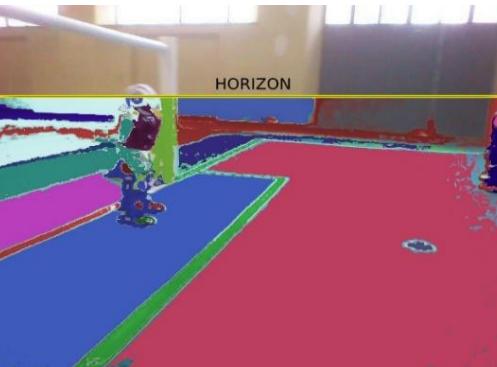




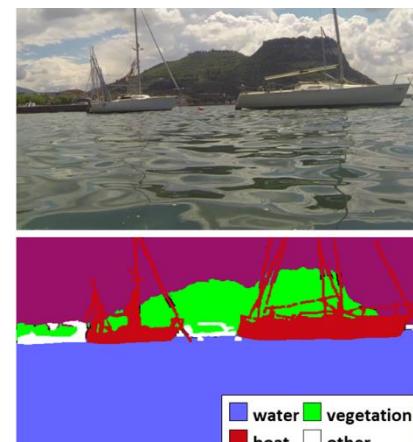
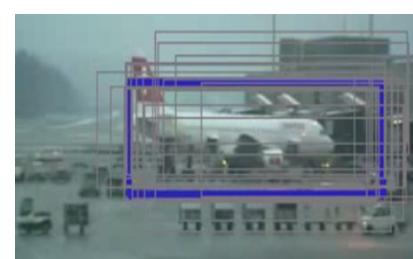
UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA

# *Corso di Visione e Percezione*

# Feature Matching



Docente  
Domenico D. Bloisi



# Domenico Daniele Bloisi

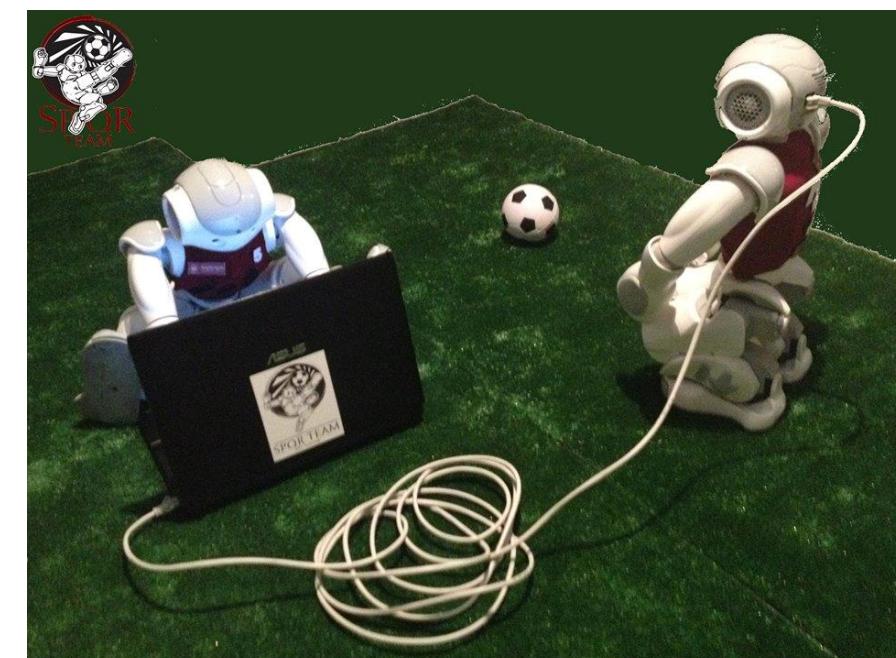
---

- Professore Associato  
Dipartimento di Matematica, Informatica  
ed Economia  
Università degli studi della Basilicata

<http://web.unibas.it/bloisi>

- SPQR Robot Soccer Team  
Dipartimento di Informatica, Automatica  
e Gestionale Università degli studi di  
Roma “La Sapienza”

<http://spqr.diag.uniroma1.it>



# UNIBAS Wolves <https://sites.google.com/unibas.it/wolves>

---



- UNIBAS WOLVES is the robot soccer team of the University of Basilicata. Established in 2019, it is focussed on developing software for NAO soccer robots participating in RoboCup competitions.
- UNIBAS WOLVES team is twinned with [SPQR Team](#) at Sapienza University of Rome.



# Informazioni sul corso

---

- Home page del corso:  
<https://web.unibas.it/bloisi/corsi/visione-e-percezione.html>
- Docente: Domenico Daniele Bloisi
- Periodo: Il semestre marzo 2022 – giugno 2022
  - Martedì dalle 15:00 alle 17:00 (Aula Copernico)
  - Mercoledì dalle 8:30 alle 10:30 (Aula Copernico)

# Ricevimento

---

- Durante il periodo delle lezioni:

Mercoledì dalle 11:00 alle 12:30 → Edificio 3D, II piano, stanza 15

Si invitano gli studenti a controllare regolarmente la bacheca degli avvisi per eventuali variazioni

- Al di fuori del periodo delle lezioni:

da concordare con il docente tramite email

Per prenotare un appuntamento inviare  
una email a

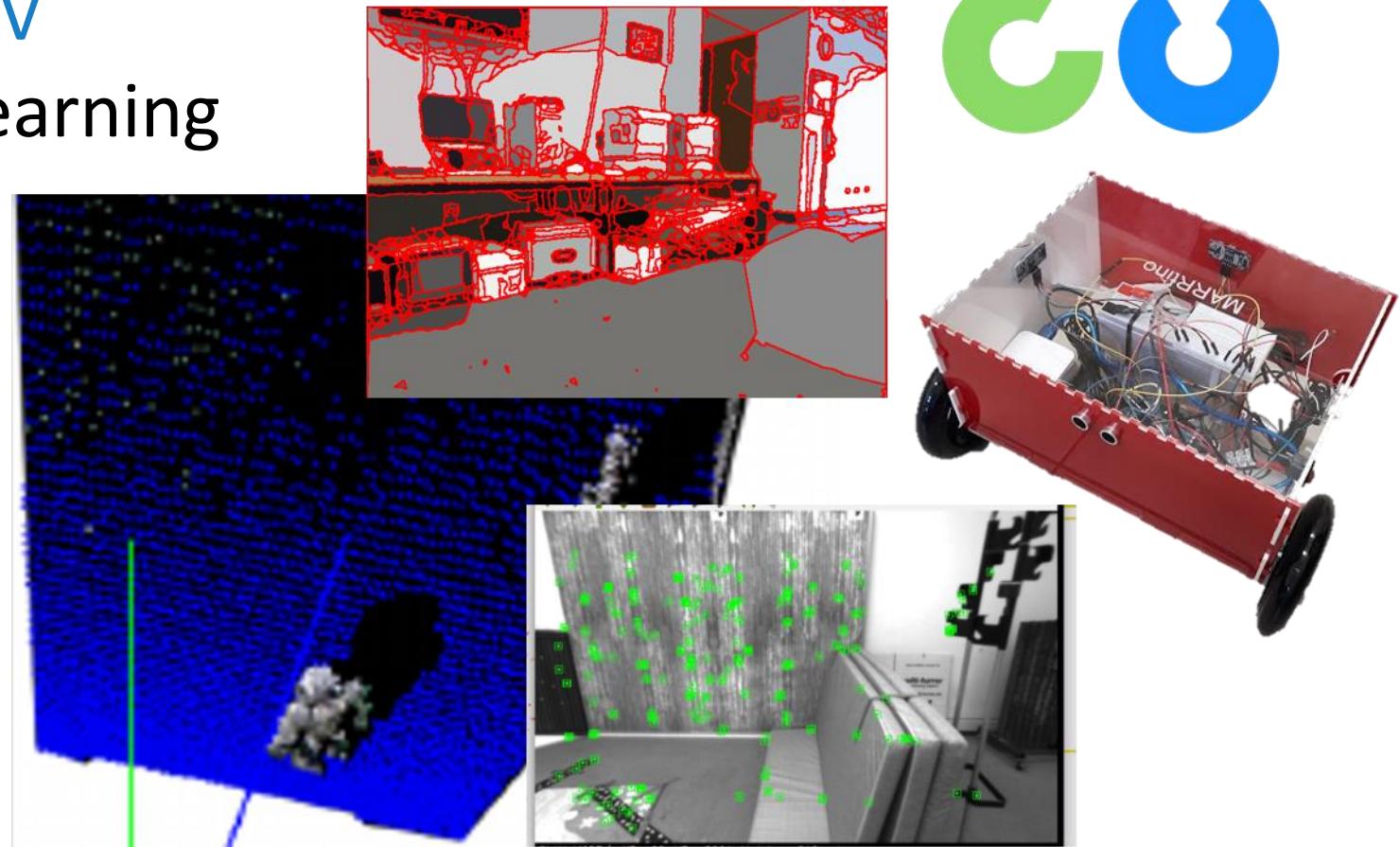
[domenico.bloisi@unibas.it](mailto:domenico.bloisi@unibas.it)



# Programma – Visione e Percezione

---

- Introduzione al linguaggio Python
- Elaborazione delle immagini con Python
- [Percezione 2D – OpenCV](#)
- Introduzione al Deep Learning
- ROS
- Il paradigma publisher and subscriber
- Simulatori
- Percezione 3D - PCL



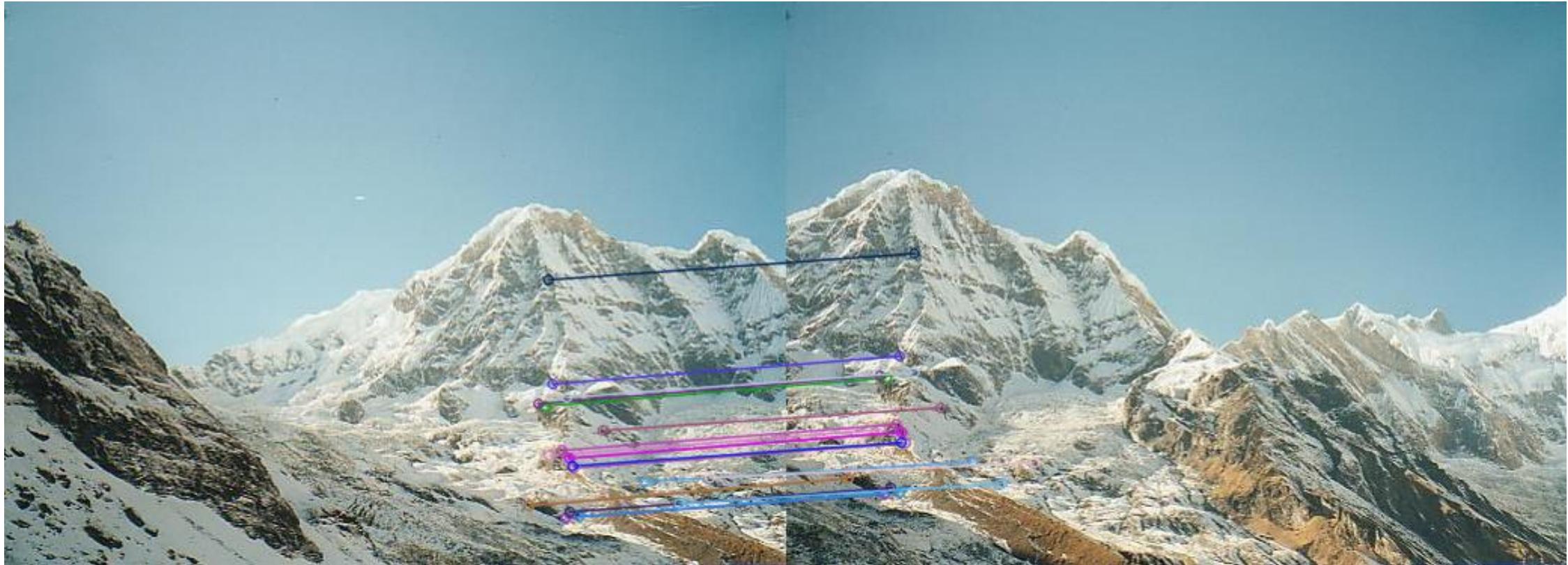
# Riferimenti

---

- Queste slide sono adattate da  
Noah Snavely - CS5670: Computer Vision  
"Lecture 5: Feature descriptors and matching"  
"Lecture 9: RANSAC"
- I contenuti fanno riferimento ai capitoli 3 e 4 del libro  
"Computer Vision: Algorithms and Applications"  
di Richard Szeliski, disponibile al seguente indirizzo  
<http://szeliski.org/Book/>

# Problem: Feature matching

---



# Recap

---

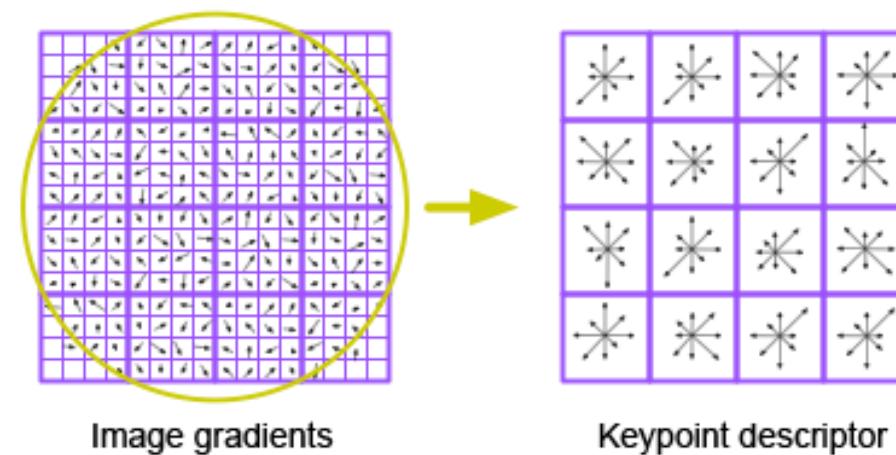
Keypoint detection: repeatable and distinctive

- Corners, blobs, stable regions
- Harris



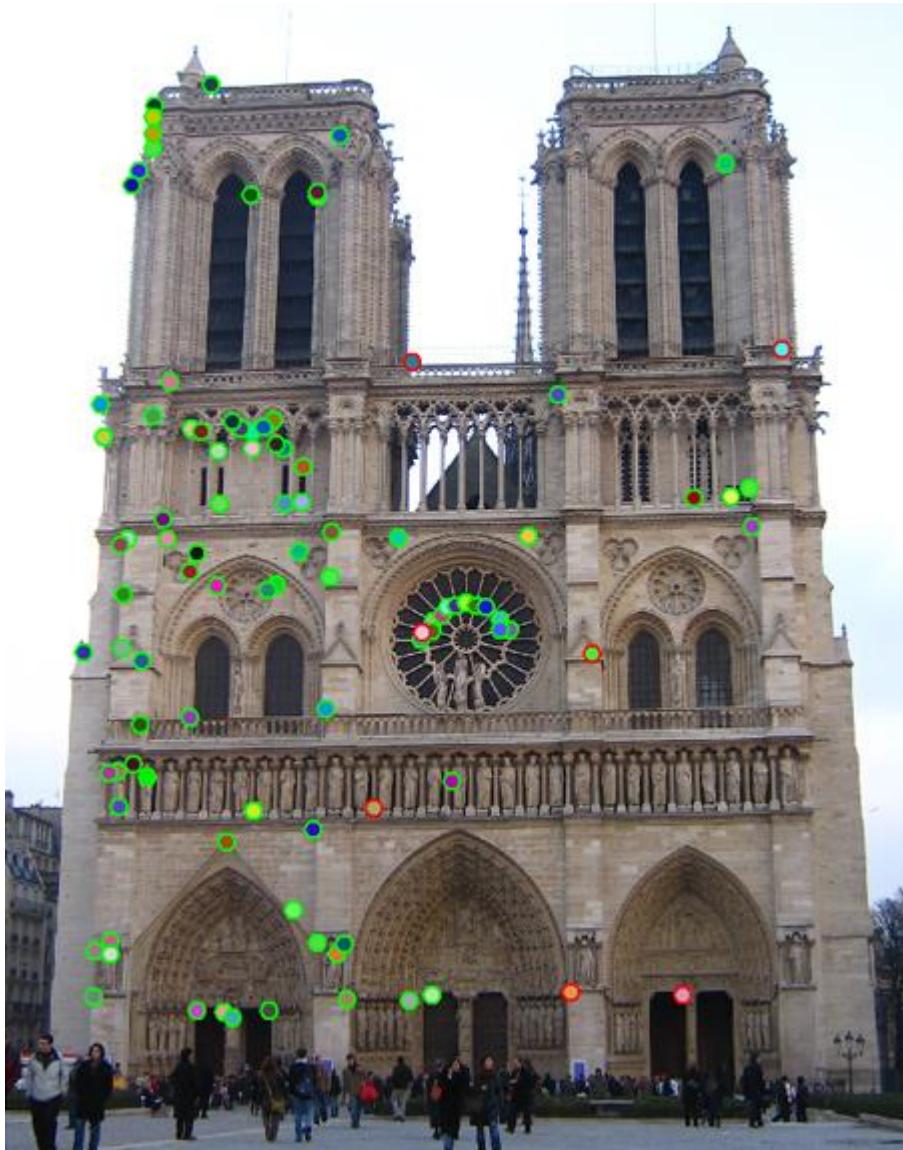
Descriptors: robust and selective

- spatial histograms of orientation
- SIFT and variants are typically good for stitching and recognition



# Which features match?

---

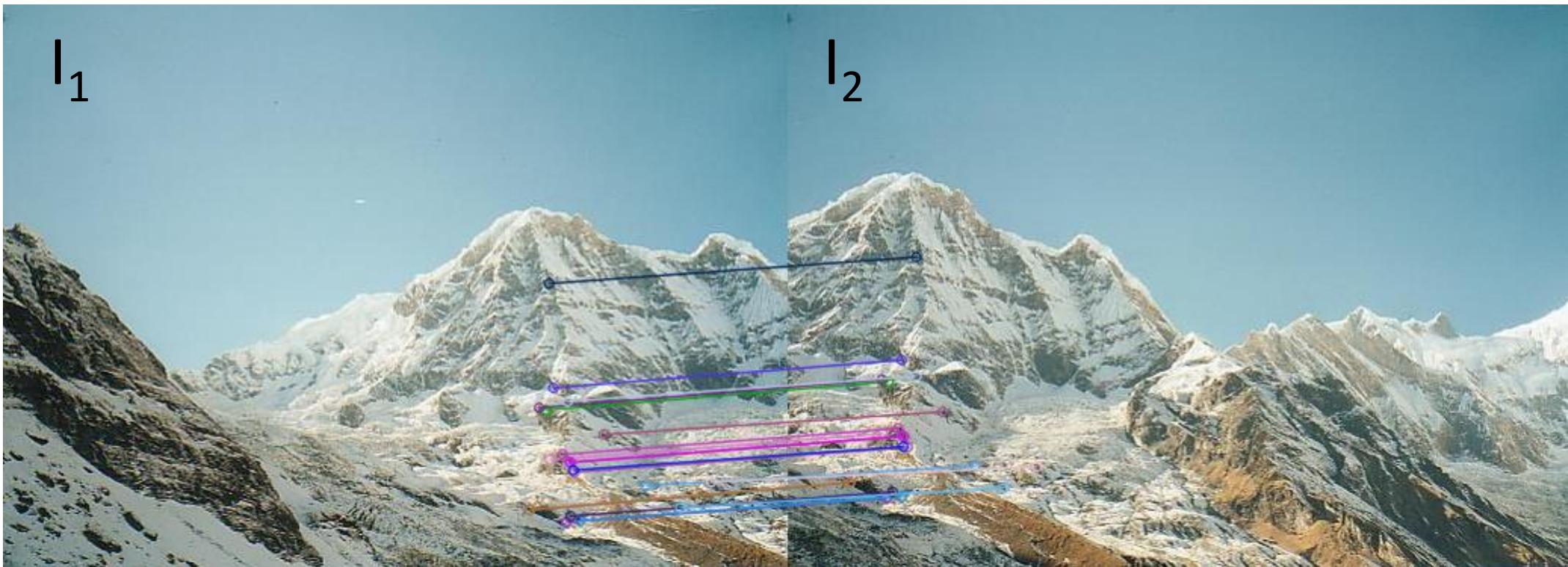


# Features matching

---

Given a feature in  $I_1$ , how to find the best match in  $I_2$ ?

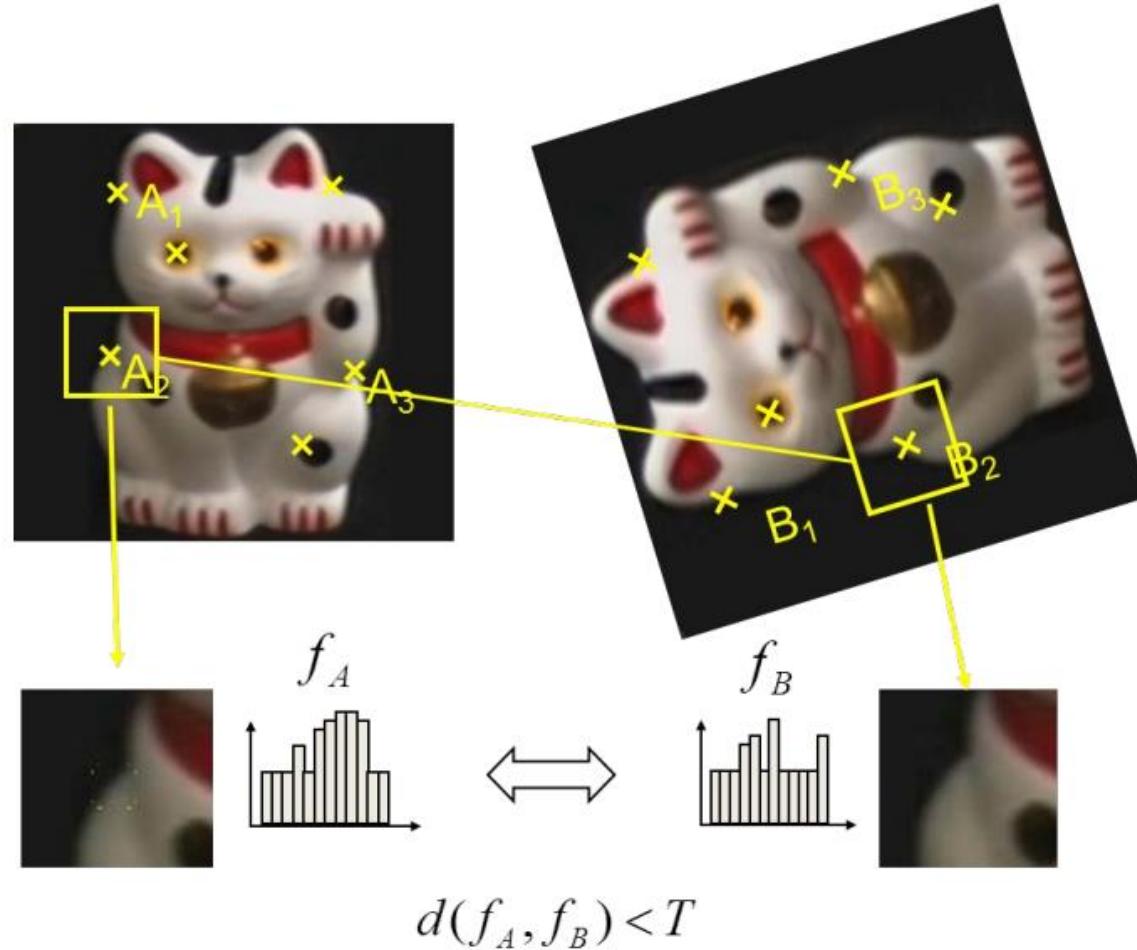
1. Define distance function that compares two descriptors
2. Test all the features in  $I_2$ , find the one with min distance



# Overview of point feature matching

---

1. Detect a set of distinct feature points
2. Define a patch around each point
3. Extract and normalize the patch
4. Compute a local descriptor
5. Match local descriptors



Source: Trym Vegard Haavardsholm

# Distance between descriptors

---

- $L_1$  distance (SAD):

