Tracking

Federico Magnolfi

27 May 2019

Outline

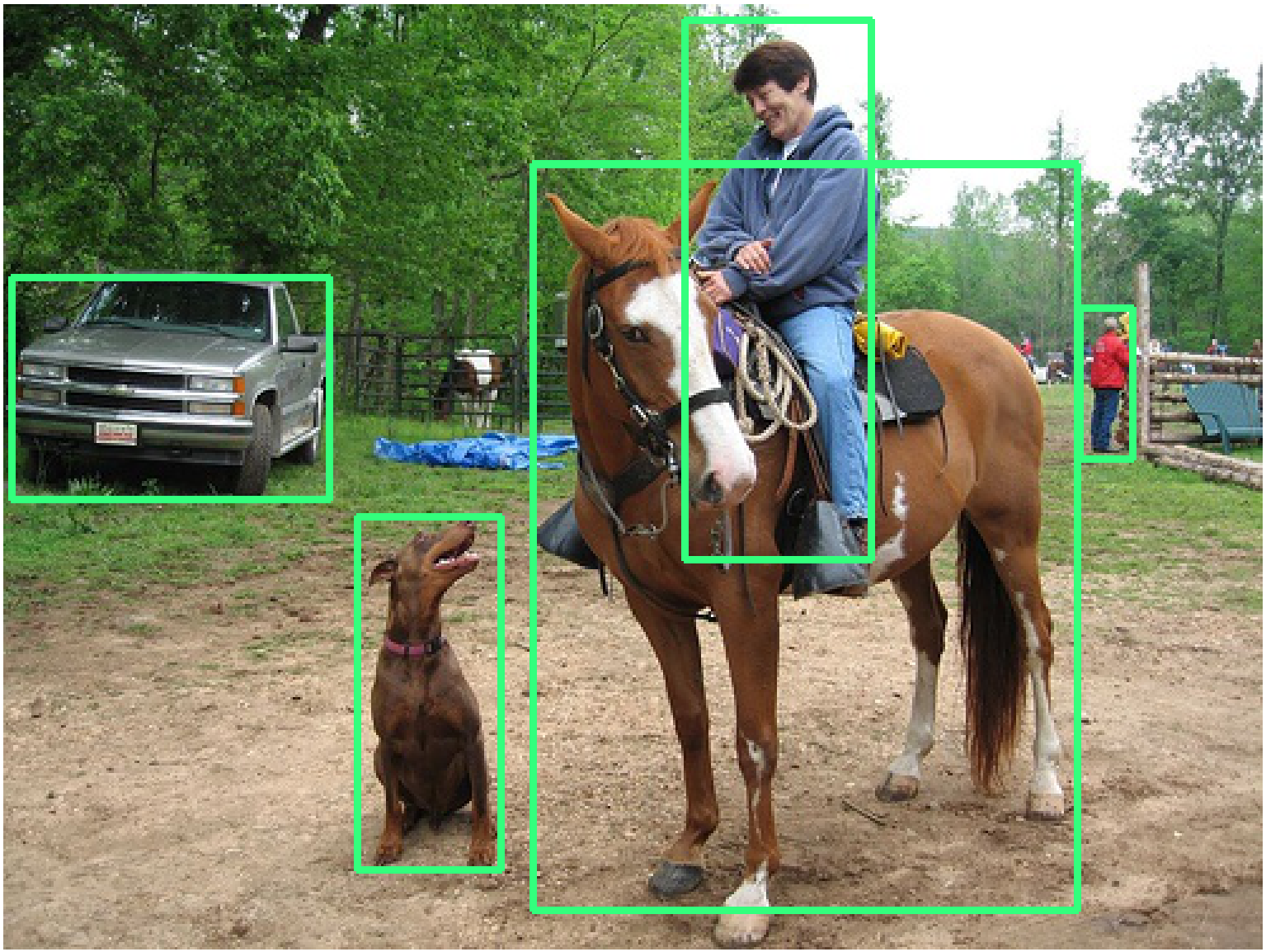
Introduction to tracking

Before tracking: detection

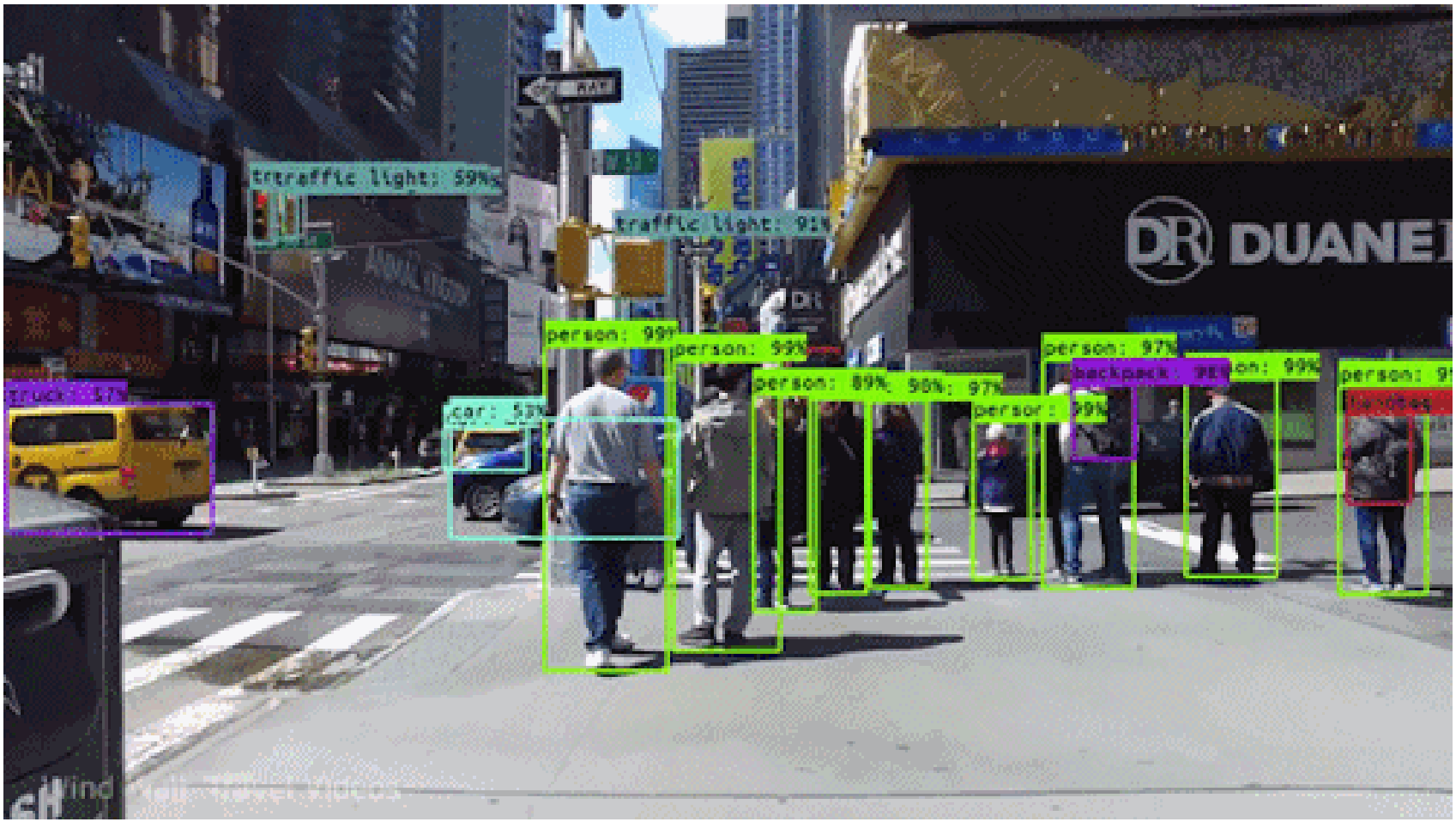
Given an image, the **detection** task consists in:

- Found objects inside the image
- Put a bounding box around each object
- Eventually classify each object (recognition task)

Detection example 1



Detection example 2



Object detection

There are many ways to do object detection:

- Template matching
- Background subtraction and contours analysis
- Neural Networks

Each of them has its pros and cons.

After detection: tracking

Questions about **tracking**:

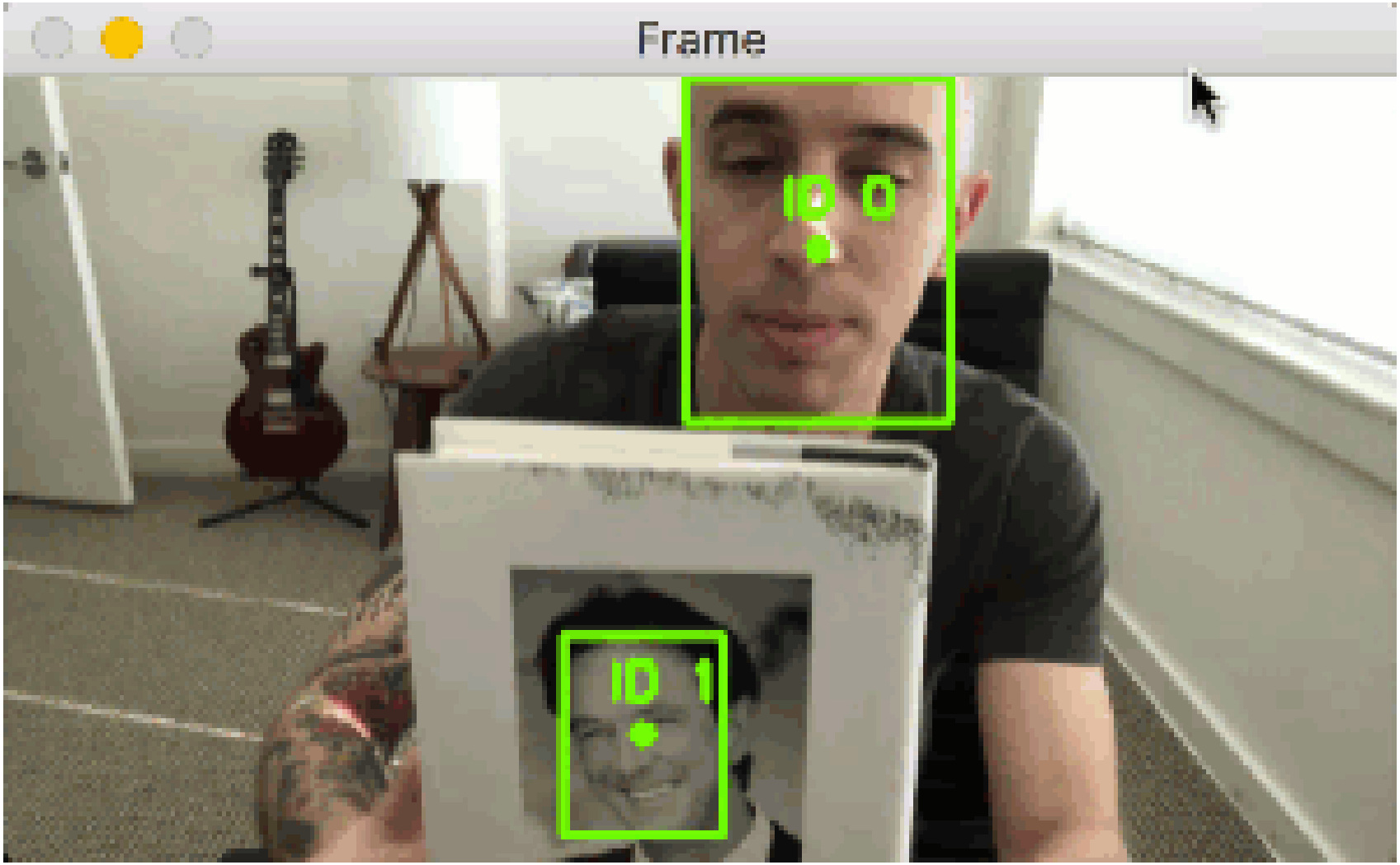
- What is it?
- Why use it?
- When use it?
- Where use it?
- How to use it?

