Lecture Notes on

# Model-based Visual Tracking

Dr.–Ing. Giorgio Panin

TUM – Informatik VI (Robotics and Embedded Systems)

January 2009

**PART I - MODELS AND BASIC TOOLS FOR 3D TRACKING** ..... **5**

LECTURE 2 – WORLD AND CAMERA GEOMETRY REPRESENTATION ..... 5
*The World-camera model* ..... *5*
*Representation of the extrinsic transformation (body-to-camera)* ..... *9*
*Intrinsic camera model* ..... *18*
*Camera calibration and object pose estimation* ..... *20*
LECTURE 3 – 3D POSE ESTIMATION FROM POINT CORRESPONDENCES ..... 23
*3D pose estimation problem* ..... *23*
*Least-squares estimation* ..... *25*
*Linear LSE optimization* ..... *27*
*Nonlinear LSE: Gauss-Newton and Levenberg-Marquardt* ..... *31*
*Robust LSE* ..... *39*
M-estimators ..... 40
RANSAC ..... 43
LECTURE 4 – BAYESIAN TRACKING (I) ..... 48
*Object state model* ..... *49*
*Dynamic model* ..... *50*
*Measurement model: the Likelihood function* ..... *54*
*General Bayesian tracking scheme: prediction-correction* ..... *61*
Correction step: Bayes'rule ..... 63
Prediction step ..... 64
Bayesian tracking equation ..... 65
*Possible models for the posterior pdf* ..... *67*
LECTURE 5 – BAYESIAN TRACKING (II) ..... 72
*Possible implementations of the tracking scheme* ..... *72*
*Linear+Gaussian case: the Kalman Filter* ..... *74*
Multi-variate Gaussian distribution ..... 74
Motion and measurement models ..... 75
Predition-correction equations ..... 76
*Nonlinear+Gaussian case: the Extended Kalman Filter* ..... *80*
*Non-gaussian situation: a multiple-hypothesis measurement model* ..... *82*
First approach to the multi-modal case: mixture of Gaussians ..... 83
*Most general case: Monte-Carlo sampling scheme (Particle Filters)* ..... *84*
Factored sampling: discrete implementation of Bayes'rule ..... 84
Prediction step: Monte-Carlo sampling from the prior distribution ..... 86
The complete Particle Filter scheme for tracking ..... 89

**PART II – VISUAL MODALITIES FOR OBJECT TRACKING** ..... **90**

LECTURE 6 – COLOR-BASED OBJECT TRACKING ..... 90
*Color-space representations* ..... *93*
*Modeling color distributions* ..... *102*
*The Mean-Shift Algorithm: I – Definition* ..... *112*
*The Mean-shift algorithm: II – Color segmentation* ..... *119*
*The Mean-shift algorithm: III – Object tracking* ..... *124*
*Bayesian tracking for a color-based modality* ..... *131*
LECTURE 7 – THE KANADE-LUCAS-TOMASI FEATURES TRACKER ..... 135
*Local keypoint-based tracking* ..... *135*
Definition: local features ..... 136
Keypoint descriptors database ..... 138
On-line features detection vs. tracking ..... 140
Invariance properties for features detection ..... 142
*The KLT feature tracking algorithm* ..... *144*
Optical flow conditions ..... 145
Solution for the translational model ..... 147
KLT: good features selection ..... 150
KLT: on-line quality check ..... 152
Solution for the 6dof affine model ..... 153
KLT: the full algorithm ..... 155
LECTURE 8 – FEATURE DETECTION METHODS: THE SIFT APPROACH ..... 156
*Features detection* ..... *156*
*Invariance properties* ..... *158*
*The Harris-Stephens feature detector* ..... *159*
Detection with the auto-correlation matrix ..... 161

Harris “cornerness” measure ..... 163
Invariance properties of Harris’ detector ..... 169
*Scale-space representation* ..... *172*
Gaussian image filtering ..... 173
Scale-space example ..... 175
The NLoG kernel and the scale-space theorem ..... 177
*SIFT: Scale-Invariant Features Transform* ..... *179*
Computing the DoG scale-space ..... 179
Detecting invariant features ..... 181
Subsampling the Gaussian pyramid (Octaves) ..... 183
Refine features detection ..... 186
Building an invariant descriptor ..... 187
Examples ..... 194
Matching features ..... 197
SIFT - Resume ..... 201
LECTURE 9 – CONTOUR TRACKING USING THE IMAGE EDGE MAP ..... 205
*Definition and motivations* ..... *205*
*Modeling the Object Contour* ..... *209*
*Obtaining the image edge map for tracking* ..... *211*
*Using the edge map for 3D pose estimation* ..... *212*
Contour-based pose estimation in real-time: the RAPiD Algorithm ..... 214
*Using explicit features extracted from the edge map* ..... *217*
Problem formulation ..... 217
Extract image segments ..... 218
Define the segment projection (Warp) ..... 219
*The segment-based pose estimation procedure* ..... *220*
Define the LSE error to be optimized: segment distances ..... 221
The Gauss-Newton step for LSE optimization ..... 225
Comparison between segment and edge map for tracking ..... 227
Bayesian estimation with dynamic models ..... 228
LECTURE 10 – CONTOUR TRACKING USING LIKELIHOOD FUNCTIONS ..... 229
*Representation of curvilinear shapes with B-Splines* ..... *229*
B-Spline basis functions ..... 231
Properties of B-Splines ..... 233
*Multi-modal contour Likelihood* ..... *235*
Multiple-hypothesis edge measurement ..... 237
The complete contour Likelihood ..... 240
Particle Filters for contour tracking – the CONDENSATION approach ..... 242
*Contour tracking using color region statistics* ..... *246*
Motivation of the main idea ..... 246
Color Likelihood definition and the CCD algorithm ..... 250
Modeling the two-sided color statistics ..... 251
Color separation criterion ..... 254
Maximum Likelihood pose hypothesis ..... 257
Refining the cost function ..... 259
1 – Split the optimization in two steps ..... 259
2 – Blurring the statistics ..... 260
3 – Using local statistics for multi-modal distributions ..... 262
Optimizing the Likelihood with Gauss-Newton ..... 264
Adding prior knowledge for MAP estimation ..... 268
The overall CCD pose estimation algorithm ..... 269
LECTURE 11 – ACTIVE APPEARANCE MODELS ..... 273
*Template modeling and tracking* ..... *273*
*Active Appearance Models* ..... *278*
*The Warp function* ..... *281*
*Training the AAM* ..... *283*
Principal Component Analysis ..... 284
Shape model training with PCA ..... 289
Appearance model training ..... 290
*Tracking an AAM* ..... *291*
State definition ..... 292
Optimization of similarity measures ..... 294
LECTURE 12 – THE LUCAS-KANADE ALGORITHM FOR TEMPLATE TRACKING ..... 295
*Piece-wise affine Warp for deformable templates* ..... *295*
*Two steps: estimating pose and appearance parameters* ..... *298*
*The Lucas-Kanade algorithm for pose estimation* ..... *300*
*First speed improvement: the forwards-compositional approach* ..... *305*

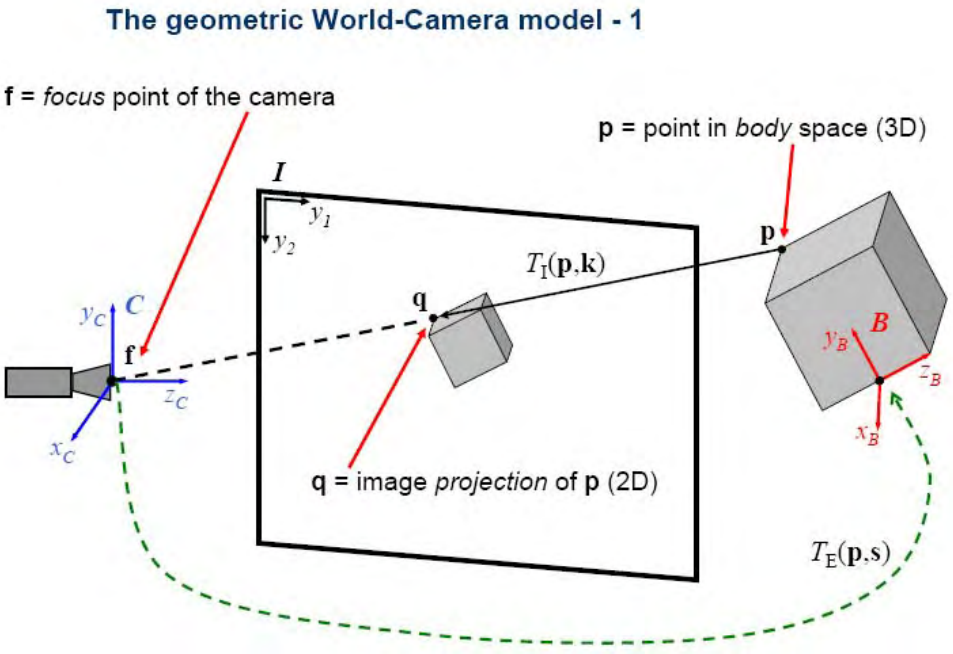
*Second improvement: the inverse-compositional approach* ..... *310*
*Appearance estimation: the appearance subspace decomposition* ..... *314*
*Modified Lucas-Kanade for pose+appearance estimation* ..... *318*
Improving convergence with multi-resolution ..... 320
Estimating 3D pose parameters with a combined (2D+3D) approach ..... 321
LECTURE 13 – ROBUST TEMPLATE SIMILARITY FUNCTIONS ..... 322
*Robustness issues in template tracking* ..... *323*
*Improving robustness of the similarity function* ..... *325*
*Mutual Information for template tracking* ..... *328*
Introduction: information theory ..... 328
Entropy and coding ..... 331
Image entropy computation with histograms ..... 335
Joint image entropy as similarity measure ..... 339
Mutual Information as similarity measure ..... 341
Matching templates with MI ..... 344
Comparison with SSD ..... 346
Optimizing Mutual Information with a Levenberg-Marquardt approach ..... 347
The full MI optimization algorithm ..... 349
**SELECTED BIBLIOGRAPHIC REFERENCES ..... 350**

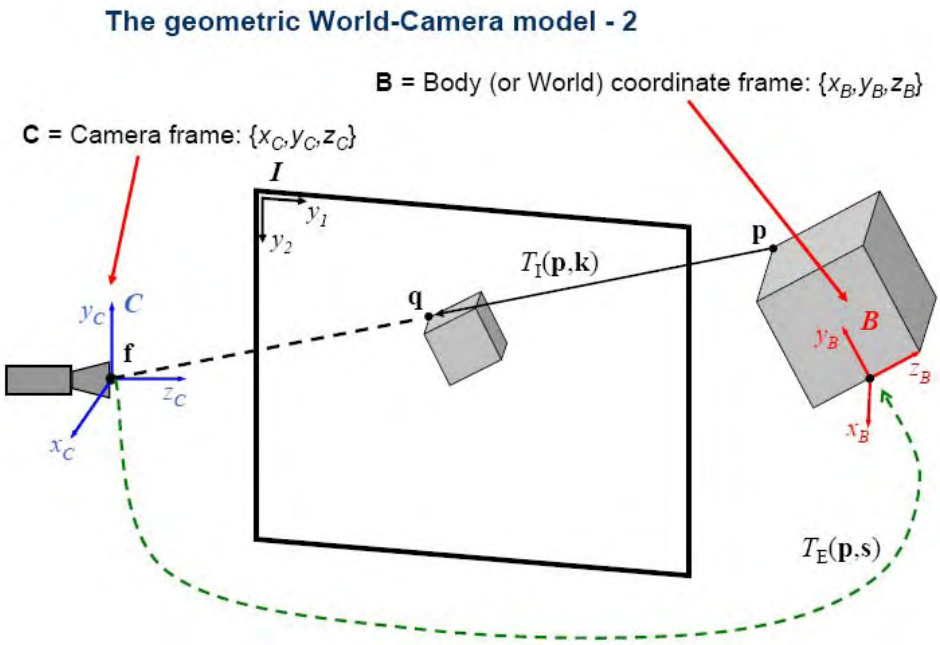
# Part I - Models and basic tools for 3D Tracking

## *Lecture 2 – World and camera geometry representation*

When talking about 3D visual tracking,, we must define the representation of the geometry of world and the image formation models.

### The World-camera model






The world items for 3D tracking consist in our sensor (i.e. the camera), the object model and the coordinate systems for both.

We use two main coordinate systems, that we call camera and body (or world) reference frames. These systems are represented by the origin and 3 orthogonal axes:

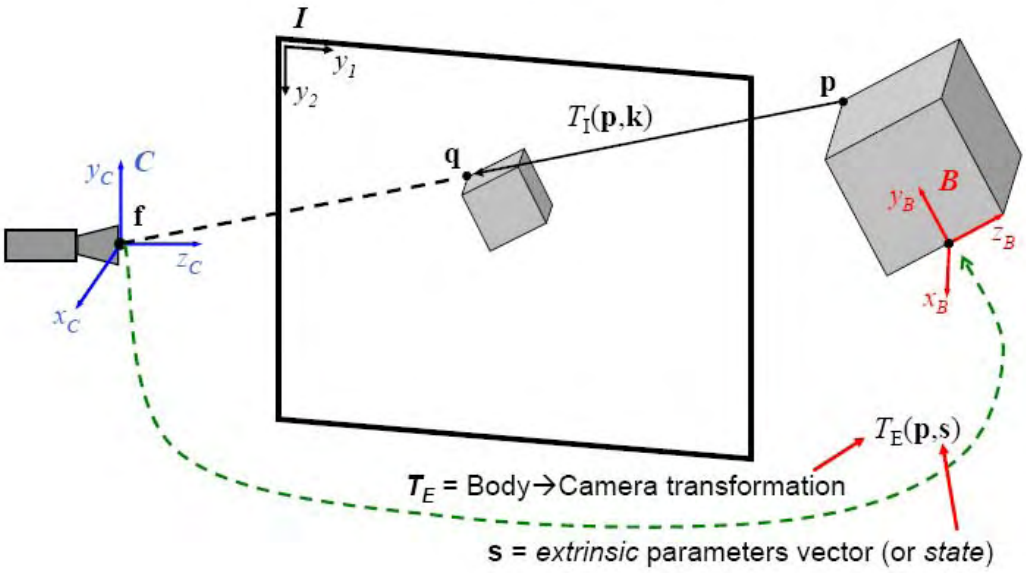
Camera frame = $(O_C, x_C, y_C, z_C)$
Body frame = $(O_B, x_B, y_B, z_B)$

The two reference frames are used to represent the 3-dimensional coordinates of a point in space **p**.

The body reference frame is used to describe positions of points that belong to the object; if theobject is rigid, these points have fixed coordinates with respect to the body frame.
Therefore, the body frame is used to describe the structure of the object to be tracked.
March 3, 2009
The camera reference frame is attached to the camera, with the origin coincident with the camera lens, and z axis oriented along the optical axis (depth direction).
In order to keep right-handed frames, we need to orient the y-axis downwards. This is necessary because for 3D space transformation all quantities (rotations etc.) are usually referred to right-handed coordinate systems.

The camera frame is used to project points from space to screen (3D/2D), by using the intrinsic transformation model.

## The geometric World-Camera model - 3

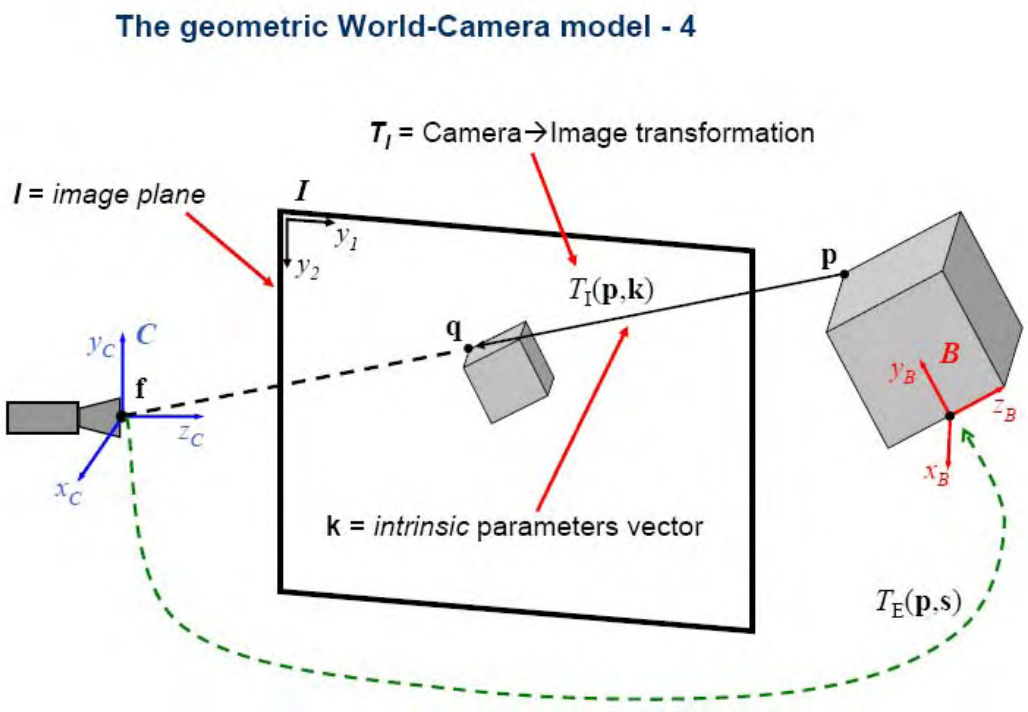



Therefore, we need to define the transformation that maps body coordinates to camera coordinates. This is called Extrinsic Transformation $T_E(\mathbf{p},\mathbf{s})$ of a point **p**, with parameters **s**.

The parameter vector **s** is called *state* or *pose* parameters, and it says how the object frame is positioned in space with respect to the camera frame. It contains information about the orientation and translation between the two frames.

By indicating with ${}^{C}\mathbf{p}$, ${}^{B}\mathbf{p}$ the coordinates of **p** with respect to the two frames, we can write

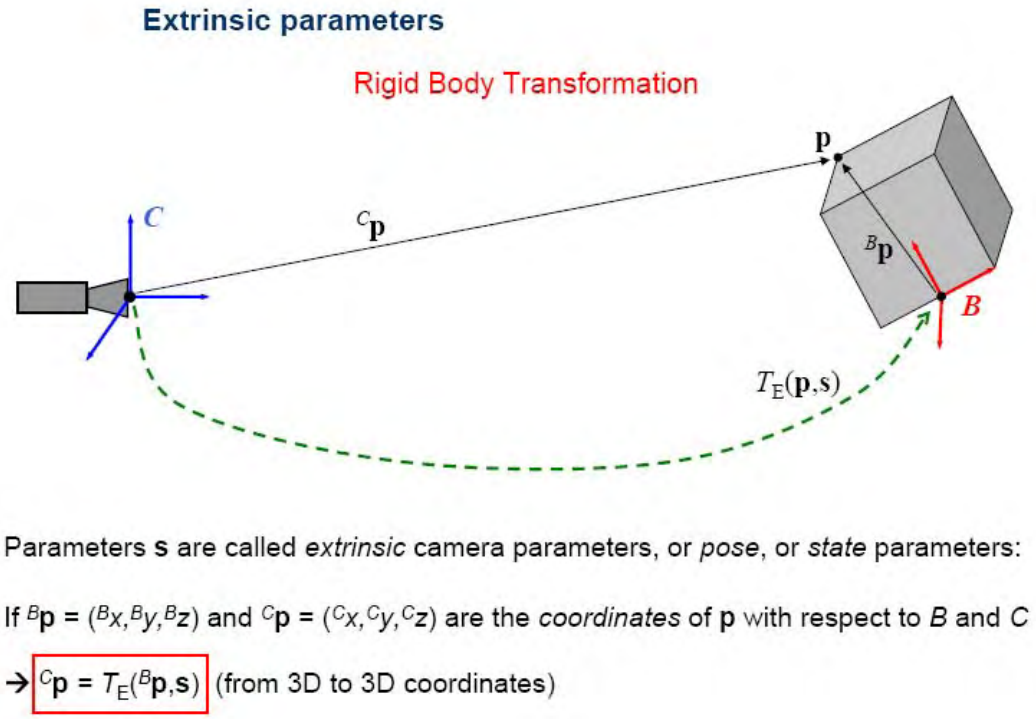
$${}^{C}\mathbf{p} = T_E({}^{B}\mathbf{p},\mathbf{s})$$



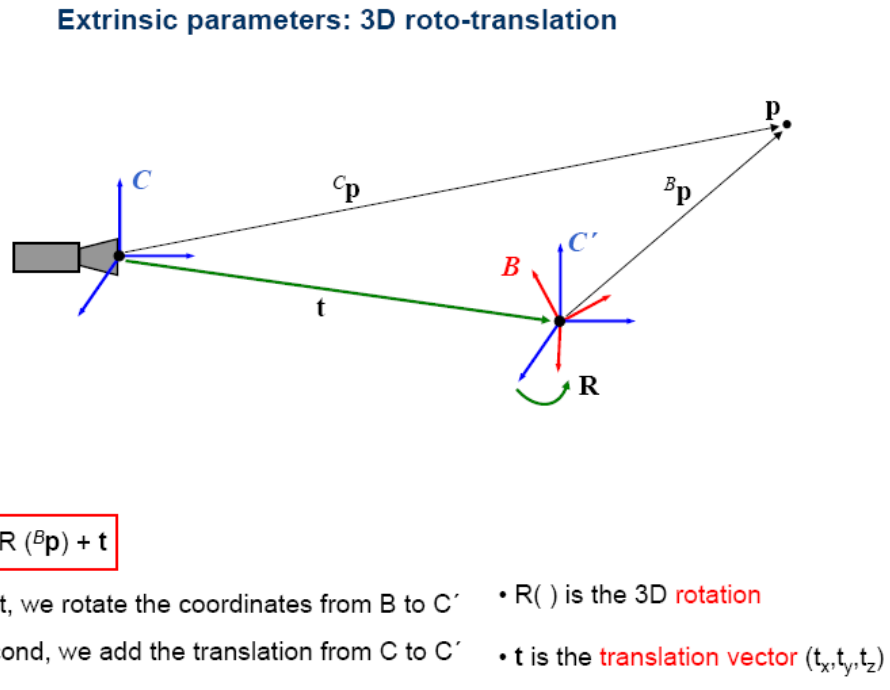

The other main transformation maps points from camera to screen coordinates: this is called intrinsic transformation model $T_I(\mathbf{p},\mathbf{k})$ of a point in camera coordinates ${}^C\mathbf{p}$ to the screen pixel position **q**, with intrinsic parameters **k**.

The intrinsic parameters are related to the internal structure of our digital camera (focal length, pixel resolution, etc.) and the intrinsic transformation is also called image acquisition model; it says how our camera "sees" the world, by mapping space points in its own coordinate system to image pixels.

**Representation of the extrinsic transformation (body-to-camera)**




Pose parameters s are used in order to define the extrinsic transformation, which our case corresponds to a rigid roto-translation (or rigid body tranformation). This transformation maps from 3D to 3D coordinates of a point p, from body reference frame to camera frame.



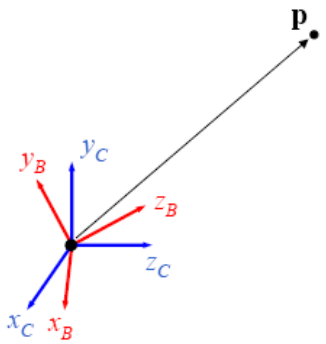

The rigid body transformation operates as a roto-translation: first, we rotate the coordinates of point p, and afterwards we add a translation term t.

$$^{C}\mathbf{p} = R(^{B}\mathbf{p}) + \mathbf{t}$$

This corresponds, geometrically speaking, to the introduction of an intermediate coordinate frame C' between C and B, which has axes parallel to C but origin coincident with B. Therefore, we can say that the coordinates of P are mapped first from B to C' (3D rotation), and then add the translation vector t = C'-C.

**How do we represent 3D rotations?**

Rotation Matrix




$$^{C}\mathbf{p} = R \cdot {}^{B}\mathbf{p}$$

$$\mathbf{R} = \begin{bmatrix} ^{C}\mathbf{x}_B & ^{C}\mathbf{y}_B & ^{C}\mathbf{z}_B \end{bmatrix} = \begin{bmatrix} ^{C}x_{B,1} & ^{C}y_{B,1} & ^{C}z_{B,1} \\ ^{C}x_{B,2} & ^{C}y_{B,2} & ^{C}z_{B,2} \\ ^{C}x_{B,3} & ^{C}y_{B,3} & ^{C}z_{B,3} \end{bmatrix}$$

Orthogonal matrix: $\mathbf{R}^T\mathbf{R} = \mathbf{I}$

$$^{C}\mathbf{x}_B \cdot {}^{C}\mathbf{y}_B = {}^{C}\mathbf{y}_B \cdot {}^{C}\mathbf{z}_B = {}^{C}\mathbf{x}_B \cdot {}^{C}\mathbf{z}_B = 0$$

$$\left\| {}^{C}\mathbf{x}_B \right\| = \left\| {}^{C}\mathbf{y}_B \right\| = \left\| {}^{C}\mathbf{z}_B \right\| = 1$$

We have 9 elements + 6 conditions → We have 3 degrees of freedom (*dof*)

A pure 3D rotation is a function that maps 3D coordinates of points or vectors in space between cartesian frames, when the origin of the coordinate frames is the same (no translation).

The next question is: how can we perform (and represent) a rotation in space?
First, we define the function in the most general way, through the rotation matrix R.

$$^{C}p = R\,{}^{B}p$$

This is a generic linear transformation, through a (3x3) matrix R whose columns contain the 3 axes of the body frame ($x_B$,$y_B$,$z_B$), written in terms of the camera frame:

$$\mathbf{R} = \begin{bmatrix} ^{C}\mathbf{x}_B & ^{C}\mathbf{y}_B & ^{C}\mathbf{z}_B \end{bmatrix} = \begin{bmatrix} ^{C}x_{B,1} & ^{C}y_{B,1} & ^{C}z_{B,1} \\ ^{C}x_{B,2} & ^{C}y_{B,2} & ^{C}z_{B,2} \\ ^{C}x_{B,3} & ^{C}y_{B,3} & ^{C}z_{B,3} \end{bmatrix}$$

Of course, if the two frames are coincident ($x_B$=$x_C$, $y_B$=$y_C$, $z_B$=$z_C$) then R=I.

We consider always orthogonal right-handed frames, which geometrically means that:

- the axes are always orthogonal one another: $x \cdot y = y \cdot z = x \cdot z = 0$
- they have unit length: $\|x\| = \|y\| = \|z\| = 1$

- their ordering (x,y,z) follows the rule $z = x \times y$, where $\times$ is the cross product between vectors (right-hand rule)

Because of the first two conditions, in a transformation of coordinates between orthogonal frames, the matrix R has always the property:

$$R^T R = I$$

while the third condition ensures that $|R| = +1$.

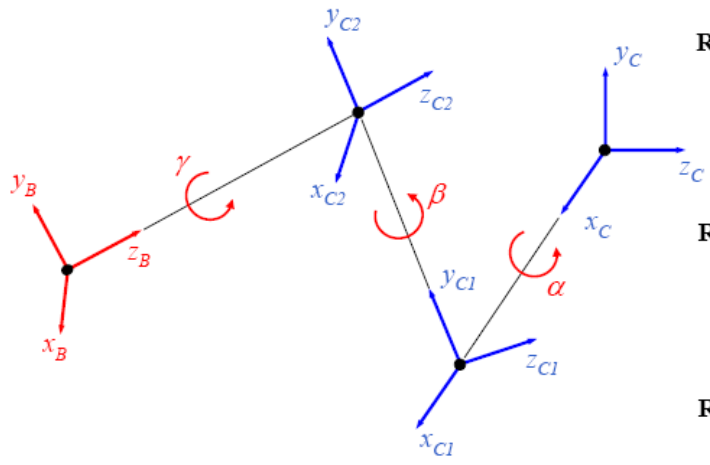
NOTE: the third condition is true also if both frames are left-handed; but not if one is left-handed and the other one right-handed! This is also called constant *polarity* condition.

Equation (…) correspond to the set of 6 constraints above mentioned, and therefore it reduced the number of free parameters inside R from 9 to 3=9-6.
This means that a rigid rotation in space has always 3 degrees of freedom = number of free parameters necessary to specify the transformation.

Therefore, we can use more compact ways to represent rigid rotations, instead of the full (3x3) elements of R.

**Rotations : 1 - Euler angles**

Euler angles (x-y-z)




$$\mathbf{R}_X(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

$$\mathbf{R}_Y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & -\sin(\beta) \\ 0 & 1 & 0 \\ \sin(\beta) & 0 & \cos(\beta) \end{bmatrix}$$

$$\mathbf{R}_Z(\gamma) = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Sequence of 3 *elementary* rotations around (x,y,z) $\quad \mathbf{R}(\alpha,\beta,\gamma) = \mathbf{R}_X(\alpha) \cdot \mathbf{R}_Y(\beta) \cdot \mathbf{R}_Z(\gamma)$

The first, and more intuitive, way to represent 3D rotations is given by elementary rotation matrices, or Euler angles.

We can obtain any 3D rotation with a proper sequence of 3 rotations around elementary (x,y,z) axes.

For this purpose, we first define the sequence of axes use to perform the rotation: for example, first x, then y and finally z.
These rotations are obtained in cascade (one after the other), with 3 angles (α,β,γ).
Note that the (x,y,z) axes are the respective axes of each frame: x is the x-axis of frame C, y is the y-axis of frame C1, and z the z-axis of frame C2.

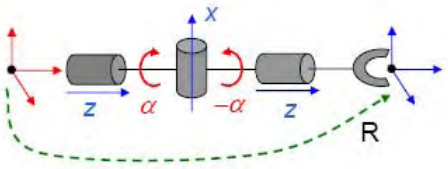
In this way, we can say that the rotation matrix is obtained as a multiplication of elementary matrices, each one function of the respective angle:

$$\mathbf{R}(\alpha,\beta,\gamma) = \mathbf{R}_X(\alpha)\cdot\mathbf{R}_Y(\beta)\cdot\mathbf{R}_Z(\gamma)$$

**Rotations : 1 - Euler angles**

- Advantages: easy to understand, and only 3 parameters
- Disadvantages: ambiguity (singularity) of the representation for some particular configurations

Example of typical singularity:
(z-x-z) axes in robotics




R = I for every 3 angles $(\alpha,0,-\alpha)$!

This representation has the advantage of being easy to understand and to compute; but has also some disadvantages.
First, if we wish to solve the inverse problem (given R, find the 3 angles (α,β,γ)), there is always a 2-fold ambiguity: that is, there are always two different triplets (α1,β1,γ1), (α2,β2,γ2) in the range [-180°,+180°] give the same R.
And moreover, there are cases where infinite triplets (α,β,γ) give the same R: these are called representation singularities.

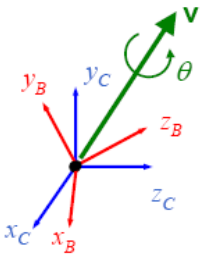
For example, in robotics often the Euler (z-x-z) angles are used for the robot wrist, that are also the real mechanical axes of the structure. In this case, the singularity is exactly when R=I, that correspond to infinite possible triplets (α,0,–α).

Whatever the choice of axes, for Euler angles there is always a singular configuration, corresponding to a single R and infinite triplets.

In the gyroscopes terminology, this is also called gymbal lock problem, and for many measurement instruments it can be a serious problem; for our visual tracking applications, it can produce a bad behavior in algorithms that try to estimate the 3D pose of an object from any kind of measurement (points, edges, etc.), when the real rotation R is close to a singular one.

On the other hand, if for a particular problem we are sure that the object will never rotate close to a singular configuration, we can safely use Euler angles, because of their computational simplicity and clear geometric representation. For example, if we track a vehicle driving on a road, we are sure that it will never flip "upside-down", so we can define the axes in a way that the singularity occurs only in this non-realistic situation.

## Rotations : 2 - Axis-angle representation




Rodrigues' formula

$$\mathbf{R}(\mathbf{v},\theta) = \mathbf{I} + [\mathbf{v}\times]\sin(\theta) + [\mathbf{v}\times]^2(1-\cos(\theta))$$

$$[\mathbf{v}\times] = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

$$\|\mathbf{v}\| = 1$$

No singularity for every $\theta \neq 0$ : for every R, there is only: $(\mathbf{v},\theta)$ and $(-\mathbf{v},-\theta)$

Problem when $\theta \to 0$ : we use the *small angle approximation*

If $\theta \to 0$, then

$\sin(\theta) \approx \theta$

$\cos(\theta) \approx 1$

$$\mathbf{R} = \mathbf{I} + [\mathbf{v}\times]\theta = \mathbf{I} + [\mathbf{w}\times]$$

$$\mathbf{w} = \mathbf{v}\theta = \begin{bmatrix} w_x & w_y & w_z \end{bmatrix}^T$$

Another way of representing 3D rotation uses the equivalent angle-axis pair.
Any 3D rotation can be also obtained with a single rotation through an axis v (with unit length) of an angle θ.
This correspond to the following formula:

$$\mathbf{R}(\mathbf{v},\theta) = \mathbf{I} + [\mathbf{v}\times]\sin(\theta) + [\mathbf{v}\times]^2(1-\cos(\theta))$$

$$[\mathbf{v}\times] = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

$$\|\mathbf{v}\| = 1$$

which is also called Rodrigues' formula.
NOTE: A property of the rotation axis v is that it has the same projection (coordinates) with respect to both frames, so there is no ambiguity in the formula.
This representation (v,θ) has also 3 degrees of freedom, because v has norm 1. Generally speaking, we have always two axis-angle pairs (v,θ) and (-v,-θ) for the same rotation R, apart from the case R=I, where we still have a singularity.
In fact, any pair (v,0) gives R=I; but this singularity can be now better solved, by using the more compact rotation vector w=vθ.

## Rotations : 2 - Axis-angle representation

We can also write (more compact)

$$\mathbf{R}(\mathbf{w}) = \mathbf{I} + [\mathbf{w}\times]\frac{\sin(\|\mathbf{w}\|)}{\|\mathbf{w}\|} + [\mathbf{w}\times]^2\frac{1-\cos(\|\mathbf{w}\|)}{\|\mathbf{w}\|}$$

$$\mathbf{w} = \mathbf{v}\theta = [w_x \quad w_y \quad w_z]^T$$
$$\theta = \|\mathbf{w}\|$$

- Advantages: only 3 parameters ($w_x$, $w_y$, $w_z$), and no singularity
- Disadvantages: the formula is more complex to compute

With the rotation vector w, we can write

$$\mathbf{R}(\mathbf{w}) = \mathbf{I} + [\mathbf{w}\times]\frac{\sin(\|\mathbf{w}\|)}{\|\mathbf{w}\|} + [\mathbf{w}\times]^2\frac{1-\cos(\|\mathbf{w}\|)}{\|\mathbf{w}\|}$$

$$\mathbf{w} = \mathbf{v}\theta = [w_x \quad w_y \quad w_z]^T$$
$$\theta = \|\mathbf{w}\|$$

which is always good, because for w~0 we can approximate

$$\mathbf{R} \approx \mathbf{I} + [\mathbf{w}\times]$$

An advantage of this representation is that avoids singularities, but at a price of more complex computations and derivatives.
This representation still has 3 degrees of freedom, that correspond to the 3 components of the rotation vector w (with respect of any of the two frames).

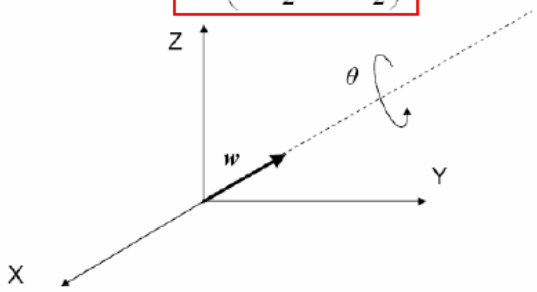
## Rotations : 3 - Quaternions

Quaternions are hyper-complex numbers that can be written as the linear combination $a+bi+cj+dk$, with $i^2 = j^2 = k^2 = ijk = -1$.

Can also be interpreted as a scalar plus a 3- vector: $(a, \mathbf{v})$.

A rotation about the unit vector $w$ by an angle $\theta$ can be represented by the unit quaternion:

$$q = \left( \cos\frac{\theta}{2}, w \sin\frac{\theta}{2} \right)$$




## Rotations : 3 - Quaternions

Quaternion product

$$q_1 \cdot q_2 = (a_1, \mathbf{v}_1) \cdot (a_2, \mathbf{v}_2) = (a_1 a_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, a_1 \mathbf{v}_2 + a_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2)$$

To rotate a 3D point $\mathbf{M}$: write it as a quaternion $p = (0, \mathbf{M})$, and take the rotated point $p'$ to be

$$p' = q.p.\bar{q} \quad \text{with} \quad \bar{q} = \left( \cos\frac{\theta}{2}, -w \sin\frac{\theta}{2} \right)$$

No gimbal lock.

**Parameterization of the rotation using the 4 coordinates of a quaternion $q$:**

1. No constraint and rotation performed using $\frac{q}{\|q\|}$ → singularity: $kq$ yields the same rotation whatever the value of k > 0;
2. Additional constraint: norm of q must be constrained to be equal to 1,for example by adding the quadratic term $K(1-\|q\|^2)$.

Quaternions are another way of representing 3D rotations, this time in a redundant way: we have 4 instead of 3 numbers, plus 1 constraint (equation) to be satisfied by the 4 numbers.

The redundancy is introduced in order to completely remove singularity from the representaiton of 3D rotations.

A quaternion represents 3D rotations in a very similar way as complex numbers represent rotations on the plane.

We remember that complex numbers are given by c = a + jb with the rule (definition) $j^2$=-1.

$$\mathbf{R} = \begin{bmatrix} \cos\theta & sin\theta \\ -sin\theta & \cos\theta \end{bmatrix}$$

If we represent a planar rotation of an angle q, this correspond to a (2x2) matrix R

that is also represented by a complex number c = cosθ + jsinθ, with ||c||2 = 1.

In 3 dimensions, we have "hyper-complex" numbers with 4 parameters:

q = a + ib + jc + kd

obeying the rules: i2 = j2 = k2 = -1 and ijk = -1 (Hamilton axioms).
These numbers are called quaternions; we can also write a quaternion as

q = (a,v), where a is the scalar part and v = (b,c,d) is the vector part of the quaternion.

Multiplication between quaternions, observing Hamilton rules, is done in the following way:

$$q_1 \cdot q_2 = (a_1, \mathbf{v}_1) \cdot (a_2, \mathbf{v}_2) = (a_1 a_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, a_1 \mathbf{v}_2 + a_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2)$$

If we also impose the constraint ||q||2 = (a2+b2+c2+d2) = 1, then we have unit quaternions, and they are used to represent 3D rotations.

Mathematically, rotations are performed in the following way: we first write a 3D vector p as a quaternion with null scalar part: (0,p), and then we compute the product:

$$(0, {}^{C}p) = q \cdot (0, {}^{B}p) \cdot \bar{q}$$

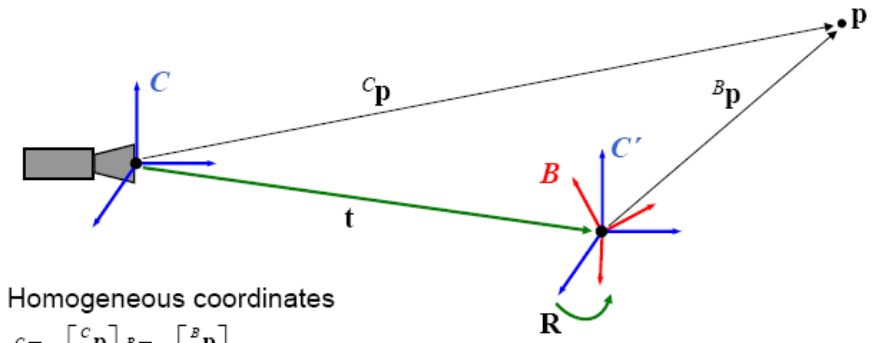
where $\bar{q} = (a, -v)$ is the conjugate of q.
If we write q = (cos θ/2, w sin θ/2) with a = cos θ/2, (b,c,d) = w sin θ/2, then (w,θ) is exactly the axis-angle pair for the rotation.

An advantage of this representation is that there are no singularities, because every possible rotation matrix correspond to exactly two quaternions: R ←→ (q, -q).
The formula that converts quaternions to rotation matrices and vice-versa is also very compact, and without sin(), cos() functions (only 2nd degree polynomials), therefore very good for computing derivatives, cost functions etc.

A disadvantage is that now we have 4 parameters, plus the constraint ||q||=1, so this is a redundant representation like R. But of course, this single contraint is much easier to deal with, with respect to the orthogonality constraint R'R=I.

The unit norm constraint for q has to be ensured (or at least enforced), whenever a quaternion is updated (for example in a pose estimation problem); this can be obtained for example by dividing q/||q|| after each modification, or by adding a penalty term K(||q||-1)2 to the cost function to be optimized, with K a high coefficient.

## Complete 3D transformation matrix (roto-translation)




Homogeneous coordinates

$$^{C}\bar{\mathbf{p}} = \begin{bmatrix} ^{C}\mathbf{p} \\ 1 \end{bmatrix}; \; ^{B}\bar{\mathbf{p}} = \begin{bmatrix} ^{B}\mathbf{p} \\ 1 \end{bmatrix}$$

$$\mathbf{T}(\alpha,\beta,\gamma,t_x,t_y,t_z) = \begin{bmatrix} & & & t_x \\ & \mathbf{R}(\alpha,\beta,\gamma) & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

State vector (example: Euler angles)

$$^{C}\bar{\mathbf{p}} = \mathbf{T}(\mathbf{s})\, ^{B}\bar{\mathbf{p}}$$

$$\mathbf{s} = (\alpha,\beta,\gamma,t_x,t_y,t_z)$$

The extrinsic roto-translation function is then obtained by adding the translation vector t = C'-C to the rotated coordinates

Cp = R Bp + t

this relationship can be more compactly written by introducing the homogeneous coordinates representation:

$$\bar{p} = [\, p \quad 1]^T$$

and the overall Transformation matrix

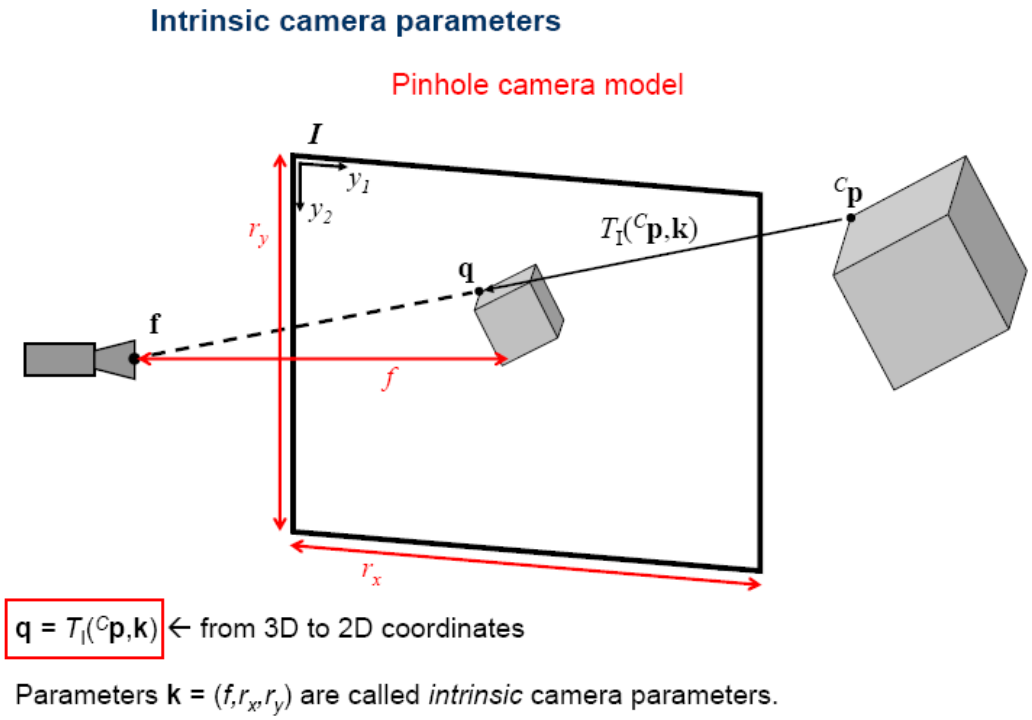
$$\mathbf{T}(\alpha,\beta,\gamma,t_x,t_y,t_z) = \begin{bmatrix} & & & t_x \\ & \mathbf{R}(\alpha,\beta,\gamma) & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

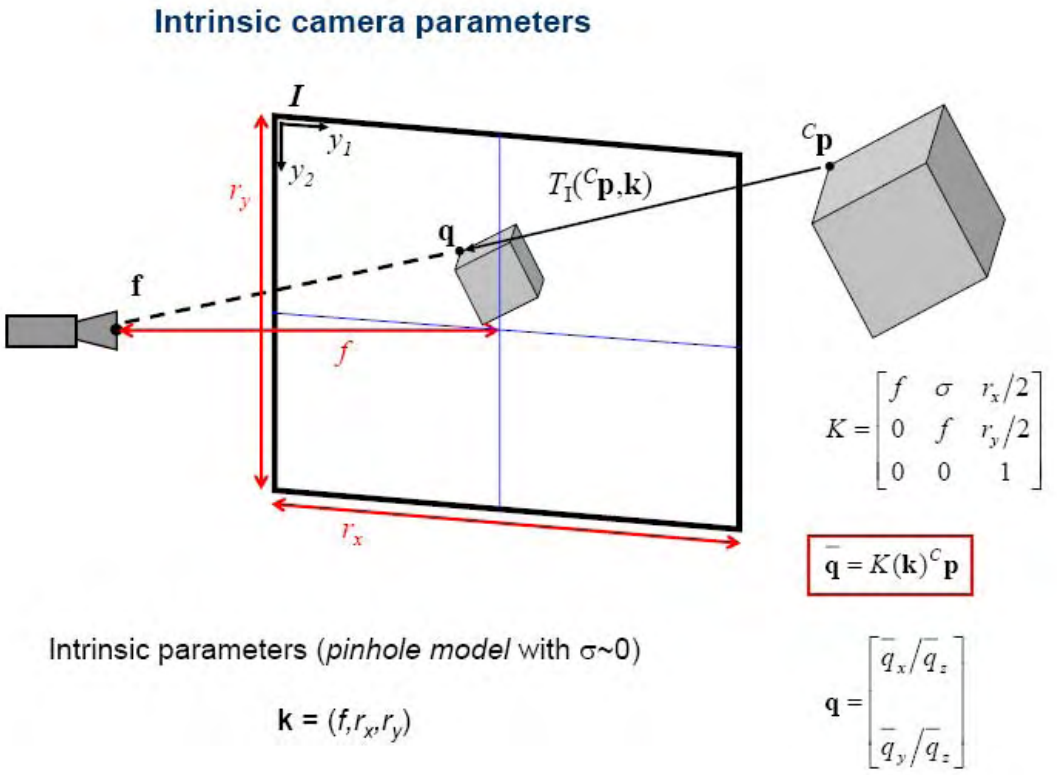
so that

$$^{C}\bar{\mathbf{p}} = \mathbf{T}(\mathbf{s})\, ^{B}\bar{\mathbf{p}}$$

where $\mathbf{s} = (\alpha,\beta,\gamma,t_x,t_y,t_z)$ is the state (or pose) vector of the object with 6 degrees of freedom.

**Intrinsic camera model**




The intrinsic transformation maps 3D points from camera space to 2D image coordinates (pixels).
This transformation is related to the acquisition system (camera lens, CCD device, etc.) and can be modeled in different ways, more or less precise with respect to the real physical system.
We consider here a simple and well-known model, the pinhole camera model, that is realistic and at the same time simple enough for our tracking algorithms.

The pinhole camera model consists in 3 parameters: focal length, horizontal and vertical resolution of the screen. These are called intrinsic camera parameters k=(f,rx,ry).
The focal length *f* is the distance between the camera frame origin (also called projection center) and the virtual screen, which is the plane where space points are projected. A point is then projected by casting a ray from the projeciton center to the point, and looking for the intersection with the screen. In this model, pixel units are equal to screen positions, so that points at depth f have x and y coordinates equal to pixel coordinates. Since image coordinates are usually expressed with respect to the lower left corner of the screen, we must also add half the resolutions to the projected coordinates.

The final transformation, from p=(px,py,pz) to q=(qx,qy) is therefore

$$q_x = \frac{p_x}{p_z} f + \frac{r_x}{2}$$

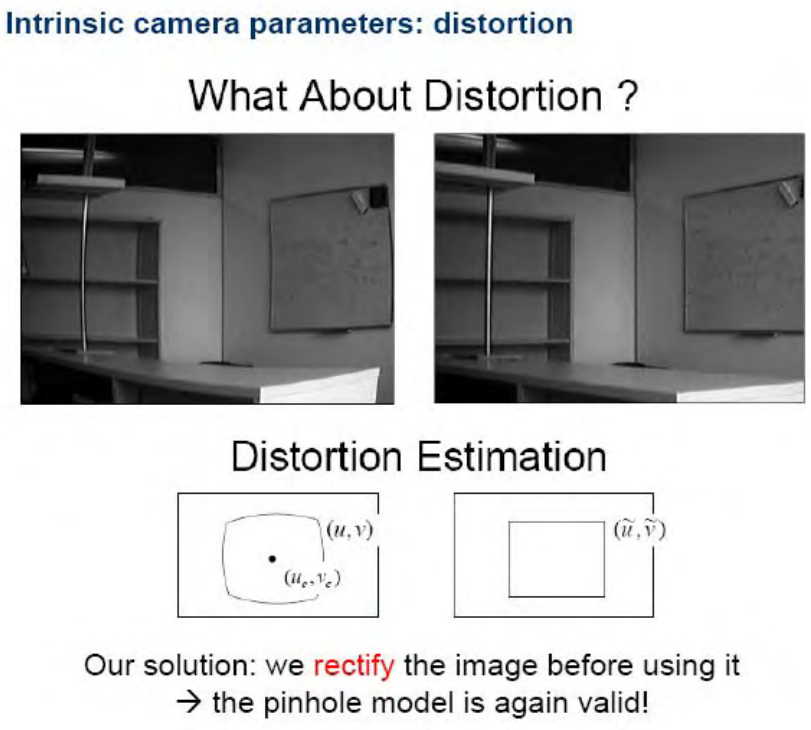
$$q_y = \frac{p_y}{p_z} f + \frac{r_x}{2}$$

which, in homogeneous coordinates, can also be written as

$$\bar{\mathbf{q}} = K(\mathbf{k})^C \mathbf{p}$$

$$K = \begin{bmatrix} f & 0 & r_x/2 \\ 0 & f & r_y/2 \\ 0 & 0 & 1 \end{bmatrix}$$

where K is the intrinsic matrix; real pixel cordinates are then given by $q = (\bar{q}_x / \bar{q}_z, \bar{q}_y / \bar{q}_z)$.




The simple pinhole model does not take into account nonlinear distortion, which often is present in camera systems.

Nonlinear distortion effects can be noticed by looking a straight lines, that in the image may get some curvature. In wide-angle lenses this effect is very strong, while for small focal length (e.g. 1000mm) it is usually less noticeable.
Distortion can be included into the projection model, but this results in a complex computation with nonlinear terms, which is not recommended for real-time tracking purposes.
Instead, if the distortion parameters are known in advanced (by performing a standard camera calibration procedure) another possibility is to remove distortion effects from the image, by inverting the distortion phenomenon: this pre-processing is called image rectification, and it is a fast procedure that can be done before using the image for our tracking tasks. Afterwards, we can reliably use a pinhole model. The price to pay is that rectification, which is an image (pixel-based) processing, can introduce artifacts, that is, spurious pixel patterns in small areas, that can a little bit reduce the image quality, and disturb a non-robust tracker. But usually, this disturbance is very low, and no more than the normal image noise level, therefore this is the preferred solution for our purposes.

## Camera calibration and object pose estimation

### Projection Matrix

We have

- Extrinsic transformation: ${}^{C}\mathbf{p} = T_E({}^{B}\mathbf{p},\mathbf{s})$ ← from 3D to 3D
- Intrinsic transformation: $\mathbf{q} = T_I({}^{C}\mathbf{p},\mathbf{k})$ ← from 3D to 2D

We can combine them into one relationship

$$\bar{\mathbf{q}} = \mathbf{K}\left[R \mid \mathbf{t}\right] {}^{B}\bar{\mathbf{p}} = P\, {}^{B}\bar{\mathbf{p}}$$

with P = *Projection Matrix* (3x4)

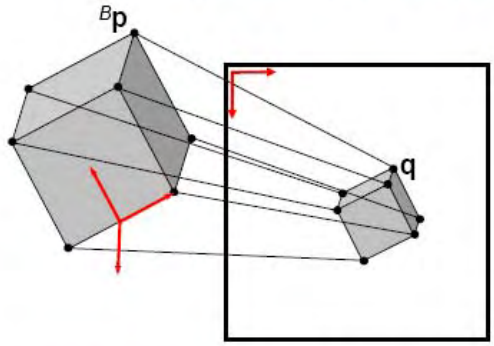
$\mathbf{q} = T({}^{B}\mathbf{p},\mathbf{s},\mathbf{k})$ ← *global transformation*, from body to image (extrinsic+intrinsic)

Intrinsic and extrinsic transformations can be combined together in the global transformation, from body to screen coordinates, by introducing the (4x3) Projection matrix P.

$$\bar{\mathbf{q}} = \mathbf{K}\left[R \mid \mathbf{t}\right] {}^{B}\bar{\mathbf{p}} = P\, {}^{B}\bar{\mathbf{p}}$$

This matrix is used often in camera calibration problems, when both intrinsic and extrinsic (pose) parameters have to be estimated at the same time; for example, in the Direct Linear Transform algorithm – DLT, where P is directly estimated form 3D/2D point correspondences.

## Camera calibration and pose estimation




Given a set of *N* correspondences:

$({}^{B}\mathbf{p}_1 \leftrightarrow \mathbf{q}_1)$
$({}^{B}\mathbf{p}_2 \leftrightarrow \mathbf{q}_2)$
…
$({}^{B}\mathbf{p}_N \leftrightarrow \mathbf{q}_N)$

between 3D and 2D points

$$\bar{\mathbf{q}} = \mathbf{K}\left[R \mid \mathbf{t}\right] {}^{B}\bar{\mathbf{p}} = P\, {}^{B}\bar{\mathbf{p}}$$

- Camera calibration: We want to find **k** and **s** (*off-line*)
  - Estimate full *P* (3x4) matrix, directly from the *N* correspondences (Direct Linear Transform Algorithm, DLT)
  - Then extract *K*, *R*, **t** from *P*
- Pose estimation and tracking: We already have **k** and we want to find only **s** (*on-line*)
  - Non-Linear Least Squares estimation (see next Lecture)

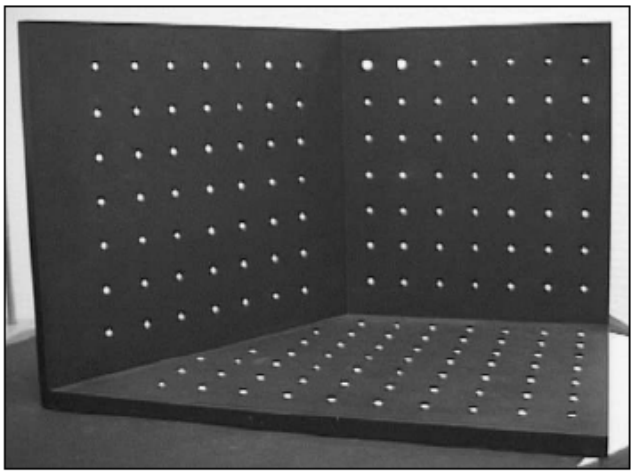
We distinguish here between two fundamental problems in computer vision: pose estimation and camera calibration.
In particular, if we have a set of correspondences between 3D body points and 2D image coordinates (for example after performing feature points matching), we can consider two situations.

If we already know the intrinsic camera parameters, we can estimate the extrinsic pose parameters of the object in space (roto-translation): this is a pose estimation problem, and we use it for 3D tracking.
If we do not know any information both abou object pose and camera parameters, then we have a camera calibration problem; for this task we usually need more than one image, that is multiple view correspondences I between 3D points and 2D coordinates. In order to solve this problem, usually one estimates first the global Projection matrix (for example using the DLT algorithm) and afterwards extract from P the intrinsic parameters k and extrinsic (multiple) pose parameters s1,…sI.
Camera calibration is computationally expensive, and usually performed off-line, in order to estimate the k parameters for subsequent tracking.

## Example of camera calibration and pose estimation


The algorithm finds both **k** *and* the chessboard poses $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_N$
given 3D-2D point correspondences, from the chessboard model to the image.

This is how a 3D calibration pattern looks like:



A calibration pattern is used in order to perform this task almost automatically; in particular, often 3D calibration patterns are used because of a good distribution of points in space, which facilitates the optimization from a numerical point of view, also when the pattern is shown from different points of view. Another very popular calibration pattern is a chessboard with known size [mm] and number of squares. This is also employed in standard open-source libraries, for example the Matlab Calibration Toolbox or the equivalent OpenCv calibration functions.
In these cases, the user has to show the chessboard a number of times (for example 5) and the algorithm should do everything in an automatic way, with minimal user assistance required only to monitor the chessboard detection result (for example, to detect false matchings).

Camera calibration requires more precision in the identification of point correspondences, and need to be done only once for a given camera; that is the reason fo using special objects (patterns) where given points can be reliably identfied in the image.

Pose estimation and tracking is instead a simpler problem that can be eventually solved in real-time, provided of course good point correspondences for more generic objects.

## *Lecture 3 – 3D Pose estimation from Point Correspondences*

### 3D pose estimation problem

The pose estimation problem can be formulated as follows: given a set of *M* correspondences

$({}^{B}\mathbf{p}_1 \leftrightarrow \mathbf{q}_1)$
$({}^{B}\mathbf{p}_2 \leftrightarrow \mathbf{q}_2)$
...
$({}^{B}\mathbf{p}_M \leftrightarrow \mathbf{q}_M)$

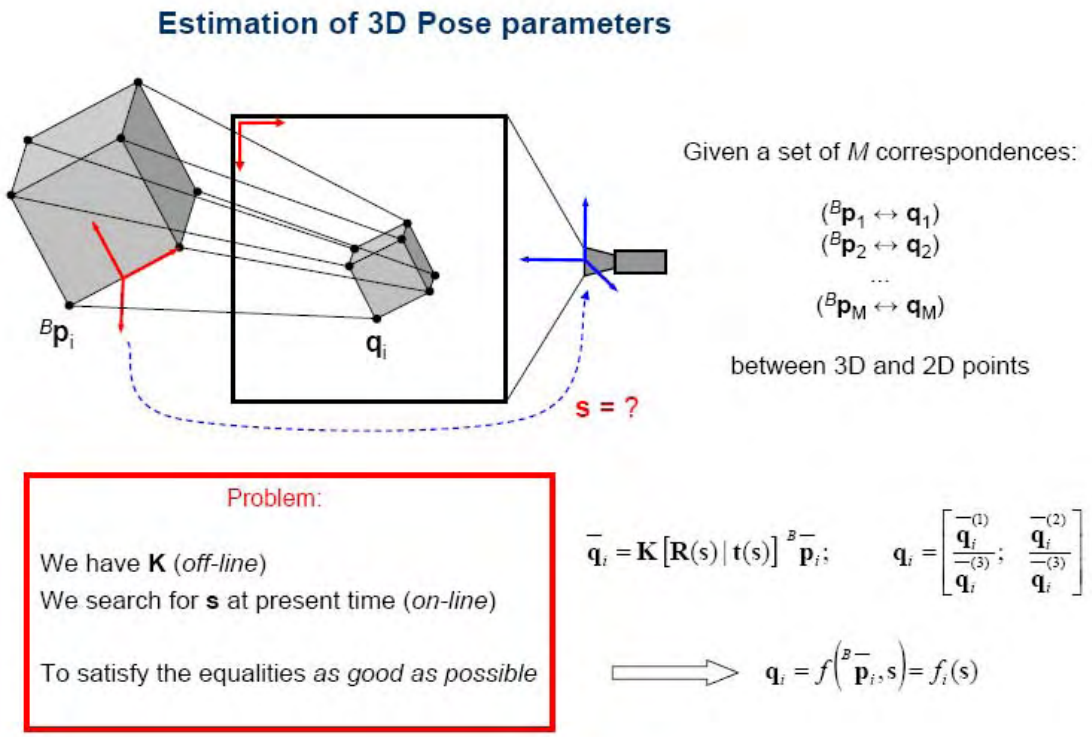
between 3D and 2D points at unknown pose parameter s with given intrinsic parameters k, find s so that the following equalities are satisfied „as well as possible“

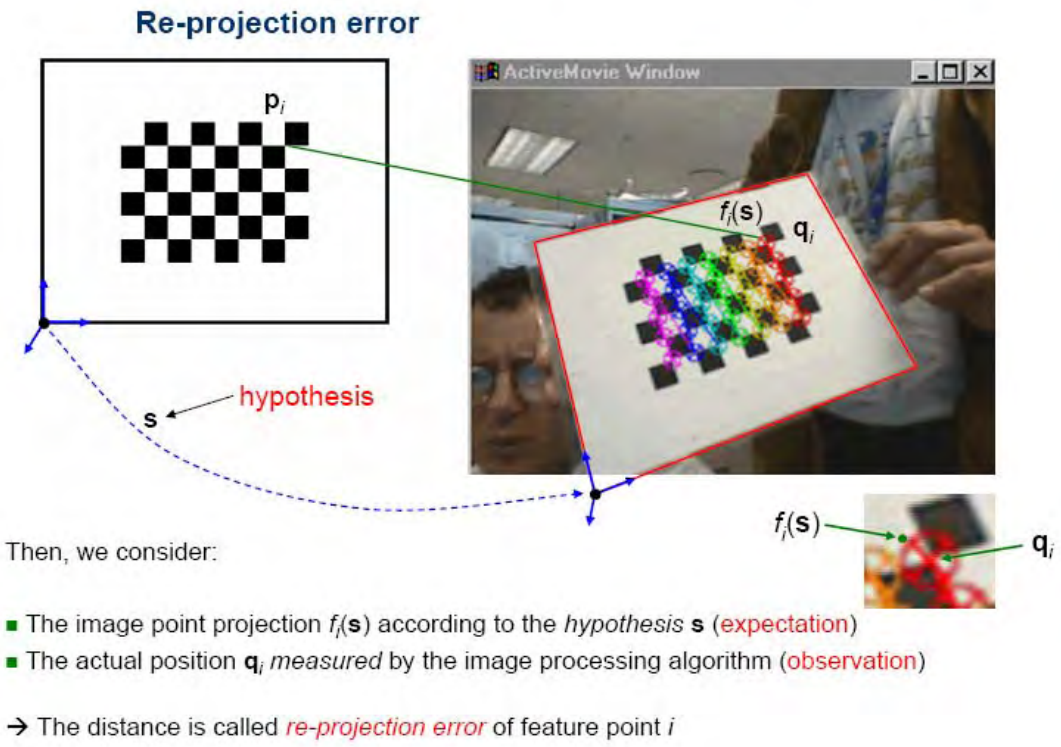
$$q_i \cong f({}^{B}p_i, s)$$

where f(p,s) is the projection from body to screen (in homogeneous coordinates):

$$\overline{q} = K\left[R(s) \quad t(s)\right]{}^{B}\overline{p}$$

and this can be solved as a non-linear least-squares optimization.
In particular, we can observe that the non-linearity of this problem comes from two sources: the rotation matrix is nonlinear in s (whatever representation we use), and the mapping from body to screen (3D/2D) requires finally a division: qx/qz and qy/qz (using homogeneous coordinates).






In a camera calibration or pose estimation problem, we first use an image processing algorithm able to detect and localize feature points like, for example, the corners of a chessboard, with pixel (or subpixel) precision. These positions, qi, are also called measurements, because extracted from the physical observation instrument, that is, our digital camera.

For a given pose hypothesis s, the respective 3D points pi project onto another, expected position fi(s)=f(s,pi).
The distance between expected projection of a 3D point p and the observed value q is called re-projection of feature i:

$$e_i = \| f_i(s) - q_i \|$$

and they are zero only in the ideal case: ei=0 if the pose s is the real one, and the measurements onto the image are perfect (no noise, no distortion effects).

## Least-squares estimation

### Goal of LSE

Cost Function

Define the error function (or *cost* function)

$$C(\mathbf{s}) = \sum_{i=1}^{M} \left\| f_i(\mathbf{s}) - \mathbf{q}_i \right\|^2 = \sum_{i=1}^{M} \left\| e_i(\mathbf{s}) \right\|^2$$ ← Sum of Squared Differences (SSD)

Goal of LSE: minimize C

Goal of Least-Squares pose estimation: find $\mathbf{s}^* = \arg\min_{\mathbf{s}} C(\mathbf{s})$

$\mathbf{s}^*$ is the *minimizer* of $C(\mathbf{s})$ (*optimum* parameter value)

The sum of squared errors is the overall re-projection error (or cost function). This is called sum of squared differences (SSD) error, and it is a very general form of cost function, used for many diferent estimation problems (edge-based, template-based, color-based, etc.)
This error is zero only in the ideal case, but because of residual imaging distortions, noise, etc. the identified points q have some random errors, even when the pose hypothesis s is the "real" one.
The goal of pose estimation is then to minimize this function with respect to s:

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} C(\mathbf{s})$$

This notation means: s* is the *minimizer* of C with respect to s, that is, the value that gives the minimum re-projection error. This should not be confused with min C(s), which is the *minimum value* of C (that is, the value of C in s*)!
The subscript s is useful, especially when the function has more than one argument (for example C(s,t), where t is the time).

## LSE and NLSE

### Two main situations

- The functions $f_i$ are *linear* $f_i = A_i s$

Linear least squares estimation (LSE) → Gauss (~1800)

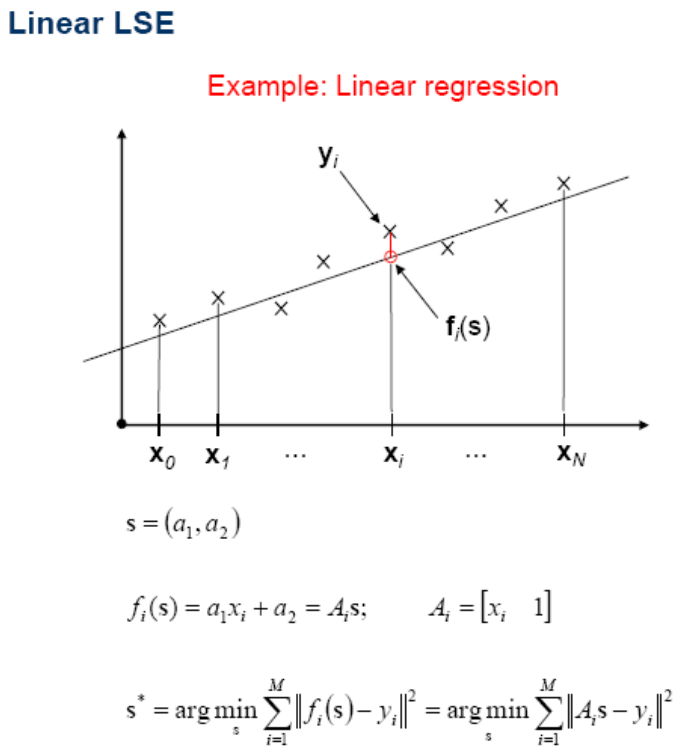
- The functions $f_i$ are *non-linear*

General Nonlinear LSE problem → Gauss-Newton or Levenberg-Marquardt (1963)

... The second problem needs the first, to be solved!

The problem of minimizing an SSD cost function belongs to the general class of Least-Squares Estimation (LSE) problems.
We can make a first gross distinction between linear and non-linear LSE.
Linear LSE are problems where the fi(s) terms in C are linear functions of the state variable: fi(s) = As+b. This is a simpler problem, and has been solved by Gauss around 1800, in the context of computing orbital parameters of some planets from several observations (measurements).
Nonlinear LSE are more general, and they arise in many contexts, in particular in our case of 3D pose estimation from point correspondences. This happens whenever the fi(s) are nonlinear functions of s.
Nonlinear LSE need a more sophisticated algorithm, which is called Gauss-Newton, or the improved version of Levenberg-Marquardt (1963). These algorithms decompose the optimization problem in several steps, each one using a *linearized* version of the function fi(s) around a different point s1,s2,… until the result is stable enough (*convergence*).

## Linear LSE optimization




A typical example in statistics of linear LSE estimation is the linear regression problem. We consider the problem when the observation space q is mono-dimensional, but of course it can be generalized
In this problem, we have a set of 1D observations yi and a linear model (a straight line in the x,y plane) that must fit the data as well as possible.
Therefore, here both model points x and observations are 1D, and a model pose (or state) hypothesis s is given by the line coefficients y=ax+b, s=[a,b].

## Linear LSE

We can write everything in a compact form:

$$\mathbf{q} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} \} M \qquad A = \underbrace{\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_M \end{bmatrix}}_{N} \} M$$

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \sum_{i=1}^{M} \|A_i \mathbf{s} - y_i\|^2 \quad \Longrightarrow \quad \mathbf{s}^* = \arg\min_{\mathbf{s}} \|A\mathbf{s} - \mathbf{q}\|^2$$

Solution: $\boxed{\begin{aligned} \mathbf{s}^* &= A^+ \mathbf{q} \\ A^+ &= (A^T A)^{-1} A^T \end{aligned}}$ $A^+$ is the *pseudo-inverse* of $A$

Pseudo-inverse matrix: $A^+A = I$ and $AA^+ \neq I$ (left-inverse but not right-inverse!)

Because A is rectangular ($M > N$) If A is square ($M = N$) then $A^+ = A^{-1}$

A linear LSE problem can be written in a more compact form, by grouping all observations and linear relationships into large matrices and vectors, therefore getting rid of the sum:

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \|A\mathbf{s} - \mathbf{q}\|^2$$

and this formulation allows to solve the problem (Gauss) using the formula:

$$\mathbf{s}^* = A^+ \mathbf{q}$$
$$A^+ = (A^T A)^{-1} A^T$$

where A+ is called pseudo-inverse of A. Since A is (mxn), where m are the observations and n the state variables (2), we always have a rectangular matrix (more equations than unknown) and the pseudo-inverse is a left-inverse but not a right inverse of A.
NOTE: generally speaking, for every LSE problem it is recommended to have as many observations as possible, much more than state variables. This gives numerical stability in presence of errors, apart from large unpredicted errors (outliers), that we will see in the following.

## Weighted LSE

To give more *confidence* to the measurements that we expect more *reliable*

→ we assign a different *weight* to each measurement, $w_i$

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \sum_{i=1}^{M} w_i \left\| A_i \mathbf{s} - \mathbf{q}_i \right\|^2$$

More compact: $$\mathbf{s}^* = \arg\min_{\mathbf{s}} \left\| W(A\mathbf{s} - \mathbf{q}) \right\|^2$$

$$W = \begin{bmatrix} \sqrt{w_1} & & 0 \\ & \ddots & \\ 0 & & \sqrt{w_M} \end{bmatrix}$$ = Weight matrix (diagonal)

→ Solution: $$\mathbf{s}^* = A_W^+ \mathbf{q}$$ $$A_W^+ = (A^T W A)^{-1} A^T W$$

$A_W^+$ = *weighted pseudo-inverse* of *A*

If we know in advance ("a priori") that for some reason some measurements are more reliable than others (for example, they correspond to feature points that are less ambiguous to identify in an image), then we can include this knowledge into the SSD cost function by using weights:

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \sum_{i=1}^{M} w_i \left\| A_i \mathbf{s} - \mathbf{q}_i \right\|^2$$

where we give a higher coefficient to reliable measurements, and lower to the others: in this way, measurement that we expect to have higher errors will influence less the solution; in the ideal case (no errors) the optimal solution s* remains of course the same.

This problem is called weighted LSE, and it can be applied to linear or nonlinear functions fi(s) as well. For the linear case, the solution is

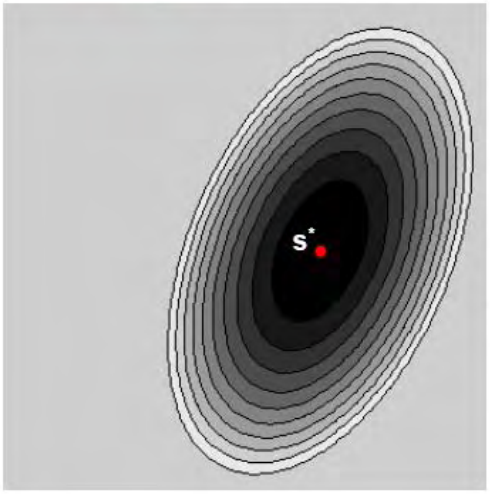
$$\mathbf{s}^* = A_W^+ \mathbf{q}$$

$$A_W^+ = (A^T W A)^{-1} A^T W$$

where (A+W) is the weighted pseudo-inverse of A, and W is the diagonal weight matrix W=diag(w1,…,wN).
NOTE: W is usually chosen to be diagonal, beacuse we consider measurements to be independent one another; an off-diagonal element wij would mean that measurement i depends on measurement j. For nost cases, and in particular for image features extraction techniques, this independence assumption can be considered correct, since features points are independently identified and extracted from an image. In the case of a chessboard, this is actually not 100% true, because positional relationships influence also the image processing and search algorithm; in this case, however, the approximation of independence and equal weights is still good and simple enough to be used for the camera calibration procedure.

## Linear (weighted) LSE cost function

Level sets of $C = \|W(A\mathbf{s} - \mathbf{q})\|^2$




$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \|W(A\mathbf{s} - \mathbf{q})\|^2$$

- No need for $\mathbf{s}_0$
- Closed-form (1-step) solution: $\mathbf{s}^* = (A_w)^+ \mathbf{q}$

**Exercise: try it with Matlab!**

Linear LSE are problems with a well-defined and unique solution, that can be always computed in one-step. Level sets of the cost function are ellipses (hyperellipsoids, in more dimensions), because the function is quadratic in s: it contains only 2nd order polynomial terms (x2, y2, z2, xy, xz, etc.).
Moreover, we do not need for any initial "guess" s0 to find the solution s*, because of the one-step Gauss' formula.

These properties are unfortunately not true for nonlinear problems, as we will see in the following.

## Nonlinear LSE: Gauss-Newton and Levenberg-Marquardt

### Non-linear LSE

We have: $C(\mathbf{s}) = \sum_{i=1}^{M} \left\| f_i(\mathbf{s}) - \mathbf{q}_i \right\|^2 = \sum_{i=1}^{M} \left\| e_i(\mathbf{s}) \right\|^2$

Now the $f_i$ are non-linear

More compact: $\mathbf{s}^* = \arg\min_{\mathbf{s}} \left\| \mathbf{f}(\mathbf{s}) - \mathbf{q} \right\|^2$ with $\mathbf{f} = \begin{bmatrix} f_1(\mathbf{s}) \\ f_2(\mathbf{s}) \\ \vdots \\ f_M(\mathbf{s}) \end{bmatrix} M$

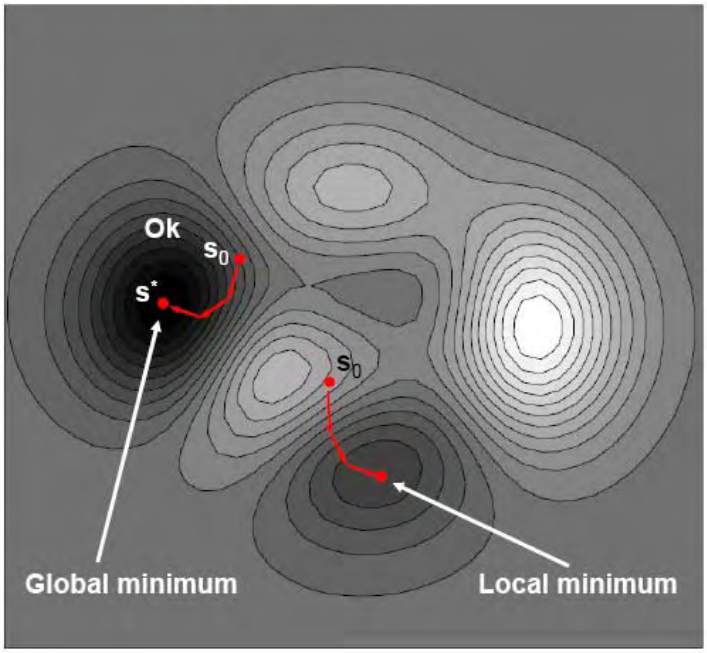
How can we solve this problem?

Nonlinear LSE problems can always be written in the general form

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \left\| \mathbf{f}(\mathbf{s}) - \mathbf{q} \right\|^2$$

where q is the set of real measurements (for example, detected feature points) and f(s) is a vector-valued function, that maps from a state-space vector s to a vector of expected measurements f(s); as for the linear case, the dimension of measurement vector should be much greater than state space, in order to have a reliable and stable estimation s*.

## NLSE optimization




Level sets of $C = \|\mathbf{f}(\mathbf{s}) - \mathbf{q}\|^2$

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} C(\mathbf{s})$$

- Need $\mathbf{s}_0$ (close to $\mathbf{s}^*$)
- Multiple steps solution

This problem is generally much more difficult than the linear one, since the behavior of the cost function C(s) presents several local optima, besides the global one s*. Local optima are ponts where the function has an optimum value (in our case, minimum) but only restricted to a neighborhood s around the point.

## NLSE optimization

There is no closed-form (*1-step*) solution anymore! → We need iterative optimization.

So, we need two things more than for linear LSE

1. An *initial guess* of the pose $\mathbf{s}_0$
2. Solve a sequence of *linearized* problems (in $\Delta\mathbf{s}_k$)

$\mathbf{s}_0 \rightarrow \mathbf{s}_1 = \mathbf{s}_0 + \Delta\mathbf{s}_0$

$\mathbf{s}_1 \rightarrow \mathbf{s}_2 = \mathbf{s}_1 + \Delta\mathbf{s}_1$

$\mathbf{s}_2 \rightarrow \mathbf{s}_3 = \mathbf{s}_2 + \Delta\mathbf{s}_2$

...

*N* linear problems in $\Delta\mathbf{s}_k$

Moreover, even in cases where there is only one optimum s*, we do not have anymore a one-step solution, since the function is not a quadratic form.

But still, we can say that around a local optimum the behavior is nearly quadratic, and this approximation is at best if we take a very small neighborhod of s*.
Therefore, methods for solving nonlinear LSE problems perform a linearization of C(s) around points near s*, that leads to a multiple-step optimization procedure.

Generally speaking, nonlinear optimization needs two main things more: an initial guess s0, which should be close enough to s* (that means, in the basin of attraction of s*); and multiple linearized optimizations, that find increments Ds and refine the solution; and a convergence criterion, that says when the algorithm should stop (for example, when the increment becomes small enough).

### Linearization of NLSE

**f** is a nonlinear *vector* function: N → M

Idea: **f**(**s**) can be *approximated* with a linear function around the point $\mathbf{s}_0$

Linearization = First-order Taylor expansion of **f**

Take a point $\mathbf{s}_0$ and a small *increment* $\Delta\mathbf{s}$

$$\mathbf{f}(\mathbf{s}_0+\Delta\mathbf{s}) = \underbrace{\mathbf{f}(\mathbf{s}_0)+J(\mathbf{s}_0)\Delta\mathbf{s}}_{linear} + \underbrace{O(\Delta\mathbf{s}^2)}_{high-order} \cong \mathbf{f}(\mathbf{s}_0)+J(\mathbf{s}_0)\Delta\mathbf{s}$$

where $J(\mathbf{s}_0)$ is the Jacobian matrix of **f**, computed in $\mathbf{s}_0$:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial s_1} & \cdots & \frac{\partial f_1}{\partial s_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial f_M}{\partial s_1} & \cdots & \frac{\partial f_M}{\partial s_N} \end{bmatrix}$$

Linearizing a function means using the first (linear) term of the Taylor'series expansion of f(s).

The Taylor series is used in order to write f(s) as an infinite sum (series) of polynomials of increasing degree, and uses a reference point s0, which can be arbitrary:

$$\mathbf{f}(\mathbf{s}_0+\Delta\mathbf{s}) = \underbrace{\mathbf{f}(\mathbf{s}_0)+J(\mathbf{s}_0)\Delta\mathbf{s}}_{linear} + \underbrace{O(\Delta\mathbf{s}^2)}_{high-order} \cong \mathbf{f}(\mathbf{s}_0)+J(\mathbf{s}_0)\Delta\mathbf{s}$$

Since we have a vector-valued function (from N- to M-vectors), the first Taylor coefficient is an (NxM) matrix, that contains the first derivatives of f in s0. This is called Jacobian matrix of f in s0.

If we stop the expansion to the first order, and ignore the higher order terms, that contain higher powers of Ds, we obtain an approximation of f(s) around s0, which is linear in Ds.

## Gauss-Newton algorithm

The linearized problem (LSE) is: $\Delta \mathbf{s}^* = \arg\min_{\Delta s} \left\| J(\mathbf{s}_0)\Delta \mathbf{s} - (\mathbf{q} - \mathbf{f}(\mathbf{s}_0)) \right\|^2$

And the solution is: $\Delta \mathbf{s}^* = J^+ \cdot (\mathbf{q} - \mathbf{f})$ $\quad \mathbf{s}^* = \mathbf{s}_0 + \Delta \mathbf{s}^*$

But, since this is an approximation of **f**(**s**), **s*** will not be the real optimum!

→ We need to iterate (repeat)

1. Set $\mathbf{s}_0 + \Delta \mathbf{s}^* \rightarrow \mathbf{s}_1$
2. Repeat the computation above in $\mathbf{s}_1$

...until the increment Δ**s*** is small enough: $\left\| \Delta \mathbf{s}^* \right\| < \varepsilon$

By using this approximation in C(s), we have a linear LSE, that can be solved with the Gauss method.

$\Delta \mathbf{s}^* = J^+ \cdot (\mathbf{q} - \mathbf{f})$ $\quad \mathbf{s}^* = \mathbf{s}_0 + \Delta \mathbf{s}^*$

The linear approximation of f is good only in a small neighborhood of s0, therefore the solution to the linearized LSE will be only an approximate solution to the real nonlinear problem, and then we will need to repeat the process (iterate): compute Jacobian matrix, and update the solution.
This is the Gauss-Newton algorithm for nonlinear LSE problems. The algorithm can be terminated when ||Ds|| is small enough.

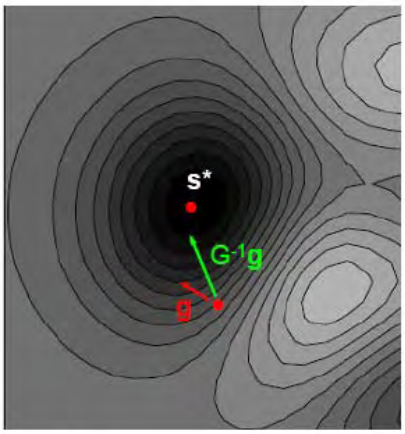
## Gauss-Newton matrix

This is the Gauss-Newton Algorithm:

- Solve several *Gauss* (LSE) problems
- Increment $\Delta\mathbf{s}^*$ = *quasi-Newton* step

We can write:

$$\Delta\mathbf{s}^* = (J^T J)^{-1}\left[J^T(\mathbf{q}-f)\right] = G^{-1}\mathbf{g}$$




$G = (J^T J)$ is the *Gauss-Newton matrix* (NxN)

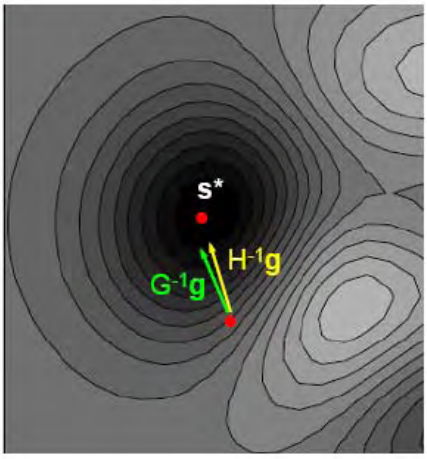
$\mathbf{g}$ is the neg-*gradient* of $C$ $\quad \mathbf{g} = J^T(\mathbf{q}-f) = -\frac{1}{2}\begin{bmatrix}\frac{\partial C}{\partial s_1}\\ \cdots \\ \frac{\partial C}{\partial s_N}\end{bmatrix}$

## Property of Gauss-Newton matrix

Property of *G*

$G = (J^T J)$ approximates
the (*NxN*) *Hessian matrix* of $C$ around $\mathbf{s}^*$

$$H_C = \begin{bmatrix}\frac{\partial^2 C}{\partial s_1^2} & \cdots & \frac{\partial^2 C}{\partial s_1 \partial s_N} \\ \cdots & \cdots & \cdots \\ \frac{\partial^2 C}{\partial s_1 \partial s_N} & \cdots & \frac{\partial^2 C}{\partial s_N^2}\end{bmatrix} \cong (J_{\mathbf{f}}^T J_{\mathbf{f}}) = G$$



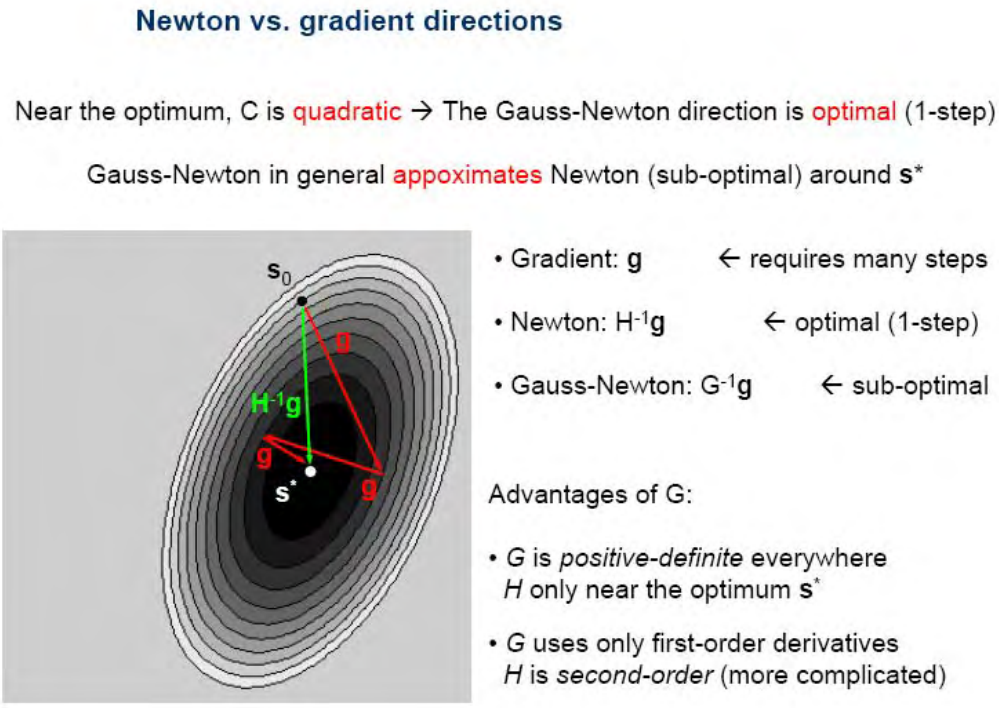

→ We have a quasi-Newton optimization step: $\Delta\mathbf{s}^* \cong H_C^{-1} \cdot \mathbf{g}_C$

The matrix (J'J) is also called Gauss-Newton matrix, while the other term J(q-f) is also the neg-gradient of the cost function C(s) (it can be verified).
This is similar to the more general (but also more complex) Newton optimization method for nonlinear functions:

$$\Delta\mathbf{s}^* = H_C^{-1} \cdot \mathbf{g}_C$$

where HC is the Hessian matrix, and gC the gradient of C in s0; the Hessian matrix is the matrix of second derivatives of C.
NOTE: remember that C is a scalar-valued function, from N to 1 dimension; therefore the first derivatives matrix (Jacobian) of C is also called gradient, and is a Nx1 vector; while second derivatives are NxN, and constitute a symmetric matrix H, called Hessian.
This is not to be confused with the Jacobian of f, Jf, which is a MxN matrix of a vector-valued function!

The Gauss-Newton matrix of C is obviously defined only if C is an SSD cost function C = ||f-q||2, while H is a more general definition (also for other nonlinear functions C). But in the case of SSD functions, Gauss-Newton (Jf'Jf) is actually an approximation of H, and it is good around the optimal point s*.
This is why this method is called Gauss-Newton: solve a sequence of linear LSE problems (Gauss) by using an update rule that approximates Newton optimization step.




If the problem were linear, the Gauss, Newton and Gauss-Newton update steps are the same, and the solution is found in one-step.
The simple neg-gradient -g direction, is also called steepest-descent direction (the direction where the slope of the function is maximal) at point s0. This direction is not optimal, and requires many descent steps, when level sets are ellipsoids with high eccentricity, that is very different axes length.

Generally speaking, for nonlinear LSE Gauss-Newton approximates the Newton direction, but has some important advantages: first it does not need to compute second derivatives; second, it gives a matrix J'J which is always positive-semidefinite (i.e. the eigenvalues are always >=0), whereas H can be also indefinite (with eigenvalues of different signs), and give bad problems when starting far from the optimum s*.

## Levenberg-Marquardt Algorithm

Problem: what happens if *G* is *singular* (it cannot be inverted)?

Levenberg-Marquardt modification:

$$\Delta \mathbf{s}^* = (J^T J + \lambda I)^{-1} J^T (\mathbf{q} - f)$$

Add a positive term $\lambda > 0$ to combine gradient and GN directions
→ We have more *robustness*:

- If $\lambda$ is big, we have the *steepest-descent* solution $\Delta \mathbf{s}^* \cong \frac{1}{\lambda}\mathbf{g}$

  → we do not trust *G*, so we go more in the *gradient* direction: **g**

- If $\lambda$ is small, we have the Gauss-Newton step

  → we trust more *G*, so we go in the Gauss-Newton direction: ($G^{-1}\mathbf{g}$)

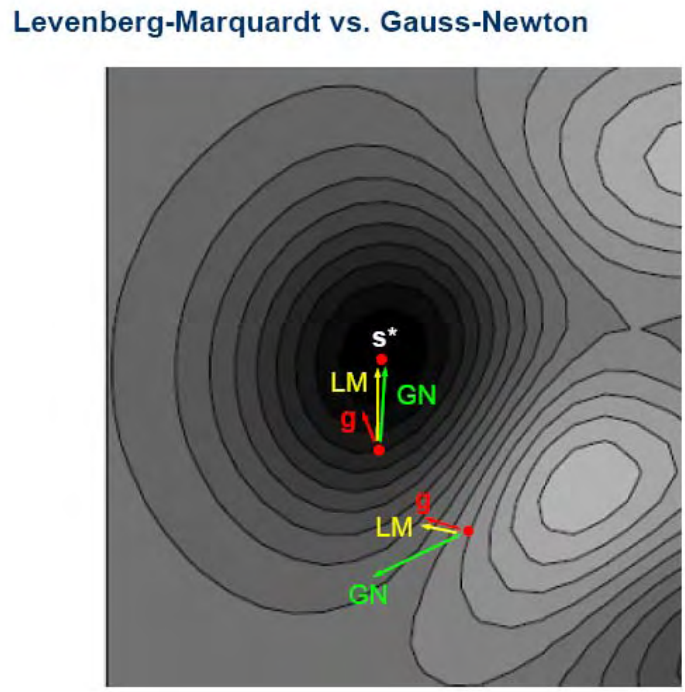
Of course, also G can become singular (with some null eigenvalues) far from the optimum, and therefore an improved algorithm has been developed by Levenberg and Marquardt (1963).

$$\Delta \mathbf{s}^* = (J^T J + \lambda I)^{-1} J^T (\mathbf{q} - f)$$

The LM correction to the Gauss-Newton matrix is given by adding a term lI, with l>0, so that the Gauss-Newton matrix becomes positive definite, and can be safely inverted.
The choice of l influences our optimization procedure: if l is high, so that the second term predominates over the first, the optimization step goes more in the direction of the steepest-descent (i.e. the gradient of C(s))

$$\Delta \mathbf{s}^* \cong \frac{1}{\lambda}\mathbf{g}$$

while a low value for l means going more in the GN direction.




Therefore, Levenberg-Marquardt specifies also the rules for choosing l, and modifying it during the optimization (so that we have a kind of “adaptation” rule). This makes the algorithm robust and flexible for many situations.
The idea is to keep a high, safe value for l when the GN matrix is close to singular, or (which is analogous) when the update step gives not a good result (no descent of C), while decreasing l when the behavior of GN is better, that is when we can “trust” more GN, which converges faster and better than the steepest descent.

These two situations happen, roughly, when we are far from the optimum (GN is badly defined), and when we are near (GN is ok, and the cost function C behaves more in a “quadratic” way, with almost ellipsoidal level sets).

Finally, the LM rule is: compute the Ds update with the current value of l, and, before applying it, test:

- If the updated value of C(si+Ds) is not decreasing w.r.t. C(si), then reject the update and increase l*10
- If the new value is ok (descreasing), then accept the update si+Ds and reduce l/10.

It can be experimentally observed, for most functions, that l will be kept high in the first optimization steps (if we start with s0 far from the optimum s*), while it will decrease near to the optimum.

### Levenberg-Marquardt vs. Gauss-Newton

**Gauss-Newton optimization step**

At iteration *k*: given $\mathbf{s}_k$

1. Solve for the increment $\Delta\mathbf{s}^*$ with the GN formula $\Delta\mathbf{s}^* = (J^T J)^{-1} J^T (\mathbf{q} - f)$
2. Set the new parameters $\mathbf{s}_{k+1} = \mathbf{s}_k + \Delta\mathbf{s}^*$

**Levenberg-Marquardt optimization step**

At iteration *k*: given $\lambda$ and $\mathbf{s}_k$

1. Solve for the increment $\Delta\mathbf{s}^*$ with the LM formula $\Delta\mathbf{s}^* = (J^T J + \lambda I)^{-1} J^T (\mathbf{q} - f)$
2. Try the new parameters $\mathbf{s}_{k+1} = \mathbf{s}_k + \Delta\mathbf{s}^*$ :

   2a. If the cost function is decreasing $C(\mathbf{s}_{k+1}) < C(\mathbf{s}_k)$ accept $\mathbf{s}_{k+1}$ and decrease $\lambda \rightarrow \lambda/10$

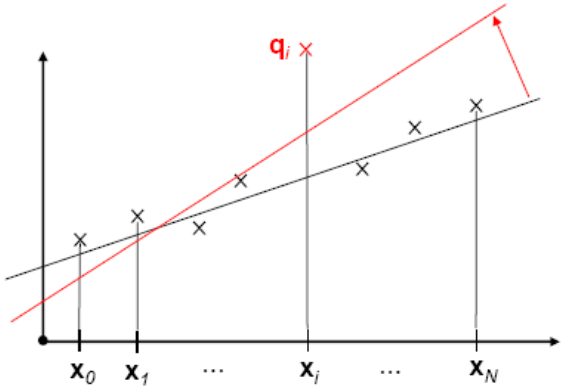
   2b. Else, reject $\mathbf{s}_{k+1}$, increase $\lambda \rightarrow 10\lambda$ and try again from $\mathbf{s}_k$

## Robust LSE

### Problem: Outliers

Outlier = a single measurement that falls out of the *average statistics* of the sample

What happens if we have outliers ?




Even a single bad measurement $\mathbf{q}_i$ can modify the result!

→ the standard LSE and NLSE estimators are not robust to outliers

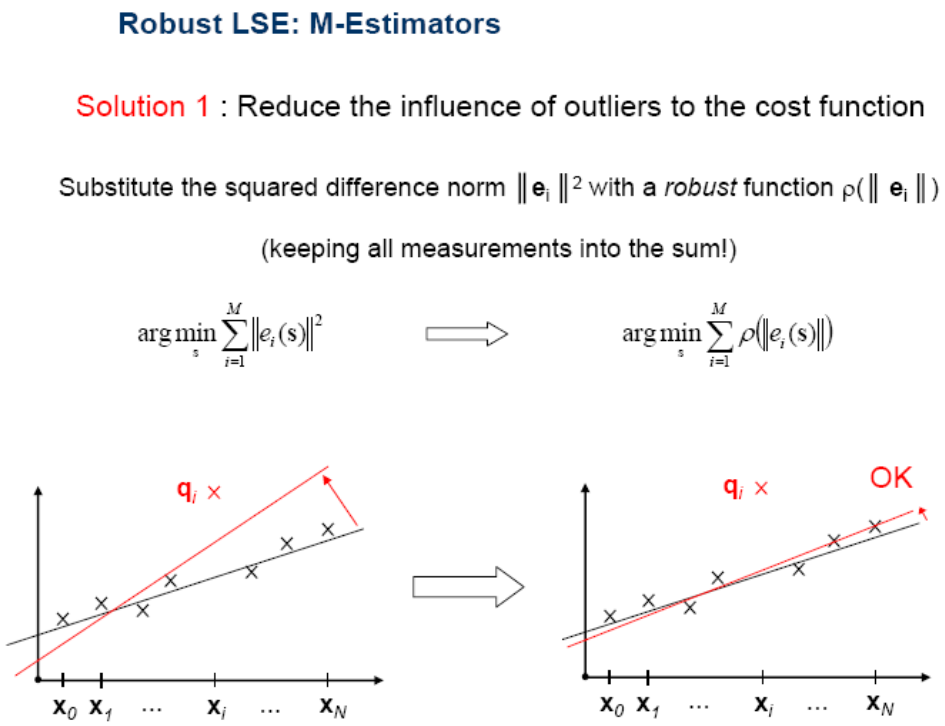
Outliers are defined in statistics as sample points that fall outside the expected sample distribution.

This defnition, however, assumes one fundamental thing: to define first a model of sample distribution.

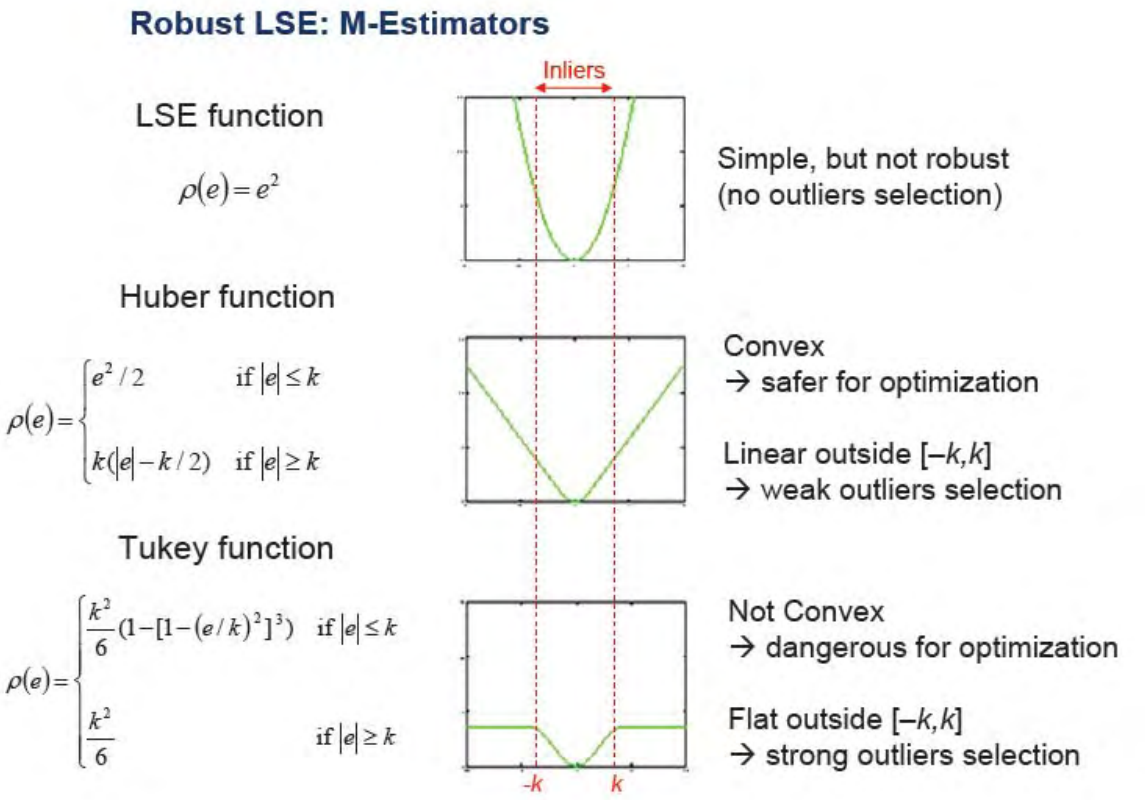
In fact, if we model our data with a linear relationship + noise (for example, Gaussian), then we can say that the point qi is an outlier: its error with respect tot the underlying line model is too big with respect to the other points of the set.
But of course, if our model is different (a parabola, for example), we could call different points of the same set “outliers”.

Once we define the underlying model (linear or not), we can consider the problem: how much the LSE estimation is influenced by outliers?
This is the problem of robust statistics: standard (linear and nonlinear) LSE are not robust to outliers: even one single outlier can heavily modify the state estimate.

***M-estimators***




We have two main approaches to perform robust LSE estimation.
One solution is to consider still all points into the sum C(s), but to modify the function in order to reduce the influence of outliers: this amounts to substitute the SSD (squared error) terms with a different function, that behaves like SSD only for small error values, while reducing the value for high errors, above a pre-defined theshold (outlier threshold).




These functions are called M-estimators.
Two well-known examples, used in computer vision, are the Tukey and Huber functions.
The Huber function becomes linear beyond the threshold, which is reduced with respect to the quadratic SSD function. Its shape is still convex, like SSD, therefore keeping the same convergence properties for the cost function C(s) (region of convergence, etc.).
The Tukey function becomes flat beyond the threshold, which reduces more dramatically outliers influence; but the convexity property is not respected anymore, and therefore the optimization algorithm can fail to converge if s0 is far from the optimum (smaller convergence area).
A good strategy therefore is to use the Huber function at the beginning, when far from the optimum, while "switching" to the Tukey function afterwards.

