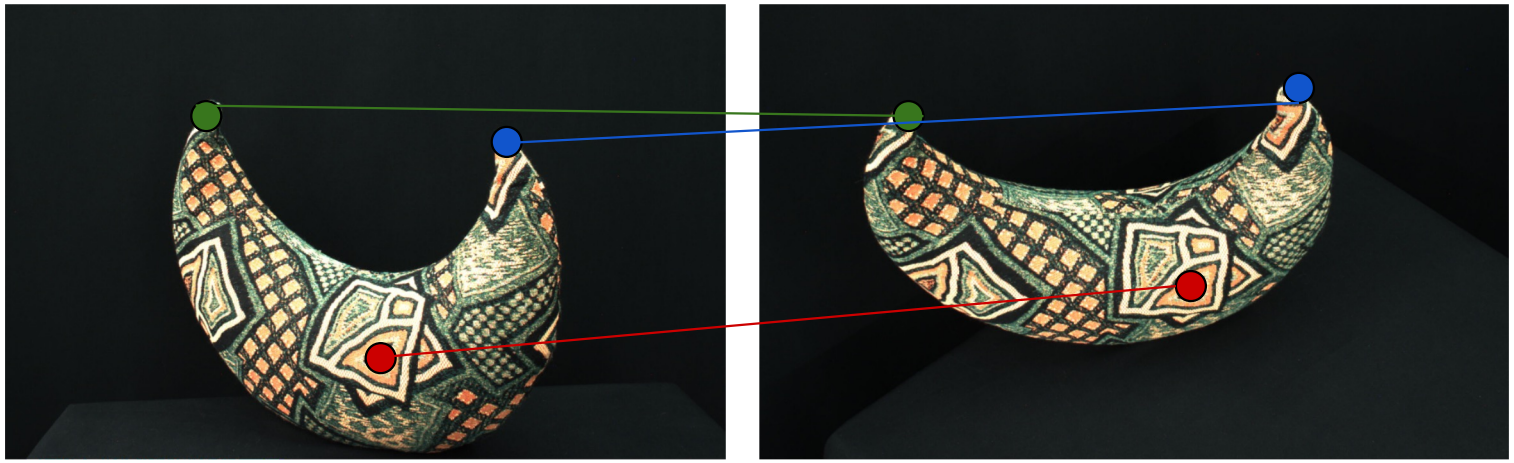
Geometric Computing and Computer Vision

Feature matching

Oleg Voynov
slides and images borrowed from a variety of sources, incl. slides by Nikolai Poliarnyi and others

FEATURE POINTS



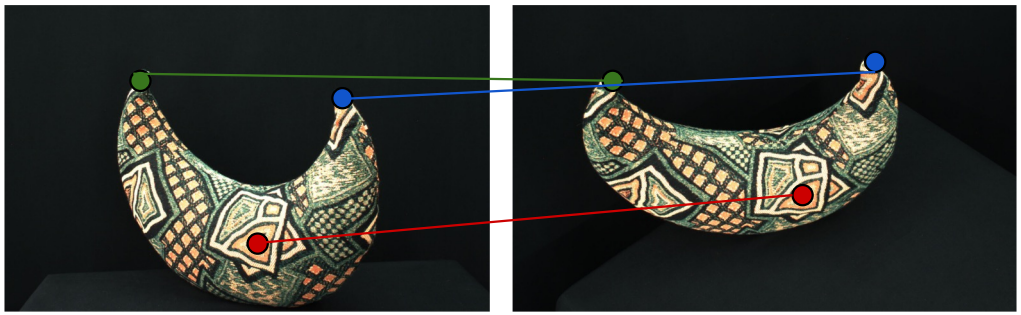
A - First view of image

B- Second view of image

- We learned how to detect and describe features / keypoints in each image independently
- **Now we want to reliably find corresponding features, i.e. correspondences / matches, robustly to errors and noise in the feature extraction process**

Q: How do we find a pair of corresponding features?

MATCHING



A - First view of image

B - Second view of image

$$\sqrt{\sum_k (H_1(k) - H_2(k))^2}$$

For each feature / keypoint in one image, we look for the closest one in the other image, in terms of some distance in descriptor space

MATCHING

BRUTE-FORCE

Input:

- M images
- N keypoints on each image
- Each keypoint has a 128-dimensional SIFT descriptor

Output (for each pair of images A and B):

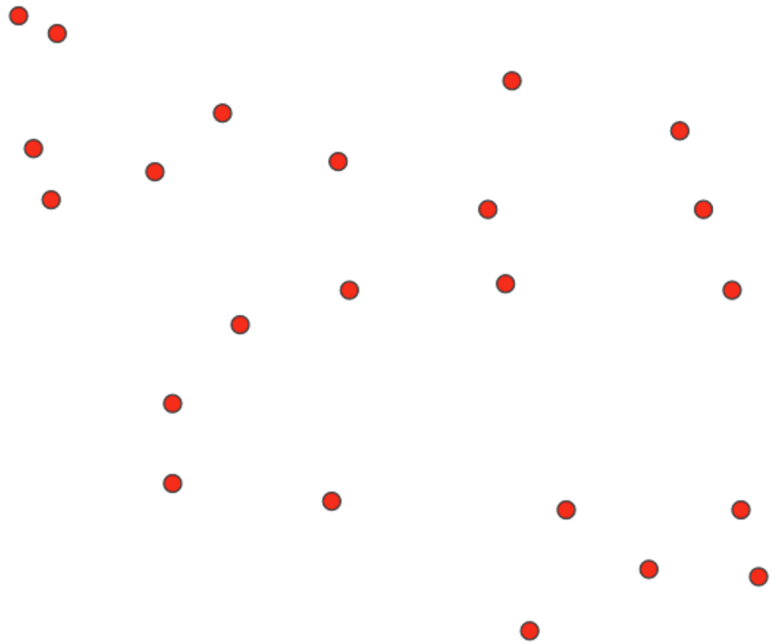
- **Nearest-Neighbor:** for each keypoint from A, find the most similar keypoint from B

Algorithmic complexity: $O(M^2 \cdot N^2)$

Q: How to speed up the second factor?
How to accelerate nearest neighbor search?

MATCHING POINTS

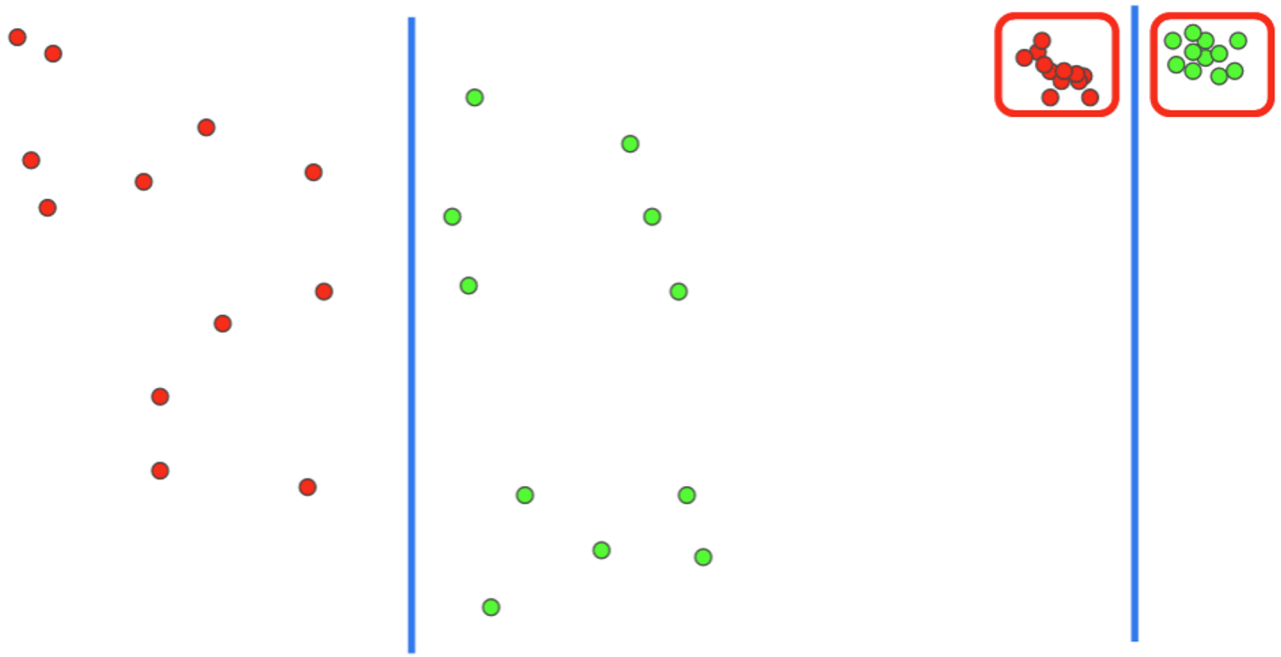
KD-tree



The goal:
to partition the space so that we can quickly find points that are close to a given point

MATCHING POINTS

KD-tree

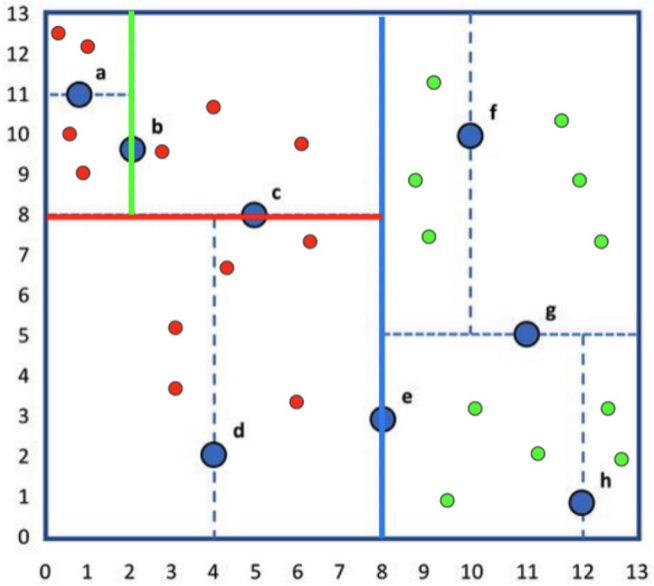


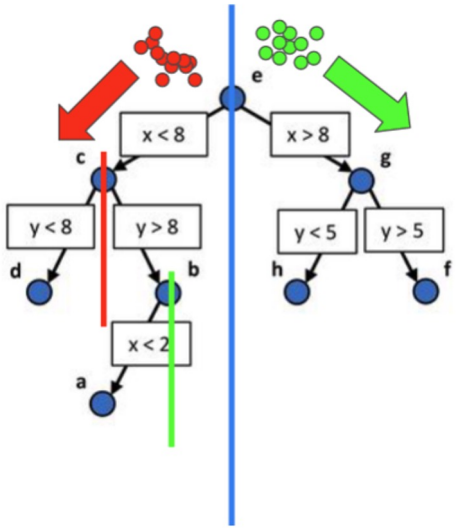
HOW WE CAN DO THIS?

Split in half along some axis

MATCHING POINTS

KD-tree

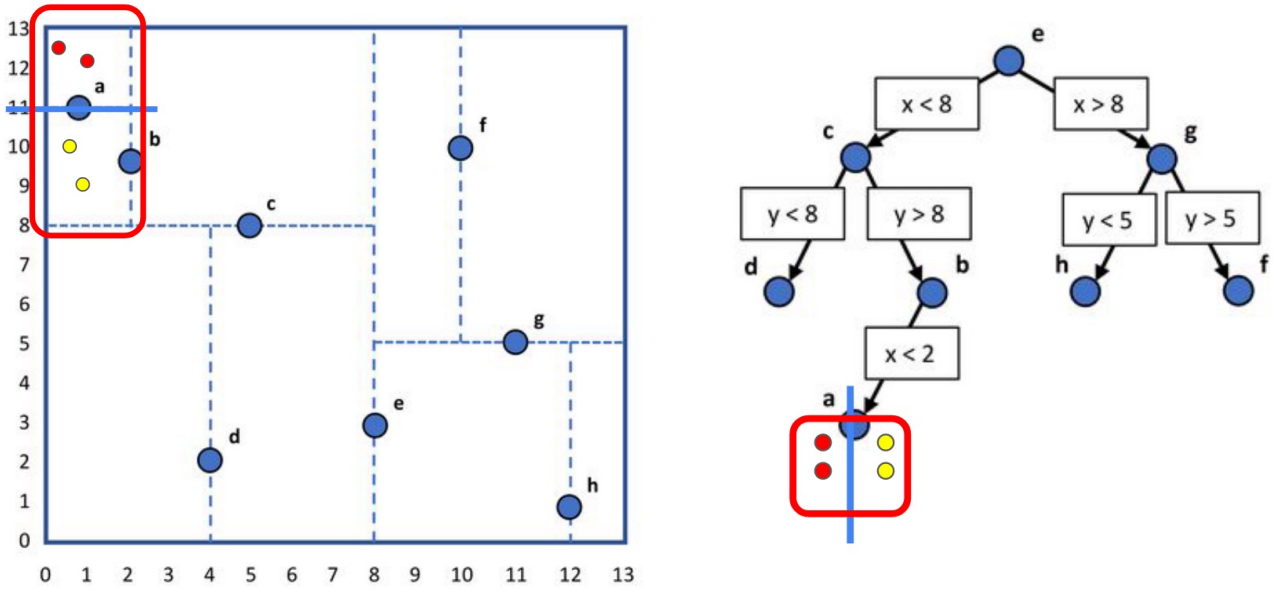




KD-tree graph

- Split points in half along X, then split in half along Y, and so on while alternating dimensions
- Each leaf contains points “packed” within a small region of space
- In 128 dimensions, we can split only a few times along each dimension

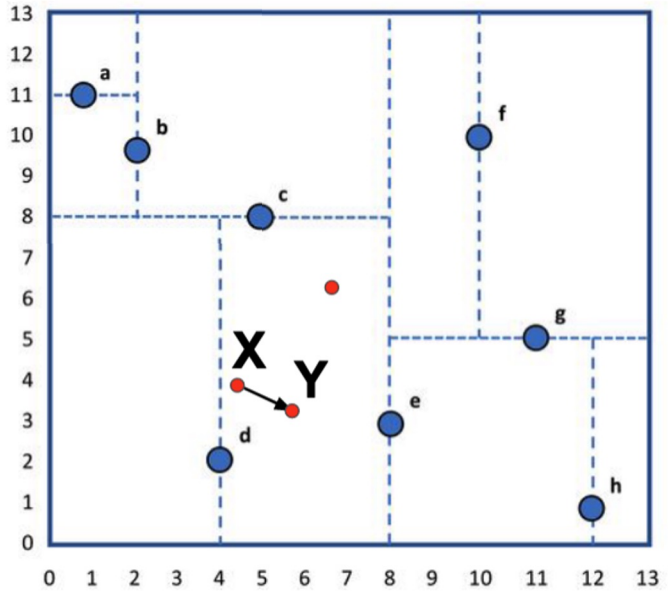
KD-tree

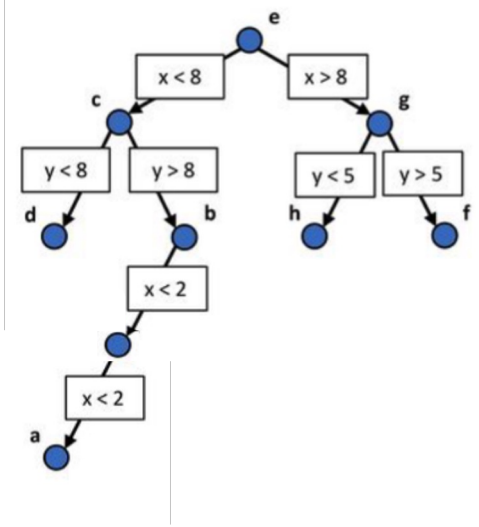


Outer / terminal nodes, i.e. the smallest cells, are called **leaves** of the tree