What is tracking

Object tracking is the process of:

- Taking an initial set of object detections (a set of bounding boxes)
- Creating a unique ID for each of the initial detections
- Tracking each of the objects as they move across frames in a video, maintaining the assignment of unique IDs

What is tracking



Why use tracking

- Easy to assign IDs to objects
- Can be much faster than object detection, this is ideal for real-time contexts and when there are many objects to track
- Can help when detection fails

When use tracking

- When we want to count the objects that pass in front of the camera for surveillance purposes (e.g.: people, vehicles...)
- When detection only is not enough: we want to follow an object (e.g.: to subsequently classify an action made by a person)
- When we can't allow heavy detection algorithms on each frame, due to time and/or hardware constraints
- When we want to help detection task with more informations

Where use tracking

Many applications:

- Surveillance cameras (both private and public: home, airports...)
- Traffic cameras
- Cameras that count people getting on public transport (e.g.: bus, tram, ship...), both for statistics gathering and additional checks
- Help detection task when failure is not allowed (e.g.: autonomous driving)

How to use tracking

- Determine the role of the tracker in your application
- Choose desired tracking algorithm (depending by the context)
- Combine it with object detection and eventually other tasks

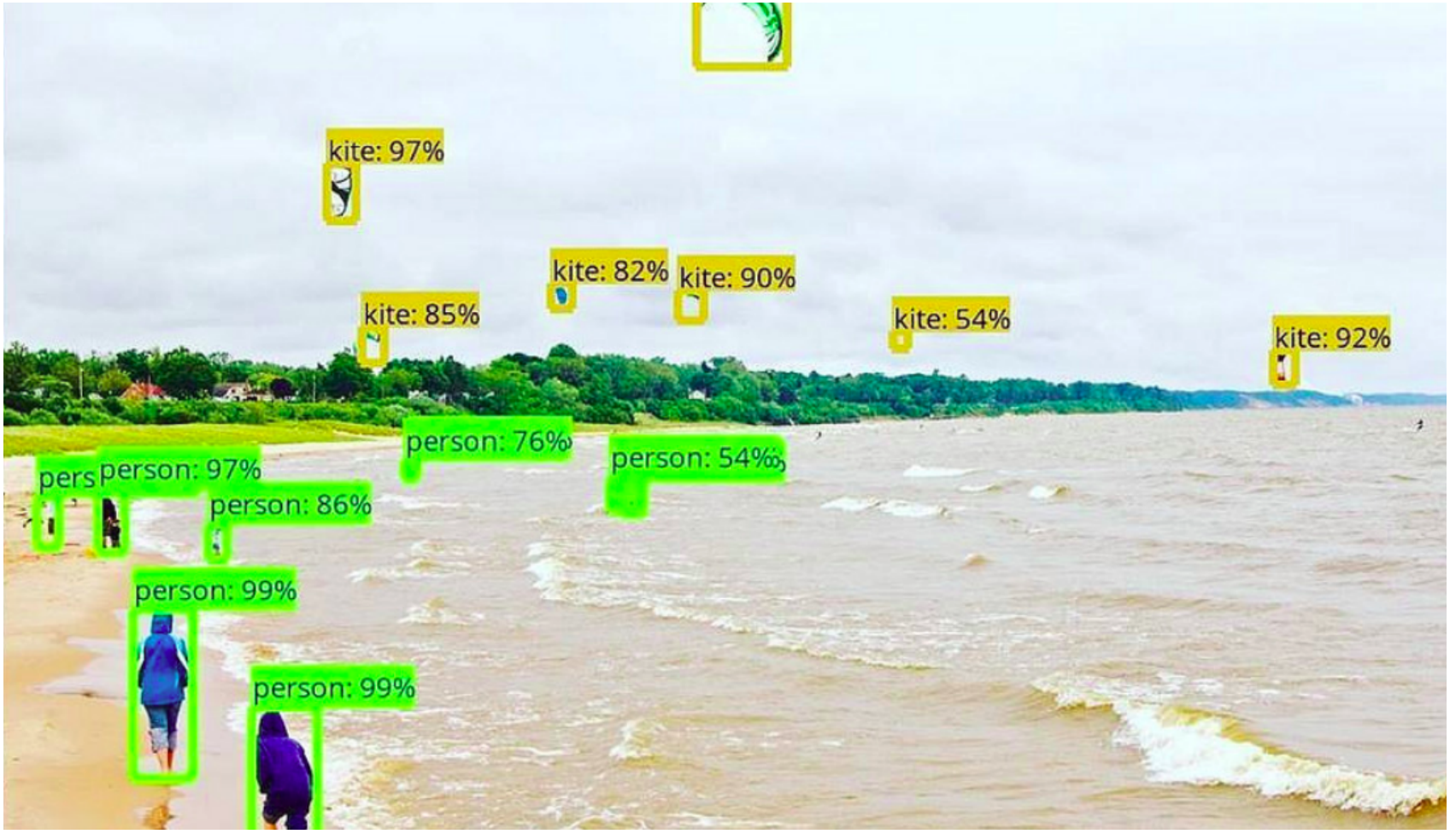
Difficulties

There are many difficulties:

- Many objects: tracking becomes computationally expensive
- Similar objects can be confused with each other, and so their IDs exchanged
- Changes in lighting and shape of the object: the object can be not very similar w.r.t. previous images of itself
- Drift phenomenon: errors in tracking can propagate in time until the tracker loses the subject, or follows different objects
- Occlusion: other objects can transit between tracked object and the camera
- Failure: the tracker loses the object

Different tracking algorithms behave differently in the face of these difficulties

Difficulties example 1: many objects

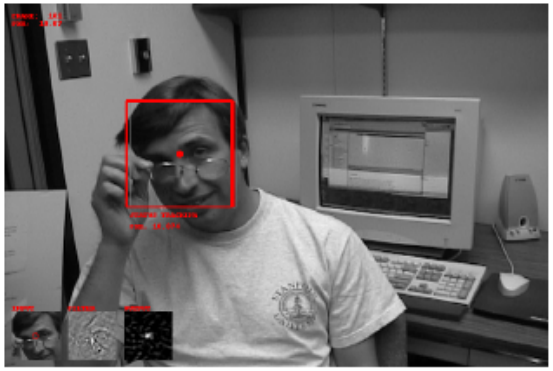


Difficulties example 2: many similar objects

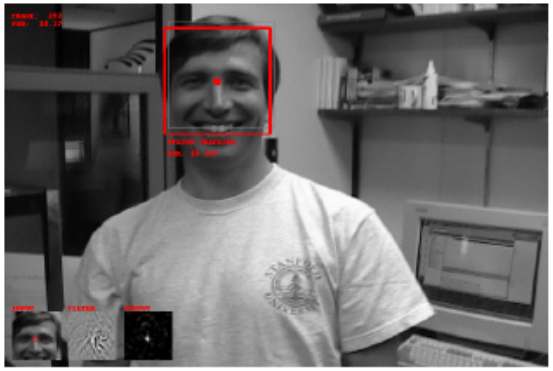
Difficulties example 3: drifting

Difficulties example 4

A - Occlusion



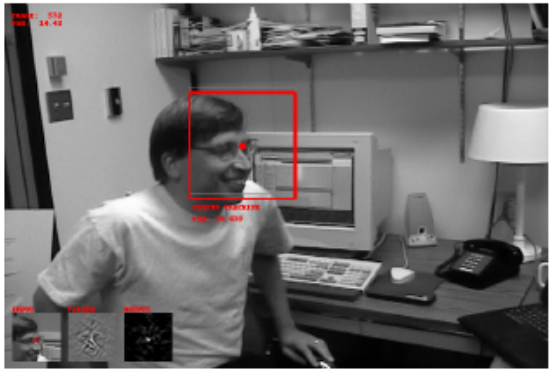
B - Remove Glasses



C - Fast Motion

D - Rotation

E - Off Frame



F - Rotation

Other concerns

There are various criteria for which the various tracking algorithms differ:

- Overall tracking quality: we would like all objects to be followed precisely and without false positives
- Robustness to overcome difficulties
- Performance: each tracker algorithm is capable of process only a certain amount of frames per second

For some applications real time tracking is required. Sometimes, tracking can be performed offline.

Introduction recap

- Tracking is useful to assign IDs to objects in video
- Tracking can be faster than detection
- Many aspects to evaluate of a tracking algorithm

Object tracking

Tracking theory overview

We'll talk about:

- Tracking general concepts
- Different approaches to tracking
- A quick look over tracking algorithms implemented in OpenCV
- An in-depth look at some of these algorithms

Tracking

Tracking objective

Associate target objects in consecutive video frames

Tracking requirement

A bounding box for the object to be tracked

Tracking pseudo-API

```
"""
TRACKER CREATION
@img_frame: a complete frame of the video
@obj_bbox: 4-tuple representing the object bounding
    box
"""
tracker = Tracker(img_frame, obj_bbox)

"""
OBJECT TRACKING
@img_new_frame: frame where to look for the object
returns: bounding box of the object in the new frame
"""
new_obj_bbox = tracker.update(img_new_frame)

```

In the case of multiple objects: create a tracker for each object.

Tradeoffs

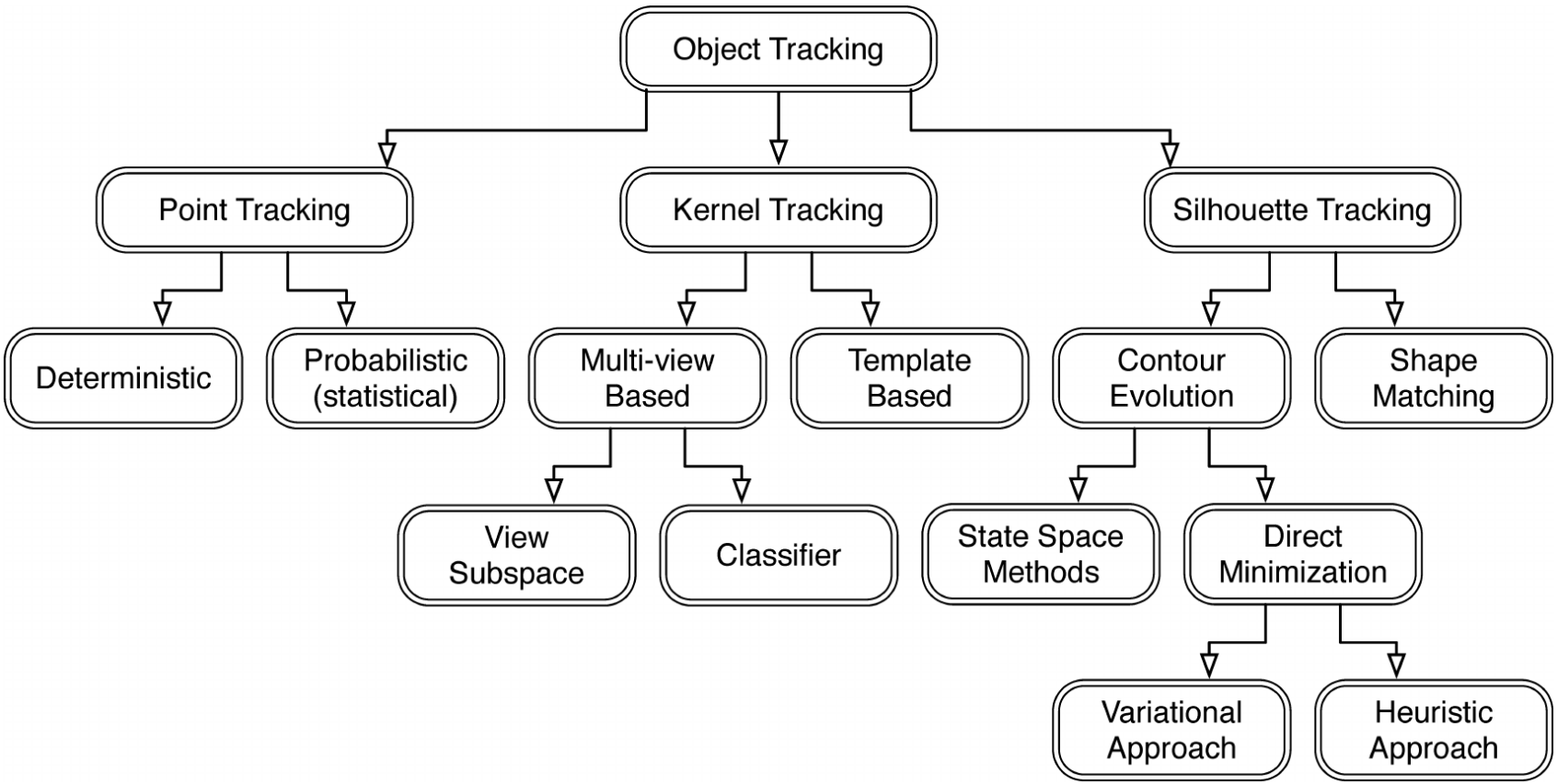
Quality vs speed

In order to find best (more probable) bounding box, it is necessary to evaluate for many candidate bounding boxes how well they fit: the more candidates there are, the slower the algorithm is.

Adaptability vs robustness

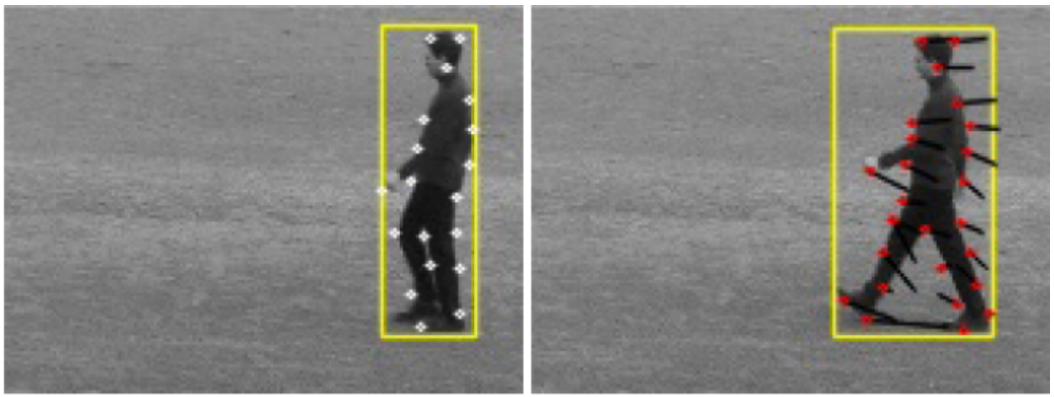
An object can change its lighting and shape: a good tracker should adapt to the new appearance of the object. This is dangerous because too high adaptability can reduce robustness, leading to the drifting problem and/or misdetections in presence of occlusions.

