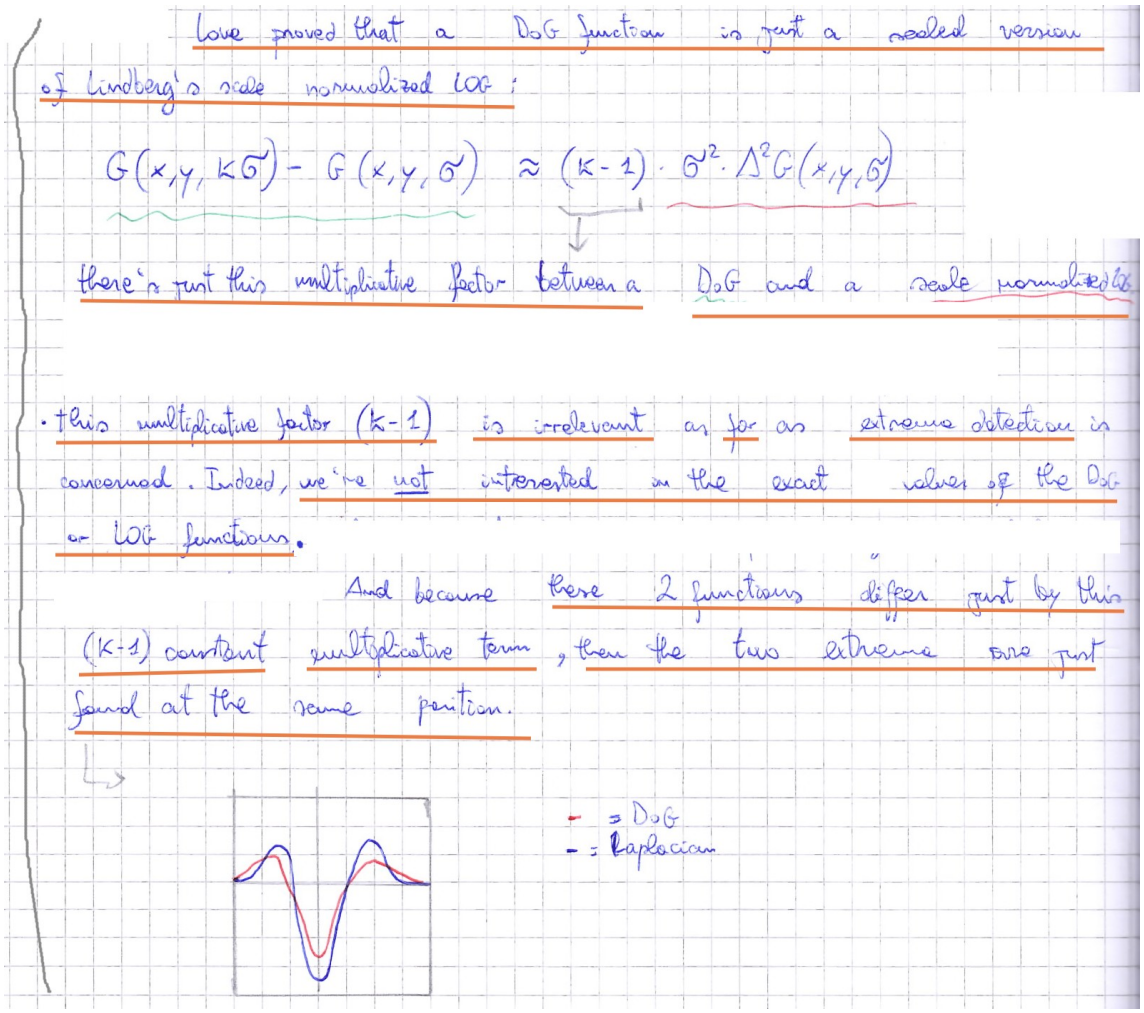


Figure 9.30

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:

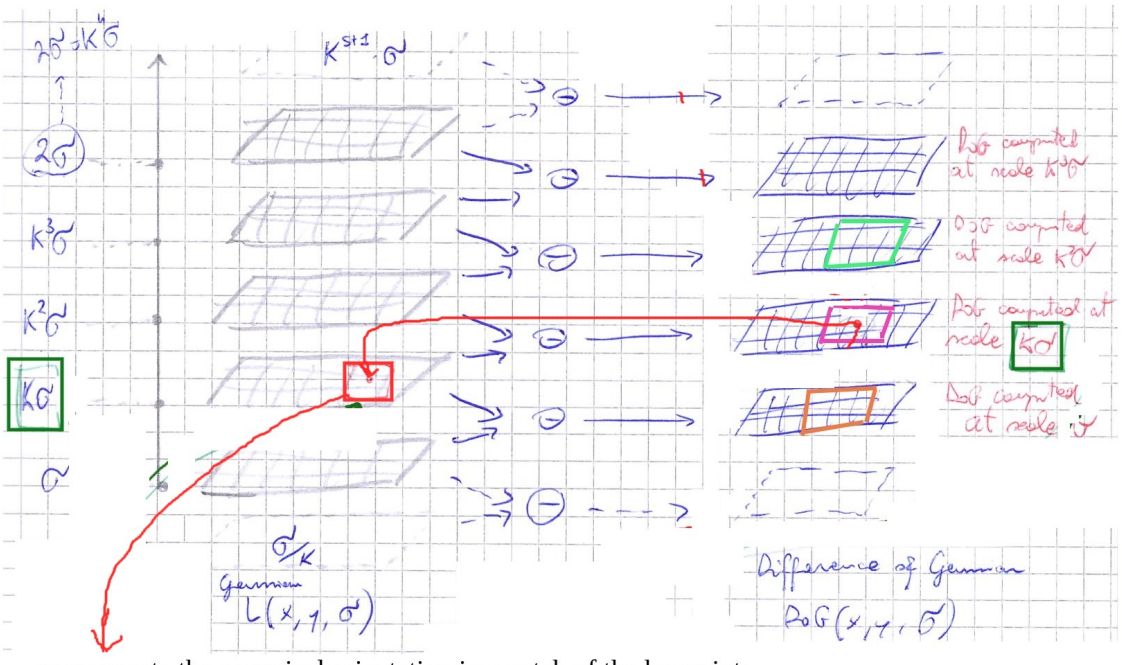
- I compute the descriptor at a right scale => it has the right level of blurring => the descrtiptor is invariant wrt scale.

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

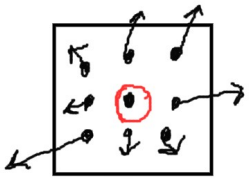
- we first want to find a canonical orientation of the patch of the keypoint => to have a descriptor which is rotation invariant => because we want to match features invariantly to rotation

To compute the canonical orientation we create an orientation histogram:

- divide angular range into bins (in SIFT 10° => 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:



= 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

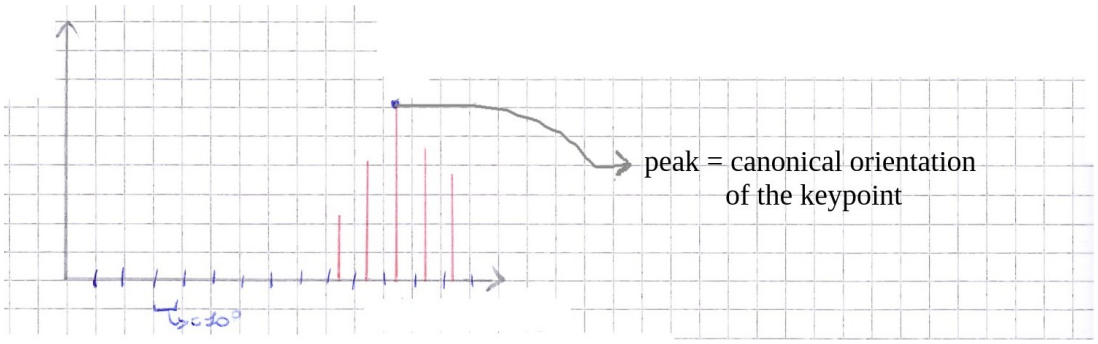


Figure 9.32

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

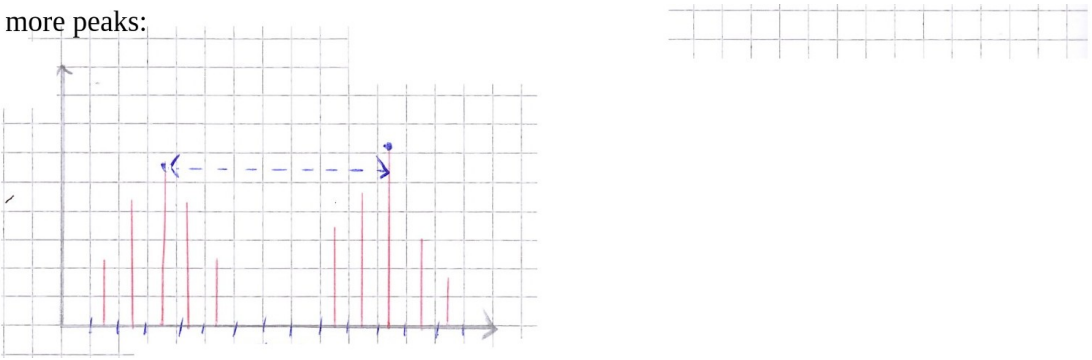


Figure 9.33

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 (=> invariant to rotation thanks to the canonical orientation given by the orientation histogram).

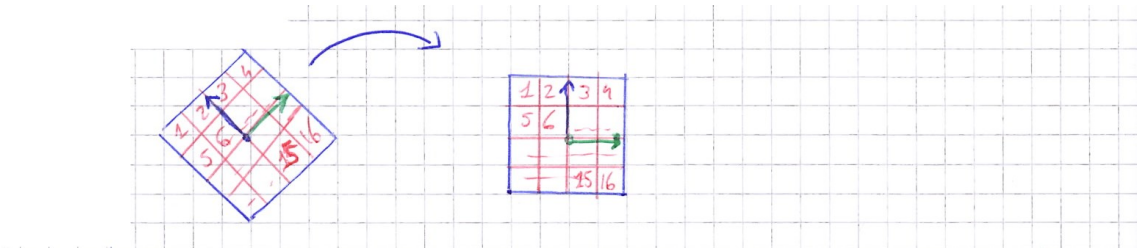


Figure 9.34

- 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

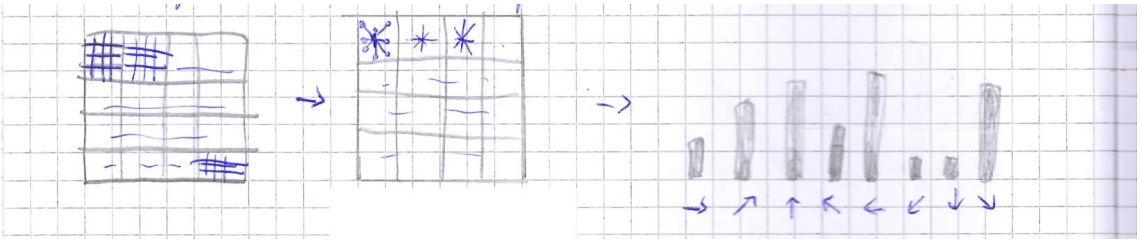


Figure 9.35

=> 8 bins per histogram x 16 sub-regions = 128 dimensional vector (= SIFT descriptor)

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

9.5.1 Validating matches in SIFT

Lowe's ratio -> Figure 9.4.

Chapter 10

Object Detection

10.1 Template matching

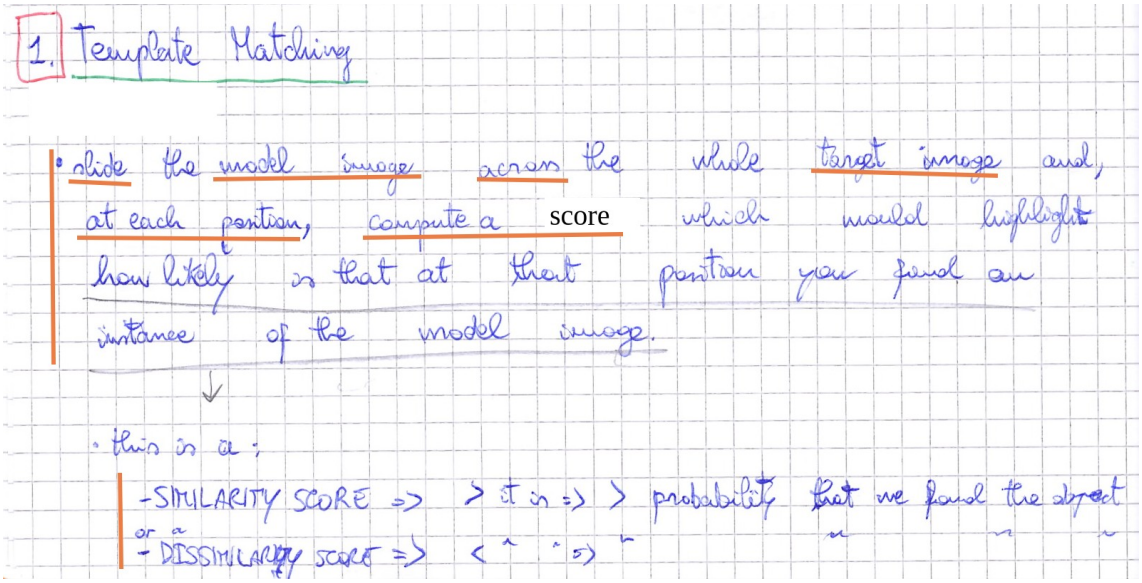


Figure 10.1

10.1.1 SSD

What are the (dis)similarity functions?

