COMPUTER VISION AND IMAGE PROCESSING

LAB SESSION 5

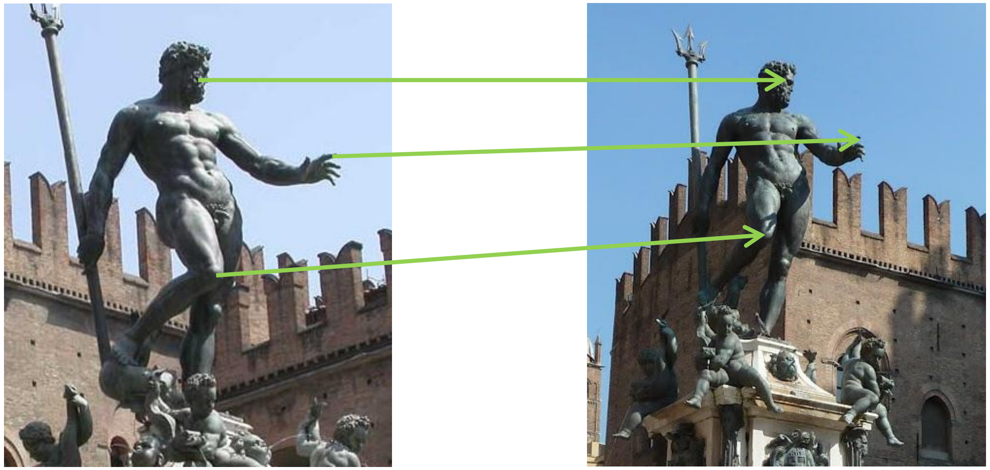
Local invariant Features for Object Detection

Alessio Tonioni - alessio.tonioni@unibo.it

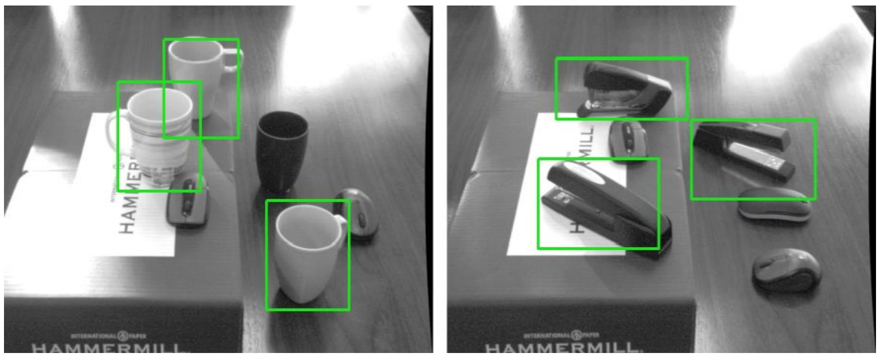
Correspondences

A great variety of computer vision problems can be dealt with finding **corresponding points** between images.

Corresponding (or homologous) points: image points which are the projection of the same 3D position from different points of view.
Being projection their appearance can vary greatly between one image and the other so establishing correspondences may be difficult.

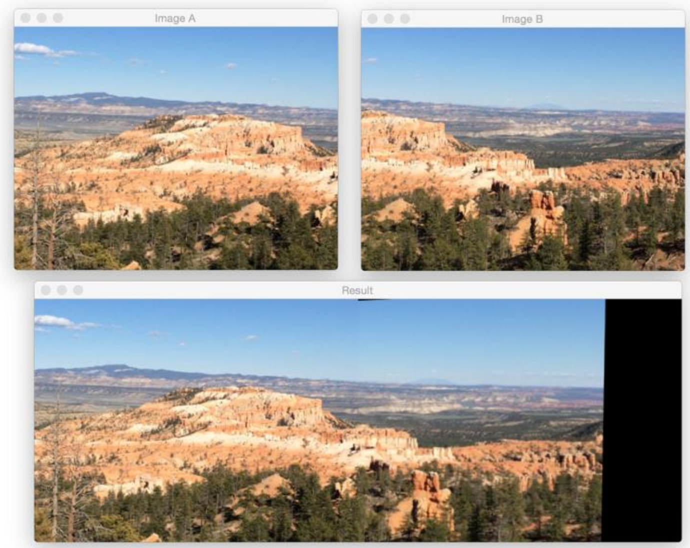


Exemplar tasks



Object Detection

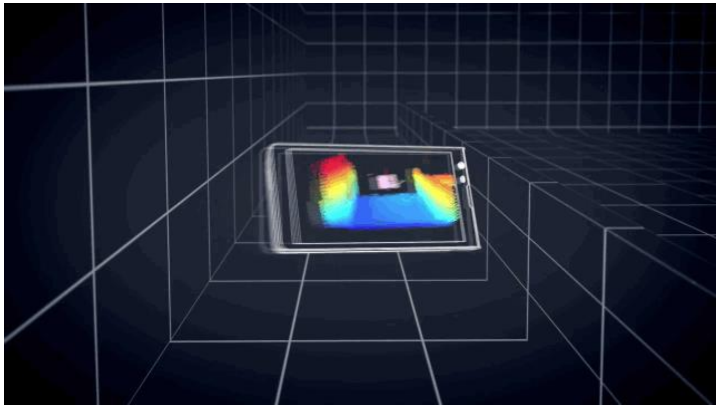
Visual Search



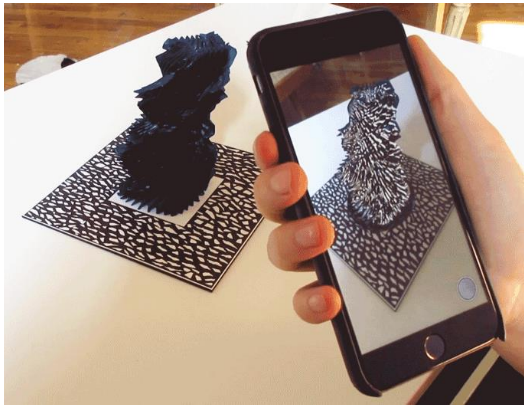
Mosaicing (a.k.a. panorama photo)

