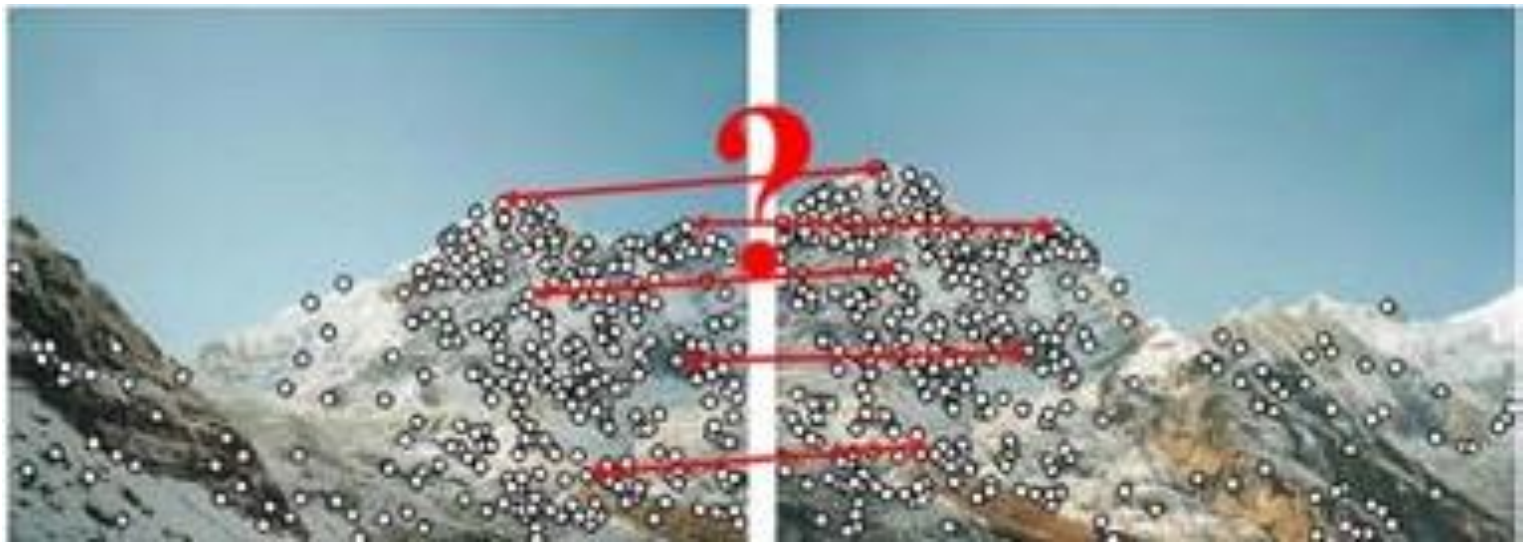
Feature point description and matching

IN4393 – Computer Vision

Introduction

- Last week, we learned how to detect stable, invariant feature points
- How can we use these feature points to match objects in images?



Feature point descriptors

Rotation-invariance

- Like keypoint detectors, descriptors should be scale- and rotation-invariant:
  - However, simple rotation-invariant descriptors have poor discriminability

Rotation-invariance

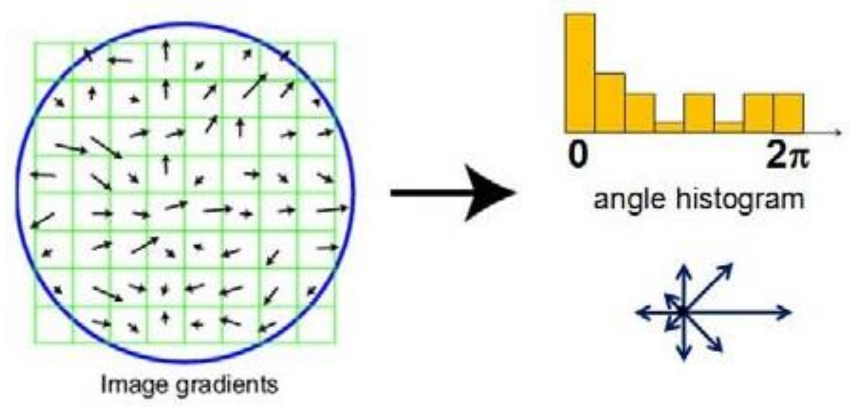
- Like keypoint detectors, descriptors should be scale- and rotation-invariant:
  - However, simple rotation-invariant descriptors have poor discriminability
- A better approach is to estimate the *dominant orientation* at a keypoint:
  - Simple approach estimates dominant orientation from the Gaussian-weighted horizontal and vertical gradients:

$$\alpha(\mathbf{x}) = \mathrm{atan2}\left(\sum_{\mathbf{x}_i \in \mathcal{N}_\mathbf{x}} w(\mathbf{x}_i) I_y(\mathbf{x}_i), \sum_{\mathbf{x}_i \in \mathcal{N}_\mathbf{x}} w(\mathbf{x}_i) I_x(\mathbf{x}_i)\right)$$

  - Could also look at the principal eigenvector of the second-order matrix

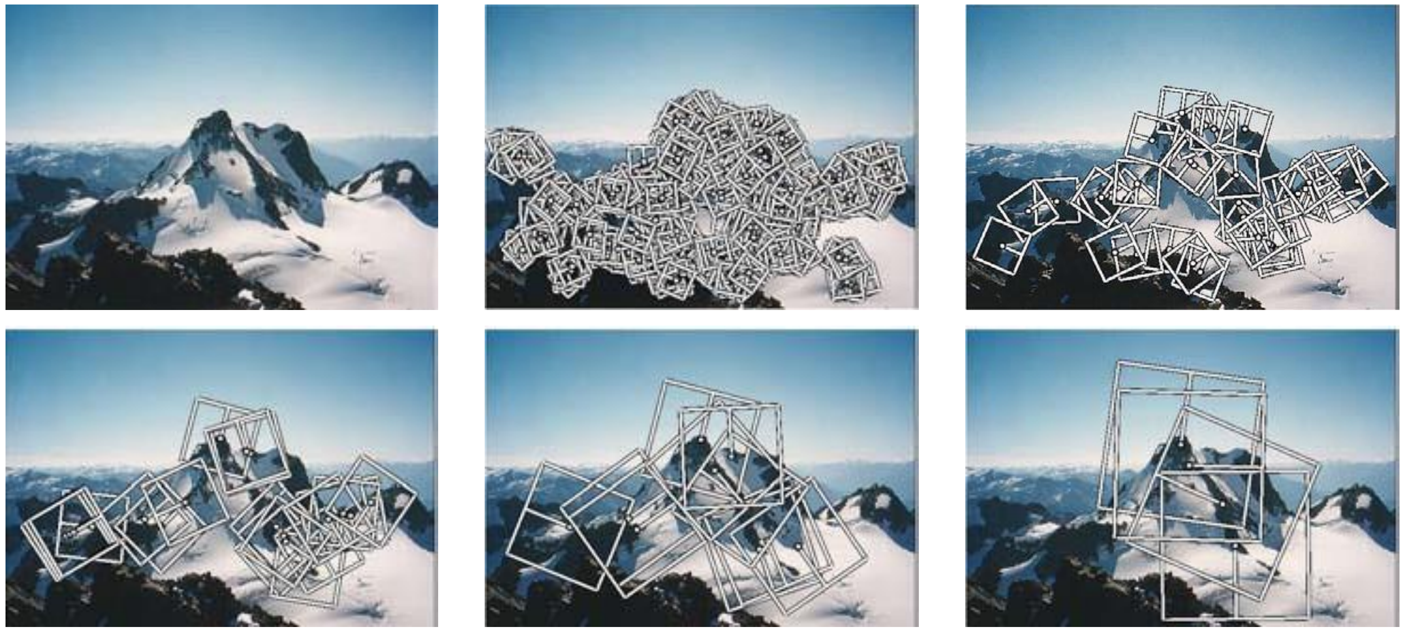
Rotation-invariance

- When the gradients are small, *orientation histograms* are more reliable:



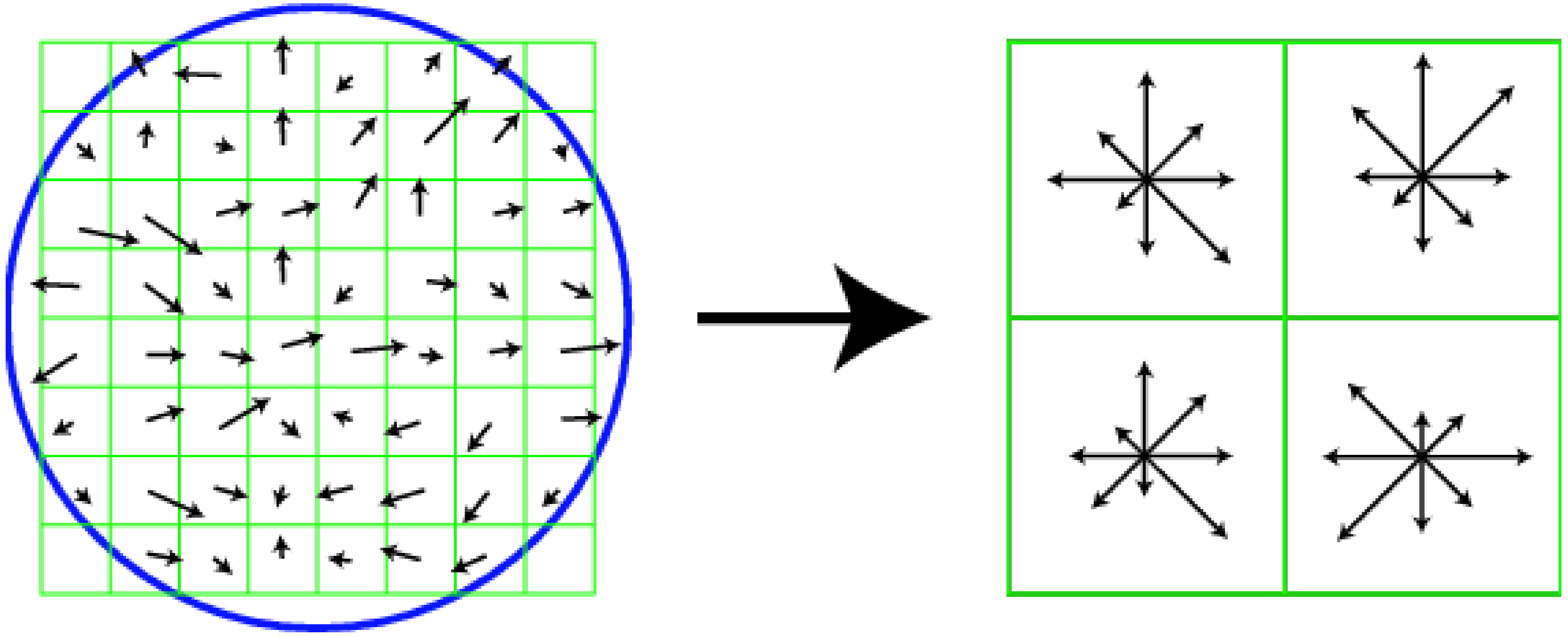
* SIFT uses 36 orientation bins here

- Illustration of dominant-orientation estimation at multiple scales:



SIFT Descriptor

- The SIFT descriptor measures orientation histograms in small blocks:


(a) image gradients (b) keypoint descriptor

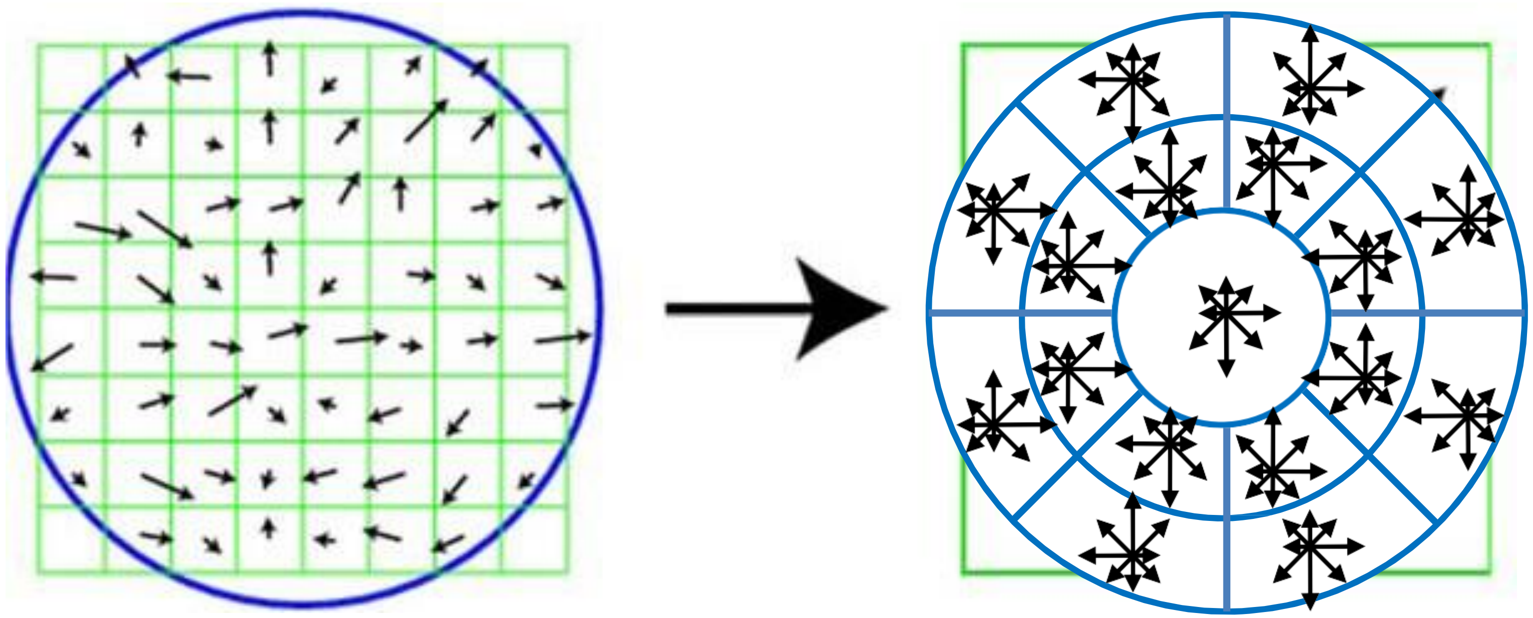
- Image gradients taken *at the right scale*, *relative to the dominant orientation*

SIFT Descriptor

- Details of the standard SIFT descriptor:
  - Uses a 16x16 pixel window, with 4x4 pixel quadrants and 8 orientation bins
  - Soft addition of gradient magnitudes to histogram bins using trilinear interpolation
  - The 128D descriptor is normalized to unit-length to reduce contrast and gain effects
  - Values are clipped to 0.2 and descriptor is renormalized to deal with high contrast
  - Sometimes PCA is applied to reduce dimensionality of SIFT descriptors (PCA-SIFT)

GLOH descriptor

- Variant of SIFT that uses *log-polar* binning structure:



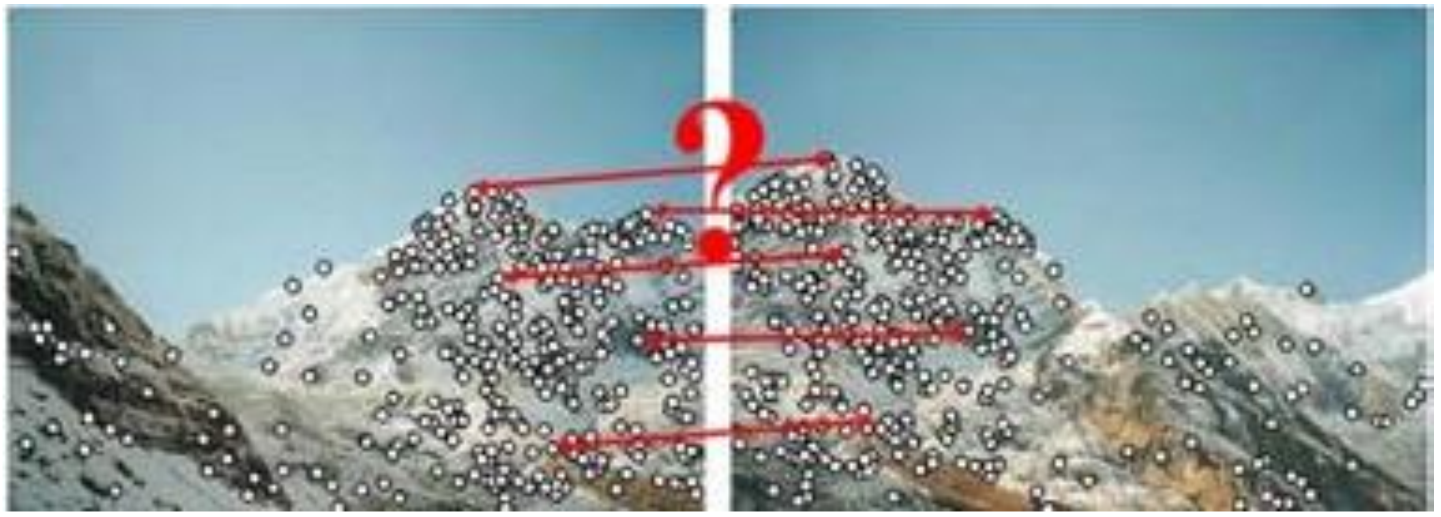
(a) image gradients (b) keypoint descriptor

- Intuition behind GLOH (**Gradient Location and Orientation Histogram**) is that you should not split up the central image part around the feature point