## Levenberg-Marquardt with M-Estimators

Robust LSE: $\mathbf{s}^* = \arg\min_{\mathbf{s}} \sum_{i=1}^{M} \rho(\|e_i(\mathbf{s})\|)$

Can we still apply Levenberg-Marquardt algorithm?

**Yes**: use the *Weight matrix* $\sum_{i=1}^{M} \rho(\|e_i(\mathbf{s})\|) = \sum_{i=1}^{M} w_i \|e_i(\mathbf{s})\|^2$ with $w_i = \frac{\rho(e_i)}{\|e_i\|^2}$

→ Re-weighted NLSE

→ Levenberg-Marquardt step: $\Delta\mathbf{s}^* = (J^T W J + \lambda I)^{-1} J^T W \cdot \mathbf{e}$

NOTE: Re-weighted = W must be updated at every step $\mathbf{s}_k$ !

The Levenberg-Marquardt (or Gauss-Newton) algorithms can be still applied to the case of M-estimators. In fact, even though the cost function is not anymore an SSD, we can re-write it as a weighted SSD by introducing weights

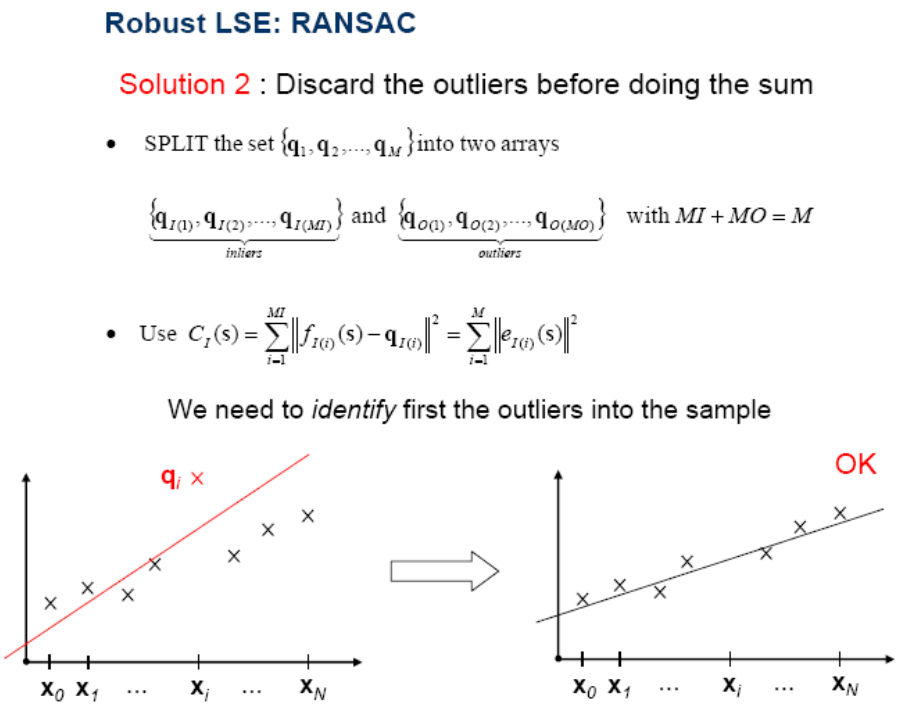
$$w_i = \frac{\rho(e_i)}{\|e_i\|^2}$$

This time the weights are not constant, so we have to update them after each step. This is called re-weighted nonlinear LSE, and the algorithm is the re-weighted Levenberg-Marquardt, which uses the weighted pseudo-inverse of J in place of J+

$$\Delta\mathbf{s}^* = (J^T W J + \lambda I)^{-1} J^T W \cdot \mathbf{e}$$

and W is the diagonal matrix of all weights wi.

***RANSAC***

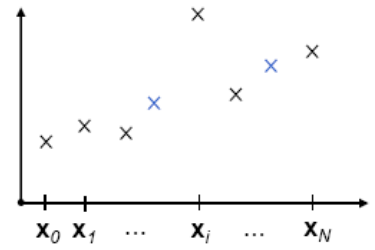



The other alternative is to detect and discard the outliers from the sum, before doing the optimization, and then perform a standard LSE over the remaining points (inliers).

A standard procedure to perform this task is called RANSAC (Random Sample Consensus), and is based on a random search for outliers, by picking many small, random subsets of sample points, and trying the consequent hypotheses on s to detect outliers. The best subset, with less outliers, will be reported and used to discard outliers, and provides also an initial estimate s0, that can be used afterwards by the LSE optimization.

## Robust LSE: RANSAC

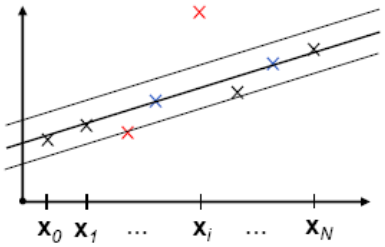
RANSAC : RANdom SAmple Consensus




1 - Take a random sample

The smallest possible set to do LSE!
→ Here: 2 points

NO need for Levenberg-Marquardt!




2 – Estimate **s** and compute outliers: 2

The sample subset used by RANSAC should be as small as possible, enough to perform a non-ambiguous estimation (1 solution only). This is because we need many random evaluations.

For example, for the linear regression problem, a subset of 2 points is enough for estimating a model hypothesis (straight line = 2 state parameters). By using a minimal subset, the estimation is doen with simpler and faster methods, instead of the full LSE; in this case, the solution is also exact (the line that goes through the 2 points).

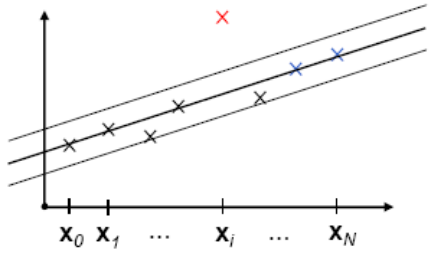
Once a pose hypothesis is generated, the remaining points are tested against this line, and outliers are computed; a common procedure is to set a threshold proportional to the standard deviation $\sigma$ of the error computed for all points (apart the 2 points of the subsample). If a point has an error more than, for example, $3\sigma$, it will be classified as outlier with respect to this hypothesis.

This random evaluation of outliers is repeated many times, in order to guarantee a high probability of detecting the "best" case, with the minimal number of outliers; it would be of course too expensive to evaluate hypotheses for all possible subsets, if the sample set is large.

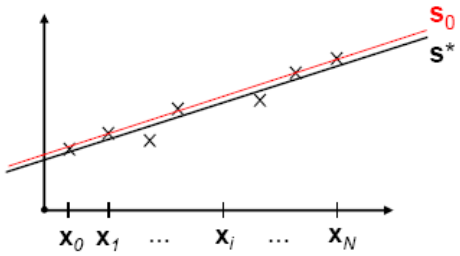
Of course, this means two things: first, we are not 100% sure that we will eventually get the “best” case, and moreover, there can be more than one best case, with a minimal number of outliers.
Therefore, we must rely on probability theory, by computing the number of trials that give for example a P=0.99 of detecting the best case(s) by generating random choices. For a given desired P, the number of trials required will increase dramatically with the dimension of both the sample set and the random subsets.

## Robust LSE: RANSAC

RANSAC : RANdom SAmple Consensus




Keep the best case and remove the outliers (1)




Refine the estimation using *all* the inliers (Levenberg-Marquardt)

At the end, by keeping (one of) the best case, we can eliminate outliers, and we also have an initial estimate s0 computed with the best point subset, that can be used by the final, LSE estimation, using all inliers.

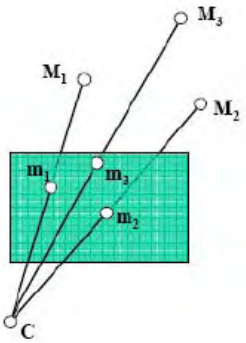
## Robust LSE: RANSAC

How do we apply RANSAC to 3D pose estimation?

Repeat

- Take at random 3 points from the N correspondences $({}^{B}\mathbf{p}_i \leftrightarrow \mathbf{q}_i)$
- Compute the pose **s** with a P3P algorithm (Perspective from 3 Points)
- Compute the outliers from the remaining points

...until we find the „best case“ → Result: $\mathbf{s}_0$ + inliers




P3P: Perspective from 3 Points algorithm

It is based on simple geometry → no Levenberg-Marquardt!

For our case of 3D pose estimation problem, the minimal number of points for each RANSAC hypothesis is 3: with 3 body-to-screen point correspondences (3D/2D), one can compute the pose parameters s without ambiguity using standard geometry considerations, apart from very few “pathological” configurations that can be detected and avoided. This is called P3P algorithm: Perspective from 3 Points.

## Robust LSE: RANSAC vs. M-Estimators

RANSAC

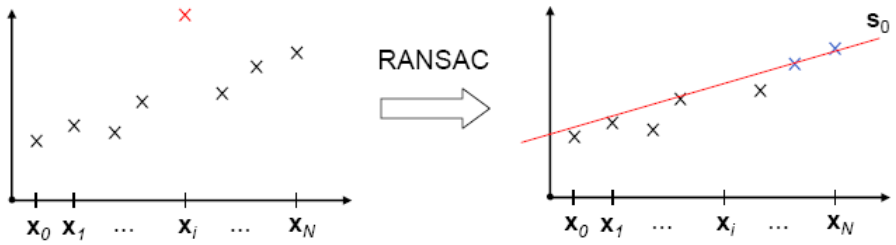
- Does not require initialization
- But: Can fail because it is based on random sampling (can keep bad points)

M-Estimators

- Deterministic
- But: require initial guess $\mathbf{s}_0$, and can fail if $\mathbf{s}_0$ is far from the optimum $\mathbf{s}^*$

RANSAC is a powerful method for eliminating outliers, but has no 100% guarantee to get the best solution, and still some outliers can remain into the sample. Therefore, after the RANSAC procedure, one can perform a robust LSE estimation using M-estimators, starting from the s0 hypothesis given by RANSAC itself.

## Robust LSE: RANSAC + M-Estimators

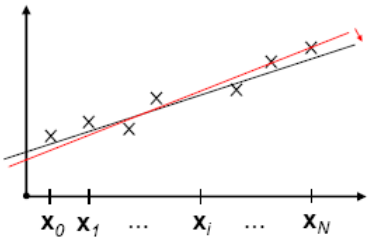



M-Estimator

More robustness:

Combine RANSAC + M-Estimators

1. RANSAC gives $\mathbf{s}_0$
2. Use M-Estimator starting from $\mathbf{s}_0$



## Resume

ALGORITHM

(very) Robust 3D pose estimation from point correspondences

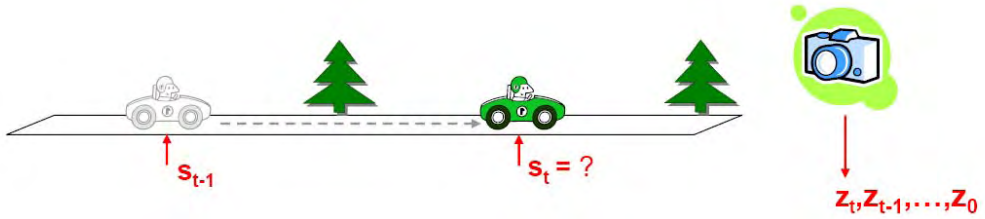
1. Take all the 3D/2D point correspondences: $({}^{B}\mathbf{p}_i \leftrightarrow \mathbf{q}_i)$
2. Run RANSAC (with P3P algorithm) to remove outliers and find $\mathbf{s}_0$
3. Start Least-Squares estimation from $\mathbf{s}_0$ with Levenberg-Marquardt and M-Estimators, and find $\mathbf{s}^*$

By combining both techniques, we obtain a general, robust and complete algorithm for model-based statistical estimation of parameters (not only 3D pose, of course).

**What do we mean by "tracking"?**

Given a sequence of images in time: $I_0, I_1, I_2, \ldots, I_t$

Tracking = Estimate $\mathbf{s}_t$, the object state at time *t*, by considering:



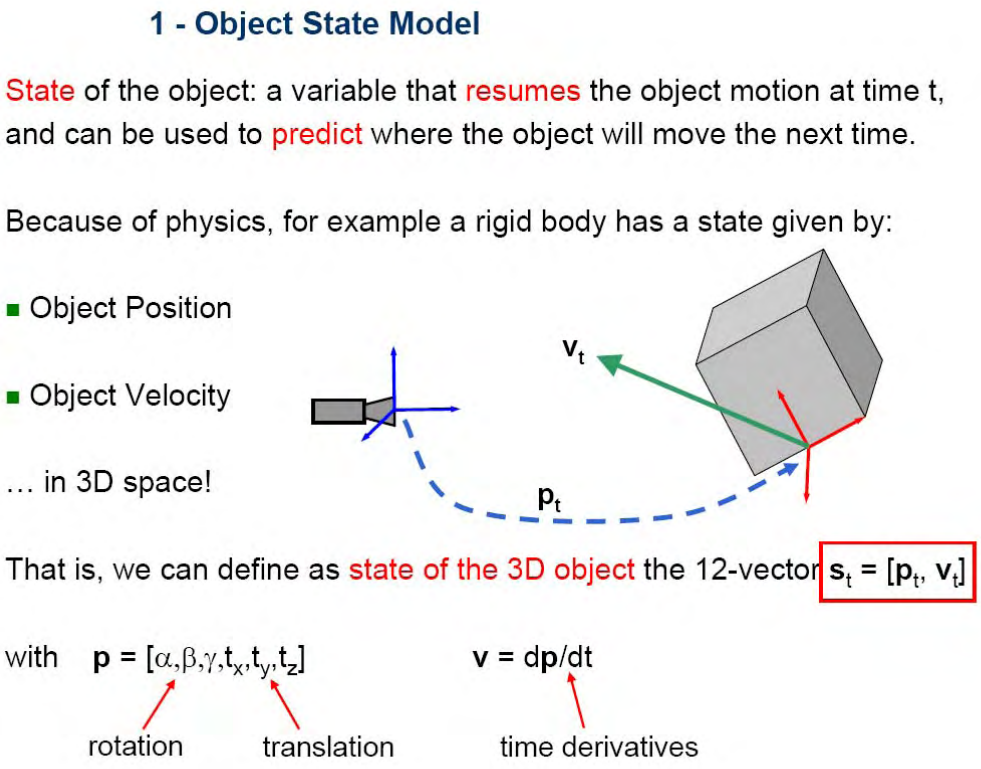

- The previous estimate of the state: $\mathbf{s}_{t-1}$
- The current measurement (observation): $\mathbf{z}_t$
- All the previous measurements (observation history): $ZH_{t-1} = (\mathbf{z}_{t-1}, \ldots, \mathbf{z}_0)$

Visual tracking is a general name given to the problem of estimating the state of (one or more) objects in time, using a sequence of visual measurements, that is, a sequence of images and image processing operations.

Generally speaking, the term "tracking" is used to define all sequential estimation problems that involve one or more sensors, providing a sequence of measurements in time.

In our case, the sensor is visual, and given by our CCD camera; the "raw" acquired signal is the image, which is discrete in space (pixels) and intensity (gray- or color-values).

**Object state model**




The state of the system is represented by a variable s that can be used to resume the object motion, and it should be "rich" enough to enable single-step state predictions over time, by using a probabilistic motion model:

$s_t = f(s_{t-1})$

Because of physics' laws, for an object with mass we need to define its state as s=[p,v] position + velocity variables; in the case of 3D tracking, a rigid object has for example 6+6 = 12 state variables, where the velocity is the time derivative of p (roto-translation pose parameters). If we define the state just as s=p, we can still design a tracking system, but at the price of motion models that need two values of s for each prediction: $s_t = f(s_{t-1},s_{t-2})$.

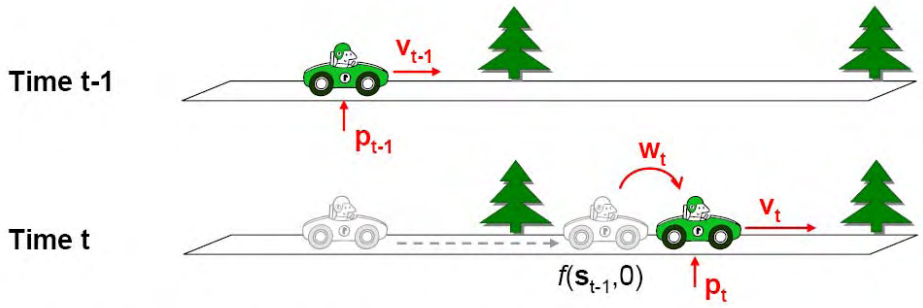
NOTE: the possibility of using only the last value $s_{t-1}$ for a prediction is also called *Markov* property: the probability distribution of next state is conditioned only on the last state, and not on the previous ones.

**Dynamic model**

## 2 - Motion Model

Motion model = a probabilistic model that describes the state evolution in *time*.

$\mathbf{s}_t$ is: (deterministic function of $\mathbf{s}_{t-1}$) + (random component $\mathbf{w}$): $\mathbf{s}_t = f(\mathbf{s}_{t-1}, \mathbf{w}_t)$




- Deterministic law *f*: this comes from physics

→ in absence of perturbation (**w**=0), it gives the state prediction.

- Random variable **w**: it gives the uncertainty of our prediction.

Motion models always consist of a deterministic part and a random component.

$$\mathbf{s}_t = f(\mathbf{s}_{t-1}, \mathbf{w}_t)$$

The deterministic part f(s,0) is also called prediction: it is a function (linear or nonlinear) that predicts the next "most probable" state value, that is, in absence of expected perturbations w=0; the random component w is also called motion noise, and models unexpected perturbations. Usually (but not always) the noise components is additive with respect to the prediction

$$\mathbf{s}_t = f(\mathbf{s}_{t-1}) + \mathbf{w}_t$$

We can consider here some typical motion models. For sake of simplicity, we will define them for the case where the object is a point mass, not a rigid 3D object; the extension to rigid object is a little bit more complex, and involves different terms for linear and angular velocities (rigid body kinematics), but the ideas are still the same.

## Motion Model – Examples

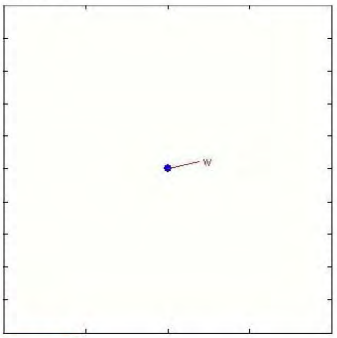
### 1 - Brownian Motion

We assume that the object, in the time interval $\Delta t$ between *t-1* and *t*, has:

- random velocity $\mathbf{v}_t=\mathbf{w}_t$, normally distributed
- the velocity is independent from time to time (white noise)

$\mathbf{p}_t = \mathbf{p}_{t-1} + \mathbf{w}_t \Delta t$

$\mathbf{w}_t$ = velocity between *t-1* and *t*: $P(\mathbf{w}_t) = \text{Gauss}(0, \Lambda_w)$




Here, the state is only the position: $\mathbf{s}_t = \mathbf{p}_t$ !

→ The probabilistic model is $P(\mathbf{s}_t|\mathbf{s}_{t-1}) = \text{Gauss}(\mathbf{s}_{t-1}, \Lambda_w \Delta t^2)$

Brownian motion is the most simple to model and, at the same time, the most difficult to track reliably. It consists in a purely random velocity vector at any time, which is also a white process: noise at time t is independent from noise at time t-1.
In this case, we can define the state as s=p, position only, since there is no prediction available about v, and no correlation in time.

$\mathbf{p}_t = \mathbf{p}_{t-1} + \mathbf{w}_t \Delta t$

w is assumed to be a zero-mean, Gaussian random variable with covariance matrix $\mathbf{\Lambda}_w$ The corresponding probabilistic model for s=p is then

$P(\mathbf{s}_t|\mathbf{s}_{t-1}) = \text{Gauss}(\mathbf{s}_{t-1}, \Lambda_w \Delta t^2)$

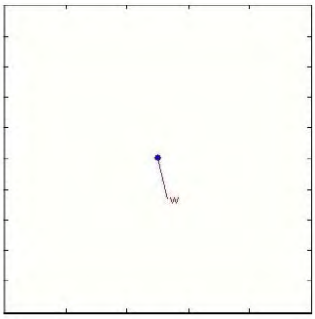
This model is good in two situations: for objects with very low mass (no inertia), or when the sample time Dt is too high (low tracking rate); in the second case, in fact, between two sample times there is enough time also for a heavy target to make any “maneuvering”, and change velocity in an arbitrary way; in other words, we loose part of our prediction abilities due to a too long Dt.

We can also say that Brownian motion gives a probabilistic model with a higher entropy (uncertainty) value for the position variable p; it can be shown that this uncertainty is proportional to Dt.

## Motion Model - Examples

### 2 - WNA = White Noise Acceleration

We assume that the object, in the time interval $\Delta t$ between *t-1* and *t*, has acceleration $\mathbf{w}_t$ (=noise), normally distributed




$$\mathbf{p}_t = \mathbf{p}_{t-1} + \mathbf{v}_{t-1}\,\Delta t + \mathbf{w}_t\,\Delta t^2$$
$$\mathbf{v}_t = \mathbf{v}_{t-1} + \mathbf{w}_t\,\Delta t$$

$\mathbf{w}_t$ = acceleration between *t-1* and *t* :

$P(\mathbf{w}_t) = \text{Gauss}(0, \Lambda_w)$

We can write : $\mathbf{s}_t = A\,\mathbf{s}_{t-1} + B\,\mathbf{w}_t$, $A = \begin{bmatrix} I & I\Delta t \\ 0 & I \end{bmatrix}$ $B = \begin{bmatrix} I\Delta t^2 \\ I\Delta t \end{bmatrix}$

→ The probabilistic model is $P(\mathbf{s}_t|\mathbf{s}_{t-1}) = \text{Gauss}\,(A\,\mathbf{s}_{t-1}, B\Lambda_w B^T)$

For objects with mass and reasonably fast sample times, we can use the White Noise Aceleration (WNA) model, with much better prediction abilities.
In this case, in absence of unpredicted external forces or internal maneuvering, the trajectory should be rectilinear with the current velocity v, therefore the predicted position is st+1 = st+vt*Dt; an unpredicted acceleration a can be seen as a perturbation, and modeled as motion noise w=a.

In a general WNA model, full motion equations can be written as

$$\mathbf{p}_t = \mathbf{p}_{t-1} + \mathbf{v}_{t-1}\,\Delta t + (1/2)\,\mathbf{w}_t\,\Delta t^2$$
$$\mathbf{v}_t = \mathbf{v}_{t-1} + \mathbf{w}_t\,\Delta t$$

where w is again a Gaussian white noise with covariance matrix $\mathbf{\Lambda}_w$

This equation can be compactly written in the state s=[p,v] and the corresponding probabilistic model is also a Gaussian,

$P(\mathbf{s}_t|\mathbf{s}_{t-1}) = \text{Gauss}\,(A\,\mathbf{s}_{t-1}, B\Lambda_w B^T)$

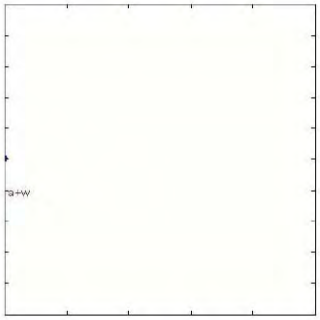
In a WNA model, it can be seen that the uncertainty on pt is lower than for Brownian motion, and proportional to Dt2.

## Motion Model - Examples

### 3 - Constant acceleration + perturbation

The object, in the time interval $\Delta t$ between *t-1* and *t*, has:

- constant acceleration **a** +
- random perturbation $\mathbf{w}_t$ of **a**, normally distributed



$$\mathbf{p}_t = \mathbf{p}_{t-1} + \mathbf{v}_{t-1}\,\Delta t + (\mathbf{a}+\mathbf{w}_t)\,\Delta t^2$$
$$\mathbf{v}_t = \mathbf{v}_{t-1} + (\mathbf{a}+\mathbf{w}_t)\,\Delta t$$

$\mathbf{w}_t$ = acceleration perturbation between *t-1* and *t*:
$P(\mathbf{w}_t) = \text{Gauss}(0, \Lambda_w)$

We can write : $\mathbf{s}_t = A\,\mathbf{s}_{t-1} + C + B\,\mathbf{w}_t$ $\quad A = \begin{bmatrix} I & I\Delta t \\ 0 & I \end{bmatrix}$ $\quad B = \begin{bmatrix} I\Delta t^2 \\ I\Delta t \end{bmatrix}$ $\quad C = \begin{bmatrix} \mathbf{a}I\Delta t^2 \\ \mathbf{a}I\Delta t \end{bmatrix}$

The probabilistic model is $P(\mathbf{s}_t|\mathbf{s}_{t-1}) = \text{Gauss}\,(A\,\mathbf{s}_{t-1}+C,\ B\Lambda_w B^T)$

Another motion model, similar to the WNA, can be considered when the object is falling under gravity (ballistic model); in this case, we can still use a WNA, but the average acceleration is not zero but –g, the downwards gravity acceleration.
Perturbations are in this case random forces due to air viscosity, or other sources, which add to the total acceleration a=w-g.

$$\mathbf{p}_t = \mathbf{p}_{t-1} + \mathbf{v}_{t-1}\,\Delta t + (\mathbf{a}+\mathbf{w}_t)\,\Delta t^2$$
$$\mathbf{v}_t = \mathbf{v}_{t-1} + (\mathbf{a}+\mathbf{w}_t)\,\Delta t$$

This correspond to a probabilistic model with the same uncertainty (noise) characteristics for the random component, but a different prediction (deterministic part).

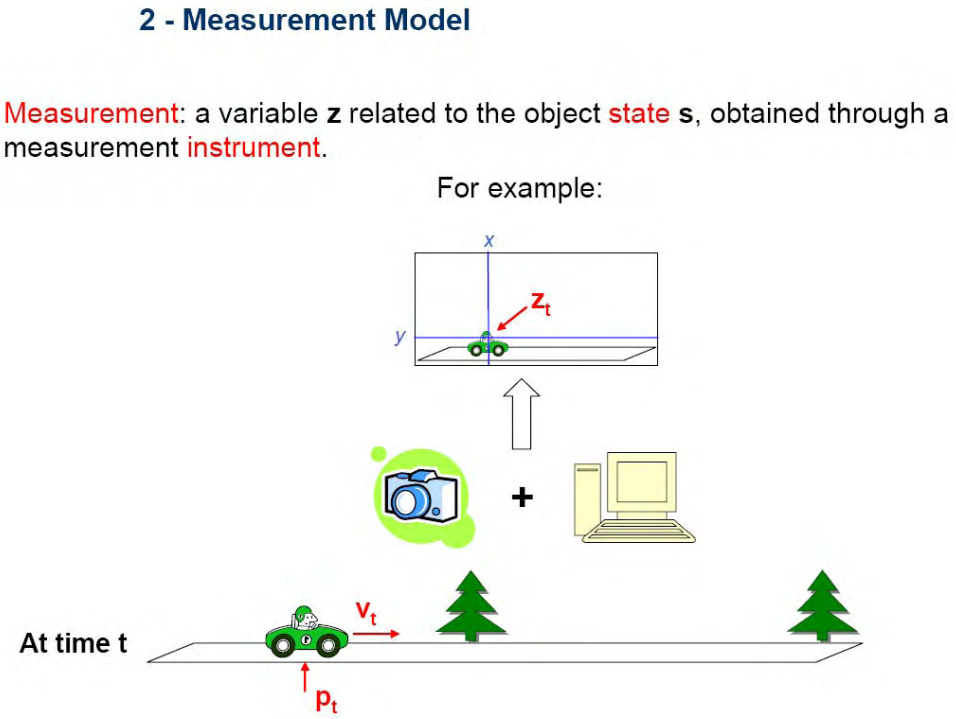
$$P(\mathbf{s}_t|\mathbf{s}_{t-1}) = \text{Gauss}\,(A\,\mathbf{s}_{t-1}+C,\ B\Lambda_w B^T)$$

The models considered up to now are examples of free, unbounded trajectories, with no fixed point (attractor).
We can mention also motion models where the trajectory is expected to oscillate, or float, around a given point in space. These models are useful when we expect the object pose to be limited to a range, around a given base position p0; for example, in 3D face tracking, the head of a person is obviously rotating to a limited extent, around the frontal position (it cannot turn 360° around!).

These cases involve a different model, which is called damped-spring motion…

(TODO)

**Measurement model: the Likelihood function**




As for general tracking systems, a sensor can perform some signal processing before providing the "measurement" to the system. This processing can be more or less sophisticated, ranging from simple filtering, noise reduction, contrast enhancement etc., to extracting salient features (lines, corner points, color blobs etc.) up to a full model-based pose estimation with possibly multiple hypotheses.

A sensor that performs this kind of processing, giving the output as "measurement" variable is often called a smart sensor (or virtual sensor), and consists of both the hardware sensing device + the software processing/interpretation algorithm.

This implies a much more general definition of "measurement", that we can organize for our convenience into low- middle- and high-level measurement typologies.

NOTE: this classification is not perhaps an official one (actually there is still no "official" classification about visual tracking measurement models), but it is very often encountered in practice, and we need it in this Course in order to introduce in a more systematic way our tracking methodologies.

Low-level measurement examples are: either the raw (color or gray) image as it is, or the image after some basic pixel-based processing (noise filtering, edge pixel extraction, contrast enhancement, etc.). Low-level measurements can be also called pixel-level, or image-oriented measurements, and their dimension is the same of the full image = number of pixels.

Middle-level examples include feature-level operations: extract salient regions or geometrical items that can be "seen" into the pixel matrix, for example color blobs, line segments of edges, corner points, textured regions, etc. Generally speaking, every kind of feature has a descriptor, which specify peculiar properties necessary to identify the feature: for example, a uniform color region may be described by the size, shape and distinctive main color, a segment by its length, position and orientation, a characteristic local point by its appearance (a small window of color pixels, or a more abstract description), etc.
From an image, a lot of different features of interest can be identified, from coarse and inform color or motion blobs, to simple edge segments or circular shapes, up to very distinctive local points (e.g. the SIFT algorithm); the choice of course is left only to the imagination of the designer of computer vision algorithms.

In this context, model-free grouping of single features (for example, joining collinear segments into straight lines) can be considered still a middle-level operation.
We can also call middle-level measurements feature-level, or feature-oriented, and the dimension is given by (number of selected features)x(descriptor dimension for each feature), which is of course much less than the full pixel matrix.

High-level measurements go one step further, and use the local image features for a model-based estimation: for example, a pose estimation from 3D point correspondences, as we have seen, involves matching point features from a 3D model to the image, and finding a pose hypothesis s* that best "explains" this set of matchings. This is something that we can also call an "image understanding" procedure.
In the same way, we may call high-level measurements also model-level or object-oriented, and the dimension is very small = dimension of pose parameters p of our model. If the model of the object is an articulated, or deformable shape, of course the dimension of z may increase, but in any case it is of the same order of the state variable s.

The choice between low- middle- or high-level measurements is by a large extent free, and depends on the algorithm we decide to use for tracking. In any case, a general rule is that a low-level measurement needs a sophisticated tracking algorithm, whereas high-level modules are by themselves computationally expensive, but also allow the design of fast and simple tracking methods, since the measurement z can be directly commensured to the state variable s (in this case, z is the position part of s, z=p*). On the other hand, if a measurement variable z is rich with details, and is not just a "synthetic" information z=s*, the tracking system is generally more robust.

Of course, the more complexity load is put onto the sensor system, the easier is the tracking system to design. And, as often happens, the best is in between: middle-level measurement have the advantage of extracting enough synthetic measurements z (in the form of elementary geometric features), but still rich enough to design robust trackers.
But, since not all computer vision algorithms provide this kind of measurement, we must be prepared to deal with all the above mentioned typologies, when designing our tracking system.

Once we have defined the measurement space z for the tracker, we also need to specify a probabilistic measurement model.

A measurement model is a probabilistic model that describes both the deterministic (ideal) relationship between the real state s and the expected measurement z from our sensor, and the overall uncertainty (random component) of the measurement process. This is usually called Likelihood model of the sensory (measurement) system.

For example, if we know the real state of the object s=[p,v], and our measurement instrument is a full pose estimation algorithm, we expect, in absence of noise, calibration errors, etc., to have perfect point correspondences, and to get a perfect estimate p*=p.

Therefore, the expected measurement is zexp = Hs, where H=[I 0], which is a very simple and linear relationship. In the real case, because of noise, bad point matchings, camera parameters etc. we also expect to have some uncertainty, that we can usually model as an additive Gaussian noise z = zexp + e, where e is N(0,Se) with zero mean and Se covariance matrix; the covariance matrix reflects different uncertainties, for example in position and orientation values.

This measurement model: z = Hs+e is equivalent to a probabilistic model: P(z|s), the probability of z for a given state, which in this case is a Gaussian, with mean value Hs and covariance matrix Se.

If also the motion model is linear+Gaussian, this simple example allows to design the tracker without difficulty, by using a standard Kalman Filter.

Unfortunately, for most measurement models the resulting Likelihood is nonlinear, non Gaussian, and often very complex, forcing to use more sophisticated and general means for designing the tracking system. For

this purpose, we will introduce next the Particle Filters approach, which is a very general and powerful Bayesian tracking paradigm.

### Measurement: definitions

The "measurement instrument" can be defined as:

- The camera system alone (e.g. digital camera + frame grabber)

→ The measurement result is the *whole image* **z=I** (a full matrix of color pixels)

OR

- The Camera + Image processing algorithm (e.g. features extraction)

→ The measurement result **z** is a set of *features*, $\mathbf{z}=(\mathbf{q}_1,\mathbf{q}_2,\ldots,\mathbf{q}_N)$

OR

- Camera + Image processing + Pose estimation algorithm (e.g. LSE minimization)

→ The measurement result **z** is directly a measure of the state (only the pose!) $\mathbf{z}\approx\mathbf{p}$

### Measurement Model = Likelihood

Probabilistic model for the measurement process = Likelihood model

Question: which measurement **z** do we expect, *if* the state is **s**?
Solution: for a given hypothesis **s**, we try to simulate the measurement process.

1. If the object has state **s** in space, an ideal measurement instrument would give an output $\mathbf{z}_{exp}$.

   → The expected measurement is a function : $\mathbf{z}_{exp} = g(\mathbf{s})$

2. Since there may be uncertainty or false measurements (the instrument is not perfect), we also expect some measurement noise **v**

   → Therefore $\mathbf{z} = g(\mathbf{s},\mathbf{v})$, and $\mathbf{z}_{exp} = g(\mathbf{s},0)$, where **v** = random variable

This tells the probability $P(\mathbf{z}|\mathbf{s})$ of having a measurement **z**, if the state is s.

$P(\mathbf{z}_t \mid \mathbf{s}_t)$ ← Likelihood of the hypothesis **s** = Probability of **z**, given **s**

Using our classification of measurement models for visual tracking, we can now give a few examples in order to clarify the concept of Likelihood.

If the real object state s is given, generally speaking we can say that the measurement vector z, which can be a more or less complex and large variable, is modeled again with a deterministic and a random (probabilistic) component.

z = g(s,v)

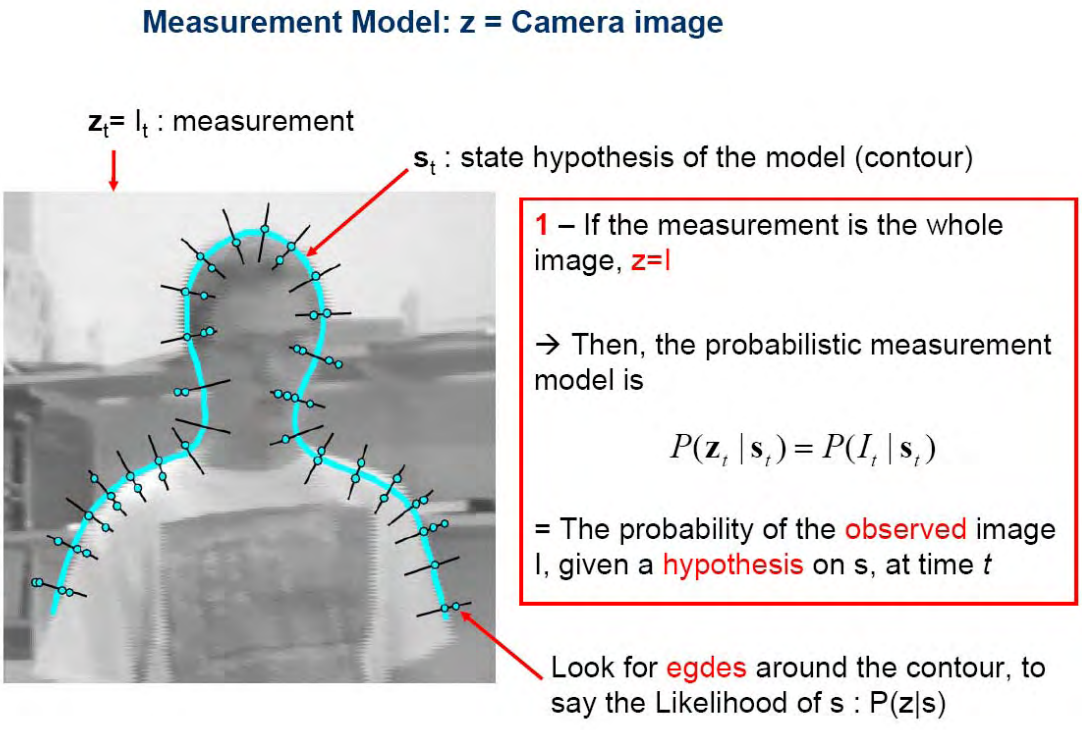
The deterministic component is the expected measurement for the hypothesis s: zexp = g(s,0); this function accounts for the measurement vector that we expect to receive from our "virtual sensor", if the sensor were an ideal, perfectly working sensor with no noise etc. In other words, g(s,0) is the sensor model.

The other term is a random component v, that accounts for unpredicted uncertainty in the measurement process: this source of uncertainty comes from many factors, starting from hardware issues (camera noise, distortion), up to algorithmic imprecisions in the measurement process (for example, bad camera projection model, outliers for a pose estimation algorithms, imprecise and/or false edge detections for features extraction, etc.).

Measurement models, unlike motion models, can be of very different complexity, ranging from simple linear+Gaussian relations like z = Hs+v, up to extremely complex nonlinear models, with multiple hypotheses and "false alarm" cases.

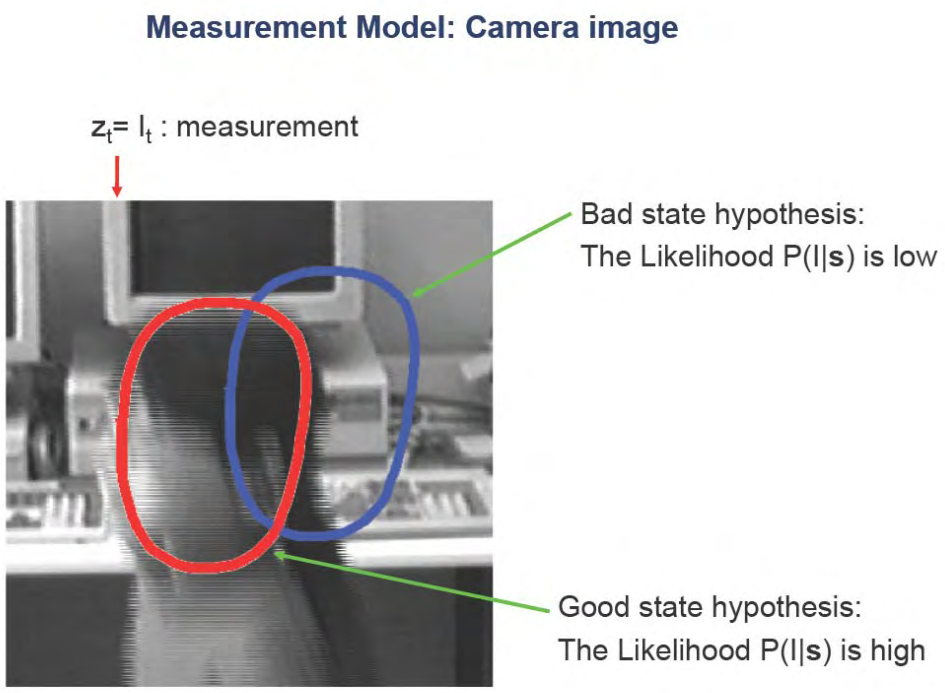
In any case, a general model of the form z=g(s,v) leads to a probability distribution P(z|s), which is called the Likelihood function.

As already mentioned, a Likelihood model will be more complex when the measurement instrument is more "primitive", that is, when z is a more coarse information obtained at an early processing stage, or even just the raw image itself. This choice can be seen as a "computational complexity balance" between measurement module and tracking system.

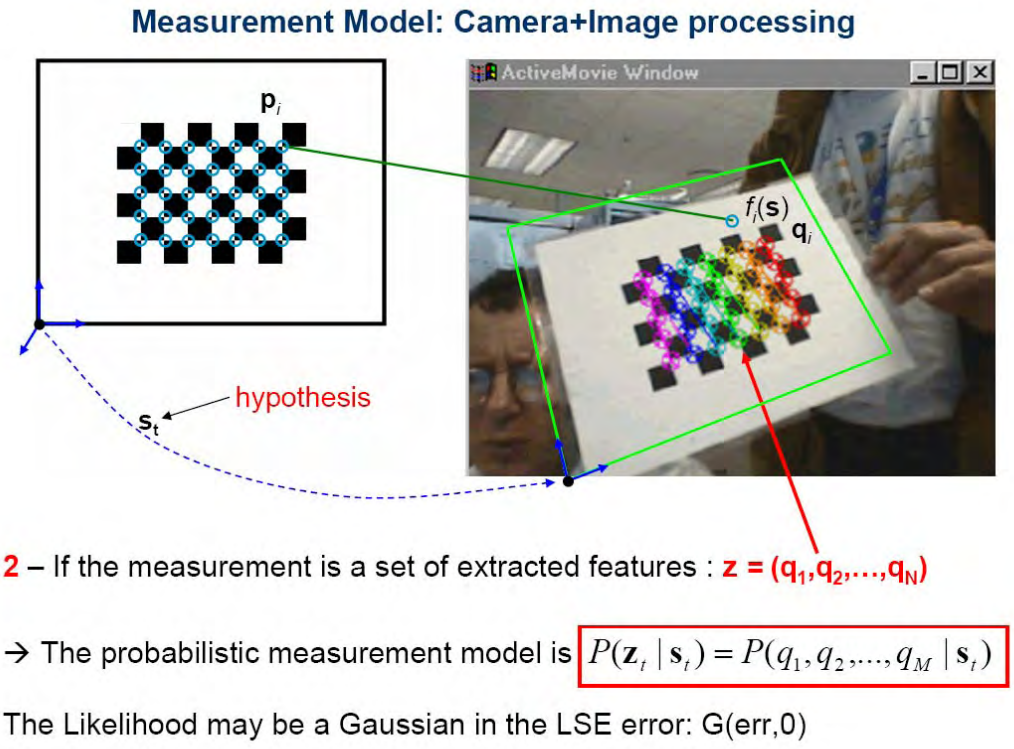


Example of Likelihod function when z is a full pixel map of image edges; in this case, the only processing applied is an edge detector, which gives for every pixel a 0/1 value (edge/no-edge): this is also called edge map.
The example is applied to a countour-based tracking system, where the object (the body) is modeled through the external silhouette, and pose parameters s correspond to a given planar position of the contour + a scale factor.
For a given hypothesis s, the contour model is projected onto the image, and, in the ideal case, one would expect an edge map where edge points in the image are located exactly (and only) over the hypothetic contour line. This edge map would then be our expected measurement zexp: edge pixels under the contour, and no edges elsewhere.
Of course, this is an ideal case, and this simple silhouette model does not acount for evtl. interior edges or external (background) edges; therefore, we also expect spurious edge points inside and outside; finally, since the image contains noise and the shape is not exact, the extracted edge map will give edge points not precisely under the contour, even in the correct pose s, and some points may not be detected at all.
All of these sources of uncertainty require a complex, multiple-hpothesis measurement model z = g(s,v) with a proper random uncertainty variables v and a proper (nonlinear) function g(s,v).
Therefore the Likelihood model in this case is, as we already expected, a quite complex one. Several simplifications are needed in order to make it easier to compute, for example searching edge points only to alimited extent along normal directions, and only for a fixed set of positions along the contour; but still the function remains a nonlinear and non-Gaussian one.

For this kind of measurement, the tracking system is more complex, and requires a proper methodology, for example the particle filters approach (see next Lecture).




We can say, also intuitively, that the red hypothesis has a higher Likelihood with respect to the blue one: contour points of the former are all located over strong edge points, whereas the latter has many points located onto uniform regions, with no edge points.

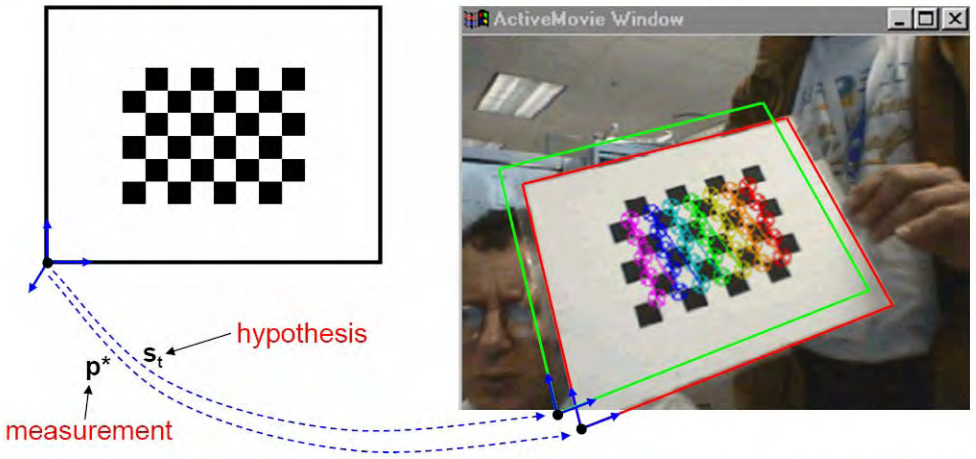



Another example is the chessboard pattern used for camera calibration and pose estimation problems.
We can first consider the set of detected feature points in the image as a middle-level measurement z. In this case, a probabilistic model of our measurement is given by the relationship between state s and expected position of all points: $z_{exp,i} = f(s,p_i)$, where f is the nonlinear 3D/2D projection function.
The random component (noise) v takes into account the imprecision of local points estimation in the image, due to image noise, and unmodeled distortion in the camera projection; therefore, we have zi = fi(s)+vi, where the noise values vi are usually assumed independent, zero-mean and Gaussian random variables.

This gives a relatively simple Likelihood model: $P(z|s) = Gauss(z_{exp}(s),\Lambda_v)$ with $\Lambda_v$ the joint covariance of measurement noise $\Lambda_v = \Lambda_{v1}*\Lambda_{v2}*..*\Lambda_{vn}$.

If also the motion model is (non)linear+Gaussian, tracking in this case can be performed with an Extended Kalman Filter.

**Measurement Model: Camera+Image processing+Pose est.**




**3** – If we run features extraction + LSE minimization (Levenberg-Marquardt)

we get a 3D pose estimate (roto-translation): $\mathbf{z_t = p^*}$ ← this is our measurement!

The Likelihood is $P(\mathbf{z}_t \mid \mathbf{s}_t) = P(\mathbf{p}_t^* \mid \mathbf{s}_t)$ (e.g. A Gaussian around **s**)

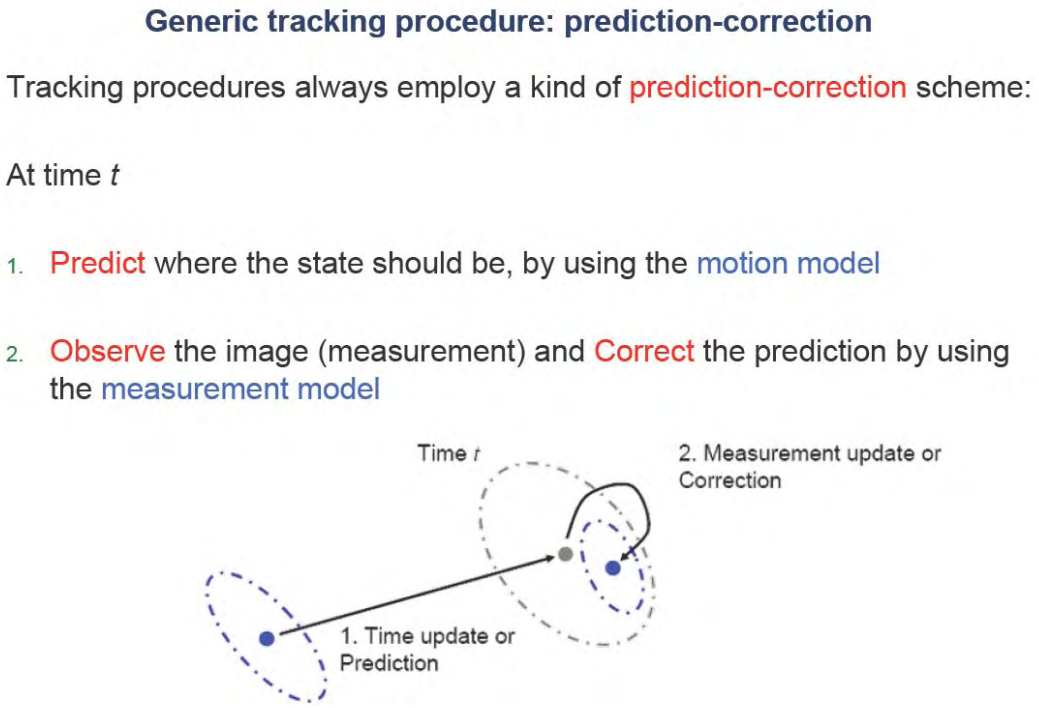
By still considering the previous example, we can also define our measurement process as a high-level one: the output z is the result of a nonlinear LSE pose estimation (see the previous Lecture): an estimate of the roto-translation p* of the object.

Therefore, in this case the measurement and the real state of the object (position only) are directly commensurable: z = p*+v, where v can be a Gaussian noise in the roto-translation parameters, representing the uncertainty about rotation angles and translation parameters of our LSE estimation algorithm.

We can also say z = Hs+v, where H=[I 0] extracts the first 6-dimensional part of s, that is, the position, and we can see that now there is no complex or nonlinear function mapping from s to zexp, which makes the Likelihood model very simple: $P(z|s) = Gauss(Hs,\Lambda_v)$.

If also the motion model is linear+Gaussian, tracking in this case can be performed with a standard Kalman Filter.

**General Bayesian tracking scheme: prediction-correction**




Bayesian tracking, in its more general formulation, is always solved as a two-step procedure each time: a (prediction+correction) scheme.

Prediction has the effect of using the probabilistic motion model in order to assert in which state (where, which velocity, etc.) the target object will be next time, given the current knowledge about the state.
Of course, this knowledge is always of a probabilistic nature; therefore it has some uncertainty which, in the example below, is represented as an average position plus an uncertainty ellipsoid around.
The prediction step always increases this uncertainty by some amount, because it adds the uncertainty of the motion model itself: even if we would know exactly where the object is now (with no uncertainty), because of the motion model we will have some uncertainty over the next, predicted state.

And this motion uncertainty, as we have seen, grows with the length of the time step Dt: intuitively, the more we wait, the more our uncertainty about the target state will grow up.

The result of a prediction step from time t-1 to t is a probability distribution over s, which is called prior probability of the state at time t. Prior means “before doing a measurement”, or “in absence of the information” z at time t.

The next step, correction, modifies the prior by using the information provided by the measurement performed at time t, $z_t$.
With this information, uncertainty about s is reduced, and the estimate is modified; in the Kalman Filter language, the difference between the expected measurement in the prior position, $z_{exp,t} = f(s_{prior},t)$ and the observed real measurement zt, is also called innovation: $n = z_{exp}-z$.

The probability distribution that results from the correction step is called posterior distribution p(st|zt), that is, the probability of the state s at time t, after the measurement zt has been done.

## Probabilistic model

What is the real goal of tracking?

- Sometimes, we are interested in a single estimation result: the "best" state $\mathbf{s}_t$ hypothesis (e.g. the most probable)
- But usually, we need to know also the *uncertainty* (or *belief*) of the hypothesis
- And we need to know about other possible hypotheses as well

…that means, we are more interested in a probabilistic model for the estimation!

→ We look for a whole function of the state $\mathbf{s}_t$ at time *t*: $P(\mathbf{s}_t \mid \mathbf{z}_t, \mathbf{z}_{t-1}, \ldots, \mathbf{z}_0)$

which is called posterior probability, or *belief*, of a state hypothesis $\mathbf{s}_t$, given all the measurements **z**, available up to the present time *t*

The goal of Bayesian tracking is the following: to re-compute the full probability distribution of the state s at present time t, by taking into account all the information available up to time t.

The complete information is given by the set of all measurements up to now performed, $Z_t = (z_1, z_2, \ldots, z_t)$, which is also called "measurement history at time t".

So, the goal of Bayesian tracking is to compute $P(s_t|Z_t)$, that is the posterior probability of s at time t. This function is also called belief about s at time t, and it is a full *pdf* (probability density function) over all possible states.

There are many ways in which we can represent P(s|z), which depend on the nature of motion and measurement models of the system to be tracked.

### *Correction step: Bayes'rule*

**Bayes' rule**

Problem: we are not able to obtain directly the posterior probability, P(s | **z**)

Solution: Bayes' rule tells how to compute it

Bayes: $P(A \mid B) = \dfrac{P(B \mid A)P(A)}{P(B)}$

**Likelihood** **Prior probability**

In our case: $P(\mathbf{s}_t \mid \mathbf{z}_t, ZH_{t-1}) = \dfrac{P(\mathbf{z}_t \mid \mathbf{s}_t, ZH_{t-1})\ P(\mathbf{s}_t \mid ZH_{t-1})}{P(\mathbf{z}_t \mid ZH_{t-1})}$

**Likelihood depend only on s** **(independent on $s_t$)**

Therefore: $P(s_t \mid z_t) \propto P(z_t \mid s_t) \cdot P(s_t \mid z_{t-1},\ldots z_0)$

Bayesian tracking takes the name from the well-known Bayes' rule of probability theory.
NOTE: a much more general class of problems in statistics and prob. theory are formulated as a Bayesian estimation framework; of course, they all involve a Likelihood function and a prior knowledge (pdf), and the Bayes'rule; in the tracking context, prior knowledge comes from a prediction step over time, therefore it is called Bayesian tracking.

In general estimation problems, the prior knowledge can be very weak, therefore not everybody agrees in including this knowledge and using Bayes'rule; but in the case of tracking, the prior pdf takes into account all past measurements, and a reasonable motion model, therefore it is a very important and meaningful term, which motivates the development of Bayesian tracking schemes.

By splitting the measurement history Zt in the present zt and past measurements Zt-1=(zt-1,…,z0), we can write Bayes'rule for P(st|Zt), in terms of the current Likelihood P(zt|st,Zt-1) and prior probability P(st|Zt-1), where the past measurements are always given.

A fundamental assumption is also that the current measurement zt (Likelihood model) is not influenced by the past measurements Zt-1; this can be assumed for most physical instruments, and also for all of our (camera+computer vision algorithm) measurement modules: in other words, what we will measure now (from the current image) does not depend in any way from the past acquisition/processing procedures performed, but is completely determined by current real state of the target object, st, the physical enviromnent around, with noise etc. We can also say, "the observation instrument has no memory", whereas of course the object state does!

Bayes'rule tells us that we can compute the posterior probability P(s|z) if we know P(s) and P(z|s); the measurement z is constant, and given from our instrument (as we have seen), while s is the variable we are interested in. Therefore, the other term P(z) can be neglected, or computed afterwards (it is just a normalization term, to make sure that the integral of P(s|z) be 1).

$$P(s_t \mid z_t) \propto P(z_t \mid s_t) \cdot P(s_t \mid z_{t-1},\ldots z_0)$$

This rule correspond to the "correction" step depicted in the previous Slides. In order to apply this rule, we need then the Likelihood model and the prior state probability, at time t.

### Prior probability

How do we compute the prior probability of $\mathbf{s}_t$?

$$P(s_t \mid z_{t-1},\ldots,z_0) = \int_{s_{t-1}} P(s_t \mid s_{t-1}) \cdot P(s_{t-1} \mid z_{t-1},\ldots,z_0)$$

Prior probability — Motion Model — Posterior in (t-1)

The prior probability can be computed by using

- The posterior P at previous time step (t-1)
- The motion model, from time (t-1) to time t

***Prediction step***

The next question then is how to compute the prior probability of state st, given the past knowledge Zt-1?

The solution to this problem can be formulated in terms of the previous posterior estimation, P(st-1|Zt-1), through the motion model.

$$P(s_t \mid z_{t-1},\ldots,z_0) = \int_{s_{t-1}} P(s_t \mid s_{t-1}) \cdot P(s_{t-1} \mid z_{t-1},\ldots,z_0)$$

This equation corresponds to the prediction step.

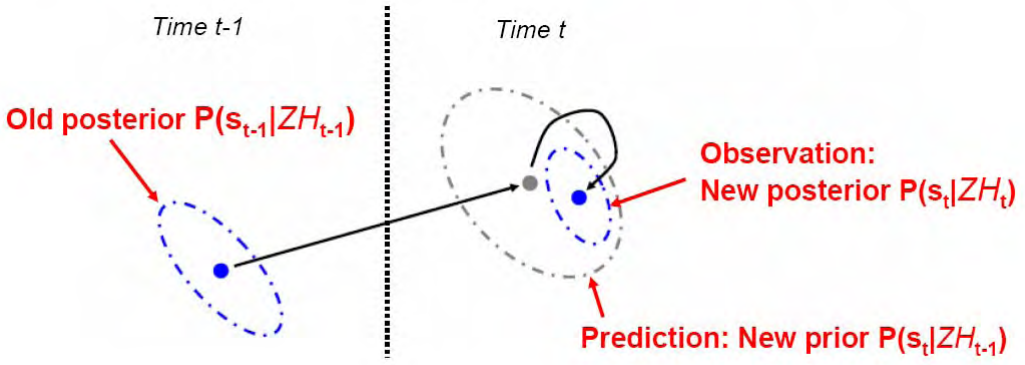
***Bayesian tracking equation***

## Meaning of Bayesian Tracking

We have two steps in Bayesian tracking:

1. Prior probability of $\mathbf{s}_t$ = the probability of $\mathbf{s}_t$ before doing the measurement $\mathbf{z}_t$
→ It is a prediction : (previous posterior + motion model)

2. Posterior of $\mathbf{s}_t$ = the probability of $\mathbf{s}_t$ after doing the measurement $\mathbf{z}_t$
→ It comes from Bayes' rule : (current prior + measurement model)




## Bayesian Tracking Equation

## RESUME

1. Compute the prior probability (before $\mathbf{z}_t$):

$$P(s_t \mid z_{t-1},\ldots,z_0) = \int \underbrace{P(s_t \mid s_{t-1})}_{\text{Motion Model}} \cdot \underbrace{P(s_{t-1} \mid z_{t-1},\ldots,z_0)}_{\text{Posterior in (t-1)}}$$

2. Use the Bayes' rule (after $\mathbf{z}_t$):

$$P(s_t \mid z_t) \propto \underbrace{P(z_t \mid s_t)}_{\text{likelihood of z}} \cdot \underbrace{P(s_t \mid z_{t-1},\ldots z_0)}_{\text{prior probability}}$$

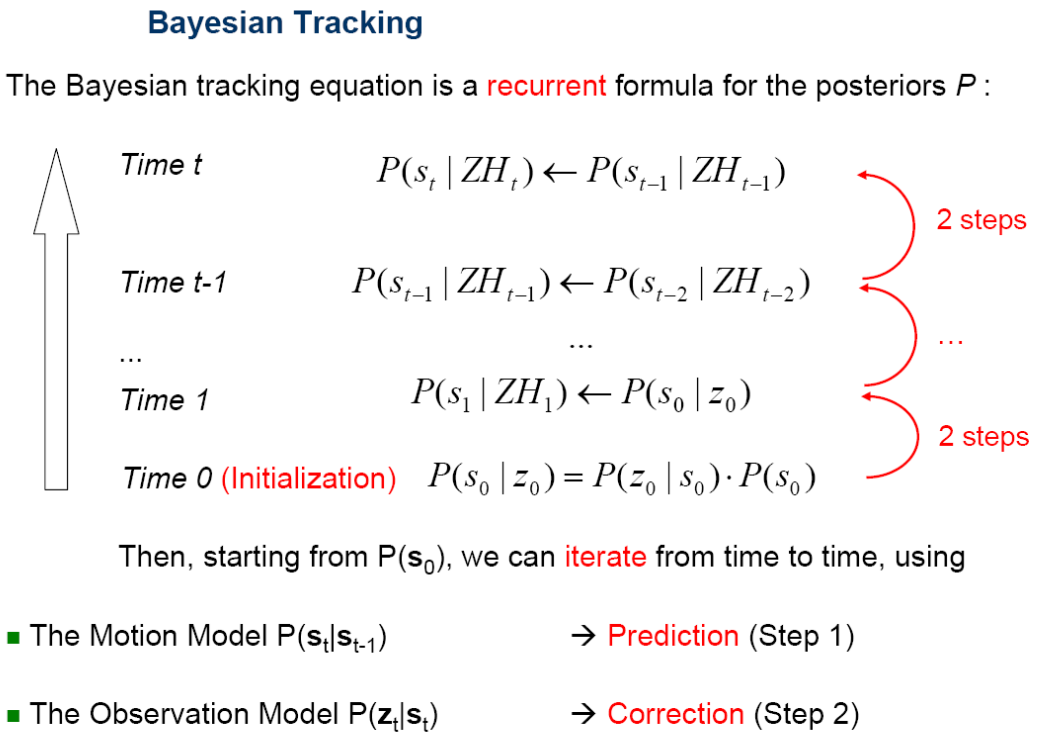
Put everything together: we have the Bayesian Tracking equation

$$\underbrace{P(s_t \mid z_t,\ldots z_0)}_{\text{Posterior in t}} \propto \underbrace{P(z_t \mid s_t)}_{\text{Observation model}} \int_{s_{t-1}} \underbrace{P(s_t \mid s_{t-1})}_{\text{Motion model}} \cdot \underbrace{P(s_{t-1} \mid z_{t-1},\ldots z_0)}_{\text{Posterior in (t-1)}}$$

The two equations of Bayesian tracking at time t can also be compactly expressed by the Bayesian tracking fundamental equation

$$\underbrace{P(s_t \mid z_t,\ldots z_0)}_{\text{Posterior in t}} \propto \underbrace{P(z_t \mid s_t)}_{\text{Observation model}} \int_{s_{t-1}} \underbrace{P(s_t \mid s_{t-1})}_{\text{Motion model}} \cdot \underbrace{P(s_{t-1} \mid z_{t-1},\ldots z_0)}_{\text{Posterior in (t-1)}}$$

Which resumes the propagation of posterior probability from time t-1 to time t, using the motion and observation probabilistic models.




As we can see, this is a recurrent formula for updating the posterior, from time to time.
Therefore, it needs to be initialized at time 0.

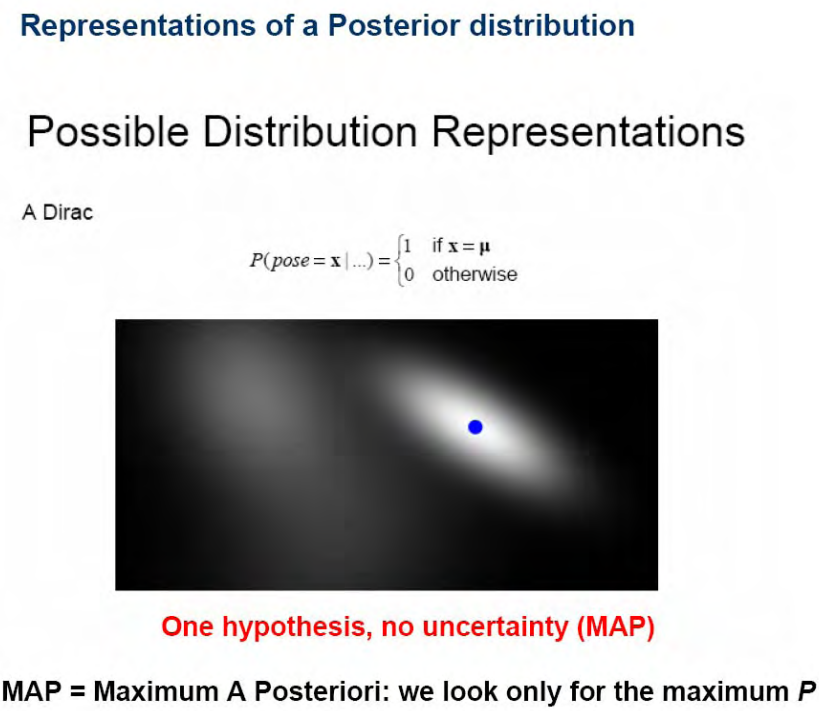
The initialization needs the knowledge of an absolute prior P(s0), which is the knowledge that we have on the state at the very beginning of the sequence, in absence of any measurement.

This knowledge must be given in advance for the tracking problem, like motion and measurement models. If no initial knowledge is available, then one can use a very large and arbitrary distribution (high uncertainty); there are two choices that are commonly used for this purpose:

- if the tracker needs Gaussian probabilities (e.g. a Kalman filter) then one must use a Gaussian also for P(s0); therefore, one may choose P(s0) with an initial (arbitrary) mean and a very large covariance matrix.

- if we are completely free to choose P(s0), then the best choice is a uniform distribution, in a suitable (large) range of values for s0. This corresponds to the “principle of sufficient reason” of Laplace: in absence of any knowledge about a random event (s0), the only reasonable choice is to assign to every value equal probabilities. Of course, we need at least to fix some boundaries for s0 (a “multi-dimensional box”).

**Possible models for the posterior pdf**




The Bayesian tracking equation is a very general, but also very abstract formula.
In order to develop concrete tracking methodologies, we need now to consider the different, particular cases for representing the posterior density (belief) and the other distributions involved.

The first, most simple example is the so-called MAP approach: MAP = Maximum-a-Posteriori.
In this approach, we do not track the full pdf P(st|zt), but rather its maximum value over all possible states.
This is a reasonable approach if the only thing we are interested in is an estimate of position/velocity of the target, but not the uncertainty that we have over these parameters.
NOTE: From a mathematical point of view, this corresponds to representing the posterior pdf with just a dirac (impulse) probability function, centered onto the MAP state.

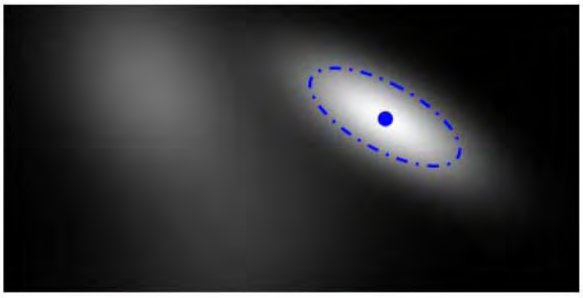
Of course, with such a representation, we loose a lot of informations, that usually are of great interest: if we know the uncertainty associated with the estimate, first of all we can say whether the estimate itself is meaningful for our purposes, or if the tracking is going to be lost (too high uncertainty). And moreover, there may be other, local maxima (peaks) in the pdf that correspond to other hypotheses about the location of the target.
In this example, we can say that at current estimation step there are two main peaks of P(s|z), each one with its uncertainty (area).

## Representations of Posterior distribution

### Possible Distribution Representations

A Gaussian ($\boldsymbol{\mu}$, $\boldsymbol{\Sigma}$)

$$P(pose = \mathbf{x} \mid ...) = N(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$



**One hypothesis + uncertainty**

A better representation involves using one multi-variate Gaussian probability distribution.
In this case, the mean value of the Gaussian is used to represent the “best” estimate s, and the uncertainty about s is given by the covariance matrix, which gives an elliptical (in more dimensions, actually a hyper-ellipsoidal) region of confidence for the target.

This representation is used by the Kalman filter, where propagating the belief in time correspond to propagating the mean and covariance matrix.

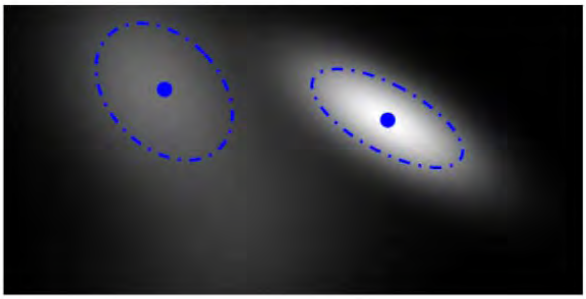
Of course, we still keep a single hypothesis about the state, neglecting the other regions of P where there may be also some other hypotheses (of course less likely, but still possible). And sometimes, even for a single hypothesis an ellipsoidal area may not be a good approximation of the “true” pdf.

## Representations of Posterior distribution

### Possible Distribution Representations

A Mixture of Gaussians $((\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1), (\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2), \ldots)$

$$P(pose = \mathbf{x} \mid \ldots) \propto \sum_i \frac{1}{\sqrt{(2\pi)^n |\boldsymbol{\Sigma}_i|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_i)^T \Sigma_i^{-1} (\mathbf{x}-\boldsymbol{\mu}_i)\right)$$



**Small, fixed number of hypotheses + uncertainties**

In order to keep into account multiple hypotheses, a first idea is to use more Gaussians: a mixture of Gaussians is just a weighted sum of a fixed number of Gaussians, each one with its mean and covariance matrix.

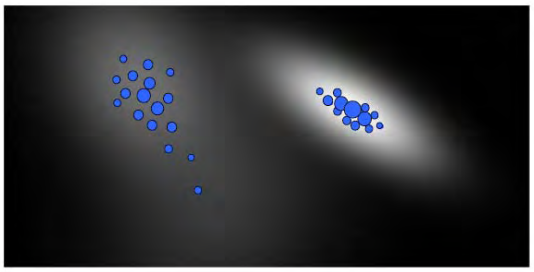
This representation has some weakness, in that: we need to know in advance the number of hypotheses, and they should not change in time. Instead, very often happens that some target hypotheses almost disappear, or temporary “fuse together” into one, while new ones may appear as well.
From this point of view, very much depends on our measurement model, that is, on the Likelihood function of our visual observation modality.

Moreover, designing a tracker with this distribution is much more involved than a Kalman filter, since also the weights w of the mixture must be updated in time; therefore, using just a set of M Kalman filters is generally not possible (or not recommended).

## Representations of Posterior distribution

### Possible Distribution Representations

A set of particles



**Any number of hypotheses + uncertainties**

A much more flexible and general representation is given instead by Monte-Carlo approximations, which in the Bayesian tracking context are also called Particle Filters.

Particle Filters represent the belief P(S|z) through a set of many discrete hypotheses, called “Particles”. Each particle is a pair (s,w) of an inividual state hypothesis + a weight, and the whole set “represents” the continuous pdf in a discrete way, but consistent with probability theory (see the Factored Sampling Theorem of next Lecture).

With this distribution, the full shape of P(s|z) into the non-zero regions is reproduced faithfully, provided we use enough particles. This number grows with two factors: the dimension of the state-space (here is 2), and the precision that we need for computing the values of interest from this distribution (for example, mean value, uncertainties, etc.). This correspond more or less to the “law of large numbers” of statistics.
And, as always happens, unfortunately the number of particles needed (keeping the same precision) grows exponentially with the space dimension (curse of dimensionality).

## Conclusions and resume

- Tracking = Estimating the object state at each time, using
  - Previous state estimate
  - Current measurement
  - Measurement history

- State of the object = a variable related to the physical system, that can be used to make predictions from time to time

- Motion model = a probabilistic model of the object motion from time to time

- Measurement model (Likelihood) = a probabilistic model of the measurement instrument, i.e. the probability of getting a measurement, for a given state hypothesis

- Prior distribution = the probability distribution of the state, given only the previous measurements history

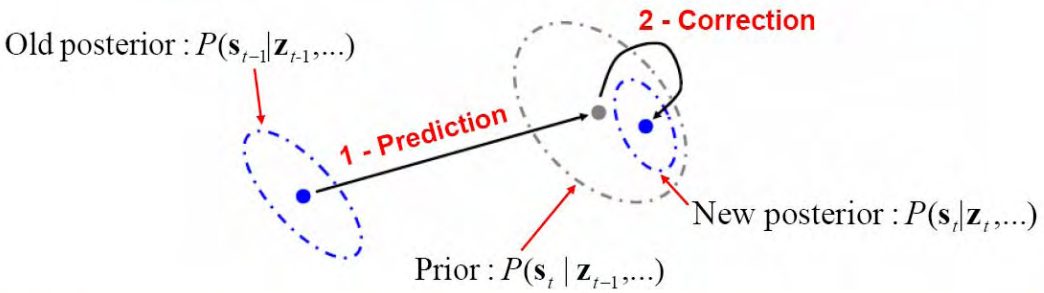
- Posterior distribution = the probability distribution of the state, given the measurements history + the current measurement

- Bayesian Tracking = estimation of the posterior probability distribution, by estimating prior distribution and then applying Bayes' rule (prediction+correction scheme)

**Possible implementations of the tracking scheme**

**Bayesian Tracking: Resume**

**Bayesian Tracking: Prediction-Correction scheme**




1. Prediction → Prior $P_t$ (*before* $\mathbf{z}_t$) : $P(s_t \mid z_{t-1},...,z_0) = \int_{s_{t-1}} \underbrace{P(s_t \mid s_{t-1})}_{\text{Motion Model}} \cdot \underbrace{P(s_{t-1} \mid z_{t-1},...,z_0)}_{\text{Posterior in (t-1)}}$

2. Correction → Posterior $P_t$ (*after* $\mathbf{z}_t$) : $P(s_t \mid z_t) \propto \underbrace{P(z_t \mid s_t)}_{\text{Likelihood}} \cdot \underbrace{P(s_t \mid z_{t-1},...z_0)}_{\text{Prior in t}}$

**Bayesian Tracking: Resume**

**Bayesian Tracking Equation**

$$\underbrace{P(s_t \mid z_t,...z_0)}_{\text{Posterior in t}} \propto \underbrace{P(z_t \mid s_t)}_{\text{Observation model}} \int_{s_{t-1}} \underbrace{P(s_t \mid s_{t-1})}_{\text{Motion model}} \cdot \underbrace{P(s_{t-1} \mid z_{t-1},...z_0)}_{\text{Posterior in (t-1)}}$$

- Motion model: (deterministic function *f* + random variable **w**)

$$\mathbf{s}_t = f(\mathbf{s}_{t-1}, \mathbf{w}_t) \Rightarrow P(\mathbf{s}_t \mid \mathbf{s}_{t-1})$$

**Motion Noise**

- Observation model: (deterministic function *h* + random variable **v**)

$$\mathbf{z}_t = h(\mathbf{s}_t, \mathbf{v}_t) \Rightarrow P(\mathbf{z}_t \mid \mathbf{s}_t) \leftarrow \textbf{Likelihood}$$

**Measurement Noise**

In the Bayesian tracking scheme, both motion and measurement models consist of a deterministic function of the state s, and a random component.

The deterministic component of the motion model is used to do the prediction step: expected next state sexp,t+1, given the current hypothesis st. For the measurement model, it gives the expected measurement zexp,t, given the state hypothesis st.
The random components are also called, respectively, motion and measurement noise.
Both models are equivalent to probability distributions: P(st+1|st) and P(zt|st); the second, in particular, is also called the Likelihood model.

### How do we implement Bayesian tracking?

## Main situations

**1.** The motion and observation models are both (Linear + Gaussian)

→ The Bayesian estimator is the Kalman Filter (Kalman, 1960)

**2.** The motion and observation models are both (Nonlinear + Gaussian)

→ We can use the Extended Kalman Filter

**3.** The motion or observation models are non-linear and non-Gaussian

→ We need Particle Filters (Isard and Blake, 1998)

Depending on the assumptions on the form of probability distributions involved, we distinguish between three main tracking methodologies:

- If both motion and measurement models are (linear+Gaussian), that is f() and h() are linear in s, and w,v are Gaussian white noises, then the Bayesian tracking scheme reduces to the Kalman Filter (R.E. Kalman, 1960) In this case, we can say that “everything is Gaussian”: the probability distributions (motion and Likelihood), the initial prior P(s0) and, as a consequence, all prior and posterior distributions P(st) and P(st|zt).

- If f() and/or h() are nonlinear functions, and the noise vectors are still Gaussian, we can use the Extended Kalman Filter, which is an approximation that uses Jacobian matrices (linearization) of f() and h(). We cannot say anymore that everything is Gaussian, but this approximation provides good enough in most cases; in any case, of course the result will not be optimal (i.e. Bayesian) anymore.

- If f() and/or h() are nonlinear, or one of the noise vectors is not Gaussian, we have to resort to a more general tracking methodology: Particle Filters. This is necessary when, for example, the measurement model provides multiple hypotheses, of which at most one can be generated by the real target, and the others are “false alarms” arising from background clutter, etc.

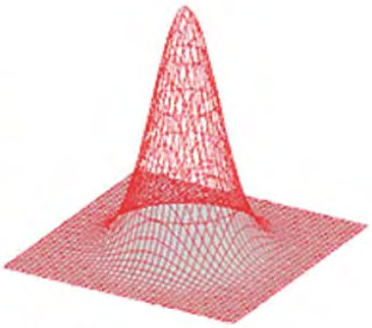
## Linear+Gaussian case: the Kalman Filter

### *Multi-variate Gaussian distribution*

**Multivariate Gaussian distribution**

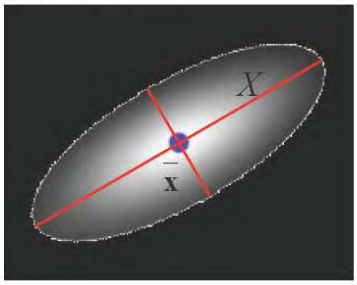
**Multi-variate Gaussian Distribution:**

$$Gauss(\bar{\mathbf{x}}, X) = \frac{1}{(2\pi)^{N/2}\sqrt{|X|}} \exp\left(-\frac{1}{2}\left(\mathbf{x}-\bar{\mathbf{x}}\right)^T X^{-1}\left(\mathbf{x}-\bar{\mathbf{x}}\right)\right)$$



We can represent the probabilities P with only two quantities:

- The Mean vector $\bar{\mathbf{x}}$
- The (NxN) Covariance Matrix ***X***

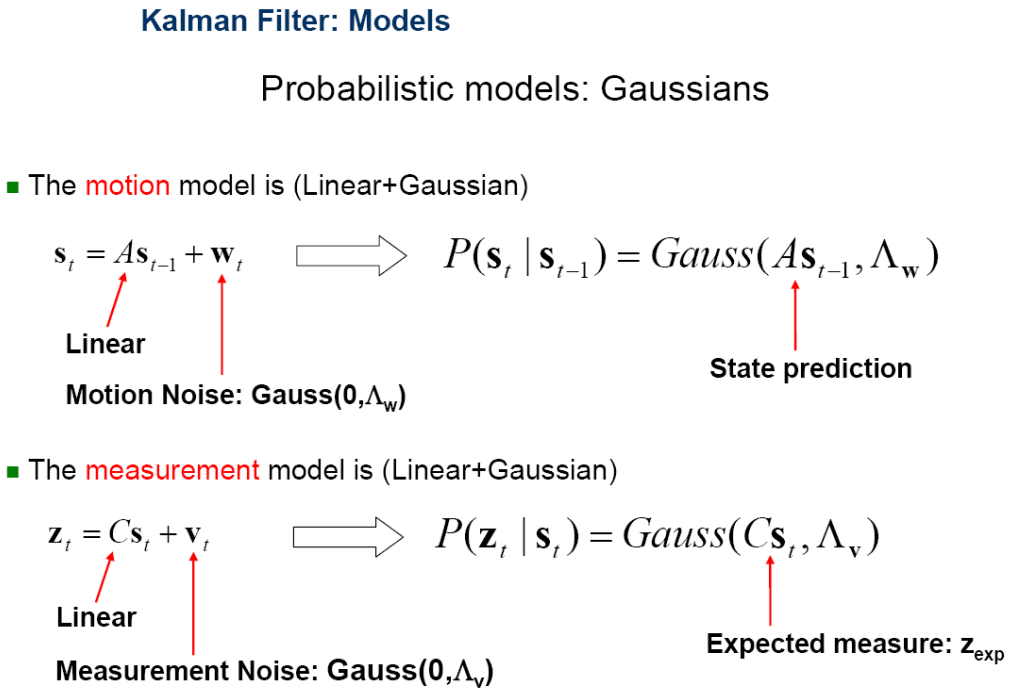



A multi-variate Gaussian distribution (in N-dimensions)

$$Gauss(\bar{\mathbf{x}}, X) = \frac{1}{(2\pi)^{N/2}\sqrt{|X|}} \exp\left(-\frac{1}{2}\left(\mathbf{x}-\bar{\mathbf{x}}\right)^T X^{-1}\left(\mathbf{x}-\bar{\mathbf{x}}\right)\right)$$

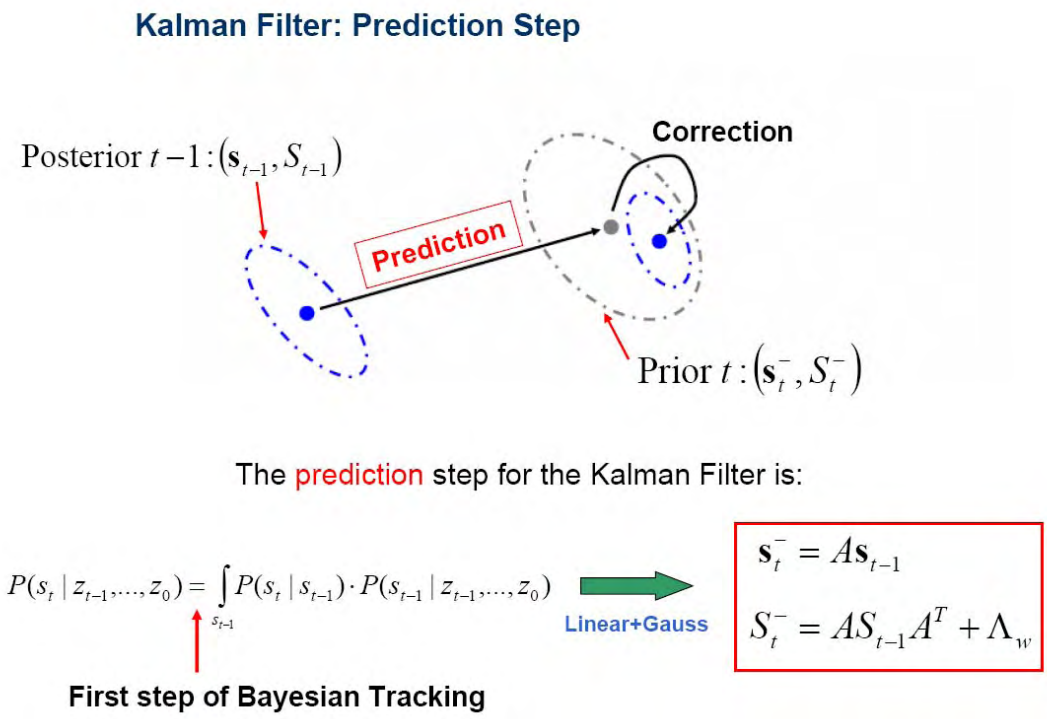
Can be completely represented by only two quantities: the N-mean vector x_bar and the (NxN) covariance matrix X.
The covariance matrix is symmetric and positive definite; its N orthogonal eigenvectors form the main axes of an ellipsoid in N-space, and the length of these axes are proportional to the respective (positive) eigenvalues. The center of the ellipsoid is x_bar.
This is the uncertainty ellipsoid, and we can say that most of the Gaussian distribution is contained inside this volume. A smaller ellipsoid indicates a higher confidence about x (low uncertainty), where the uncertainty can be of course different along different directions in N-space.

***Motion and measurement models***




If motion and measurement equations are (linear+Gaussian), all probabilistic models are Gaussian; if also the initial prior P(s0) is a Gaussian, then everything is Gaussian, and the corresponding Bayesian tracking equations (prediction+correction steps) reduce to the Kalman Filter equations, which deal exclusively with mean and covariance matrices of the probability functions.

***Predition-correction equations***




Prediction in Kalman Filter is obtained through the following equations for mean and covariance matrices of the state

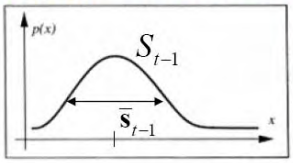
$$\mathbf{s}_t^- = A\mathbf{s}_{t-1}$$
$$S_t^- = AS_{t-1}A^T + \Lambda_w$$

where the (-) sign denotes prediction (prior) quantities, obtained in absence of the current measurement z.

## Kalman Filter: Prediction Step

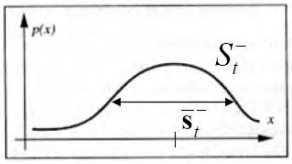
Prediction Equation: Compute the prior in t

$P(s_{t-1}|z_{t-1})$ = Old posterior




Motion model: $P(s_t|s_{t-1})$

$P(s_t)$ = Prior




**Diffusion:**
**The uncertainty S increases**

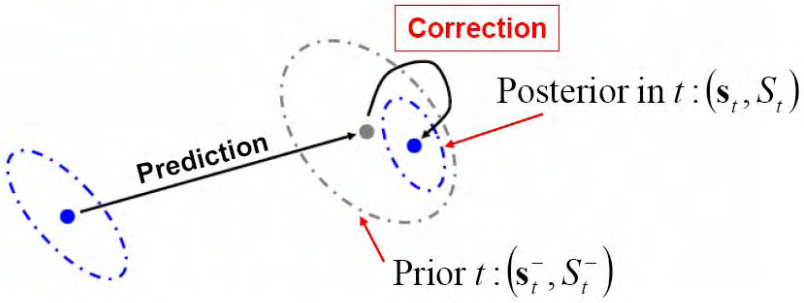
$$\mathbf{s}_t^- = A\mathbf{s}_{t-1}$$

$$S_t^- = AS_{t-1}A^T + \Lambda_w$$

← Motion uncertainty = $\Lambda_w$

From this equation, the diffusion effect of prediction (increasing uncertainty about the state estimate) can be readily seen, where the added uncertainty comes in part from the added motion noise covariance Lw.

## Kalman Filter: Correction Step




The correction step for the Kalman Filter is:

$$P(s_t \mid z_t) \propto P(z_t \mid s_t) \cdot P(s_t \mid z_{t-1}, \ldots z_0)$$

Second step of Bayesian Tracking

Linear+Gauss

$$s_t = s_t^- + G_t(z_t - Cs_t^-)$$

$$S_t = S_t^- - G_t C S_t^-$$

and

$$G_t = S_t^- C^T (CS_t^- C^T + \Lambda_v)^{-1}$$

Correction step in Kalman Filter is obtained through the following equation, where s and S are the posterior mean and covariance matrices of P(s|z):

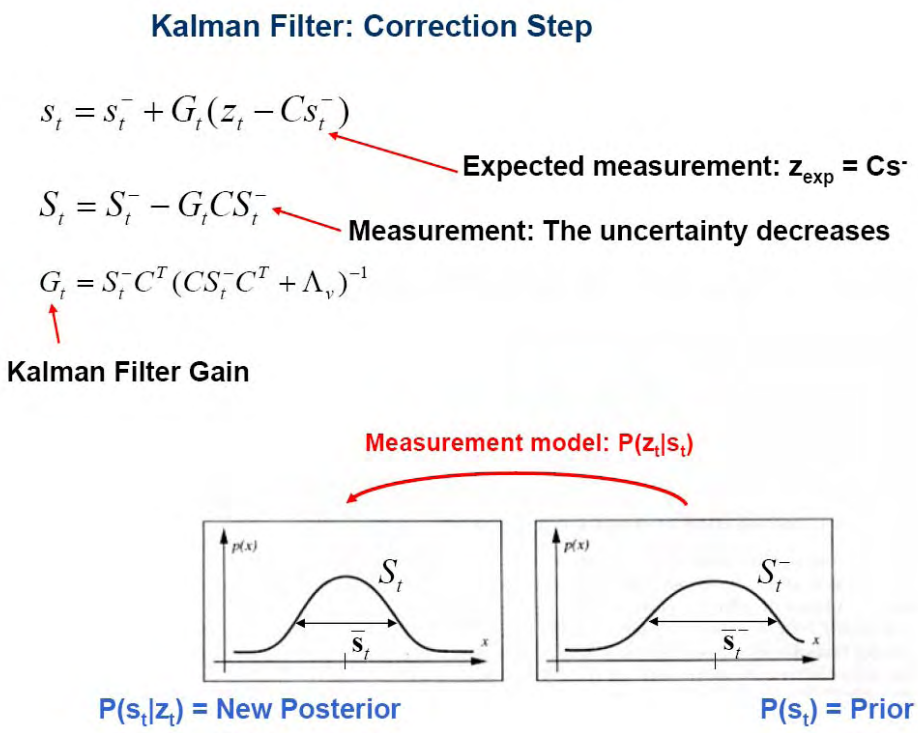
$$s_t = s_t^- + G_t(z_t - Cs_t^-)$$

$$S_t = S_t^- - G_t C S_t^-$$

where

$$G_t = S_t^- C^T (C S_t^- C^T + \Lambda_v)^{-1}$$

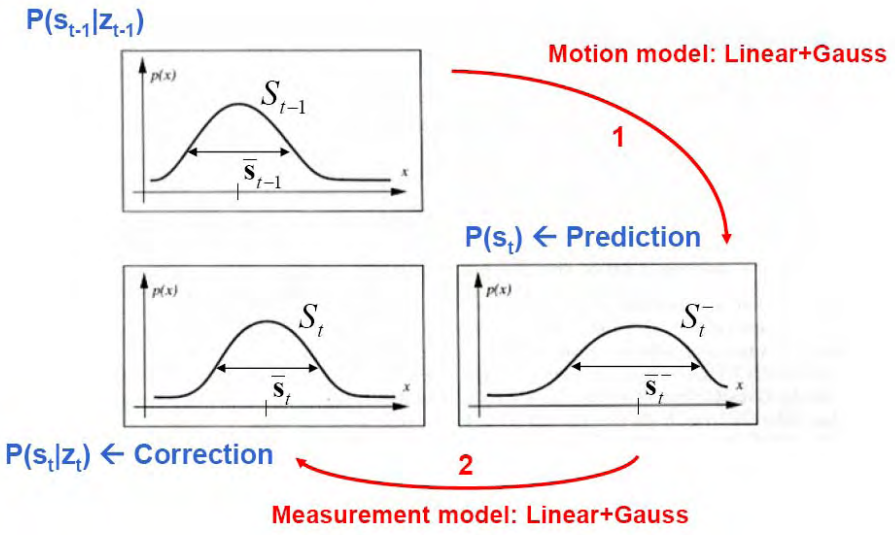
is the Kalman Filter gain.




This step reduces uncertainty about the state, through the measurement z; the correction of the mean value s is obtained through the difference (z-zexp) between expected and actual measurement, and this difference is also called “innovation” in the tracking literature.

## Resume

### Kalman Filter




### Kalman Filter: the full algorithm

- Initialization : at time 0, given the initial prior density $\mathrm{P}(s_0) = \mathrm{Gauss}(s_0^-, S_0^-)$ get the first measure $z_0$, and compute the first posterior

$$s_0 = s_0^- + G_0(z_0 - Cs_0^-)$$
$$S_0 = S_0^- - G_0 C S_0^-$$
$$G_0 = S_0^- C^T (C S_0^- C^T + \Lambda_v)^{-1}$$

- Tracking : at time t, given the posterior at time t-1, do
  - Prediction: $s_t^- = A s_{t-1} \qquad S_t^- = A S_{t-1} A^T + \Lambda_w$
  - Correction: $s_t = s_t^- + G_t(z_t - Cs_t^-)$
    $S_t = S_t^- - G_t C S_t^-$
    $G_t = S_t^- C^T (C S_t^- C^T + \Lambda_v)^{-1}$

A complete tracking scheme requires also an initial step. In Kalman filtering, this is obtained through the initial prior P(s0), which must be given in advance, and must be also Gaussian.
Of course, the initial estimation step will consist only in a correction, given z0, since no prediction from previous time is available; therefore, the absolute knowledge P(s0) is used as initial prior over s0.

**Nonlinear+Gaussian case: the Extended Kalman Filter**

### Extenden Kalman Filter

Probabilistic models:

- The motion model is (Nonlinear + Gaussian)

$$\mathbf{s}_t = f(\mathbf{s}_{t-1}) + \mathbf{w}_t \Rightarrow \text{Prediction } (\mathbf{s}_t)^- = f(\mathbf{s}_{t-1},0)$$

**Nonlinear**

**Motion Noise: Gauss(0,$\Lambda_w$)**

- The measurement model is (Nonlinear + Gaussian)

$$\mathbf{z}_t = h(\mathbf{s}_t) + \mathbf{v}_t \Rightarrow \text{Expected measurement } \mathbf{z}_{exp} = h(\mathbf{s}_t,0)$$

**Nonlinear**

**Measurement Noise: Gauss(0,$\Lambda_v$)**

### Extended Kalman Filter

Solution: we linearize the functions *f* and *h*
→ Compute the Jacobian matrices

*f* is linearized around $\mathbf{s}_{t-1}$ : $A_t = \left.\frac{\partial f}{\partial s}\right|_{s_{t-1}}$

Prediction :

$$s_t^- = f(s_{t-1},0)$$
$$S_t^- = A_t S_{t-1} A_t^T + \Lambda_w$$

*h* is linearized around $\mathbf{s}_t$ : $C_t = \left.\frac{\partial h}{\partial s}\right|_{s_t}$

Correction :

$$s_t = s_t^- + G_t(z_t - h(s_t^-,0))$$
$$S_t = S_t^- - G_t C_t S_t^-$$
$$G_t = S_t^- C_t^T (C S_t^- C_t^T + \Lambda_v)^{-1}$$

If we relax the assumption about linearity for motion and/or measurement models, we cannot say that everything is Gaussian anymore, even if both noise vectors w and v are still Gaussian and additive.
In this case, we can still approximate the Kalman Filter equation if we linearize the functions f() and h() around the respective state values.

The expected next state st+1 (motion)and expected zt (measurement) are still given by the respective nonlinear functions f(st-1) and h(st-), in order to compute the prior and posterior mean state values. Instead, the covariance propagation steps use a linearized version of f and h. This linearization is obtained through the respective Jacobian matrices:

$$A_t = \left.\frac{\partial f}{\partial s}\right|_{s_{t-1}} \qquad C_t = \left.\frac{\partial h}{\partial s}\right|_{s_t}$$

Which "act" like the A and C matrices of the linear Kalman filter, for computing prior and posterior covariance matrices, and for the Kalman gain.

This tracker is called Extended Kalman Filter (EKF), and reduces to the standard Kalman Filter if f() and h() are linear: the Jacobian matrices become exactly A and C.

### Extended Kalman Filter : limitations

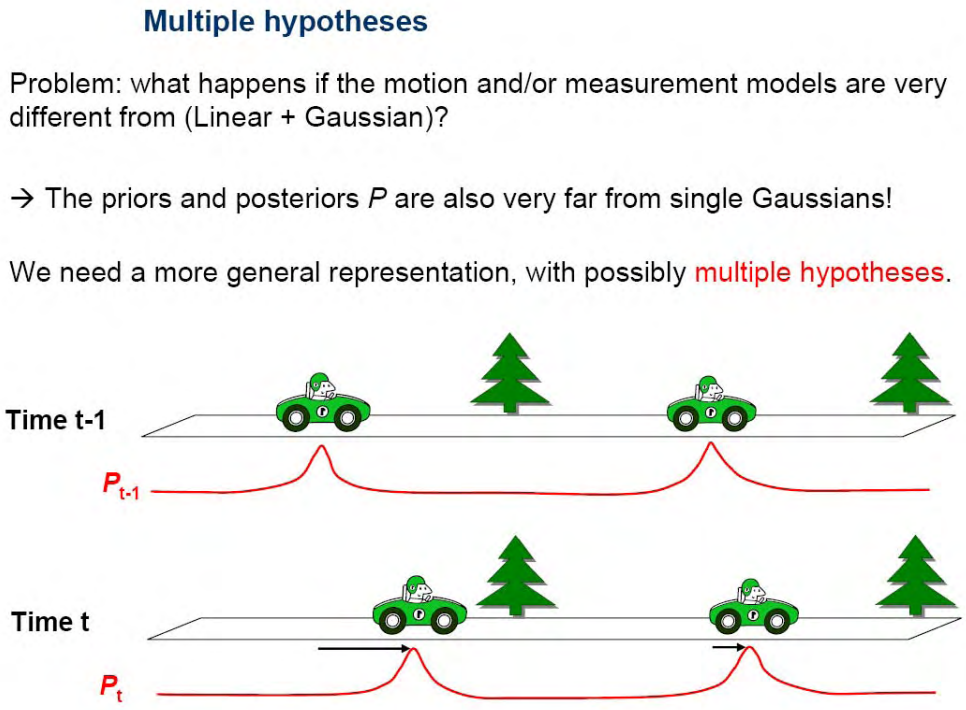
- Kalman Filter was optimal (the exact Bayesian tracker) for linear+Gaussian models
- Problem with EKF: the functions *f* and *h* are linearized (approximated)

  → The solution is not optimal
- And, if the noises are not Gaussian, it does not work

EKF is not the most general Bayesian tracker!

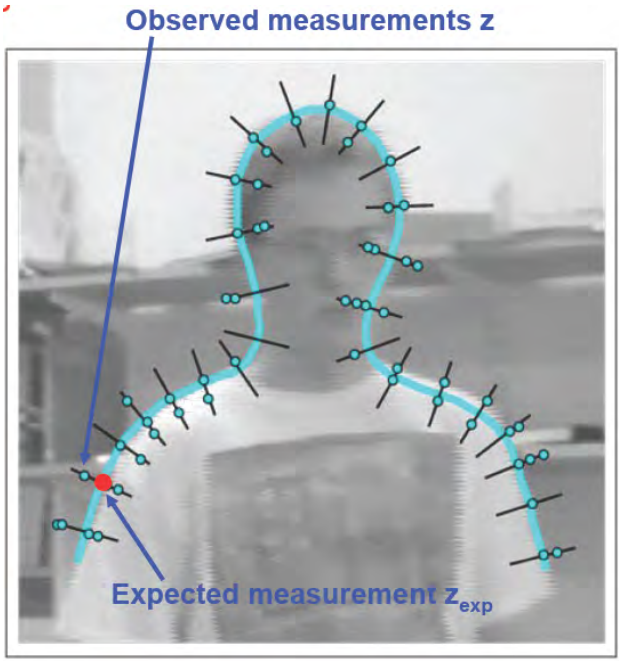
This filter is not the optimal Bayes' tracker, because of the linearization, which appxorimates f() and h(). Therefore, it works well as long as the two approximations are good, and nonlinear effects can be locally neglected (not too wide steps from time to time, and not too much non-linear behavior of f and h around the respective state values).

**Non-gaussian situation: a multiple-hypothesis measurement model**




In many cases, the measurement model provides multiple hypotheses. That means, for a measurement variable z we can infer more, simultaneous possible localizations for the target pose, or for individual features associated with the target.

In the picture, we have an object-level measurement (high level) which is directly a pose estimation of the car, but this time in a situation when another car comes into the visible field of the camera.




Another example (more concrete for our task): in contour tracking we can often have an ambiguity, by observing similar edges in the vicinity of a silhouette hypothesis s: which image edge corresponds to the model edge of the object being searched?

In this situation, we can either select the nearest detected edge, or for more robustness keep all the available hypotheses without discarding anyone. In the second case, we need a Likelihood function P(z|s) which has more peaks (a *multi-modal* distribution) that cannot definitely be modeled with a single Gaussian, so that a Kalman Filter cannot be applied.

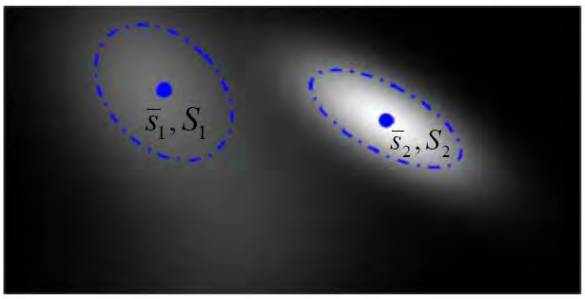
***First approach to the multi-modal case: mixture of Gaussians***

**Mixture of Gaussians**

A first possibility: approximate P with a mixture of Gaussians

$$P(s) = \sum_{k=1}^{K} w_k \frac{1}{(2\pi)^{n/2}\sqrt{|S_k|}} \exp\left(-\frac{1}{2}(s-\bar{s}_k)^T S_k^{-1}(s-\bar{s}_k)\right)$$

It is a weighted sum of K Gaussians (weights = $w_k$)




Problems:

- It is much more complicated than 1 Gaussian (no Kalman Filter anymore!)
- We need to have a known and constant number of hypotheses, K

A Mixture of Gaussians can model a multi-modal distribution, but usually we do not know in advance how many meaningful peaks (modes) are present, and this number can change during time.

Moreover, tracking a mixture of Gaussians needs also the estimation of the weights wk, together with the mean and covariance matrix, and for this problem just a bank of K independent Kalman Filters cannot be applied.

Therefore, we need a more general tracking method, like as the Particle Filters approach.

