

Image Correspondence

Sunglok Choi, Assistant Professor, Ph.D.
Computer Science and Engineering Department, SEOULTECH
sunglok@seoultech.ac.kr | <https://mint-lab.github.io/>

Table of Contents: Finding Correspondence

▪ Feature Points

- Q) How can we detect salient points (or parts)?

▪ Feature Descriptors

- Q) How can we distinguish the points each other?

▪ Feature Matching

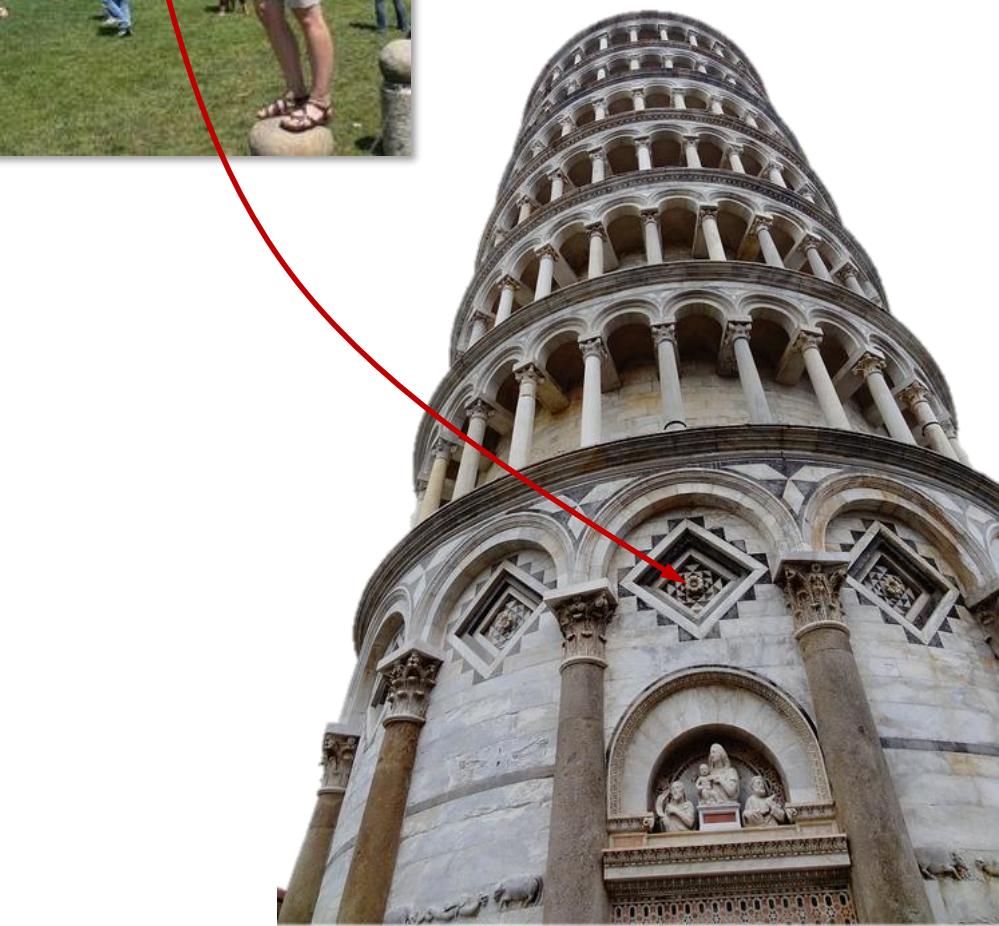
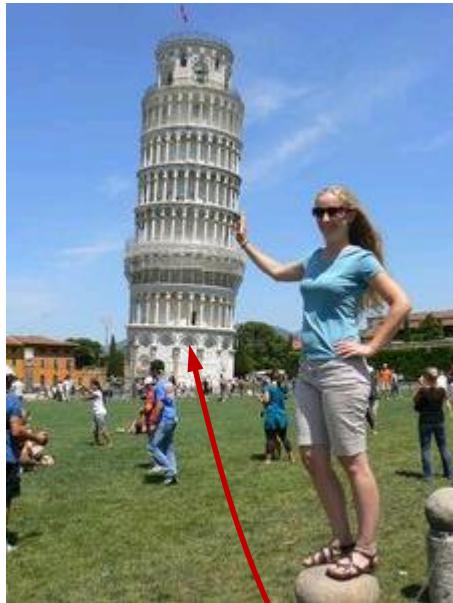
- Q) How can we associate the points across different images?

▪ Feature Tracking

- Q) How can we associate the points across their next image?

▪ Outlier Rejection

- Q) How can we select correctly matched points?



Getting Started with a Quiz

- Q) What is it?



600 pixels

Getting Started with a Quiz

- Q) What is it?



600 pixels

Getting Started with a Quiz

- Q) What is it? “Duck”



600 pixels



1095 pixels



600 pixels

- Q) Why corners (junction) instead of edges or blobs?

- Human visual systems understand objects better from corners than edges.
 - a.k.a. *feature points*, *keypoints*, *salient* points, and *interest* points

Getting Started with a Quiz

- Q) Why corners (junction) instead of edges or blobs? Where are the image patches?

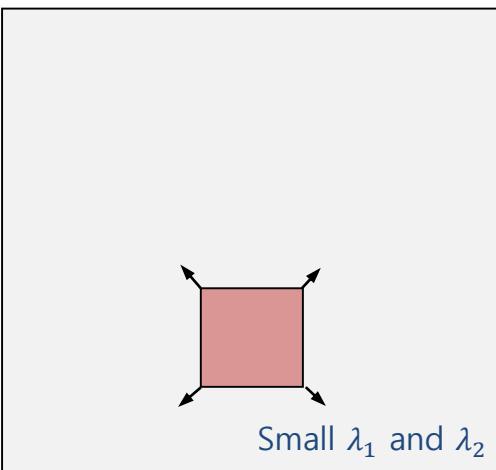


- Q) What is a good feature? (Requirements)

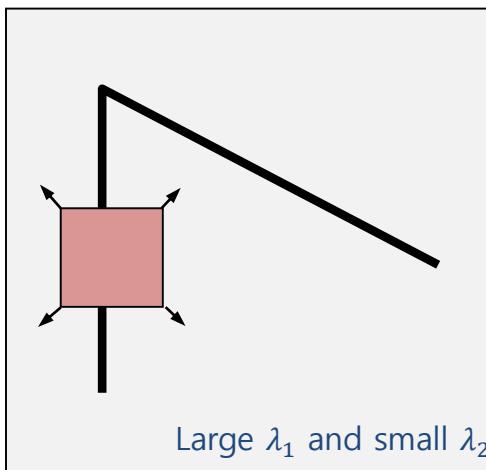
- **Repeatability** (to be invariance/robustness to transformation and noise)
- **Distinctiveness** (to be easy to distinguish or match)
- **Locality** (due to occlusion)
- Quantity (sufficient number), accuracy (localization), efficiency (computing time), ...

Feature Point) Harris Corner (1988)

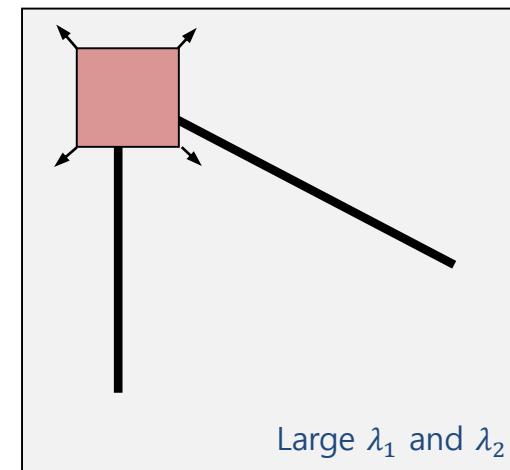
- Key idea: **Sliding window**



"flat" region:
no change
in all directions



"edge":
no change
along the edge direction



"corner":
significant change
in all directions

- Formulation

- $I(x + \Delta_x, y + \Delta_y) \approx I(x, y) + [I_x(x, y) \quad I_y(x, y)] \begin{bmatrix} \Delta_x \\ \Delta_y \end{bmatrix}$ where $I_x = \frac{\partial I}{\partial x}$ and $I_y = \frac{\partial I}{\partial y}$
- $D(\Delta_x, \Delta_y) = \sum_{(x,y) \in W} (I(x + \Delta_x, y + \Delta_y) - I(x, y))^2 \approx [\Delta_x \quad \Delta_y] \begin{bmatrix} \sum_W I_x^2 & \sum_W I_x I_y \\ \sum_W I_x I_y & \sum_W I_y^2 \end{bmatrix} \begin{bmatrix} \Delta_x \\ \Delta_y \end{bmatrix}$

M

Feature Point) Harris Corner (1988)

- Key idea: **Sliding window**

- Formulation

$$\bullet \quad I(x + \Delta_x, y + \Delta_y) \approx I(x, y) + [I_x(x, y) \quad I_y(x, y)] \begin{bmatrix} \Delta_x \\ \Delta_y \end{bmatrix} \text{ where } I_x = \frac{\partial I}{\partial x} \text{ and } I_y = \frac{\partial I}{\partial y}$$

$$\bullet \quad D(\Delta_x, \Delta_y) = \sum_{(x,y) \in W} (I(x + \Delta_x, y + \Delta_y) - I(x, y))^2 \approx [\Delta_x \quad \Delta_y] \begin{bmatrix} \sum_W I_x^2 & \sum_W I_x I_y \\ \sum_W I_x I_y & \sum_W I_y^2 \end{bmatrix} \begin{bmatrix} \Delta_x \\ \Delta_y \end{bmatrix}$$

M

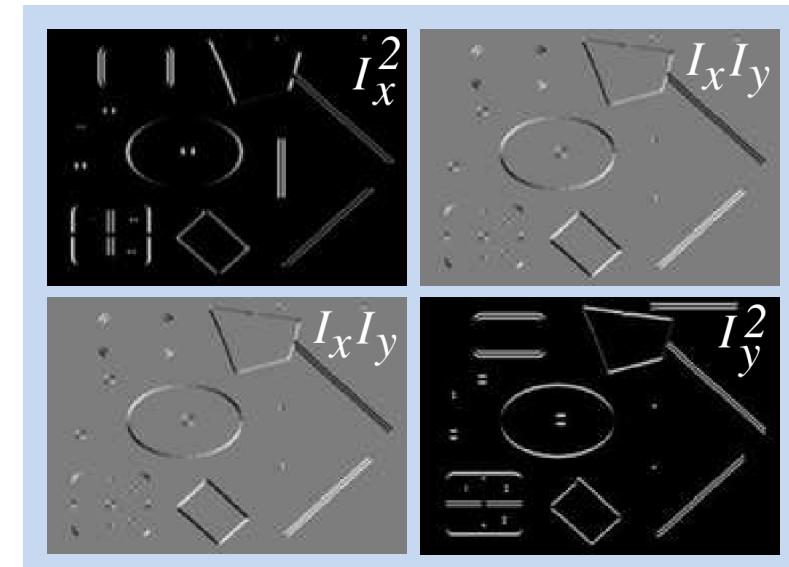
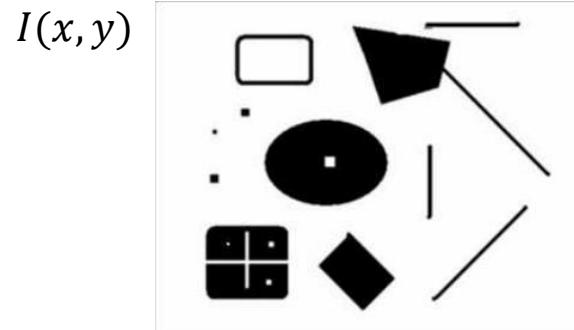
- Harris corner response**

$$\bullet \quad \text{cornerness} = \det(M) - k \text{ trace}(M)^2$$

$$\bullet \quad \text{Note) } \det(M) = \lambda_1 \lambda_2, \text{ trace}(M) = \lambda_1 + \lambda_2, \text{ and } k \in [0.04, 0.06]$$

- Note) Good-Feature-to-Track** (a.k.a. GFTT or Shi-Tomasi corner; 1994)

$$\bullet \quad \text{cornerness} = \min(\lambda_1, \lambda_2)$$



M



$$\det(M) - k \text{ trace}(M)^2$$

Feature Point) Harris Corner (1988)

- Example) **Harris corner implementation** [harris_corner_implement.py]

```
def cornerHarris(img, ksize=3, k=0.04):
    # Compute gradients and M matrix
    Ix = cv.Sobel(img, cv.CV_32F, 1, 0)
    Iy = cv.Sobel(img, cv.CV_32F, 0, 1)
    M11 = cv.GaussianBlur(Ix*Ix, (ksize, ksize), 0)
    M22 = cv.GaussianBlur(Iy*Iy, (ksize, ksize), 0)
    M12 = cv.GaussianBlur(Ix*Iy, (ksize, ksize), 0)

    # Compute Harris cornerness
    detM = M11 * M22 - M12 * M12
    traceM = M11 + M22
    cornerness = detM - k * traceM**2
    return cornerness
```

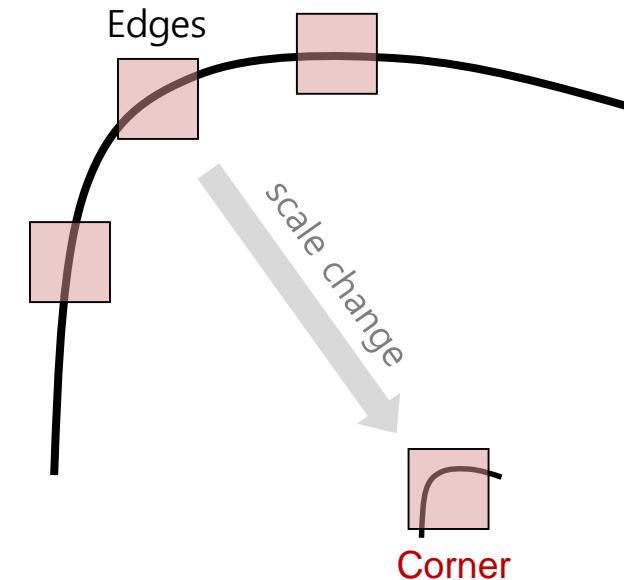
$$M = \begin{bmatrix} \sum_w I_x^2 & \sum_w I_x I_y \\ \sum_w I_x I_y & \sum_w I_y^2 \end{bmatrix}$$

$$\text{cornerness} = \det(M) - k \operatorname{trace}(M)^2$$



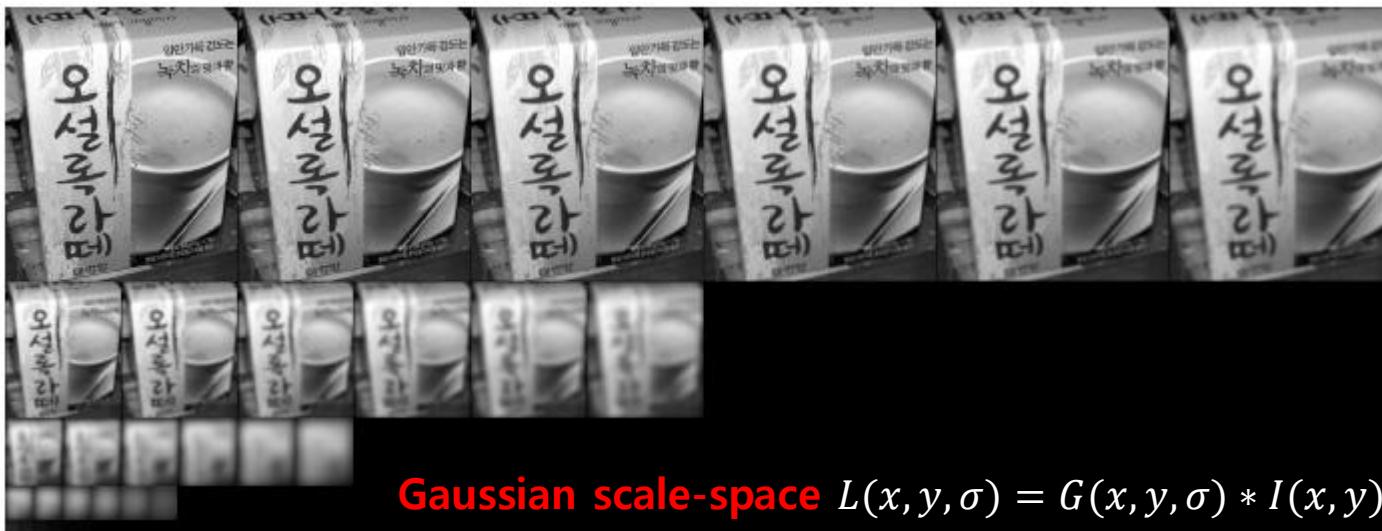
Feature Point) Harris Corner (1988)

- Properties
 - Invariant to translation, rotation, and intensity shift ($I \rightarrow I + b$) ~~intensity scaling ($I \rightarrow aI$)~~
 - Variant to **image scaling**