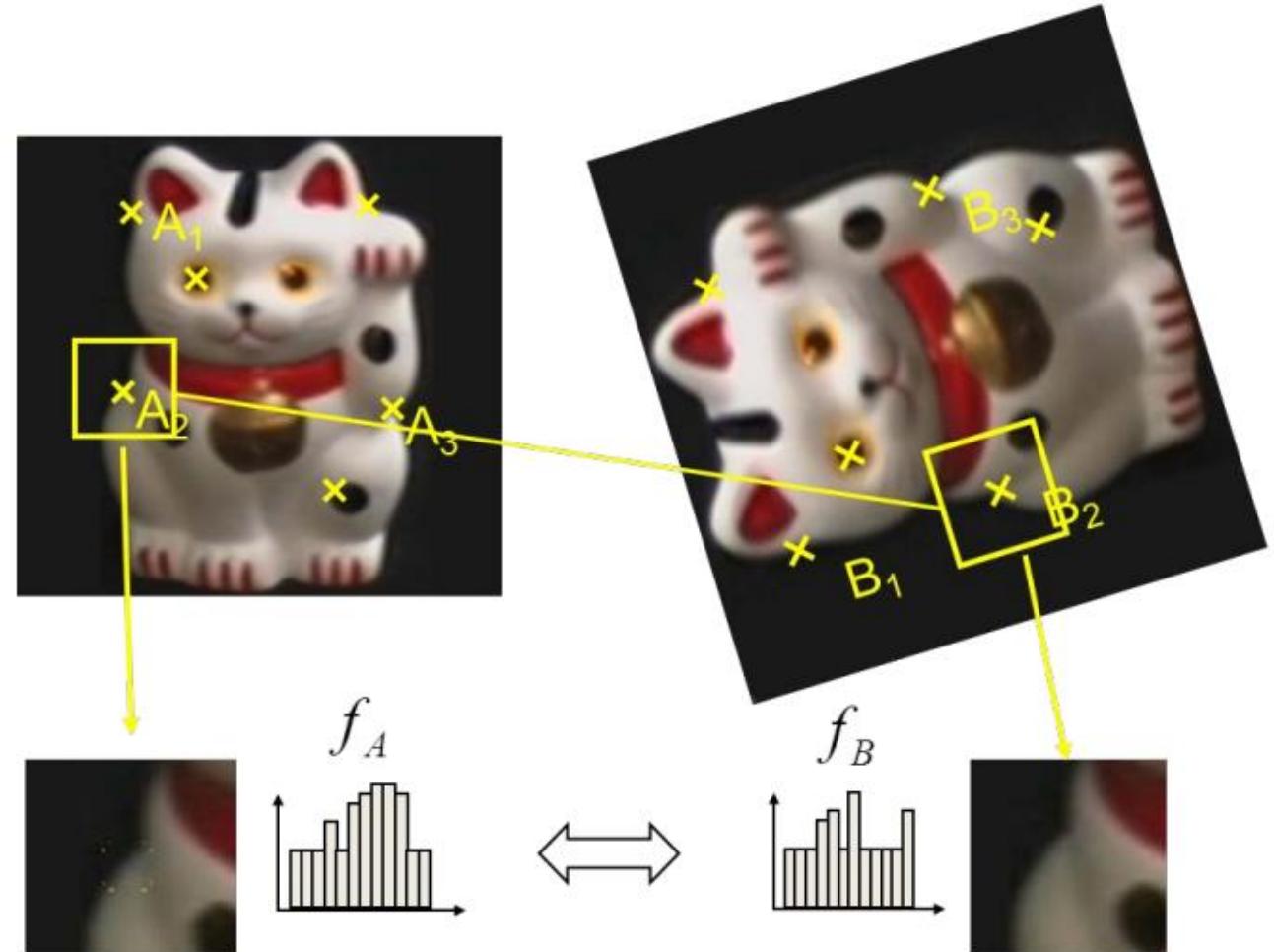
$$d(f_a, f_b) = \sum |f_a - f_b|$$

- $L_2$  distance (SSD):

$$d(f_a, f_b) = \sum (f_a - f_b)^2$$

- Hamming distance:

$$d(f_a, f_b) = \sum \text{XOR}(f_a, f_b)$$



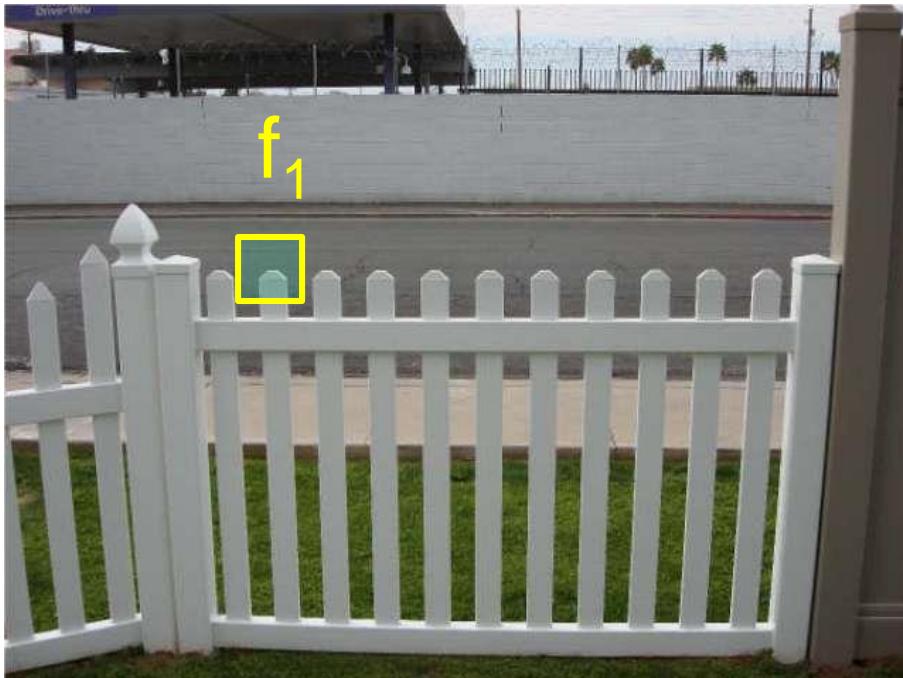
$$d(f_A, f_B) < T$$

Source: Trym Vegard Haavardsholm

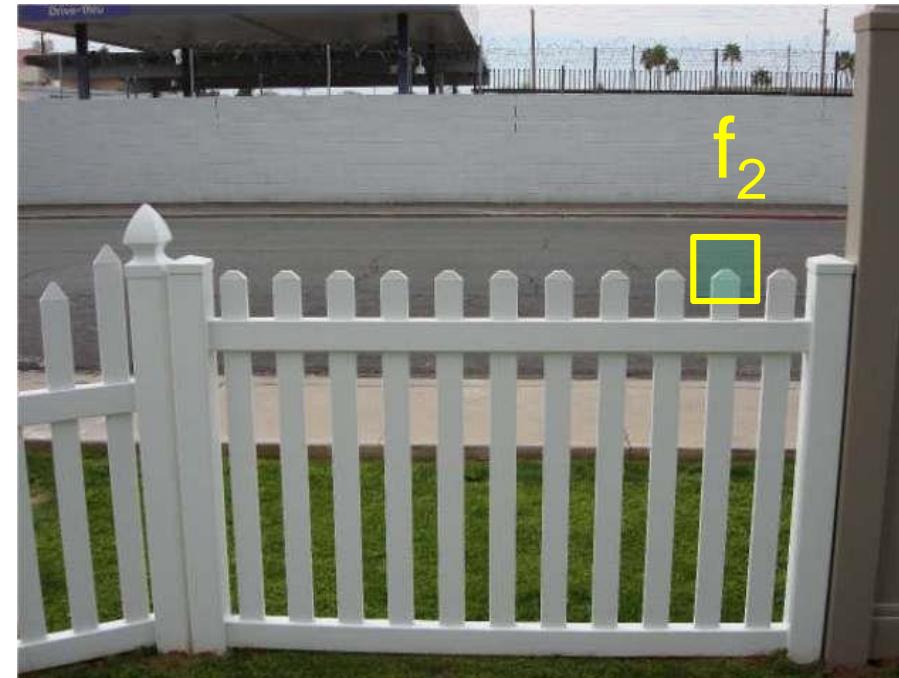
# Features distance: SSD

How to define the difference between two features  $f_1, f_2$ ?

- Simple approach:  $L_2$  distance,  $\|f_1 - f_2\|$   
i.e., sum of square differences (SSD) between entries of the two descriptors
- can give small distances for ambiguous (incorrect) matches  
i.e., does not provide a way to discard ambiguous (bad) matches



$I_1$

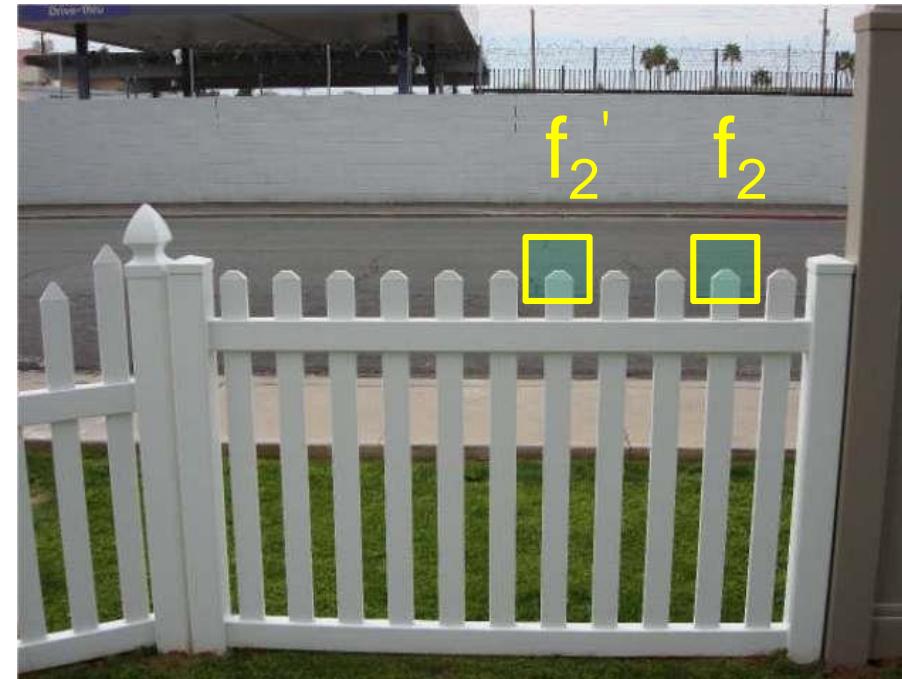
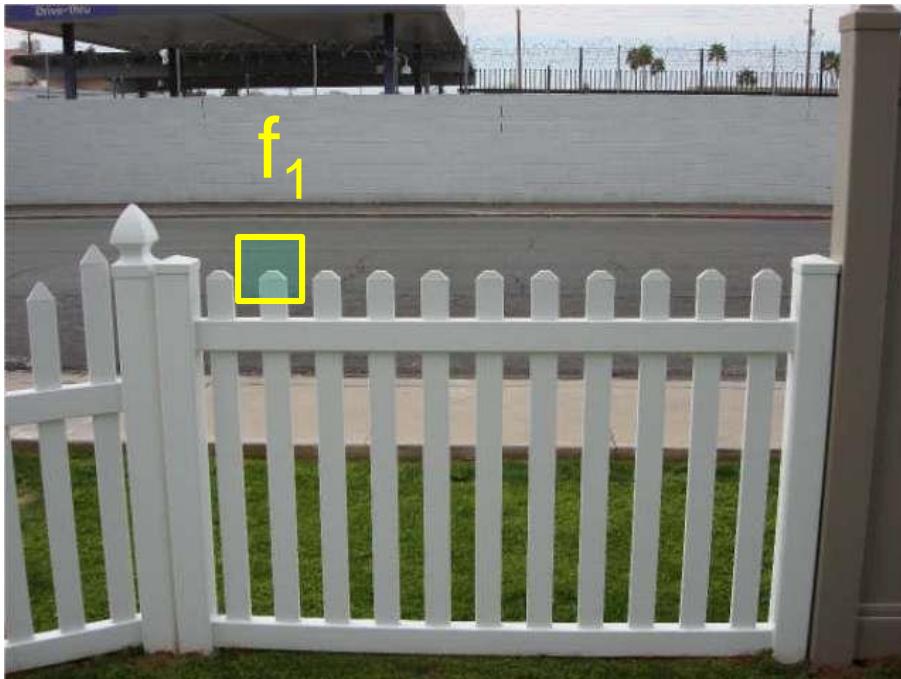


$I_2$

# Features distance: Ratio of SSDs

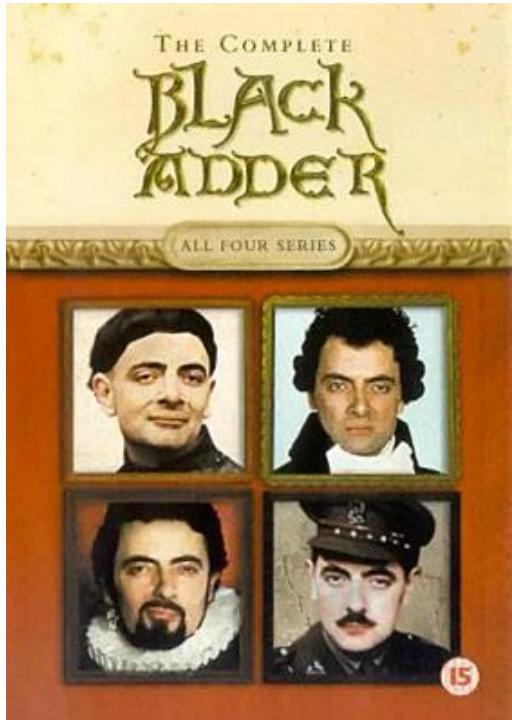
How to define the difference between two features  $f_1, f_2$ ?

- Better approach: ratio distance =  $\text{SSD}(f_1, f_2) / \text{SSD}(f_1, f_2')$ 
  - $f_2$  is best SSD match to  $f_1$  in  $I_2$
  - $f_2'$  is 2nd best SSD match to  $f_1$  in  $I_2$
  - An ambiguous/bad match will have ratio close to 1
  - Look for unique matches which have low ratio



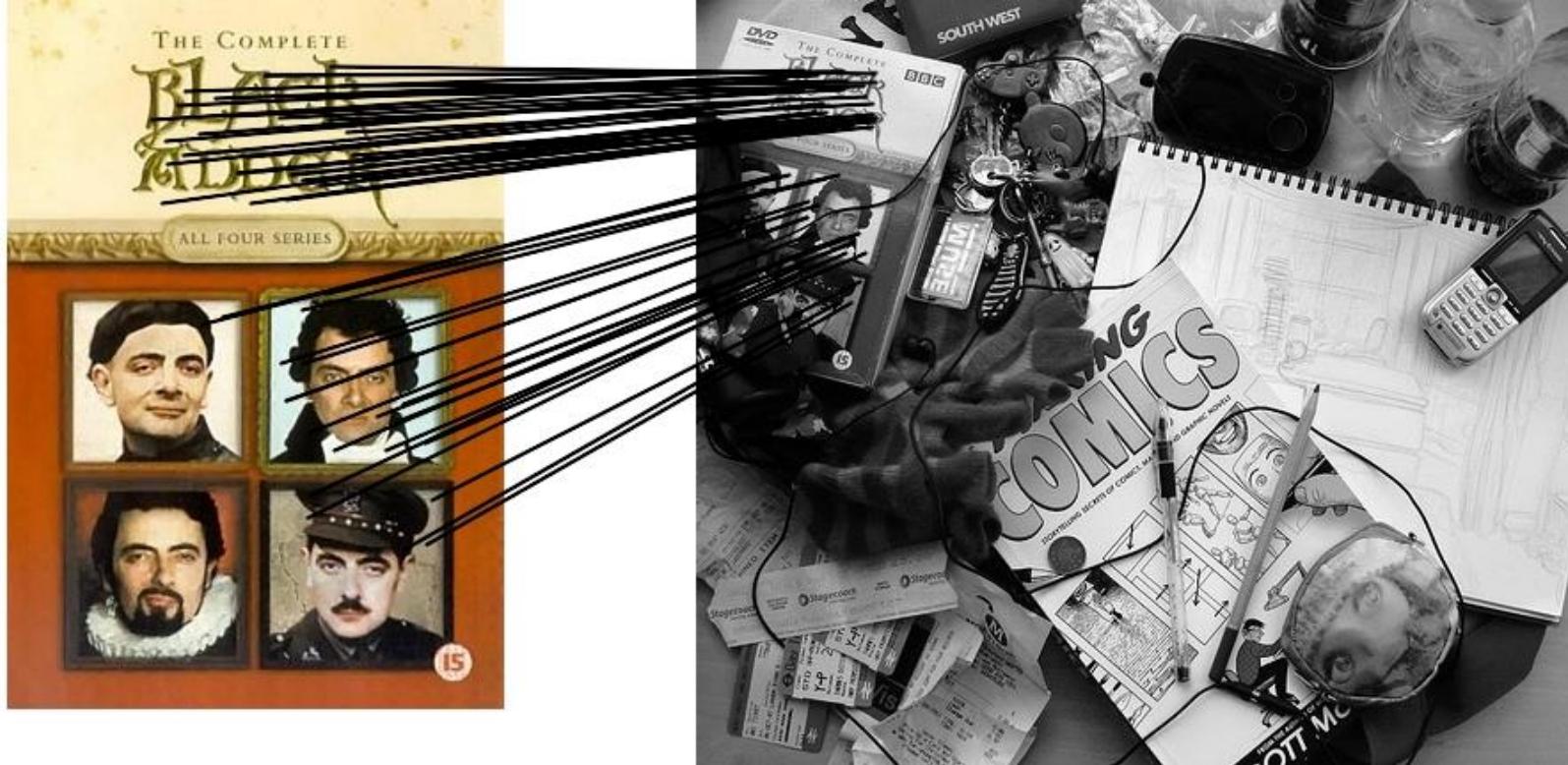
# Feature matching example

---



# Feature matching example

---



58 matches (thresholded by ratio score)

# Example in Colab: ratio test

```
orb = cv.ORB_create()

kp_1, des_1 = orb.detectAndCompute(img_1, None)
kp_2, des_2 = orb.detectAndCompute(img_2, None)

bf = cv.BFMatcher()
matches = bf.knnMatch(des_1, des_2, k=2)

# Apply ratio test
good = []
for m,n in matches:
    if m.distance < 0.7*n.distance:
        good.append([m])

img_3 = cv.drawMatchesKnn(img_1, kp_1, img_2, kp_2, good[:20], None, flags=2)

img_3_rgb = cv.cvtColor(img_3, cv.COLOR_BGR2RGB)

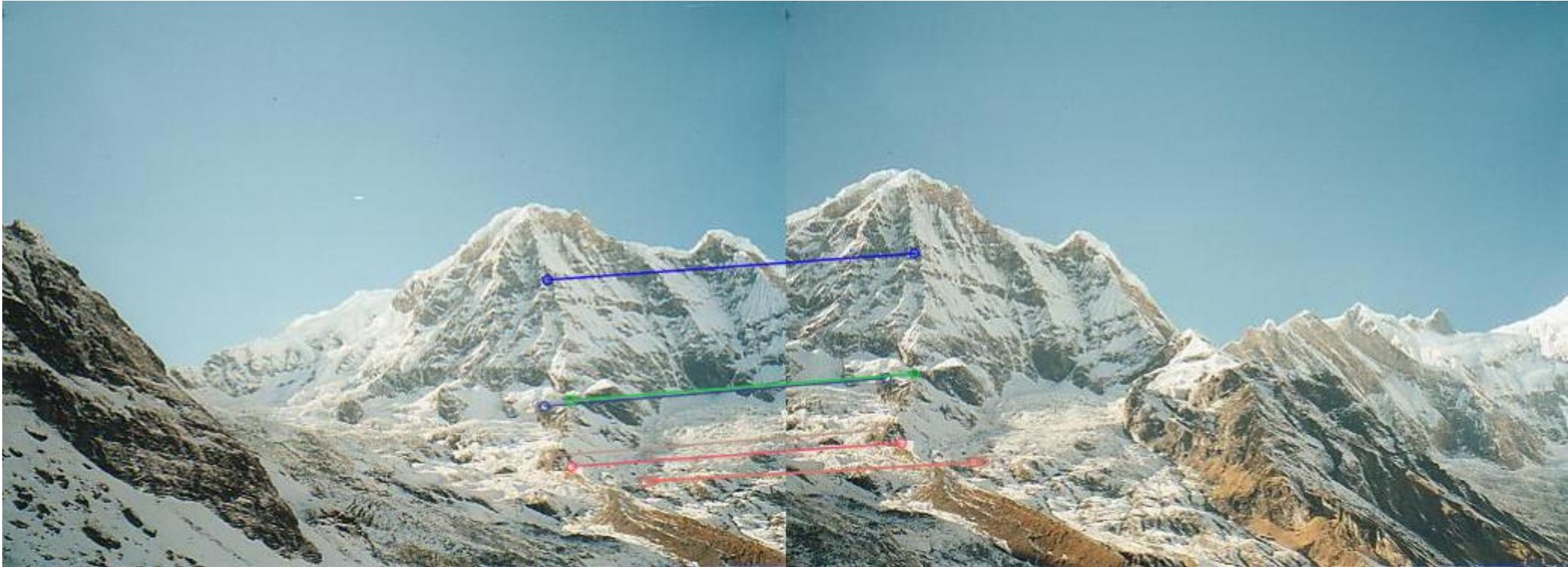
plt.axis('off')
plt.imshow(img_3_rgb)
plt.show()

cv.imwrite('matching.png', img_3)
```

`BFMatcher.knnMatch()` to get k best matches.  
In this example, we will take k=2 so that we can apply ratio test