Feature point matching

Feature point matching

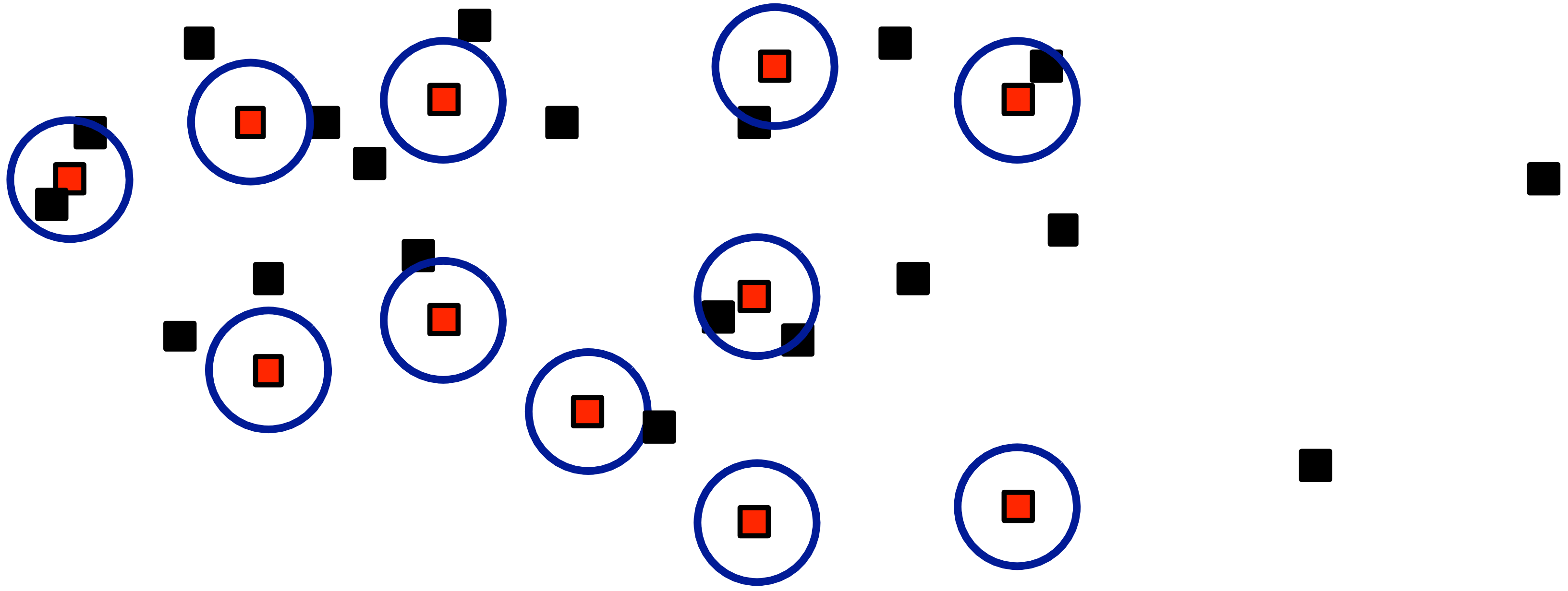
- The goal is to find corresponding points in two images for further processing:



- Key idea behind feature point matching:
  - Corresponding points have similar feature point descriptors

Matching feature points

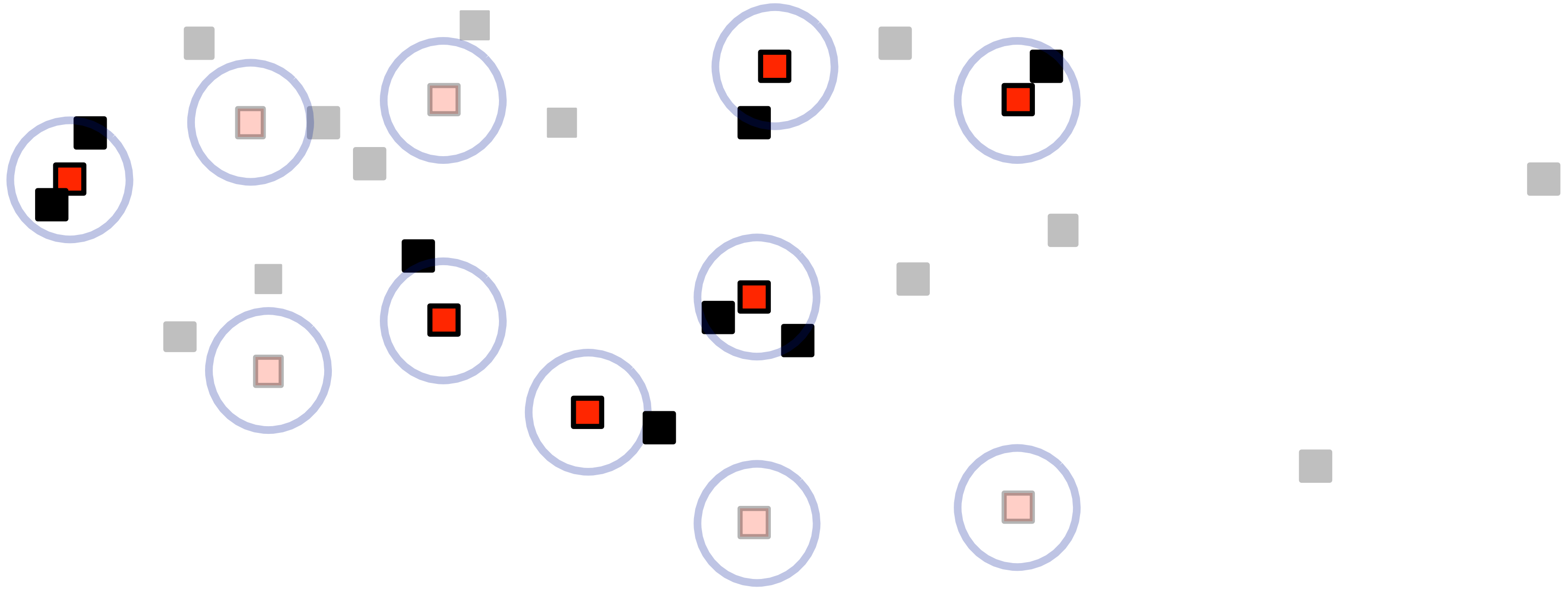
- Similar points have similar descriptors, *i.e.* are nearby in *“descriptor space”*:



- Simple approach to finding matches is to threshold Euclidean distances
  - It may be necessary to *whiten* the descriptors first: normalize feature “scales”

Matching feature points

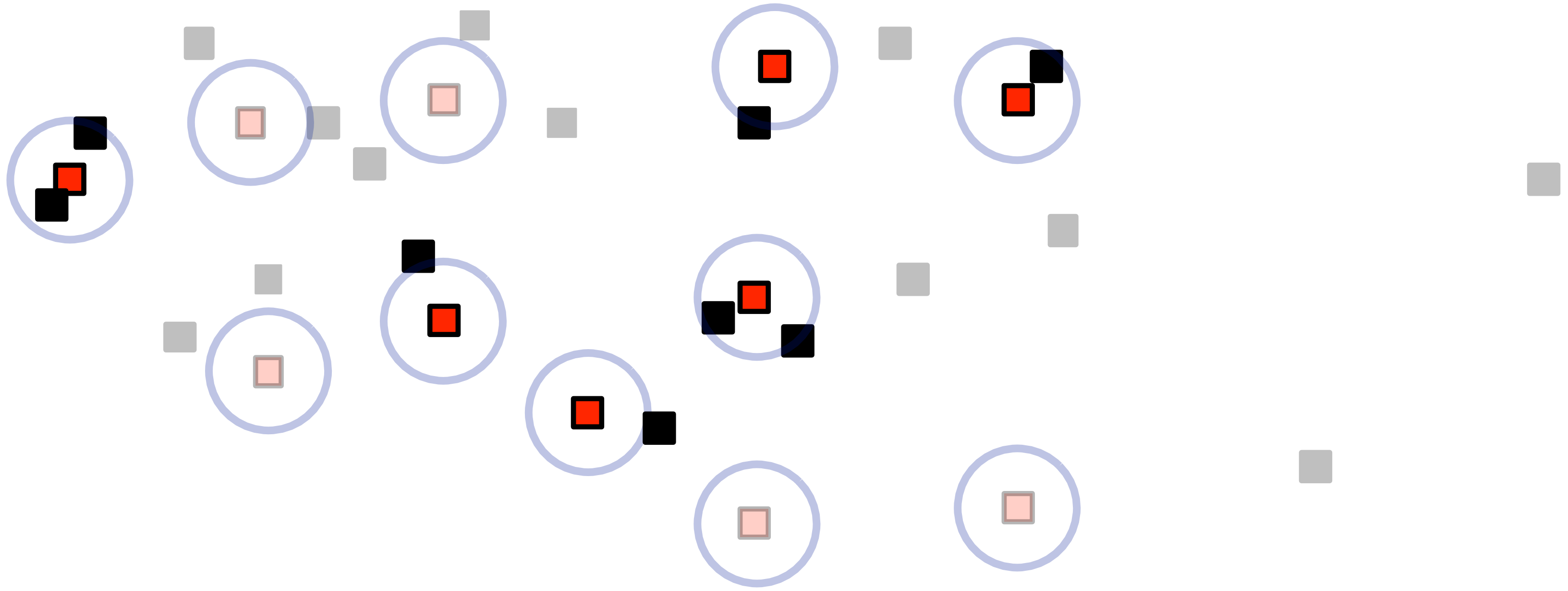
- Similar points have similar descriptors, *i.e.* are nearby in *“descriptor space”*:



- Simple approach to finding matches is to threshold Euclidean distances
    - It may be necessary to *whiten* the descriptors first: normalize feature “scales”

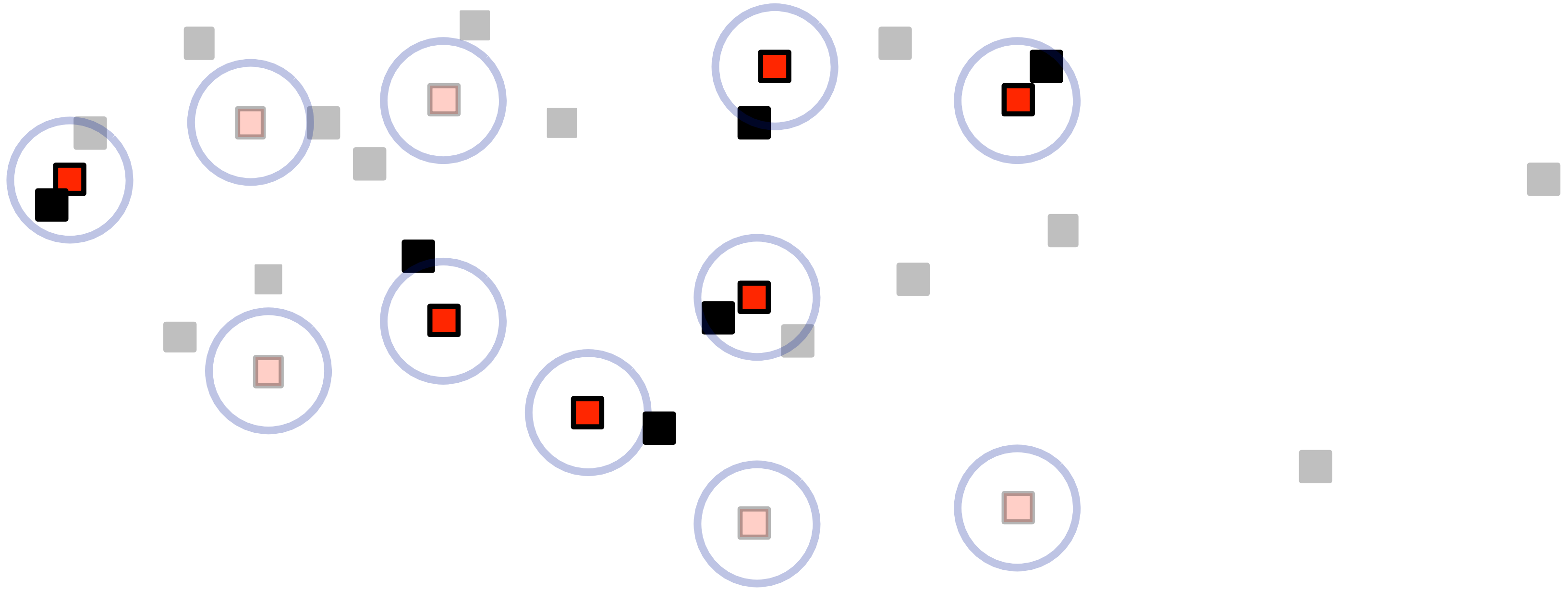
Matching feature points

- Setting threshold on Euclidean distance is hard; optimal value may vary a lot
- Moreover, a single feature point may get many potential matches
- A better approach is to restrict matches to *nearest neighbors* only:



Matching feature points

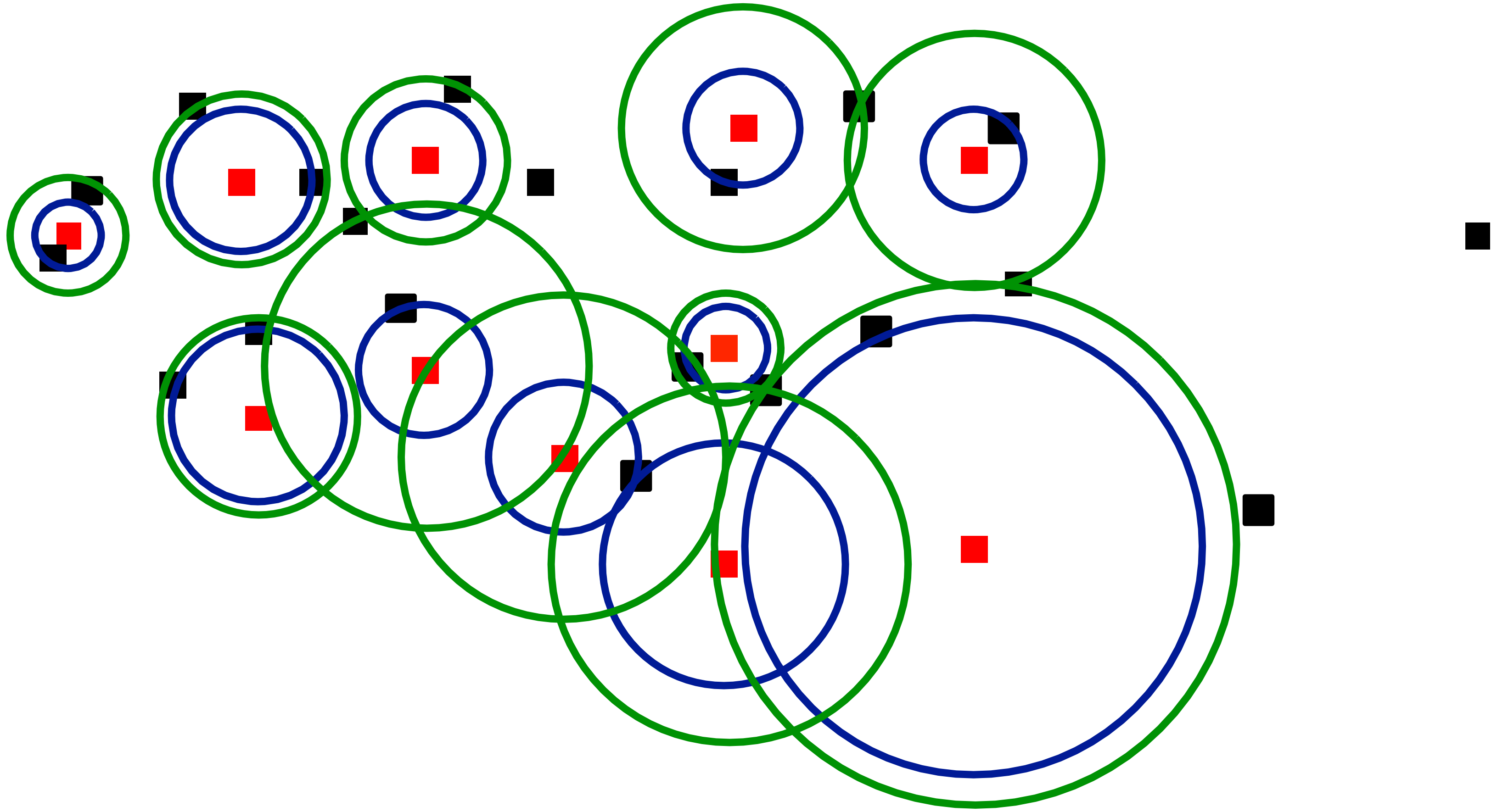
- Setting threshold on Euclidean distance is hard; optimal value may vary a lot
- Moreover, a single feature point may get many potential matches
- A better approach is to restrict matches to *nearest neighbors* only:



Matching feature points

- *Nearest neighbor distance ratio*: $NNDR = \dfrac{d(target, nearest\ neighbor\ 1)}{d(target, nearest\ neighbor\ 2)}$