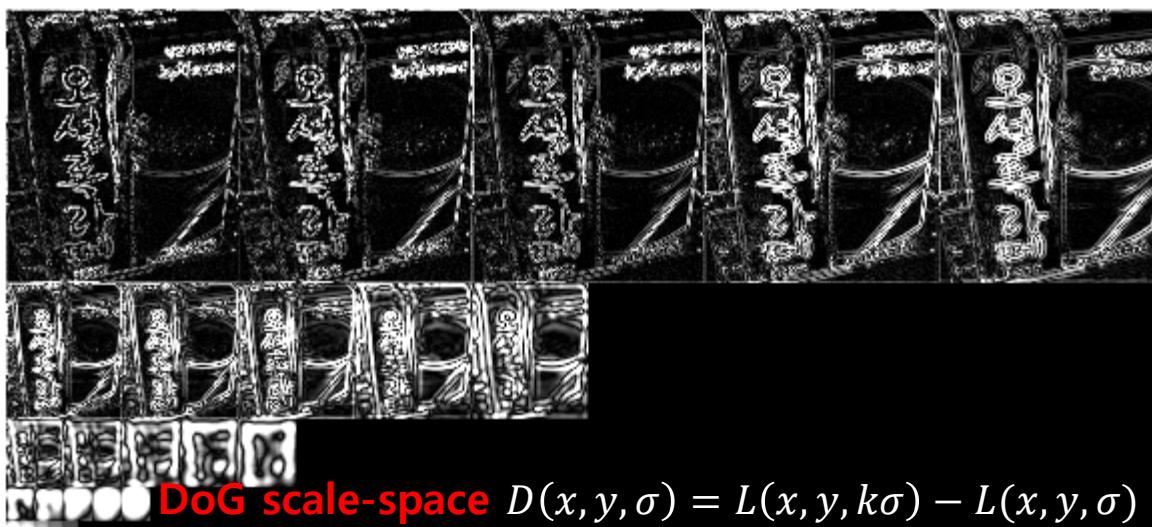


Feature Point) SIFT (Scale-Invariant Feature Transform; 1999)

- Key idea: **Scale-space** (~ image pyramid) and **DoG** (difference of Gaussian)



Gaussian scale-space $L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$

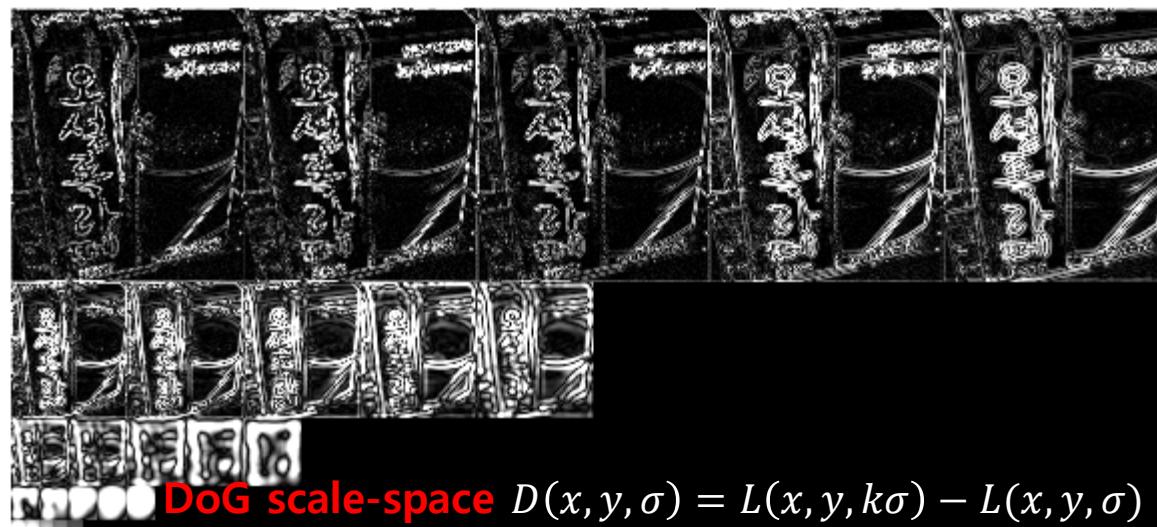
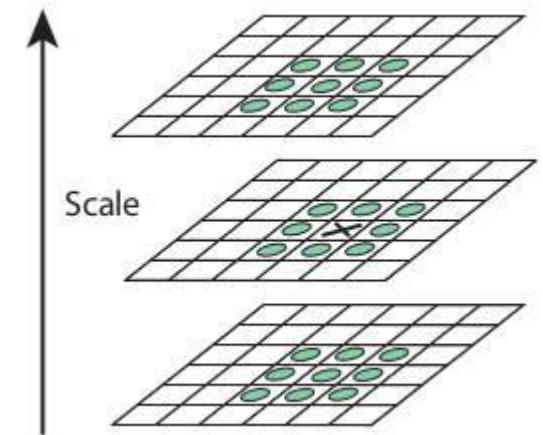


DoG scale-space $D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma)$

Feature Point) SIFT (Scale-Invariant Feature Transform; 1999)

- Key idea: **Scale-space** (~ [image pyramid](#)) and **DoG** ([difference of Gaussian](#))
- Part #1) **Feature point detection**
 1. Find **local extrema** (minima and maxima) in DoG scale-space
 2. Localize their position accurately (sub-pixel level) using 3D quadratic function
 3. Eliminate **low contrast candidates**, $|D(\mathbf{x})| < \tau$
 4. Eliminate **candidates on edges**,

$$\frac{\text{trace}(H)^2}{\det(H)} < \frac{(r+1)^2}{r} \quad \text{where} \quad H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$



$$\text{DoG scale-space } D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma)$$

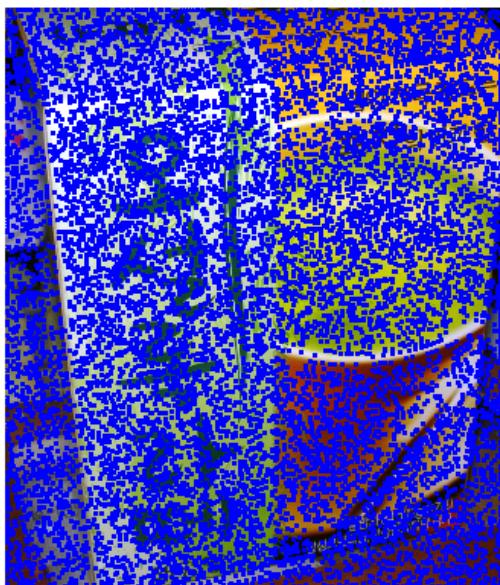
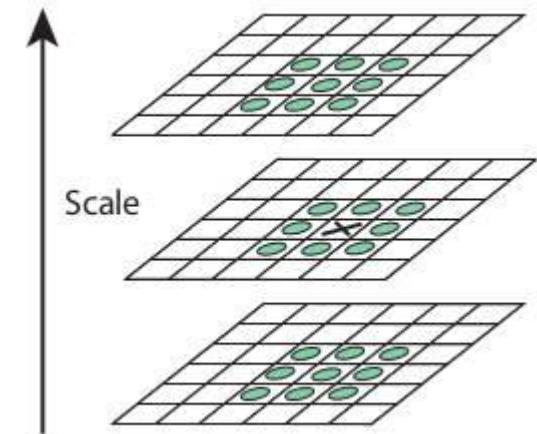
Feature Point) SIFT (Scale-Invariant Feature Transform; 1999)

- Key idea: **Scale-space** (~ [image pyramid](#)) and **DoG** ([difference of Gaussian](#))

- Part #1) **Feature point detection**

1. Find **local extrema** (minima and maxima) in DoG scale-space
2. Localize their position accurately (sub-pixel level) using 3D quadratic function
3. Eliminate **low contrast candidates**, $|D(\mathbf{x})| < \tau$
4. Eliminate **candidates on edges**,

$$\frac{\text{trace}(H)^2}{\det(H)} < \frac{(r+1)^2}{r} \quad \text{where} \quad H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$



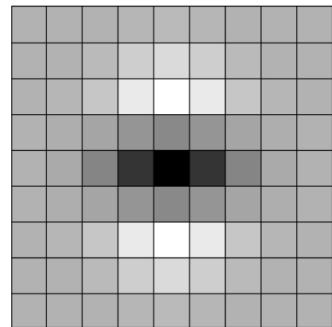
Local extrema (N: 11479)



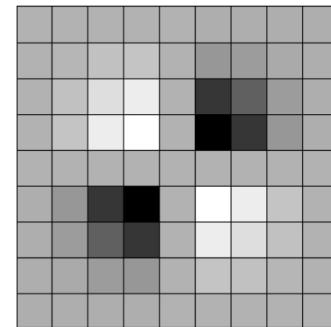
Feature points (N: 971)

Feature Point) SURF (Speeded Up Robust Features; 2006)

- Key idea: **Approximation of SIFT**
 - e.g. DoG approximation Haar-like features and **integral image**

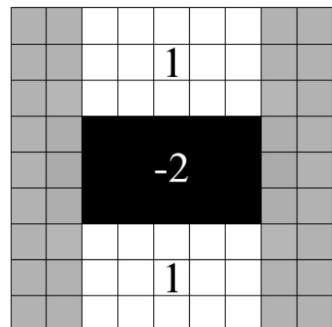


D_{yy}

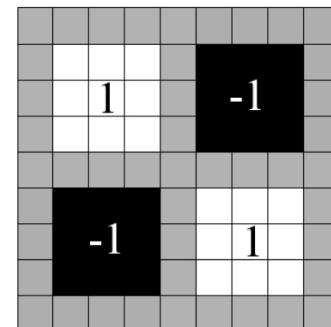


D_{xy}

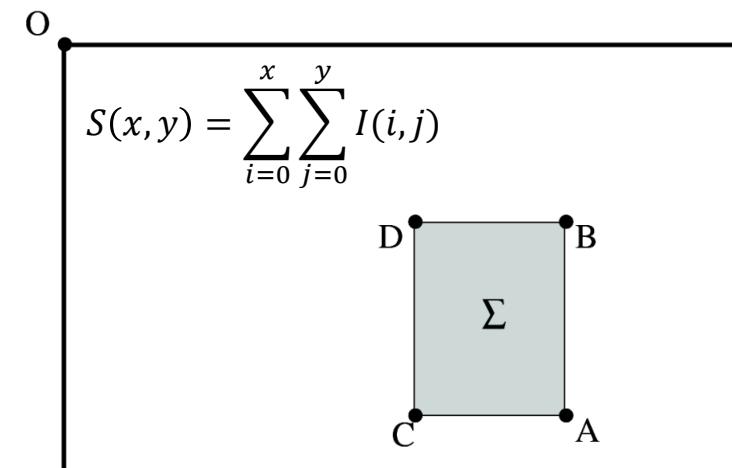
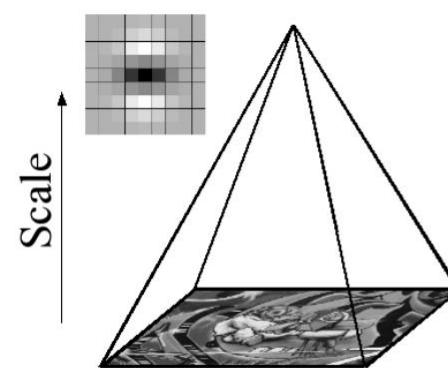
DoG approximation



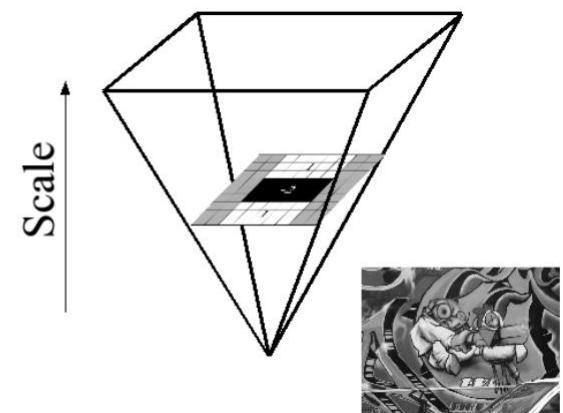
D'_{yy}



D'_{xy}



$$\Sigma = A - B - C + D$$



Feature Descriptor) SIFT (Scale-Invariant Feature Transform; 1999)

■ Part #2) Orientation assignment

- Derive magnitude and orientation of gradient of each patch

$$m(x, y) = \sqrt{(L(x + 1, y) - L(x - 1, y))^2 + (L(x, y + 1) - L(x, y - 1))^2}$$

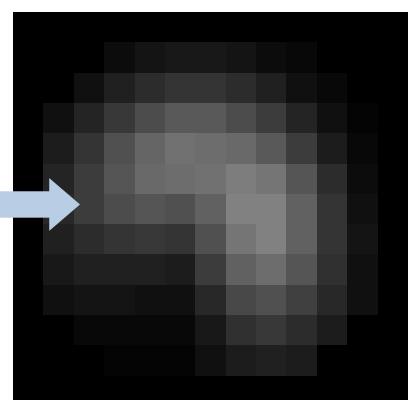
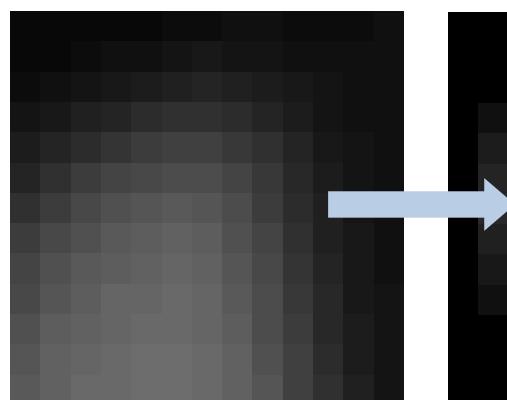
$$\theta(x, y) = \tan^{-1} \frac{L(x, y + 1) - L(x, y - 1)}{L(x + 1, y) - L(x - 1, y)}$$

- Find **the strongest orientation**

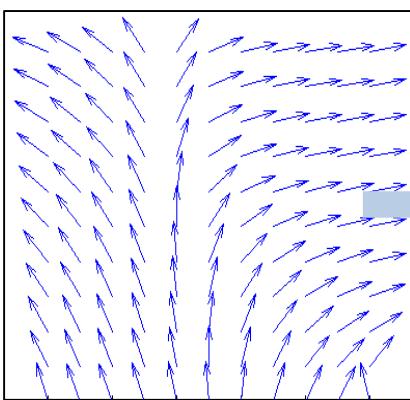
- Histogram voting (36 bins) with Gaussian-weighted magnitude



Feature scales and orientations

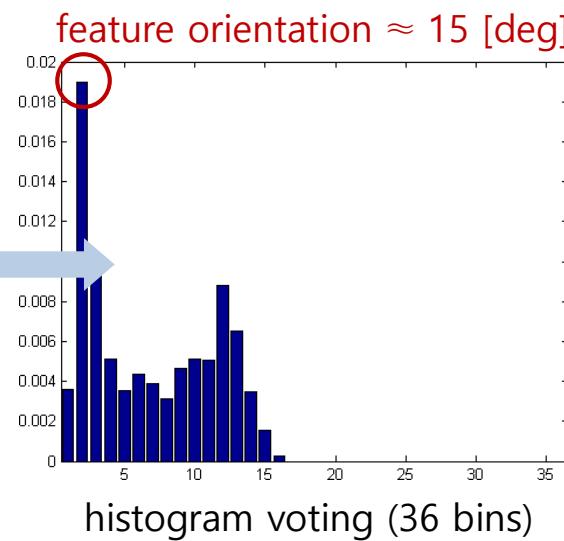


feature patch



gradient magnitude
 $m(x, y)$

gradient orientation
 $\theta(x, y)$



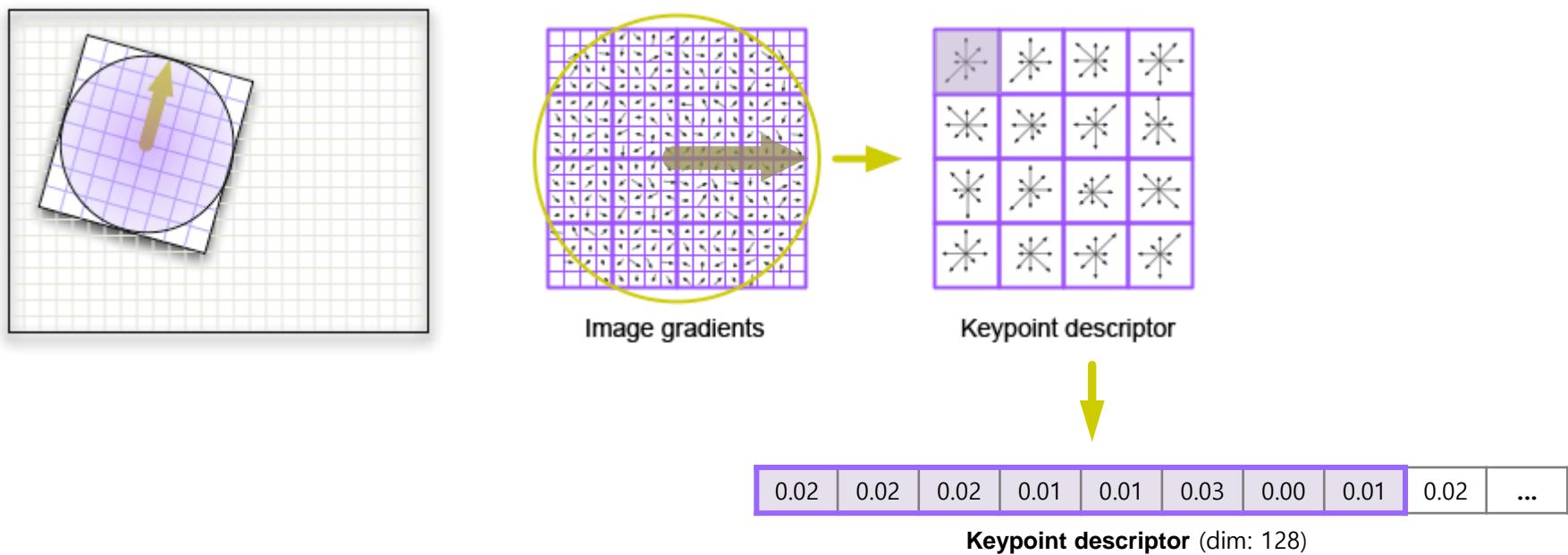
Feature Descriptor) SIFT (Scale-Invariant Feature Transform; 1999)