# Example in Colab: ratio test results

---



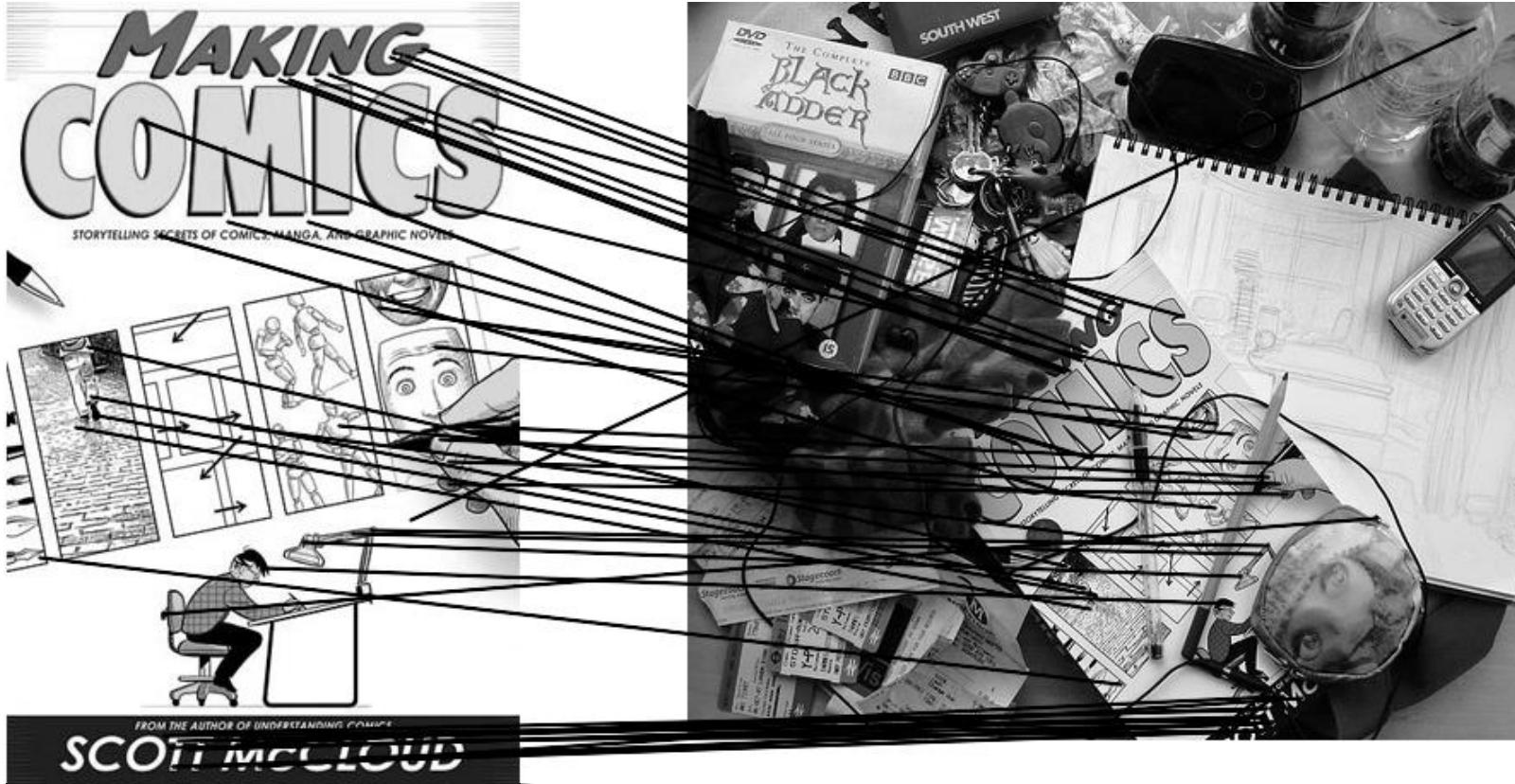
# Feature matching example

---



# Feature matching example

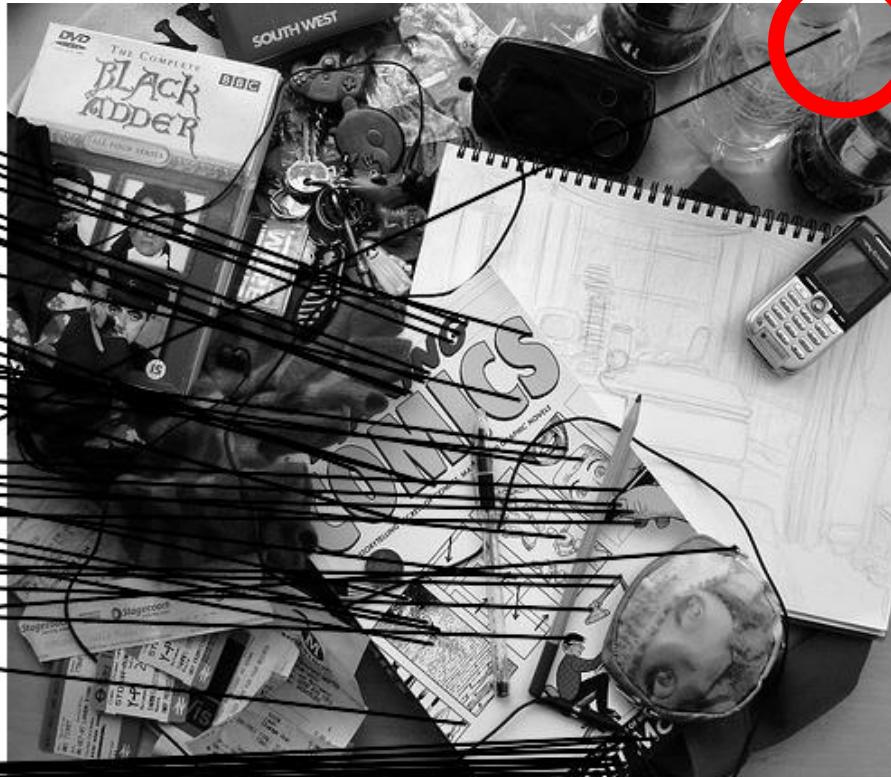
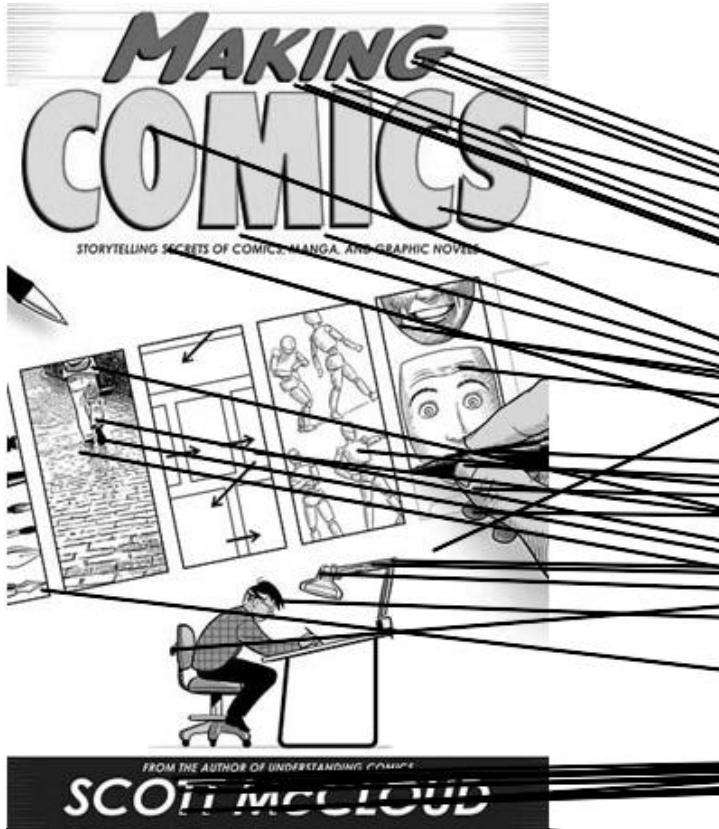
---



51 matches (thresholded by ratio score)

# Feature matching example

---

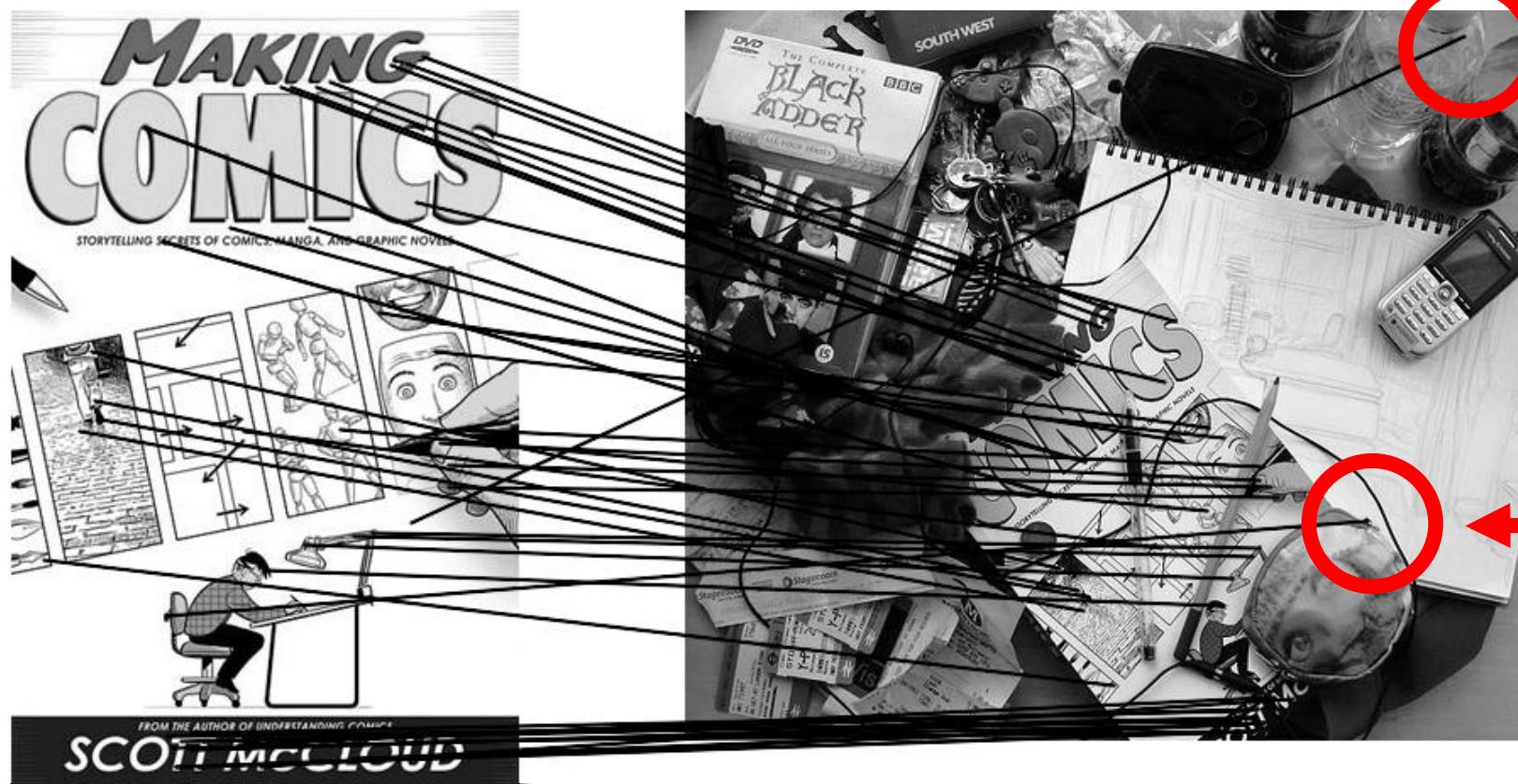


← outlier

51 matches (thresholded by ratio score)

# Feature matching example

---



outlier

outlier

51 matches (thresholded by ratio score)

# Evaluating the results

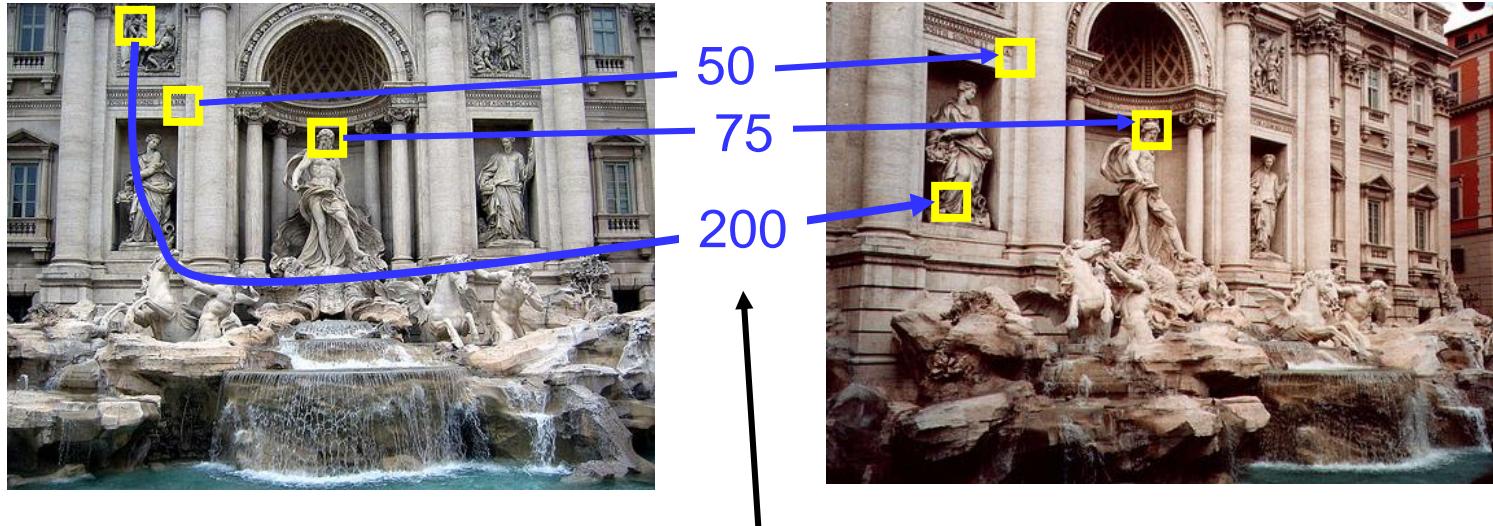
---

How can we measure the performance of a feature matcher?

# Evaluating the results

---

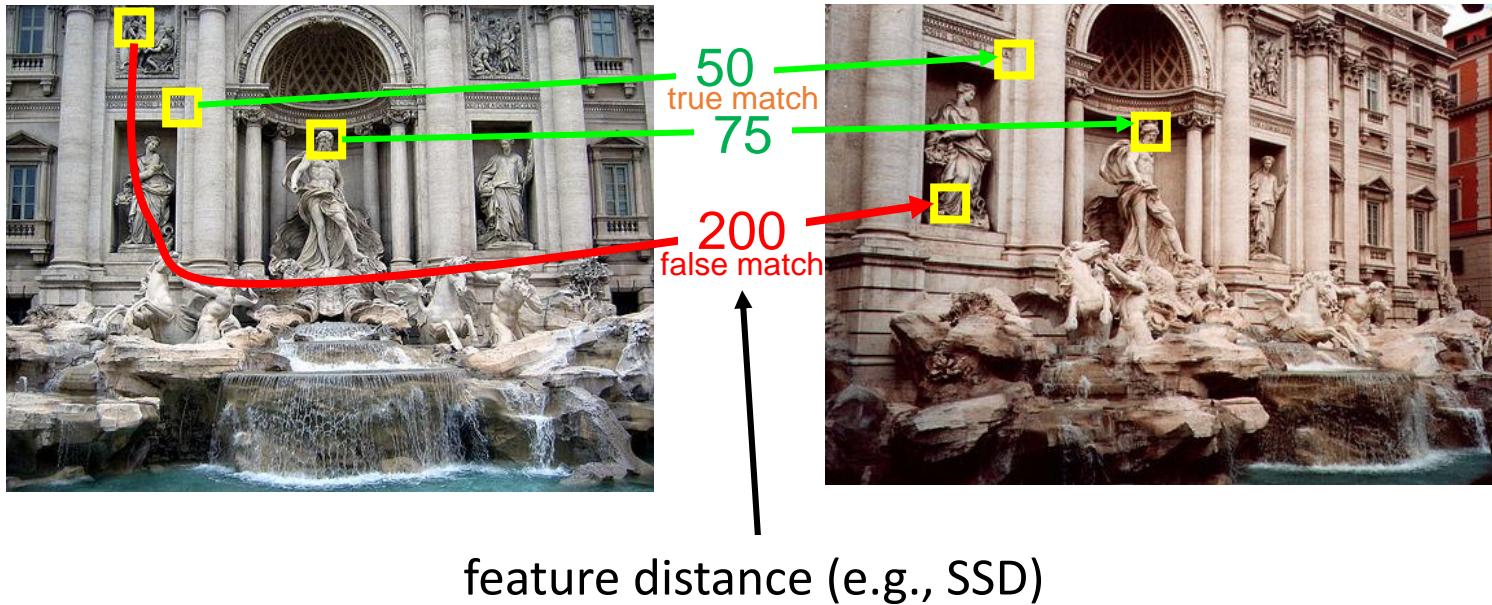
How can we measure the performance of a feature matcher?



feature distance (e.g., SSD)

# True/false positives

How can we measure the performance of a feature matcher?

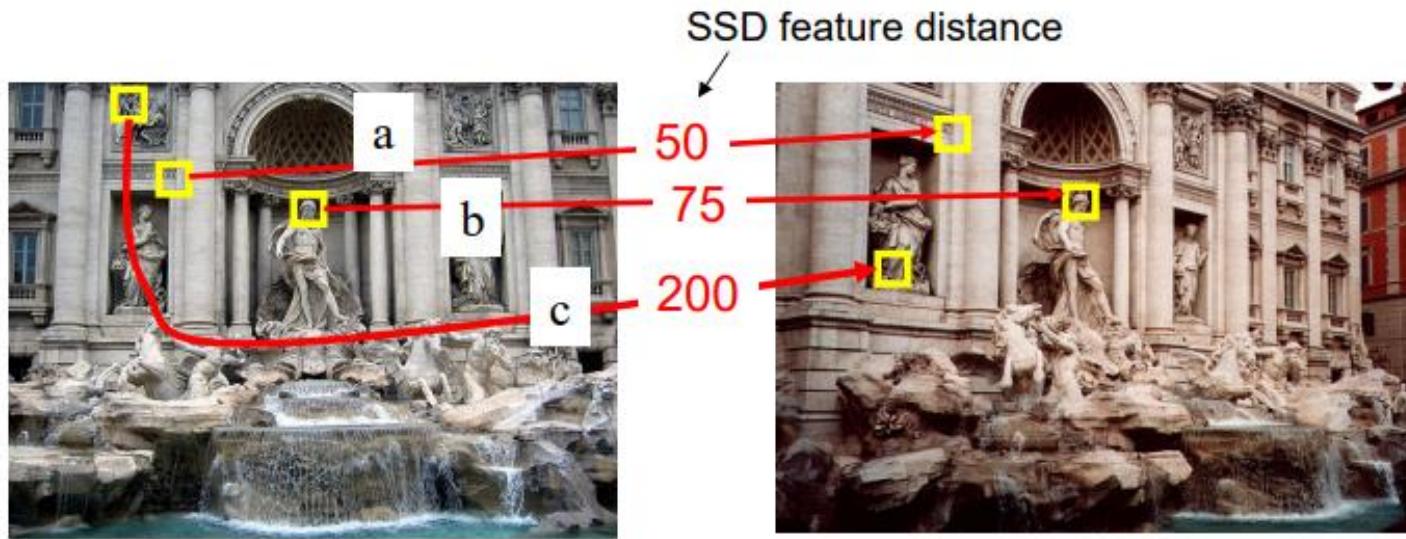


The distance threshold affects performance

- **True positives** = # of detected matches that are correct
  - Suppose we want to maximize these—how to choose threshold?
- **False positives** = # of detected matches that are incorrect
  - Suppose we want to minimize these—how to choose threshold?

# Large threshold T

Maximize  
TP



**Decision rule:** Accept match if  $\text{SSD} < T$

Example: **Large T**

$T = 250 \Rightarrow a, b, c$  are all accepted as matches

a and b are true matches (“**true positives**”)

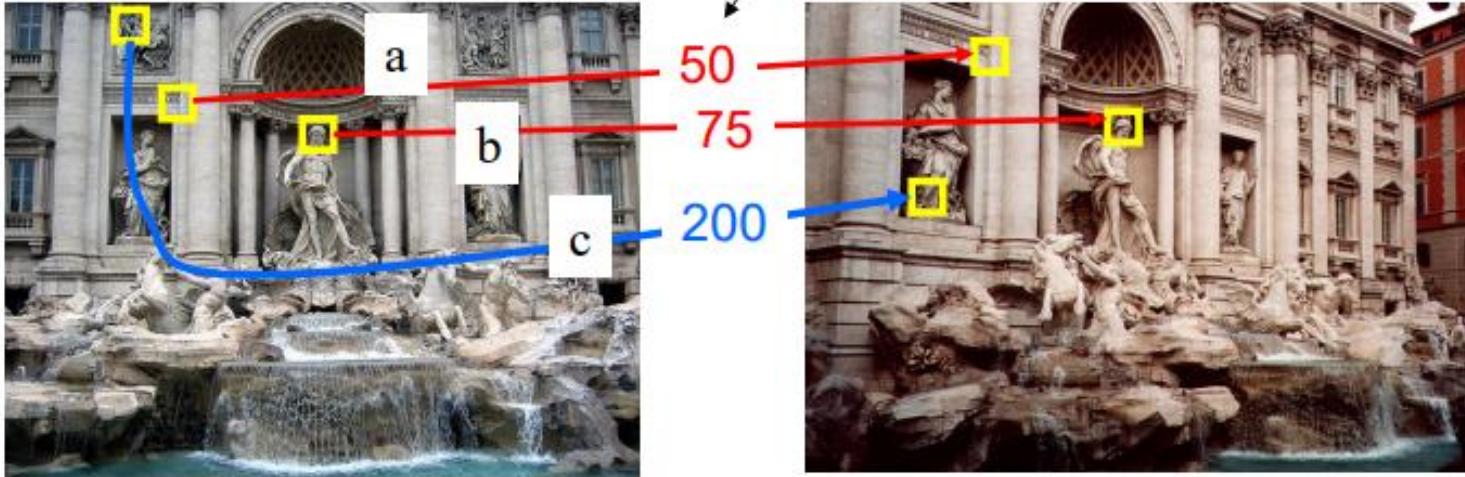
- they are actually matches

c is a false match (“**false positive**”)

- actually not a match

# Small threshold T

Minimize  
FP



**Decision rule:** Accept match if  $SSD < T$

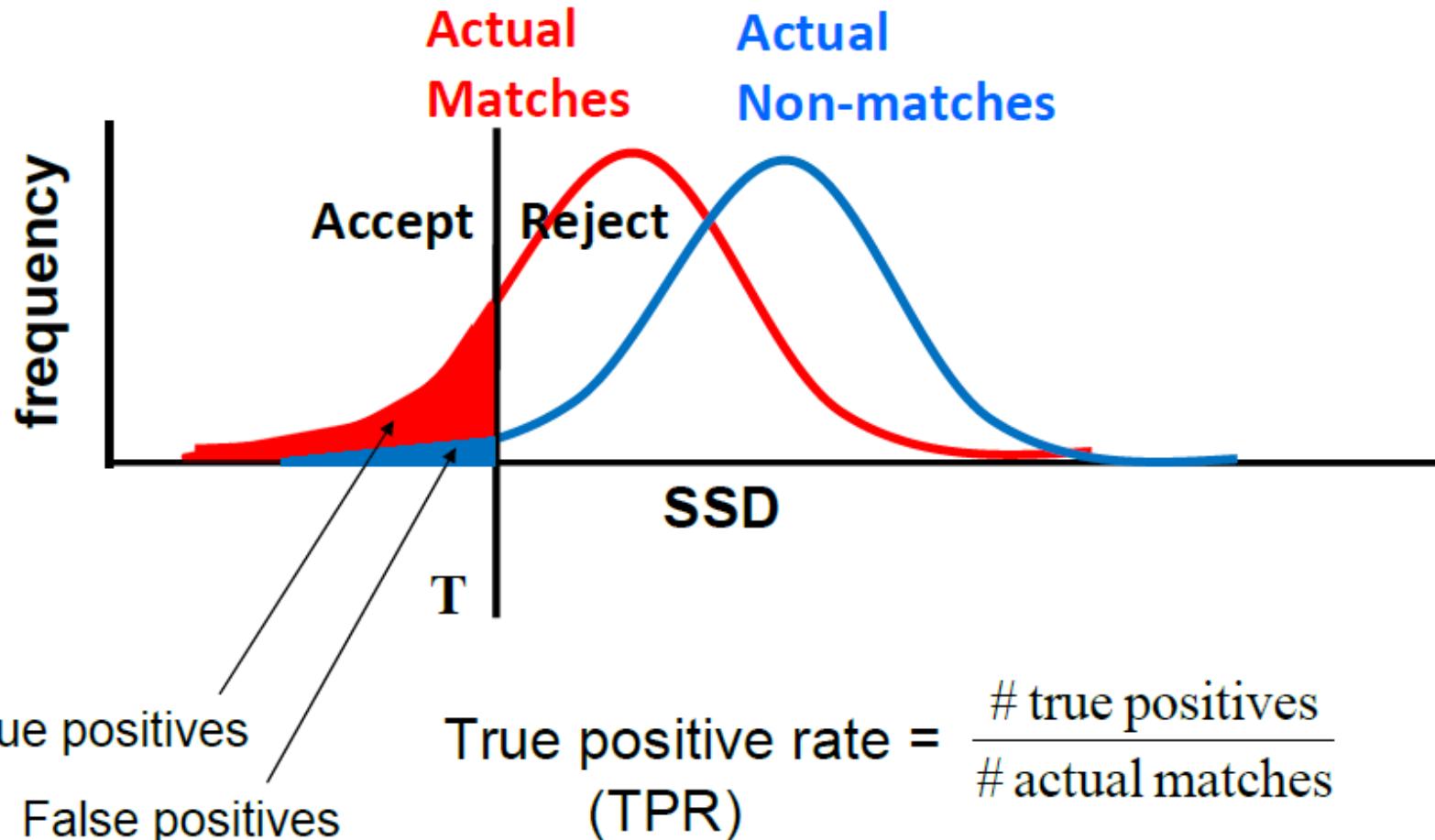
Example: **Smaller T**

$T = 100 \Rightarrow$  only a and b are accepted as matches

a and b are true matches (“true positives”)

c is no longer a “false positive” (it is a “true negative”)