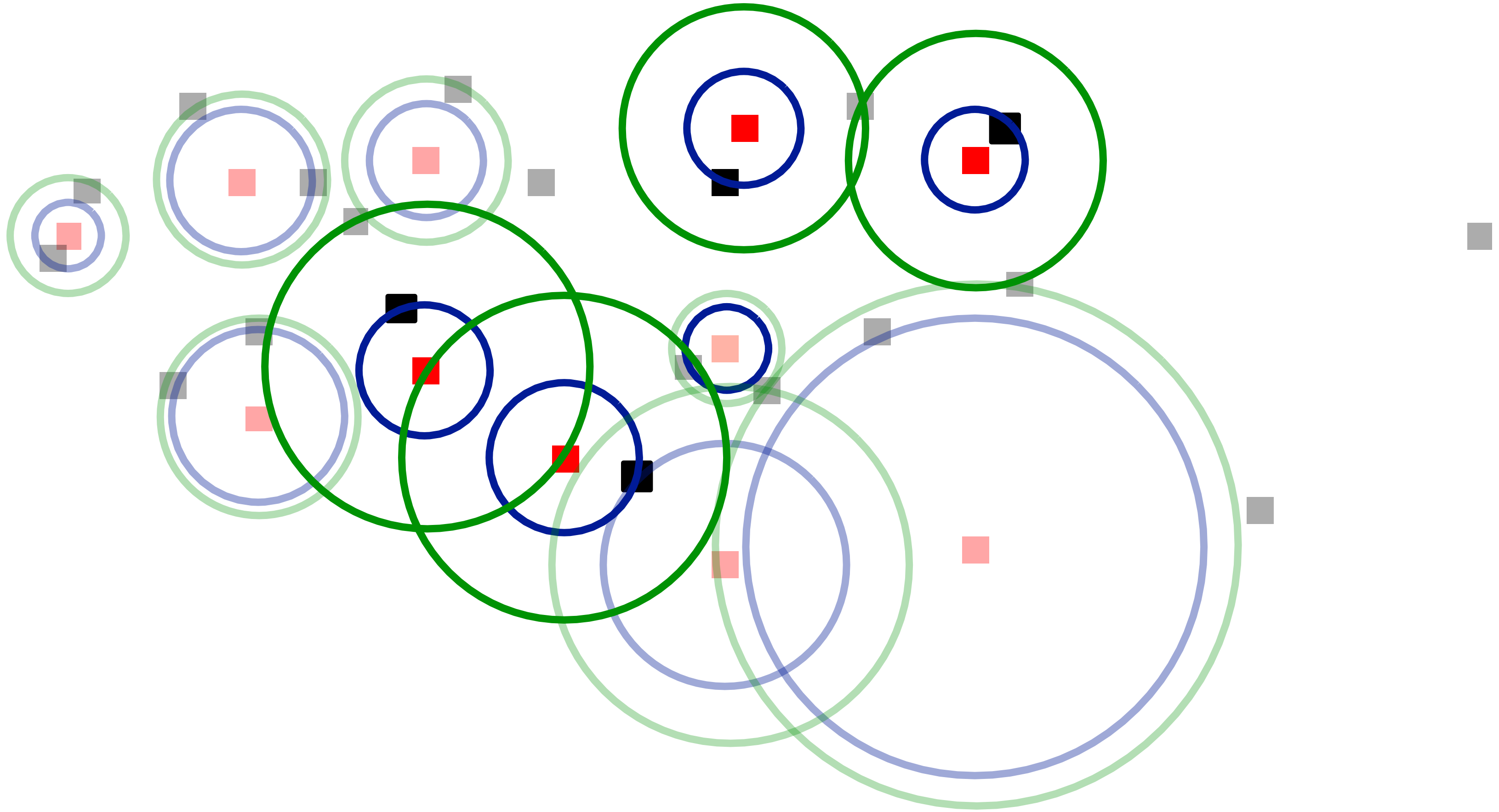
- Compares the distance to the first neighbor with that to the second neighbor:



Matching feature points

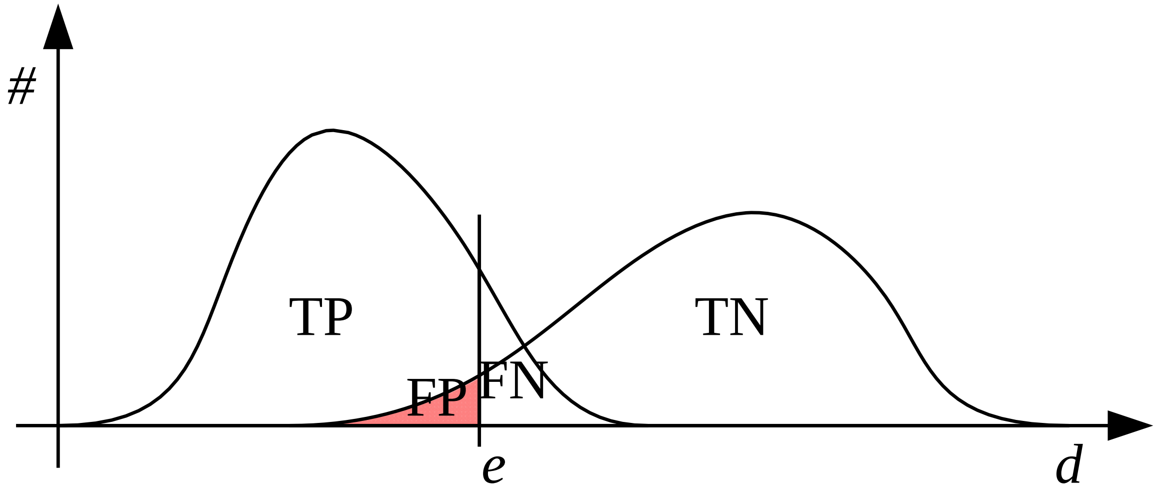
- *Nearest neighbor distance ratio*: $NNDR = \frac{d(target, nearest\ neighbor\ 1)}{d(target, nearest\ neighbor\ 2)}$

- Compares the distance to the first neighbor with that to the second neighbor:



Evaluating quality of matching

- *True positives* (TP): number of matches that were correctly detected
- *False positives* (FP): number of non-matches that were erroneously detected
- *True negatives* (TN): number of non-matches that were correctly rejected
- *False negatives* (FN): number of matches that were not detected



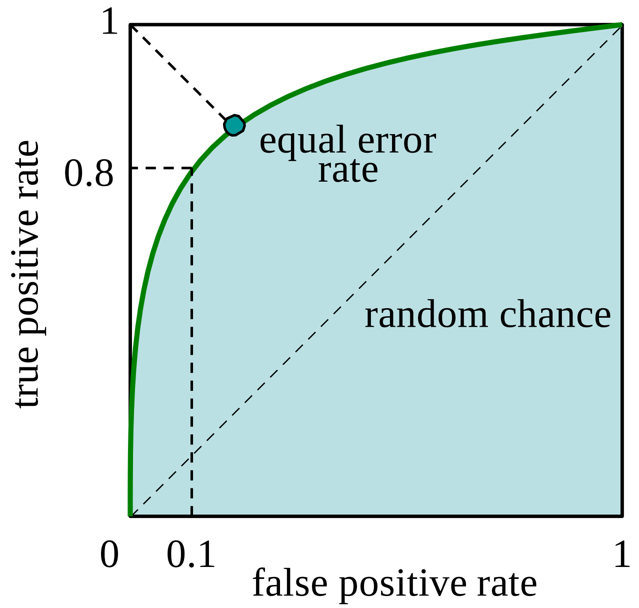
Evaluating quality of matching

- *True positive rate*: percentage of true matches that is indeed proposed

- *False positive rate*: percentage of non-matches that is erroneously proposed

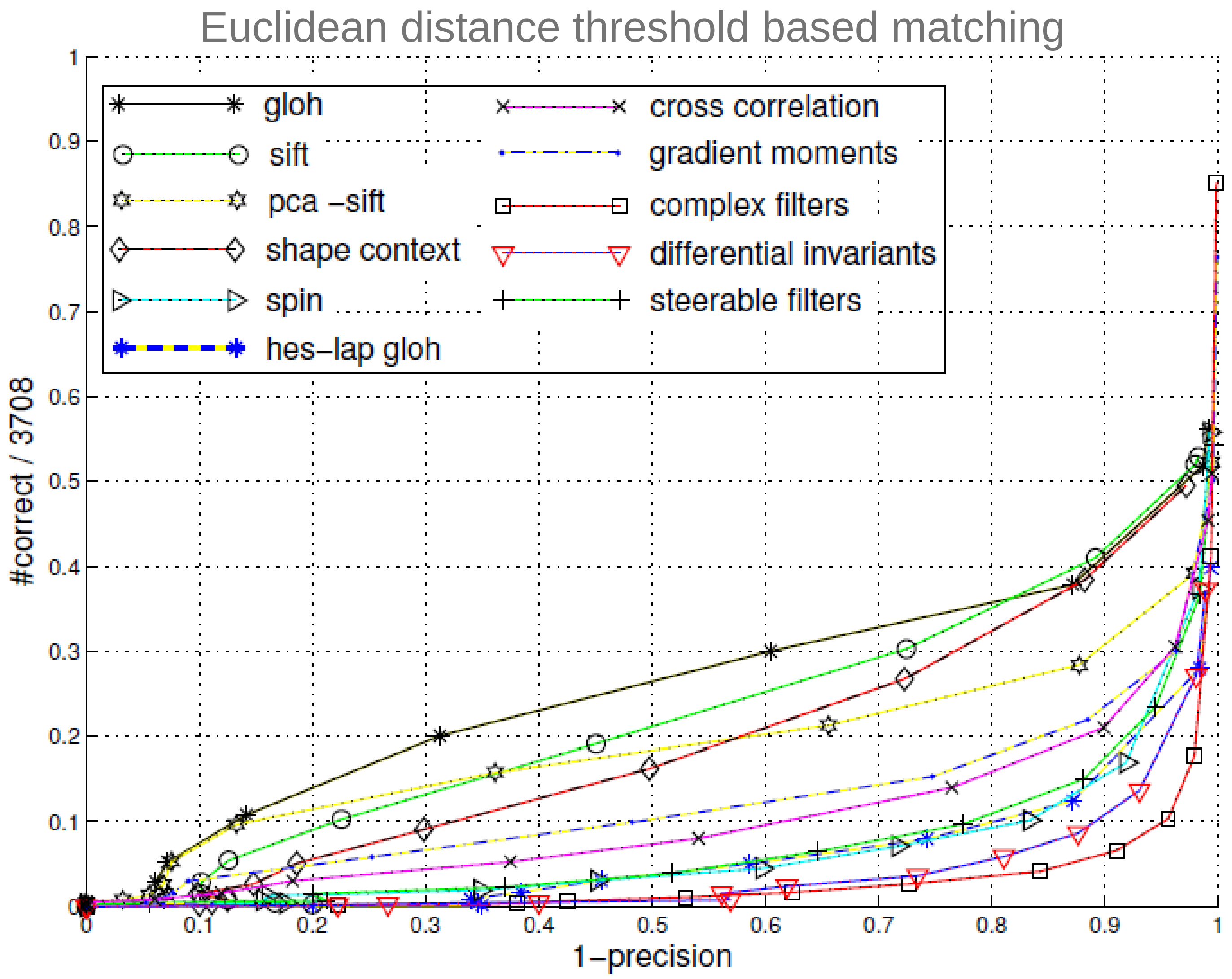
$$TPR = \frac{TP}{TP + FN} = \frac{TP}{P} \qquad FPR = \frac{FP}{FP + TN} = \frac{FP}{N}$$

- *Receiver-operating characteristic* (ROC) curve relates TPR and FPR:



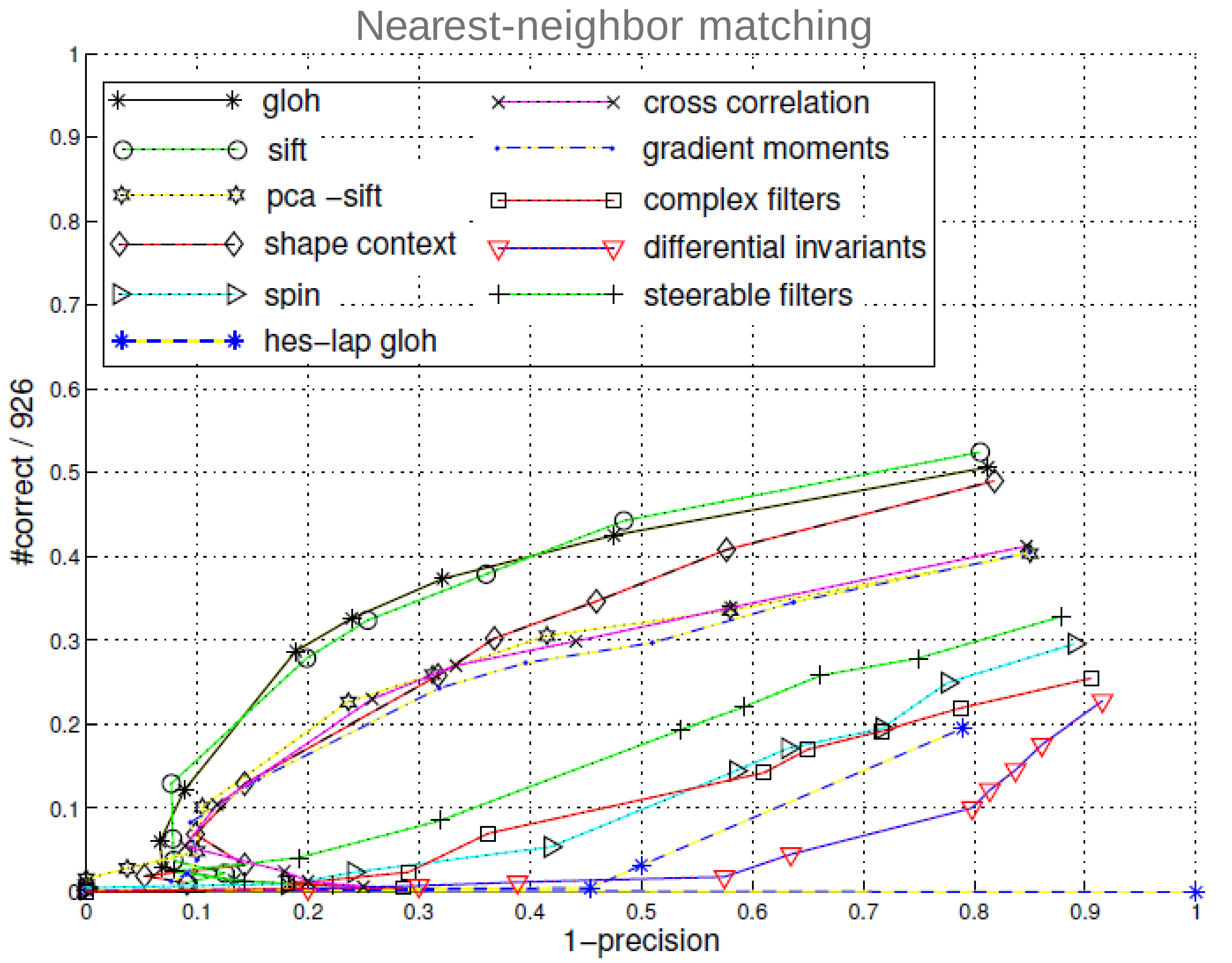
Comparing matching approaches

- Euclidean vs. nearest-neighbor vs. NNDR for various descriptors:



Comparing matching approaches

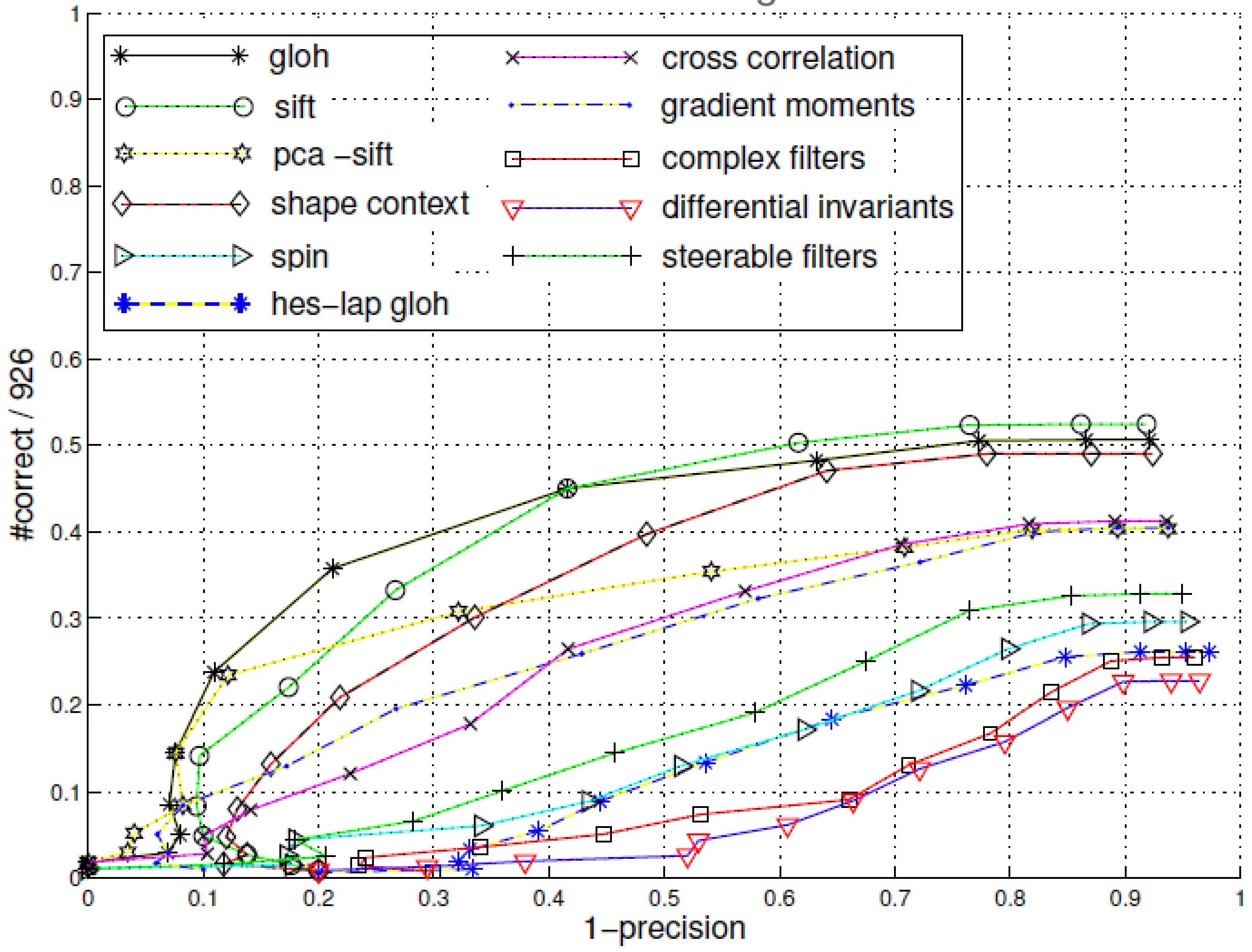
- Euclidean vs. nearest-neighbor vs. NNDR for various descriptors:



Comparing matching approaches

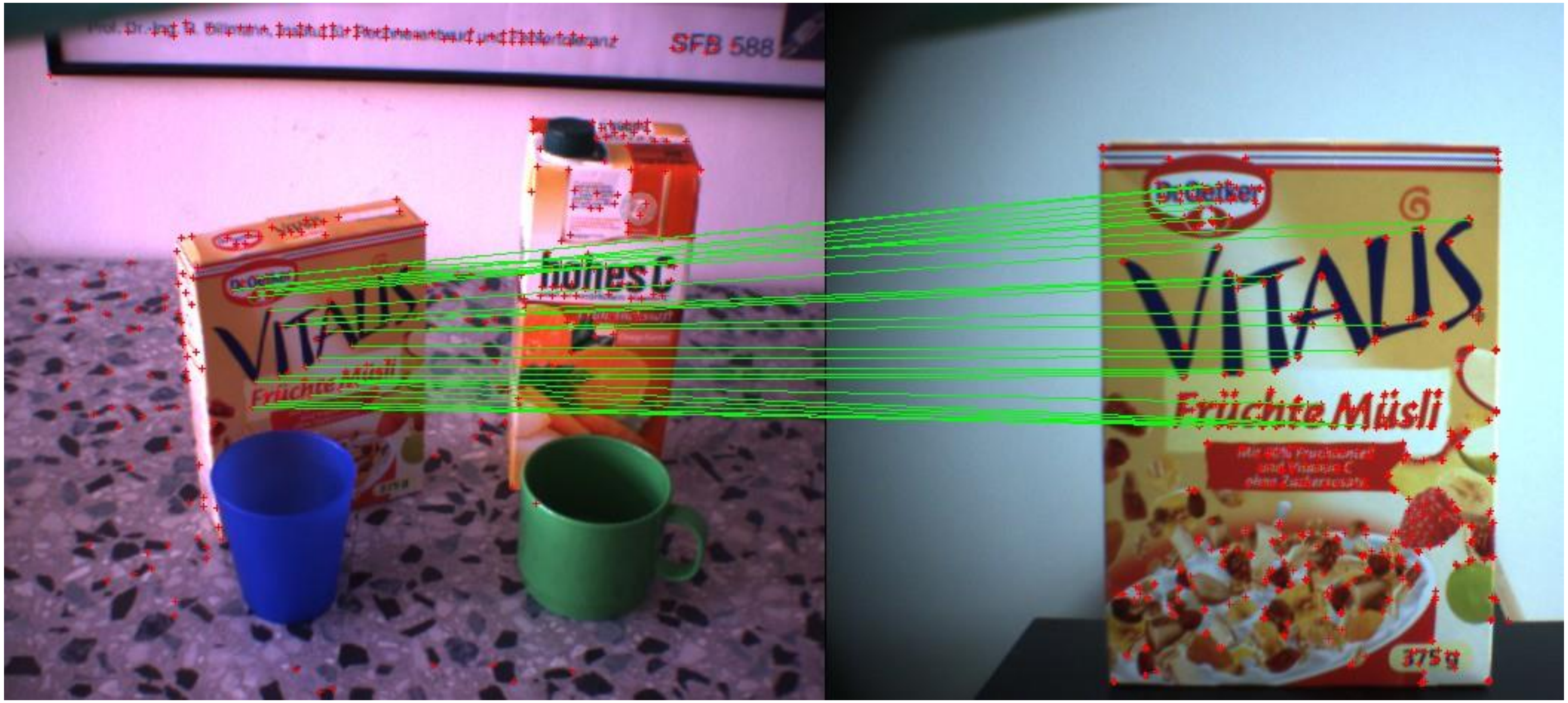
- Euclidean vs. nearest-neighbor vs. NNDR for various descriptors:

NNDR matching



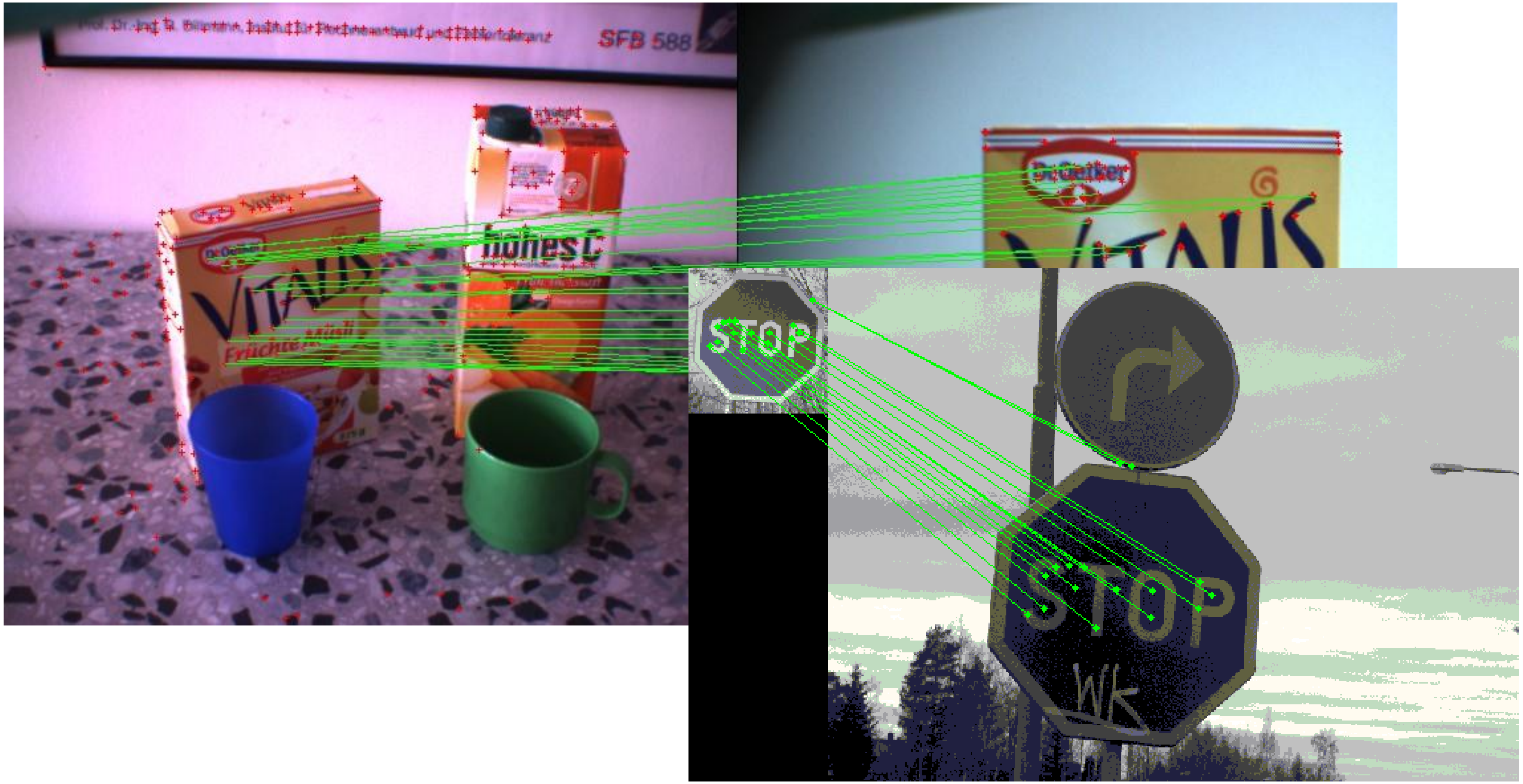
Example: Object recognition

- Given a database of labeled objects, you could do *object recognition*:



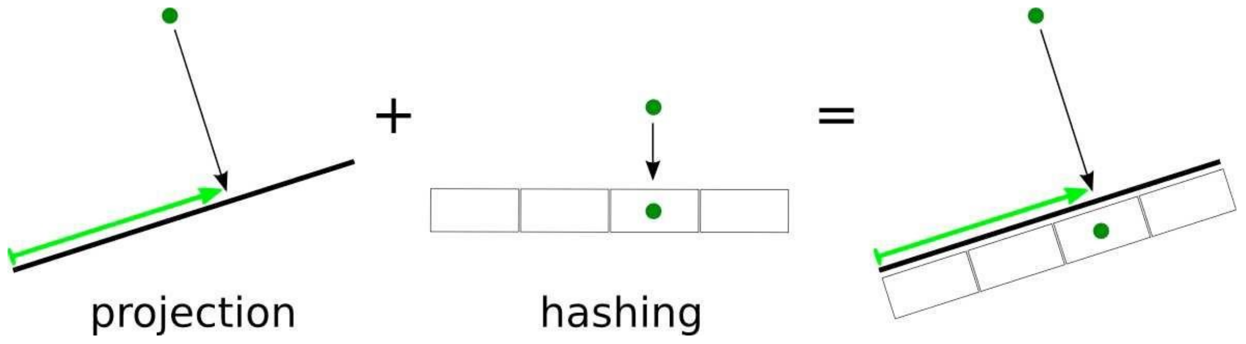
Example: Object recognition

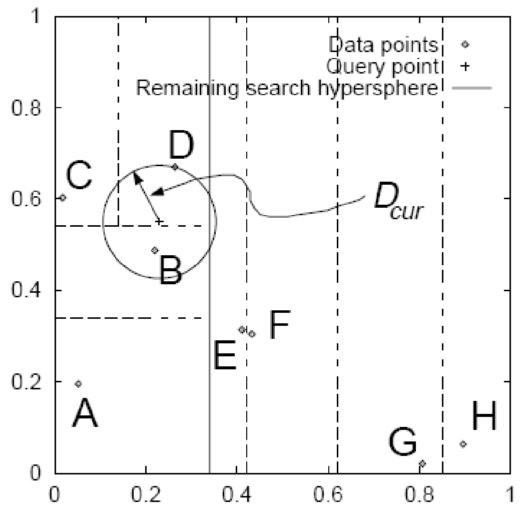
- Given a database of labeled objects, you could do *object recognition*:

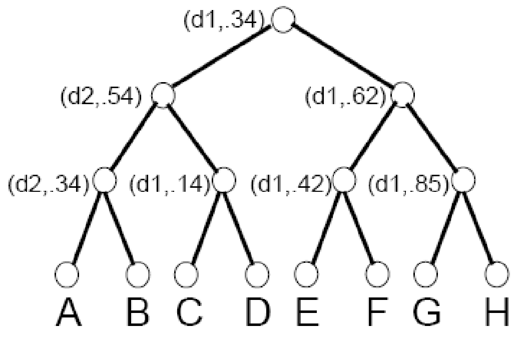


Efficient matching

- Naively matching two sets of feature points is $\mathcal{O}(NM)$
- *Locality sensitive hashing* or *kd-trees* may speed up nearest neighbor search

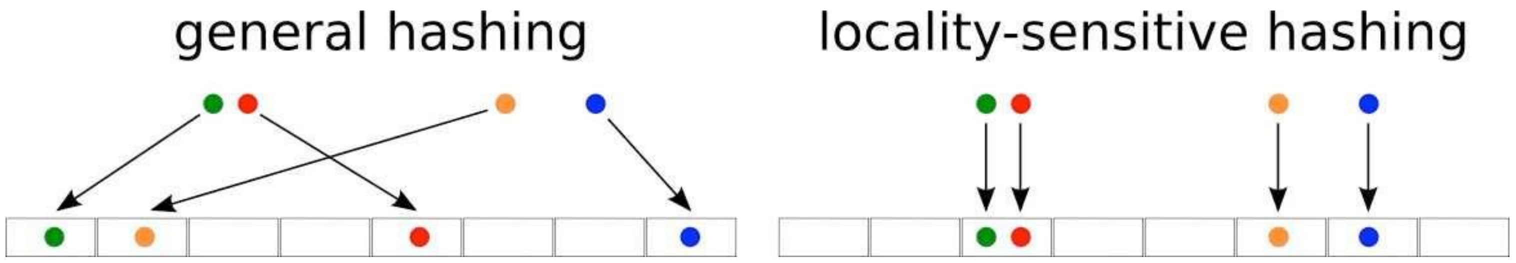






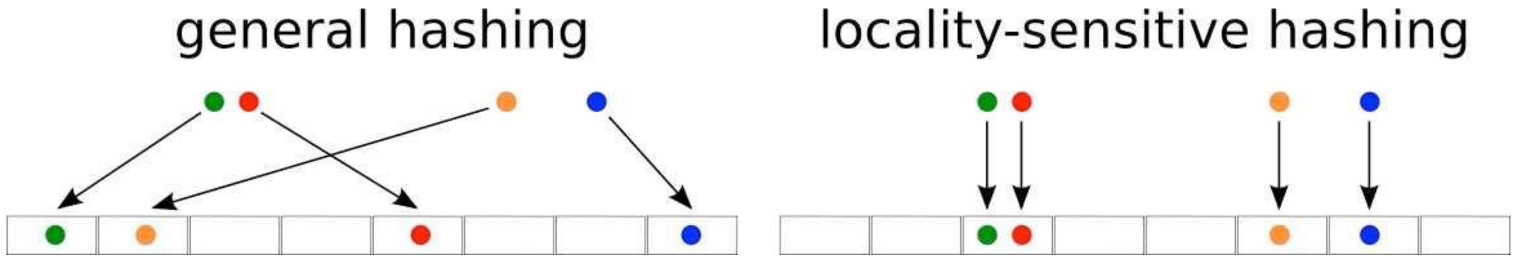
Locality-sensitive hashing

- LSH uses hashing functions that take “location” of object in consideration:

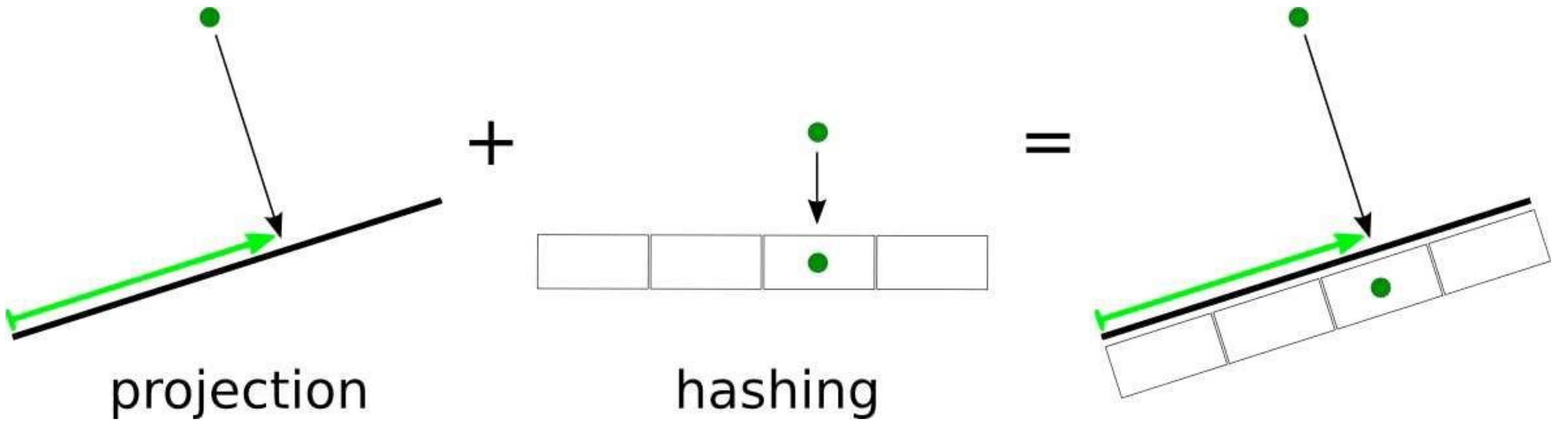


Locality-sensitive hashing

- LSH uses hashing functions that take “location” of object in consideration:



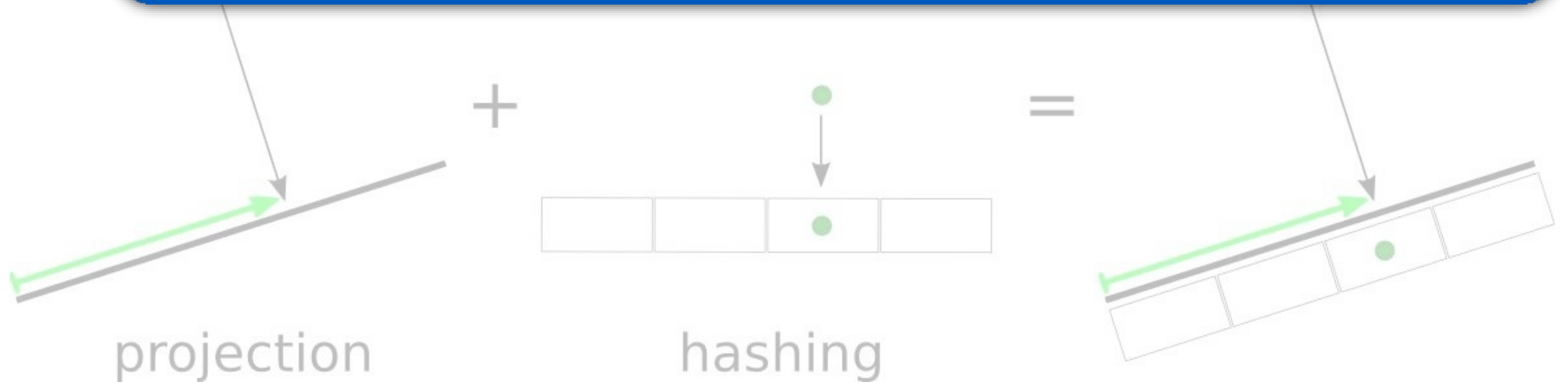
- Example of a locality-sensitive hashing function for points in a space:
  - Project the point onto a random subspace; divide result into 4 buckets (2 bits)



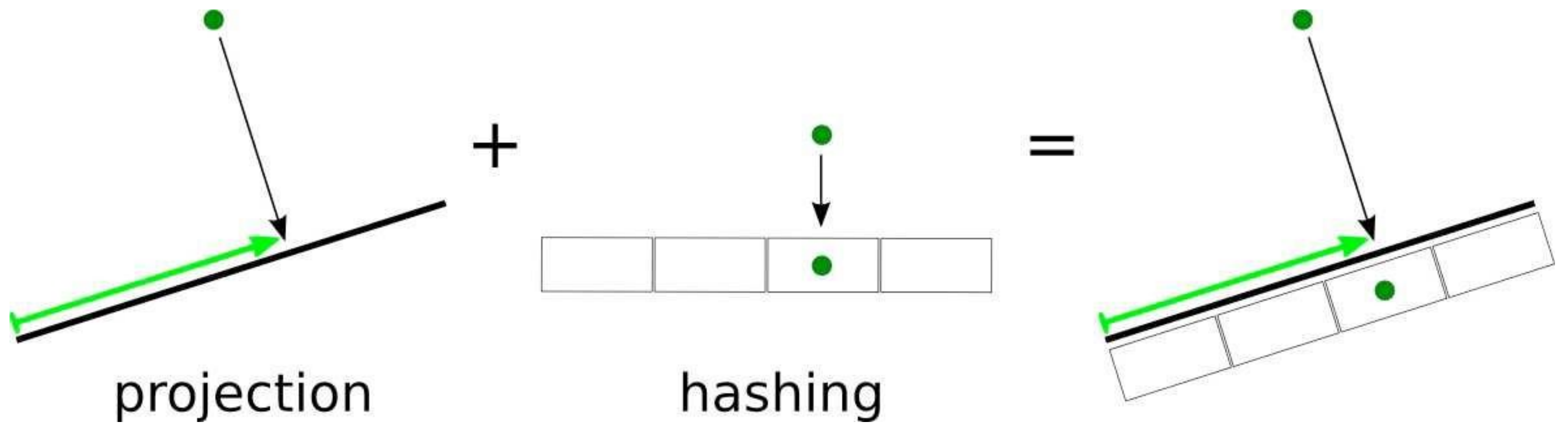
Locality-sensitive hashing

- LSH uses hashing functions that take “location” of object in consideration:
- Simple ex
  - Project (2 bits)

Whilst normal hash functions try to **minimize** the probability of collision, LSH hash functions try to **maximize** probability of similar items colliding.