KD-tree

- **How do we use this for nearest neighbor search?**





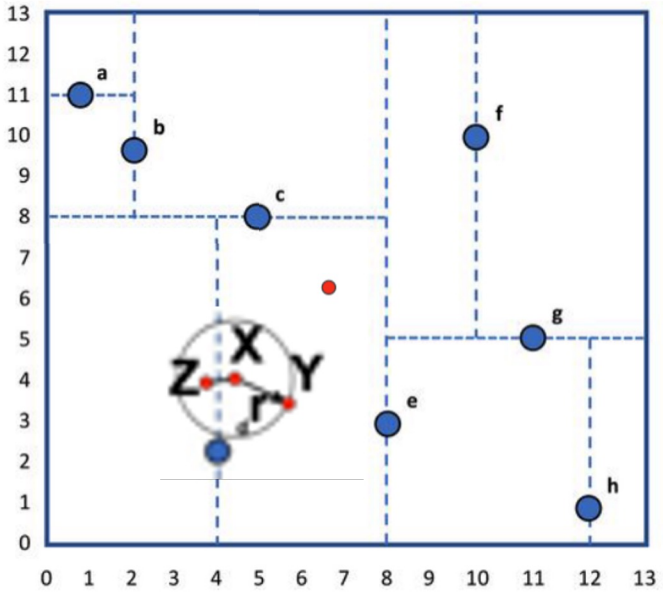
- Suppose we searching for the nearest neighbor of X
- Go to leaf containing X — these are points that are roughly in the same region as X
- Search for the closest point in the leaf

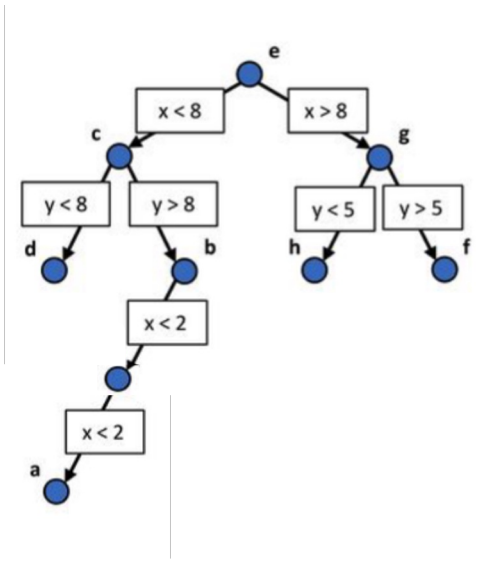
Q:

Is it sufficient to search only within the leaf?

KD-tree

- **How do we use this for nearest neighbor search?**





Q:

Is it enough to search only within the leaf?

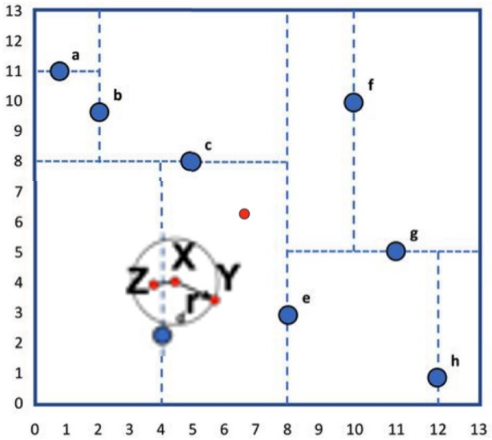
No, because the nearest neighbor might be across the boundary

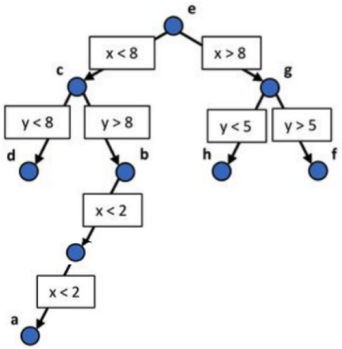
MATCHING

BRUTE-FORCE

Input:

- M images
- N keypoints on each image
- Each keypoint has a 128-dimensional SIFT descriptor





Output (for each pair of images A and B):

- Nearest-Neighbor: for each keypoint from A, find the most similar keypoint from B

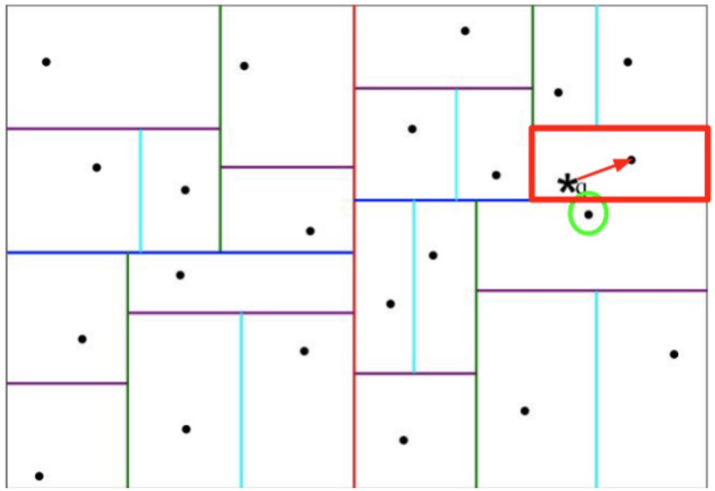
Algorithmic complexity with KD-Tree $O(M^2N^2) \rightarrow O(M^2N \log N)$

But only on average. In worst, case still this

Approximate Nearest Neighbor Search

FLANN

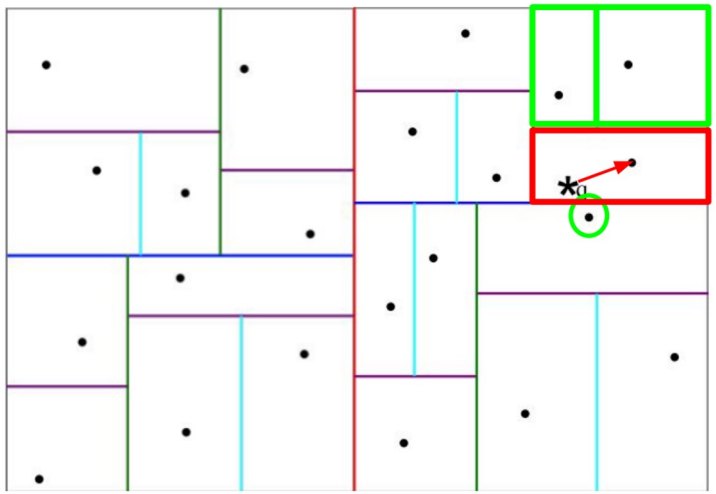
- The worst case: the nearest neighbor for the query point **q** is across the boundary of one of the root cells
- Our search starts in the leaf



Approximate Nearest Neighbor Search

FLANN

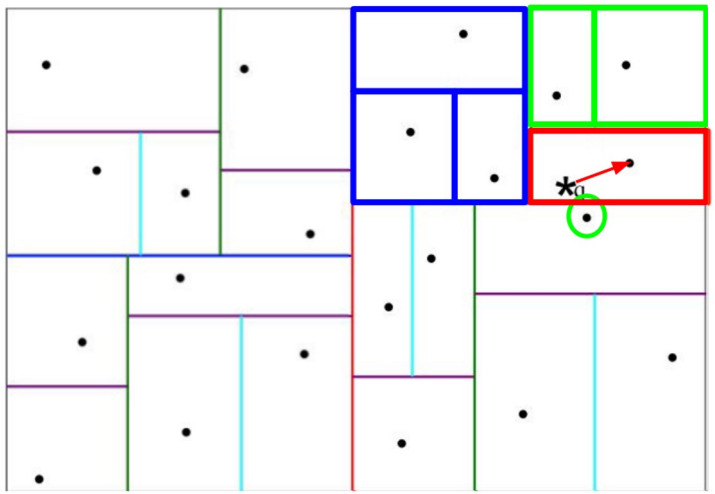
- The worst case: the nearest neighbor for the query point **q** is across the boundary of one of the root cells
- Our search starts in the leaf
- Then moves up one level, then again, and so on: **slowly walks across the whole tree**



Approximate Nearest Neighbor Search

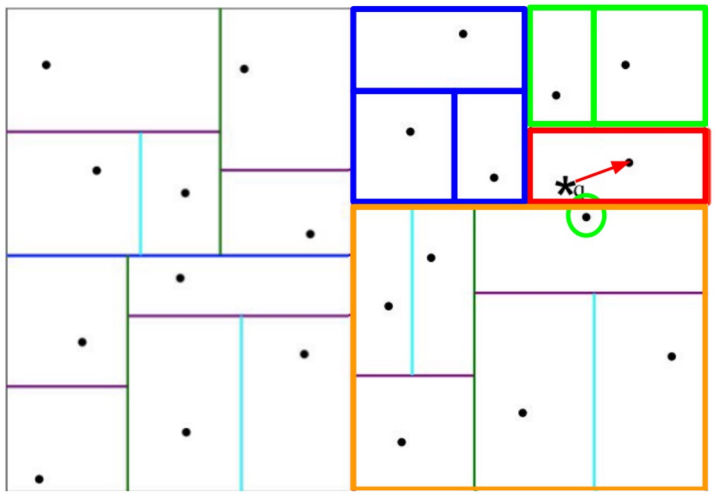
FLANN

- The worst case: the nearest neighbor for the query point **q** is across the boundary of one of the root cells
- Our search starts in the leaf
- Then moves up one level, then again, and so on: **slowly walks across the whole tree**



FLANN

- The worst case: the nearest neighbor for the query point **q** is across the boundary of one of the root cells
- Our search starts in the leaf
- Then moves up one level, then again, and so on: **slowly walks across the whole tree**

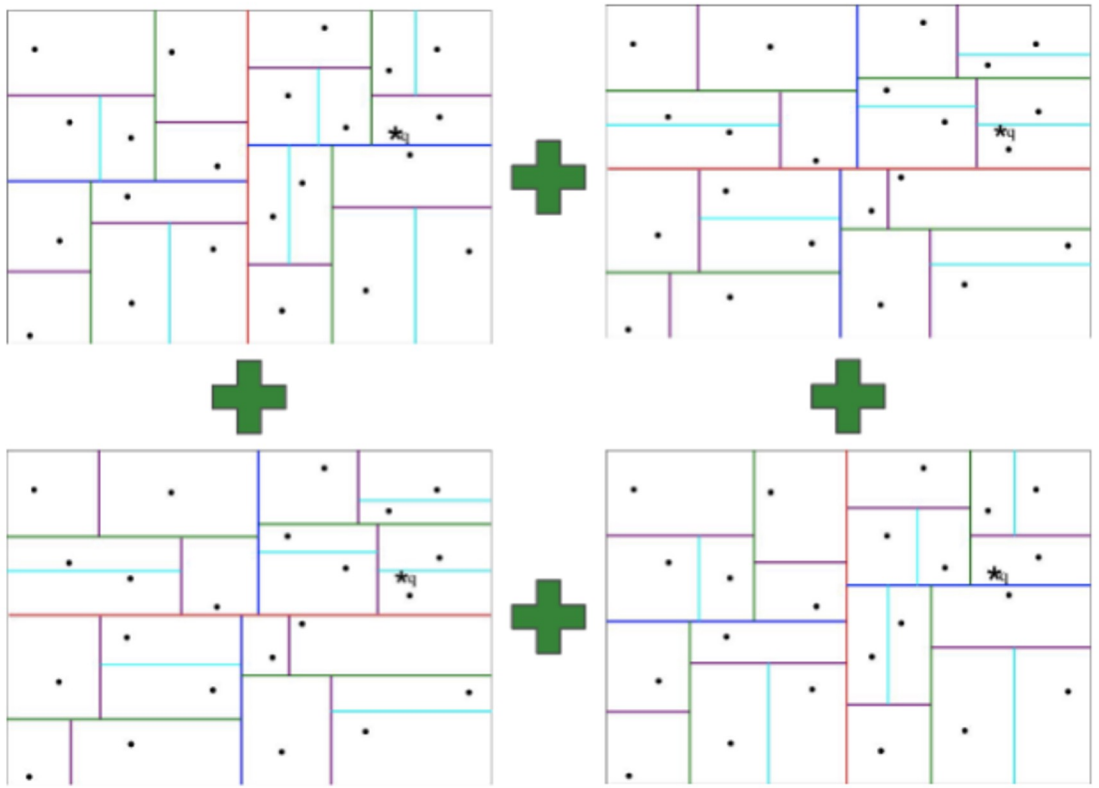


Q: How to avoid this problem, when the query point is near the boundary of a root cell?

Approximate Nearest Neighbor Search

FLANN

- Build several trees, each with a random order of axes for splitting
- In each tree, limit the backtracking depth, for example, just search with the leaf
- This gives an approximate solution in each tree, but the probability of missing the true nearest neighbor across all the trees is reduced

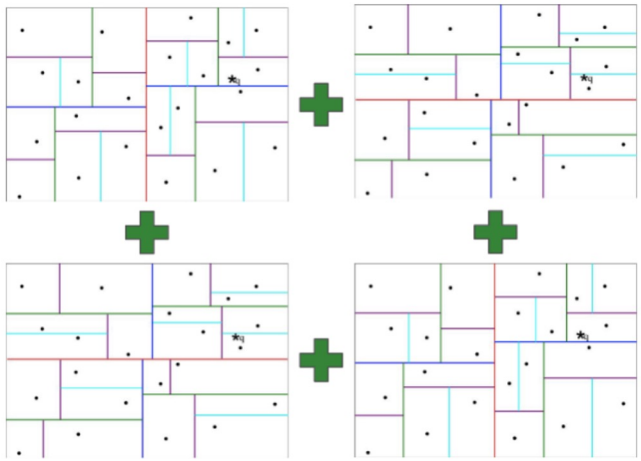


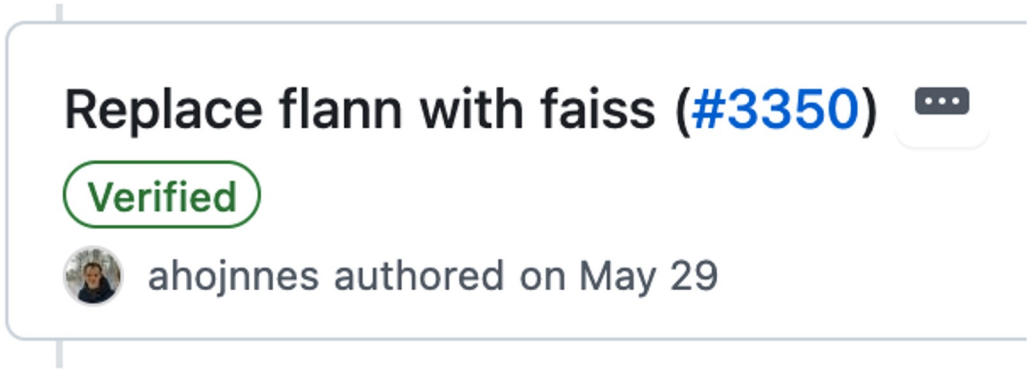
Approximate Nearest Neighbor Search

FLANN

This method is used in the FLANN library

Used in OpenCV, was used in COLMAP until recently





MATCHING

BRUTE-FORCE

Input:

- M images
- N keypoints on each image
- Each keypoint has a 128-dimensional SIFT descriptor

Output (for each pair of images A and B):

- **Nearest-Neighbor:** for each keypoint from A, find the most similar keypoint from B

Algorithmic complexity: $O(M^2 \cdot N^2)$

Q: How to speed up the first factor?