Exemplar tasks

3D Reconstruction



Camera Tracking

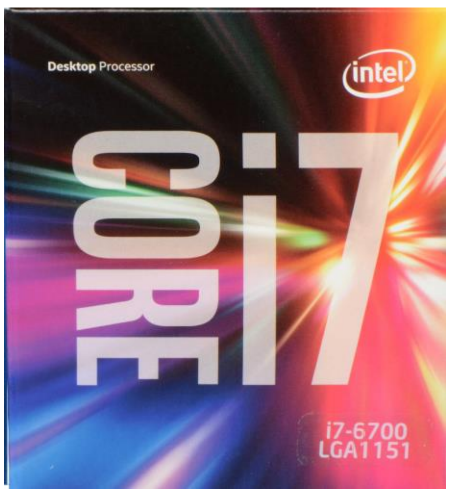


SLAM

Augmented Reality

Object Detection

Task: detect instances of objects in images (*scenes*), given one or more reference image depicting them (*models*).



Model image

Scene Image

Object Detection - Difficulties

Q: What can go wrong? What a good detection system should handle?

- Scale invariance: object in *scene* may appear at any scale, not only at the same resolution used for the *model* images.
- Rotation invariance: object may appear rotated or skewed in the *scene*.
- Photometric invariance: object may appear in any light condition.
- Occlusion: portion of the objects may not be visible in the *scene*.
- Perspective distortion: Object may appear fairly different if viewed from different camera viewpoint.

Object Detection - CNN

State of the art methods are based on *machine learning,* especially convolutional neural networks (i.e. CNN).

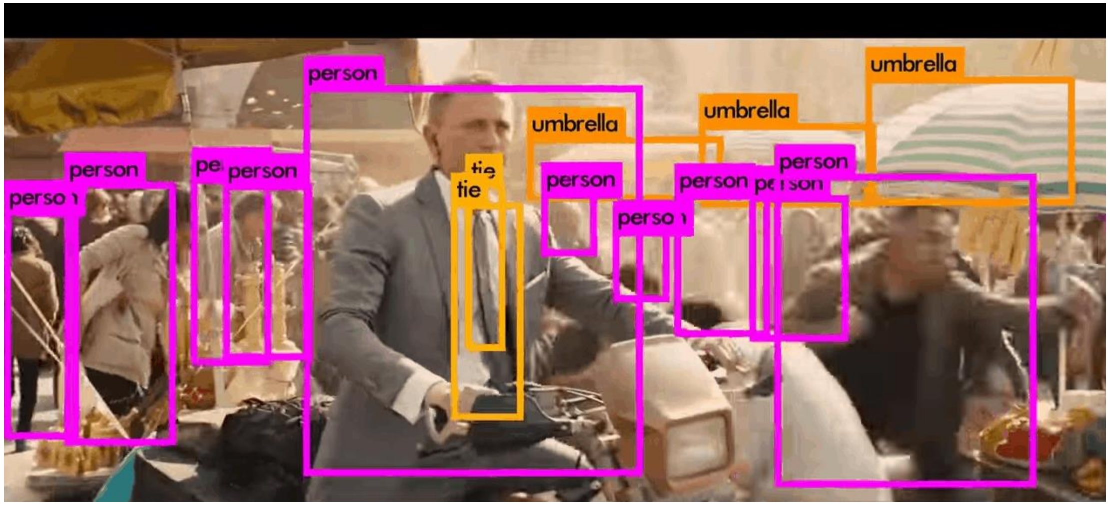
Pro: Astonishing real time performances for the detection of complex and highly variable categories of objects (persons, animals, vehicles…)

Cons: Requires thousand of *model* images for each category of objects we want to recognize and a lot of time to train the algorithm.

Example:

- Faster R-CNN [1]
- Yolo [2]
- SSD [3]

YoloV2 watching *Skyfall.* →



Local Invariant Features Paradigm

State of the art approach before the *machine learning* revolution. Proposed by David Lowe in 2004[4], allows to successfully identify objects in *scene* from a single *model* image per object.

Pros:

- Quite effective for the detection of textured objects.
- Scale/rotation and illuminance invariance.
- Works under partial occlusion.
- Only one *model* image per object required.
- Fully implementable in openCV with few lines of code.

Cons:

- Suffers from changes in camera viewpoint.
- Can be slow when the number of objects to recognize increases.
- Does not work well with deformable objects or to detect categories of objects.

Local Invariant Features Paradigm

Four steps:

1. Detection:
   Identify salient repeatable points (***Keypoints***) in *model* and *scene* images.
2. Description:
   Create a unique description of each point, usually based on its local pixel neighborhood.
3. Matching:
   Match point from *scene* and *model* according to a similarity function between the descriptors.
4. Position Estimation:
   Estimate the position of the object in the *scene* image given enough matching points.

1 - Keypoints Detection

Identify in each image a set of points that have good qualities for the following description and matching phases.
We call these points keypoints. A good keypoint detector should be:

- **Repeatable:** find the same keypoints in different views of the same object despite the transformation undergone by the image (i.e. different point of view, different illumination…).

- **Distinctive:** find keypoints surrounded by *informative* patterns of intensities (i.e. good candidate for creating a unique description and improve the probability of matching).

- **Fast:** it must be applied to each pixel on the image to find the most salient ones.

Q: Can we just grab random points? Can we use all of them?