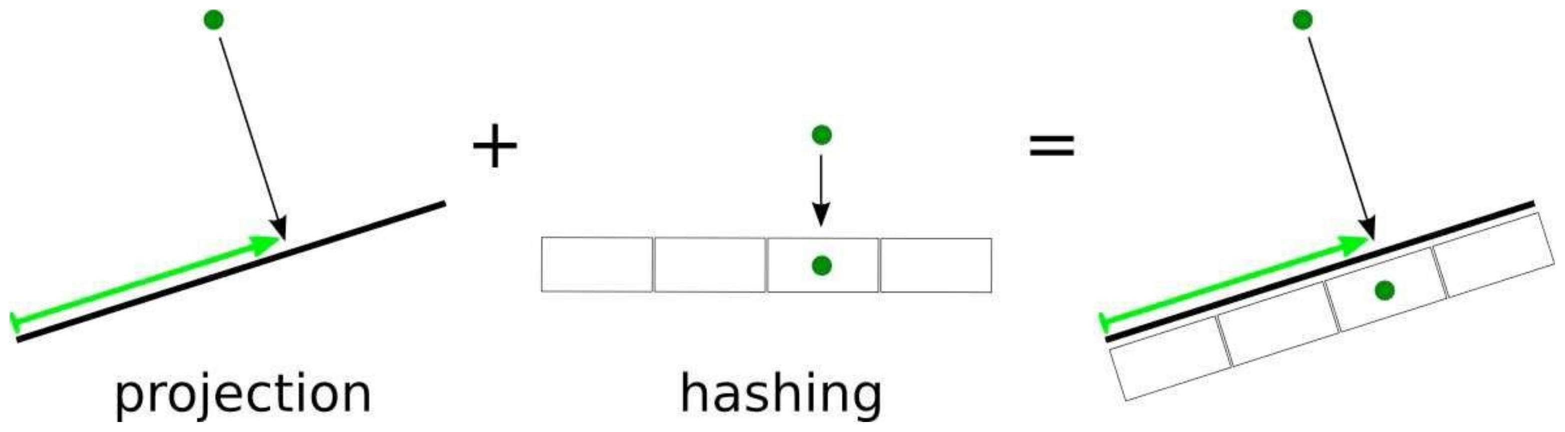
Locality-sensitive hashing



- Mathematically, we could express this locality sensitive hash function as:

$$h(\mathbf{x}) = \begin{cases} 0 & \text{if} \quad \mathbf{w}^\top \mathbf{x} \leq -\tau \\ 1 & \text{if} \quad -\tau < \mathbf{w}^\top \mathbf{x} \leq 0 \\ 2 & \text{if} \quad 0 < \mathbf{w}^\top \mathbf{x} \leq \tau \\ 3 & \text{if} \quad \mathbf{w}^\top \mathbf{x} > \tau \end{cases}$$

Locality-sensitive hashing

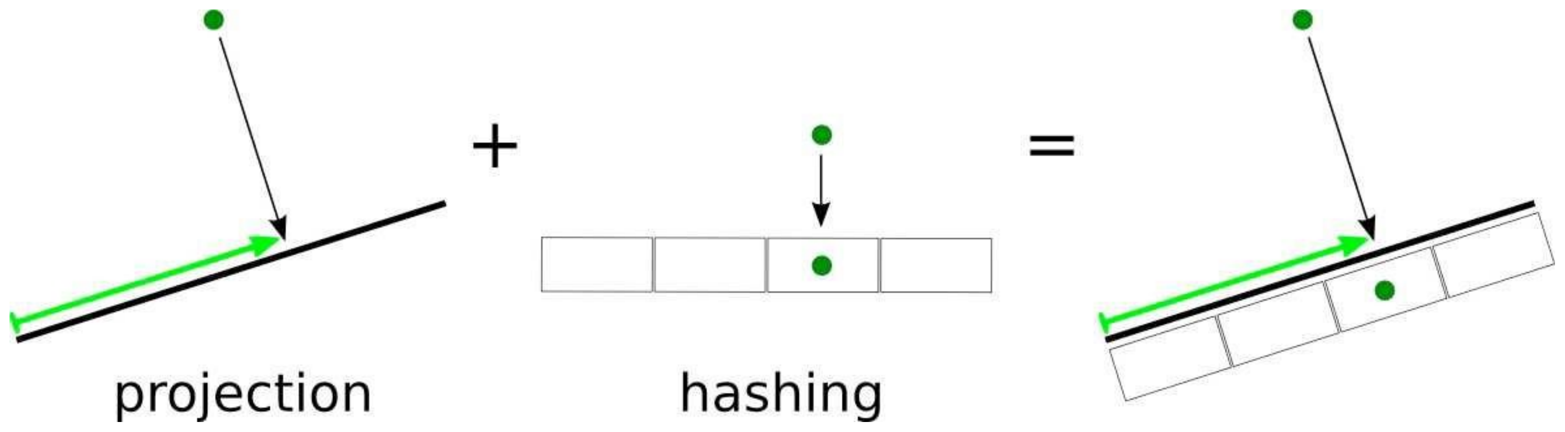


- Mathematically, we could express this locality sensitive hash function as:

$$h(\mathbf{x}) = \begin{cases} 0 & \text{if} \quad \mathbf{w}^\top \mathbf{x} \leq -\tau \\ 1 & \text{if} \quad -\tau < \mathbf{w}^\top \mathbf{x} \leq 0 \\ 2 & \text{if} \quad 0 < \mathbf{w}^\top \mathbf{x} \leq \tau \\ 3 & \text{if} \quad \mathbf{w}^\top \mathbf{x} > \tau \end{cases}$$

random projection

Locality-sensitive hashing



- Mathematically, we could express this locality sensitive hash function as:

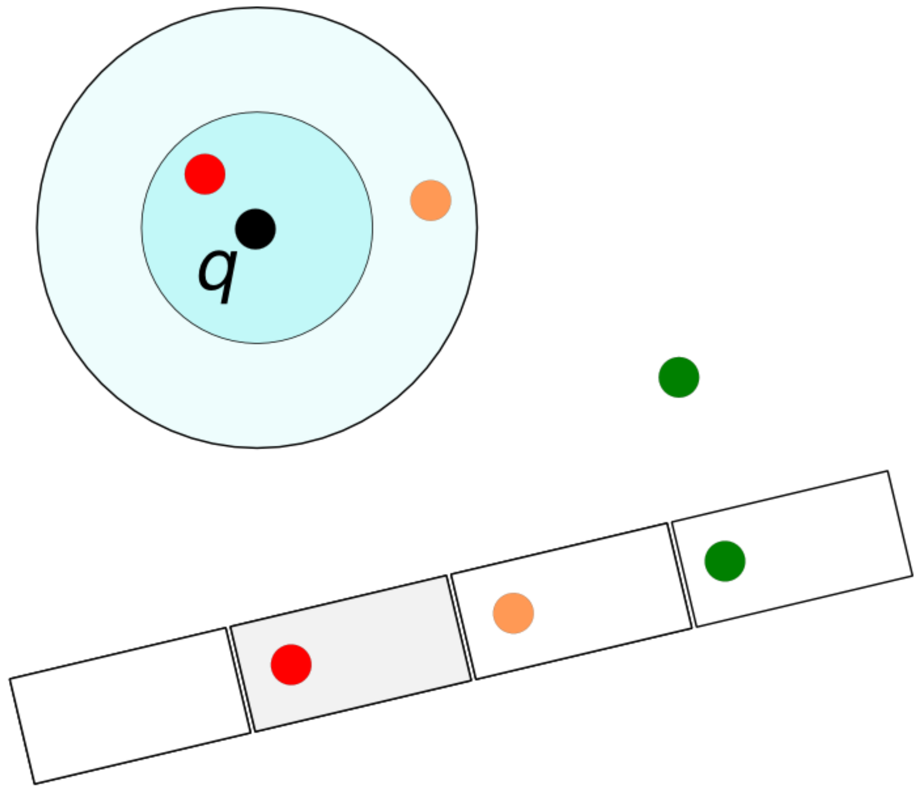
$$h(\mathbf{x}) = \begin{cases} 0 & \text{if} \quad \mathbf{w}^\top \mathbf{x} \le -\tau \\ 1 & \text{if} \quad -\tau < \mathbf{w}^\top \mathbf{x} \le 0 \\ 2 & \text{if} \quad 0 < \mathbf{w}^\top \mathbf{x} \le \tau \\ 3 & \text{if} \quad \mathbf{w}^\top \mathbf{x} > \tau \end{cases}$$

random projection

threshold parameter

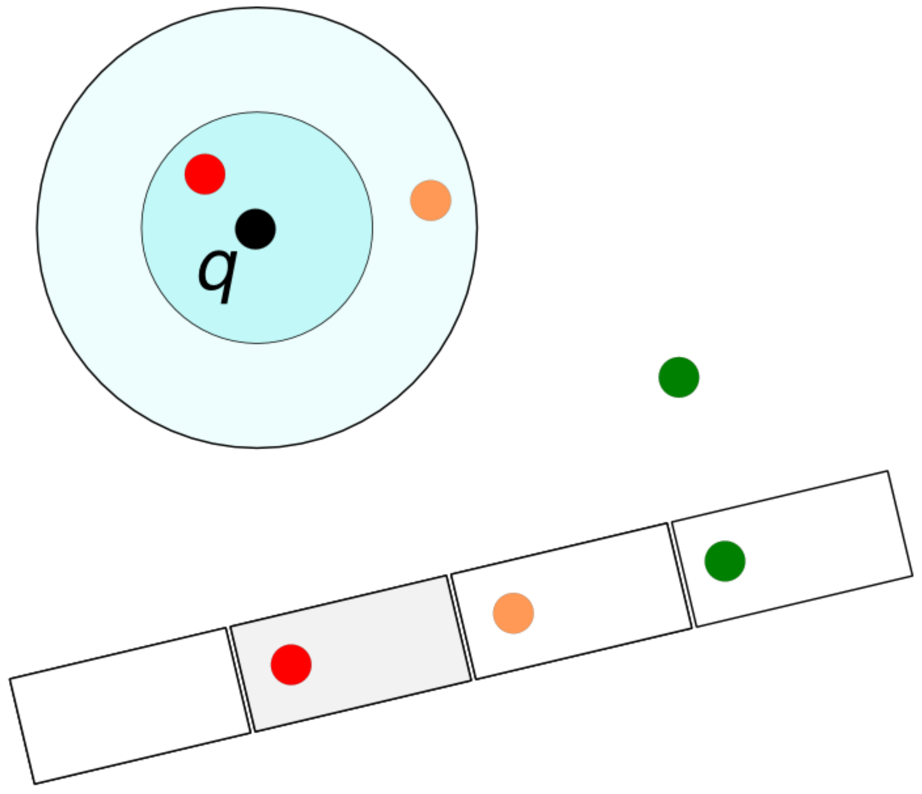
Locality-sensitive hashing

- Retrieval of nearest neighbors of a query point *q* using LSH works as follows:



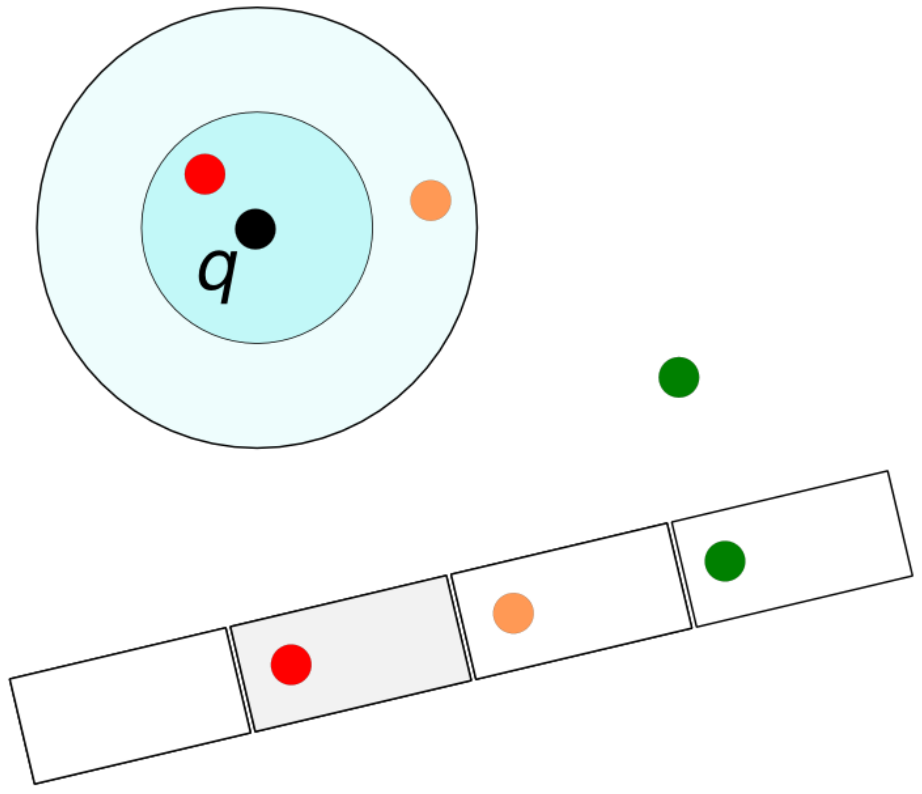
Locality-sensitive hashing

- Retrieval of nearest neighbors of a query point *q* using LSH works as follows:
  - Hash all data points using locality-sensitive hash



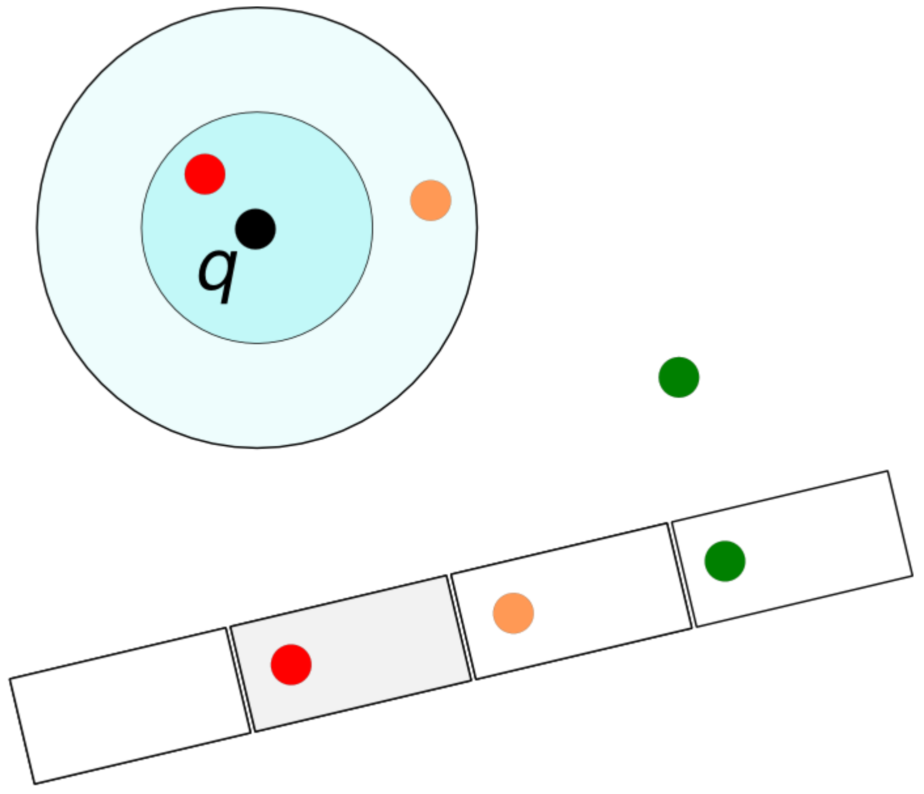
Locality-sensitive hashing

- Retrieval of nearest neighbors of a query point *q* using LSH works as follows:
  - Hash all data points using locality-sensitive hash
  - Compute locality-sensitive hash of query point