Taxonomy of tracking methods

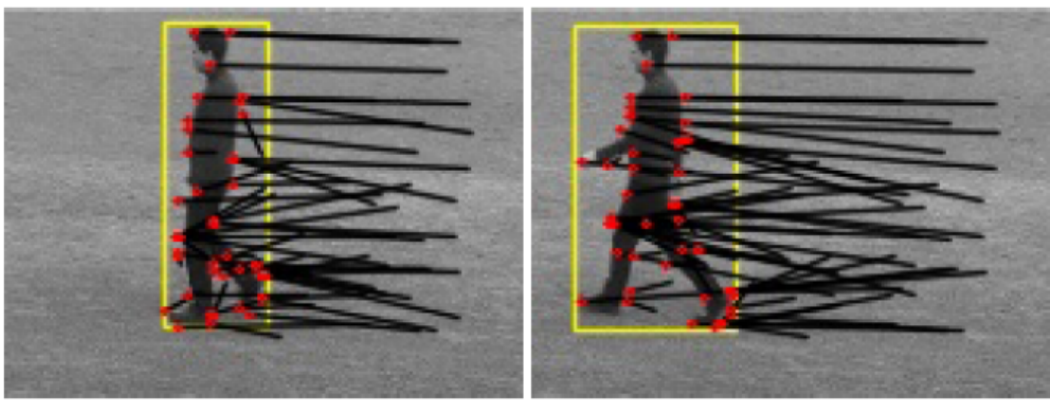


In addition, machine learning approaches have been proposed in recent years, mainly based on convolutional neural networks.

Point tracking



(a) Frame: 1. (b) Frame: 5.



(c) Frame: 27. (d) Frame: 37.

The detected objects are represented by points, and the tracking of these points is based on the previous object states which can include positions of the objects and their motion.

Point tracking: pros and cons

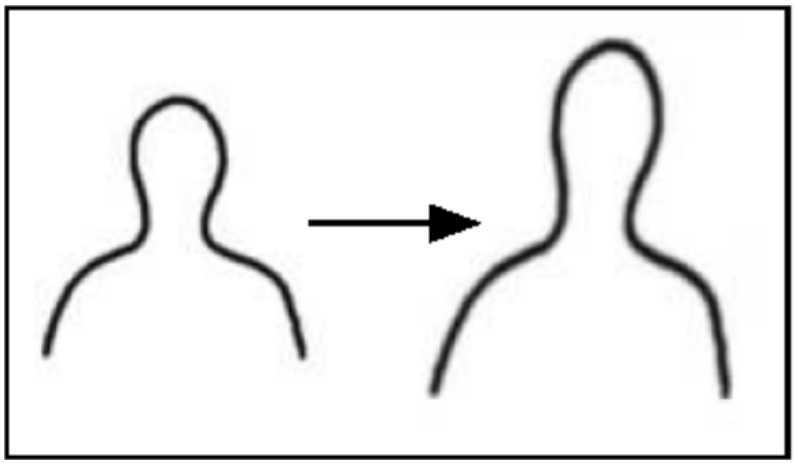
Pros:

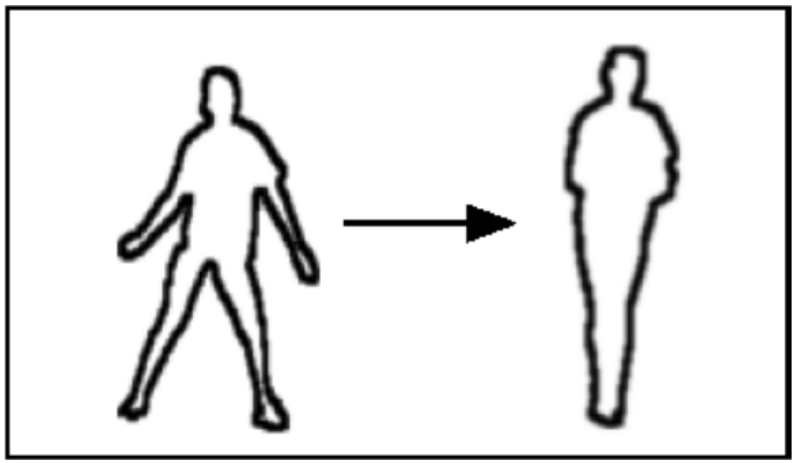
- Suitable in case of small objects, representable with few points (better one point only)
- robust to illumination changes

Cons:

- Not very robust to other challenges, especially in complex scenarios such as occlusion, object rotation...

Silhouette tracking





Tracking is performed by estimating the object region in each frame. Silhouette tracking methods use the information encoded inside the object region. Given the object models, silhouettes are tracked by either shape matching or contour evolution.

Silhouette tracking: pros and cons

Pros:

- Flexible in object representation, can handle a large variety of object shapes, including complex non rigid shapes
- Object detection is only needed when the object first appears in the scene

Cons:

- Occlusions can be a problem

OpenCV tracking algorithms

- Boosting (2006): low quality, based on an online version of AdaBoost (+ are only the chosen bboxes, - randomly from bg)
- MIL (2010): good quality, improves Boosting using more +
- TLD (2010): robust over scale changes and occlusions, but many false positives
- MedianFlow (2010): low quality, estimate trajectory checking previous and next frames
- MOSSE (2010): good quality, robust, very fast
- KCF (2014): improves MIL in quality and speed
- GOTURN (2016): high quality, based on CNN
- CSRT (2017): high quality, not the fastest

+/−: positive/negative examples

OpenCV tracking algorithms

- Boosting (2006): low quality, based on an online version of AdaBoost (+ are only the chosen bboxes, - randomly from bg)

+/−: positive/negative examples

OpenCV tracking algorithms

- Boosting (2006): low quality, based on an online version of AdaBoost (+ are only the chosen bboxes, - randomly from bg)
- MIL (2010): good quality, improves Boosting using more +

+/−: positive/negative examples

OpenCV tracking algorithms

- MIL (2010): good quality, improves Boosting using more +
- TLD (2010): robust over scale changes and occlusions, but many false positives

+/−: positive/negative examples

OpenCV tracking algorithms

- MIL (2010): good quality, improves Boosting using more +
- MedianFlow (2010): low quality, estimate trajectory checking previous and next frames

+/−: positive/negative examples

OpenCV tracking algorithms

- MIL (2010): good quality, improves Boosting using more +

- MOSSE (2010): good quality, robust, very fast

+/−: positive/negative examples

OpenCV tracking algorithms

- MIL (2010): good quality, improves Boosting using more +

- MOSSE (2010): good quality, robust, very fast
- KCF (2014): improves MIL in quality and speed

+/−: positive/negative examples

OpenCV tracking algorithms

- MOSSE (2010): good quality, robust, very fast
- KCF (2014): improves MIL in quality and speed
- GOTURN (2016): high quality, based on CNN

OpenCV tracking algorithms

- MOSSE (2010): good quality, robust, very fast
- KCF (2014): improves MIL in quality and speed

- CSRT (2017): high quality, not the fastest

OpenCV tracking algorithms

Comparison

We will compare these three algorithms first in theory and then in laboratory practice:

- MOSSE (2010): good quality, robust, very fast
- KCF (2014): improves MIL in quality and speed
- CSRT (2017): high quality, not the fastest

MOSSE, KCF & CSRT overview

Similarities

- Based on **correlation filters** (so they are *Kernel Trackers*)
- Recognition of **tracking failure**

Differences

- Tracking quality (in ascending order: MOSSE, KCF, CSRT)
- Speed (in descending order: MOSSE, KCF, CSRT)

Correlation filters

How correlation filters work

- A filter (or a group of filters) mantains the information about the object to be tracked
- To find the new object (*location step*), is made a convolution between the filter and the search window in the new frame
- The peak of the convolution indicate the center of the new location
- Filter is updated with new information (*update step*)
- Actually, convolution can be calculated in Fourier domain

MOSSE, KCF & CSRT algorithms differ in how they mantain informations in the filter, and in the choice of the search window

Actors and notation in future slides

Operators

$\circledast$: convolution operator (correlation actually)
$\odot$: element-wise product (also divisions will be element-wise)
$\mathcal{F}$: Fourier transform
*: complex conjugate

Variables

$i \triangleq$ *image*	$I \triangleq \mathcal{F}(i)$
$g \triangleq$ *groundtruth* (desired output)	$G \triangleq \mathcal{F}(g)$
$f \triangleq$ *filter*	$F \triangleq \mathcal{F}(f)$
$o \triangleq$ *output* (after applying filter to image)	$O \triangleq \mathcal{F}(o)$

All of these variables are matrices; subscript k will be used to indicate k-th element of a dataset (e.g.: i_k is the k-th image).

Update step in correlation filters

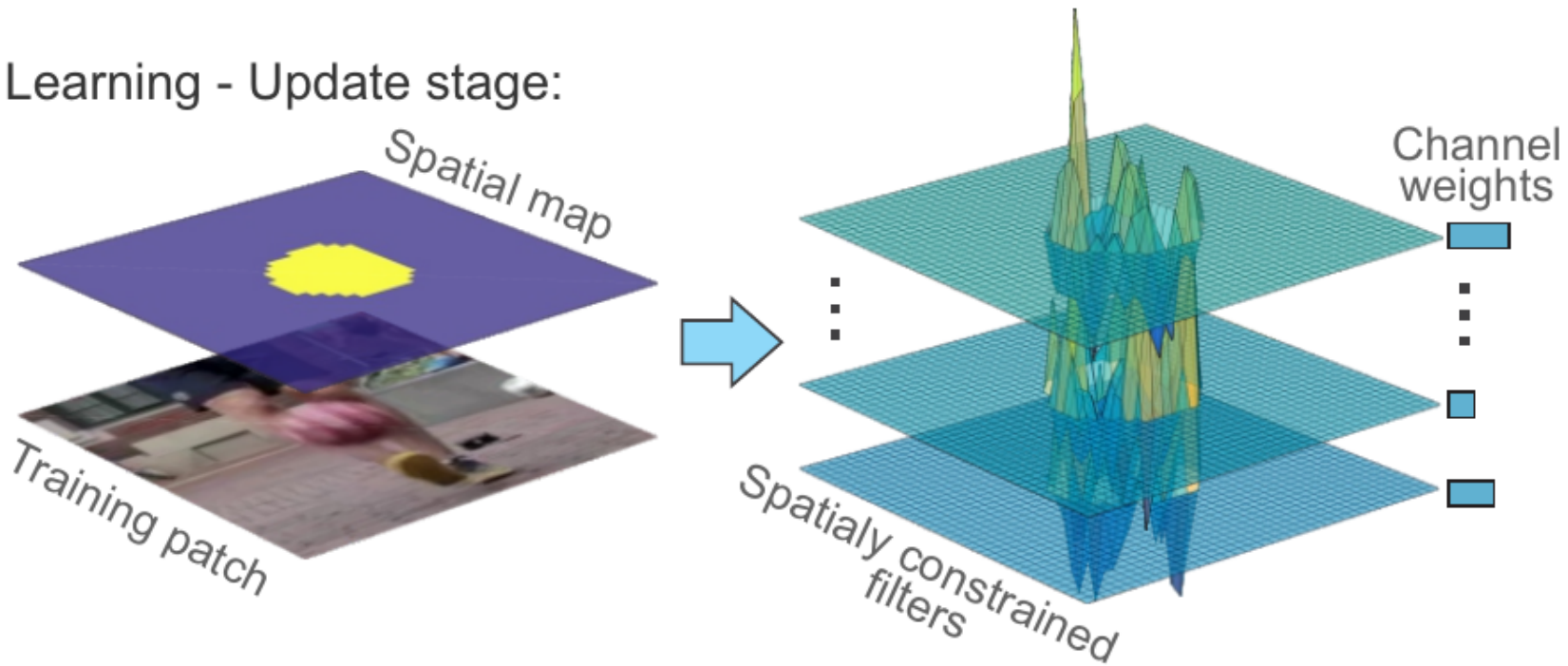
Filter based trackers model the appearance of objects using filters trained on example images. Remark: no offline training is done, the **training is completely online**, i.e. the filter is learned in real-time analyzing the frames of the video.

How the training/update of the filter works?

Depends on the tracker, but usually:

1. We create a set of training images, by perturbing the first crop (i.e. slightly modifying the content of the initial bounding box)
2. To each training image, we associate a ground truth (desired output): the ground truth is almost always a 2D gaussian centered in the middle of the bounding box

Update step in correlation filters: graphical intuition



Terminology

Some remarks on terminology:

- Training/Update are synonyms in the literature
- Even if the training is completely online (i.e. the filter is learned in real-time analyzing the frames of the video) we differentiate between initial updates (made on the first frame) and online updates (made on the following frames)
- For efficiency reasons, the filter is not correlated with the entire image but with only a portion of it, which depends on the tracker: when we correlate the image to the filter, we will refer implicitly to this subregion
- Training (Search) region is the portion of the image used during update (localization).