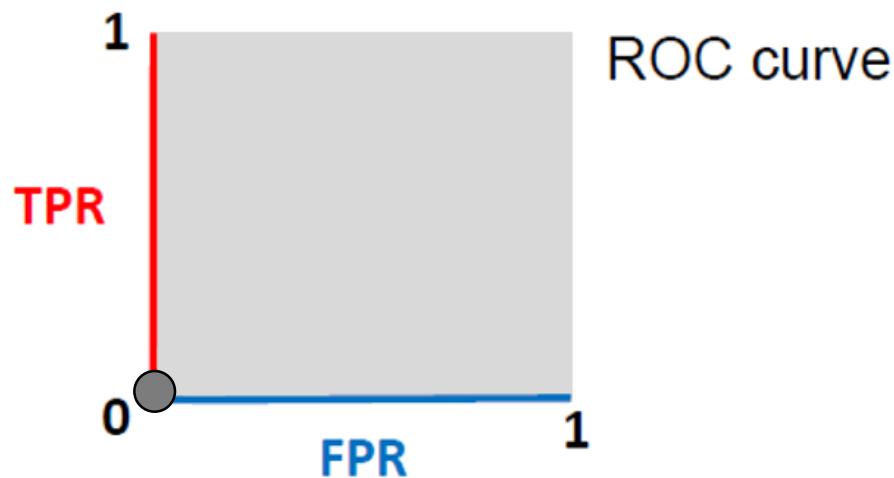
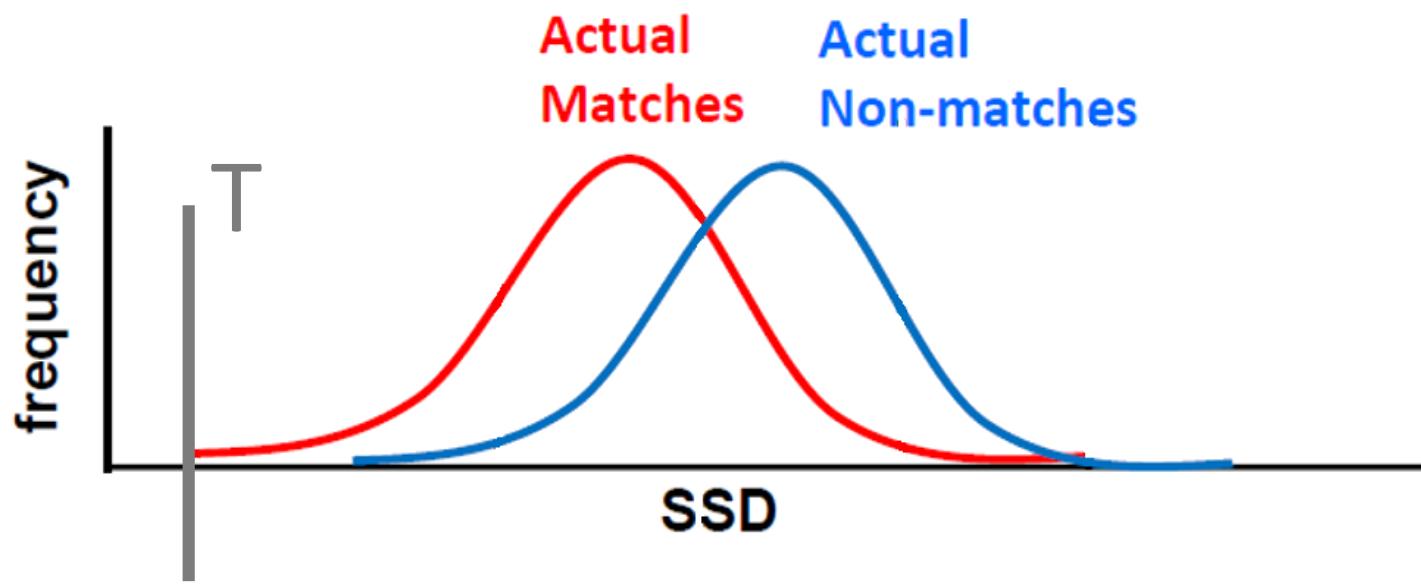
# True positives and false positives



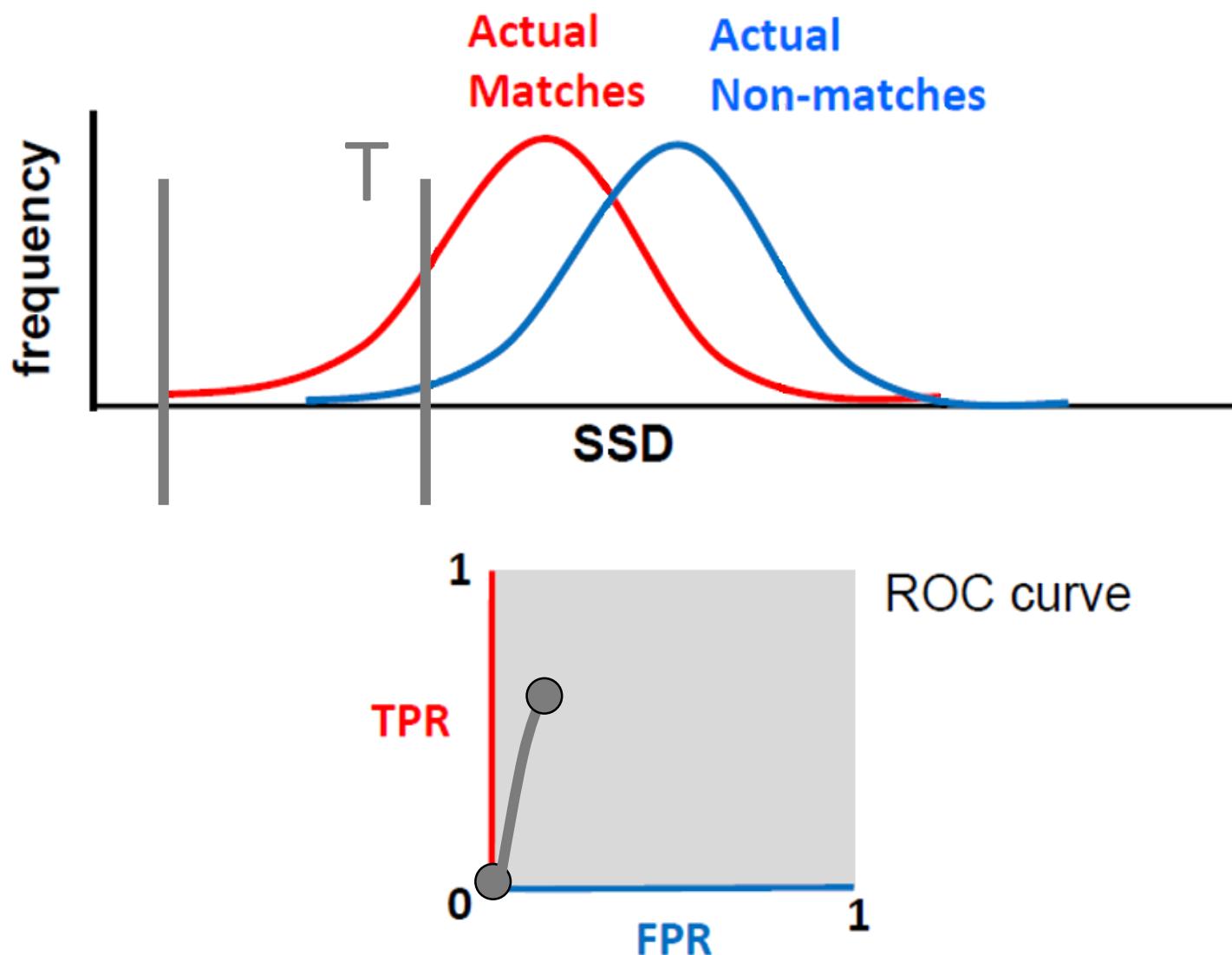
True positive rate =  $\frac{\# \text{ true positives}}{\# \text{ actual matches}}$

False positive rate =  $\frac{\# \text{ false positives}}{\# \text{ actual nonmatches}}$

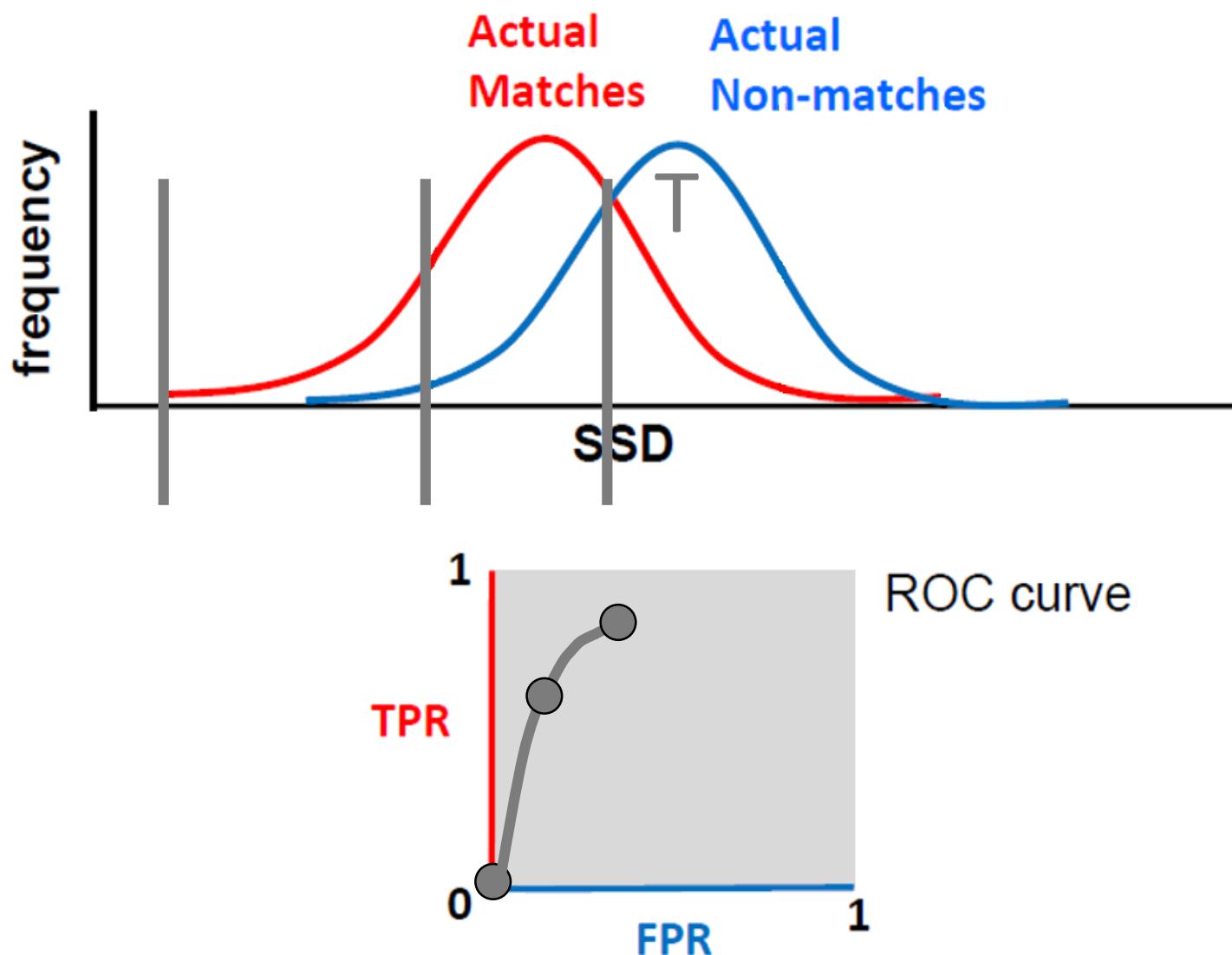
# Receiver Operating Characteristic (ROC) curve



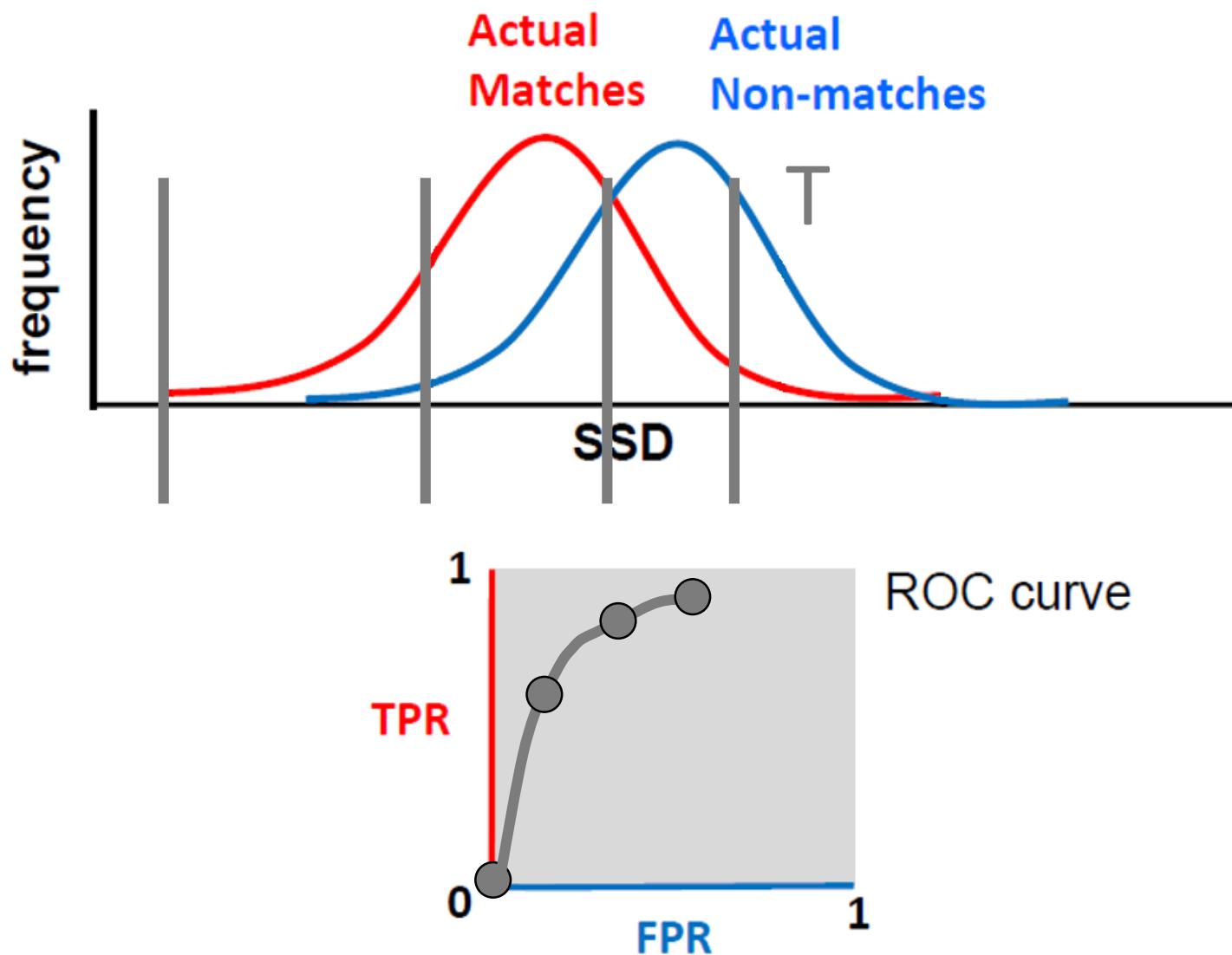
# Receiver Operating Characteristic (ROC) curve



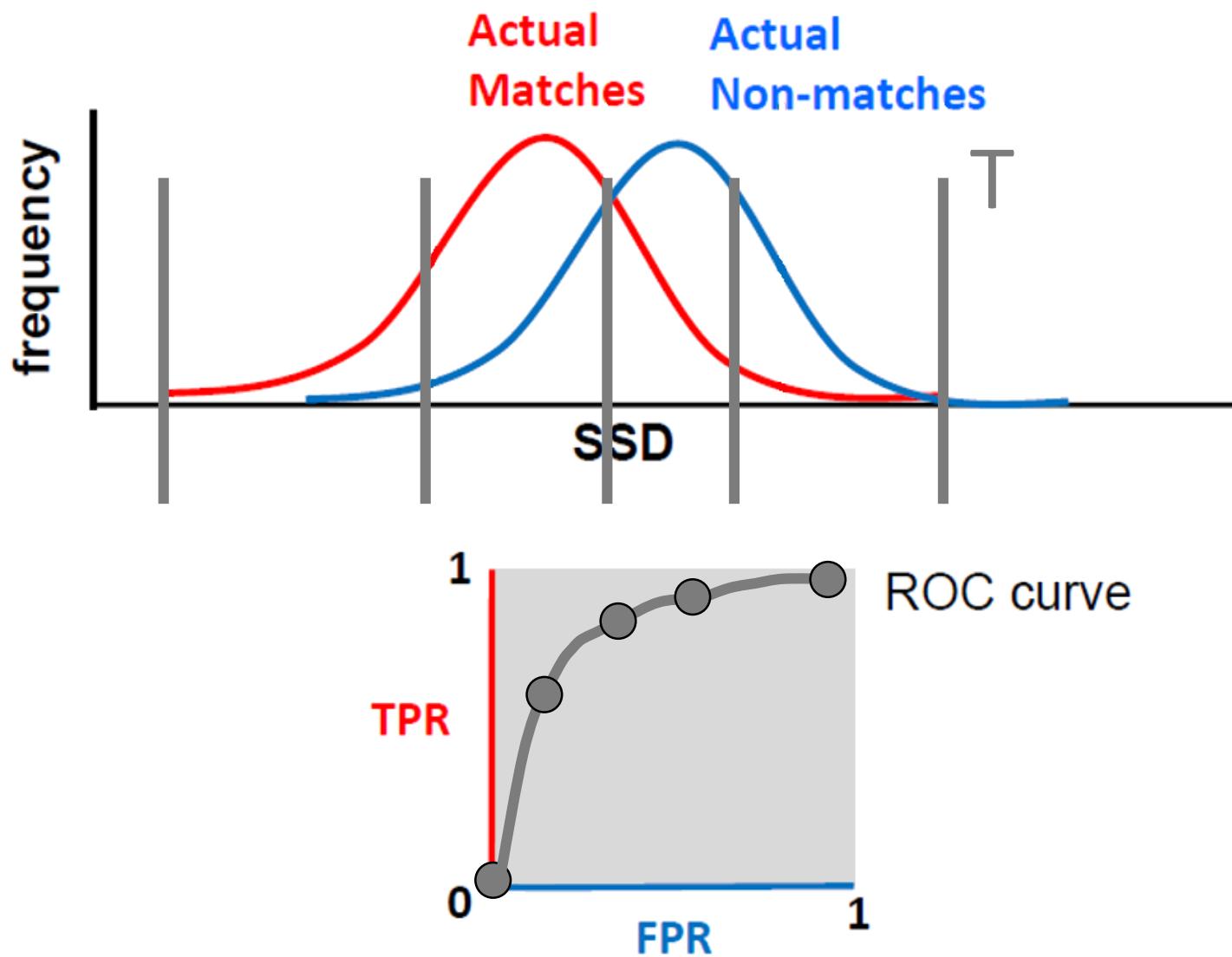
# Receiver Operating Characteristic (ROC) curve



# Receiver Operating Characteristic (ROC) curve

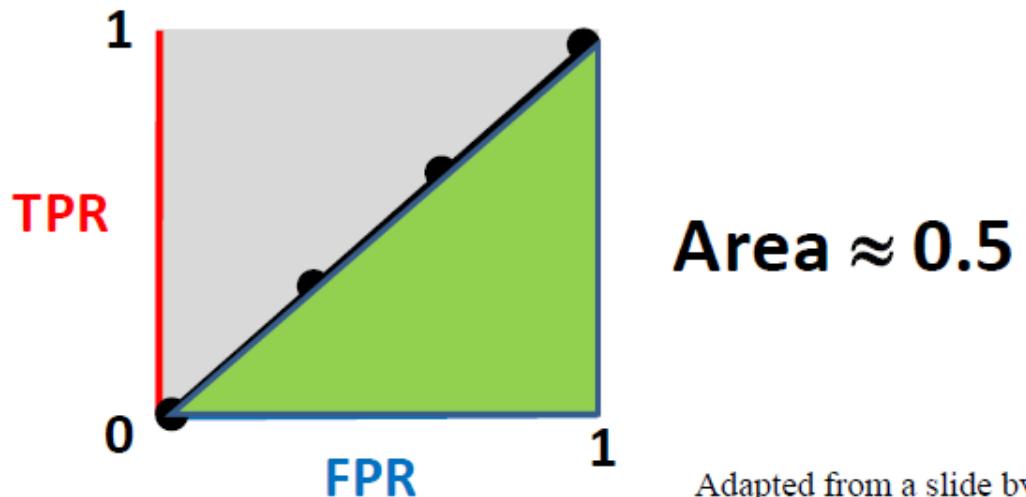
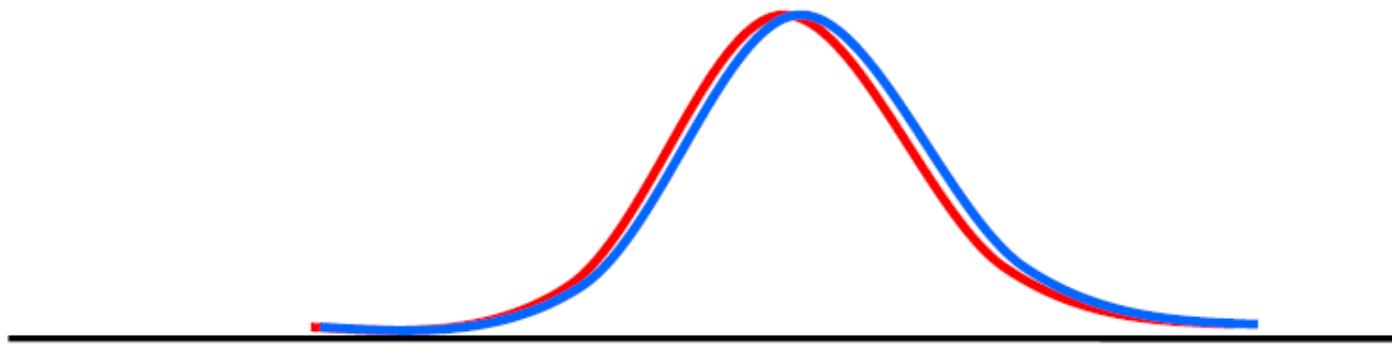