MATCHING

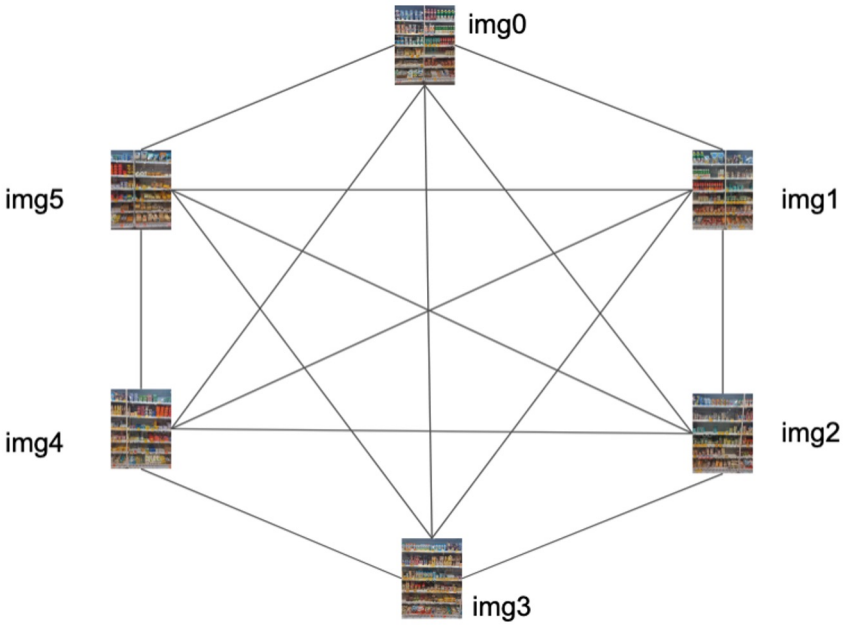
Preselection by Nearest Frames

If the images are sequential or proximity is known from GPS, we can only find correspondences between the images close on the timeline or in space

$$O(M^2N^2) \rightarrow O(MN^2)$$

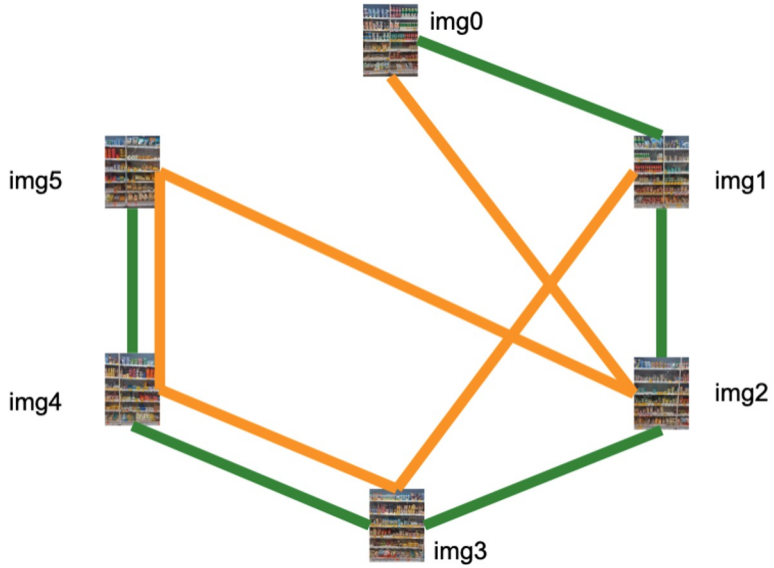
Preselection by Coarse matching

$O(M^2)$ pairs of matching with a small number of strong features ($N_{strong} = 1000$)



Preselection by Coarse matching

- $O(M^2)$ pairs of matching with a small number of strong features ($N_{strong} = 1000$)
- Find two maximum spanning trees based on the number of matches
- $O(M)$ pairs of matching with the full set of features ($N = 40000$)
- Total complexity (matching only):
  $O(M^2 N_{strong}^2) + O(MN^2)$

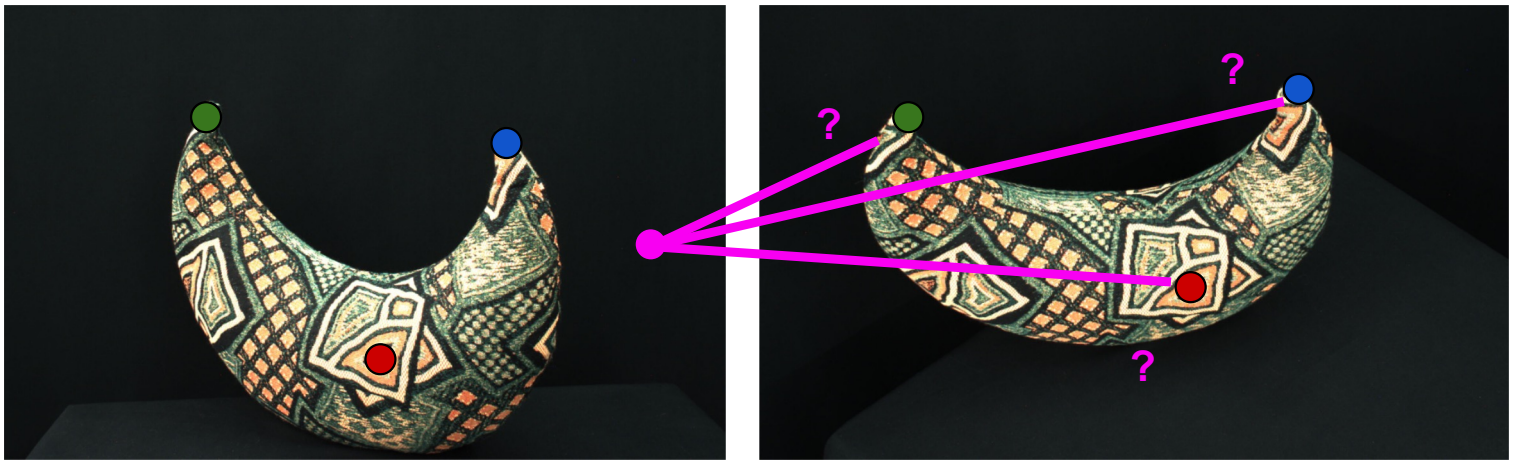


There are other options in COLMAP

- **Exhaustive Matching**: If the number of images in your dataset is relatively low (up to several hundreds), this matching mode should be fast enough and leads to the best reconstruction results. Here, every image is matched against every other image, while the block size determines how many images are loaded from disk into memory at the same time.
- **Sequential Matching**: This mode is useful if the images are acquired in sequential order, e.g., by a video camera. In this case, consecutive frames have visual overlap and there is no need to match all image pairs exhaustively. Instead, consecutively captured images are matched against each other. This matching mode has built-in loop detection based on a vocabulary tree, where every N-th image (*loop_detection_period*) is matched against its visually most similar images (*loop_detection_num_images*). Note that image file names must be ordered sequentially (e.g., *image0001.jpg*, *image0002.jpg*, etc.). The order in the database is not relevant, since the images are explicitly ordered according to their file names. Note that loop detection requires a pre-trained vocabulary tree. A default tree will be automatically downloaded and cached. More trees are available and can be downloaded from https://demuc.de/colmap/. In case rigs and frames are configured appropriately in the database, sequential matching will automatically match all images in consecutive frames against each other.
- **Vocabulary Tree Matching**: In this matching mode [schoenberger16vote], every image is matched against its visual nearest neighbors using a vocabulary tree with spatial re-ranking. This is the recommended matching mode for large image collections (several thousands). This requires a pre-trained vocabulary tree, that can be downloaded from https://demuc.de/colmap/.
- **Spatial Matching**: This matching mode matches every image against its spatial nearest neighbors. Spatial locations can be manually set in the database management. By default, COLMAP also extracts GPS information from EXIF and uses it for spatial nearest neighbor search. If accurate prior location information is available, this is the recommended matching mode.
- **Transitive Matching**: This matching mode uses the transitive relations of already existing feature matches to produce a more complete matching graph. If an image A matches to an image B and B matches to C, then this matcher attempts to match A to C directly.
- **Custom Matching**: This mode allows to specify individual image pairs for matching or to import individual feature matches. To specify image pairs, you have to provide a text file with one image pair per line:

MATCHING

FILTERING



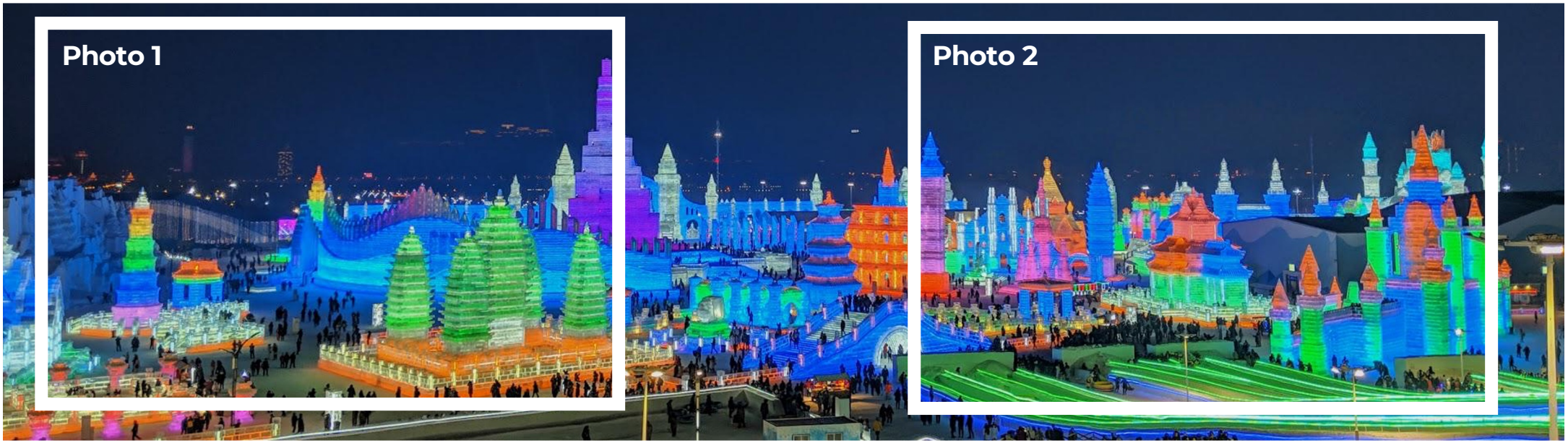
We learned how to accelerate matching, let's talk about making matching robust to errors in feature extraction

Until now, we searched for matches for all keypoints, but some keypoints may not have a good match

Q: What are examples when we will have a lot of false matches?

MATCHING

FALSE CORRESPONDENCES



Photos capture two completely different parts of the scene — all matches will be false, even for good features

It is necessary to be able to filter out false matches

FILTERING

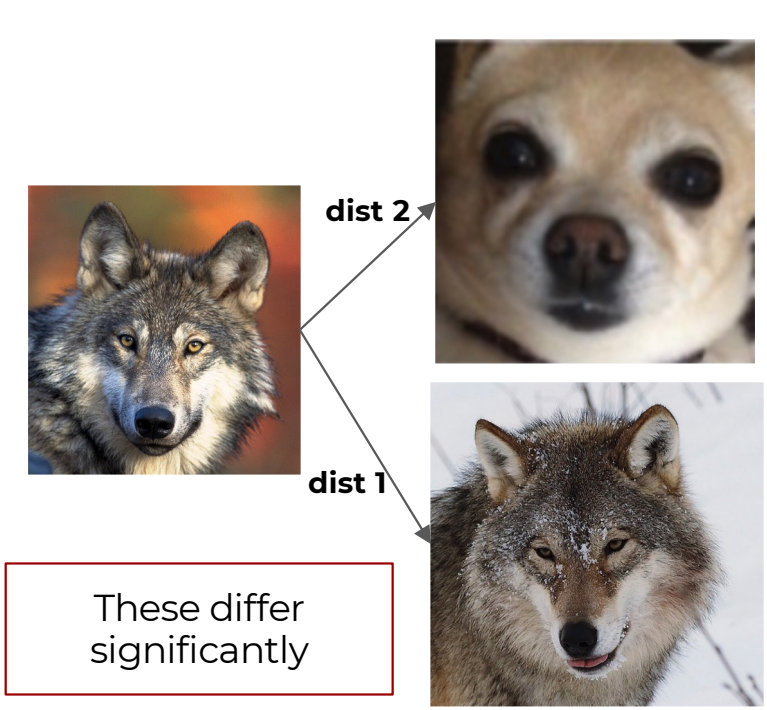
Suppose there are two candidate matches

Q: How do the distances to them differ when only one of them is incorrect and when both are incorrect?

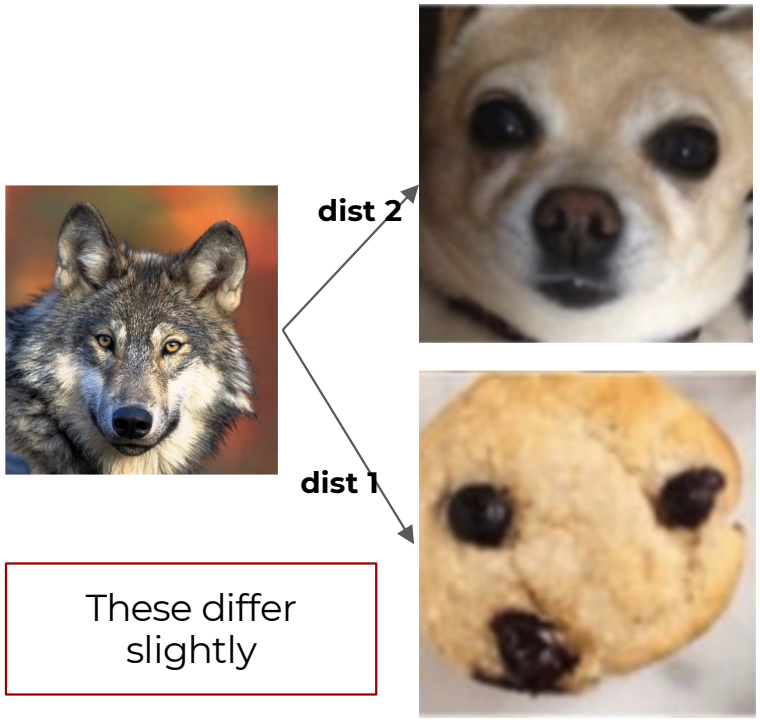
FILTERING MATCHES

K-Ratio Test / Lowe's Ratio Test

1. We search not only for the nearest neighbor ($dist_1$), but also for the second closest ($dist_2$).
2. If the difference is small ($dist_1 > dist_2 \times 0.8$), there is a risk of error (ambiguity).
3. If the difference is large ($dist_1 < dist_2 \times 0.8$), the match is reliable.