**Most general case: Monte-Carlo sampling scheme (Particle Filters)**

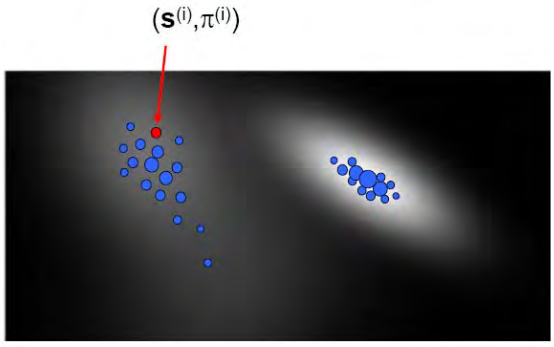
***Factored sampling: discrete implementation of Bayes'rule***

**How can we represent a more general *P* ?**

**PARTICLES REPRESENTATION**

A more general representation for a probability distribution: set of Particles

Particle = a state hypothesis + weight: $(\mathbf{s}^{(i)},\pi^{(i)})$ $i=1,...,n$




**Factored Sampling (Monte-Carlo technique)**

**Approximate P(s|z) with particles → Factored Sampling**

**A set of N particles** $(s^{(i)}, \pi^{(i)})$ **represents the distribution *P*(s|z) with arbitrary (probabilistic) precision for** $N \to \infty$, **if:**

- **$s^{(i)}$ are randomly generated from the prior** $P(s)$
- **$\pi^{(i)}$ are given by the Likelihood** $\pi^{(i)} = P(z \mid s^{(i)}) / \sum_i P(z \mid s^{(i)})$

This is a Monte-Carlo (randomized) approximation of Bayes' rule:

- Bayes: P(s|z) = k P(z|s) P(s), with k not dependent on s
- Factored sampling: P(s|z) is "represented" by a random sample $(s^{(i)}, \pi^{(i)})$ distributed as P(s) and weighted by P(z|s)

## Meaning of the Factored Sampling representation

### The set of particles represents P(s|z)

- P(**s**|**z**) is a continuous distribution (**s** is a real-valued vector)
- $(\mathbf{s}^{(i)}, \pi^{(i)})$ is a discrete set

...and they have all the same statistical properties: for every function g( )

$$\int_s g(s)P(s \mid z) \approx \sum_{i=1}^{n} \pi^{(i)} g(s^{(i)})$$

For example:

- Mean vector $\bar{s} = \int_s P(s \mid z) \approx \sum_{i=1}^{N} \pi^{(i)} s^{(i)}$
- Covariance matrix $S = \int_s (ss^T - \bar{s}^2) P(s \mid z) \approx \sum_{i=1}^{N} \pi^{(i)} (s^{(i)} s^{(i)T} - \bar{s}^2)$

A Particle Set is a set of individual state hypotheses, associated with respective (scalar) weights.
This set represents in a discrete way the continuous posterior distribution. The idea behind this is given by the Factored Sampling Theorem.

This is an equivalent of the Bayes' rule, when using the particles representation.

That is, we can represent with arbitrary precision (in a probabilistic sense) the posterior P(s|z) if we construct a particle set with the following properties:

- the state hypotheses s(i) are sampled at random from the prior distribution P(s)
- the weights of these hypotheses are computed as the Likelihood values w(i) = P(z|s(i))

Which is a Monte-Carlo equivalent of Bayes'rule.
In general statistics, Monte-Carlo techniques are methods for estimating properties of a given distribution P(x) like as mean, covariance matrix etc., by generating random sample points from P(x) or other distributions that are related to P(x) in some way.

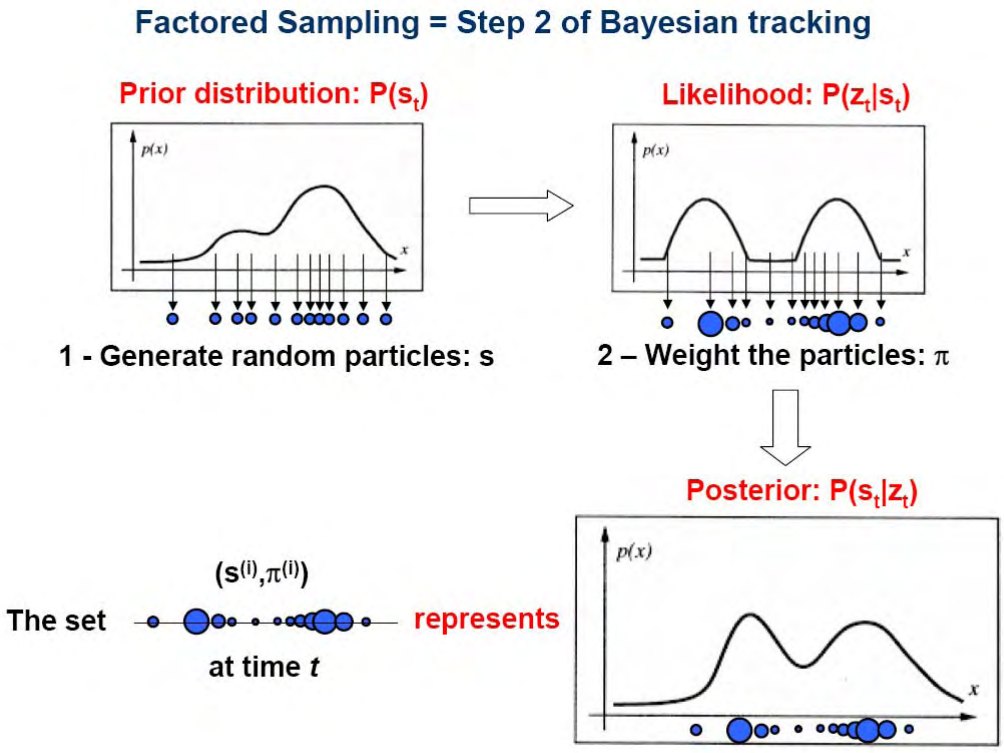
The meaning of this representation is the following one: we can estimate from the set of particles any property of the original P(x) by a discrete average (sum) instead of the integral function:

$$\int_s g(s)P(s \mid z) \approx \sum_{i=1}^{n} \pi^{(i)} g(s^{(i)})$$

Where g(s) is an arbitrary function, that we can define in order to extract the quantities of interest for us.
In particular, in order to estimate the mean and covariance matrix of our belief (posterior) pdf, we have:

$$\bar{s} = \int_s P(s \mid z) \approx \sum_{i=1}^{N} \pi^{(i)} s^{(i)}$$

$$S = \int_s (ss^T - \bar{s}^2) P(s \mid z) \approx \sum_{i=1}^{N} \pi^{(i)} (s^{(i)} s^{(i)T} - \bar{s}^2)$$




Factored sampling corresponds to Step 2 of general Bayesian tracking (correction step).
As we can see from the example picture, now the pdf (prior and posterior) need not anymore to be Gaussians, and the Likelihood model can be a multi-modal distribution, keeping all the multiple measurement hypotheses available from the instrument.

***Prediction step: Monte-Carlo sampling from the prior distribution***

In order to perform Step 1 (prediction step) now we need to represent and compute the prior, $P(s_t|Z_{t-1})$.

## Sample from the prior = Step 1 of Bayesian tracking

## Sample from the prior distribution

Now we need to sample $(\mathbf{s}_t)^{(i)}$ from the prior = generate random positions from

$$P(s_t \mid ZH_{t-1}) = \int_{s_{t-1}} P(s_t \mid s_{t-1}) \cdot P(s_{t-1} \mid ZH_{t-1})$$

But the old posterior was only a set of discrete particles $(\mathbf{s}_{t-1},\pi_{t-1})^{(i)}$ !

→ We approximate the integral, by using the old particles and weights

$$P(s_t \mid ZH_{t-1}) \approx \sum_{j=1}^{N} P(s_t \mid s_{t-1}^{(j)}) \cdot \pi_{t-1}^{(j)}$$

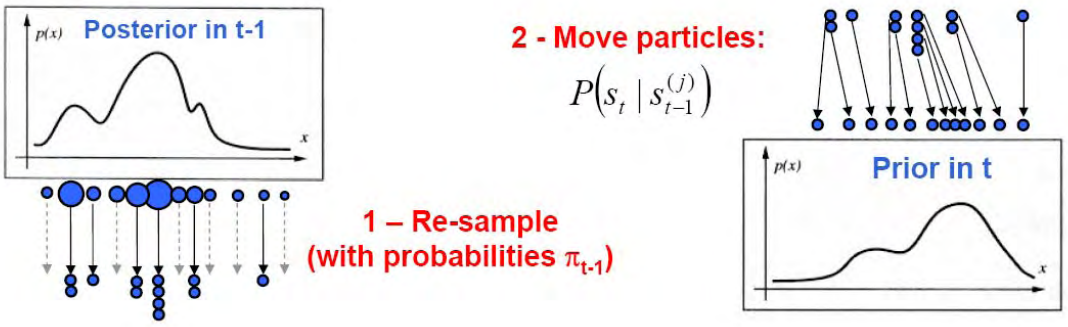
With this approximation, the prior is a weighted sum of motion probabilities, from each one of the old particles positions!

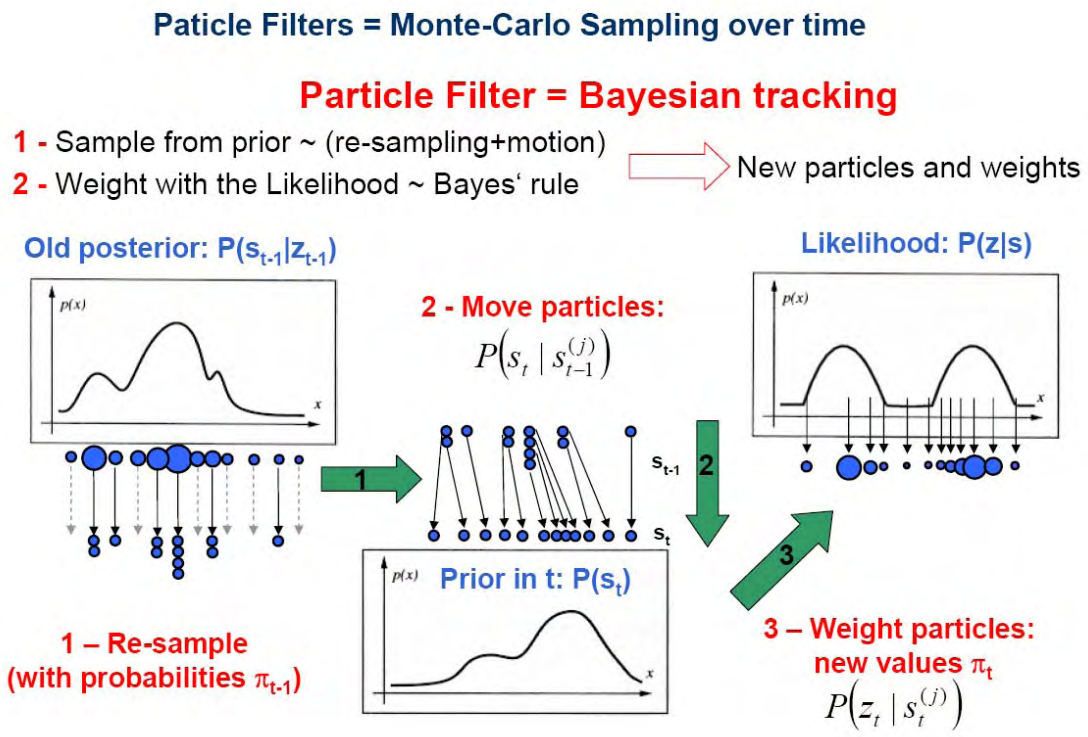
## Sample from the prior = Step 1 of Bayesian tracking

**LOOP - Repeat N times:**

**1.** Re-sample: Choose one from the old paricles set $(\mathbf{s}_{t-1})^{(j)}$, by taking a random number *j* in (1,...,N) with probabilities $(\pi_{t-1}^{(1)},...,\pi_{t-1}^{(N)})$

**2.** Random motion: Move the chosen particle in a new random position, according to the motion model $P(s_t \mid s_{t-1}^{(j)})$






We recall the first formula (prediction) of Bayes'tracking, that involves the previous posterior and the motion model.

$$P(s_t \mid ZH_{t-1}) = \int_{s_{t-1}} P(s_t \mid s_{t-1}) \cdot P(s_{t-1} \mid ZH_{t-1})$$

In our case, the previous P(s|z) is represented by another set of particles. Therefore, we can approximate the integral in a discrete way, if we consider the particles P(st-1|zt-1) as a sum of many Dirac (impulse) functions, centered in the respective positions s(i)t-1, with magnitudes w(i)t-1.

Therefore, the integral becomes a sum over the previous particles

$$P(s_t \mid ZH_{t-1}) \approx \sum_{j=1}^{N} P(s_t \mid s_{t-1}^{(j)}) \cdot \pi_{t-1}^{(j)}$$

that approximates the prior at time t.
This representation gives us a way to generate the new particle set s(i)t in two steps:

1 – Pick at random one of the old particles, selecting an integer j=1,…,N with probabilities w(j)t-1
2 – Generate the new position s(j)t from the motion model P(st|s(j)t-1), centered in the old particle position

The first step is also called re-sampling, and has the effect of picking (with repetition) the old positions with higher weights, while eliminating low weight particles (that are not selected at all).
The second step is called diffusion, because it moves the particles to new positions according to random motion, therefore "spreading" them around and increasing uncertainty.

Re-sampling and diffusion, taken together constitute our prediction of the particle set, which will have a higher uncertainty (like in the Kalman Filter case), that will be reduced by the correction step using the measurement model.

***The complete Particle Filter scheme for tracking***

**CONDENSATION – The full algorithm**

- Initialization: construct a first sample set $(s_0^{(i)}, \pi_0^{(i)})$ by random sampling from the given initial prior density $P(s_0)$ and weight it with the initial likelihood $P(z_0|s_0)$
- Tracking: from the old sample set $(s_{t-1}^{(i)}, \pi_{t-1}^{(i)})$

LOOP: for i=1,…,N

- Re-sample : take at random a particle j by generating a random number in (1,…,N) with probabilities $(\pi_{t-1}^{(1)},...,\pi_{t-1}^{(N)})$
- Random motion (diffusion): move the particle in a random new position, by using the motion model applied to the particle *j* : $P(s_t \mid s_{t-1}^{(j)})$
- Weight (measurement) : with the measurement $\mathbf{z}_t$, give weight to the particle, according to the Likelihood $P(z_t \mid s_t^{(j)})$

In the computer vision literature, Particle Filters are also known as Condensation algorithm (Conditional Density Propagation), where the conditional density is just the posterior P(s|z) density of s, conditioned on z. It has been used in particular for contour-based tracking by Isard and Blake, who invented this name for the filter, in 1998.

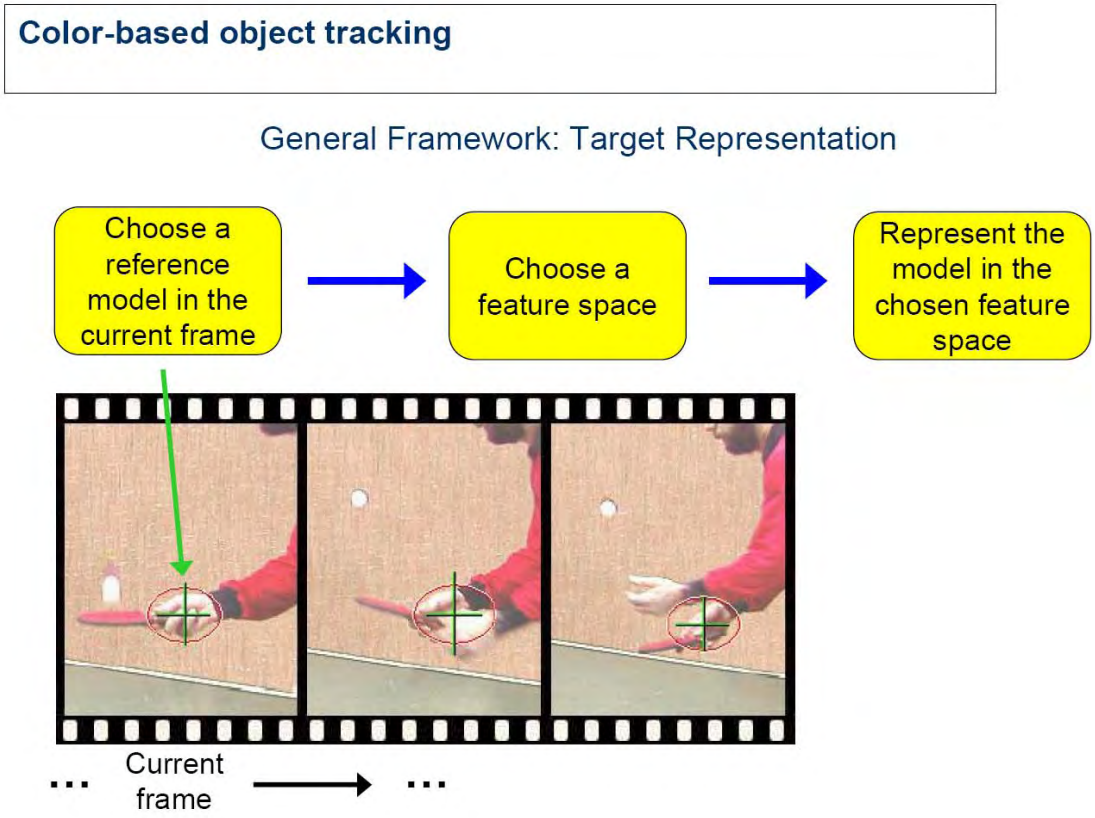
By resuming, also in the case of Particle Filters we need of course the initialization, using an initial prior P(s0) available.

In this case, we are again not restricted to Gaussians; therefore, in case we do not know nothing about the initial state s0 (in absence of measurements), we can use the uniform prior distribution over a bounding box for s0, which is more appropriate.

For the initial step (correction only), we need to sample s0 directly from the absolute prior P(s0) and give the weights according to the first measurement w0(i) = P(z0|s0(i)).

# Part II – Visual modalities for object tracking

## *Lecture 6 – Color-based object tracking*




Introduction:

Color-based object tracking consists in obtaining the pose of an object, by using a model of its color distribution: for example, if we wish to track a hand, we can look at regions of pixels with skin-color values.

This requires the specification of a more or less precise statistical model description: in simple words, for a multi-colored object we could say “this object, from a given viewpoint, shows 70% red pixels and 30% green pixels”.

That kind of description is formalized by a (possibly multi-modal) color statistics: what is the probability of observing a given color, into the pixel area where the object is located:

$P(x=c \mid x \in O)$

Where c is the observed color at pixel x, and O is the “object class” (as opposed to the background class, B).

This is the color likelihood of x, supposed to belong to O.
Then, if we want to classify individual pixels as belonging to the object or background, we can say:

$x \in O$ if $P(x=c \mid x \in O) > t$; else $x \in B$

with a given probability threshold t.

If we also have a background color statistics: $P(x=c \mid x \in B)$, then we can do a more robust reasoning:

$x \in O$ if $P(x=c \mid x \in O) > P(x=c \mid x \in B)$; else $x \in B$

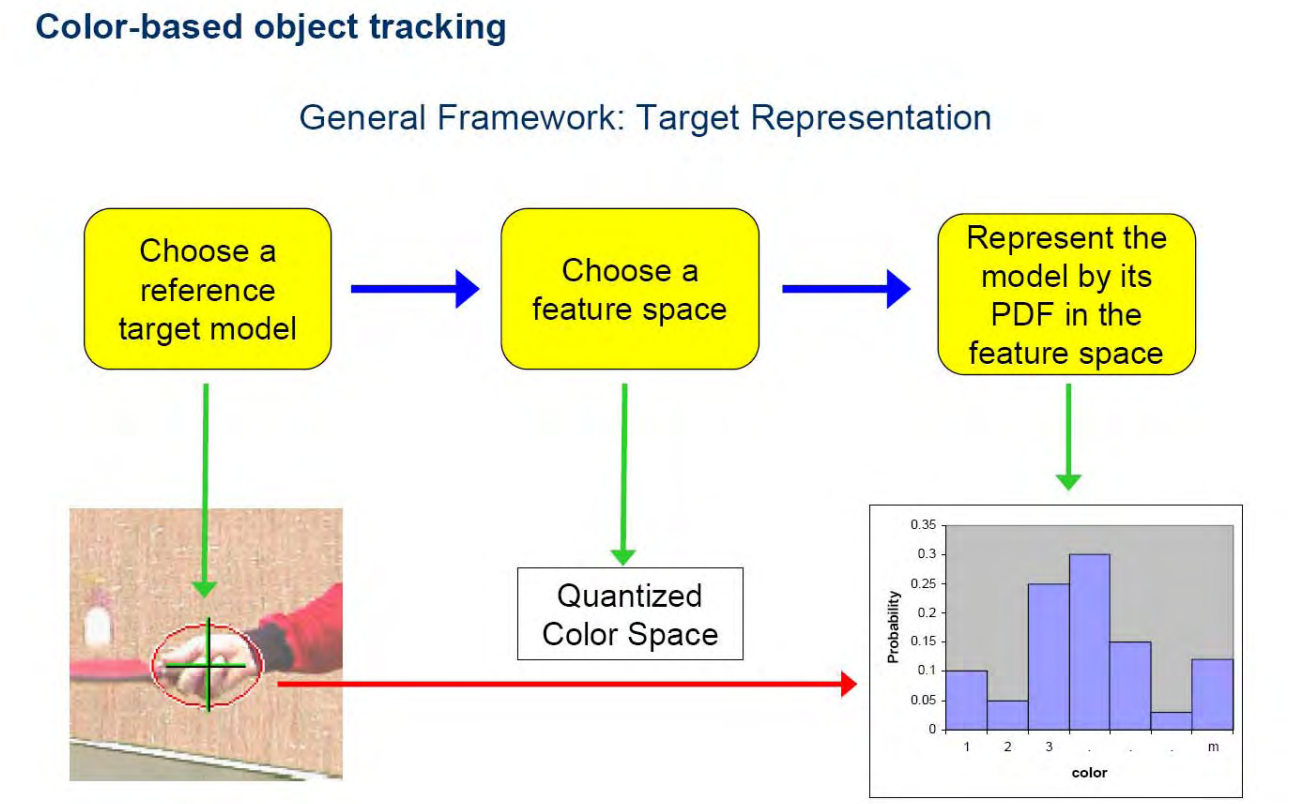
that uses no threshold, but rather compares the two probabilities of x belonging to O or to B, respectively.

A background statistics may be also pixel-dependent: each pixel location may have a different, expected color value, such as the gray pixels on the bottom and the brown pixels on the walls in the picture above.

As we can see from the picture (left frame), a color statistics can be obtained from a reference image, manually initialized, selected by the user.
Pixels collected from this frame are used to build the statistical representation $P(x=c \mid x \in O)$.
This representation constituters our *feature space* for color-based tracking.

We notice here how this kind of description has no information about the *location* of expected colors: in other words, by saying *how many* pixels in the region have a given color (e.g. 70% red+ 30% green), we do not say *which* pixels are expected to be red and which ones green, respectively.

Therefore, this model is of a purely *photometric* nature: no geometric information (feature location, size, etc.) is contained in it.

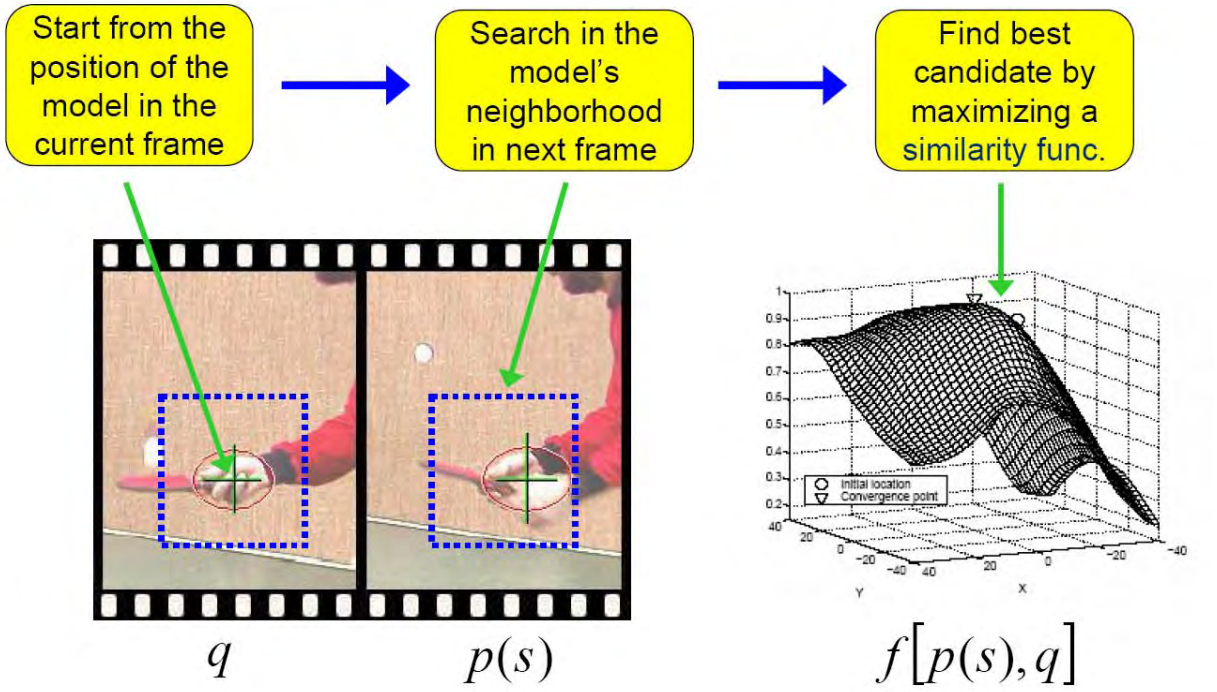



Once a reference picture of the target has been taken, a suitable feature space (in our case, a quantized *color space*) needs to be chosen, as well as the representation form of the probability distribution.
For example, the picture above shows how a color *histogram* can be used to represent the distribution: each cell of the histogram corresponds to a given color range (e.g. red, green, yellow, blue, etc.), and the histogram is obtained by collecting color pixels from the reference model, and accumulating them in the respective histogram cells. Finally, the histogram is normalized by dividing every cell by the sum of cells (so that the sum of probabilities is 1).

A color histogram is a feature descriptor, which has as many degrees of freedom as (number of cells – 1), since the last one can be obtained from the others (the sum is always 1).

## Mean-Shift Object Tracking

### Maximizing the similarity function

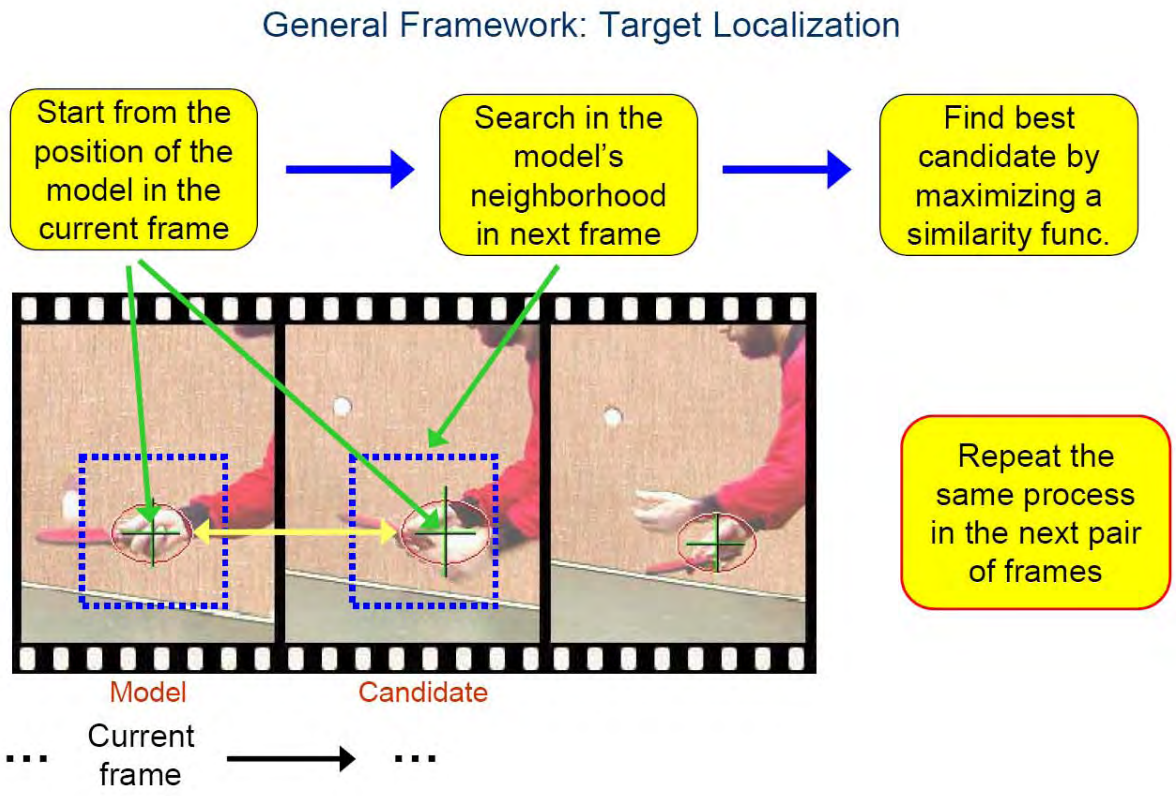



After the feature-space model (q) is obtained, it can be used in order to localize the object in subsequent frames: in fact, by moving the window to another position (s) and re-computing the histogram in the new image region, we get a different feature descriptor p(s).
By comparing the two histograms using a suitable *similarity* (or *likelihood*) measure f(p,q), we can search for the position s that *maximizes* this similarity.

This is a Maximum-Likelihood problem in feature-space, for which we can find the highest value in a *neighborhood* of the initial point $s_0$ (that means, it is a local search method).

In order to do the optimization, we need to formulate the likelihood f( ) in an appropriate way: it should be a smooth surface in state-space (s), and we should be able to compute easily also the derivatives with respect to s.

The mean-shift approach, that will be explained later on, is one such maximization methods, which can be applied if the probability densities p and q are represented by *kernel* functions.




**Color-space representations**

In order to perform color-based tracking, we need first to talk about the color-space representation of our image and model.

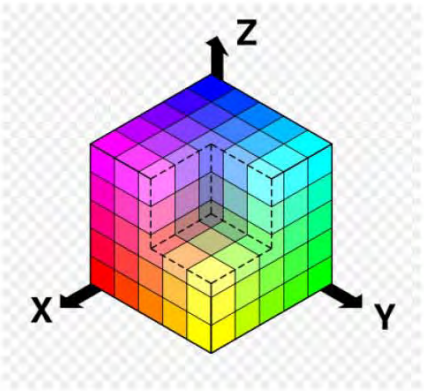
Each unique color that can be observed by human vision (or by a digital camera) and represented on a media such as paper, monitor screen, etc., can be represented by a t-uple of coordinates, which are most usually 3.

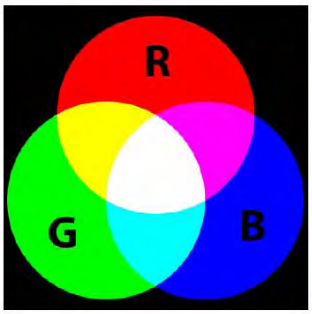
There are many different representations of color, which are called color-spaces.

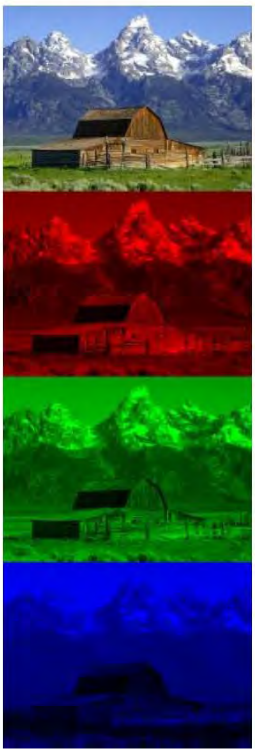
Color space = a mathematical model describing the way color can be represented by t-uples of numbers (typically t=3)

Color can be represented in different ways

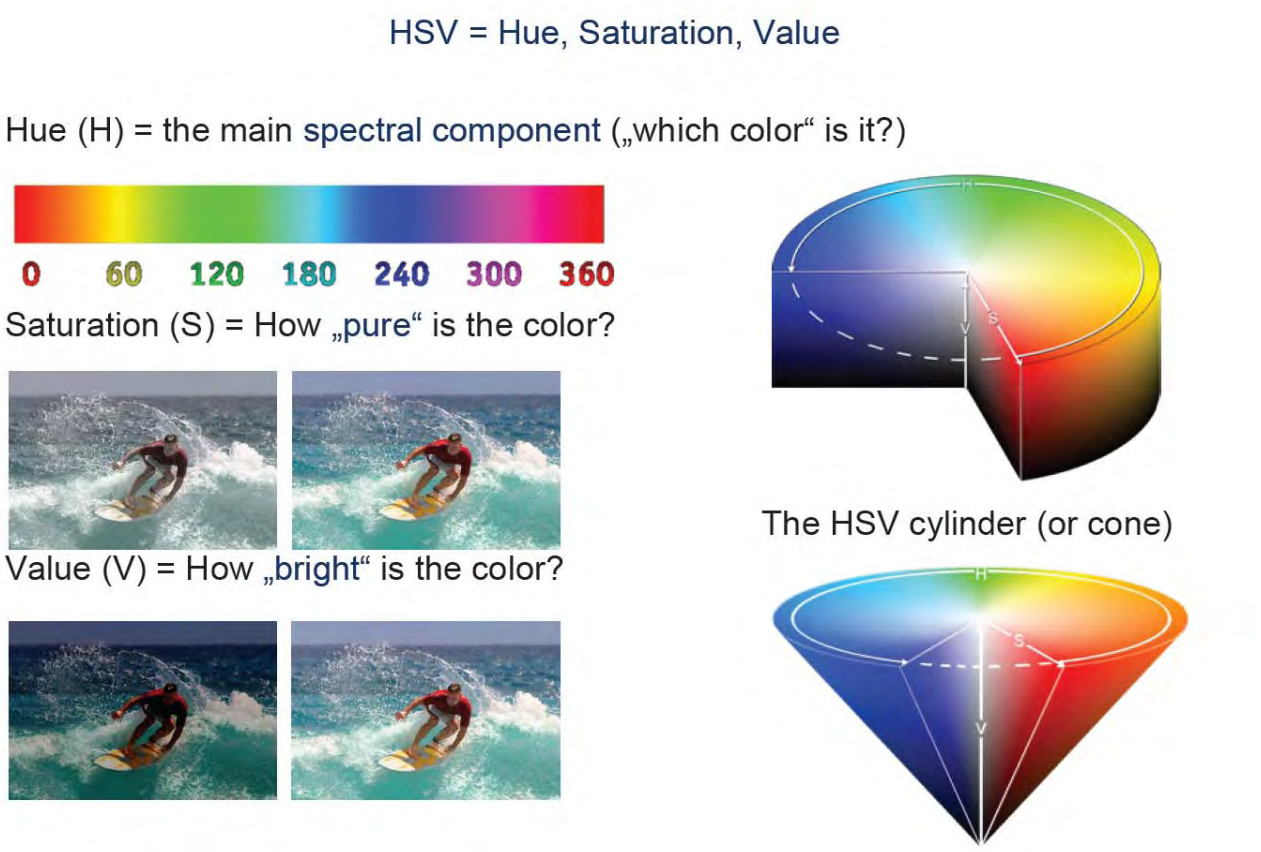
On a computer: RGB – red, green, blue components



Usually, on a PC we use the RGB representation: the amount of red, green and blue components that make up the color. They correspond to superimposing different intensities of the three primary lights onto a screen, which are mixed to produce the desired color.




The RGB representation is quite popular; however, it does not exactly correspond to a natural description of visible colors, in terms of perceptual properties (such as “how bright a given color looks”), nor it corresponds to the real physiology of the human retina (i.e. the color receptor cells).
A more perceptually meaningful representation is the HSV (Hue-Saturation-Value).
The first component (Hue) represents the *main spectral component* of the color: in other words, it answers the question “which color is this”? (a red, a green, a yellow, a violet, …). Physically, it corresponds to the main *frequency* that the light wave carries.
The Saturation component tells how *pure* the frequency spectrum is: if a color is highly saturated, it has only the main component (the hue), whereas a non-saturated color is a large mixture of all nearby frequencies, and looks less “pure” (or less “colourful”). The extreme case are gray colors (which include also black and white), that contain an equal mixture of all visible frequencies.
Finally, the Value is the light *intensity* (or perceived brightness), which roughly corresponds to the amplitude, or energy, of the light wave.

Unlike the RGB “cube” (which is a linear space), the HSV color space can be more conveniently represented on a *cylinder*, where the main axis (V) represents varying brighness, the radius is the saturation component (that is null on the axis of grey colors), and the angular component is the Hue value.

As we can see, for low values of V all colors tend to appear black, while at high values they are very well distinguished. Therefore, a better representation is the HSV *cone*, where colors are distributed in a more uniform way for all HSV triplets.

Nonlinear conversion form RGB to HSV:

Let (r,g,b) be between [0,1], and

max = max(r,g,b); min = min(r,g,b)

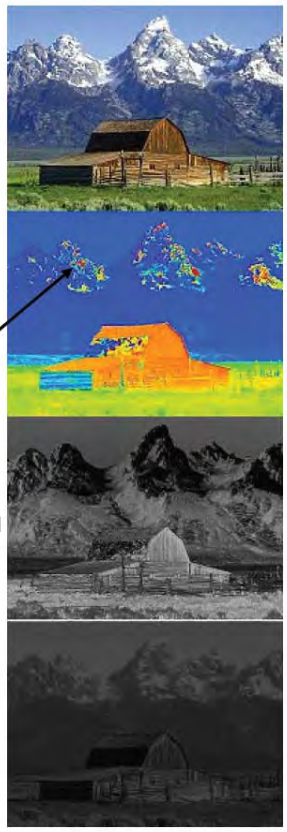
$$h = \begin{cases} 0 & \text{if } \max = \min \\ (60° \times \frac{g-b}{\max - \min} + 0°) \bmod 360°, & \text{if } \max = r \\ 60° \times \frac{b-r}{\max - \min} + 120°, & \text{if } \max = g \\ 60° \times \frac{r-g}{\max - \min} + 240°, & \text{if } \max = b \end{cases}$$

$$s = \begin{cases} 0, & \text{if } \max = 0 \\ \frac{\max - \min}{\max} = 1 - \frac{\min}{\max}, & \text{otherwise} \end{cases}$$

$$v = \max$$

RGB→HSV: there are singularities (S=0 → any H)

HSV→RGB: No singularities



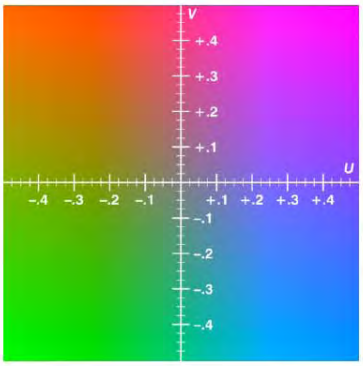

This is in any case a non-linear representation which may have *singularities*: in fact, for colors with S=0, the Hue component makes no difference (there is no “main” component). Therefore, the conversion between this space and another space like RGB contains a singularity: for a given HSV value, the RGB value is always unique, but for a given RGB color, there may be infinite HSV values (when S=0).

Y = Luminance (~brightness)

U,V = Chrominance
Used in color TV transmission

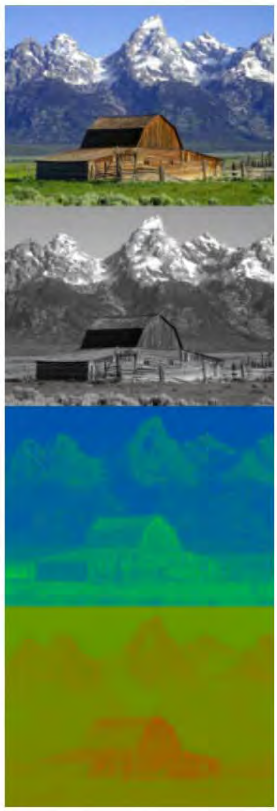
(Digital TV → Cr,Cb)

Conversion YUV / RGB: Linear




$$\begin{bmatrix} Y' \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.14713 & -0.28886 & 0.436 \\ 0.615 & -0.51499 & -0.10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.13983 \\ 1 & -0.39465 & -0.58060 \\ 1 & 2.03211 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ U \\ V \end{bmatrix}$$

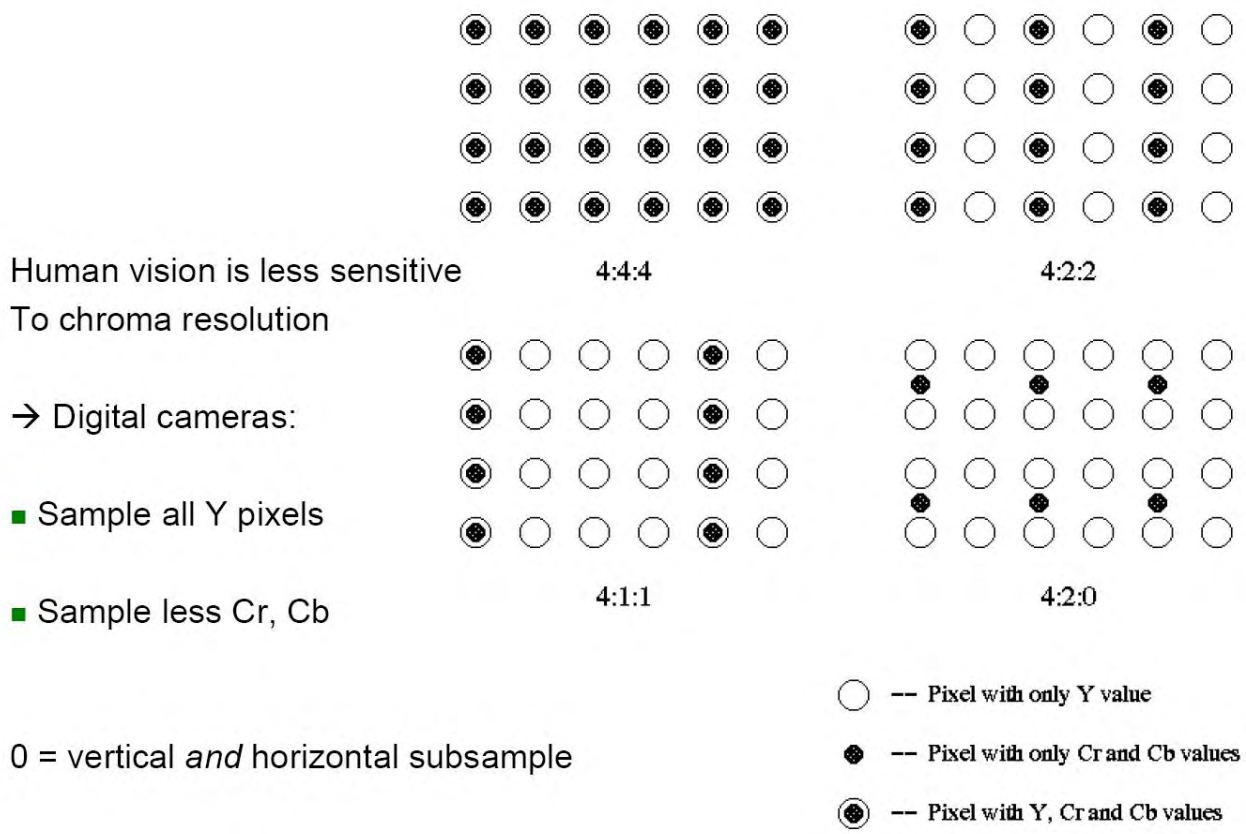


Y

U

V

Another representation, mostly used by digital cameras (CCD sensors), is the YUV space: this is similar to the HSV representation, in that it separates the pure brightness component (Y, which is related to the Value, but not exactly the same) from the two *chromatic components* (U,V).
However, the meaning of (U,V) plane is quite different from (H,S), and in particular, this representation has a *linear* relationship with the RGB space.




In digital cameras, in order to reduce the transmission bandwidth, often input pixels are sub-sampled: not all values of (Y,U,V) are sent along the cable, but some intermediate values are missing, and they need to be interpolated (often, simply repeating the last one) in the resulting image.
Sub-sampling takes place on the U,V channels only, since human vision has a lower spatial resolution for chromatic components, than for the intensity values. In fact, intensity values are a more important part of early visual processing (which is the processing done by the neuron layers of the retina).
There are several sub-sampling schemes in commercial devices, that are indicated by 3 numbers Y:U:V. The 4:4:4 format corresponds to the full transmission. 4:2:2 takes 2 values of (U,V) every 4 of Y (that is, one every two pixels are missing). 4:1:1 is even more sub-sampled, and 4:2:0 indicates that pixels are subsampled on both the horizontal and vertical direction: only one color value every two pixels, on both axes, is detected and sent along the cable.

## CIE–XYZ color space

(*CIE = International Commission on Illumination*)

Human receptors for color = retinal cone cells

X, Y, Z = tristimulus values → related (but not the same) to the cone responses

XYZ are unique for each perceived color

I(λ) = spectral distribution of a colored light

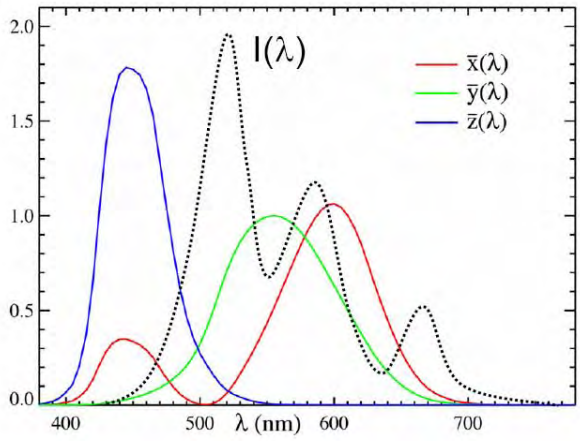
λ = wavelength

$$X = \int_0^\infty I(\lambda)\,\overline{x}(\lambda)\,d\lambda$$

$$Y = \int_0^\infty I(\lambda)\,\overline{y}(\lambda)\,d\lambda$$

$$Z = \int_0^\infty I(\lambda)\,\overline{z}(\lambda)\,d\lambda$$

Y = Luminosity (or brightness)




Another color-space, more related to human perception of color, has been developed by the CIE organization in the early 1930s and refined into the CIE-XYZ specification of 1970(?).

This model corresponds to the *tristimulus* concept: human receptors for color (retinal cone cells) compute one of three possible values, obtained by filtering the input light spectrum I(λ) by means of the tristimulus spectral responses (corresponding to the three color curves x(λ),y(λ),z(λ) above). This is achieved as an integral value, over all visible spectrum λ (with wavelength measured in nm)

Applying the tristimulus responses give, respectively, the X,Y,Z values of the colored light.

The three curves very roughly correspond to the red, green and blue components of the perceived color light. As a result, any light spectrum I(λ) with the same (X,Y,Z) values is perceived as the *same color*; therefore, our perceived color-space is three-dimensional.

In particular, the second component Y, roughly corresponds to the brightness (or luminance) of the color, but X and Z are not “purely chromatic” components (such as Hue and Saturation, or U and V, etc.)

## CIE–xyY normalized color space

(x,y) = chromaticity values

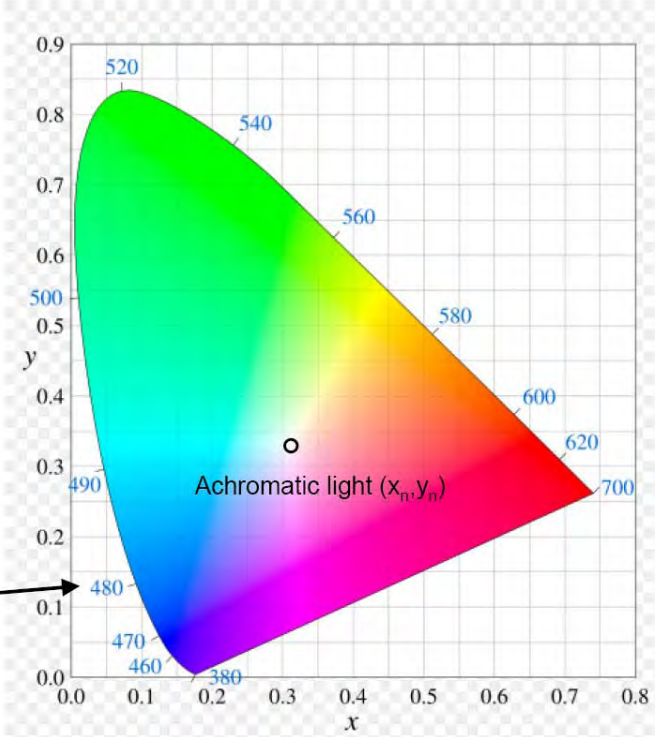
Y = brightness

$$x = \frac{X}{X+Y+Z}$$
$$y = \frac{Y}{X+Y+Z}$$
$$z = \frac{Z}{X+Y+Z} = 1 - x - y$$

The (x,y) coordinates cover all of the visible colors

Boundary = monochromatic locus

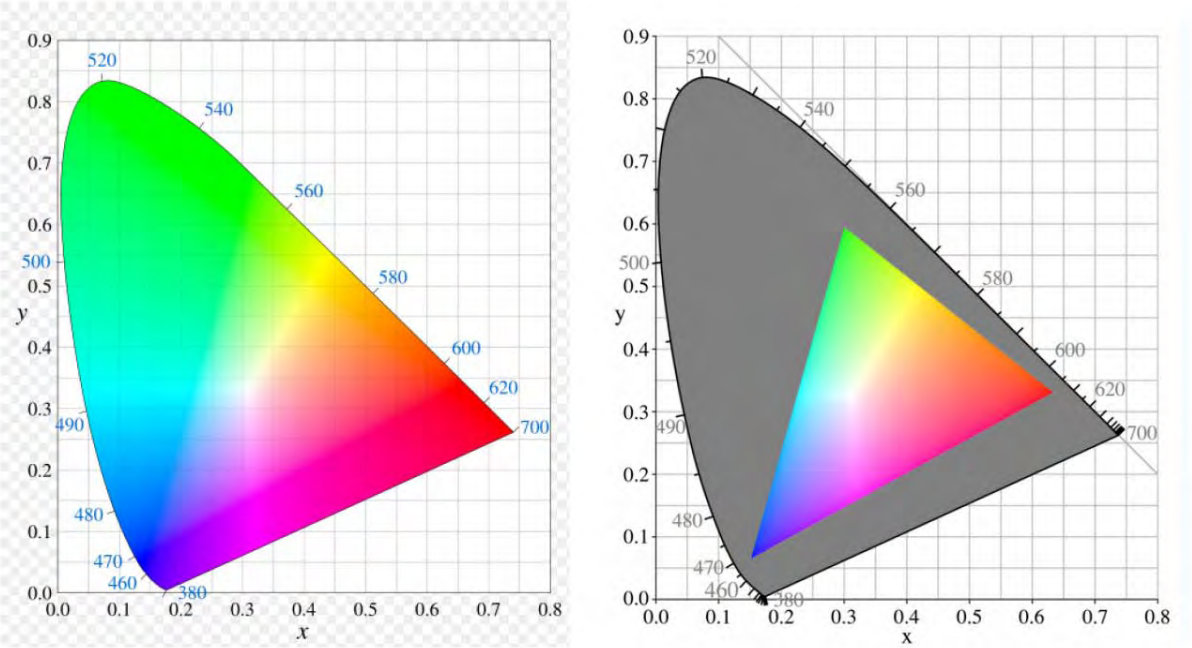
Wavelength λ of pure colors [nm]




If we normalize the X,Y,Z components, dividing by their sum, we get (x,y,z) coordinates, which sum to 1. Therefore, only two of them can be specified. And this normalization produces purely chromatic coordinates, independent of the overall brightness.
Then, by considering (x,y,Y), where Y is the brightness, we get a more suitable color space, with the desired separation between chroma and luminance components: this is the CIE-xyY space.

We can see that, on the (x,y) plane, visible colors occupy a “horse-shoe” region: the center is the “achromatic light”, with no saturation (gray), and the border contains pure colors (monochromatic locus). Along the border, the wavelength is specified, corresponding to the respective pure color.

Comparison: visible *vs*. monitor-representable colors




The visible color region is much larger than a common visualization device (such as a monitor, or a color TV) can actually display.
In fact, on the right side this sub-region is displayed, which contains all unique colors that can be displayed by a typical monitor.
All other colors seen here are just “copies” of the ones on the triangle border, but we can see many more colors, that are actually present in nature.

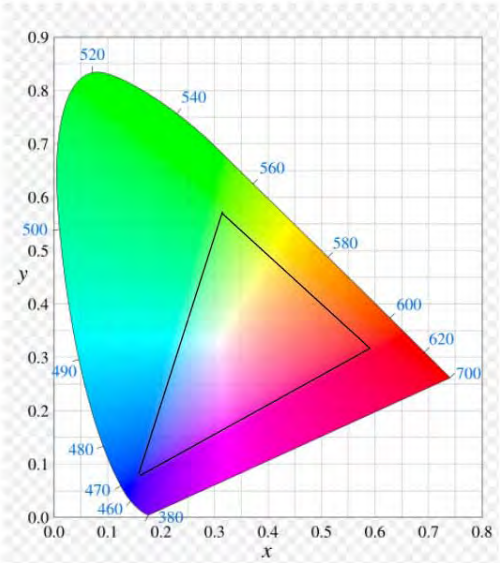
L*u*v*: transform XYZ in order to linearize perceptual color differences

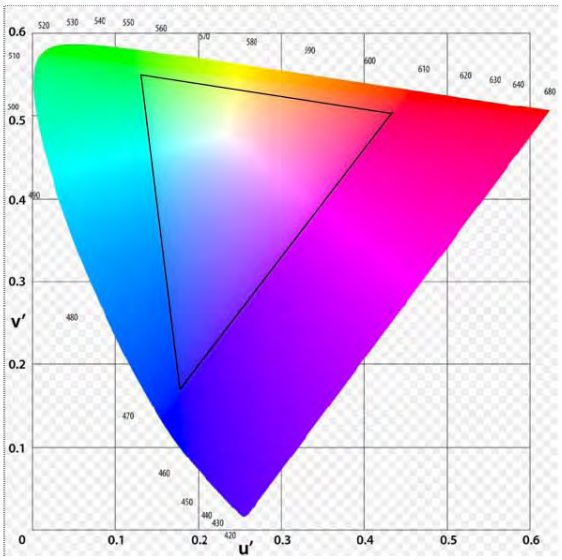
XYZ: Nonlinear distribution

- Similar colors may be far apart
- Different colors may be close

L*u*v*: Linear distribution

- Similar colors are close
- Different colors are far apart




Looking at the interior colors of the triangle, the problem with the CIE-xyY space is the *non-uniformity* of perceptual color differences: in the center, two colors that look similar can be far apart, while on the border, two different colors are close.

This is not desirable for computer vision, since clustering colors in an image works better if their perceptual differences are roughly proportional (i.e. linear) to their distance in color-space. In other words, “perceptual clusters” of pixels are better distributed, and separated, in color space.

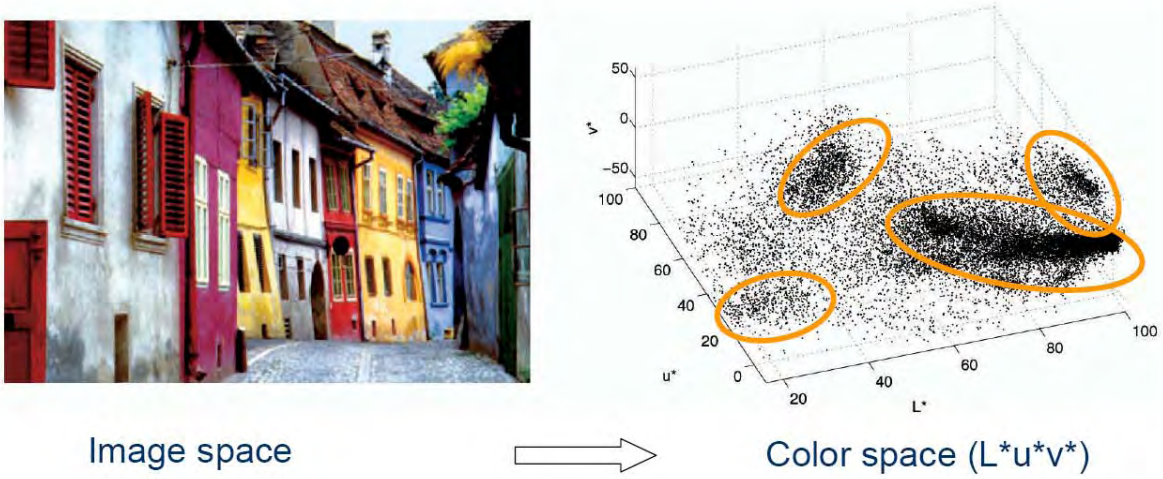
Therefore, the L*u*v color space attempts to modify the xyY space through a non-linear transformation, so that this “perceptual linearity” is achieved: similar colors are close, different ones are far apart.

**Modeling color distributions**

An image (or a sub-region) contains a set of colored pixels.

The ensemble of pixels can be characterized by its color statistics

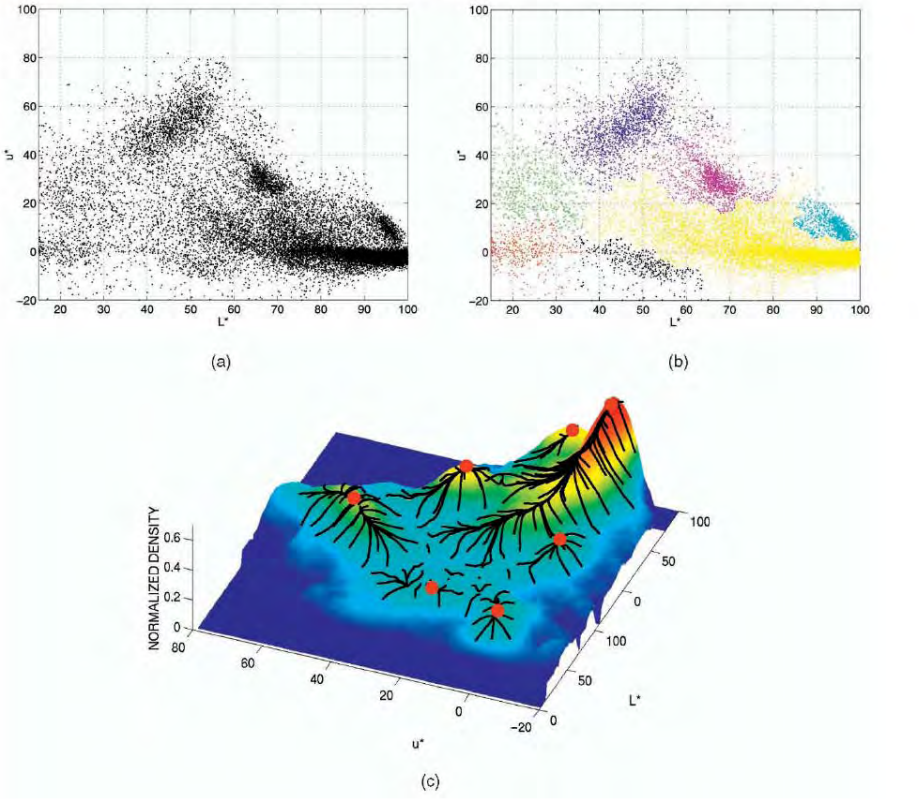
Color statistics = density in color space represents the probability of occurrence of a given color




In order to perform a color segmentation, we can look at the distribution of color pixels, in a suitable color space. Often, the L*u*v* space is chosen, because of the linear distribution of perceptually different colors, so that pixels form good-shaped clusters in this space.

Clustering is a procedure which finds high-density regions in color space, and assigns pixels to each cluster. Most often, clusters have an ellipsoidal shape, since they represent a mixture of Gaussian distribution (or color-likelihood function).

## Color clustering: find local maxima (modes) of the color distribution




A clustering method looks for the local maxima (or *modes*) of this color likelihood, and assigns each pixel to the respective *basin of attraction*: all pixels for which the local maximization leads to a given mode, are assigned to this cluster.

What is needed for such a clustering procedure is, therefore: defining the color likelihood (that should be proportional to the local point *density*), and the local maximization procedure.

Segmentation = look for clusters in feature space

Clusters = groups of pixels with

- Similar color (close in color space)
- Similar position (close in image space)

} Feature space = (x,y,l,u,v)

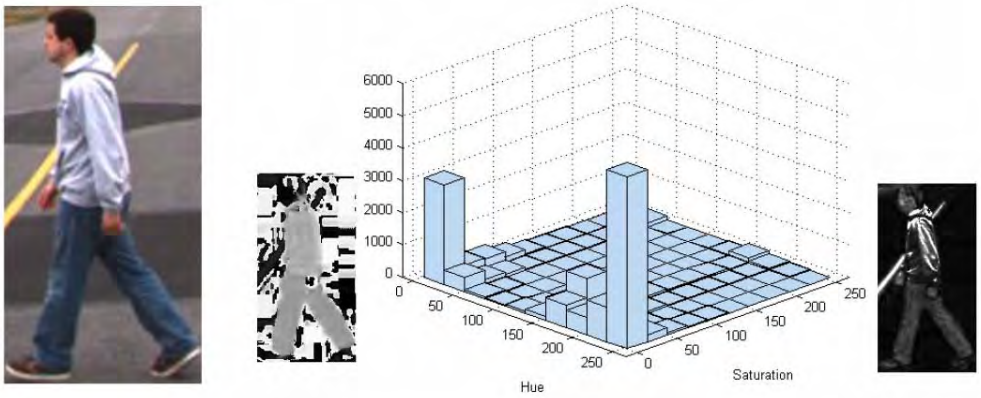
Usually, clustering only in color space is not sufficient: in fact, for a good image segmentation, we are also interested in spatial relationships between pixels (for example, the chair has the same color of the stripes on the painting above, but they are two different objects).

Therefore, in image segmentation, the feature space consists of both position (x,y) and color values (l,u,v), so that clustering takes place in a 5-dimensional space.

Statistical representation 1: color histograms

Histograms can be given in 1- 2- or 3- color space
(typically, only Hue and Saturation channels)

→ Non-parametric (the only parameter is the number of cells)



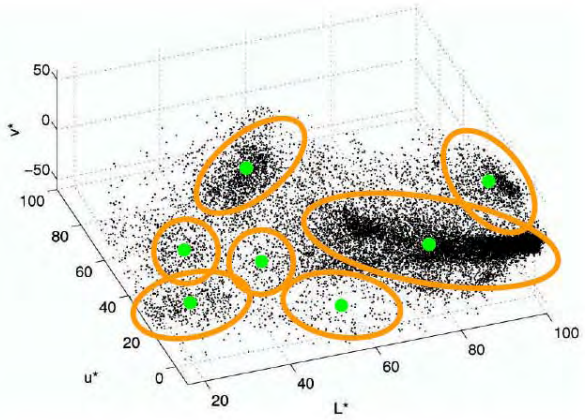

A first representation of the color likelihood is given by histograms: each bin represents a given interval of color values, where pixels are collected. High peaks on the histograms represent prevalent colors in the image.

Histograms are easy to understand and to collect, and they are a non-parametric representation: the only parameter is the number of bins (i.e. the resolution in color-space), which however should be carefully selected.
A too low resolution (few bins) leads to a poor and ambiguous representation, since it neglects small color differences; on the other hand, too many bins lead to an “over-fitting”, that is, noise and small color variations give very different histograms, and therefore many small segmentation regions.

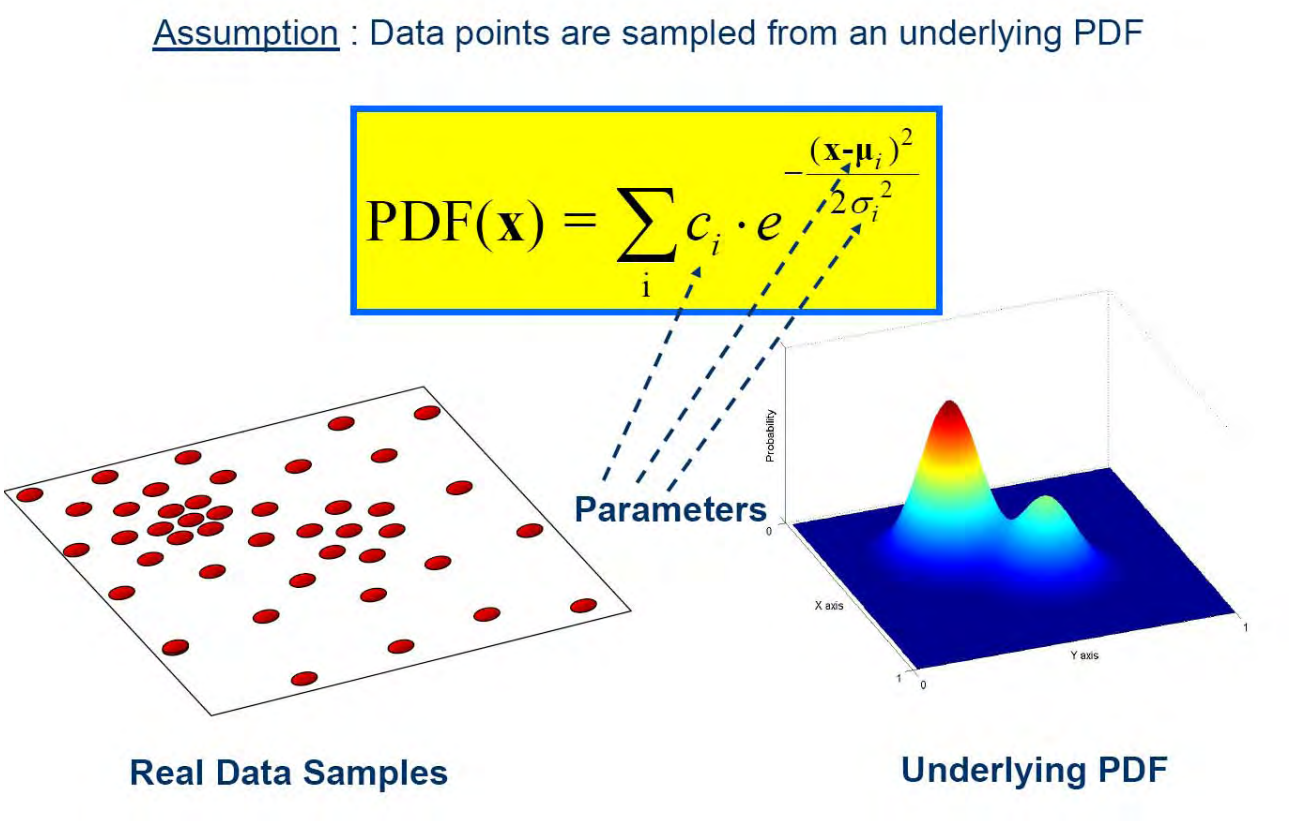
There is no standard rule to select the bin size, but it should be anyway proportional to the sample size: in fact, a higher resolution can be used when many pixels are available, while for a small sample size, the number of bins should be kept low.

Statistical representation 2: mixture of Gaussians

→ Highly parametric: all means, covariances and weights are needed

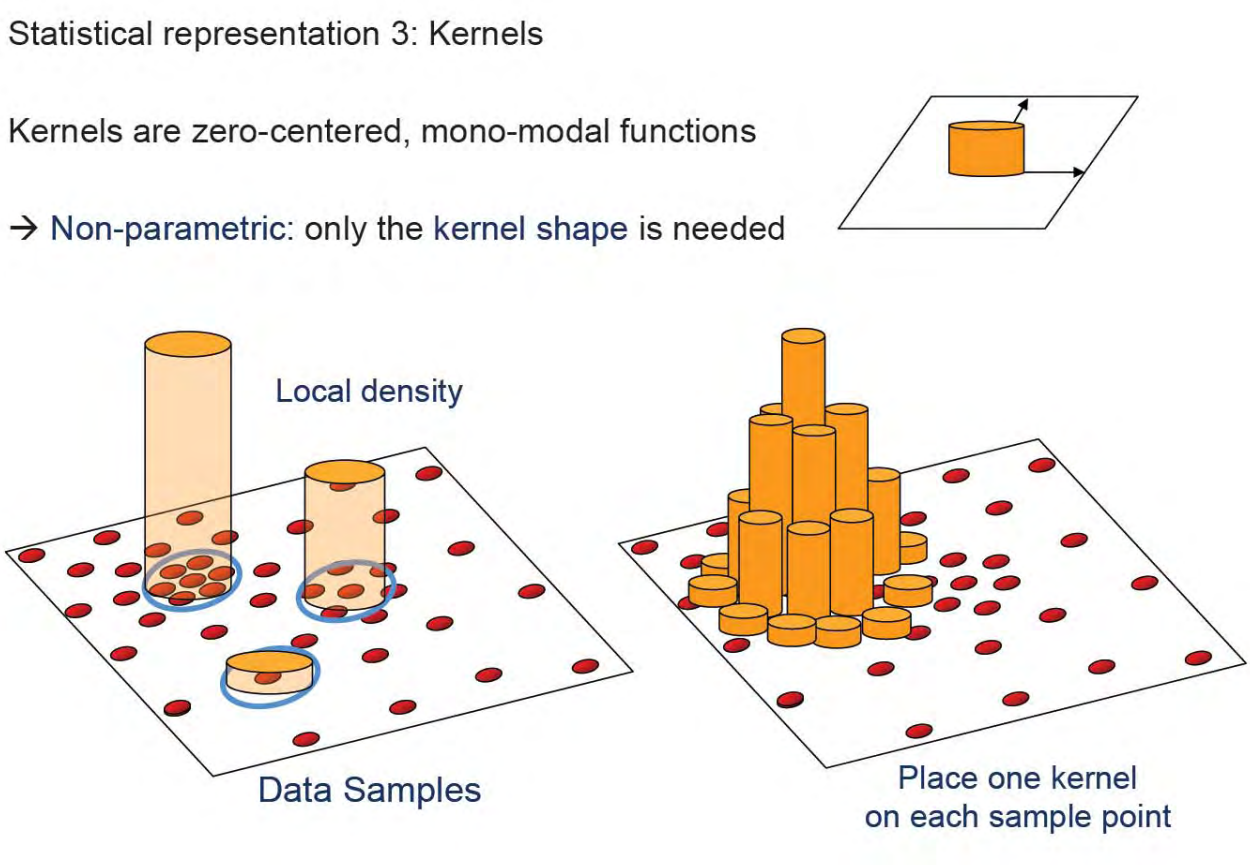



Another useful representation is given by Mixtures of Gaussians: in fact, each cluster can be usually approximated by a multi-variate Gaussian distribution, with given mean vector (the mode, or average value) and covariance matrix (the extension and shape of the cluster).




However, this model is highly parametric: all mean vectors, covariances, weights, and even the number of Gaussian components (i) have to be determined, for a given sample set. This can be accomplished by using the Expectation-Maximization (EM) algorithm, which has some computational complexity, but for low-dimensional spaces like this, can be efficiently implemented.
Still, the problem of estimating the number of components (which is an integer value) has to be solved, and in the literature some methods (or simple rules) have been indicated for this purpose.

The advantage of using Mixtures of Gaussians is the smoothness of the resulting color likelihood: all derivatives can be computed, which is very useful for tracking (frame-to-frame estimation of the likelihood function)
For image segmentation, the Gaussian already represent the modes of the distribution, unless too many components are present (in which case, two clusters which are too near can always be “merged” in one).




A third representation, which joins the best properties of both the previous ones, is given by *kernels*.

Kernels are a non-parametric representation, that unlike histograms is also smooth and differentiable, so that local modes can be found by gradient ascent maximization, in a relatively quick time.

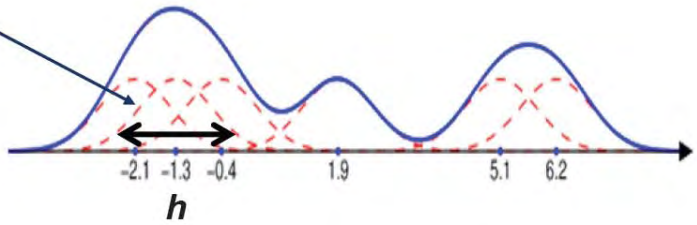
The idea behind kernels is to place a uni-modal function with a compact support (kernel) onto each sample point, and summing them together to obtain the resulting function.

This looks like a mixture of Gaussians (the kernel can also be a Gaussian), but it is not: no parameters (mean and covariances) have to be specified, since all kernels are located on sample points, and have a unique covariance, which is the only parameter to be specified.

In this way, high-density regions will have many neighboring contributions, and give high likelihood values, and vice-versa for low-density regions.

$$\hat{f}_h(x) = \frac{1}{nh^d}\sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

$$K(x) = Gauss(x)$$




Typical kernel properties:

1. Compact support $\|\mathbf{x}\| > t \Rightarrow K(\mathbf{x}) = 0$
2. OR: Exponential decay $\lim_{||\mathbf{x}||\to\infty} ||\mathbf{x}||^d K(\mathbf{x}) d\mathbf{x} = 0$
3. Normalized $\int_{R^d} K(\mathbf{x}) d\mathbf{x} = 1$
4. Symmetric $\int_{R^d} \mathbf{x} K(\mathbf{x}) d\mathbf{x} = 0$

The kernel function K(x) has some basic properties, in order to provide a meaningful probability density estimate (f(x)).

In particular, the first condition (compact support) ensures a limited influence to each point from the overall sample set: only the nearest neighbor samples contribute to the value of f(x).
Alternatively, one can specify a kernel with very fast (exponential) decay to 0, such as a Gaussian.

In any case, the integral of the kernel function must be 1 (normalization), so that summing up all kernels, and dividing by N, keeps the integral of f(x) to 1.

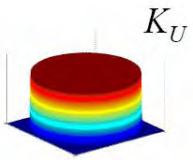
ERROR: the normalization coefficient ($1/h^d$) should be already included in K(x)!

Finally, usually it is desirable to keep symmetry of the kernel around the origin.

## Radially–symmetric kernels (*d* = space dimension)

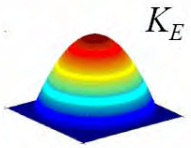
**Uniform**

$$K_U(\mathbf{x}) = \begin{cases} c_d; & \|\mathbf{x}\| \le 1 \\ 0; & \|\mathbf{x}\| > 1 \end{cases}$$



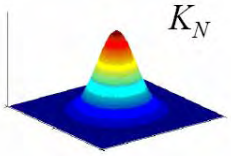

**Epanechnikov**

$$K_E(\mathbf{x}) = \begin{cases} \frac{1}{2} c_d^{-1}(d+2)(1-\|\mathbf{x}\|^2); & \|\mathbf{x}\| \le 1 \\ 0; & \|\mathbf{x}\| > 1 \end{cases}$$




**Normal (Gaussian)**

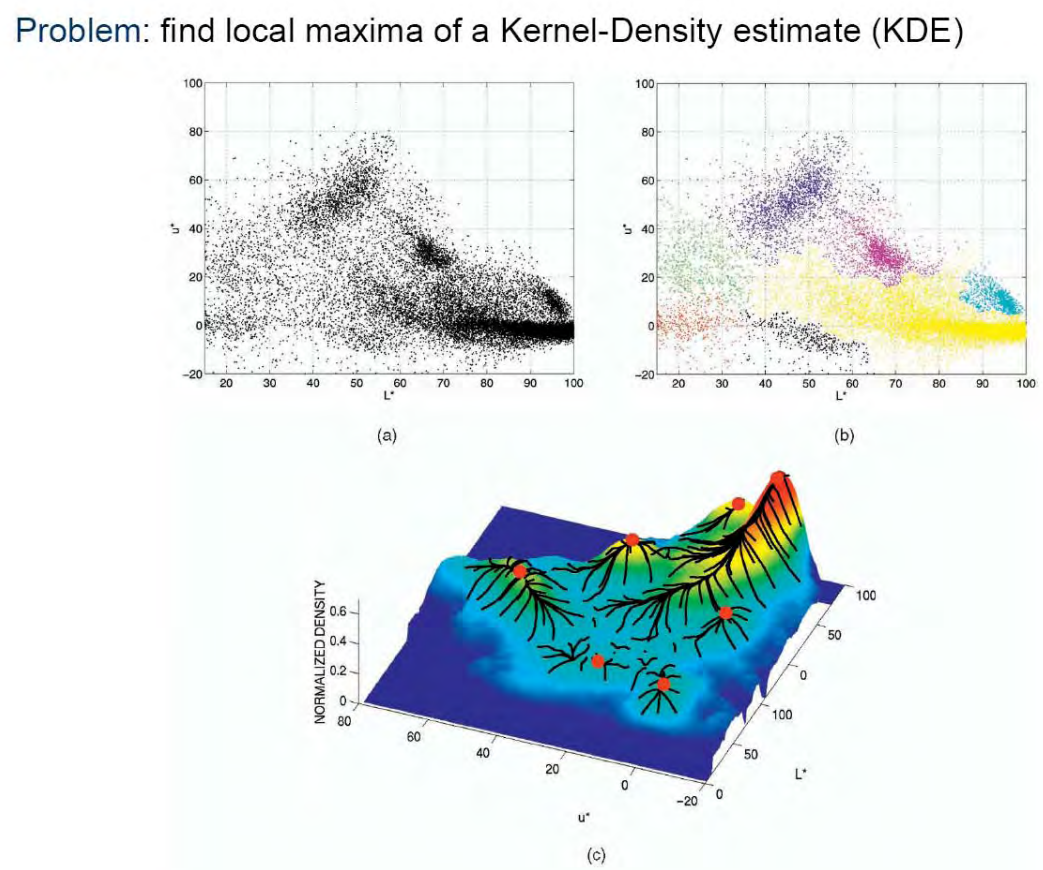
$$K_N(\mathbf{x}) = (2\pi)^{-d/2} e^{-\frac{1}{2}\|\mathbf{x}\|^2}$$




The most common choice is a radially symmetric kernel, meaning that the extension and shape of K(x) is the same in all directions.
In this case, we can substitute x with its norm ||x||, so that these Kernels can be conveniently expressed by a scalar function, K(||x||).

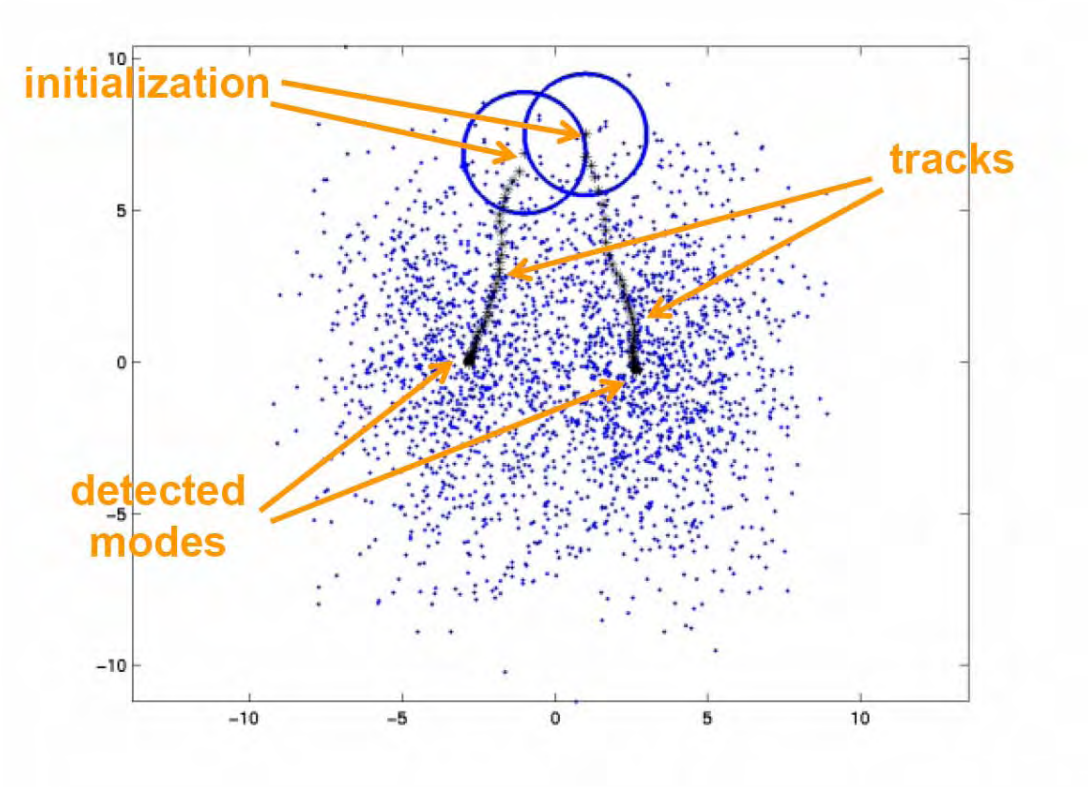
Well-known examples are: the uniform kernel (constant), the Epanechnikov kernel (quadratic) and the normal (or Gaussian) kernel. The first two have compact support, while the Gaussian does not; but anyway, it has an exponential decay, so that outside the covariance region, $K_N(x) \sim 0$.

**The Mean-Shift Algorithm: I – Definition**




As we said in the previous Section, clustering can be seen as finding the modes (peaks, or local maxima) of an underlying *likelihood* density, that models the probability from which the observed sample set is supposed to have been generated.

By using kernel estimators, this density function is given by a smooth and differentiable function, that can be locally optimized, starting from any sample point.



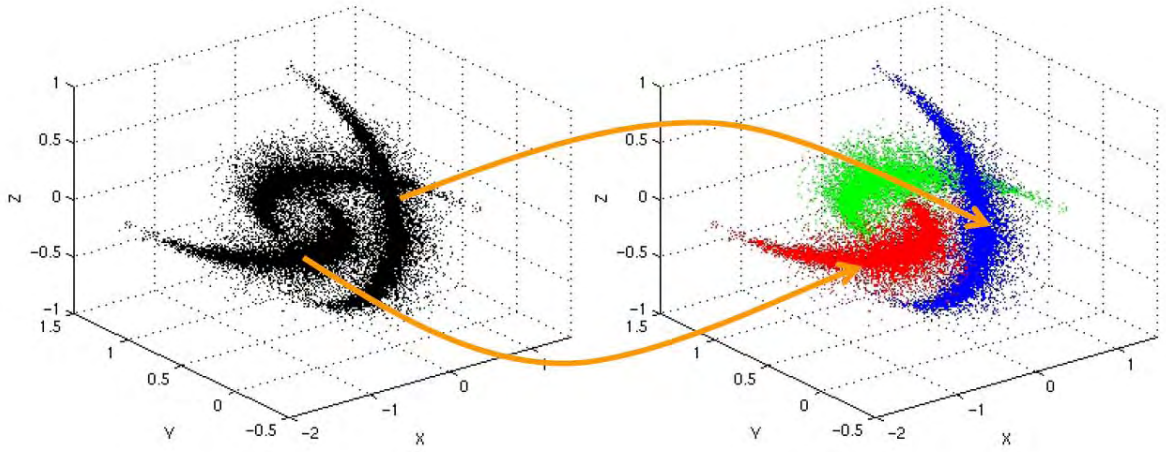

Each local maximization produces a *trajectory* in the feature space, which converges to one of the modes of the kernel density estimate. In particular, several trajectories converge to the same mode, and therefore form a cluster.

This procedure has the advantage of being completely unsupervised: apart from the sample data, no other information is given a-priori, with the exception of the choice of kernel (shape and size).

So, we compute automatically both the number of clusters, their centers, and assign all the cluster points.

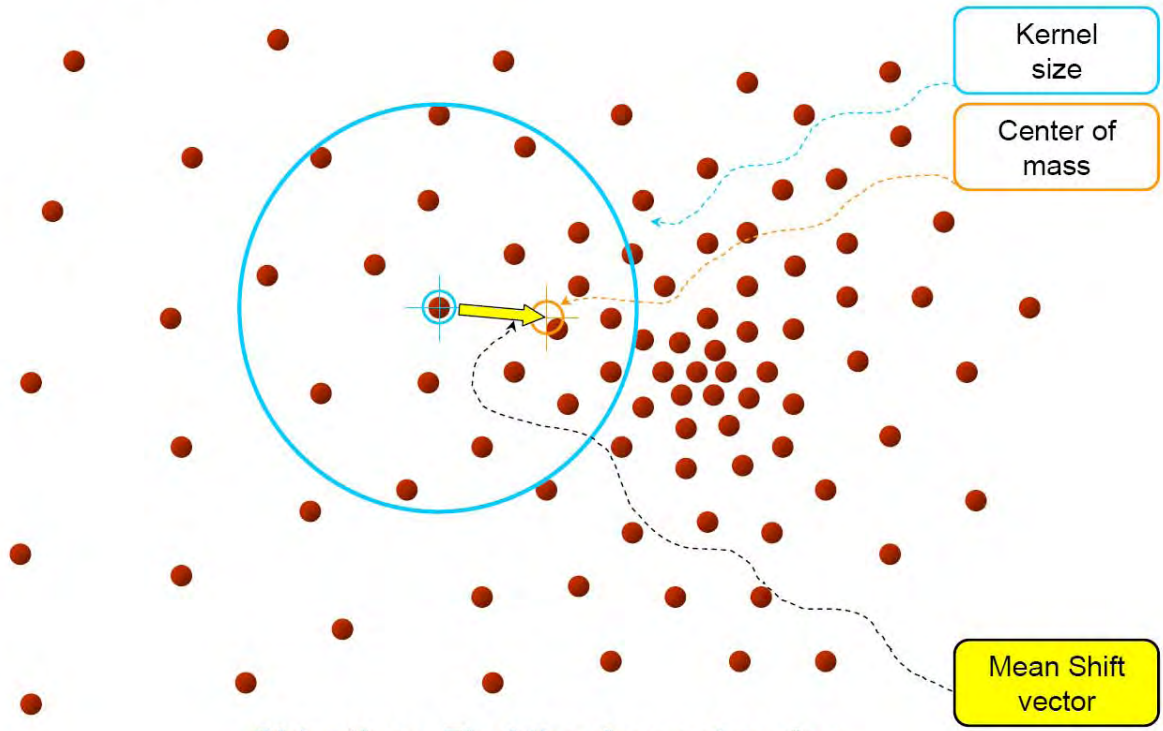
## Kernel-based clustering:

Can deal also with highly non-Gaussian clusters



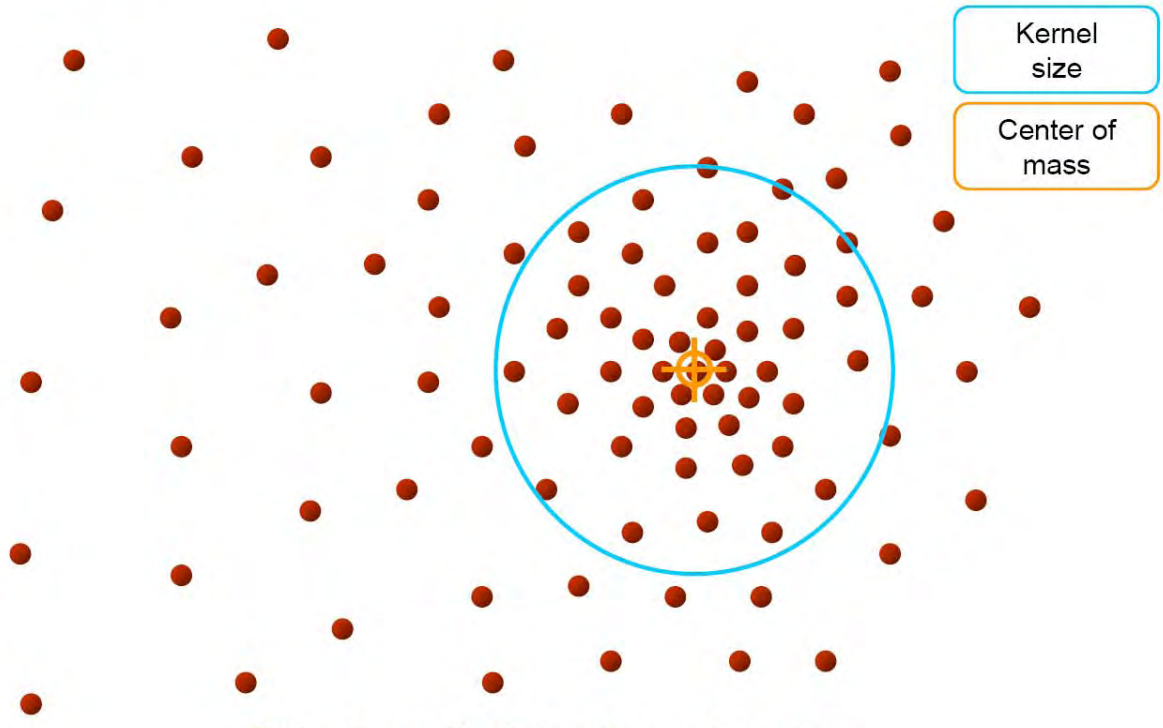

Another advantage of kernel density estimators, is the generality with respect to the underlying function, and cluster shapes: also very non-linear and non-Gaussian clusters can be separated with this method.

## Intuitive Description




**Objective : Find the densest region**

## Intuitive Description




**Objective : Find the densest region**

Intuitively, the goal of mean-shift procedure is the following: starting from a given sample point, we try to go in the direction of maximum density increase (i.e. the gradient of f(x)), evaluated in a region around the point. This region (blue circle) contains all sample points that have influence over f(x), therefore it has the size of the kernel K( ).

Afterwards, we move to the new point and do the procedure again, and iterate until convergence: we will then find the maximum density point, at least locally. Starting from any point in the picture above, lead to the same mode, so they belong to the same cluster.

## Gradient of Kernel Density

1–dimensional kernel: $k(x)$

*d*–dimensional kernel (radially symmetric): $K(\mathbf{x}) = c \cdot k(\|\mathbf{x}\|)$

Kernel density estimate: $\hat{f}(\mathbf{x}) = c\frac{1}{nh^d}\sum_{i=1}^{n} K\left(\left\|\frac{\mathbf{x}-\mathbf{x}_i}{h}\right\|^2\right)$

h = kernel size

For optimization we do not need $f(\mathbf{x})$:

ONLY the gradient !

Density gradient:

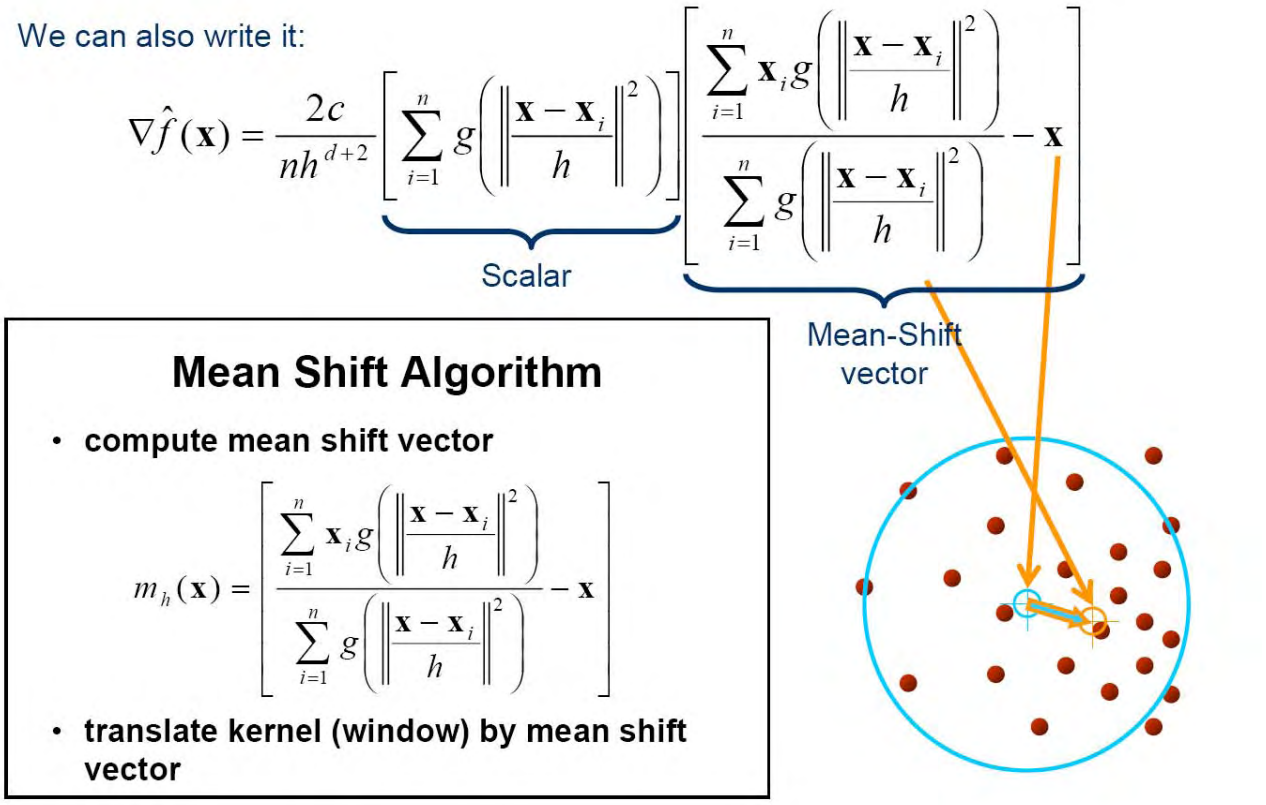
$$\nabla\hat{f}(\mathbf{x}) = \frac{2c}{nh^{d+2}}\sum_{i=1}^{n}(\mathbf{x}_i - \mathbf{x})g\left(\left\|\frac{\mathbf{x}-\mathbf{x}_i}{h}\right\|^2\right)$$

Kernel gradient:

$$g(\mathbf{x}) \equiv -k'(\mathbf{x})$$

For a radially symmetric kernel k(||x||), more precisely, the gradient of the kernel density can be evaluated with the formula above, where g(x) is the first derivative of k.

As we can see, also the gradient of f(x) has the form of a kernel density estimate, but with kernel g(x), and multiplied by the vector joining each point $x_i$ (in the neighborhood) to x.




By re-writing the formula above, we can see that the gradient of f(x) is given by two terms: a scalar term (kernel density estimate using g(x)), and a vector term, which is called the “mean-shift” vector.

The mean-shift vector gives the direction of motion from x to the new point (which is the new “mean” of the density). When the density is locally maximum at x, no shift will be present, and the algorithm will stop on x.

1. Run Mean Shift to find the stationary points

   - To detect modes: run multiple times, start from each feature point.

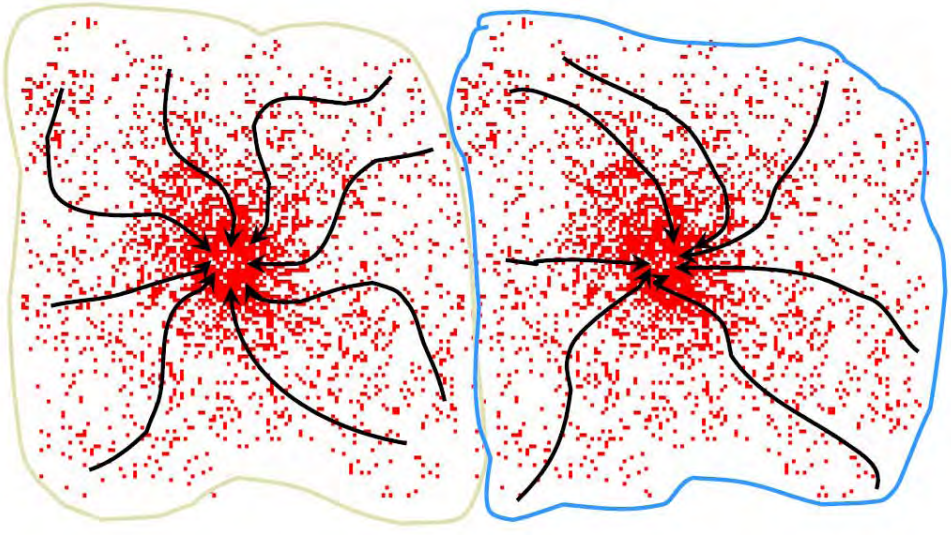
2. Prune the stationary points (local maxima)

   - Merge modes at a distance of less than the bandwidth.

3. Assign points to modes (cluster)

   - The basin of attraction of each mode delineates a cluster of arbitrary shape.

- Cluster: all data points in the attraction basin of a mode
- Attraction basin: the region for which all trajectories lead to the same mode



**The Mean-shift algorithm: II – Color segmentation**

Cluster pixels according to:

- Spatial position
- Color values

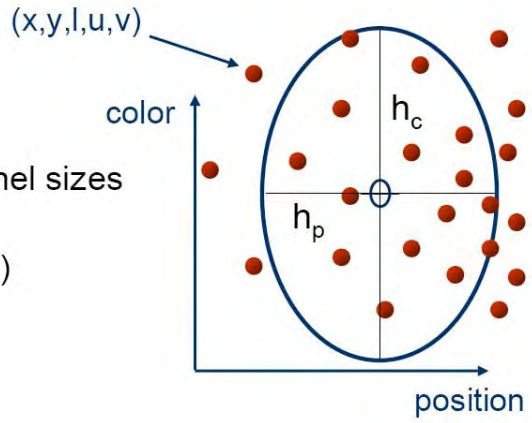
→ Define a 5-dimensional feature vector (descriptor):

(x,y,l,u,v) = position + color

→ geometric + photometric information

Now define the kernel in this space: use 2 kernel sizes

- $h_p$ = kernel size for position components (x,y)
- $h_c$ = kernel size for color components (l,u,v)

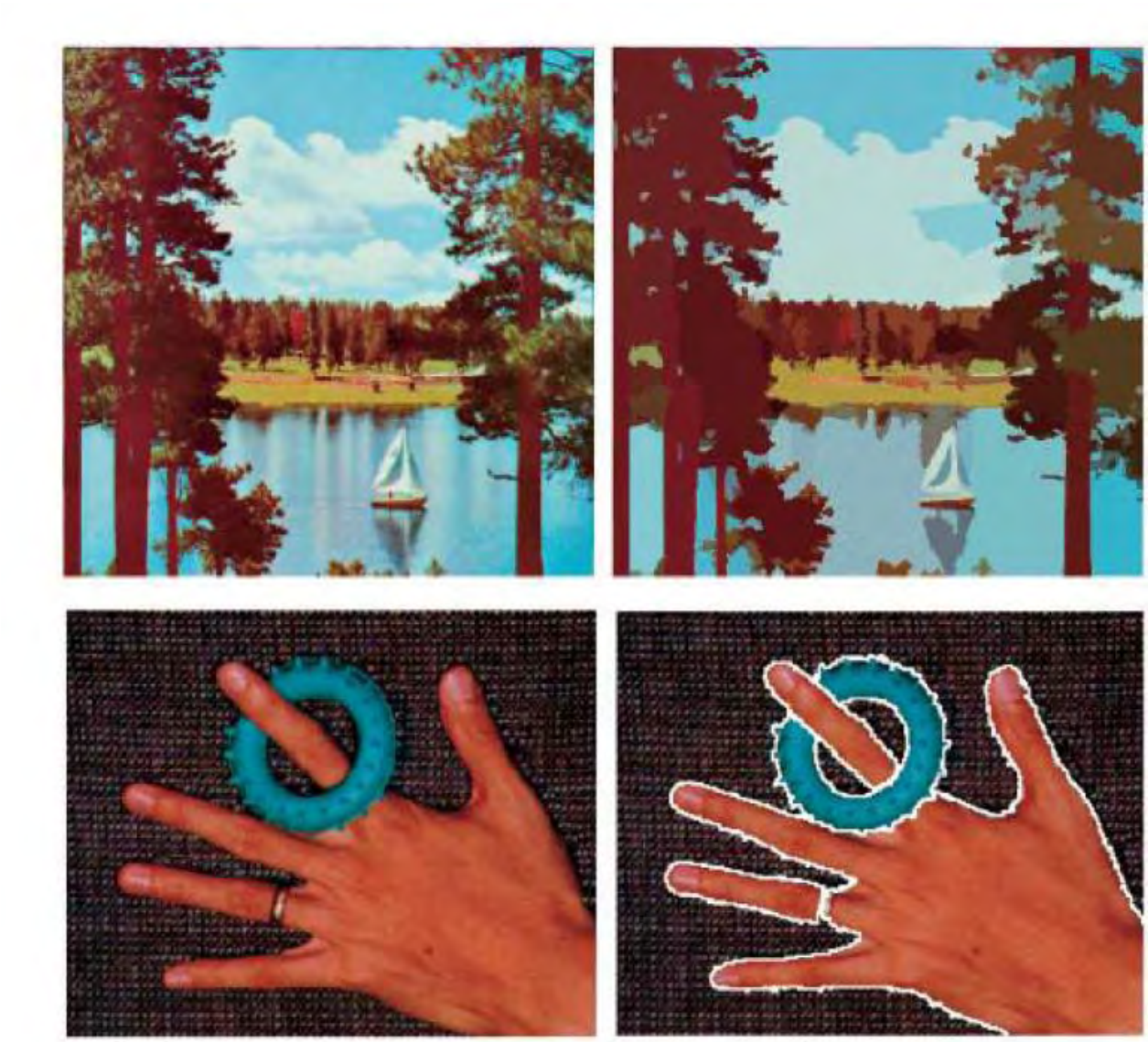



For color segmentation, it is important to include both geometric (positional) and photometric (optical) information. Therefore, a good methodology involves a 5-dimensional feature space, containing for every pixel its position (x,y) and color (l,u,v) in a joint descriptor.

In this case, the geometric and photometric components may have a different numerical range, therefore two kernel sizes ($h_p$ and $h_c$) are recommended. The mean-shift procedure can be easily applied also for non-isotropic kernels (i.e. with different sizes along different dimensions).

Some example of mean-shift image segmentation follow.