# Receiver Operating Characteristic (ROC) curve



# If the features selected were bad...

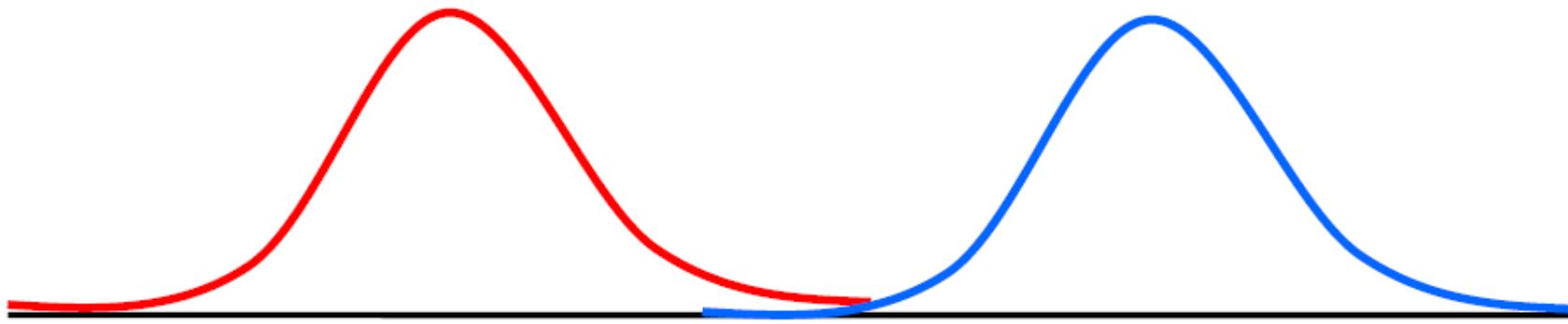
---



Adapted from a slide by Shin Kira

# If the features selected were good...

---

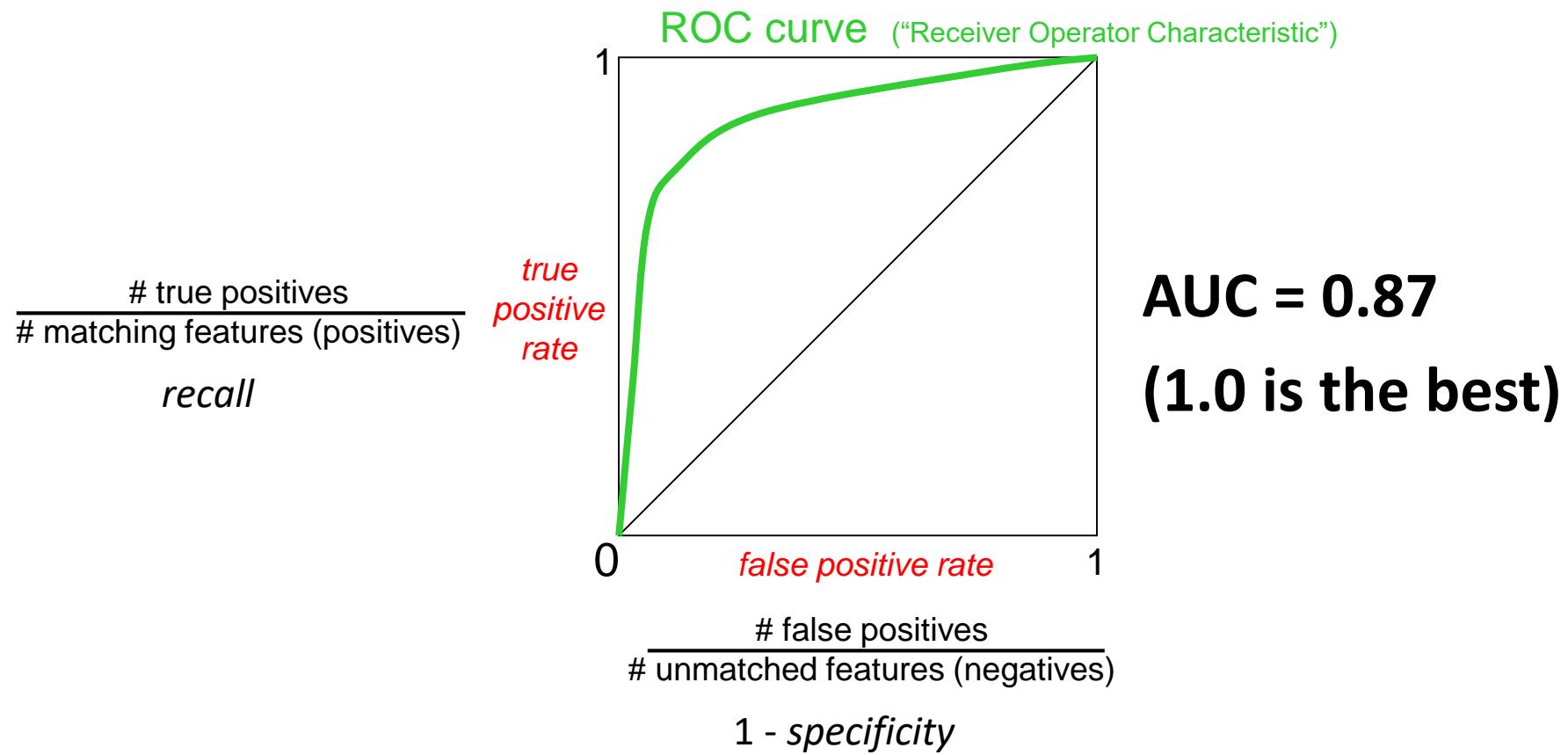


Adapted from a slide by Shin Kira

# Area under the curve

---

Single number: Area Under the Curve (AUC)



# Feature matching: example using ORB

The screenshot shows a Jupyter Notebook interface with the following details:

- File:** matching.ipynb
- Toolbar:** File, Edit, View, Insert, Runtime, Tools, Help, CODE, TEXT, CELL, CELL.
- Table of contents:** Code snippets, Files.
- Files:** UPLOAD, REFRESH, sample\_data, montagna-1.jpg, montagna-2.jpg.
- Code Cell:** Contains Python code for feature matching using ORB.
- Code Content:**

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

img_1 = cv.imread("montagna-1.jpg")
gray_1 = cv.cvtColor(img_1, cv.COLOR_BGR2GRAY)

img_2 = cv.imread("montagna-2.jpg")
gray_2 = cv.cvtColor(img_2, cv.COLOR_BGR2GRAY)

orb = cv.ORB_create()

kp_1, des_1 = orb.detectAndCompute(img_1, None)
kp_2, des_2 = orb.detectAndCompute(img_2, None)

bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)

matches = bf.match(des_1, des_2)

matches = sorted(matches, key = lambda x:x.distance)

img_3 = cv.drawMatches(img_1, kp_1, img_2, kp_2, matches[:20], None, flags=2)

img_3_rgb = cv.cvtColor(img_3, cv.COLOR_GRAY2RGB)

plt.axis('off')
plt.imshow(img_3_rgb)
plt.show()

cv.imwrite('matching.png', img_3)
```
- Output:** Two URLs pointing to images: <https://dbloisi.github.io/corsi/images/montagna-1.jpg> and <https://dbloisi.github.io/corsi/images/montagna-2.jpg>.

# ORB

---



```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

img_1 = cv.imread("montagna-1.jpg")
gray_1 = cv.cvtColor(img_1, cv.COLOR_BGR2GRAY)

img_2 = cv.imread("montagna-2.jpg")
gray_2 = cv.cvtColor(img_2, cv.COLOR_BGR2GRAY)

orb = cv.ORB_create()

kp_1, des_1 = orb.detectAndCompute(img_1, None)
kp_2, des_2 = orb.detectAndCompute(img_2, None)
```

# Brute force matching

---

Brute-Force matcher is simple. It takes the descriptor of one feature in first set and is matched with all other features in second set using some distance calculation. And the closest one is returned.

Hamming distance:

$$d(f_a, f_b) = \sum \text{XOR}(f_a, f_b)$$



```
bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)

matches = bf.match(des_1, des_2)
```

# Cross check test

---

- Choose matches  $(f_a, f_b)$  so that
  - $f_b$  is the best match for  $f_a$  in  $I_b$
  - And  $f_a$  is the best match for  $f_b$  in  $I_a$
- Alternative to ratio test



# Cross check test

---

- Choose matches  $(f_a, f_b)$  so that
  - $f_b$  is the best match for  $f_a$  in  $I_b$
  - And  $f_a$  is the best match for  $f_b$  in  $I_a$
- Alternative to ratio test



# Sorting

---

Matches are sorted in ascending order of their distances so that best matches (with low distance) come to front.

```
bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)

matches = bf.match(des_1, des_2)

matches = sorted(matches, key = lambda x:x.distance)
```



In Python, le funzioni lambda, dette anche funzioni anonime, sono funzioni che vengono usate per un periodo di tempo limitato e sono legate a funzioni di più alto livello

# Result

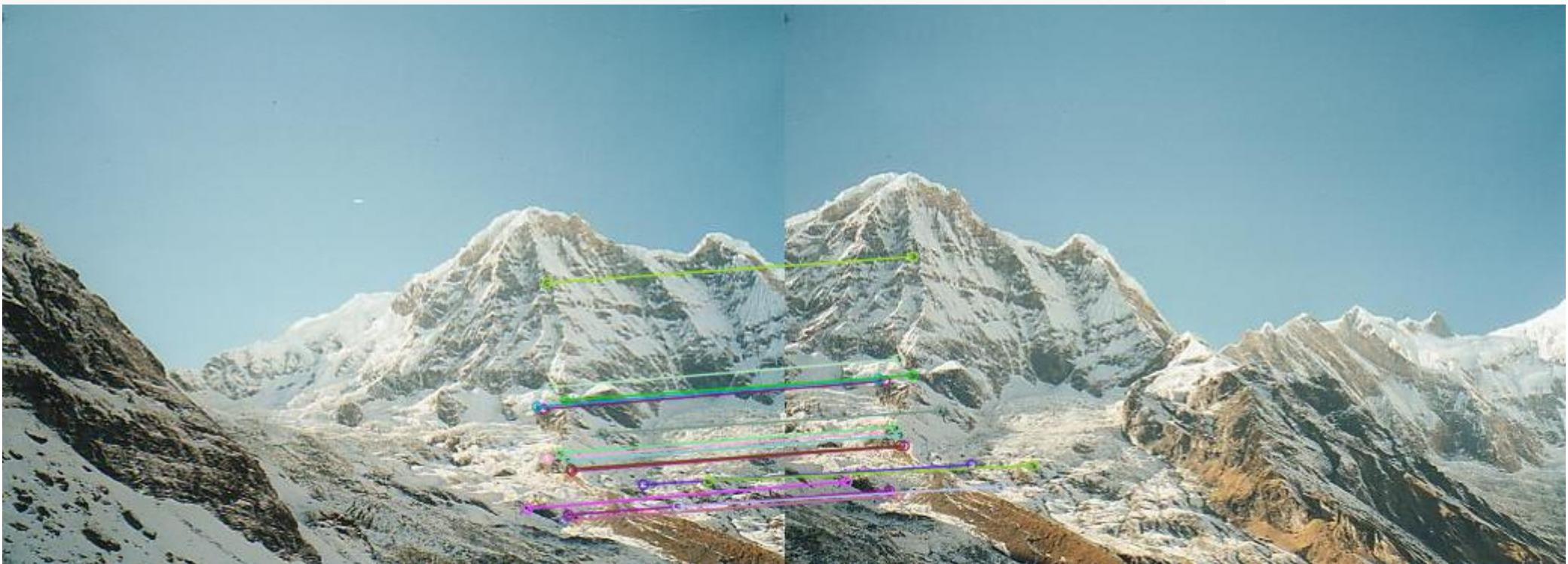
---

```
img_3 = cv.drawMatches(img_1, kp_1, img_2, kp_2, matches[:20], None, flags=2)

img_3_rgb = cv.cvtColor(img_3, cv.COLOR_BGR2RGB)

plt.axis('off')
plt.imshow(img_3_rgb)
plt.show()

cv.imwrite('matching.png', img_3)
```



# SIFT Example

---



Univ4.jpg



Univ3.jpg

# SIFT Example

---



# SIFT vs ORB

---



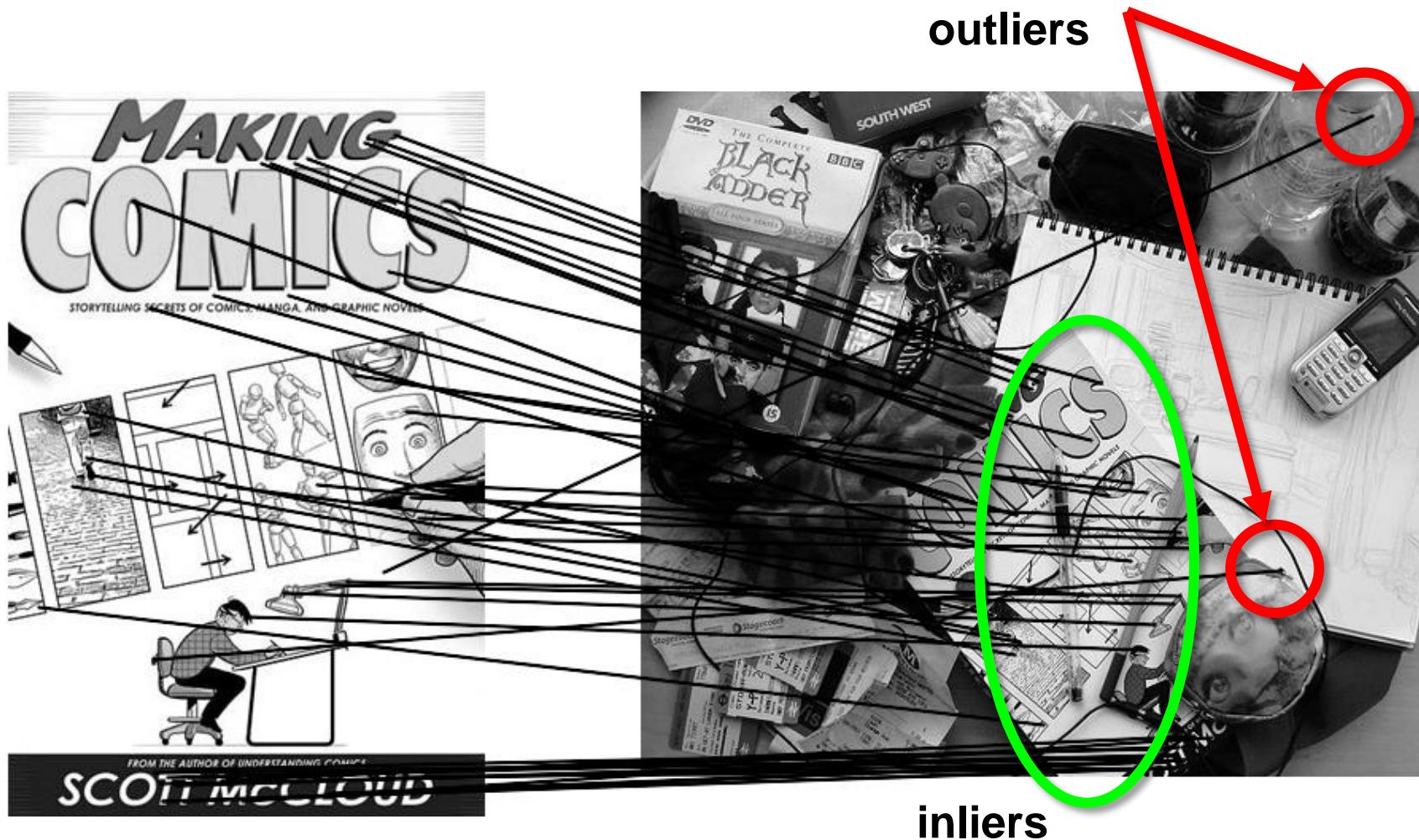
<https://dbloisi.github.io/corsi/lezionivep/sift.ipynb>



<https://dbloisi.github.io/corsi/lezionivep/orb.ipynb>

# Excluding outliers

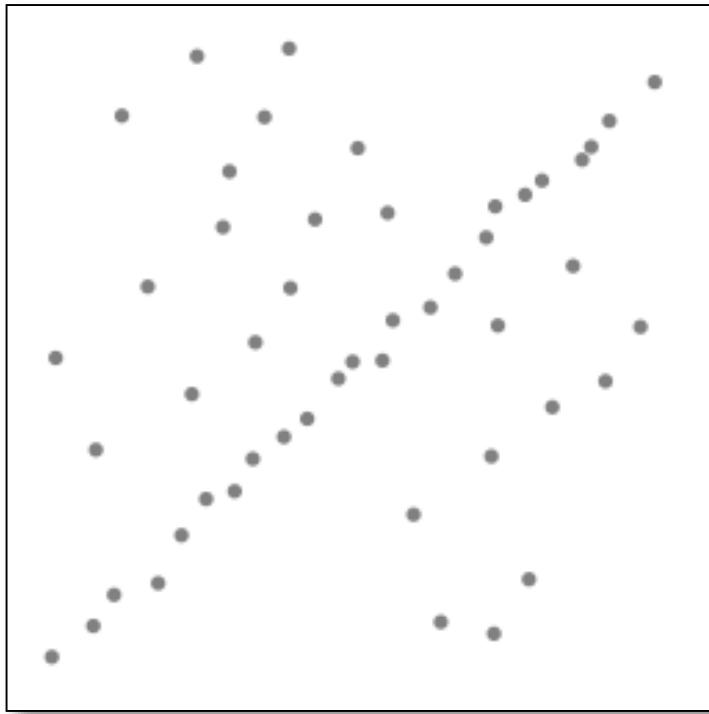
---



# Robustness

---

Problem: Fit a line to these datapoints



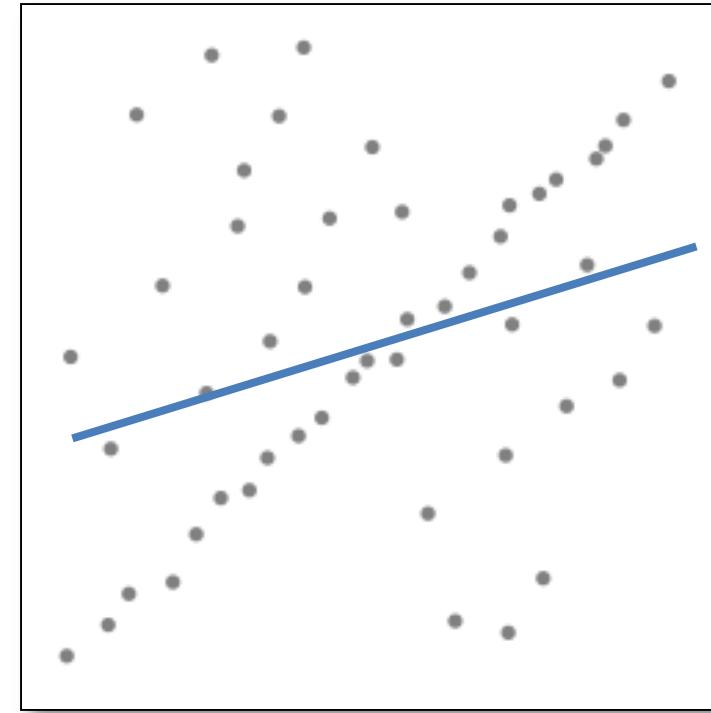
# Robustness

---

Problem: Fit a line to these datapoints



Least squares fit  
→

A large blue arrow pointing from the left scatter plot towards the right scatter plot, indicating the process of fitting a line to the data.

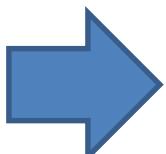
# Least Squares Fit

---

```
1 import numpy as np
2 import cv2 as cv
3 import math
4
5 # Read image
6 im = cv.imread("punti.png", cv.IMREAD_GRAYSCALE)
7
8 # Setup SimpleBlobDetector parameters.
9 params = cv.SimpleBlobDetector_Params()
10
11 # Change thresholds
12 params.minThreshold = 10;
13 params.maxThreshold = 200;
14
15 # Set up the detector with default parameters.
16 detector = cv.SimpleBlobDetector_create(params)
17
18 # Detect blobs.
19 keypoints = detector.detect(im)
```



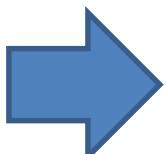
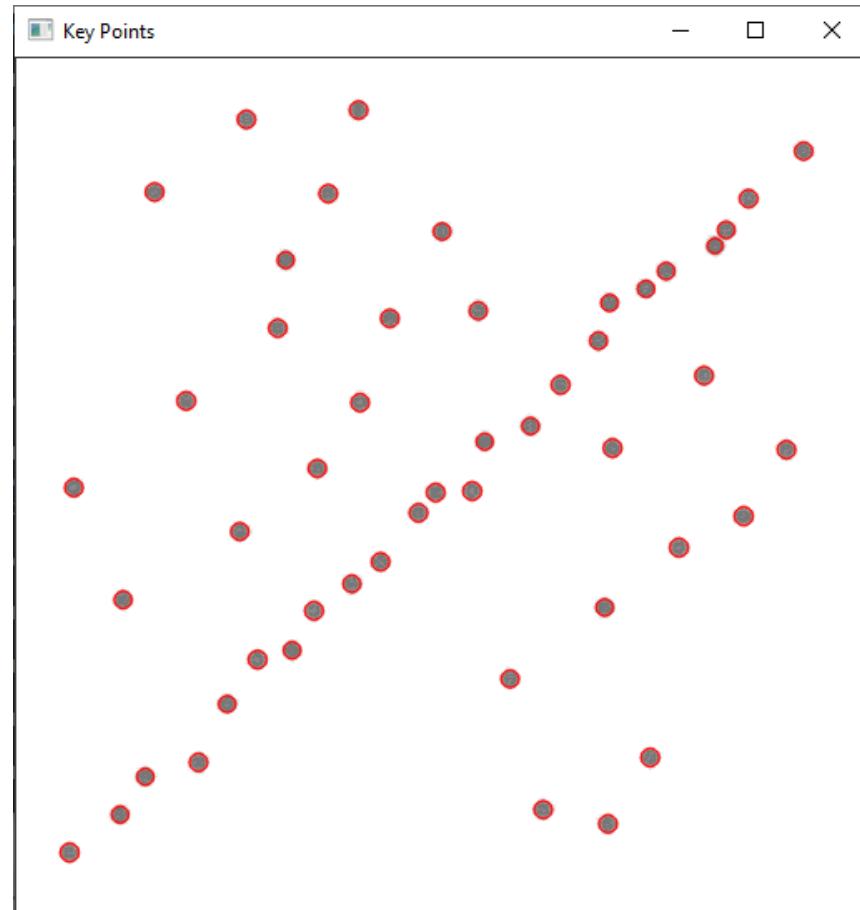
punti.png



# Least Squares Fit

---

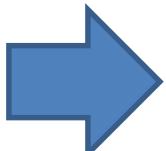
```
20
21     # Draw detected blobs as red circles.
22     # cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures the size of the circle corresponds to the size of blob
23     im_with_keypoints = cv.drawKeypoints(im, keypoints, np.array([]), (0,0,255), cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
24
25     cv.namedWindow("Key Points", cv.WINDOW_AUTOSIZE);
26     cv.imshow("Key Points", im_with_keypoints);
27     cv.waitKey(0);
```



# Least Squares Fit

---

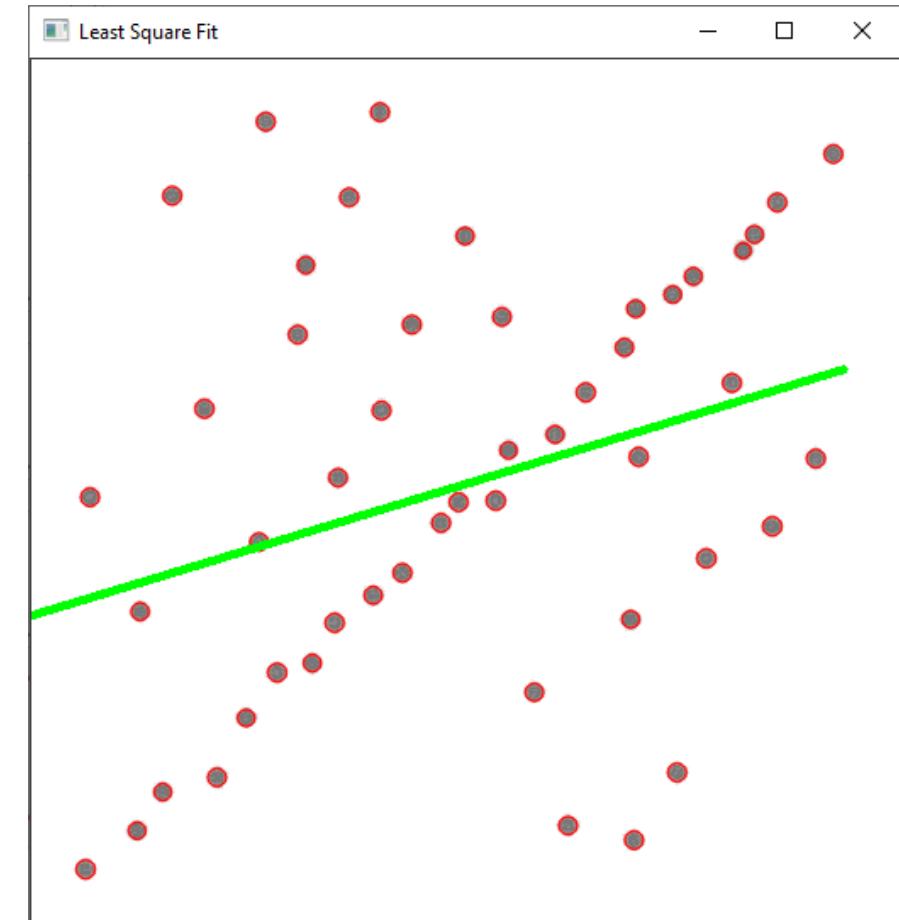
```
28
29     v = []
30     for elem in keypoints:
31         #print(elem.pt[0])
32         v.append([elem.pt[0],elem.pt[1]])
33     points = np.array(v)
34
35     y = points[:,0]
36     x = points[:,1]
37
38     m, c = np.polyfit(x, y, 1)      # calculate least square fit line
39
40     # calculate two coordinates (x1,y1), (x2,y2) on the line
41     angle = np.arctan(m)
42     x1, y1, length = 0, int(c), 500
43     x2 = int(round(math.ceil(x1 + length * np.cos(angle)),0))
44     y2 = int(round(math.ceil(y1 + length * np.sin(angle)),0))
```



# Least Squares Fit

---