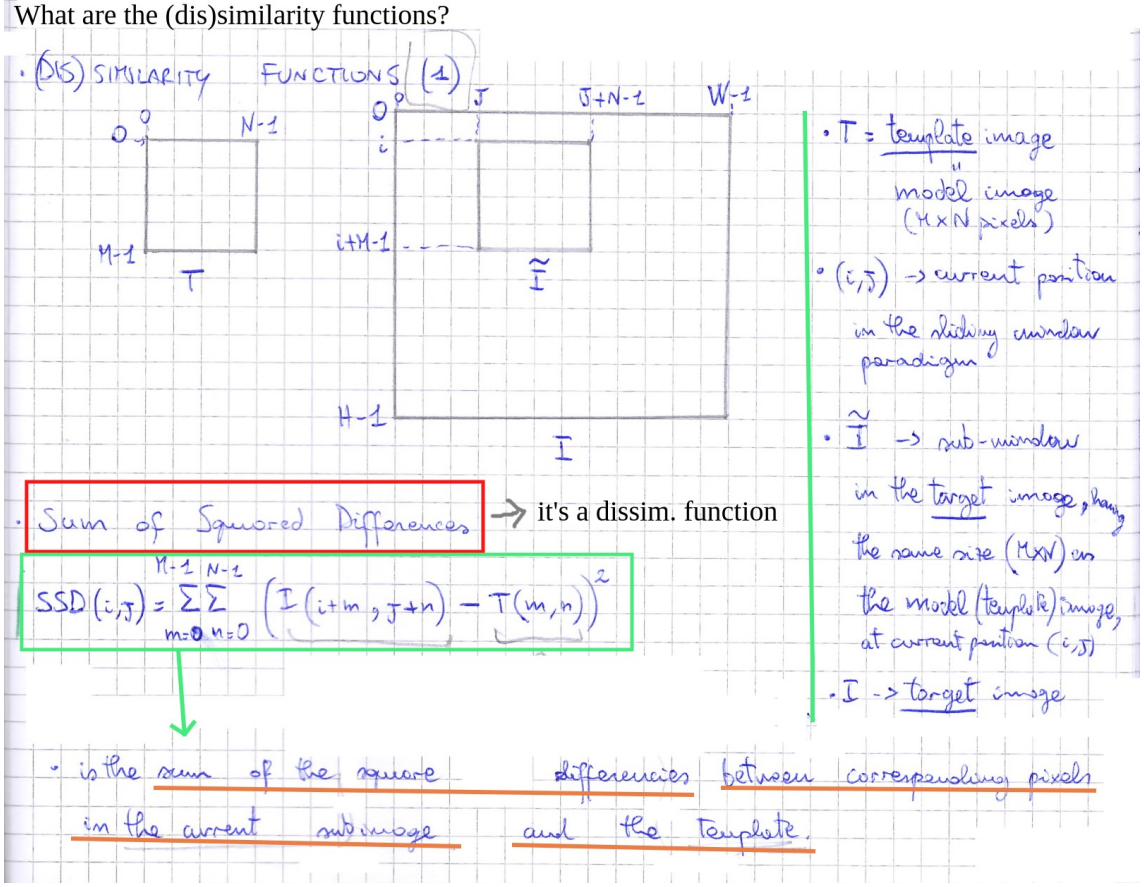


Figure 10.2

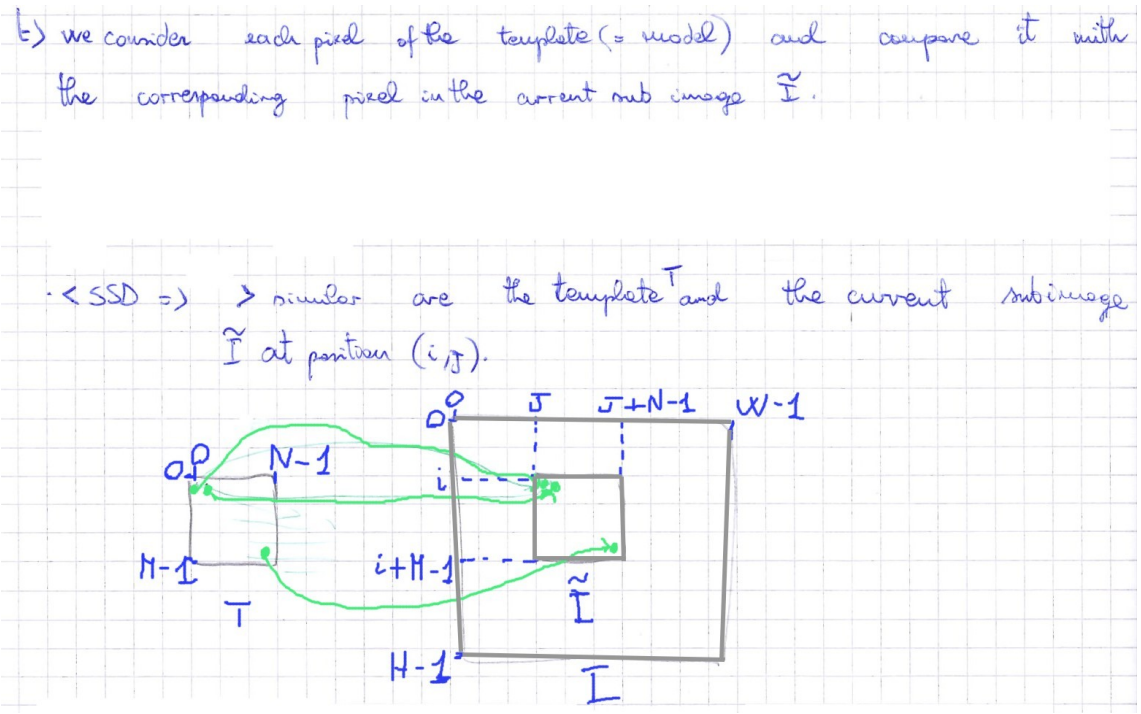


Figure 10.3

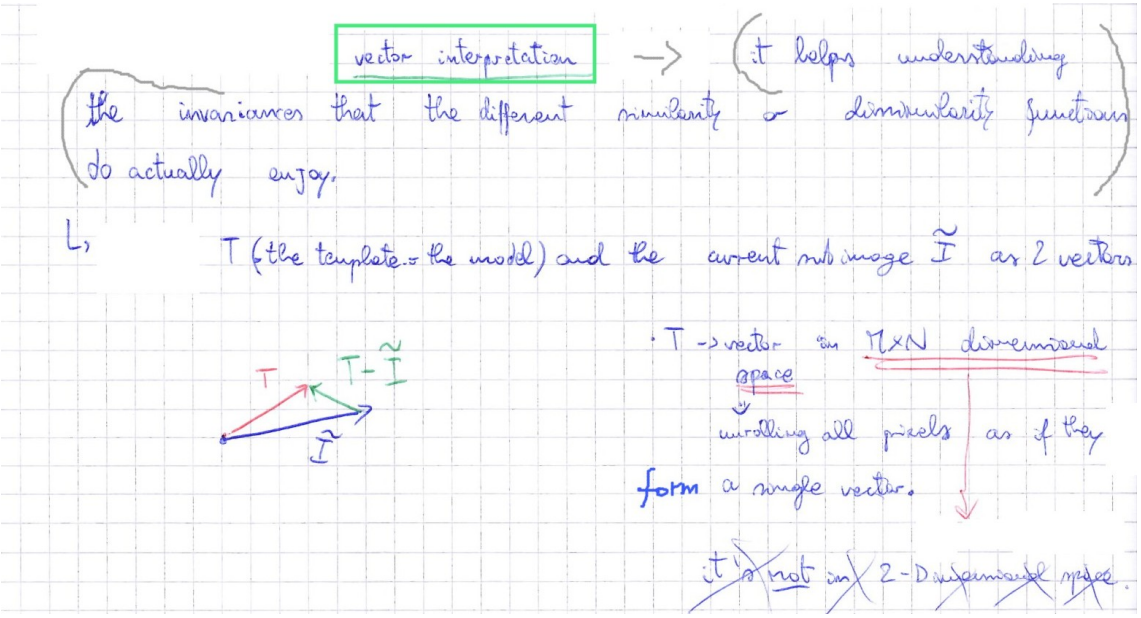


Figure 10.4

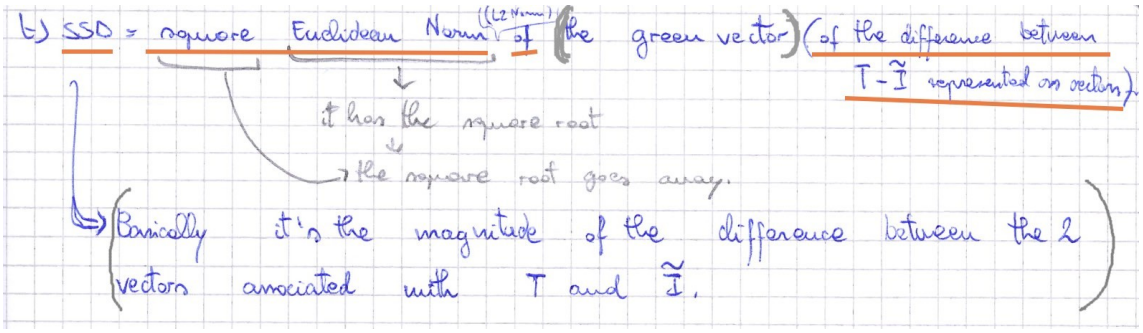


Figure 10.5

10.1.2 SAD

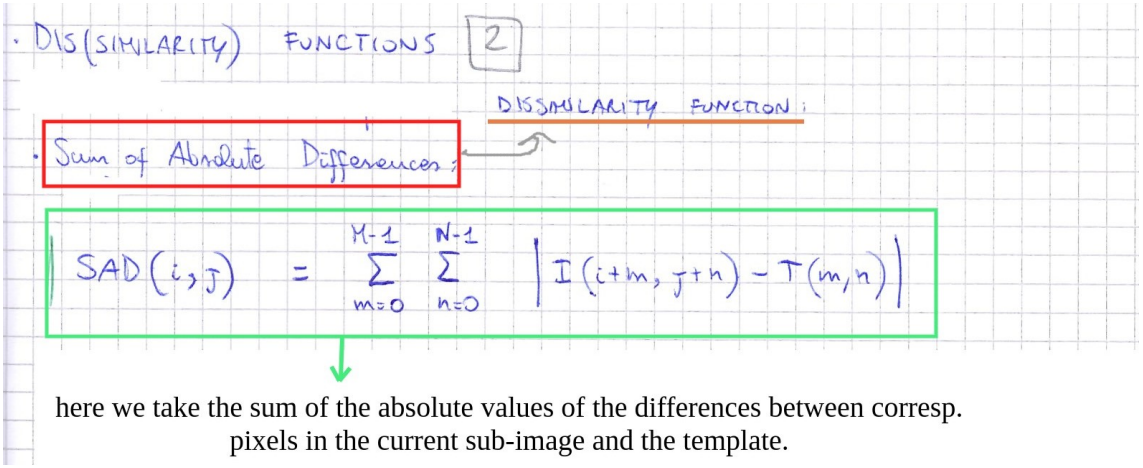


Figure 10.6

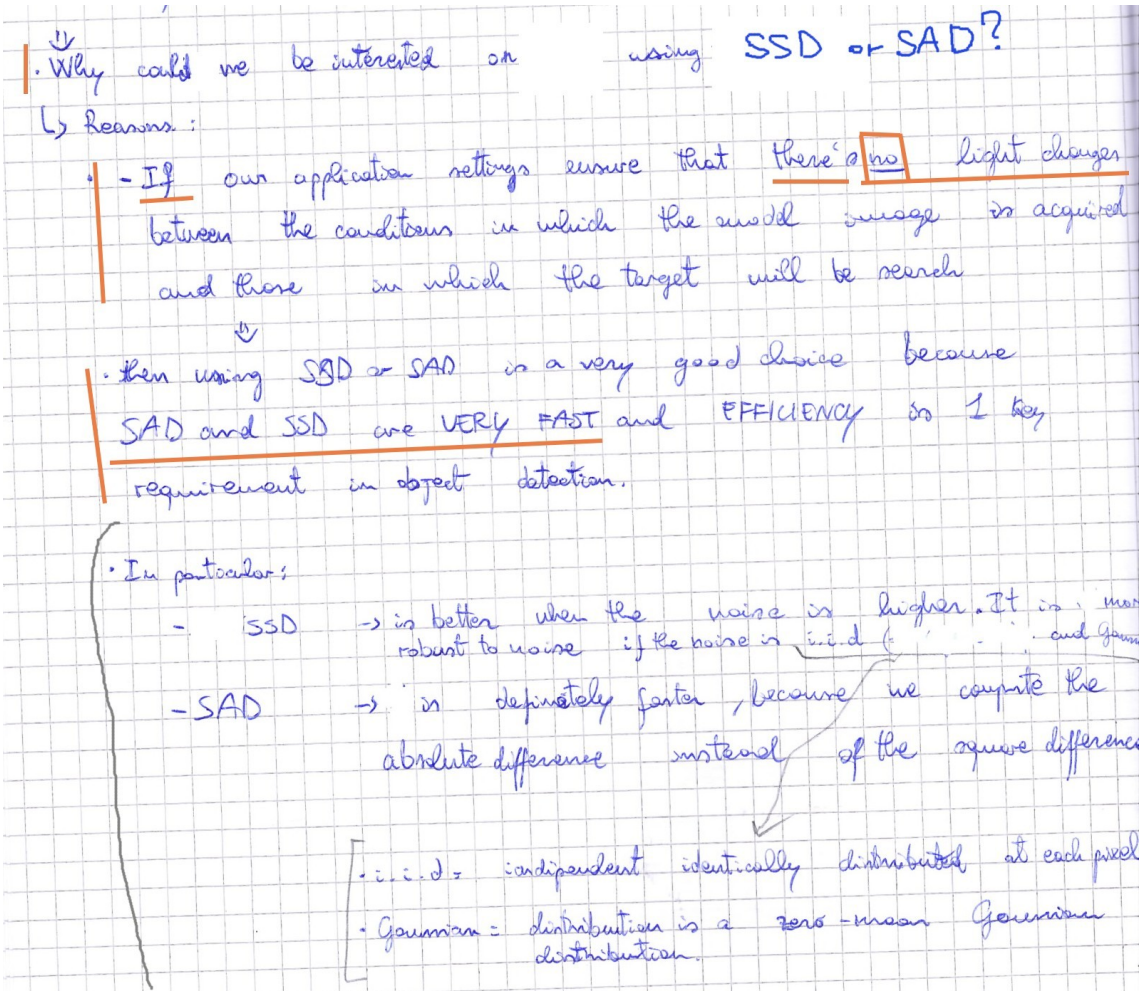


Figure 10.7

10.1.3 NCC

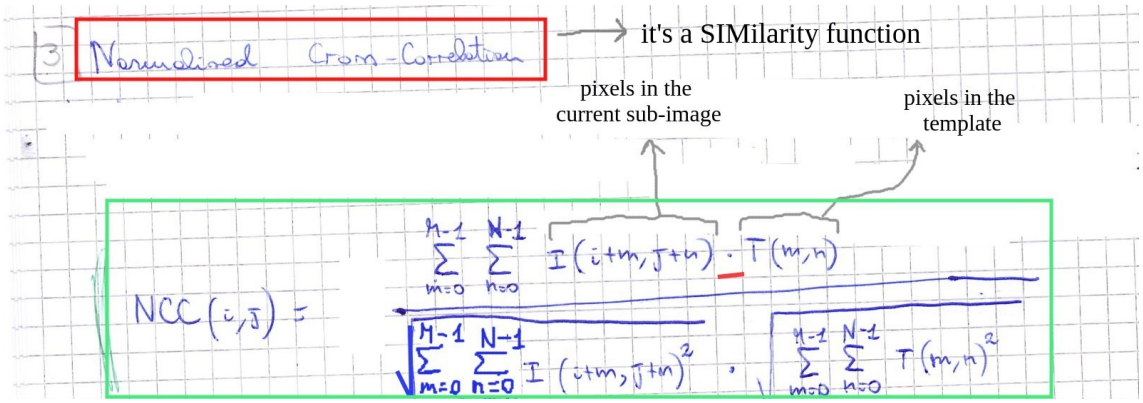


Figure 10.8

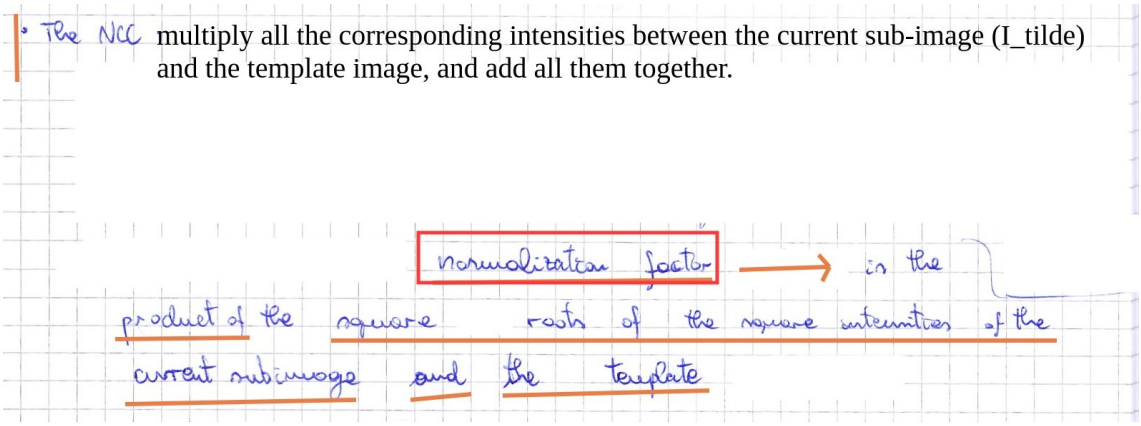


Figure 10.9

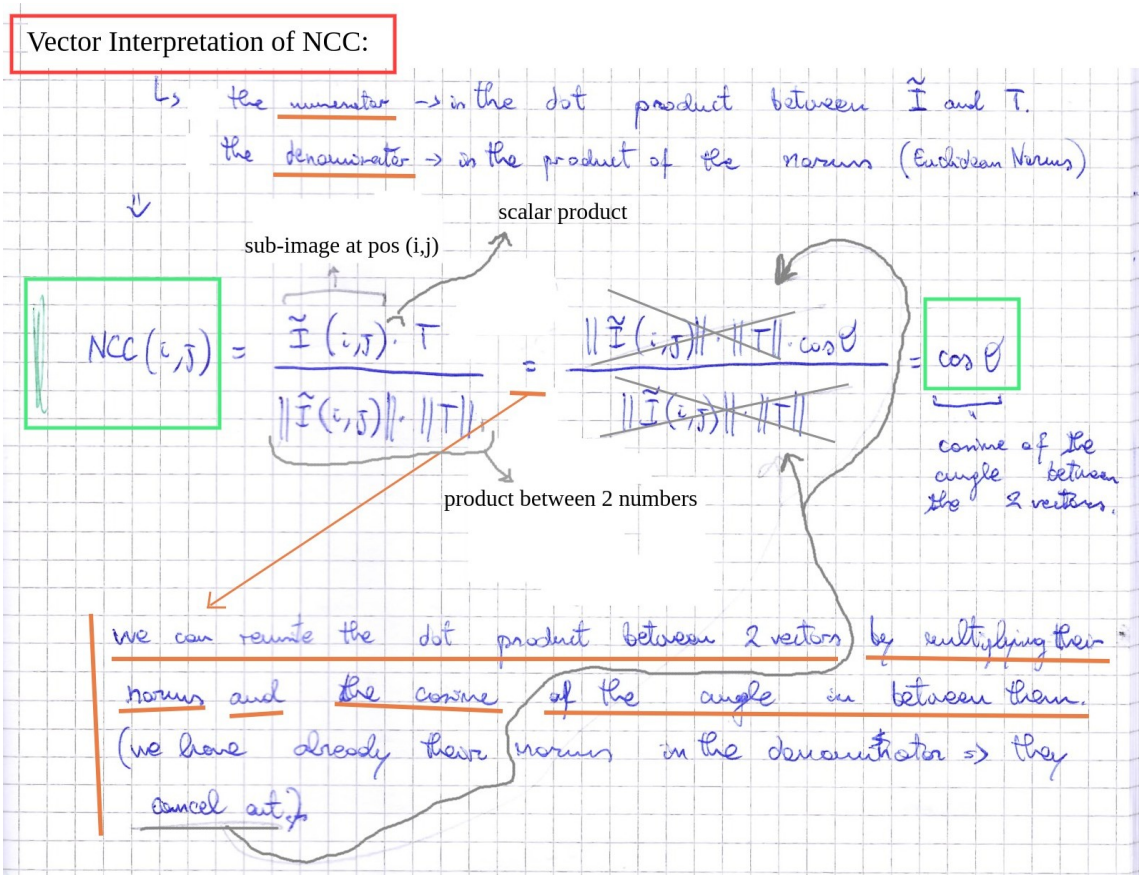


Figure 10.10

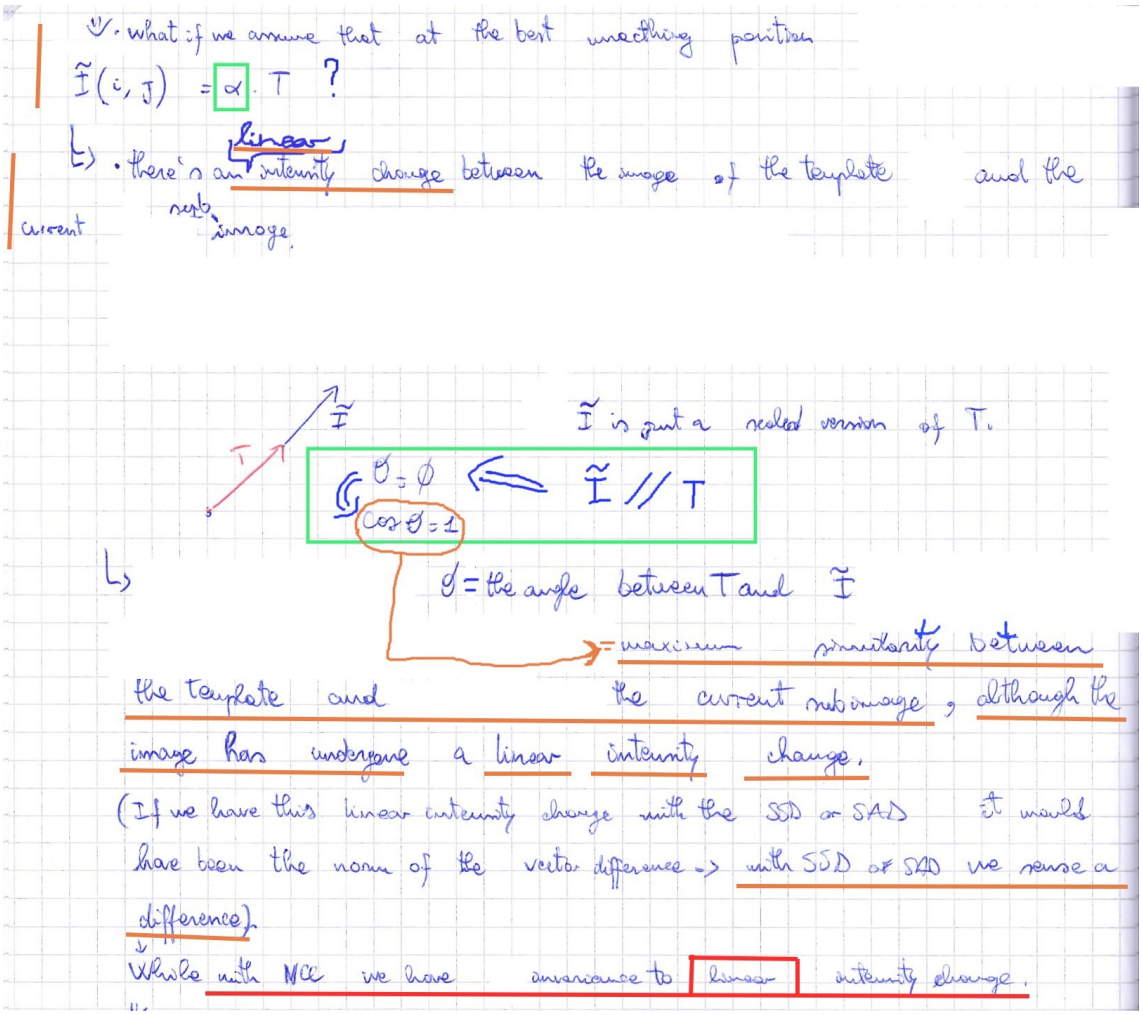


Figure 10.11

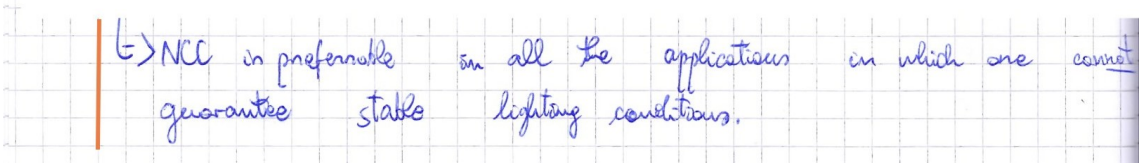


Figure 10.12

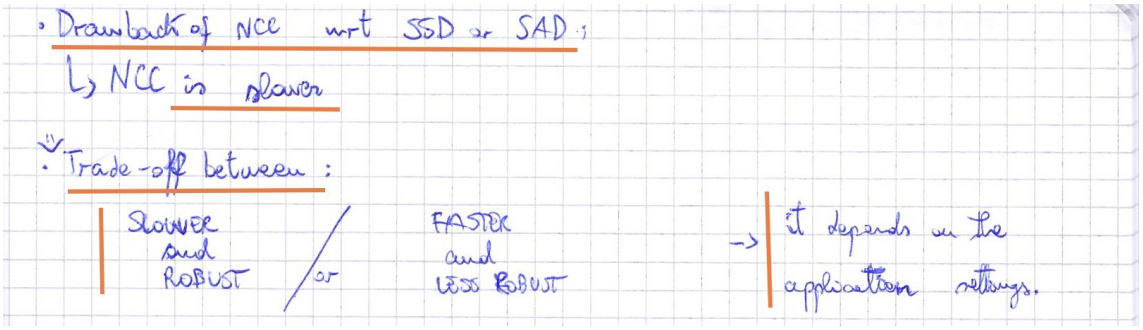


Figure 10.13

10.1.4 ZNCC

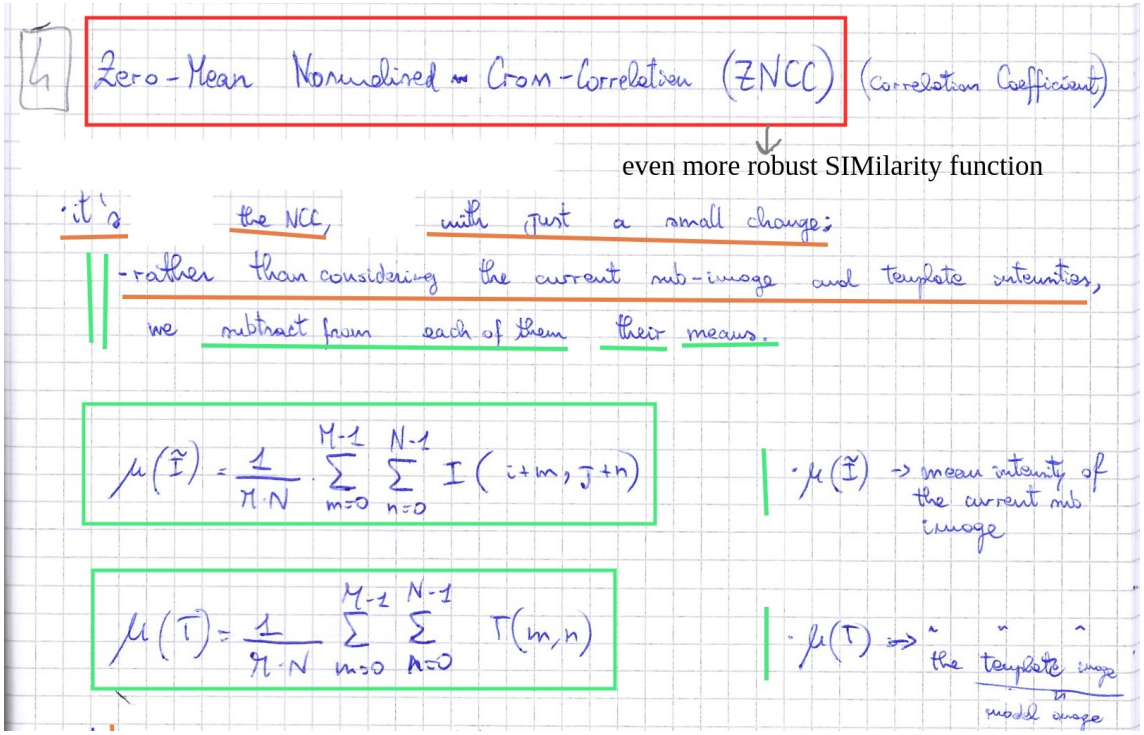


Figure 10.14

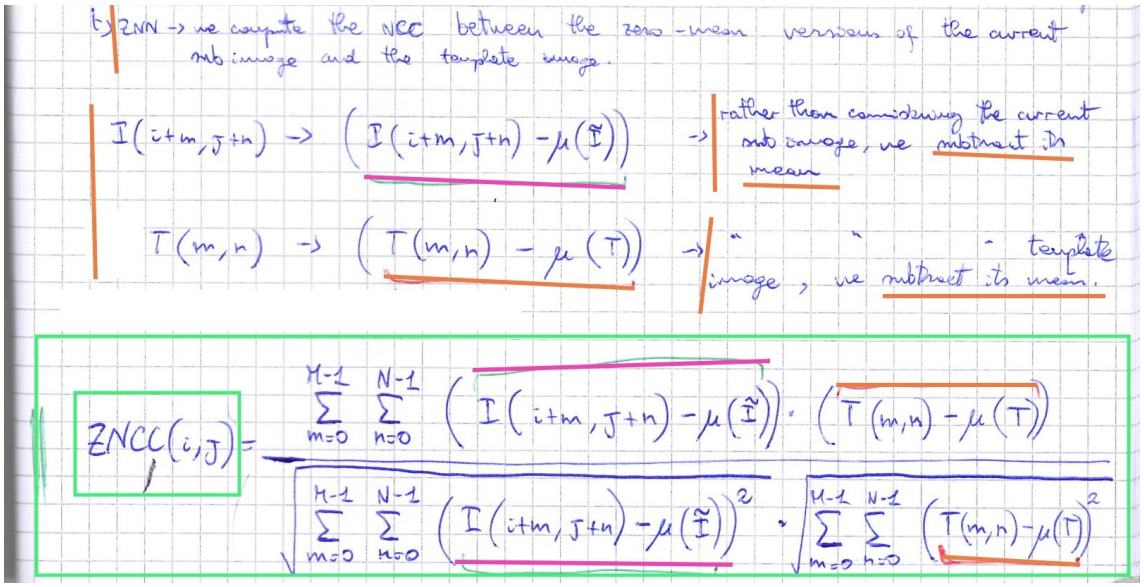


Figure 10.15

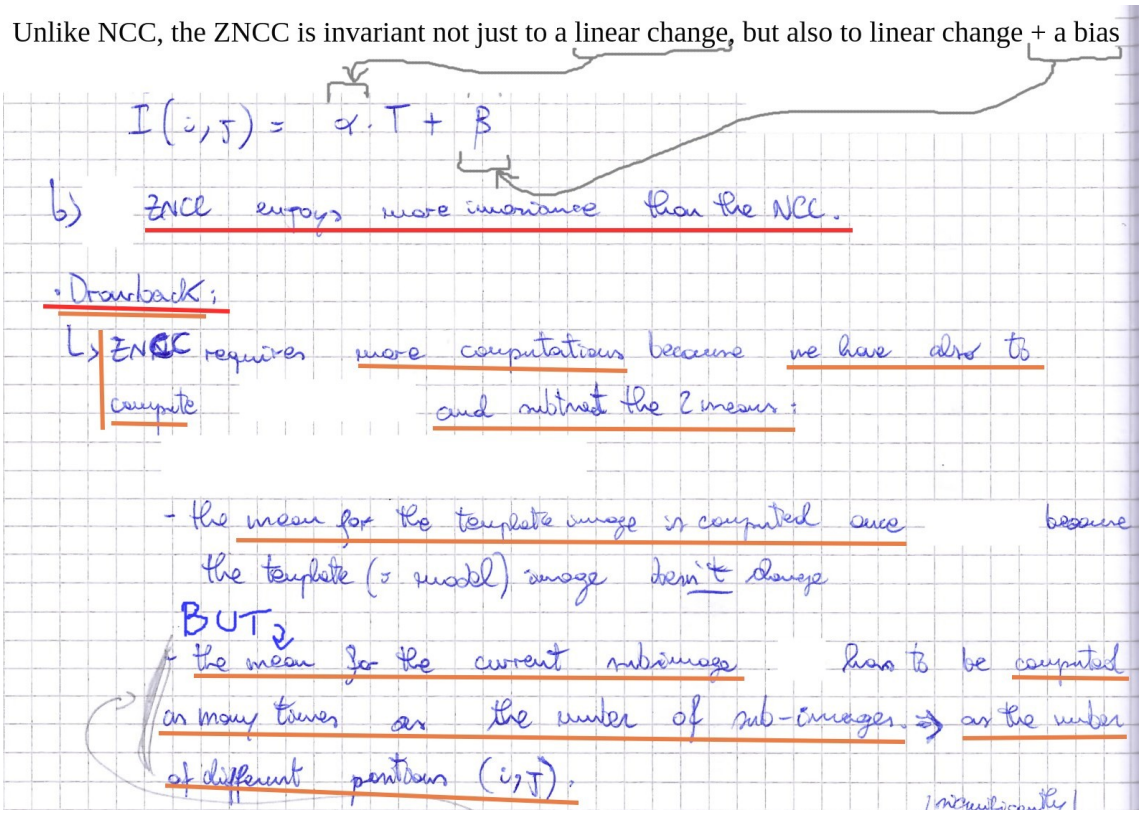