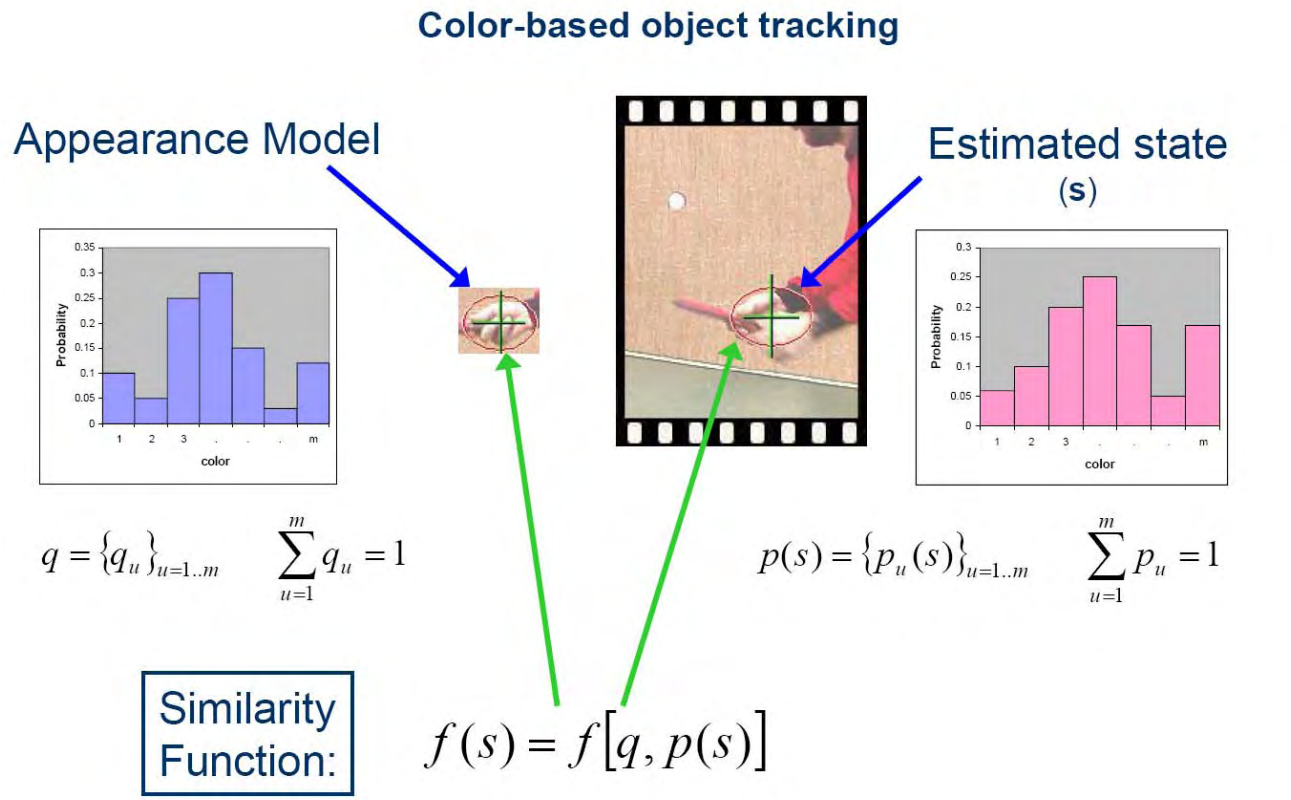
In this example, a gray-level image is shown (where the feature space is given by (x,y,l)). The color example shows how objects with the same color (e.g. the red chair and the red stripes on the wall) belong to different clusters, since they are separated in the position (although not in the color) components.



Natural scenes like the one above are generally more difficult to segment, since they contain complex textures, and complex shapes (for example, the trees).

The picture below shows how a highly non-Gaussian region (the blue ring), is still correctly separated from the finger and the hand. This is due to the generality of the kernel-based representation (a Gaussian mitzure could not perform as well).

**The Mean-shift algorithm: III – Object tracking**




Color-based object tracking can also be performed with the mean-shift algorithm; in this Chapter we will see how this can be accomplished frame-by-frame, by maximizing a color matching likelihood using a reference model of the object.

In particular, we need to define an appearance model of the object to be tracked, and a feature space where to define the descriptor for matching it inside the new image.

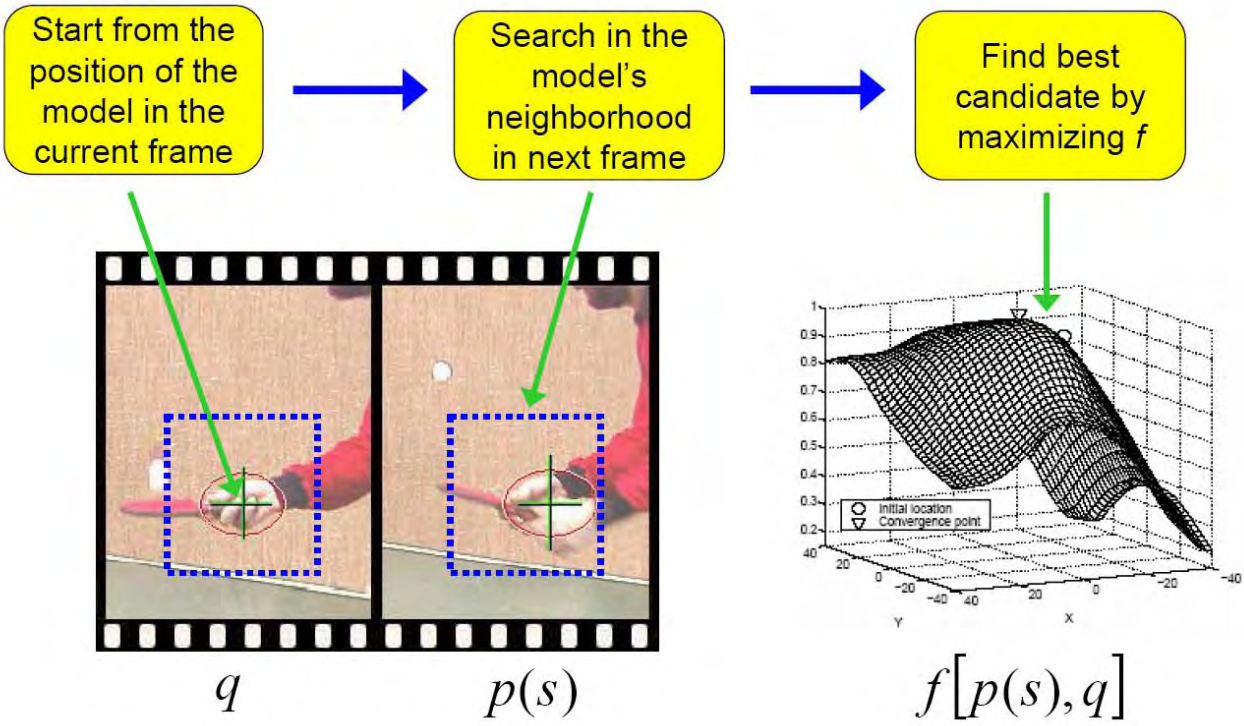
The descriptor here is given by a color histogram, obtained by collecting all the pixels inside the reference window (left).

This histogram can be described by a 1-dimensional vector $q_u$, where *u* is the bin index (although we may use two or three color channels, a 2- or 3-dimensional histogram can always be "vectorized" into one dimension).

On the new image, for a given pose hypothesis s (in this case a simple 2D translation) we can do the same and collect pixels in a new histogram $p_u(s)$. Both histograms are normalized to sum 1 (since they represent a probability distribution).

Then, we can compute a similarity function f(q,p) between histograms, that tells how good the state hypothesis *s* is.

## Maximizing the similarity function

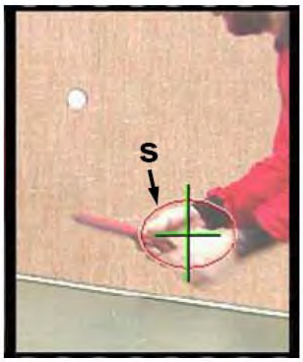



After defining f(p,q) we can look for the local maximum w.r.t. $s_{t-1}$, starting from the previous state $s_t$. Since from frame to frame the object does not move very much, if the similarity function is well-behaved (i.e. smooth, and with a large local peak), then we should most probably obtain the correct matching pose.

## How can we compare two color histograms?

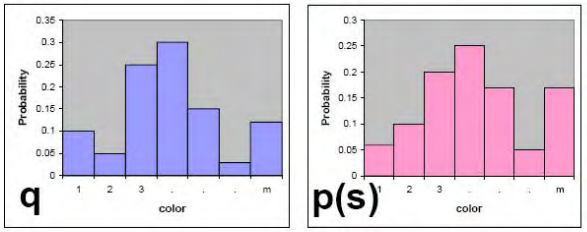
*f* = Bhattacharyya coefficient

$$f[p(s),q] = \sum_{u=1}^{m} \sqrt{p_u(s) q_u}$$




*B* = Bhattacharyya distance

$$B(s) = \sqrt{1 - f[p(s),q]}$$




## Minimizing *B* is equivalent to maximize *f*

A similarity function can always be converted into a *distance* function B (with the standard metric properties), so that maximizing f is equivalent to minimizing B.
In particular, here B is the so-called Bhattacharyya distance, defined by the formula above.

## Histogram similarity function

Target model: $\vec{q} = (q_1, \ldots, q_m)$

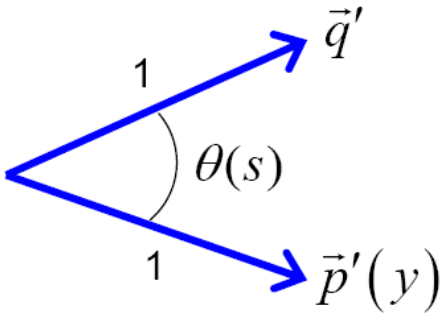
Target candidate: $\vec{p}(s) = (p_1(s), \ldots, p_m(s))$

Similarity function: $f(s) = f[\vec{p}(s), \vec{q}]$

Bhattacharyya Coefficient = divergence measure

$$\vec{q}' = \left(\sqrt{q_1}, \ldots, \sqrt{q_m}\right)$$

$$\vec{p}'(s) = \left(\sqrt{p_1(s)}, \ldots, \sqrt{p_m(s)}\right)$$




$$f(s) = \cos\theta(s) = \frac{p'(s)^T q'}{\|p'(s)\| \cdot \|q'\|} = \sum_{u=1}^{m} \sqrt{p_u(s) q_u}$$

In order to understand the Bhattacharyya distance, it is useful to represent it geometrically: in fact, by taking the vector of square roots of all entries (for q and p), we get two vectors q' and p', which form an angle in an N-dimensional space (where N is the number of histogram bins) θ.
Then, the B. distance is simply the cosine of this angle (which can never be more than 90° since both p' and q' have positive entries) or, equivalently, the scalar product of p' and q', divided by both vector norms.

Therefore, the B. distance between the two distributions is called a divergence measure: it is related to the "divergence" angle of the two distributions, in feature-space.

There are other well-known examples of divergence measures, such as the Kullback-Leibler divergence, which can be defined both in finite-dimensional space (such as histograms) as well as infinite-dimensional ones (such as generic, continuous-valued representation of probability distributions).

## Approximating the Similarity Function

$$f(s) = \sum_{u=1}^{m} \sqrt{p_u(s) q_u}$$

Previous location: $s_0$

Candidate location: $s$

1. Linearization of f around $s_0$: $f(s) \approx f(s_0) + f'(s_0) \cdot (s - s_0)$

$$\Rightarrow \quad f(s) \approx \sum_{u=1}^{m} \sqrt{p_u(s_0) q_u} + \sum_{u=1}^{m} \frac{1}{2} \sqrt{\frac{q_u}{p_u(s_0)}} p'_u(s_0) \cdot (s - s_0)$$

2. Linearization of $p_u$ around $s_0$: $p_u(s) \approx p_u(s_0) + p'_u(s_0)(s - s_0)$

$$\Rightarrow \quad f(s) \approx \sum_{u=1}^{m} \sqrt{p_u(s_0) q_u} + \sum_{u=1}^{m} \frac{1}{2} \sqrt{\frac{q_u}{p_u(s_0)}} \left( p_u(s) - p_u(s_0) \right)$$

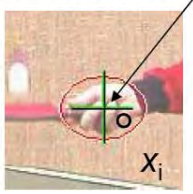
In order to optimize the Similarity function with mean-shift, we need to express it somehow in terms of a kernel-based representation, and the color histograms are not well-suited for this task.

First, we perform a linearization of f around $s_0$ (the previous state estimate), by using the first-order Taylor expansion. This leads to two terms: a constant one (depends only on $s_0$) and a term linear in s.

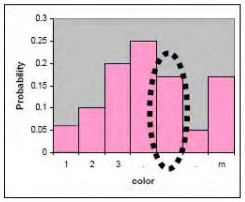
Next, we need to compute the derivative of $p_u$' in $s_0$ (inside the second term). This derivative can be obtained by a second linearization, of $p_u(s)$ around $s_0$. This allows computing it in terms of $p_u(s)$ and $p_u(s_0)$.

## Weighting pixel contributions with kernel

Candidate state (**s**)




$u=b(x_i)$




**Pixel contributions to $p_u$**

$$p_u(s) = \frac{1}{n} \sum_{i:b(x_i)=u} 1$$ **Standard**

$$p_u(s) = C_h \sum_{i:b(x_i)=u} k\left(\left\| \frac{s - x_i}{h} \right\|^2\right)$$ **Fuzzy**

Normalization factor

Pixel weight

$\{x_i\}_{i=1..n}$ Target pixel locations

$b(x)$ The color bin index (1..*m*) of pixel *x*

$k(x)$ A radially symmetric kernel

- Peripheral pixels are affected by occlusion and background interference

In order to arrive to the kernel-based representation of f(s), we make the histogram smooth in terms of s: that means, we express each bin contribution (from the respective pixels) in a fuzzy way, where the membership of pixel $x_i$ to the bin u (to which its color belongs) is weighted by its distance from the region center, s, through a kernel-shaped function (for example, a Gaussian with approximately the size of the region).

In this way, pixels near to the center contribute more to the histogram than pixels in the periphery, which is also plausible with the fact that central pixels have a higher probability of belonging to the object (and not to the background).

The result is that now $p_u(s)$ is a smooth function of s, and expressed by a sum of kernels.

Note that here k(x,y) is a kernel in *position* only, not in color space; in fact, the color information is included as the *bin* $b(x_i)$ to which the point $x_i$ contributes.

## Approximating the Similarity Function

Linear approx. (around $\mathbf{s}_0$)

$$f(s) \approx \underbrace{\frac{1}{2}\sum_{u=1}^{m}\sqrt{p_u(s_0)q_u}}_{\text{Independent of } \mathbf{s}} + \frac{1}{2}\sum_{u=1}^{m} p_u(s)\sqrt{\frac{q_u}{p_u(s_0)}}$$

$$p_u(s) = C_h \sum_{i:b(x_i)=u} k\left(\left\|\frac{s-x_i}{h}\right\|^2\right)$$

$$f(s) \approx \frac{C_h}{2}\sum_{i=1}^{n} w_i k\left(\left\|\frac{s-x_i}{h}\right\|^2\right) \qquad w_i = \sqrt{\frac{\hat{q}_{b(\mathbf{x}_i)}}{\hat{p}_{b(\mathbf{x}_i)}(s_0)}}$$

Weighted kernel density estimate (in **s**)

Now we can express f(s) in terms of the kernel k(), and we obtain a *weighted* kernel representation, where the weights $w_i$ contain the color information, collected inside the histograms p and q.

## Object tracking: re-weighted mean-shift

The mode of $\frac{C_h}{2}\sum_{i=1}^{n} w_i(s_0) k\left(\left\|\frac{s-x_i}{h}\right\|^2\right)$ = local maximum of *f*

**Original Mean-Shift:**

Find mode of $c\sum_{i=1}^{n} k\left(\left\|\frac{s-x_i}{h}\right\|^2\right)$ Iterate: $s_1 = \frac{\sum_{i=1}^{n} x_i g\left(\left\|\frac{s_0-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} g\left(\left\|\frac{s_0-x_i}{h}\right\|^2\right)}$

**Weighted Mean-Shift:**

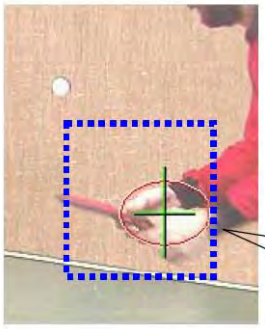
Find mode of $c\sum_{i=1}^{n} w_i k\left(\left\|\frac{y-x_i}{h}\right\|^2\right)$ Iterate: $s_1 = \frac{\sum_{i=1}^{n} x_i w_i g\left(\left\|\frac{s_0-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n} w_i g\left(\left\|\frac{s_0-x_i}{h}\right\|^2\right)}$

So, we can now apply a *weighted mean-shift* procedure in order to reach a local maximum of f (single mode), starting from $s_0$. The difference between the original and weighted mean-shift is given by the presence of $w_i$, in both numerator and denominator of the mean-shift vector.

## Maximizing the Similarity Function

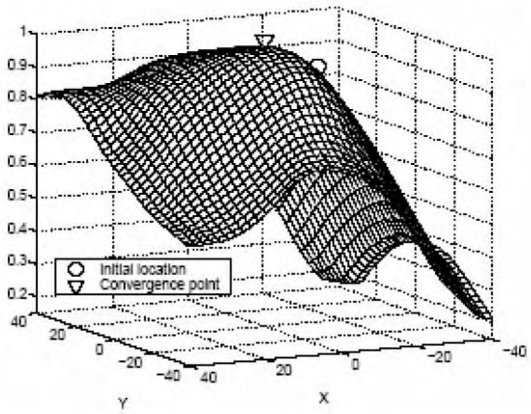
The mode of $\frac{C_h}{2}\sum_{i=1}^{n} w_i k\left(\left\|\frac{s-x_i}{h}\right\|^2\right)$ = local maximum of *f*

**Important Assumption:**

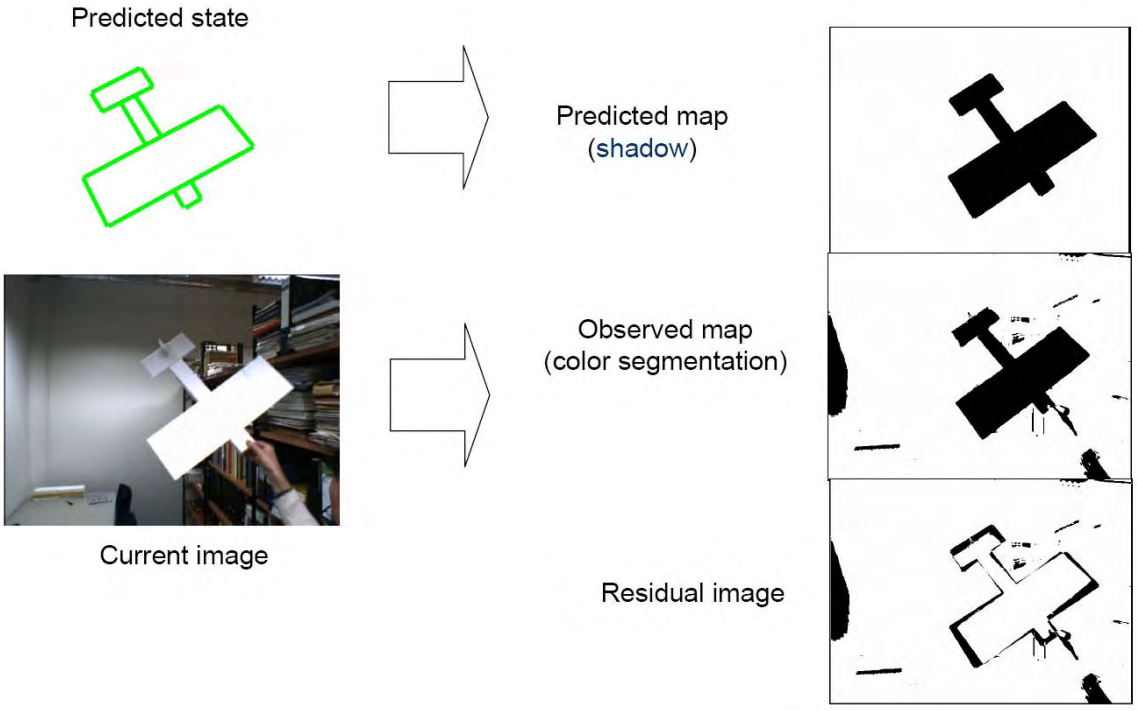


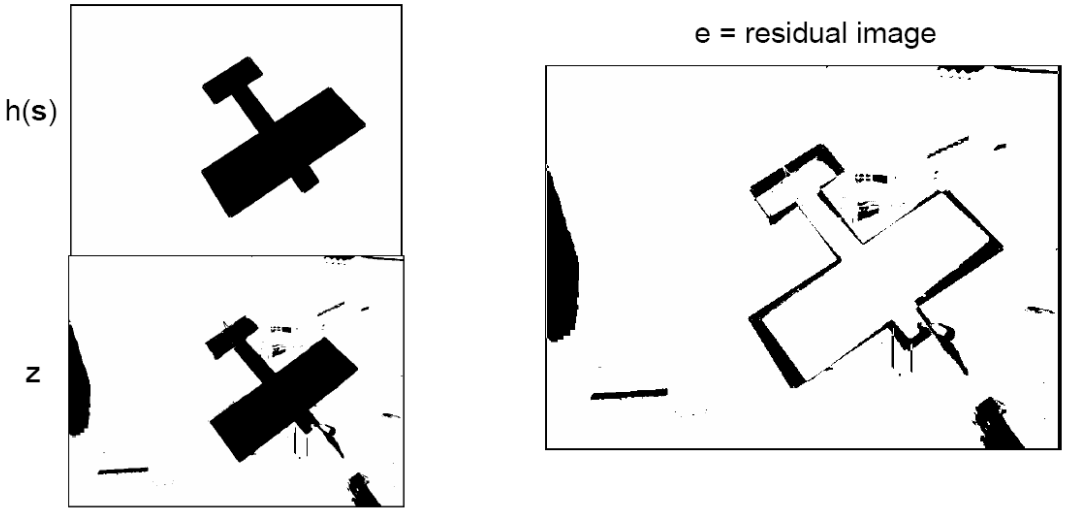
The model provides sufficient discrimination

One peak in the neighborhood



**Bayesian tracking for a color-based modality**




Pixel-wise binary difference = XOR

$$e_{x,y}^2(\mathbf{s}) = (z_{x,y} - h_{x,y}(\mathbf{s}))^2 = \mathrm{XOR}(z, h(\mathbf{s}))$$

Gaussian likelihood: $P(\mathbf{z} \mid \mathbf{s}) = C \cdot \exp\left( -\sum_{x,y} e_{x,y}^2(\mathbf{s}) \right)$

In a color-based visual modality, we can include dynamical models for tracking if we properly define the Bayesian filter and the measurement and dynamical models.

Concerning the measurement, by resuming our classification into three measurement levels for object tracking:

At pixel level, a color-based visual modality computes the predicted map h(s) at a given pose hypothesis (corresponding to an ideal, noise-free segmentation).

The actual measurement z is given by the color segmentation of the current image, which also contains noise and background clutter, as well as missing points in the object region.

The residual image is given by the X-OR of the two binary maps, collecting the differences between expectation and observation.

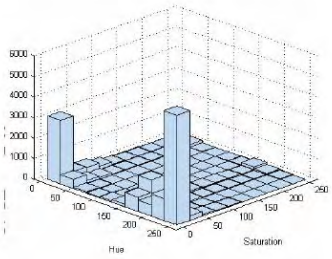
In this case, a particle filter can be used (since this is a highly non-linear and non-Gaussian process).
Since the expectation h(s) has to be computed many times (for each particle) a good solution is to implement this procedure on graphics hardware (GPU) where rendering the object silhouette can be performed very fast.

Photometric feature = Hue-Saturation color histogram

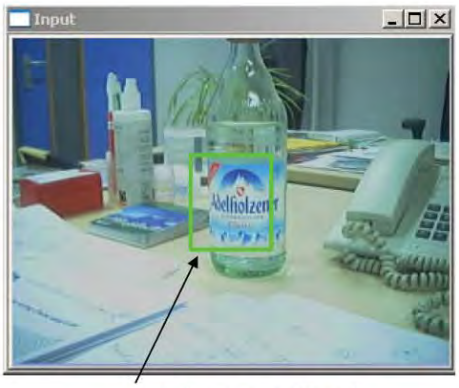
h = expected histogram (appearance model)
z = observed histogram (measurement)
P(z|s) = Likelihood (Gauss. in Bhattacharyya distance)

$$P(\mathbf{z} \mid \mathbf{s}) \propto \exp\left(-\frac{1}{2}\frac{B(\mathbf{s})^2}{\sigma^2}\right)$$ ← $\sigma^2$ = measurement noise variance

Appearance model




h (expectation)

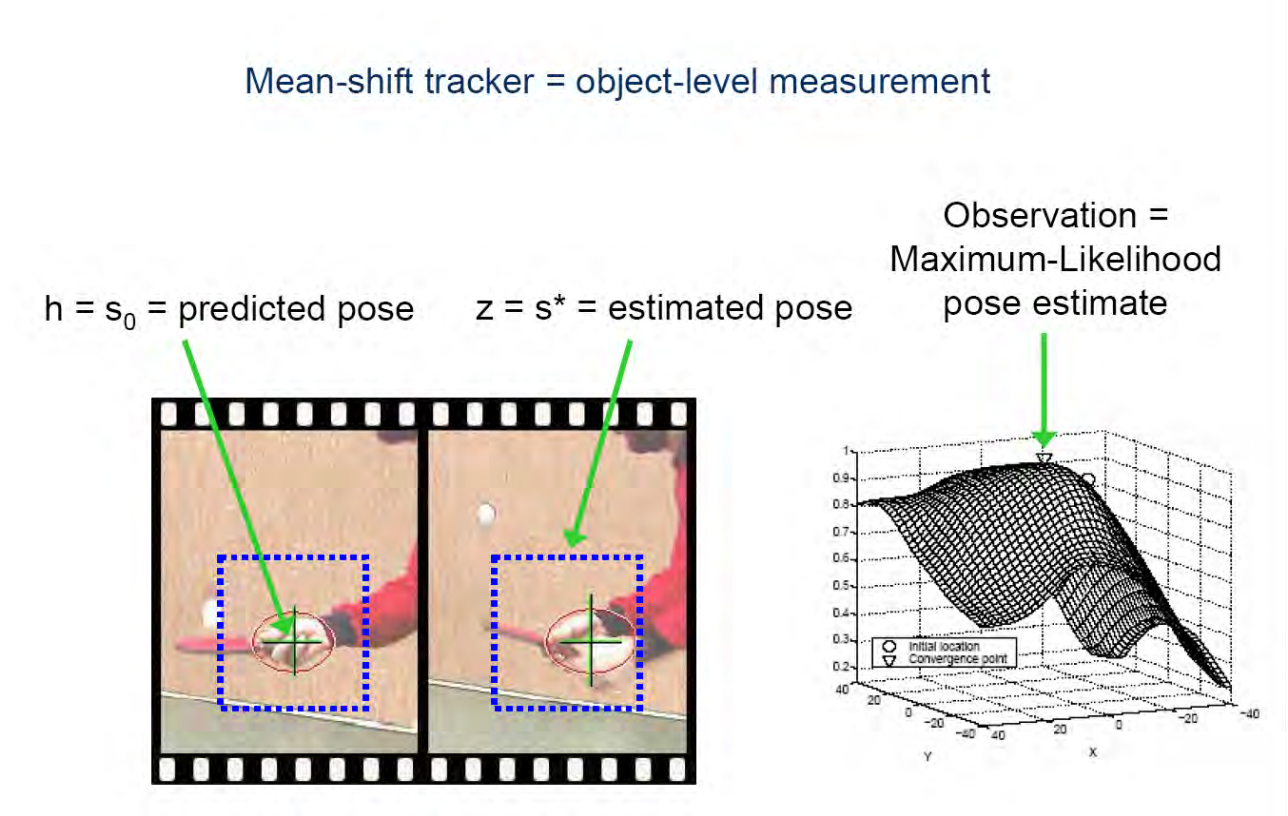



z (observation)

At feature-level, we consider the expected color distribution *h* (represented by the reference histogram), and match it with the observed color histogram *z* of the underlying pixels, in the predicted area at pose *s*.

In this case, the overall residual between *z* and *h* is given by the Bhattacharyya distance between the two histograms. In order to obtain a Likelihood value (as probability distribution), we can use a Gaussian-like distribution, where the B distance is used instead of the standard Mahalanobis distance, and with a suitable variance $\sigma^2$.
Note that in this case, we cannot speak of a Gaussian distribution (because of B), but rather of a more general "Gibbs-class" distribution, which has this exponential form.

Therefore, this likelihood cannot be used with a Kalman (or Extended Kalman) filter, so we can use a particle filter instead.




And finally, at object-level, after running mean-shift optimization, we get a pose-space estimate p*, which constitutes the measurement z, while the expected measurement h is the prior pose $p_0$ (predicted from the previous frame).

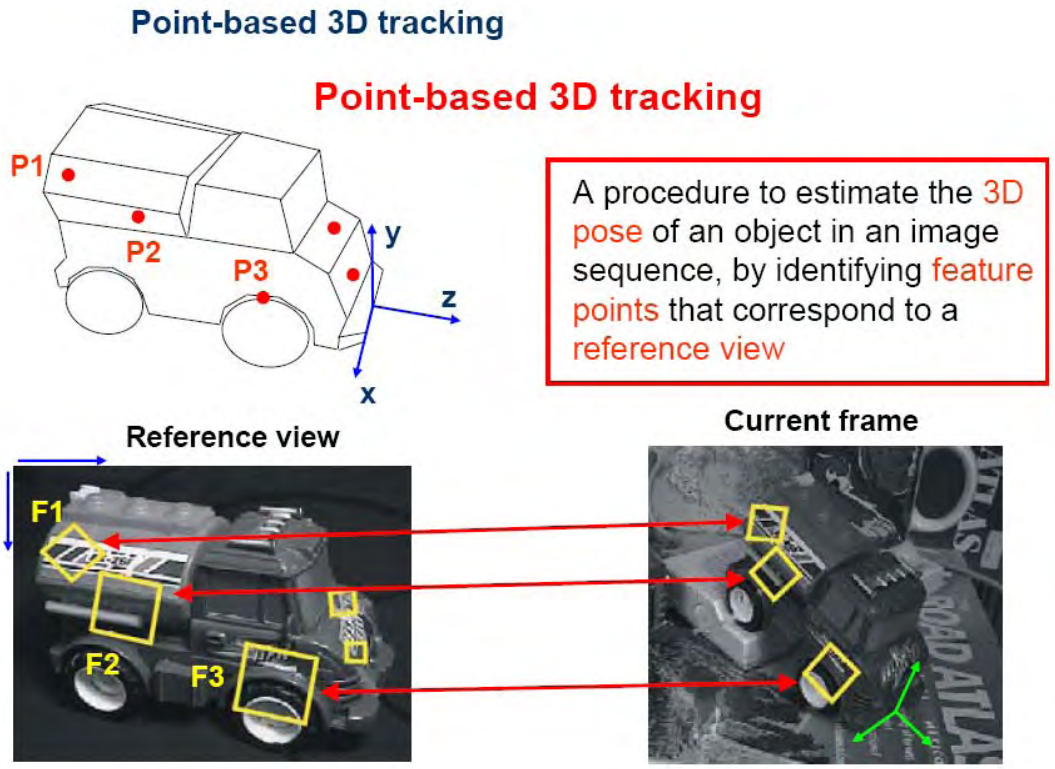
In this case, a standard Kalman filter can be used in order to perform Bayesian tracking.

# Resume: Color-based object tracking

- Color space definitions
- Modeling the color distribution of an image (or an object)
  - Histograms
  - Mixtures of Gaussians
  - Kernel densities
- Maximizing kernel densities: the mean-shift algorithm
  - Application 1: Kernel-based color segmentation
  - Application 2: Kernel-based object tracking
- The 3 measurement levels for color-based Bayesian tracking

## *Lecture 7 – The Kanade-Lucas-Tomasi Features Tracker*

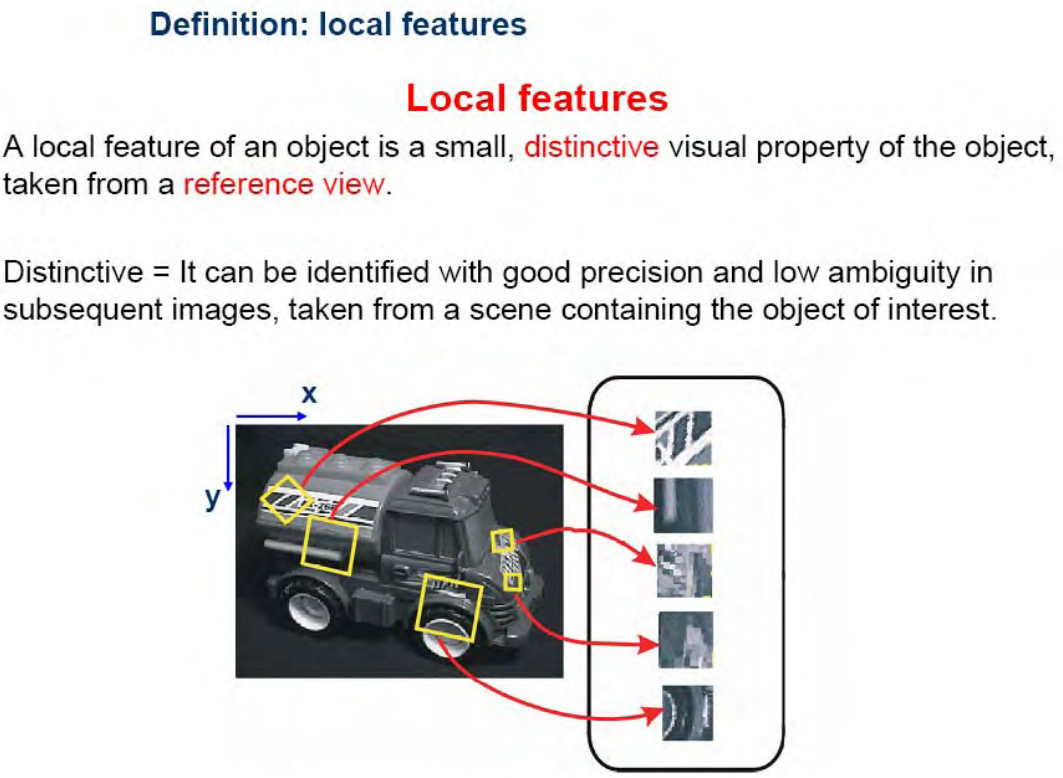
### Local keypoint-based tracking




Point-based visual tracking employs a set of feature points, that belong to the visible surface of the object to be tracked.

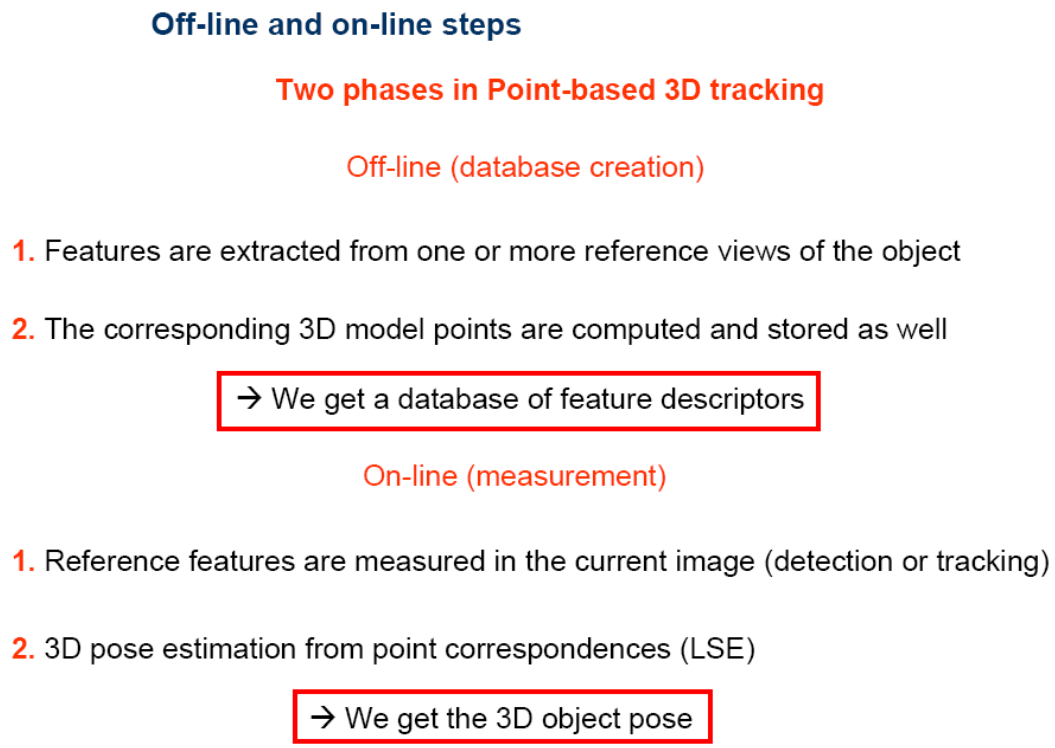
These points are uniquely identified from one or more reference views of the object.
When these points are detected, or tracked, into the current image, a procedure for 3D pose estimation can be performed, yielding an estimate of the state s*. This can be used as measurement variable (z) for a Bayesian tracking scheme, where the measurement is of a high-level (i.e. object-level) type.
NOTE: As we have seen, the measurement can also be defined as the set of feature points themselves z=(q1,…,qN) which is a middle-level (feature-level) type, and in this case the Likelihood model will be defined between expected and observed feature points. The choice between the two is absolutely free, and influenced only by application-specific arguments.

***Definition: local features***




A point feature is also called *local* feature, because they describe a specific visual property which is localized in a very small area of the object surface.
This property is of a general type: it can be a small distinctive pattern of colors/grey values, or a local configuration of edges (a corner), etc.

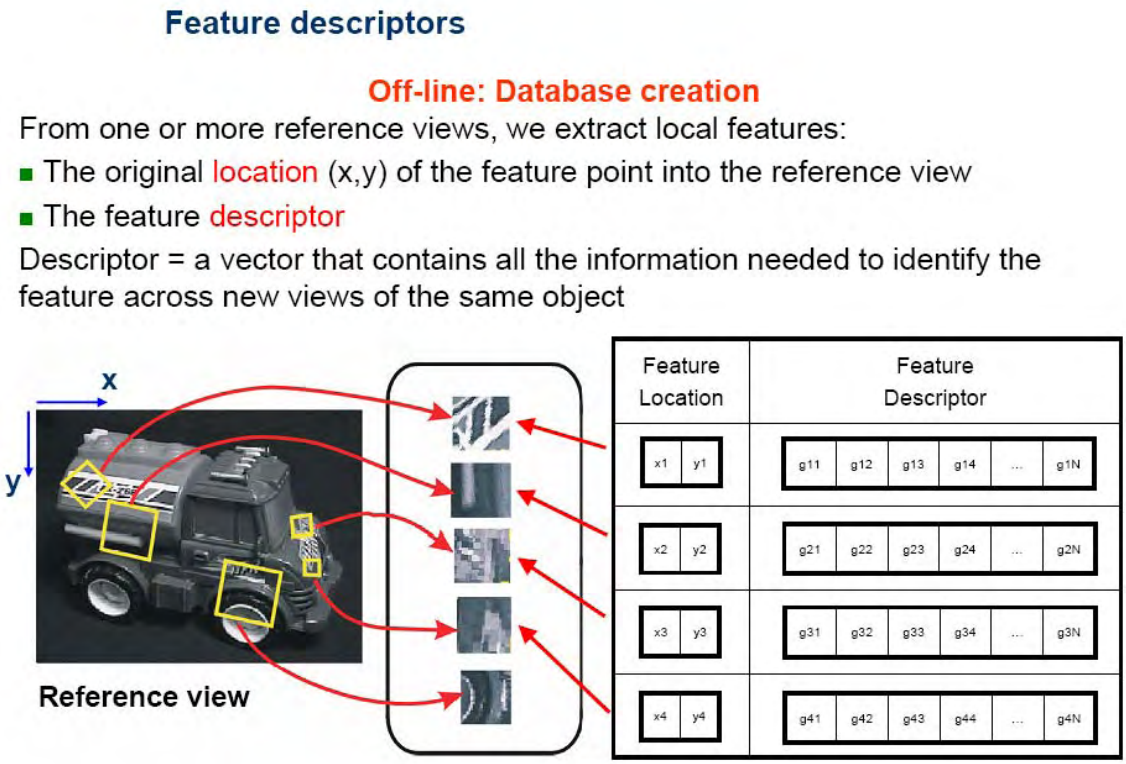
Two most important properties for a local features are: it should be a distinctive pattern (a pattern which is very unlikely to be found in other parts of the image), but at the same time it should be easy to be identified again from different views of the object.




In model-based tracking, we always have two main phases: modeling (off-line) and tracking (on-line).
For the case of point-based 3D tracking, the following tasks are involved.

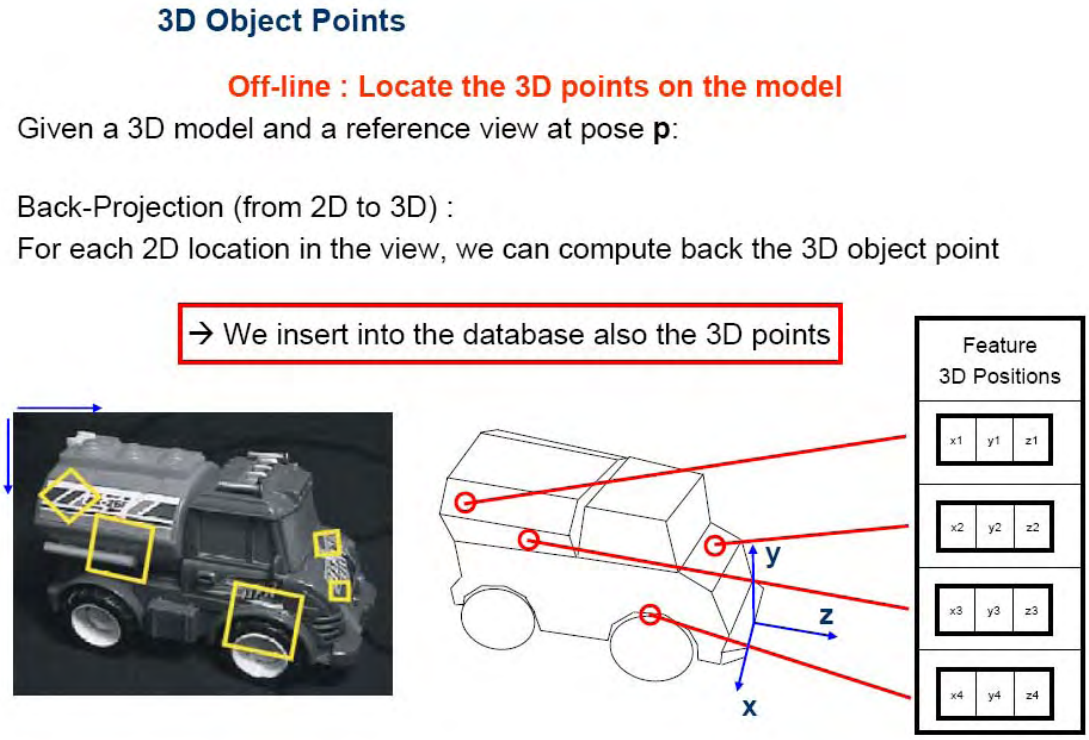
Off-line, we need to compute a database of relevant points from the object, by taking one or more reference views.
The database contains all the information needed to identify the local pattern in subsequent images: this is called the feature descriptor. Moreover, since each feature point corresponds to a unique 3D point from the object surface, we need to store this information as well, that is the (x,y,z) coordinates of the point, referred to the body frame.

On-line, we have two alternatives: features detection (frame-by-frame) or tracking (frame-to-frame).




***Keypoint descriptors database***

A descriptor is a vector containing all informations needed to identify the local feature i.e. the pattern. The local pattern can be described in very different ways, according to the degree of robustness that we need for matching and identifying it in different views.
The simplest example of descriptor is given by the grey-values of the window itself, stored in a long row. But this descriptor not always has the properties needed for a good tracking (invariance), as we will see next.




When features are collected from a reference view of the object, the local positions in the image (x,y) can be used in order to get back the corresponding 3D points on the object surface.

This procedure is called *back-projection*, from image to space, and it is possible in this case, since we know already the pose of the object in the view.

Back-projection can be done using the *depth-map* of the CAD model, rendered at the given pose: a depth map contains the depth (z) of each pixel of the rendered model at that pose, and these informations are sufficient to reconstruct the original coordinates of the point, in body frame.

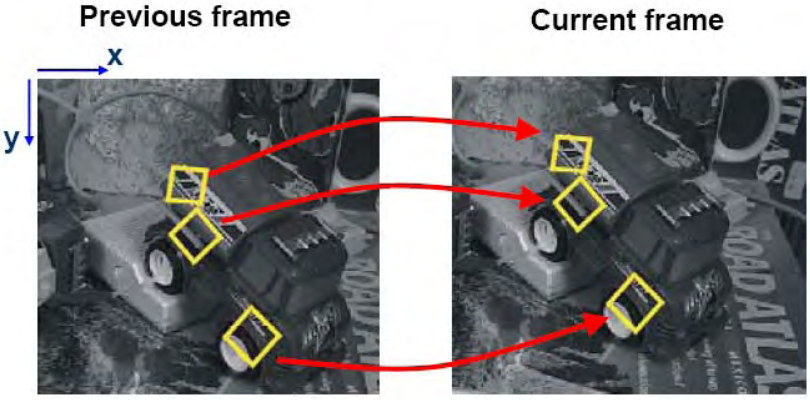
***On-line features detection vs. tracking***

## Features Tracking

### On-line Solution 1: Features Tracking

1. Time 0: The features are identified in the first frame of the sequence
2. Time t: They are followed (tracked) from frame t-1

→ We do not need particular invariance properties




## Feature descriptors for tracking

How can we represent (describe) a local feature?

First idea:

We can simply store the gray (or color) values of the N pixels from the reference window; example (N = 25x25 = 625)

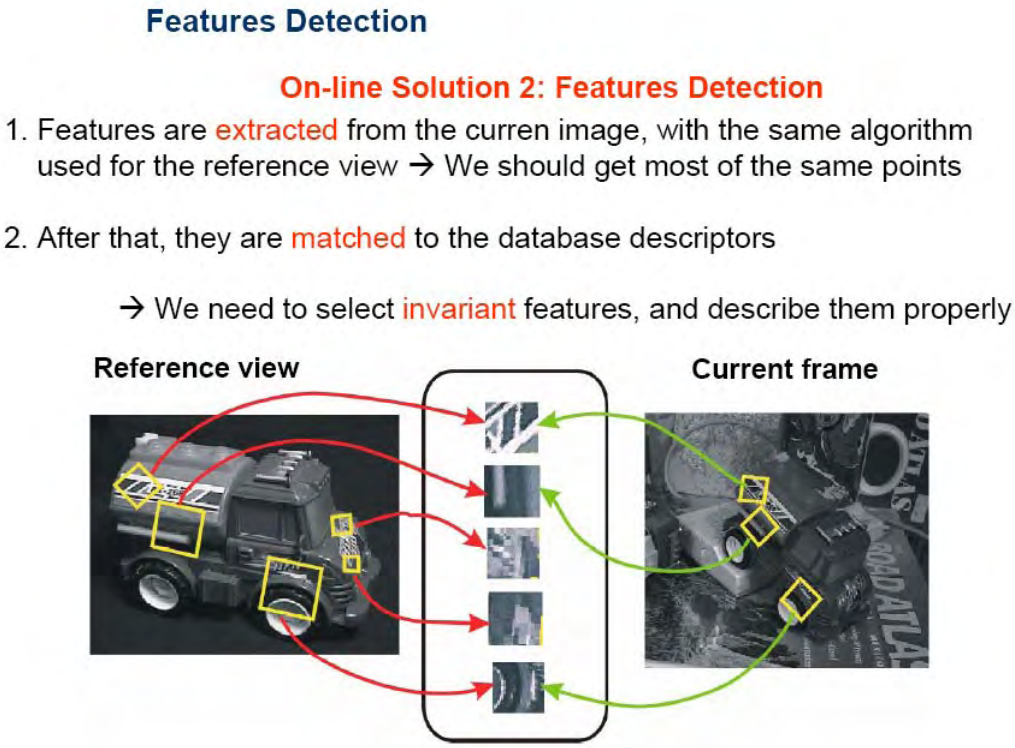

→ Not invariant, but good for tracking, because frame (t) is similar to frame (t-1)

A first solution to point-based tracking involves first tracking the local features themselves, from frame to frame.

This idea is easier to implement, faster, and does not require the selection of particular invariance properties for the feature points, which therefore can also be selected in a large number.

This is because from frame to frame, the object view changes little, as well as the light etc., so that one can expect a very similar grey pattern to be found near the previous location.

In this case, a simple grey-level descriptor is sufficient.

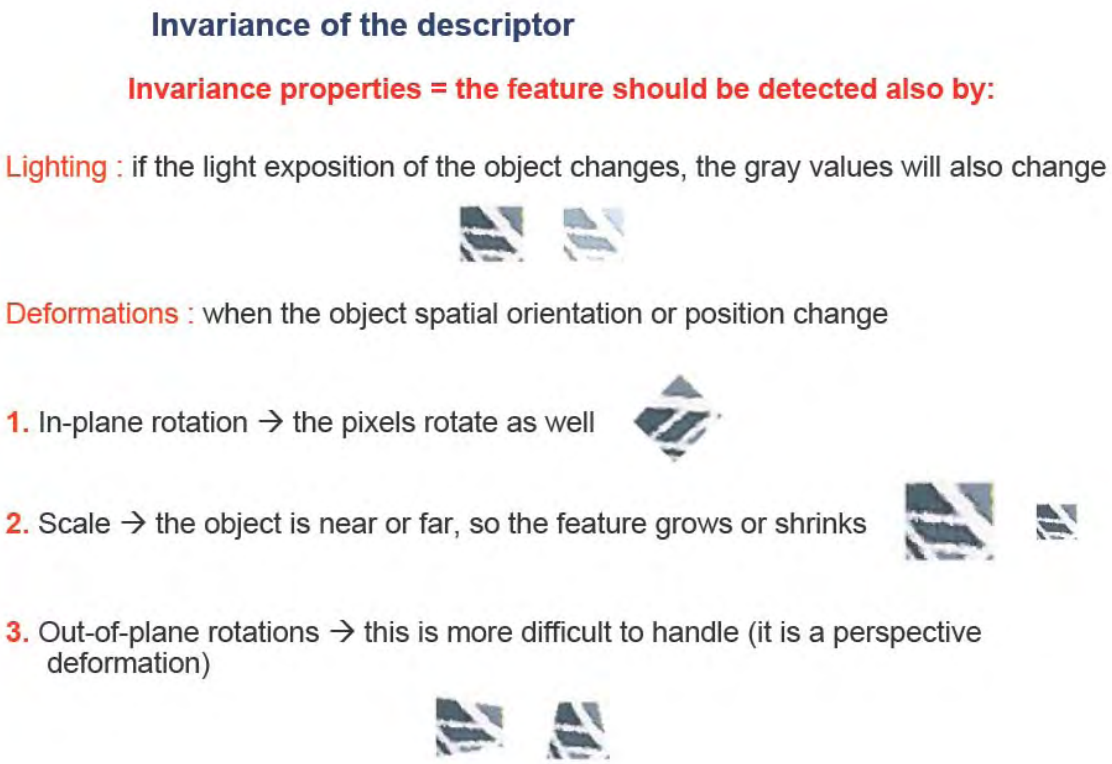



The other possibility is to perform a new detection of features in the whole image at every frame, that need afterwards to be matched to the original ones from the reference view.

In this case, since the pose, light, etc. can be a very different one with respect to the reference, we need more invariance properties.

From the model point of view, this has two main consequences: the first is that less feature points can be selected from the object (only points with “special” properties), the second is that we also need to use a different, invariant descriptor, (the grey-level window does not have these properties).

***Invariance properties for features detection***




The most important invariance properties that we need to consider are:

- invariance to light: not only the external environment light can change in time, but actually the incident direction of light onto the object changes with the pose, and therefore the local grey pattern

- invariance to geometric deformations of the pattern: when the space pose (roto-translation) is different, this reflects on the appearance of the image patch, which basically can rotate, scale, and deform according to the perspective transformation

Therefore, in order to perform frame-by-frame detection and matching of local features, our method should be robust with respect to these modifications, so that the feature point can be found and matched with the correct one into the reference database.

## Features detection vs. tracking

### Detection

- Advantages: it searches over the whole image (can be a very different view)
- Disadvantages: less stable (jitter), less features, complex algorithm (slow)

### Tracking

- Advantage: quite stable (the view changes little), more points, fast algorithm
- Disadvantage: it can loose the track (drift problem)

→ We can combine them together: **(Detection+Tracking)**

- At the beginning, match the features with the reference image (detection)
- After, track them from time to time, monitoring the quality of each feature
- If a feature gets lost (drift), it can be replaced by the detection algorithm

The reason to consider both approaches for on-line tracking is that they have complimentary advantages/disadvantages.

Detection can find the object also when the image is very different from the reference view, which happens for example in the initial frame of a video sequence, where no previous knowledge is still available.
A main disadvantage is that, because of the global search, it is a complex, slow and not very precise, even with the best methodologies today available (e.g. SIFT).
In particular, low precision results in a jittering pose estimation (not very stable and accurate).

Features tracking, instead, is stable, precise and fast, because based on local search algorithms (features are searched only in a neighborhood of the old ones).
The main disadvantage in this case is the drift problem: whenever a feature gets occluded from external objects, or disappears from the visible area, it will be mis-tracked and never recover again, and all the pose estimation result will be definitely lost.

Therefore, features tracking alone is not sufficient: it needs a *re-initialization* procedure, which can be supplied by the other methodology, that in this case acts as a "supervisor" of the tracking process.

In order to obtain this result, of course we also need a criterion to check at each time if a feature is getting lost, to be eventually replaced in order to keep the number of points to a sufficient level for 3D pose estimation.

**The KLT feature tracking algorithm**

## The KLT feature tracking algorithm

### Idea of KLT features tracker

The KLT algorithm does two things:

- Select features from the reference view that are good for tracking

- Given the initial feature locations, track them over subsequent images

What does it mean „good for tracking“?

It means that the features should be reliable for identification over the sequence, from the KLT algorithm itself.

Therefore, we need first to talk about the tracking method

The KLT tracker is a well-known algorithm for frame-to-frame tracking of local features.

This algorithm also provides the initial selection of feature points good for tracking.

The criterion for this selection is of course different from the one used by invariant detection methods like as SIFT. In particular, it is less restrictive since the points do not need particular invariance properties, but only to be distinctive, and reliable to be followed from frame to frame.

***Optical flow conditions***

**Optical Flow equation**

How does KLT tracking works?

KLT looks for the feature pattern (i.e. the gray-pixels window) into the current image, by computing the transformation that minimizes the error (LSE estimation) between the previous gray values and the current one.

Which transformation does KLT use?

The transformation can be of two types:

- Simple translation of the image patch ← 2 dof

- Affine transformation ← 6 dof

The tracking part of KLT solves a LSE problem for searching a small patch of grey-pixels of the current image over the subsequent one.

This task must be separately accomplished for every feature point, and for the purpose of speed it employs simple and linear models of transformation in the image plane:

- 2D Translation
- 6D Affine (linear+translation) deformation

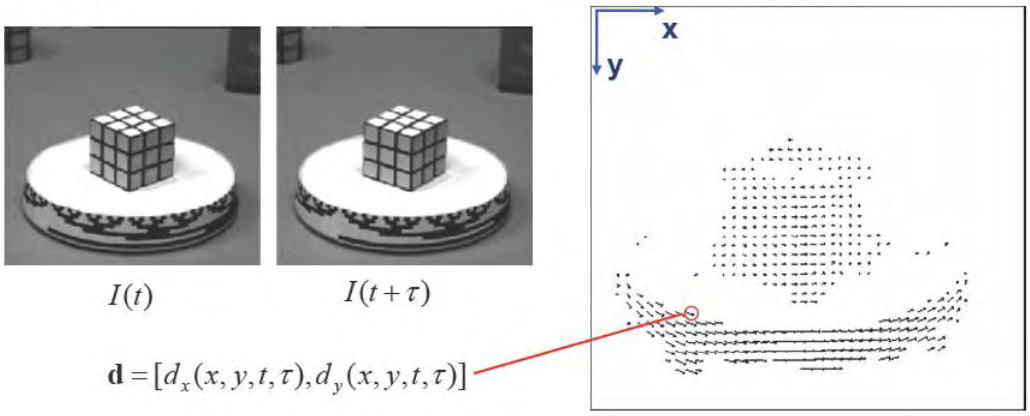
These models can be reliably employed for frame-to-frame tracking, because of the small difference between the two images.

## Optical Flow equation

Optical Flow is a general function that describes how the visible pixels move, for a small displacement of the viewpoint and/or the objects inside the scene.

$$I(x,y,t+\tau) = I(x-d_x(x,y,t,\tau), y-d_y(x,y,t,\tau), t)$$

The pixels of I move, from time t to time (t+τ), by a variable displacement $\mathbf{d}=(d_x,d_y)$.




$I(t)$ $I(t+\tau)$

$$\mathbf{d} = [d_x(x,y,t,\tau), d_y(x,y,t,\tau)]$$

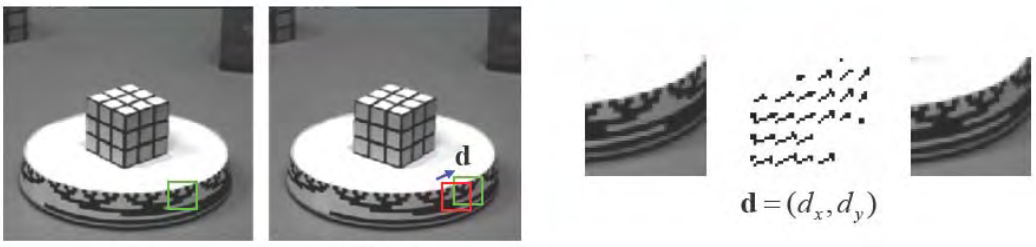
## Optical flow for features tracking

How can we use this fact for features tracking?

- If the window is small → The pixel displacements are ~ parallel (pure translation)
- If τ is small → The new pose is close to the previous one

→ We use the approximated Optical Flow equation: **d** is ~constant

$$I(x,y,t+\tau) = I(x-d_x(t,\tau), y-d_y(t,\tau), t)$$




$I(t)$ $I(t+\tau)$

$$\mathbf{d} = (d_x, d_y)$$

The Optical flow function describes corresponding pixels motion between two images of the same scenario, in the most general case:

$$I(x,y,t+\tau) = I(x-d_x(x,y,t,\tau), y-d_y(x,y,t,\tau), t)$$

The two terms dx, dy express pixel displacements, and they are function of:

- the original pixel location x,y
- the previous and current time in the sequence

This is because:

- different pixels can be the image of different real-world parts, eventually moving with different velocities
- even when two pixels belong to the same, rigid object, their image motion can be different because located at different depths, etc.
- moreover, since motion can be complex (roto-translation), over a long time difference, the displacement between pixels cannot be approximated with a simple translation

If we impose two conditions:

- consider a small window of pixels
- consider a small time difference $\tau$

then we can assume a *parallel* translation for each pixel inside the window; in this case, dx and dy are constant over the window (pure translation of the feature window).

### *Solution for the translational model*

#### Pure translation model

We have the two images I(t) and I(t+$\tau$), and we have to solve for $d_x$, $d_y$

$$I(x, y, t+\tau) = I(x-d_x, y-d_y, t)$$

We can write this equation as an LSE problem:

$$\mathbf{d}^* = \arg\min_{\mathbf{d}} \sum_{x,y} \left\| I(x, y, t+\tau) - I(x-d_x, y-d_y, t) \right\|^2$$

Where the sum is over the feature window (x,y). We can also write:

$$\arg\min_{\mathbf{d}} \sum_{\mathbf{q}_i} \left\| J(\mathbf{q}_i) - I(\mathbf{q}_i - \mathbf{d}) \right\|^2; \quad \mathbf{q}_i = (x, y); J = I(t+\tau)$$

This is a non-linear LSE problem (the image values I(q-d) are generic functions of **d**!).

If we put all image values J($\mathbf{q}_i$) and I($\mathbf{q}_i$-**d**) on two vectors **J**, **I**(**d**), we can write

$$\mathbf{d}^* = \arg\min_{\mathbf{d}} \left\| J - I(\mathbf{d}) \right\|^2$$

When the approximation holds, we can track a feature from frame to frame, by solving a LSE problem in (dx,dy)

$$\mathbf{d}^* = \arg\min_{\mathbf{d}} \sum_{x,y} \left\| I(x, y, t+\tau) - I(x-d_x, y-d_y, t) \right\|^2$$

NOTE: although the transformation between image positions is linear, the overall cost function is nonlinear, because the image values (grey-level) are nonlinear and unknown functions of pixel positions!

## KLT tracking equation

KLT tracker : Solves the NLSE problem with the Gauss-Newton algorithm:

**Gauss-Newton optimization (KLT)**

k=0: start from **d**=(0,0)

Repeat (k): given $\mathbf{d}_k$

1. Solve for the increment $\Delta\mathbf{d}$ with the formula $(A^T A)\Delta\mathbf{d} = A^T(J - I)$
2. Set the new parameters $\mathbf{d}_{k+1} = \mathbf{d}_k + \Delta\mathbf{d}$

The Jacobian matrix (Mx2) is: $$A = \begin{bmatrix} \frac{\partial I(q_1 - d_x)}{\partial d_x} & \frac{\partial I(q_1 - d_y)}{\partial d_y} \\ \cdots & \cdots \\ \frac{\partial I(q_M - d_x)}{\partial d_x} & \frac{\partial I(q_M - d_y)}{\partial d_y} \end{bmatrix} = \begin{bmatrix} I_x(q_1) & I_y(q_1) \\ \cdots & \cdots \\ I_x(q_M) & I_y(q_M) \end{bmatrix}$$

For this problem, KLT uses the standard Gauss-Newton algorithm. In this case, the Jacobian matrix A is also constant (not dependent on dx, dy); therefore, it can be computed just once, at the beginning of the optimization.

## Auto-correlation matrix

What exactly does the KLT tracking algorithm?

Each Gauss-Newton step solves a linear equation

$$(A^T A)\Delta\mathbf{d} = A^T(J - I(\mathbf{d})) \Rightarrow Z \cdot \Delta\mathbf{d} = \mathbf{z}(\mathbf{d})$$

What is Z?

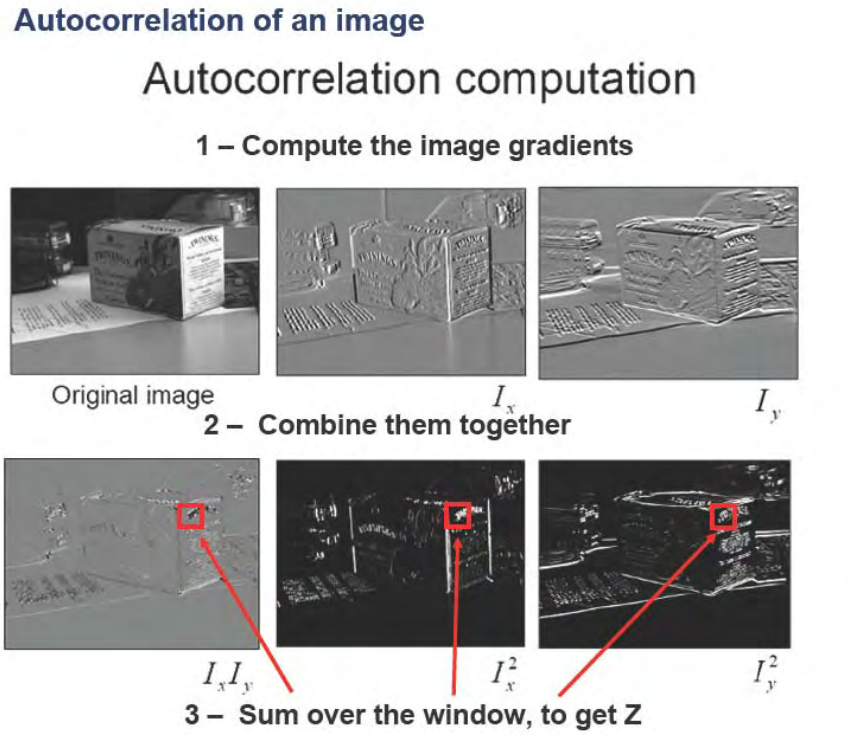
$$Z = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \quad \mathbf{z}(\mathbf{d}) = \begin{bmatrix} -\sum I_x (J - I(\mathbf{d})) \\ -\sum I_y (J - I(\mathbf{d})) \end{bmatrix}$$

Z is a *constant matrix* related to the feature window gradients $I_x$, $I_y$ (not dependent on **d**).

Z is called Auto-correlation matrix : the „correlation of the feature with itself“

If we write the equation for each Gauss-Newton step (linearized LSE), we find a linear (2x2) system, with a constant coefficient matrix Z.

This matrix is also called auto-correlation matrix, and it is related to the gradients inside the feature window; it expresses the correlation of the window with itself (see also next Lecture), and it can be used also in order to select good features to track.




**Gaussian-weighted feature window**

To have a more plausible (robust) solution:
We give more importance to the central pixels of the window, and less to the peripheral pixels

→ We add weights, with a 2D Gaussian centered in the window

$$Z = \sum_{(x,y)} Gauss(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}; \qquad \mathbf{z} = -\sum_{(x,y)} Gauss(x,y) \begin{bmatrix} I_x (J-I) \\ I_y (J-I) \end{bmatrix}$$

**Property of Z: it can be used to select feature points!**

In order to improve robustness in features tracking, it is generally better to give different weights to the pixels, where central pixels are more important than peripheral ones; this is obtained through a 2d Gaussian function with proper variance.

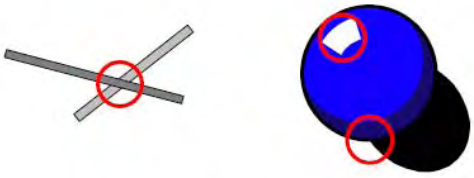
***KLT: good features selection***

## False features

### Problem of false features

Some corner-like patches from the reference image do not represent real points on the object!

Examples: Depth and lighting illusions



- Depth: If the point in the image comes from the projection of two lines at different depths → It is no real object point
- Lighting: If there is a light spot or a shadow → It looks like a corner

## KLT selection criterion

### Solution 2: KLT Features Selection

KLT selection criterion: select the best features for the KLT tracking algorithm
→The Z matrix must be good for solving the KLT equation:

$$Z^{*}\Delta\mathbf{d} = \mathbf{z}$$

When is this equation good to solve?

- **Z** should be invertible
- **Z** should not be too small due to noise

→ eigenvalues $\lambda_1$ and $\lambda_2$ of **Z** should not be too small

- **Z** should be well-conditioned

→ $\lambda_1 / \lambda_2$ should not be too large ($\lambda_1$ = larger eigenvalue)

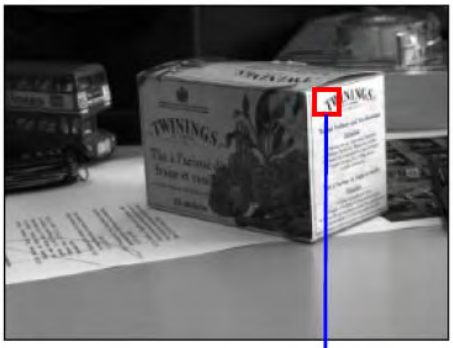
**→ KLT criterion: Good feature to track ⇔ $\min(\lambda_1,\lambda_2) > \text{threshold}$**

Another criterion to select features for tracking is the KLT criterion.
This is again based on the Z matrix, but this time considers a good feature one that can be reliably found in the next image, that is, when the Gauss-Newton equation is good to be solved (well-conditioned system).

This means that both the eigenvalues of Z should be high, and not too different from each other.
High eigenvalues → the feature point shows a high variability along all directions.
Not too different eigenvalues → the intensity variability is similar along all direction ("corner-like").

## KLT selection criterion

By looking at the eigenvalues of Z ($\lambda_1 > \lambda_2$) we can select features to track



$\lambda_1$

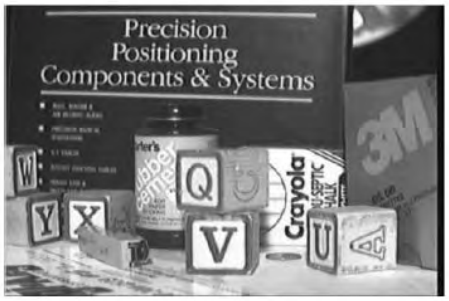
$\lambda_2$

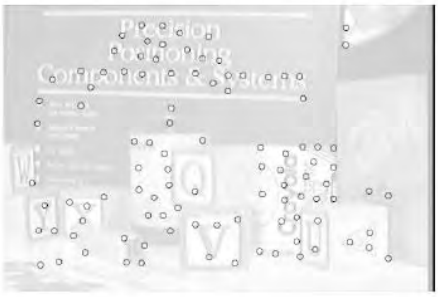
## Select features for KLT

### Good features to track

We select good features to track by searching over all the windows (e.g. 25x25) in the reference image $I_0$ :

- Compute $Z_0 = \sum_{(x,y)} Gauss(x,y) \begin{bmatrix} I_{0x}^2 & I_{0x}I_{0y} \\ I_{0x}I_{0y} & I_{0y}^2 \end{bmatrix}; \quad (\lambda_1, \lambda_2) = eig(Z_0)$
- Select the feature window if $\min(\lambda_1,\lambda_2)$>treshold
- Remove feature windows too near each other (minimum distance)



After looking for windows with this property, KLT also removes features that are too near each other, by setting a minimum distance proportional to the window size.

Many image patches that look like corner points are in fact no real features: they do not belong to any existing object point in space.

This is the problem of false features, and typically happens in case of depth illusions (when two object at different depth appear intersecting on the image) and in case of shadows or light spots.

All feature tracking methods also have drift problems: a feature track can get lost, and never recovered again.

This happens for two reasons:

- It was a false feature, therefore when the scene changes, it does not exists anymore
- The motion was too fast, so that in the next frame the point was too far for the region of convergence of Gauss-Newton optimization
- The feature is not visible anymore, either because has been occluded, or covered by light/shadows, or because it is not visible anymore from the new camera point of view.

We need then an automatic way for detecting this event, and eventually terminate the track for this feature.

***KLT: on-line quality check***

**KLT: Check features quality in time**

**Solution to drift: check quality and remove bad features**

The feature to track (descriptor) comes from the reference image $I_0$
→ We can use the descriptor itself to monitor the quality of tracking!

At time t, we have an estimation of the feature position F in the image I(t)
After estimating the position, we could just check the residual SSD error:

$$e = \sum_{\mathbf{q}_i} \left\| I(\mathbf{q}_i, t) - I(\mathbf{q}_i - \mathbf{d}, 0) \right\|^2$$

But this is not good, because in a long time difference $\tau$, the simple translation model **d** is not valid (even for a small window F)!

Then, for large displacement, we need a more precise model for the Optical Flow of the window F: the affine (6dof) deformation model is better

In order to check the quality of tracked features, we can use the original feature window (descriptor) from the reference image: if we track the same feature point, its appearance should be similar to the original one, apart from eventual light changes; on the contrary, if the feature has drifted or occluded, then we are tracking a very different patch.

This can be done by computing the SSD error between grey values of the two windows, which is the same cost function used for the KLT tracking algorithm (Gauss-Newton optimization).

But, in order to do this comparison, we must transform the pixel coordinates from the reference to the current position of the feature.

And for this purpose, a simple (x,y) translation model is not sufficient anymore, since the current viewpoint can be very different from the reference one (remember that the optical flow approximation is good only for small viewpoint differences, that is, between consecutive frames).

### *Solution for the 6dof affine model*

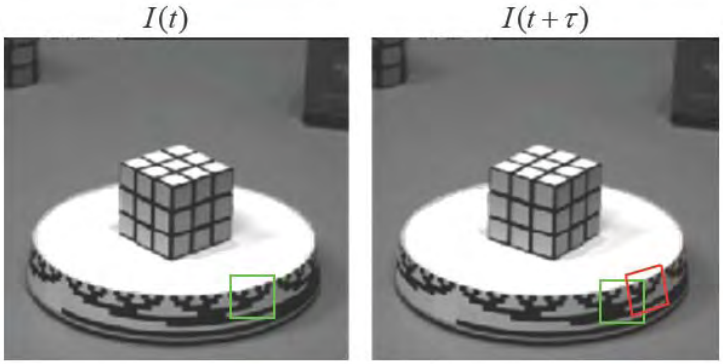
**Affine (6dof) deformation model**

A better approximation: for large $\tau$, we use the affine model

Affine = Linear+translation

$$I(q, t+\tau) = I(Dq - \mathbf{d}, t)$$

$D$ = (2x2) linear deformation matrix, **d** = translation vector




→ It has (4+2=6) dof, and can model also deformations of the feature window

The affine (6dof) deformation model is more general than the pure translation, and appropriate for this purpose.

Here, we have a linear+constant transformation, where *d* is the translation vector, while the (2x2) matrix D is the linear deformation matrix.
This kind of transformation is still simple enough for an LSE optimization (Gauss-Newton), and allows a rectangular window to be rotated, stretched and skewed (but always keeping parallelism between opposite sides).

## KLT: Solve for affine deformations

The LSE problem becomes:

$$(D^*, \mathbf{d}^*) = \arg\min_{(D,\mathbf{d})} \sum_{\mathbf{q}} \left\| I(\mathbf{q},t) - I(D\mathbf{q} - d, 0) \right\|^2$$

→ We write **g** = [$D_{11}$,$D_{12}$,$D_{21}$,$D_{22}$,$d_x$,$d_y$], and we apply Gauss-Newton

Gauss-Newton optimization (affine model)

k=0: start from **g** = [1,0,0,1,$d_x$,$d_y$] ← use the translation ($d_x$,$d_y$) from the KLT tracker

Repeat (k): given $\mathbf{g}_k$

1. Solve for the increment $\Delta\mathbf{g}$ with the formula $(A^T A)\Delta\mathbf{g} = A^T (J - I)$
2. Update affine parameters $\mathbf{g}_{k+1} = \mathbf{g}_k + \Delta\mathbf{g}$

Here the Jacobian matrix is more complex (Mx6): $A = \left[ \frac{\partial I(\mathbf{g},0)}{\partial g_j} \right] = \cdots$

## Solution to the affine problem

How is the Gauss-Newton step for the affine case?

In tracking (pure translation **d**) we had the Jacobian matrix A (Mx2)

$$(A^T A)\Delta\mathbf{d} = A^T (J - I)$$
$$Z\Delta\mathbf{d} = \mathbf{z}$$

Now we have A (Mx6)

$$(A^T A)\Delta\mathbf{g} = A^T (J - I)$$
$$W\Delta\mathbf{g} = \mathbf{w}$$

Where

$$W = \begin{bmatrix} U & V \\ V^T & Z \end{bmatrix}, \mathbf{w} = \begin{bmatrix} \mathbf{u} \\ \mathbf{z} \end{bmatrix}$$

U is (4x4), V (4x2), and **u** is (4x1) vector (a bit more complex)
Z and **z** are the same as before

In this context, in order to warp the original (reference) feature window onto the current one, we use Gauss-Newton, starting from the estimated translation $d_0$ (from the KLT tracker), and estimating all the 6 dof (D*,d*) (where d* of course can be slightly different from $d_0$).

A Gauss-Newton loop with this model needs to compute the related (Mx6) constant Jacobian matrix A, and the linearized LSE system at each step is obtained with the (6x6) matrix W, which is also constant throughout the optimization, and the 6-vector w.
NOTE: A (and W) are constant throughout the tracking sequence, because they are referred to the reference image; therefore, they can be computed off-line, before tracking. Instead, the Z matrix of KLT is computed once per frame, and it is constant only inside the Gauss-Newton loop. The right-hand sides (in both cases) are instead computed at every GN iteration, and every frame.

After the optimization, we have the possibility of computing the residual SSD error between the reference feature window and the warped one from the new image. If this error is above a threshold, we can say that the feature has been lost (drift, etc.), and we can discard it from the set.

***KLT: the full algorithm***

**KLT : the full algorithm**

Time 0

Select: Take the reference image I(0), and select good features to track:
- Try all possible windows of size (25x25) and compute the Z matrix
- If the eigenvalues of Z satisfy $\min(\lambda_1,\lambda_2)>$threshold, take it as a candidate feature window
- Remove feature windows too near each other (minimum distance)

Time t

Track: For each new frame I(t), given the previous features $F_i(t-1)$:
- Solve the translation problem $\mathbf{d}_i$ from the previous image I(t-1) to I(t)
- Update the feature windows $F_i(t) \rightarrow F_i(t-1)+\mathbf{d}_i$

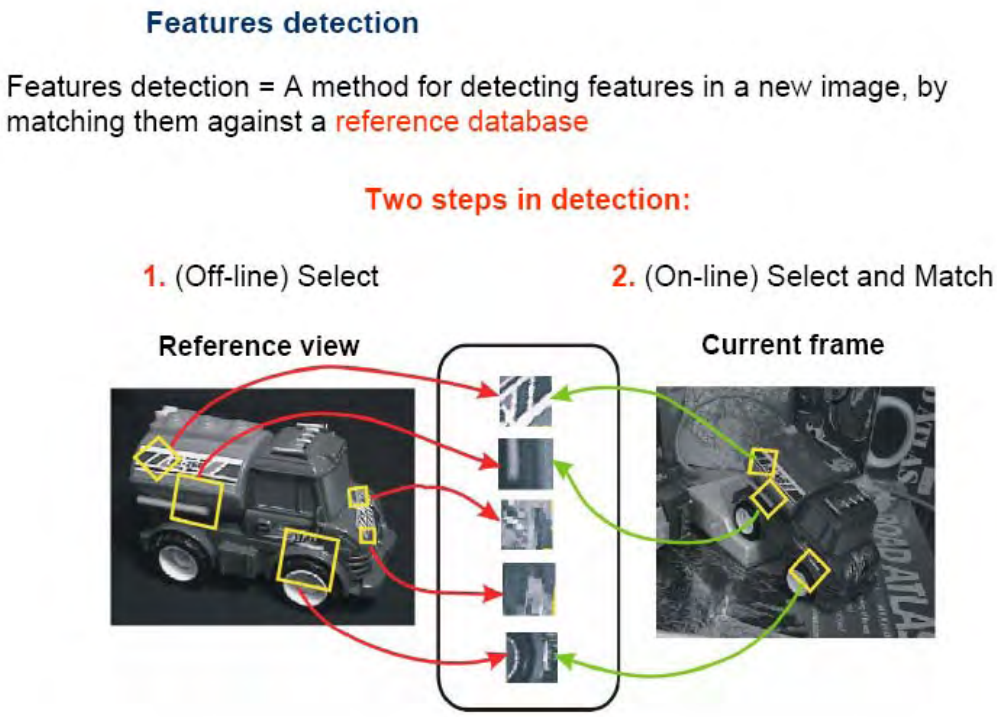
Check: For each feature, check if the quality is still good:
- Solve the affine estimation from the reference image I(0) to I(t) starting from the estimated translation $\mathbf{d}_i$, and find $(D_i,\mathbf{d}_i)^*$
- Compute the residual error: $e_i = \sum_{\mathbf{q}} \left\| I^{(t)}(\mathbf{q}) - I^{(0)}(D_i^*\mathbf{q} - \mathbf{d}_i^*) \right\|^2$
- If the error is too big, remove the feature

A sketch of the complete KLT algorithm: Selection (first frame), tracking (frame to frame) and check quality (after tracking). If too many features get removed from the last step, we also need to replace them with new ones, by selection of new features from the visible surface of the object.

Since now the view is very different from the first (or from the reference) frame, this replacement can be better done by using an invariant feature detection algorithm, e.g. SIFT.

## *Lecture 8 – Feature Detection Methods: the SIFT Approach*

### Features detection




For features detection, we use a database of reference points, that we try to match with the features extracted from the current image.

Therefore, now the correspondence problem must also be solved: which feature points in the new image correspond to the reference points?

This is different from feature tracking, where each feature was individually tracked across subsequent frames, and therefore there was no problem of identification (matching).

## Features detection

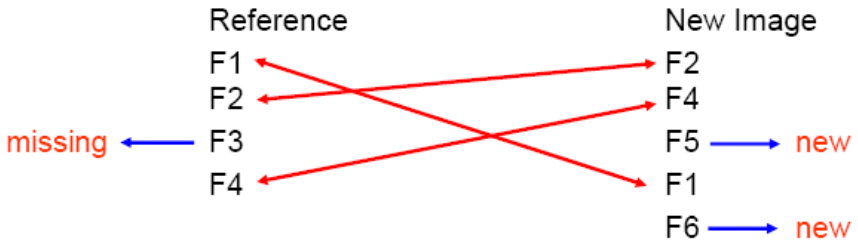
### Detection: Two main steps

1. Extract features in all images using the same algorithm of the reference

→ Hopefully, we get most of the same points

But usually we get also new points, and some of the old ones may be missing

2. Match the new features to the old ones (reference database):




Features detection proceeds in two steps: selecting new features, and matching them to the database.

The key idea is the following: we use an algorithm for extracting feature points which should be invariant to light or viewpoint change; therefore, we expect to select from the new image almost the same set of points that we selected in the reference frame.

Of course, since the new image is rather different (with more objects, etc.) we also expect some of the new detected points are not present into the old database, and also that some of the database points are not missing from the new image detection.

But, as long as the detection algorithm has good invariance properties, we expect to have still many good matchings.

## Invariance properties

### Invariance

#### Desired Invariance properties

For detection, we need invariance: the features should be correctly identified also in the presence of light and object pose changes

We need these properties, because the current image comes from a generally different pose/lighting than the original one.

Most important invariance properties for local features are: invariance to light, to in-plane rotations, to scale, and (to some extent) to out-of-plane rotations. The latter are the most difficult to achieve, since out-of-plane rotations of the object will result in perspective deformations of the feature gray-scale pattern.

### Obtaining invariance

#### The two fundamental issues for invariance

**1.** Selection: search for features with good invariance properties

Invariance properties are more difficult to find (the auto-correlation matrix $Z$ is not enough).

→ Invariant features selection

**2.** Description: describe the features in such a way the descriptor does not change with rotations etc.

The gray-value window itself is not invariant.

→ Transform it into an invariant descriptor

The invariance requirement translates into two main requirements for the detection method:

- The selection algorithm (which extracts “good” features for detection) should give similar results in presence of the above mentioned effects (light, etc.). In other words, it should detect almost the same set of points, apart from a few missing ones, or new detections.
Since these features are always found in a new image, even in different view situations, they are called *invariant features*.

- Once that invariant features have been detected, they should also be described in such a way that we can *match* (i.e. compare) them correctly → The grey-pattern itself (a MxM matrix of grey pixels) is not anymore a good descriptor for the local feature, and we need to transform it into an *invariant descriptor*.

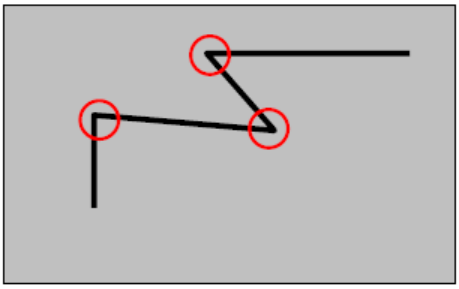
The second requirement (invariance of the description) means that we must encode the grey pattern into a more abstract vector (= *descriptor*) that remains almost the same also when the pattern undergoes a change in scale, intensity, etc.

### The Harris-Stephens feature detector

**Harris-Stephens detector**

**First idea: use The Harris corner detector**

**(Harris, 1988)**

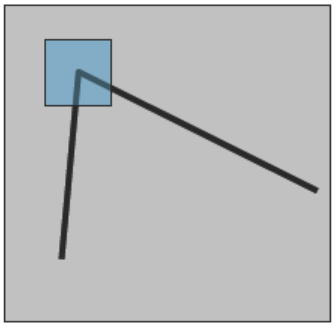


## Harris Detector: Basic Idea

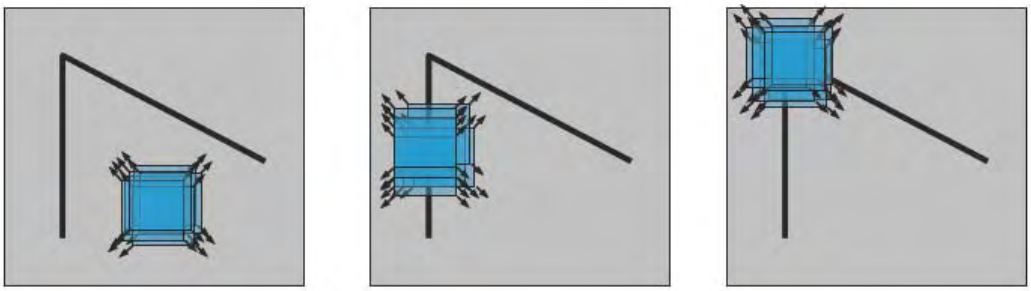
### Corner-like features

Idea: we should get points recognizable by looking through a small window

Shifting a window in *any direction* should give *a large change* in intensity



## Harris Detector: Basic Idea



"flat" region: no change in all directions

"edge": no change along the edge direction

"corner": significant change in all directions

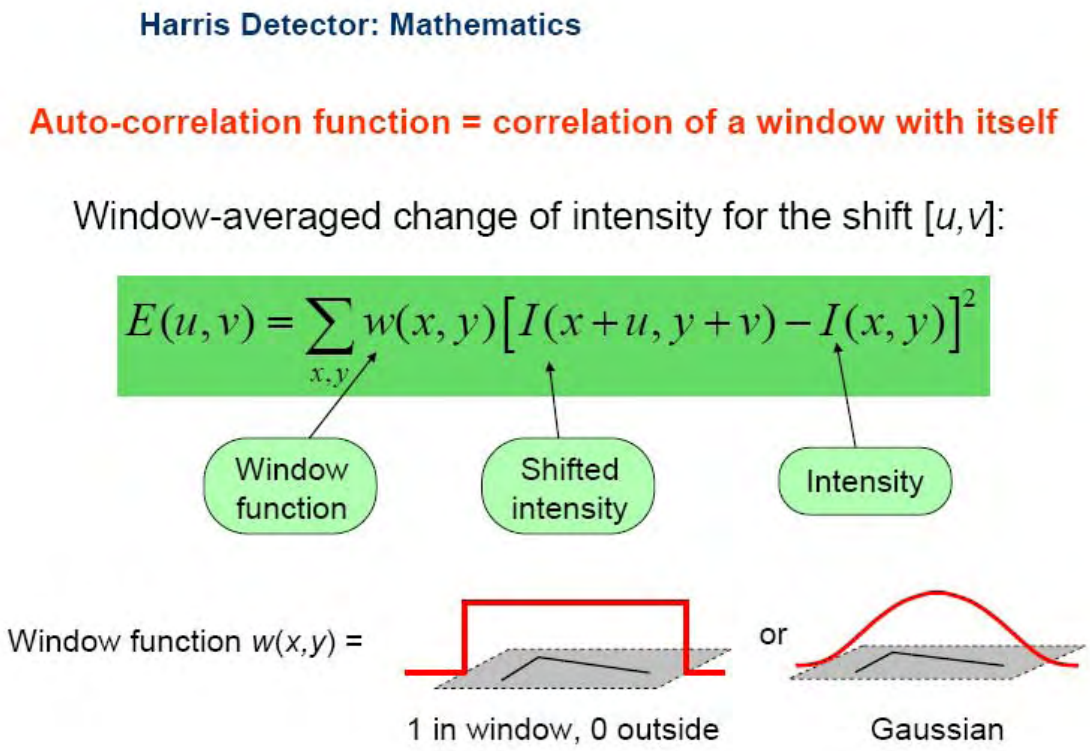
The first idea (Harris, 1988) for detecting invariant features is to use a selection criterion based on a specific property of a feature point: the "cornerness" measure.

This measure says how much a given image window represents a corner point of an object in the scene.

It is based on a simple principle: a feature window in the image represents a corner point if, when moving the window of a small amount in all possible directions, we get a large variation of the gray-pattern.

This idea allows to distinguish a corner feature window from other windows centered on *flat regions*, where small variations are observed when moving the window, or *edges*, where the gray pattern has a large variation only for displacements in the direction orthogonal to the edge, but not along the edge.

***Detection with the auto-correlation matrix***




From a mathematic point of view, the "cornerness" measure can be computed using the autocorrelation function E.

This function is basically the "correlation between a signal and its shifted version", where the shift is 2-dimensional (u,v); or, in other terms, the SSD between the window gray pattern and the shifted pattern.

If a window represents a corner, then E should be increase rapidly for all possible small 2D displacements (u,v) (for example, a few pixels in all directions).

A better function is the weighted autocorrelation, where a weight term w(x,y) is introduced, that gives more importance to differences in the central pixels of the original window, rather than the periphery pixels.

The window, as usually, is chosen to be a 2D Gaussian with covariance proportional to the window size (in order to cover the area properly).

## Harris Detector: Mathematics

### Approximate *E*: the Auto-correlation matrix

For small shifts [*u*,*v*], use a *bilinear* approximation of I

$$I(x+u, y+v) = I(x,y) + \begin{bmatrix} I_x(x,y) & I_y(x,y) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

$$\Rightarrow E(u,v) \cong \sum_{x,y} w(x,y) \left( \begin{bmatrix} I_x(x,y) & I_y(x,y) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \right)^2 = \begin{bmatrix} u, v \end{bmatrix} Z \begin{bmatrix} u \\ v \end{bmatrix}$$

where *Z* is the 2×2 auto-correlation matrix:

$$Z = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

In order to make computations easier, the autocorrelation E is then approximated with a bilinear function: in other words, since u and v are small, instead of using the shifted window I(x+u,y+v), we approximate it to the first order in u and v (bi-linear), by taking the image gradient in (x,y).

The result is a quadratic form in (u,v), that approximates E: a second-degree polynomial (quadratic) where the (2x2) coefficient matrix Z is called auto-correlation matrix.

We have already seen this matrix for the KLT algorithm, where Z is used both to select good features to track, and to track them across subsequent frames (pure translation model).

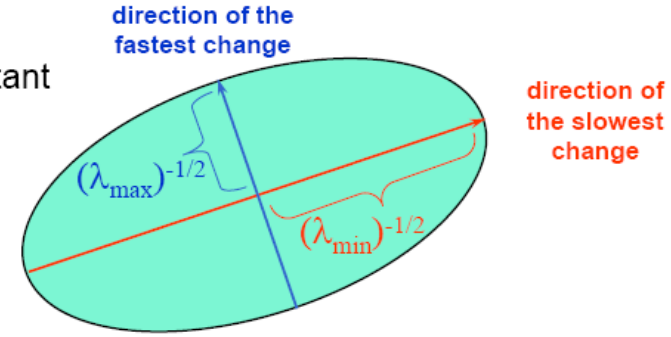
### Harris Detector: Mathematics

Intensity change in shifting window: eigenvalue analysis

$$E(u,v) \cong [u,v]\ Z \begin{bmatrix} u \\ v \end{bmatrix}$$

$\lambda_1, \lambda_2$ – eigenvalues of $Z$

Ellipse $E(u,v)$ = constant

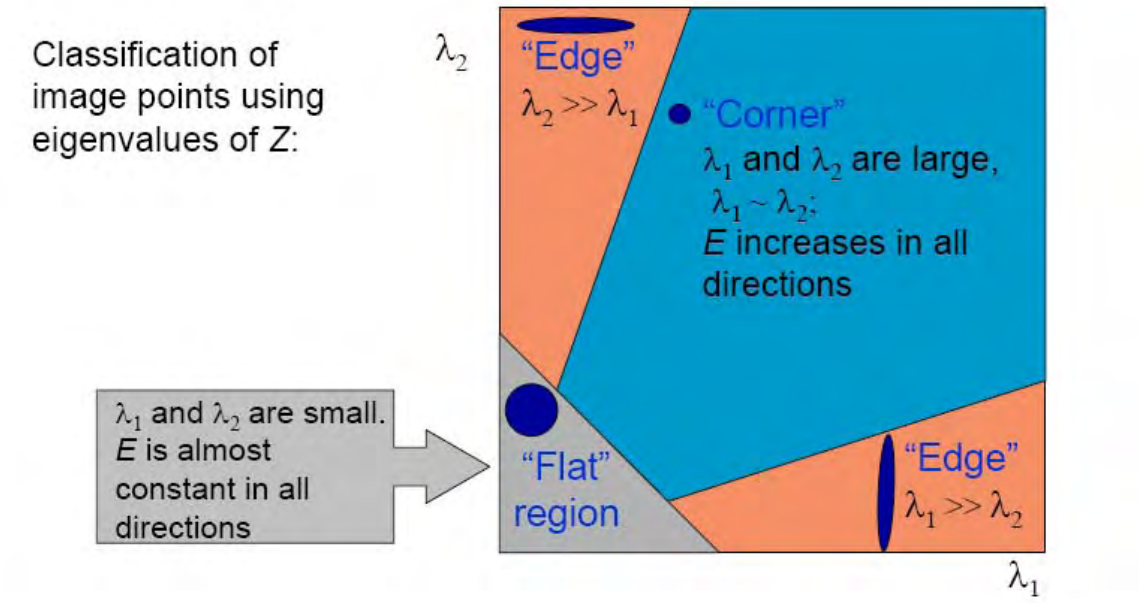



The two orthogonal eigenvectors/eigenvalues of Z (which is a symmetric, positive-definite matrix) represent the axes of the elliptical level sets of E (or actually, the approximation of E for small u,v).

In particular, the largest eigenvalue corresponds to the direction of fastest change in E, while the other corresponds to the slowest change direction of E.

***Harris "cornerness" measure***

## Harris Detector: Mathematics




Therefore, we have a criterion for selecting "corner-like" points, by computing the two eigenvalues of Z.

- Flat region: there is a slow increase of E along all directions → both eigenvalues are small
- Edge: one eigenvalue is large, the other is small
- Corner: both eigenvalues are large

### Harris Detector: Mathematics

**Measure of corner response:**

$$R = \det Z - k(\text{trace } Z)^2$$

$$\det Z = \lambda_1 \lambda_2$$
$$\text{trace } Z = \lambda_1 + \lambda_2$$

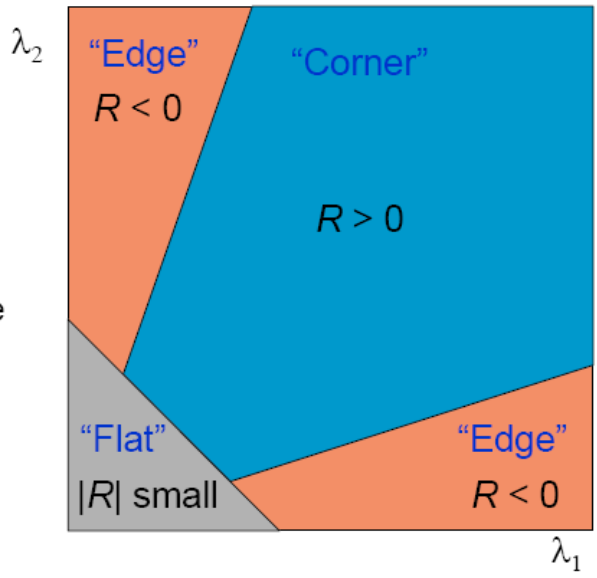
($k$ – empirical constant, $k$ = 0.04-0.06)

The Harris-Stephens selection Algorithm:

1. Find points with large corner response function ($R$ > threshold)
2. Take the points of local maxima of $R$

### Harris Detector: Mathematics

- $R$ depends only on eigenvalues of $Z$
- $R$ is large for a corner
- $R$ is negative with large magnitude for an edge
- $|R|$ is small for a flat region




The Harris measure of cornerness is defined by R, where the constant k is to some extent arbitrary, but in the range 0.04-0.06.

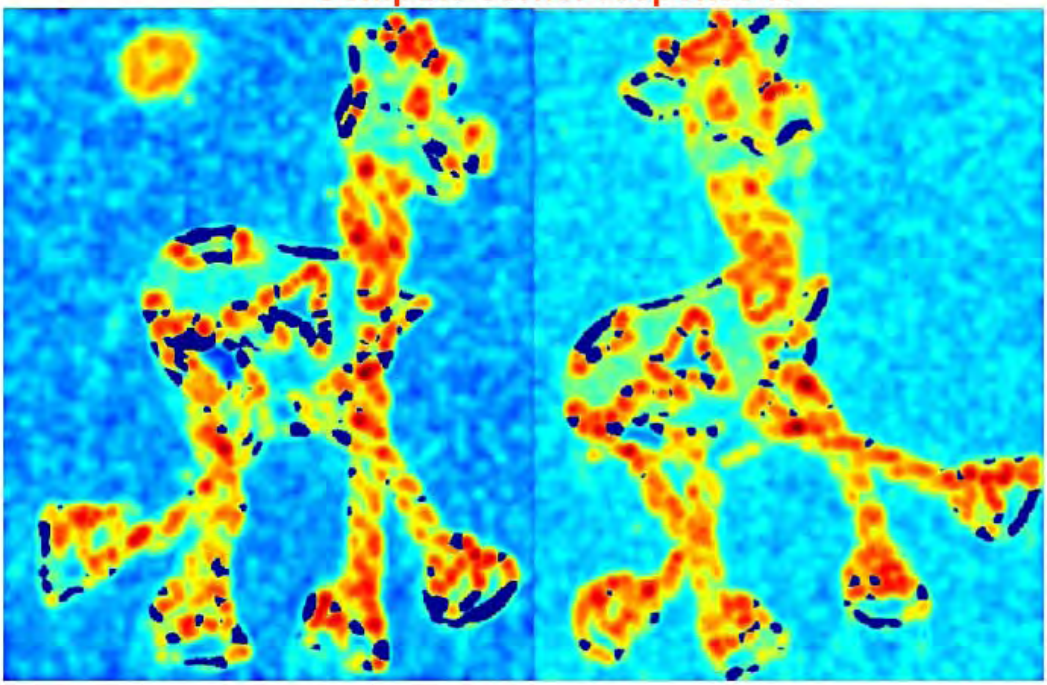
- Flat region: R is small (it can be positive or negative)
- Edge: R is large and negative
- Corner: R is large and positive

## Harris Detector: Workflow

**Example: Original images**

## Harris Detector: Workflow

**Compute corner response *R***

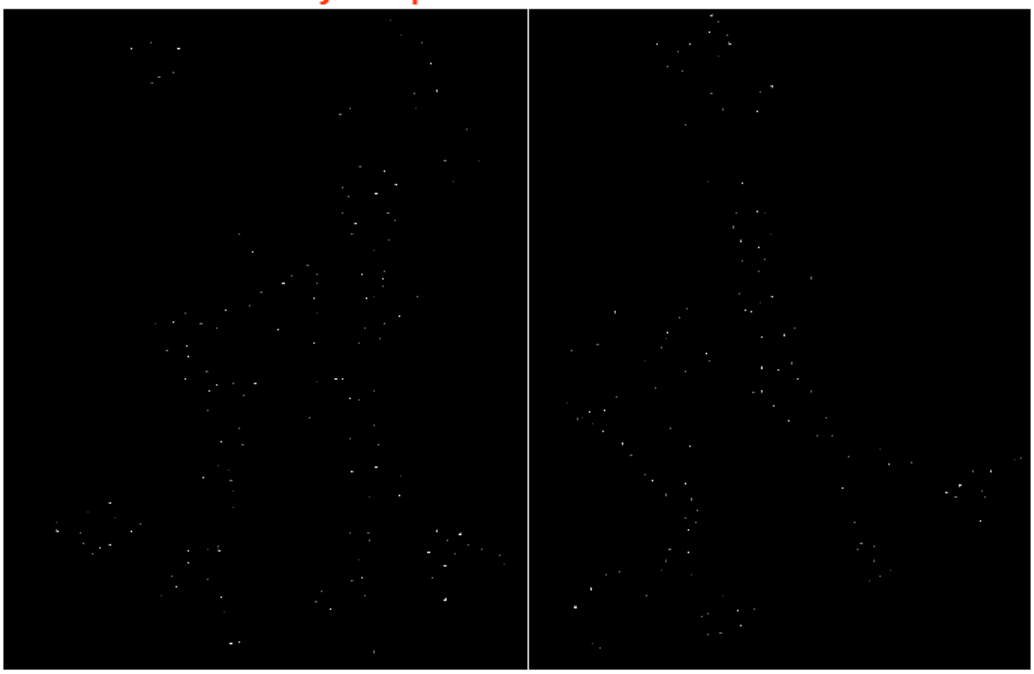


## Harris Detector: Workflow

Find points with large corner response: $R$>threshold

## Harris Detector: Workflow

Take only the points of local maxima of $R$




The Harris algorithm performs the following steps to detect corners:

- From an original image (gray-scale), compute the R response at every window position (NOTE: the window size is selected in advance, typically ~ 25x25 pixels).

- Set a positive threshold on R (it can be fixed or adaptive), and segment the image in “corner-regions”

- In order to find single corner points, for each corner region take only the local maximum of R.

As we can see from the example images, most of the same points are found on the object, even if there is a rotation and a different light → The Harris detector has some invariance properties.