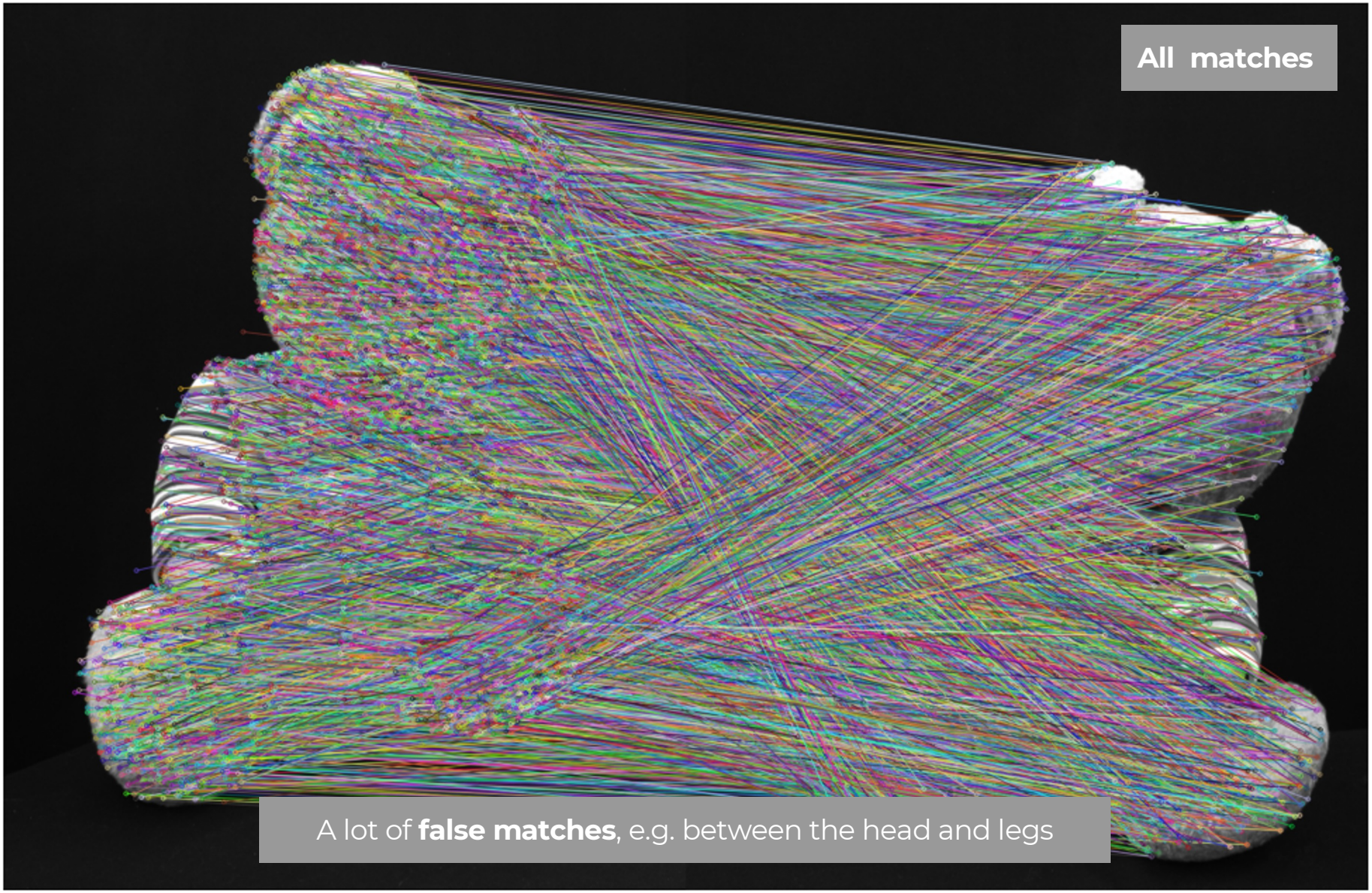


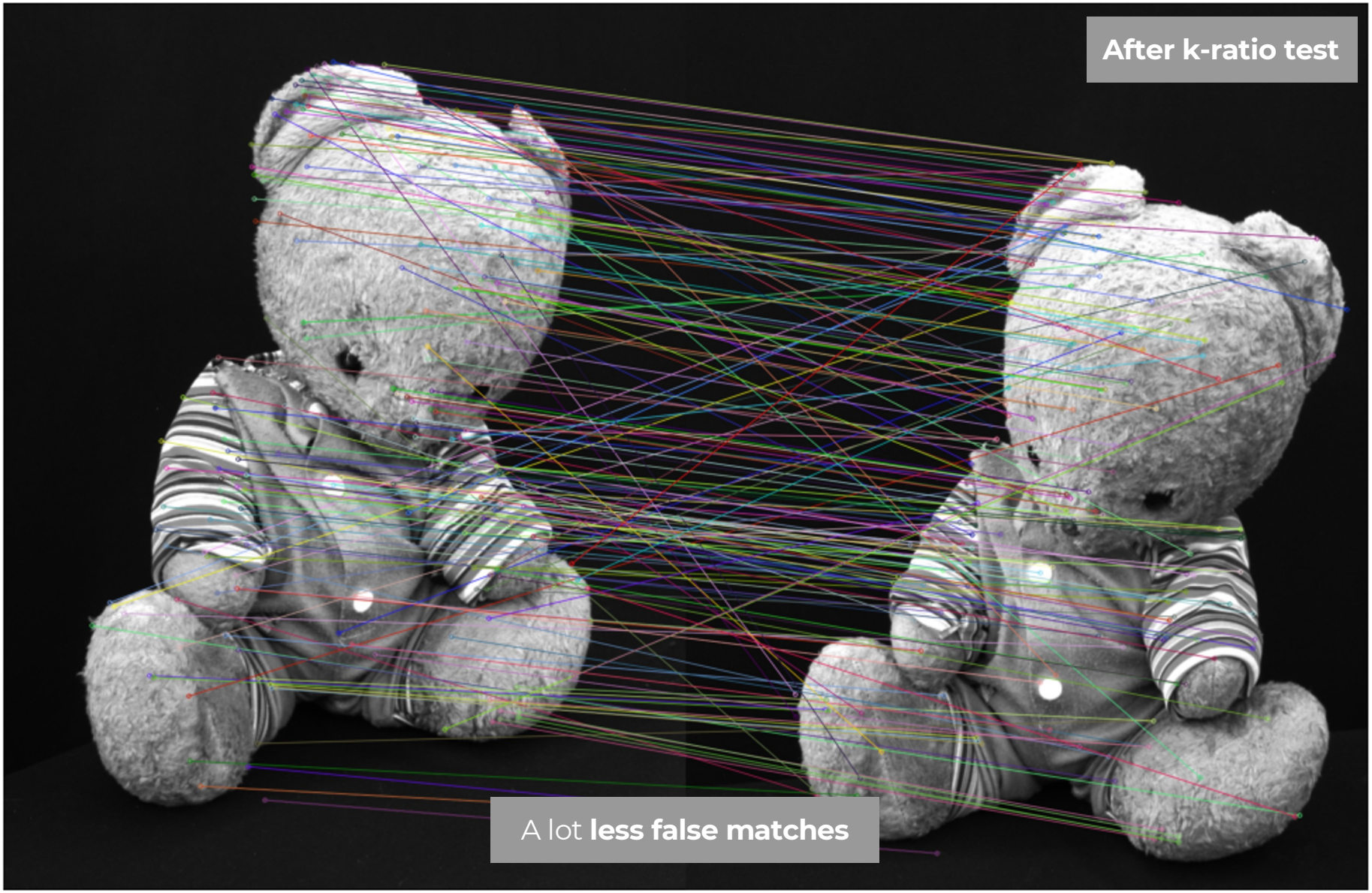
These differ significantly

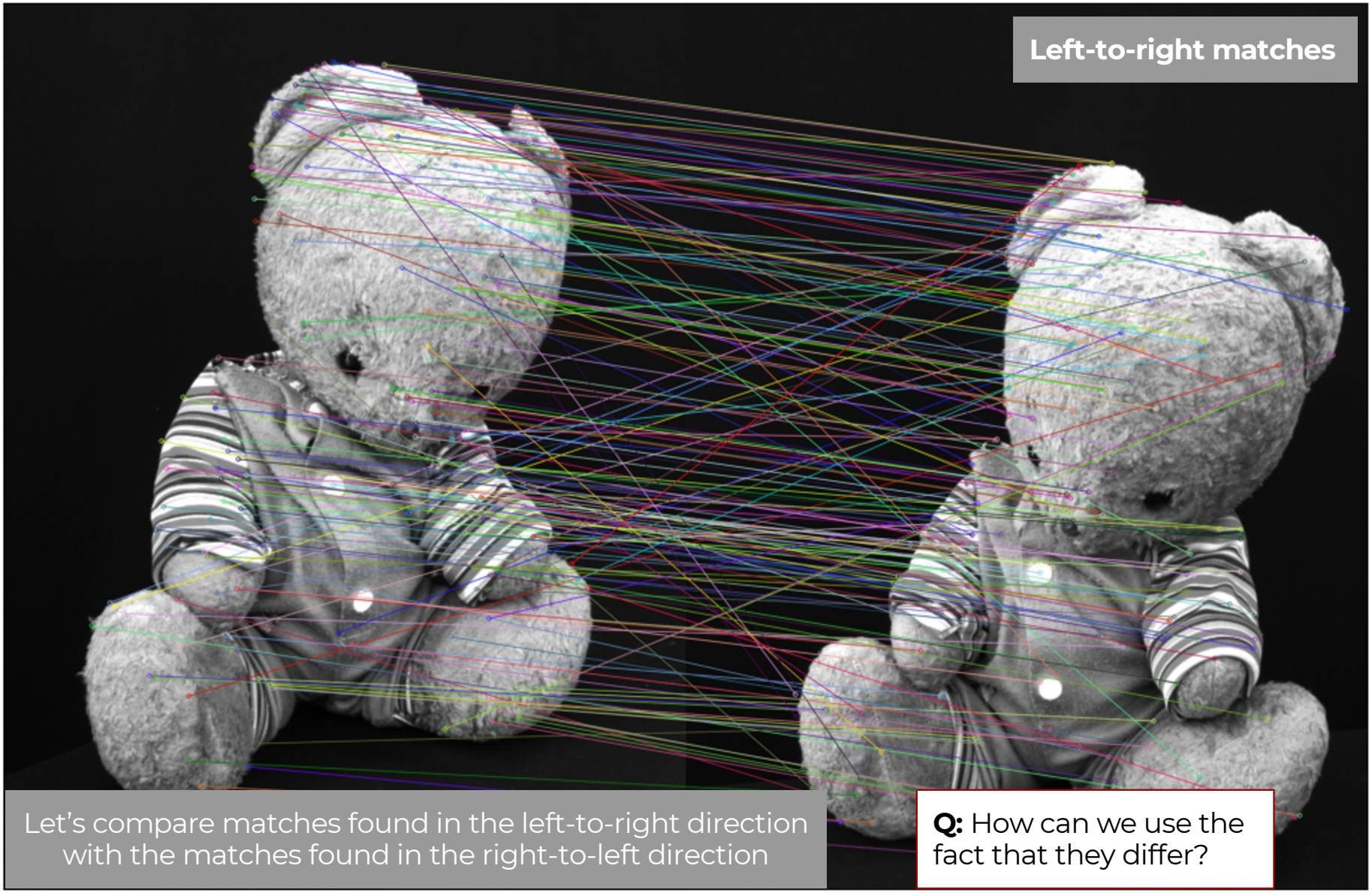


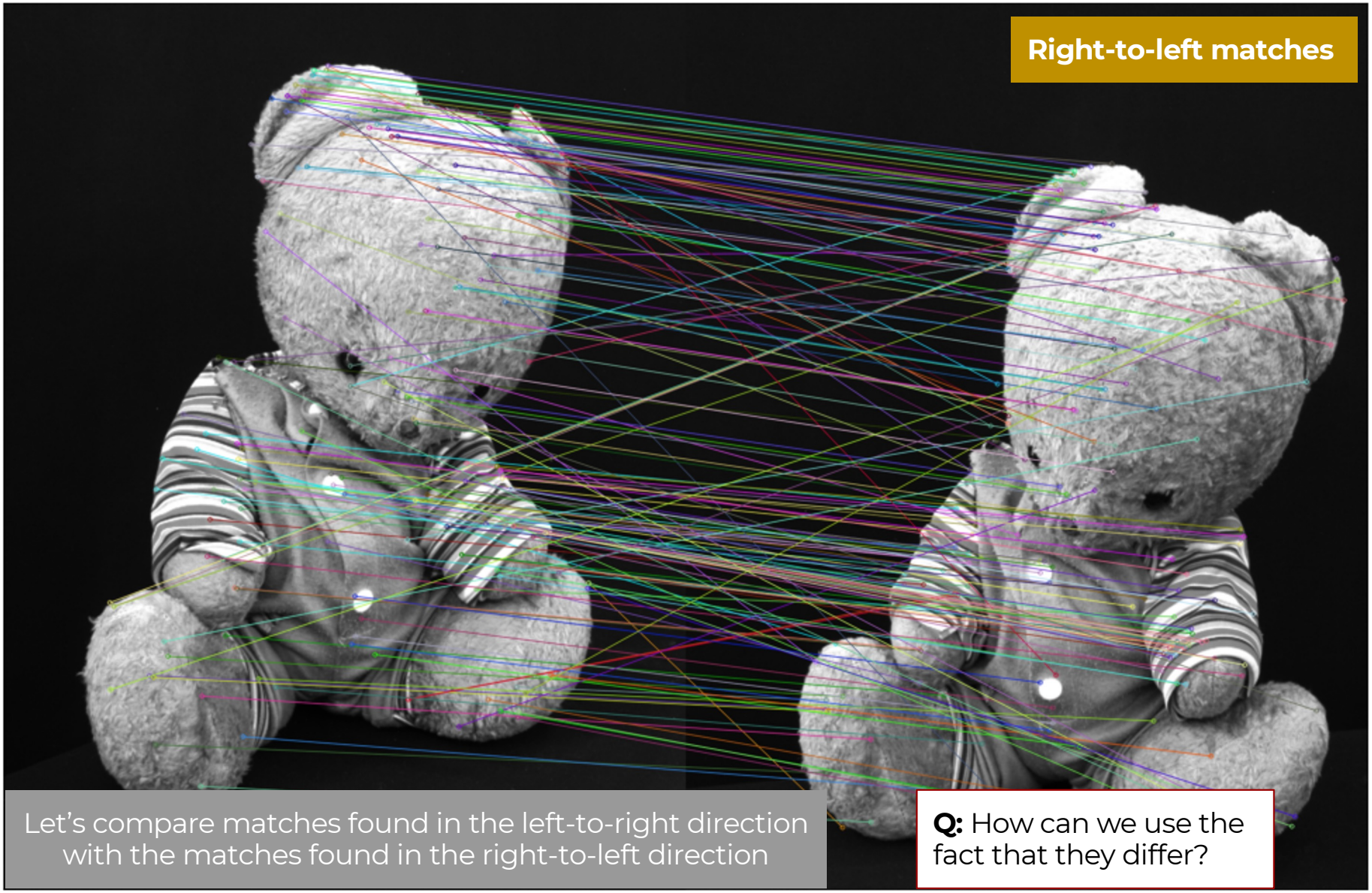
These differ slightly

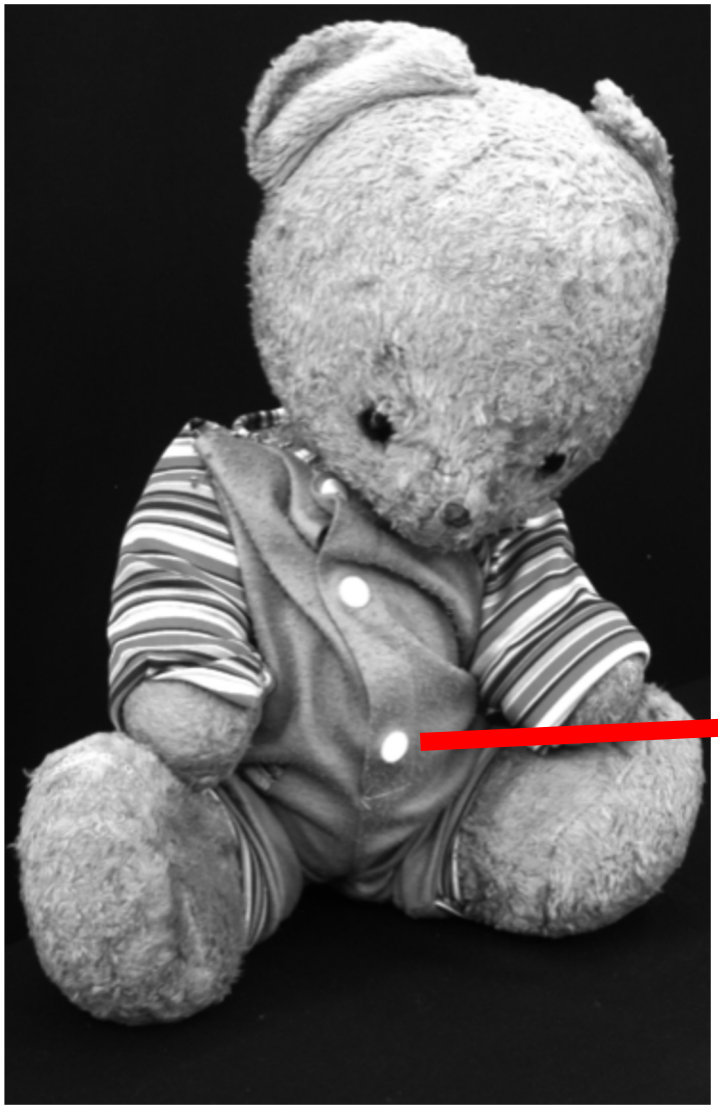

Left photo
Right photo
Let's match keypoints in these two photos


All matches
A lot of **false matches**, e.g. between the head and legs

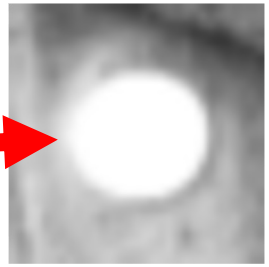

After k-ratio test
A lot **less false matches**


Left-to-right matches