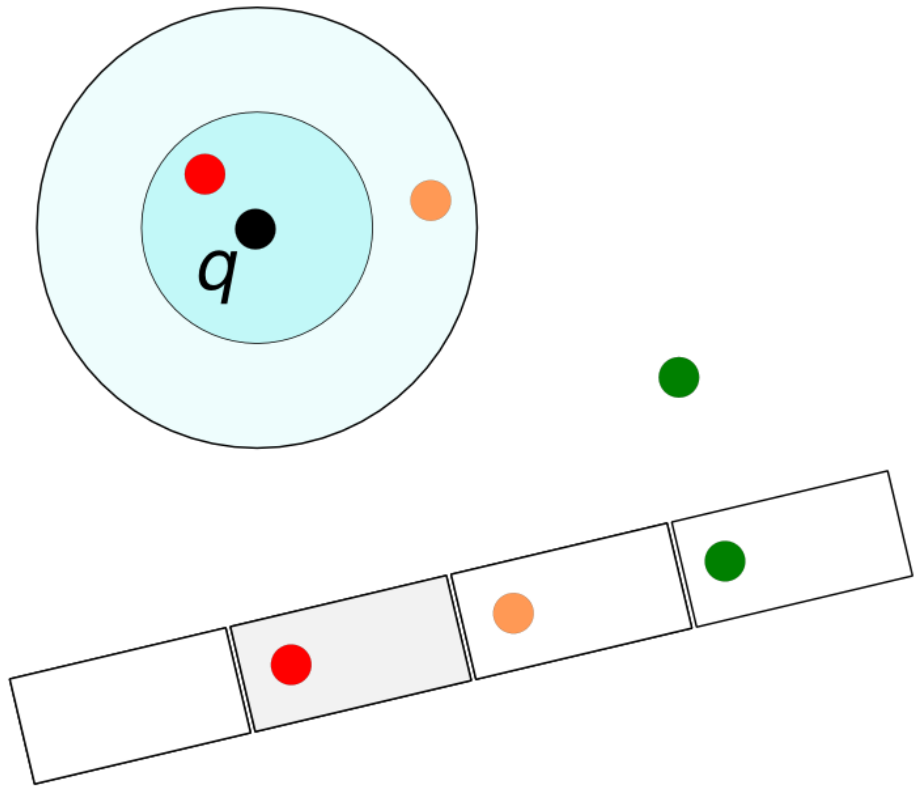
Locality-sensitive hashing

- Retrieval of nearest neighbors of a query point *q* using LSH works as follows:
  - Hash all data points using locality-sensitive hash
  - Compute locality-sensitive hash of query point
  - All data points in the bucket are candidate near neighbors



Locality-sensitive hashing

- Retrieval of nearest neighbors of a query point *q* using LSH works as follows:
  - Hash all data points using locality-sensitive hash
  - Compute locality-sensitive hash of query point
  - All data points in the bucket are candidate near neighbors
  - Compute distances to candidate points to find true nearest neighbors

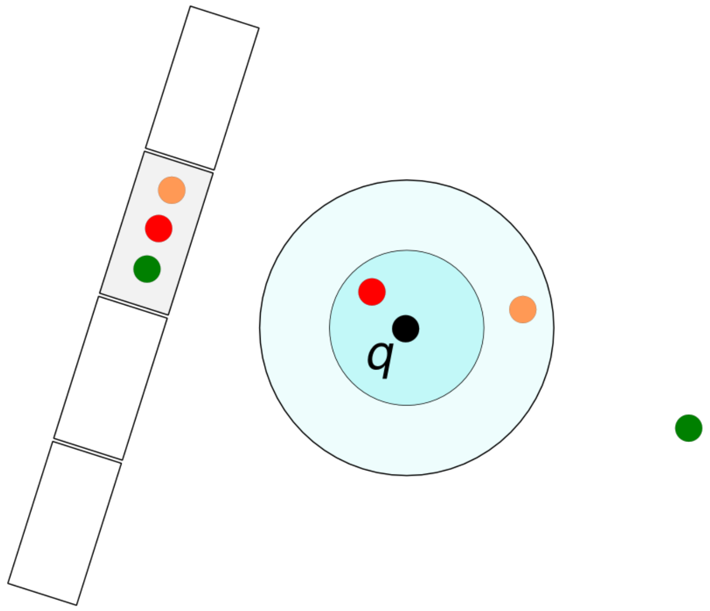


Locality-sensitive hashing

- LSH projections may be “unlucky” in two main ways:

Locality-sensitive hashing

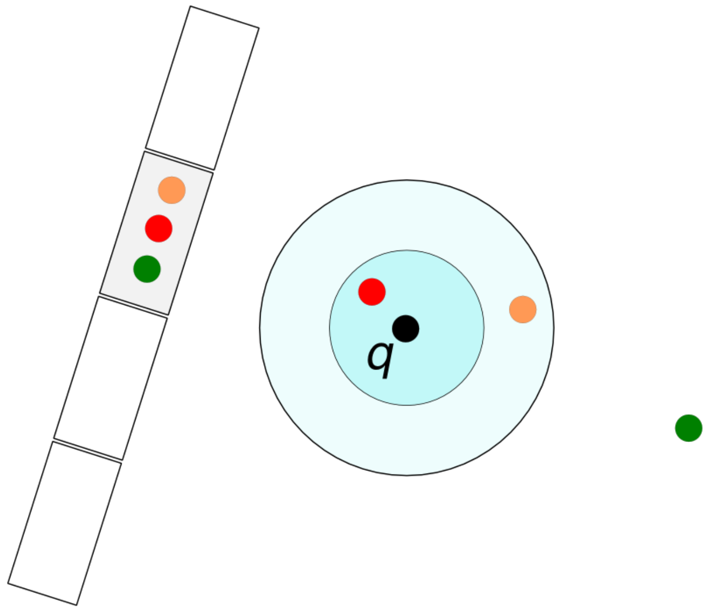
- LSH projections may be “unlucky” in two main ways:



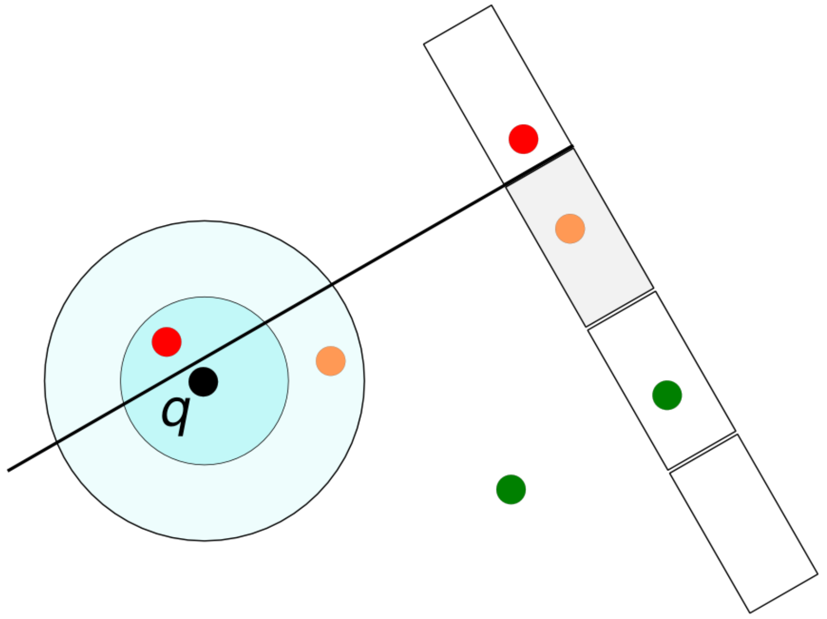
“Collision”: Distant points hashed in the same bucket

Locality-sensitive hashing

- LSH projections may be “unlucky” in two main ways:



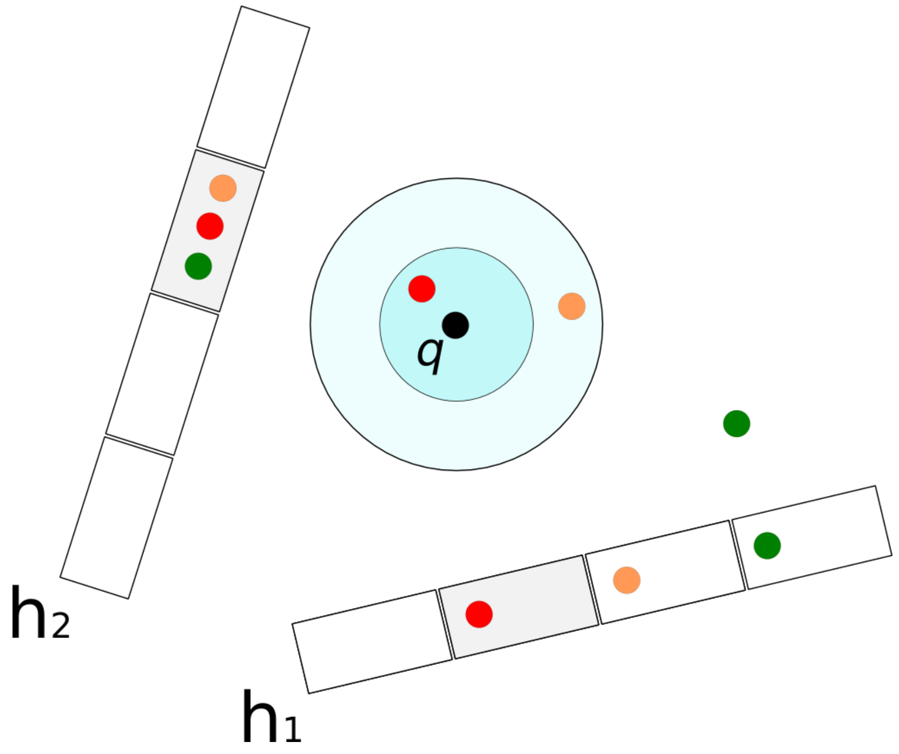
“Collision”: Distant points hashed in the same bucket



“Split”: Nearby points hashed in different buckets

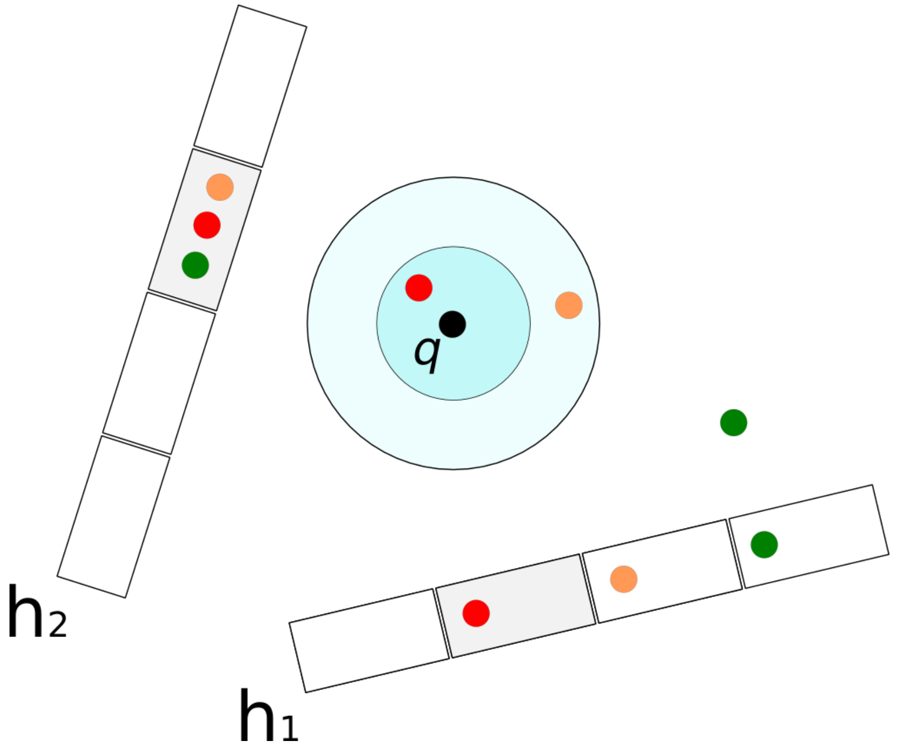
Locality-sensitive hashing

- Using multiple projections in an LSH resolves “collisions”:



Locality-sensitive hashing

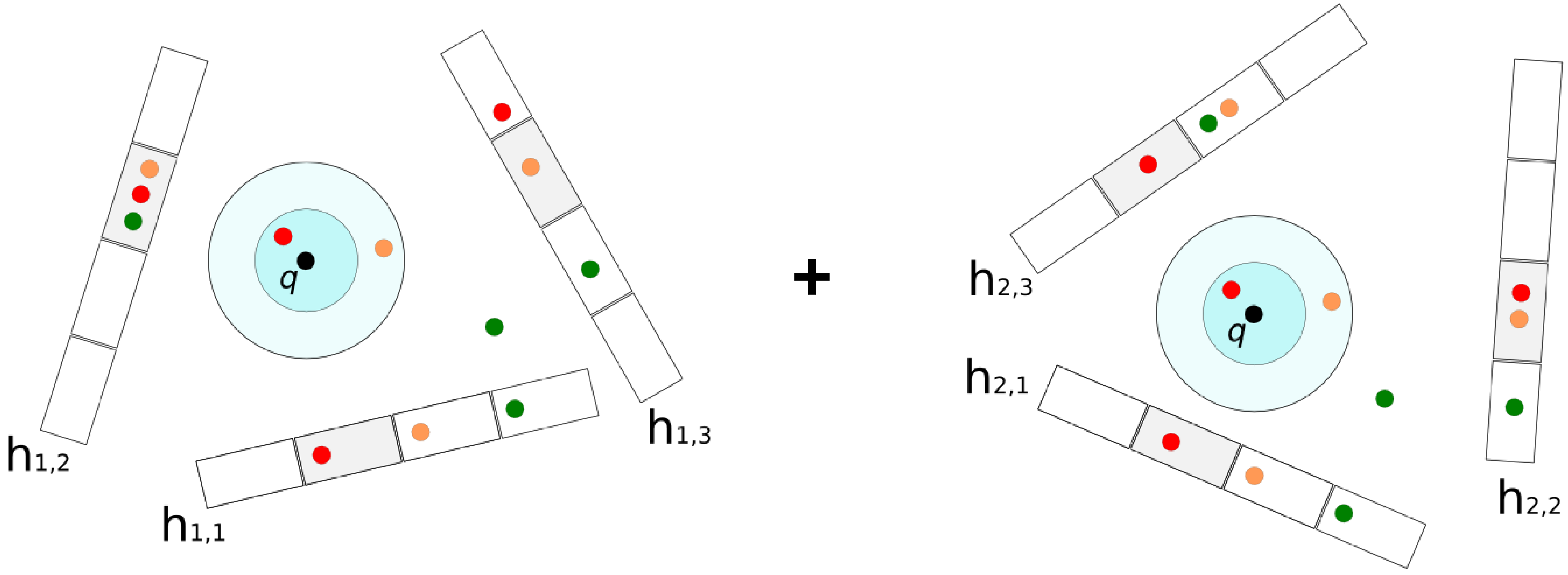
- Using multiple projections in an LSH resolves “collisions”:



- The LSH is given by a concatenation of all individual buckets

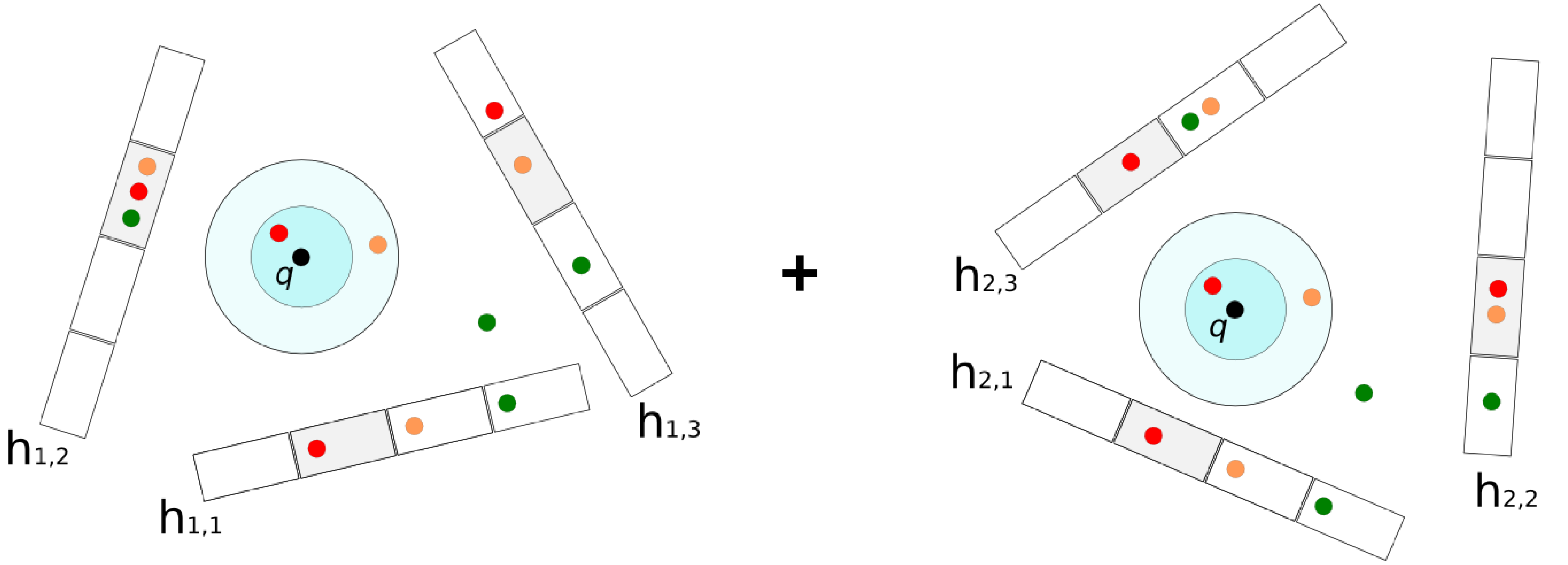
Locality-sensitive hashing

- Using multiple separate hash tables when doing LSH resolves “splits”:



Locality-sensitive hashing

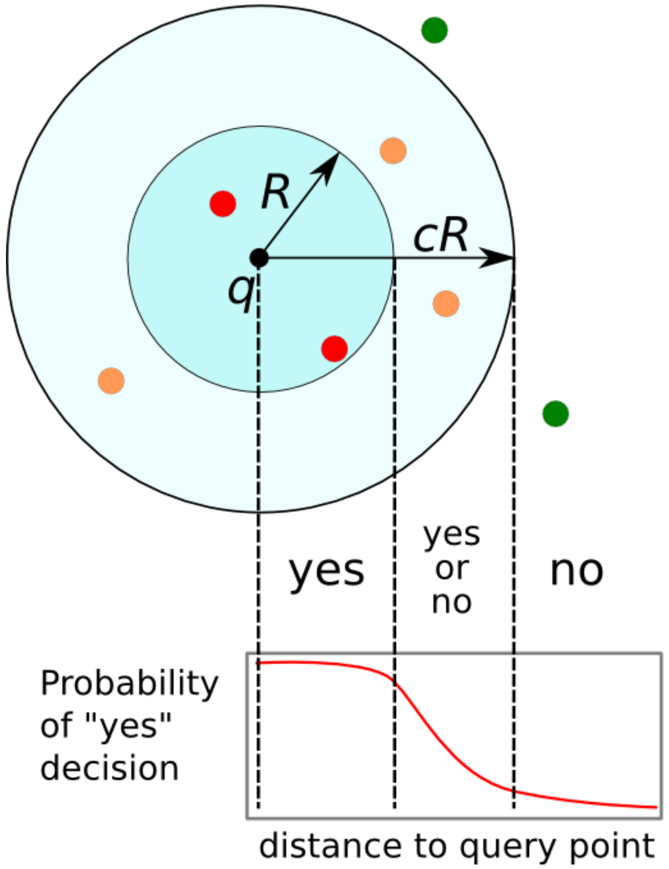
- Using multiple separate hash tables when doing LSH resolves "splits":

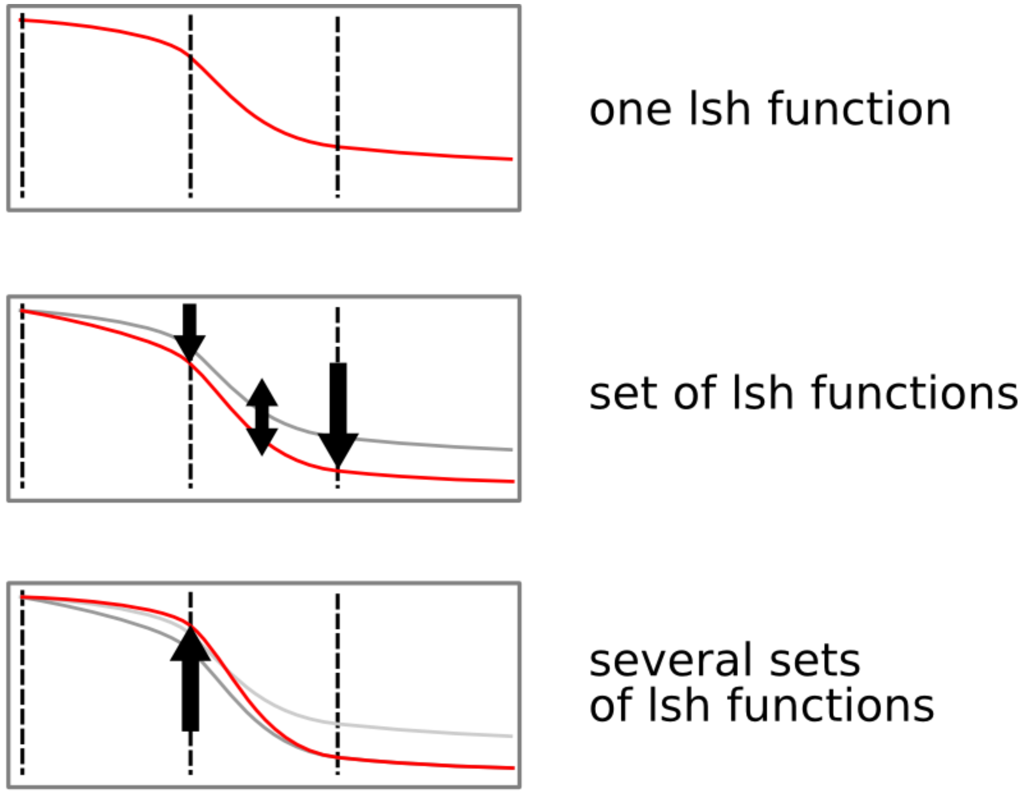


- Points are candidate neighbors if candidate in *any* of the hash tables

Locality-sensitive hashing

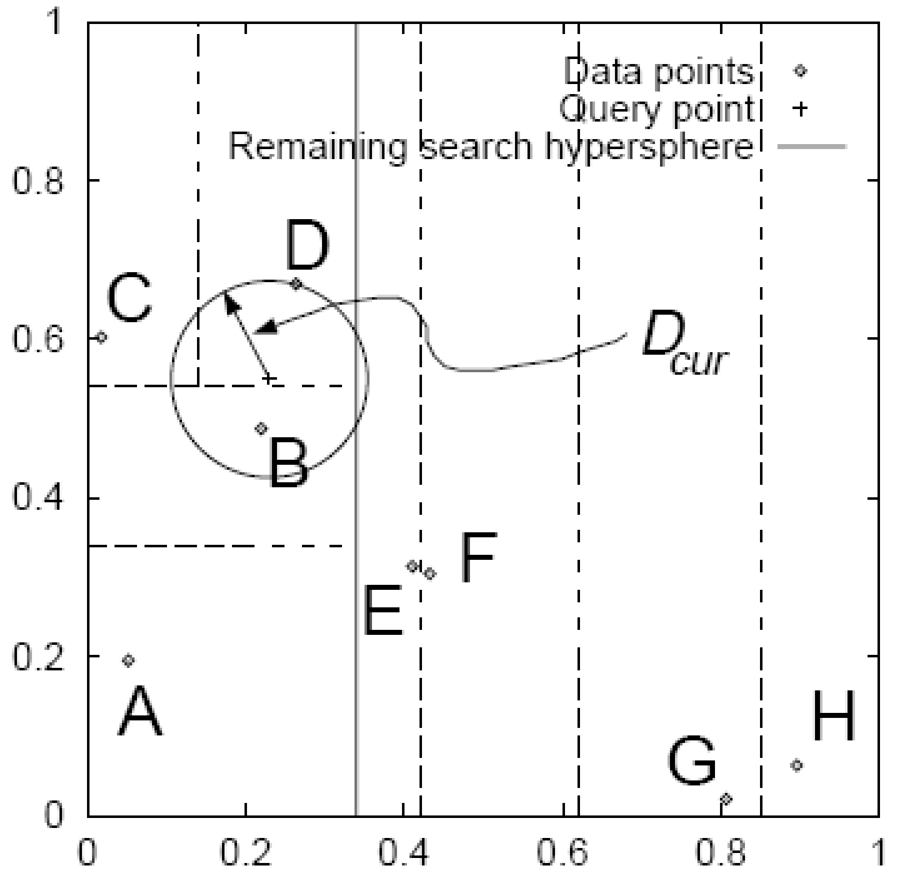
- Error analysis of locality-sensitive hashing:

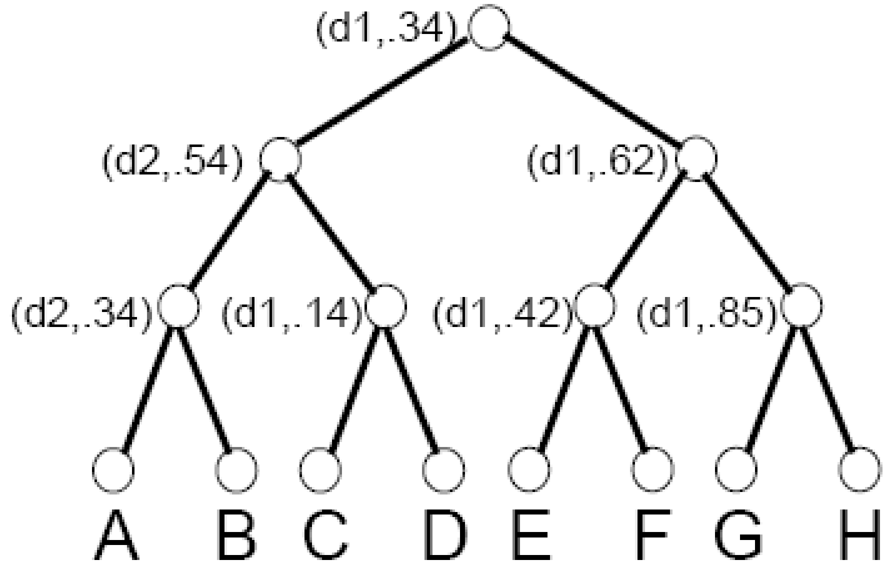




kd-trees

- Tree structure that optimally splits a random dimension at each level:



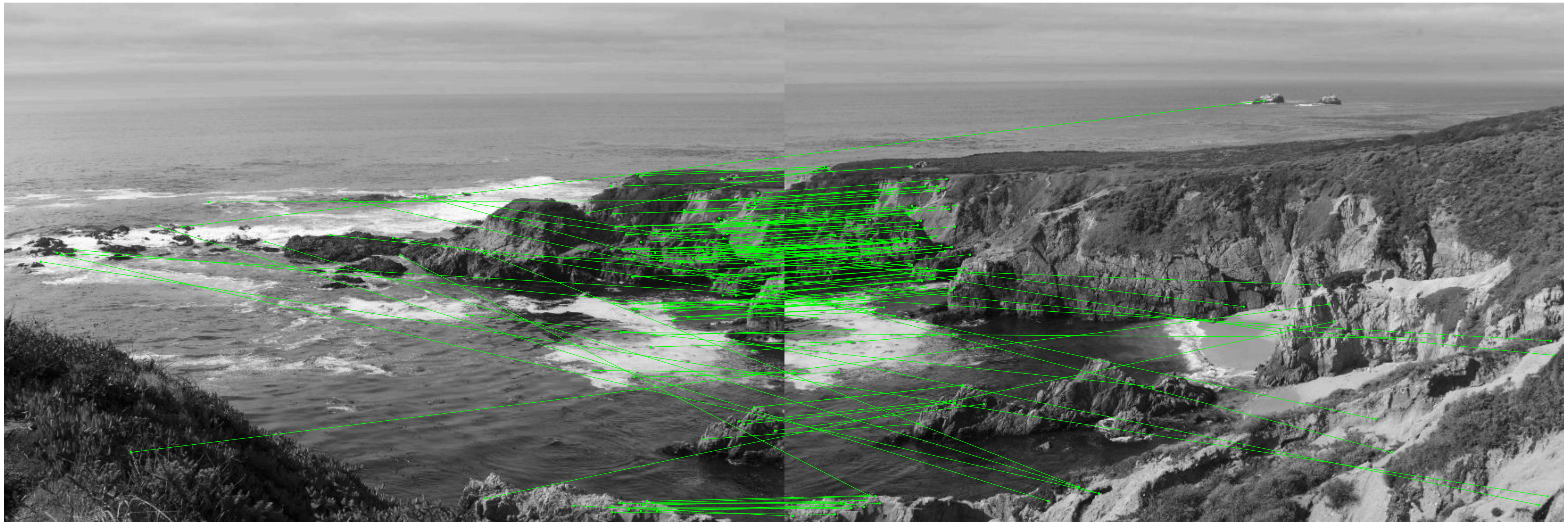


kd-trees

- Finding the nearest neighbor of a target point using a kd-tree:
    - Identify the bin in which the target point is located
    - Compute distance to nearest neighbor inside this bin (D_{cur})
    - Perform depth-first search on the kd-tree:
        - Prune all cells that are further away than D_{cur}
        - If we arrive at a leaf, search for nearer neighbors, and update D_{cur}

SIFT Matches

- Matches SIFT finds are not generally free of errors:

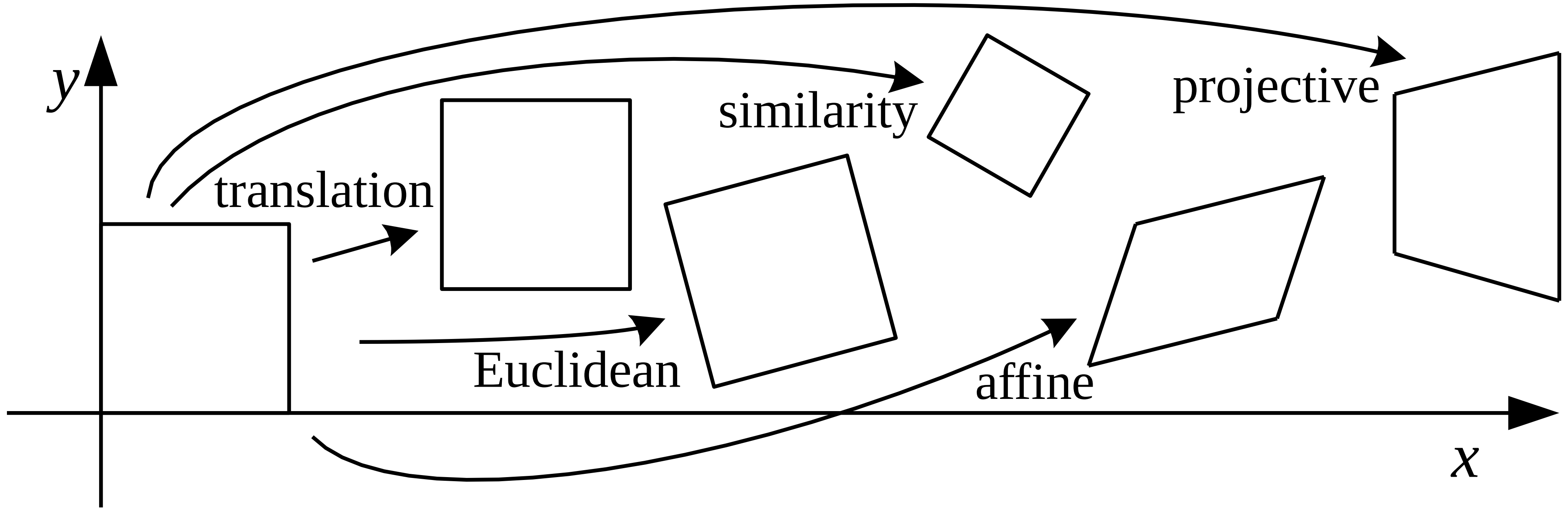


Feature-based alignment

Feature-based alignment

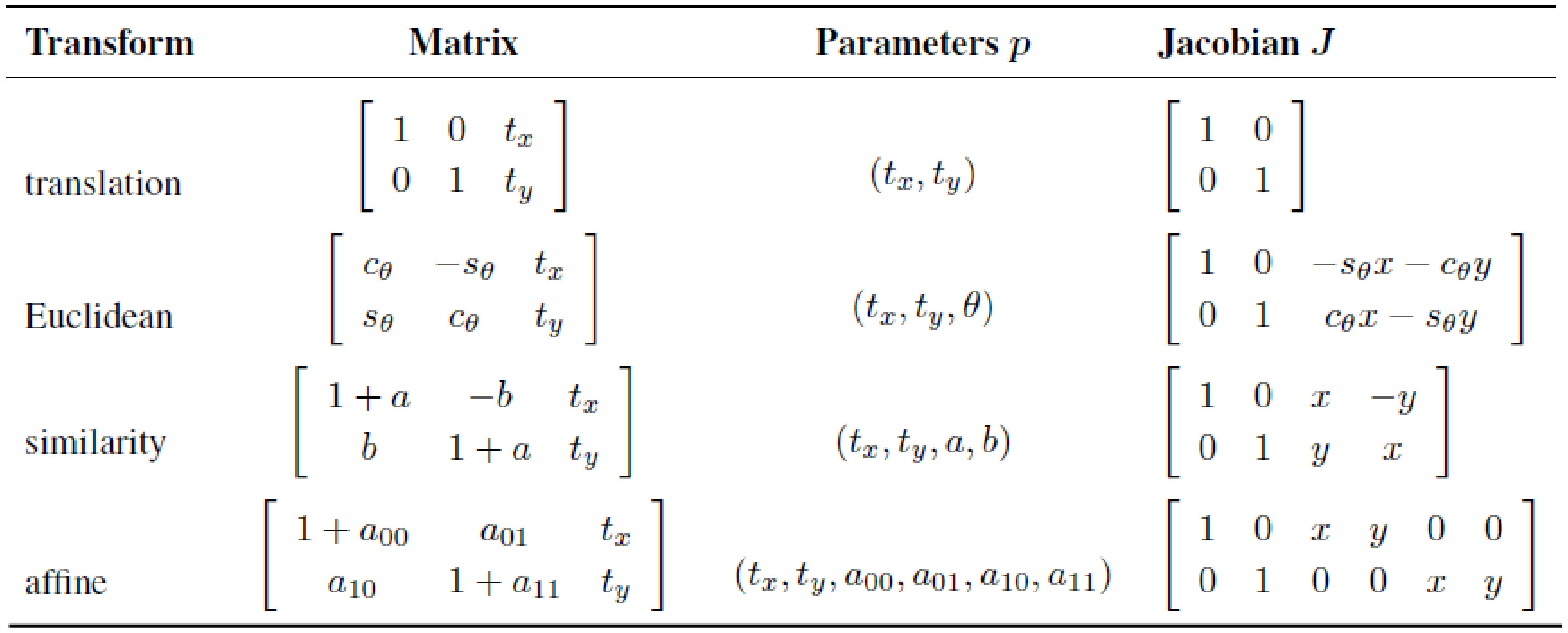
- Estimating the motion between two images based on set of matched points

- Basic collection of 2D (planar) *coordinate transformations*:



Feature-based alignment

- Basic 2D coordinate transformations $\mathbf{x}' = f(\mathbf{x}; \mathbf{p})$:



Transform	Matrix	Parameters p	Jacobian J
translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$	(t_x, t_y)	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Euclidean	$\begin{bmatrix} c_\theta & -s_\theta & t_x \\ s_\theta & c_\theta & t_y \end{bmatrix}$	(t_x, t_y, θ)	$\begin{bmatrix} 1 & 0 & -s_\theta x - c_\theta y \\ 0 & 1 & c_\theta x - s_\theta y \end{bmatrix}$
similarity	$\begin{bmatrix} 1+a & -b & t_x \\ b & 1+a & t_y \end{bmatrix}$	(t_x, t_y, a, b)	$\begin{bmatrix} 1 & 0 & x & -y \\ 0 & 1 & y & x \end{bmatrix}$
affine	$\begin{bmatrix} 1+a_{00} & a_{01} & t_x \\ a_{10} & 1+a_{11} & t_y \end{bmatrix}$	$(t_x, t_y, a_{00}, a_{01}, a_{10}, a_{11})$	$\begin{bmatrix} 1 & 0 & x & y & 0 & 0 \\ 0 & 1 & 0 & 0 & x & y \end{bmatrix}$

- These transformations use *augmented vector* representation: $\mathbf{x} = [x \quad y \quad 1]^T$

Feature-based alignment

- *Least squares* provides a simple way to estimate parameters **p** of transform

- Assuming a set of matched feature points $\{(\mathbf{x}_i, \mathbf{x}'_i)\}_{i=1,\ldots,N}$, we minimize:

$$E = \sum_i \|f(\mathbf{x}_i; \mathbf{p}) - \mathbf{x}'_i\|^2$$

- If transformation is linear in parameters, closed-form solution exists!