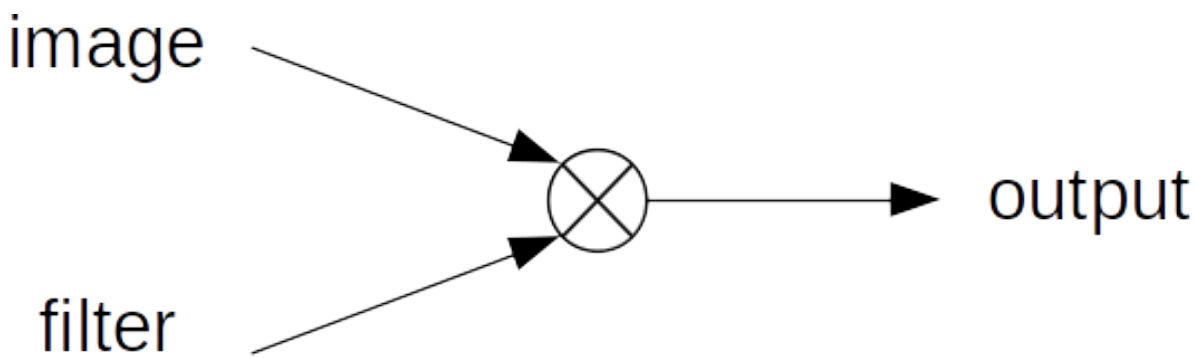
Localization step in correlation filters

Localization task: find the tracked object in a test image

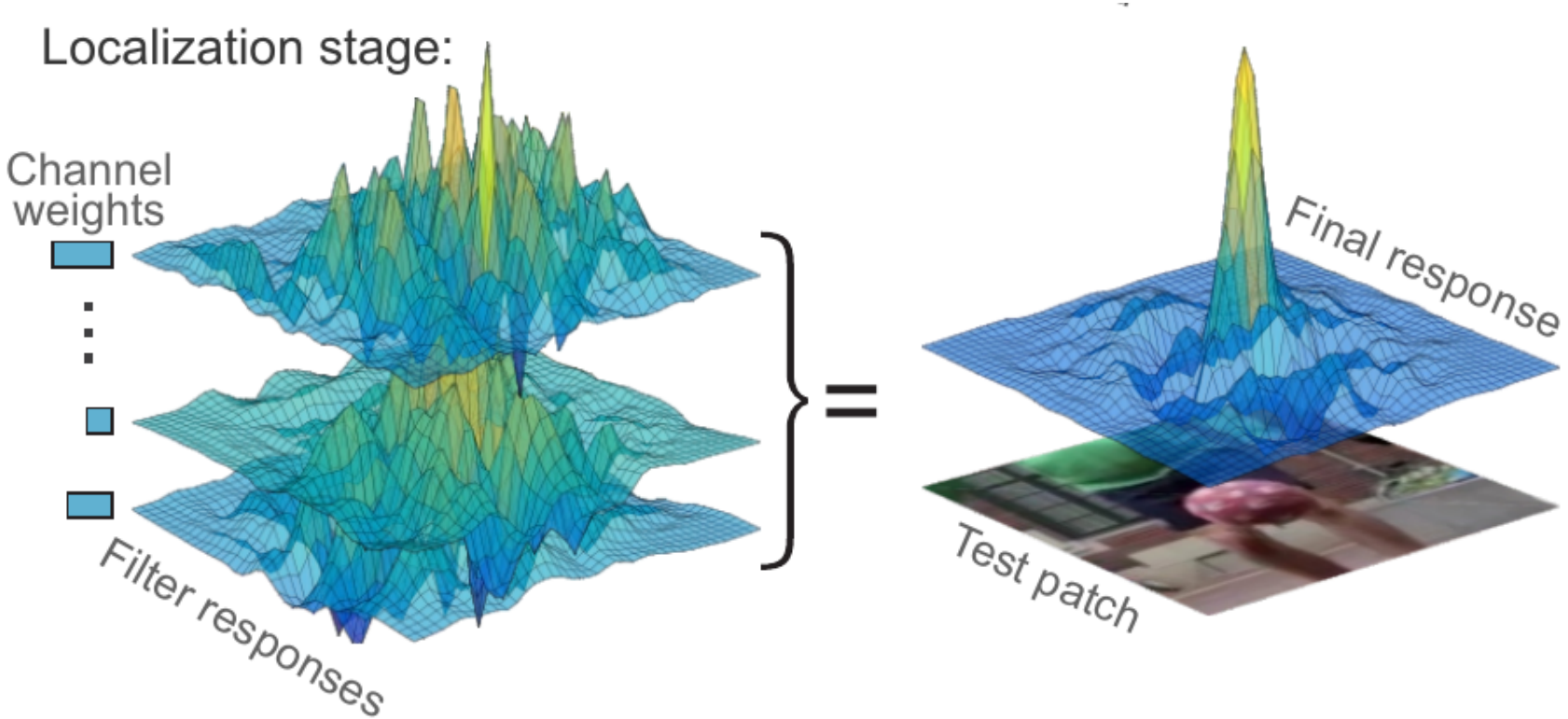
To find the target position in a single-channel image:

1. the output is calculated as the convolution of the test image with the filter [a]
2. target position is the one for which the peak value is observed in the output

[a] We'll see that this calculation can be generalized to the multi-channel case



Localization step in correlation filters: graphical intuition



Peak to Sidelobe Ratio

What is PSR

PSR (Peak to Sidelobe Ratio) is a measure of peak strength w.r.t. the rest of the signal. It is defined as:

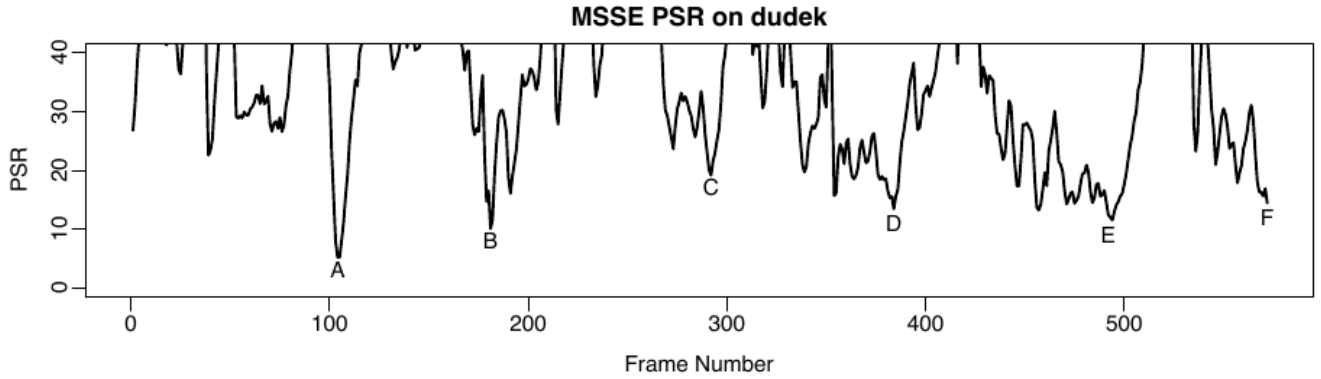
$$PSR \triangleq \frac{o_{max} - \mu}{\sigma}$$

with:

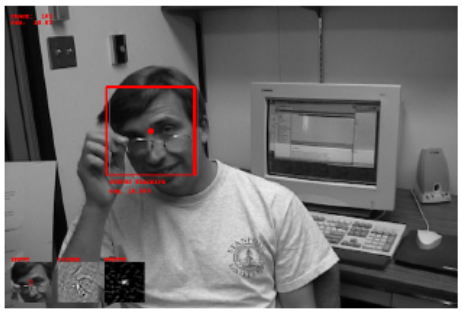
- o_{max}: maximum (peak) value of the output signal
- μ: mean value of the output signal (except values in a neighbourhood of the peak)
- σ: standard deviation of the output signal (except values in a neighbourhood of the peak)

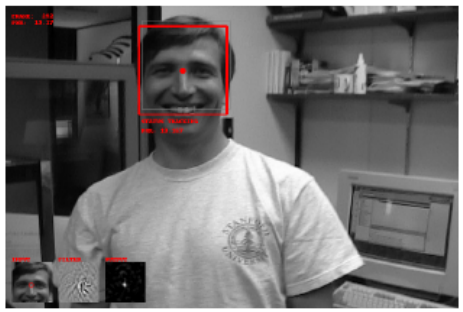
PSR is useful for failure detection: if it is too low, tracking has failed.

Monitoring PSR



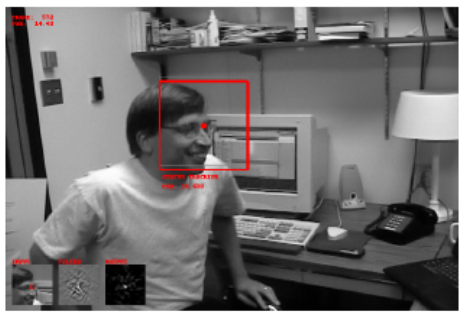

A - Occlusion


B - Remove Glasses


C - Fast Motion


D - Rotation


E - Off Frame


F - Rotation

MOSSE (2010)

Minimum Output Sum of Squared Error

MOSSE (Minimum Output Sum of Squared Error)

MOSSE tracker main aspects:

- Introduced in 2010
- Correlation filters were not commonly used before MOSSE
- It is very fast, and also robust to variations
- Gray-scale only (automatic conversion in OpenCV)
- Capable of detecting tracking failure and recovering from occlusion

MOSSE: training images and ground truth

Training images

Remind that tracker initialization requires a frame and a bounding box: the crop (content of the bounding box) is a training image. To have more images, MOSSE perturbs the crop 8 times, using random affine transformations.

Groundtruth

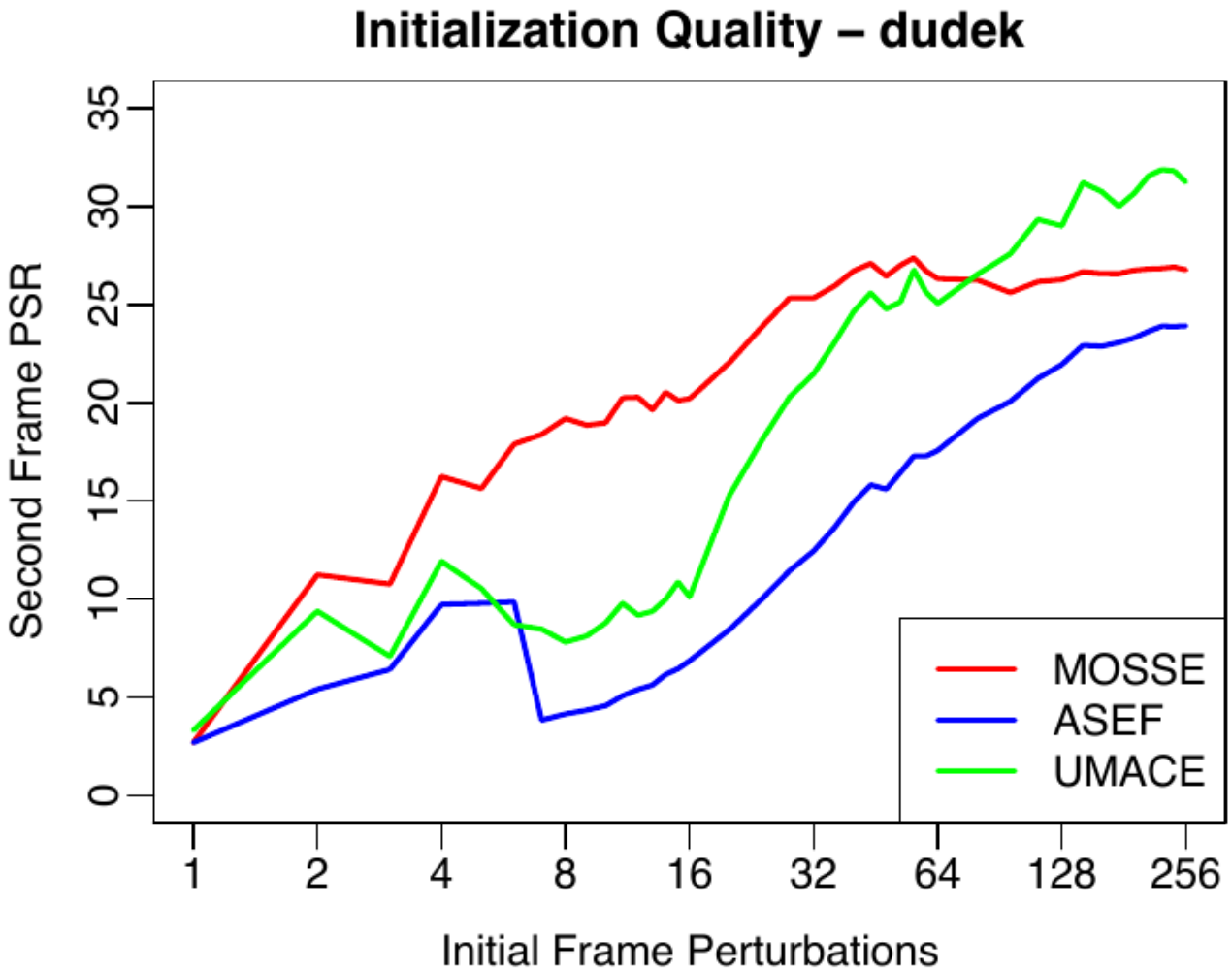
Given a training image, its ground truth is a compact 2D Gaussian shaped peak centered in the middle of the bounding box.