```
46 # draw line on the color image
47 cv.line(im_with_keypoints, (x1, y1), (x2, y2), (0,255,0), 3, cv.LINE_8)
48
49 # show output the image
50 cv.namedWindow("Least Square Fit", cv.WINDOW_AUTOSIZE);
51 cv.imshow("Least Square Fit", im_with_keypoints);
52 cv.waitKey(0);
53 cv.destroyAllWindows()
```



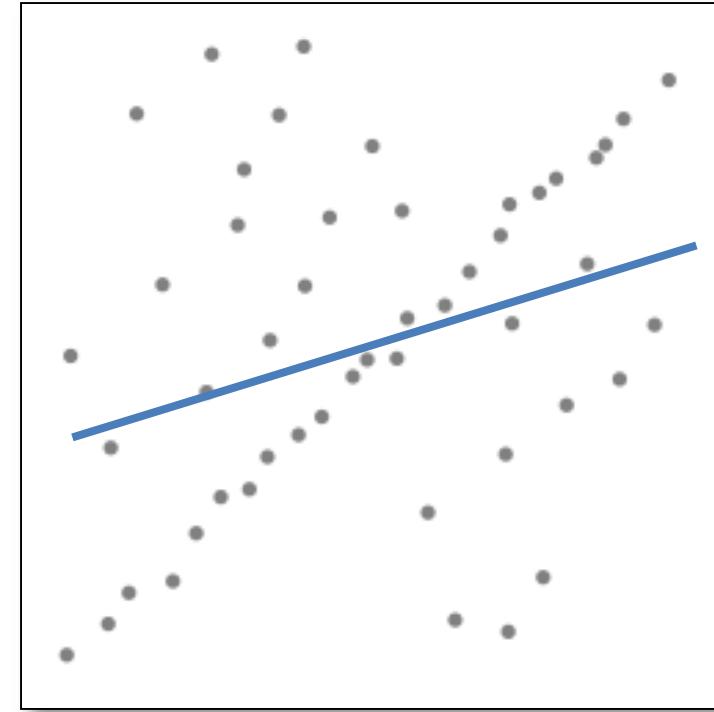
# Robustness

---

Problem: Fit a line to these datapoints



Least squares fit



**How can we fix this?**

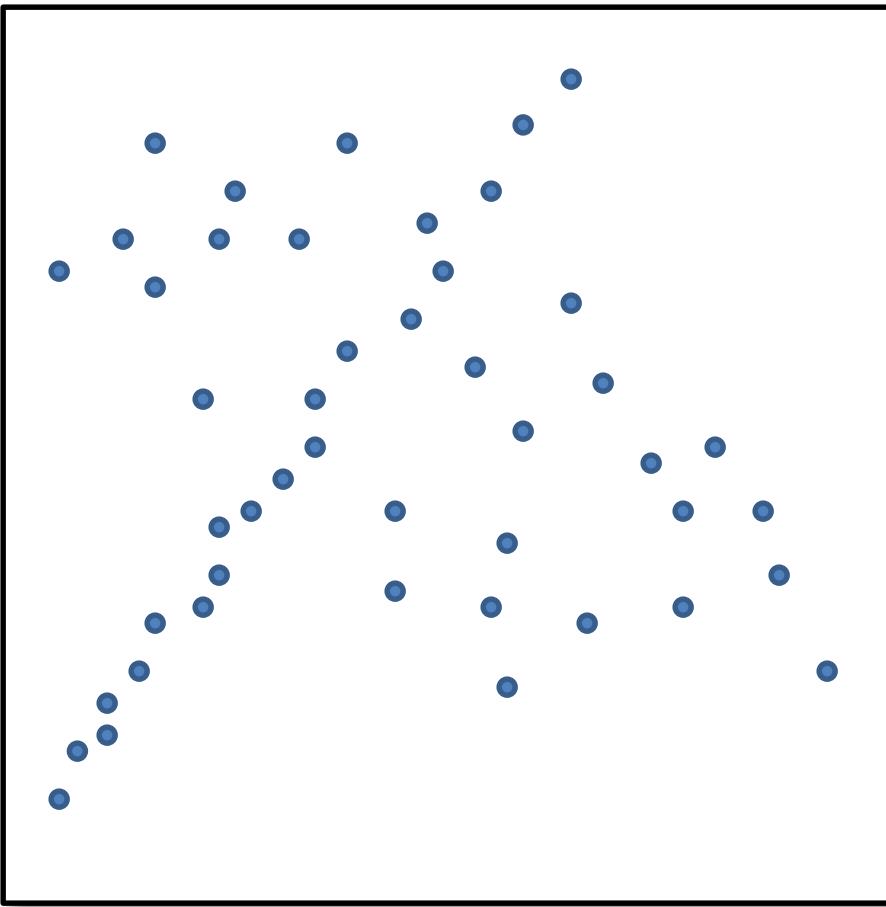
# Idea

---

- Given a hypothesized line
- Count the number of points that “agree” with the line
  - “Agree” = within a small distance of the line
  - I.e., the **inliers** to that line
- For all possible lines, select the one with the largest number of inliers

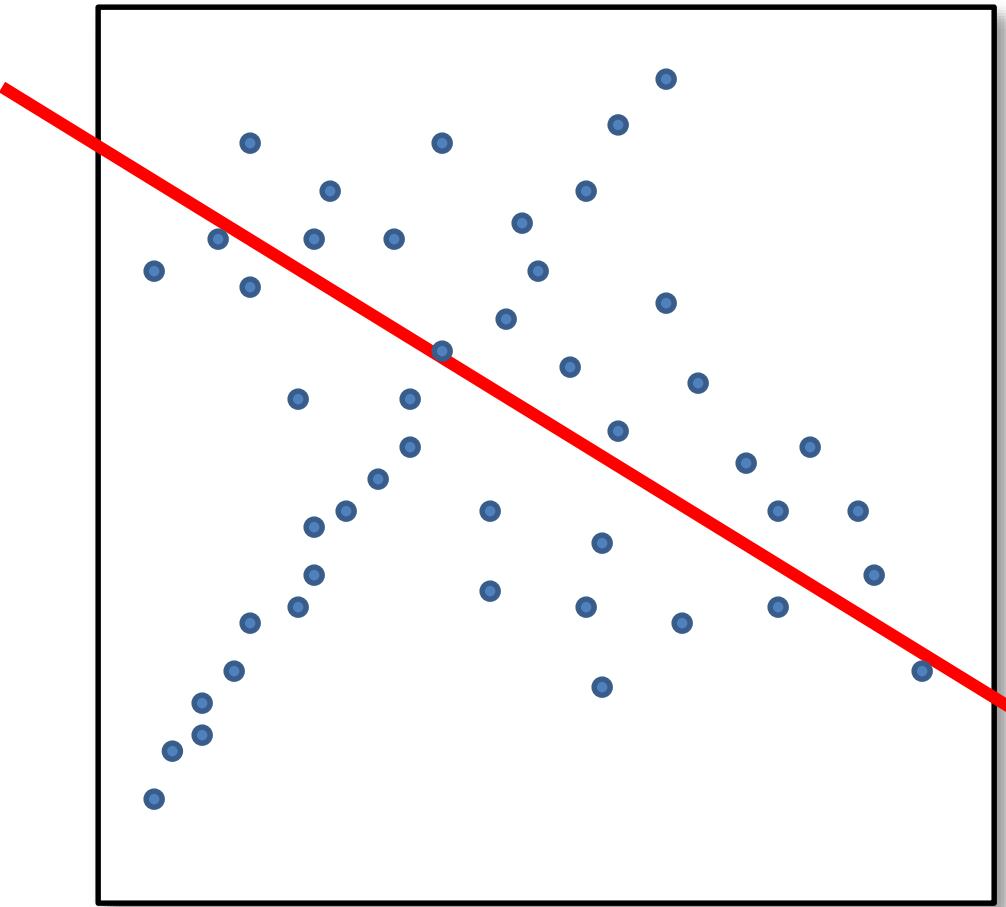
# Counting inliers

---



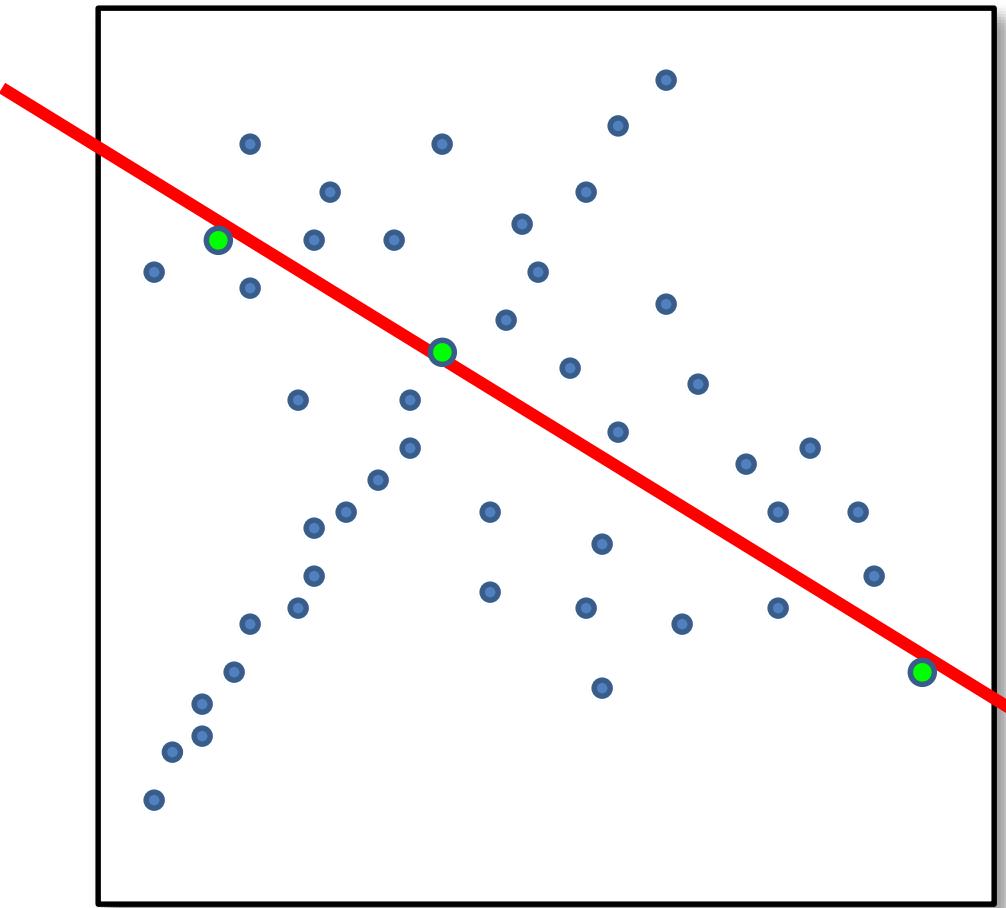
# Counting inliers

---



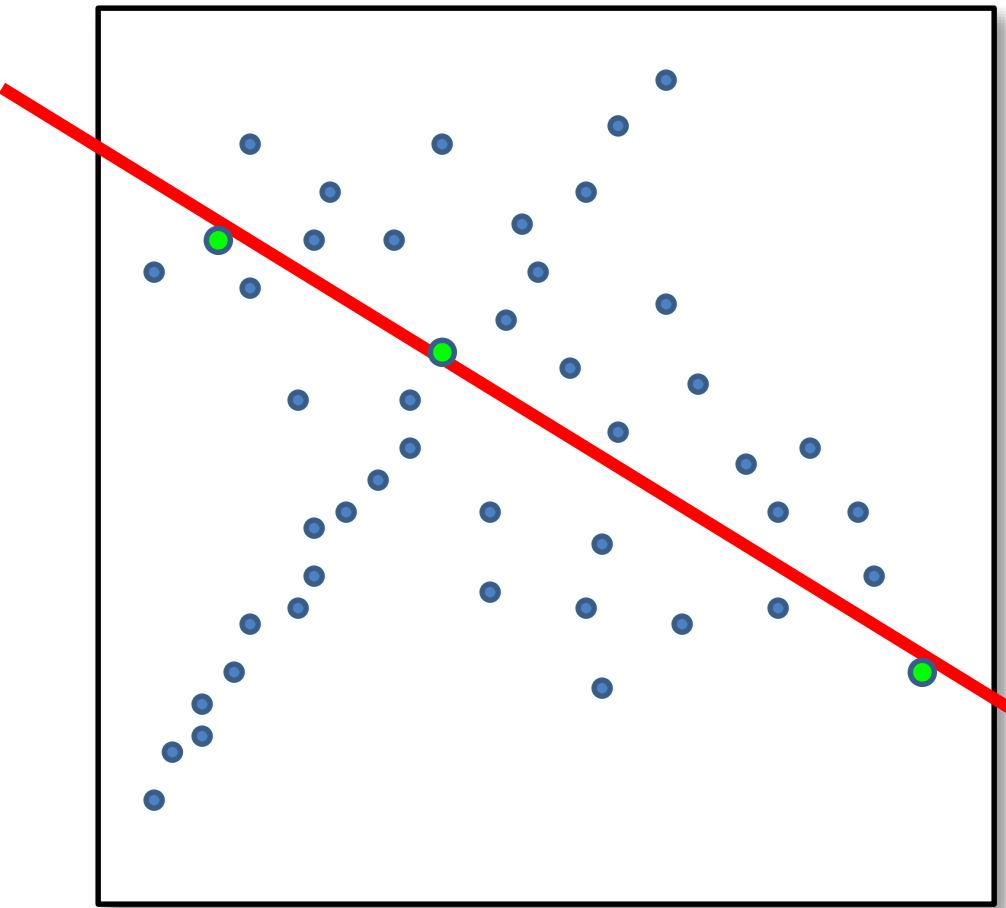
# Counting inliers

---



# Counting inliers

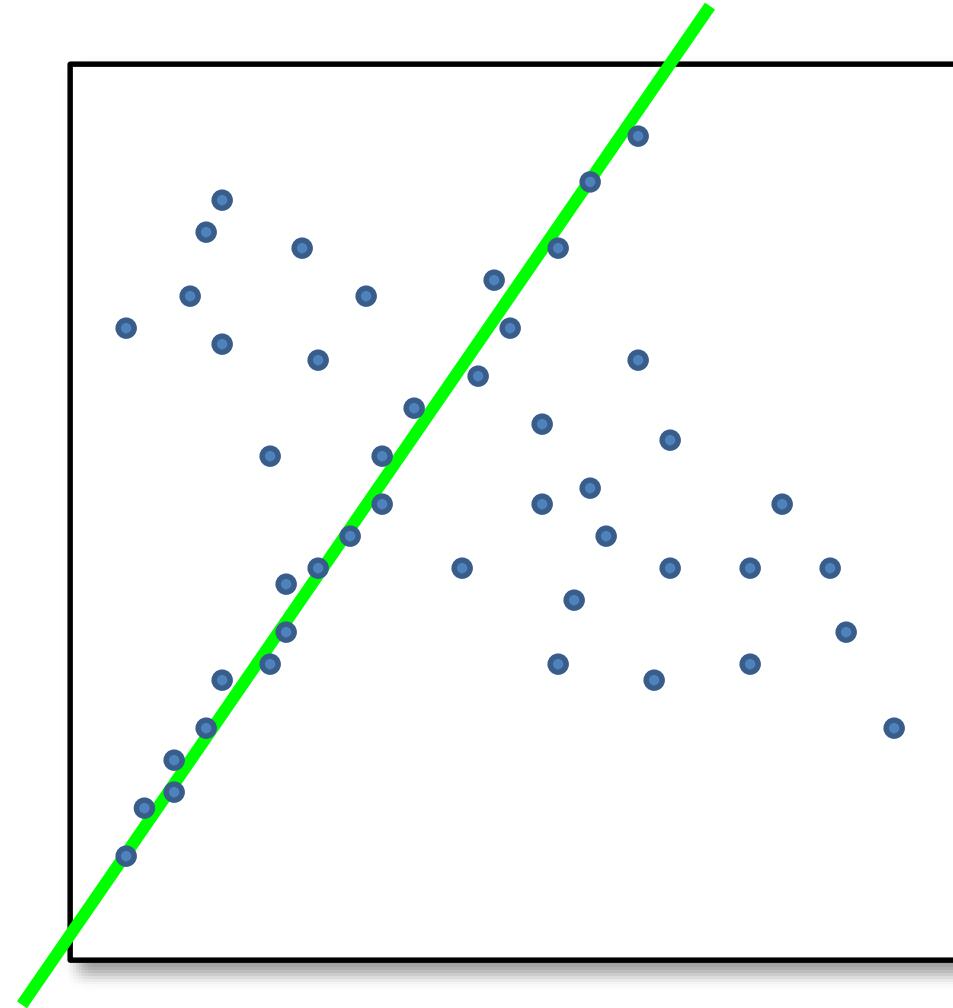
---



**Inliers: 3**

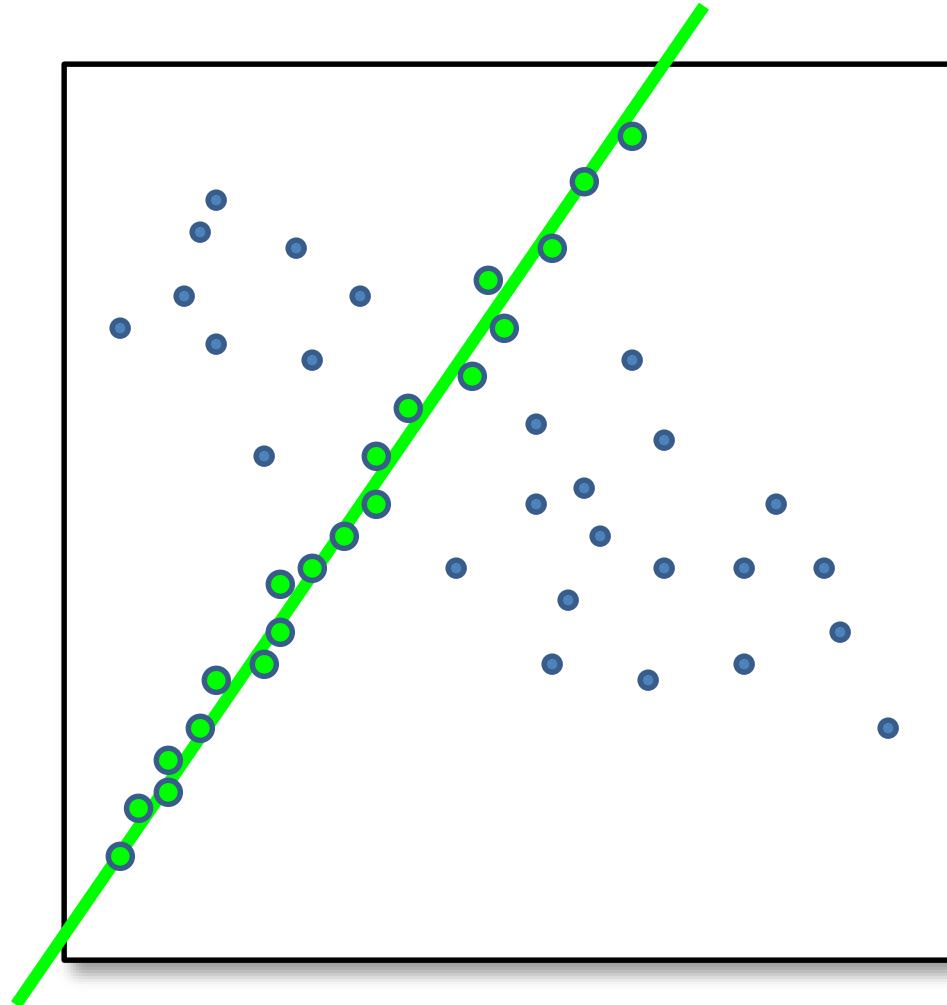
# Counting inliers

---



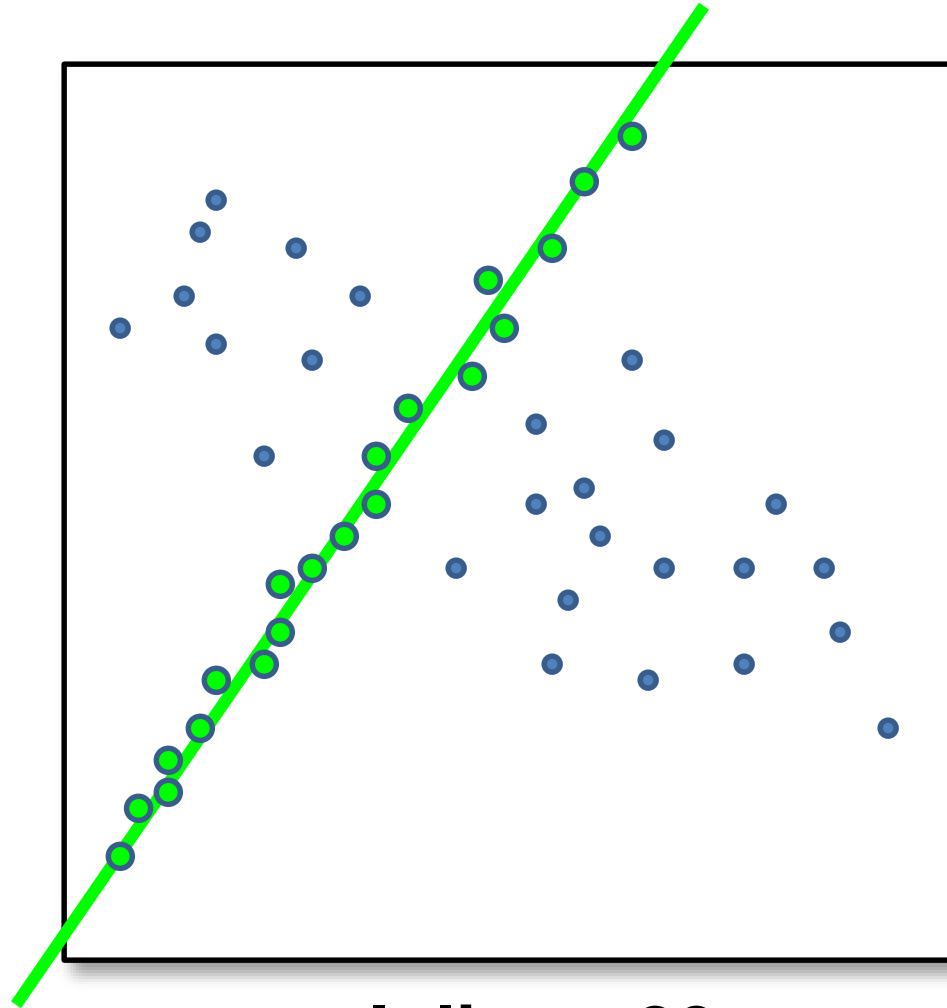
# Counting inliers

---



# Counting inliers

---



# How do we find the best line?

---

- Unlike least-squares, no simple closed-form solution
- Hypothesize-and-test
  - Try out many lines, keep the best one
  - Which lines?

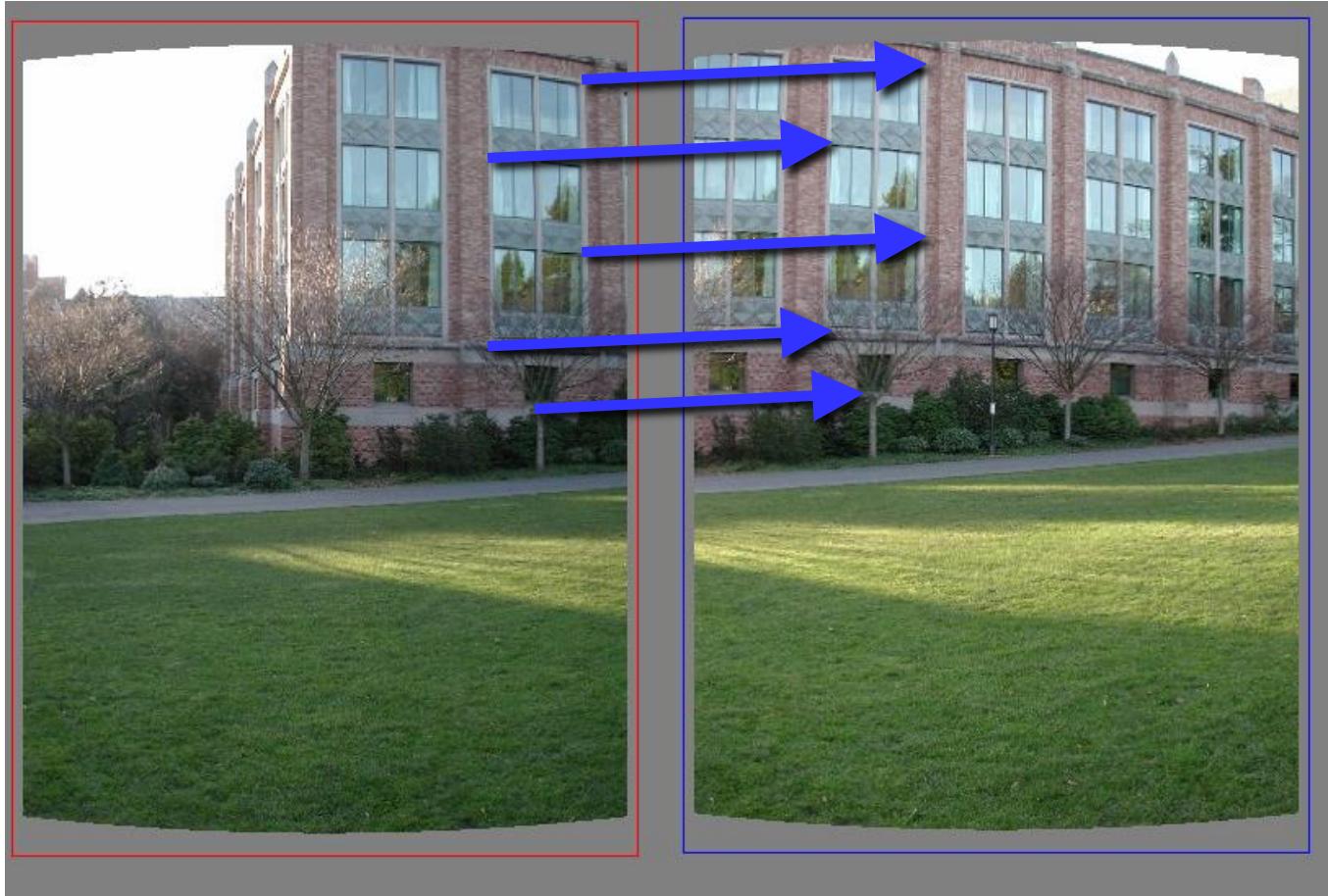
# Translations

---



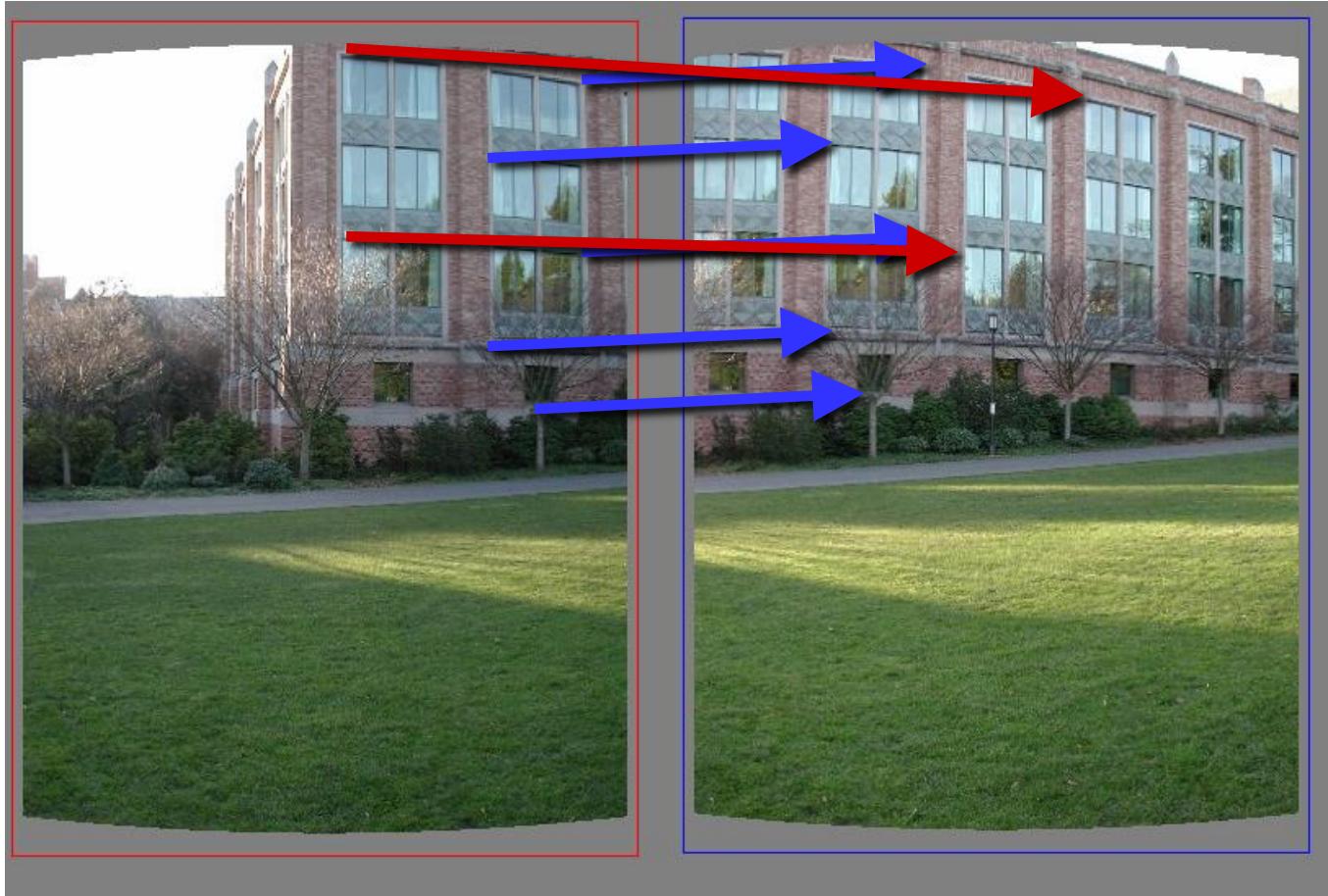
# Translations

---



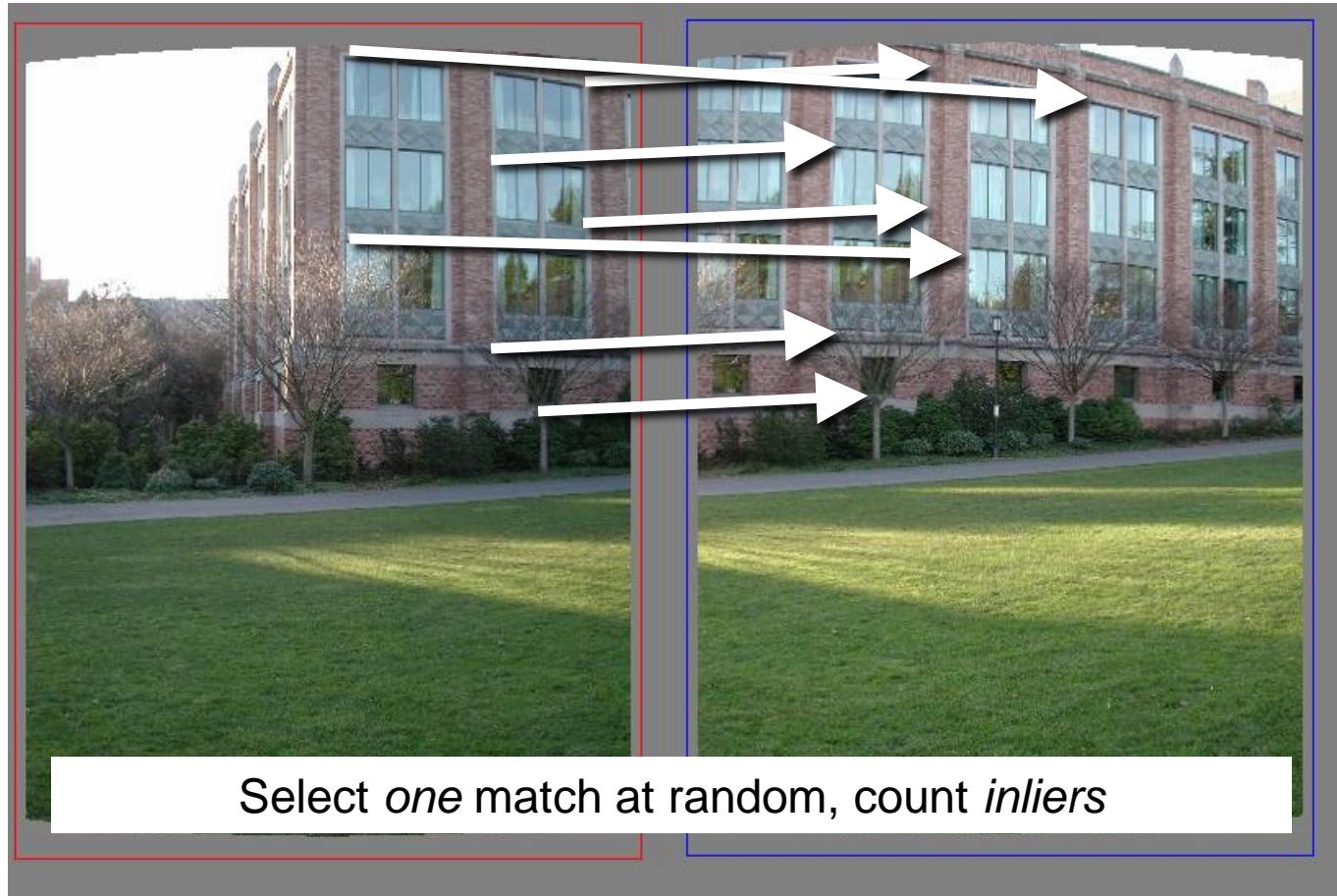
# Translations

---



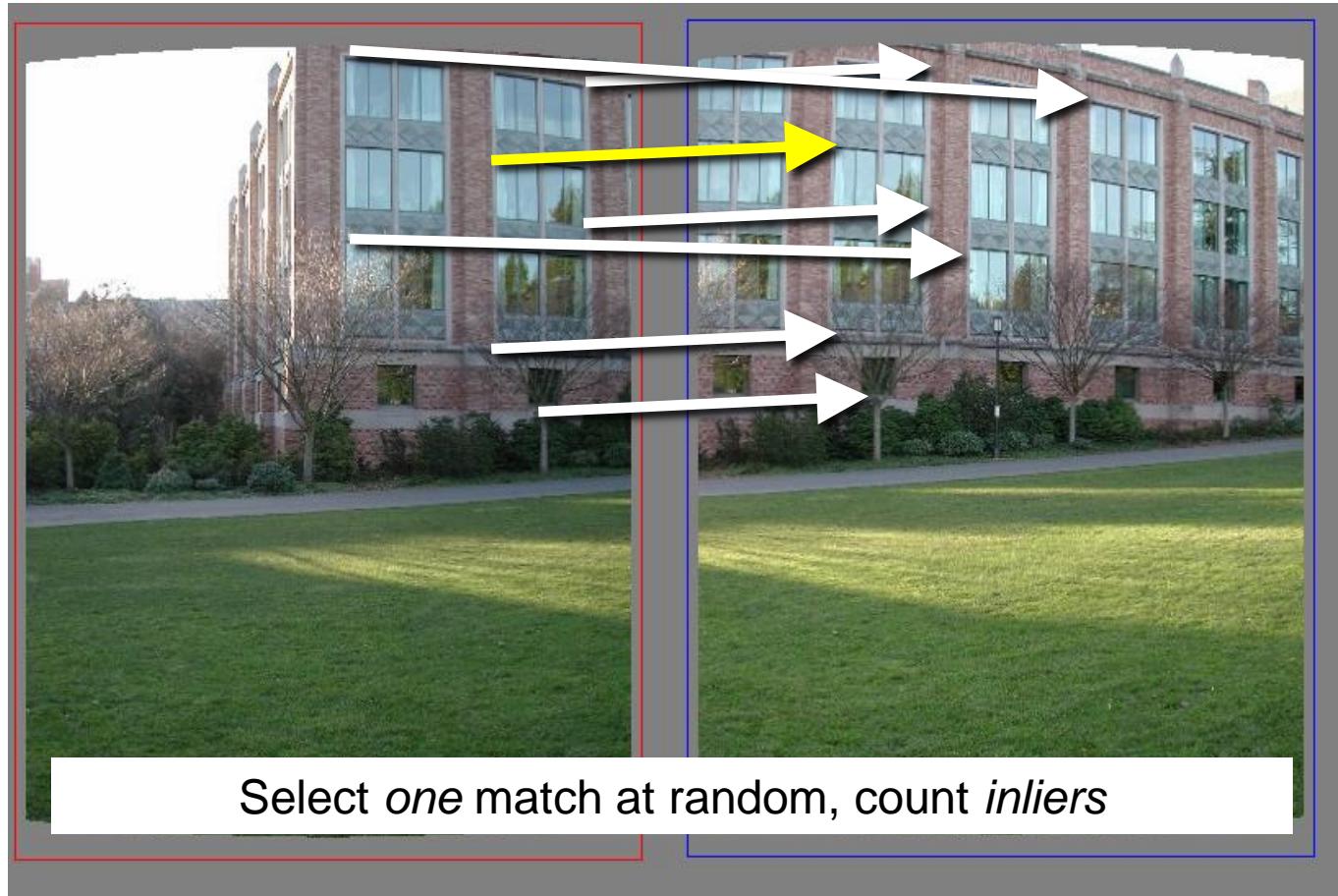
# RANdom SAmple Consensus

---



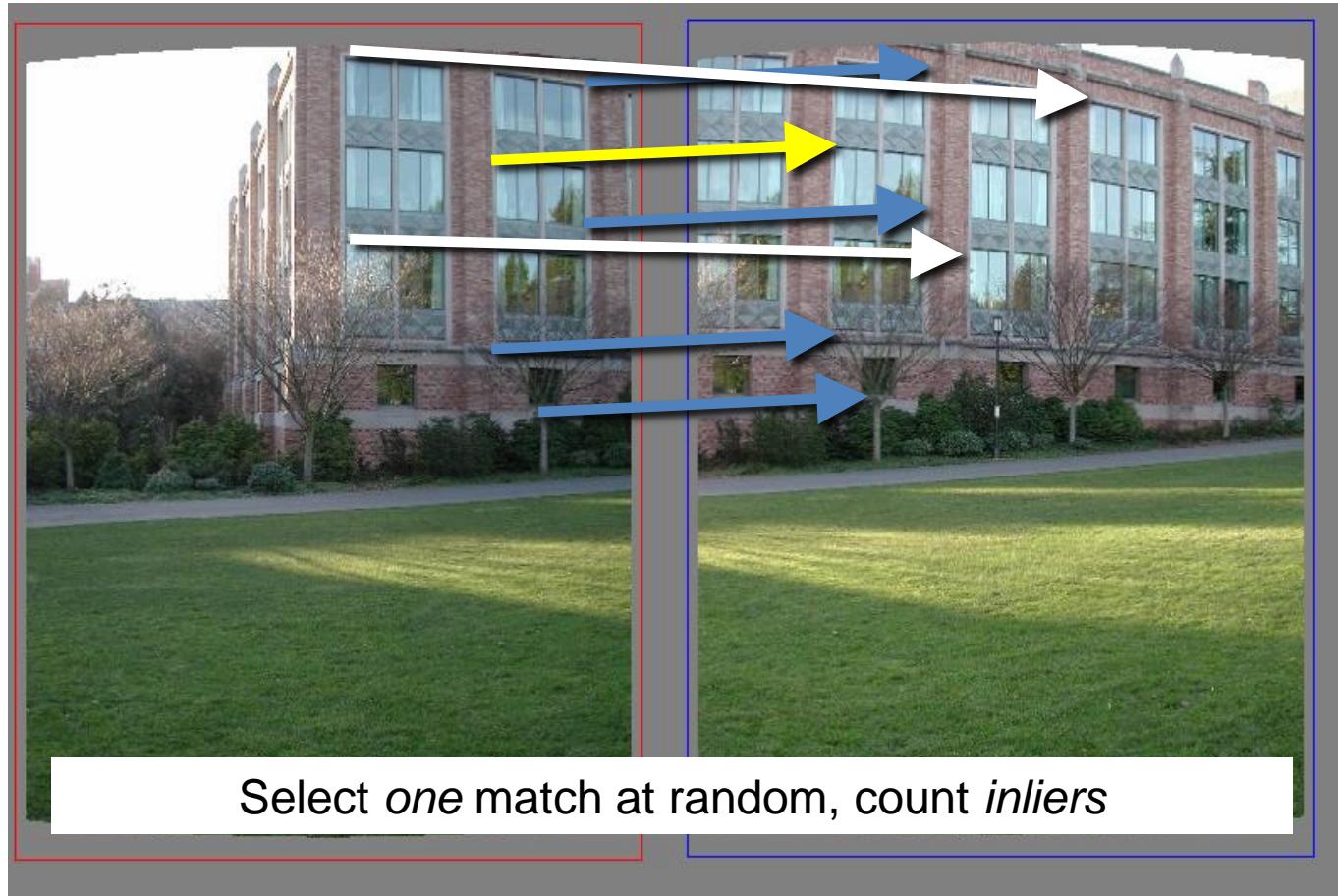
# RANdom SAmple Consensus

---



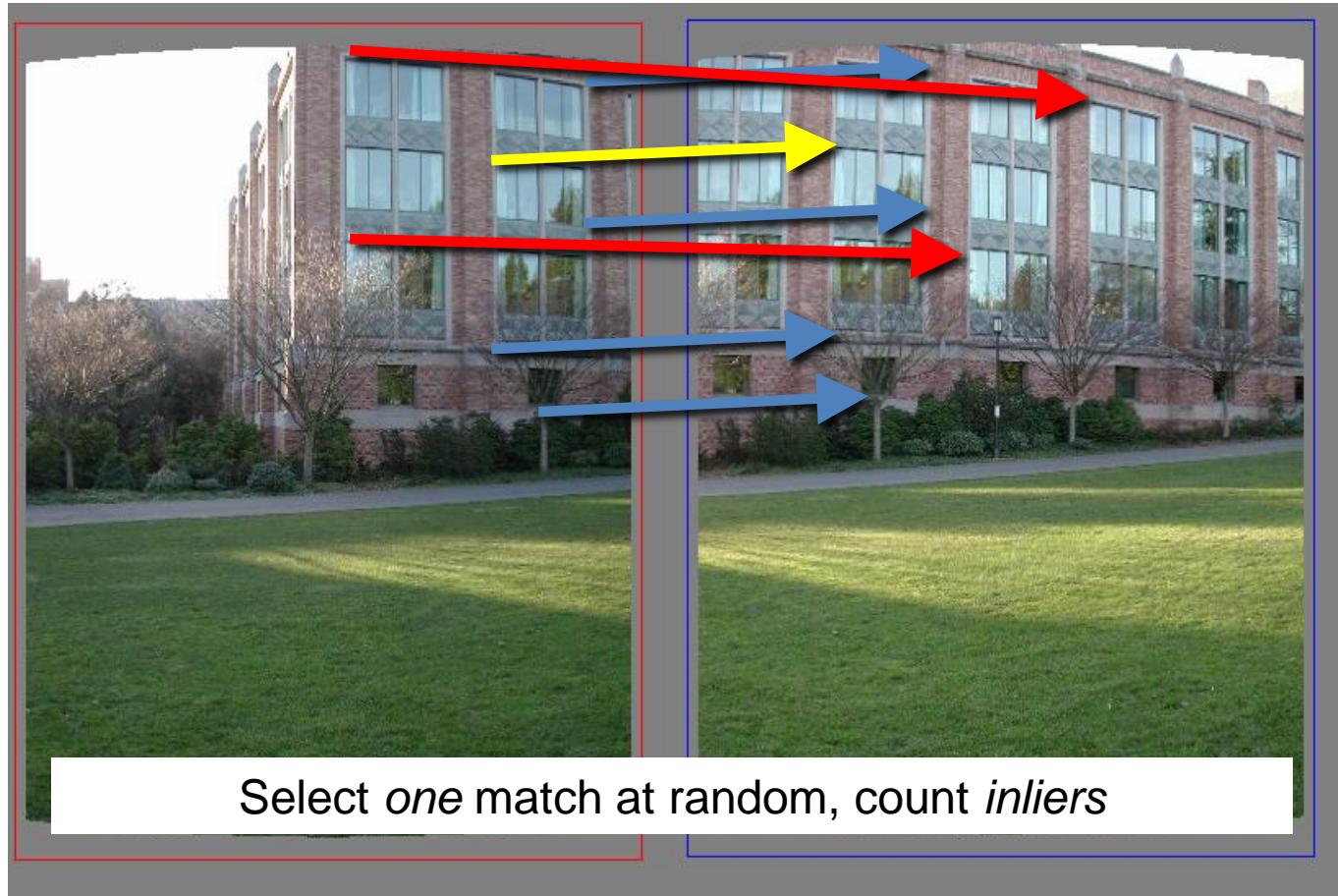
# RANdom SAmple Consensus

---



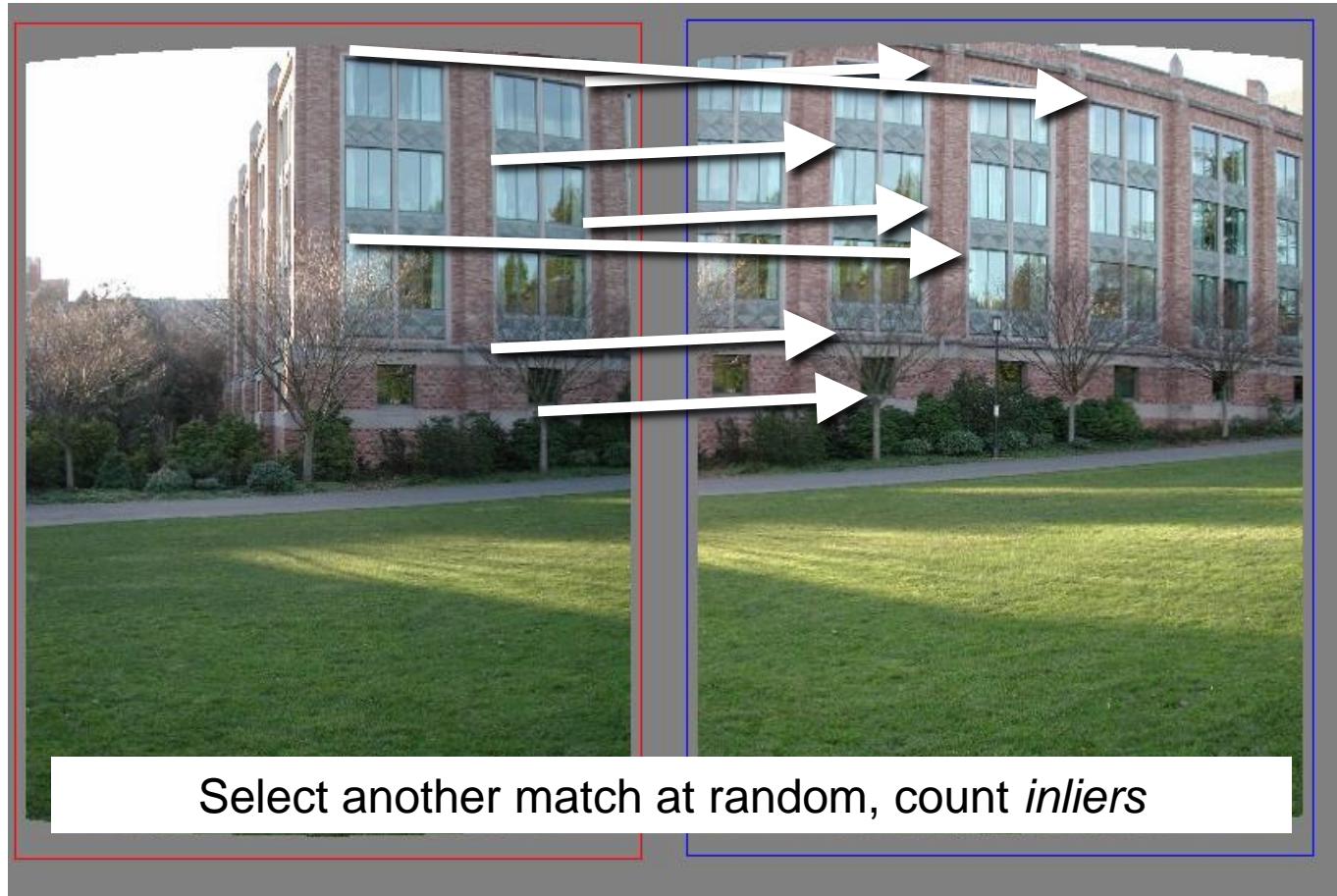
# RANdom SAmple Consensus

---



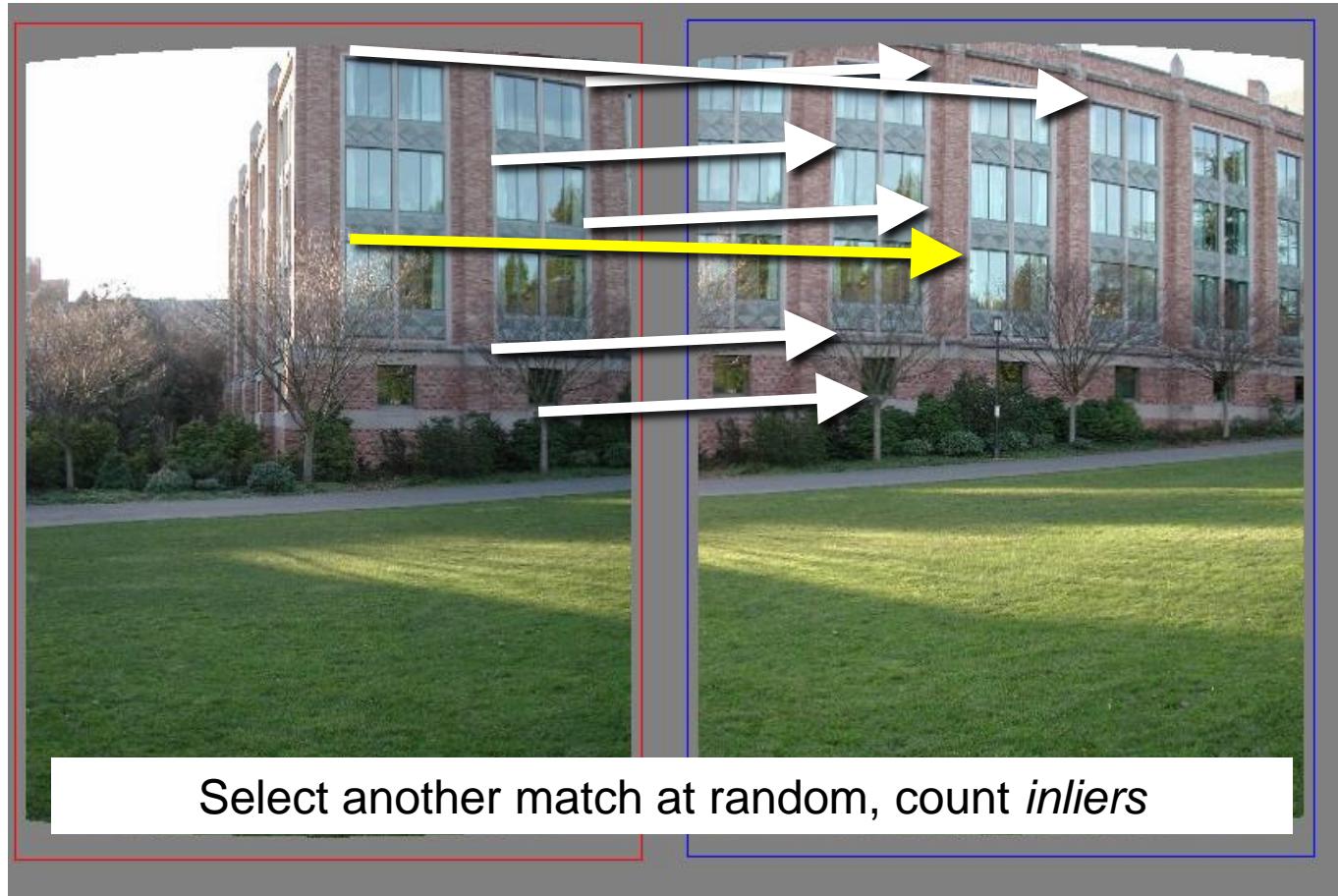
# RANdom SAmple Consensus

---



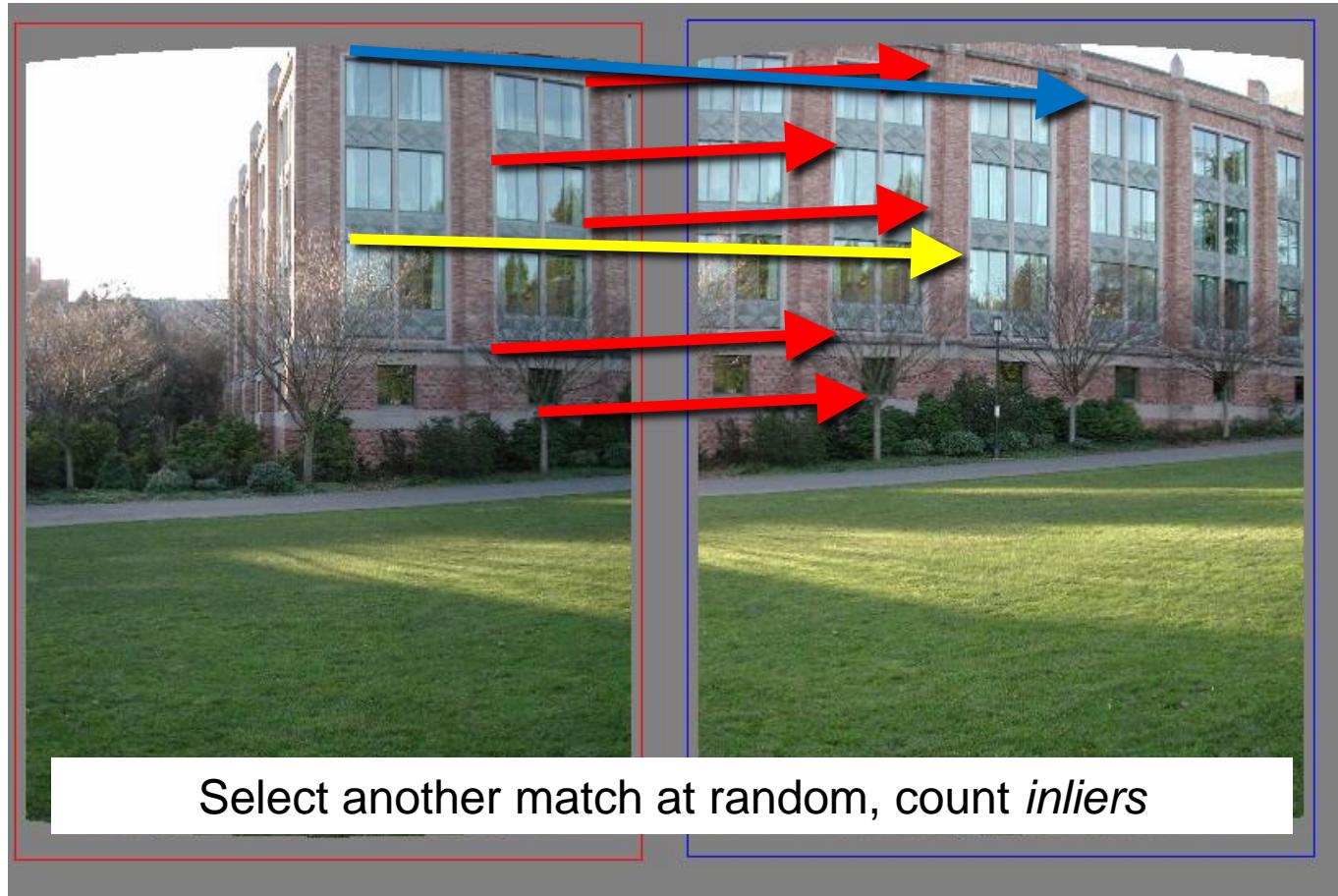
# RANdom SAmple Consensus

---



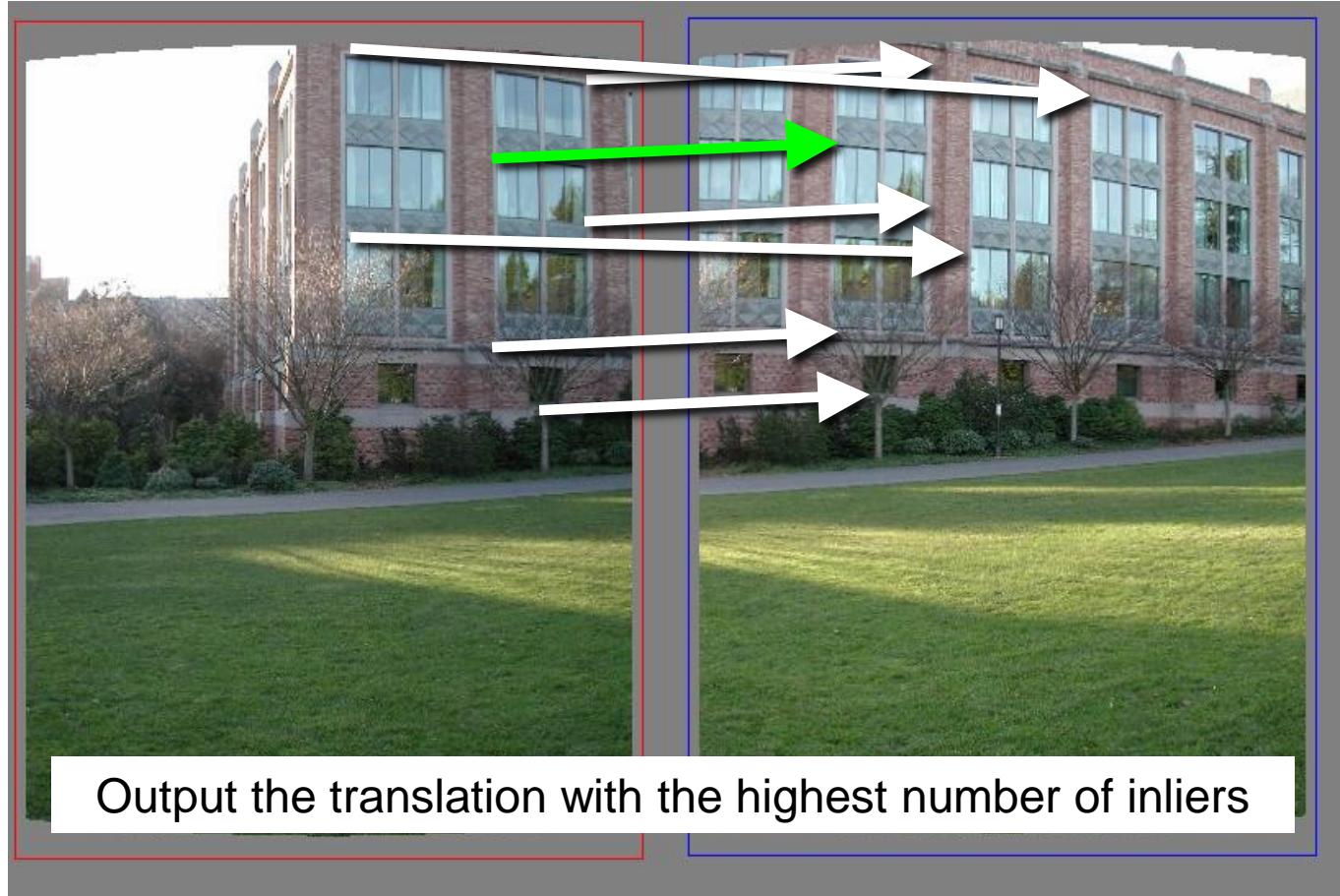
# RANdom SAmple Consensus

---



# RANdom SAmple Consensus

---



# RANSAC

---

- **Inlier threshold** related to the amount of noise we expect in inliers
  - Often model noise as Gaussian w/ some standard deviation (e.g. 3 pixels)
- **Number of rounds** related to the percentage of outliers we expect, and the probability of success we'd like to guarantee
  - Suppose there are 20% outliers, and we want to find the correct answer with 99% probability
  - How many rounds do we need?

# RANSAC pros and cons

---

- **Pros**

- Simple and general
- Applicable to many different problems
- Often works well in practice

- **Cons**

- Parameters to tune
- Sometimes too many iterations are required
- Can fail for extremely low inlier ratios
- We can often do better than brute-force sampling

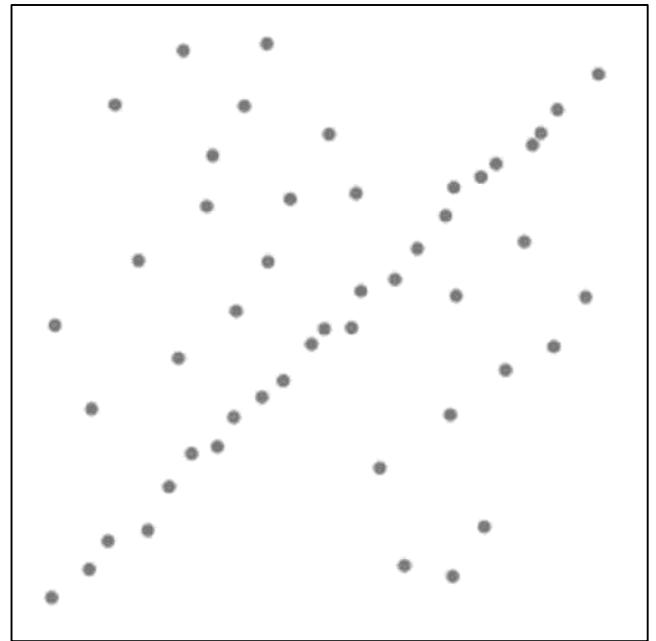
# RANSAC

---

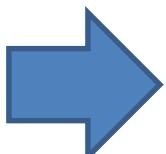
- Idea:
  - All the inliers will agree with each other on the translation vector; the (hopefully small) number of outliers will (hopefully) disagree with each other
- RANSAC only has guarantees if there are < 50% outliers

# Fitline OpenCV