Let’s compare matches found in the left-to-right direction with the matches found in the right-to-left direction
Q: How can we use the fact that they differ?

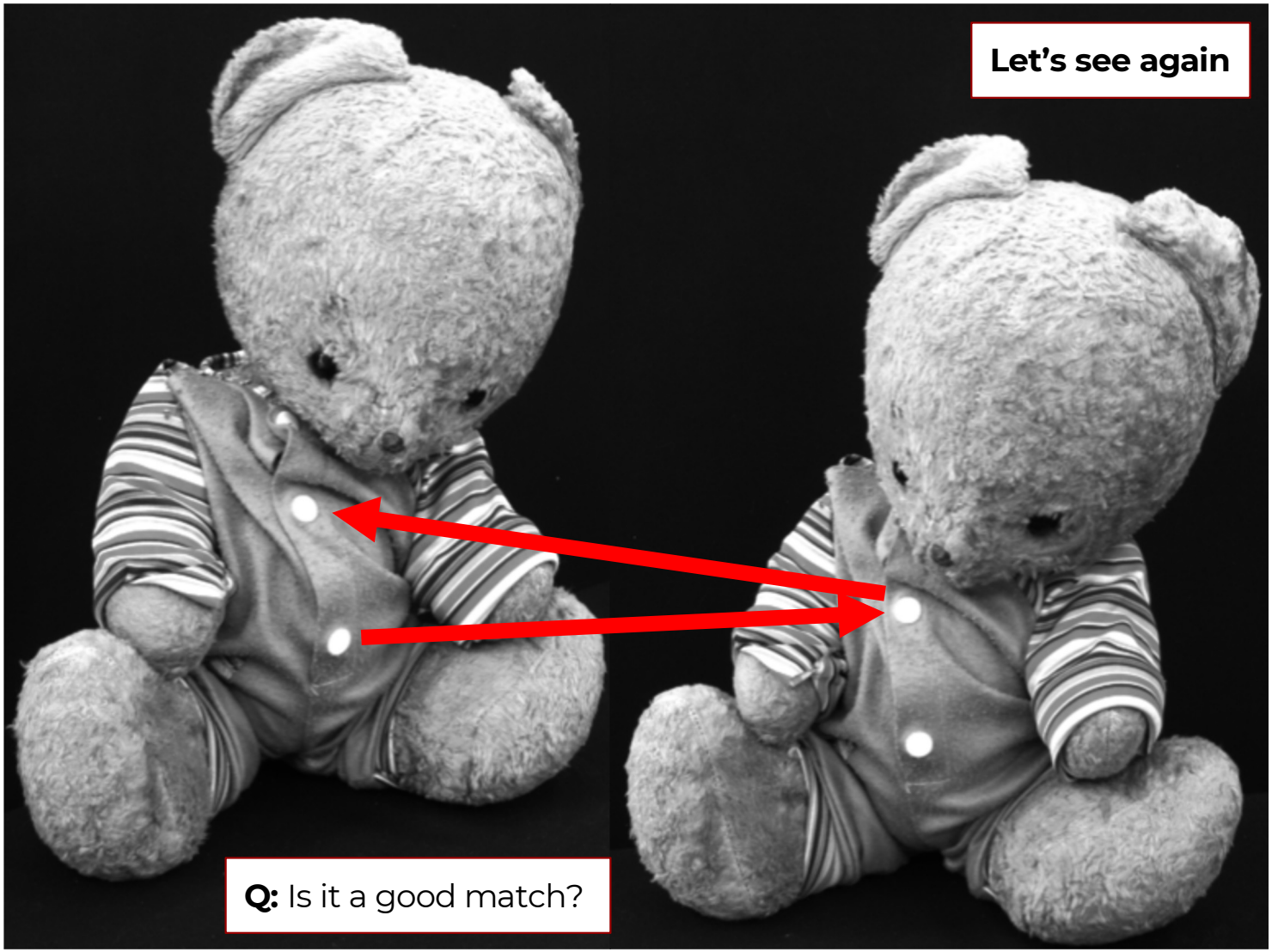

Right-to-left matches
Let’s compare matches found in the left-to-right direction with the matches found in the right-to-left direction
Q: How can we use the fact that they differ?



We find a match



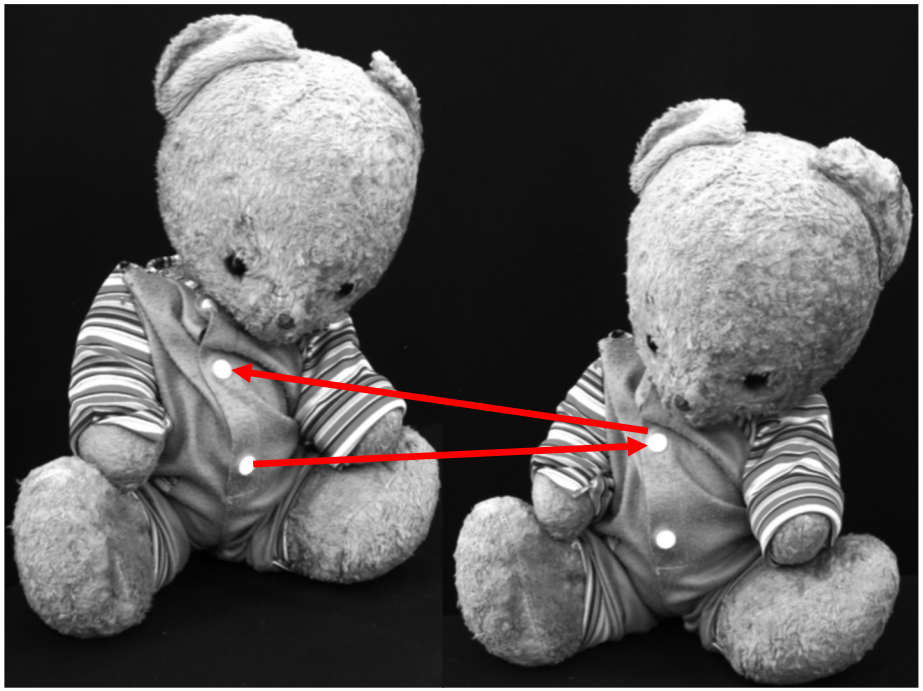
Q: Is it a good match?

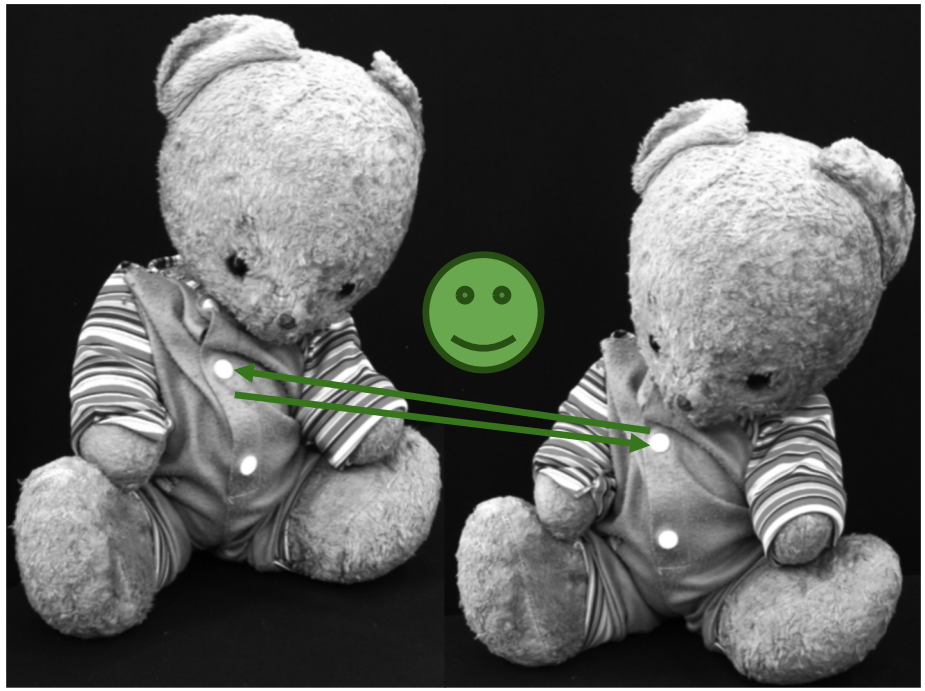

Let’s see again
Q: Is it a good match?