## Harris Detector: Summary

Resume: Harris detector

Average intensity change in direction [*u*,*v*] can be expressed as a quadratic form:

$$E(u,v) \cong [u,v]\ Z \begin{bmatrix} u \\ v \end{bmatrix}$$

Describe a point in terms of eigenvalues of *Z*:
*measure of corner response*

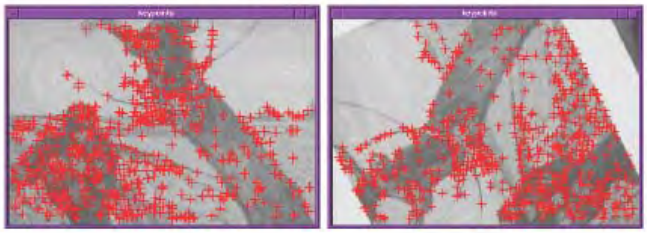
$$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$$

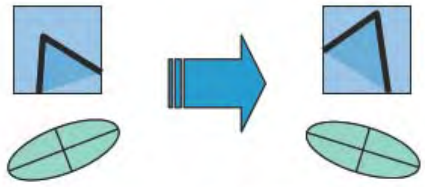
A good (corner) point should have a *large intensity change* in *all directions*, i.e. *R* should be a large positive number

***Invariance properties of Harris' detector***

## Harris Detector: Some Properties

**Rotation invariance: OK**





The ellipse rotates but its shape (i.e. the eigenvalues of Z) remain the same

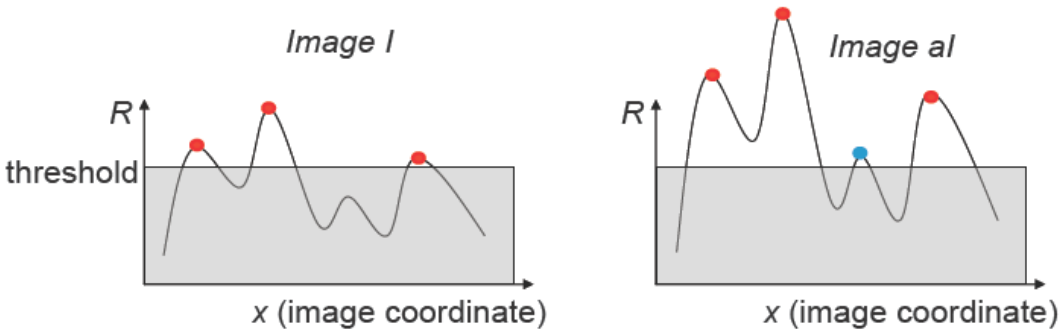
*Corner response R* is invariant to image rotation

## Harris Detector: Some Properties

### Partial invariance to light intensity

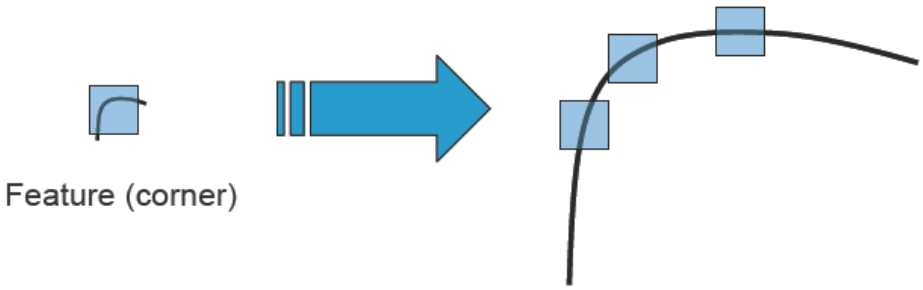
✓ Only derivatives are used ($I_x$,$I_y$) =>
Invariance to intensity shift $I \rightarrow I + b$

✓ Partial invariance to intensity scale: $I \rightarrow a\, I$

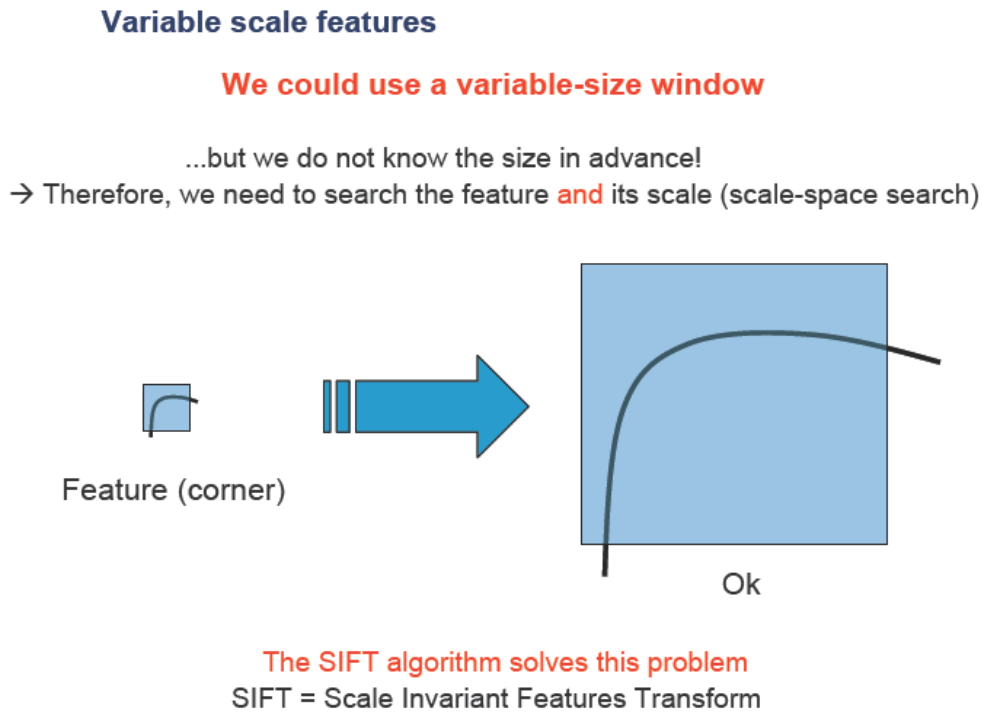



## Harris Detector: Some Properties

### Scale invariance is not respected!




The feature cannot be found anymore (only edges)




Concerning invariance:

- Harris is invariant to in-plane rotations, because the auto-correlation matrix Z for a rotated window has different directions (eigenvectors) but not eigenvalues → R is almost the same

- It is partially invariant to light changes:

If the change is a brightness shift (I+c), the gradients inside the window are the same, therefore Z does not change.
If the change is about contrast (aI) then the gradients change magnitude, and so R→ (aR); therefore, if the threshold is not adapted, we will get more or less corner points, but the old positions are not changed.

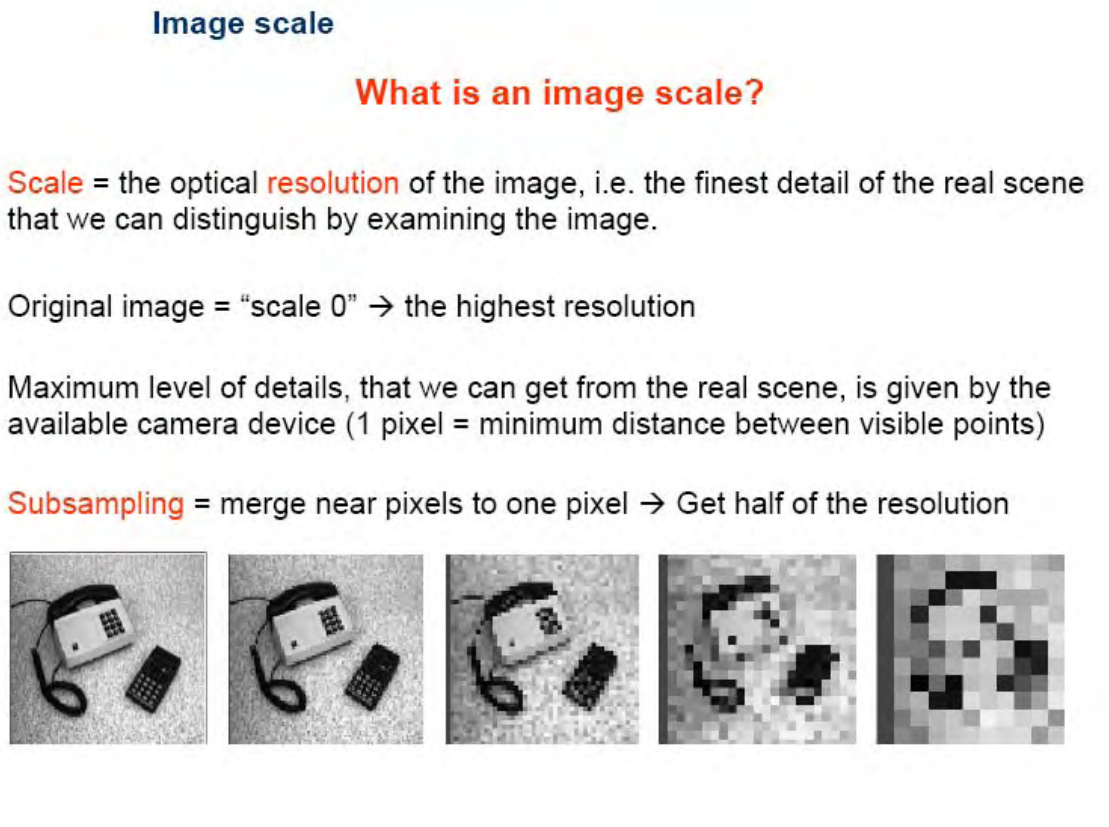
- It is NOT invariant to scale changes: a feature that is detected as corner point, when the scale increases cannot be found anymore, because the feature window (25x25) is not changed!

If we would be able also to know the "correct" window size, we could solve the problem of scale invariance: for a given feature point, select also the scale (=size).
But we cannot do it with the Harris detector.

Another method that solves this problem, more complex but much better, is SIFT: Scale Invariant Features Transform.

This method selects good features by detecting not only their position in the image, but also the scale, searching in an augmented (3D) space of image and scale coordinates (x,y,s) which is also called the *scale-space*.

## Scale-space representation




When we get an image through a digital camera device, the optical resolution is given by the pixel size, which also define the size of the finest detail of the physical scene that can be distinguished.

Therefore, we can say that the original image has the maximal resolution, that we call "scale 1".

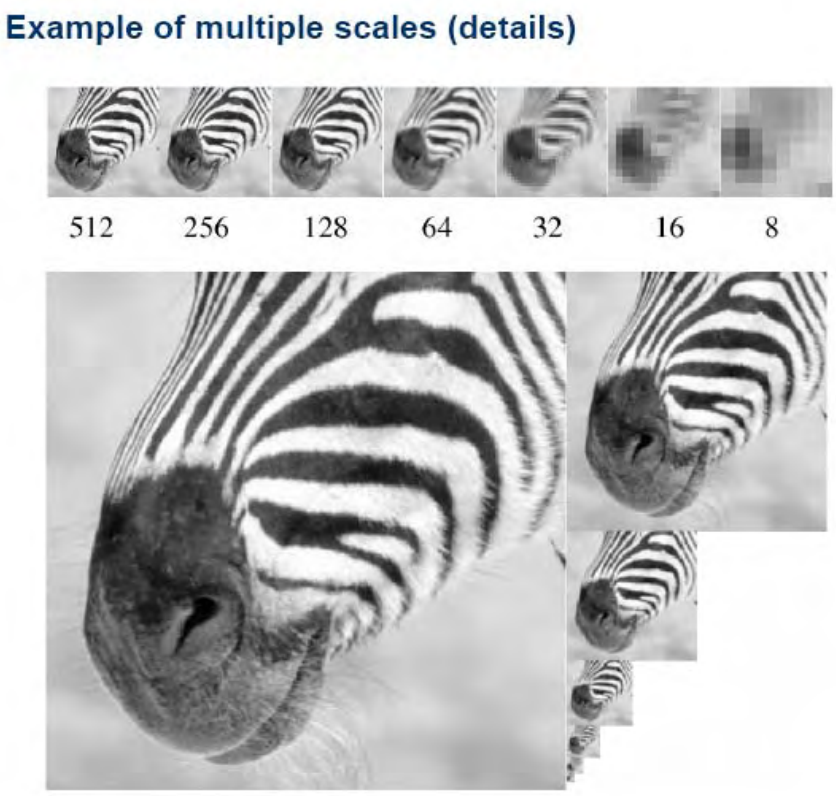
If we perform a sub-sampling, by merging 4 pixels together (2 on both x and y axes), we reduce the number of pixels, and the size of the smallest physical details that can be distinguished decreases of the same amount.

In scale-space language, we say that the scale has been doubled. The term "scale" is related to the size of smallest object details that can be perceived from the digital image.

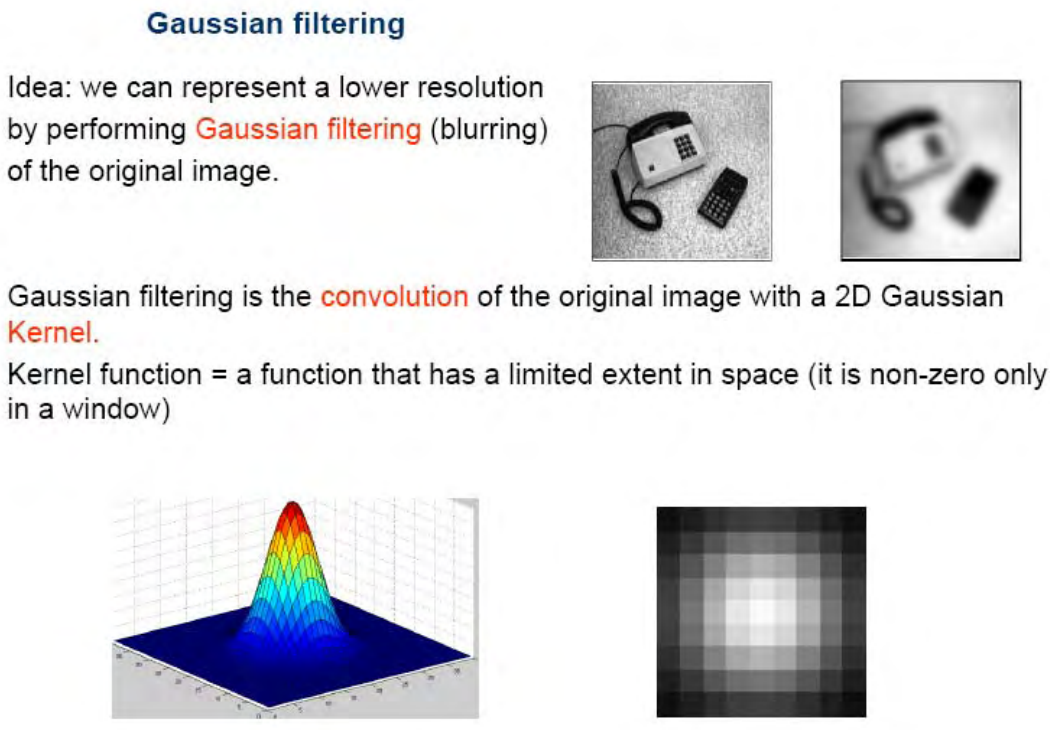
For example, in these images we can see how the phone keyboard (as a whole) can be still identified up to the third scale, whereas the individual keys are visible only up to the second scale. The last scale barely allows to distinguish the handle, the cable and the small calculator, as identifiable intensity "blobs".

In this sense, we can say that the feature "telephone keyboard" exists at scale 2^2=4, whereas the individual keys are visible at scale 2^0=1, and the calculator at scale 2^3=8.

This justifies the idea of a "scale-space" (x,y,s) for identifying features: a feature exists in a given location (x,y) = center, and at a given scale s.




***Gaussian image filtering***




As we have seen, different scales can be obtained by subsampling the original image, which reduces its size of powers of two.

If we want to keep the size in pixels of the original image, and represent it in a continuous scale-space s (not just powers of two), then we can apply an equivalent principle: Gaussian filtering (blurring).

Gaussian filtering is a 2D convolution operation, between the image and a Kernel function (in this case, a 2D Gaussian).

$$J(x,y) = (G * I)(x,y) = \sum_{(u,v)} G(u,v) I(x+u, y+v)$$

A Kernel function G(u,v) is a function which covers a limited area: for a Gaussian, this area is proportional to the covariance matrix.

When we apply a Gaussian filter, we have the effect of blurring the image: this basically means that we loose small details like edges or near corner points, that are “smoothed” and merged together.

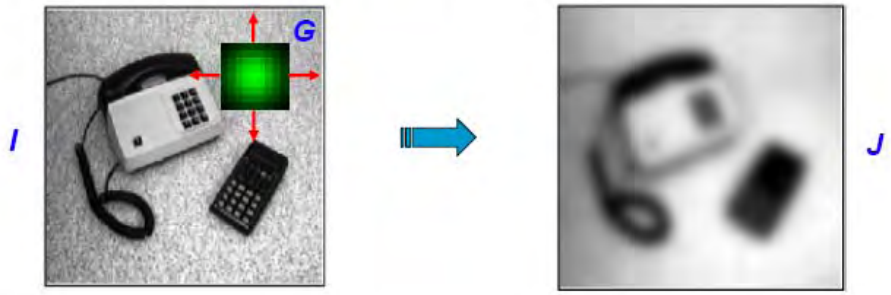
**2D Images Convolution**

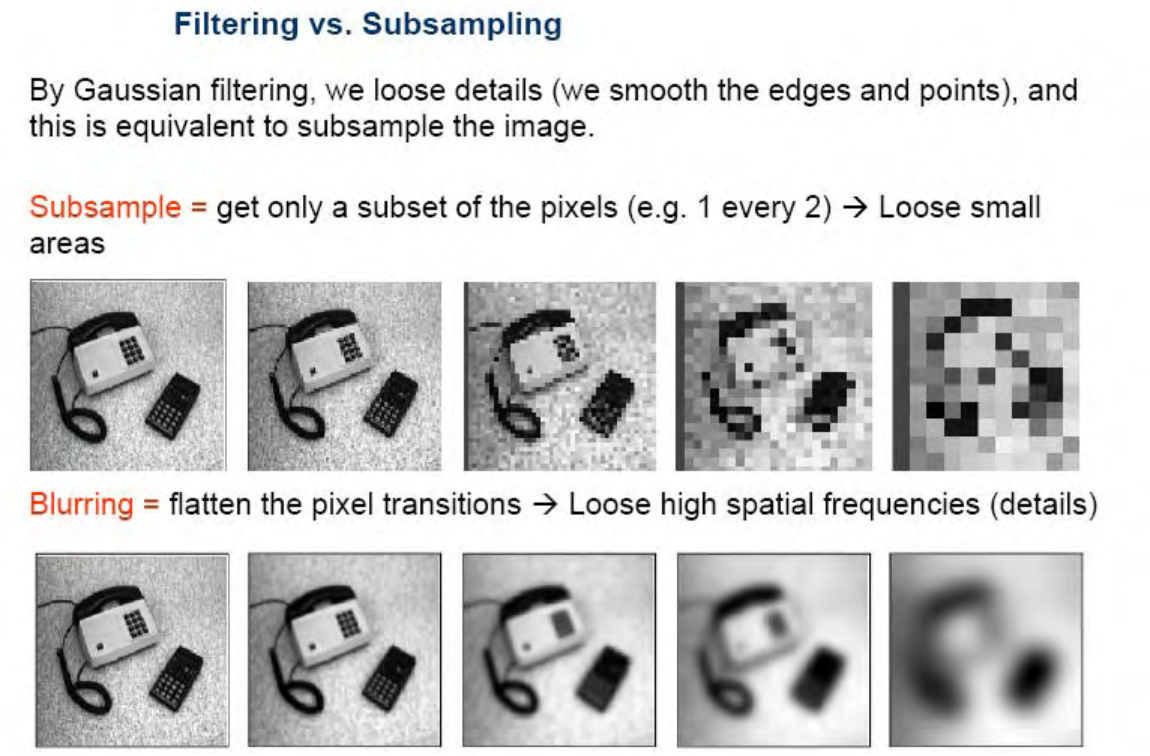
**2D Convolution operation**

Take the signal (i.e. the image) and multiply it point by point by a mobile window (NxN) which is the kernel function translated into the point:

$$J(x,y) = (G * I)(x,y) = \sum_{(u,v)} G(u,v) I(x+u, y+v)$$

Convolution has the effect of “blurring” the signal: it reduces the sharp variations in space → reduces the distinguishable details.






This is equivalent (in scale-space theory) to the subsampling operation.

The larger the covariance of a Gaussian Kernel, the higher will be the equivalent scale of the filtered image.

***Scale-space example***

## Scale-space concept

Scale-space is a 3D representation of an image with 3 coordinates: $F(x,y,\sigma)$
$(x,y)$ = pixel coordinates; $\sigma$ = scale (resolution)

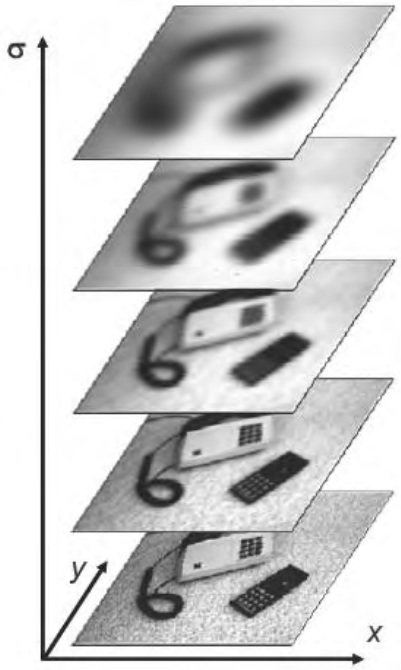
What is $\sigma$?
$\sigma$ = variance of the 2D Gaussian Kernel

The Gaussians have diagonal covariance matrix:
$S = \sigma I$ (circular windows).

How many scales do we consider?
We use exponentially increasing scales: $\sigma_{i+1} = k^*\sigma_i$

$\sigma_0 = 0 \rightarrow$ no filter = full resolution
$\sigma_1 = \sigma$
$\sigma_2 = k^*\sigma_1 = k^*\sigma$
$\sigma_3 = k^*\sigma_2 = k^{2*}\sigma$

with k = 1.4 (typically)




This leads to the concrete idea of (continuous) scale-space: we can take the original image and apply different Gaussian filters, in order to obtain all the scales we want.

A scale-space representation of the image is therefore a 3D space (x,y,s) of image gray values, where (x,y) are pixel positions and s is the scale.

In this representation, we take a diagonal covariance matrix S = sI, with the only parameter s.

Usually, to build a discrete scale-space the scales are sampled in a logarithmic way: the next discrete scale is

$s_{i+1} = ks_i$

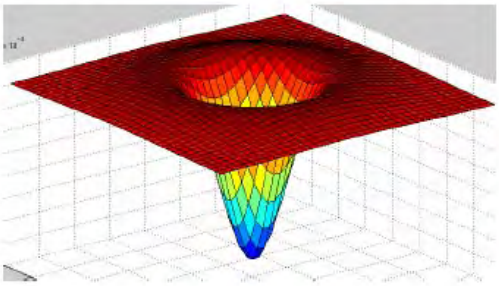
with $1<k<2$ (typically k=1.4).

***The NLoG kernel and the scale-space theorem***

**Normalized Laplacian of Gaussian (NLoG)**

**Scale-space idea:**
**To get scale-invariant features, apply a scale-space Kernel to the image.**

NLoG Kernel = Scale-Normalized Laplacian-of-Gaussian (with size σ)



$$\sigma^2\nabla^2 G = \sigma^2(G_{xx} + G_{yy}) =$$

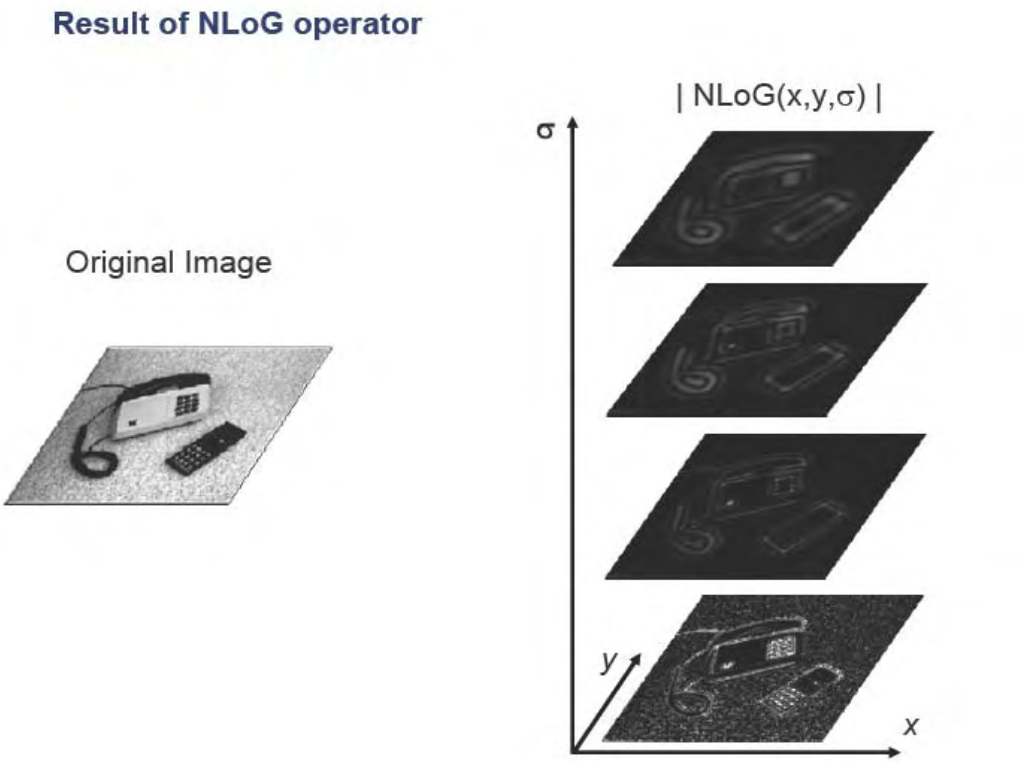
$$= -\frac{1}{\pi\sigma^2}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right] e^{-\frac{x^2+y^2}{2\sigma^2}}$$

If we apply it to the image we have a function NLoG(x,y,σ) (in scale-space):

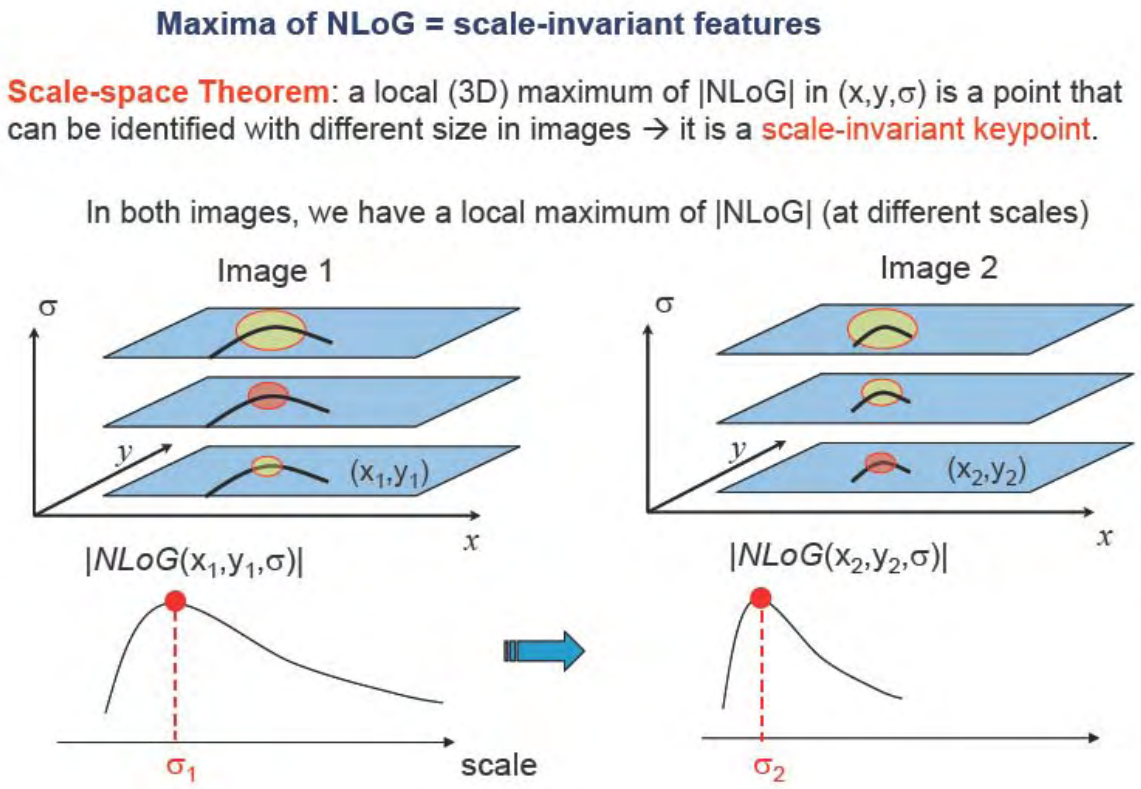
$$NLoG(x, y, \sigma) = (\sigma^2\nabla^2 G) * I(x, y)$$

Scale-space theory is useful in order to detect scale-invariant features.
In fact, a basic result from this theory is that we can select invariant features in scale-space by applying the NLoG operator to each image.

NLoG is the Normalized Laplacian-of-Gaussian kernel function, which is well known in edge detection (for a single image).
Normalization comes from the scale term $s^2$, multiplying the LoG (scale-normalized LoG).




This example shows a NLoG scale-space for the original image on the left.




The scale-space theorem basically says the following:

If we search in NLoG scale-space for local maximal (in 3D) point, then we find feature points with given location and scale (x,y,s) that are invariant to scale and rotation.
This means that, if we take a second image which has been re-scaled and we do the same detection, we will detect more or less the same keypoints, but of course in different position and scale (and, eventually, different orientation).

This is the principle used in SIFT:

SIFT = Scale Invariant Features Transform

meaning that we transform the images in a representation that can be used to select features which are invariant to scale.

But also, once we find the keypoints, we need to represent them in a way that allows the correct matching at different scales and orientation (the SIFT invariant descriptor) → Therefore, the transformation involves also building the descriptor itself, which cannot be just the grey-value pattern (like in KLT).

## SIFT: Scale-Invariant Features Transform

### *Computing the DoG scale-space*

**Difference of Gaussians (DoG) Kernel**

**How can we compute NLoG?**

We use DoG Kernel = Difference of Gaussians: it is the difference between two Gaussians at near scales

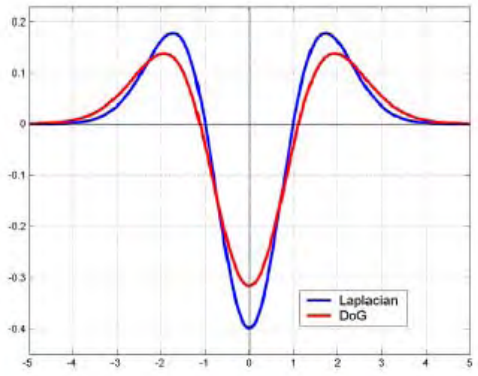
Laplacian Kernel

$$NLoG(x, y, \sigma) = \sigma^2 \nabla^2 G$$

Difference of Gaussians Kernel

$$DoG(x, y, \sigma) = G(x, y, k\sigma) - G(x, y, \sigma)$$

**NOTE: both kernels find features invariant to *scale* and *rotation***

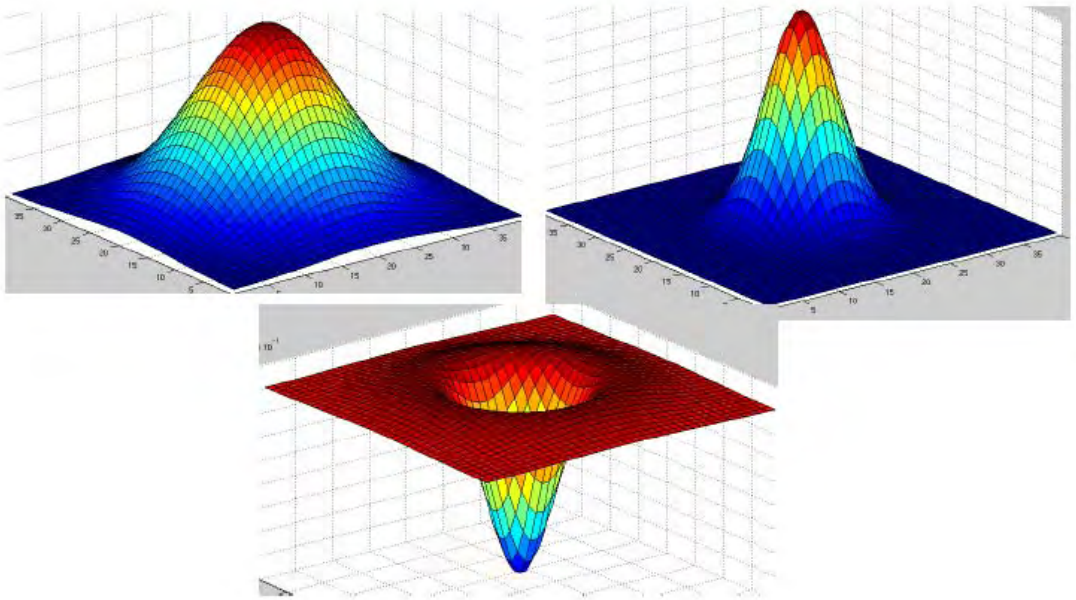



The NLoG can be computed more efficiently by using an approximation: Difference of Gaussians (DoG).

The DoG is a good approximation if the difference in scale is small enough. Both kernels, being symmetric in (x,y), detect features which are invariant to planar rotation as well as scale.

## Difference of Gaussians

The 2-dimensional DoG kernel



## Maxima of DoG

**SIFT: Use DoG to select invariant features**

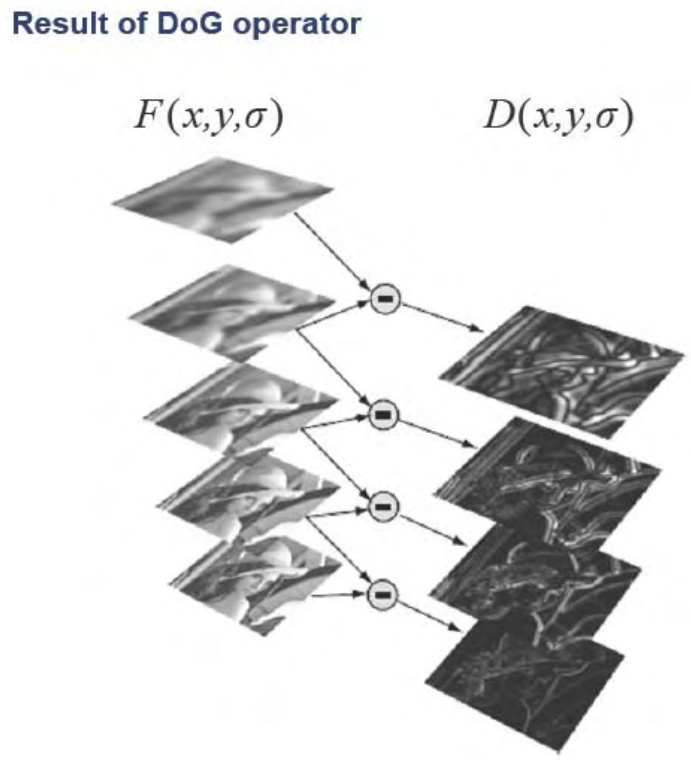
**1.** Filter the image N times with Gaussians, and get the scales $\sigma_0,\sigma_1,\ldots$
→ F(x,y,σ)

$$F(x,y,\sigma) = G(x,y,\sigma) * I(x,y)$$

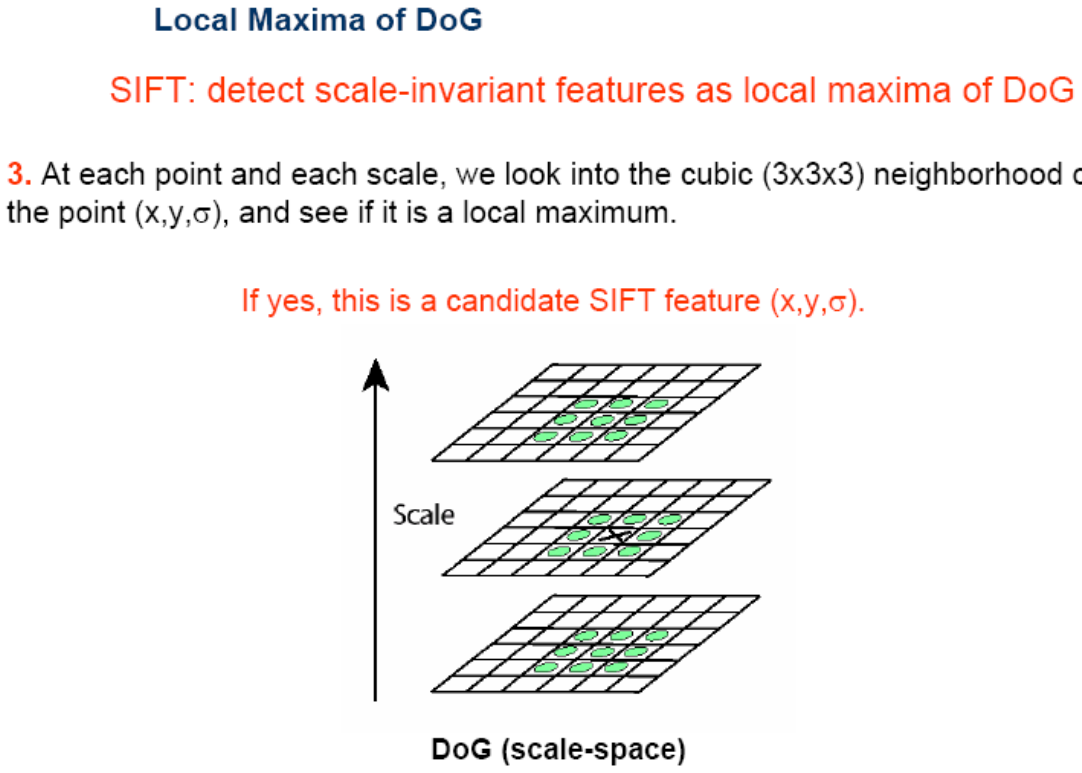
**2.** Compute the absolute difference between images at adjacent scales (σ,kσ)
→ D(x,y,σ)

$$D(x,y,\sigma) = (G(x,y,k\sigma) - G(x,y,\sigma)) * I(x,y)$$
$$D(x,y,\sigma) = F(x,y,k\sigma) - F(x,y,\sigma)$$

In order to compute the DoG, we can first compute the scale-space F(x,y,s) representation of the image I, and then simply taking the differences between adjacent images.

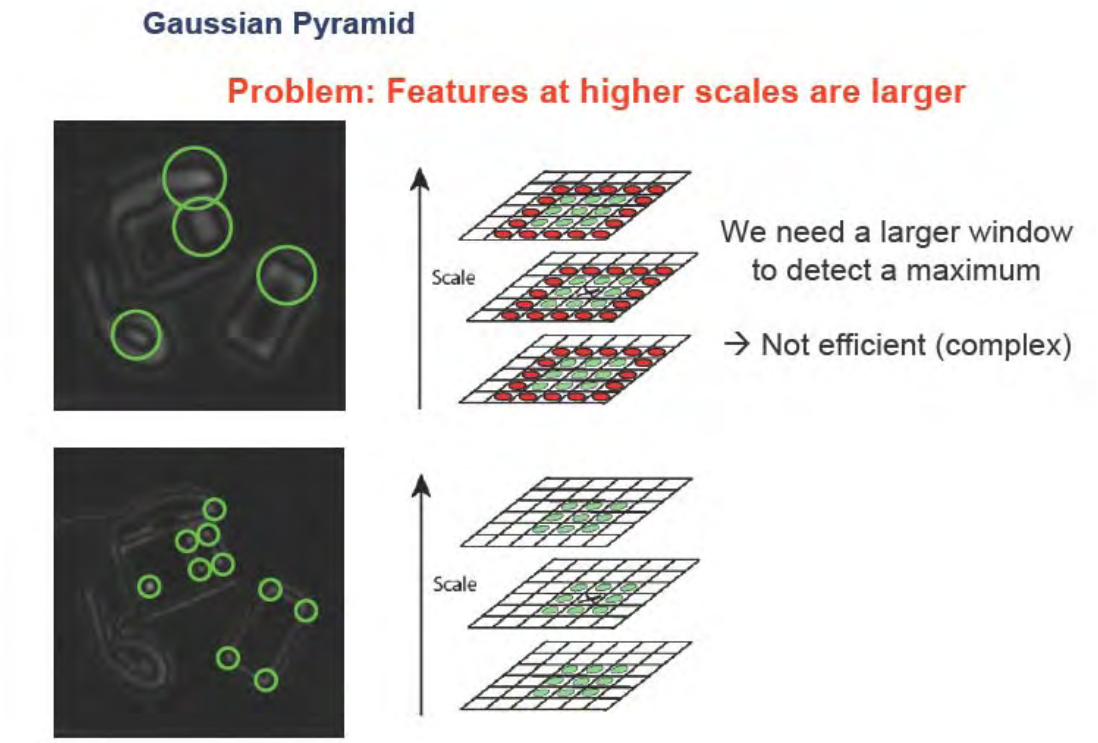



***Detecting invariant features***




From the scale-space theorem, features detection in the DoG scale-space is performed by looking for local maxima of the absolute value |DoG(x,y,s)| in 3 dimensions.

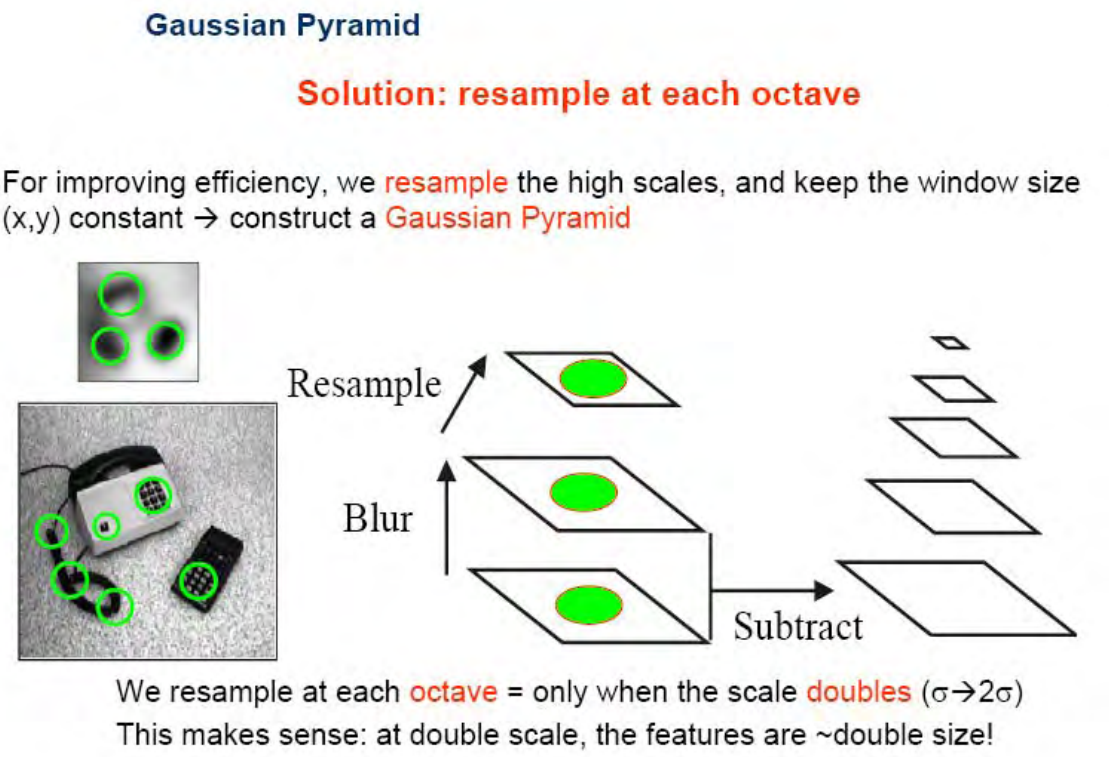
This can be simply done by looking at a (3x3x3) cubic neighborhood of every point (x,y,s).




The problem is that a fixed size of the neighborhood is not good for all scales, since we expect to get larger features at larger scales!

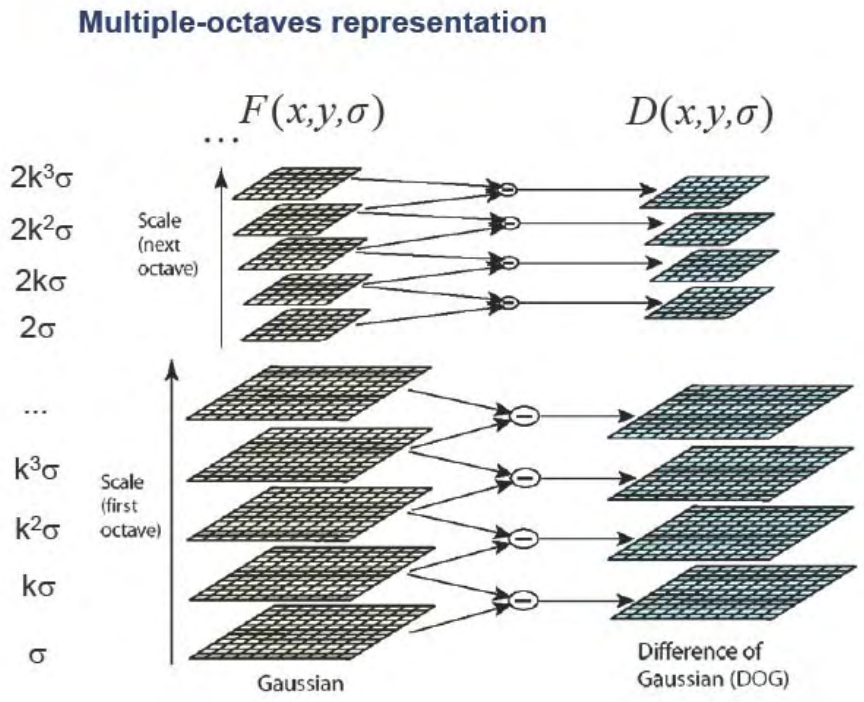
It would then be unpractical to increase the size of the neighborhood, since this would cost much more computational time.

***Subsampling the Gaussian pyramid (Octaves)***



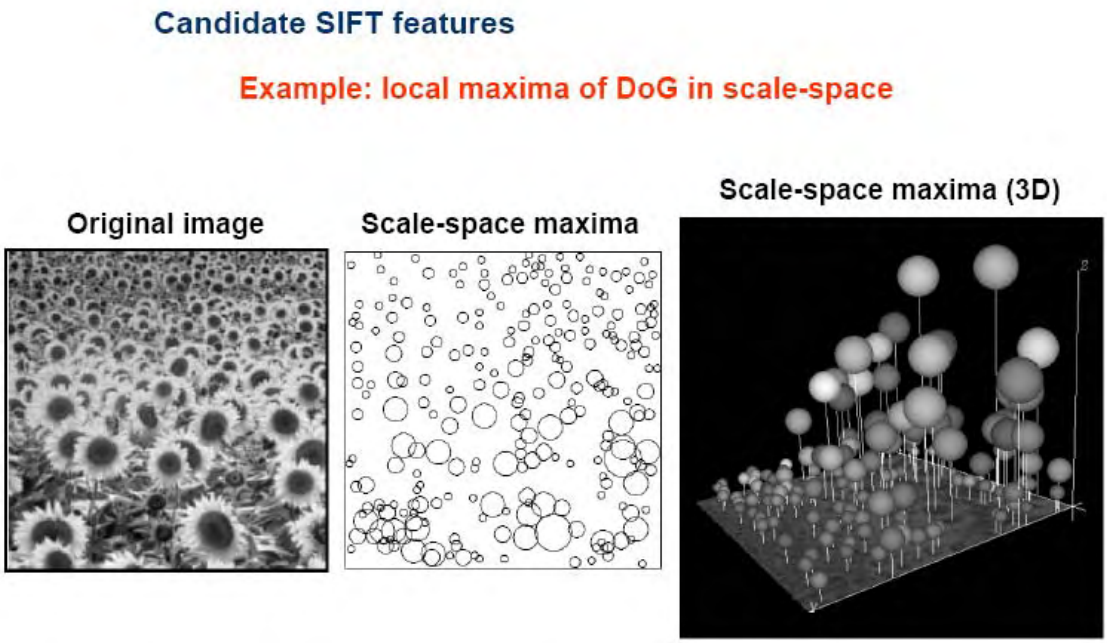

Therefore, another solution is to subsample the higher scales: every double scale s→2s we reduce the size of the image by a factor 2.

Of course, the intermediate scales between s and 2s will keep the same size, and we perform the subsampling only at each octave = double scale, where we expect to get features of ~ double size.



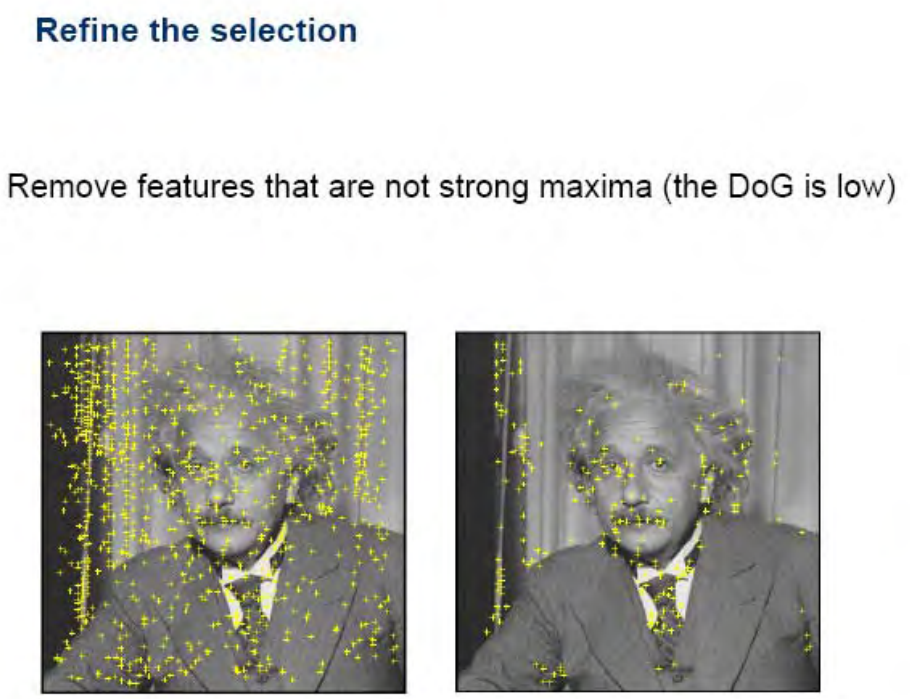

This gives the scale-space pyramid (both image and the DoG).

In order to compute the DoG correctly, we must consider the difference between image sizes, which happens when changing octave.
For this case, SIFT uses a simple trick: before subsampling an image, we keep also the original version; therefore, we can use the proper “version” of this image for the DoG difference (the larger with the previous image, the smaller with the next).




An interesting example of detected invariant features; on the right, we can see the maxima represented in 3D scale-space.

***Refine features detection***

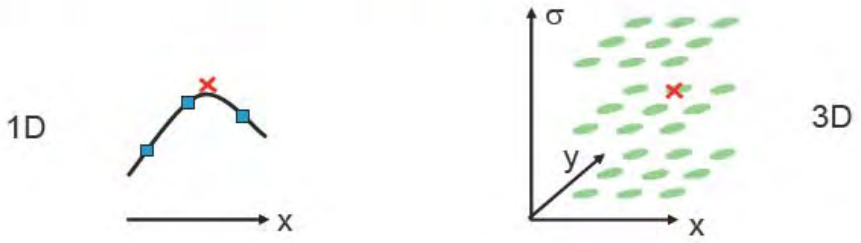



The next step in SIFT is to remove less significant features (weak maxima of the DoG).
These points are local maxima (in their neighborhood), but the DoG function is very small; therefore, they can be removed by putting a threshold on the DoG.

**Refine the estimated location**

**Refine the estimate with sub-pixel accuracy in space and scale**

For each remaining feature, fit a quadratic function to the DoG neighborhood
→ Compute center with sub-pixel accuracy by setting the first derivative to zero.




Afterwards, the remaining points can be localized with better (sub-pixel and sub-scale) accuracy. This can be done by fitting a 3D quadratic function in scale-space, that interpolates all the cubic neighborhood of the point in DoG, and the local maximum can be better detected.

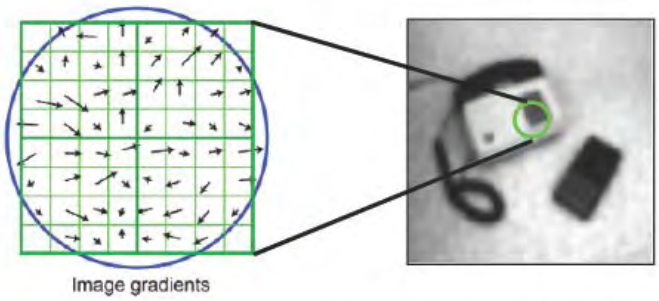
***Building an invariant descriptor***

**SIFT descriptor**

**How do we describe the features?**

We need an invariant descriptor; SIFT specifies also the description.

1. extract the image gradients $I_x$, $I_y$ at the selected scale σ, in a (16x16) window.




NOTE: Remember that at each octave the images are also subsampled!
So, a (16x16) window covers an area which is actually equivalent to (32x32), (64x64), etc…

Once we have the SIFT features, we need a way to represent them (descriptor) in a way that they can be matched between different images.

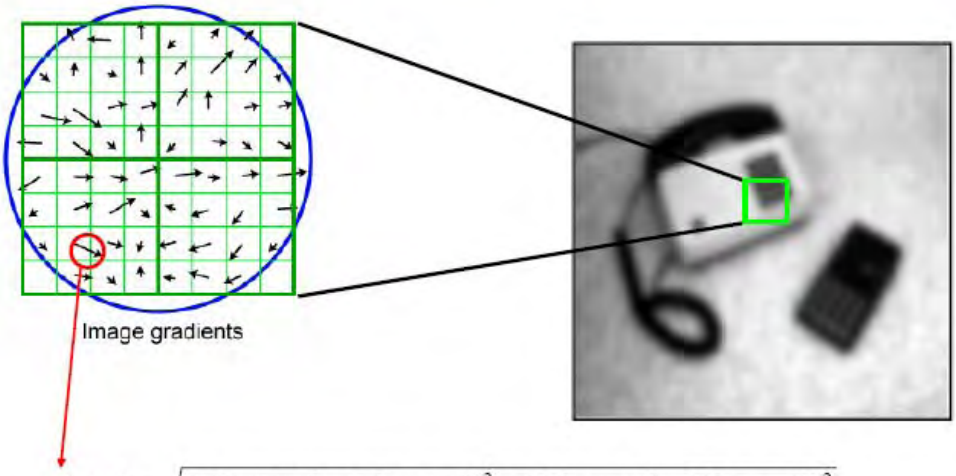
NOTE: remember that the gray-value pattern is not invariant!
Therefore, we also need an invariant descriptor.

In SIFT, instead of using the gray values, we can use the image gradients: this makes the description less dependent on lighting, since for a change $I \rightarrow aI+b$ the gradient vectors change magnitude but not orientation.

The window used for a SIFT feature is about 16x16 pixels; since the images are also subsampled at double scale (pyramid), actually the feature window will be doubled at each octave.

## Gradients computation

**2. Compute the gradient magnitudes and orientations inside the window (at the respective scale)**

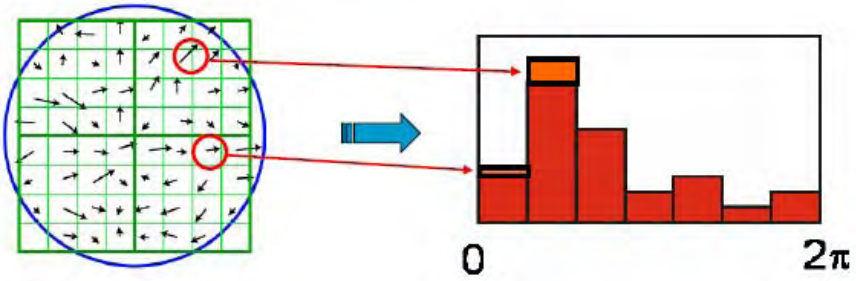



$$m(x,y) = \sqrt{(F(x+1,y) - F(x-1,y))^2 + (F(x,y+1) - F(x,y-1))^2}$$

$$\theta(x,y) = \text{atan}((F(x,y+1) - F(x,y-1)) / (F(x+1,y) - F(x-1,y)))$$

## Orientations histogram

**3. Collect the orientations θ(x,y) in a weighted histogram**




Weighted histogram = the contribution of each orientation to the respective bin is proportional to the gradient magnitude and the distance to the center (w=Gauss)

$\theta_i \rightarrow \theta_i + m(x,y)w(x,y)$

The low magnitude gradients are less important and subject to noise, therefore less meaningful for the descriptor. The central pixels are also more important.

From the pixel window at the detected scale, we can compute image gradient magnitudes and orientations.

Gradient orientations are then collected in an orientation histogram, where each bin represents an interval of angles.

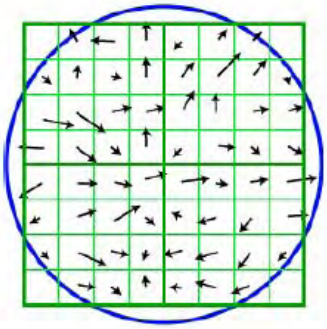
Each gradient contributes in a weighted manner to the histogram: the weight of the contribution is proportional to the magnitude; in this way, we consider large gradients more significant, because smaller ones can be affected by image noise, and their orientation is less reliable.

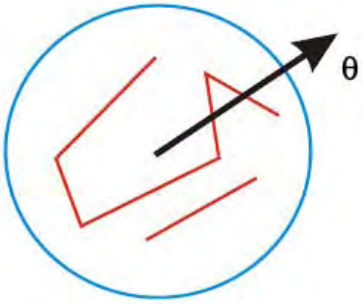
Moreover, we also give a higher weight to the central pixels (as usually), with a Gaussian weight function of covariance proportional to the image scale.

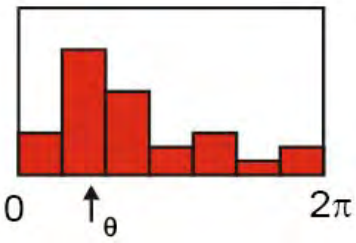
### Feature orientation

**4. Look for the highest histogram peak**

This is the prevalent orientation
of the image gradients over the window,
and we call it orientation of the feature (θ).

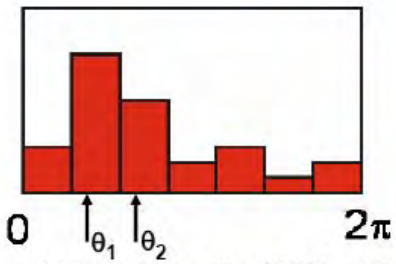


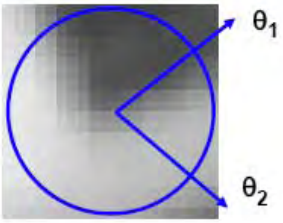


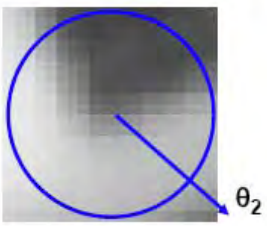

### Multiple orientations

**5. Look also for peaks (if any) that have similar value to the highest one**




In case of multiple peaks (80% of the maximum), we duplicate the feature:

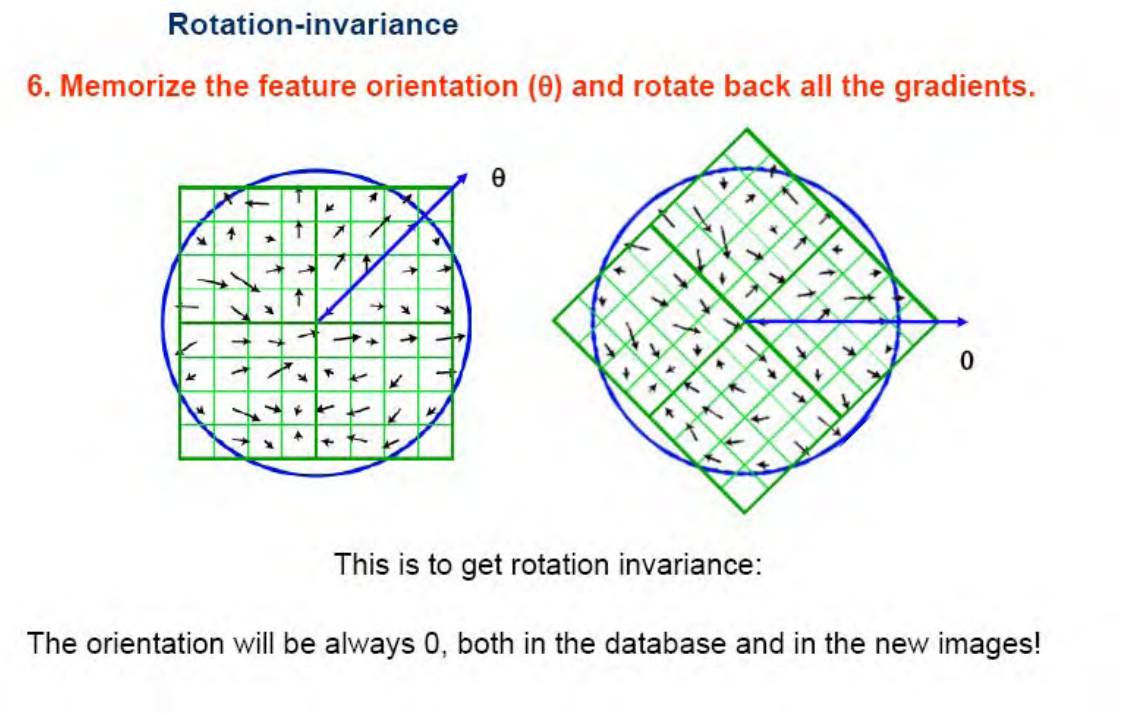




$(x,y,\sigma,\theta_1)$ $(x,y,\sigma,\theta_2)$

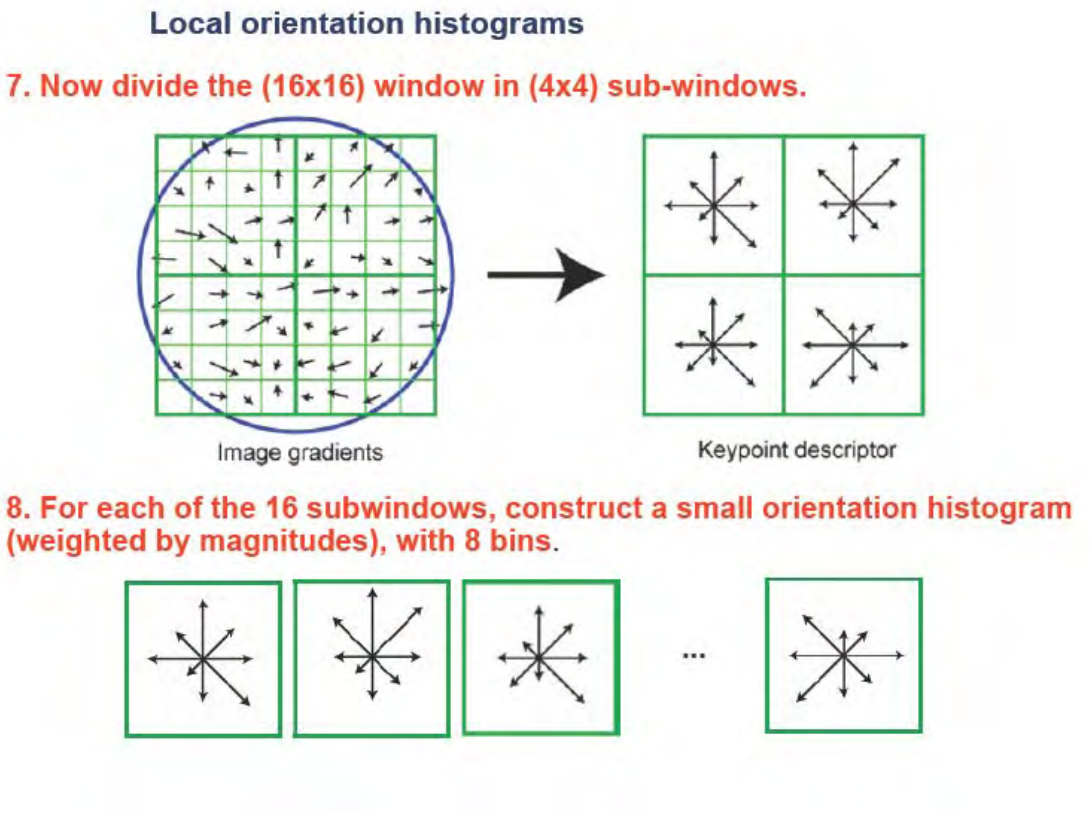
(the features have the same position and scale, but different orientation)

The orientation histogram is used to compute the principal orientation of the feature, which is the highest peak.

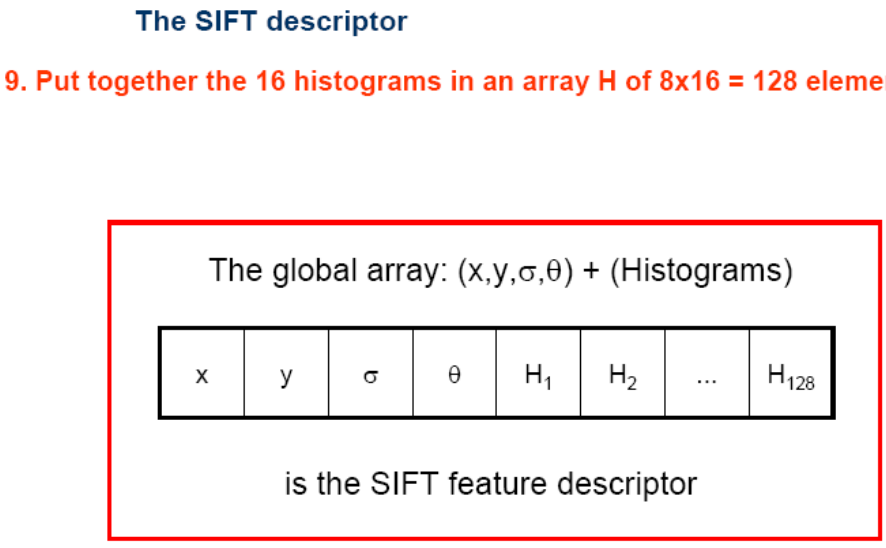
If there are other significant peaks (>80% of the main one), then the feature is duplicated, by assigning a different principal orientation to each copy.




The main orientation θ is stored, and the feature gradients are rotated back, so to have orientation θ=0. This step is important to provide rotation invariance of the descriptor.

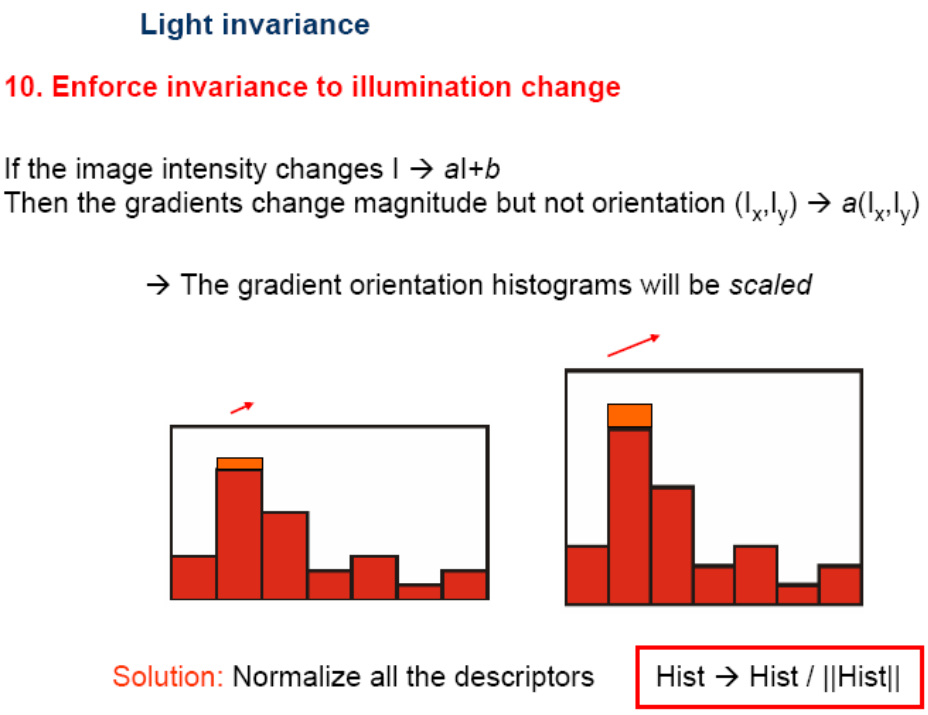



Finally, the main feature window is subdivided into smaller sub-windows (4x4) and for each one, a local orientation histogram is computed (in the same way as the global one, but with less bins (8) because there are less vectors).




The resulting array of 16 histograms with 8 bins is stored in the database, along with the position, scale and main orientation of the feature.

This is the SIFT invariant descriptor.




If the light changes I → aI+b, then the magnitude of the gradients will be M → aM, and the orientations are the same.

Therefore, the weighted histograms will be just scaled by a, and we can obtain also light invariance, by normalizing everything in the Euclidean norm (sum of squares).

***Examples***

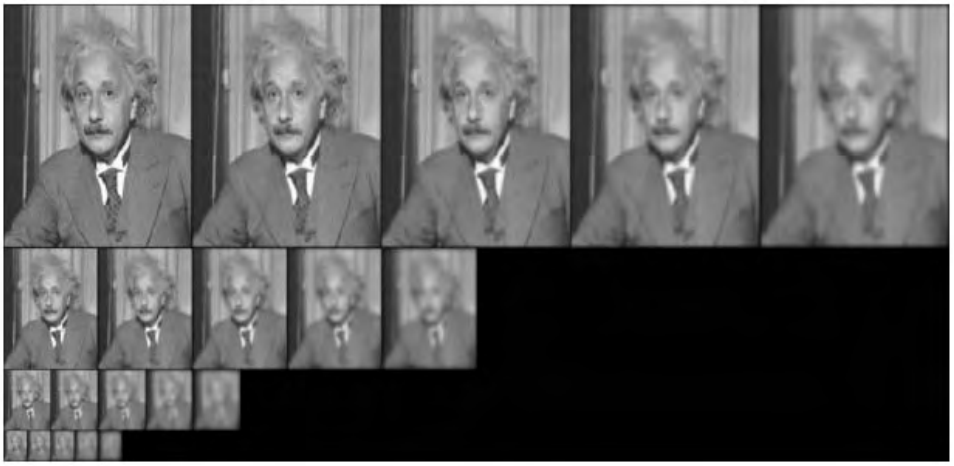
### SIFT features example

How does a SIFT set look like?

The arrows represent orientation and scale (circle size) of the feature: $(x,y,\theta,\sigma)$

### SIFT: Example of features selection

1. Scale-space (Gaussian pyramid) of the image $F(x,y,\sigma)$

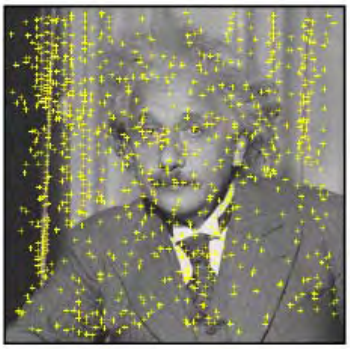


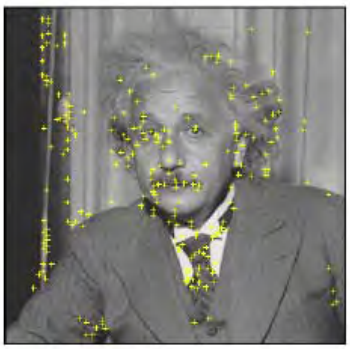
## SIFT: Example of features selection

2 . Difference of Gaussians $D(x,y,\sigma)$

## SIFT: Example of features selection

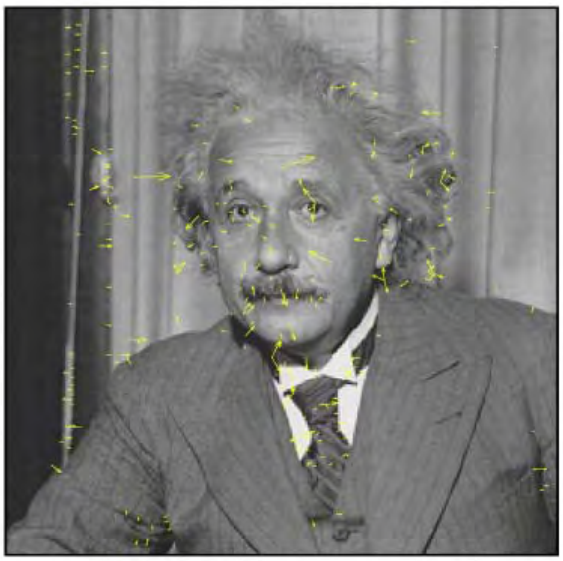
3. Candidate SIFT locations (x,y), and refine of the estimate





## SIFT: Example of features selection

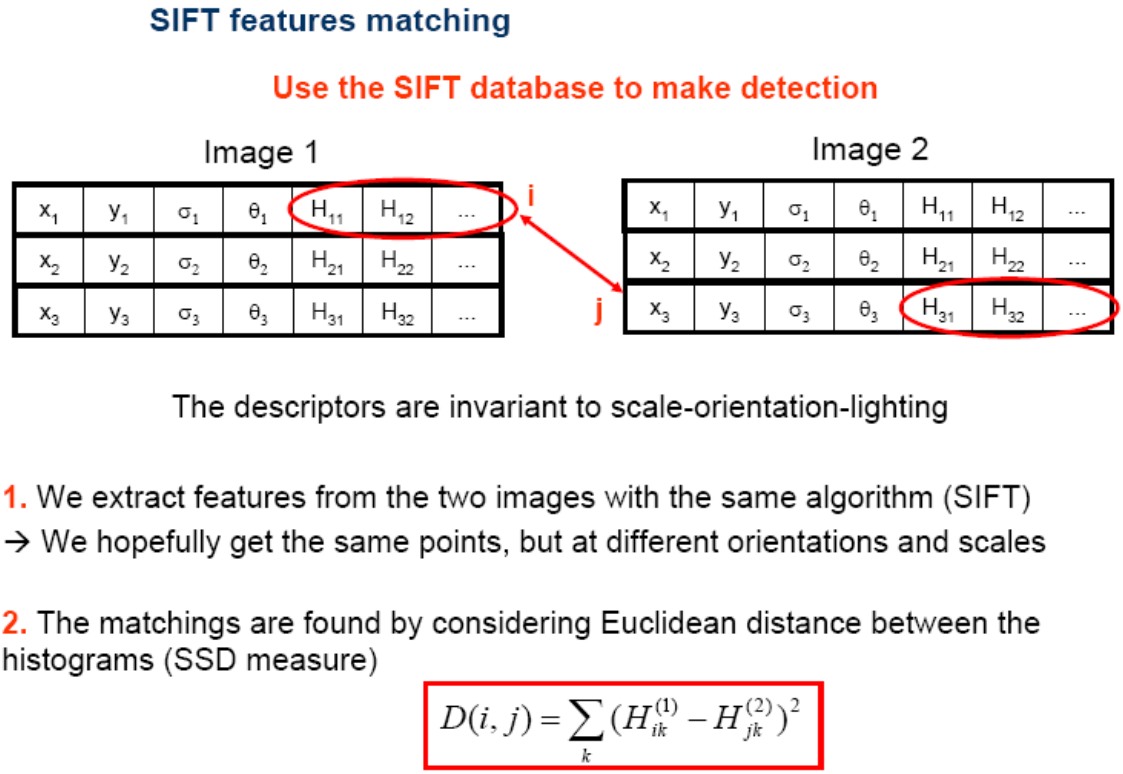
4. Extracted SIFT features, with orientations and scales



## SIFT: Example of features selection

5. SIFT features extracted from the rotated and scaled image
→ Most of the same features are selected! (with new scale and orientation)

***Matching features***




In order to perform features matching, the SIFT detection algorithm is applied to both images, and the two databases are compared to find correspondent points.

If also the new image contains the reference object, we will hopefully detect most of its points, but of course with different scales and orientations.

Since the SIFT descriptors are invariant to scale, rotations and lighting, we can directly compare two features by computing the Euclidean distance (SSD) between the descriptor vectors.

Therefore, matching features will be selected by looking at the minimum Euclidean distance between all possible pairs.

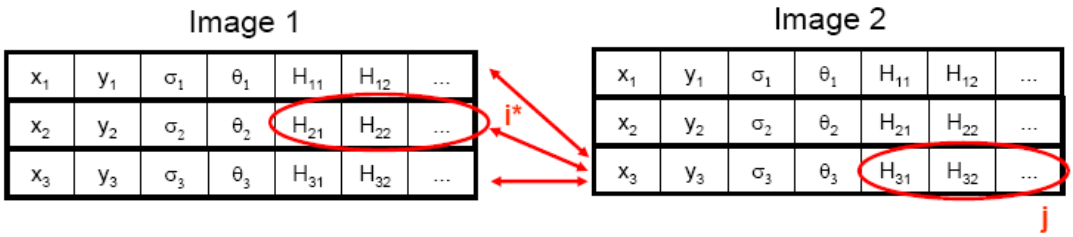
## Search for matchings

### How can we select the correct matchings?

First step: find the nearest neighbor

For every feature **j** in the new image, search the one in the reference database **i*** with lowest Euclidean distance

$$i^* = \arg\min_i D(i, j)$$




## Discard new features

Problem: if a feature **j** is new, it does not correspond to any point in the database
→ It should be just discarded from the matchings

To decide whether to discard **j**, we may look at the minimum distance value D: if still high, then **j** is not good.

But this does not work: it does not exist a unique threshold $D_{min}$ for all features and all possible views!

Better: check the ratio between closest and second-closest neighbors:

$$r = D_{min}/D_{min2} < 0.6$$

If r is low: the feature has only one strong match in the database → OK

A first idea to do the matching is: for every feature in the new image, look into the reference database for the minimum distance one.

But this does not work good, because some keypoints in the new image are not existing in the reference image, so they should not be matched with any reference feature!

This could be done by putting a threshold on the minimum distance, and discard a new feature if the minimum distance with the old database is still too high.

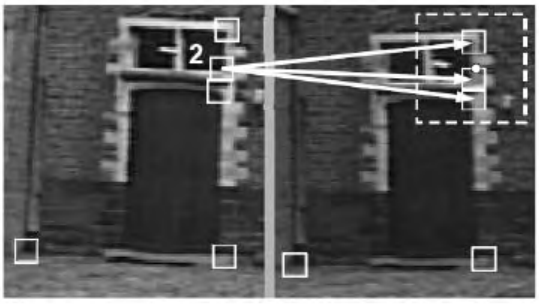
But still, there is no absolute threshold value for this problem, because when the scale, light, etc. is different, the difference can be high also for the correct matching.

Instead, the method suggested in SIFT is to use the ratio between the best and the second best matching: if this ratio is low, then the feature has only one strong matching in the database, and therefore this should be very likely a correct matching.

**Multiple matchings**

Another problem: with this method, we have no symmetry: more points from new image could be matched to the same point of the reference!

(If we expect that there exist only one instance of the object in a picture!)




**Feature i=2 has 3 matchings j**

Another problem concerns the possibility of multiple matchings: if we do a mono-lateral matching scheme (from the new to the old database), it can happen that the same old feature is matched to more new features; this is of course an impossible event.

## Two-sided matching

Better: Do a two-sided matching

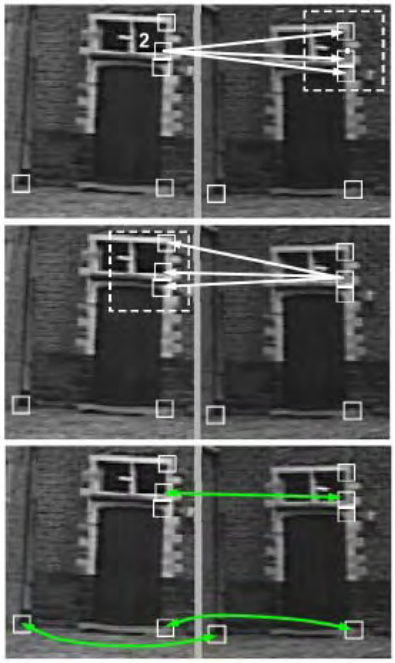
1. Look for matchings i→j

For j=1,...,$N_j$ $i^* = \arg\min_i D(i,j)$

2. Reverse: look for mathings i←j

For i=1,...,$N_i$ $j^* = \arg\min_j D(i,j)$

3. Finally, keep only the matchings (i*,j*) that are bidirectionally identified



A better scheme is a bi-lateral matching: first, match features from the new to the old database, and then do the same in the opposite direction.
Afterwards, we keep only the matchings that have been found in both directions.

NOTE: This does not ensure 100% that we do not still have multiple matchings, but at least most of them will be removed. If we need to absolutely avoid multiple matching groups, then we need a further check to remove them, for example by keeping only the strongest matching of the group.

For an image pair, we can have thousands of SIFT features on both sides, therefore testing every matching in both directions can be a quite expensive process.

For this purpose, there are also faster methods based on binary search trees, which are a bit complex to explain here, and can be found in the literature about the SIFT algorithm.

***SIFT - Resume***

## Resume: SIFT Algorithm

### SIFT = Scale Invariant Features Transform

- **Select features**
  1. Compute the Gaussian Pyramid $F(x,y,\sigma)$ by Gaussian filtering
  2. Subsample the octaves
  3. Compute the DoG pyramid $F(x,y,k\sigma)$-$F(x,y,\sigma)$
  4. Look for local 3D maxima of DoG
  5. Remove weak maxima, and refine the remaining ones (quadratic fit)
- **Compute the descriptors**
  1. Compute the image gradients over a 16x16 window
  2. Compute the global orientations histogram, weighted by the magnitudes
  3. Get the prevalent orientation, and evtl. the secondary orientations
  4. Rotate back the gradients so the orientation is 0 (rotation invariance)
  5. Split the window in 4x4 subwindows, and compute the local orientation histograms
  6. Normalize the histograms to get light invariance
  7. Store the informations $(x,y,\sigma,\theta)$ and $H_1,...,H_{128}$ in the database
- **Match the new image to the database**
  1. Select the SIFT features from the new image and compute the descriptors
  2. Find the matches, by searching the minimum Euclidean distance between descriptors
  3. Remove wrong matches by looking at the ratio first- and second-nearest-neighbors
  4. Use a bi-lateral matching scheme (two times, reversing the role of images)

## SIFT for object recognition

Example: Detect the presence of 3D objects in a complex image.
SIFT detects features and match them to the databases

Correct matchings (the train is found twice!)



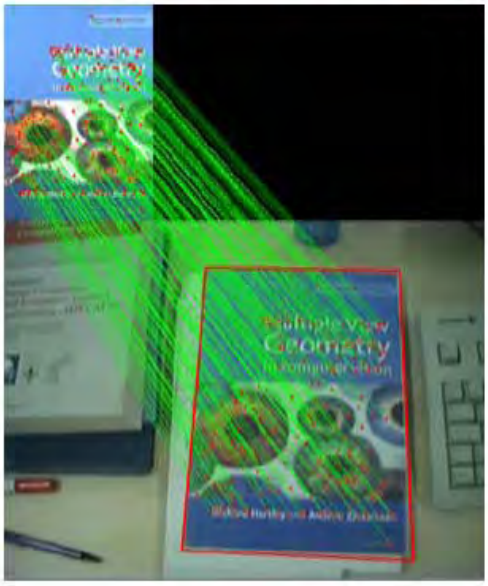
SIFT can be used also for multiple object recognition, without performing an explicit pose estimation.

## SIFT for 3D object tracking

3D pose estimation (frame-by-frame) with SIFT matching

Detection problems: jitter and slow tracking (3 fps)

→ need to combine features tracking (KLT) and detection (SIFT)!




When SIFT features detection is used for an LSE pose estimation (see Lecture 3), we have a point-based tracking method which is actually frame-by-frame performed (every image is processed without taking into account the previous one).

This is in fact slow, if we use SIFT at every frame (about 2-3 fps) and not very accurate (jitter), as we have anticipated before.

Therefore, the best strategy for point-based object tracking is to combine off-line (for example SIFT) and on-line (for example KLT) feature point tracking.

An example in the literature of this combination can also be found in [Lepetit], which uses a different approach for both modalities (off-line and on-line), but the idea is the same.

## Who uses SIFT in the industry?

**Sony Aibo (Evolution Robotics)**

**SIFT usage:**

- **Recognize charging station**
- **Communicate with visual cards**

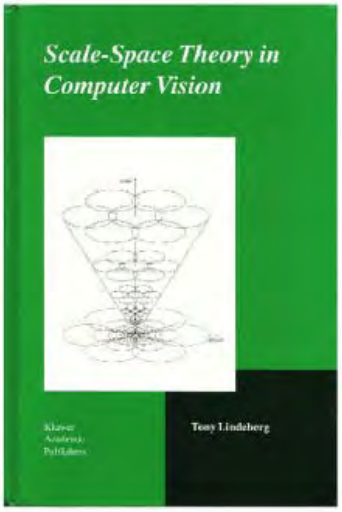

SIFT has been used by the AIBO robot of Honda, which needs an object recognition system to recognize visual cards to communicate with a human operator.

## Who invented SIFT?

Scale-space theory is due to **Tony Lindberg** (1994)

The SIFT algorithm comes from this theory, and was patented by David Lowe:

*"Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image"*
David G. Lowe, US Patent 6,711,293 (March 23, 2004).




**David Lowe**
Computer Science Department
2366 Main Mall
University of British Columbia
Vancouver, B.C., V6T 1Z4, Canada

The inventor of the SIFT algorithm is David Lowe, who holds a patent for it.

The basic theory of scale-space has been instead mostly developed by Tony Lindeberg 10 years before, and described in a quite famous book published by Kluwer.

## *Lecture 9 – Contour tracking using the image edge map*

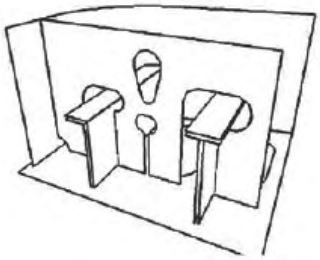
### Definition and motivations

**Contour-based tracking**

**Contour-based 3D tracking**

Contour-based tracking = a procedure to estimate the 3D pose of an object by using contour lines extracted from a full 3D (CAD) Model

Contour model = a model that describes the lines (boundaries) of an object, that separate the object from the background, or the internal parts of the surface



CAD 3D model

Image

Contour-based tracking aims to estimating the 3D pose by using only the boundary (contour) visible lines of the object.
For this task, we need a 3D contour model, which can be provided by a standard CAD application.
The contour lines separate either between the object and the background (external contours) or between internal parts of the object itself (internal contours).

The latter usually correspond to weaker edges, since the object surface parts usually have the same color, or reflectance properties; instead, the external lines usually have a strong separation in the image, if the background is very different from the object, and they can be more reliable for tracking.

## Contour vs. Point-based Tracking

### Motivations for contour tracking

1. Edges are usually well-defined (sharp transitions), for many different poses, light conditions, etc.


→ Advantage: more robustness and precision w.r.t. pose and light

2. We can obtain good tracking with low computational time, because contours are rather easy to detect and to track (while point features require complex selection-description-matching)

→ Advantage: faster detection and tracking (real-time guaranteed)

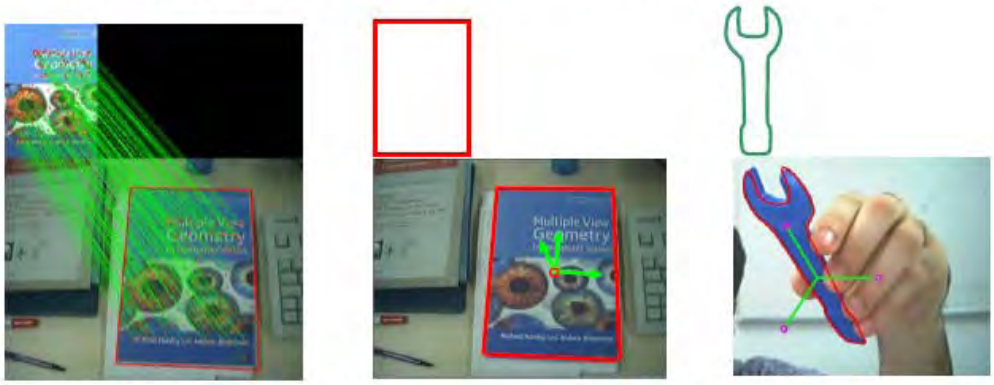
Contour tracking has the advantage of using a simple and, at the same time, robust feature (edges): a long edge line can be clearly identified in the picture also when the light and pose is changing, or when partial occlusions are present.

Moreover, edges are simple to detect and to use for tracking, unlike local keypoints like SIFT, which require a complex computation for detection and matching.
Therefore, we can obtain a higher frame-rate for tracking, which highly improves the quality of the result.

## Contour vs. Point-based Tracking

### Motivations for contour tracking

3. Contours depend only on the object shape (geometry), while feature points require a distinctive surface appearance (not good for uniform colors or textures)



Point-based Contour-based

→ Advantage: we can track a large class of objects or environment items, that have no feature points on the surface

Another advantage is that we can also track objects that have no textured surface or distinctive keypoints, but a distinctive contour line.

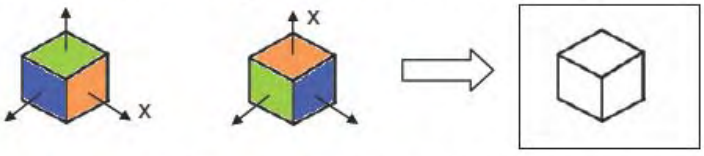
## Contour vs. Point-based Tracking

1. Disadvantage: It is possible to do contour-based tracking, only if the 3D object has a contour which is sufficiently complex, to identify the 3D position without ambiguity


OK

(only 1 pose is possible)

A simple cube can be ambiguous, because of its many symmetries




A revolving surface is even worse: we have no clue about the axis rotation!

The requirement for contour-based 3D tracking is to have a non-ambiguous (possibly complex enough) contour model, in order to be able to uniquely determine the object pose from the projected 2D image lines only (silhouette).

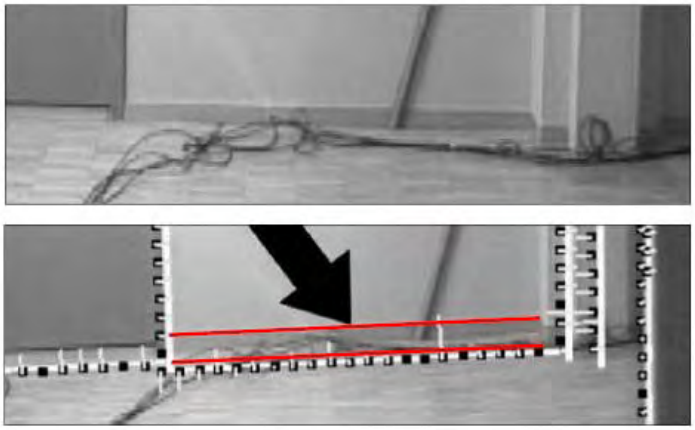
This makes impossible, for example, to track fully a revolving surface (cylinders etc.); in this case, we can still estimate part of the pose, and for the remaining degrees of freedom we need to integrate another visual modality (e.g. point- or template-based).

**Contour vs. Point-based Tracking**

2. Disadvantage: Edges are not distinctive features like points

→ They have weak identity (ambiguity)
Less robustness to false edges: we may follow the wrong edge!

Example : if we find two image edges close together, which is the right one?



A second disadvantage is the ambiguity of the edge features themselves: unlike distinctive keypoints, the matching problem here is more difficult, because it is possibly to have very similar edged, near one another.

## Modeling the Object Contour

### Contour modeling

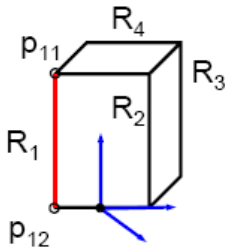
**How can we model the borders of an object?**

We can use straight line segments or curves

Polyhedral : we use segments
to model the boundaries

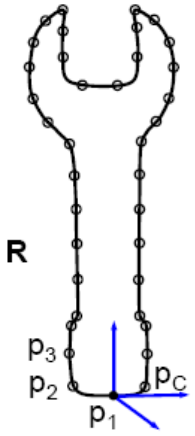
$R_i = (p_{i1}, p_{i2})$

$p_{i1} = (x_1, y_1, z_1)$
$p_{i2} = (x_2, y_2, z_2)$




More general shapes : we need curves
→ B-Splines with Control Points

$R_i = (p_1, p_2, \ldots, p_C)$




In order to model the object contour, we need a representation of lines.

For straight line segments, we can just store the two end-points (in 3D space).
For curved shapes, usually B-Splines are instead used.
A B-Spline representation of a curve line (see next Lecture) allows to model the contour using a finite set of *control points* (again, in 3D space).

When we need to project the model contour from space to image, we can do every computation by projecting only the 3D contour points, in the standard way.

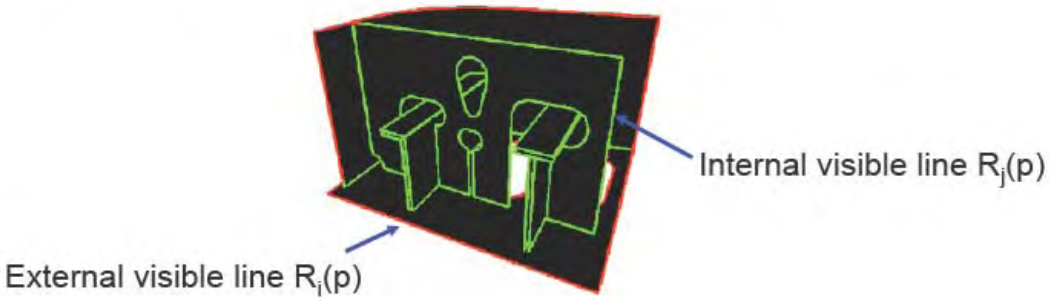
## Contour lines visibility

**Problem: at a given pose, only some contour lines are visible**

We can find the visible model edges, for a given pose p, by projecting the 3D CAD model and test every line for visibility

→ OpenGL can render automatically very complex models, and helps to test the visibilty of lines and surfaces

We can also say if a model edge is internal or external, at pose p.




A very important issue in 3D tracking is the model visibility at a given pose hypothesis s.

In the case of contours, we need to determine which lines of the model are visible from the camera, and use only these lines for computations (matching, measurements, etc.).

We can also distinguish between internal and external visible lines, which are different from pose to pose.

## Measurement type

**Measurement variable z for tracking**

**A.** We can consider the whole image as measurement z=I.

→ The observation instrument is just the camera (z = the pixels matrix)

**B.** We can extract from the image the edge map E, which is another image z=E

→ Our instrument is: Camera + Edge map algorithm (e.g. Canny)

**C.** We can extract from I some edge primitives (short segments, or curves), which are a finite set of geometric features z = ($Q_1$,$Q_2$,...)

→ Our instrument is: Camera + Edge map + Geometric primitives extraction (e.g. Lowe's algorithm, 1987)

For tracking, we always need to define our measurement variable z.

In contour-based tracking, z can again be distinguished among the three levels:

- Pixel-level (A,B): z is a map of pixel-wise values (for example, an edge map, or the gray-scale image itself)
  Example: the Condensation algorithm of (Isard and Blake)

- Feature-level (C): z is a set of detected lines (e.g. segments) from the edge map; in this case, z is a set of geometric primitives, which we use for tracking, into our Likelihood model P(z|s)
  Example: the Lowe's algorithm for segment detection, followed by an EKF tracker (Koller et al.)

- Object-level: z is directly a state (pose) estimate z=p*, obtained after a cost function optimization
  Example: the RAPiD algorithm and its variants, or the CCD algorithm (ML/MAP optimization)

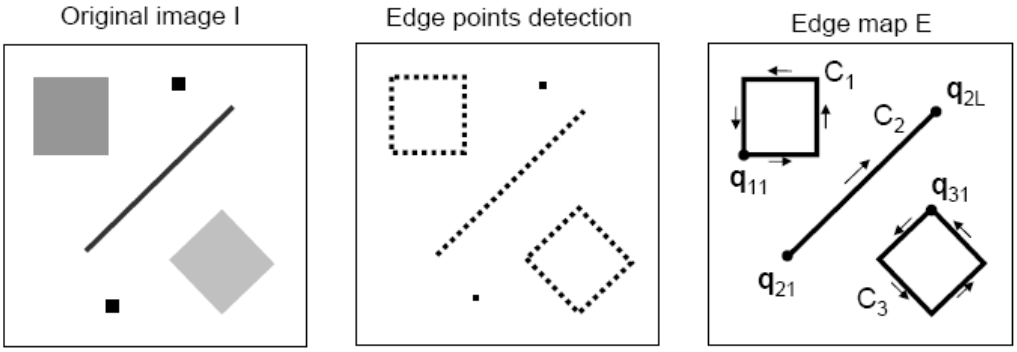
### Obtaining the image edge map for tracking

**Edge map**

**Edge map definition**

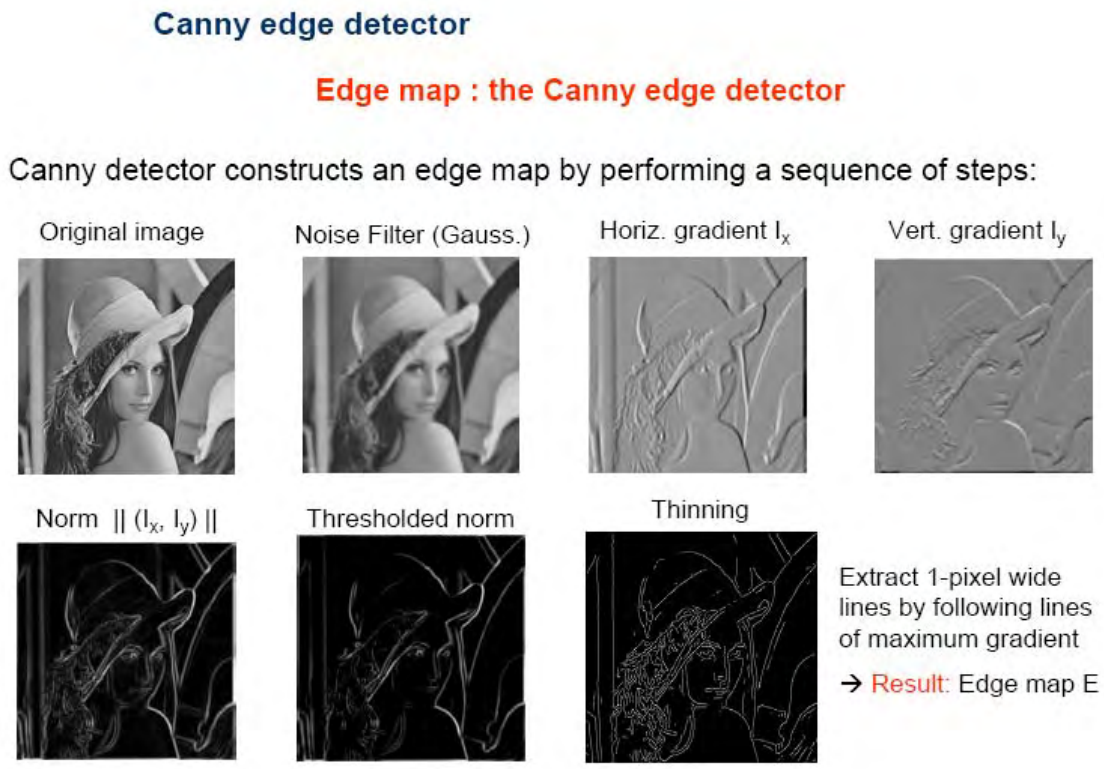
An edge map is a complete map of the image containing 1s where a significant edge point is detected (e.g where the gradient is above a threshold)

The edge points are linked in continuous sequences of image contours $C_i$.
Each sequence $C_i$ is a linked list of edge points $C_i = (\mathbf{q}_{i1},...,\mathbf{q}_{iL})$
NOTE: $C_i$ is not a geometric primitive, but only a long list of pixels!



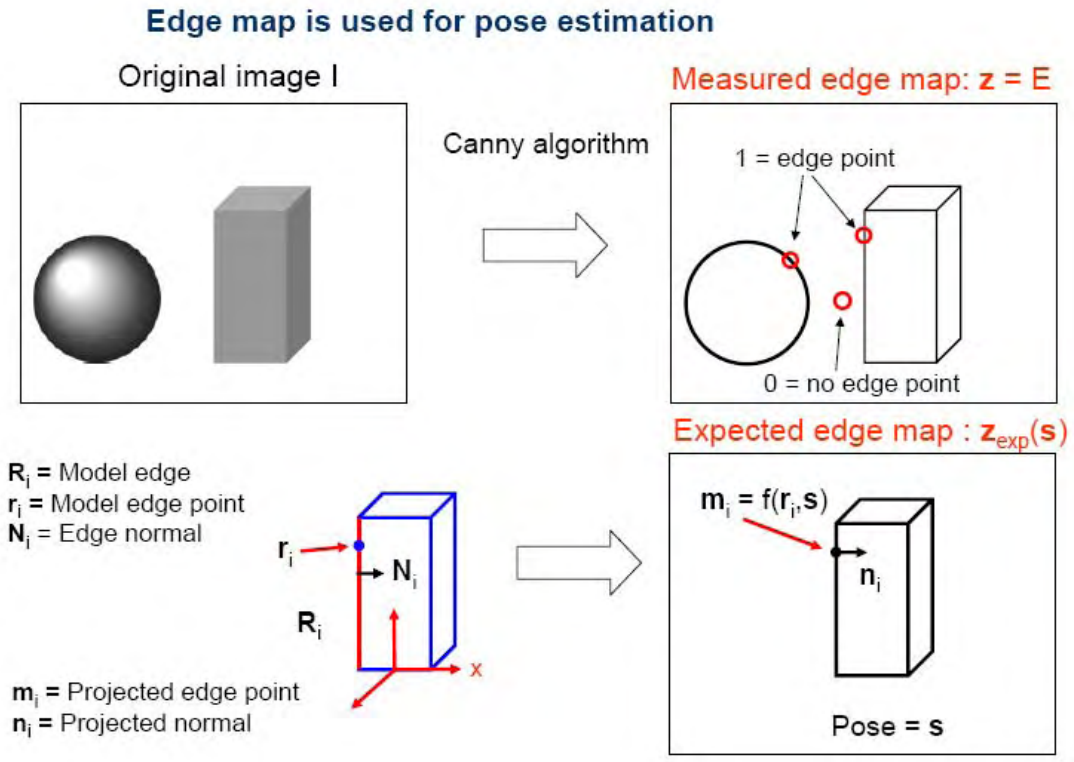

An edge map is a binary image (0/1 values) obtained from the original image, which contains 1 where a possible edge has been detected.