1 - Keypoints Detection

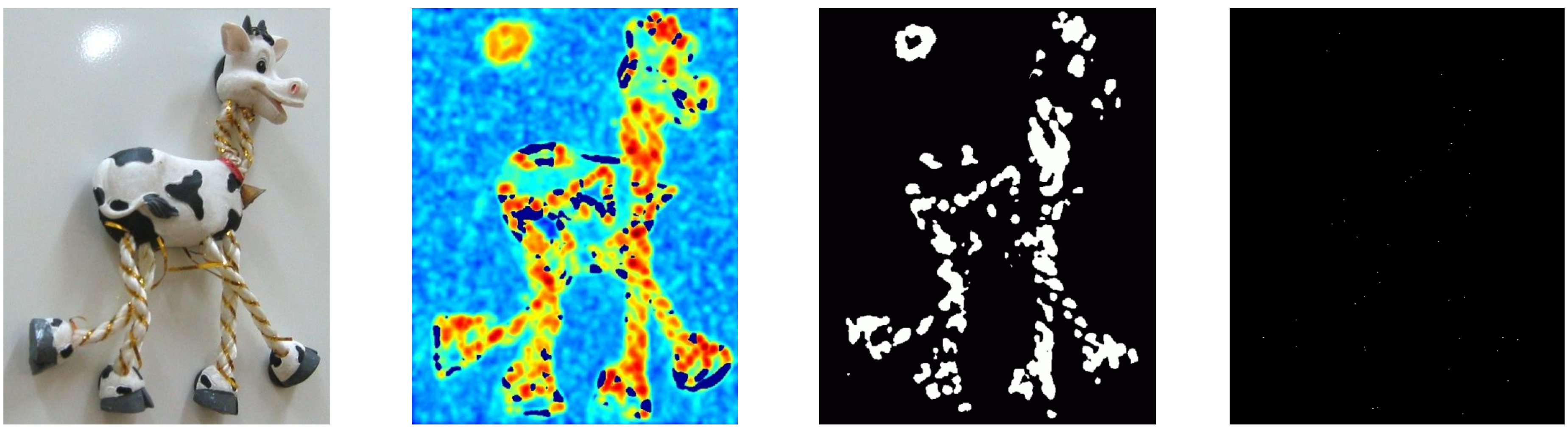
Different algorithms proposed over the years, quite often associated with a suitable descriptor:

- Difference of Gaussian (DOG) → SIFT [4]
- Fast-Hessian Detector → SURF [5]
- Features From Accelerated Segment Test (FAST) → BRISK [6], ORB [7]
- …

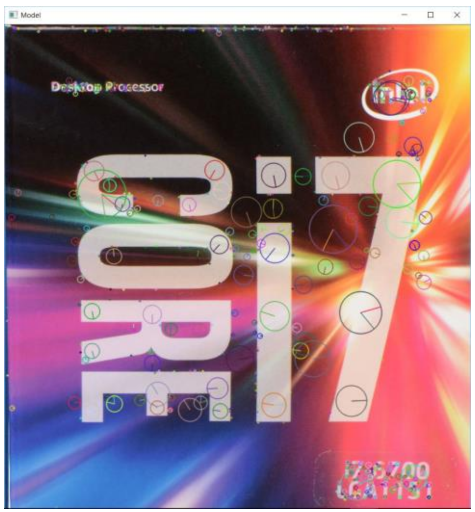
1 - Keypoints Detection

Common schema:

1. Compute a <u>saliency score</u> for each pixel location based on the response to different mathematical operators.
2. Keep only the points that are local maxima.
3. For each keypoint estimate the '*scale*' at which it is salient (*scale invariance*) and the orientation *(rotation invariance)*.



1 - Keypoints Detection



Keypoints extracted on the *model* (left) and *scene* (right) using DOG keypoint detector.

2 – Keypoints Description

Compute for each keypoint a unique description usually based on the nearby pixels (*descriptor support*).
A good keypoint descriptor should be:

- **Repeatable:** the descriptions computed at homologus points should be as similar as possible.
- **Distinctive:** capture the salient informations around the keypoint despite various nuisances (e.g. light changes).
- **Compact:** minimize memory occupancy to allow efficient matching.
- **Fast:** it is usually applied to hundred or thousand of keypoints in each image.

Q: Given the patch surrounding a keypoint, can we use raw pixel intensities as descriptor?

2 – Keypoints Description

Different algorithms provides different descriptions, the common idea is to describe keypoints using an array (*histogram*) of values that encodes the appearance of its local neighborhood.

The size of the support depends on the scale associated to the keypoint (i.e. ***scale invariance***).

The descriptor are computed according to the orientation associated to the keypoint (i.e ***rotation invariance***).

2 – Keypoint Description

The histogram used for the description could be made of:

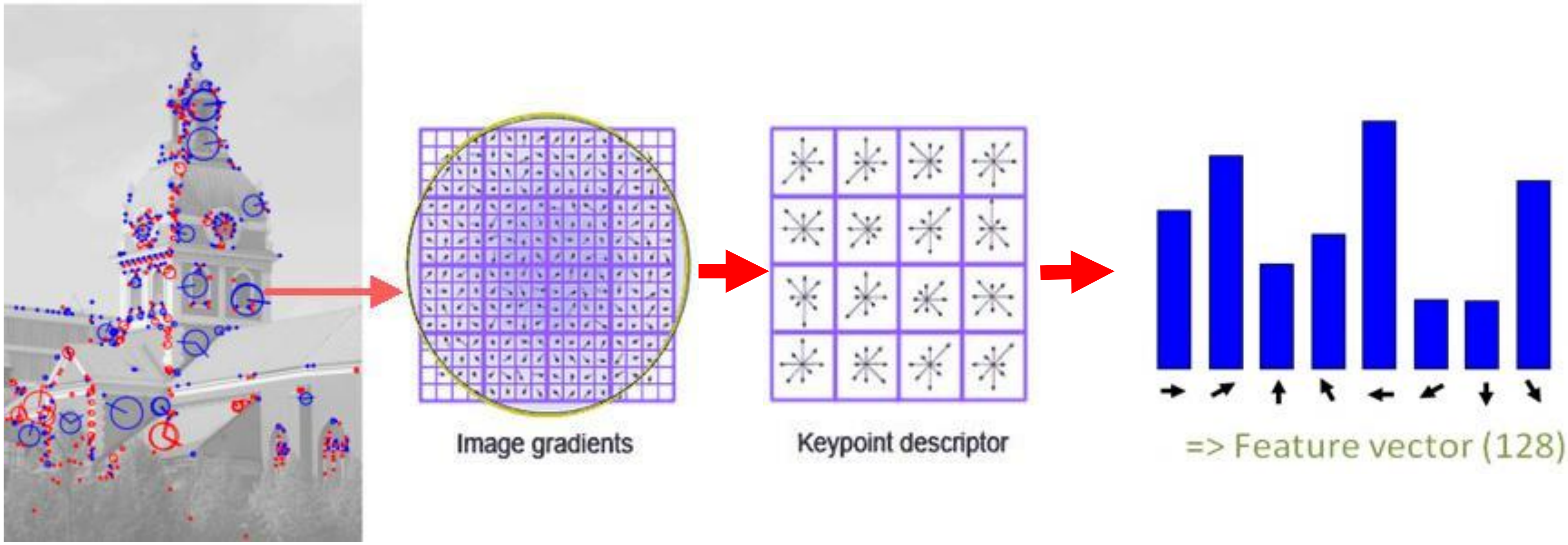
- floats → more distinctive, high memory footprint
- bits → less distinctive, small memory footprint (*binary descriptors*)