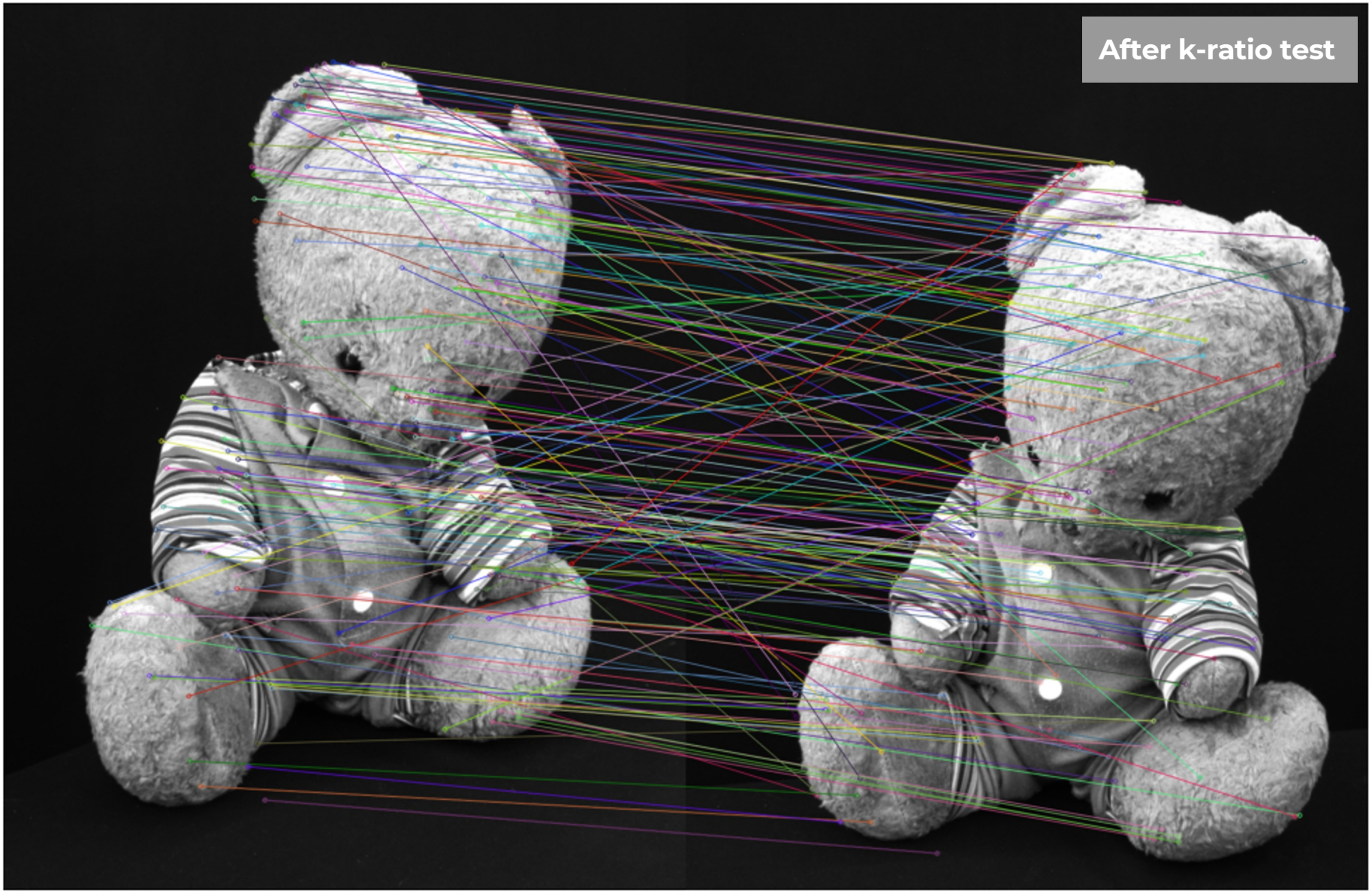
FILTERING MATCHES

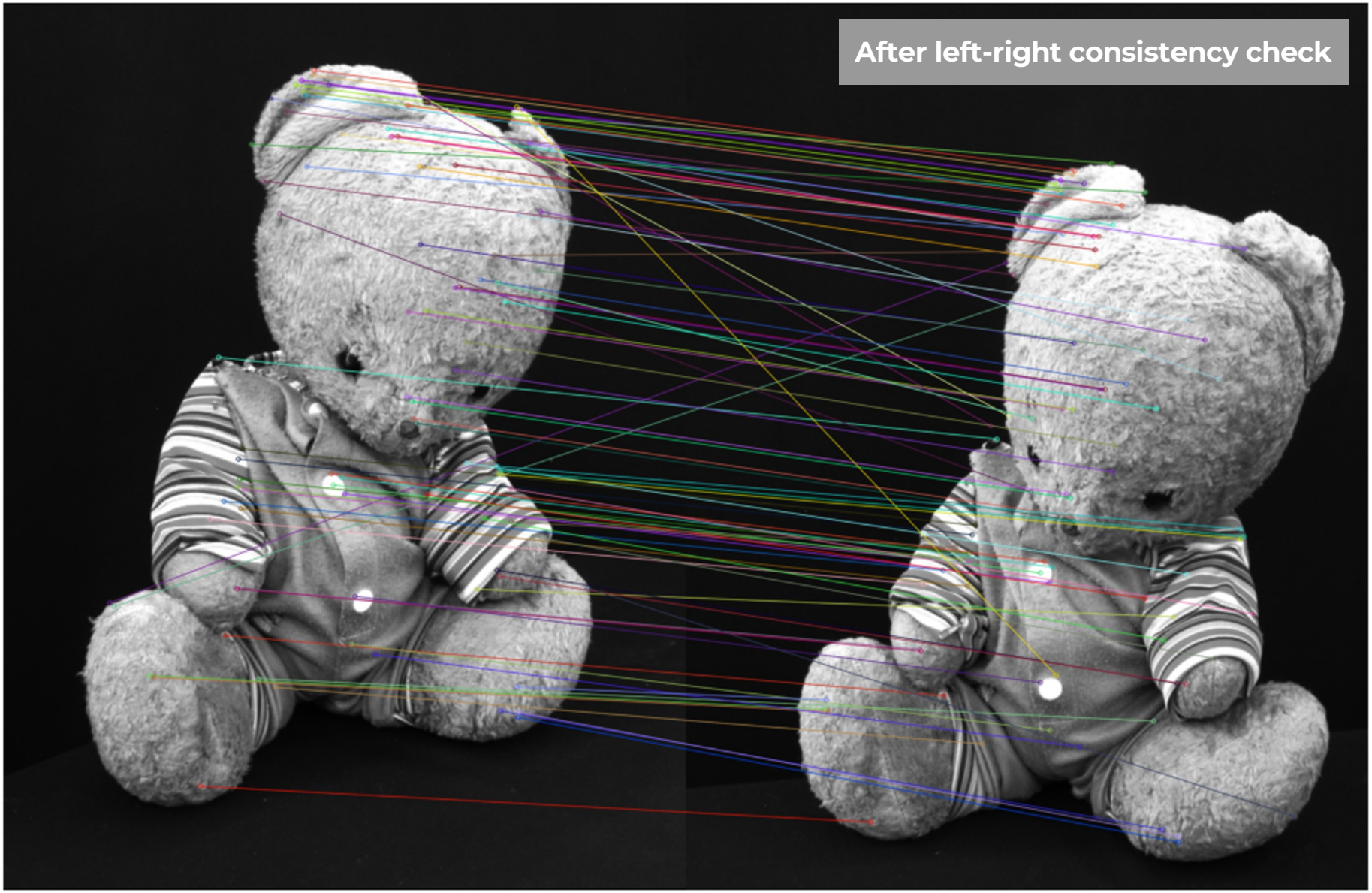
Left-Right Check

We match from left to right and from right to left, and keep only the matches consistent with both directions



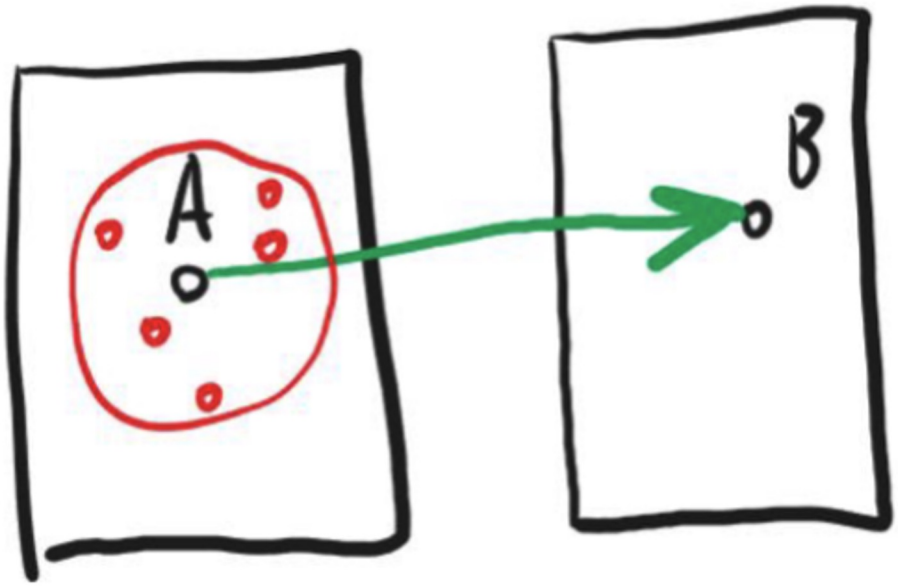



After k-ratio test


After left-right consistency check

MATCHING

FILTERING

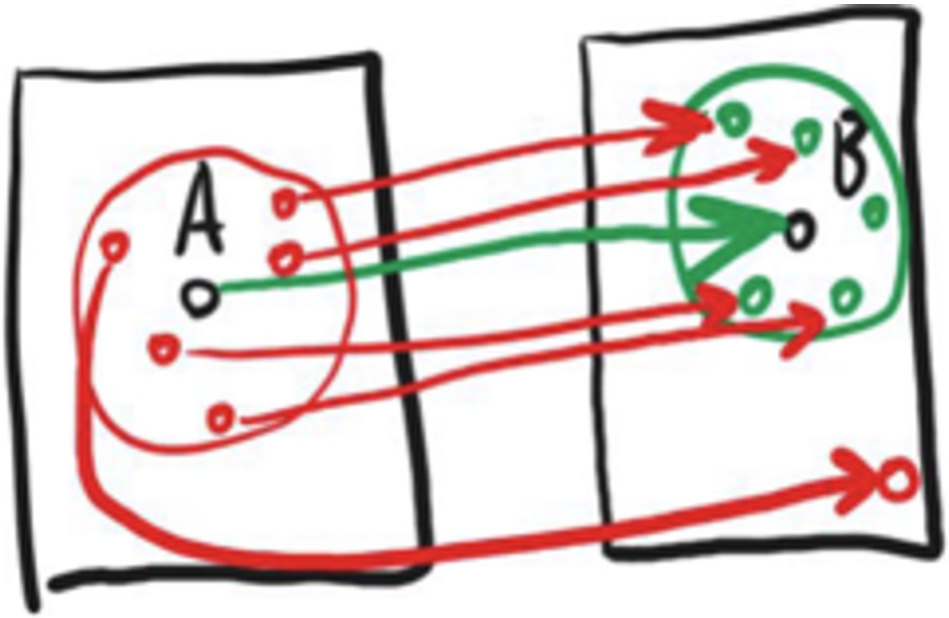


Q: Point B matches point A. Where can we expect the matches for the points around A to located?

FILTERING MATCHES

Cluster Filtering

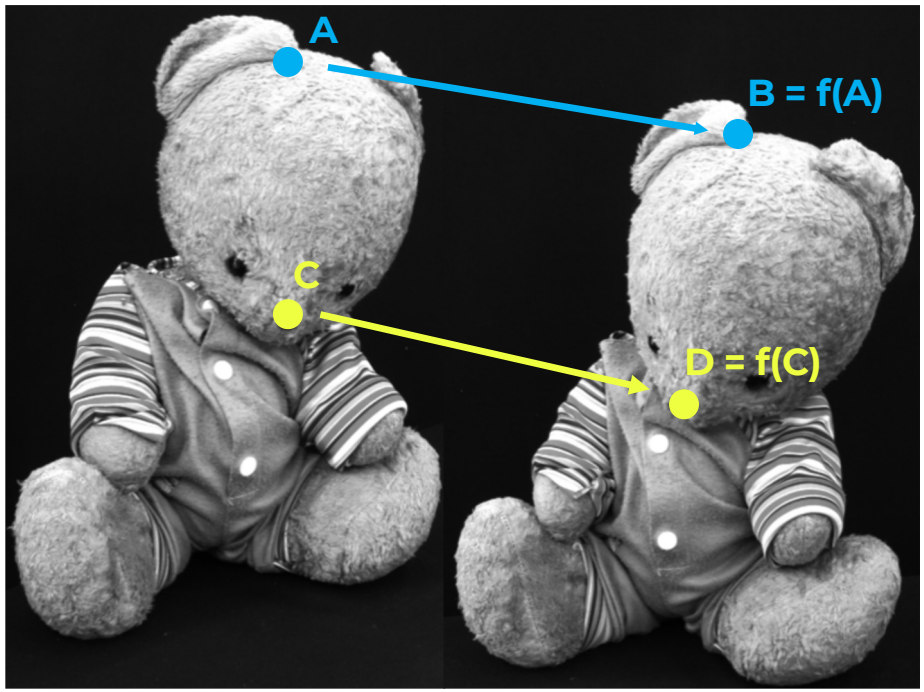
Keep the match if, e.g., more than half of points from the neighborhood “move” across the views accordingly



Geometric consistency

Let's go further and expect some global geometric consistency of good matches.

I.e., the coordinates of the matching points in the two images are related with some transformation **f**, same for all matches.

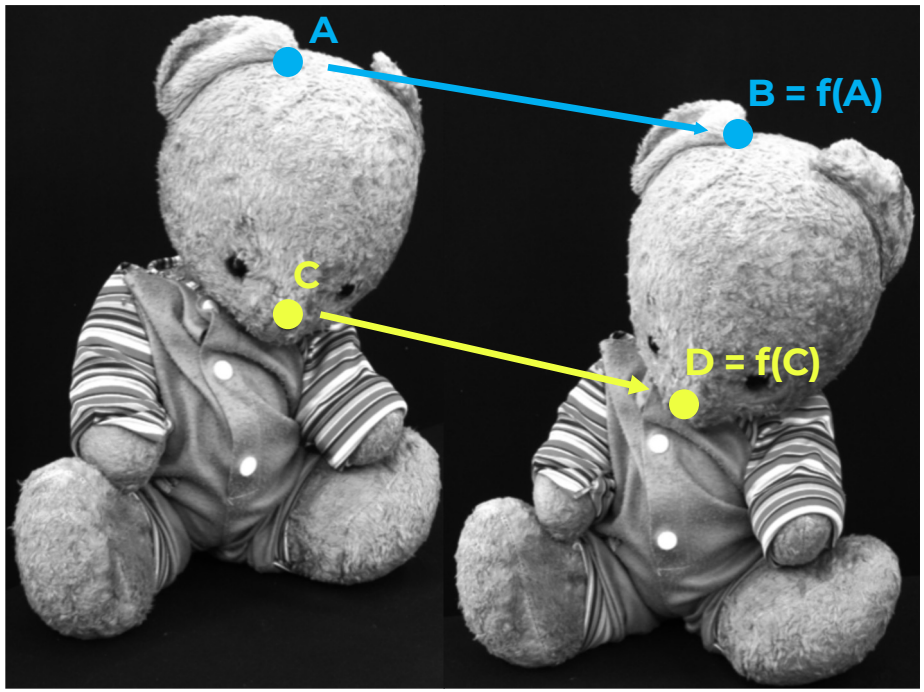


Q: If we know **f**, how can we use it to filter out bad matches?

Geometric consistency

Let's go further and expect some global geometric consistency of good matches.

I.e., the coordinates of the matching points in the two images are related with some transformation **f**, same for all matches.



Q: If we know **f**, how can we use it to filter out bad matches?

Check matches for consistency with **f**, e.g., reject matches where D != f(C).

But where to get **f** from?
Assume some parametric model for **f** and find using good matches — but need to find good matches first.

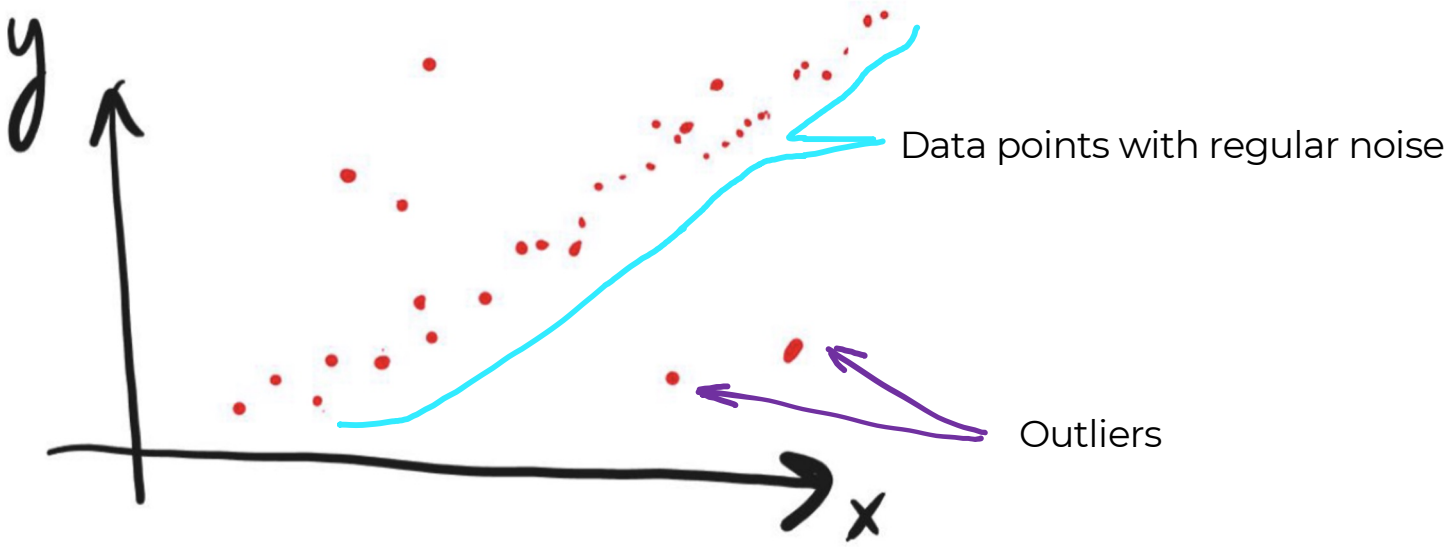
Chicken and egg problem.