Figure 10.16

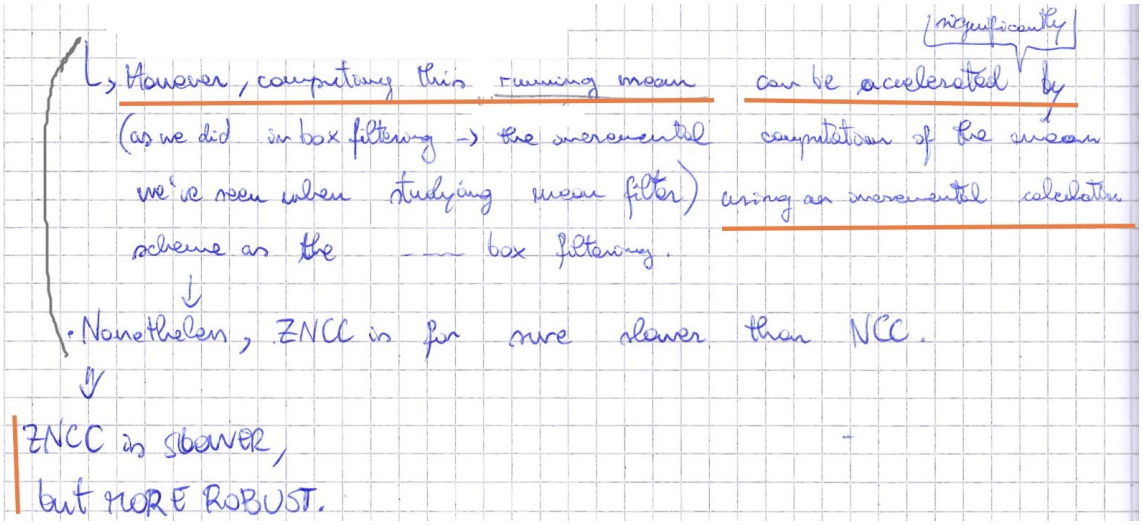


Figure 10.17

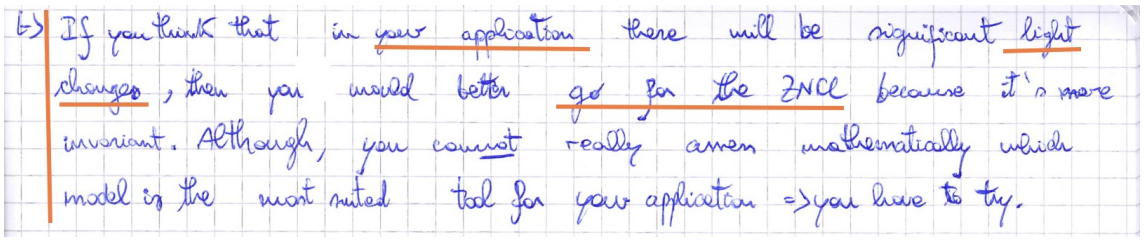


Figure 10.18

10.1.5 Fast Template Matching

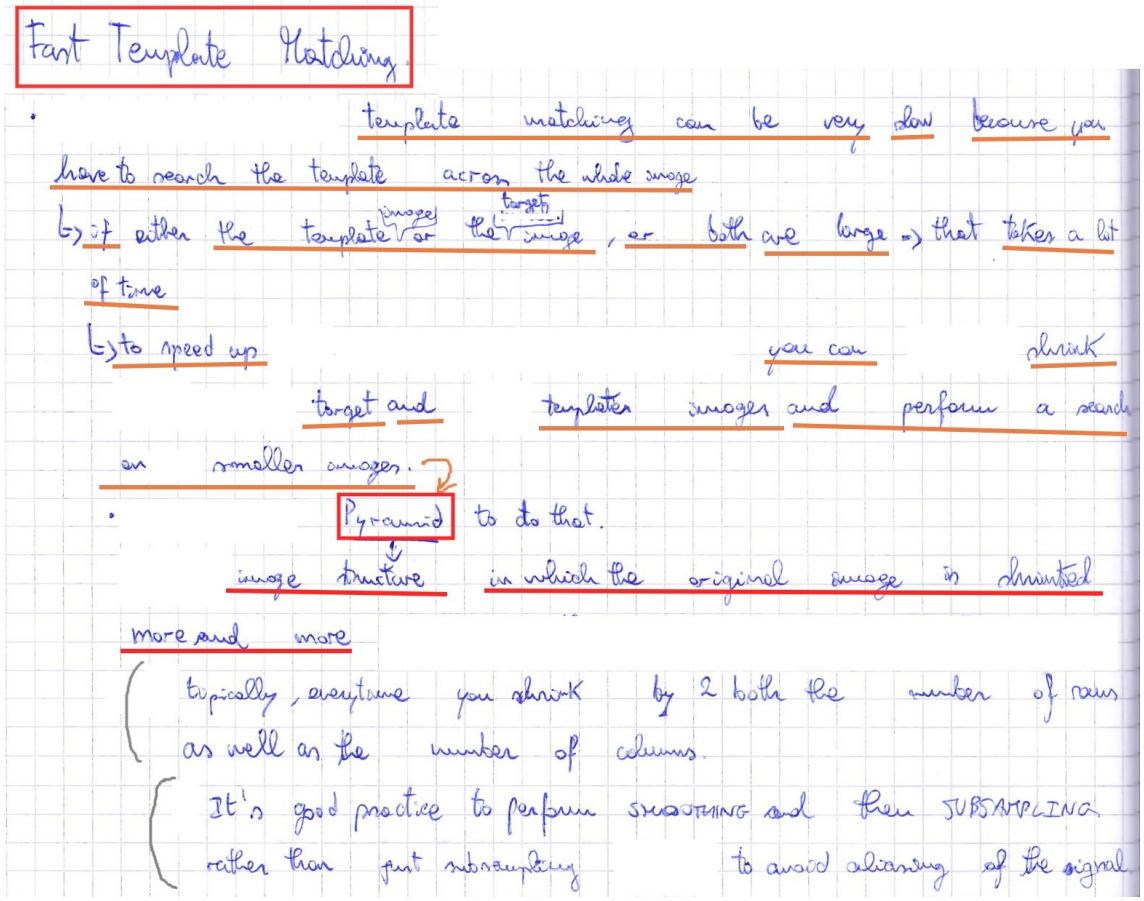


Figure 10.19

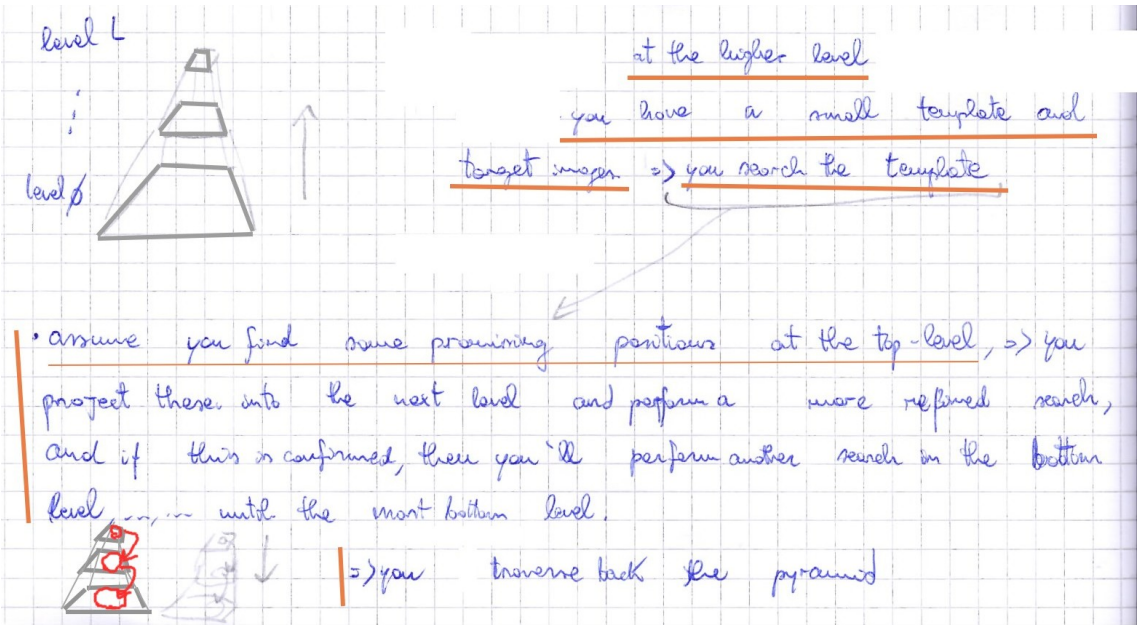


Figure 10.20

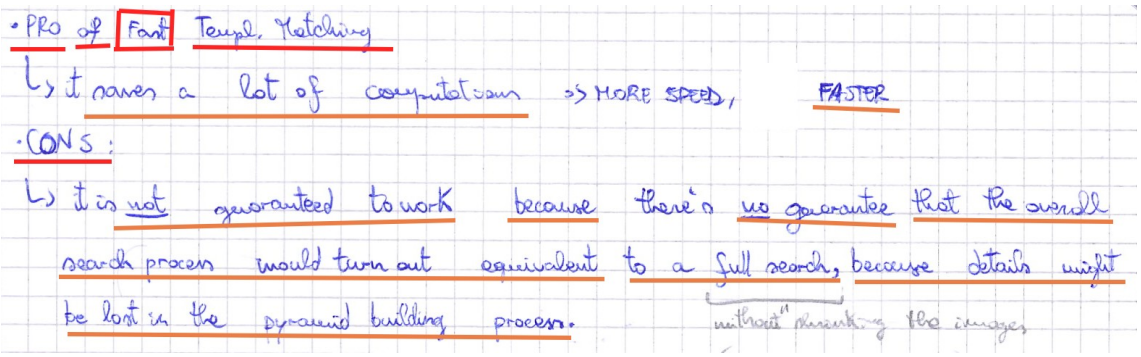


Figure 10.21

10.2 Shape-based matching algorithm

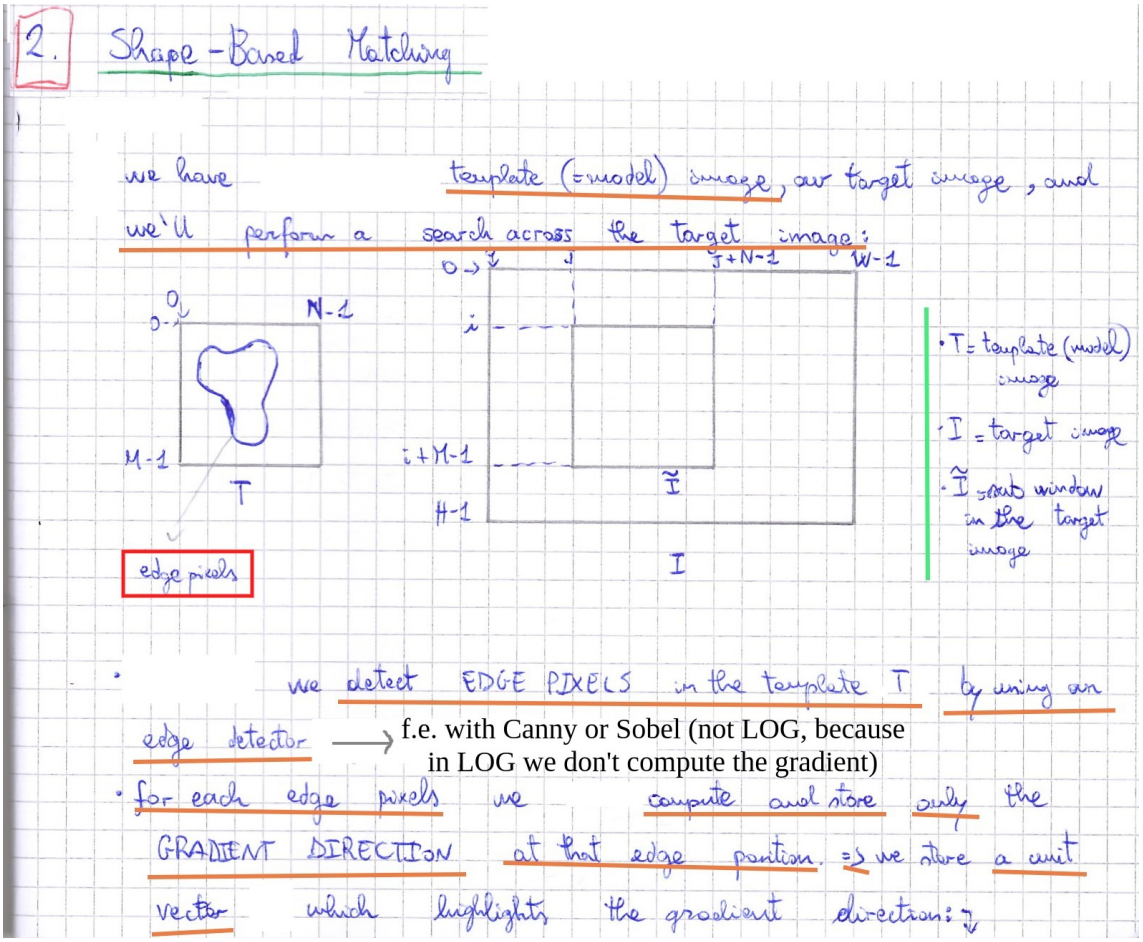


Figure 10.22

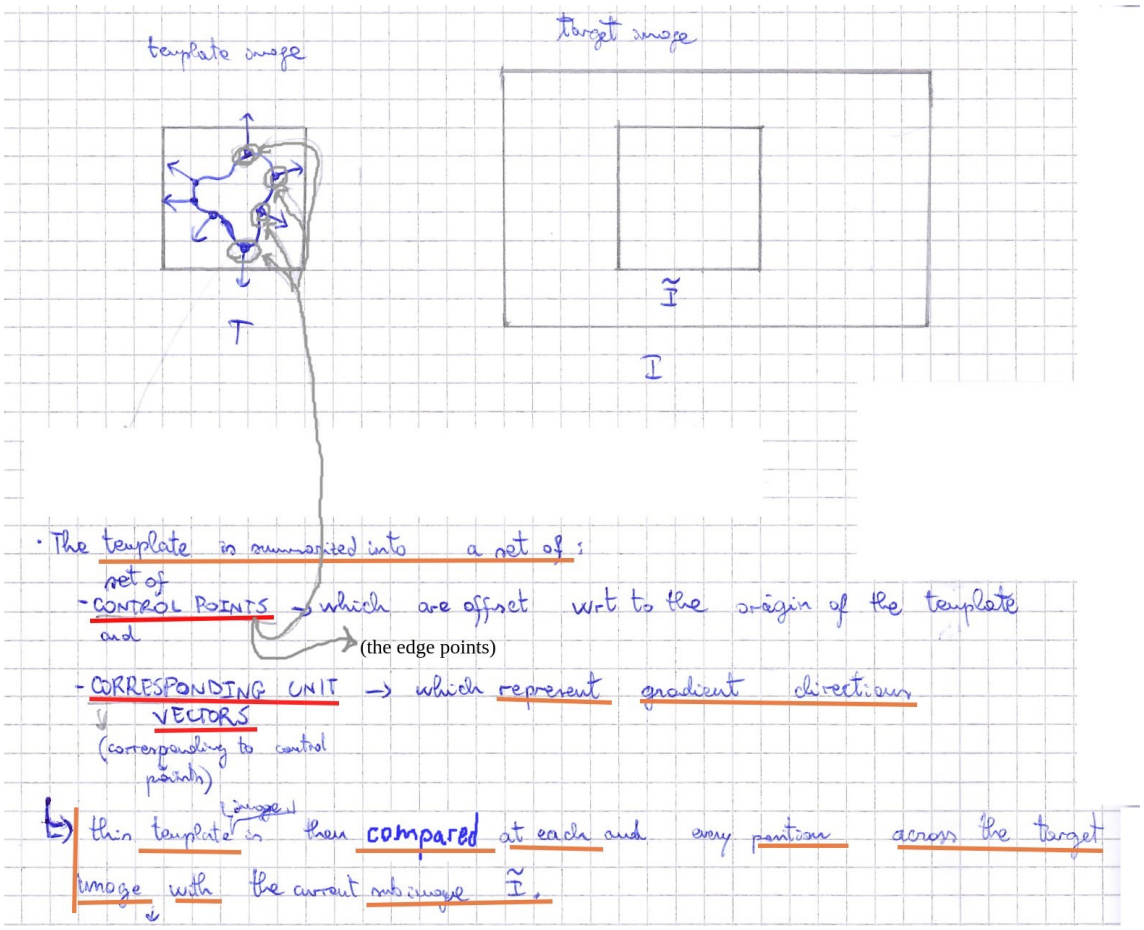

template image
target image
T
Ĩ
I
• The template is summarized into a set of :
- CONTROL POINTS → which are offset w.r.t to the origin of the template
(the edge points)
- CORRESPONDING UNIT VECTORS → which represent gradient directions
(corresponding to control points)
b) this template is then compared at each and every portion across the target image with the current sub image Ĩ.