Some well known algorithms:

1. SIFT [4]: 128 floats array (4.096 bytes for each descriptor)
2. SURF [5]: 64 floats array (2.048 bytes for each descriptor)
3. BRISK[6]: 512 bit array (64 bytes for each descriptor)
4. ORB [7]: 256 bit array (32 bytes for each descriptor)

2 – Keypoint Description

A study case: SIFT.



3 – Feature Matching

Descriptors extracted from the *scene* are compared with those extracted from the *models* to find couples of similar ones.

Classic Nearest Neighbour (NN) Search problem:
Given a set of n points $R = \{r_0, \dots r_n\}$, a query point q and a distance function D; find the point $r_{nn} \in R$ such that

$$D(q, r_{nn}) \leq D(q, r_k) \quad \forall r_k \in R$$

In our scenario points are feature vectors and the distance function is *Euclidean distance* for floats or *Hamming distance* for bits.

Q: What is a naive solution to this problem?

3 – Feature Matching

Naive idea - Brute force matcher:

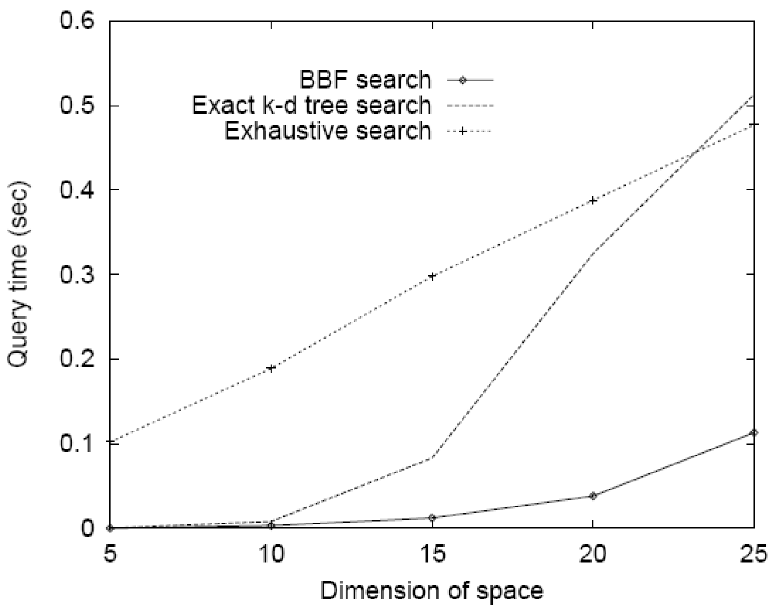
For each keypoint q detected in scene compute all the $D(q, r_{nn})$ to find the minimum.

Too slow to be applied in a lot of application, may sometimes be used with binary descriptor (distance function is a simple XOR between the descriptors).

Smart idea – indexing technique

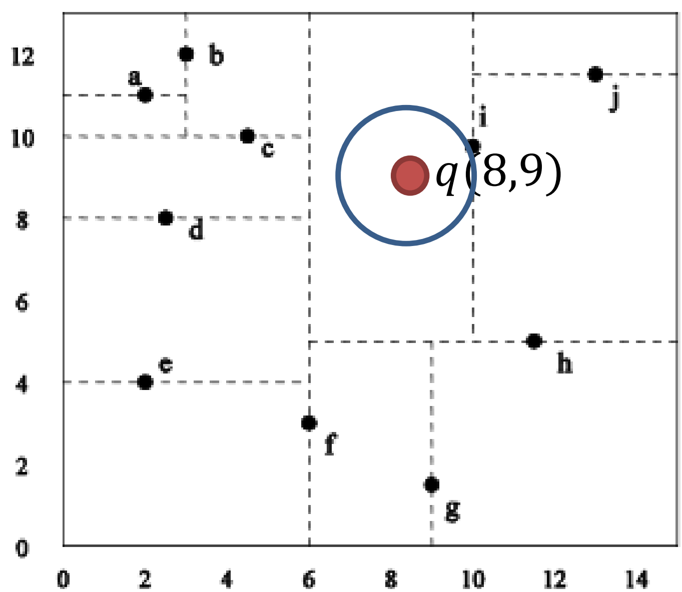
Use efficient indexing techniques borrowed from database management to speed up the search:

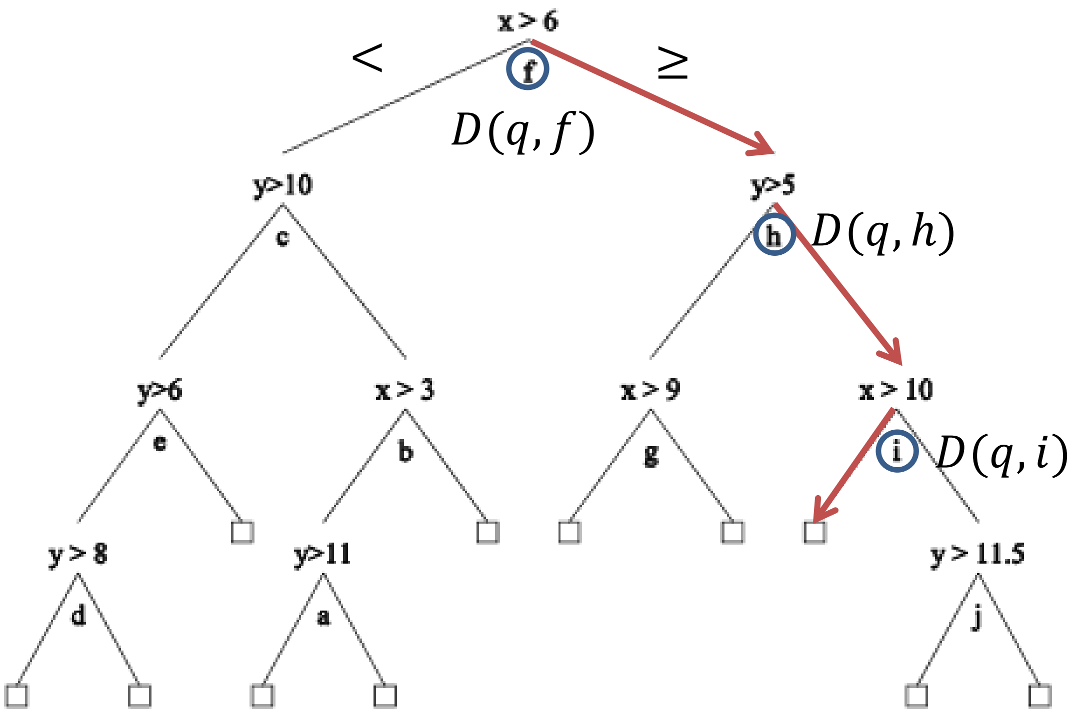
- Kd-tree [8] – exact
- BBF [9] – approximated
- LSH [10] –for binary descriptor



3 – Feature Matching

A simple case study: kd-tree with two dimension



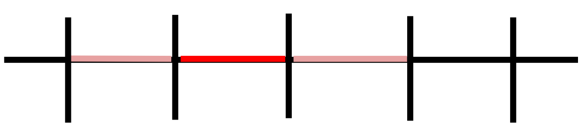


One time cost to build the tree, logaritmic number of distances to compute for each q.

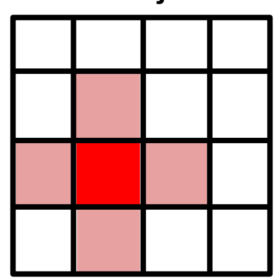
3 – Feature Matching

Kd-tree may be thought of as partitioning the space into 'bins', during backtracking the bins adjacent to the one containing the found leaf may be examined.
However the number of bins grow exponentially with the dimension of the space, so kd-tree does not work well for highly dimensional space.