FILTERING MATCHES

RANSAC

Example: RANSAC for fitting a line to noisy points with outliers.

Goal: find a line corresponding the largest number of points.

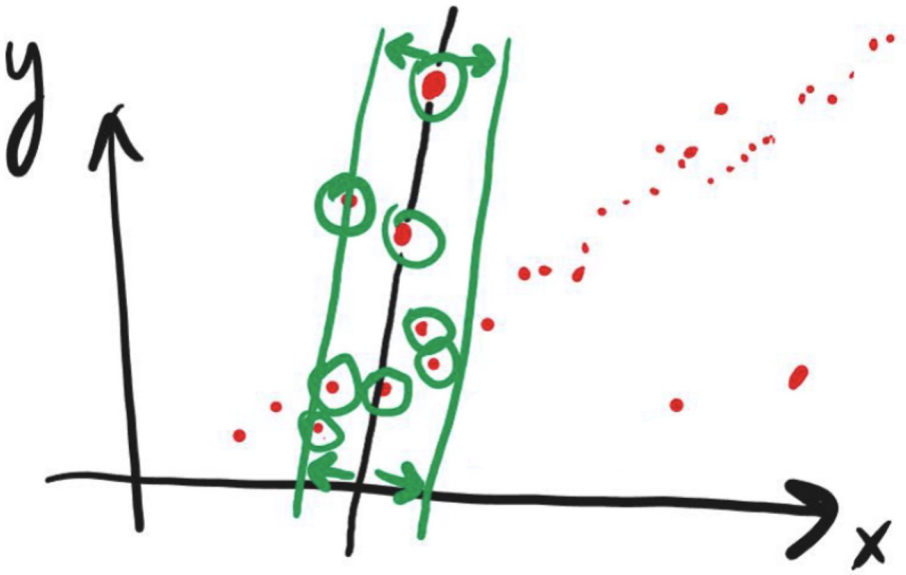


FILTERING MATCHES

RANSAC

Example: RANSAC for fitting a line to noisy points with outliers.

Goal: find a line corresponding the largest number of points.



Algorithm:

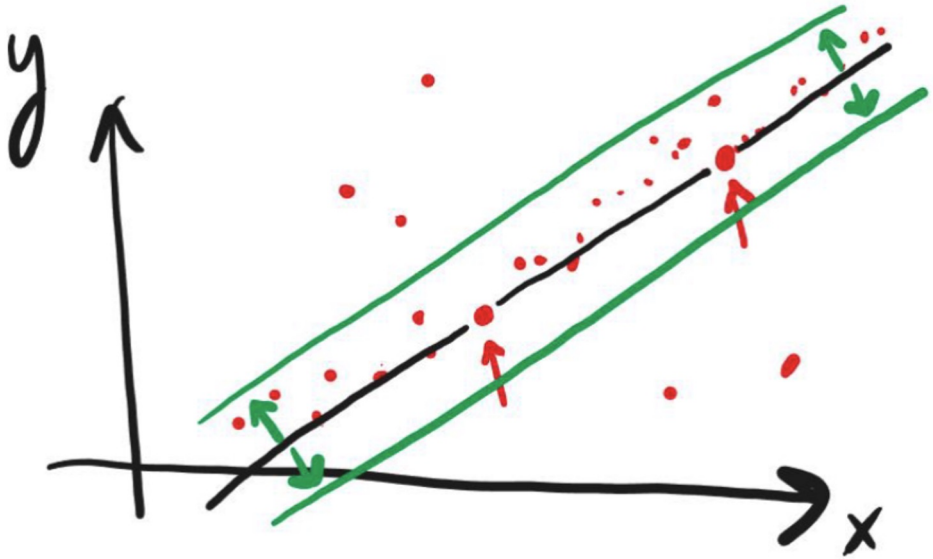
1. Draw a line by a random pair of points
2. Count the number of points corresponding to this line, e.g., that are within a certain threshold. These are called **inliers**

FILTERING MATCHES

RANSAC

Example: RANSAC for fitting a line to noisy points with outliers.

Goal: find a line corresponding the largest number of points.



Algorithm:

1. Draw a line by a random pair of points
2. Count the number of points corresponding to this line, e.g., that are within a certain threshold. These are called **inliers**
3. Repeat from step 1. If now we have more inliers — keep the new line as the best solution so far
4. Repeat until we find a good enough solution or for a certain number of iterations

FILTERING MATCHES

RANSAC

How to apply for fitting a parabola to noisy points with outliers?

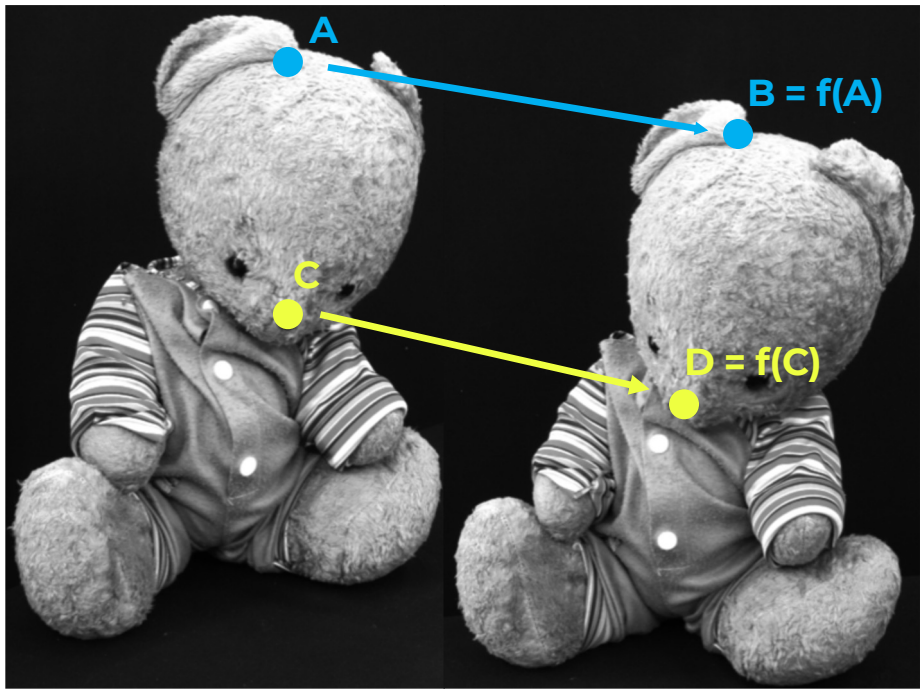
What solution will RANSCAC find if the data comes from two parabolas?

After we found one parabola, how can we find the second one using RANSAC?

FILTERING MATCHES

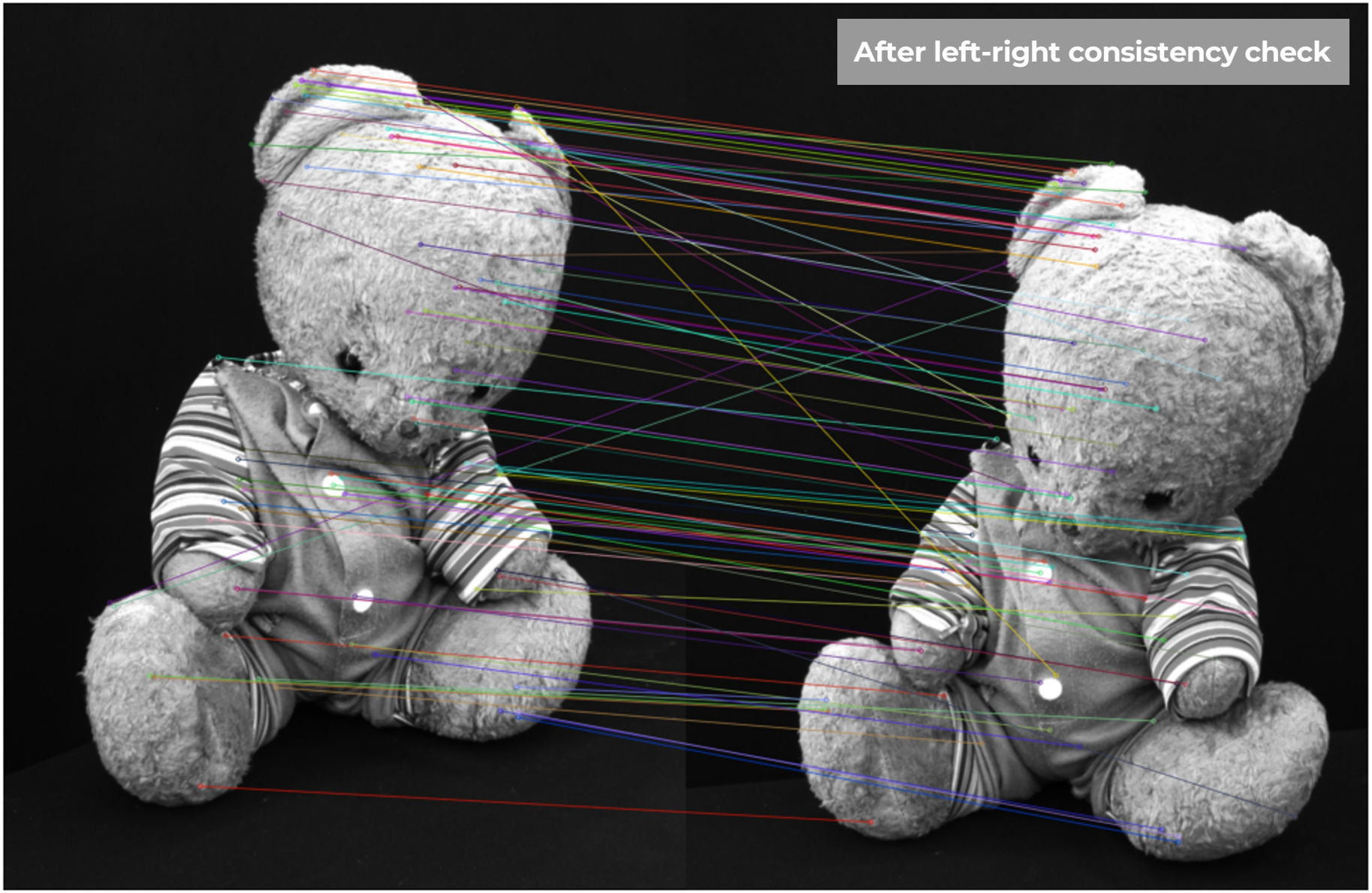
Geometric consistency using RANSAC

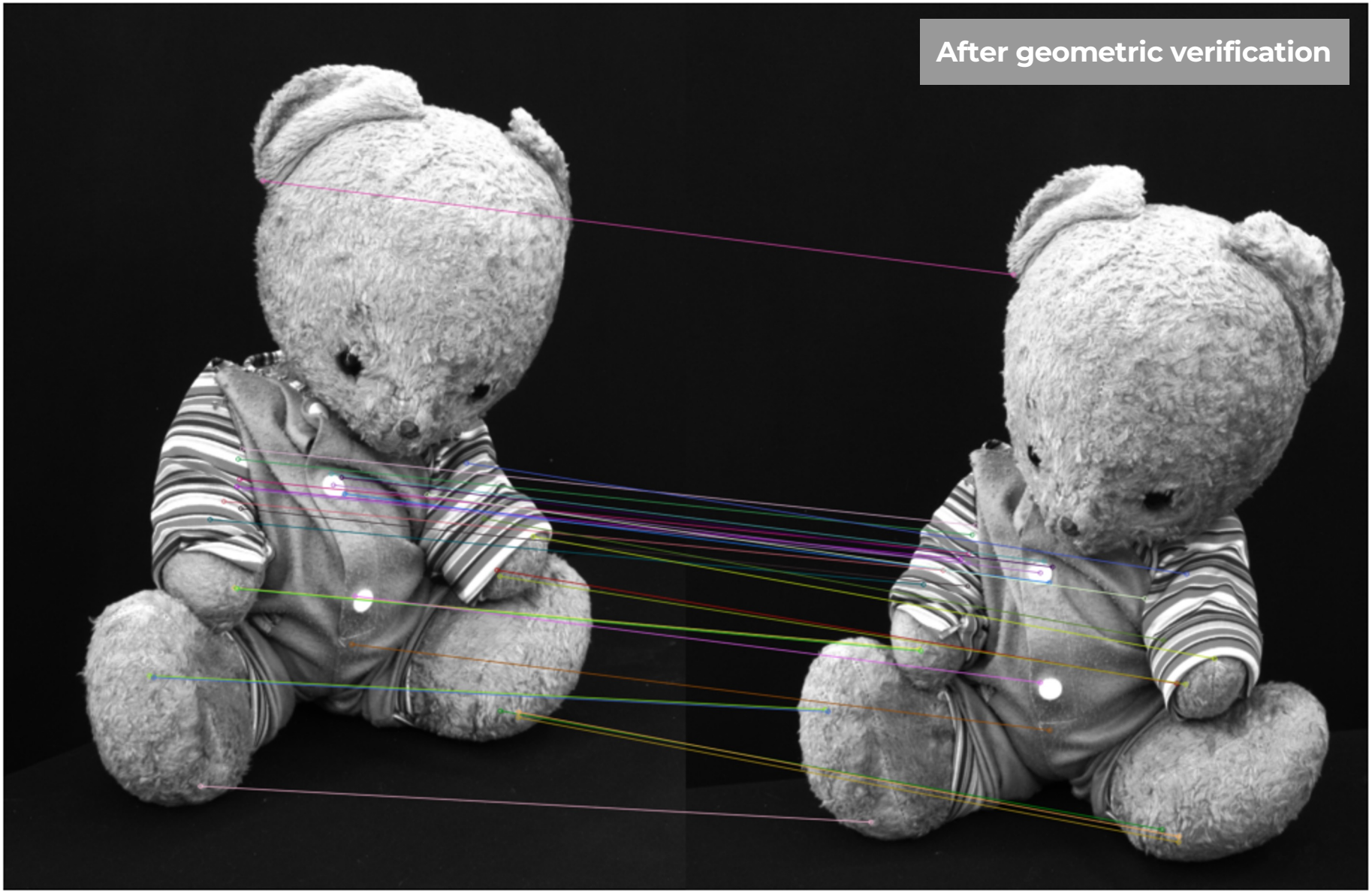
How to apply this for filtration of matches using geometric consistency check?