■ Part #3) Feature descriptor extraction

1. Build a 4×4 gradient histogram (8 bins) from each patch (16×16 pixels)
 - Use Gaussian-weighted magnitude again
 - Use relative angles w.r.t. the assigned feature orientation
2. Encode the histogram into a 128-dimensional vector
 - Apply normalization to be an unit vector

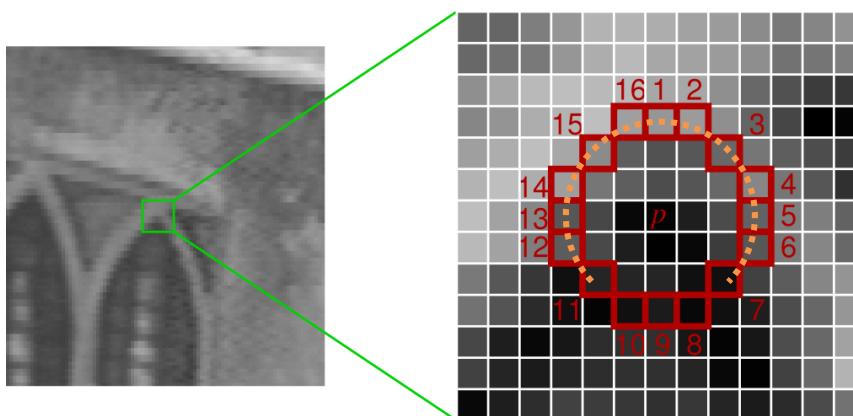


Feature scales and orientations

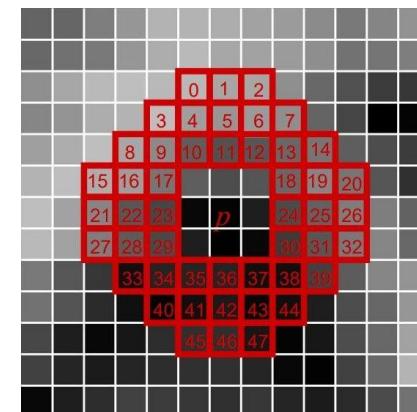


Feature Point) FAST (Features from Accelerated Segment Test; 2006)

- Key idea: **Intensity check of continuous arc of n pixels**
 - Is this point p a corner? (I_p : intensity at p , t : the intensity threshold)
 - Is a segment of n continuous pixels brighter than $I_p + t$? (OR) Is the segment darker than $I_p - t$?
 - Note) High-speed non-corner rejection: Checking intensity values at 1, 9, 5, and 13 (n : 12)



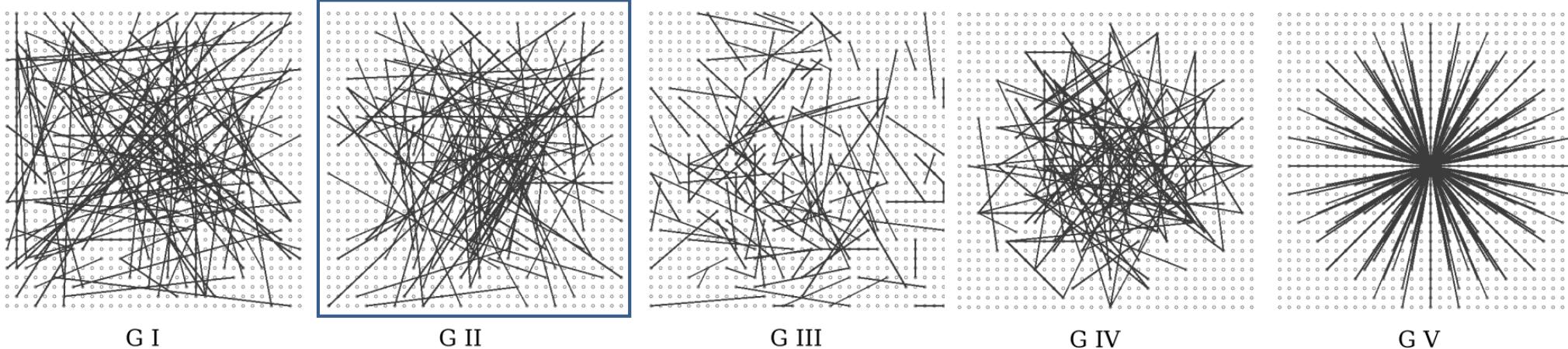
- Too many corners! → Non-maximum suppression
- Versions
 - FAST-9 (n : 9; `cv.FastFeatureDetector_TYPE_9_16`), FAST-12 (n : 12), ...
 - FAST-ER: Training a decision tree to enhance repeatability with more pixels



Feature Descriptor) BRIEF (Binary Robust Independent Elementary Features; 2010)

- Key idea: **Intensity comparison of a sequence of random pairs (binary test)**

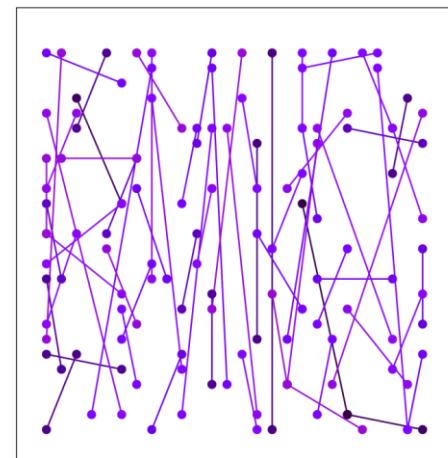
- Path size: 31 x 31 pixels (Note: Applying smoothing for stability and repeatability)



- Descriptor size: 128 tests (128 bits) → 16 bytes
 - Note) SIFT: 128-dimensional vector → 512 bytes
- Versions: The number of tests
 - BRIEF-32, BRIEF-64, BRIEF-128 (16 bytes), BRIEF-256 (32 bytes), BRIEF-512 (64 bytes), ...
- Combination examples
 - SIFT feature points + BRIEF descriptors
 - FAST feature points + BRIEF descriptors

Feature Point and Descriptor) ORB (Oriented FAST and rotated BRIEF, 2011)

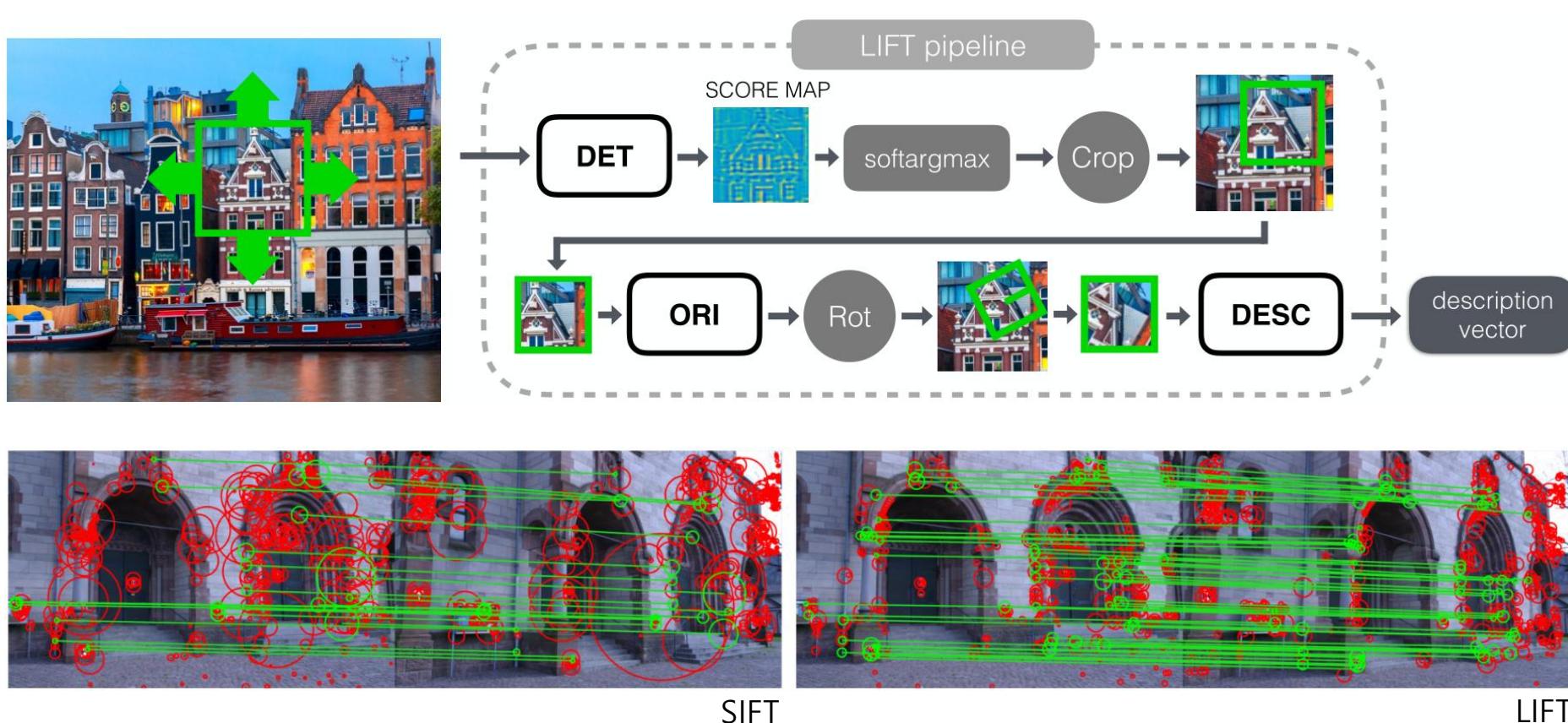
- Key idea: **Adding rotation invariance to BRIEF**
 - **Oriented FAST**
 - Generate scale pyramid for scale invariance
 - Detect *FAST-9* points (filtering with Harris corner response)
 - Calculate feature orientation by *intensity centroid* $C = (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}})$
$$\theta = \tan^{-1} \frac{m_{01}}{m_{10}} \quad \text{where} \quad m_{pq} = \sum_{x,y} x^p y^q I(x, y)$$
 - **Rotation-aware BRIEF**
 - Extract BRIEF descriptors w.r.t. the known orientation
 - Use better comparison pairs trained by greedy search
- Combination (default): **ORB**
 - FAST-9 detector (with orientation) + BRIEF-256 descriptor (with trained pairs)
- Computing time
 - ORB: **15.3 [msec]** / SURF: 217.3 [msec] / SIFT: 5228.7 [msec] @ 24 images (640x480) in Pascal dataset



Feature Point and Descriptor) LIFT (Learned Invariant Feature Transform; 2016)

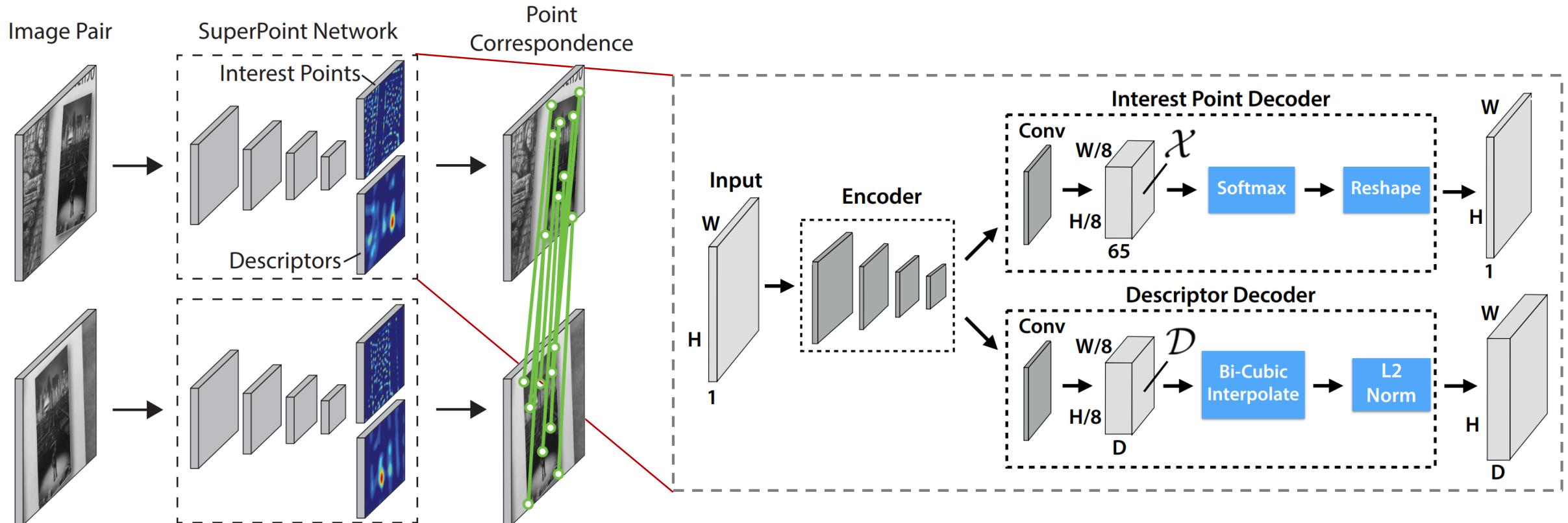
- Key idea: **Deep neural network**

- CNN network: **DET** (feature detector) + **ORI** (orientation estimator) + **DESC** (feature descriptor)
- Training data: Photo Tourism dataset with [VisualSfM](#) (SIFT)



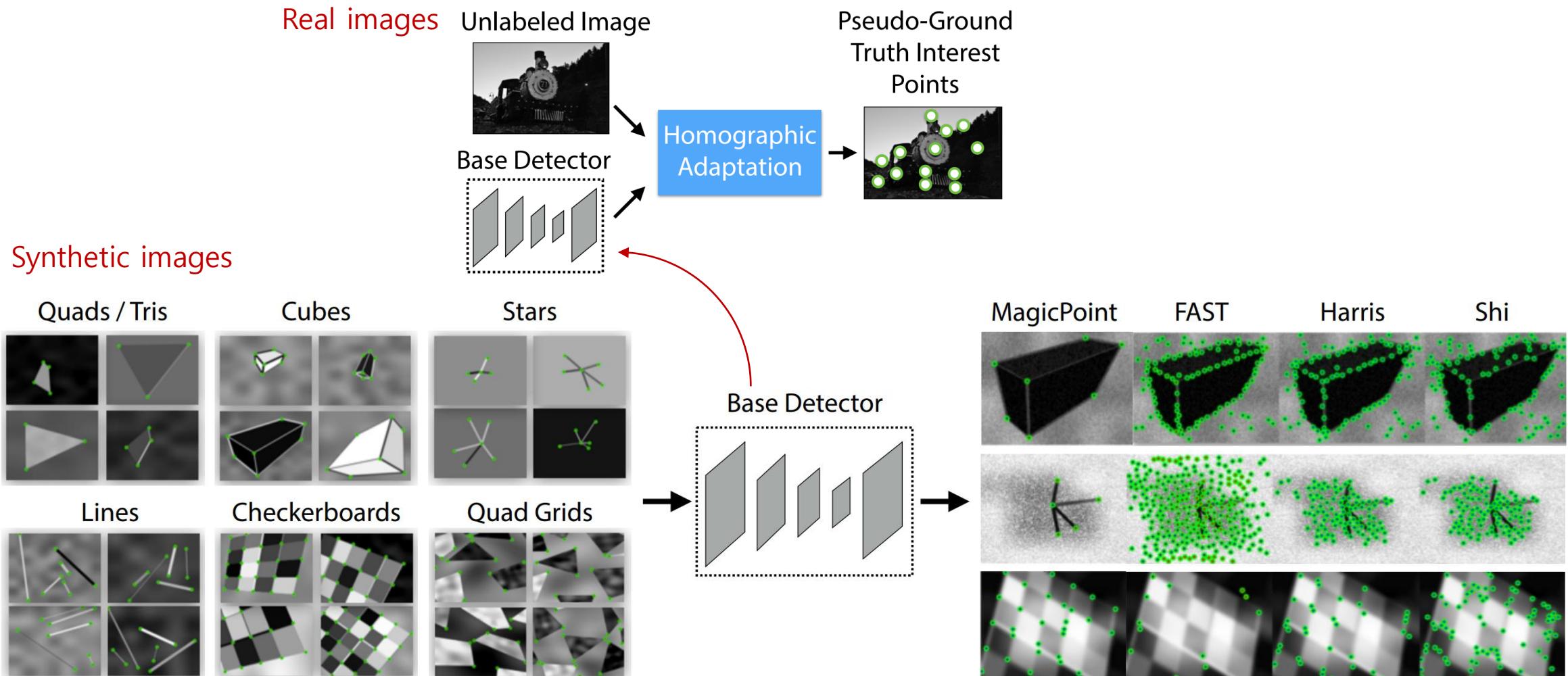
Feature Point and Descriptor) SuperPoint (2017)