Online training images

Filter needs to quickly adapt in order to follow objects: after a localization, the new crop (without perturbations) is used to update the filter.

MOSSE: how many perturbations?

Given a measure of filter quality, you can empirically find a good number of perturbations.



MOSSE: update step

Filter initialization

We want to solve this problem:

$$F^* \leftarrow arg \min_{F^*} \sum_k ||I_k \odot F^* - G_k||^2$$

The closed-form solution is:

$N \leftarrow \sum_k G_k \odot I_k^*$

$D \leftarrow \sum_k I_k \odot I_k^*$

$F^* \leftarrow N/D$

$\circledast$: convolution (correlation) operator

$\odot$: element-wise product

$\mathcal{F}$: Fourier transform

*: complex conjugate

$I \triangleq \mathcal{F}(image)$

$G \triangleq \mathcal{F}(groundtruth)$

$F \triangleq \mathcal{F}(filter)$

$O \triangleq \mathcal{F}(output)$

MOSSE: online update step

Online update step

Numerator and denominator of F^* are updated by exponential moving average:

$$N \leftarrow \eta G \odot I^* + (1 - \eta)N$$

$$D \leftarrow \eta I \odot I^* + (1 - \eta)D$$

$$F^* \leftarrow N/D$$

They found that $\eta = 0.125$ allows the filter to quickly adapt to appearance changes while still maintaining a robust filter.

MOSSE: localization step

Localization step

$output \leftarrow image \circledast filter$

$(x_{new}, y_{new}) \leftarrow arg \max_{(x,y)} output(x, y)$

Localization step actually done in Fourier domain

$I = \mathcal{F}(image);$

$F = \mathcal{F}(filter);$

$O = I \odot F^{*};$

$output = \mathcal{F}^{-1}(O);$

$(x_{new}, y_{new}) \leftarrow arg \max_{(x,y)} output(x, y)$

MOSSE: additional observations

- The optimization problem is formulated in Fourier domain
- The search window as a number of pixels proportional to P, where P is the number of pixels in the filter
- The computational complexity is $\mathcal{O}(P \log P)$: this upper bound is given by the FFTs

MOSSE: example

This figure shows the ability of MOSSE tracker to quickly adapt to scale and rotation changes. It is also capable of detecting tracking failure and recovering from occlusion.

KCF (2014)

Kernelized Correlation Filters

KCF (Kernelized Correlation Filters)

KCF tracker main aspects:

- Introduced in 2014
- Tries to generalize the MOSSE approach seeing it as a linear regression problem
- Non-linear regressions lead to non-linear filters, which can improve overall tracking quality
- Non-linearity requires a little overhead, but the same computational complexity is maintained
- Can handle multiple channels simultaneously, whether they are color channels or HOG descriptors

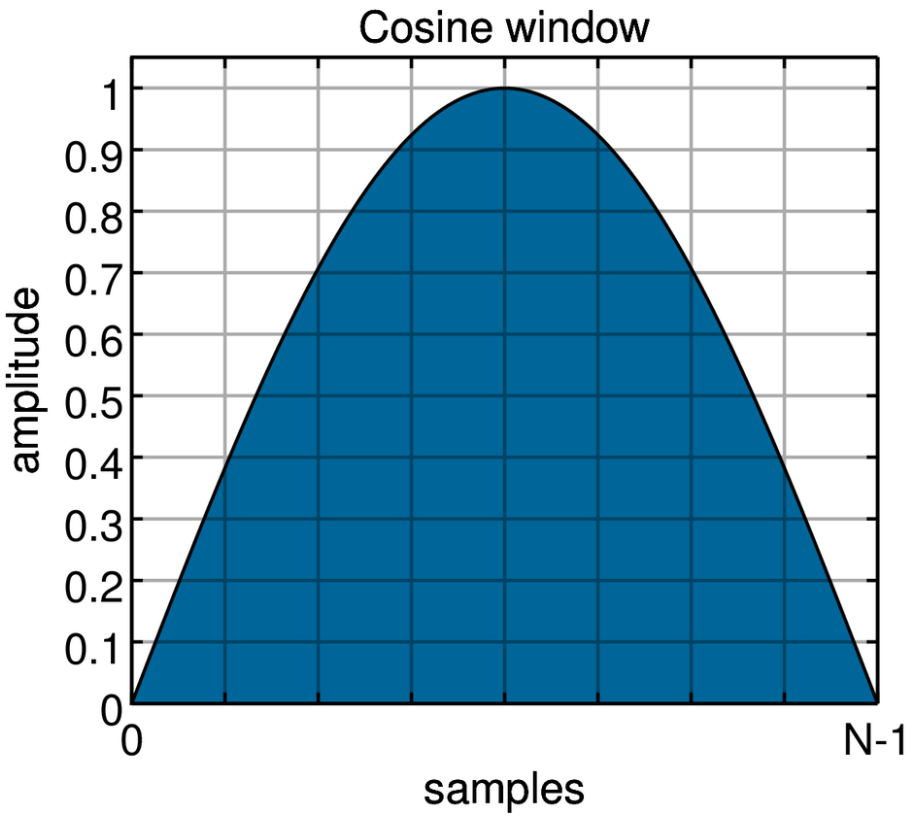
KCF: training images

Training images are cyclic shifts of the original crop, which is larger than the target, to provide some context.

Figure: Examples of vertical cyclic shifts of a base sample. Our Fourier domain formulation allows us to train a tracker with *all* possible cyclic shifts of a base sample, both vertical and horizontal, without iterating them explicitly. Artifacts from the wrapped-around edges can be seen (top of the left-most image), but are mitigated by the cosine window and padding.

KCF: cosine window

The input patches are weighted by a cosine window, which smoothly removes discontinuities at the image boundaries caused by the cyclic shifts.



KCF: how can we achieve non-linearity?

First of all, we need to generalize the MOSSE approach, that is:

$$F^* \leftarrow arg \min_{F^*} \sum_k ||I_k \odot F^* - G_k||^2$$

Generalization

- Element-wise product in the Fourier domain is a convolution in the space domain
- Convolution is a linear operator and therefore can be reformulated[a] as a Matrix product

[a] Using Toeplitz Matrices and some reshape, steps omitted here.

So let's rewrite the objective function in the time domain, with the Matrix product instead of the convolution.

KCF: linear update step

Linear update step as reformulation of MOSSE

We want to solve this problem:

$$f \leftarrow arg \min_f \sum_k ||f^T i_k - g_k||^2 + \lambda ||f||^2$$

The closed-form solution is:

$$F \leftarrow \frac{\sum_k G_k \odot I_k^*}{\lambda \mathbb{1} + \sum_k I_k \odot I_k^*}$$

Where $\mathbb{1}$ is a matrix of ones and λ is a positive penalty hyperparameter: the use of λ is another difference with MOSSE.
Remark: this is a Ridge regression problem, i.e. a linear regression with L2 regularization.

KCF: how we can achieve non-linearity with the kernel trick

Mapping the images

We can map an image i to a non-linear feature-space using a function $\phi(i)$.
We can define the kernel function K as: $K(i_1, i_2) = \phi^T(i_1)\phi(i_2)$.

Linear regression (MOSSE)

$\exists \alpha$ *s.t.*, for the optimal filter f:

$$f = \sum_j \alpha_j i_j$$

The output for input image i is:

$$output = f^T i = \sum_j \alpha_j i_j^T i$$

Non-linear regression

$\exists \alpha$ *s.t.*, for the optimal filter f:

$$f = \sum_j \alpha_j \phi(i_j)$$

The output for input image i is:

$$output = f^T i = \sum_j \alpha_j K(i_j, i)$$

Remark: the function ϕ has to satisfy some properties.

KCF: non-linear update step

Non-linear update step

We want to solve this problem:

$$\alpha \leftarrow arg \min_{\alpha} \sum_{k} || \sum_{j} \alpha_j K(i_j, i_k) - g_k||^2 + \lambda||f||^2$$

For which we now a closed form solution.

Online update step

We train a new model at the new position, and linearly interpolate the obtained values of the ones from the previous frame, to provide the tracker with some memory.

KCF: localization step

Linear localization step

$$output \leftarrow image \circledast filter$$

$$(x_{new}, y_{new}) \leftarrow arg \max_{(x,y)} output(x, y)$$

KCF: localization step

Linear localization step

$$output \leftarrow \sum_{c=1}^{C} image_c \circledast filter_c$$

$$(x_{new}, y_{new}) \leftarrow arg \max_{(x,y)} output(x, y)$$

With:

- c: channel index
- C: number of channels

KCF: localization step

Linear localization step

$$output \leftarrow \sum_{c=1}^{C} image_c \circledast filter_c$$

$$(x_{new}, y_{new}) \leftarrow arg \max_{(x,y)} output(x, y)$$

With:

- c: channel index
- C: number of channels

Non-linear localization step

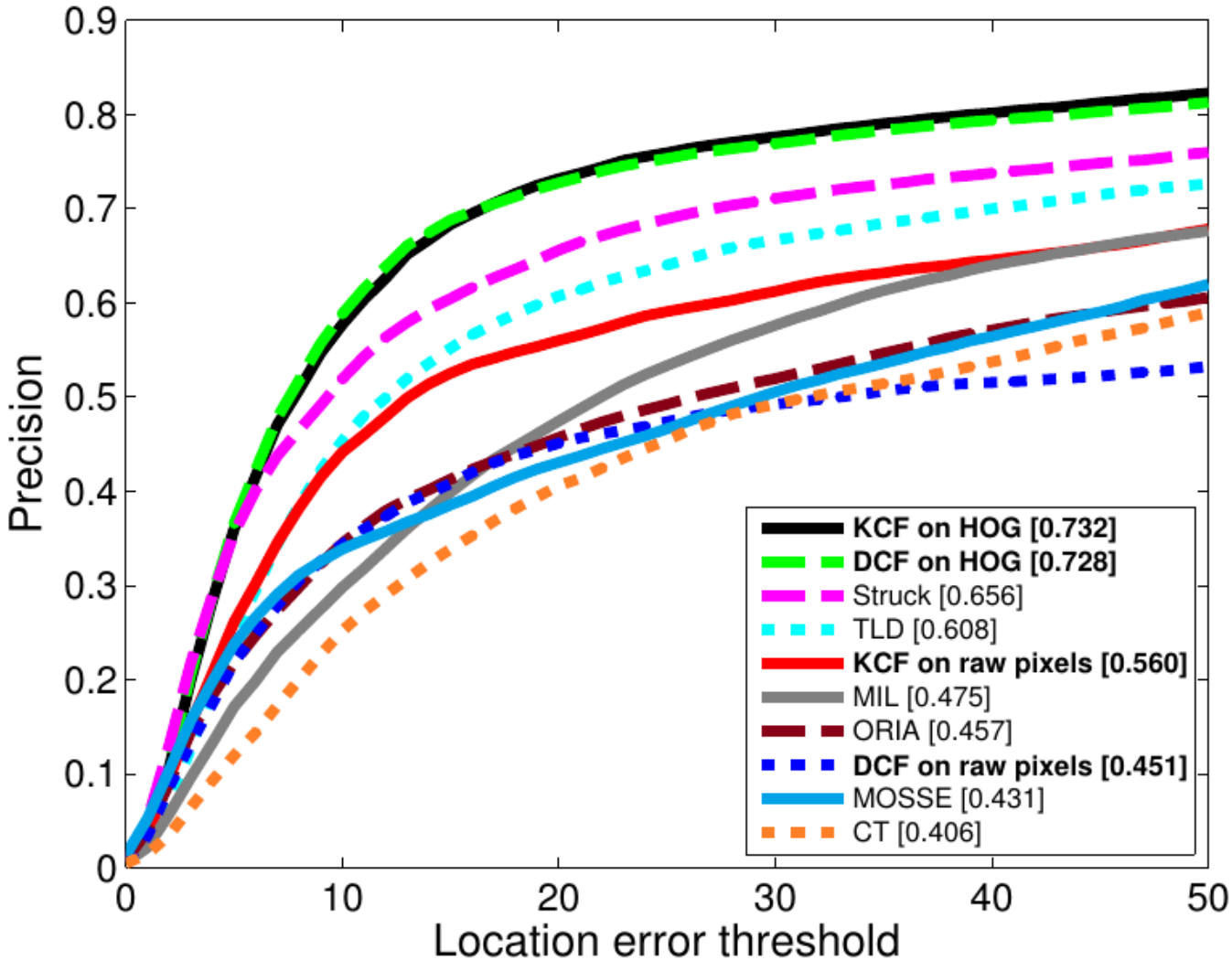
$$output \leftarrow \sum_{c=1}^{C} \sum_{j} \alpha_{jc} K(i_{jc}, image_c)$$

$$(x_{new}, y_{new}) \leftarrow arg \max_{(x,y)} output(x, y)$$

KCF: algorithm in Matlab code, with a Gaussian kernel

```
% INPUTS
%  imgTrain: training image patch, m x n x c
%  gndTr: regression target, Gaussian-shaped, m x n
%  imgTest: test image patch, m x n x c
% OUTPUT
%  responses: detection score for each location, m x n

function alpha = train(imgTrain, gndTr, sigma, lambda)
  k = kernel_correlation(imgTrain, imgTrain, sigma);
  alpha = fft2(gndTr) ./ (fft2(k) + lambda);
end

function responses = detect(alpha, imgTrain, imgTest, sigma)
  k = kernel_correlation(imgTest, imgTrain, sigma);
  responses = real(ifft2(alpha .* fft2(k)));
end

function k = kernel_correlation(img1, img2, sigma)
  c = ifft2(sum(conj(fft2(img1)) .* fft2(img2), 3));
  d = img1(:)'*img1(:) + img2(:)'*img2(:) - 2 * c;
  k = exp(-1 / sigma^2 * abs(d) / numel(d));
end

```

KCF: quality comparison with other algorithms



Experimental results on 50 videos dataset.