Algorithm:

1. Fit **f** to a random set of matches
2. Count the number of inlier matches corresponding to this **f**
3. Repeat from step 1 while keeping the **f** with the largest number of corresponding matches as the best solution so far
4. Repeat until we find a good enough solution or for a certain number of iterations


After left-right consistency check


After geometric verification

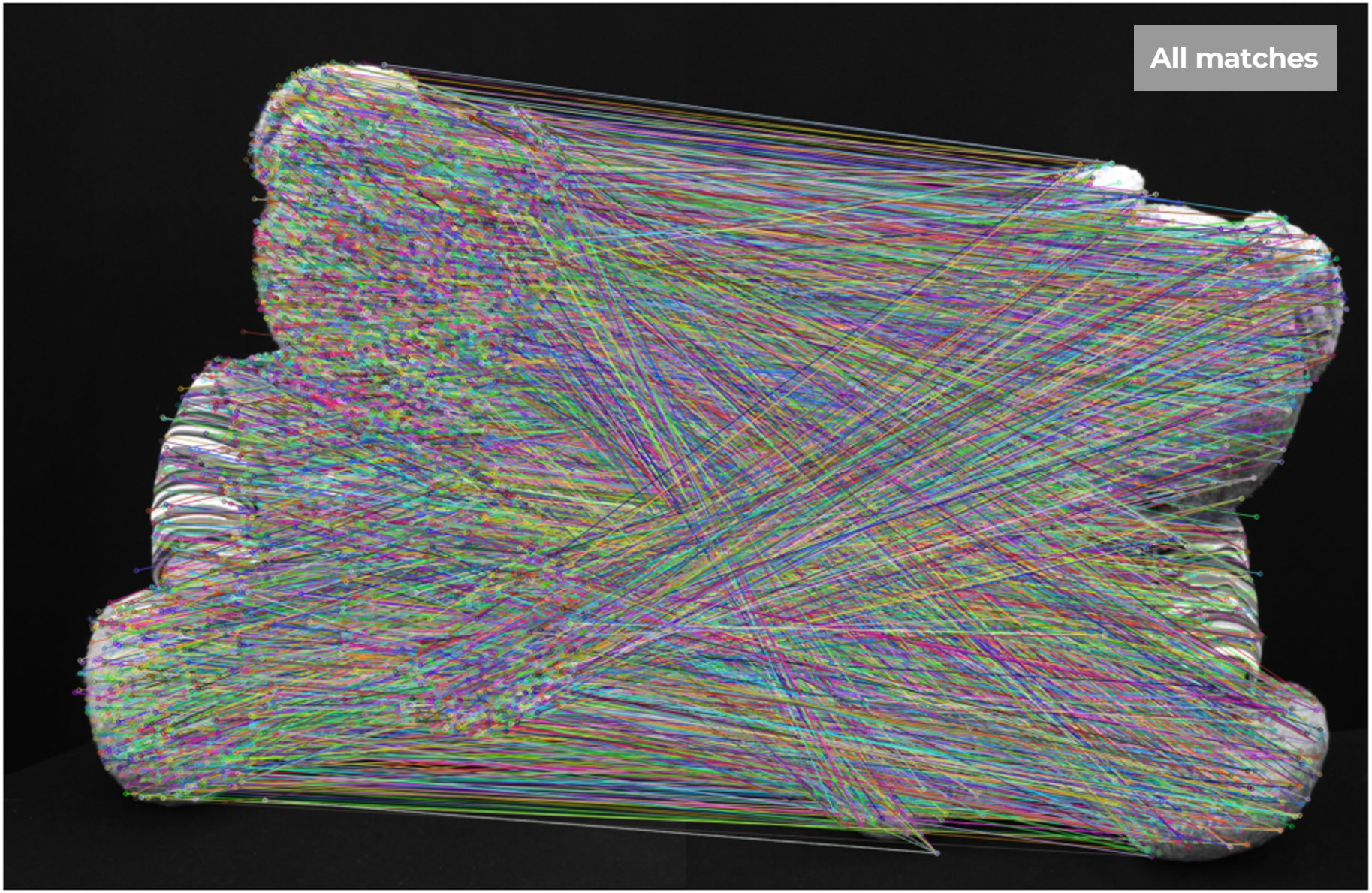
SUMMARY

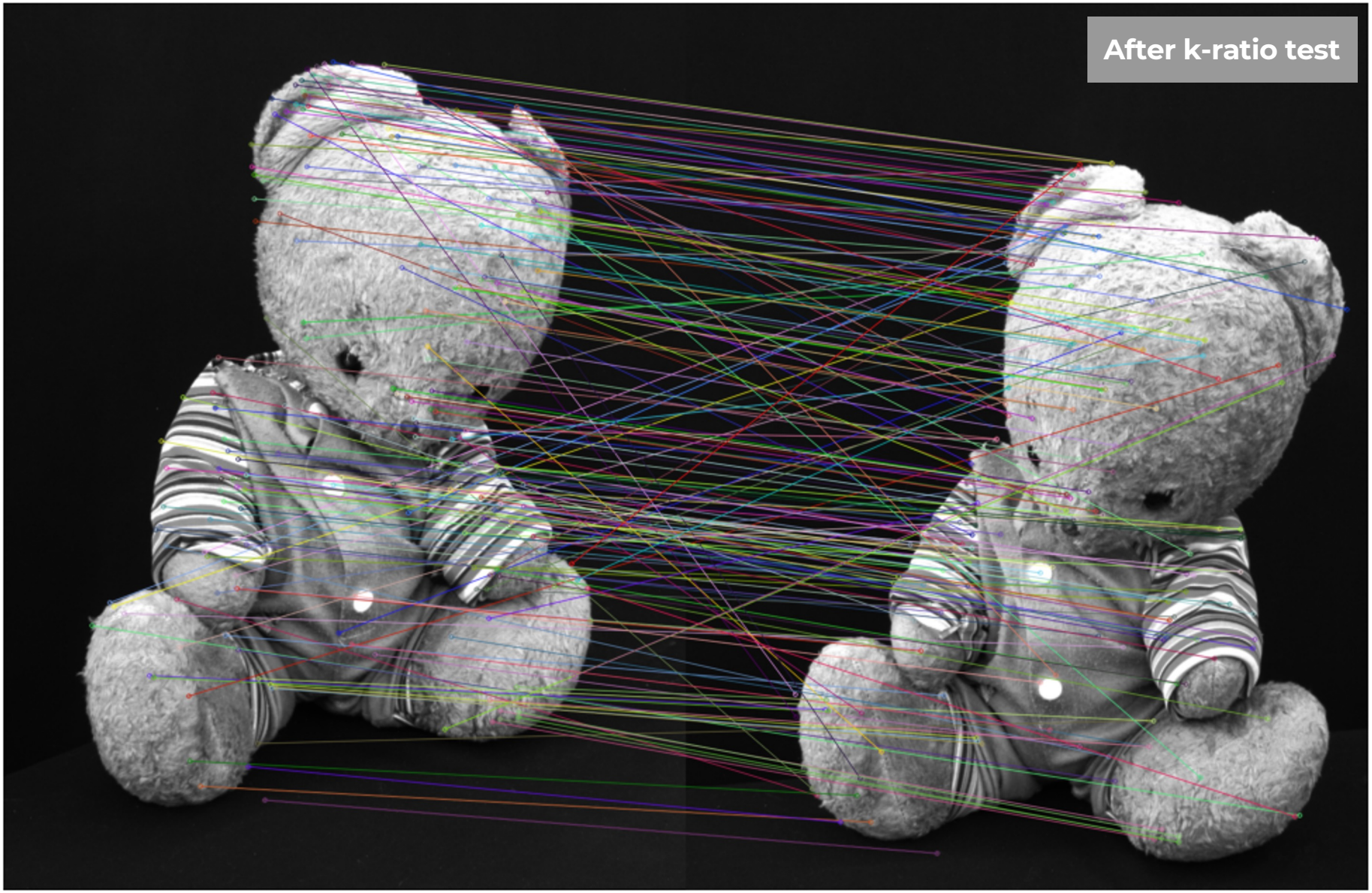
Making feature matching faster:

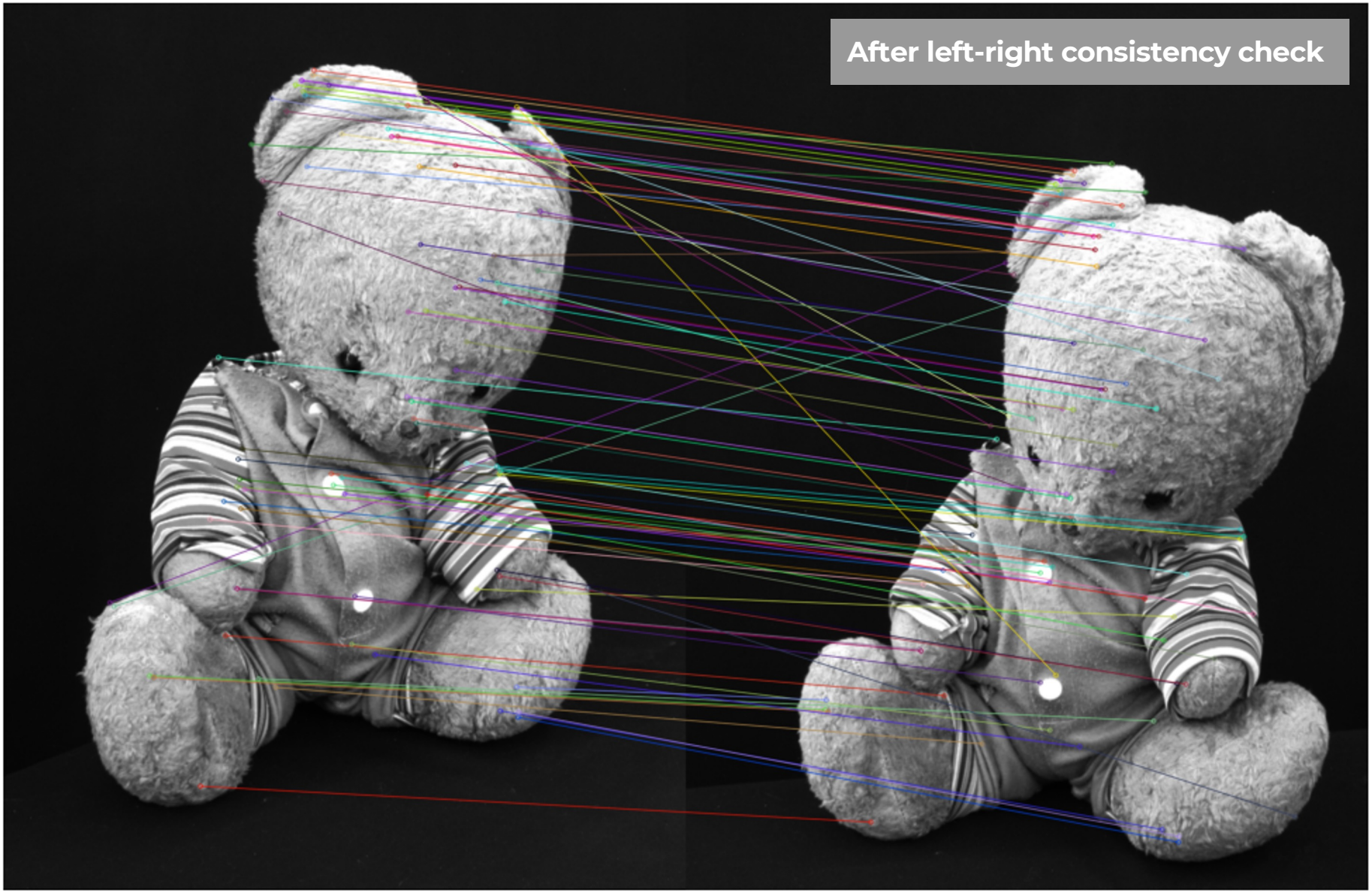
- KD-Tree for fast nearest neighbor search (on average)
- FLANN for fast approximate nearest neighbor (in the worst case)
- Approaches for preselection of image pairs to avoid unnecessary exhaustive matching

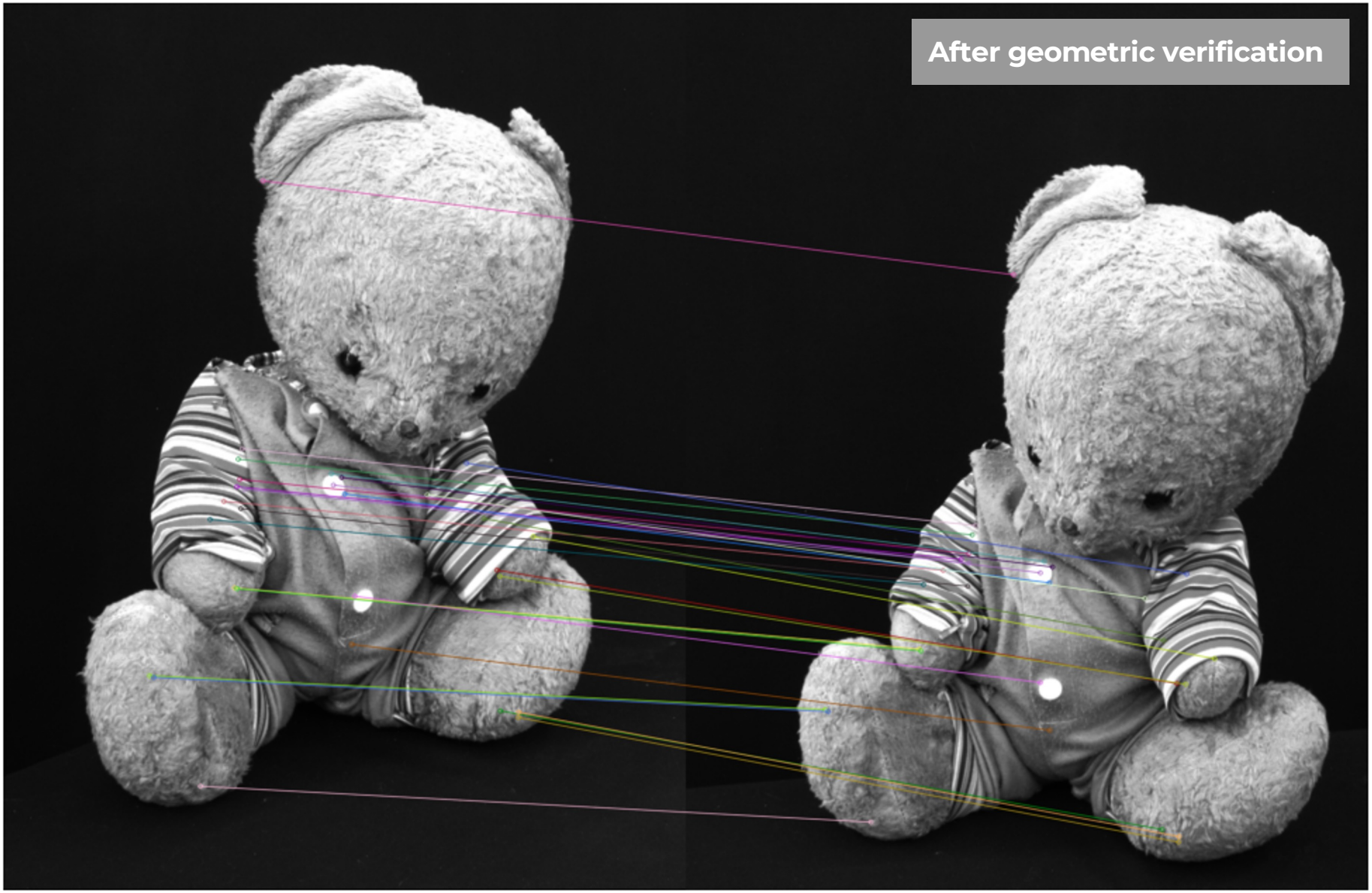
Making feature matching more robust, i.e., filtration of mismatches:

- K-ratio (Lowe's ratio) test for discarding non-confident matches, with no distinctive nearest neighbor
- Left-right consistency check for discarding nonsymmetric matches
- Cluster filtration and Geometric verification for only keeping the matches consistent with some transformation


All matches


After k-ratio test


After left-right consistency check


After geometric verification