- Key idea: **Self-supervised training with homography transformation**
 - CNN network: Encoder (~ VGG) + Decoders (for point and descriptor)



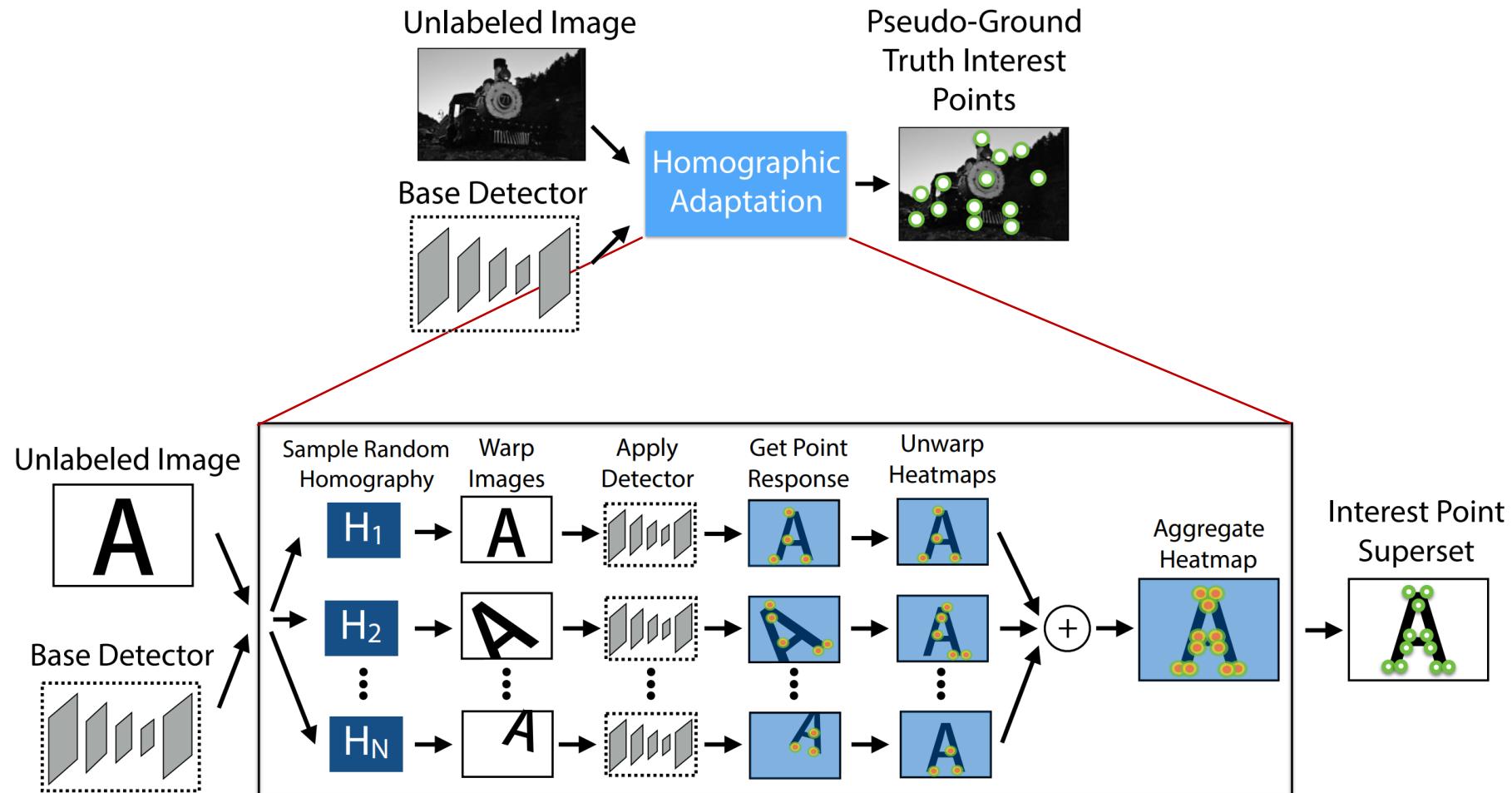
Feature Point and Descriptor) SuperPoint (2017)

- Key idea: **Self-supervised training with homography transformation**
 - Training data generation: The base detector, *MagicPoint* → Ground truth interest points



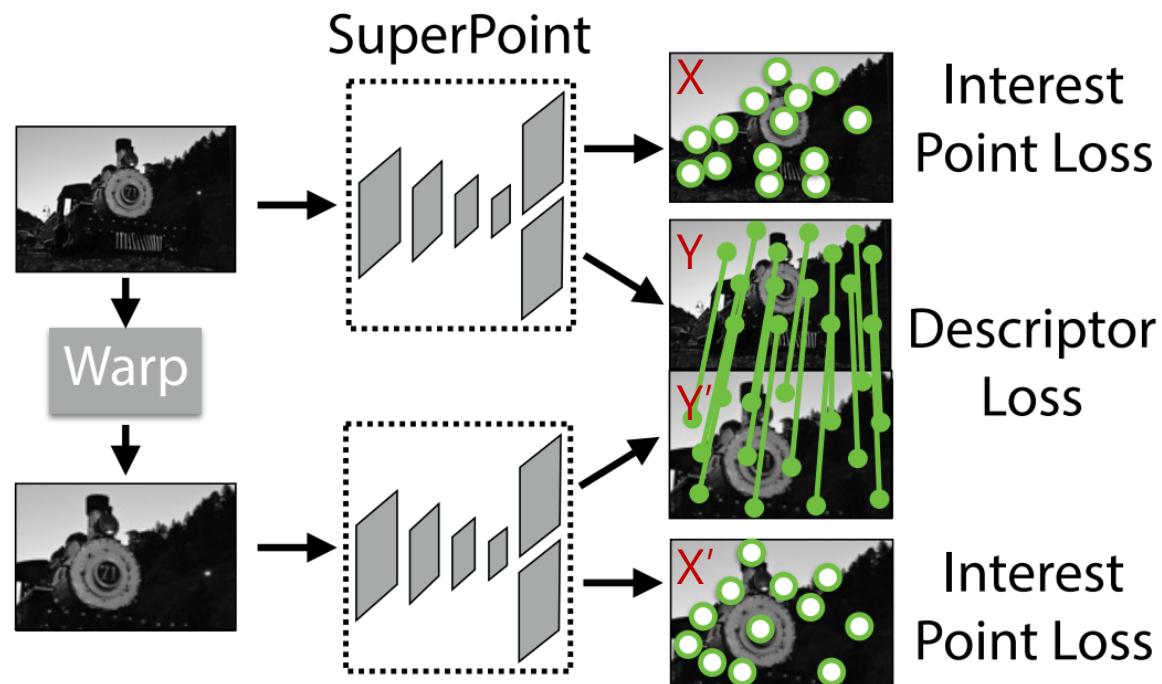
Feature Point and Descriptor) SuperPoint (2017)

- Key idea: **Self-supervised training with homography transformation**
 - Training data generation: The base detector, *MagicPoint* → Ground truth interest points



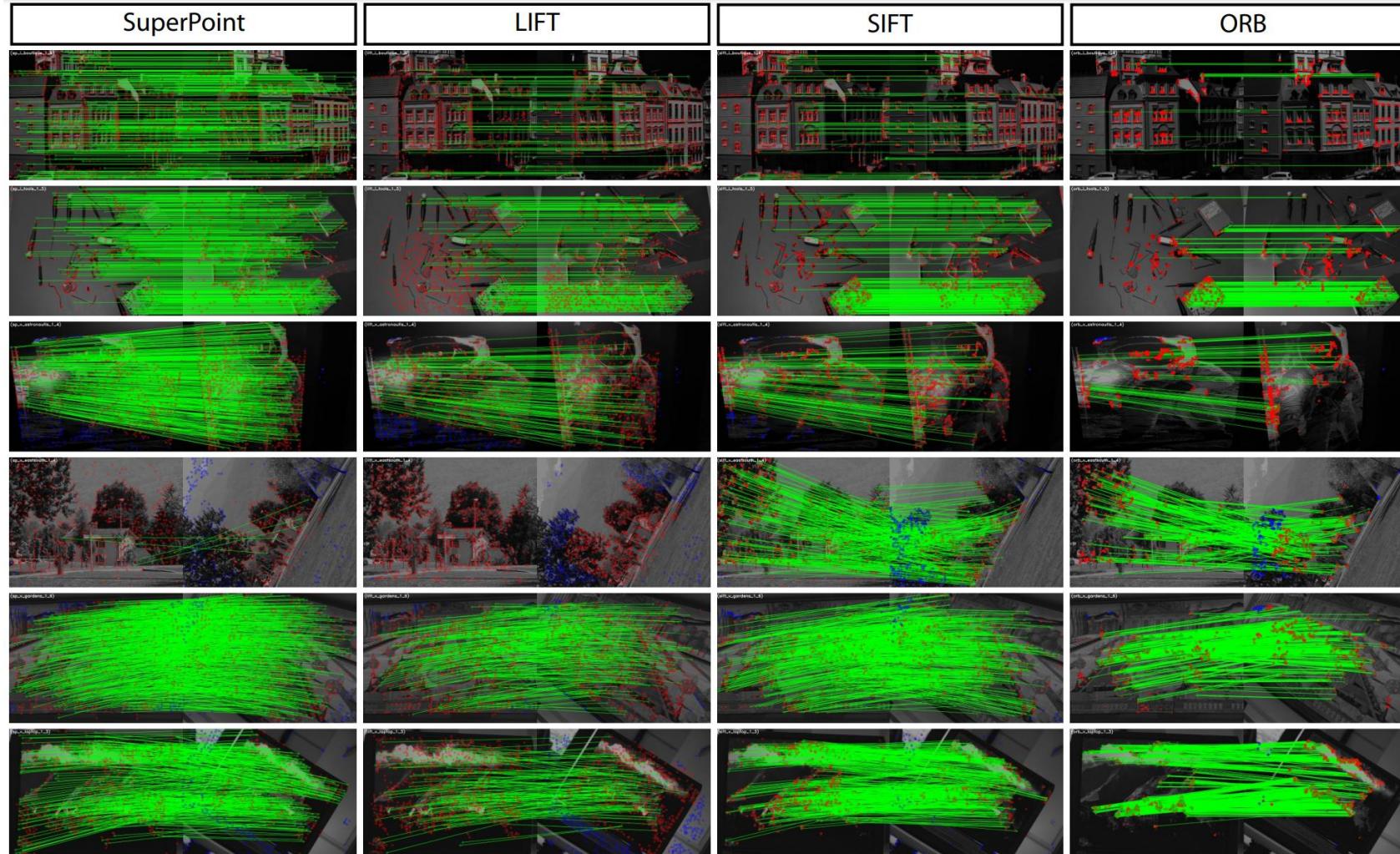
Feature Point and Descriptor) SuperPoint (2017)

- Key idea: **Self-supervised training with homography transformation**
 - Training data augmentation: Random homography transformation
 - Loss functions: Interest Point Loss (X, Y) + Interest Point Loss (X', Y') + Descriptor Loss (Y, Y')



Feature Point and Descriptor) SuperPoint (2017)

- Key idea: **Self-supervised training with homography transformation**
 - Real-time performance: **70 FPS** (13 msec) on 640x480 images with NVIDIA Titan X GPU



Freiburg RGBD:



Microsoft 7 Scenes:



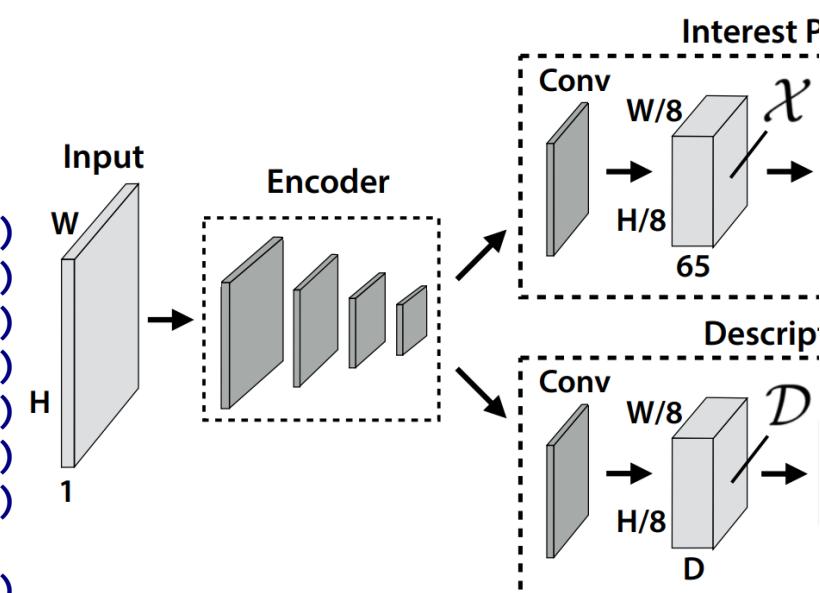
MonoVO:



Feature Point and Descriptor) SuperPoint (2017)

- Example) **SuperPoint implementation** [demo_superpoint.py] @ [Github \(authors\)](#)