KCF: performance comparison with other algorithms

	Algorithm	Feature	Mean precision (20 px)	Mean FPS
Proposed	KCF	HOG	**73.2%**	172
	DCF		**72.8%**	**292**
	KCF	Raw pixels	56.0%	154
	DCF		45.1%	278
Other algorithms	Struck [7]		65.6%	20
	TLD [4]		60.8%	28
	MOSSE [9]		43.1%	**615**
	MIL [5]		47.5%	38
	ORIA [14]		45.7%	9
	CT [3]		40.6%	64

Experimental results on 50 videos dataset.

CSRT (2017)

Channel and Spatial Reliability Tracker

CSRT (Channel and Spatial Reliability Tracker)

As KCF, this is a multi-channel tracker: it can use color images (and also features like HOG). This type of tracker introduces two important concepts:

- Channel reliability: channel weights reflect channel-wise quality of filters
- Spatial reliability: adjusts the filter support to the part of the object suitable for tracking

CSRT: localization step

Localization step

$$output \leftarrow image \circledast filter$$

$$(x_{new}, y_{new}) \leftarrow arg \max_{(x,y)} output(x, y)$$

CSRT: localization step

Localization step

$$output \leftarrow \sum_{c=1}^{C} image_c \circledast filter_c$$

$$(x_{new}, y_{new}) \leftarrow arg \max_{(x,y)} output(x, y)$$

With:

- c: channel index
- C: number of channels

CSRT: localization step

Localization step

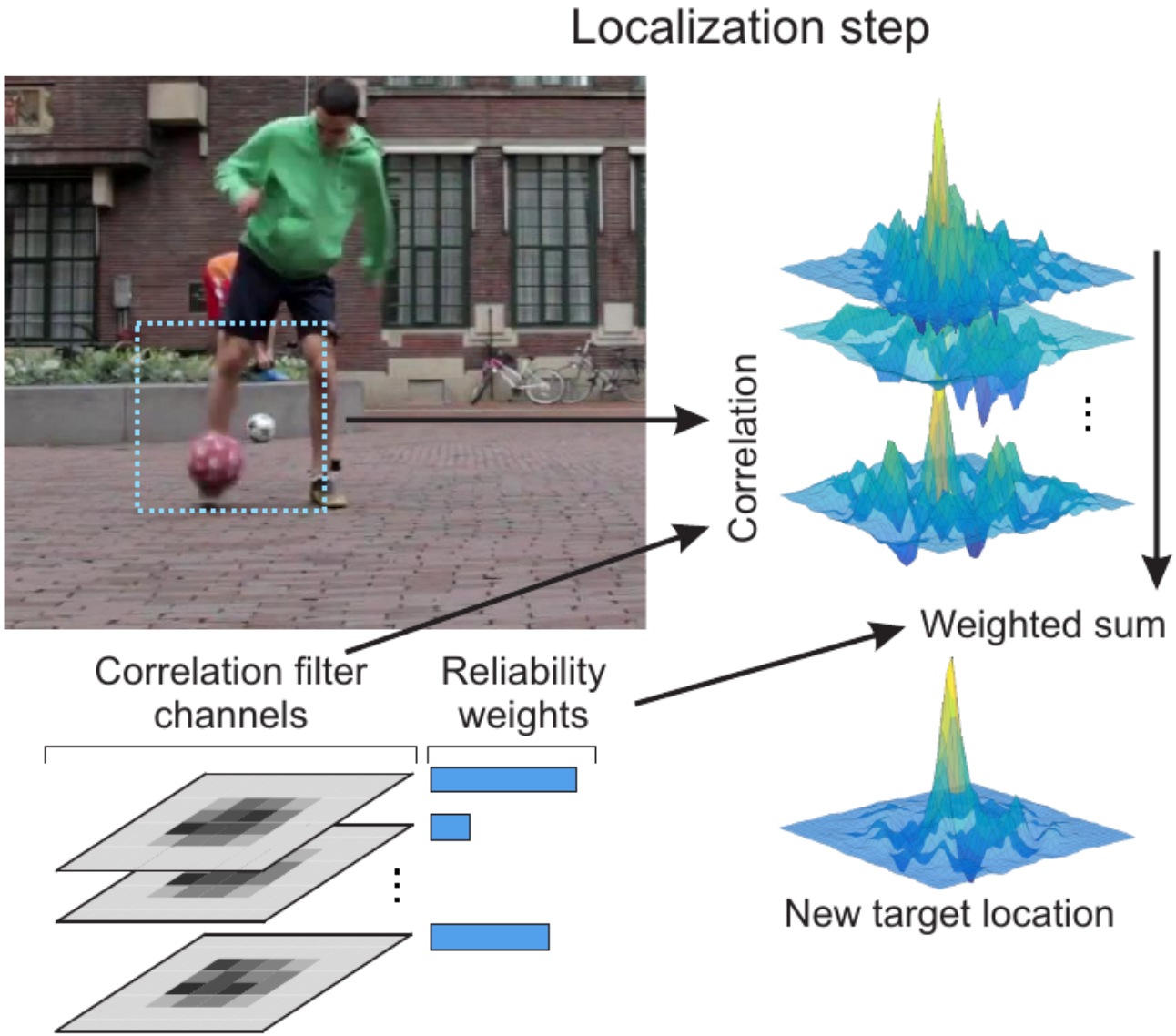
$$output \leftarrow \sum_{c=1}^{C} image_c \circledast filter_c \; w_c$$

$$(x_{new}, y_{new}) \leftarrow arg \max_{(x,y)} output(x, y)$$

With:

- c: channel index
- C: number of channels
- w_c: weight of channel c
- $\sum_{c=1}^{C} w_c = 1$

CSRT: localization step



CSRT: channel reliability

Many channels can be used simultaneously: channel reliability allows to give different importance (weight) to the various channels.
The weight w_c for the channel c is given by the product of:

- channel learning reliability $w_c^{(lrn)}$, calculated in the filter learning stage
- channel detection reliability $w_c^{(det)}$, calculated in the target localization stage

CSRT: channel learning reliability

The idea is that the higher the response of the filter applied on train image, the more important the channel should be.

$$w_c^{(lrn)} = \max(i_c \circledast f_c)$$

CSRT: channel detection reliability

The idea is that if there is only a valid target position, the ratio between outputs in 2^{nd} local maxima ($o_c^{(2)}$) and in 1^{st} local maxima ($o_c^{(1)}$) should be low (i.e., $o_c^{(1)} >> o_c^{(2)}$).

$$w_c^{(det)} = \quad 1 - \frac{o_c^{(2)}}{o_c^{(1)}}$$

$o_c^{(j)}$ is the j-th local highest output value in channel c.

CSRT: channel detection reliability

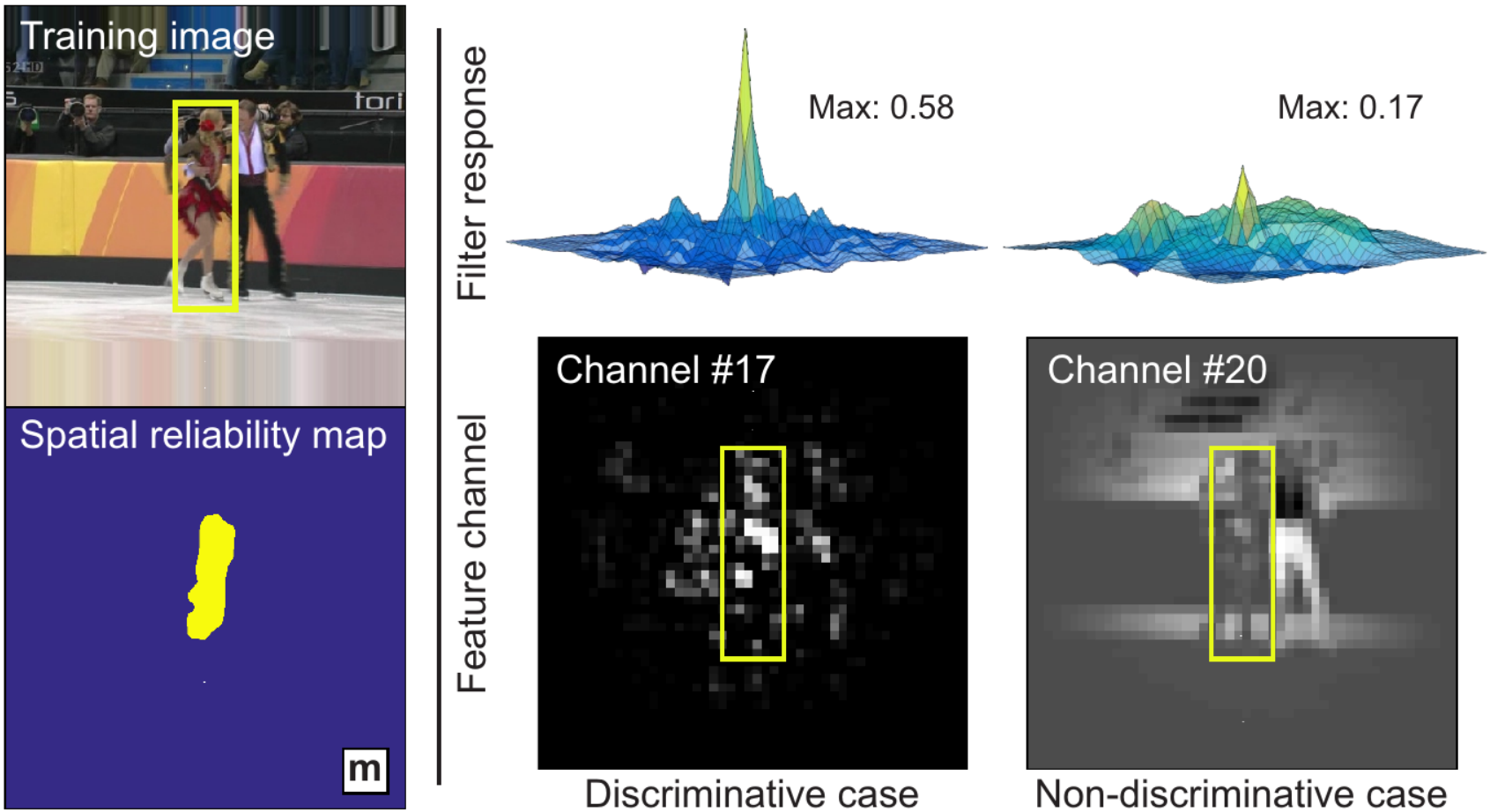
The idea is that **if there is only a valid target position**, the ratio between outputs in 2^{nd} local maxima ($o_c^{(2)}$) and in 1^{st} local maxima ($o_c^{(1)}$) should be low (i.e., $o_c^{(1)} >> o_c^{(2)}$).

$$w_c^{(det)} = \max(1 - \frac{o_c^{(2)}}{o_c^{(1)}},\ 0.5)$$

But the assumption made is not necessarily true, therefore the weight is forced to be $>= 0.5$.

$o_c^{(j)}$ is the j-th local highest output value in channel c.

CSRT: channel reliability



On a discriminative feature channel the filter response is much stronger and less noisy than on a non-discriminative channel.

CSRT: spatial reliability

The *spatial reliability map* is a binary mask that specifies locations of zero and non-zero pixels in the learned filter.

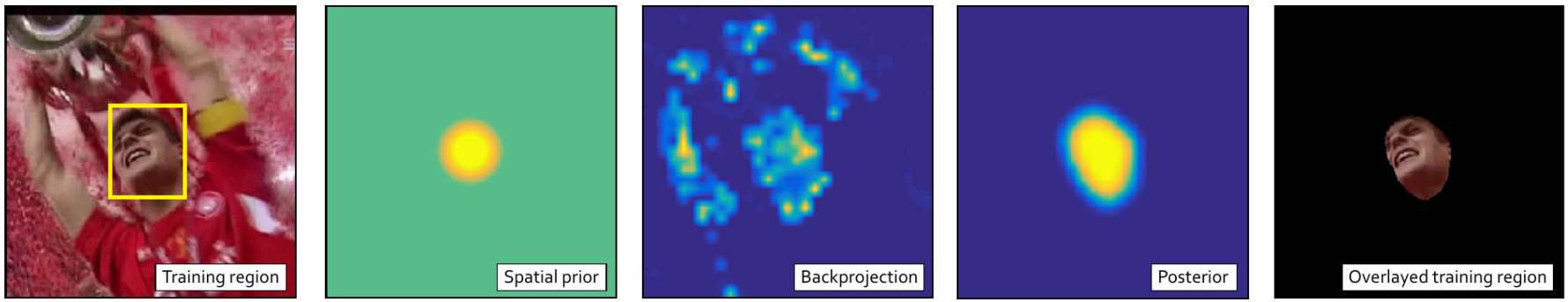
Why a spatial reliability map?