$R^1 \rightarrow$ 2 adjacent bins



$R^2 \rightarrow$ 4 adjacent bins



$R^3 \rightarrow$ 6 adjacent bins

$R^4 \rightarrow$ 8 adjacent bins

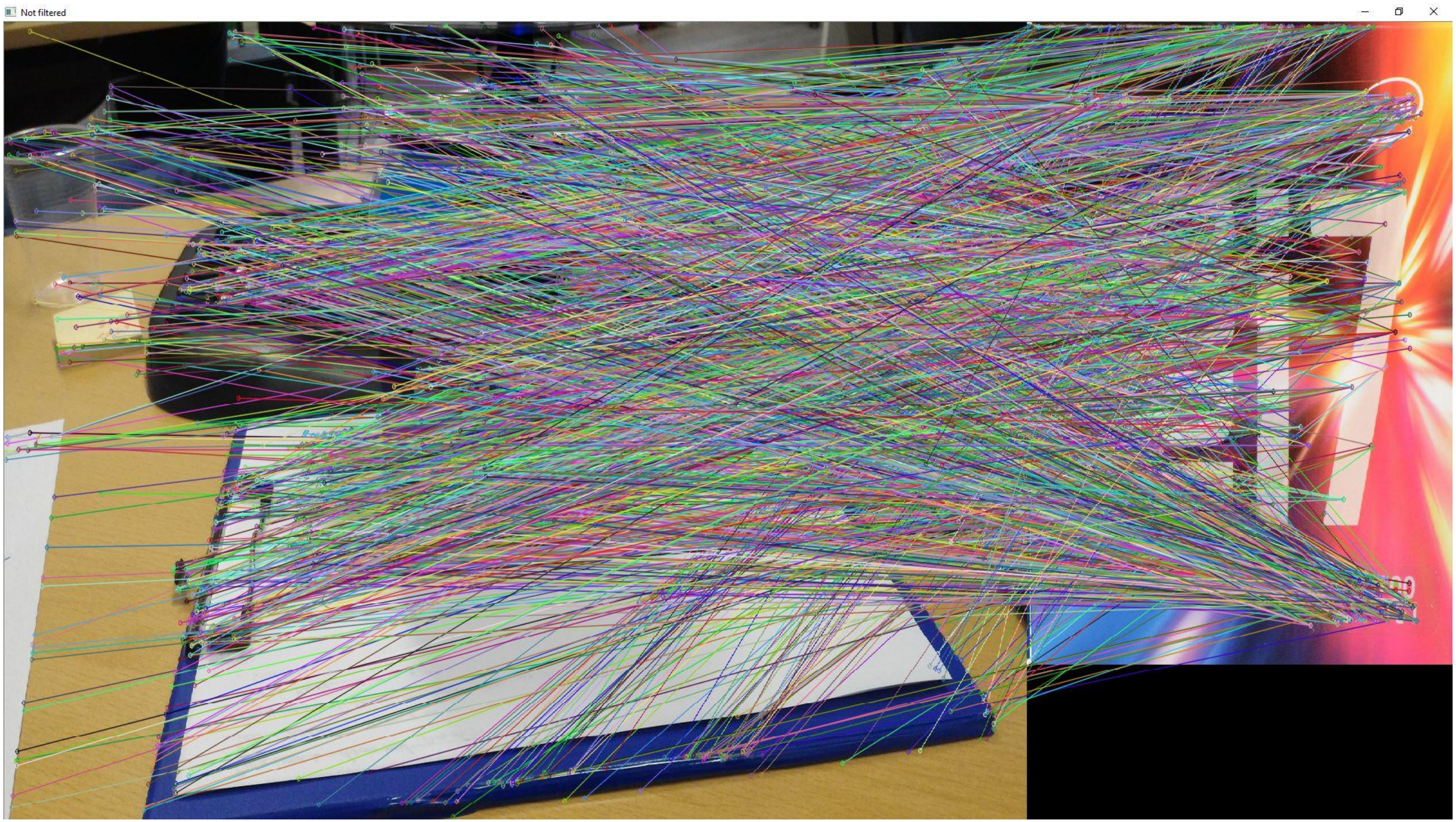
$\vdots$

$R^n \rightarrow$ $2n$ adjacent bins

Features space are highly dimensional!

Approximate technique like [9] help speeding up the search.

3 – Feature Matching



3 – Feature Matching

Problem:
Not all the couples of matching points (m, s) are correct, how can we remove some of the completely wrong ones?

Naive Way:
Threshold on distance, keep only the couples with *small* enough distance.

$$D(s, m) < \tau$$
$$\tau \epsilon [0, \infty]$$

Better Way:
Threshold on the ratio of the distance between the nearest point and the second nearest point.

$$\frac{D(s, m_1)}{D(s, m_2)} < \tau$$
$$\tau \in [0,1]$$

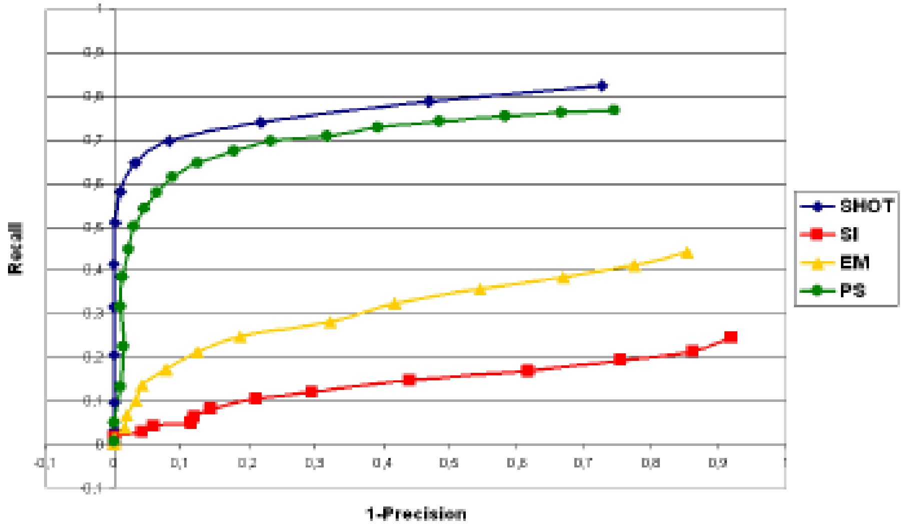
3 – Feature Matching

3 – Feature Matching

To asses the performance of the matching process we can use Precision (1-Precision) – Recall curves.

Given TP number of correct matches, FP number of false matches and P max number of possible matches we can define:

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{P}$$



As long as one tries to gather more matches these become less precise.

4 - Position Estimation

Q: How can we find the position of the object in *scene*? (hint: did you remember camera calibration?)

To find the position we have to compute, given the correspondences, a suitable transformation that brings points from the *model reference system* to the *scene one.*

Homography: transformation that relates any two images of the same planar surface under the pinhole camera model.
An homography is a $3x3$ matrix that transforms points expressed in homogeneous coordinates; it can be decomposed in a *rotation, a translation* and a *perspective distortion.*

4 - Position Estimation

Given corresponding couples of points (m, s) in R^2 with $m \in\ C_m$ and $s \in\ C_s$ estimate an homography means solving a linear system.

$$M = \begin{bmatrix} x_1 & \dots & x_n \\ y_1 & \dots & y_n \\ 1 & \dots & 1 \end{bmatrix} \quad \forall\, m = (x_k, y_k, 1) \in C_m$$

$$S = \begin{bmatrix} x_1 & \dots & x_n \\ y_1 & \dots & y_n \\ 1 & \dots & 1 \end{bmatrix} \quad \forall\, s = (x_k, y_k, 1) \in C_s$$

$$H * M = S$$

Usually those systems are over-constrained problems with no exact solution: solve minimizing error with the least square solution [4].

4 - Position Estimation

Bounding box obtained by transforming the corner of the *model* image in the *scene* image reference system. Homography computed using *least square solution*.

4 - Position Estimation

Problem: some of the match are completely wrong! The estimate homography can be quite bad…

Use *Random Sample Consensus* (RANSAC) [11], an algorithm to fit a parametric model to noisy data.

In our case estimate an homography from good matches while identifying and discarding the wrong ones.

4 - Position Estimation

Ransac main idea:

Given a set of observation $O = \{o_1 \dots o_n\}$ and a certain parametric model M, repeat iteratively:

1. Pick a random (small) subset I of O called *inlier set*.
2. Fit a model M_i according to the observations in I.
3. Test all the other observations against M_i, add to a new set C (*consensus set*) all the observations that fit M_i according to a model specific *loss function*.
4. If the consensus set is bigger than the one associated with the current best model M_b, proceed to step 5, other way return to step 1.
5. Re-compute M_I according to the observations in $I \cup C$, then set $M_b = M_I$. Restart from step 1.