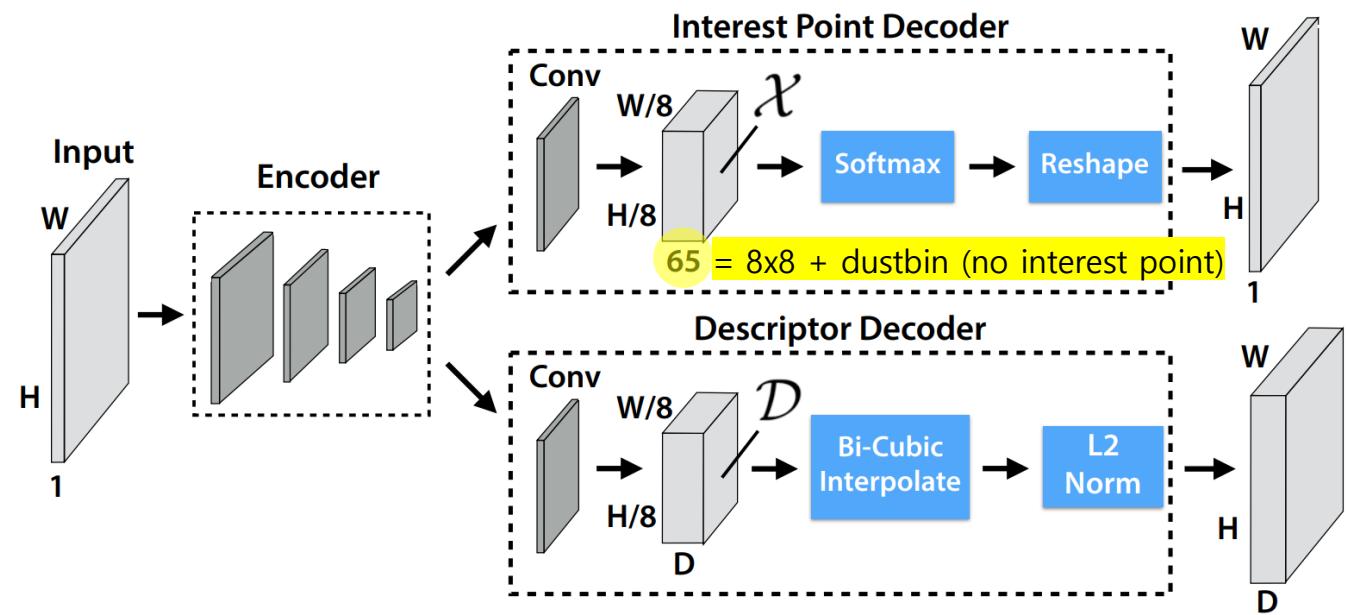
```
class SuperPointNet(torch.nn.Module):
    """ Pytorch definition of SuperPoint Network. """
    def __init__(self):
        super(SuperPointNet, self).__init__()
        self.relu = torch.nn.ReLU(inplace=True)
        self.pool = torch.nn.MaxPool2d(kernel_size=2, stride=2)
        c1, c2, c3, c4, c5, d1 = 64, 64, 128, 128, 256, 256
        # Shared Encoder.
        self.conv1a = torch.nn.Conv2d(1, c1, kernel_size=3, stride=1, padding=1)
        self.conv1b = torch.nn.Conv2d(c1, c1, kernel_size=3, stride=1, padding=1)
        self.conv2a = torch.nn.Conv2d(c1, c2, kernel_size=3, stride=1, padding=1)
        self.conv2b = torch.nn.Conv2d(c2, c2, kernel_size=3, stride=1, padding=1)
        self.conv3a = torch.nn.Conv2d(c2, c3, kernel_size=3, stride=1, padding=1)
        self.conv3b = torch.nn.Conv2d(c3, c3, kernel_size=3, stride=1, padding=1)
        self.conv4a = torch.nn.Conv2d(c3, c4, kernel_size=3, stride=1, padding=1)
        self.conv4b = torch.nn.Conv2d(c4, c4, kernel_size=3, stride=1, padding=1)
        # Detector Head.
        self.convPa = torch.nn.Conv2d(c4, c5, kernel_size=3, stride=1, padding=1)
        self.convPb = torch.nn.Conv2d(c5, 65, kernel_size=1, stride=1, padding=0)
        # Descriptor Head.
        self.convDa = torch.nn.Conv2d(c4, c5, kernel_size=3, stride=1, padding=1)
        self.convDb = torch.nn.Conv2d(c5, d1, kernel_size=1, stride=1, padding=0)
```



```

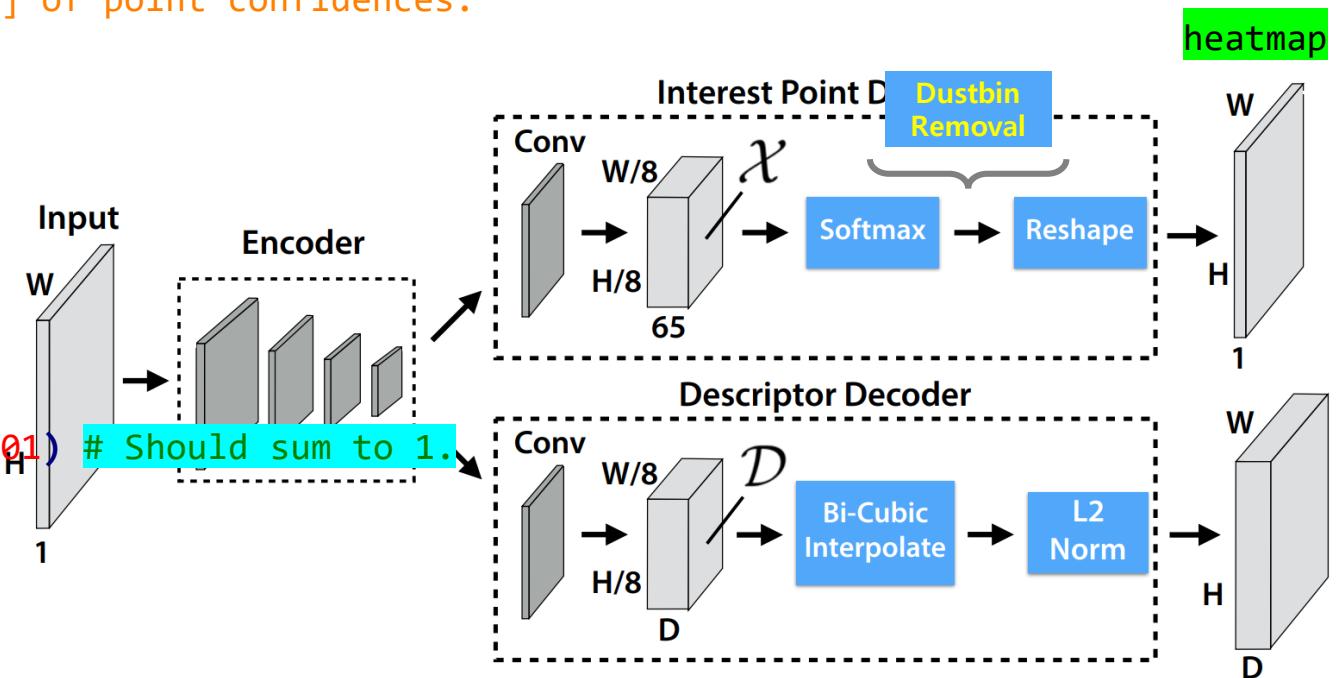
class SuperPointNet(torch.nn.Module):
    ...
    def forward(self, x):
        """ Forward pass that jointly computes unprocessed point and descriptor
        tensors.
        Input
            x: Image pytorch tensor shaped N x 1 x H x W.
        Output
            semi: Output point pytorch tensor shaped N x 65 x H/8 x W/8. (Note: 65 channels includes a dustbin.)
            desc: Output descriptor pytorch tensor shaped N x 256 x H/8 x W/8.
        """
        # Shared Encoder.
        x = self.relu(self.conv1a(x))
        x = self.relu(self.conv1b(x))
        x = self.pool(x)
        x = self.relu(self.conv2a(x))
        x = self.relu(self.conv2b(x))
        x = self.pool(x)
        x = self.relu(self.conv3a(x))
        x = self.relu(self.conv3b(x))
        x = self.pool(x)
        x = self.relu(self.conv4a(x))
        x = self.relu(self.conv4b(x))
        # Detector Head.
        cPa = self.relu(self.convPa(x))
        semi = self.convPb(cPa)
        # Descriptor Head.
        cDa = self.relu(self.convDa(x))
        desc = self.convDb(cDa)
        dn = torch.norm(desc, p=2, dim=1) # Compute the norm.
        desc = desc.div(torch.unsqueeze(dn, 1)) # Divide by norm to normalize.
        return semi, desc

```



```

class SuperPointNet(torch.nn.Module):
    ...
    def run(self, img):
        """ Process a numpy image to extract points and descriptors.
        Input
            img - HxW numpy float32 input image in range [0,1].
        Output
            corners - 3xN numpy array with corners [x_i, y_i, confidence_i]^T.
            desc - 256xN numpy array of corresponding unit normalized descriptors.
            heatmap - HxW numpy heatmap in range [0,1] of point confidences.
        """
        ...
        ...
        # Forward pass of network.
        outs = self.net.forward(inp)
        semi, coarse_desc = outs[0], outs[1]
        # Convert pytorch -> numpy.
        semi = semi.data.cpu().numpy().squeeze()
        # --- Process points.
        dense = np.exp(semi) # Softmax.
        dense = dense / (np.sum(dense, axis=0)+.00001) # Should sum to 1.
        # Remove dustbin.
        nodust = dense[:-1, :, :]
        # Reshape to get full resolution heatmap.
        Hc = int(H / self.cell)
        Wc = int(W / self.cell)
        nodust = nodust.transpose(1, 2, 0)
        heatmap = np.reshape(nodust, [Hc, Wc, self.cell, self.cell])
        heatmap = np.transpose(heatmap, [0, 2, 1, 3])
        heatmap = np.reshape(heatmap, [Hc*self.cell, Wc*self.cell])
        ...
    
```



```

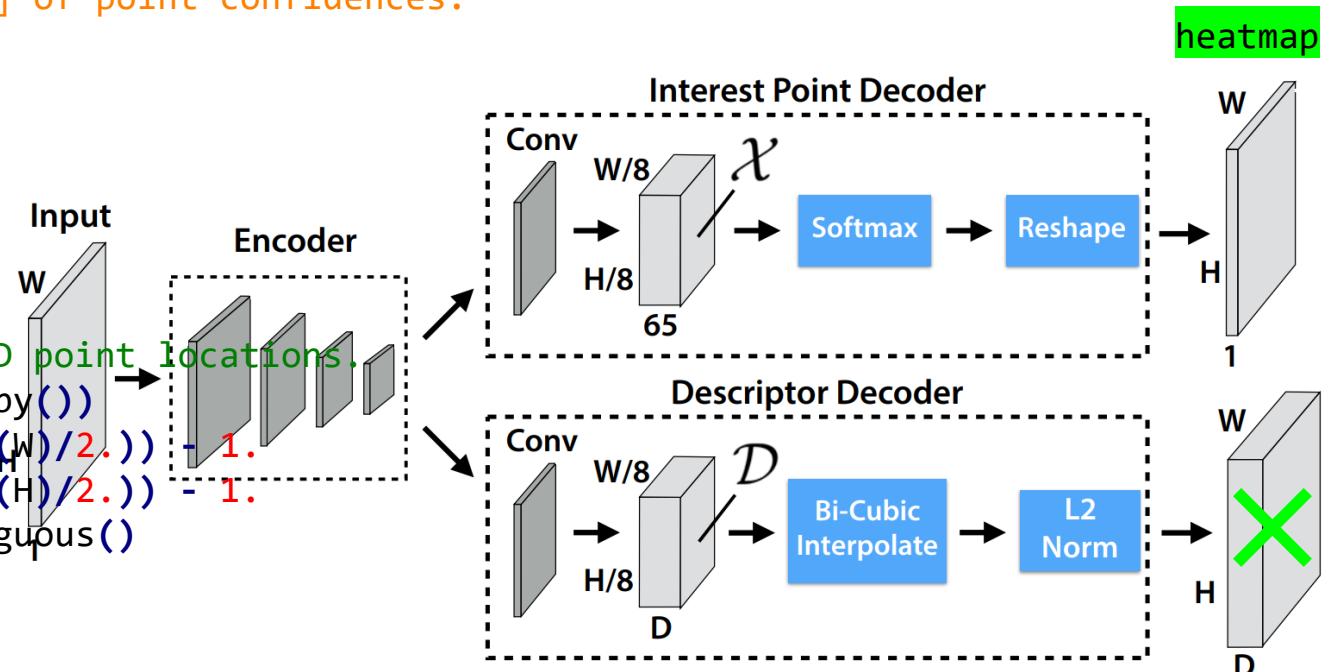
class SuperPointNet(torch.nn.Module):

...
def run(self, img):
    """ Process a numpy image to extract points and descriptors.

    Input
        img - HxW numpy float32 input image in range [0,1].
    Output
        corners - 3xN numpy array with corners [x_i, y_i, confidence_i]^T.
        desc - 256xN numpy array of corresponding unit normalized descriptors.
        heatmap - HxW numpy heatmap in range [0,1] of point confidences.
    """
    ...

    ...
    # --- Process descriptor.
    D = coarse_desc.shape[1]
    if pts.shape[1] == 0:
        desc = np.zeros((D, 0))
    else:
        # Interpolate into descriptor map using 2D point locations.
        samp_pts = torch.from_numpy(pts[:2, :].copy())
        samp_pts[0, :] = (samp_pts[0, :] / (float(W)/2.)) - 1.
        samp_pts[1, :] = (samp_pts[1, :] / (float(H)/2.)) - 1.
        samp_pts = samp_pts.transpose(0, 1).contiguous()
        samp_pts = samp_pts.view(1, 1, -1, 2)
        samp_pts = samp_pts.float()
        if self.cuda:
            samp_pts = samp_pts.cuda()
        desc = torch.nn.functional.grid\_sample(coarse_desc, samp_pts)
        desc = desc.data.cpu().numpy().reshape(D, -1)
        desc /= np.linalg.norm(desc, axis=0)[np.newaxis, :]
    return pts, desc, heatmap

```



Summary) Feature Points and Descriptors

Feature Points	Gradient-based <ul style="list-style-type: none">▪ Harris▪ GFTT (a.k.a. Shi-Tomasi)▪ SIFT▪ SURF	Intensity-based <ul style="list-style-type: none">▪ FAST	DL-based <ul style="list-style-type: none">▪ LIFT▪ SuperPoint
Feature Descriptor	Real-valued <ul style="list-style-type: none">▪ SIFT▪ SURF	Binary-valued <ul style="list-style-type: none">▪ BRIEF▪ ORB (FAST+BRIEF)	Real-valued (DL-based) <ul style="list-style-type: none">▪ LIFT▪ SuperPoint
Advantages Disadvantages	(+) Accurate (-) Slow	(+) Fast (-) Inaccurate (+) Less storage	(+) Accurate (+) Fast (-) GPU requirement

Table of Contents

▪ Feature Points

- Gradient-based: Harris corner, GFTT corner, SIFT, SURF
- Intensity-based: FAST
- DL-based: LIFT, SuperPoint

▪ Feature Descriptors

- Real-valued: SIFT, SURF (DL-based: LIFT, SuperPoint)
- Binary-valued: BRIEF, ORB

▪ Feature Matching

- Q) How can we associate the points across different images?

▪ Feature Tracking

- Q) How can we associate the points across their next image?

▪ Outlier Rejection

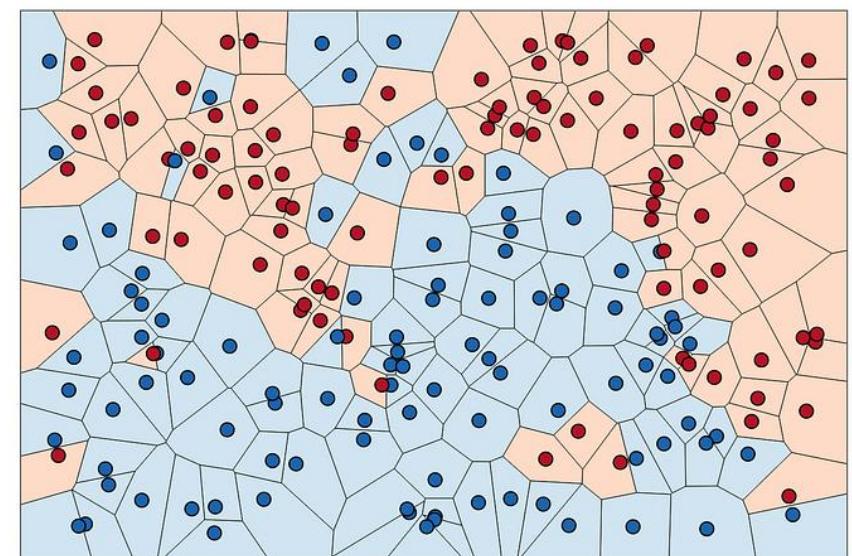
- Q) How can we select correctly matched points?