It can be stored as a list of pixel sequences (more compact), or just as a whole binary image.




A good way to obtain an edge map for tracking is the Canny algorithm, which finds connected sequences of 1-pixel wide edge lines, by following lines of (locally) maximum image gradient.

### Using the edge map for 3D pose estimation

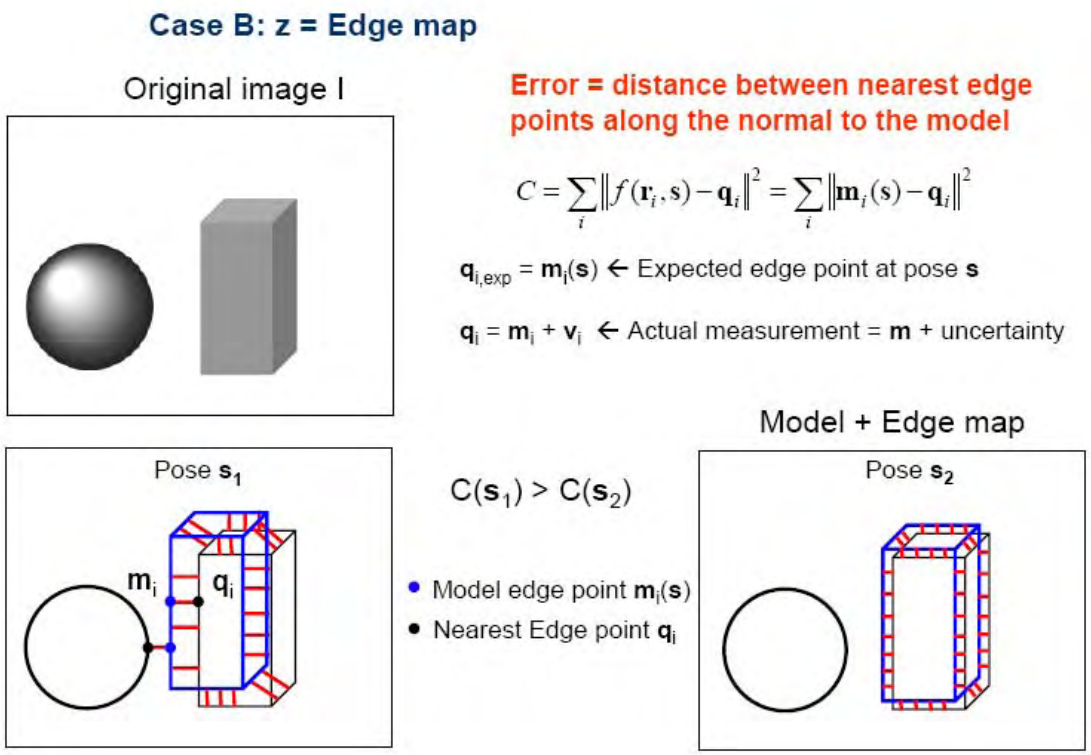



The edge map can be used for pose estimation, by defining a suitable Likelihood function.

For this purpose, we need to project at pose s the visible object lines, which give us the expected edge map ($z_{exp}$). This corresponds to an ideal, noise-free measurement if the pose were the correct one.

In order to obtain the expected edge map at the given pose hypothesis, every visible model edge Ri is projected onto the image.

From a practical point of view, instead of the full 3D edge model, only some points are selected and projected onto the image; the normal direction ni to the edge line at point mi is also computed, an used for computing the Likelihood of the observed edge map w.r.t. the ideal edge map at pose s: P(z|s).




This type of contour tracking methods then evaluates the error between zexp and z, by searching the nearest edge point to each projected model point mi, along the normal direction ni.
The error is defined as a standard SSD measurement using the distances between expected and nearest edges found on the map.

In the ideal case, of course, SSD should be 0: that is, every projected model point mi should be exactly onto an image edge point. Due to errors, image noise, missing edge detections and false edges, this will never happen, so we search again for the “best” pose, that minimizes SSD distance.

If we use standard SSD, minimizing the cost function is always equivalent to maximizing a Gaussian Likelihood function: z = zexp+v, where v is Gaussian with 0 mean and given covariance matrix.

As we can see from the example picture, sometimes this is not the best choice: the nearest edge point can be the wrong one!

A better approach (we will see it next time) considers instead all edge points found along the normal (up to a maximal distance), that can be more than one: this is a multiple hypothesis measurement, and gives a multi-modal Likelihood function, not Gaussian anymore but with several peaks (=modes), one for each edge point found.

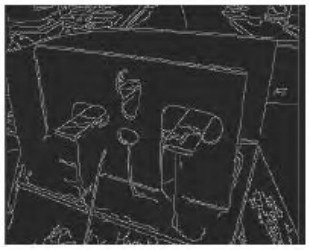
***Contour-based pose estimation in real-time: the RAPiD Algorithm***

**The RAPiD algorithm**

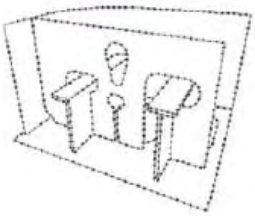
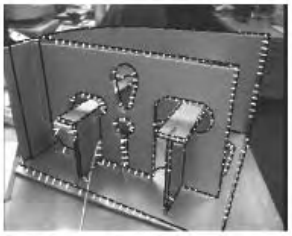
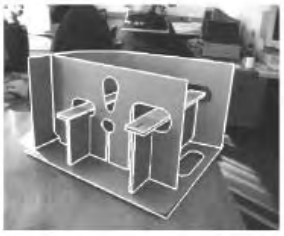
**RAPiD Algorithm [Harris 1992]**

The model points are fitted to the edge map with a LSE procedure

1. Compute edge map → e.g. Canny algorithm

2. Fit model points to the edge points → estimate pose **s**

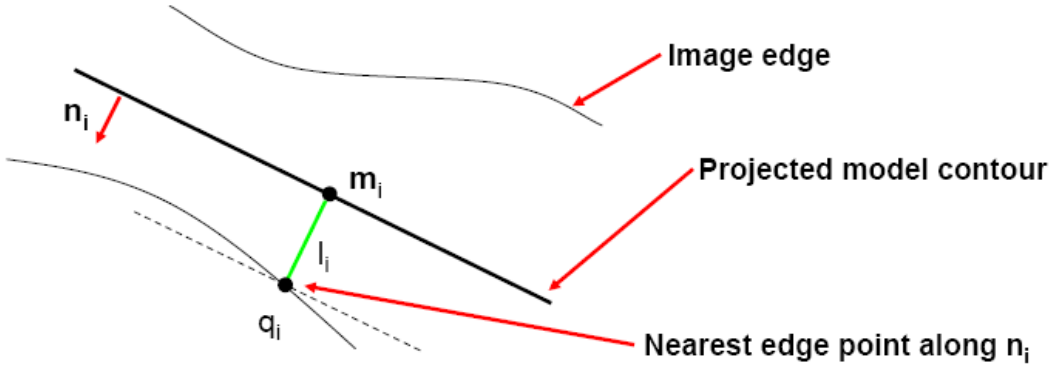




The RAPiD Algorithm has been developed by using the idea above described: minimize SSD between expected and measured edge points along the normal (nearest neighbor only).

**The RAPiD algorithm**

**STEP 1: Find the nearest edge point**

1. Project the model contour point $\mathbf{m}_i(\mathbf{s}) = f(\mathbf{r}_i,\mathbf{s})$ and the normal vector $\mathbf{n}_i=f(\mathbf{N}_i,\mathbf{s})$
2. Search the nearest edge point $\mathbf{q}_i$ on the map, along the normal
3. LSE Error = sum of squared distances $C = \sum_i \|\mathbf{m}_i(\mathbf{s}) - \mathbf{q}_i\|^2 = \sum_i l_i^2$



The first step, as we have seen, amounts to project the model edge point and normal, and find the nearest edge in the Canny map along the normal direction.

The projection from 3D model to screen is again the same used for points (see Lecture 3):

$m_i = f(r_i,s)$

## The RAPiD algorithm

### STEP 2: LSE optimization

Consider a small 3D pose displacement $\Delta\mathbf{s} = (\Delta\alpha, \Delta\beta, \Delta\gamma, \Delta t_x, \Delta t_y, \Delta t_z)$

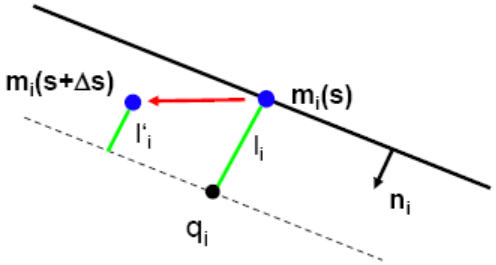
1. For small change of position/orientation we linearize the projection

$$\mathbf{m}_i(\mathbf{s}+\Delta\mathbf{s}) = \mathbf{m}_i(\mathbf{s}) + J_i(\mathbf{s})\Delta\mathbf{s}$$

where $J_i$ is the (2x6) Jacobian matrix of the projection $f$: $J_i = \partial\mathbf{m}_i / \partial\mathbf{s}$

2. Compute the new distance (assuming ~ the same normal $n_i$)

$$l_i' = l_i - \mathbf{n}_i^T J_i \Delta\mathbf{s}$$




The second step linearizes the model projection for small parameter changes Ds, by using the Jacobian matrix at each projected point:
$J_i = \partial m_i/\partial s = \partial f(r_i,s)/ \partial s$

Then, another approximation (for small displacement Δs) is made, by assuming that also the normal direction ni does not change.
At this point, the new distance between q and $m_i(s+\Delta s)$ along the normal $n_i$ is

$l_i' = l_i - n_i' J_i \Delta s$

## The RAPiD algorithm

### Gauss-Newton Step

1. Consider the new error (linearized LSE):

$$C(\mathbf{s}+\Delta\mathbf{s}) = \sum_i \left(l_i'\right)^2 = \sum_i \left(l_i - \mathbf{n}_i^T J_i \Delta\mathbf{s}\right)^2$$

2. Minimize it w.r.t. Δs:

$$\Delta\mathbf{s} = A^{+}\mathbf{b} \qquad A = \begin{bmatrix} \mathbf{n}_1^T J_1 \\ \cdots \\ \mathbf{n}_M^T J_M \end{bmatrix}; \qquad b = \begin{bmatrix} l_1 \\ \cdots \\ l_M \end{bmatrix}$$

3. Update rotation and translation parameters **s** → **s**+Δ**s**

Repeat the procedure (STEP 1+2) with the updated pose **s**, until convergence.

This is equivalent to a Gauss-Newton minimization

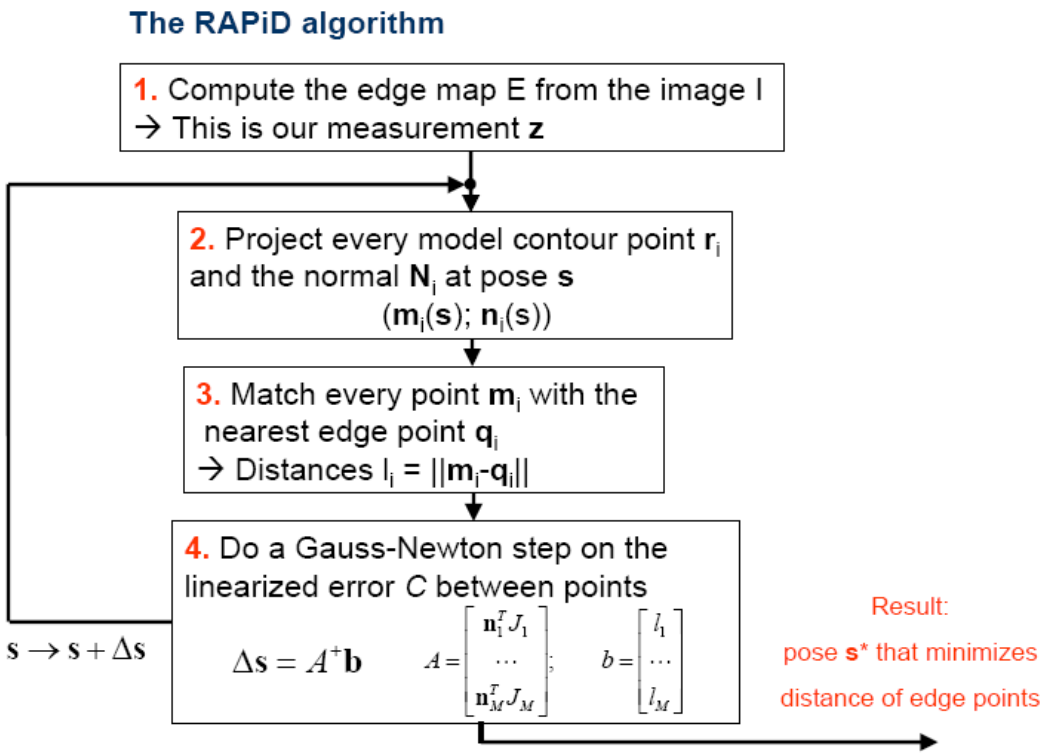
→ We find the minimum LSE pose **s*** to match model and image edge points

This gives a linearized new cost value C(s+Δs)

If we search for the increment Δs minimizing the linearized error C, we have $\Delta s = A^{+}b$, and we can update the parameter s.

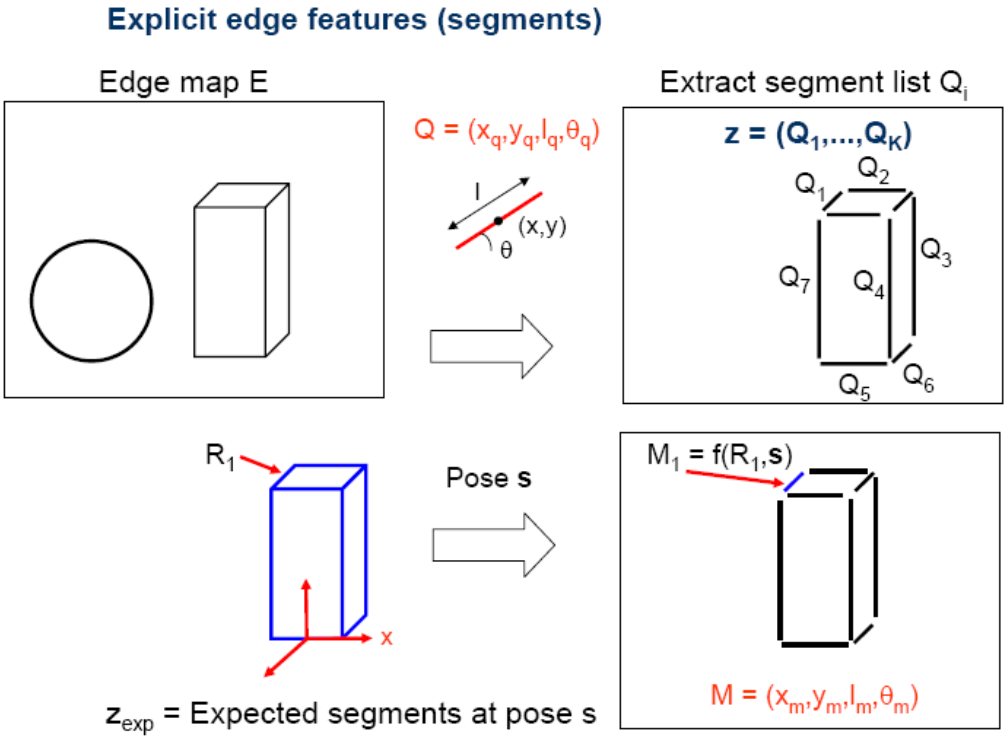
Since we linearize C, this is only an approximation, and we need to repeat the procedure with the new point $m_i$, new normal $n_i$, and new nearest edge point $q_i$.

This is almost equivalent to the standard Gauss-Newton algorithm.




**Using explicit features extracted from the edge map**

***Problem formulation***



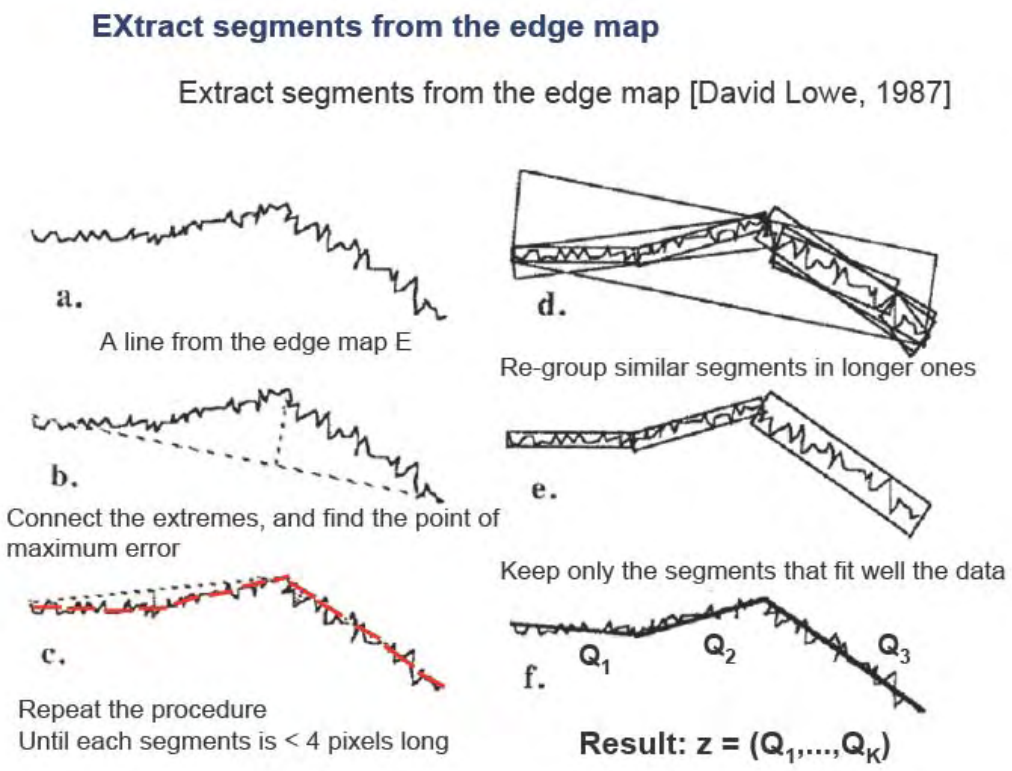

When we define a feature-level measurement for contour-based tracking, we need a further processing step.

In this case, the observed image features Z are a set of line segments, that we try directly to match to the model contour of the object.

In particular, after computing the Canny edge map, we need a procedure that joins individual, consecutive edge points into segments, and gives as output a list of geometric descriptors, for example $Q_i = (x,y,l,\theta)_i$, where x,y are the image coordinates of the center point, l is the length and θ the angle with respect to the x axis.

NOTE: in this case, edge points are joined into geometric structures (segments), therefore we are working on a different space, which we may call the segment space (x,y,l,θ).

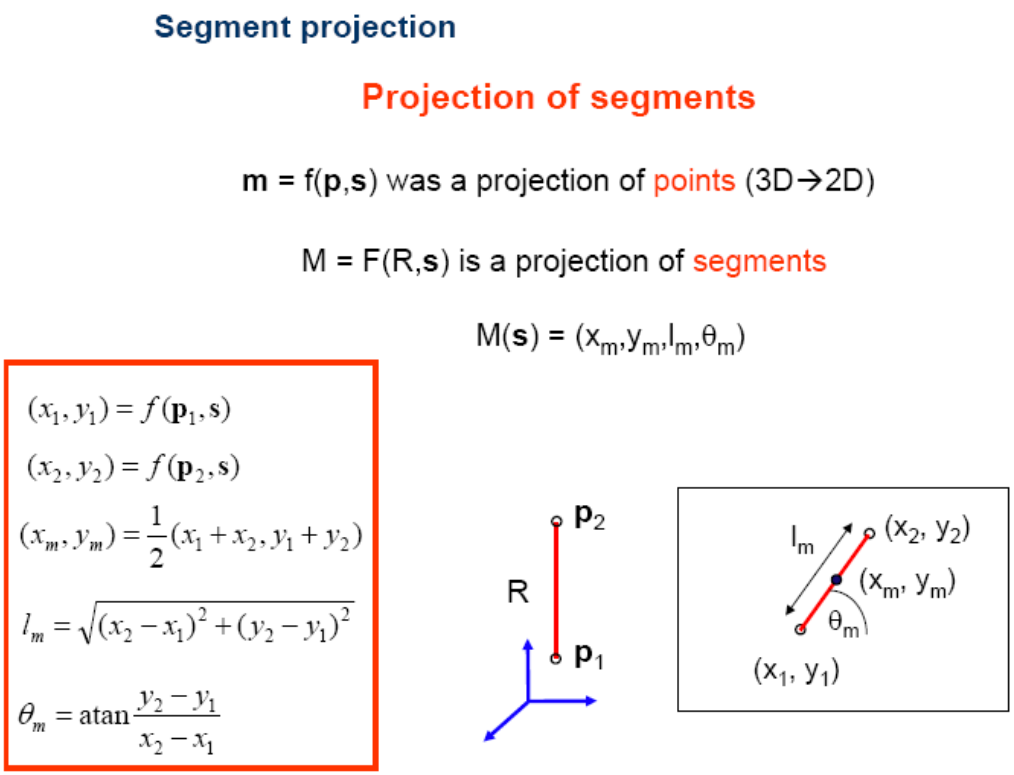
### *Extract image segments*




An algorithm for extracting line segments from an edge map has been done by David Lowe in 1987.

The procedure amounts to take a single, connected Canny edge line (a), connecting the extreme points and split it by finding the maximum distance point (b), into sub-sections, whch are in turn split again; this procedure (c) stops when all of the single segments are shorter than a threshold; then, the procedure merges together short segments into longer ones (d), such that the SSD difference between constructed segments and image edge points is minimized (e) (segments must fit the image points), and the result is a set of “optimal” segments Q that fit the image edge points (f).

***Define the segment projection (Warp)***




Now we have a set of features (segments) that we wish to match to the model contours.

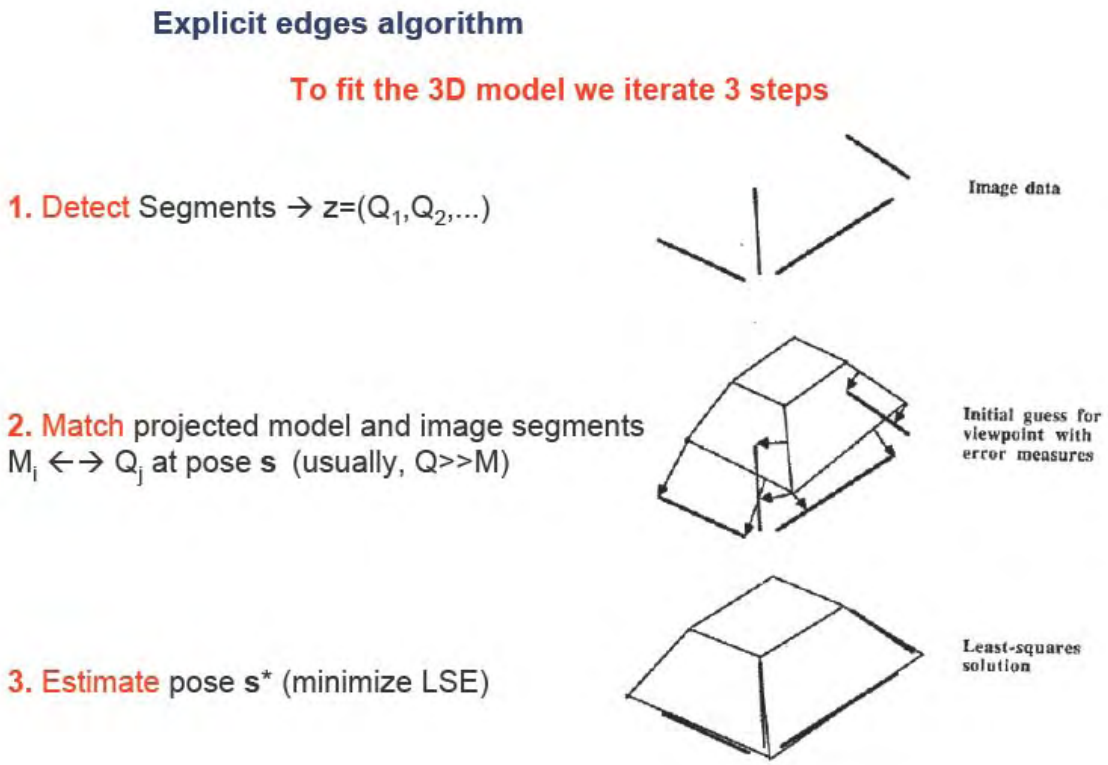
For this purpose, we first need to project model segments from 3D space to the image, by using our extrinsic+intrinsic projection function f(P,s), where P is a 3D body point and s is the roto-translation pose vector.

A model segment R is given by the extreme points in 3D space, $P_1$ and $P_2$.

If we project the points f($P_1$,s) and f(P2,s), we obtain two image points (x1,y1) and (x2,y2), and we can compute the middle point ($x_m$,$y_m$), the length l and the angle θ.

In this way, we have a mapping from the model segment R and the image segment Q, that we can globally represent as a nonlinear function Q=f(R,s).

**The segment-based pose estimation procedure**



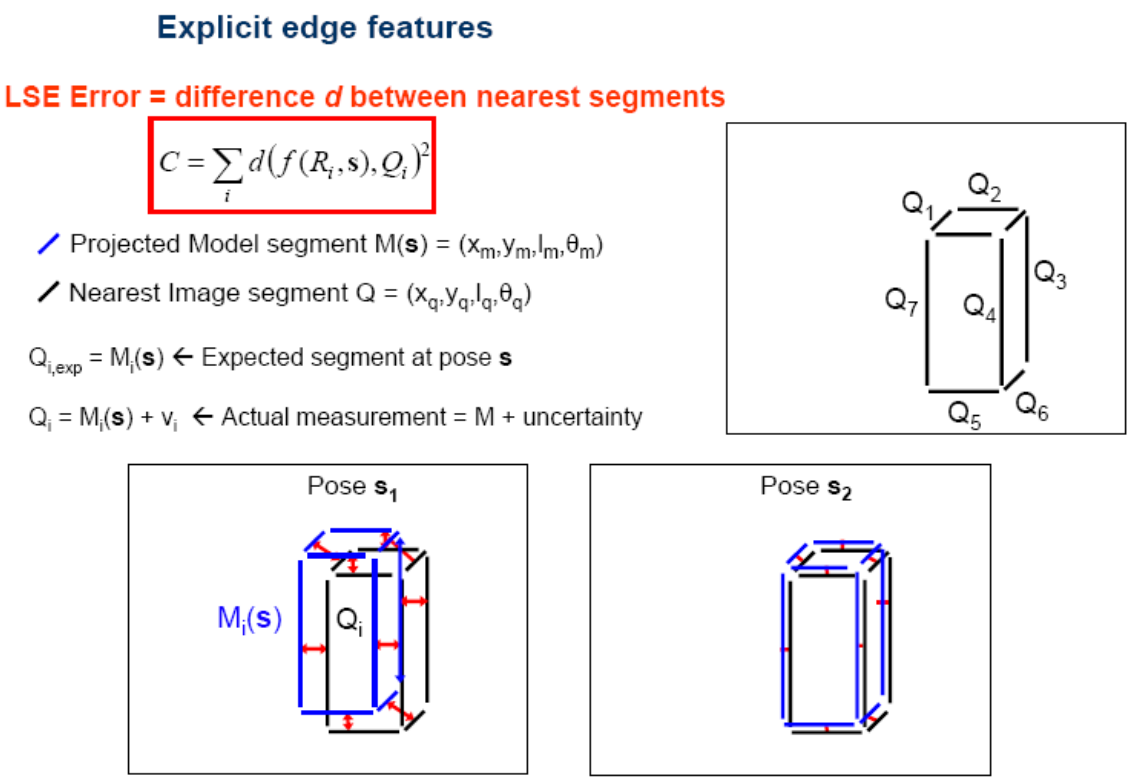

In order match segments, we also need a distance measure, between an expected feature $Q_{exp}$ and the observed one Q.

We will next define this distance measure, which is again a Mahalanobis distance.

The general procedure for estimating the object pose s is then:

1. Detect segments, to get the measurement $z=Q_1,...,Q_M$ (for example, using the Lowe's algorithm)
2. For a pose hypothesis s, match visible model and image segments by looking for the minimum distance (with the Mahalanobis distance for segments). Since usually the number of observed segments M is much greater than the visible model segments N, we can have ambiguities in the matching process (false matchings), that are solved during the pose optimization process
3. Estimate pose, by doing a Gauss-Newton optimization step

***Define the LSE error to be optimized: segment distances***




The LSE error to be optimized is, in this case, a sum of squared Mahalanobis distances.
If we do a single-hypothesis measurement, that is, we select only the nearest segment to each model segment for matching, then we have a nonlinear function+Gaussian noise model (zexp = h(s) + v), which can be inserted later on into an Extended Kalman Filter for tracking.

Now we do an object-level measurement, since we optimize the LSE error, to get the Maximum-Likelihood state estimate (maximum Likelihood for Gaussian noise = minimum SSD error).

## Distance measure

### Definition of distance between segments

The error is given by the difference between segment primitives

Model and image segments: $M_i = (x_{mi}, y_{mi}, l_{mi}, \theta_{mi})$ $Q_j = (x_{qi}, y_{qj}, l_{qj}, \theta_{qj})$

How can we define a distance between segments $d(M,Q)$ ?

We could use the usual Euclidean distance (in 4D)

$$d(M,Q)^2 = (x_m - x_q)^2 + (y_m - y_q)^2 + (l_m - l_q)^2 + (\theta_m - \theta_q)^2$$

But this is not good: the 4 descriptors have different meaning and importance!

The Mahalanobis distance between segments takes into account a different importance (weight) for the 4 different components (x,y middle point, length and angle).
This is better than the standard Euclidean distance, where the weights are the same, since the reliability of the middle point, for example, is more than the angle or the length, which instead are more uncertain because of the noise.

## Distance measure

### Mahalanobis distance between segments

→ Better: we use the Mahalanobis distance = Distance weighted by a matrix $\Lambda$

$$d(M,Q)^2 = (M-Q)^T \Lambda^{-1} (M-Q)$$

$\Lambda$ = (4x4) covariance matrix = uncertainty in the parameters of M and Q.

If $\Lambda$=I we have the standard Euclidean distance

For each pair $(M_i, Q_i)$ we use a different covariance matrix

$$\Lambda_i = \Lambda_{mi} + \Lambda_{qi}$$

→ If the segments are less reliable, the contribution to the total distance is lower

$\Lambda_i$ are pre-computed for every pair $(M_i, Q_i)$ before doing the optimization.

The covariance matrix of this distance, in turn, can be obtained as a sum of two matrices: the projected model segment and the detected image segment covariance matrices.

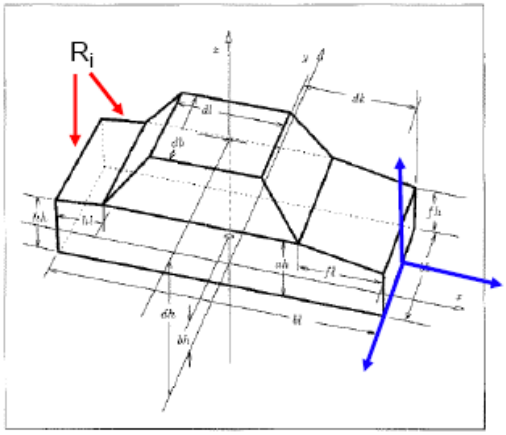
These matrices reflect the uncertainty in the respective parameters, and they can be even different for individual segments (for example, very short model segments have a higher uncertainty, therefore a higher covariance).

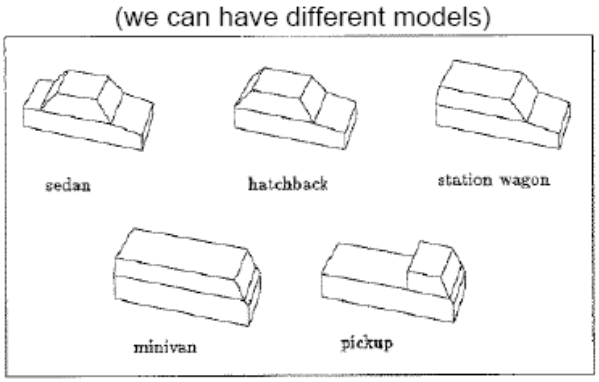
## Explicit edges algorithm

### Example : A car-tracking algorithm [Koller, Daniilidis, Nagel 1993]

Task: we want to track the 3D model of a car.

We represent the model as a polyhedral with 3D segments $R_i$ (i=1,...,K)
The visibility test is performed at each pose by simple geometric rules
(this model is quite simple, so we do not need OpenGL)



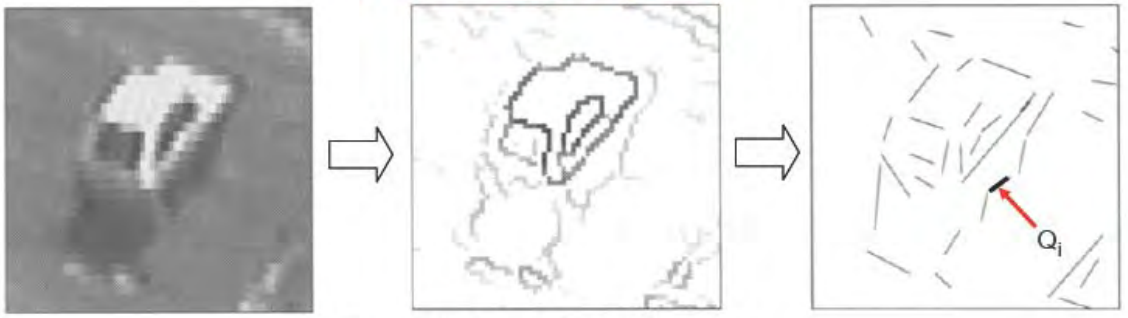




As an example of this methodology, apart from the already mentioned work by D. Lowe (1987) we can mention the paper by Koller et al. (1993), where a 3D car model is tracked using explicit edges.

The contour model consists only of segments, and at a given pose a visibility test based on standard geometric rules (no self-occlusions) is performed in order to determine the ones used for pose estimation.

## Explicit edges algorithm

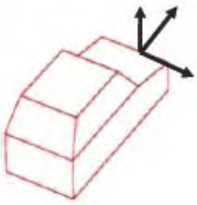
### STEP 1: Detect segments

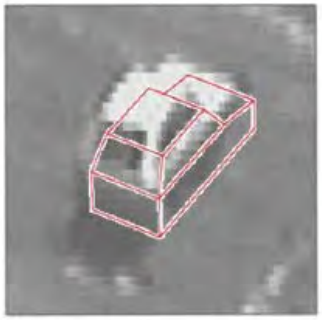



At the initial guess **s** of the pose

NOTE: Not all segments are correctly matched, if the pose is not correct

→ We need to refine the matchings at each optimization step





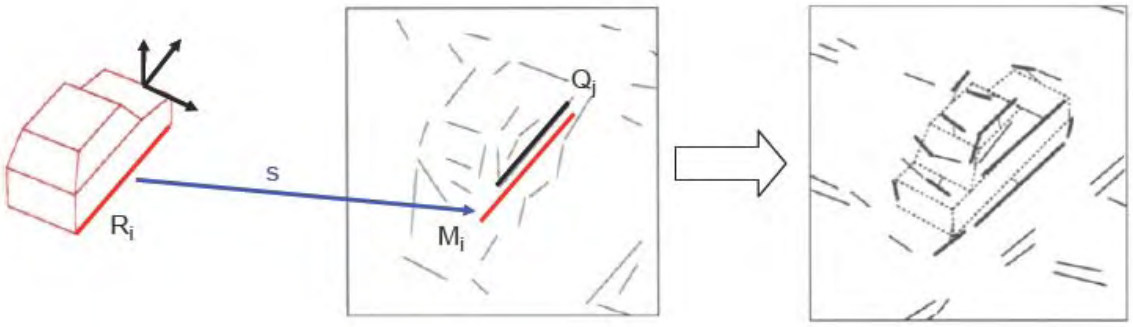
## Explicit edges algorithm

### STEP 2. Matching segments

For each projected model segment $M_i$ at pose s (hypothesis)

Match $M_i \leftrightarrow Q_j$ such that $Q_i = \arg\min_{Q} d\left(M_i(\mathbf{s}), Q_i\right)$

→ Find the image segment closest to the projected model segment




From the image, we can build the edge map and detect segments; at pose s, we match projected model segments f(R,s) with image segments Q, by searching for the nearest one (in the Mahalanobis metric).

If the initial pose guess s is not correct, of course we will match some wrong edges; therefore, the matching process must be repeated after each correction (Gauss-Newton optimization) step.

***The Gauss-Newton step for LSE optimization***

**Explicit edges algorithm**

**STEP 3. Minimize LSE**

The error at pose **s** (after matching) is

$$C = \sum_i d(M_i(\mathbf{s}), Q_i)^2$$

We can write it as a weighted LSE function

$$C = \left\| \mathbf{W}(\mathbf{M}(\mathbf{s}) - \mathbf{Q}) \right\|^2 \quad \mathbf{M}(\mathbf{s}) = \begin{bmatrix} M_1 \\ \cdots \\ M_K \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} Q_1 \\ \cdots \\ Q_K \end{bmatrix} \quad W = \begin{bmatrix} \sqrt{\Lambda_1} & 0 & 0 & \cdots \\ 0 & \sqrt{\Lambda_2} & 0 & \cdots \\ 0 & 0 & \sqrt{\Lambda_3} & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

**W** is the weight matrix : it contains all the covariances $\Lambda_i$.

→ We can use weighted Levenberg-Marquardt

$$\Delta\mathbf{s} = \left(\mathbf{J}^T\mathbf{W}\mathbf{J} + \lambda\mathbf{I}\right)^{-1}\mathbf{W}\mathbf{J}^T(\mathbf{Q} - \mathbf{M}) \qquad \mathbf{s} \rightarrow \mathbf{s} + \Delta\mathbf{s}$$

**J** = ∂**M**/ ∂**s** ← Jacobian matrix (4K x 6) of the segment projection

LSE minimization uses the same Mahalanobis distance, in order to compute the SSD error.

This can be re-written in a weighted, standard LSE form, if we build a weight matrix W that contains the covariance matrix of the Mahalanobis distance between segments ($\Lambda_i$).

Therefore, we can do a weighted Levenberg-Marquardt step (=Gauss-Newton, plus the correction factor $\lambda I$).

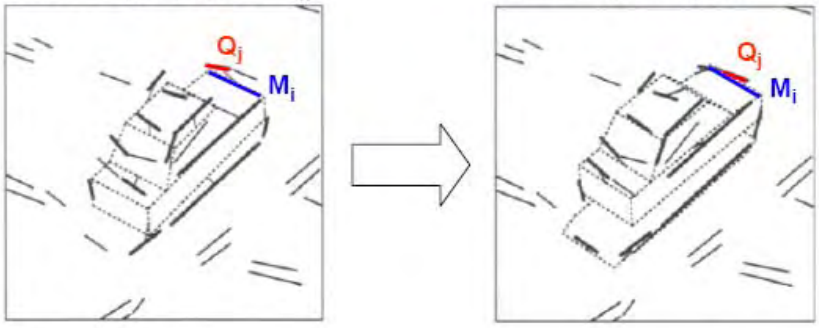
In order to perform LM, we need to calculate also the Jacobian of the mapping between 3D and 2D model segments, that we defined earlier.

This is a (4x6) matrix for each of the K visible model segments, so that we have a (4Kx6) total Jacobian matrix.

## Explicit edges algorithm

After Levenberg-Marquardt, re-compute the matchings (Step 1)

Matchings at pose $s_k$ | Matchings at pose $s_{k+1}$

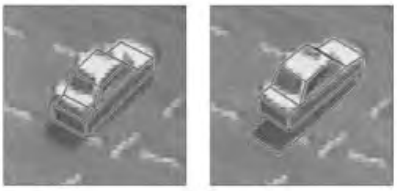



Stop the estimation loop:

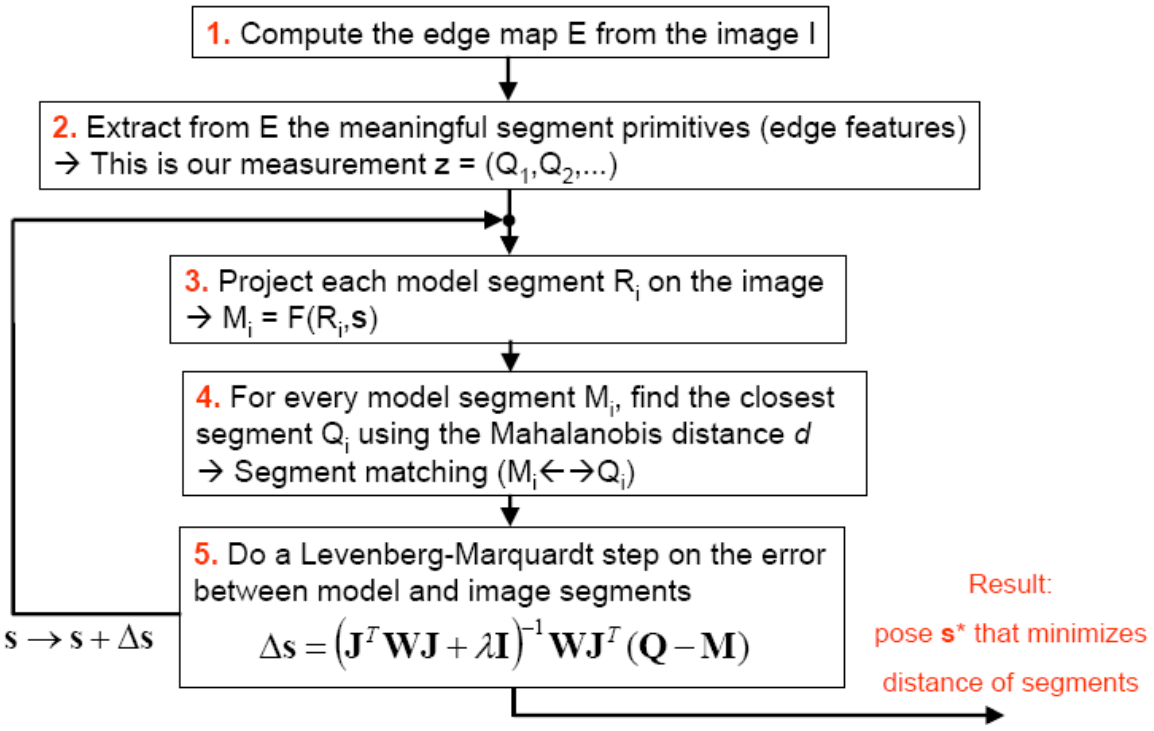
A. If the final LSE error is C < Threshold

OR

B. If the matchings ($M_i \leftrightarrow Q_j$) do not change anymore



## Pose estimation algorithm with explicit edges




After a single LM optimization step, we get a new pose hypothesis $s_k \rightarrow s_{k+1}$, which should be closer to the correct value; then, we can refine also the matching between model and image segments, by searching again the minimum distance segments with the Mahalanobis metric.

The algorithm stops when the pose increment $||s_{k+1}-s_k||$ is below a threshold, or when at the next step, the correspondences between segments M←→Q do not change anymore.

***Comparison between segment and edge map for tracking***

**Conclusion: RAPiD vs. Explicit edges**

**Edge map (RAPiD)**

- Advantage: the measurement **z** = E is cheap (only the Canny detector)
- Advantage: the matching is safer (many model points)
- Disadvantage: LSE optimization is costly (many model points) $C = \sum_i \left\| \mathbf{m}_i(\mathbf{s}) - \mathbf{q}_i \right\|^2$

**Explicit edges**

- Disadvantage: the measurement **z** = $(Q_1,...,Q_K)$ is costly (extract segments)
- Disadvantage: the initial matching is not safe (many similar segments Q against few model segments in the database) → False matches
- Advantage: LSE optimization is cheap (a few model segments) $C = \sum_i d(M_i, Q_i)^2$

A comparison between pixel-level and feature-level contour tracking (RAPiD vs. explicit edges).

RAPiD requires less image processing (only the edge map is needed) and uses many edge points from the model. Therefore, it is more flexible and more robust with respect to noise: if a single model point from a segment is mismatched, the contribution of this error is smaller, when compared to a mismatch between entire segments (second method).

A disadvantage in terms of computational complexity is the use of a larger Jacobian matrix (many points). Instead, explicit edges require more processing, to get the segments Q from the edge map (Lowe's algorithm), and give also a more compact information, that can be more sensitive to noise.

An advantage is of course the smaller dimension of J, which gives a faster optimization time.

***Bayesian estimation with dynamic models***

## Final note: Bayesian tracking

### Bayesian improvement

Both algorithms can be plugged-in a Bayesian predictor/corrector (e.g. Kalman)

Remember: **z** is the measurement, and P(**z**|**s**) is the Likelihood (Gaussian)

- RAPiD: P(**z**|**s**) = Gauss($\mathbf{z}_{exp}$,C)
where $\mathbf{z}_{exp}$ are the expected edge points positions, $\mathbf{m}_i$
→ all put in a big vector **m** → P(**z**|**s**) = Gauss(**m**,C)

- Explicit edges: P(**z**|**s**) = Gauss(**M**,C)

But of course, nobody forbids to use **z** = **s*** (the pose estimation after the full algorithm)! → This is a "Type 3" measurement!

Then, in both cases we can also use P(**z**|**s**) = Gauss(**s***,C)

After this choice, we can design the proper Kalman Filter.
(And of course, we have to choose the right covariance matrix C!)

In order to implement Bayesian tracking, both methods are suitable for KF or EKF implementations.

In particular, RAPiD can be used in an EKF context, if **instead** of the Gauss-Newton optimization we directly use Z (the edge map) in order to compute the Likelihood P(z|s), by putting together all of the projected model points $m_i$ into a big vector **m** (=expected measurement, $z_{exp}$), the corresponding nearest image edge points in another vector **q**(= observed z) and measure the Likelihood as a Gaussian (z-$z_{exp}$(s)) with a given covariance matrix.

For an EKF implementation, of course we need also the Jacobian of $z_{exp}$(s), which is the Jacobian of the usual body-to-screen projection f(P,s).

In this case, we can say that we use again a feature-level measurement z, where the features are the nearest-neighbor edge points to the model points, in the edge map.

The same can be done for explicit edges, in this case in segment space (features=segments), with the proper Jacobian matrix.

Another choice for both algorithms is instead to perform the LSE optimization, and use the estimated pose s*=z as an object-level measurement.

In this case, a standard Kalman Filter can be implemented.

NOTE: whatever filter we implement, we must be careful to put the "right" covariance matrix C of the measurement noise!

## *Lecture 10 – Contour tracking using Likelihood functions*

### Representation of curvilinear shapes with B-Splines

**Representing a curve**

A parametric curve in the plane is represented by

$$\mathbf{c}(l) \rightarrow \big(x(l),\, y(l)\big)$$
$$0 \le l \le L$$


We are interested in curves that are

- Complex (can be of any shape we want)
- Smooth
- Computationally cheap

This can be achieved by connecting many small polynomial (low degree) curves: the B-Splines

A parametric curve model can represent all points along a regular curve, in plane or 3D space, by a scalar parameter *l* (curvilinear coordinate) that runs in a continuous way from 0 (start point) to L (end point) and represents the intermediate curve length.

One of such representations is given by the uniform B-spline, which provide a smooth and differentiable curve, in order to model different and eventually complex shapes with relatively low effort.

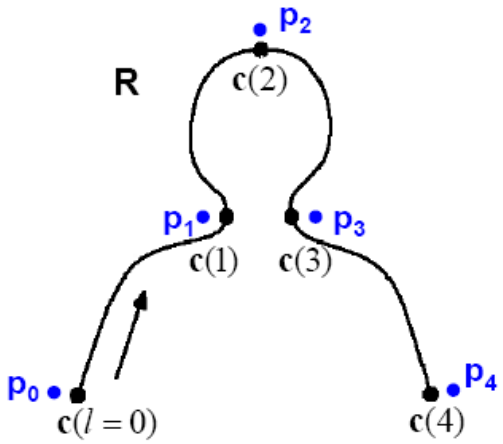
## Describe curves: B-Splines

B-Spline = a curve parametrized by a set of control points

R = $(\mathbf{p}_0,\mathbf{p}_1,\ldots,\mathbf{p}_L)$;

A B-Spline is defined by: $\mathbf{c}(l) = \sum_{i=0}^{L} \mathbf{p}_i B(l-i)$

- A Basis function B(l)
- The control points $\mathbf{p}_i$




l is a continuous parameter (curvilinear coordinate) running from 0 to L
The control points are approximated at l=0,1,...,L

A uniform B-spline is represented by a set of control points, in plane or space; the locations of these points provide the shape of the curve, and every intermediate point c(l) is obtained through a linear combination of the control points, through the B-splne basis function B(l).

If there are L control points, and we use them directly as basis coefficients (as in the formula above), then at integer curvilinear coordinates l=0,1,...,L the curve approximates (but not exactly interpolates!) the control points.
If we need an exact interpolation of the points, then the B-spline coefficients are a bit more complex to compute, for which we need to solve a system of linear equations.

As we can see, in order to model the shape we need to specify both the Basis function (a piece-wise polynomial function) and the control points.

***B-Spline basis functions***

**Spline basis functions**

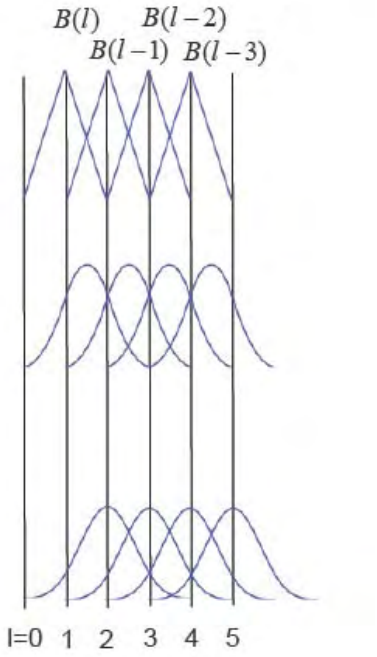
Basis Spline functions B(l) = a piece-wise polynomial of degree n (n=1,2,3,...)

Example of Basis Spline functions:

Degree 1 (Linear) $$B(l)=\begin{cases} l, & 0<l<1 \\ 2-l, & 1<l<2 \end{cases}$$

Degree 2 (Quadratic) $$B(l)=\begin{cases} \frac{1}{2}l^2, & 0<l<1 \\ -l^2+3l-\frac{3}{2}, & 1<l<2 \\ \frac{1}{2}l^2-3l+\frac{9}{2}, & 2<l<3 \end{cases}$$

Degree 3 (Cubic) $$B(l)=\begin{cases} \frac{1}{6}l^3, & 0<l<1 \\ -\frac{1}{2}l^3+2l^2-2l+\frac{2}{3}, & 1<l<2 \\ \frac{1}{2}l^3-4l^2+10l-\frac{22}{3}, & 2<l<3 \\ -\frac{1}{6}l^3+2l^2-8l+\frac{32}{3}, & 3<l<4 \end{cases}$$

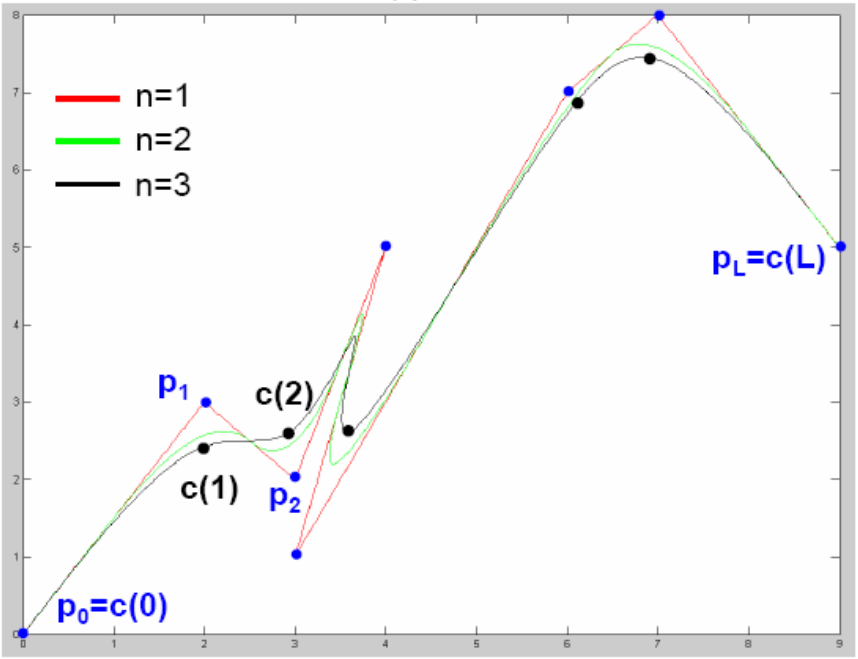



Basis functions are given by piece-wise polynomials of degree n, and they have a limited support of n+1. For example, the linear function covers only two unit intervals for l, the second degree only three, and so on. They also provide a n-1 differentiability: linear b-splines are continuous but not differentiable (0-order differentiability), quadratic splines are 1-st order differentiable, and so on.

The first three basis functions are the most used.

## Example with different basis functions

$$\mathbf{c}(l) = \sum_{i=0}^{L} B(l-i)\mathbf{p}_i$$




As the example shows, for the same set of control points we have an increasing quality of approximation for increasing degree; but for a too high degree, we instead have a worse curve, with a “spiky” behaviour, because of too many degrees of freedom of the curve (in the approximation theory this is a well-known fact).

Therefore, a quadratic or cubic spline is usually the best choice, which give a 1-st or 2-nd order differentiability.

***Properties of B-Splines***

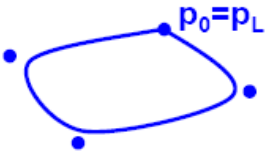
### B-spline properties

If we desire to start exactly from the first point, and end on the last, we need to duplicate them:

R = ($\mathbf{p}_0,\mathbf{p}_0,\mathbf{p}_1,\ldots,\mathbf{p}_L,\mathbf{p}_L$)

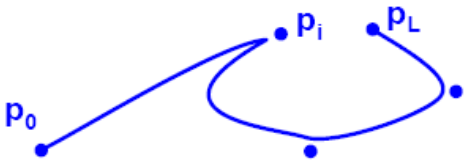

If we model a closed curve, then the last control point is the same as the first (and duplicate!)

R = ($\mathbf{p}_0,\mathbf{p}_0,\mathbf{p}_1,\ldots,\mathbf{p}_{L-1},\mathbf{p}_0,\mathbf{p}_0$)




If we need a „broken“ point in $\mathbf{p}_i$, we have to duplicate it:

R = ($\mathbf{p}_0,\ldots,\mathbf{p}_{i-1}$, $\mathbf{p}_i$, $\mathbf{p}_i$, $\mathbf{p}_{i+1},\ldots$)




A “broken point” is given by a discontinuity in the first derivative, in order to model a “corner” in the object contour.

### Normal to a B-Spline

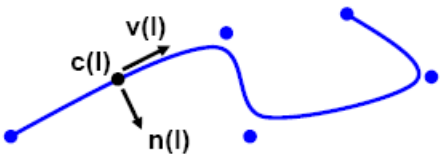
The normal direction to a B-spline curve in a point t, is easily computed:

The tangent direction is the derivatve d**c**/dt:

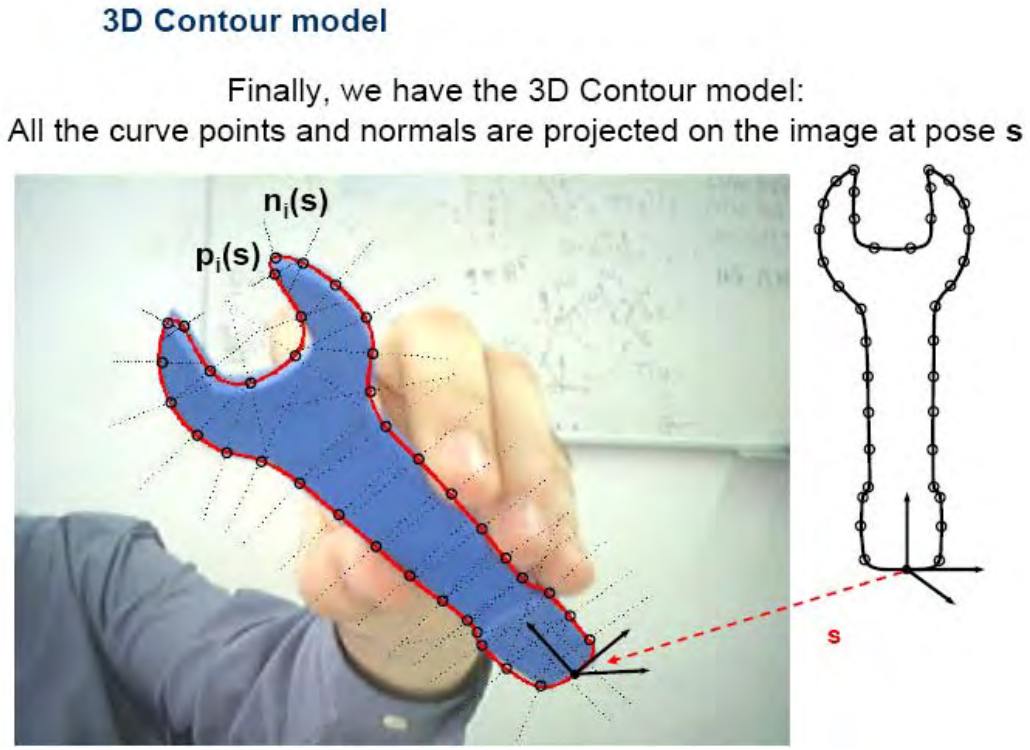
$$\mathbf{v}(l) = \frac{d}{dl}\sum_{i=0}^{L}\mathbf{p}_i B(l-i) = \sum_{i=0}^{L}\mathbf{p}_i B'(l-i)$$

Example : quadratic splines $B'(l) = \begin{cases} l, & 0<l<1 \\ -2l+3, & 1<l<2 \\ l-3, & 2<l<3 \end{cases}$

The normal direction is **n**(l) = [-$v_y$(l),$v_x$(l)]



For contour tracking we usually need also to compute the normal direction to the contour line, that in case of B-splines can be obtained by using the derivative of the basis function, and the same coefficients used for the curve points.




By using 3D control points in a reference frame (body frame) we can model the object contour, and project directly the control points onto the image, with the projective camera transformation used so far.

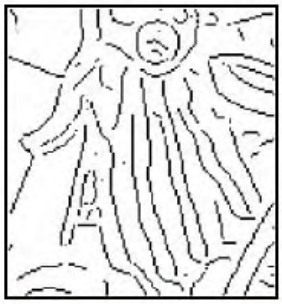
At pose s, we obtain therefore any curve point, and its normal direction, both projected onto the image plane.

**Multi-modal contour Likelihood**

**Need for multi-modal Likelihood contribution at each point**

Last time, we saw how to minimize a contour LSE error using the edge map E and model contour points (the RAPiD algorithm).

Question: How can we better deal with false (neighboring) edges in general?


We need to define a Likelihood function with multiple hypothesis: P(**z|s**) must be a multi-modal distribution.

Multi-modal = a function with more peaks (modes), related to different edge hypotheses.

Edge models often contain ambiguous information: in this example, it is very difficult from the edge map to distinguish between real finger edges and shadows.

This happens because of the low distinctive properties of edges: a straight edge is just a line segment, with the only information concerning length and orientation; therefore, parallel lines are almost impossible to distinguish between one another. In contrast, local keypoints (e.g. SIFT) carry a very specific distinctive property, given by their descriptor over a pixel window.

For edges, a multiple likelihood model is better suited: if we make a hypothesis on the state S (picture on the right, we may have multiple hypotheses for the corresponding measurement (edge pixel) to each contour point along the normal.

Only one can be the “true” edge, while the others are shadows, or different edges. In the tracking literature, we talk about “target-generated” measurement, as opposed to “false alarms” or “clutter” measurements.

The problem of finding the correct measurement and associating it to the real state S is also called “data association” problem: which edge points actually correspond to the object, and which are false alarms?

If we just take the nearest edge hypothesis, we may do the wrong association; a better idea is to keep all of the measurements, and give them a weight (association probability) that tells the probability for each edge point to have been generated from the real target or not.

This is a multiple hypothesis measurement, and in presence of clutter situations (many false alarms), it is much more robust than the nearest-neighbor approach, where the nearest is associated with probability 1 to the target, and the others with probability 0.

Of course, in this case we do not have a single Gaussian anymore for P(z|s), but rather a multi-modal distribution (multiple peaks).
Therefore, we cannot use Kalman or EKF anymore, but we can instead use Particle Filters.

## Multi-modal Likelihood

The Likelihood function answers to the question: what is the probability that the observed edge map z=E was generated by the contour hypothesis **s**?

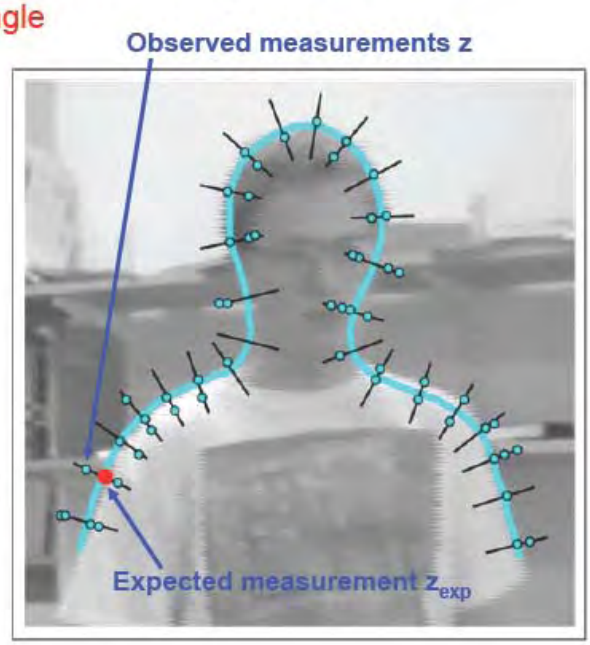
If the pose were **s**, we would expect a single contour, located exactly on the blue line

The measurement function is of the kind

$$\mathbf{z} = h(\mathbf{s},\mathbf{v})$$

$\mathbf{z}_{exp} = h(\mathbf{s},0)$ is an edge map with only one edge on the projected silhouette

**v** is a multi-modal noise (non-Gaussian!) which gives rise to multiple edges where only one can be the true one, or no-true edge at all!




A multi-modal likelihood can be modeled by a function h(s), which tells the expected measurement (expected contour, or edge map) plus a noise vector which is not Gaussian but multi-modal, which models the presence of multiple measurements in the vicinity of the projected curve.

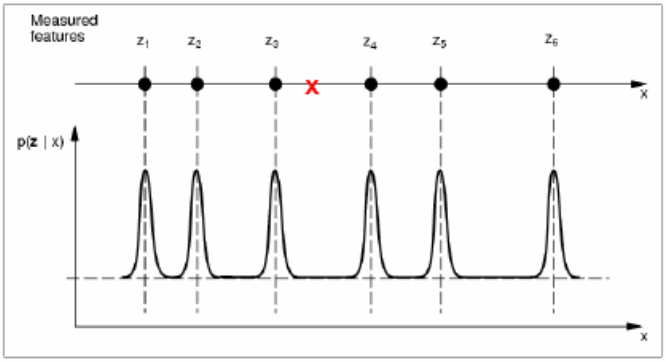
As the example shows, there can be one, no one, or multiple associated measurements for each contour position (expected measurement).

***Multiple-hypothesis edge measurement***

**Multiple hypotheses measurement**

A simple case: mono-dimensional observation with multiple hypotheses.
The state is x, a 1D coordinate, and z is also a 1D measurement.
The expected measurement is $z_{exp} = x$ ← We expect only one measurement.

But the measurement instrument gives M possible answers $(z_1,...,z_M)$




Only one can be the „true“ one, the others must be false measurements.
If we do not know it, then each $z_m$ has a probability $\phi_m$ to be the true one
And there is also the possibility that all $z_m$ are false → false alarm case

In order to understand this likelihood model, we consider first a mono-dimensional case: the state hypothesis (x) is also the expected measurement, while our “measurement instrument” gives several output hypotheses.

Only one can be target-generated, or eventually no one (missing detection), but no more than one; the remaining ones must be false alarms (clutter).

If we have no clue a-priori, we must assume all $z_i$ to have the same probability $\phi$ of being the true one.
If all $z_i$ are false, then we have the missing detection case, with probability q.

## Multiple hypotheses measurement

$\phi_m$ = {probability that the true measurement is $z_m$}

$q = 1 - \sum_m \phi_m$ = probability that all $z_m$ are false

If the true measurement were $z_m$, then the Likelihood is (for example) a Gaussian around $z_m$, with covariance $\sigma$: Gauss(x,$z_m$,$\sigma$)

$$P(z \mid x, \phi_m) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left( -\frac{(x - z_m)^2}{2\sigma^2} \right)$$

If all $z_m$ are false, then we have no guess about the expected measure
→ uniform (constant) probability for all x

Put all together: the overall Likelihood is a sum of Gaussians (each one taken with probability $\phi_m$)

$$P(z \mid x) \propto q + \sum_m \phi_m P(z \mid x, \phi_m)$$

If $\phi_m$ are all the same (1/M), we have $$P(z \mid x) \propto q + \frac{1}{\sqrt{2\pi}\sigma M} \sum_m \exp\left( -\frac{(x - z_m)^2}{2\sigma^2} \right)$$

The true measurement is distributed as a Gaussian around the expected measurement = x (standard model), while false alarms are instead uniformly generated in the visual field.

The result is a measurement model which is a sum of Gaussians around each $z_i$, weighted by the probabilities of $z_i$ being the true one, plus a constant (uniform) noise term for the false alarms. This is a multi-modal distribution P(z|x).

## Contour Likelihood function

Go back to contour tracking:

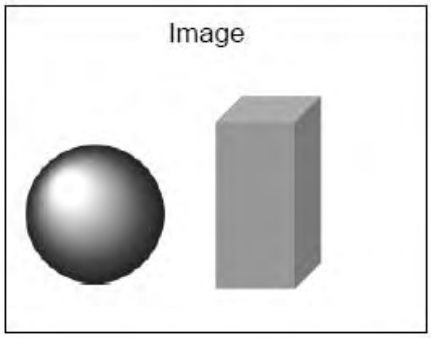
The measurement is the full edge map: **z** = E
An edge map is the set of all edge points (connected into sequences)

For a pose hypothesis **s**, take a contour point **m**. Along the normal, we find a lot of edge points, and we can do a multiple-hypothesis measurement.

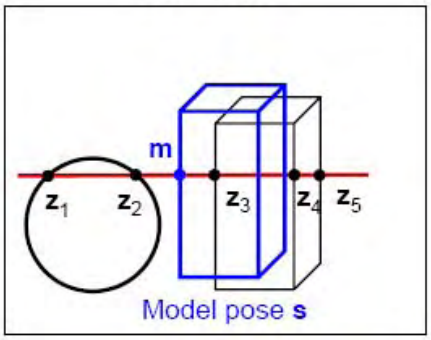
Example: Edge measurements are ($\mathbf{z}_1,...,\mathbf{z}_5$)

$$P(\mathbf{z} \mid \mathbf{m}) \propto q + \lambda \sum_m \exp\left( -\frac{\|\mathbf{m} - \mathbf{z}_m\|^2}{2\sigma^2} \right)$$

At a distance $||\mathbf{m}-\mathbf{z}_m|| >> \sigma$
we can approximate $\exp(\mathbf{m}-\mathbf{z}_m) \sim 0$
→ We exclude far points (e.g. $\mathbf{z}_1,\mathbf{z}_4,\mathbf{z}_5$)




Edge map E




By going back to the 2D contour, we can use the formula before developed for each contour position, along the respective normal. In this case, zi are observed edge points along the normal, in the vicinity of the edgehypothesis.

For computational simplicity, only the points within a distance of 2-3σ are considered, since far points do not contribute to the Likelihood (Gaussian is almost 0).

***The complete contour Likelihood***

**Contour Likelihood function**

If we do the same for all model points, assuming they are independent one another, we have the global contour Likelihood at pose **s**

$$P(\mathbf{z}\,|\,\mathbf{s}) = \prod_{i=1}^{N} P(\mathbf{z}\,|\,m_i)$$

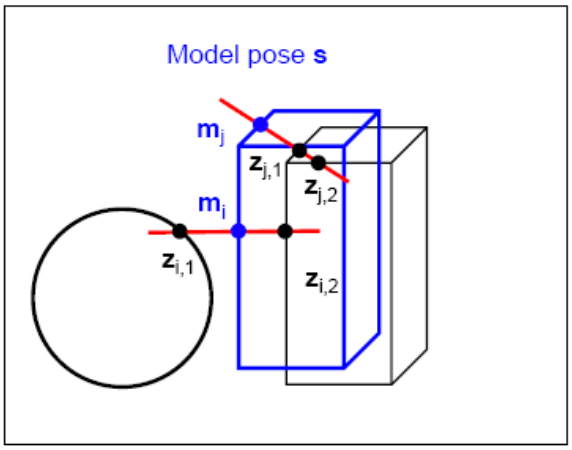
That is:

$$P(\mathbf{z}\,|\,\mathbf{s}) \propto \prod_{i=1}^{N}\left(q + \lambda \sum_{m} \exp\left(-\frac{\|\mathbf{m}_i - \mathbf{z}_{i,m}\|^2}{2\sigma^2}\right)\right)$$

$\mathbf{z}_{i,m}$ are points on the edge map correspoinding to $\mathbf{m}_i$

This function maps every edge point on the map to one or more model points

Edge map E




→ It is a complete Likelihood P(E|s), covering all the edge map E.

By considering the measurement process to be independent for different contour positions, we can multiply individual contributions together, and obtain the overall Likelihood of the contour P(z|s).

This Likelihood in principle (if the number of sample position goes to infinity) should cover every edge point, therefore being a complete P(E|s), where E=Z is our “pixel-level measurement”.

In practice, since we use a finite number of positions, it is an approximation, but sufficient for our tracking purposes.

## Contour Likelihood function

We can also approximate this function by considering only the nearest edge **z**:

Single hypothesis $\mathbf{m}_i \leftrightarrow \mathbf{z}_i$ (like RAPiD)

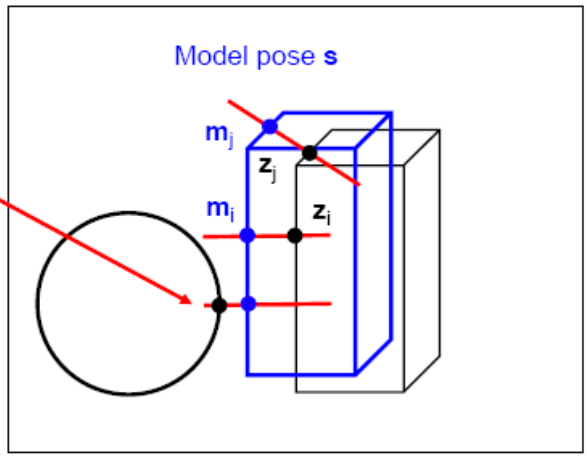
$$P(\mathbf{z} \mid \mathbf{s}) \propto \prod_{i=1}^{N} \exp\left( -\frac{\|\mathbf{m}_i - \mathbf{z}_i\|^2}{2\sigma^2} \right) = \exp \sum_{i=1}^{N} \left( -\frac{\|\mathbf{m}_i - \mathbf{z}_i\|^2}{2\sigma^2} \right)$$

This is a weaker model:

The nearest edge can be a wrong one, and we discard other hypotheses, which may include the true one!

But also a cheap model:

The Likelihood is a Gaussian




If we would use, instead, the nearest-neighbor contour likelihood (the same used for RAPiD pose optimization), we have a cheaper model (actually a Gaussian model), but much less robust to cluttered background, noise, and other object edges.

In this case, we can use an Extended Kalman filter, since the model is z = h(s)+v, where h is nonlinear (3D to 2D projection of contour points).

***Particle Filters for contour tracking – the CONDENSATION approach***

**Non linear and non Gaussian measurement → CONDENSATION**

**How can we use contour Likelihood to do Bayesian tracking?**

The expected measurement is the non-linear 3D/2D projection of points from model contour to screen.

→ We cannot use Kalman filter

We could use EKF, only if the uncertainty (noise) were a unimodal Gaussian.

But the uncertainty is multi-modal! (Multiple edge hypotheses)

So, here we definitely need Particle Filters (CONDENSATION)

Particle Filters for 3D contour tracking

- The model has state **s**, given by the 3D pose of the object.
- The motion model can be standard Linear+Gaussian (e.g. a WNA model)
- The measurement model is the contour Likelihood (non-linear/non-Gaussian).

Particle Filters can be used with this Likelihood function, which is nonlinear and non-Gaussian.

## CONDENSATION for contour tracking

### Particle Filters applied to contour tracking

A particle is a state hypothesis+weight $(\mathbf{s}_t^{(k)}, \pi_t^{(k)})$

For example, if the pose is: **s** = (x,y,w)

(x,y) = center of contour, w = size
(The true state is actually [**s**,d**s**/dt])

A particles set at time t is a set
of contour hypotheses, each one with a weight:

$$\pi_t^{(k)} = P(\mathbf{z} \mid \mathbf{s}_t^{(k)})$$

The weight is the Likelihood, and the particles are sampled from the Prior probability at time t:

$$P(\mathbf{s}_t|Z_{t-1}) \rightarrow (\mathbf{s}_t)^{(1)}, (\mathbf{s}_t)^{(2)}, \ldots, (\mathbf{s}_t)^{(N)}$$

In this example, a set of particles is a set of contour hypotheses, and the weight is P(Z|S) which is represented here with the contour thickness.
High Likelihood hypotheses are distributed around the correct one, where most edges are observed near to the projected contour.

## Particle representation of contour posterior

Factored Sampling Theorem: the Particles represent the posterior $P(\mathbf{s}_t|\mathbf{z}_t)$

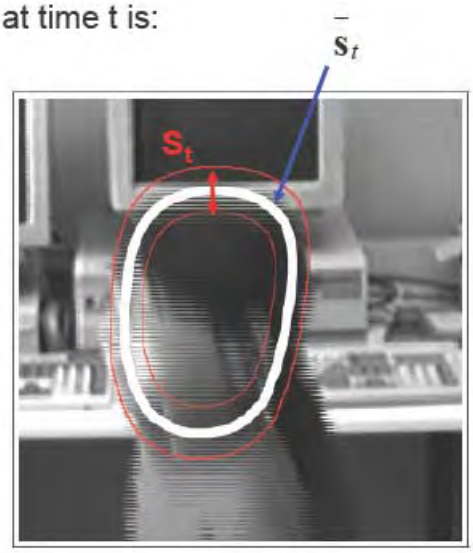
→ So, we can get from the particles set $(\mathbf{s}_t^{(k)}, \pi_t^{(k)})$ every information we need.

For example, the average (mean value) state at time t is:

$$\bar{\mathbf{s}}_t = \sum_{k=1}^{N} \pi_t^{(k)} \mathbf{s}_t^{(k)}$$

and the uncertainty (covariance matrix) is:

$$S_t = \sum_{k=1}^{N} \pi_t^{(k)} (\mathbf{s}_t^{(k)} \mathbf{s}_t^{(k)T} - \bar{\mathbf{s}}_t^{2})$$

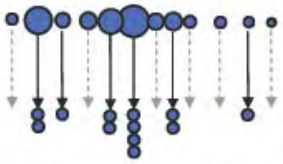



With the particle representation, we can compute the average value of the output distribution, and the covariance matrix as well, in order to monitor the tracking quality.

## Sketch of Condensation

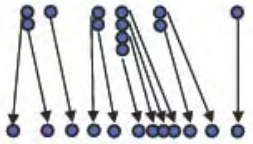
### CONDENSATION Tracking - At time t

**1.** Re-sample the old particles set $\mathbf{s}_{t-1}^{(k)}$ with probabilities $\pi_{t-1}^{(k)}$
→ re-sampled particles $\mathbf{s}'^{(1)}_{t-1}, \mathbf{s}'^{(2)}_{t-1}, \ldots, \mathbf{s}'^{(N)}_{t-1}$



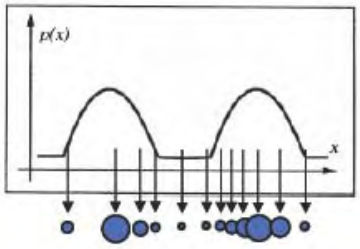
**2.** Move the particles $\mathbf{s}'^{(k)}_{t-1}$: simulate random motion by generating random values $\mathbf{w}_t$, and moving $\mathbf{s}_t^{(k)} = g(\mathbf{s}'^{(k)}_{t-1}, \mathbf{w}_t) = A\mathbf{s}'^{(k)}_{t-1} + B\mathbf{w}_t$

(Motion model: for example, a WNA model)



**3.** Compute the contour Likelihoods, and re-weight the new particles: $\pi_t^{(k)} = P(\mathbf{z} \mid \mathbf{s}_t^{(k)})$

$$P(\mathbf{z} \mid \mathbf{s}) \propto \prod_{i=1}^{N} \left( q + \lambda \sum_{m} \exp\left( -\frac{\|\mathbf{m}_i - \mathbf{z}_{i,m}\|^2}{2\sigma^2} \right) \right)$$




This resumes the particle filtering scheme, applied to contour tracking (see Lecture 5).

## CONDENSATION - features

### CONDENSATION for contour tracking

Advantage: it is very robust and flexible

- It resists to clutter, false measurements etc. because it can keep multiple hypotheses into account (Likelihood P(z|s))

- It can model also complex motion or shapes because of the generality of Particle Filters

Disadvantage: it is hardly real-time…

- The required sample size N (number of particle hypotheses) grows **exponentially** with the state dimension!
→ In 3D tracking, the state is 6-dimensional → We need N=$10^6$ or so

- The multiple-hypotheses Likelihood by itself is a complex function to compute.

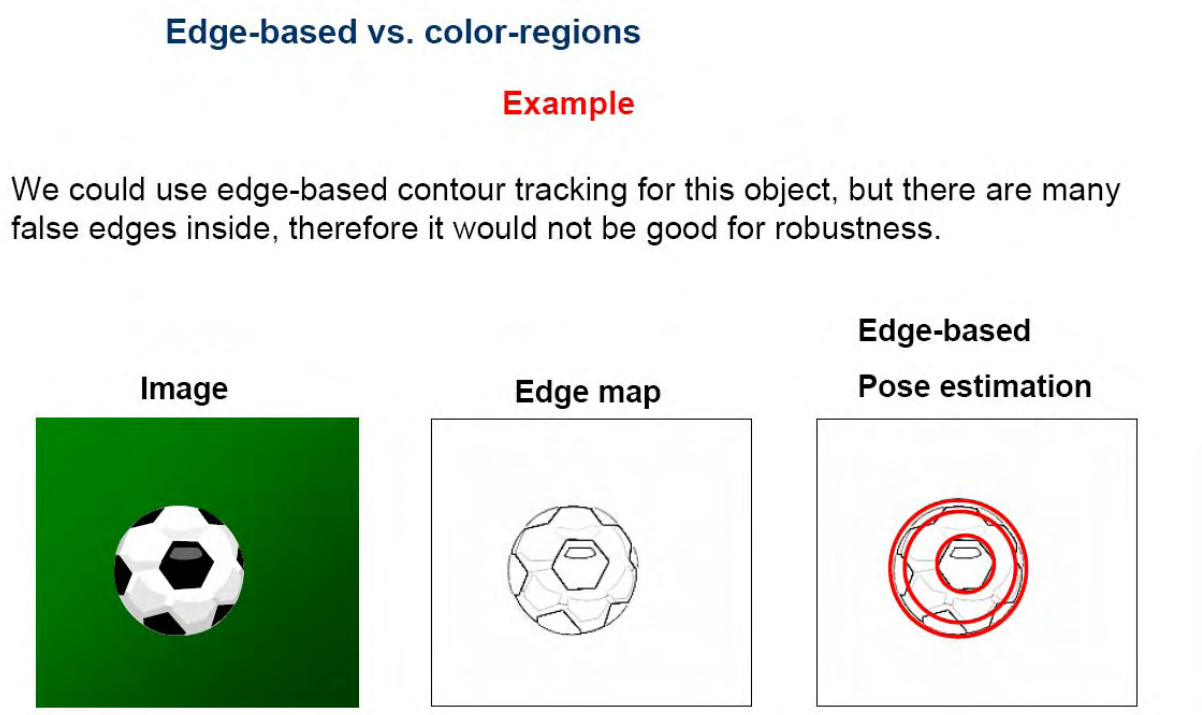
Particle Filters have the advantage of being flexible to nonlinear motion and/or measurement models, but they suffer from computational complexity, because the required number of particles to represent the posterior P(s|z) grows exponentially with the state dimension.

Therefore, for 3D tracking problems they offer still a challenging situation.

There are several ways to improve this approach: one is given by the Importance Sampling method (ICondensation) which allows introducing a complimentary low-level visual modality (e.g. color blobs) that helps to focus the sampling process, and allows to reduce the number of particles.
Another possibility is to distribute the computation between parallel processing units, since each particle can be processed (Likelihood and Motion) independently from the others.

### Contour tracking using color region statistics

#### *Motivation of the main idea*




In some situations, there are many false edges inside the object, that therefore is difficult to track using the edge map.

Nevertheless, we still wish to use the contour model of the object (in this case a circle) for tracking, since it is visually well-identified, and at the same time is very simple (no 3D information).

A human can well match by sight the position of the ball in the image, since as we can see the contrast between color distributions inside vs. outside is very sharp.

Therefore, we can obtain another way of matching contours to images, based on color statistics separation instead of matching intensity edges (like the edge map above).

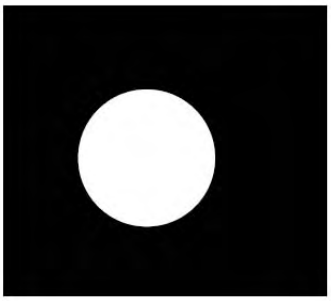
## Color statistics separation vs. edge-based tracking

Now the idea is pretty different: we try to find the contour not looking for strong edges (i.e. local transitions), but maximizing separation between regions (inside vs. outside).

This is like image segmentation: in a segmentation process, we try to isolate perceptually meaningful parts of the image.

Perceptually meaningful = regions that are meaningful to human sight.

We do this, because many objects (or environment parts) can be better distinguished by looking at the color distributions inside and outside.



## Segmentation-based contour estimation

**Example**

The ball inside is black/white, outside we have green.

→ We can search for the pose that maximizes separation between color statistics (i.e. segmentation)

Bad contour hypothesis:

- Inside : black/white
- Outside : green + some black/white

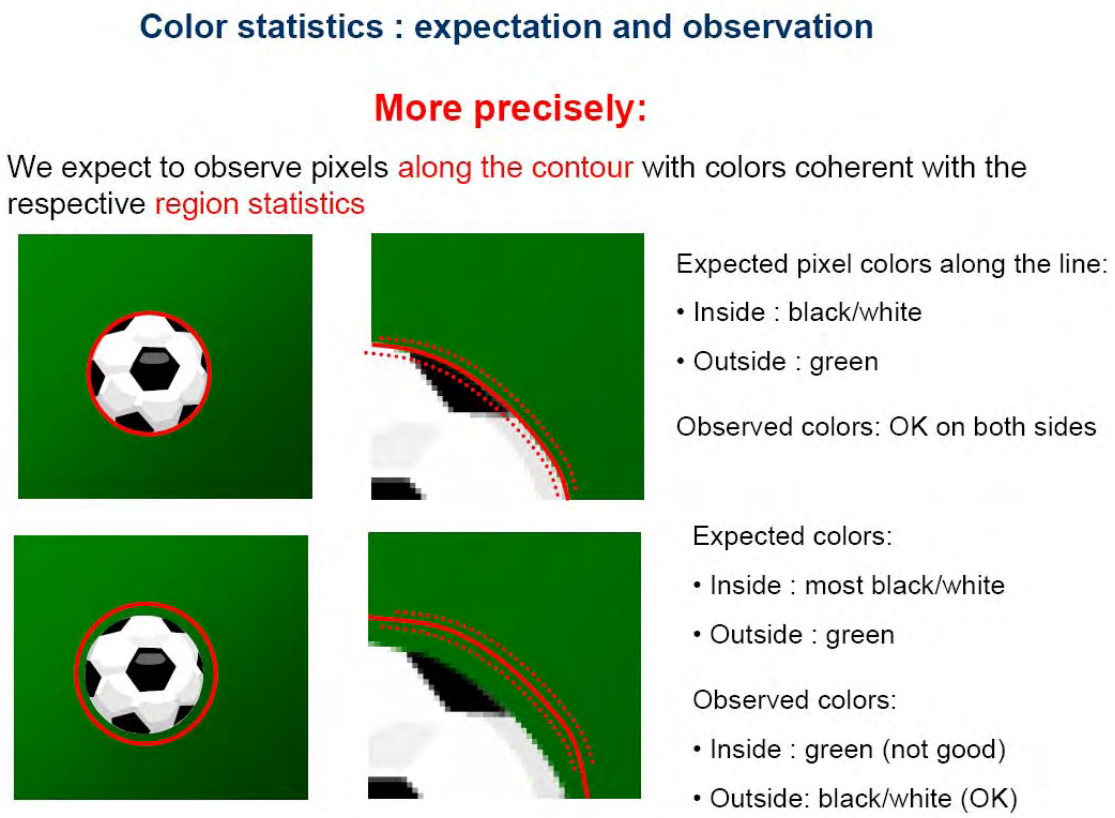
Bad contour hypothesis

- Inside : black/white + some green
- Outside : green

Perfect segmentation:

- Inside : only black/white
- Outside : only green

This is again contour-based tracking, since we use the contour model of the object as the main visual feature to match against an image; the difference with the edge-map approach is the matching criterion: we use region color statistics instead of edges (intensity transitions) in the image.




Idea: If the contour position is well separating the two regions, we should see along the contour (on the respective side) color pixels that are well coherent with therespective region statistics.

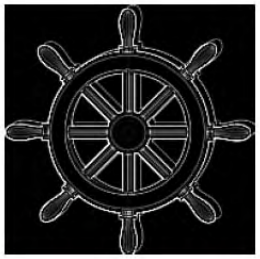
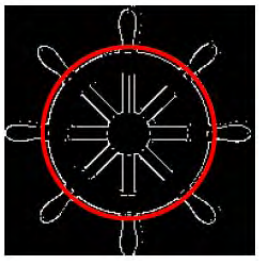
For example: by collecting pixels inside and outside, in the both cases we have inside an almost black+white statistics, and outside almost green.
But in the second case, we observe along the contour on the inner side, green pixels, which are not in accordance with the statistics.
Therefore, we can say that the first contour hypothesis better fits both color statistics than the second one.

## Example of edge-based tracking

A counter-example:

Here we have no chance of using color segmentation! For every pose hypothesis s, inside and outside we always have white+brown color statistics.

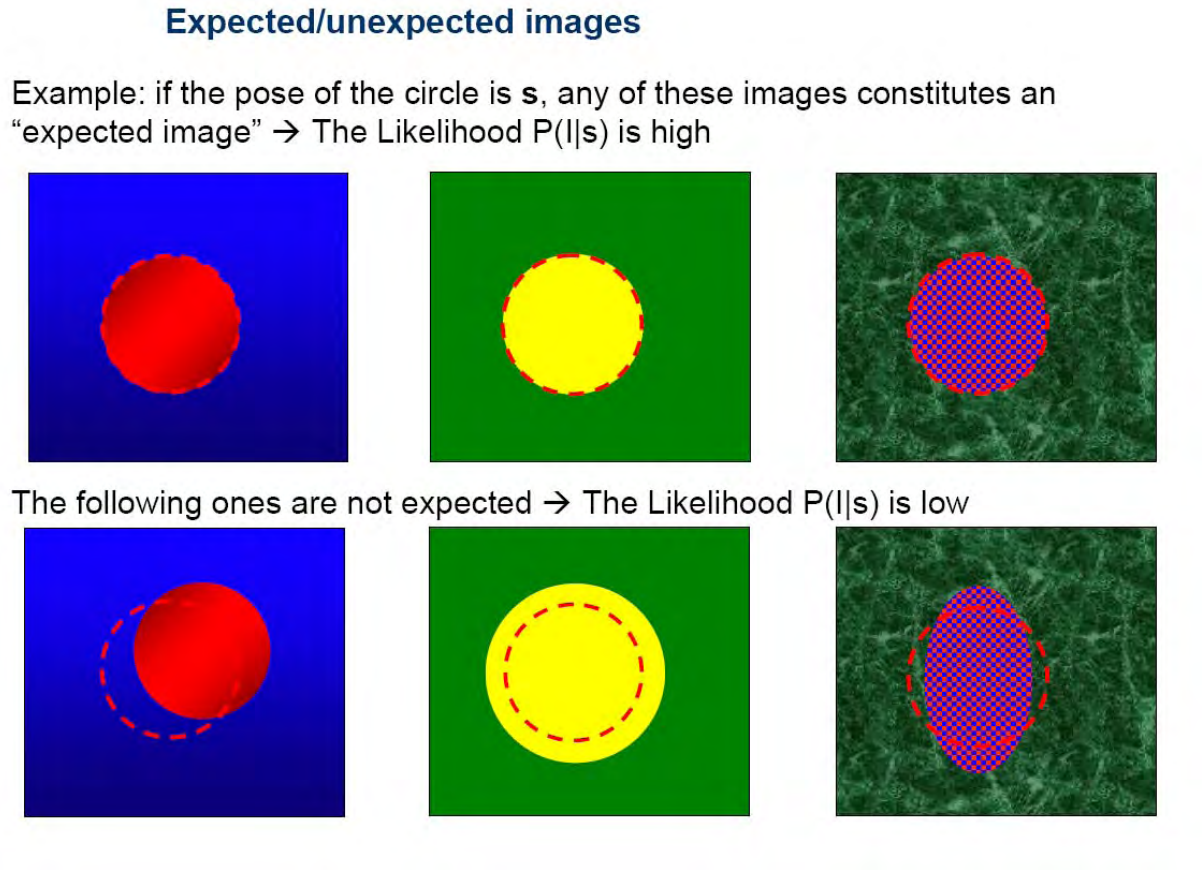
Insted, with edge-based tracking, we have opportunity to track the correct circle

→ Every tracking situation has a proper approach (usually more than one!)

The color separation approach works well only if there is actually a real separation between inside and outside.
In this case, since there are wide holes, inside we see also the background, and there is no real color separation on the correct pose hypothesis. Therefore we need to use edge maps again.

***Color Likelihood definition and the CCD algorithm***

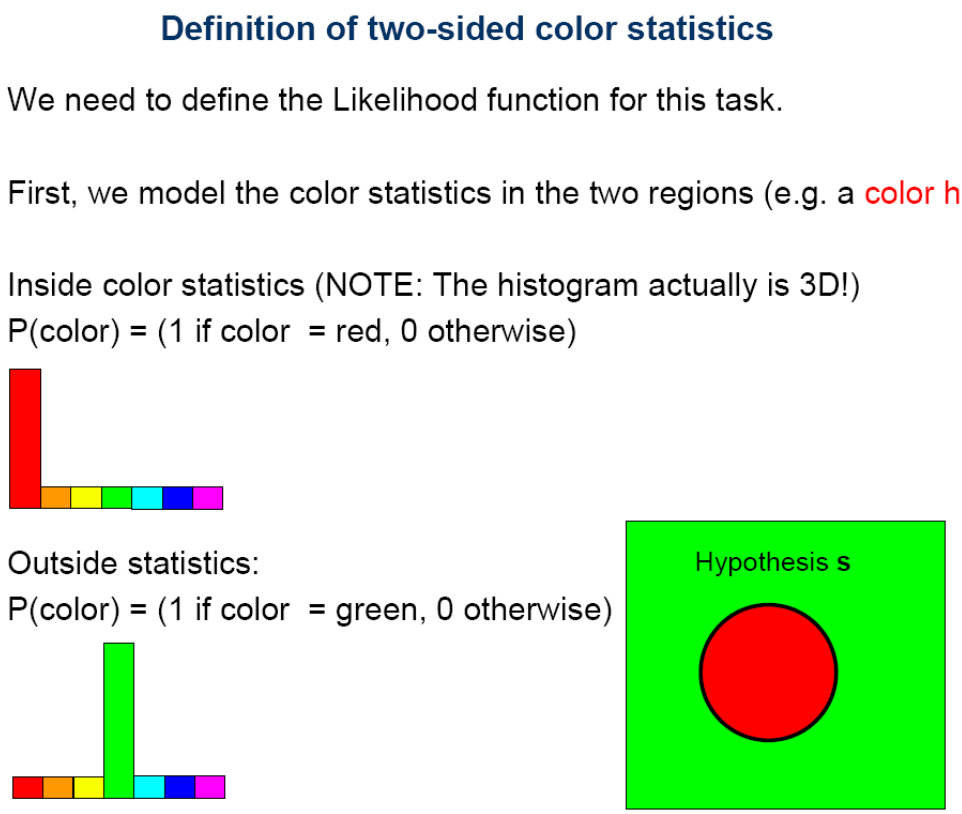



In this approach we can see the measurement z as an object-level one: we try to minimize a complex SSD cost function, which will be a color separation index, which is equivalent to maximizing a Likelihood function.

(Remember: min SSD = max Likelihood, if the measurement uncertainty is Gaussian)

In what follows, we will define the color separation cost function, and how to optimize it. This is the CCD algorithm.

***Modeling the two-sided color statistics***




We start with the simplest case: a uniform color object against a uniform background.

In this case, we can look at the color histograms inside and outside, in order to model color statistics.

A color histogram is a histogram in color space (usually the HSV space = Hue-Saturation-Intensity), where each bin corresponds to the probability of a given color (or a small range of colors).

In the example, inside the contour the bin corresponding to the red color as a high value, and the statistics are modeled by a probability 1 for red and 0 for the other colors.

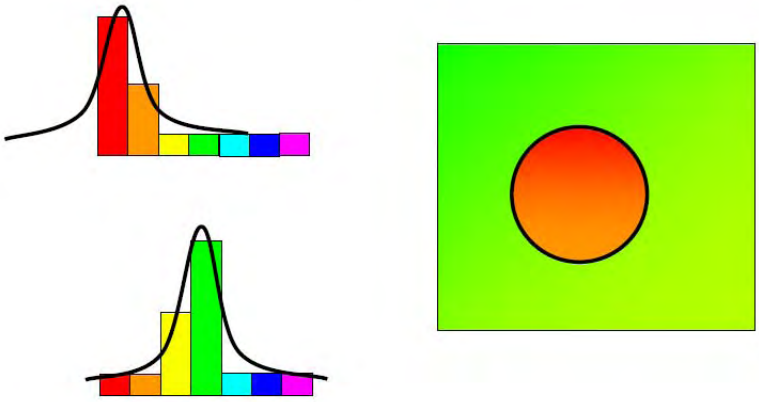
This is a discrete way of modeling the color distributions, and actually not well suited for our optimization purposes, but meant only as a simple graphical representation.

## Gaussian color statistics

More often, the color is not uniform, so we can better model with a RGB Gaussian distribution (3D):

$P(RGB|inside) = Gauss(\mathbf{m}_{RGB1}, C_{RGB1})$

$P(RGB|outside) = Gauss(\mathbf{m}_{RGB2}, C_{RGB2})$



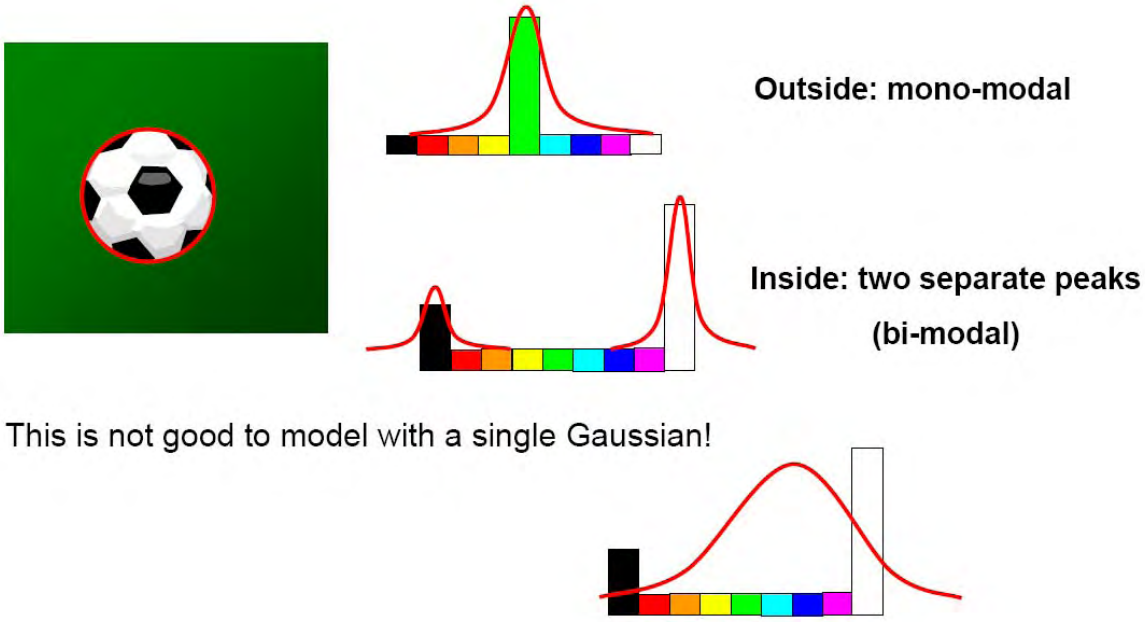
Usually we cannot say that we have uniform color distributions, but colors with different values distributed around an average: inside the average is still red, but also near orange values are present, and the same outside.

The histogram is not a very good representation in this case: then, we use Gaussians in color space.
A Gaussian in color space is a 3-variate Gaussian with a color mean and a (3x3) color covariance matrix.

This models the two color statistics with more accuracy.

## Multi-modal color statistics

In a more general case, we can have a multi-modal distribution




This is not good to model with a single Gaussian!



Color statistics can be multi-modal: a single, global Gaussian in color space is not sufficient to represent the distribution.
We will consider this problem later on, when defining the so-called *local* color statistics.

Now we need to define the SSD cost function to minimize, that constitutes our color separation index.

***Color separation criterion***

## CCD criterion for contour fitting: maximize color separation

If we use color statistics, we can describe the fit of the contour by looking at separation between the two sides.

Assuming that a Gaussian is a good model for each region, we can compute the statistics (mean, covariance) in 3 dimensions: Gauss($\mathbf{m}_A$,$C_A$), Gauss($\mathbf{m}_B$,$C_B$)

The mean (average) RGB colors for the two sides are

$$\mathbf{m}^A = \frac{1}{N_A}\sum_{i=1}^{N_A}\left(r_i^A, g_i^A, b_i^A\right) \qquad \mathbf{m}^B = \frac{1}{N_B}\sum_{i=1}^{N_B}\left(r_i^B, g_i^B, b_i^B\right)$$

Where the sums are over the respective pixels of A and B.

Afterwards, the (3x3) covariance matrices are

$$C^A = \frac{1}{N_A}\sum_{i=1}^{N_A}\left(r_i^A, g_i^A, b_i^A\right)\left(r_i^A, g_i^A, b_i^A\right)^T - \mathbf{m}_A\mathbf{m}_A^T$$

$$C^B = \frac{1}{N_B}\sum_{i=1}^{N_B}\left(r_i^B, g_i^B, b_i^B\right)\left(r_i^B, g_i^B, b_i^B\right)^T - \mathbf{m}_B\mathbf{m}_B^T$$

The first step in CCD consists in computing color statistics for the two regions.

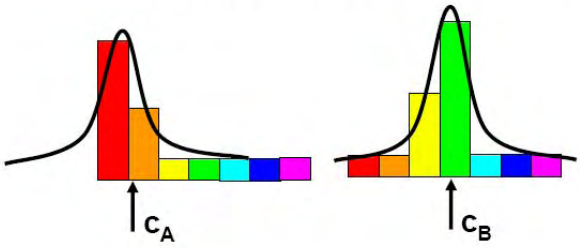
We choose to work in the RGB color space, and we can compute the sample mean and covariance of the color distributions: at a pose hypothesis s, we project the model contour onto the image, and collect all color pixels from the respective region (inside A, and outside B).

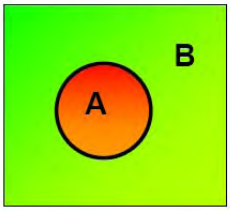
The color mean and covariance matrices for region A are referred by $m^A$ and $C^A$, and the same for B.