Figure 10.23

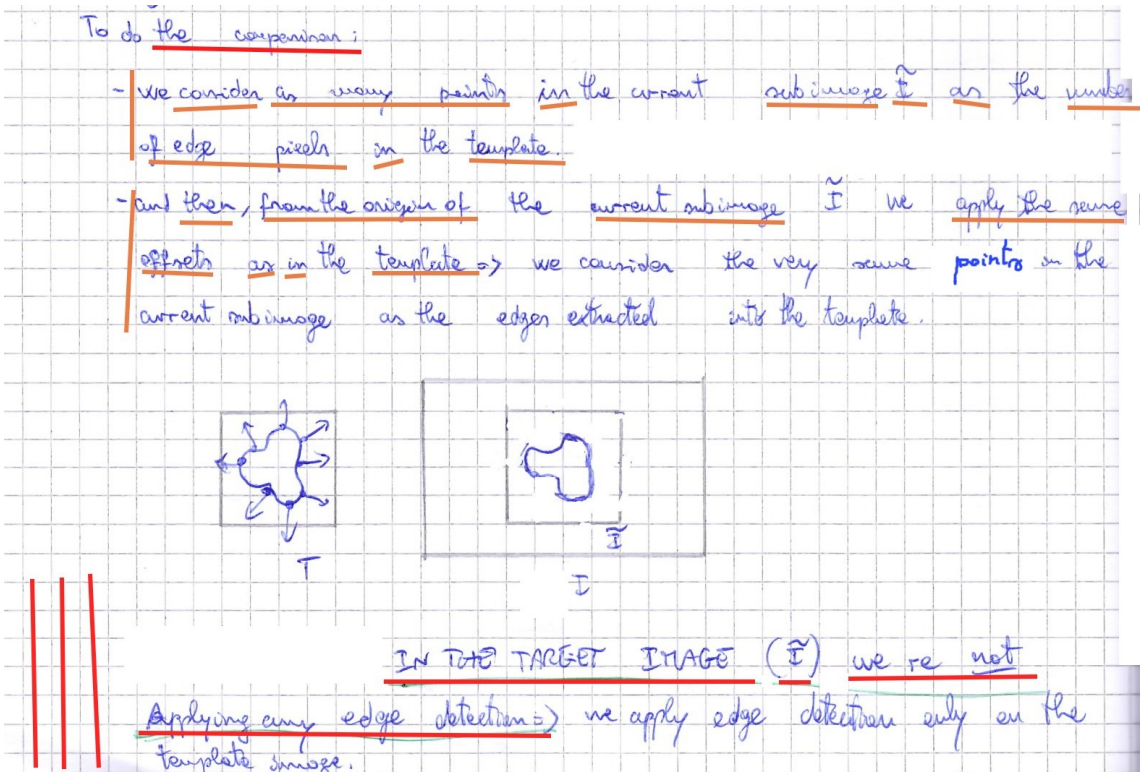


Figure 10.24

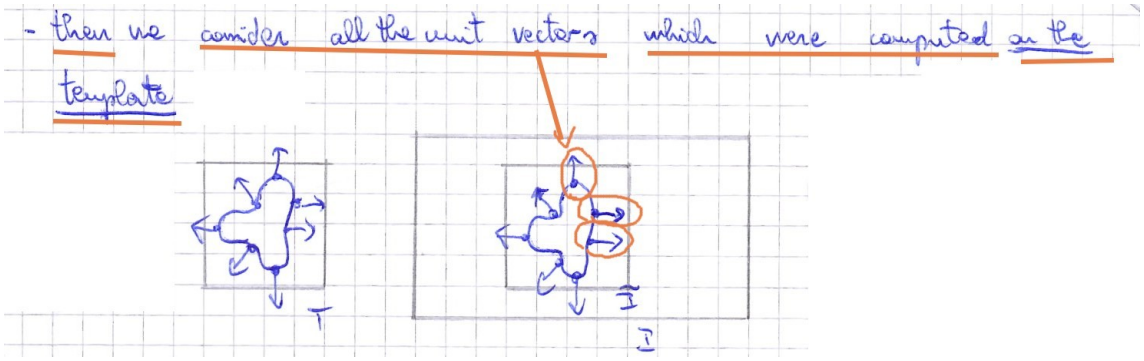


Figure 10.25

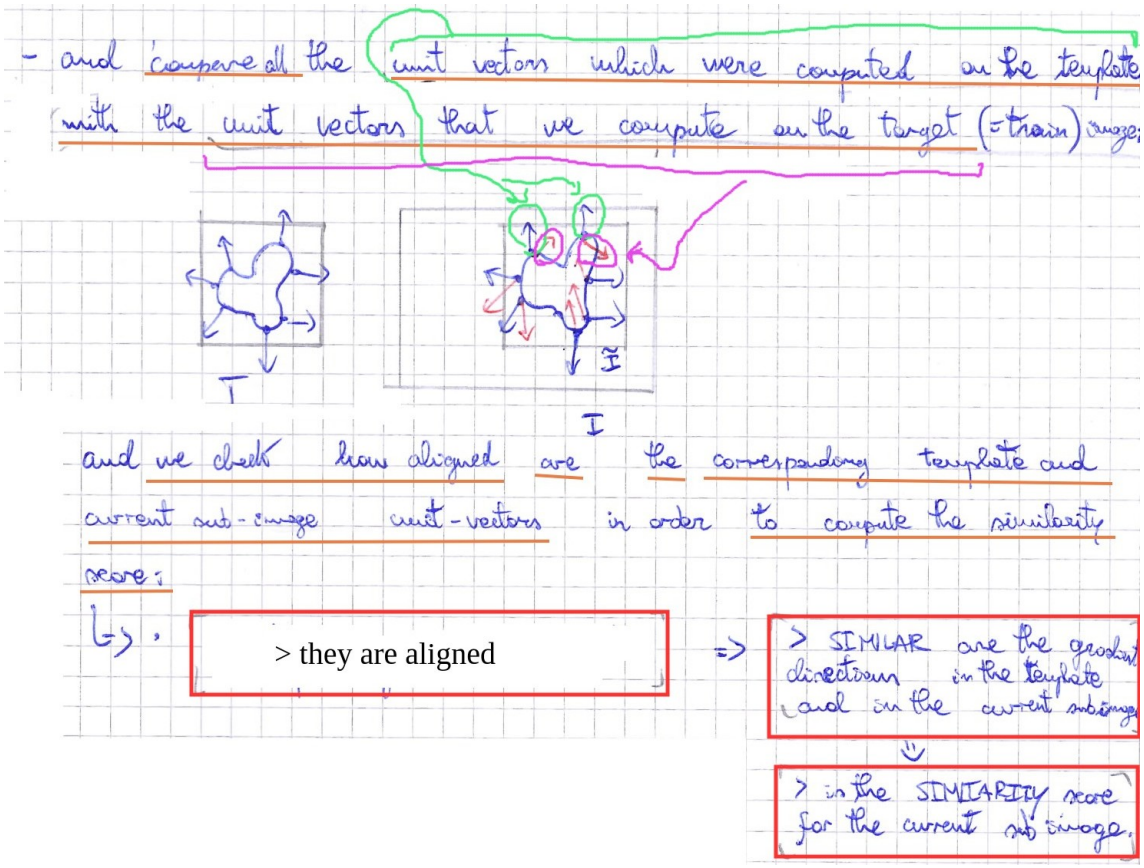

- and compare all the unit vectors which were computed on the template
with the unit vectors that we compute on the target (=train) image;
T
I
and we check how aligned are the corresponding template and
current sub-image unit-vectors in order to compute the similarity
score:
↳ .
> they are aligned
=>
> SIMILAR are the gradient directions in the template and in the current subimage
> in the SIMILARITY score for the current sub image.