Linear least squares

- Consider the *linear least squares* optimization problem:

$$g(\mathbf{x}) = \|\underbrace{\mathbf{A}\mathbf{x} - \mathbf{b}}_{\text{residual } r \text{ (linear in parameters)}}\|^2 \qquad \underbrace{\mathbf{x}^*}_{\text{optimal parameters}} = \underset{\mathbf{x}}{\operatorname{argmin}}\, g(\mathbf{x})$$

Linear least squares

- Consider the *linear least squares* optimization problem:

$$g(\mathbf{x}) = \|\underbrace{\mathbf{A}\mathbf{x} - \mathbf{b}}_{\text{residual } r \text{ (linear in parameters)}}\|^2 \qquad \underbrace{\mathbf{x}^*}_{\text{optimal parameters}} = \operatorname*{argmin}_{\mathbf{x}} g(\mathbf{x})$$

- We can solve such an optimization problem by *setting the gradient to zero*:

$$\frac{\partial g}{\partial \mathbf{x}} = 2\mathbf{A}^{\mathrm{T}}(\mathbf{A}\mathbf{x} - \mathbf{b}) = 0$$

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{x} = \mathbf{A}^{\mathrm{T}}\mathbf{b}$$

$$\mathbf{x} = \underbrace{(\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{T}}}_{\text{pseudo-inverse of } \mathbf{A}}\mathbf{b}$$

Feature-based alignment

- For translation, similarity and affine transforms, the movement is *linear* in **p**:

$$E = \sum_i \|f(\mathbf{x}_i;\mathbf{p}) - \mathbf{x}'_i\|^2 = \sum_i \|\mathbf{x}_i + J(\mathbf{x}_i)\mathbf{p} - \mathbf{x}'_i\|^2$$
$$= \sum_i \|J(\mathbf{x}_i)\mathbf{p} - \Delta\mathbf{x}_i\|^2$$

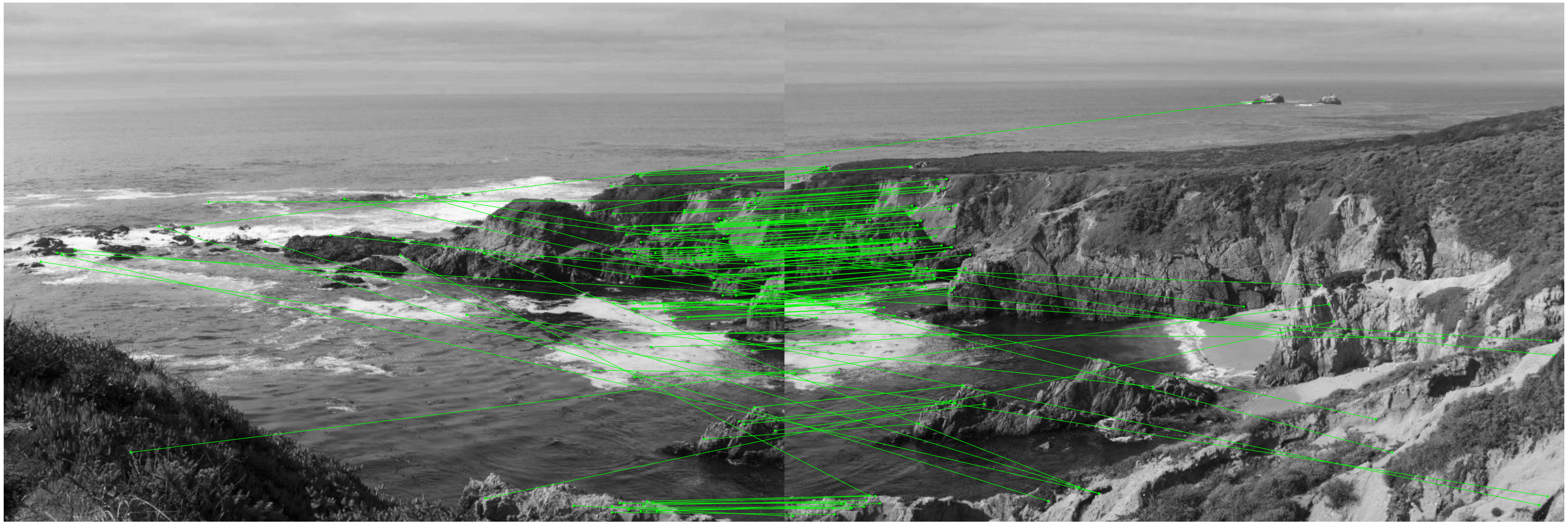
with: $\Delta\mathbf{x}_i = \mathbf{x}'_i - \mathbf{x}_i$

- The optimal solution for the problem is thus given in closed form:

$$\mathbf{p}^* = \left(\sum_i J(\mathbf{x}_i)^\top J(\mathbf{x}_i)\right)^{-1} \sum_i J^\top(\mathbf{x}_i)\Delta\mathbf{x}_i$$

Example: Panography

- Simple example of feature-based alignment: Panography (translation model)



- Let’s work out the optimal translation based on correspondences

Example: Panography

- Our aim is to find the optimal translation based on keypoint correspondences:

$$g(t_x, t_y) = \sum_{n=1}^{N} \| \begin{bmatrix} x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ t_x & t_y \end{bmatrix} - \begin{bmatrix} x'_n \\ y'_n \end{bmatrix} \|^2$$

$$= \sum_{n=1}^{N} \| \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \end{bmatrix} - \begin{bmatrix} x_n - x'_n \\ y_n - y'_n \end{bmatrix} \|^2$$

$$= \| \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \dots & \dots \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \end{bmatrix} - \begin{bmatrix} x_1 - x'_1 \\ y_1 - y'_1 \\ \dots \\ x_N - x'_N \\ y_N - y'_N \end{bmatrix} \|^2$$

Example: Panography

- Our aim is to find the optimal translation based on keypoint correspondences:

$$g(t_x, t_y) = \sum_{n=1}^{N} \| \begin{bmatrix} x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ t_x & t_y \end{bmatrix} - \begin{bmatrix} x'_n \\ y'_n \end{bmatrix} \|^2$$

$$= \sum_{n=1}^{N} \| \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \end{bmatrix} - \begin{bmatrix} x_n - x'_n \\ y_n - y'_n \end{bmatrix} \|^2$$

$$= \| \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \dots & \dots \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \end{bmatrix} - \begin{bmatrix} x_1 - x'_1 \\ y_1 - y'_1 \\ \dots \\ x_N - x'_N \\ y_N - y'_N \end{bmatrix} \|^2$$

- This can be recognized as a standard *linear least-squares* problem!

RANSAC

- Erroneous matches may highly influence our estimate for t_x and t_y

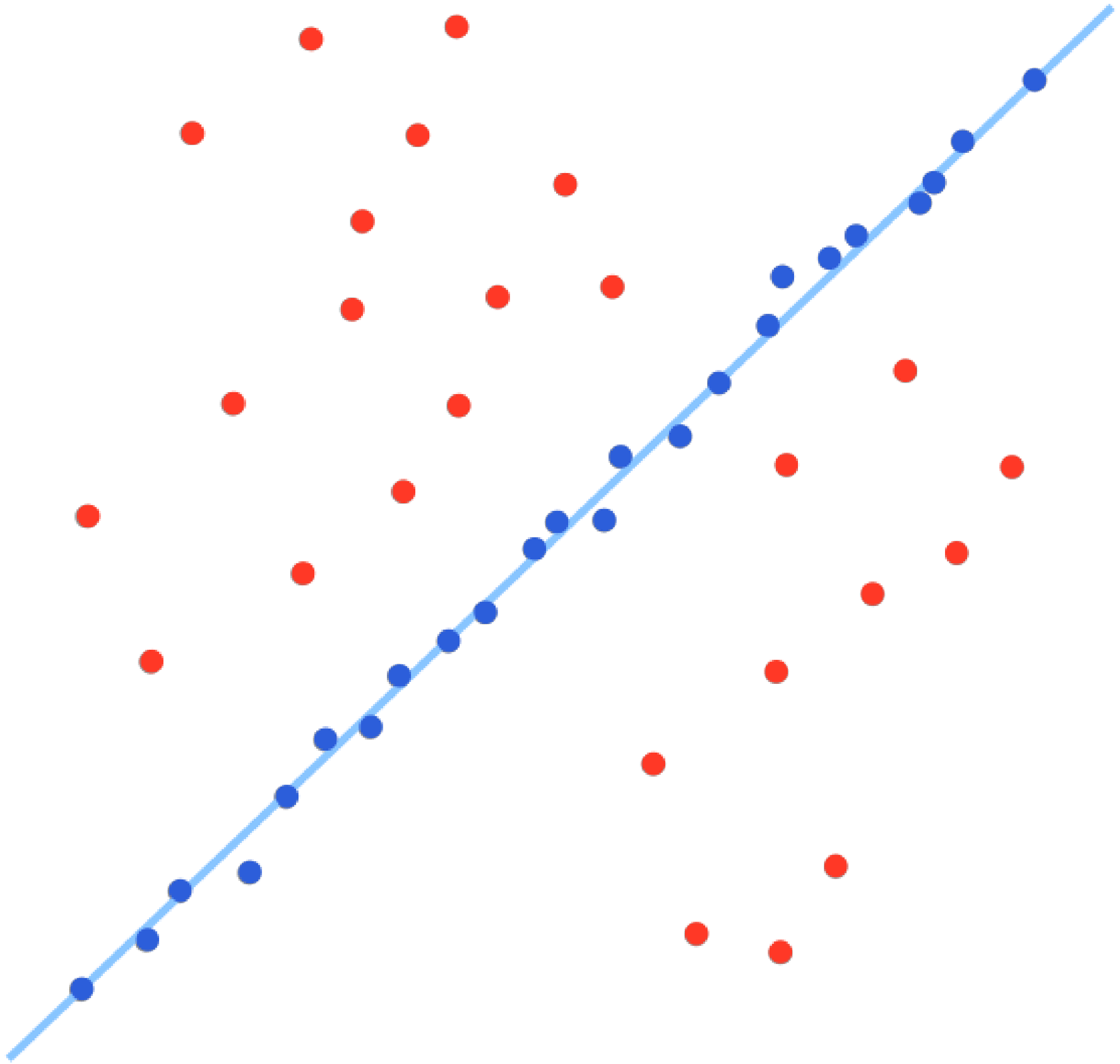
RANSAC

- Erroneous matches may highly influence our estimate for t_x and t_y

- A *RANSAC* algorithm for fitting a panography would roughly work as follows:
  - *Fit model* on current inliers: solve linear-least squares problem
  - *Determine inliers*: *e.g.*, find points for which $\|x_n + t_x - x'_n\|^2 + \|y_n + t_y - y'_n\|^2$ is small, and consider those as the new inliers
  - Return to the first step using the new inliers
  - Keep track of the best model so far (in terms of the squared error) that has “sufficient” inliers

RANSAC

- *RAndom SAmple Consensus* aims to fit model whilst being robust to outliers:



RANSAC

- RANSAC is an iterative algorithm that works using the following steps:

  1) Model is fitted to the *hypothetical inliers*

  2) Data are tested against the fitted model to determine hypothetical inliers

  3) Return to step 1) until sufficient points are classified as inliers (or fixed number of times)

  4) Keep track of best model so far during iterations

RANSAC

- RANSAC is an iterative algorithm that works using the following steps:

  1) Model is fitted to the *hypothetical inliers*

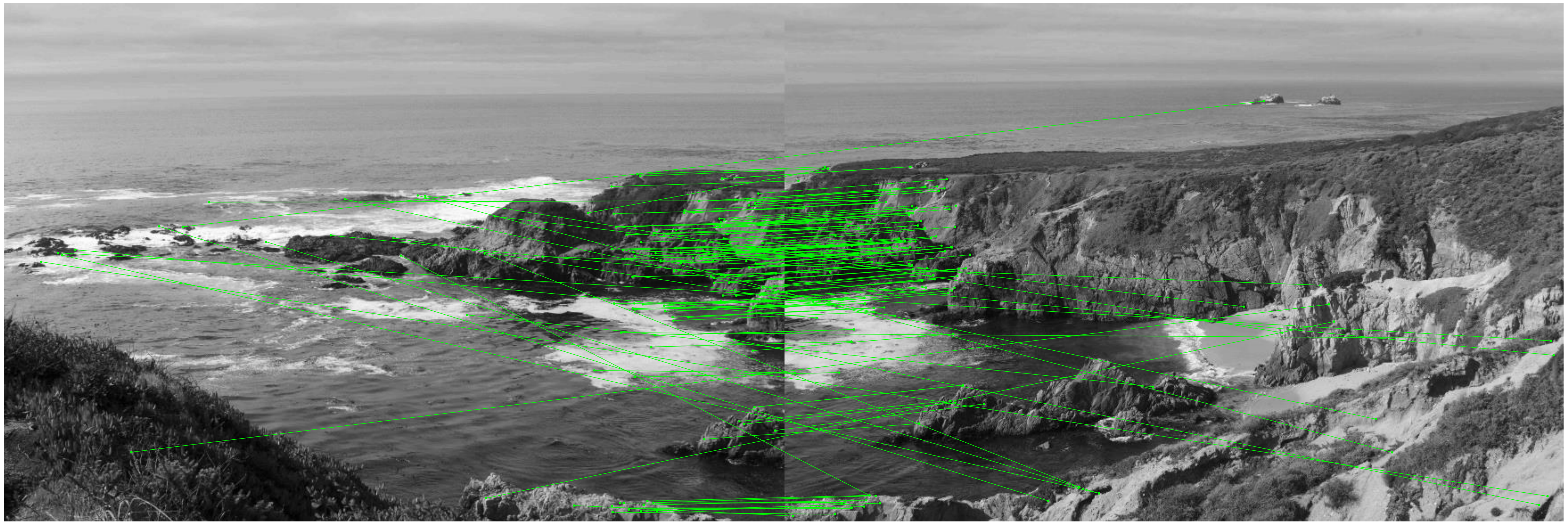
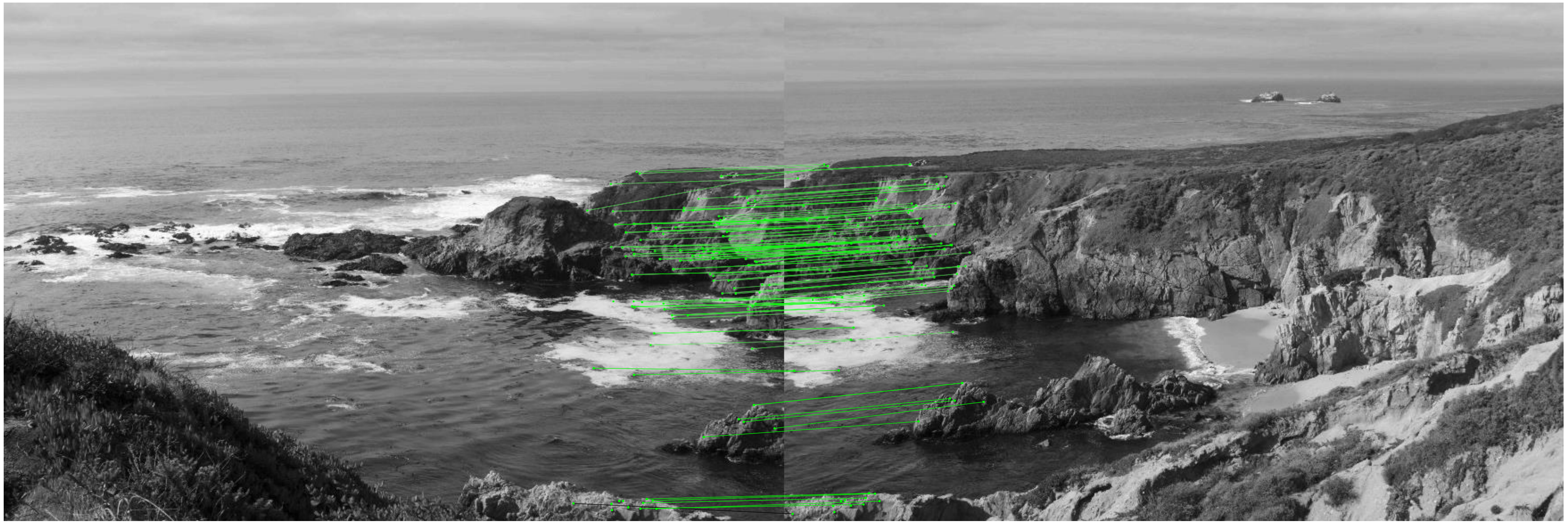
  2) Data are tested against the fitted model to determine hypothetical inliers

  3) Return to step 1) until sufficient points are classified as inliers (or fixed number of times)

  4) Keep track of best model so far during iterations

- No upper bound on number of iterations available; RANSAC may take forever

- RANSAC has two magic parameters: How far can data be from the model to be considered inlier? And how much data is needed to accept the model?

Reading material: Section 4 and 6