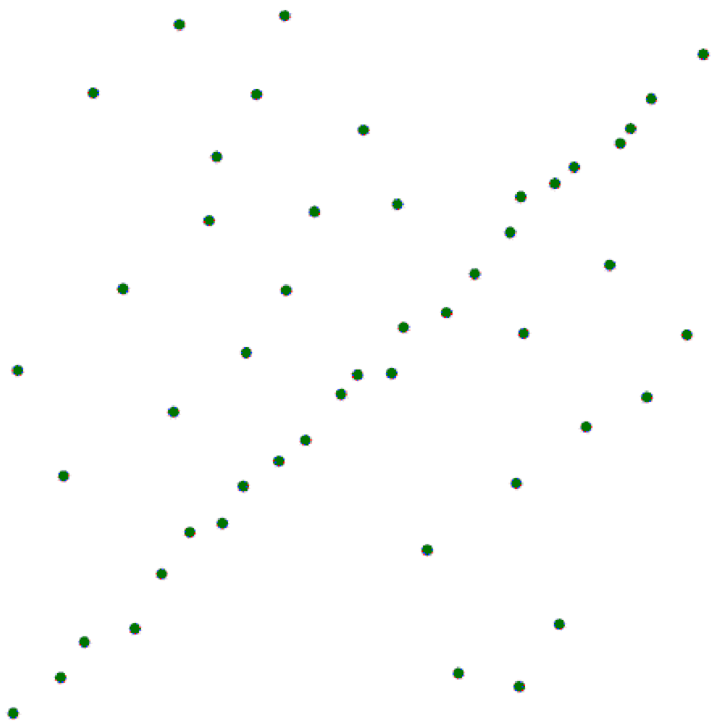
The procedure is repeated for a fixed amount of steps, at the end the best model is returned.

4 - Position Estimation

A simple example of RANSAC used to fit a line to a set of 2D point with *Euclidean distance* as *loss function.*

In blue M_I, in green M_b.

In our case homography as parametric model, and reprojection error as *loss function.*



4 - Position Estimation

Bounding box obtained by transforming the corner of the *model* image in the *scene* image reference system. Homography computed using RANSAC.

OpenCV - Implementation

We are going to use OpenCV to implement an object detection pipeline based on local features!

- **Good news:** All the steps seen so far can be deployed easily using native function in OpenCV. ☺
- **Bad news:** Unfortunately the C interface does not provide methods for feature detection… ☹
  If you want to use feature detection just convert IplImages to cv::Mat and use them accordingly. ☺

```
IplImage * ipl;
cv::Mat m = cv::cvarrToMat(ipl);
```

OpenCV - Implementation

Regarding feature descriptor and detector there are slightly differences between OpenCV version 2.4.x and 3.x. , mainly on how the algorithms are created.

One major diference: OpenCV 3.x has removed SIFT and SURF from the algorithms distributed with the standard auto-installing library. If you want to use them you should build the library from sources adding the *opencv_contrib/xfeatures2d* module.

Some useful links (ask your tutor in case of trouble):

- OpenCV github page
- OpenCV_contrib github page (take a look at the readme)
- Tutorial on how to build opencv from sources

Lab Material – at Home

- Download the zip file from the web page of the Course.

- Extract it and you should have 3 folders:
  - **BIN:** Contains demo files for camera tracking.
  - **Data:** Contains test images.
  - **Src:** Contains the two source files we are going to work with.

- Take a look at "readme.txt" and use CMake-GUI to create a project for your favorite IDE (or just copy the *.cpp* files in *src* in a project already set up to use OpenCV).
  Cmake allows the creation of cross-platform friendly *cpp* program. ☺

Lab Material – at School

- Download the zip file from the web page of the Course.

- We are going to work with project **Lab_session_4,** source file ***Local_Features.cpp***

- Camera track not available due to the lack of webcam ☹

Lab Material

Inside the *src* folder you will find two *.cpp* file:

- **Local_Feature.cpp:** A complete object detection pipeline based on feature matching that reads two images from files (one as *model* and one as *scene*) and try to localize the object.
- **Camera Track.cpp:** Allows to specify the image to load as *model*, then use the live stream from a webcam to perform object detection. Using the slider you can adjust the τ used to filter matches with ratio threshold.

Local_Feature.cpp - includes

Using OpenCV 2.4.x, if you want to use SIFT or SURF remember to include and initialize the proper module using:

```
#include "opencv2/nonfree/nonfree.hpp
…
Int main(int argc,char**argv){
        cv::initModule_nonfree();
        …
}
```

Local_Feature.cpp – Feature2D

Both Detector and Descriptor are implemented in Opencv using the common abstract class *cv::Feature2D*. The syntax used to **create** a detector/descriptor change slightly from OpenCV 2.4.x and 3.x

- **OpenCV 2.4.x:**

  ```
  cv::Ptr<feature_type> cv::Feature2D::create(const std::string feature_type);
  ```

  e.g.24: `cv::Ptr<cv::Feature2D> detector=cv::Feature2D::create("BRISK");`

  Line 24 in Local_Feature.cpp

- **OpenCV 3.x:**

  Each algorithm has its own static *create* method.

  e.g.27: `cv::Ptr<cv::Feature2D> detector = cv::BRISK::create();`

Local_Feature.cpp – Feature2D

The object just created can be used both for detection and description.

In this small demo we are going to use one of this four descriptor with a companion detector:

- ORB [7]
- BRISK [6]
- SIFT [4]
- SURF [5]

Many other detectors or descriptors available [in opencv](#).

Local_Feature.cpp – KP Detection

Keypoint are represented in Opencv with the [cv::Keypoint]struct.