Figure 10.26

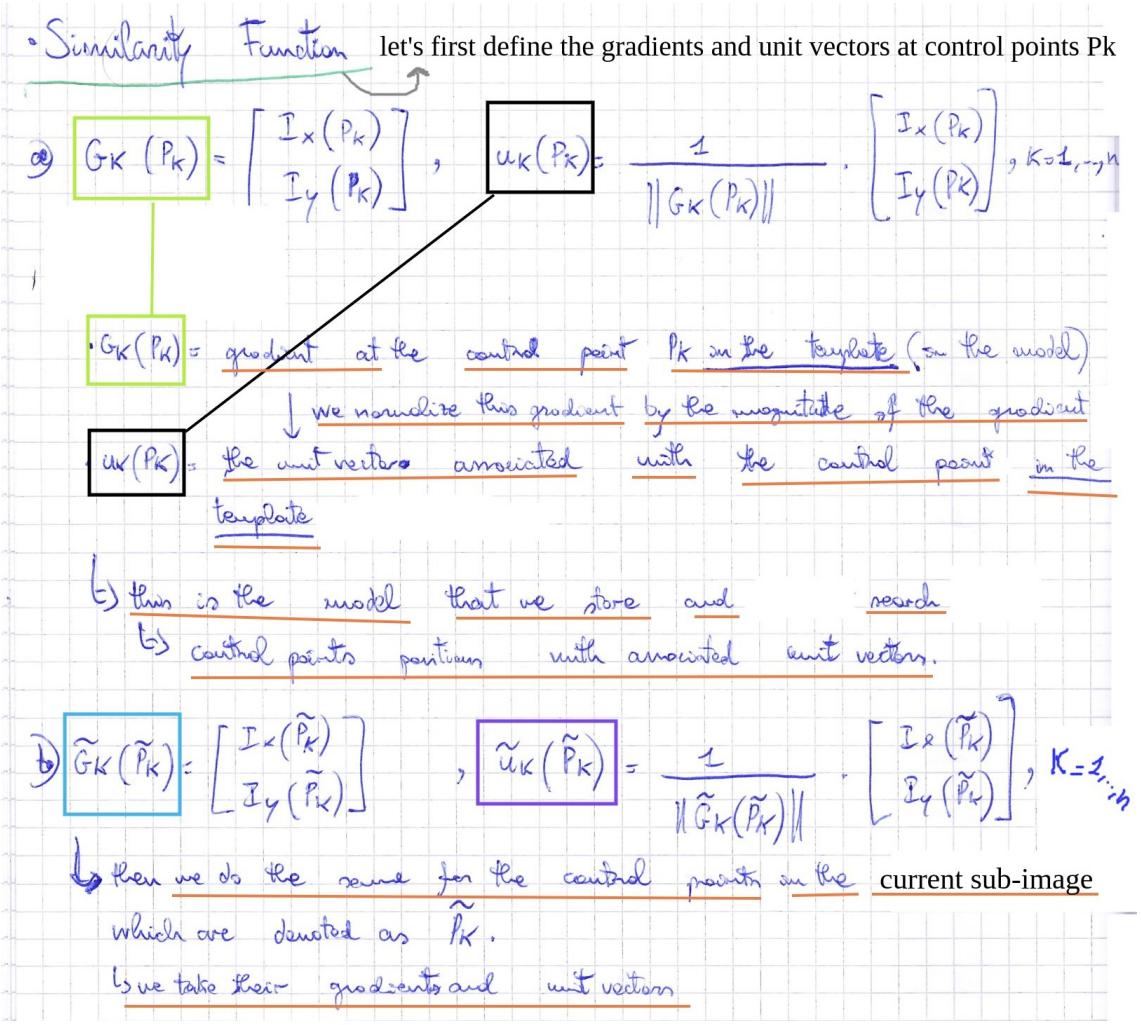


Figure 10.27

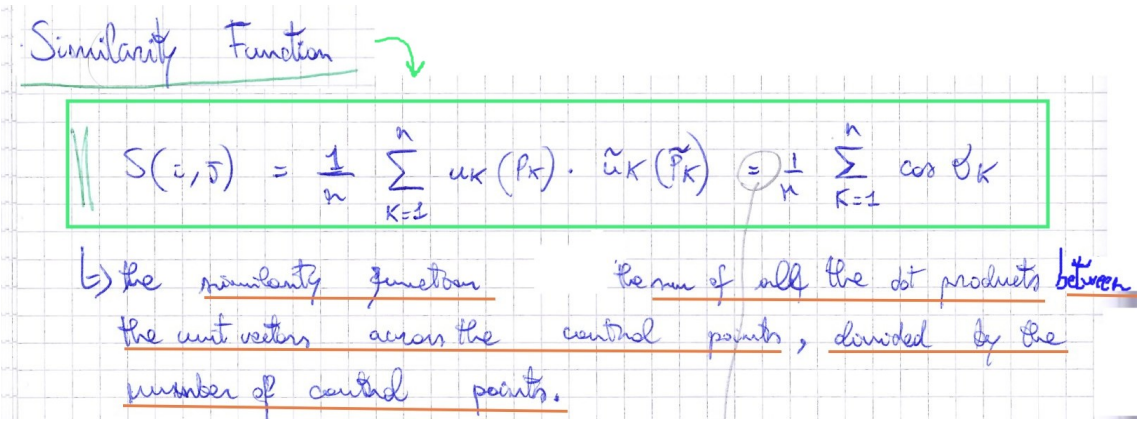


Figure 10.28

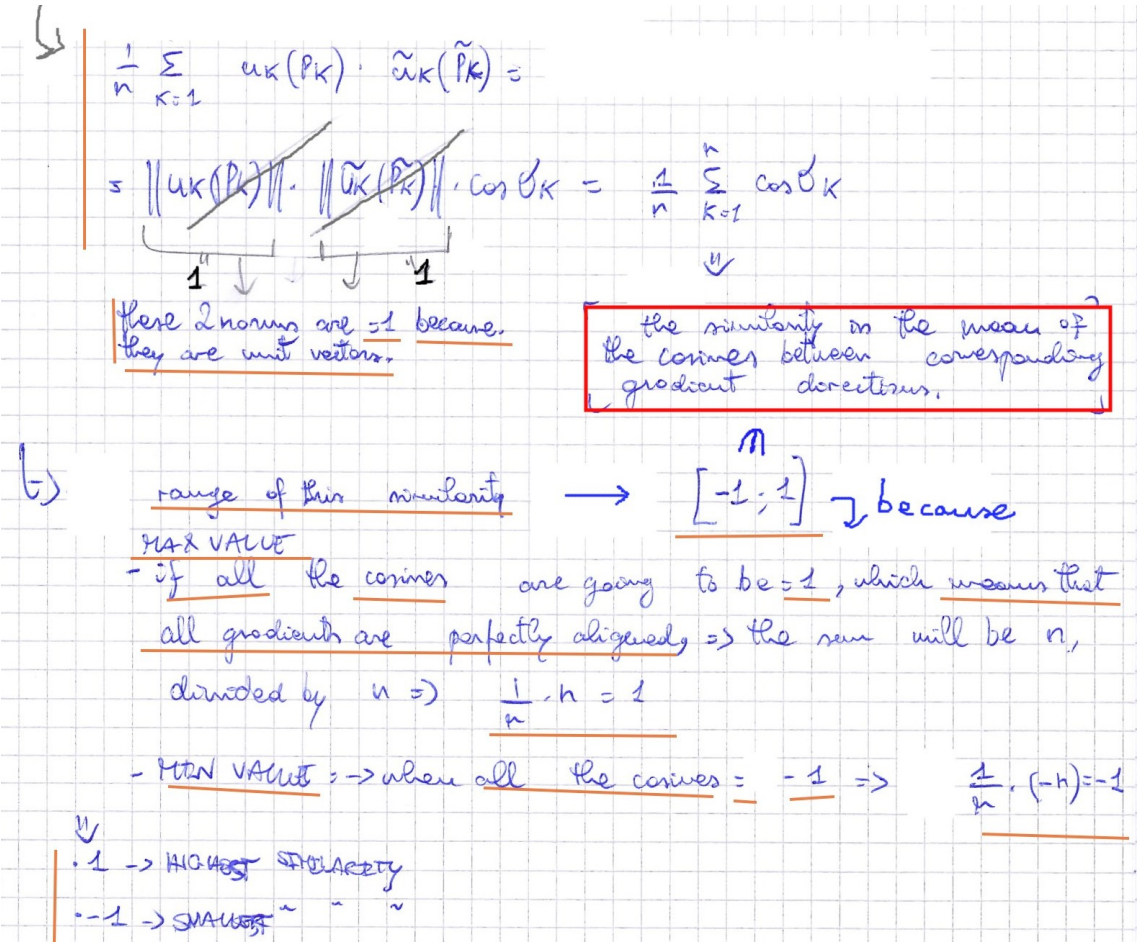


Figure 10.29

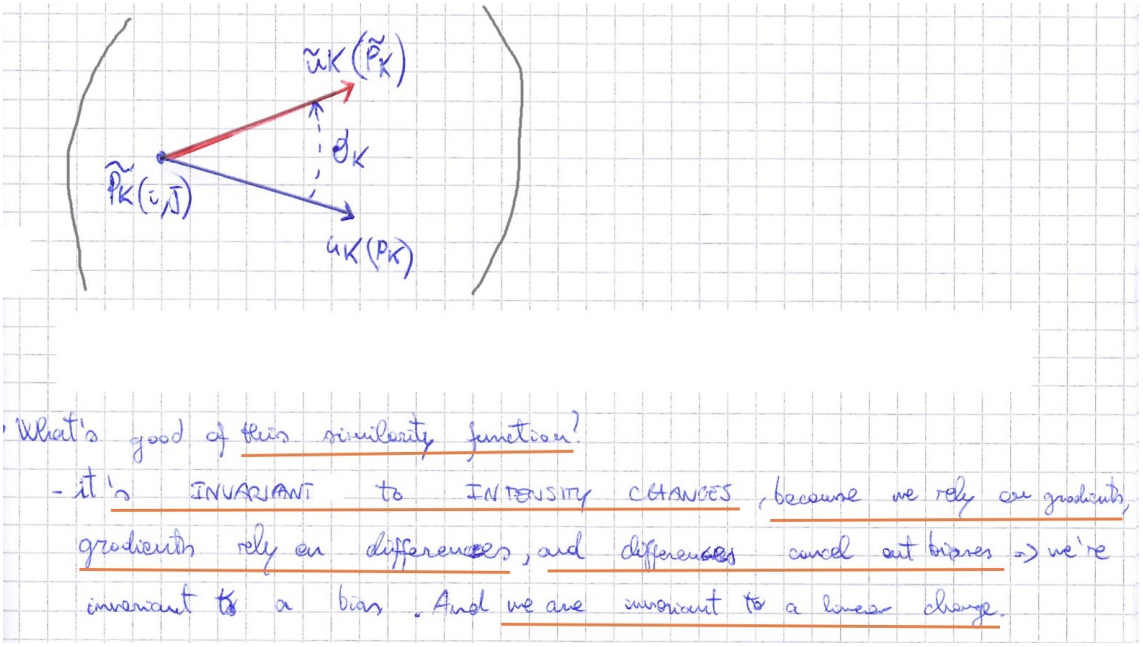


Figure 10.30

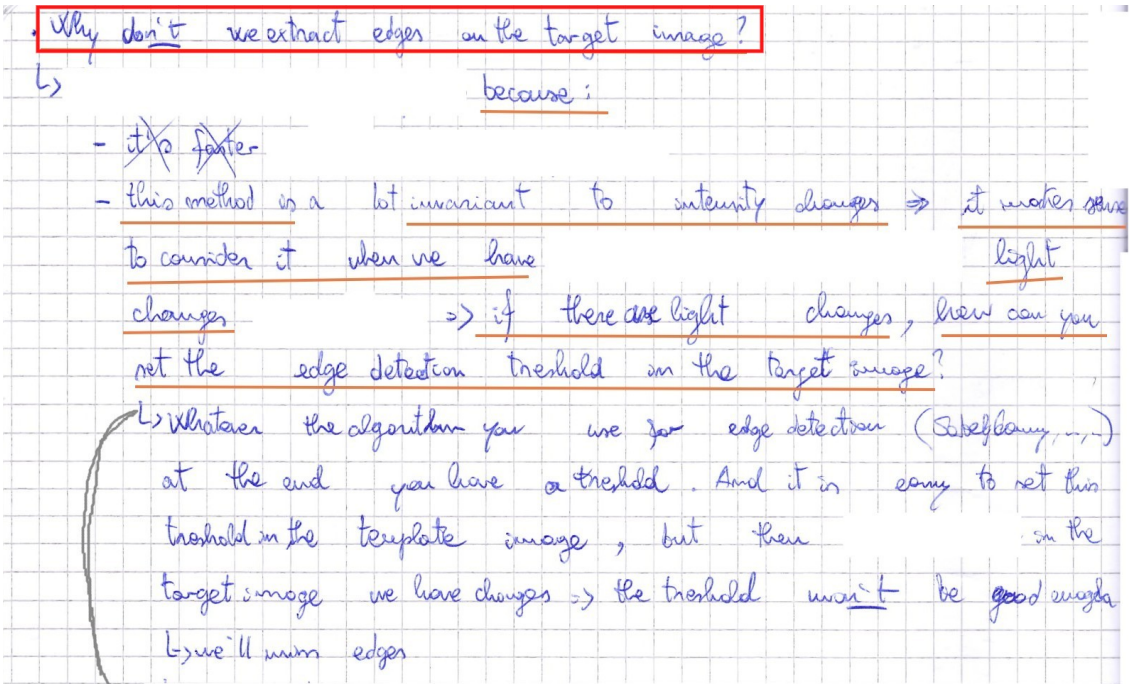


Figure 10.31

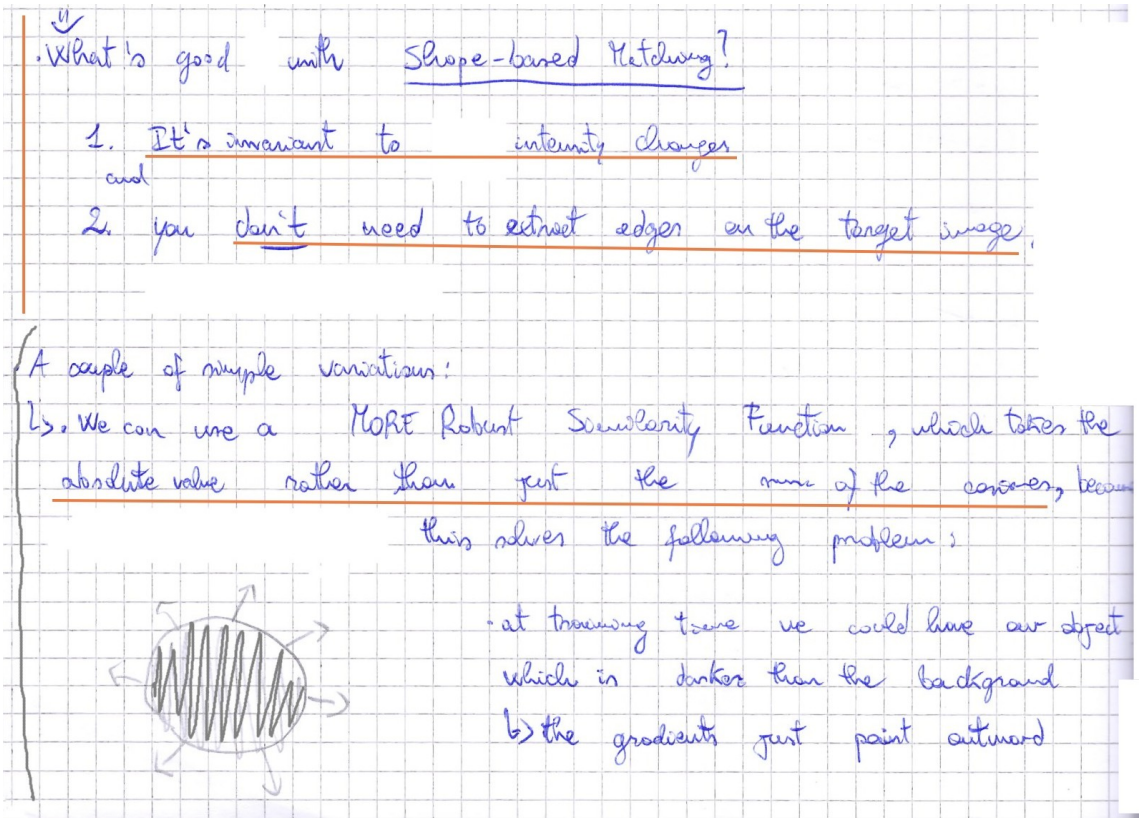


Figure 10.32

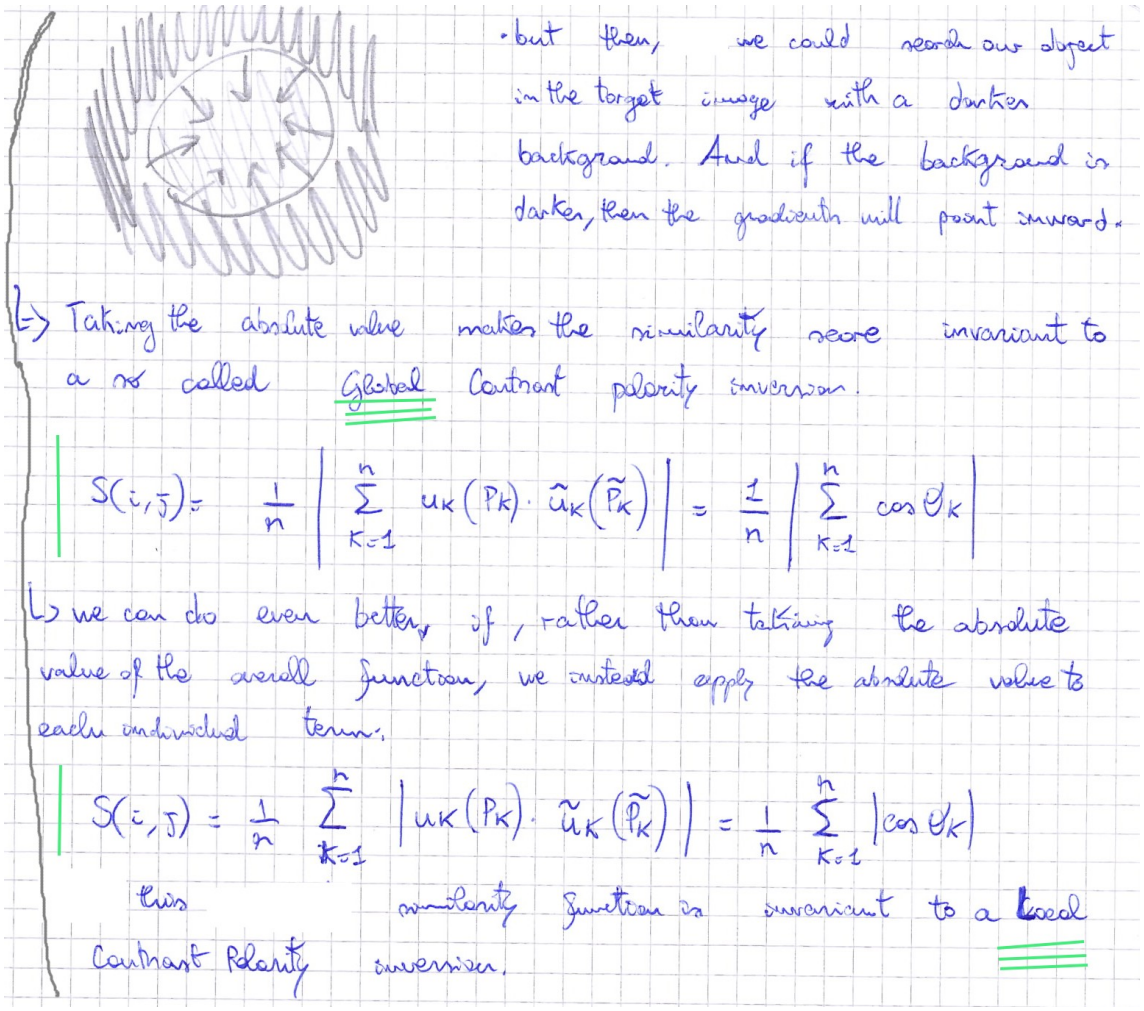

• but then, we could search our object in the target image with a darker background. And if the background is darker, then the gradients will point inward.
↳ Taking the absolute value makes the similarity score invariant to a so called Global Contrast polarity inversion.
$S(i,j) = \frac{1}{n}\left|\sum_{k=1}^{n} u_k(p_k)\cdot\tilde{u}_k(\tilde{p}_k)\right| = \frac{1}{n}\left|\sum_{k=1}^{n}\cos\vartheta_k\right|$
↳ 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:
$S(i,j) = \frac{1}{n}\sum_{k=1}^{n}\left|u_k(p_k)\cdot\tilde{u}_k(\tilde{p}_k)\right| = \frac{1}{n}\sum_{k=1}^{n}\left|\cos\vartheta_k\right|$
this similarity function is invariant to a local Contrast Polarity inversion.