Not all the pixels inside the bounding box are equally important for the tracking task: some pixels will belong to the foreground (i.e., to the tracked object) while others will belong to the background.

Create the mask is a task of segmentation: for this purpose two histograms are maintained, one for the background and one for the foreground (histograms are then updated by exponential moving average).

CSRT: spatial reliability

In the image below it is intuitively represented how the spatial reliability map can be obtained.



Some remarks on CSRT training images:

- Spatial reliability enables an arbitrary large search region (useful to have information about the background)
- Cyclic shifts are not made

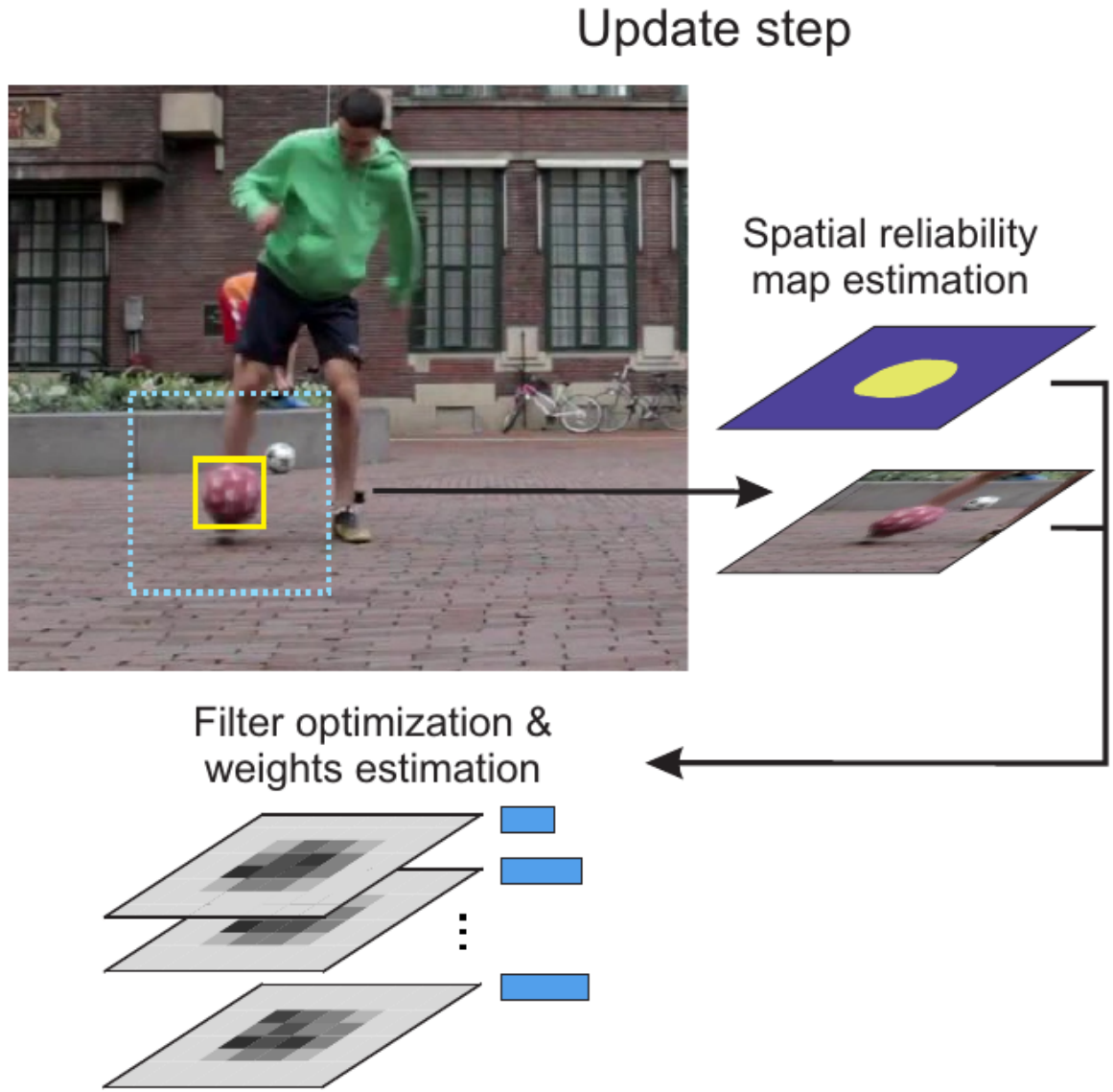
CSRT: update step

Constrained correlation filter learning

Optimal filter is found using the same objective function as in KCF, but adding the constraint: $f \equiv f \odot m$
This constraint determines that using the same approach as in MOSSE or KCF becomes intractable. Therefore, the solution is sought using an iterative approach.

Where the *spatial reliability map* m is a binary mask that specifies locations of zero and non-zero pixels in the learned filter.

CSRT: update step



CSRT: online update step

Online update step

Both filters and channel weights are updated by exponential moving average.

CSRT: pseudocode

Require:

Image $\mathbf{I}_t$, object position on previous frame $\mathbf{p}_{t-1}$, scale s_{t-1}, filter $\mathbf{h}_{t-1}$, color histograms $\mathbf{c}_{t-1}$, channel reliability $\mathbf{w}_{t-1}$.

Ensure:

Position $\mathbf{p}_t$, scale s_t and updated models.

Localization and scale estimation:

1: New target location $\mathbf{p}_t$: position of the maximum in correlation between $\mathbf{h}_{t-1}$ and image patch features $\mathbf{f}$ extracted on position $\mathbf{p}_{t-1}$ and weighted by the channel reliability scores $\mathbf{w}$ (Sect. 3.3).

2: Using per-channel responses, estimate detection reliability $\tilde{\mathbf{w}}^{(\mathrm{det})}$ (Sect. 3.3).

3: Using location $\mathbf{p}_t$, estimate new scale s_t.

Update:

4: Extract foreground and background histograms $\tilde{\mathbf{c}}^f$, $\tilde{\mathbf{c}}^b$.

5: Update foreground and background histograms $\mathbf{c}_t^f = (1-\eta_c)\mathbf{c}_{t-1}^f + \eta_c\tilde{\mathbf{c}}^f$, $\mathbf{c}_t^b = (1-\eta_c)\mathbf{c}_{t-1}^b + \eta_c\tilde{\mathbf{c}}^b$.

6: Estimate reliability map $\mathbf{m}$ (Sect. 3.2).

7: Estimate a new filter $\tilde{\mathbf{h}}$ using $\mathbf{m}$ (Algorithm 1).

8: Estimate learning channel reliability $\tilde{\mathbf{w}}^{(\mathrm{lrn})}$ from $\mathbf{h}$ (Sect. 3.3).

9: Calculate channel reliability $\tilde{\mathbf{w}} = \tilde{\mathbf{w}}^{(\mathrm{lrn})} \odot \tilde{\mathbf{w}}^{(\mathrm{det})}$

10: Update filter $\mathbf{h}_t = (1-\eta)\mathbf{h}_{t-1} + \eta\tilde{\mathbf{h}}$.

11: Update channel reliability $\mathbf{w}_t = (1-\eta)\mathbf{w}_{t-1} + \eta\tilde{\mathbf{w}}$.

CSRT: performance comparison with other algorithms

Tracker		EAO	A_{av}	R_{av}	fps
CSRT		① 0.338	② 0.51	① 0.85	③ 13.0
CCOT	ECCV2016	② 0.331	① 0.52	① 0.85	0.6
CCOT*	ECCV2016	③ 0.274	① 0.52	② 1.18	1.0
SRDCF	ICCV2015	0.247	① 0.52	③ 1.50	7.3
KCF	PAMI2015	0.192	③ 0.48	2.03	① 115.7
DSST	PAMI2016	0.181	③ 0.48	2.52	② 18.6
Struck	ICCV2011	0.142	0.42	3.37	8.5

EAO, A_{av}, fps: HIGHER is better

R_{av}: LOWER is better

MOSSE vs KCF vs CSRT

As tracking algorithms based on correlation filters, we have seen:

- MOSSE (2010)
- KCF (2014)
- CSRT (2017)

$\downarrow$ better quality of tracking
$\uparrow$ faster

Which one to choose?

The choice depends on the computational capabilities of your device, and on the latency/throughput required by your application. In general, you would like to choose the most accurate that respects the computational constraints.

Theory recap

We've seen:

- Different mathematical approaches to tracking
- Tradeoffs to keep in mind when choosing a tracking algorithm
- How can a tracker algorithm recognize failure

Now we'll try to understand more practical concepts

Application

Some considerations

- We will use OpenCV 4.1.0 for Python: different versions can have different function signatures, or even missing features
- The size of the video frame strongly affects both the calculation time and the results of the algorithms: it should be fixed at the beginning and no longer changed (a lower resolution can speed up very much the scripts without losing in tracking quality)

Case study description

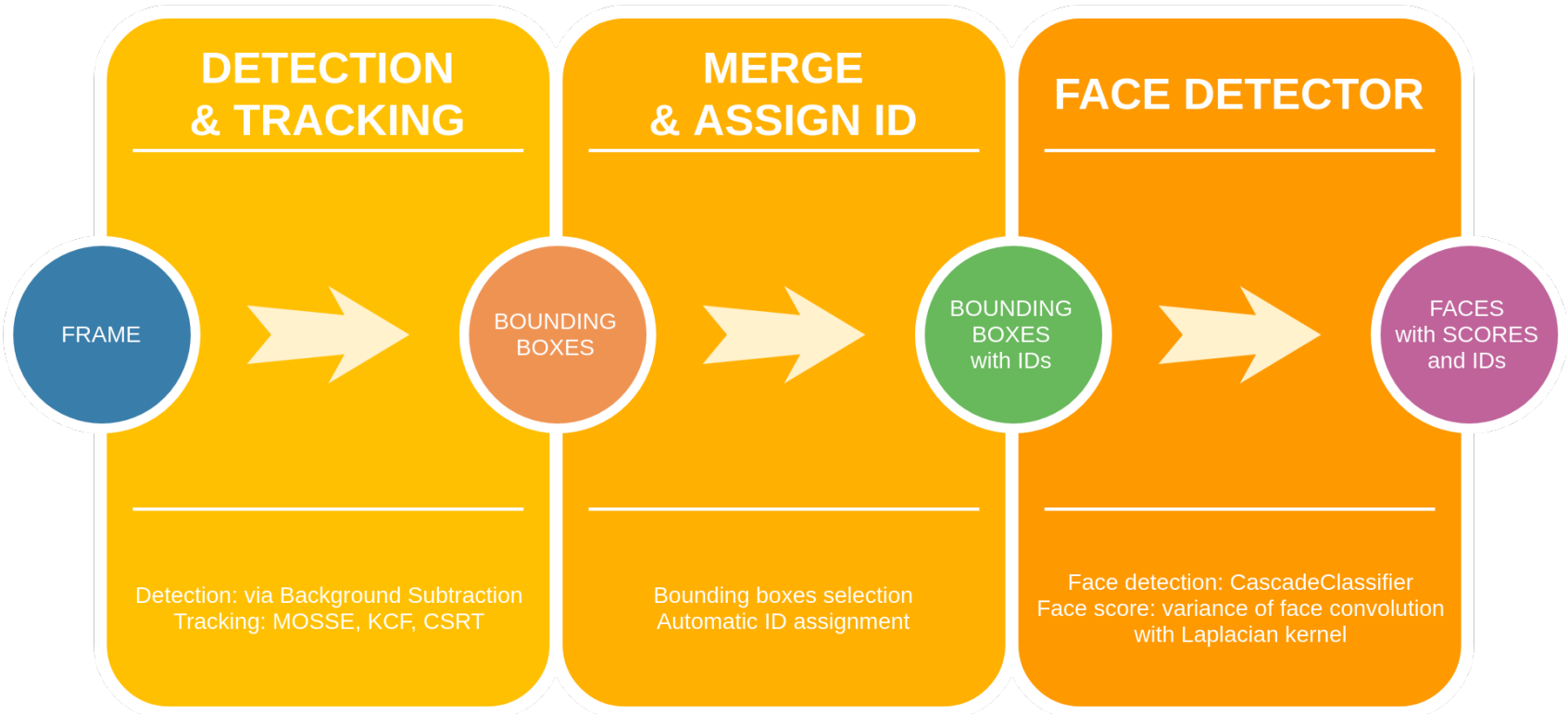
Description

We have a video camera and potentially many people could pass by it.

Objective

We want to register the (best) face(s) of each person.

Pipeline


DETECTION & TRACKING
MERGE & ASSIGN ID
FACE DETECTOR
FRAME
BOUNDING BOXES
BOUNDING BOXES with IDs
FACES with SCORES and IDs
Detection: via Background Subtraction
Tracking: MOSSE, KCF, CSRT
Bounding boxes selection
Automatic ID assignment
Face detection: CascadeClassifier
Face score: variance of face convolution with Laplacian kernel

Detection/Tracking task

How to properly take advantage of tracking?

The answer is not obvious: there are different possibilities.
Questions:

- When to do detection?
- When to do tracking?
- It makes sense to do both detection and tracking in the same frame?
- If so, how to treat the detection and tracking outputs? How to merge them?