Feature Matching) For Real-valued Descriptors

▪ Real-valued descriptors

- Distance measures
 - Euclidean distance: $l_2(\mathbf{d}, \mathbf{d}') = \|\mathbf{d} - \mathbf{d}'\|_2$ (\downarrow : similar)
 - Cosine similarity: $s_c(\mathbf{d}, \mathbf{d}') = \frac{\mathbf{d} \cdot \mathbf{d}'}{\|\mathbf{d}\| \|\mathbf{d}'\|}$ (\uparrow : similar)
 - Note) Matching measures can be combined or advanced.
 - e.g. The ratio of the best and second best similarity $>$ threshold
 - It may select more distinguishable feature matching.
- Matching algorithms
 - Brute-force search
 - Time complexity: $O(N)$ for N descriptors
 - Approximated nearest neighborhood search (ANN search)
 - Time complexity: $O(\log N)$ or less for N descriptors
 - Note) Big-ANN Competition (recent: NeurIPS 2023)



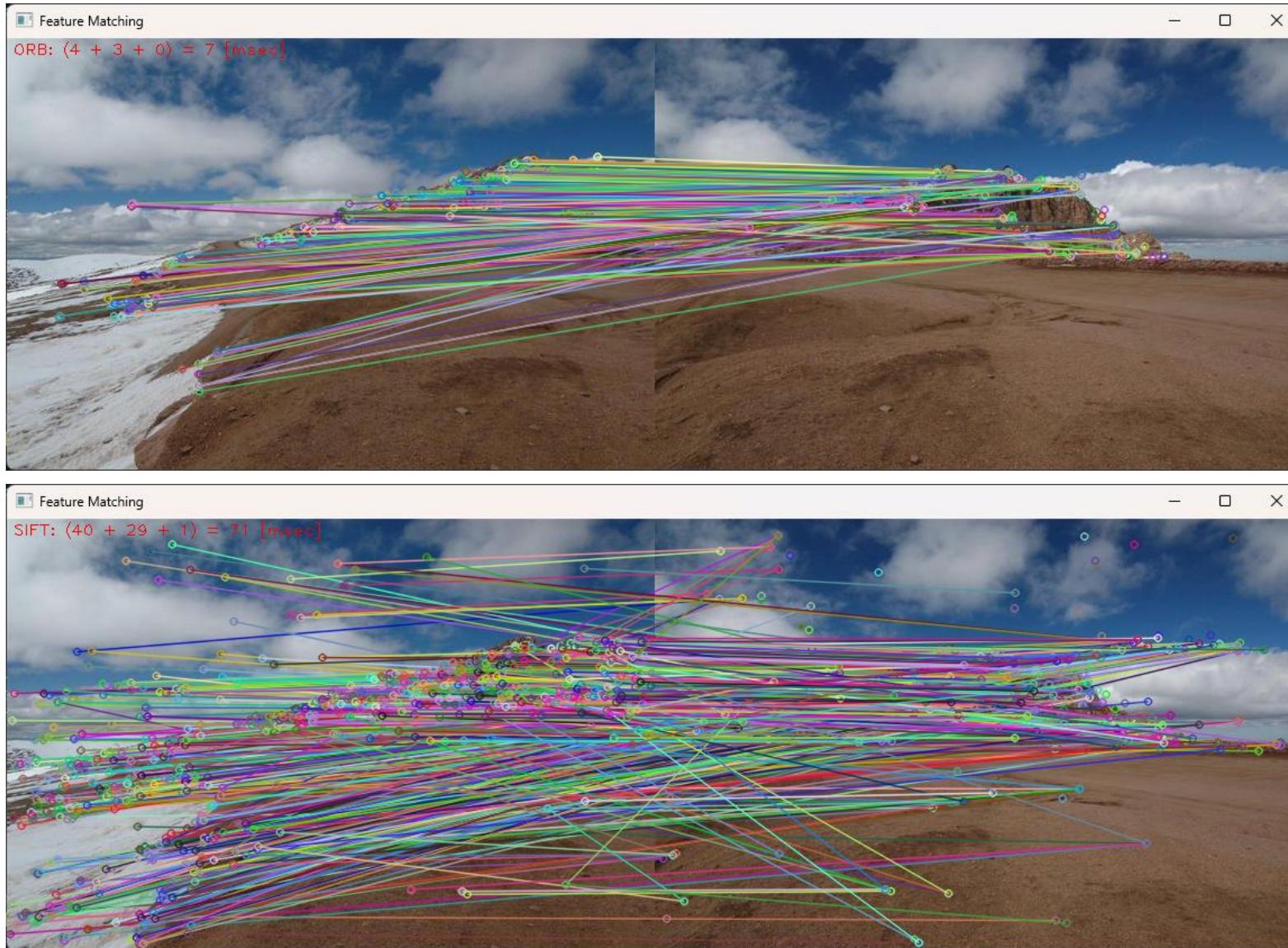
Feature Matching) For Binary-valued Descriptors

- **Binary-valued descriptors**

- Distance measures
 - Hamming distance: $l_h(\mathbf{d}, \mathbf{d}') = \sum_i (d_i \neq d'_i)$
 - e.g. 4-bit descriptors
 - $l_h(0110, \textcolor{red}{1}110) = 1$ vs. (6, 14)
 - $l_h(0110, 011\textcolor{red}{1}) = 1$ vs. (6, 7)
 - $l_h(0110, 01\textcolor{red}{0}1) = 2$ vs. (6, 5)
 - Note) Hamming distance is the L_1 -norm with binary-valued descriptors.
 - Matching algorithms
 - Brute-force search
 - Time complexity: $O(N)$ for N descriptors

Feature Matching

- Example) Feature matching comparison [feature_matching.py]



Feature Matching

- Example) **Feature matching comparison** [feature_matching.py]

```
# Load two images
img1 = cv.imread('../data/hill01.jpg')
img2 = cv.imread('../data/hill02.jpg')

# Instantiate feature detectors and matchers
# Note) You can specify options for each detector in its creation.
features = [
    ...
    {'name': 'FAST',      'detector': cv.FastFeatureDetector_create(),
     'matcher' : None}, # No descriptor
    ...
    {'name': 'ORB',       'detector': cv.ORB_create(),
     'matcher' : cv.DescriptorMatcher_create('BruteForce-Hamming')},
    {'name': 'SIFT',       'detector': cv.SIFT_create(),
     'matcher' : cv.DescriptorMatcher_create('BruteForce')},
]

```

Feature Matching

- Example) Feature matching comparison [feature_matching.py]

```
# Detect feature points
keypoints1 = features[f_select]['detector'].detect(img1)
keypoints2 = features[f_select]['detector'].detect(img2)

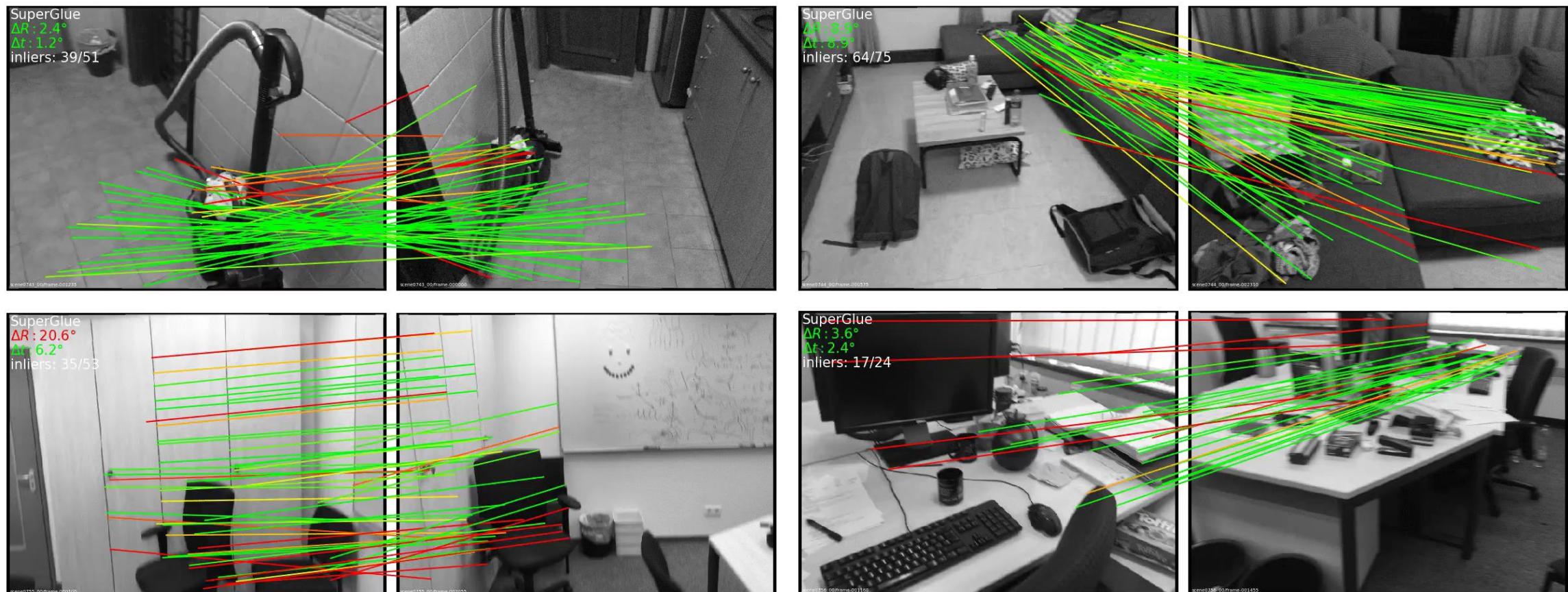
if features[f_select]['matcher'] is not None:
    # Extract feature descriptors
    keypoints1, descriptors1 = features[f_select]['detector'].compute(img1, keypoints1)
    keypoints2, descriptors2 = features[f_select]['detector'].compute(img2, keypoints2)

    # Match the feature descriptors
    match = features[f_select]['matcher'].match(descriptors1, descriptors2)

# Show the matched image
if features[f_select]['matcher'] is not None:
    img_merged = cv.drawMatches(img1, keypoints1, img2, keypoints2, match, None)
else:
    img1_keypts = cv.drawKeypoints(img1, keypoints1, None)
    img2_keypts = cv.drawKeypoints(img2, keypoints2, None)
    img_merged = np.hstack((img1_keypts, img2_keypts))
info = features[f_select]['name'] + ...
cv.putText(img_merged, info, (5, 15), cv.FONT_HERSHEY_PLAIN, 1, (0, 0, 255))
cv.imshow('Feature Matching', img_merged)
```

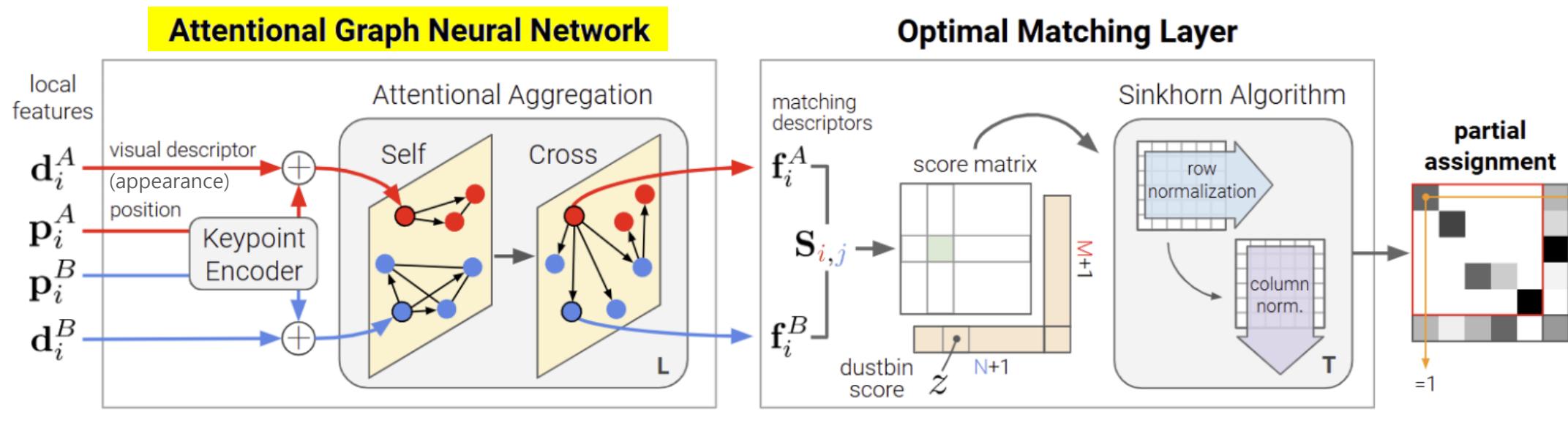
Feature Matching) SuperGlue (2020) → LightGlue (2023)

- Key idea: **Encoding feature points and descriptors using a graph neural network with attention**
 - For extreme wide-baseline image pairs in real-time on GPU
 - e.g. Relative pose estimation on the [ScanNet dataset](#) (more **correct matches** and less **mismatches**)

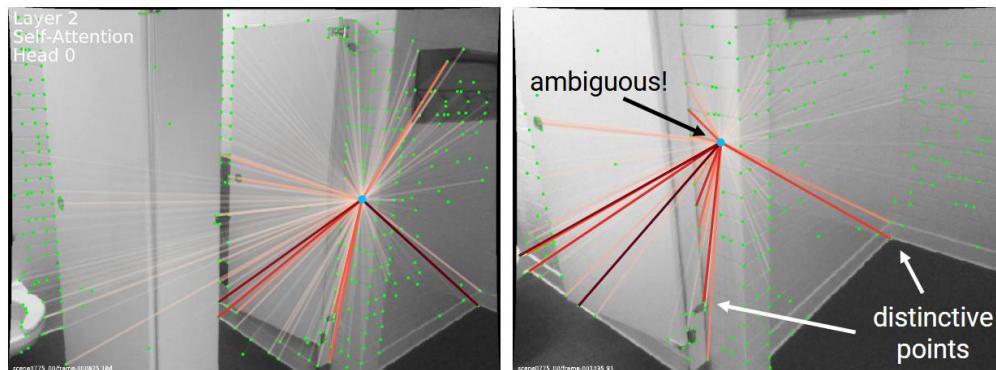


Feature Matching) SuperGlue (2020)

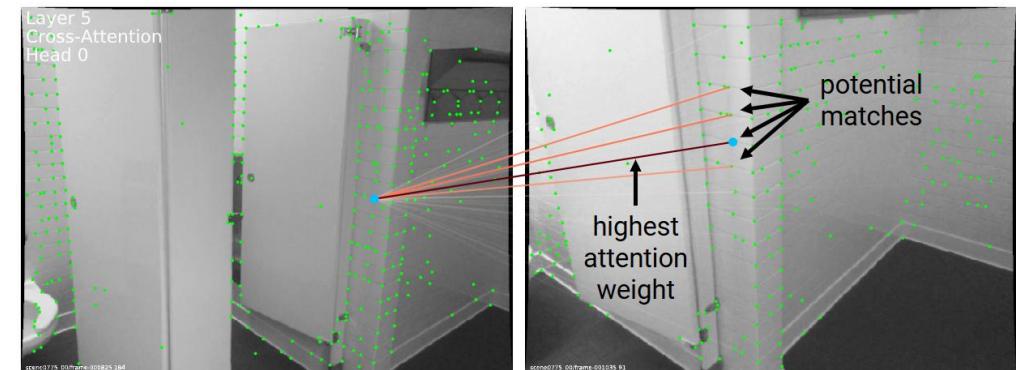
- Key idea: **Encoding feature points and descriptors using a graph neural network with attention**



Self-attention: Intra-image information flow

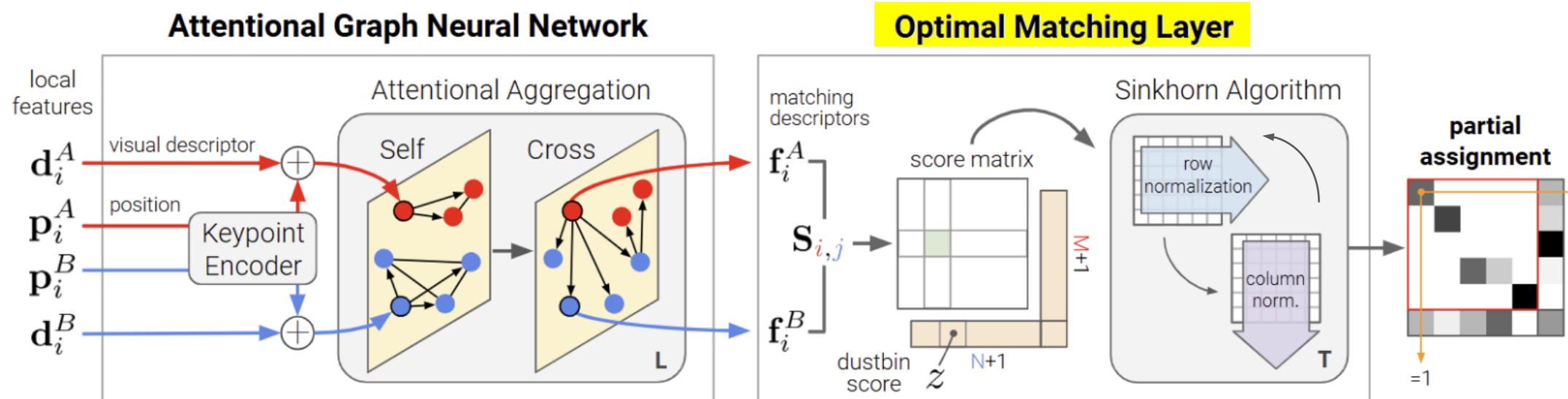


Cross-attention: Inter-image communication



Feature Matching) SuperGlue (2020)

- Key idea: **Encoding feature points and descriptors using a graph neural network with attention**



- Matching score $S_{i,j}$: Inner product of \mathbf{f}_i^A and \mathbf{f}_j^B
- Sinkhorn algorithm: Differentiable & soft [Hungarian algorithm](#)

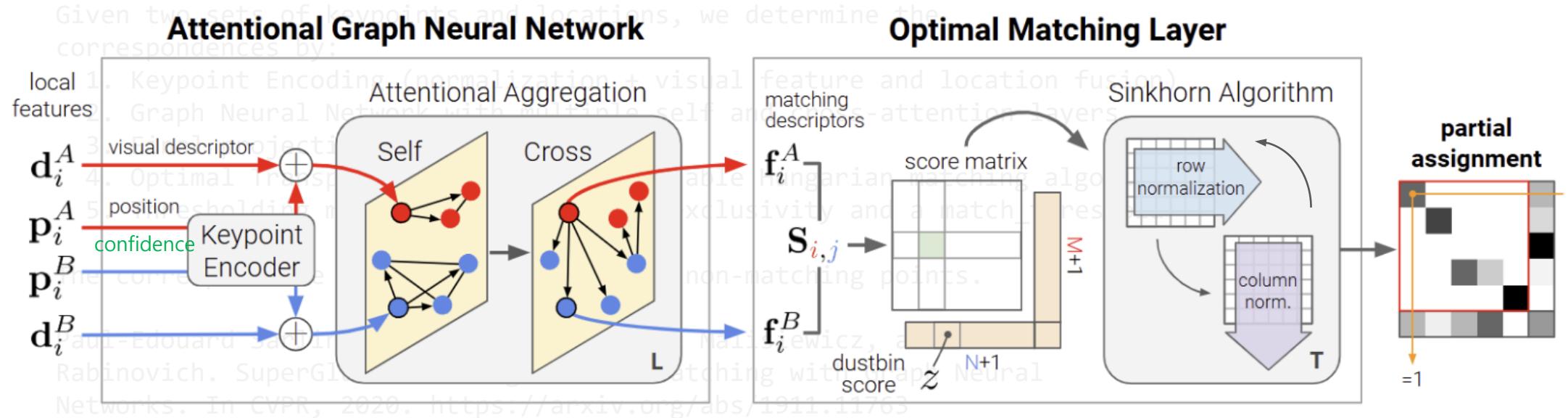
e.g. Optimal assignment problem

Task Worker \	Task 1	Task 2	Task 3
Worker 1	\$8	\$4	\$7
Worker 2	\$5	\$2	\$3
Worker 3	\$9	\$4	\$8

Feature Matching) SuperGlue (2020)

- Example) **SuperGlue implementation** [models/superglue.py] @ [Github \(authors\)](#)

```
class SuperGlue(nn.Module):  
    """SuperGlue feature matching middle-end
```



```
....  
default_config = {  
    'descriptor_dim': 256,  
    'weights': 'indoor',  
    'keypoint_encoder': [32, 64, 128, 256], # 4-layer CNN: [3, 32, 128, 256, 256] channels (kernel size = 1)  
    'GNN_layers': ['self', 'cross'] * 9, # 18-layer attention network + 1-layer CNN (final projection)  
    'sinkhorn_iterations': 100, # T = 100  
    'match_threshold': 0.2,  
}
```

Feature Extraction and Matching

- Example) Feature matching comparison: [Image Matching WebUI](#)

Spaces | Realcat /image-matching-webui | like 86 | Running

App Files Community 2

Image Matching Structure from Motion(under-dev)

Image Matching WebUI

This Space demonstrates [Image Matching WebUI](#) by vincent qin. Feel free to play with it, or duplicate to run image matching without a queue!