This structure has some useful fields:

- *pt* → point location in pixel coordinates.
- *size* → diameter of the meaningful keypoint neighborhood.
- *angle* → estimated orientation of the keypoint.
- …

Local_Feature.cpp – KP Detection

For the detection, first create a *std::vector* (similar to C arrays, but with no fixed size) to store the detected keypoints:

```
std::vector<cv::KeyPoint> keypoints;
```

Then use the [*detect*]() method of *cv::Feature2D* to find keypoints on an image.

```
detect(cv::Mat& image, std::vector<cv::KeyPoint>& keypoints)
```

e.g.`35: detector->detect(model, keypoints_model);`

Demo Time!

Local_Feature.cpp – Description

Descriptors are represented in OpenCV as simple arrays of values. For convenience they are stored in 2D matrices, one descriptor for each row with the row number corresponding to the id of the associated keypoint.

To compute the descriptors given a *std::vector<cv::Keypoint>* use the compute method of *cv::Feature2D*.

```
compute(cv::Mat& image, std::vector<cv::KeyPoint>& keypoints,
cv::Mat& descriptors)
```

```
e.g. 57: descriptor->compute(model, keypoints_model,
descriptor_model);
```

Keypoint at position 0 in keypoint_model has its descriptor at row 0 of descriptor model.

Demo Time!

Local_Feature.cpp – Matching

Create a suitable descriptor matcher according to the type of descriptor used.

Descriptor matchers in OpenCV are all implementation of the abstract class *cv::DescriptorMatcher*, depending on the type of descriptor we have to use a different matcher:

- **Binary descs** *(ORB,BRISK)*: LSH [10]
  e.g. 67: `cv::Ptr<cv::DescriptorMatcher> matcher = cv::makePtr<cv::FlannBasedMatcher>(new cv::flann::LshIndexParams(10, 20, 2));`

- **Float descs** (SIFT,SURF): BBF[9]
  e.g. 65: `cv::Ptr<cv::DescriptorMatcher> matcher = cv::DescriptorMatcher::create("FlannBased");`

Local_Feature.cpp – Matching

Use the method *add* to populate the matcher with the descriptors computed on the models.

```
add(const std::vector<cv::Mat>& descriptors);
```

e.g.73: `matcher->add(models_descriptors);`

`descriptors` contains one matrix for each model to detect.

To get the actual couples of matching keypoints:

```
knnMatch(cv::Mat& scene_desc,
std::vector<std::vector<cv::Dmatch>>& matches, int k)
```

`k` being the number of NN to find for each query point.

e.g.77: `matcher->knnMatch(descriptor_scene, matches, 2);`

ratio test!

Local_Feature.cpp – Matching

Matches are represented in OpenCV with the *cv::DMatch* structure, it encodes in its fields all the useful information about the couple of keypoints:

- ***queryIdx***: index of the keypoint in the *scene* image.
- ***trainIdx***: index of the keypoint in the *model* image.
- ***imgIdx***: index of the *model* in the vector passed to the *add* method
- ***distance***: distance between the descriptors.

Demo Time!

Local_Feature.cpp – Pose

Finally compute the homography that transforms points from the *model* space to the *scene* one using [*findHomograhy*].

```
Cv::Mat cv::findHomography(std::vector<cv::Point> srcPoint,
std::vector<cv::Point> dstPoint, int method)
```

`method` can be one of: 0 (all couples used), CV_RANSAC (ransac), CV_LMEDS (least square error).

```
e.g.119:cv::Mat homography=cv::findHomography(model_points,
scene_points, CV_RANSAC);
```

Local_Feature.cpp – Pose

It is now possible to use the homography matrix to transform points from the *model* reference system to the *scene* one, useful for example to draw the object bounding box. We can use [*cv::perspectiveTransform*]

```
cv::perspectiveTransform(std::vector<cv::Point>& src,
std:vector<cv::Point>& dest, cv::Mat& transformMat)
```

This method takes the points in `src`, transform them according to `transformMat` and save the result in `dest`.

It is also possible to transform a whole image using [*cv::warpPerspective*]

Local_Features.cpp - exercise

- Take a look at Local_Feature.cpp
- Modify it to localize two different objects at the same time:
  1. Load two model images.
  2. Compute Keypoints in both.
  3. Compute descriptors in both.
  4. Compute matches between the *scene* descriptors and the *models*.
  5. Compute the two homographies.
  6. Display two bounding boxes.

Bibliography

1. Ren, Shaoqing, et al. "Faster R-CNN: Towards real-time object detection with region proposal networks." *Advances in neural information processing systems*. 2015.
2. Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." *arXiv preprint arXiv:1506.02640* (2015).
3. Liu, Wei, et al. "SSD: Single Shot MultiBox Detector." *arXiv preprint arXiv:1512.02325* (2015).
4. Lowe, David G. "Distinctive image features from scale-invariant keypoints." *International journal of computer vision* 60.2 (2004): 91-110.
5. Bay, Herbert, Tinne Tuytelaars, and Luc Van Gool. "Surf: Speeded up robust features." *European conference on computer vision*. Springer Berlin Heidelberg, 2006.

Bibliography

6. Leutenegger, Stefan, Margarita Chli, and Roland Y. Siegwart. "BRISK: Binary robust invariant scalable keypoints." *2011 International conference on computer vision*. IEEE, 2011.
7. Rublee, Ethan, et al. "ORB: An efficient alternative to SIFT or SURF." *2011 International conference on computer vision*. IEEE, 2011.
8. Friedman, Jerome H., Jon Louis Bentley, and Raphael Ari Finkel. "An algorithm for finding best matches in logarithmic expected time." *ACM Transactions on Mathematical Software (TOMS)* 3.3 (1977): 209-226.
9. Muja, Marius, and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration." *VISAPP (1)* 2.331-340 (2009): 2.

Bibliography

10. Indyk, Piotr, and Rajeev Motwani. "Approximate nearest neighbors: towards removing the curse of dimensionality." *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 1998.
11. Fischler, Martin A., and Robert C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography." *Communications of the ACM* 24.6 (1981): 381-395.