```
1 import cv2 as cv
2 import numpy as np;
3
4 # Read image
5 im = cv.imread("punti.png", cv.IMREAD_GRAYSCALE)
6
7 # Setup SimpleBlobDetector parameters.
8 params = cv.SimpleBlobDetector_Params()
9
10 # Change thresholds
11 params.minThreshold = 10;
12 params.maxThreshold = 200;
13
14 # Set up the detector with default parameters.
15 detector = cv.SimpleBlobDetector_create(params)
16
17 # Detect blobs.
18 keypoints = detector.detect(im)
19
20 v = []
21 for elem in keypoints:
22     #print(elem.pt[0])
23     v.append([elem.pt[0], elem.pt[1]])
24
25 points = np.array(v)
```

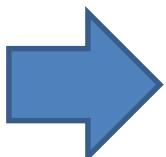
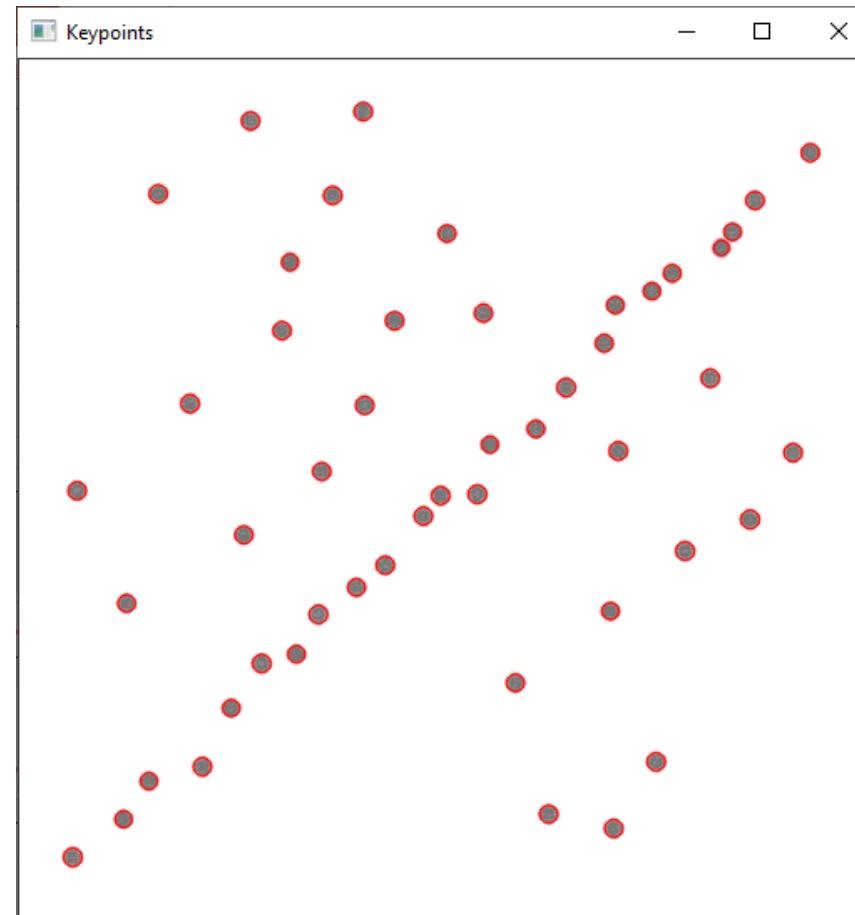


punti.png



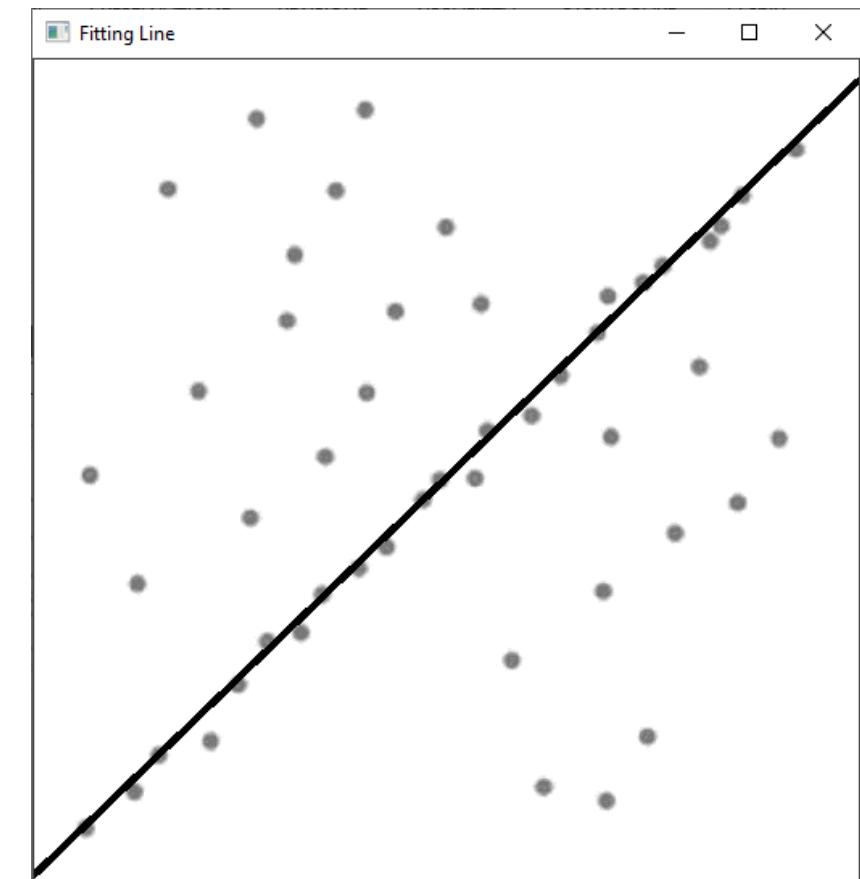
# Fitline OpenCV