For more details about supported local features and matchers, please refer to <https://github.com/Vincentqyw/image-matching-webui>

All algorithms run on CPU for inference, causing slow speeds and high latency. For faster inference, please download the [source code](#) for local deployment.

Your feedback is valuable to me. Please do not hesitate to report any bugs [here](#).

Matching Model: superpoint+superGlue

Image Source: upload, webcam, clipboard

Image 0: **Image 1:**

Open for More: Keypoints: **Open for More: Raw Matches (Green for good matches, Red for bad):** **Open for More: Ransac Matches (Green for good matches, Red for bad):** **Open for More: Matches Statistics:** #Matches: 155

Reset **Run Match**

Advanced Setting:

Open for More: Examples: Examples (click one of the images below to Run Match). Thx: WxBs

Image 0	Image 1	Match threshold	Max features	Keypoint threshold	Matching Mod
		0.1	2000	0.01	disk

Open for More: Warped Image: **Image 1 - warped**

Feature Template Matching) For Image Patches

- Raw image patches (or histogram) can be used as descriptors.

- Distance measures

- SAD (sum of absolute difference): $l_1(\mathbf{d}, \mathbf{d}') = \|\mathbf{d} - \mathbf{d}'\|_1 = \sum_i |d_i - d'_i|$

- SSD (sum of squared difference): $l_2^2(\mathbf{d}, \mathbf{d}') = \|\mathbf{d} - \mathbf{d}'\|_2^2 = \sum_i (d_i - d'_i)^2$

- Cross-correlation: $s_{cc}(\mathbf{d}, \mathbf{d}') = \mathbf{d} \cdot \mathbf{d}' = \sum_i d_i d'_i$

- NCC (normalized cross-correlation): $s_{NCC}(\mathbf{d}, \mathbf{d}') = \frac{\mathbf{d} \cdot \mathbf{d}'}{\|\mathbf{d}\| \|\mathbf{d}'\|}$ (~ cosine similarity)

- ZNCC (zero-mean normalized cross-correlation): $s_{ZNCC}(\mathbf{d}, \mathbf{d}') = \frac{\bar{\mathbf{d}} \cdot \bar{\mathbf{d}}'}{\|\bar{\mathbf{d}}\| \|\bar{\mathbf{d}}'\|}$ where $\bar{\mathbf{d}} = \mathbf{d} - E(\mathbf{d})$ and $\bar{\mathbf{d}}' = \mathbf{d}' - E(\mathbf{d}')$

- Matching algorithms

- Sliding window (\mathbf{d} : )

Matching Result



SSD

Detected Point



SSD

Matching Result



Cross-correlation

Detected Point



NCC

Matching Result



Detected Point



Feature Tracking) Lukas-Kanade Optical Flow (1981)

- Key idea: **Finding movement of a patch whose pixel values are same**

- Brightness constancy constraint: $I(x, y, t) = I(x + \Delta_x, y + \Delta_y, t + \Delta_t)$

$$I_x \frac{\Delta_x}{\Delta_t} + I_y \frac{\Delta_y}{\Delta_t} + I_t = 0 \quad \leftarrow \quad I(x + \Delta_x, y + \Delta_y, t + \Delta_t) \approx I(x, y, t) + I_x \Delta_x + I_y \Delta_y + I_t \Delta_t$$

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad \text{where} \quad \mathbf{A} = [I_x \quad I_y], \quad \mathbf{x} = [\Delta_x, \Delta_y]^\top, \quad \text{and} \quad \mathbf{b} = [-I_t] \quad (\Delta_t = 1)$$

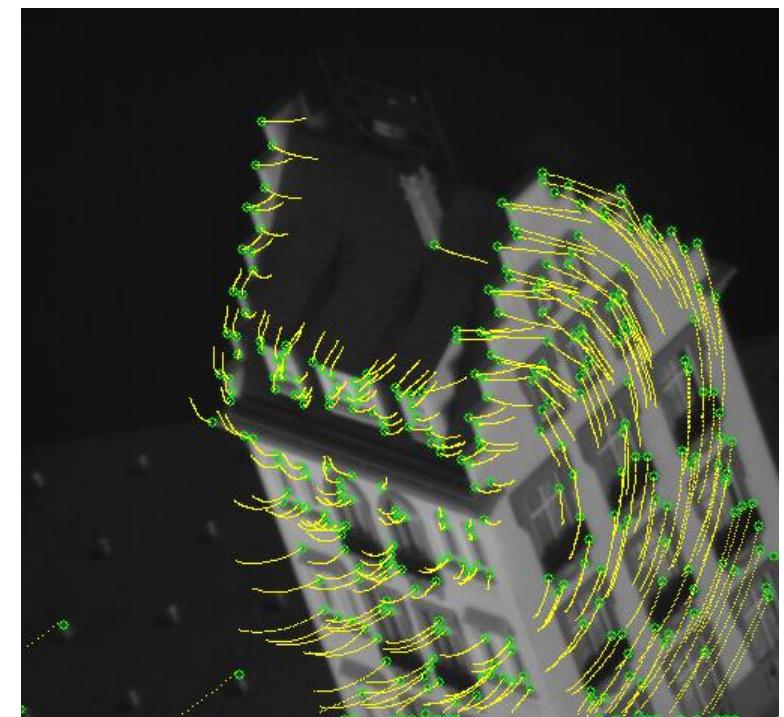
$$\therefore \mathbf{x} = \mathbf{A}^\dagger \mathbf{b}$$

- Combination: **KLT tracker**

- GFTT (a.k.a. Shi-Tomasi) detector + Lukas-Kanade optical flow

- Advantages and disadvantages (feature tracking vs. matching)

- (+) No descriptor required (\rightarrow fast and compact)
 - (-) Continuous feature tracking causes drift errors.
 - (-) Not working in wide-baseline cases
 - (+) Able to control matching range



Feature Tracking) Lukas-Kanade Optical Flow (1981)

- Example) **Feature matching comparison** [feature_tracking_klt.py]



Feature Tracking) Lukas-Kanade Optical Flow (1981)

- Example) **Feature matching comparison** [feature_tracking_klt.py]

```
# Open a video and get an initial image
video = cv.VideoCapture(video_file)
assert video.isOpened()

_, gray_prev = video.read()
if gray_prev.ndim >= 3 and gray_prev.shape[2] > 1:
    gray_prev = cv.cvtColor(gray_prev, cv.COLOR_BGR2GRAY)

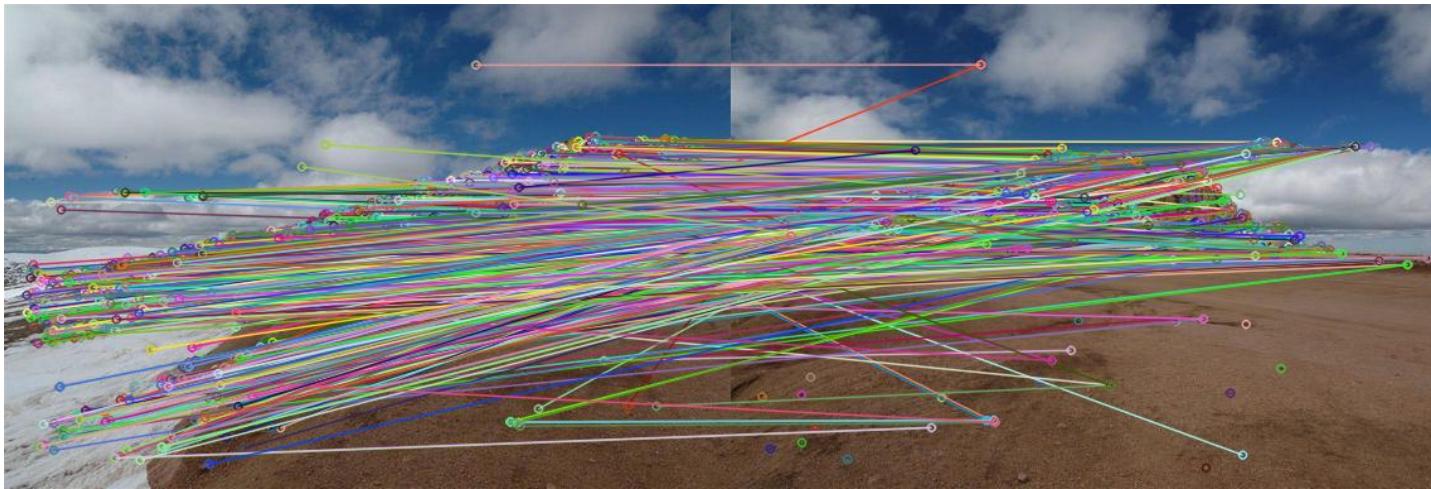
# Run the KLT feature tracker
while True:
    # Grab an image from the video
    valid, img = video.read()
    if not valid:
        break
    if img.ndim >= 3 and img.shape[2] > 1:
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    else:
        gray = img.copy()

    # Extract optical flow
    pts_prev = cv.goodFeaturesToTrack(gray_prev, 2000, 0.01, 10)
    pts, status, error = cv.calcOpticalFlowPyrLK(gray_prev, gray, pts_prev, None)
    gray_prev = gray

    # Show the optical flow on the image
    if img.ndim < 3 or img.shape[2] < 3:
        img = cv.cvtColor(img, cv.COLOR_GRAY2BGR)
```

Why Outliers?

Putative feature matches (inliers + outliers)



After applying RANSAC (inliers)

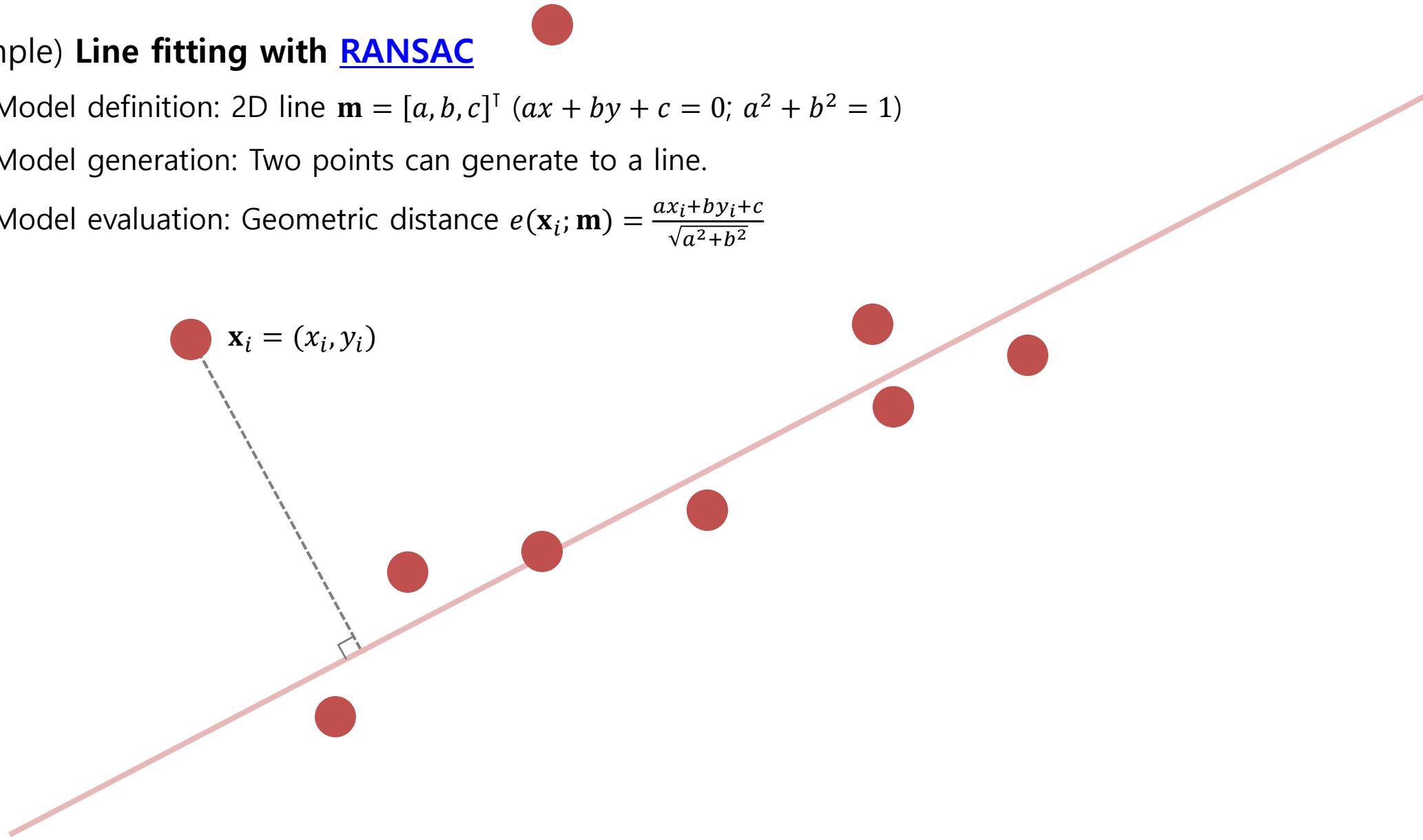


Model: Homography H (Parameters: 9 elements of H)

Outlier Rejection) RANSAC (Random Sample Consensus; 1981)