Figure 10.33

10.3 GHT

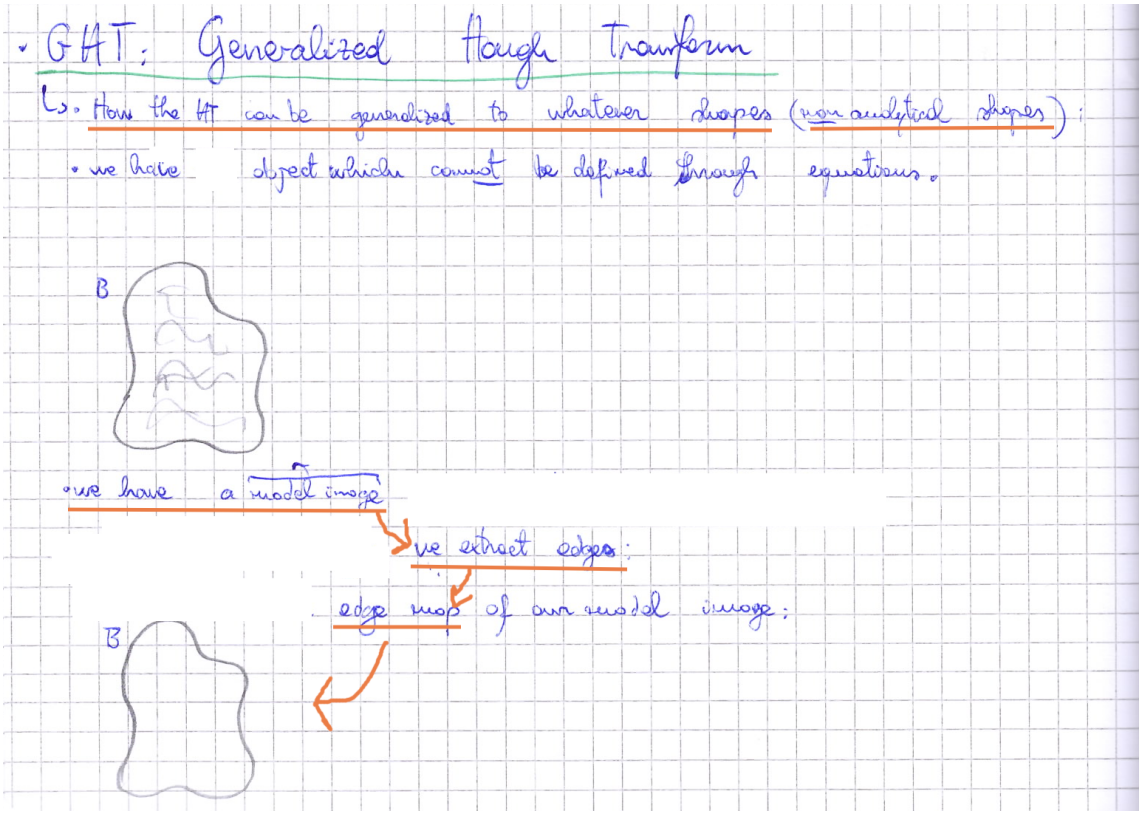


Figure 10.34

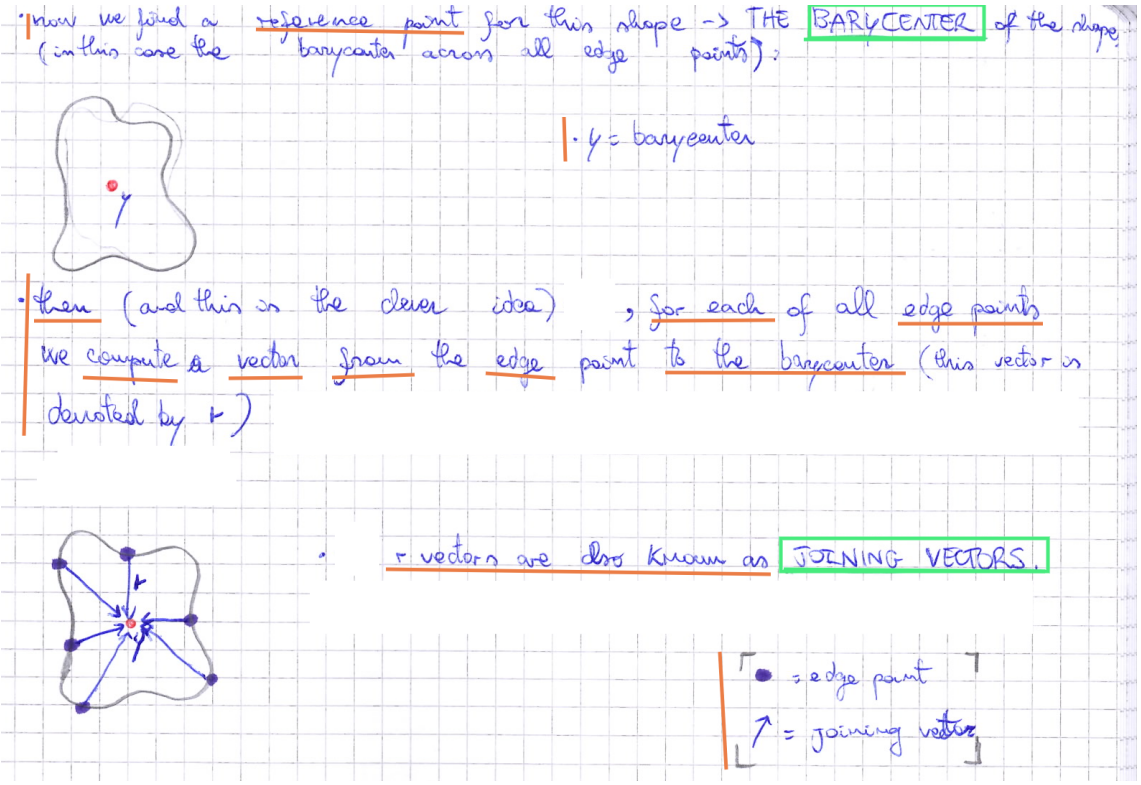


Figure 10.35

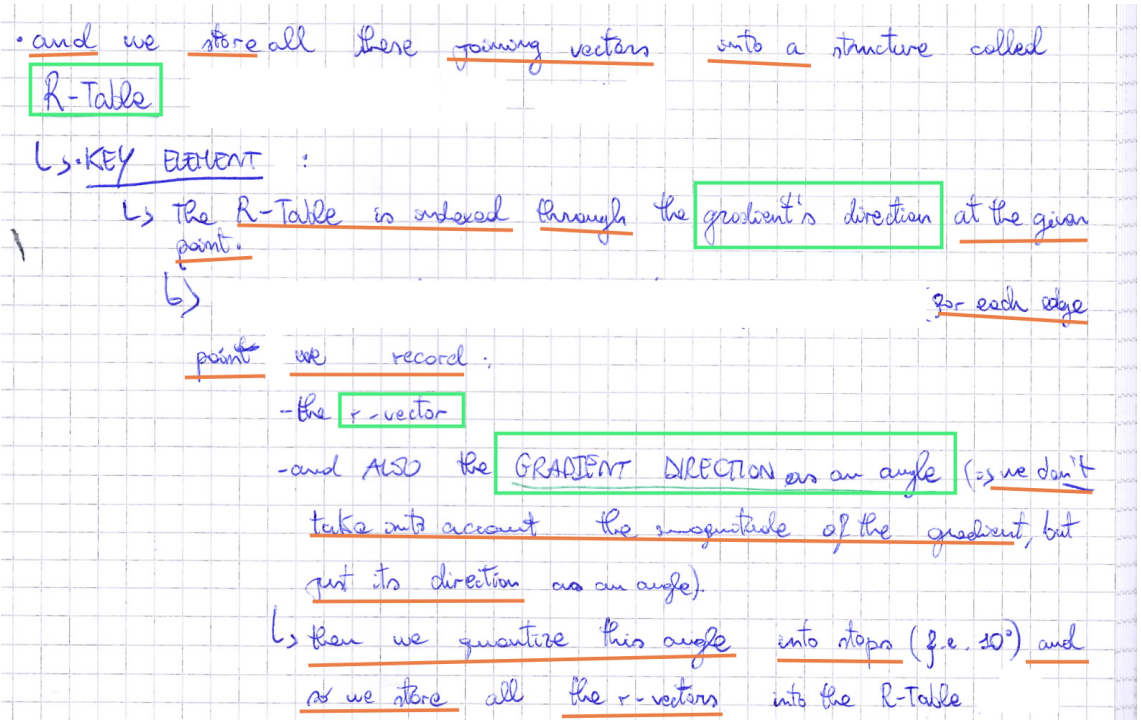


Figure 10.36

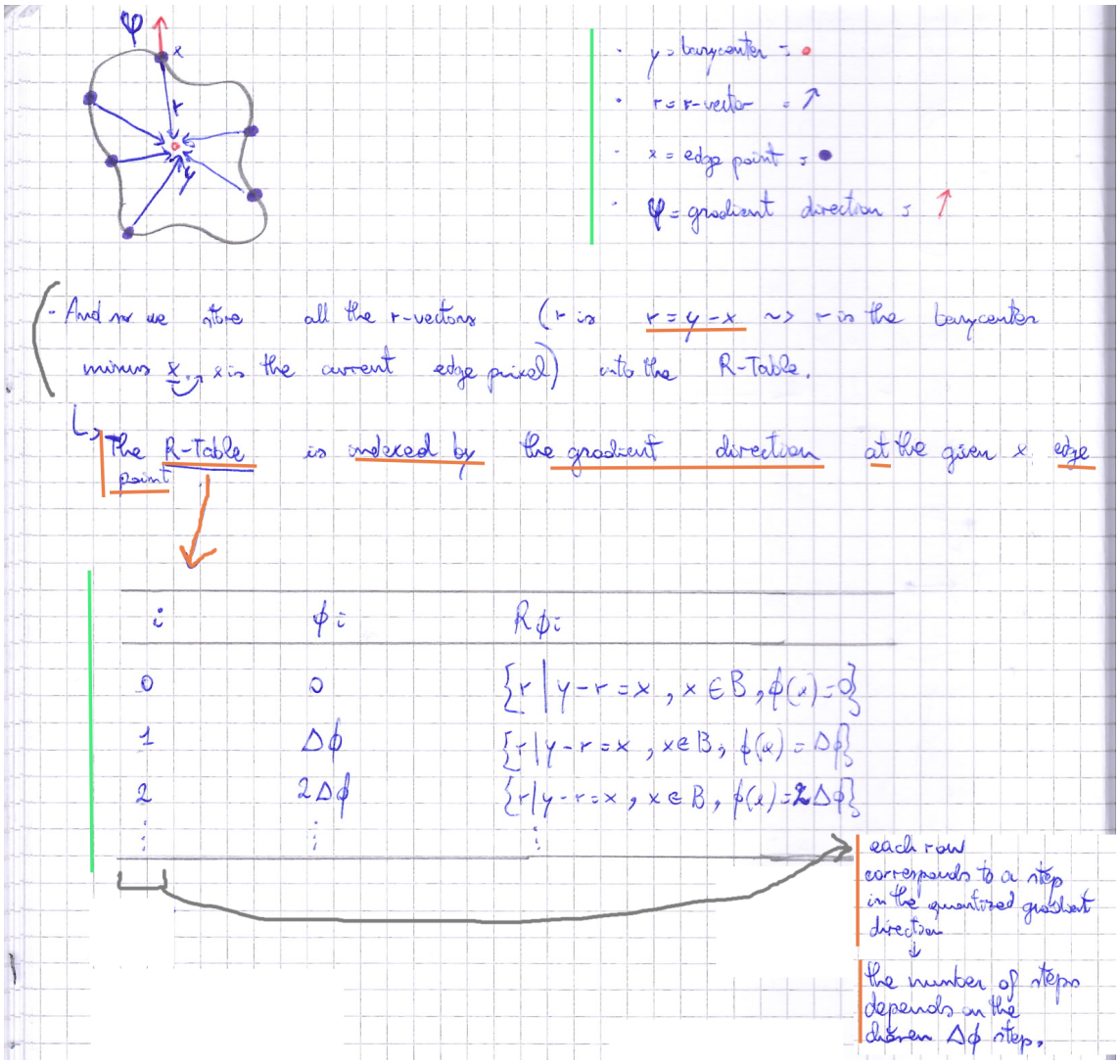


Figure 10.37

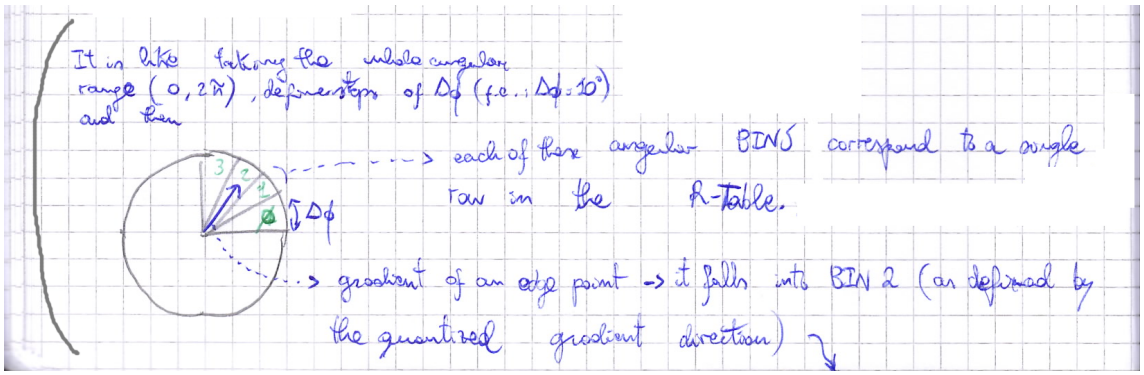


Figure 10.38

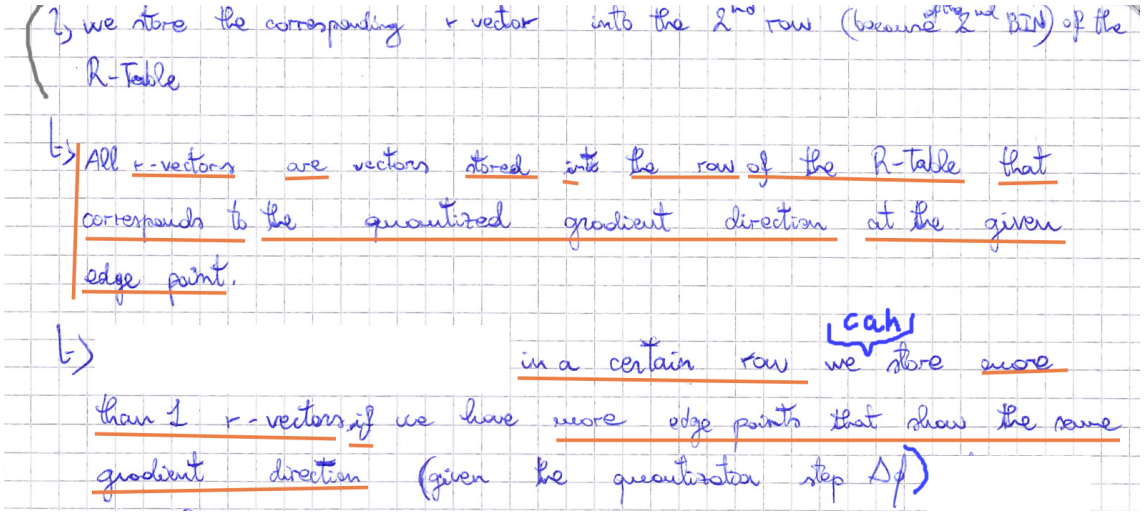


Figure 10.39

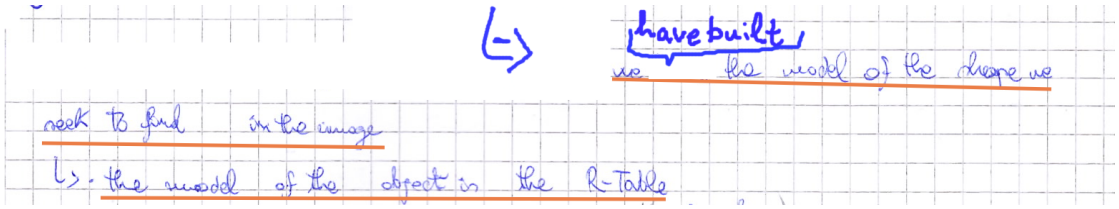


Figure 10.40

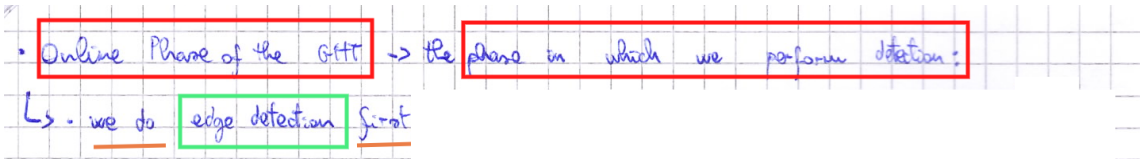


Figure 10.41

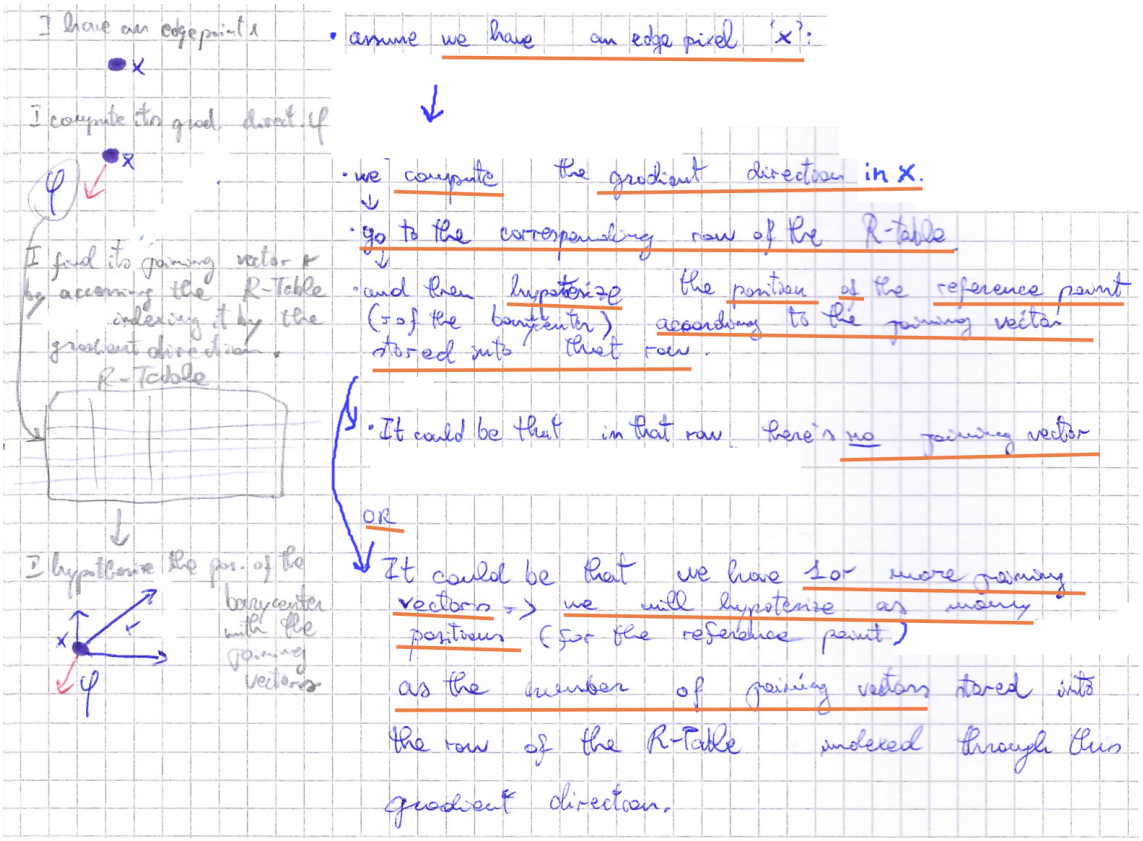


Figure 10.42

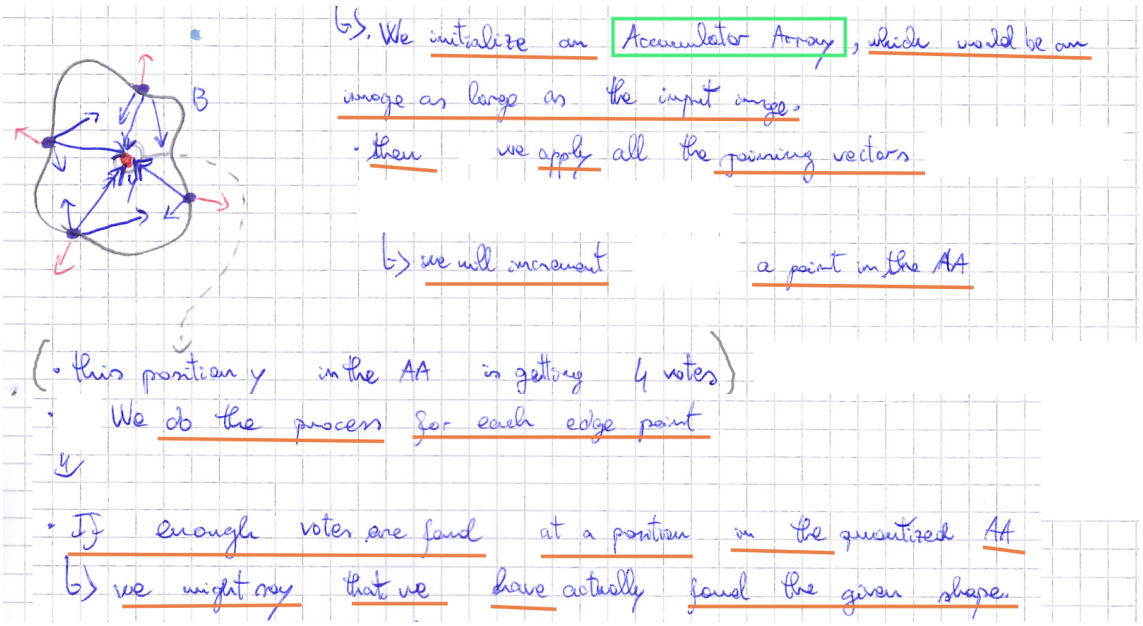


Figure 10.43

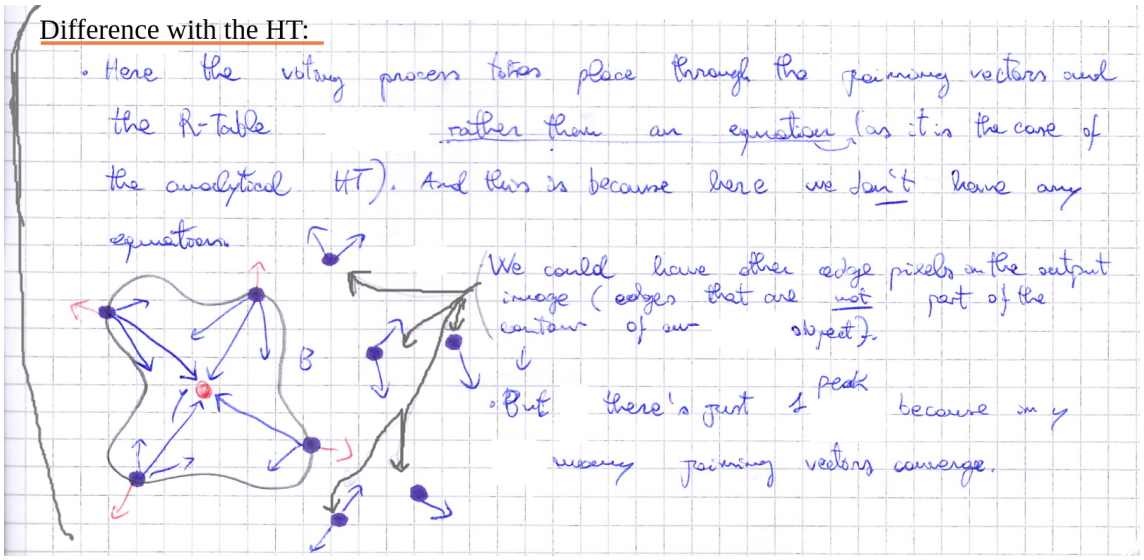


Figure 10.44

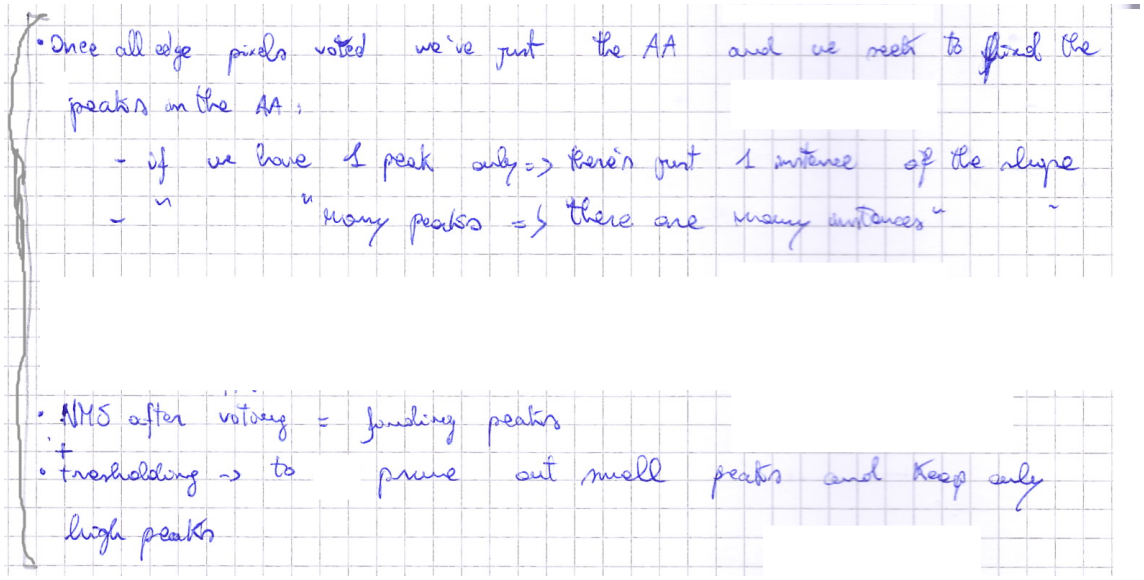


Figure 10.45

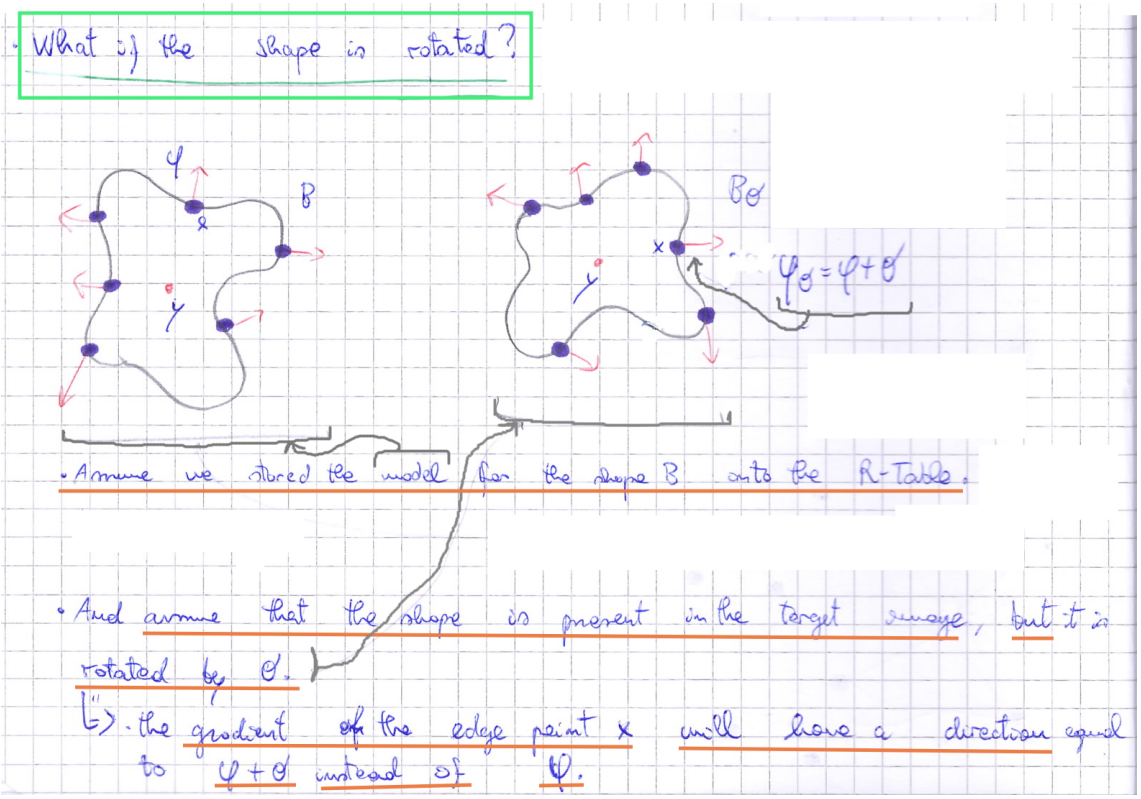


Figure 10.46

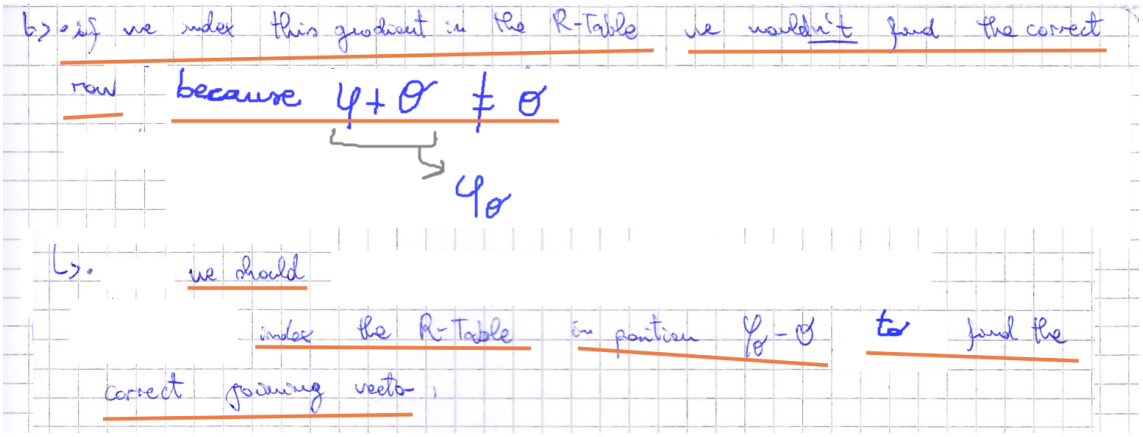


Figure 10.47

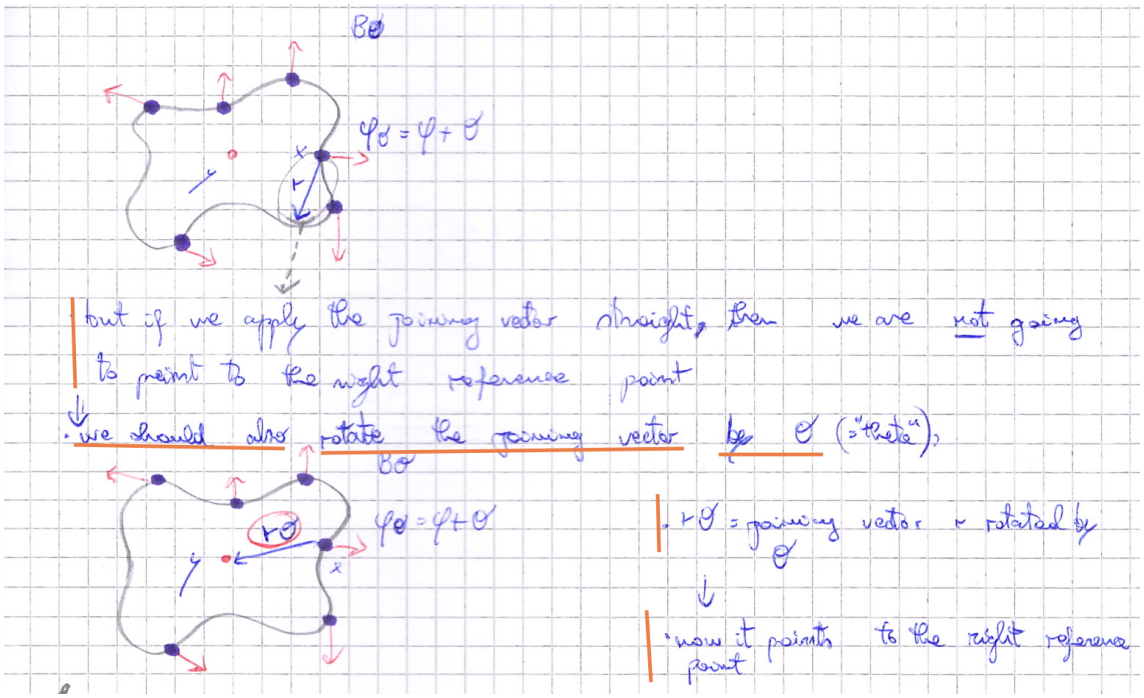


Figure 10.48

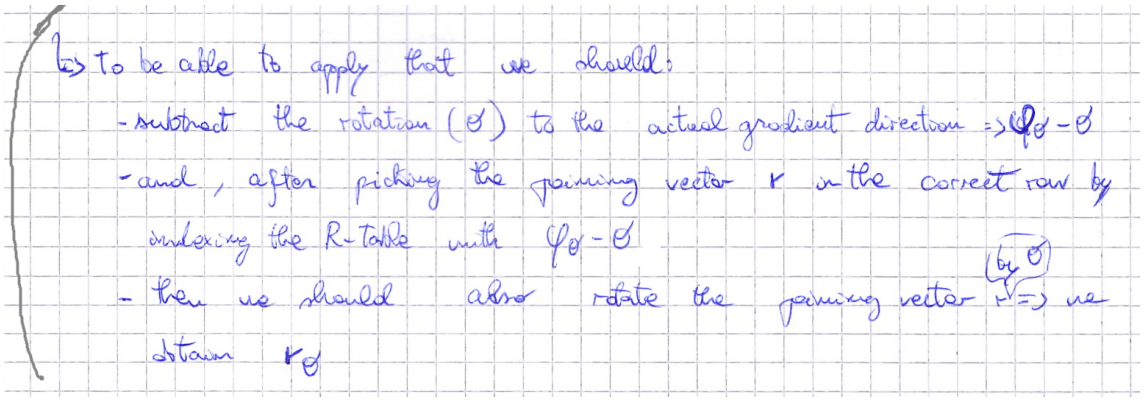


Figure 10.49

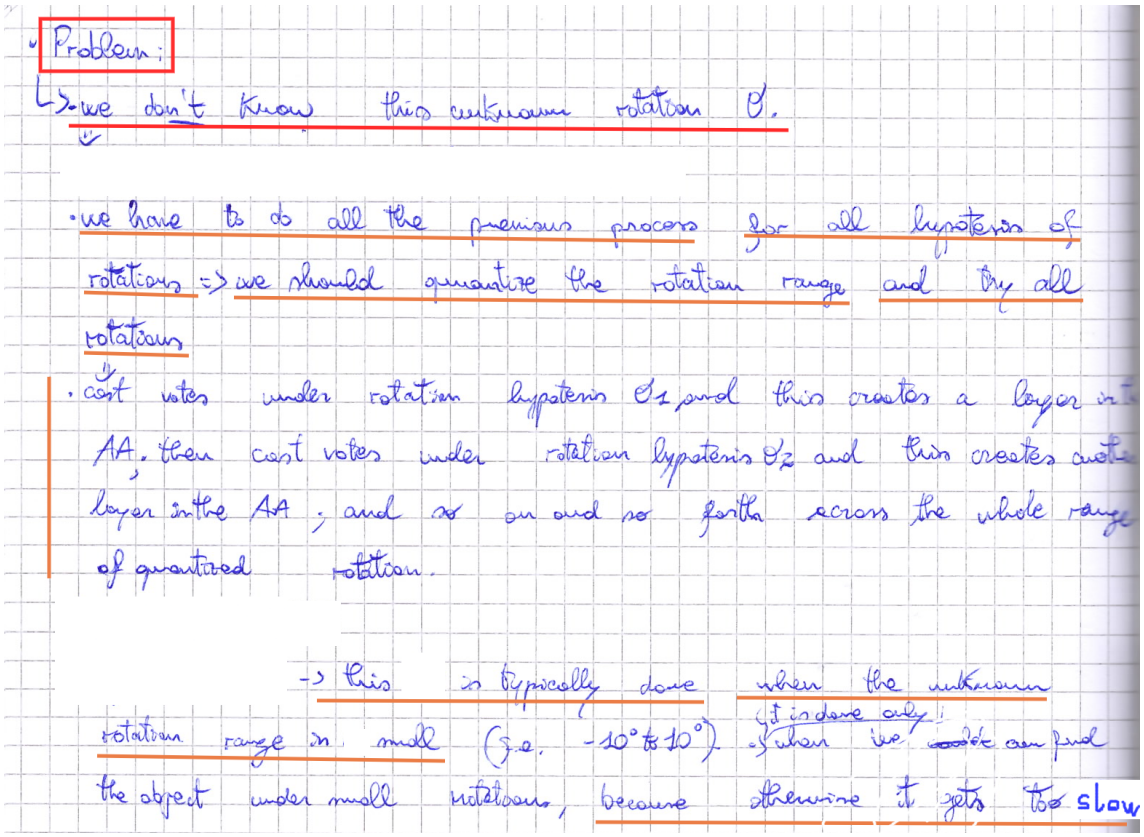

Problem:
we don't know this unknown rotation θ.
we have to do all the previous process for all hypotesis of rotations => we should quantize the rotation range and try all rotations
cast votes under rotation hypotesis θ1 and this creates a layer in AA, then cast votes under rotation hypotesis θ2 and this 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 (i.e. -10° to 10°) (it is done only) when we can find the object under small rotations, because otherwise it gets too slow