Detection/Tracking task: bad ideas

Bad idea #1

We can make detection on the first frame, and track all detected objects in next frames.
Why it's a (very) bad idea:

- An object should appear in the first frame (necessary condition), otherwise it will never be detected
- It's very unlikely that objects of interest appears in the beginning of the video

Detection/Tracking task: bad ideas

Bad idea #2

We can make detection on the first frame, and track all detected objects in next frames. When all trackers fail, we do another detection, and so on.
Why it's a bad idea:

- An advantage of video tracking should be using fewer computational resources: in this approach, detection could be done in all frames
- A tracker can fail if the tracked object is temporarily occluded, so the tracker should keep looking for the object for a while
- When an object is tracked, no new objects can be detected.

Detection/Tracking task: ideas

Idea #1

We can make detection periodically starting in the first frame, and track all detected objects in next frames.
Why it can work:

- Since detection is made periodically, we can ideally detect any objects that appears for a number of frames greater or equal to the period
- Using only tracking in many frames can lead to using fewer resources: this of course is not always true, but it is if the objects appear sporadically

We must be careful to keep objects IDs in different periods.
Unfortunately, we have introduced the *period* hyperparameter, and last but not the least, in the case of overlapping we have to decide which bounding boxes maintain in transition frames (the first frame of the period, in which both tracking and tracking is done).

Detection/Tracking task: ideas

Idea #2

In the event that bounding boxes obtained from the detection overlap with bounding boxes obtained from the tracking, which of them maintain?

- Why might it be better to keep those from the detection: if tracking has begun drifting, using the detection output can help to re-center the object in the bounding box
- Why might it be better to keep those from the tracking: the tracker has information about what the object was in the previous frames and so can know better how the object might look

We can't say right now which approach is better, we just have to try them both in the laboratory.

Detection/Tracking task: ideas

Idea #3

In the case an object is (partially) occluded, the tracker can fail: how long to wait for the object to reappear?
We have to be particularly careful when a disappeared object is re-detected by the detector than by the tracker: we don't want to identify it as a new object!

Detection/Tracking task: ideas

Idea #4

In transition frames, the output of the detection isn't automatically related with the output of the tracking: we have to decide how to relate a detection bounding box with a tracking bounding box. To make these relations, we need some criteria: for example, we can use IoU (Intersection over Union) or Euclidean distance; ideally, if two bounding boxes refer to the same object, the IoU will be high while the distance of the centers will be low.

How to reach the objective

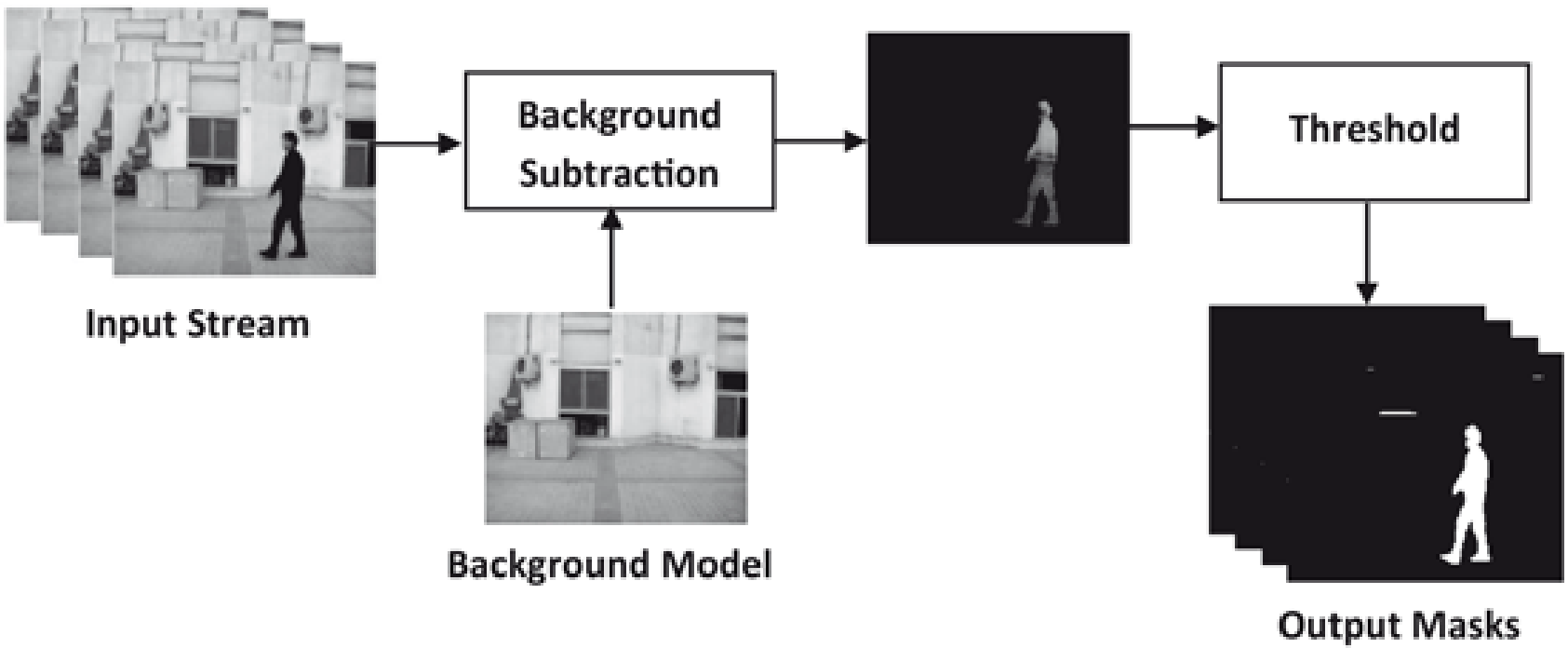
These are the general steps, for each frame:

1. Background subtraction to extract foreground objects
2. Process b/w image to remove noise
3. Object detection by contours analysis
4. Get tracking results (first frame excluded), and continue tracking for next frame
5. Merge detection and tracking results, assigning the right ID to objects already detected and an auto-incremental ID to new objects
6. For each object, search for a face inside it
7. Give a sharpness score to each face found

At the end, best faces are saved for each object.

Background subtraction

With these methods is possible to learn, frame by frame, which pixels represents the background.
Subtracting the background from the frame allows highlighting foreground objects.



Background subtraction in OpenCV

There are many BG subtractor algorithms in OpenCV, example creation of some background subtractor (with their default parameters):

```
bM = cv2.bgsegm.createBackgroundSubtractorMOG(history=200,
      nmixtures=5, backgroundRatio=0.7, noiseSigma=0)
bM2 = cv2.createBackgroundSubtractorMOG2(history=500,
      varThreshold=16, detectShadows=True)
bG = cv2.bgsegm.createBackgroundSubtractorGMG(
      initializationFrames=120, decisionThreshold=0.8)
bK = cv2.createBackgroundSubtractorKNN(history=500,
      dist2Threshold=400.0, detectShadows=True)
```

Example usage of a background subtractor:

```
fgmask = myBgSubtractor.apply(frame)
```

Noise removal

Background subtraction algorithms are not perfect: noise removal is needed. The kind of noise depends on the background subtractor algorithm and on the video in analysis.
To remove noise, we made these steps:

1. Median blur smoothing
2. Dilation

The dilation step is needed to reduce unwanted object splitting.

Object detection

To achieve object detection, we analyze contours: for each contour, we get the corresponding bounding rectangle, discarding small ones (small if area of rectangle is < 5000).

Tracking and ID management

In transition frames, we do both detection and tracking. So, these frames have two set of bounding boxes: the detected ones (DBBS), and the tracked ones from previous frame (TBBS). Detection bounding boxes are considered the effective objects bounding boxes. Bounding boxes from tracking decides the ID assigned to the objects, using the following criterion:

- for each box pair (d, t), with $d \in DBBS$, $T \in TBBS$, is being calculated IoU(d,t)
- pairs are ordered for decreasing IoU
- ID binding occurs according to this order (Note: t has already an ID)

DBBS: Detected Bounding Boxes Set; TBBS: Tracked Bounding Boxes Set

Tracking in OpenCV

There are different trackers in OpenCV, each one with the same interface. We pick CSRT as example but the other are analogous.

Tracker creation

```
tracker = cv2.TrackerCSRT_create()
```

Tracker initialization

```
tracker.init(frame, boundingBox)
# boundingBox as (left, top, width, height)
```

Tracker locate and update

```
(success, boundingBox) = trackers.update(frame)
# success is False if tracking has failed, True otherwise
# boundingBox is the location of the tracked object in the frame
```

MultiTracking in OpenCV

OpenCV has a MultiTracker class that can handle many Trackers at the same time.

MultiTracker issue (at least in OpenCV 4.1.0)

MultiTracker update method doesn't return success for each specific tracker: success is False if **at least one** tracker has failed. Therefore, it's impossible to differentiate failure management between trackers.

To avoid MultiTracker issue, we implement a class with the same interface, but that returns a list of successes instead of only a boolean success.

Face detection

An OpenCV CascadeClassifier is used: it is a class to detect objects in a video stream. The class exposes a method "load" to load a .xml classifier file: we use an HaarCascade pre-trained to detect faces inside the objects.
HaarCascade is a machine learning based approach where a cascade (ensemble) function is trained from a lot of positive and negative images.
The method to do the detection on an image is called "detectMultiScale".

Face score

We calculate the face score as the variance of the convolution between the gray-scale image of the face and the Laplacian kernel. The Laplacian kernel is:

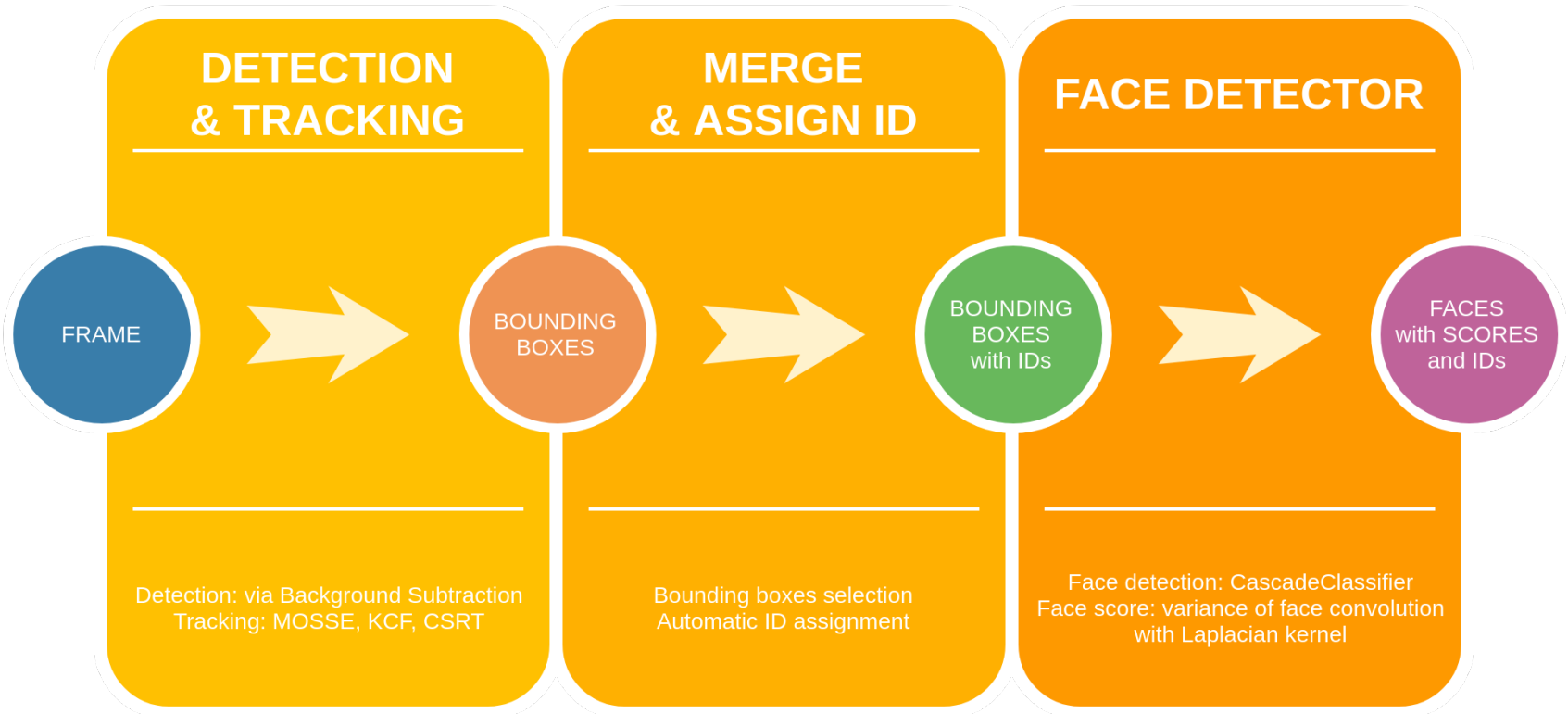
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The corresponding Python code is:

```
score = cv2.Laplacian(imgFaceGray, cv2.CV_64F).var()
```

The higher the score, the better the quality of the image (less blur). A threshold of 100 should be good (values range from 0 to above 1 thousand).

Application recap



Libraries

The following libraries are required for the exercise on Thursday:

- opencv-python 4.1.0
- opencv-contrib-python 4.1.0
- imutils
- matplotlib