## Likelihood: first definition

We can define the Likelihood function P(I | s) in two steps:

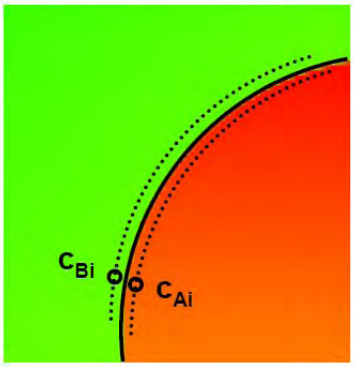
**1.** Compute the local color statistics at pose s




**2.** Look at pixels on the two sides of the contour and compute how well they are predicted by the respective statistics:

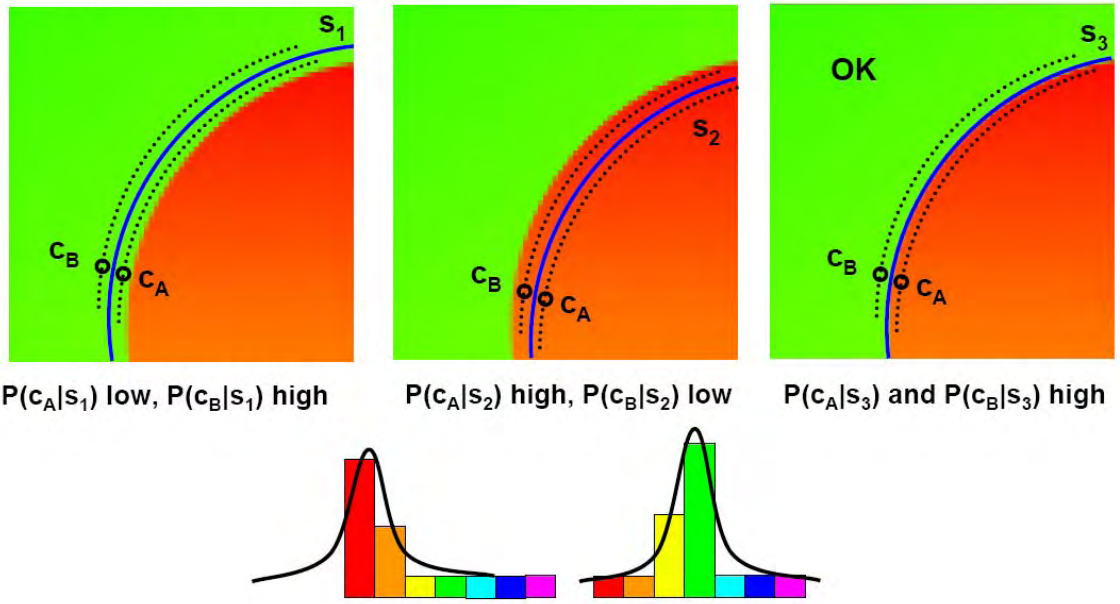
$P(\mathbf{c}_{Ai}) = \text{Gauss}(\mathbf{c}_{Ai}\text{-}\mathbf{m}_A, C_A)$
$P(\mathbf{c}_{Bi}) = \text{Gauss}(\mathbf{c}_{Bi}\text{-}\mathbf{m}_B, C_B)$




Once we have the color statistics, approximated by two Gaussians, we can do a first definition of our Likelihood function for the contour, P(Image | s), by considering a set of sample points $c_{Ai}$ and $c_{Bi}$ along the contour line on the two sides, and see how well the observed colors are predicted from the respective statistics: $P(c_{Ai})$ and $P(c_{Bi})$.

## Separation criterion

Every single point on each side (A/B) will have an observed color $\mathbf{c}_i$ (RGB).
If the separation is good, all the pixels along the contour are well predicted by the respective color statistics




If the separation is good, then all sample points have a high probability, that means, they “fit well” with the respective statistics.

***Maximum Likelihood pose hypothesis***

## Maximum Likelihood hypothesis

The probability of each color is a Gaussian, and the overall Likelihood of the hypothesis **s** will be the product:

$$P(I \mid \mathbf{s}) = \prod_i Gauss(\mathbf{c}_i^{(A)} - \mathbf{m}^A, C^A)) \cdot Gauss(\mathbf{c}_i^{(B)} - \mathbf{m}^{(B)}, C^{(B)}))$$

$$\propto \exp\left( -\sum_i (\mathbf{c}_i^{(A)} - \mathbf{m}^A)^T (C^A)^{-1} (\mathbf{c}_i^{(A)} - \mathbf{m}^A) - \sum_i (\mathbf{c}_i^{(B)} - \mathbf{m}^B)^T (C^B)^{-1} (\mathbf{c}_i^{(B)} - \mathbf{m}^B) \right) =$$

$$\exp\left( -\sum_i d_A(\mathbf{c}_i^{(A)}, \mathbf{m}^A)^2 - \sum_i d_B(\mathbf{c}_i^{(B)}, \mathbf{m}^B)^2 \right)$$

Where $d_A(\mathbf{c}_i^{(A)}, \mathbf{m}^A)^2 = (\mathbf{c}_i^{(A)} - \mathbf{m}^A)^T (C^A)^{-1} (\mathbf{c}_i^{(A)} - \mathbf{m}^A)$
are the Mahalanobis distances of each pixel to the respective statistics (A or B)

If we maximize P(I|s), we find the maximum Likelihood hypothesis

$$\arg\max_{\mathbf{s}} P(I \mid \mathbf{s}) = \arg\min_{\mathbf{s}} \left( \sum_i d_A(\mathbf{c}_i^A, \mathbf{m}^A)^2 + \sum_i d_B(\mathbf{c}_i^B, \mathbf{m}^B)^2 \right)$$

In mathematical terms, we compute all sample point probabilities, and multiply them together: this corresponds to an independence assumption (every observed point color does not depend on the others).

The result for P(I|s) is a global Gaussian, where the exponent is the sum of Mahalanobis distances, between expected and observed colors, at each contour point.

As we know, this is the standard case for Gaussians: maximizing a Gaussian probability is equivalent to minimizing a Mahalanobis distance, which can always be written as a weighted SSD cost function.

So, it seems that we formulated the problem in a standard nonlinear LSE way. But there is still a problem.

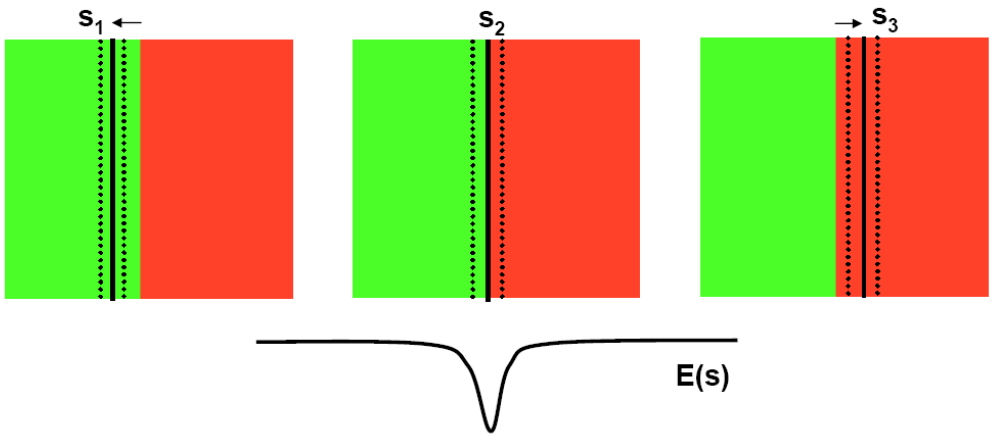
## The color-separating cost function

The cost function to be optimized is

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \left( \sum_i d_A(\mathbf{c}_i^A, \mathbf{m}^A)^2 + \sum_i d_B(\mathbf{c}_i^B, \mathbf{m}^B)^2 \right) = \arg\min_{\mathbf{s}} E(\mathbf{s})$$

Where everything inside is a function of $\mathbf{s}$ : $\mathbf{c}^{A/B}(\mathbf{s}), \mathbf{m}^{A/B}(\mathbf{s}), C^{A/B}(\mathbf{s})$

Let us take a very simple example




Although the cost function is a SSD (sum of squared Mahalanobis distances), this is not a standard LSE problem, since everything inside E(s) depends on s: both the color statistics (m,C) and the color points $c_{Ai}$, $c_{Bi}$.

If only the sample points $c_{Ai}$, $c_{Bi}$ were dependent on s, then we would have a standard LSE, and we could use Gauss-Newton for optimizing it.

By taking a simple example like the one above, we can see also another problem: the shape of this cost function is very narrow, therefore in order to optimize it successfully, we would need to start from a very near initial guess to the optimal pose.

*Refining the cost function*

### Problem of the cost function

This function has two problems:

**1.** It is too narrow to be safely optimized (we need a very close initial guess $\mathbf{s}_0$)

**2.** In any case, it is too complex to make derivatives, and not in a standard nonlinear LSE form

The second problems arises because everything inside E depends on s:

$$E(s) = \sum_i d_A(\mathbf{c}_i^A(\mathbf{s}), \mathbf{m}^A(\mathbf{s}))^2 + \sum_i d_B(\mathbf{c}_i^B(\mathbf{s}), \mathbf{m}^B(\mathbf{s}))^2$$

- The color region statistics ($\mathbf{m}(\mathbf{s})$,$C(\mathbf{s})$)

- The observed pixel colors $\mathbf{c}(\mathbf{s})$

So the derivatives are extremely complicated!

*1 – Split the optimization in two steps*

### Solution 1: split optimization in two subtasks

In order to solve the second problem, CCD iterates two steps:

Iterate:

1 - Compute color statistics ($\mathbf{m}(\mathbf{s})$,$C(\mathbf{s})$)

2 - Optimize the cost function E, by keeping ($\mathbf{m}$,C) fixed, and considering only the observed color $\mathbf{c}(\mathbf{s})$ dependent on the state:

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \left( \sum_i d_A(\mathbf{c}_i^A(\mathbf{s}), \mathbf{m}^A)^2 + \sum_i d_B(\mathbf{c}_i^B(\mathbf{s}), \mathbf{m}^B)^2 \right) = \arg\min_{\mathbf{s}} E_2(\mathbf{s})$$

3 – Update **s**, and repeat 1 and 2

The error $E_2$ is a standard (weighted) nonlinear LSE

→ We can do a weighted Levenberg-Marquardt update of **s**!

This is fast and efficient

A first approach to the second problem is to split the optimization in two subtasks: first compute color statistics (m,C), then optimize the LSE function, by considering only $c_{Ai}$, $c_{Bi}$ to be dependent on the pose s, and keeping (m,C) fixed.

The statistics (m,C) will be then updated after each Gauss-Newton pose update s+Δs.

This is reasonable, since color statistics do not change very much from one pose to the next, therefore their influence to the error is much less fast than the influence of observed color pixels $c_{Ai}$, $c_{Bi}$, which instead change much faster with the pose parameters.

We can call the new cost function $E_2(s)$, which has the same value of E(s) for a given pose, but where the color statistics (m,C) terms are considered not dependent on s, when computing the derivatives.

This solves only the second problem, but the first one (i.e. the fact that E(s) has a very narrow convergence region) is even worse with $E_2(s)$!

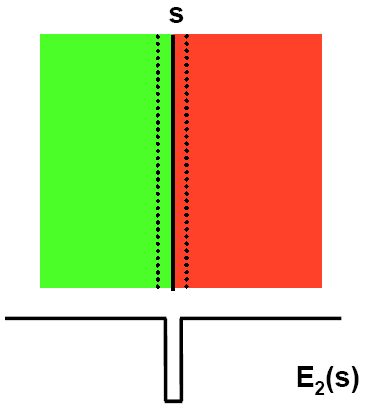
### *2 – Blurring the statistics*

**Second problem of E**

So, we solved the second issue. But the first problem becomes even worse.

If we try to optimize the error $E_2$ considering only the dependence of **c**(**s**), for a given statistics (m,C) we have something like this:

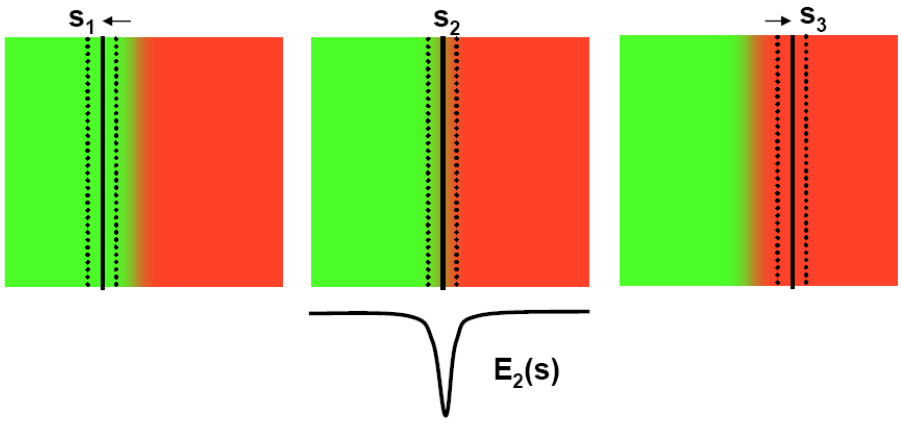
$$E_2(\mathbf{s}) = \sum_i d_A(\mathbf{c}_i^A(\mathbf{s}), \mathbf{m}^A)^2 + \sum_i d_B(\mathbf{c}_i^B(\mathbf{s}), \mathbf{m}^B)^2$$




where the statistics (m,C) are fixed from Step 1 (not dependent on **s**).
This is because we have a sharp transition when the line crosses the color edge.

This phenomenon is due to the fact that in this example, there is a very sharp transition between color values. Therefore, by taking sample points near the contour, their classification changes very quickly when we cross the boundary between color regions.

For this problem, we can blur the image:
Apply a Gaussian Filter to the original color image



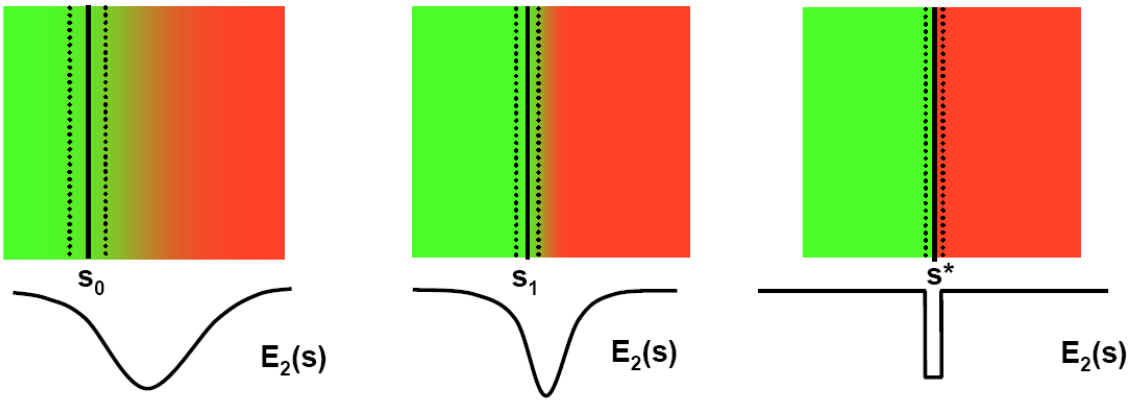

Now there is no sharp transition, and the cost function $E_2$ is also smooth.

## Solution 2: Multi-resolution approach

What is the price to pay, when we do blurring?
We loose details, so the optimization will not be precise anymore!

Therefore, we apply the multi-resolution principle:

- Start with a lower resolution (more blur) at pose $\mathbf{s}_0$, when we are far from the optimum
- After each pose update **s**, increase the resolution and shrink the points




This problem is actually common to many different Computer Vision problems, where the cost function becomes too sharp for a successful optimization, if the first pose guess $s_0$ is far from the correct one.

A common solution is given by a multi-resolution approach: by blurring the original image of an increasing amount, we get images with less sharp transitions everywhere, but also less precise details.

The effect on the respective cost function is that they get smoother, with a larger convergence area, but less precise optimal value.

Therefore, a multi-resolution approach consists in starting the optimization from $s_0$, using the smoother (more blurred) version of the image, performing a full Gauss-Newton (or L-M) loop, increasing the resolution and repeating the optimization again, starting from the previous result $s_1$.

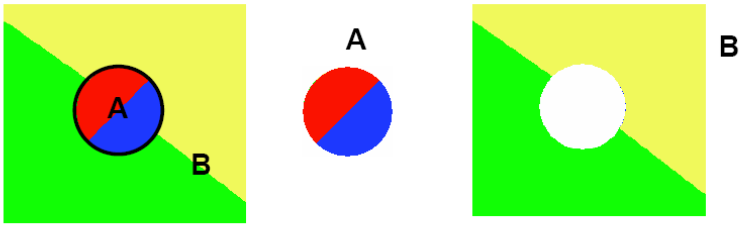
In our case, a further benefit for the optimization is obtained keep the sample points more far to the contour line for the first loop, and shrink their distance for the next loops. This widens even more the low resolution versions.

### *3 – Using local statistics for multi-modal distributions*

**Third problem : Multi-modal color distributions**

So far, so good... with this simple example (two regions with unimodal color)

But in real scenes, we have multi-modal color statistics. Just two global Gaussians $(m_A,C_A)$, $(m_B,C_B)$ cannot be a good description.

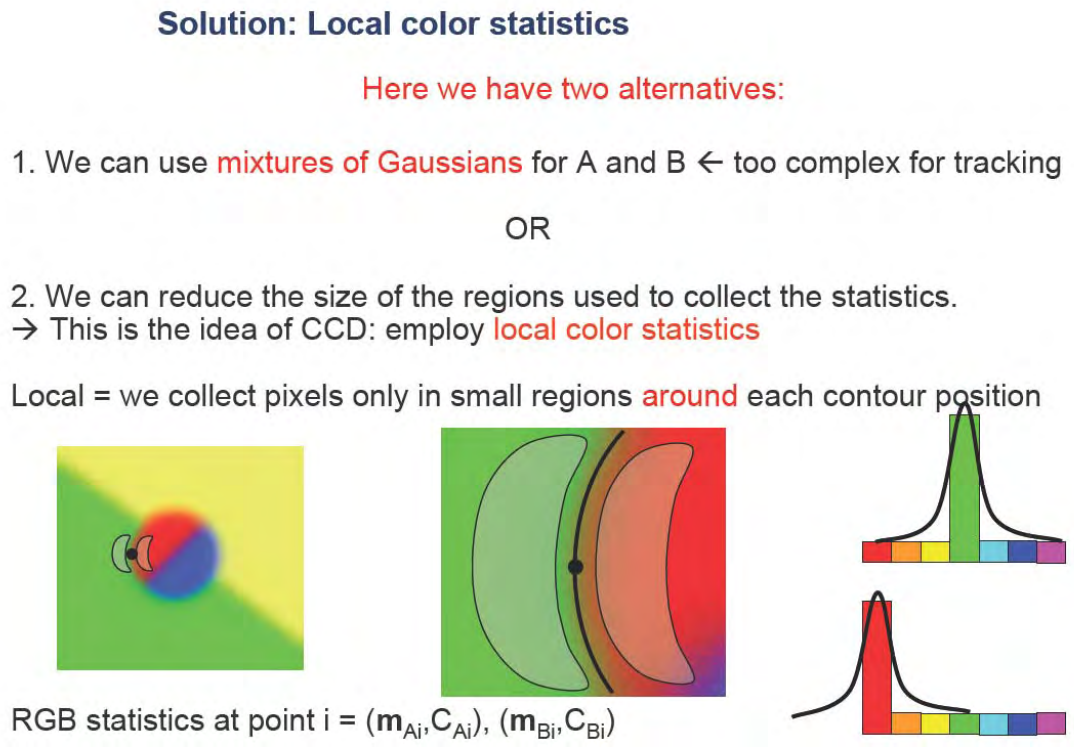



Pixel colors along the contour will be badly predicted from the statistics, if the regions are not well described by a single Gaussian (m,C).

So, we need more flexible color statistics!

Now we have to consider the problem mentioned in the beginning: multi-modality of color distributions. In fact, in a situation like the one above (which happens quite often) we cannot model color statistics with single Gaussians.

In real scenes, this is true at least for the background, which is unpredictable and can have multi-modal color statistics.




We could think about using mixtures of Gaussians, but this gives a too complex approach for a real-time tracking algorithm: the number of Gaussians is unknown, and should be estimated at each optimization step.

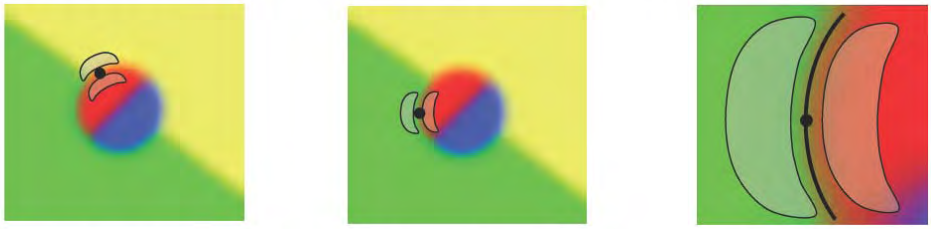
A better idea, used in CCD, is to use local color statistics: in fact, although the full region had a multi-modal color distribution, in a neighborhood of a contour position we can say that the statistics are mono-modal, and can be still approximated by Gaussians.

Therefore, for every contour position *i* we take a local (small) area around, and compute the respective statistics $(m_i,C_i)$.

## Likelihood function with Local statistics

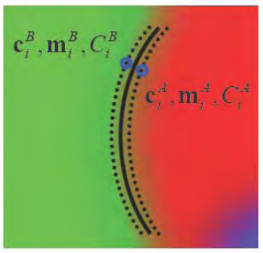
The estimation steps become: at pose **s**

1. For each contour position i, compute local statistics in 2 windows : ($\mathbf{m}_{Ai}$,$C_{Ai}$), ($\mathbf{m}_{Bi}$,$C_{Bi}$)



2. Compute the Likelihood, by looking at observed colors along the line

$$P(I \mid \mathbf{s}) = \prod_i Gauss(\mathbf{c}_i^A(\mathbf{s}) \mid \mathbf{m}_i^A, C_i^A) \cdot Gauss(\mathbf{c}_i^B(\mathbf{s}) \mid \mathbf{m}_i^B, C_i^B)$$




3. Update the maximum Likelihood pose $\mathbf{s}^* = \arg\max_{\mathbf{s}} P(I \mid \mathbf{s})$
and repeat the process

In this way, we can re-formulate the Likelihood (and SSD) function, and use the same Gauss-Newton approach developed before: here, the only difference is that now (m,C) are dependent on i (contour position) but not on s, of course.

After each GN step, we will re-compute local statistics as well.

### *Optimizing the Likelihood with Gauss-Newton*

## Maximum Likelihood pose

The maximum Likelihood pose now is

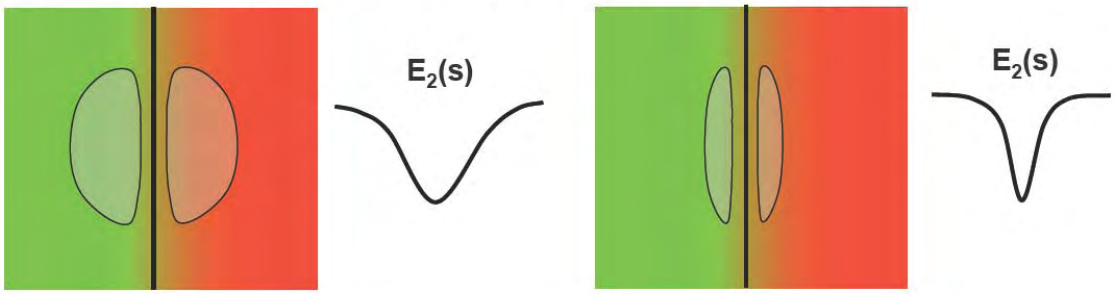
$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \left( \sum_i d_{Ai}(\mathbf{c}_i^A(\mathbf{s}), \mathbf{m}_i^A)^2 + \sum_i d_{Bi}(\mathbf{c}_i^B(\mathbf{s}), \mathbf{m}_i^B)^2 \right)$$

With $d_{Ai}, d_{Bi}$ = Local Mahalanobis distances

$$d_{Ai}(\mathbf{c}_i^A, \mathbf{m}_i^A)^2 = (\mathbf{c}_i^A - \mathbf{m}_i^A)^T (C_i^A)^{-1} (\mathbf{c}_i^A - \mathbf{m}_i^A)$$

$$d_{Bi}(\mathbf{c}_i^B, \mathbf{m}_i^B)^2 = (\mathbf{c}_i^B - \mathbf{m}_i^B)^T (C_i^B)^{-1} (\mathbf{c}_i^B - \mathbf{m}_i^B)$$

Next question: how large are the areas for local statistics?




Local areas must be not too small, since otherwise the assumption of independence of (m,C) on s is violated: they will change too fast with s, and this dependence could not be anymore neglected!

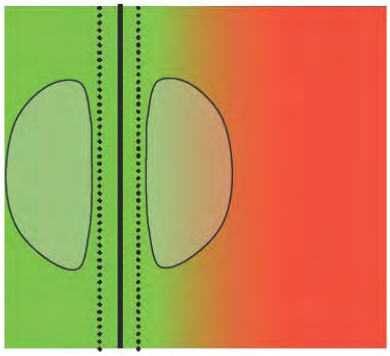
But they must also be not too large, otherwise the assumption of mono-modality (single color distribution ~Gaussian) would instead be violated.

Finally, for good convergence properties, local areas for collecting statistics are kept wider for low-resolution images, and decreasing for higher resolutions.

## Choice of the Local statistics area

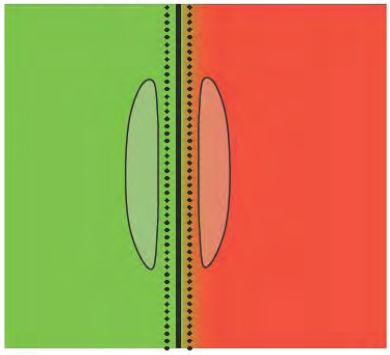
- Far from the optimum, it is better to take large areas, otherwise we may fail the optimization

We have less precision,
but a wide function E(s) and $E_2(s)$



- Afterwards we can decrease the areas as well

E(s), $E_2(s)$ get narrow, but more precise



## Levenberg-Marquardt optimization of $E_2$

Now we can apply Levenberg-Marquardt

First, we write the cost function as a standard NLSE:

$$E_2(\mathbf{s}) = \left( \sum_i d_{Ai}(\mathbf{c}_i^A(\mathbf{s}), \mathbf{m}_i^A)^2 + \sum_i d_{Bi}(\mathbf{c}_i^B(\mathbf{s}), \mathbf{m}_i^B)^2 \right) =$$

$$= \sum_i (\mathbf{c}_i^A - \mathbf{m}_i^A)^T (C_i^A)^{-1} (\mathbf{c}_i^A - \mathbf{m}_i^A) + (\mathbf{c}_i^B - \mathbf{m}_i^B)^T (C_i^B)^{-1} (\mathbf{c}_i^B - \mathbf{m}_i^B)$$

$$= \sum_i \left\| W(\mathbf{c}(\mathbf{s}) - \mathbf{m}) \right\|^2$$

$$\mathbf{c}(\mathbf{s}) = \begin{bmatrix} \mathbf{c}_1^A \\ \cdots \\ \mathbf{c}_N^A \\ \mathbf{c}_1^B \\ \cdots \\ \mathbf{c}_N^B \end{bmatrix} \quad \mathbf{m} = \begin{bmatrix} \mathbf{m}_1^A \\ \cdots \\ \mathbf{m}_N^A \\ \mathbf{m}_1^B \\ \cdots \\ \mathbf{m}_N^B \end{bmatrix} \quad W = \begin{bmatrix} \sqrt{(C_1^A)^{-1}} & & & & & 0 \\ & \cdots & & & & \\ & & \sqrt{(C_N^A)^{-1}} & & & \\ & & & \sqrt{(C_1^B)^{-1}} & & \\ & & & & \cdots & \\ 0 & & & & & \sqrt{(C_N^B)^{-1}} \end{bmatrix}$$

At each step, after that local statistics are computed, optimizing E2(s) is a standard, weighted nonlinear LSE problem, for which a Gauss-Newton (or Levenberg-Marquardt) step can be performed.

In this formulation, at each position i the observed color values $c_{Ai}(s)$ and $c_{Bi}(s)$ have to be matched against the predicted (mean) colors $m_{Ai}$, $m_{Bi}$, in the Mahalanobis distance sense, and the weight matrix W contains all of the color covariance matrices.

## Gradient computation

The L-M parameters update is

$$\Delta \mathbf{s} = (J^T W J + \lambda I)^{-1} J^T W (\mathbf{c}(\mathbf{s}) - \mathbf{m})$$

Where $J(s) = \left[\frac{\partial \mathbf{c}}{\partial \mathbf{s}}\right]$ is the color Jacobian matrix (3Nx6) of the pixels $\mathbf{c}_i$

This matrix is given by $J(s) = \begin{bmatrix} \frac{\partial \mathbf{c}_1}{\partial \mathbf{s}} \\ \cdots \\ \frac{\partial \mathbf{c}_N}{\partial \mathbf{s}} \end{bmatrix}$

$$\frac{\partial \mathbf{c}_i}{\partial \mathbf{s}} = \begin{bmatrix} r_x & r_y \\ g_x & g_y \\ b_x & b_y \end{bmatrix} \cdot \frac{\partial f_i}{\partial \mathbf{s}}$$

Where $r_x, g_x, b_x$ and $r_y, b_y, g_y$ are the horizontal and vertical color gradients at point $\mathbf{c}_i$, and the other term is the usual Jacobian (2x6) of 3D point projection f.

The Jacobian matrix for this problem contains the image color gradients (3x2) and the usual Jacobian (2x6) of the nonlinear projections $f_i(s)$ (from body to screen contour points).

***Adding prior knowledge for MAP estimation***

**MAP optimization**

Up to now, we did a Maximum-Likelihood estimate…
But the CCD algorithm is actually a MAP (Bayesian) method!

MAP = Maximum-A-Posteriori

We look for the maximum posterior probability (Bayes)

$$\mathbf{s}^* = \arg\max_{\mathbf{s}} P(\mathbf{s} \mid \mathbf{z}) = \arg\max_{\mathbf{s}} (P(\mathbf{z} \mid \mathbf{s}) P(\mathbf{s}))$$

Where P(**s**) is the prior at time and P(**z**|**s**) the Likelihood functions, at time t

NOTE: In CCD we do not estimate the full posterior P(**s**|**z**)

Reason : the full Likelihood P(**z**|**s**) is too complex, so we can only maximize it!

CCD is therefore a Maximum-Likelihood pose estimation algorithm, where the Likelihood maximized color separation between the two regions.

A final improvement of CCD can be obtained if we have also a prior information P(s) about the pose (for example, from the previous frame estimation).

In fact, in this case we can use Bayes' rule, and multiply it with the Likelihood, to obtain a MAP (maximum-a-posteriori) pose estimation s*.

In this case, we do not estimate a full state posterior (Bayesian tracking), but we just maximize P(s|z), because the Likelihood P(z|s) is far too complex to be represented.

## MAP optimization

### CCD algorithm - MAP optimization

If the prior is a Gaussian with given parameters ($\mathbf{s}_0$,$C_0$), where $\mathbf{s}_0$ is the initial state hypothesis, then:

$$\mathbf{s}^* = \arg\max_{\mathbf{s}}(P(\mathbf{z}\,|\,\mathbf{s})P(\mathbf{s})) = \arg\min_{\mathbf{s}}(E(\mathbf{s}) + d_0(\mathbf{s},\mathbf{s}_0))$$

Where $d_0$ is the Mahalanobis distance between $\mathbf{s}$ and $\mathbf{s}_0$, with covariance matrix $C_0$:

$$d_0(\mathbf{s},\mathbf{s}_0) = (\mathbf{s}-\mathbf{s}_0)^T C_0^{-1}(\mathbf{s}-\mathbf{s}_0)$$

To do the MAP pose estimation, we add the second term to our Levenberg-Marquardt optimization step:

$$\Delta\mathbf{s} = (J^T W J + \lambda I)^{-1} J^T W(\mathbf{c}(\mathbf{s}) - \mathbf{m}) + C_0^{-1}(\mathbf{s}-\mathbf{s}_0)$$

If the prior P(s) is modeled with another Gaussian, with mean $s_0$ and (6x6) covariance $C_0$ (in *pose* space), we can add this term to the overall Mahalanobis distance, when performing a Levenberg-Marquardt optimization step.

***The overall CCD pose estimation algorithm***

## The CCD algorithm for contour tracking

0. Initialize: set the state to prior $\mathbf{s}_0$, with uncertainty $C_0$

Repeat: at pose $\mathbf{s}_k$

1. Blur the color image

2. Compute local color statistics: at each contour position, take two areas (inside and outside) and collect color pixels to compute $(\mathbf{m}_{Ai}, C_{Ai})$ (Gaussians in RGB space)

3. Collect all color pixels along the contour $\mathbf{c}_i$

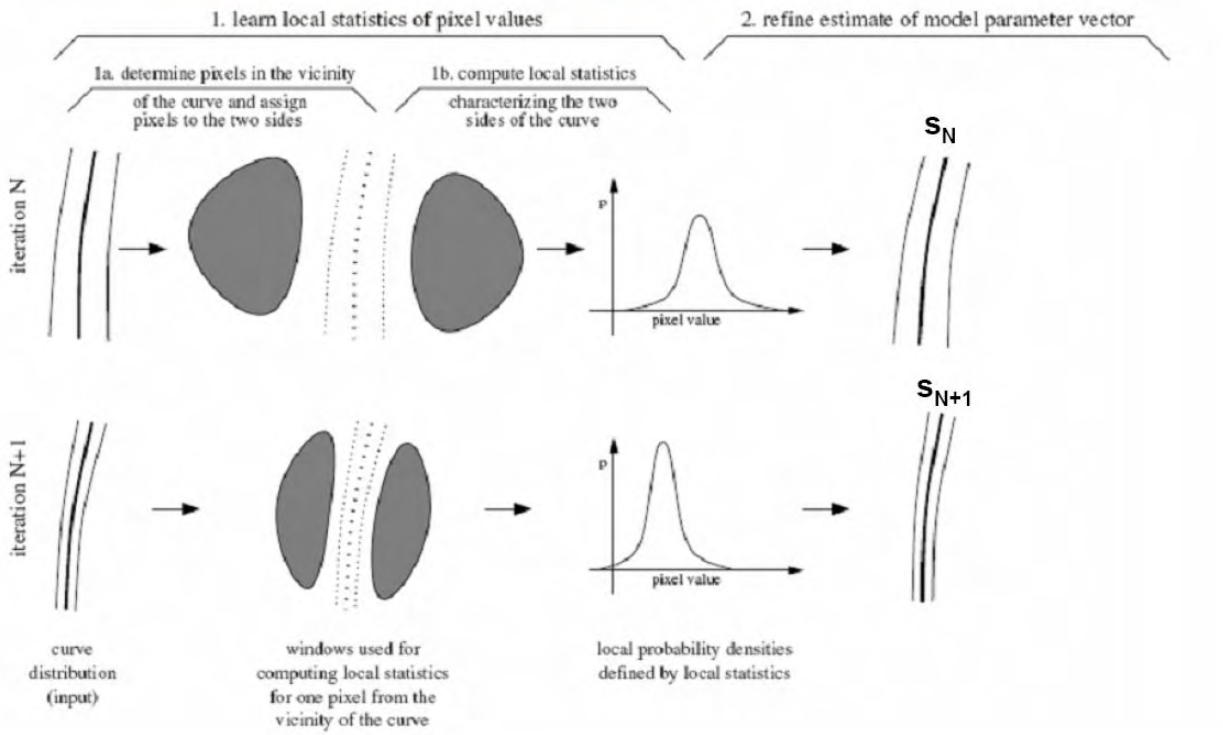
4. Compute the color Jacobian matrix (3Nx6) by multiplying the N color gradients (3x2) with the N projection Jacobians (2x6)

5. Collect all the 2N local means in a vector **m** and the 2N local covariances in a weight matrix W

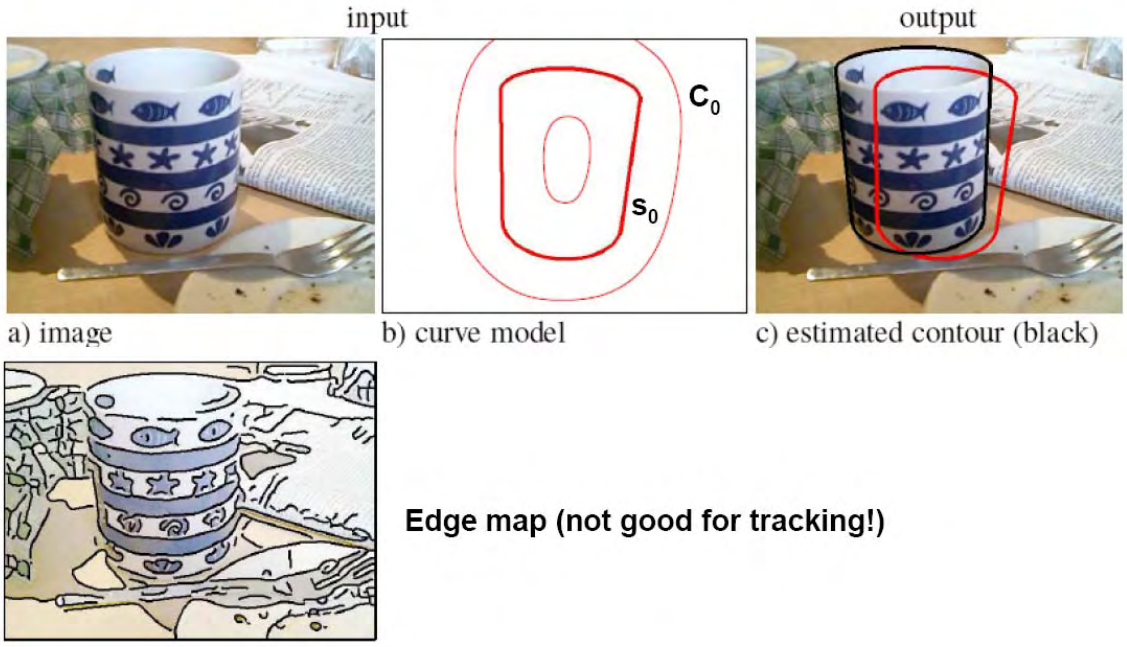
6. Perform the weighted Levenberg-Marquardt optimization step and update the pose $\mathbf{s} \rightarrow \mathbf{s}+\Delta\mathbf{s}$

7. Shrink both the local statistics area and the pixel positions around the contour, and repeat the loop.

## A graphical representation of CCD

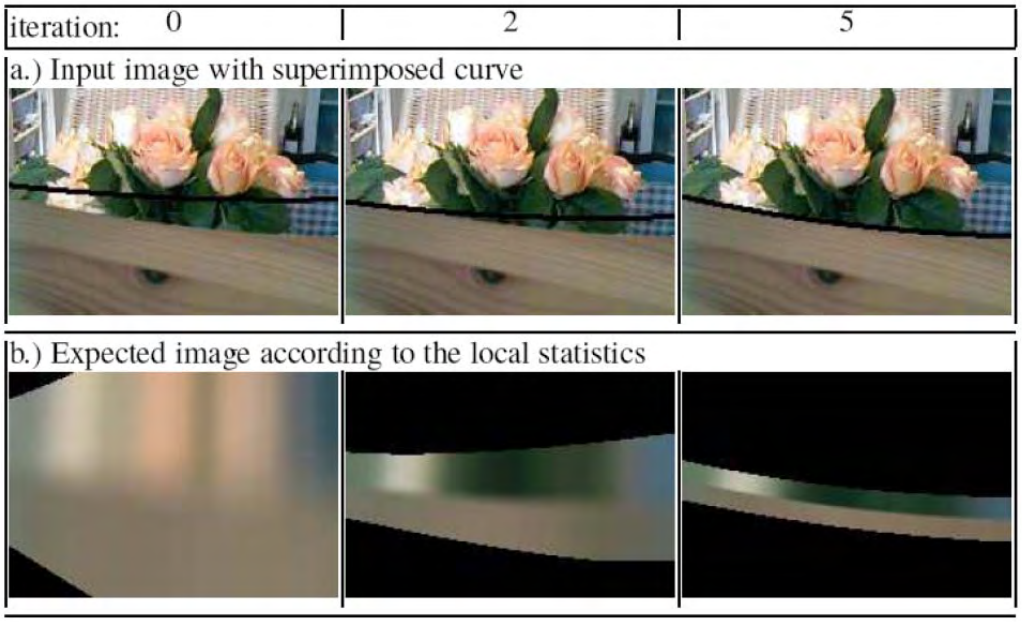


Example of CCD: Input image + Prior Probability → Output (MAP) pose




In this picture, the prior information about the position of the cup is represented by a Gaussian (s0,C0) in pose space.
By starting the optimization from $s_0$, and performing a full MAP estimation, we obtain the result shown above.

Example of CCD: Optimization steps with expected images



Here we can see how local color statistics are more blurred in the first optimization steps, while increasing precision (and decreasing convergence area as well) for the subsequent steps, where the certainty about the pose estimation increases.


This is an example where a common edge-based contour tracker would fail, due to too many edges, both inside and outside the object.

## CCD - features

## CCD Algorithm for contour tracking

Advantage: it is fast and precise

- It resists to clutter, occlusions etc. because the local statistics are a flexible representation of colors
- It can optimize very precisely the contour pose
- Even though complex, it can be very fast (the Real-Time CCD is a quicker implementation), because the Levenberg-Marquardt steps are not many (10 usually)

Disadvantage: it has no initialization…

- It is not a real Bayesian tracker (no prediction-correction scheme), so the prior must be given from outside, each time!

→ Solution: plug-in a simple Bayesian tracker (e.g. Kalman Filter) using **z**=**p***, the estimated CCD pose at time t.

- It needs to be initialized manually (to give the prior knowledge) both at the beginning and in case of tracking loss

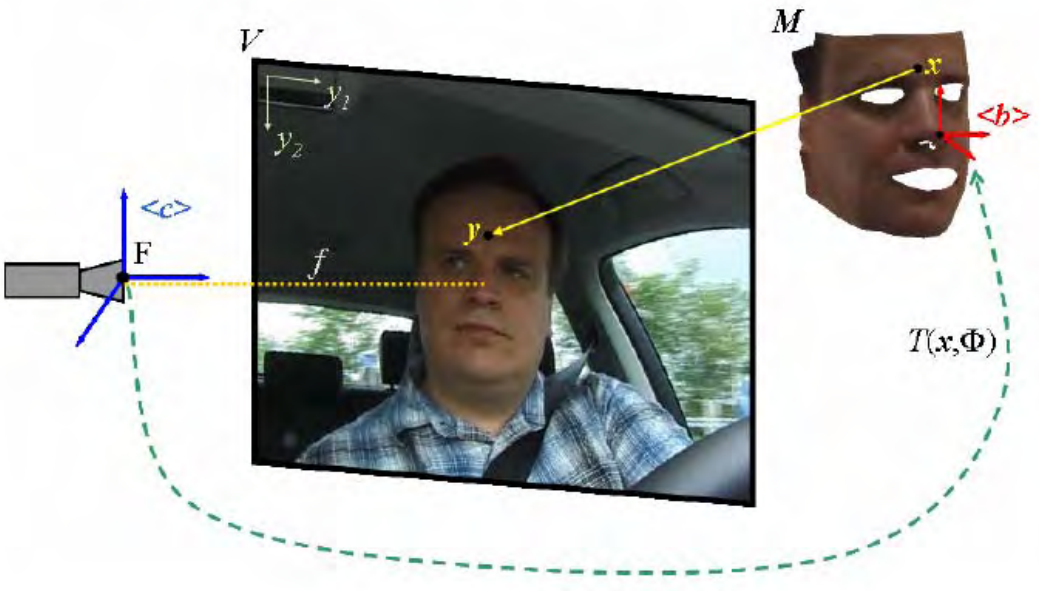
→ Solution: employ a slower, global method (maybe not precise) to initialize it: for example, SIFT features.

## *Lecture 11 – Active Appearance Models*

### Template modeling and tracking

## Teplate-based tracking

Template-based tracking = a procedure to estimate the pose of an object, by using the full surface appearance mapped over the 3D shape.

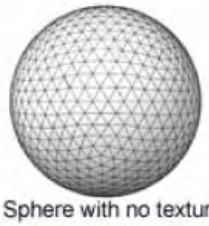


## Shape+Appearance Model

For this task we need a two-fold model:

1. The object 3D shape

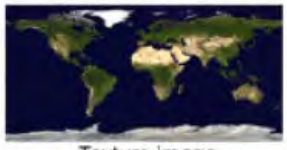
S



Sphere with no texture

2. The surface appearance, or texture image, mapped onto the shape

A



Texture image

→ Template = a complete description, with full information about the object surface, and it can be used for tracking

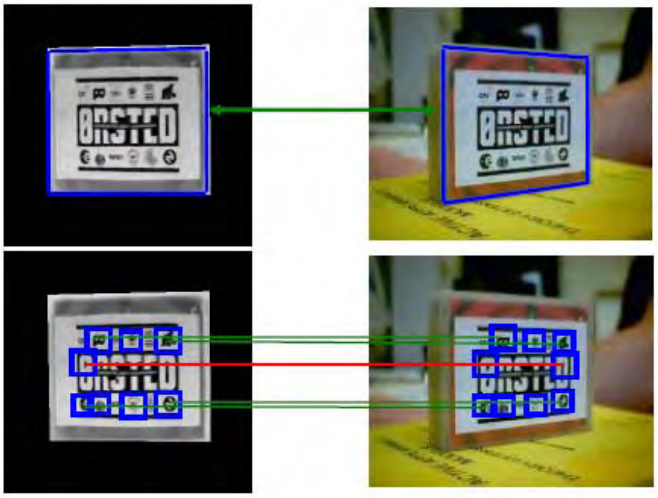
S+A

Sphere with texture

Here the model consists of the full 3D shape, together with the surface appearance.
This approach exploits the full information for tracking, and therefore it is also called a global-feature approach.

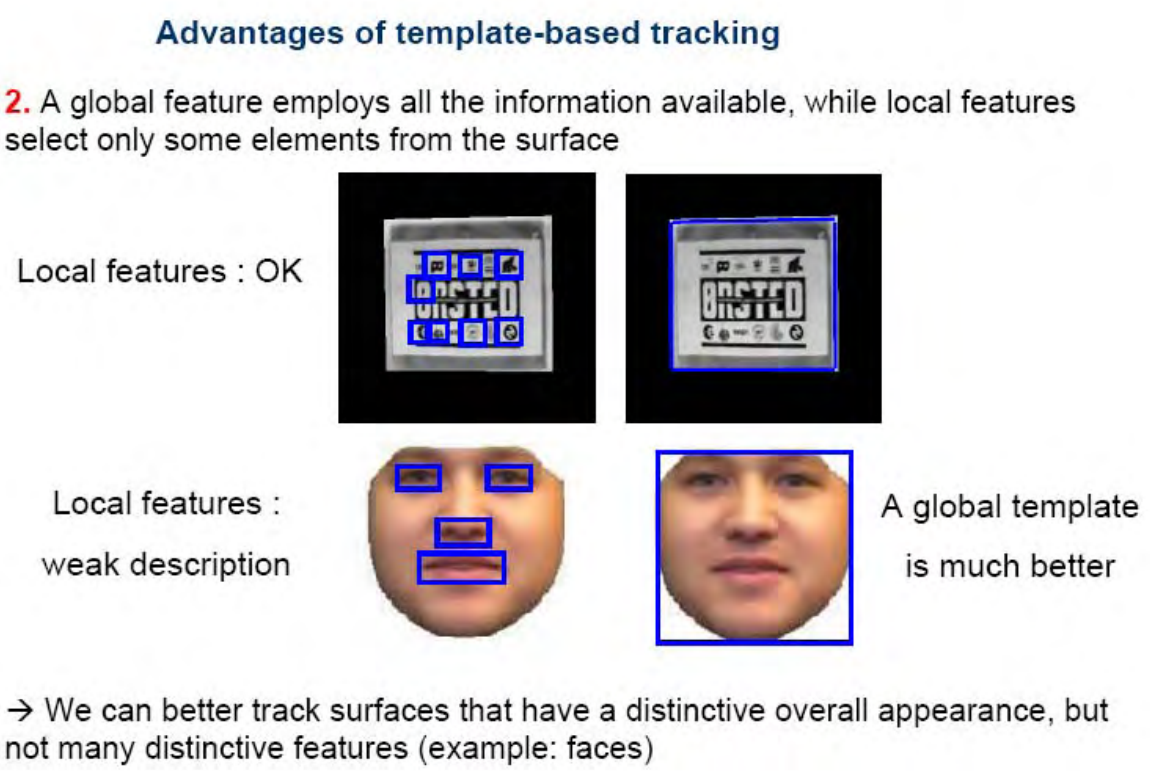
## Advantages of template-based tracking

1. We have only one global feature to track, not anymore many local features



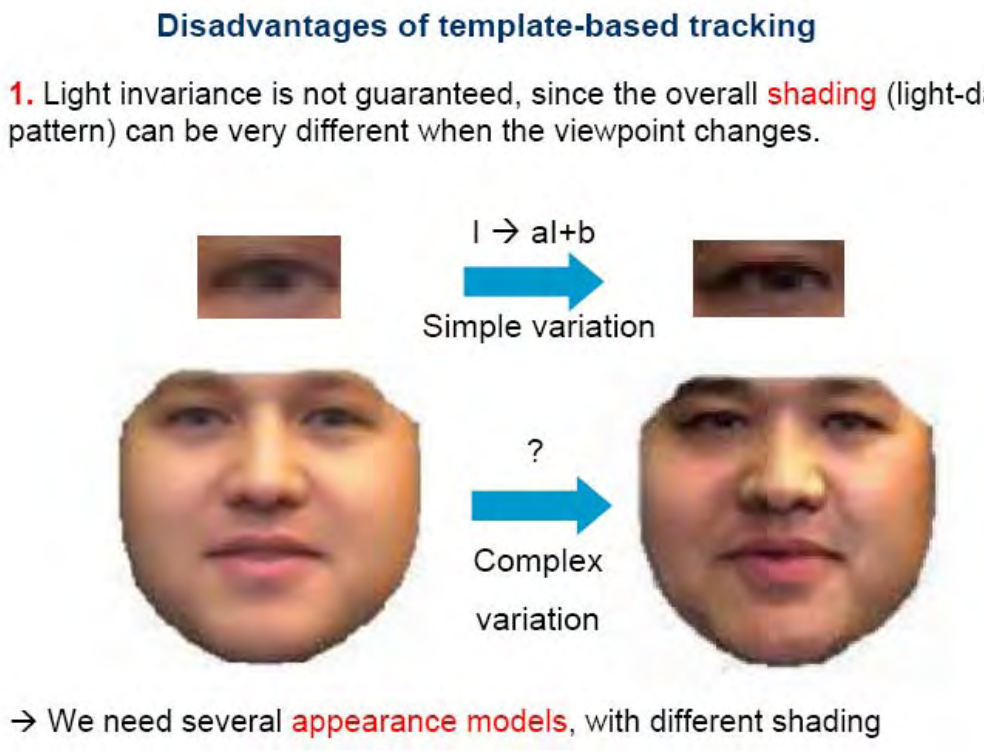
→ There is no problem of false matchings: which feature correspond to which in the database?

By using a global template, we have not anymore the problem of solving false correspondences, as we did for local features.
In other words, all of the visible template points on the surface, at a given pose hypothesis, will be all together directly mapped onto the current image for comparison.




Another advantage concerns surfaces which have not many distinctive local features, but an overall distinctive appearance.
For example, a face has few distinctive keypoints (eye corners, mouth, nose, etc.). Usually these points are not sufficient for a reliable tracking.
Instead, the full 3D face template (shape and appearance) can be visually well-matched to the image, because it exploits all of the available information about.

In some cases, however, local features are already enough for tracking (see the example above), and a template-based approach does not improve very much the quality of the result.




A disadvantage for template tracking is the fact that the light is distributed along the surface in a complex and nonlinear way.
Instead, for a local feature, a light variation usually can be modeled as a simple, overall brightness/contrast change (linear model).

This problem can be solved in two main ways: one is to use different appearance models onto the same 3D shape, in order to model unpredicted light variations and estimate them as well as the pose (appearance parameters).
The other consists in substituting the cost function (better called in this case “similarity function”), with a more robust one, which works correctly also when the light appearance is different from the original template.
If this function is robust enough, just one appearance template is again sufficient for tracking.

## Disadvantages of template-based tracking

**2.** Since the template is large collection of pixels (~100000), computations will be time-consuming.

The Optimization algorithm → should be fast and efficient also for a large sample of points

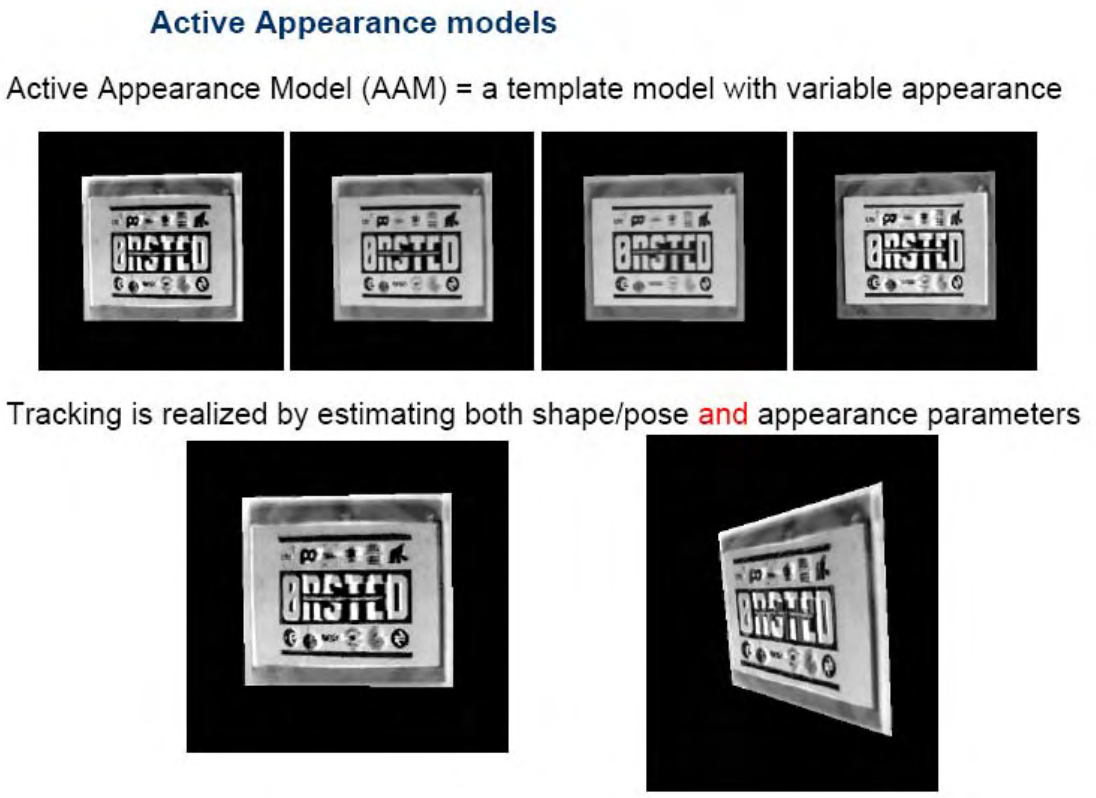
**3.** With appearance models, we still have the problem of outliers: any part of the visible surface covered by light/shadows or occlusions, is a source of outliers.


Intensity-outliers = pixels with a very different color from the predicted template.
The Cost function (*similarity* function) → should be robust to outliers!

Another disadvantage of template tracking is the computational requirements: we basically need to solve a very large LSE problem for pose estimation (and evtl. also appearance estimation, if multiple models are used). This is because we use many points, distributed along the 3D surface.
Therefore, we also need a fast and efficient optimization method.

Finally, we have to consider (as for all estimation problems) the outliers: in this case, outliers are unexpected pixel colors due to partial occlusions or light effects (spots, shadows). We call these *intensity* outliers, as opposed to the *position* outliers that we have seen for local features matching.

## Active Appearance Models

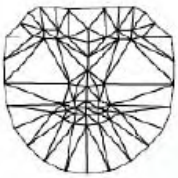



An active appearance model is a model that contains both pose and appearance parameters: in particular, appearance parameters are used to model variations due to different light exposures, different viewpoints and shadows.

## Shape Models

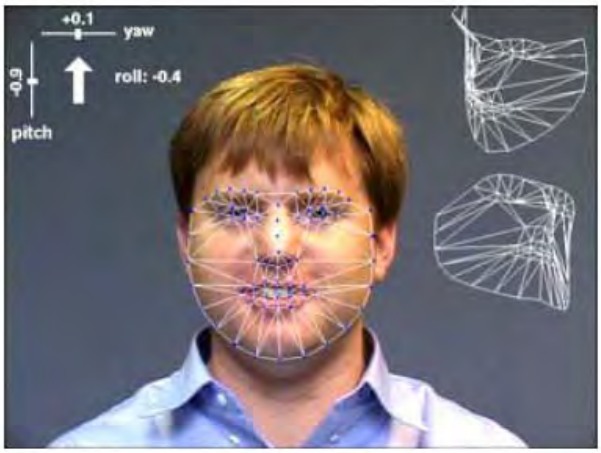
**Shape model =** a geometric model of the object (for example a CAD model).
It can be a rigid or a deformable surface.

Example: a deformable face model.
With a deformable shape we can model movements of eyes, mouth etc.
→ We have more degrees of freedom (not just 6)



Shape P = collection of vertices

$P = (x_1,y_1),\ldots,(x_M,y_M)$




The shape model of a face template usually contains parameters which can model both the overall pose (roto-translation) and individual deformations (stretch, local deformations etc.).

This corresponds to a non-rigid (deformable) model.
In the most general case, if we have a triangular mesh like the one above, we can model the shape as a collection of vertices position; for sake of simplicity, a planar deformation model is used, where vertices are defined on the (x,y) plane only, without depth information.

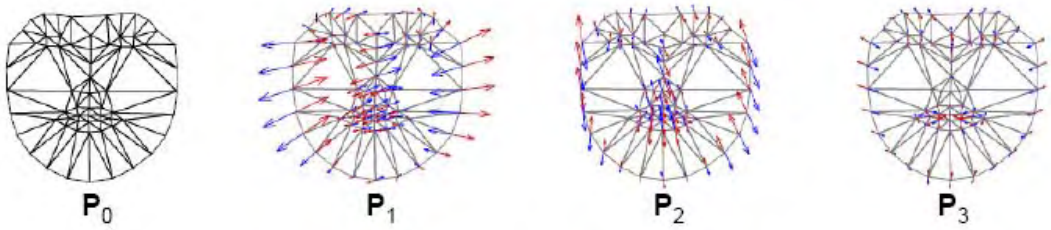
In such a model, 3D parameters are not directly estimated, but they can be subsequently obtained from the 2D deformation parameters, at a price of some complex computations more.

An alternative is to use a rigid but 3D model, which cannot accommodate expression variations (particularly the eyes and the mouth), and therefore can be a bit weaker.

## Shape Models

We can model a deformable shape **P** by using a base shape and a set of $N_p$ allowed deformations of the vertices.

Shape **P** = Base shape $\mathbf{P}_0$ + deformation vectors $\mathbf{P}_1, \mathbf{P}_2, \ldots$




→ In this way, we use a linear model: the shape is a linear combination:

$$\mathbf{P} = \mathbf{P}_0 + p_1\mathbf{P}_1 + p_2\mathbf{P}_2 + p_3\mathbf{P}_3 + \ldots$$

with $N_p$ shape parameters **p**.

Of course, this is not 3D tracking!

But the result can be used to obtain a 3D pose (with some computations more...)

If we just define as “shape” the set of all vertices of the mesh, as if they were independent one another, we would end up with a too large pose space, which makes no sense: the vertices can move only in “realistic” ways, and they are dependent one another.

Therefore, a first step is to identify and to model the allowed deformations for our object to be tracked. The result is a lower dimension “shape-space”, where deformations can be expressed as a linear combination of a base shape + base deformation vectors, in a limited number.

The pose (or better, shape) parameter is then given by a vector **p** in $N_p$ dimensions.

As already mentioned, this vector represents only a planar deformation, not 3D, so that the actual 3D pose in space (roto-translation) has to be subsequently estimated through geometric reasonings, or better through a Kalman-filter approach (see the end of Lecture ...).

**The Warp function**

## Warp function

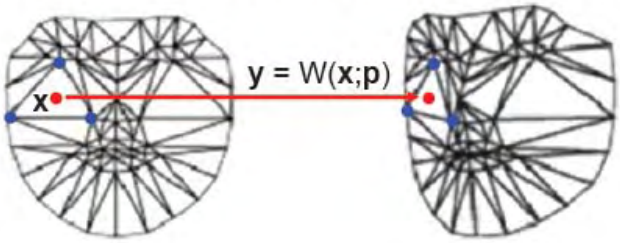
The function that maps a point **x** from the base shape to the corresponding point **y** on the shape **p** is called multi-dimensional Warp:

$$\mathbf{y} = W(\mathbf{x};\mathbf{p})$$

For example, we know the 3D/2D camera projection as Warp function, where **p** was the 3D pose (roto-translation)

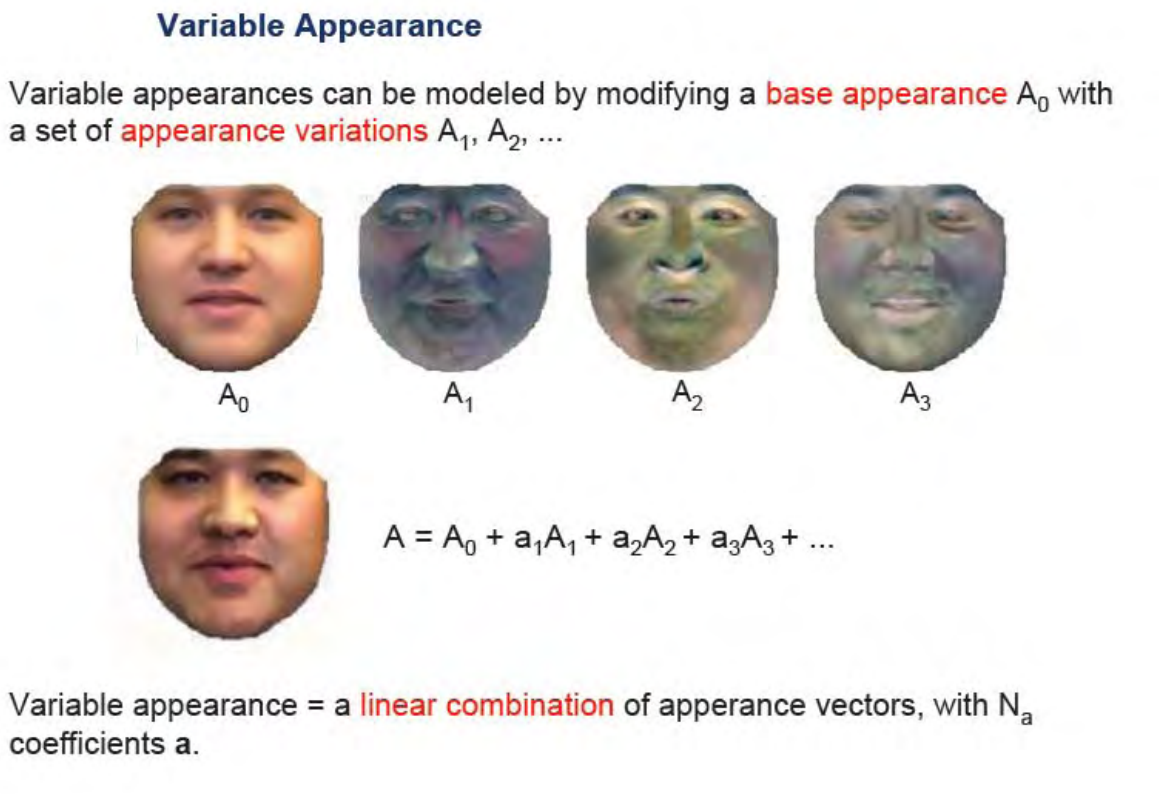
For deformable shapes, the Warp is a more complex function of the shape parameters **p**.

Example: interpolating the vertices of the triangle containing **x**.




Once we define the shape-space, we can introduce the Warp function, that maps every point x on the template to a point y on the current image, at pose hypothesis p.

Since we defined only the vertices transformation, in this case we also need to map the internal points for each triangle, which can be obtained through linear interpolation.

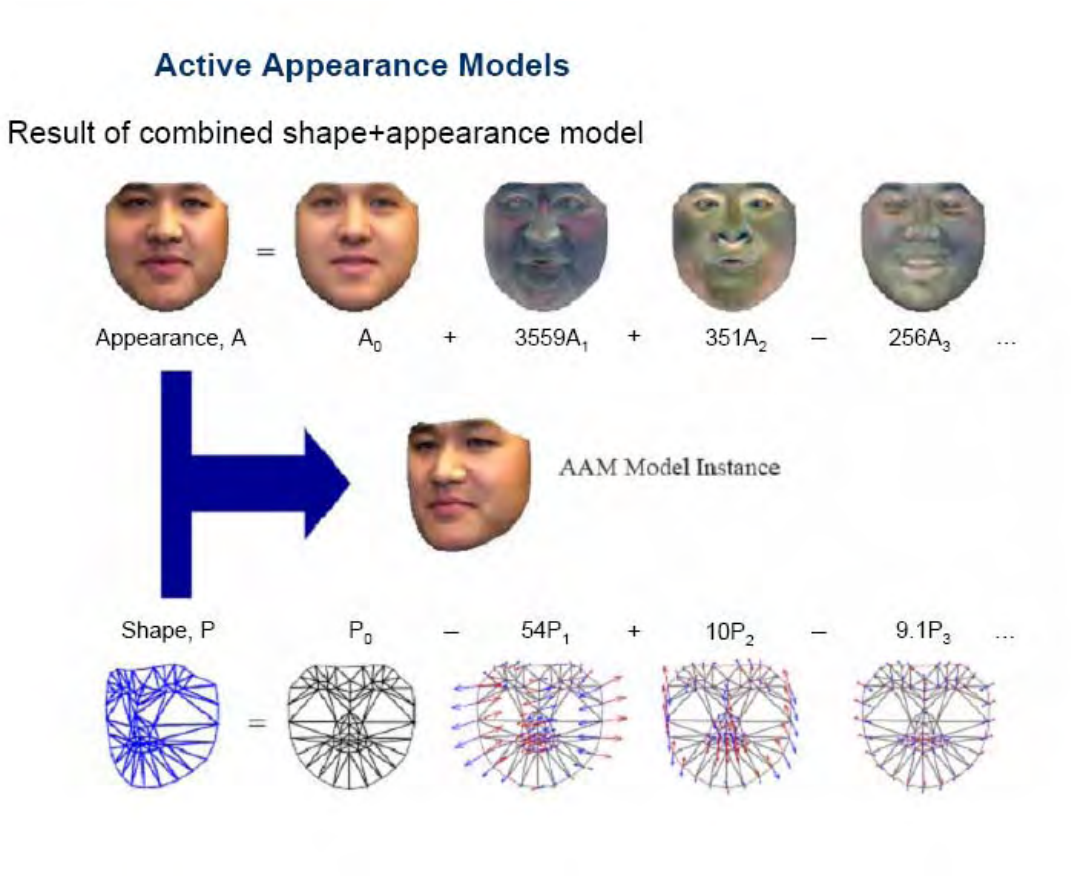



The next step for defining the model is to set up the variable appearance model.

As for the shape-space, we can define an appearance-space, which consists of the base appearance of the object, modified through a linear combination of appearance variations.

The base appearance variations should be enough to accommodate all possible (or at least, expected) variations for the specific tracking problem; but at the same time, not too many coefficients should be present, in order to avoid increasing too much the dimensionality for the estimation problem.

For both shape and appearance models, we need off-line a *training* phase: this is necessary for learning the base and the variation vectors, from a sequence of training images, taken in a variety of conditions.



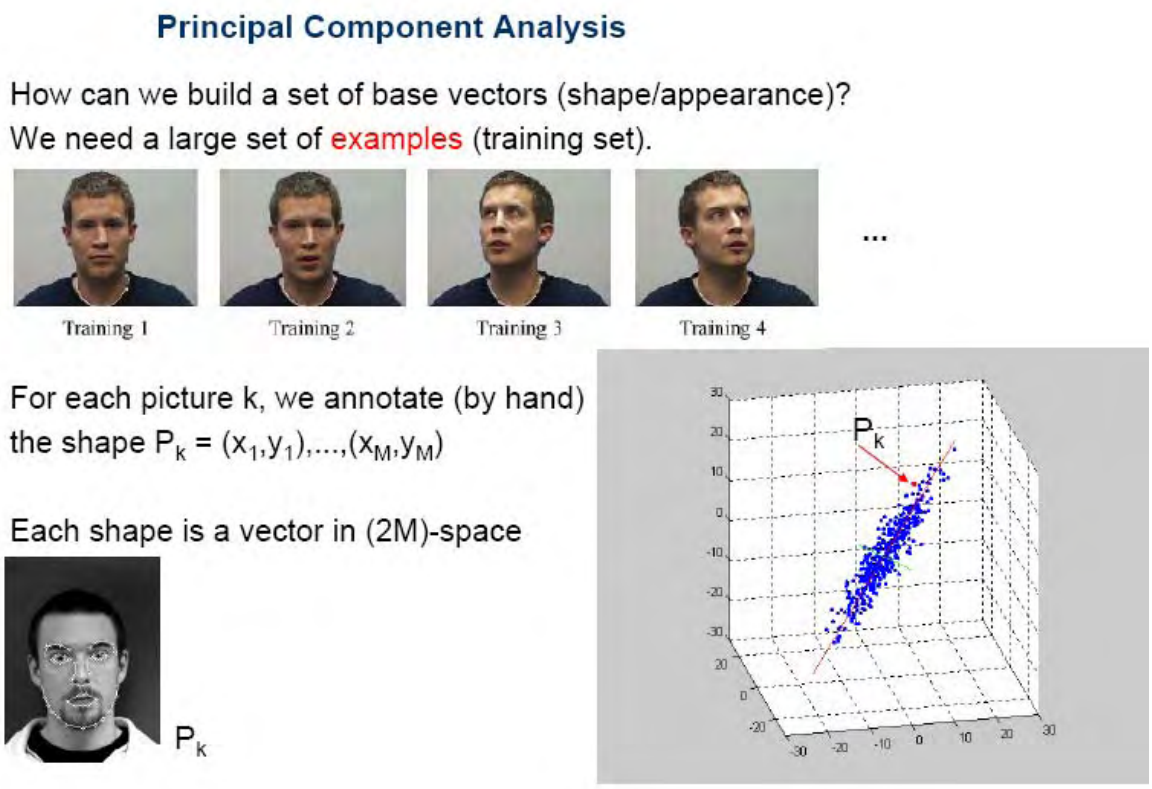

As a result, we can model any new pose, expression and light shading of the face, by combining all of the shape and appearance base vectors, and describe the new instance just through the parameters S = [P,A].

By using graphics hardware (OpenGL) we can render the new instance very quickly, by mapping the appearance A onto the triangular mesh, and attaching the texture to every triangular vertex.

This is therefore our template state-vector S, which has a much higher dimensionality than the usual 6-pose vector for 3D tracking.

### Training the AAM

***Principal Component Analysis***




The next question concerns how can we obtain the base shape and appearance vectors.

A standard procedure is depicted above.
First, we have a large set of training images of the subject we want to track. On every picture, we can annotate by hand (or with the help of semi-automatic contour matching tools) the observed shape, by matching salient features (eyes, mouth, nose, cheek contours).

This provides for each training picture the 2M-vector $P_k$.

If we represent the set of training shapes in this multi-dimensional space, usually we observe a dense *clustering* around some volume.

This means that not all shape configurations are actually possible, but only a compact set which lies in a lower-dimensional subspace.

In the picture above, in order to give the idea, we see an example in 3D space.

In particular, the situation is such that there is a principal direction (red line) along which we observe the maximal variation inside the training set, and other directions (green) where the sample variation is very small.

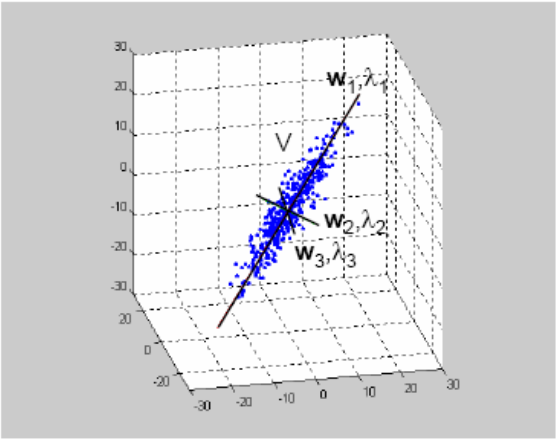
## Principal Component Analysis

On this set, we perform a PCA
**PCA = Principal Component Analysis**

Example of PCA: the set of points V is expressed into an orthogonal basis of 3 vectors: $\mathbf{w}_1,\mathbf{w}_2,\mathbf{w}_3$.

$\mathbf{w}_k$ = principal directions

$\lambda_k$ = principal components (singular values)




Idea of PCA: Only two principal directions ($\mathbf{w}_1,\mathbf{w}_2$) have meaningful values $\lambda_1,\lambda_2$

→ The basis ($\mathbf{w}_1,\mathbf{w}_2$) is sufficient to describe the training set V properly

In such a situation, it is possible to reduce the dimensionality of the 2M space, by finding the so-called principal directions of the training set.

For this purpose, the mathematical tool is called PCA: Principal Component Analysis.

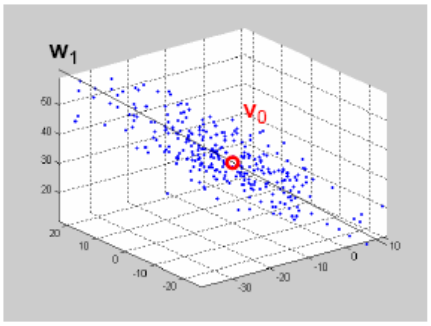
The idea behind PCA is simple: to search for the the orthogonal directions (axes) in space where the sample set has maximal variation (principal components), and discard the remaining axes.

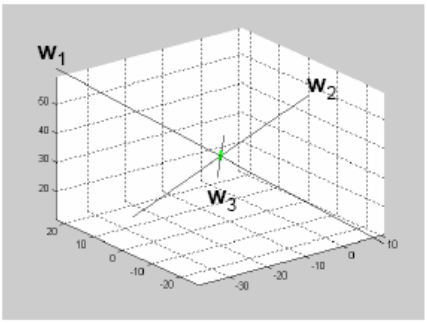
This will give an approximate description of the training set in terms of a smaller number of dimensions, that will be our linear, orthogonal basis for shape modeling.

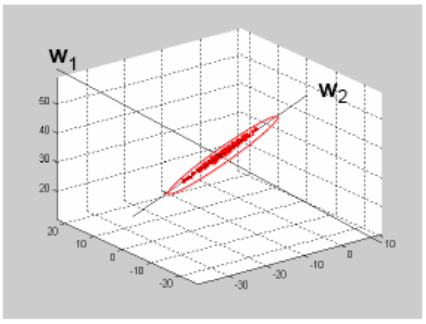
## PCA: example

### PCA Procedure

- Remove the average value from $v_k \rightarrow v_k - v_0$
- Find the maximum variance direction: $w_1$
- Remove the $w_1$ component from all data: $(1 - (v_k^T * w_1)) * v_k$ for each $v_k$
  The remaining data lie on a plane orthogonal to $w_1$
- Find the maximum variance direction: $w_2$
- Remove the $w_2$ component: $(1 - (v_k^T * w_2)) * v_k$
- $w_3$ is the remaining direction




PCA usually is performed in a sequential way:

- First, the mean point of the sample set is computed (average) v0. This will be the origin of the new coordinate system

- Afterwards, the principal axis w1, going through v0, is found: this is the direction where, when every point is projected along the line, maximal data variance is observed

- Then, the main component w1 is subtracted from all data points, that will be therefore projected in a subspace, orthogonal to w1 (here, it is a plane)

- The same procedure is repeated in this subspace, and the main direction w2 is found.

- At the end, we are left with a 1-dimensional subspace, which is the last axis of the PCA basis (w3)

The result of this procedure is a new orthogonal coordinate frame, with origin in v0 and axes (w1,w2,w3), which are the principal components of our data set.

As we can see from the picture, only the first two axes (w1,w2) are sufficient to approximately describe our data set, since the third component (along w3) is much smaller.

Therefore, any point can be approximated as v ~= v0+a1w1+a2w2, with two coefficients (a1,a2) instead of three.

In high-dimensional spaces (like the shape and appearance spaces) this procedure is extremely useful, and leads to a very high reduction of dimensionality.

## Principal Components

How do we compute the orthogonal basis $\mathbf{w}_1,...,\mathbf{w}_N$?

Let V be the matrix of all training data (NxK), with the average $\mathbf{v}_0$ subtracted.

$$V = \left[(\mathbf{v}_1 - \mathbf{v}_0) \quad \cdots \quad (\mathbf{v}_K - \mathbf{v}_0)\right]$$

Theorem : The principal components are eigenvectors $\mathbf{w}_i$ and eigenvalues $\lambda_i$ of the (NxN) data covariance matrix W

$$W = V V^T = \left[(\mathbf{v}_1 - \mathbf{v}_0) \quad \cdots \quad (\mathbf{v}_K - \mathbf{v}_0)\right] \cdot \begin{bmatrix} (\mathbf{v}_1 - \mathbf{v}_0)^T \\ \cdots \\ (\mathbf{v}_K - \mathbf{v}_0)^T \end{bmatrix}$$

W is symmetric and positive definite

→ Eigenvectors are orthogonal, and all eigenvalues are positive $\lambda_1,\lambda_2,...,\lambda_N > 0$

In mathematical form, PCA is very simple to obtain, by using the covariance matrix of the sample data, W.

In fact, it can be demonstrated that the principal components (w1,w2,...) are simply the orthogonal eigenvectors of this matrix (which is symmetric and positive definite), and the principal variances of the set V around the $\mathbf{w}_i$ axes are the respective eigenvalues $\lambda_i$.

## Principal Component Analysis

**RESUME: PCA**

Given a set of K training vectors $\mathbf{v}_1,\ldots,\mathbf{v}_K$ in N-dimensional space

1. Compute the average $\mathbf{v}_0 = \frac{1}{K}\sum_k \mathbf{v}_{0k}$

2. Subtract the average from each vector $(\mathbf{v}_1 - \mathbf{v}_0) \quad \cdots \quad (\mathbf{v}_K - \mathbf{v}_0)$

3. Find the principal components : eigenvectors $\mathbf{w}_1,\ldots,\mathbf{w}_N$ of $(V^*V^T)$ with decreasing eigenvalues $\lambda_1>\lambda_2>\ldots>\lambda_N$

Every vector can be written as $\mathbf{v}_k = \mathbf{v}_0+a_{1,k}\mathbf{w}_1+a_{2,k}\mathbf{w}_2+\ldots+a_{N,k}\mathbf{w}_N$

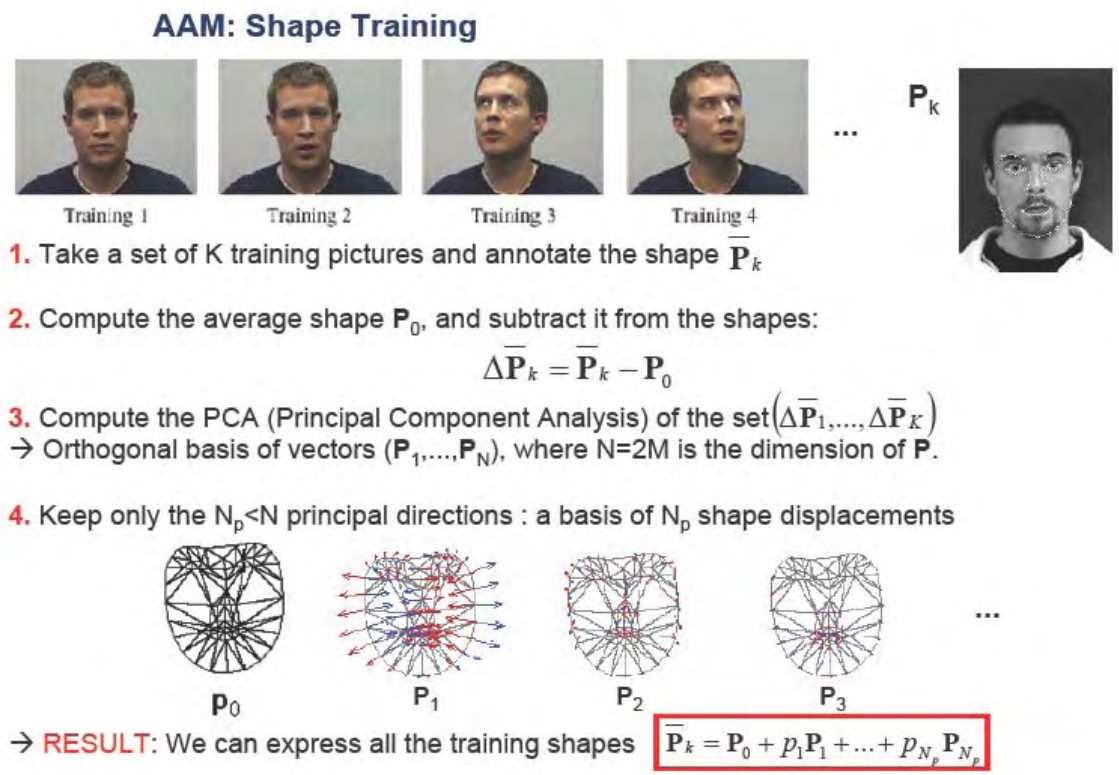
4. Keep only the first $N_p<N$ vectors $\mathbf{w}$ with large $\lambda$: $(\mathbf{w}_1,..,\mathbf{w}_{Np})$ so that the residual error is lower than a threshold (example: < 1%)

→ We approximate $\mathbf{v}_k \sim \mathbf{v}_0+a_{1,k}\mathbf{w}_1+a_{2,k}\mathbf{w}_2+\ldots+a_{Np,k}\mathbf{w}_{Np}$

Therefore, we can do a PCA of our training set, by discarding the directions w with lowest eigenvalues, in such a way to have an approximation error less than 1% (for example) over the whole set.

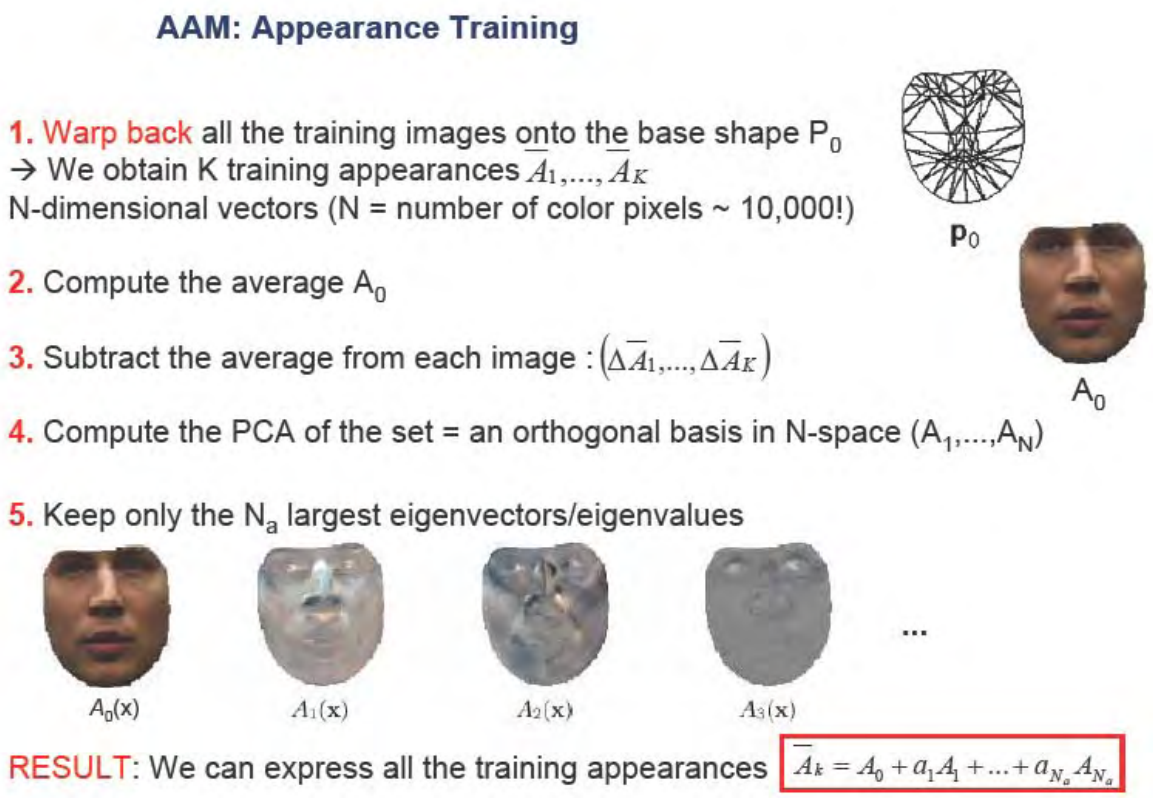
The result is a set of $N_p$ vectors $(w_1,..,w_{Np})$ plus the average vector v0.

***Shape model training with PCA***




For the shape case, the PCA procedure gives a base shape P0 and the principal deformation vectors Pi, and any new shape (vertices configuration) P will be expressed as a linear combination of Np vectors in the N=2M space, where Np<<N.

***Appearance model training***

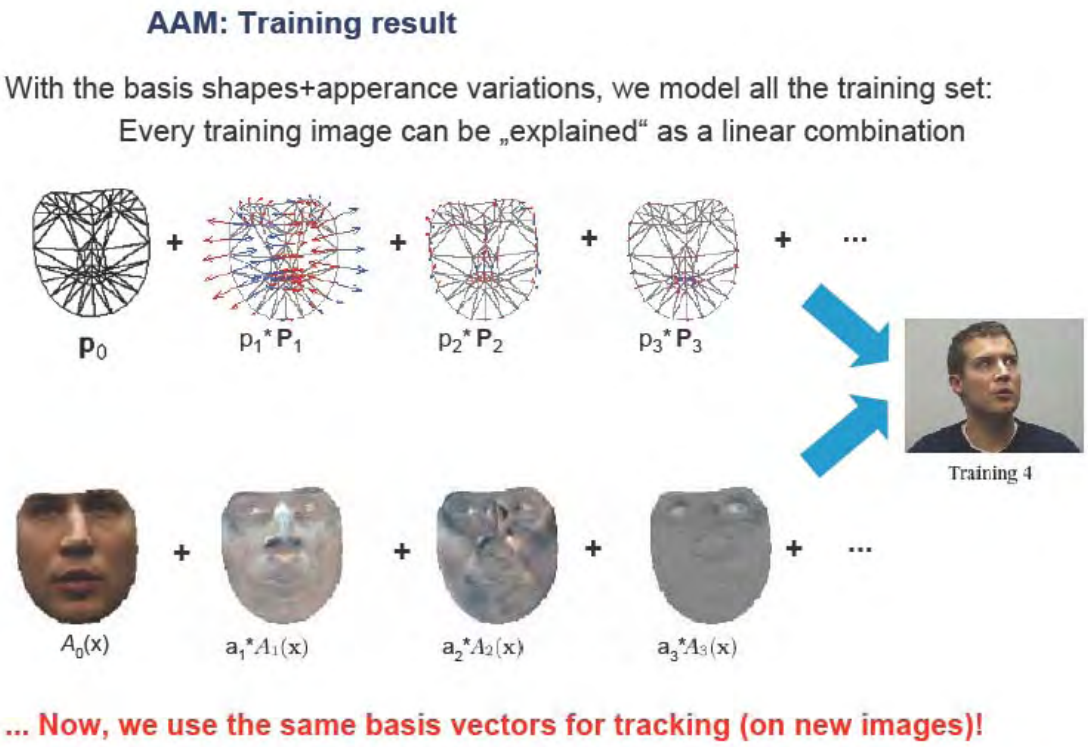



A very similar procedure can be done for training the appearance space.

In particular, for this task we first need to align all the appearances to the same shape; this can be done by warping back all the training image, from the respective annotated shapes to the average shape P0.

After all images are aligned, we can store each observed appearance in a very long vector of color pixels, which is the appearance vector A.

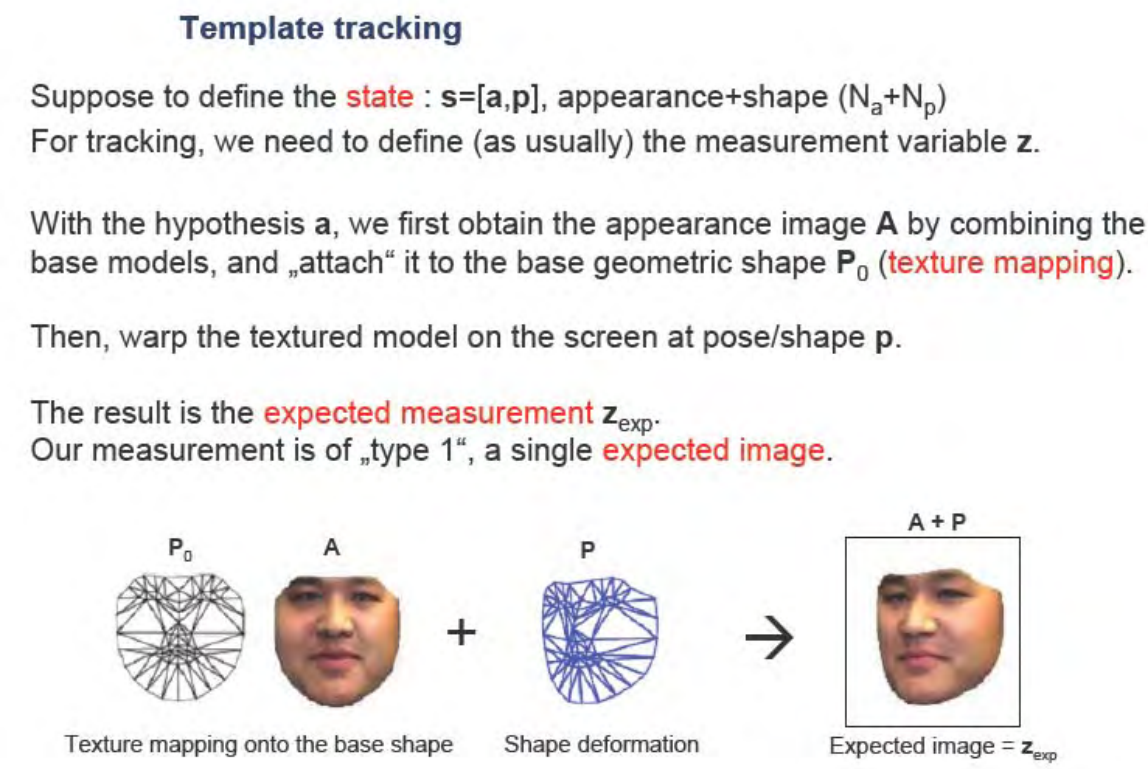
Here the use of PCA is even more important, since the A space has a very high dimension N (around 10,000 color pixels), but the observed appearances actually are distributed only along the first, very few principal directions in this space (Na), as we expect.




At the end of both PCA procedures, we finally have the two linear bases, which can be used for tracking the same face in new images.

### Tracking an AAM

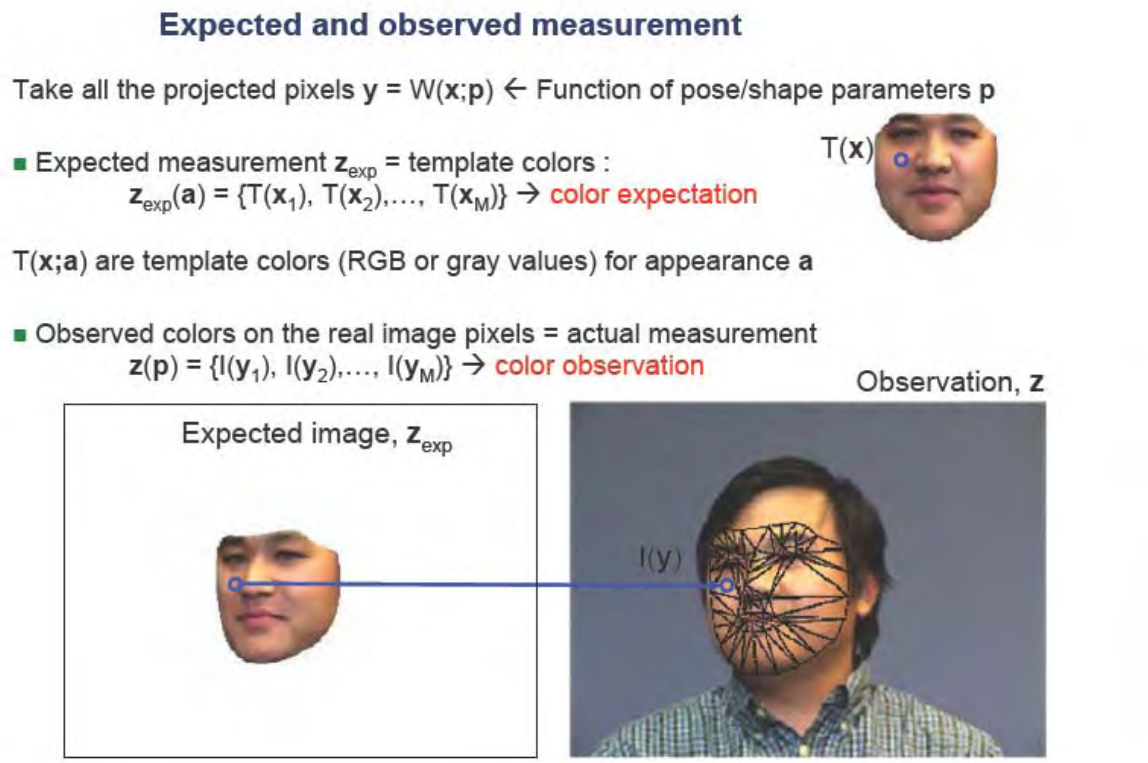
***State definition***




In Active Appearance Models, the state variable is a joint vector s=[a,p] which contains both appearance and shape parameters.

For a hypothesis s, we have an expected measurement zexp, which is the template image that we expect to observe if the hypothesis is the true one.
This is defined here as a pixel-level measurement.

This image can be compared with the real (observed) image z, in order to assess the Likelihood P(z|s) of the hypothesis.




In AAM, the real pose s is estimated through a single large, nonlinear LSE optimization, as we will see in the following Lecture.

For this purpose, the cost function is first defined, as the sum of squared differences (SSD) between expected and observed colors at corresponding pixel locations y.

NOTE: This is different to the SSD between expected and observed positions, that we used for point-based tracking, where the errors were defined in pixel coordinates (re-projection errors). Here, instead the error is defined in color (or intensity) space. But otherwise, the general formulation of the problem is of course the same.

Since the cost function here defines “how similar” the expected and observed images are, the SSD function is actually called “similarity” function.

***Optimization of similarity measures***

### Similarity measures

In order to estimate the state, now we need to define the cost function between expected and observed measurement = template similarity function

1. Sum of Squared Differences = The „classical“ function

$$E(\mathbf{a};\mathbf{p}) = \left\|\mathbf{z}_{\exp}(\mathbf{a}) - \mathbf{z}(\mathbf{p})\right\|^2 = \sum_{\mathbf{x}} \left\|T(\mathbf{x};\mathbf{a}) - I(W(\mathbf{x};\mathbf{p}))\right\|^2$$

→ We have a non-linear LSE problem

2. Robust SSD (ρ = Tukey or Huber function) → Better for outliers

$$E(\mathbf{a};\mathbf{p}) = \sum_{\mathbf{x}} \rho\left(\left\|T(\mathbf{x};\mathbf{a}) - I(W(\mathbf{x};\mathbf{p}))\right\|\right)$$

3. Normalized Cross Correlation (NCC) → More robust than SSD, but more complex to compute and to optimize

4. Mutual Information → Much more general and robust, but also very complex to compute and optimize (there are still no real-time applications available!)

For template matching we have more options to define the similarity function.
One is, as we said, classical SSD in color space.

A more robust solution for outliers uses also M-estimators.

But since here the color correspondence problem is much more critical (observed colors can be very different from the template), better and more general similarity functions have been defined.

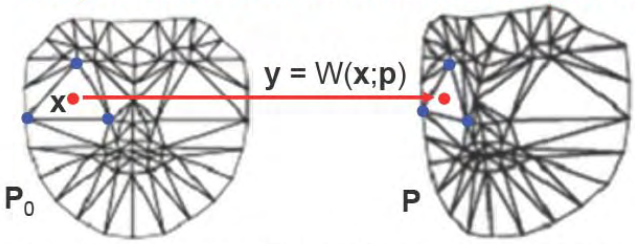
We will see in the last Lecture a very powerful one, called Mutual Information, that is used for multi-modal medical images registration (CT vs. MRI, PET etc.) where the similarity problem is much more difficult to solve than optical images.

## *Lecture 12 – The Lucas-Kanade Algorithm for template tracking*

### Piece-wise affine Warp for deformable templates

**The Warp function**

**Warp function for deformable templates**

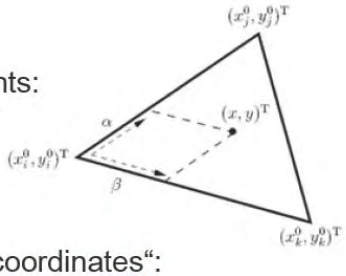



**1.** We obtain the new shape vertices **P** from $\mathbf{P}_0$ as a linear combination:

$$\mathbf{P} = \mathbf{P}_0 + p_1\mathbf{P}_1 + p_2\mathbf{P}_2 + \ldots + p_{Np}\mathbf{P}_{Np}$$

**2.** The triangle vertices are used to map also the interior points:

For a point **x** in $\mathbf{P}_0$, we first look for the triangle around **x**
The original triangle is $(x_i^0, y_i^0), (x_j^0, y_j^0), (x_k^0, y_k^0)$




→ Internal points **x** can be expressed in „triangular coordinates“:

$$\mathbf{x} = (x, y) = (x_i^0, y_i^0) + \alpha(x_j^0 - x_i^0, y_j^0 - y_i^0) + \beta(x_k^0 - x_i^0, y_k^0 - y_i^0)$$

The base shape deformation model is linear with respect to the vertices (P), but in order to map also interior points for each triangle, we need to use some kind of interpolation.

In this case, a very simple interpolation scheme uses so-called “triangular coordinates”: every internal point (x,y) can be expressed with two positive coordinates (α,β) such that (α+β) is between 0 and 1.

In particular, the 3 vertices are given by (0,0), (0,1) and (1,0).

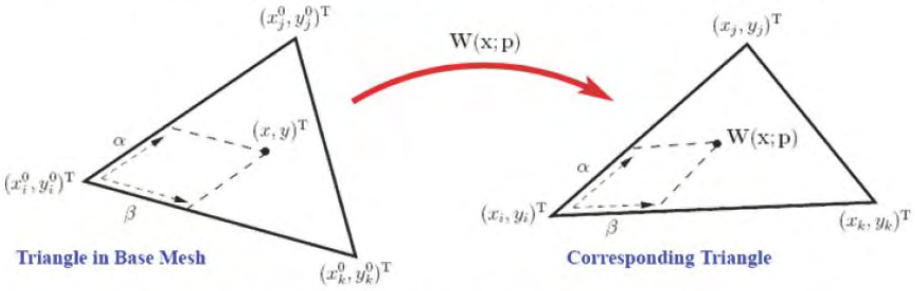
## The Warp function

We have

$$\alpha = \frac{(x - x_i^0)(y_k^0 - y_i^0) - (y - y_i^0)(x_k^0 - x_i^0)}{(x_j^0 - x_i^0)(y_k^0 - y_i^0) - (y_j^0 - y_i^0)(x_k^0 - x_i^0)} \quad \beta = \frac{(y - y_i^0)(x_j^0 - x_i^0) - (x - x_i^0)(y_j^0 - y_i^0)}{(x_j^0 - x_i^0)(y_k^0 - y_i^0) - (y_j^0 - y_i^0)(x_k^0 - x_i^0)}$$

The same coefficients (α,β) are used also for the modified triangle:

$$(x_i^0, y_i^0), (x_j^0, y_j^0), (x_k^0, y_k^0) \rightarrow (x_i, y_i), (x_j, y_j), (x_k, y_k)$$




Finally, we get the new coordinates

$$W(\mathbf{x};\mathbf{p}) = (x_i, y_i) + \alpha(x_j - x_i, y_j - y_i) + \beta(x_k - x_i, y_k - y_i)$$

For every point (x,y) the two coordinates (α,β) can be obtained with the formula above.

In this way, every internal point can be warped into the new triangle, by keeping the same coordinates (α,β), of course now referred to the new vertices.

The overall function W(x,p) therefore maps every point of the surface mesh to a new position, in 2D space:

- First the vertices $P_i$ are linearly modified through the shape coefficients **p**
- Then every internal point (α,β) of each triangle is warped to the corresponding triangle point.

## Piece-wise Affine Warp

**This is a piece-wise affine Warp**

$$W(\mathbf{x};\mathbf{p}) = (x_i, y_i) + \alpha(x_j - x_i, y_j - y_i) + \beta(x_k - x_i, y_k - y_i)$$

- **Affine:** for a given point **x** is a (linear + constant) transformation in **p**

because the new vertices are affine functions of the old ones, through **p**

$$\mathbf{P} = \mathbf{P}_0 + p_1\mathbf{P}_1 + p_2\mathbf{P}_2 + \ldots + p_{Np}\mathbf{P}_{Np}$$

$$(x_i, y_i), (x_j, y_j), (x_k, y_k) \leftarrow (x_i^0, y_i^0), (x_j^0, y_j^0), (x_k^0, y_k^0)$$

- **Piece-wise:** each point **x** has different coefficients in **p**, because:

- It has different $(\alpha,\beta)$

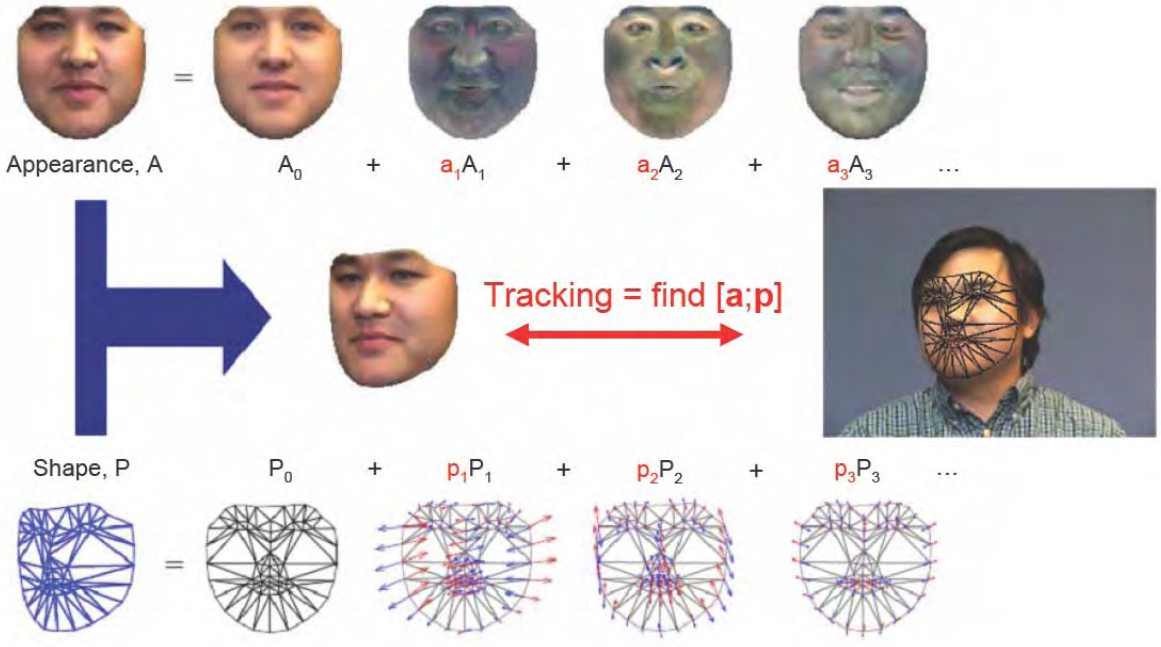
- And possibly a different base triangle $(x_i^0, y_i^0), (x_j^0, y_j^0), (x_k^0, y_k^0)$

Such a warp function is called piece-wise affine, because for a given point (x,y) the mapping is affine (linear+constant) with respect to the shape coefficients p, but the affine relationship (coefficients) are different for different points $(\alpha,\beta)$ or a different triangle (i,j,k).