Figure 10.50

Figure 10.51

10.3.1 GHT + SIFT

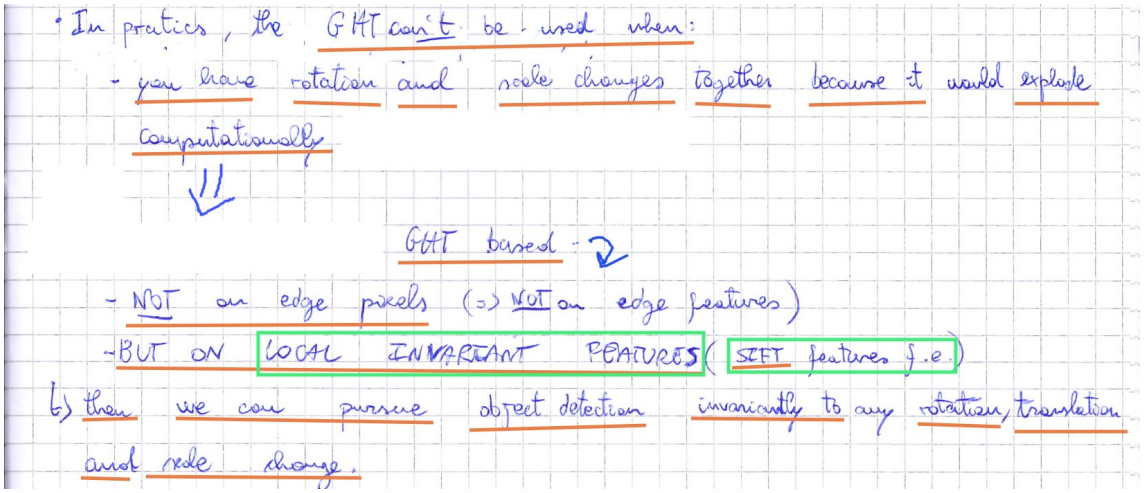


Figure 10.52

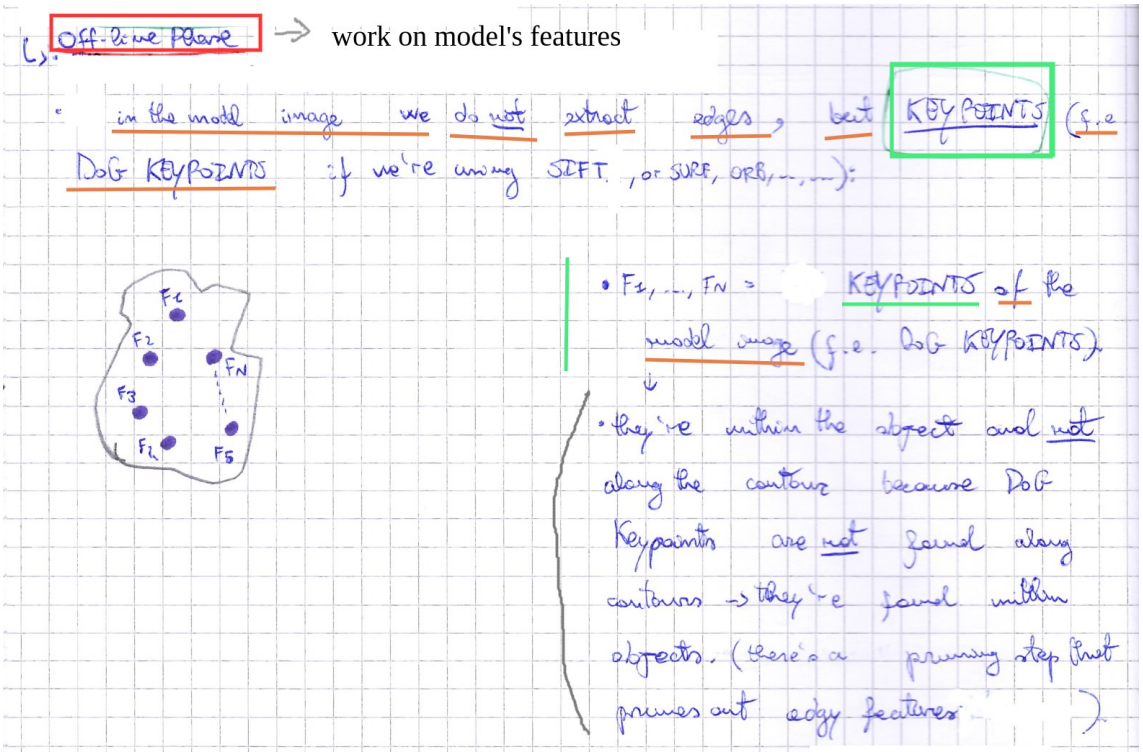


Figure 10.53

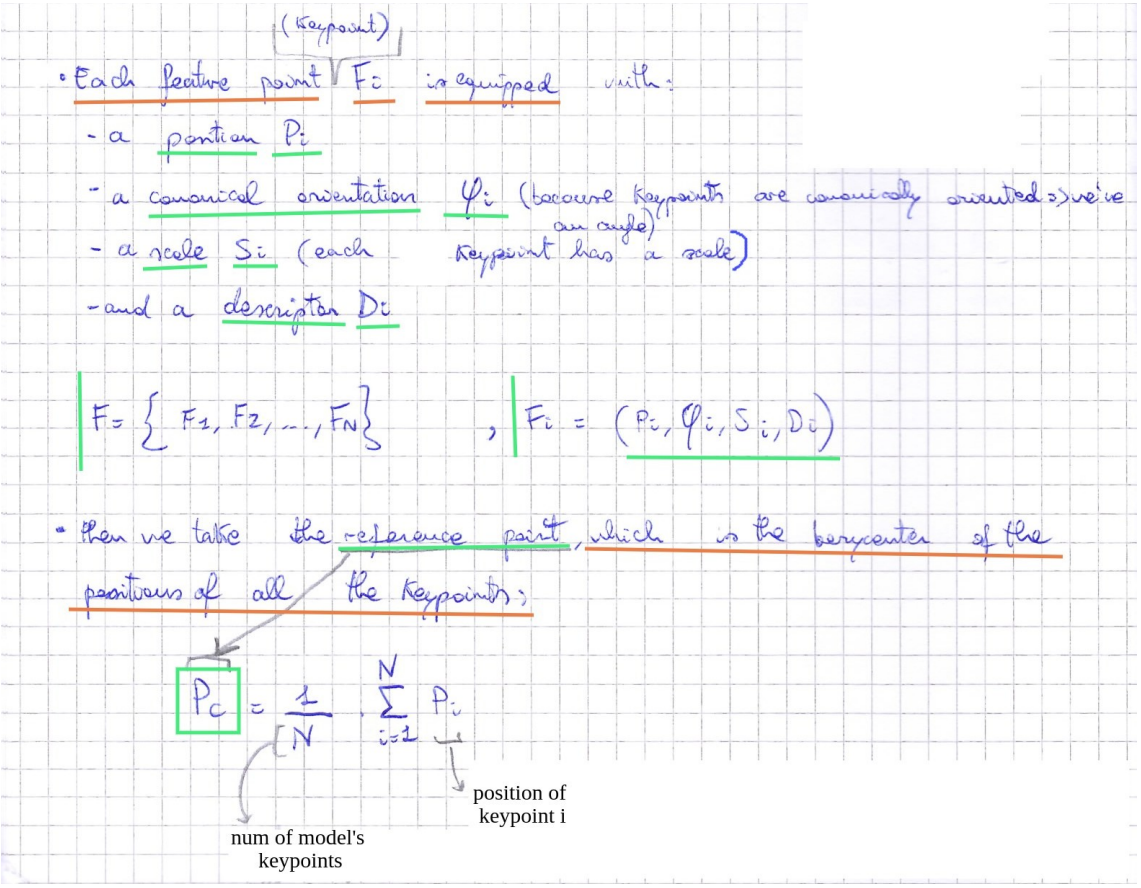


Figure 10.54

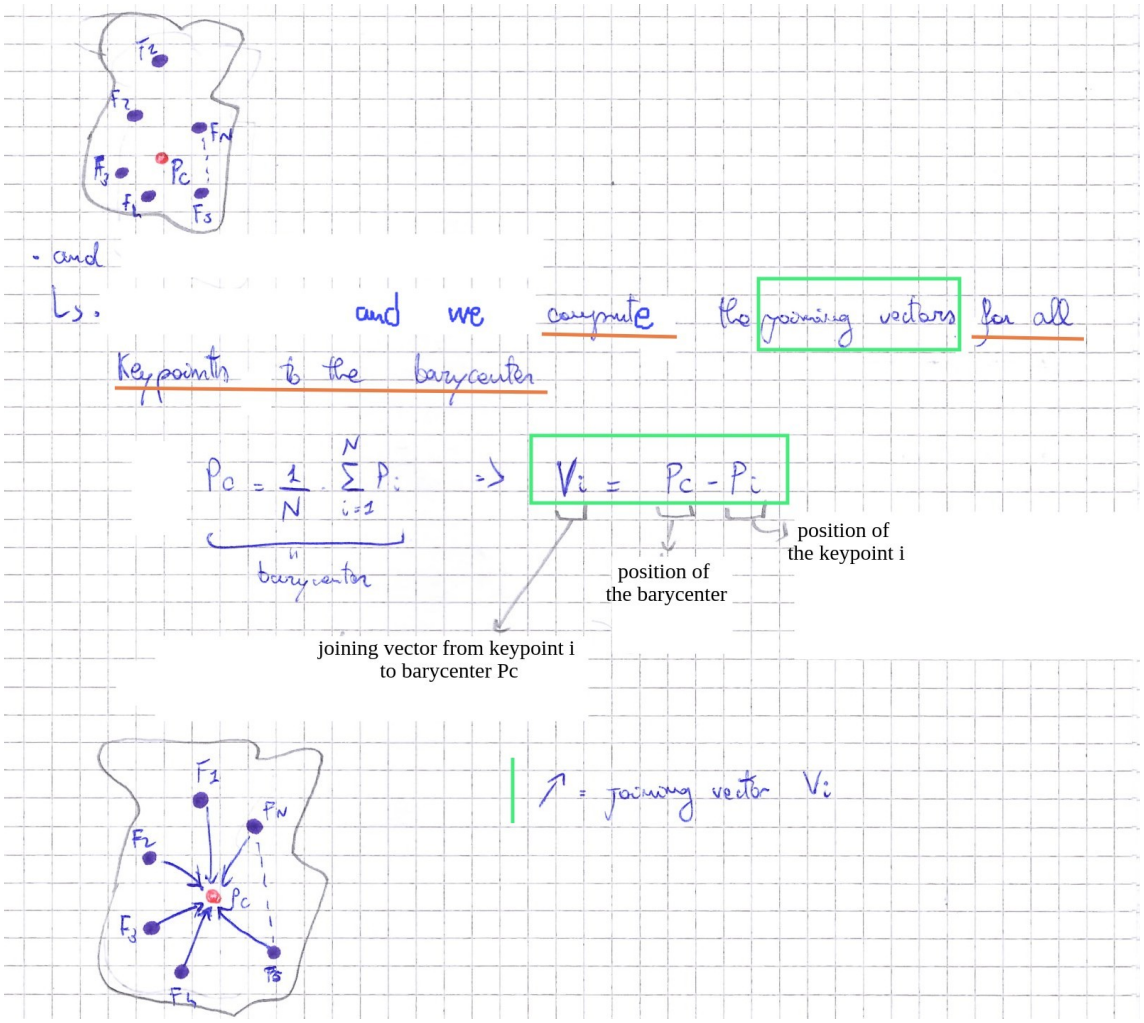


Figure 10.55

Figure 10.56

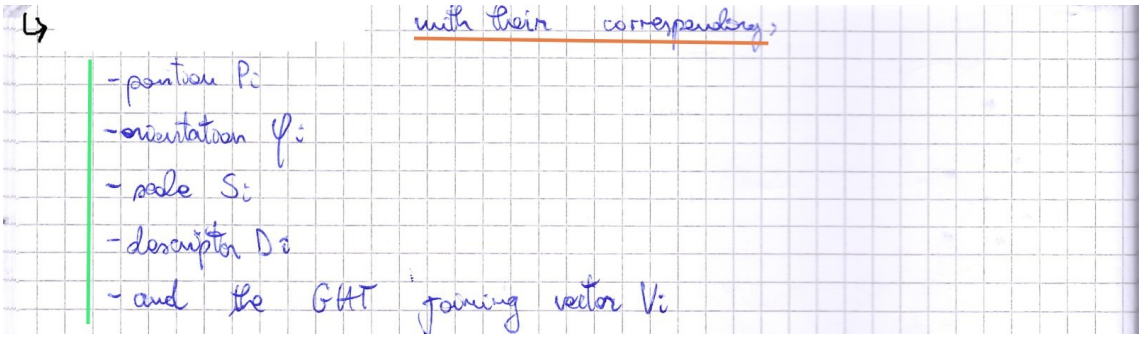


Figure 10.57

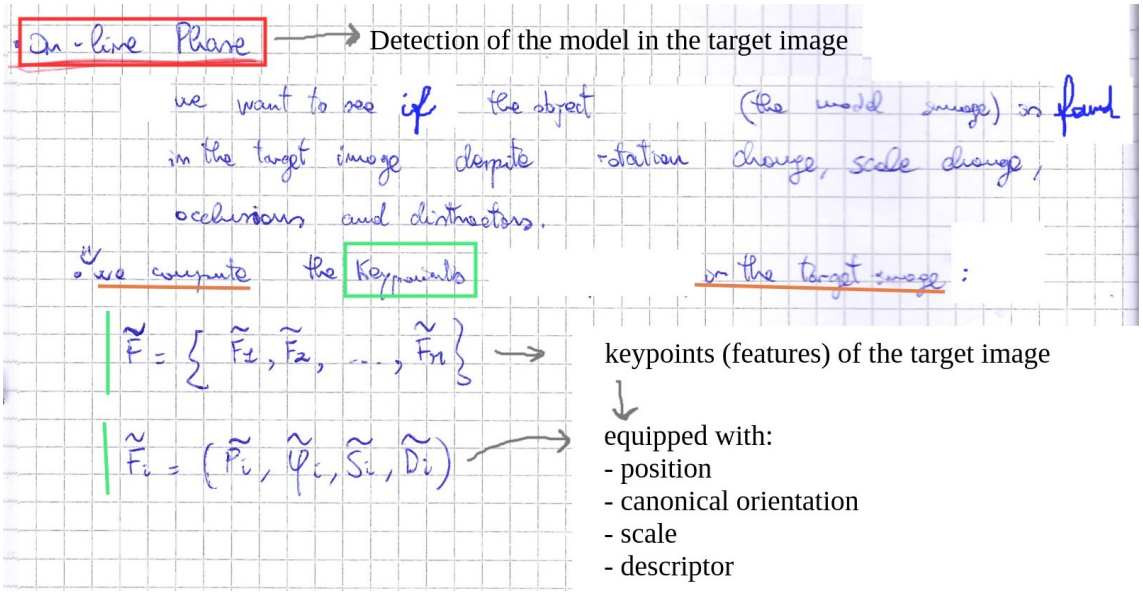


Figure 10.58

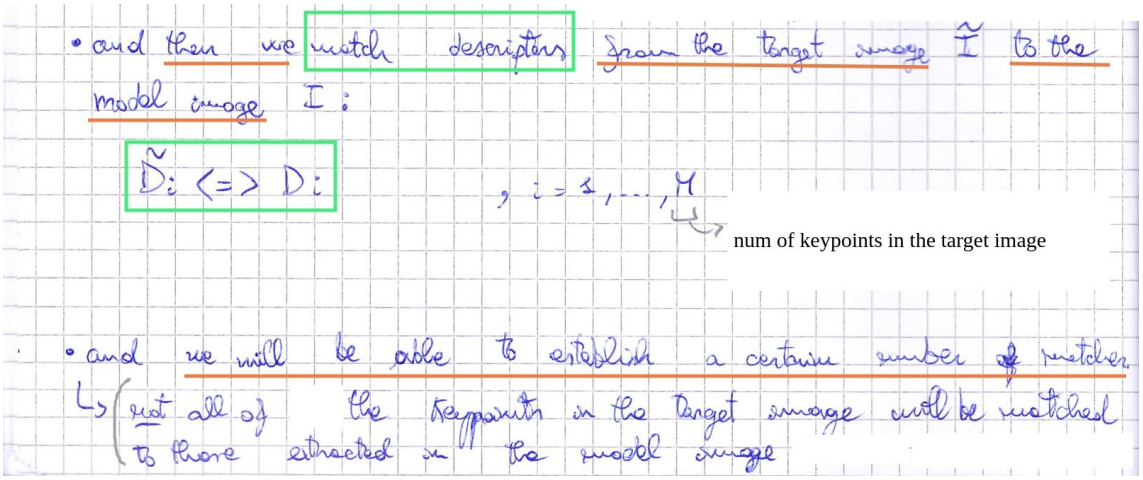


Figure 10.59

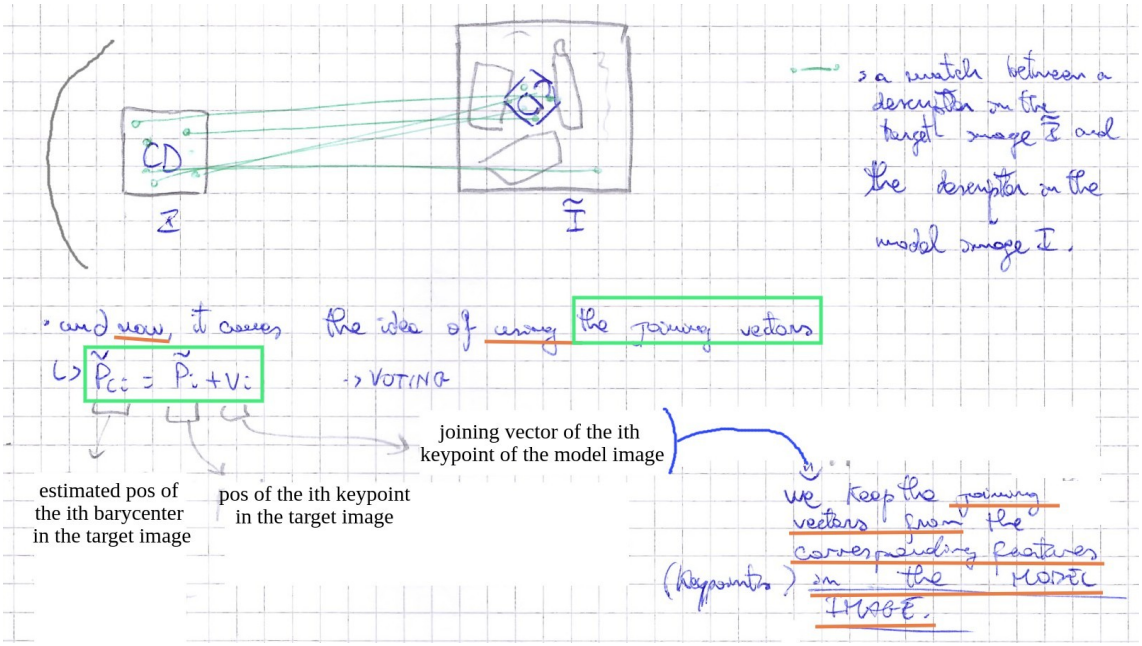


Figure 10.60

Figure 10.61

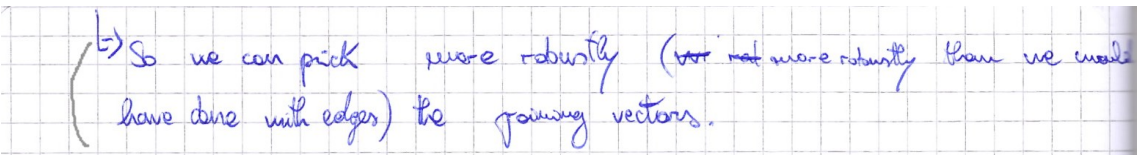


Figure 10.62

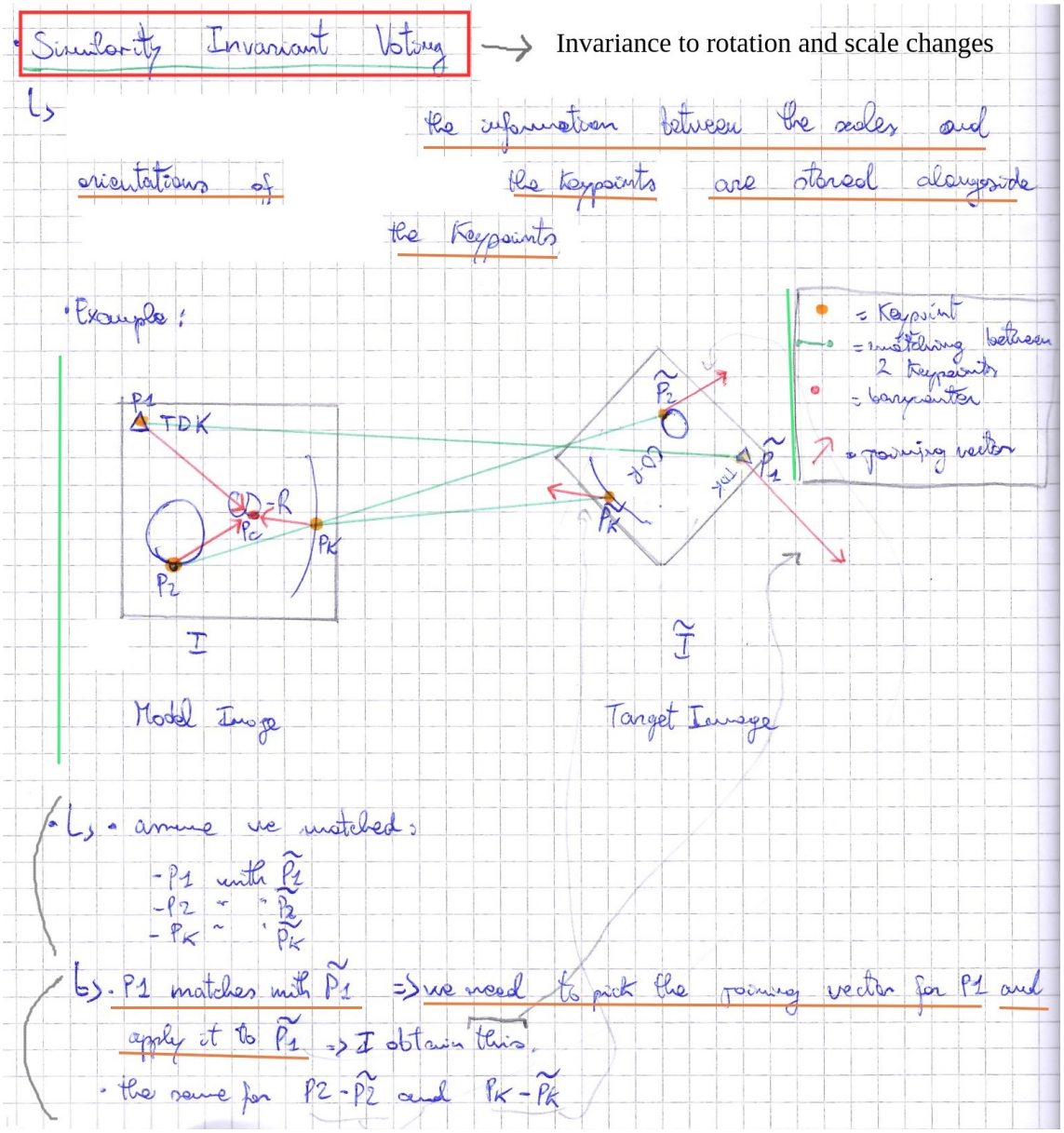

Similarity Invariant Voting
Invariance to rotation and scale changes