```
26      # Draw detected blobs as red circles.  
27      # cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures the size of the circle corresponds to the size of blob  
28      im_with_keypoints = cv.drawKeypoints(im, keypoints, np.array([]), (0,0,255), cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)  
29  
30      # Show keypoints  
31      cv.imshow("Keypoints", im_with_keypoints)  
32      cv.waitKey(0)
```



# Fitline OpenCV

```
31 # Show keypoints  
32 cv.imshow("Keypoints", im_with_keypoints)  
33 cv.waitKey(0)  
34  
35 vx, vy, x, y = cv.fitLine(np.float32(points),cv.DIST_L12, 0, 0.01, 0.01);  
36  
37 line = [float(vx),float(vy),float(x),float(y)]  
38  
39 left_pt = int((-x*vy/vx) + y)  
40 right_pt = int(((im.shape[1]-x)*vy/vx)+y)  
41 cv.line(im,(im.shape[1]-1,right_pt),(0,left_pt),0,3,cv.LINE_8)  
42  
43 # Show keypoints  
44 cv.imshow("Fitting Line", im)  
45 cv.waitKey(0)
```



# Panoramas

---

- Now we know how to create panoramas!

- Given two images:

- Step 1: Detect features
- Step 2: Match features
- Step 3: Compute a homography using RANSAC
- Step 4: Combine the images together (somehow)



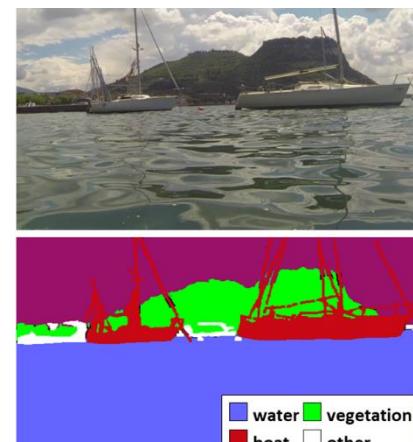
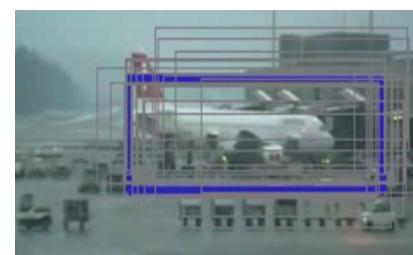
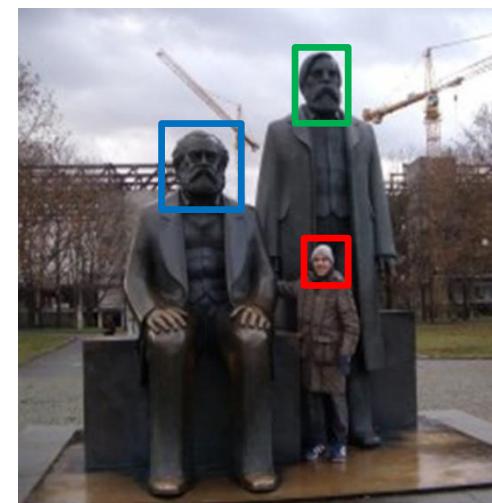
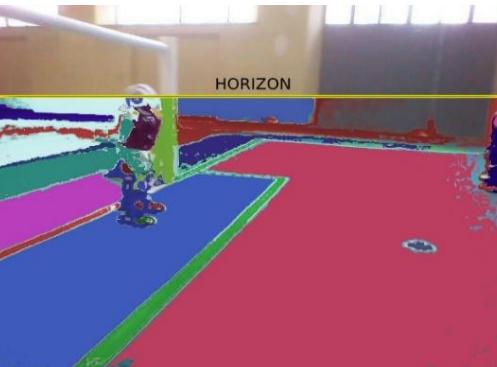
What if we have more than two images?



UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA

# *Corso di Visione e Percezione*

# Feature Matching



Docente  
Domenico D. Bloisi