## Template Tracking with Active Appearance Model

After obtaining the base shapes and appearances from training images...

→ Template tracking: estimate the (**a**;**p**) coefficients for a new image



As already stated in the previous Lecture, AAM tracking consists in estimating both shape and appearance parameters [**a**,**p**] in a new image, by using the piece-wise affine warp with coefficients **a** and **p** for every visible point (x,y) of the base mesh (template).

**Two steps: estimating pose and appearance parameters**

### Lucas-Kanade Algorithm

If the cost function is the standard SSD, starting from initial state $\mathbf{s}_0 = (\mathbf{a}_0, \mathbf{p}_0)$
We search for

$$(\mathbf{a}^*;\mathbf{p}^*) = \arg\min_{(\mathbf{a};\mathbf{p})} \sum_{\mathbf{x}} \left\| T(\mathbf{x};\mathbf{a}) - I(W(\mathbf{x};\mathbf{p})) \right\|^2$$

This is not in standard LSE form, because the state is present in both terms!

**A first, intuitive solution: split the problem in two LSE**

**1.** Optimize first the shape, with fixed appearance $\mathbf{a}_0$.

$$\mathbf{p}^* = \arg\min_{\mathbf{p}} \sum_{\mathbf{x}} \left\| T(\mathbf{x};\mathbf{a}_0) - I(W(\mathbf{x};\mathbf{p})) \right\|^2$$

→ This is Lucas-Kanade algorithm (= Gauss-Newton)

**2.** Afterwards, fix the shape in $\mathbf{p}^*$ and optimize over the appearance

$$\mathbf{a}^* = \arg\min_{\mathbf{a}} \sum_{\mathbf{x}} \left\| T(\mathbf{x};\mathbf{a}) - I(W(\mathbf{x};\mathbf{p}^*)) \right\|^2$$

The AAM estimation is a large, nonlinear LSE problem, that in principle could be solved by using the Gauss-Newton algorithm, starting from an initial estimate $[a_0,p_0]$ for shape and appearance parameters (for example, obtained from the previous frame estimation).

But here we have an additional problem: shape and appearance parameters are on two different parts of the cost function, therefore this is not a standard form for optimization.

A first idea would be, intuitively, to split the problem in two parts, by solving first for the shape parameters p (for example) with fixed appearance $a_0$, and afterwards fixing the shape on the estimated value p* and optimizing over a.

The first problem would be a standard LSE, that can be solved with Gauss-Newton: this is also known, in the template matching literature, as Lucas-Kanade algorithm.

The second problem would be even simpler (linear LSE) since the dependence of the template on the appearance parameters a is linear (see the previous picture).

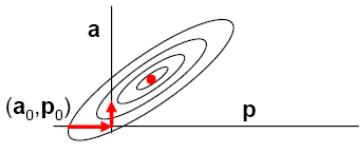
### *Caveat*

**But, be careful:**

**This is not the optimal solution of the original problem.**

**Reason: in general, we have**

$$\min_{(\mathbf{a};\mathbf{p})}\left[E(\mathbf{a},\mathbf{p})\right] \neq \min_{\mathbf{a}}\left[\min_{\mathbf{p}} E(\mathbf{a},\mathbf{p})\right]$$




**...unless we do the „right“ subdivision (in two problems)**

**→ The appearance subspace projection does the work**

Unfortunately, we cannot operate in this way, because this is not the minimum in the joint [a,p] space. This would correspond to minimize the function on two orthogonal subspaces, that is the pure appearance (a) and pure shape (p) spaces, respectively.

But still, the idea of splitting the AAM problem into two steps can be applied, if we do the “correct” subdivision, that means, by operating on the correct orthogonal subspaces (appearance subspace projection).

We will see this strategy later on.
Now we talk about the first idea, of optimizing the shape parameters p only.

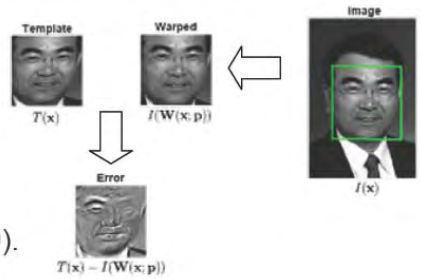
**The Lucas-Kanade algorithm for pose estimation**

## Lucas-Kanade Algorithm

### Lucas-Kanade = Template Pose estimation with LSE

The LSE problem is:

$$\mathbf{p}^* = \arg\min_{\mathbf{p}} \sum_{\mathbf{x}} \left\| I(W(\mathbf{x};\mathbf{p})) - T(\mathbf{x}) \right\|^2$$




Error = Difference between the template color and the corresponding color on the image pixel (SSD).

Gauss-Newton solution:

- Write the shape **p** as increment from the initial guess: $\mathbf{p} = \mathbf{p}_0 + \Delta\mathbf{p}$
- The error is $E(\mathbf{p}_0 + \Delta\mathbf{p}) = \sum_{\mathbf{x}} \left[ I(W(\mathbf{x};\mathbf{p}_0 + \Delta\mathbf{p})) - T(\mathbf{x}) \right]^2$
- Linearize the error around $\mathbf{p}_0$, and solve the LSE problem in $\Delta\mathbf{p}$
- Update parameters $\mathbf{p}_0 + \Delta\mathbf{p} \rightarrow \mathbf{p}_1$

...until convergence ($||\Delta\mathbf{p}|| < \varepsilon$)

The LSE problem is solved, as usually, by linearizing the cost function around the current pose estimation, for a small increment Δp, and iterating again the procedure until convergence.

This is very similar to the KLT optimization method (which actually bears the same author names); the main difference is that now we have a more general Warp function, instead of the simple translation (2dof) or affine (6dof) models; and, of course, the size of the template is much larger than the small features (25x25) used for point-based tracking.

## Lucas-Kanade Algorithm

More precisely:

$$E(\Delta\mathbf{p}) = \left\|\mathbf{f}(\mathbf{p}_0 + \Delta\mathbf{p}) - \mathbf{T}\right\|^2$$

With

$$\mathbf{f}(\mathbf{p}) = \begin{bmatrix} I(W(\mathbf{x}_1;\mathbf{p})) \\ \cdots \\ I(W(\mathbf{x}_N;\mathbf{p})) \end{bmatrix} \qquad \mathbf{T} = \begin{bmatrix} T(\mathbf{x}_1) \\ \cdots \\ T(\mathbf{x}_N) \end{bmatrix}$$

Linearization:

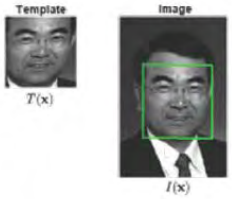
$$E(\mathbf{p}_0 + \Delta\mathbf{p}) \approx \left\|\mathbf{f}(\mathbf{p}_0) + \left.\frac{\partial \mathbf{f}}{\partial \mathbf{p}}\right|_{\mathbf{p}_0} \Delta\mathbf{p} - \mathbf{T}\right\|^2 = \left\|\mathbf{f}(\mathbf{p}_0) + J(\mathbf{p}_0)\Delta\mathbf{p} - \mathbf{T}\right\|^2$$

Gauss-Newton step:

$$\Delta\mathbf{p} = (J^T J)^{-1} J^T (\mathbf{T} - \mathbf{f})$$

## Lucas-Kanade Algorithm

### Computation of Jacobian Matrix

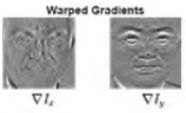



The linearization gives

$$E(\Delta\mathbf{p}) = \sum_{\mathbf{x}} \left[ \nabla I(W(\mathbf{x};\mathbf{p}_0)) \cdot \left.\frac{\partial W}{\partial \mathbf{p}}\right|_{(\mathbf{x};\mathbf{p}_0)} \cdot \Delta\mathbf{p} + I(W(\mathbf{x};\mathbf{p}_0)) - T(\mathbf{x}) \right]^2$$

Where:

- $\nabla I(\mathbf{y})$ = (1x2) image gradients (gray-scale), warped in $\mathbf{y}=W(\mathbf{x};\mathbf{p}_0)$




- $\frac{\partial W}{\partial \mathbf{p}}(\mathbf{x})$ = ($2xN_p$) Jacobians of the Warp in $\mathbf{x}$, with respect to the shape $\mathbf{p}$

Example: For a 6-dof Warp they are the usual Jacobian matrices (2x6)

The products are also called steepest-descent images $\nabla I \frac{\partial W}{\partial \mathbf{p}}(\mathbf{x}) = \left[ \nabla I \frac{\partial W}{\partial p_1}(\mathbf{x}), \ldots, \nabla I \frac{\partial W}{\partial p_{N_p}}(\mathbf{x}) \right]$


By writing the linearized cost function E(Δp), we can see how the general computation of the Jacobian at pose $p_0$ is obtained by multiplying, for every point x of the template, the projected image gradient (1x2) in y = W(x,$p_0$) with the (2xNp) Jacobian matrix of the Warp.

The result of this product can be graphically represented as a set of images, which are also called steepest-descent images.

## Lucas-Kanade Algorithm

The overall Jacobian ($N \times N_p$) is $J(\mathbf{p}) = \begin{bmatrix} \nabla I \dfrac{\partial W}{\partial \mathbf{p}}\Big|_{(\mathbf{x}_1;\mathbf{p})} \\ \cdots \\ \nabla I \dfrac{\partial W}{\partial \mathbf{p}}\Big|_{(\mathbf{x}_M;\mathbf{p})} \end{bmatrix}$

And the ($N_p \times N_p$) Gauss-Newton matrix is:

$$G(\mathbf{p}) = J^T(\mathbf{p})J(\mathbf{p}) = \sum_{\mathbf{x}} \left( \nabla I \frac{\partial W}{\partial \mathbf{p}} \right)^T \left( \nabla I \frac{\partial W}{\partial \mathbf{p}} \right)$$

So, we can write the solution $\Delta\mathbf{p}^*$ as:

$$\Delta\mathbf{p}^* = (J^T J)^{-1} J^T (\mathbf{T} - \mathbf{f}) = G^{-1} \sum_{\mathbf{x}} \left( \nabla I \frac{\partial W}{\partial \mathbf{p}} \right)^T \left[ T(\mathbf{x};\mathbf{a}) - I(W(\mathbf{x};\mathbf{p})) \right]$$

Finally, we can put everything in a more compact matrix form, and compute the Gauss-Newton step as above indicated.

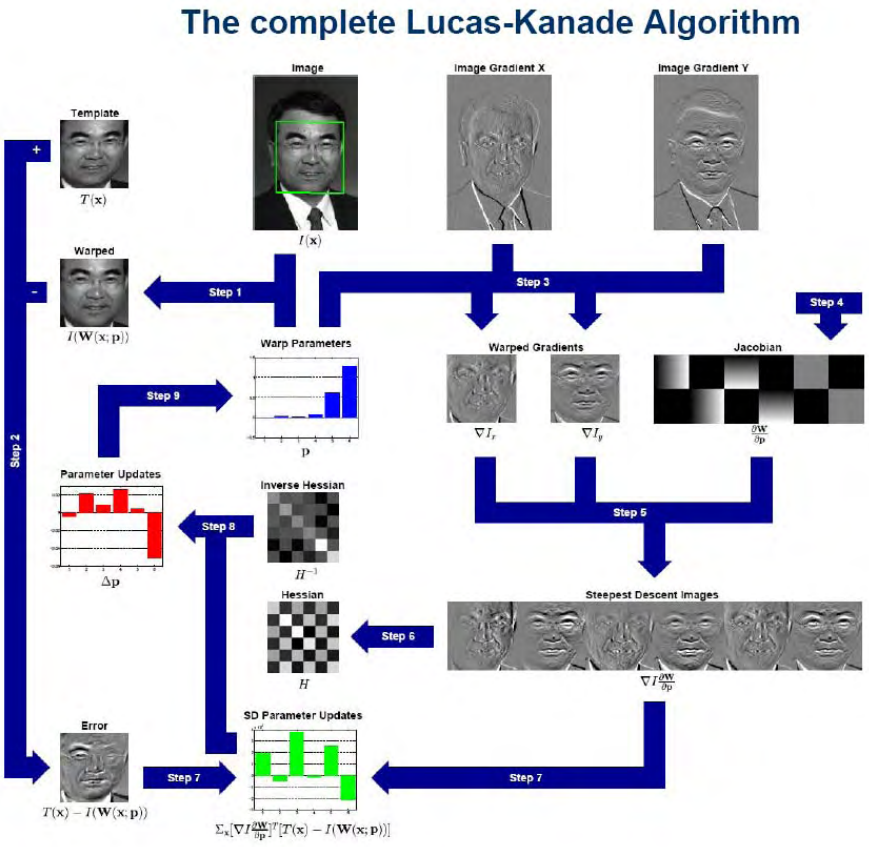
## The complete Lucas-Kanade Algorithm

Iterate: at pose **p**

For every pixel **x** of the template T(**x**), do

1. Compute the Warp W(**x**;**p**), and get the image gray values I(W(**x**;**p**))
2. Compute the difference (error image) T(**x**)-I(W(**x**;**p**))
3. Compute the image gradient $\nabla I$ at pixel W(**x**;**p**)
4. Evaluate the Jacobian matrices of the Warp $\frac{\partial W}{\partial \mathbf{p}}$ in (**x**;**p**)
5. Multiply (3) with (4) → Steepest-descent images
6. Accumulate the Gauss-Newton Matrix $G(\mathbf{p}) = \sum_{\mathbf{x}} \left( \nabla I \frac{\partial W}{\partial \mathbf{p}} \right)^T \left( \nabla I \frac{\partial W}{\partial \mathbf{p}} \right)$
7. Accumulate the second term: $J^T(\mathbf{T}-\mathbf{f}) = \sum_{\mathbf{x}} \left( \nabla I \frac{\partial W}{\partial \mathbf{p}} \right)^T \left[ T(\mathbf{x}) - I(W(\mathbf{x};\mathbf{p})) \right]$
8. After accumulating all terms (6), and (7), compute the increment $\Delta\mathbf{p} = G^{-1} J^T (\mathbf{T}-\mathbf{f})$
9. Update **p** → **p**+Δ**p**

...until ||Δ**p**||<ε



As we can see, this algorithm has a high computational complexity, because every quantity (Warp Jacobians, image gradients, and the Gauss-Newton matrix) has to be computed for each template point **x**, and again for each pose $p_i$ during the optimization.

In particular, the most expensive step is the Gauss-Newton matrix computation (Step 6), that requires for each template point x to add an ($N_p$x$N_p$) contribution.

### Speed of Lucas-Kanade

Problem: the sample size for this Gauss-Newton optimization (number of points **x**) is extremely large!

So, how fast is Lucas-Kanade for real applications?

The traditional version (as described before) is slow, because each quantity must be updated at each Gauss-Newton iteration!

In particular:

- The image gradients $\nabla I(W(\mathbf{x};\mathbf{p}))$ and Warp Jacobians $\left.\frac{\partial W}{\partial \mathbf{p}}\right|_{(\mathbf{x};\mathbf{p})}$ depend on **p**
- The Gauss-Newton matrix G, as a consequence, must always be re-computed

→ And this is the heavy part of the computation!

The problem comes from the fact that every quantity inside Lucas-Kanade depends on the shape p. Therefore, the general algorithm is very slow, and can be applied only to smaller and simpler problems like the KLT tracker (a few Warp parameters $N_p$, and a small number of template points N).

**First speed improvement: the forwards-compositional approach**

## Lucas-Kanade = Forwards-Additive Approach

## How can we improve Lucas-Kanade?

There are different ways to formulate the optimization problem.
The main difference is the update rule.

Lucas-Kanade is also called forwards-additive optimization approach:

- Forwards: the direct Warp function is used: points **x** are mapped from the template to the image, via the Warp **y** = W(**x**;**p**+Δ**p**)

$$\Delta \mathbf{p} = \arg\min_{\Delta \mathbf{p}} \sum_{\mathbf{x}} \left[ I(W(\mathbf{x}; \mathbf{p} + \Delta \mathbf{p})) - T(\mathbf{x}) \right]^2$$

- Additive: the parameters update Δ**p** are just added to the current **p**:

$$\mathbf{p}_0 + \Delta \mathbf{p} \rightarrow \mathbf{p}_1$$

We can improve Lucas-Kanade, if we try to modify the update rule (Step 9 of the algorithm).

In fact, in Gauss-Newton we solve the problem by updating p in an additive way: we compute a shape increment Δp, and add it to the previous estimate $p_0$.

This approach can also be called forwards-additive.

## Forwards-Compositional Approach

### First modification: Forwards-Compositional

IDEA: We can change the update rule in the following way

- Instead of adding an increment $\mathbf{p}+\Delta\mathbf{p}$ and computing the Warp $W(\mathbf{x};\mathbf{p}+\Delta\mathbf{p})$
- We can do a composition of warps: $W(W(\mathbf{x};\Delta\mathbf{p});\mathbf{p})$

...under the hypothesis that zero Warp is the identity Warp

$$W(\mathbf{x};\mathbf{0}) = \mathbf{x}$$

→ For a null increment $\Delta\mathbf{p}=\mathbf{0}$, we have the original warp $W(\mathbf{x};\mathbf{p})$ ← OK

Now, we search for $\Delta\mathbf{p}$ to minimize the cost function:

$$\Delta\mathbf{p} = \arg\min_{\Delta\mathbf{p}} \sum_{\mathbf{x}} \left[I(W(W(\mathbf{x};\Delta\mathbf{p});\mathbf{p}) - T(\mathbf{x})\right]^2$$

There is another way to do the optimization, which can be proven to work equivalently (convergence properties), if we restrict a little bit the set of allowed Warp functions.

This approach is called compositional: instead of adding the incremental parameter Δp, we compose the Warp with itself.

In this case, of course the meaning of the update Δp is not anymore of an “increment”, but for sake of simplicity we can still call it in this way.

In order to apply this approach, we require for the Warp an additional property: for a null shape vector Δp=0, we should obtain the identity function W(x,0) = x.

In other words, the null Warp should give the base template T unmodified, which is a property satisfied by several common Warps (like the piece-wise affine function up to now considered).

In this way, we can write the cost function in terms of this composition, instead of the additive increment, and try to minimize it with respect to Δp, as always, by linearization around Δp=0.

## Forwards-Compositional Approach

This is called compositional approach, because we compose the Warp with itself:

$$W(\mathbf{x};\mathbf{p}) \circ W(\mathbf{x};\Delta\mathbf{p})$$

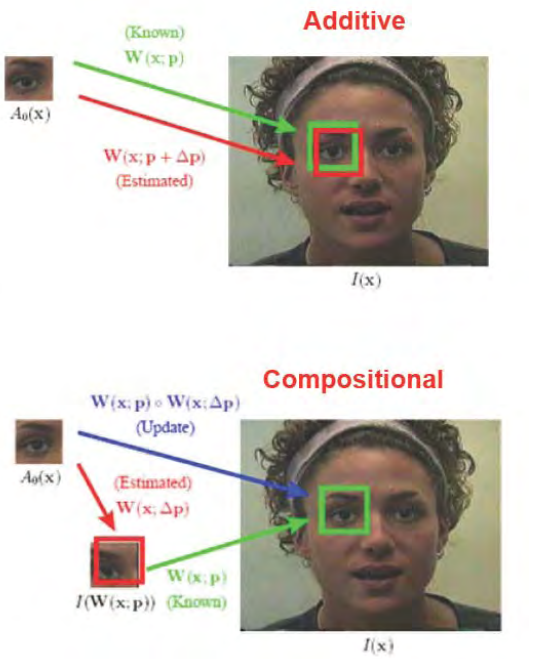
That means

1. Warp the template by a small increment

$\mathbf{x}' = W(\mathbf{x};\Delta\mathbf{p})$

2. Compose the incremental Warp with the large one:

$\mathbf{y} = W(\mathbf{x}';\mathbf{p}) = W(W(\mathbf{x};\Delta\mathbf{p});\mathbf{p})$




We call this approach forwards-compositional, since we warp the points x forwards (from the template T to the image I) but using a compositional update rule.

## Forwards-Compositional Approach

### Update rule

Update = From $\mathbf{p}_0$ and $\Delta\mathbf{p}$, find $\mathbf{p}_1$

BUT: now the meaning of $\Delta\mathbf{p}$ is different!

Instead of adding $\mathbf{p}_0+\Delta\mathbf{p} \rightarrow \mathbf{p}_1$

We must compose the Warp: $W(\mathbf{x};\mathbf{p}_0) \circ W(\mathbf{x};\Delta\mathbf{p}) \rightarrow W(\mathbf{x};\mathbf{p}_1)$

That is: we must find the new parameters $\mathbf{p}_1$ such that

$W(\mathbf{x};\mathbf{p}_1) = W(W(\mathbf{x};\Delta\mathbf{p});\mathbf{p}_0)$ for all $\mathbf{x}$!

Is this possible?

→ The answer is yes, for a large class of Warp functions.

In particular, for the piece-wise affine Warp used in AAM this is possible.

After an optimization step, this time we cannot use the result $\Delta p$ as an additive value $p_1=p_0+\Delta p$, because it does not represent an increment.

Instead, we need to compose the Warp to find the new parameter $p_1$: that is, to find p1 such that the new function $W(x,p_1)$ is equal to $W(W(x,\Delta p),p_0)$ for every point x of the template.

This looks rather complex to compute, but for a large class of Warps is actually simple, and always possible. In particular, it can be shown that for the piece-wise affine Warp this can be done.

## Forwards-Compositional Approach

The linearized cost function becomes

$$E(\Delta\mathbf{p}) = \sum_{\mathbf{x}} \left[ \nabla I(W(\mathbf{x};\mathbf{p})) \cdot \left.\frac{\partial W}{\partial \mathbf{p}}\right|_{(\mathbf{x};\mathbf{0})} \cdot \Delta\mathbf{p} + I(W(\mathbf{x};\mathbf{p})) - T(\mathbf{x}) \right]^2$$

Difference with the previous approach: now the Jacobian of the Warp $\left.\frac{\partial W}{\partial \mathbf{p}}\right|_{(\mathbf{x};\mathbf{0})}$ is evaluated at $\Delta\mathbf{p}=\mathbf{0}$!

→ Therefore, it is constant, and can be pre-computed (off-line) for all points **x**

All the rest is the same (compute the Gauss-Newton matrix, etc.) and the result is

$$\Delta\mathbf{p}^* = G^{-1} \sum_{\mathbf{x}} \left( \nabla I \frac{\partial W}{\partial \mathbf{p}} \right)^T \left[ T(\mathbf{x}) - I(W(\mathbf{x};\mathbf{p})) \right]$$

with the only difference in $\left.\frac{\partial W}{\partial \mathbf{p}}\right|_{(\mathbf{x};\mathbf{0})}$

The advantage of the compositional approach lies in a simpler optimization step: in fact, by linearizing the cost function E w.r.t. Dp, we see that the Warp Jacobians are evaluated always at pose p=0.

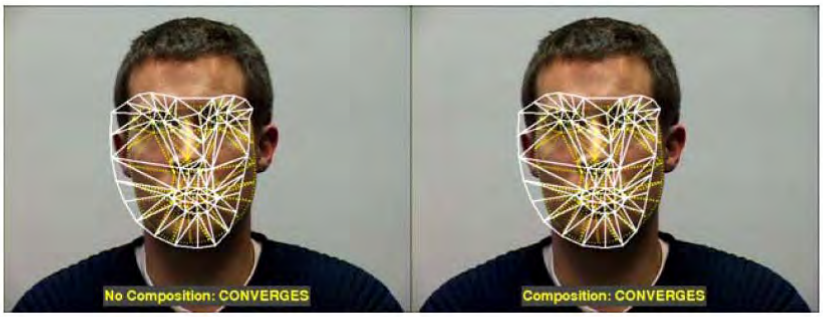
This is a big advantage, since all of these N matrices, with size (2xNp), can now be pre-computed (off-line) and stored for each point x of the base template.

Instead, the image gradients must be still evaluated for every new pose p, since they are computed at the warped image locations y = W(x,p).

## Properties

Convergence properties of compositional approach

It can be proven that this formulation converges to the correct solution as well as the additive one (and sometimes even better) ← See [Baker,Matthews]




- Advantage: off-line evaluation of Warp Jacobian matrices
- Disadvantage: a more complex update step (composition istead of addition)

Overall, this is a faster algorithm!

The advantage of having an off-line evaluation of Warp Jacobians is by far bigger than the disadvantage of a more complex update rule (compositional) for the new parameters $p_0 \rightarrow p_1$.

Therefore, whenever applicable, the forwards-compositional approach is much faster.

**Second improvement: the inverse-compositional approach**

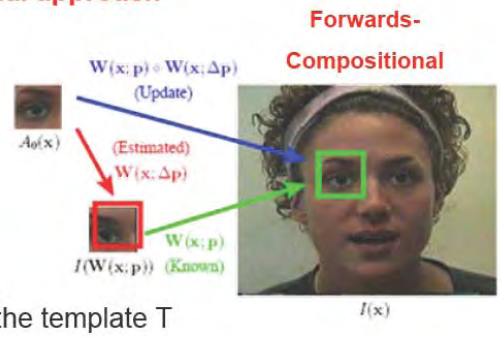
## Inverse-Compositional Approach

### Inverse-Compositional approach

Now we go a step further :

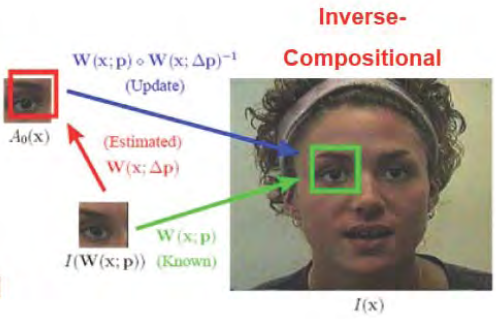
Use the inverse incremental Warp: $W^{-1}(\mathbf{x};\Delta\mathbf{p})$

$$\Delta\mathbf{p} = \arg\min_{\Delta\mathbf{p}} \sum_{\mathbf{x}} \left[ I(W(W^{-1}(\mathbf{x};\Delta\mathbf{p});\mathbf{p}_0)) - T(\mathbf{x}) \right]^2$$




which is equivalent to moving the direct Warp into the template T

$$\Delta\mathbf{p} = \arg\min_{\Delta\mathbf{p}} \sum_{\mathbf{x}} \left[ I(W(\mathbf{x};\mathbf{p}_0)) - T(W(\mathbf{x};\Delta\mathbf{p})) \right]^2$$




After minimization, the update rule is:

$$W(\mathbf{x};\mathbf{p}_0) \circ W^{-1}(\mathbf{x};\Delta\mathbf{p}) \rightarrow W(\mathbf{x};\mathbf{p}_1)$$

This is OK only if W can be composed and inverted

A step further, in the direction of a faster Lucas-Kanade optimization, consists now in considering an inverse-compositional approach.

If we do some further assumptions about the Warp, we can in fact invert it w.r.t. p, and therefore compose the inverse Warp in Δp with the base one.

This is possible for a more restricted set of warps, and it gives a more complex update rule for obtaining p1 from $p_0$ and Δp.

In this way, we can move the “incremental warp” W(x,Δp) from the image I to the template T.

## Inverse-Compositional Approach

The linearized error becomes

$$E(\Delta\mathbf{p}) = \sum_{\mathbf{x}} \left[ I(W(\mathbf{x};\mathbf{p})) - T(\mathbf{x}) - \nabla T(\mathbf{x}) \cdot \left. \frac{\partial W}{\partial \mathbf{p}} \right|_{(\mathbf{x};0)} \cdot \Delta\mathbf{p} \right]^2$$

**This is very good : all derivatives can be computed off-line**

Off-line: $G = \sum_{\mathbf{x}} \left( \nabla T \left. \frac{\partial W}{\partial \mathbf{p}} \right|_{(\mathbf{x};0)} \right)^T \left( \nabla T \left. \frac{\partial W}{\partial \mathbf{p}} \right|_{(\mathbf{x};0)} \right)$

On-line: (note the different sign!) $\Delta\mathbf{p} = G^{-1} \sum_{\mathbf{x}} \left( \nabla T \frac{\partial W}{\partial \mathbf{p}} \right)^T \left[ I(W(\mathbf{x};\mathbf{p})) - T(\mathbf{x}) \right]$

Update: find $\mathbf{p}_1$ such that, for every $\mathbf{x}$, $W(\mathbf{x};\mathbf{p}_1) = W(W^{-1}(\mathbf{x};\Delta\mathbf{p});\mathbf{p}_0)$

- Advantage: **a lot** of computations are off-line

  ← Overall: definitely faster
- Disadvantage: the update rule is quite complex

The advantage for optimization is here even bigger, since now the linearized cost function contains most terms that depend only on the template points T(x), and not on the pose p.

Therefore, all of the Gauss-Newton matrix now can be computed off-line, and only few terms remain in the on-line optimization step.

## Inverse-Compositional Approach

Off-line:

For every pixel **x** of the template T(**x**), do

1. Compute the template gradient $\nabla T(\mathbf{x})$ at **x**
2. Evaluate the Jacobian matrices of the Warp $\frac{\partial W}{\partial \mathbf{p}}$ in (**x**;**0**)
3. Multiply (1) with (2) : $\nabla T \frac{\partial W}{\partial \mathbf{p}}$
4. Accumulate the Gauss-Newton Matrix $G = \sum_{\mathbf{x}} \left( \nabla T \frac{\partial W}{\partial \mathbf{p}} \right)^T \left( \nabla T \frac{\partial W}{\partial \mathbf{p}} \right)$

Iterate: at pose **p**

For every pixel **x** of the template T(**x**), do

5. Warp the point W(**x**;**p**), and get the image gray value I(W(**x**;**p**))
6. Compute the difference (error image) T(**x**)-I(W(**x**;**p**))
7. Accumulate the second term: $\sum_{\mathbf{x}} \left( \nabla T \frac{\partial W}{\partial \mathbf{p}} \right)^T [I(W(\mathbf{x};\mathbf{p})) - T(\mathbf{x})]$
8. After accumulating all terms, compute the increment $\Delta \mathbf{p} = G^{-1} \sum_{\mathbf{x}} \left( \nabla T \frac{\partial W}{\partial \mathbf{p}} \right)^T [I(W(\mathbf{x};\mathbf{p})) - T(\mathbf{x})]$
9. Update: find $\mathbf{p}_1$ such that for all **x**: $W(\mathbf{x};\mathbf{p}_1) = W(W^{-1}(\mathbf{x};\Delta\mathbf{p});\mathbf{p}_0)$

...until $||\Delta\mathbf{p}|| < \varepsilon$

## Complete cost function

Up to now, we have seen how to optimize the SSD error with respect to the shape parameters **p**, by fixing the appearance **a**=$\mathbf{a}_0$.

$$\mathbf{p}^* = \arg\min_{\mathbf{p}} \sum_{\mathbf{x}} \left\| T(\mathbf{x};\mathbf{a}_0) - I(W(\mathbf{x};\mathbf{p})) \right\|^2$$

This was Lucas-Kanade (or one of the variants).

Now we want to optimize the full cost function

$$(\mathbf{a}^*;\mathbf{p}^*) = \arg\min_{(\mathbf{a};\mathbf{p})} \sum_{\mathbf{x}} \left\| T(\mathbf{x};\mathbf{a}) - I(W(\mathbf{x};\mathbf{p})) \right\|^2$$

The template appearance T is given by $T(\mathbf{x};\mathbf{a}) = A_0(\mathbf{x}) + \sum_{i=1}^{N_a} a_i A_i(\mathbf{x})$

So the „real“ cost function is $E(\mathbf{a};\mathbf{p}) = \sum_{\mathbf{x}} \left\| A_0(\mathbf{x}) + \sum_{i=1}^{N_a} a_i A_i(\mathbf{x}) - I(W(\mathbf{x};\mathbf{p})) \right\|^2$

In order to optimize over the full pose+appearance space, now we need to consider the dependence of the template T on the appearance parameters, $a_i$, which we kept fixed in the Lucas-Kanade approach.

The complete cost function is more complex, and not in standard LSE form anymore.

### The complete cost function

So, the next question is: how can we optimize the complete function in (**a**;**p**)?

First, we can write it in a more compact way: $\left\| \mathbf{A}_0 + \sum_{i=1}^{N_a} a_i \mathbf{A}_i - \mathbf{I}(\mathbf{p}) \right\|^2$

With (very) large vectors $\mathbf{A}_i$ and **I**

$$\mathbf{A}_0 = \begin{bmatrix} A_0(\mathbf{x}_1) \\ \cdots \\ A_0(\mathbf{x}_M) \end{bmatrix}; \mathbf{A}_1 = \begin{bmatrix} A_1(\mathbf{x}_1) \\ \cdots \\ A_1(\mathbf{x}_M) \end{bmatrix}; \cdots \qquad \mathbf{I}(\mathbf{p}) = \begin{bmatrix} I(W(\mathbf{x}_1;\mathbf{p})) \\ \cdots \\ I(W(\mathbf{x}_M;\mathbf{p})) \end{bmatrix}$$

So that we have an error vector **E**(**a**;**p**) that we want to minimize (in norm):

$$\arg\min_{(\mathbf{a};\mathbf{p})} \left\| \mathbf{E}(\mathbf{a};\mathbf{p}) \right\|^2 \qquad \mathbf{E}(\mathbf{a};\mathbf{p}) = \mathbf{A}_0 + \sum_{i=1}^{N_a} a_i \mathbf{A}_i - \mathbf{I}(\mathbf{p})$$

In the compact notation, we can see how the parameters are “distributed” between the two terms (shape and appearance).

**Appearance estimation: the appearance subspace decomposition**

## Appearance subspaces

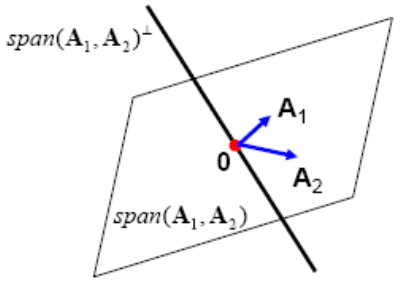
### Orthogonal appearance subspaces

Next, we take the basis appearances $\mathbf{A}_i$, and we consider the linear sub-space of $R^M$ that they generate :

$span(\mathbf{A}_i)$ : { $\mathbf{V} = a_1\mathbf{A}_1+a_2\mathbf{A}_2+...+a_{Na}\mathbf{A}_{Na}$ }

This space has dimension $N_a$<M

Example: $R^3$, with $N_a$=2:
The vectors ($\mathbf{A}_1$,$\mathbf{A}_2$) generate a plane




For each subspace of $R^M$ with dimension $N_a$, there is always an orthogonal complement:

$span(\mathbf{A}_i)^{\perp}$ = {$\mathbf{W} \perp \mathbf{V}$} ← dimension = M-$N_a$

$span(\mathbf{A}_i)^{\perp}$ = The set of all vectors **W** orthogonal to every vector **V** of $span(\mathbf{A}_i)$

$$R^M = span(\mathbf{A}_i)+span(\mathbf{A}_i)^{\perp}$$

So every vector in M-space is a unique sum **E** = **V**+**W**

In order to solve the problem, we need first to introduce a more general concept: the orthogonal decomposition of a linear space.

In fact, for a given vector space E, and a given subspace V, there always exist a unique orthogonal space $V^{\perp}$, which has the property $E = V + V^{\perp}$.

In other words, every vector **e** of E can be expressed as a unique sum of two orthogonal components, one in V and the other in $V^{\perp}$, which are the two projections of **e** onto the two subspaces.

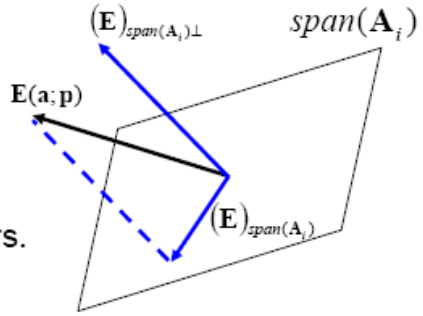
In our case, the appearance base vectors (excluding $A_0$), generate a linear subspace of $R^M$, the space of all possible appearances span($A_i$).
Therefore, there exists a unique orthogonal complement to the appearance subspace (the “impossible” appearances), so that an arbitrary pixel pattern can be expressed as the sum of a possible appearance (combination of $A_i$) plus an orthogonal term.

## Error decomposition

The solution now is to split the error vector **E** in two orthogonal components:

$$\left(\mathbf{E}\right)_{span(\mathbf{A}_i)} = \sum_{i=1}^{N_a} a_i \mathbf{A}_i; \quad a_i = \left(\mathbf{A}_i^T \cdot \mathbf{E}\right)$$

$$\left(\mathbf{E}\right)_{span(\mathbf{A}_i)\perp} = \mathbf{E} - \left(\mathbf{E}\right)_{span(\mathbf{A}_i)}$$




These components are the projections of **E** on two orthogonal subspaces → they are orthogonal vectors.

$$\mathbf{E} = \left(\mathbf{E}\right)_{span(\mathbf{A}_i)} + \left(\mathbf{E}\right)_{span(\mathbf{A}_i)\perp}$$

So, we decompose **E** as

$$\mathbf{E} = \left(\mathbf{A}_0 + \sum_{i=1}^{N_a} a_i \mathbf{A}_i - \mathbf{I}(\mathbf{p})\right)_{span(\mathbf{A}_i)} + \left(\mathbf{A}_0 + \sum_{i=1}^{N_a} a_i \mathbf{A}_i - \mathbf{I}(\mathbf{p})\right)_{span(\mathbf{A}_i)\perp}$$

And then

$$\left\|\mathbf{E}\right\|^2 = \left\|\mathbf{A}_0 + \sum_{i=1}^{N_a} a_i \mathbf{A}_i - \mathbf{I}(\mathbf{p})\right\|^2_{span(\mathbf{A}_i)} + \left\|\mathbf{A}_0 + \sum_{i=1}^{N_a} a_i \mathbf{A}_i - \mathbf{I}(\mathbf{p})\right\|^2_{span(\mathbf{A}_i)\perp}$$

By doing this decomposition, we can therefore decompose the error vector, **E**(a,p) as the sum of two terms, obtained by projecting E onto span($A_i$) and the orthogonal space, respectively.

Since the two components are orthogonal, we can also say that the overall error norm (=SSD) is the sum of the two separate norms.

## Error decomposition

The second component simplifies :

$$\left(\mathbf{A}_0 + \sum_{i=1}^{N_a} a_i \mathbf{A}_i - \mathbf{I}(\mathbf{p})\right)_{span(\mathbf{A}_i)\perp} = \left(\mathbf{A}_0 - \mathbf{I}(\mathbf{p})\right)_{span(\mathbf{A}_i)\perp}$$

because it belongs to the space orthogonal to the $\mathbf{A}_i$

→ Therefore, is not function of **a**.

The first component depends on both **a** and **p**

$$\left(\mathbf{A}_0 + \sum_{i=1}^{N_a} a_i \mathbf{A}_i - \mathbf{I}(\mathbf{p})\right)_{span(\mathbf{A}_i)} = \left(\mathbf{A}_0 - \mathbf{I}(\mathbf{p})\right)_{span(\mathbf{A}_i)} + \sum_{i=1}^{N_a} a_i \mathbf{A}_i$$

BUT:

For any **p***, the minimum value over **a** is always zero! (it has exact solution):

for i=1,...,$N_a$ $\boxed{a_i^* = \mathbf{A}_i^T \cdot (\mathbf{I}(\mathbf{p}^*) - \mathbf{A}_0)}$

If we consider the second projection (orthogonal to the $A_i$), the middle term inside just disappears, and therefore it depends only on p, and not on a.

Instead, the other projection still contains both (a,p) parameters; but, for any fixed value of p*, we can see that its minimum value is always zero, that is, it has always an exact solution in a*.
In particular, since this is a linear function of a, the solution to the appearance problem is obtained immediately, in one step.

## Two-Step (Shape+Appearance) optimization

So, we have this situation: we look for $\min_{(\mathbf{a};\mathbf{p})} E(\mathbf{a};\mathbf{p}) = \left[f(\mathbf{p}) + g(\mathbf{a},\mathbf{p})\right]$

Where for every $\mathbf{p}^*$, $\min_{\mathbf{a}} g(\mathbf{a},\mathbf{p}^*) = 0$

And then: $\min_{(\mathbf{a};\mathbf{p})}\left[f(\mathbf{p}) + g(\mathbf{a},\mathbf{p})\right] = \min_{\mathbf{p}}\left(f(\mathbf{p})\right)$ ⟹ $\mathbf{p}^*$ (minimizer of f) is also minimizer of E!

Now we can **correctly** split the optimization in two steps:

1. $\mathbf{p}^* = \arg\min_{\mathbf{p}} f(\mathbf{p})$
2. $\mathbf{a}^* = \arg\min_{\mathbf{a}} g(\mathbf{a},\mathbf{p}^*)$

→

1. $\mathbf{p}^* = \arg\min_{\mathbf{p}} \left\|\mathbf{A}_0 - \mathbf{I}(\mathbf{p})\right\|^2_{span(\mathbf{A}_i)\perp}$
2. $a_i^* = \mathbf{A}_i^T \cdot \left(\mathbf{I}(\mathbf{p}^*) - \mathbf{A}_0\right)$

This decomposition, as we can see, allows to split the joint (shape+appearance) estimation problem in the correct way.

1 - In the first problem, we solve for p*, by using Lucas-Kanade *onto the subspace* span($A_i$)$^\perp$

2 - Afterwards, we solve for a* in one step, with shape fixed to p*.

And (a*,p*) is the true optimizer of the joint SSD error function E(a,p).

**Modified Lucas-Kanade for pose+appearance estimation**

### Lucas-Kanade with sub-space projection

The first optimization is just Lucas-Kanade (shape only, with fixed base appearance $\mathbf{A}_0$).

$$\mathbf{p}^* = \arg\min_{\mathbf{p}} \left\| \mathbf{A}_0 - \mathbf{I}(\mathbf{p}) \right\|^2_{span(\mathbf{A}_i)\perp}$$

...with the only difference : the error is projected over $span(\mathbf{A}_i)^{\perp}$

→ We need to project the error **and** its derivatives into this sub-space.

The original Lucas-Kanade algorithm modifies this way (Steps 2 and 5):

At iteration (shape) **p**

**Step 2b.** Compute the Error vector and project it onto the subspace: $\left(\mathbf{A}_0 - \mathbf{I}(\mathbf{p})\right)_{span(\mathbf{A}_i)\perp}$

**Step 5b.** Compute the Error Jacobian = Steepest-Descent images and project them also onto the subspace ($N_p$ columns) $\left(\nabla I \frac{\partial W}{\partial p_j}\right)_{span(\mathbf{A}_i)\perp}$

The Lucas-Kanade algorithm has to minimize the *projection* of the error vector **E** onto a subspace, instead of the full vector **E**.

This implies only two modifications to the algorithm:

- Step 2b (error image + projection onto the subspace)

- Step 5b (steepest-descent images + projection of each image onto the subspace)

## Shape+Appearance optimization (with Lucas-Kanade)

Iterate: at pose **p** (for all points **x**), do

**1.** Compute the Warp W(**x**;**p**), and get the image gray values I(W(**x**;**p**))

**2b.** Compute the error vector and project it $[A_0(\mathbf{x})\text{-}I(W(\mathbf{x};\mathbf{p}))]_{span(A)^{\perp}}$

**3.** Compute the image gradients $\nabla I(\mathbf{x})$

**4.** Compute the Warp Jacobian $\frac{\partial W}{\partial \mathbf{p}}(\mathbf{x})$

**5.** Multiply the gradients (3) with the Jacobians (4) → Error Jacobian $\left(\nabla I \frac{\partial W}{\partial \mathbf{p}}\right)$

**5b.** Project the error Jacobian columns: $\left(\nabla I \frac{\partial W}{\partial p_j}\right)_{span(\mathbf{A}_i)^{\perp}}$

**6.** Compute the Gauss-Newton Matrix with the modified Jacobian

**7.** Compute the second term with the modified error

**8,9.** Compute the increment Δ**p** and update **p** → **p** + Δ**p**

...until $||\Delta \mathbf{p}||<\varepsilon$

→ RESULT: Optimized pose **p*** onto the subspace span($\mathbf{A}_i$)$^{\perp}$

**10.** Given **p***, optimize the appearance **a***: $a_i^* = \mathbf{A}_i^T \cdot (\mathbf{I}(\mathbf{p}^*) - \mathbf{A}_0)$

At the end, we also have the appearance parameters computation, that is solved in one step (in vector form). This is the complete AAM optimization algorithm.

## Example of AAM Tracking

In this example, we can see how the joint shape+appearance computation can give very good and realistic results, where the estimated template perfectly matches the underlying image with almost no error (bottom-rigth image).

***Improving convergence with multi-resolution***

**Improvement: multi-resolution**

Lucas-Kanade (and all the variants) can be still improved for convergence:

→ We can employ the multi-resolution approach

1. Compute two Gaussian Pyramids : Image and Template at multiple scales

2. Perform Lucas-Kanade on the lowest resolution of I and T, with initial pose $\mathbf{p}_0$

3. After optimization set $\mathbf{p}_0 = \mathbf{p}^*$, increase the resolution of I and T, and do the optimization again

...until the maximum resolution (i.e. the original image I and template T).

→ This increases both speed and robustness: estimation can start from a far initial guess $\mathbf{p}_0$, and the final number of updates is also less.

The Lucas-Kanade algorithm can also be improved by using multiple resolutions: the optimization can start from a coarse, blurred image and template, with a larger convergence region, and be repeated afterwards with increasing resolution, up to the original image and template.

Both image and appearance template, in this case, will be filtered with the same set of Gaussian filters (pyramid).

***Estimating 3D pose parameters with a combined (2D+3D) approach***

**Combined (2D+3D) AAM**

**Last (but not least…) : we are more interested in 3D tracking!**

AAM tracking gives 2D shape parameters **p**.

How can we recover the 3D pose (roto-translation) of the head from **p**?

There is a work by the same Authors [Baker, Matthews] that says how to do it.

But this is a bit complicated (and we do not have the space in this Course).
→ Whoever is interested, please refer the following paper:

**Real-Time Combined 2D+3D Active Appearance Models**
*J. Xiao, S. Baker, I. Matthews, and T. Kanade*
*Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, June, 2004

Finally, we mention here the 3D pose estimation problem.
In fact, as we have seen, with AAM we can only estimate a piece-wise affine Warp, which is a complex Warp working only in 2D. 3D estimation requires a projective model, as we know, from body space to camera image, which is non-linear and much more difficult for the optimization.

For this purpose, another idea is to use the piece-wise model up to now developed, and then extract a more compact 3D information (6 dof, global head roto-translation) out of the complex 2D deformation parameters $N_p$.

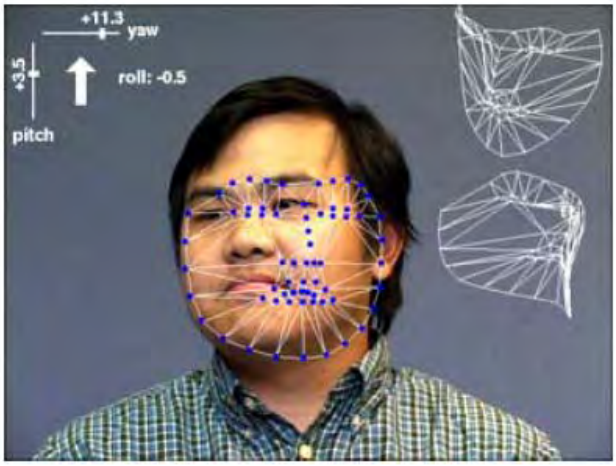
This can be done by using an Extended Kalman Filtering approach, where the 2D parameters constitute the observation vector (Z) and the 3D pose is the state (S), eventually with or without velocity component.

The proper, nonlinear measurement model, together with the 3d motion model, can be specified for this filter, by using the nonlinear 3D/2D camera projection function, and Jacobian, so that the pose S is estimated in a Bayesian tracking way.

## Combined (2D+3D) AAM

Combined 2D+3D model = after estimation of 2D shape parameters (**p**), the model is augmented with the 3D body pose.

Some single-frame estimation examples are shown below




This tracking method is called *Combined 2D+3D AAM*, where the 3D pose information is a by-product of the complete 2D affine shape parameters.

### *Lecture 13 – Robust template similarity functions*

**Robustness issues in template tracking**

## Robustness in Template Matching: outliers

In Template tracking, we need to talk about robustness

In fact, there are many situations where:

- The object can be partly occluded

(for example by other objects)

- There can be strong and unpredicted light spots, or very dark shadows

These situations are quite bad, even using active appearance models (AAM)

→ Both kind of disturbances cannot be included into the training model!

Therefore, we have outliers: a set of observed colors with a much bigger error than the others.

The problem of outliers, in template matching, can be particularly critical, since and lighting or viewpoint change of the object usually reflects in a nonlinear shading effect (dark-bright pattern).

Therefore, as we have seen, the observed appearance of the object can be very different from the original template, and this is the reason to need a multiple appearance template (AAM), in order to carry out a successful LSE pose estimation.

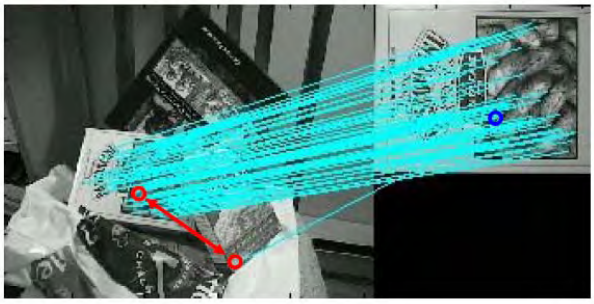
Partial occlusions can be as well a big source of outliers, and in any case pose difficult problems, since they of course cannot be modeled by the AAM.

Moreover, using an active appearance models requires in any case a training phase (PCA etc.), which can be more or less long and more or less successful.

## Intensity Outliers vs. Position Outliers

These are intensity-outliers: unexpected large color (or gray) errors.

As opposed to position-outliers: in features matching (e.g. SIFT), outliers have a very large position error, with respect to the others.



We can consider such outliers intensity-outliers, since they give big errors in color (or intensity) space, rather that re-projection errors (in 2D space), as we instead have for keypoints tracking.
And, as we can see, their percentage now can be critically high, with respect to the sample set.

## Robustness of the similarity function

Tracking fails mainly because the SSD function itself is not robust to outliers.

How can we make the template matching more robust, in presence of outliers?

SSD = At pose **p** we compute the average color (gray) squared error

$$E(T, I, \mathbf{p}) = \sum_{\mathbf{x}} \left\| T(\mathbf{x}) - I(W(\mathbf{x}; \mathbf{p})) \right\|^2$$

In template tracking, we call the cost function similarity function:

It says how much the observed image I „resembles“ the warped template at pose **p**.

The question is: do we need to use SSD as similarity function?

→ No: there are also better functions for this task.

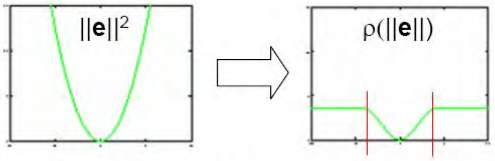
## Improving robustness of the similarity function

The question we consider now, instead, is the following one: can we use a different template matching strategy, where a simpler (possibly unique) appearance model is needed?
The answer is yes, if we substitute the cost function (traditional SSD) with something more general and more robust to this kind of outliers.

## M-Estimators

**1.** We already know M-Estimators as a robust improvement over SSD:

Examples: Tukey, Huber functions




They substitute the squared error terms $||\mathbf{e}_i||^2$ with a function $\rho(||\mathbf{e}_i||)$ which can reduce the influence of outliers

$$E(T, I, \mathbf{p}) = \sum_{\mathbf{x}} \rho \left\| T(\mathbf{x}) - I(W(\mathbf{x}; \mathbf{p})) \right\|$$

If we use M-Estimators, then Gauss-Newton can be applied (re-weighted nonlinear LSE).

...But in template matching, the outliers are more critical, and the threshold is difficult to set properly.

For a template matching we can do better.

The first idea is to use the M-estimators that we used for keypoints matching.

This has been done in several works, and improves the result very much over standard LSE. Nevertheless, the need for multiple appearance models is still critical, since a different lighting (shading) produces a very different, and nonlinear, transformation of the original template gray-pattern.

## Normalized Cross Correlation

**2.** Normalized Cross Correlation (NCC)
SSD is not robust to light changes

The Normalized Cross Correlation index is derived from SSD, in the translational case:

If the Warp is a simple translation W(**x**;**p**) = **x**-**p** the definition of NCC is

$$NCC(T,I,\mathbf{p}) = \frac{\sum_{\mathbf{x}}[T(\mathbf{x})-\bar{T}][I(\mathbf{x}-\mathbf{p})-\bar{I}]}{\sqrt{\sum_{\mathbf{x}}[T(\mathbf{x})-\bar{T}]^2 \sum_{\mathbf{x}}[I(\mathbf{x}-\mathbf{p})-\bar{I}]^2}} \qquad \bar{T} = \frac{1}{M}\sum_{\mathbf{x}} T(\mathbf{x}); \quad \bar{I} = \frac{1}{M}\sum_{\mathbf{x}} I(\mathbf{x})$$

T I I

NCC is invariant when brightness (I=T-c) and contrast (I=aT) change

...But it works only for a translational Warp, and is not so robust to occlusions
→ It is very good for local features tracking, but not for general template matching.

NCC is instead a more general measure of matching between templates, and it is widely used also for local keypoints.

The idea behind NCC is the following one: we search for the pose of the template where corresponding intensity pixels show the “most linear” intensity relationship with the image pixels.

This is more general than SSD: in fact, the latter tries to find the template pose that maximizes “similarity” (that is, equality) between pixel intensities.
If we look at the picture above, we see how the three images are related by a linear transformation (brightness+contrast change) of the kind I=aT-c. This fact maximizes NCC, but it is not good for SSD, which only searches for the “best equality” I=T.

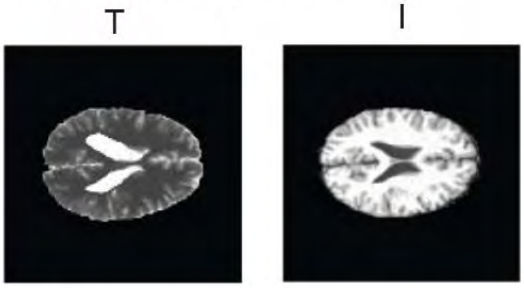
The index is of course more complex to compute and optimize, but it is very good for local features, which are small and therefore transform approximately only with linear relationships when light changes.

But unfortunately, this is not enough for a global template matching: as we can see, a global, nonplanar surface exhibits a complex transformation of intensity pattern, which is generally much nonlinear.

## Mutual Information

**3.** Mutual Information

Today, this is probably the most general measure available for image matching.
It is very much used in medical images registration:




The two images represent the same scene (a brain slice) but they are taken with different imaging devices (for example CT, MRI, PET etc.)

Multi-modal images = images obtained with different kind of sensors/devices (=modality), from the same scene or object.

If we need to match them (find the pose of T in I), we cannot use SSD, nor any robust improvement of it: the color patterns look extremely different!

→ Mutual Information can do it

We consider here a better function, which is the most general similarity index, working also with multi-modal medical images.

This is the Mutual Information similarity index.

Mutual Information allows also nonlinear relationships between template and image, because it looks for the pose where the two images show a relationship *at all* (linear or not).

For example, if we look at the example above, we have two images of the same physical thing (a given brain's slice) taken with two different procedures (CT and MRI, for example).

Since the two represent the same object, the corresponding patterns are expected to have a kind of relationship: for example, grey pixels on the left are white on the right, etc.

If they would not be correctly aligned, instead, the superimposed pixels would show less, or no relationship (=correspondence) at all.

But we do not know the relationship, and it can be nonlinear. Therefore, we need an index that is maximized when there is any kind of relationship, or *dependence*, between the patterns. This is Mutual Information.

### Mutual Information for template tracking

#### *Introduction: information theory*

In order to introduce it, we need to talk a bit about Information Theory.

## Information Theory

### Information Theory

From the WikiPedia definition:

***Information theory*** *is a discipline in applied mathematics involving the quantification of data information, with the goal of enabling as much data as possible to be reliably stored on a medium or communicated over a channel.*

The measure of data information, known as entropy, is usually expressed by the average number of bits needed for storage or communication of the message.

For example:

If a daily weather description has 3 bits of entropy, then, on the average, we can describe the daily weather with a message long no less than 3 bits per day.

Information Theory is concerned with analysis and quantification of the amount of information "contained" in a random variable.

This quantity is identified by the minimum (on the average) number of bits needed to represent a message, generated from that variable, in the most compact way as possible (compression).

This quantity is actually an average one, since we are not sure how many bits we will really need to compress this message, because it comes from a random event. But the average becomes most exact, when the length of the generated message is high (law of large numbers).

## Information Theory

### Information Theory

Mutual Information is a concept from Information Theory.

Information Theory comes from probability theory and statistics.

The most important definitions inside IT are:

### Entropy and Mutual Information

Given a random variable X = {0,1,...,N}, with probability distribution

$$P(X=0), P(X=1), ..., P(X=N)$$

- Entropy : H(X) =

1. The uncertainty we have about X (before knowing its actual value)
2. The average amount of information carried by a single outcome (value) of X

- Mutual information MI(X,Y) = the amount of information in common between two random variables, X and Y.

In order to arrive to the MI index, we need to start from the concept of Entropy.

Entropy is the *amount of uncertainty* carried by a random event x, with a given probability distribution P(x).

Before we know an outcome of x (for example, before tossing the coin), our degree of uncertainty about x is given by H(x).

After we know a particular event x, our uncertainty is reduced to 0: therefore, we also can say that this event gives us an amount H(x) of *information* about x.

NOTE: H(x) is a single number, that characterizes the whole distribution P(x). We can also say that is a *functional* quantity (function of a whole function P(x)).

Mutual Information, instead, concerns the information in common between two variables, that is, the amount of information that an output of one variable gives about the other: MI(x,y). This of course must be symmetric between x and y: MI(x,y)=MI(y,x).

*Entropy and coding*

## Entropy

### Entropy

The **entropy** *H*(X), of a discrete random variable *X* is a measure of the amount of **uncertainty** one has about the value of *X*.

For example, consider a binary variable: X = {0,1}

- If the variable has a certain value:

P(X=0) = 1; P(X=1) = 0
OR
P(X=0) = 0; P(X=1) = 1

We are sure about X in advance → H(X) = 0

- If, on the contrary, each value is equally probable (0 or 1)

P(X=0) = 0.5; P(X=1) = 0.5

We have the maximum degree of uncertainty
We cannot "bet" on any value in advance → H(X) is maximum

As a simple example of Entropy, we can consider a binary variable X={0,1}. If one of the two outcomes of X has probability 1 and the other 0, we have no uncertainty about X, and entropy is H(x)=0.
In this case, an observed value on X gives also no information, since we were already sure about it from the beginning.

If, on the opposite extreme, the two values have the same probability 0.5, then we have the maximum degree of uncertainty H(x): any of the two can happen, and we have no clue in advance (we cannot "bet" in any way).
In this case, observing an output of X gives the maximum information about it.

## Entropy

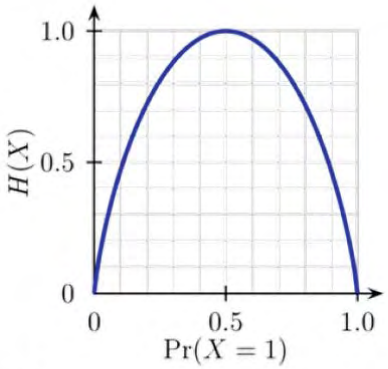
As demonstrated by Shannon: If X is a n-ary variable $\{x^{(1)},\ldots,x^{(n)}\}$

The **unique** formula for Entropy is $H(X) = -\sum_{i=1}^{n} P(x^{(i)}) \log_n P(x^{(i)})$

In case of binary variables (n=2) we have

$$H(X) = -[P(0)\log_2 P(0) + P(1)\log_2 P(1)]$$

$$P(0) = 1 - P(1)$$




[NOTE: We will omit the n index in the following]

After Shannon' definition and axioms, we get this function as the only possible entropy computation H(x) for a N-ary variable.
The binary (N=2) case is depicted above, where we can see the maximum value in the equi-probable case.

## Coding of random sequences

### Random Sequence

Suppose we have a long sequence generated L times the random variable X

$$S=(X_1,X_2,\ldots,X_L)$$

that we need to communicate to somebody else.

In the Information Theory language, these words are linked together

"Description ←→ Communication ←→ Transmission ←→ Coding ←→ Compression"

…of a data sequence S into a message M.

- Description = We use words from an alphabet, to translate S into a message
- Communication = The translated message has to be communicated to the other person
- Transmission = To communicate the message, we use a transmission channel (medium)
- Coding = To do the translation, we use a set of rules (code)
- Compression = We do a translation so that the resulting message will be shorter than S

→ To communicate S, we encode it into a shorter message, by using codewords from a code alphabet

The other person will need to decode the message, to get back the original sequence S.

We can use this definition of uncertainty, when coding a sequence originated from X, for communication over an ideal channel.

In fact, the most important thing here is to choose the right compression (=coding) scheme: that is, choose an alphabet, and the coding rules, in order to compress in the most efficient way the sequence S.

It is intuitively clear that, if we choose the codeword lengths in relationship of the behavior of S (i.e. the expected sequences have short codewords, and the unexpected ones have longer words), then we obtain the best compression efficiency, in a probabilistic sense.

## Coding

### An example of Coding

X has a probability distribution: P(*X=0*) = 0.7, P(*X=1*) = 0.3

Given a sequence generated from X:

S = (0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,1,0,0,0,1)

**Code Alphabet**: binary values {0,1} (the same of the sequence)
**Codebook**: we translate every two bits in a new word

(0,0) ←→ (0)
(0,1) ←→ (1,0)
(1,0) ←→ (1,1,0)
(1,1) ←→ (1,1,1,0)

NOTE: The code must be good also for decoding: no codeword is prefix of others
→ No ambiguity at destination

(0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,1,0,0,0,1) ←→ (0,0,1,1,0,0,0,1,1,0,0,1,0,0,1,0)

→ Encoding reduces the length (from 20 bits to 16) : Compression ratio r = (16/20)

This simple example shows the idea: the subsequence (0,0) is observed much more often (on the average) than others, since 0 has a higher probability than 1. Therefore, by assigning a short word to this subsequence, we get better compression.

In this case, we obtain a compression ratio of 16/20.

NOTE: The code should also be good for decoding: no codeword should be prefix of another one, in order to avoid ambiguity at destination! This restricts the set of possible codewords, and they will have different lengths.

## Entropy = code length

### Entropy : minimum code lenght

If we choose the codebook "wisely", we try to associate short codewords to more probable sub-sequences of X, and vice-versa.

→ The average compression ratio r will be smaller.

One of the main results from Information Theory is the

### Shannon's Source Coding Theorem

Even with the "best" coding for a given probability distribution P(X), the average compression ratio of the message will be no less than the Entropy of X:

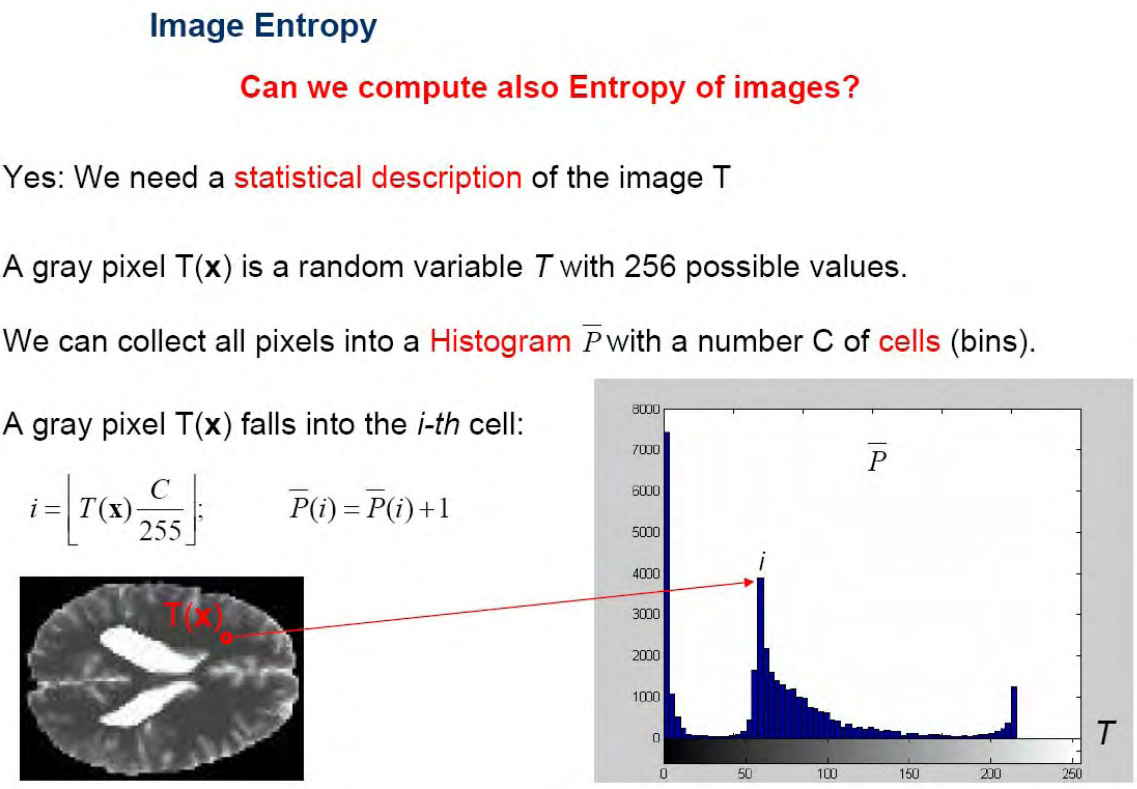
$$r_{min} = H(X).$$

→ Therefore, the Entropy of X is also the minimum encoding length (or the amount of compression) of a sequence generated from X.

The role of Entropy here is given by the Source Coding Theorem: even the "best" coding scheme cannot obtain an average compression ratio better than H(x).

That means, H(x) gives also the minimum possible encoding length of a (long) sequence generated from X.

***Image entropy computation with histograms***




Going back to images, we can compute the entropy measure of intensity (gray values) for a template T.

The procedure is the following: first, we need to estimate the distribution (statistics) of grey values P(i), where i are integer values between 0 and 255.

This statistics usually is computed by using histograms, with a given number of C bins.

Since the sample size is high (the number of pixels), this is a good statistical description of the image, and we can use it for computing H(T).

The Histogram $\overline{P}(i)$ is a discrete representation of P($T$):

$$P(T) = \overline{P}\left(\left\lfloor T\frac{C}{255}\right\rfloor\right)$$

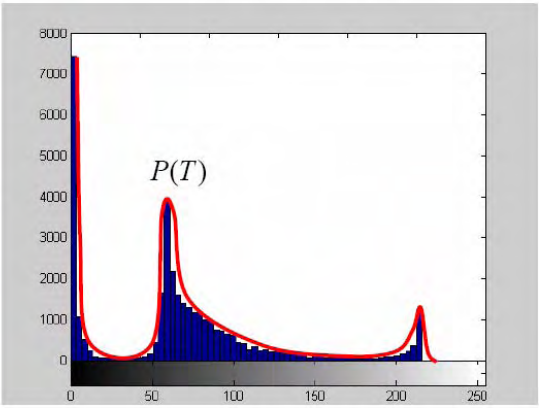
And it must be also normalized (sum = 1)

$$P(T) = \overline{P}(i) / L$$

L = total number of pixels

Therefore, we see the entire image (column-wise) as a long sequence **T** of gray values, generated from P($T$)

P($T$) → **T** = ($T_1$,$T_2$,…,$T_L$)




We can see the image as a random sequence generated from the statistics P(T). But this, of course, is a very abstract description, since in reality we have also spatial relationships between pixels! Nevertheless, this is enough for our purposes of template matching, where the joint pixel correspondences between two images are used.

## Image Entropy

### Image Entropy

Having P(T), we can compute the image entropy $H(T) = -\sum_{T=0}^{255} P(T) \log P(T)$

If we want to compress (=encode) the image, we need a codebook:
Example: using a binary alphabet, and coding one pixel at a time

(0) → (0)
(1) → (1,0)
(2) → (1,1,0)
…
(255) → (1,1,1,…,0)

→ The compressed image will be a binary message of length $L_M$ (in bits)

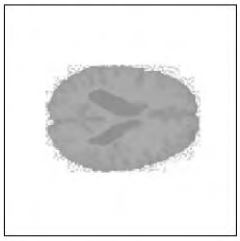
The best would be to assign short codewords to high-probability values P(T).
BUT: Even with the "best codebook", the average compression rate will be no less than the image entropy

$\min(L_M/(8*L)) = H(T)$

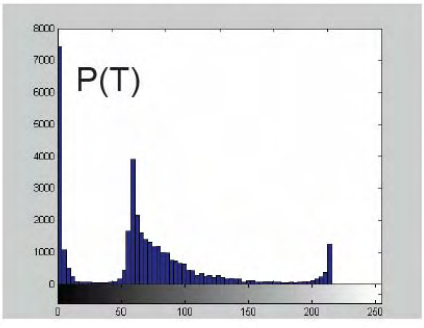
As we did before, we can obtain H(T), the image entropy, with the same formula. The meaning, in this case, is the maximal amount of compression that we can get by using the "best" compression algorithm (without loss).

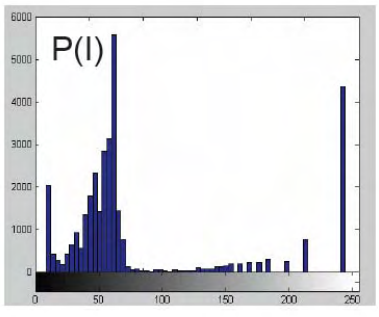
## Entropy of two images

Now consider two images, T and I



We can separately compute the Histograms, and obtain H(T) and H(I)

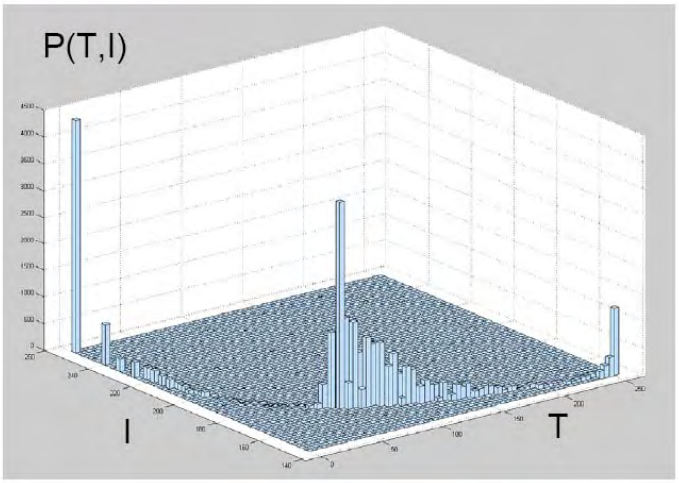


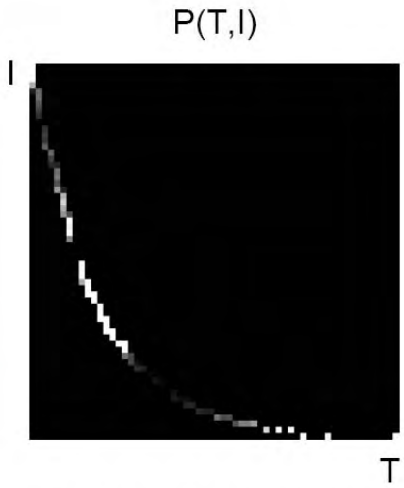



When we have two images, we can now consider their relationship.
First, we have two separate histograms, describing the two intensity statistics.

## Joint Histogram

But we can also put the corresponding pixel pairs together (T,I) and compute the Joint Histogram (2D)




P(T,I) = Probability of observing T and I at the same pixel coordinates ($\mathbf{x}_T$=$\mathbf{x}_I$).

Afterwards, we can consider the relationship between corresponding grey values (at the same pixel coordinates), and construct the so-called joint histogram.

This is a 2D histogram, which represents the joint probability of the event (T,I). Every cell gives the probability of observing a given pair of grey values (black-white, white-black, grey-white, etc.), for all (256*256) possibilities.
In fact, the number of cells is (C*C) which is less, and this is better for statistical reasons.

***Joint image entropy as similarity measure***

## Joint Entropy between two images

With the joint probability, we can compute the Joint Entropy

$$H(T,I) = -\sum_{T=0}^{255}\sum_{I=0}^{255} P(T,I)\log P(T,I)$$

Which is the uncertainty of the joint event: observed gray T and I, at the same pixel coordinates.

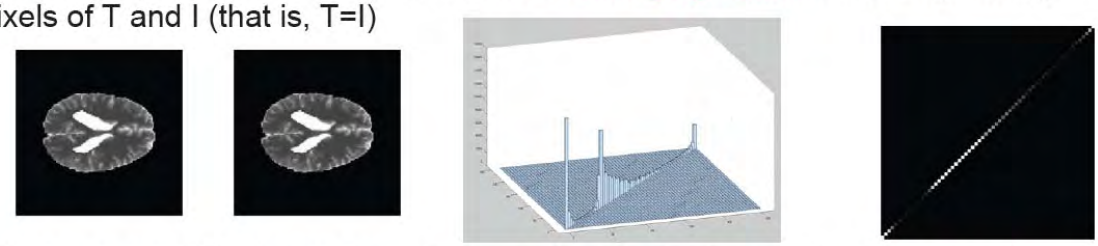
The meaning is: if H(T,I) is low, then the two images must be correlated, because only some "gray pairs" are probable, while other combinations are almost absent.

→ Whenever there is a relationship between T and I, Joint entropy has a minimum (also a non-linear relationship!)

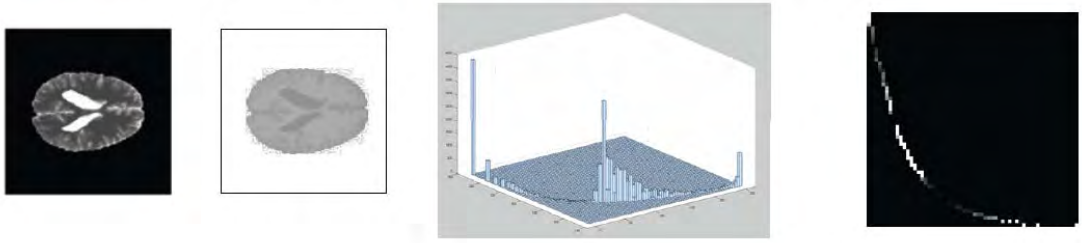
## Joint Entropy as similarity measure

This is a more general concept with respect to the SSD measure:

- Minimizing SSD aims to a pose with the same gray values in corresponding pixels of T and I (that is, T=I)



- Instead, Joint Entropy is low whenever the joint histogram lies in compact areas, identifying a more general relationship between T and I



Now, with the joint probability we have a joint Entropy, which expresses the uncertainty of the joint event (T,I).

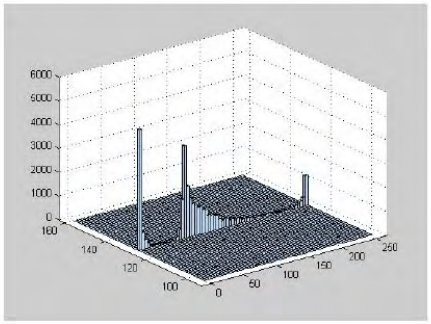
If there is a relationship, then only a few pairs will be observed (high P) and most other combinations will have null P. This gives a low joint uncertainty, that could be used as a correlation index.

## Problem of Joint Entropy: false positives

**But this is not yet the correct measure of matching**

In fact, it may introduce false positives:

If one of the two images has a constant value, Joint Entropy is also low.
But in this case, there is no visual correlation between T and I!




Still, we have the problem of false positives: here we see no relationship at all, but H(T,I) is low, as computed from the joint histogram!

***Mutual Information as similarity measure***

## Mutual Information

### A better measure: Mutual Information

Again from WikiPedia:

*In probability theory and information theory, the **mutual information**, or **transinformation**, of two random variables is a quantity that measures the mutual dependence of the two variables.*

$$MI(T,I) = H(I) - H(I \mid T) \qquad H(I \mid T) = H(I,T) - H(T)$$

H(I|T) = Conditional Entropy of I given T → Related to the conditional probability P(I|T).

Conditional entropy = the uncertainty that we have on I, if we know already the value T.

If we know the value of a variable T, on which I depends, the uncertainty about I must decrease: MI(T,I) ≥ 0

→ Therefore, MI is the information gain of I, by knowing T (and vice-versa)

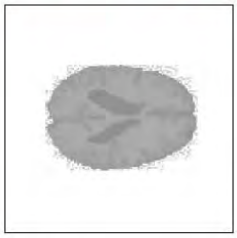
This is the reason for using MI: MI is the difference between the prior entropy of I, H(I), and the posterior (after observing T), or conditional entropy, H(I|T).

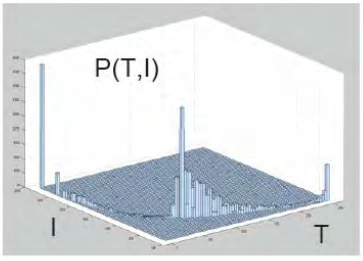
That is, MI measures the *information gain* on I, that we obtain after knowing T.

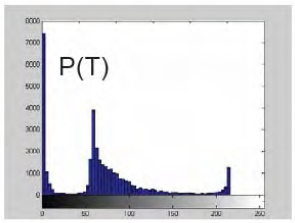
The idea is: by knowing the value of T, the uncertainty (entropy) about I must decrease. The amount of decrease in uncertainty is also the information that T gives about I, and vice-versa.

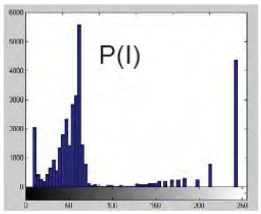
## Mutual Information between two images

How do we compute MI between two images?






We have $H(I \mid T) = H(I,T) - H(T)$

So, we can write $MI(T,I) = H(T) + H(I) - H(T,I) =$

$$= \sum_T \sum_I P(T,I) \log\left(\frac{P(T,I)}{P(T)P(I)}\right)$$

And for this computation, we need only P(T,I), because

$$P(T) = \sum_I P(T,I); \qquad P(I) = \sum_T P(T,I)$$

Concretely, MI is computed with the formula above, which requires both the joint histogram P(T,I) and the two marginal ones. But the marginal histograms can be in turn obtained from the joint, by summation over rows and columns, respectively (which is also called marginalization).

## Mutual Information as similarity function

### Why using Mutual Information as similarity measure?

Because Mutual information is an index of dependence between random variables

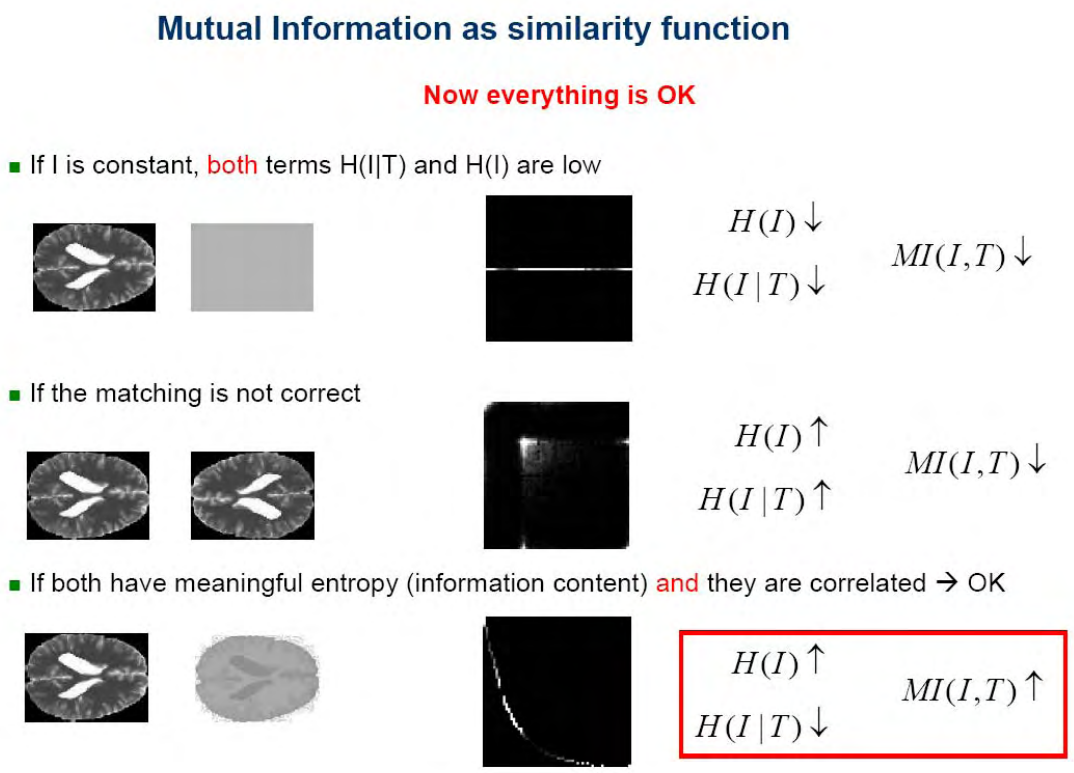
- If MI is high, the two images are very much dependent (statistically) one another, i.e. they are correlated.

→ Knowledge of T gives a lot of information about I , and they must represent the same scene or object

- If MI is low, then T and I are almost independent (pixel by pixel)

→ Knowledge of T says very little about I: they represent a different view of the same object, or a different scene.

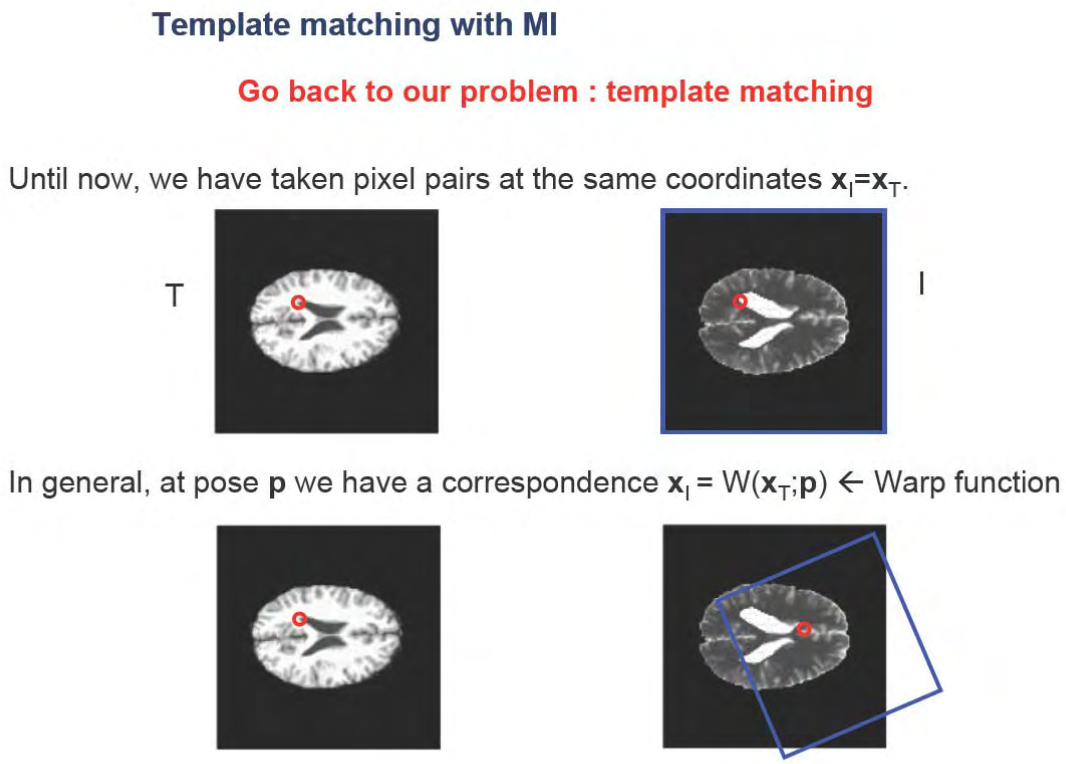
The result is the amount of dependence between T and I at the given pose: if they have a high MI, then they must depend on one another, otherwise they are independent variables.




Here we can see how the previous situation is handled by MI: in fact, MI is high only when both images have a high information content (high marginal entropies) and, at the same time, the joint entropy is low.

This is the correct way of maximizing the dependence, and gives a very general and robust matching index, that we can use for template matching, with a single appearance model, that can be very different from the current light/shading situation in the image.

***Matching templates with MI***




In template matching, we need to compute the similarity function for a given pose p.

That is, we need to take the corresponding points to the template in the image, by using the Warp W(x;p), in our case for computing Mutual Information.

## Computation of MI for a Warped template

### Step 1. Build the Joint Histogram → P(i,j)

Suppose to have a 2D histogram with (CxC) cells

At pose **p**, do

0. Reset the 2D Histogram to zero: P(i,j) = 0; (i,j) = 1,…,C

For every pixel **x** of the template T, do

1. Compute the Warped image pixel I(W(**x**;**p**))
2. Compute the Histogram cell to which the pair T(**x**),I(W(**x**;**p**)) belongs:

$$i = \left\lfloor T(\mathbf{x}) \frac{C}{255} \right\rfloor; \quad j = \left\lfloor I(W(\mathbf{x};\mathbf{p})) \frac{C}{255} \right\rfloor$$

3. Add its contribution to the cell: $P(i,j) = P(i,j) + 1$

4. At the end, normalize the probabilities: $P(i,j) = P(i,j) / \sum_{i,j} P(i,j)$

The procedure for computing MI between image and warped template at pose p is resumed above. This is the most expensive step, since we have to warp each pixel, and then sum the contribution of the observed grey value pair (template-image) to the respective histogram cell.

## Computation of MI for a Warped template

### Step 2. Compute Mutual Information

1. Compute the marginal probabilities of Template P(i) and Image P(j) from the Joint Probability P(i,j)

$$P(i) = \sum_{j=1}^{C} P(i,j); \quad P(j) = \sum_{i=1}^{C} P(i,j)$$

2. Compute Mutual Information $MI(\mathbf{p}) = \sum_{i=1}^{C} \sum_{j=1}^{C} P(i,j) \log\left( \frac{P(i,j)}{P(i)P(j)} \right)$

This is our new Cost function, to be optimized in **p**

$$\mathbf{p}^* = \arg\max_{\mathbf{p}} MI(T, I, \mathbf{p})$$

NOTE: This time the optimization is a maximization.
But it is the same: by switching sign, we have $\mathbf{p}^* = \arg\min_{\mathbf{p}}(-MI(T, I, \mathbf{p}))$

The second step of an MI evaluation is instead simpler, because the double sum is over the number of cells CxC, which are usually much less than the image (and template) size.

There is a statistical reason, in fact, for choosing a value of C which must be much lower than the sample size (number of pixel pairs). Usually, C ~ sqrt(N) is the rule of thumb for this choice.

### *Comparison with SSD*

## Mutual Information vs. SSD

**Compare MI and SSD computations**

SSD computation requires only one loop

0. Reset E=0.
For every pixel **x** of the template T, do

1. Compute the Warped image pixel I(W(**x**;**p**))

2. Add the SSD term to the sum: $E = E + \|T(\mathbf{x}) - I(W(\mathbf{x};\mathbf{p}))\|^2$

SSD looks much cheaper than MI to compute, but actually is not:

- In MI computation, for Joint Histogram computation we need to loop through all the points **x** of the template.
This has a cost which can be compared to SSD.

- The second step of MI involves nonlinear functions log( ), but the number of terms to sum is (CxC)~1000 cells, which is much less than the number of pixels **x**~100,000!

→ Overall, Step 1 determines the complexity of MI computation

The only drawback in MI is the memory usage (cache etc.) : a lot of random access to the Histogram cells in Step 1!

By comparing an MI evaluation with SSD, we can see how SSD looks at first simpler, since it requires only one pass, and only to accumulate a sum of squares.

But actually the difference is not significant, since the most expensive step in MI is only the first, due to the Warp of each template pixel onto the image. And this is almost the same cost, in terms of operations, of SSD.

The other drawback of MI is actually the use of the joint histogram, which needs a lot of random accesses (one for each pixels) on the square (CxC) matrix, and degrades system performance, while SSD of course does not have this problem.

***Optimizing Mutual Information with a Levenberg-Marquardt approach***

**Mutual Information Optimization for Template Matching**

The optimal pose is $\mathbf{p}^* = \arg\max_{\mathbf{p}} MI(T, I, \mathbf{p})$

With Gauss-Newton (SSD) optimization we have $\Delta\mathbf{p} = \mathbf{G}^{-1} \frac{\partial}{\partial \mathbf{p}} E(\mathbf{p})$

But this is not a **sum of squares**, so the G matrix does not exist anymore!

→ In this case, we have to use the Newton step, or another approximation of it.

$$\Delta\mathbf{p} = \mathbf{H}^{-1} \frac{\partial}{\partial \mathbf{p}} MI(\mathbf{p})$$

Where H is the Hessian matrix of MI : $\mathbf{H}(h,k) = \frac{\partial^2}{\partial p_h \partial p_k} MI(\mathbf{p})$

How can we compute these derivatives, to do the optimization?

For the template pose estimation, we also used the derivative of the cost function (Gauss-Newton), where the GN matrix was an approximation of the second-order Hessian matrix H, using only first derivatives (Jacobian matrix).

Here we do not have the standard Gauss-Newton algorithm anymore, since the cost function is not in a LSE form.
But still, we can approximate the Hessian matrix of Mutual Information with a first-order matrix, using only the Jacobian of the Warp function.

## Mutual Information Optimization for Template Matching

We provide here (without demonstration) the solution to this problem

- The first derivative of MI is $\frac{\partial}{\partial \mathbf{p}} MI(\mathbf{T}, \mathbf{I}(\mathbf{p})) = \sum_{i,j} \frac{\partial P(i,j)}{\partial \mathbf{p}} \log\left(\frac{P(i,j)}{P(j)}\right)$

- To compute $P(i,j)$ and $\frac{\partial P(i,j)}{\partial \mathbf{p}}$ we need to accumulate all the contributions T(**x**), I(W(**x**;**p**))

→ (This is another issue: for details, see the paper [Thevenaz et al. 2000])

- In Gauss-Newton (SSD), we used the G matrix to approximate H : G~H

Here, the correct approximation of **H** is:

$$\mathbf{H}_{MI}(h,k) \approx \sum_{i} \frac{\partial P(j)}{\partial p_h} \frac{\partial P(j)}{\partial p_k} \frac{1}{P(j)} - \sum_{i,j} \frac{\partial P(i,j)}{\partial p_h} \frac{\partial P(i,j)}{\partial p_k} \frac{1}{P(i,j)}$$

This is a good approximation, replacing Gauss-Newton and providing a fast and robust optimization method for Mutual Information.

The computation of both gradient and Hessian matrix requires the derivatives of each joint histogram cell, with respect to the pose parameters.

This computation can be done while accumulating the histrogram itself, by accumulating for each sample pair also the derivative of P(i,j) as well.

***The full MI optimization algorithm***

## Mutual Information Optimization: the Algorithm

Iterate: At pose **p**

**Step 1: Joint histogram and derivatives**

0. Reset the (CxC) histogram P(i,j)=0 and the cell gradients $\frac{\partial P(i,j)}{\partial \mathbf{p}} = \mathbf{0}$

For every pixel **x** of the template T, do

1. Get the gray-value pair [T(**x**), I(W(**x**;**p**))] $i = \left\lfloor T(\mathbf{x}) \frac{C}{255} \right\rfloor; \quad j = \left\lfloor I(W(\mathbf{x};\mathbf{p})) \frac{C}{255} \right\rfloor$
2. Find the cell (i,j) to which the pair belongs
3. Update the Joint Histogram $P(i,j)$
4. Update the Histogram derivatives $\frac{\partial P(i,j)}{\partial \mathbf{p}}$

**Step 2: MI gradient and Hessian Matrix**

5. Compute the gradient $\mathbf{g} = \frac{\partial}{\partial \mathbf{p}} MI(\mathbf{p})$
6. Compute the Hessian matrix $\mathbf{H}(h,k) = \frac{\partial^2}{\partial p_h \partial p_k} MI(\mathbf{p})$
7. Compute $\Delta\mathbf{p} = \mathbf{H}^{-1}\mathbf{g}$ and update **p** → **p**+Δ**p**

...until ||Δ**p**||<ε

This algorithm for MI optimization resembles to the Lucas-Kanade approach from many aspects, and the same improvements can be applied (forwards- and inverse-compositional, multi-resolution) for speed and robustness.

## Selected bibliographic references

**Survey papers**

- [1]
- [26]

**Lecture 2. Camera-world geometry**

Representation of rigid-body rotations

- [1] (§ 2.2) with references

Intrinsic camera parameters

- [1] (§ 2.1) with references
- The MATLAB Calibration Toolbox: http://www.vision.caltech.edu/bouguetj/calib_doc/

**Lecture 3. 3D pose estimation from point correspondences**

Linear and Nonlinear LSE (Gauss-Newton and Levenberg-Marquardt)

- [1] (§ 2.4) with references

Robust LSE (RANSAC and M-Estimators)

- [1] (§ 2.5) with references
- [2] (original paper on RANSAC)

P3P pose estimation problem:

- [1] (§ 2.3.3) with references

**Lectures 4-5. Bayesian Tracking**

Motion models (Brownian Motion, WNA)

- [3]

Bayesian tracking scheme

- [1] (§ 2.6) with references

Kalman Filter

- [1] (§ 2.6.1) with references
- http://www.cs.unc.edu/~welch/kalman/
- [3]

Extended Kalman Filter

- [1] (§ 2.6.1) with references
- [3]

Particle Filters

- [1] (§ 2.6.2.) with references
- [4]
- The Condensation web page: http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/ISARD1/condensation.html

**Lecture 6. Kanade-Lucas-Tomasi features tracker**

KLT algorithm

- http://www.ces.clemson.edu/~stb/klt/
- [5]

Optical Flow

- http://en.wikipedia.org/wiki/Optical_flow

Harris Corner detector:

- http://en.wikipedia.org/wiki/Corner_detection
- [6]

**Lecture 7. SIFT**

- [7]
- [8]
- [9]

**Lecture 8. Edge-based contour tracking**

- [1] (§ 4.1) with references
- [10]
- [11]
- http://en.wikipedia.org/wiki/Canny
- [12]
- [13]

**Lecture 9. Contour tracking using Likelihood functions**

B-Splines

- http://de.wikipedia.org/wiki/Spline, and references
- http://userpage.fu-berlin.de/~vratisla/Bildverarbeitung/Bspline/Bspline.html

CONDENSATION for contour tracking

- Official Page: http://www.robots.ox.ac.uk/~misard/condensation.html
- [4]

CCD Algorithm

- [14]

- [15]

**Lecture 10. Active Appearance Models**

AAM Webpages:

- http://www2.imm.dtu.dk/~aam/tracking
- http://www2.imm.dtu.dk/~aam/faces
- Tim Cootes' page: http://www.isbe.man.ac.uk/~bim/
- CMU Webpage: http://www.ri.cmu.edu/projects/project_448.html

AAM Papers

- [16]
- [17]

PCA

- http://de.wikipedia.org/wiki/Principal_Component_Analysis

**Lecture 11. Lucas-Kanade Algorithm for template matching**

- http://www.ri.cmu.edu/projects/project_515.html (with Matlab code)
- [20]
- [16] (Piece-wise affine Warp)
- [16] (forwards- and inverse-compositional methods)
- [16] (combined pose+appearance optimization)
- [19]

**Lecture 12. Robust Template Similarity Functions**

- [18] (M-Estimators)
- [21] (NCC)
- http://en.wikipedia.org/wiki/Information_theory
- http://en.wikipedia.org/wiki/Mutual_information
- [22] (Shannon's original paper)
- [23]
- [24]
- [25] (Optimization of MI)

**References**

**NOTE: Most of the publications below are also in a large (~40 Mb) zip file, that can be downloaded from**

**http://www6.in.tum.de/~panin/Bibliography.zip**

[1] V. Lepetit and P. Fua, Monocular Model-Based 3D Tracking of Rigid Objects: A Survey, Foundations and Trends in Computer Graphics and Vision, Vol. 1, Nr. 1, pp. 1-89, October 2005

Online : http://cvlab.epfl.ch/publications/publications/2005/LepetitF05.pdf

*This text has also references inside (as indicated in the list).*

[2] M. A. Fischler, R. C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. Comm. of the ACM, Vol 24, pp 381-395, 1981

[3] Yaakov Bar-Shalom, X.-Rong Li, Thiagalingam Kirubarajan Estimation with Applications to Tracking and Navigation, 2002

[4] Michael Isard and Andrew Blake CONDENSATION -- conditional density propagation for visual tracking Int. J. Computer Vision, 29, 1, 5--28, (1998)

[5] Jianbo Shi and Carlo Tomasi. Good Features to Track. IEEE Conference on Computer Vision and Pattern Recognition, pages 593-600, 1994.

[6] C. Harris and M. Stephens (1988). "A combined corner and edge detector". *Proceedings of the 4th Alvey Vision Conference*, pages 147--151.

[7] Lowe, D. G., “Distinctive Image Features from Scale-Invariant Keypoints”, International Journal of Computer Vision, 60, 2, pp. 91-110, 2004.

[8] Lindeberg, Tony "Feature detection with automatic scale selection", International Journal of Computer Vision, 30, 2, pp 77--116, 1998.

[9] Iryna Skrypnyk, David G. Lowe: Scene Modelling, Recognition and Tracking with Invariant Image Features. ISMAR 2004: 110-119

[10] C. J. Harris. Tracking with rigid models. In A. Blake and A. Yuille, editors, *Active Vision*. MIT Press, Cambridge, MA, 1992.

[11] J. Canny *A Computational Approach to Edge Detection*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 8, No. 6, Nov 1986.

[12] David G. Lowe: Three-Dimensional Object Recognition from Single Two-Dimensional Images. Artif. Intell. 31(3): 355-395 (1987)

[13] Model-Based Object Tracking in Monocular Image Sequences of Road Traffic Scenes. D. Koller, K. Daniilidis, H.-H. Nagel. International Journal of Computer Vision 10:3 (1993) 257--281.

[14] Robert Hanek and Michael Beetz. The Contracting Curve Density Algorithm: Fitting Parametric Curve Models to Images Using Local Self-adapting Separation Criteria. *International Journal of Computer Vision (IJCV)*, 59(3):233–258, 2004.

[15] Robert Hanek, Thorsten Schmitt, Sebastian Buck, Michael Beetz: Towards RoboCup without Color Labeling. RoboCup 2002: 179-194

[16] T.F.Cootes, G.J. Edwards and C.J.Taylor. "Active Appearance Models", in Proc. European Conference on Computer Vision 1998 (H.Burkhardt & B. Neumann Ed.s). Vol. 2, pp. 484-498, Springer, 1998.

[17] I. Matthews and S. Baker “Active Appearance Models Revisited”, International Journal of Computer Vision, Vol. 60, No. 2, November, 2004, pp. 135 - 164.

[18] B. Theobald, I. Matthews, and S. Baker, “Evaluating Error Functions for Robust Active Appearance Models”, Proceedings of the International Conference on Automatic Face and Gesture Recognition, April, 2006, pp. 149 - 154.

[19] J. Xiao, S. Baker, I. Matthews, and T. Kanade, “Real-Time Combined 2D+3D Active Appearance Models”, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, June, 2004.

[20] S. Baker and I. Matthews, “Lucas-Kanade 20 Years On: A Unifying Framework”, International Journal of Computer Vision, Vol. 56, No. 3, March, 2004, pp. 221 - 255.

[21] J. P. Lewis, “Fast Template Matching”, Vision Interface, p. 120-123, 1995.

[22] Claude E. Shannon “A Mathematical Theory of Communication”, Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656, 1948.

[23] P Viola, WM Wells III “Alignment by Maximization of Mutual Information”, International Journal of Computer Vision, 1997 – Springer

[24] Frederik Maes, André Collignon, Dirk Vandermeulen, Guy Marchal, Paul Suetens: “Multimodality Image Registration by Maximization of Mutual Information” IEEE Trans. Med. Imaging 16(2): 187-198 (1997)

[25] Thevenaz, P. Unser, M. “Optimization of mutual information for multiresolution image registration” IEEE Transactions on Image Processing, Dec 2000

[26] Yilmaz, A., Javed, O., and Shah, M. 2006. Object tracking: A survey. *ACM Comput. Surv.* 38, 4 (Dec. 2006), 13