the information between the scales and orientations of the keypoints are stored alongside the keypoints
Example:
= Keypoint
= matching between 2 keypoints
= barycenter
= joining vector
P1
TDK
R
Pc
PK
P2
I
Model Image
Target Image
assume we matched:
- P1 with P̃1
- P2 " " P̃2
- PK " " P̃K
b) P1 matches with P̃1 => we need to pick the joining vector for P1 and apply it to P̃1 => I obtain this.
the same for P2 - P̃2 and PK - P̃K

Figure 10.63

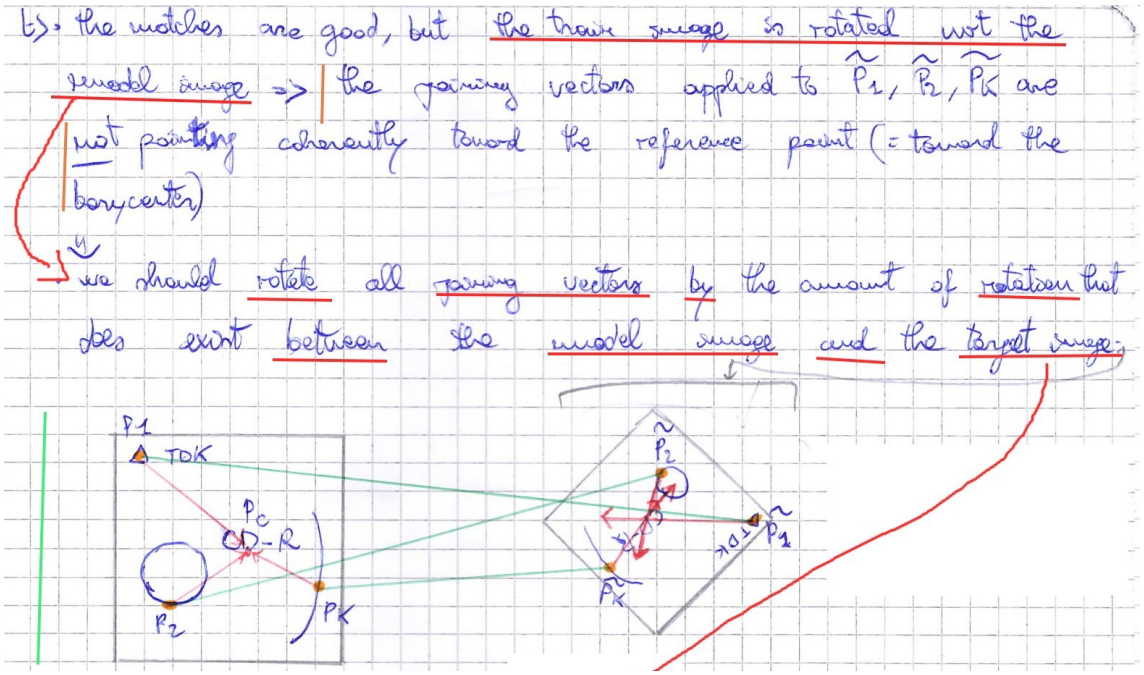


Figure 10.64

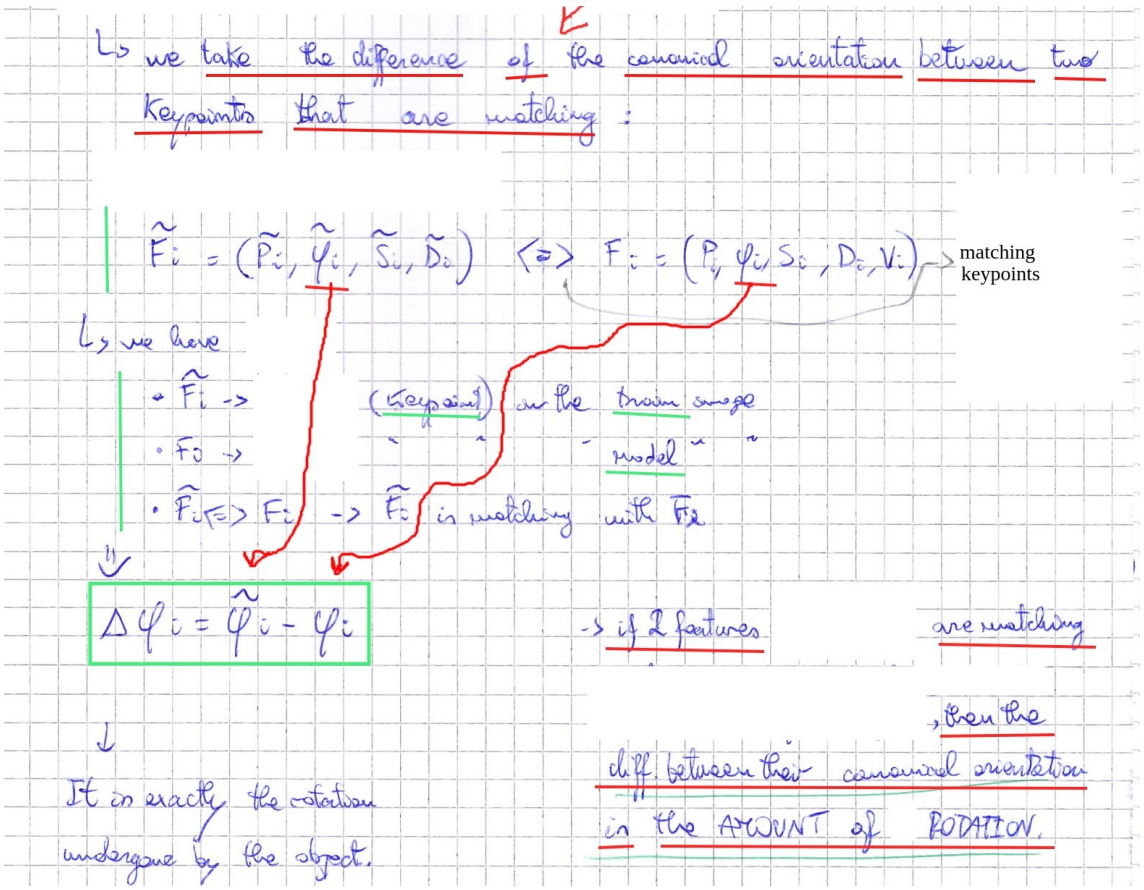


Figure 10.65

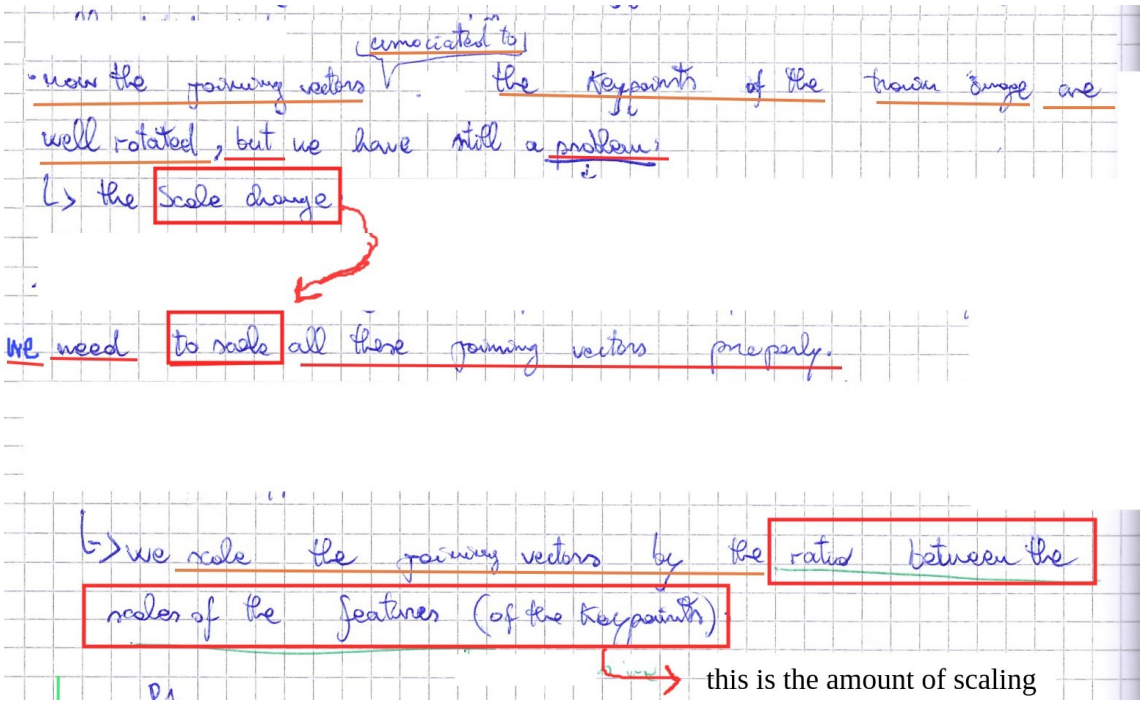


Figure 10.66

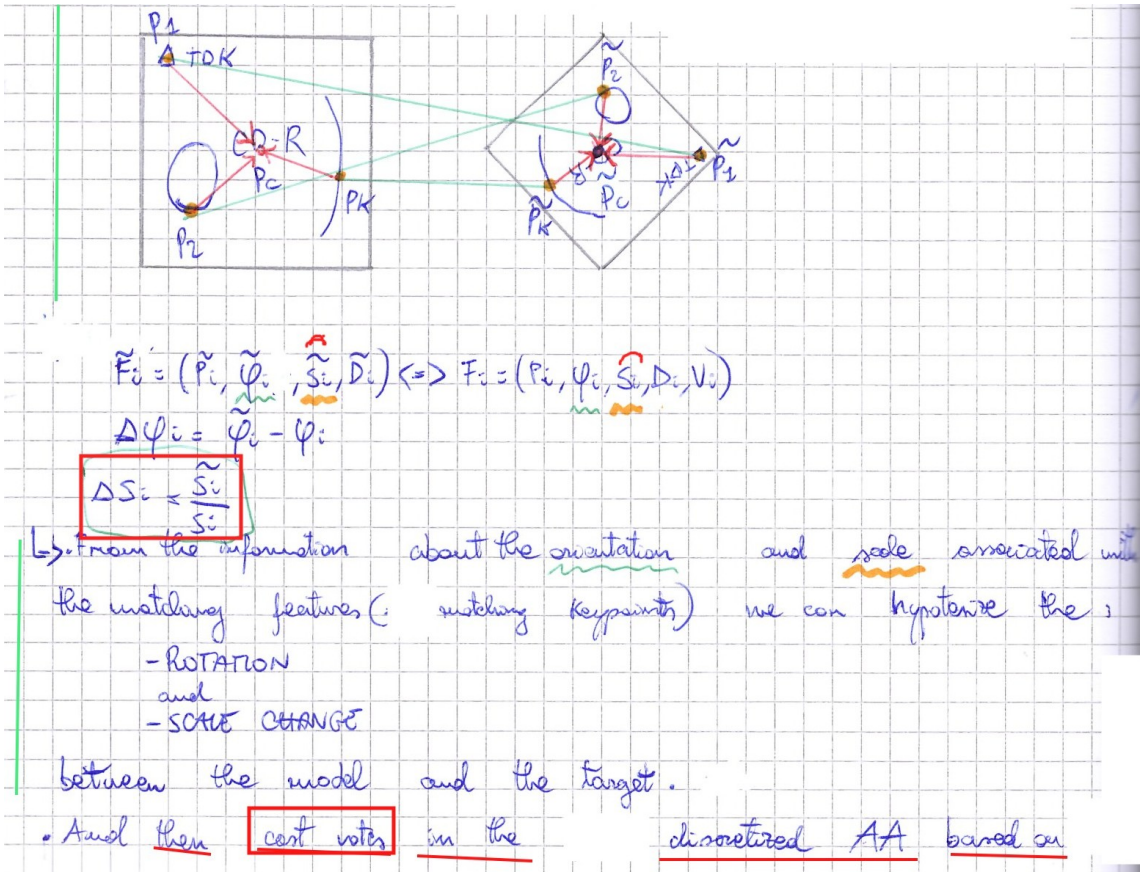


Figure 10.67

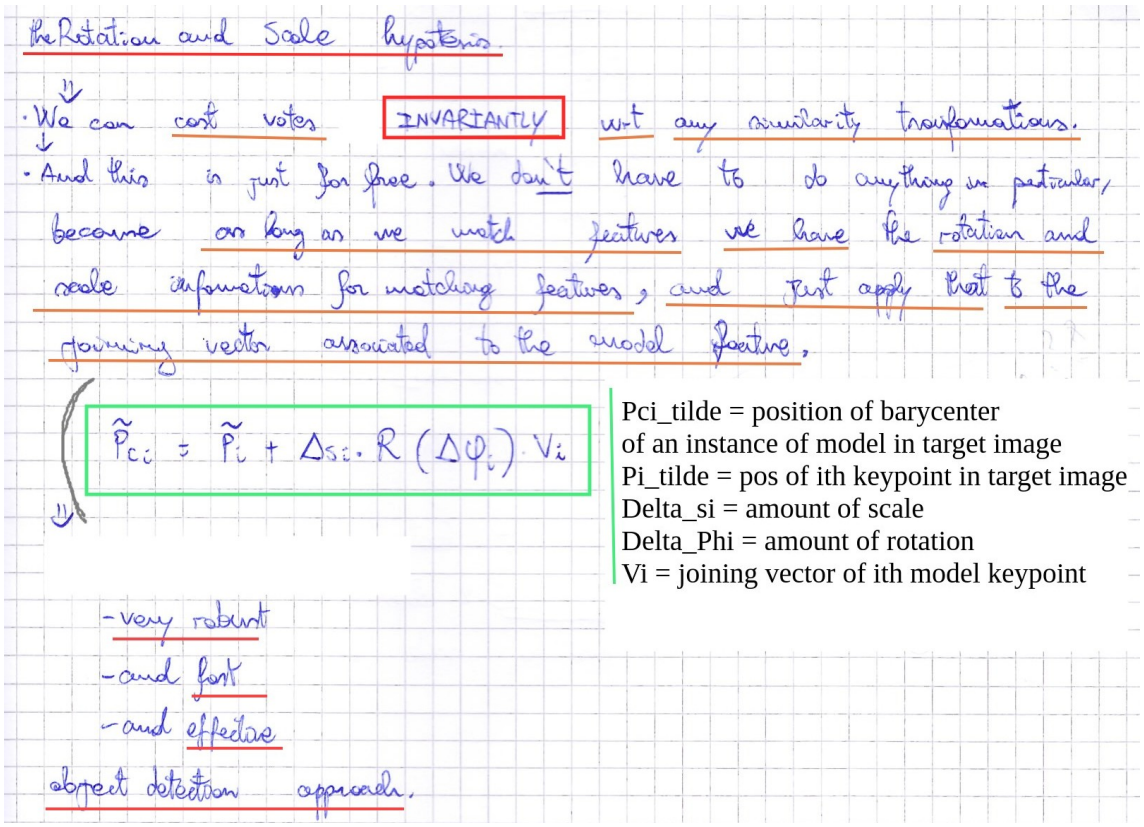


Figure 10.68

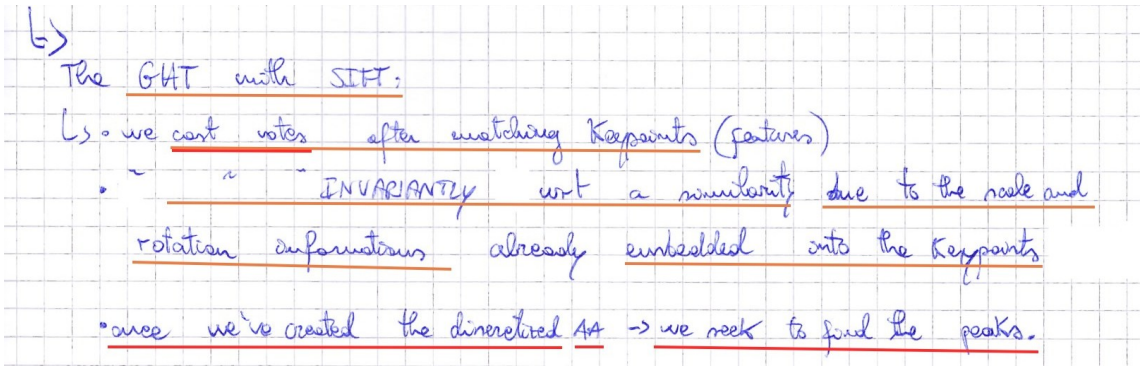


Figure 10.69