- Example) **Line fitting with RANSAC**

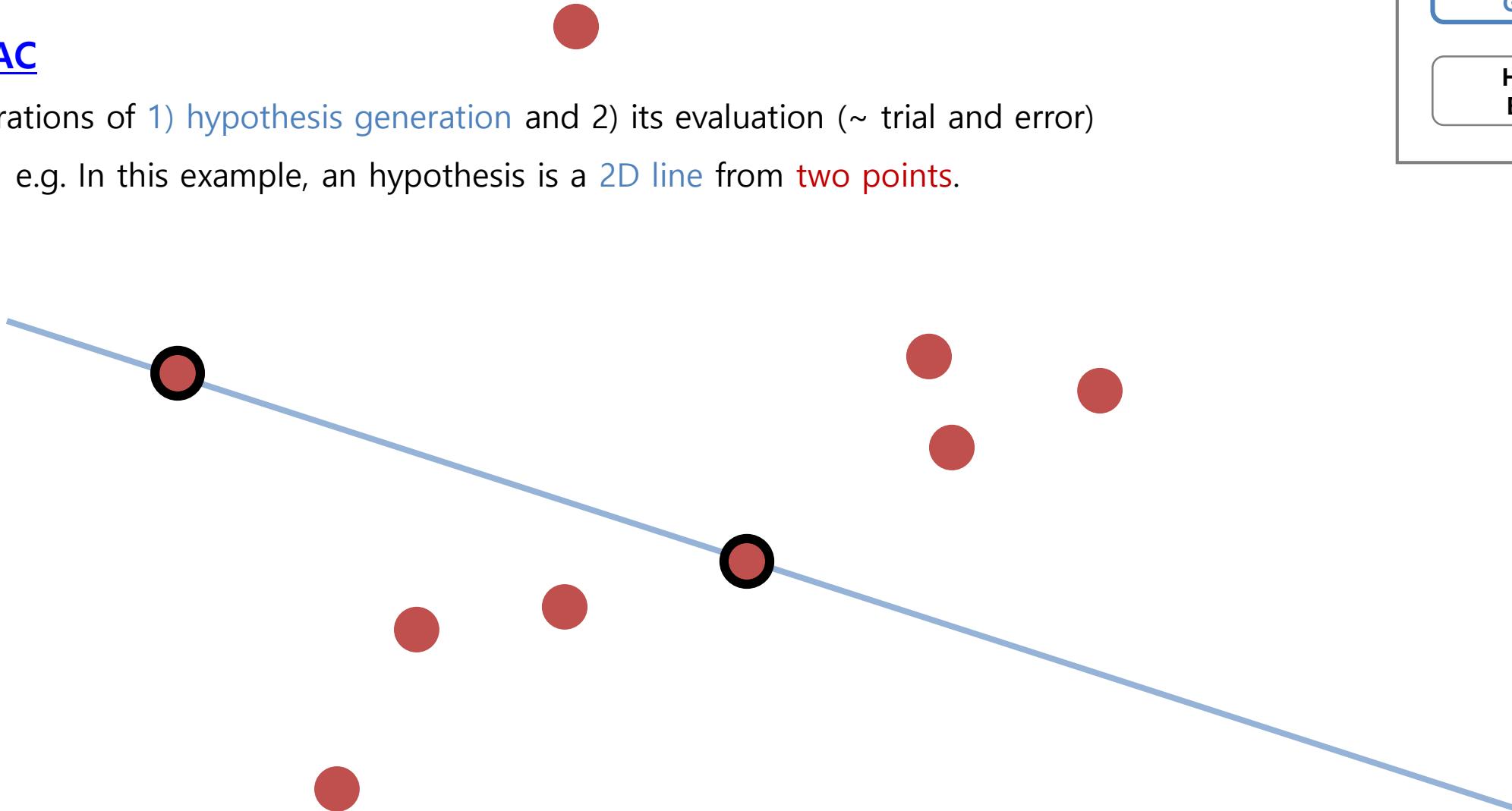
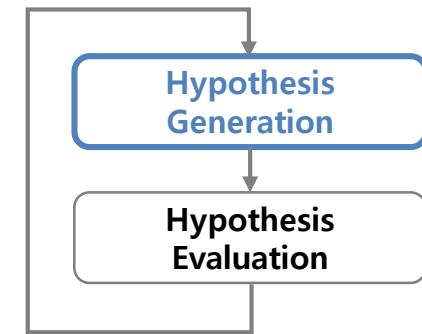
- Model definition: 2D line $\mathbf{m} = [a, b, c]^\top$ ($ax + by + c = 0$; $a^2 + b^2 = 1$)
- Model generation: Two points can generate to a line.
- Model evaluation: Geometric distance $e(\mathbf{x}_i; \mathbf{m}) = \frac{ax_i + by_i + c}{\sqrt{a^2 + b^2}}$



Outlier Rejection) RANSAC (Random Sample Consensus; 1981)

- **RANSAC**

- Iterations of 1) hypothesis generation and 2) its evaluation (~ trial and error)
 - e.g. In this example, an hypothesis is a 2D line from two points.

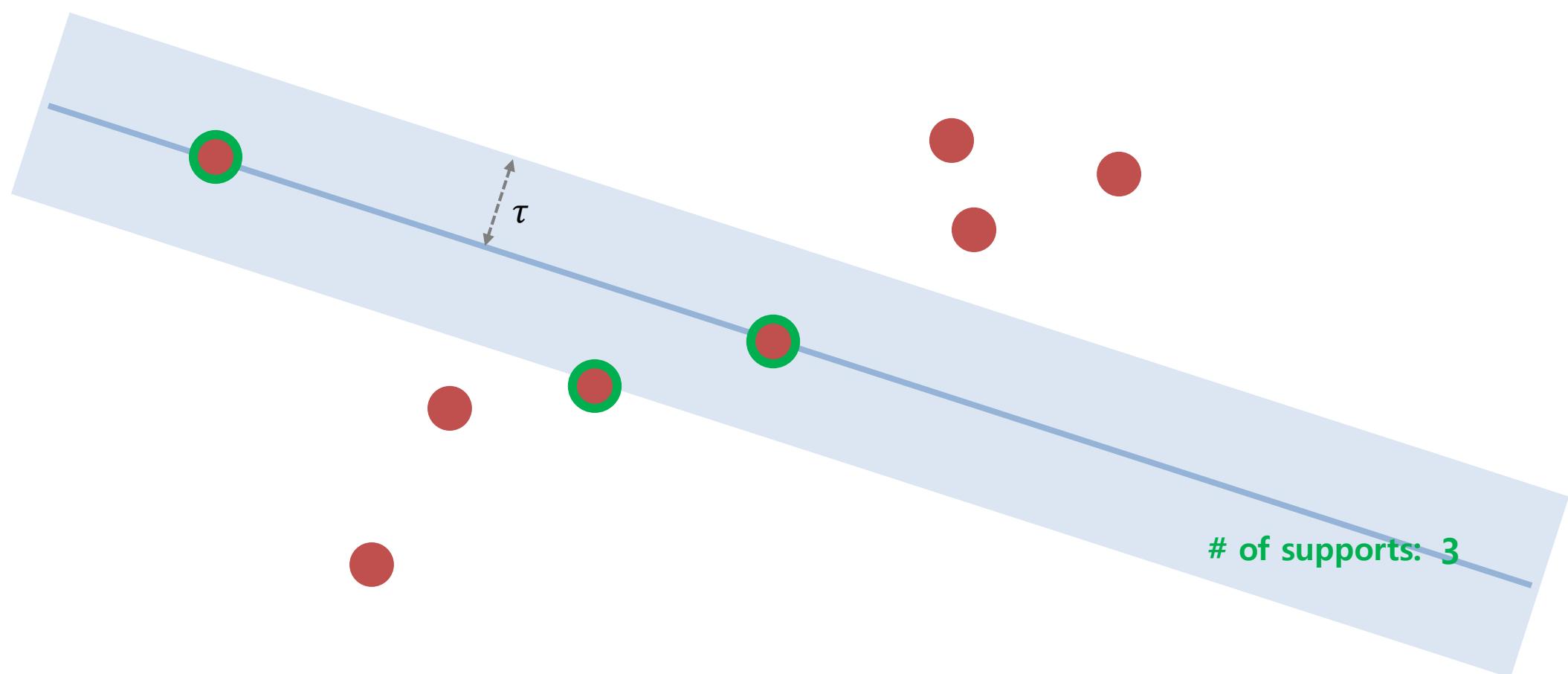
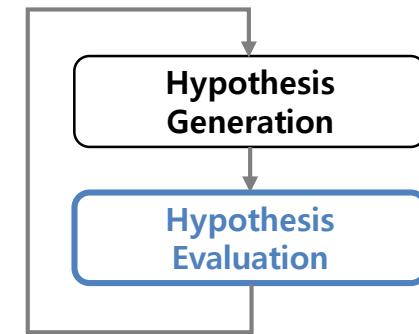


Outlier Rejection) RANSAC (Random Sample Consensus; 1981)

- **RANSAC**

- Iterations of 1) hypothesis generation and 2) its evaluation (~ trial and error)

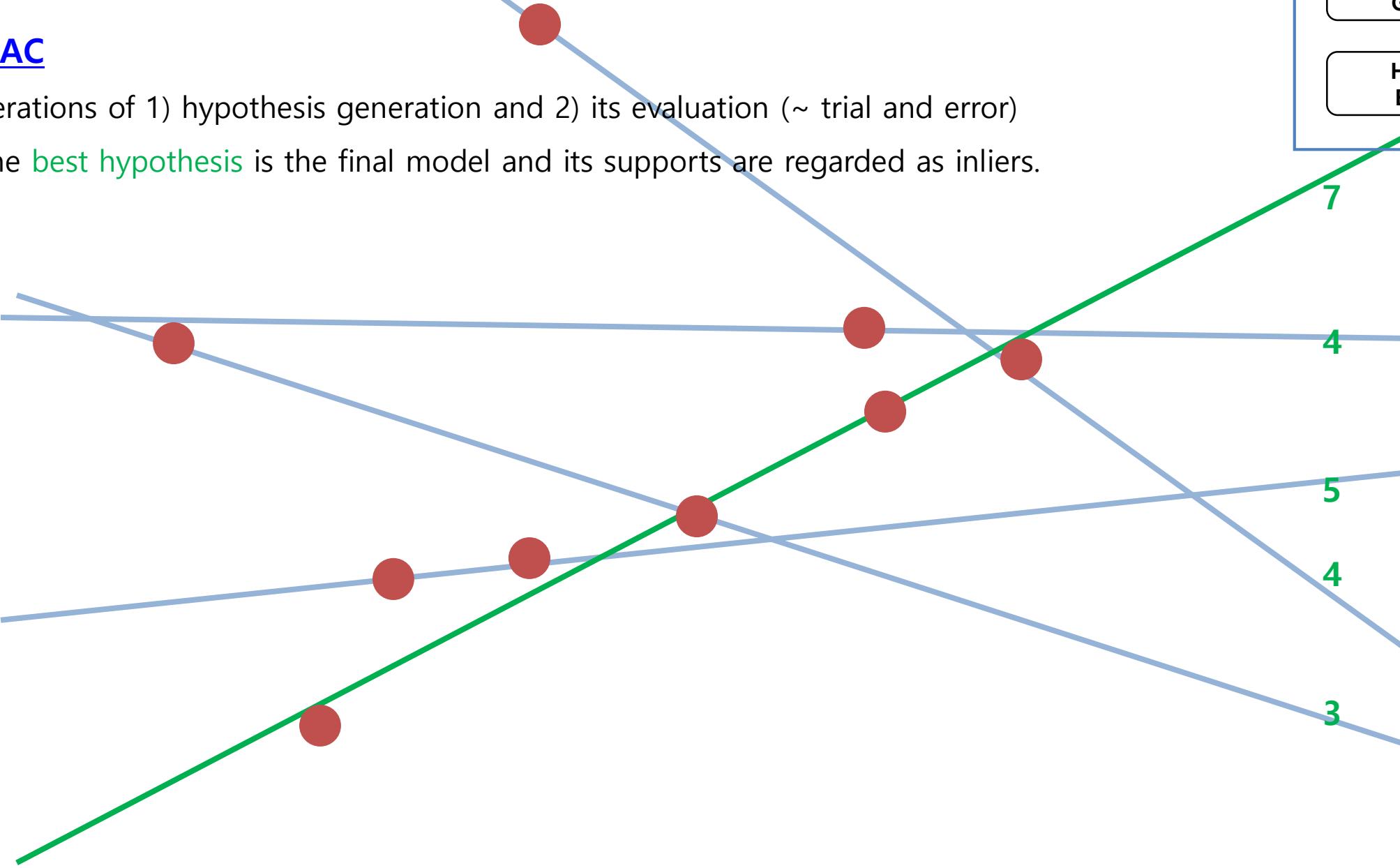
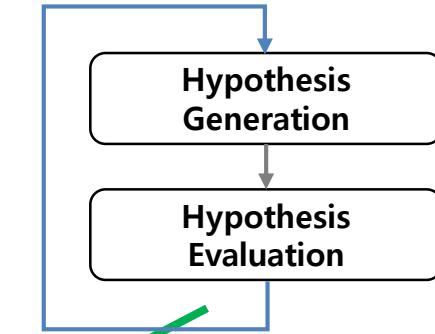
- e.g. In this example, the 2D line is supported by points within the given threshold τ .



Outlier Rejection) RANSAC (Random Sample Consensus; 1981)

- **RANSAC**

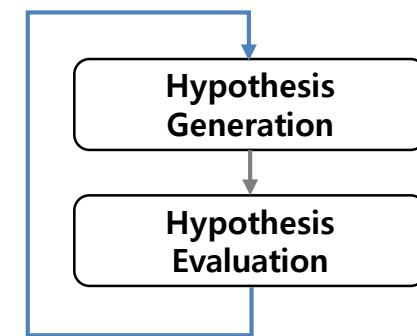
- Iterations of 1) hypothesis generation and 2) its evaluation (~ trial and error)
- The **best hypothesis** is the final model and its supports are regarded as inliers.



Outlier Rejection) RANSAC (Random Sample Consensus; 1981)

- **RANSAC**

- Iterations of 1) hypothesis generation and 2) its evaluation (~ trial and error)
- **Q) How many iterations t are required?**



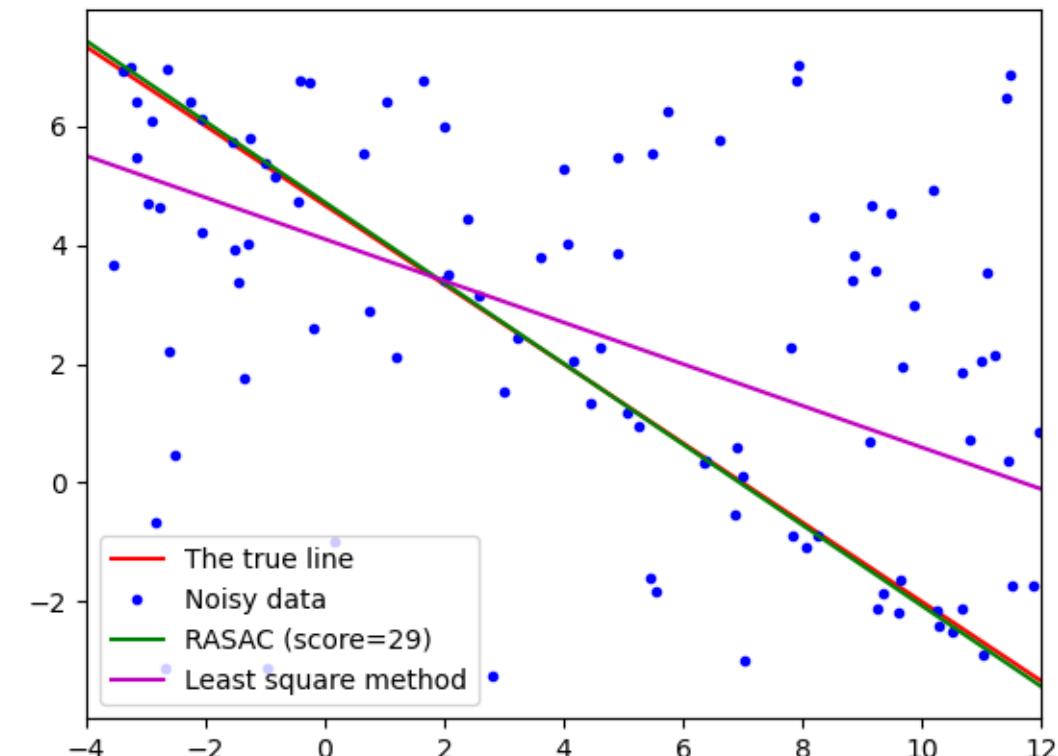
- Parameters and assumptions
 - s : Success probability (confidence level)
 - d : The number of samples for model generation
 - γ : Inlier ratio
- Success criteria: **Selecting d samples from inliers within t iterations**
 - Failure sampling: $1 - \gamma^d$
 - Failure probability: $1 - s = (1 - \gamma^d)^t$
- **The minimum number of iteration for success:** $t = \frac{\log(1-s)}{\log(1-\gamma^d)}$

Outlier Rejection) RANSAC (Random Sample Consensus; 1981)

- Example) **Line fitting with RANSAC** [line_fitting_ransac.py]

- Model definition: 2D line $\mathbf{m} = [a, b, c]^\top$ ($ax + by + c = 0$; $a^2 + b^2 = 1$)
- Model generation: Two points can generate to a line.
- Model evaluation: Geometric distance $e(\mathbf{x}_i; \mathbf{m}) = \frac{ax_i + by_i + c}{\sqrt{a^2 + b^2}}$

```
def generate_line(pts):  
    # Line model: y = ax + b  
    a = (pts[1][1] - pts[0][1]) / (pts[1][0] - pts[0][0])  
    b = pts[0][1] - a * pts[0][0]  
  
    # Line model: ax + by + c = 0 (a^2 + b^2 = 1)  
    line = np.array([a, -1, b])  
    return line / np.linalg.norm(line[:2])  
  
def evaluate_line(line, p):  
    a, b, c = line  
    x, y = p  
    return np.fabs(a*x + b*y + c)
```



Outlier Rejection) RANSAC (Random Sample Consensus; 1981)

- Example) **Line fitting with RANSAC** [line_fitting_ransac.py]

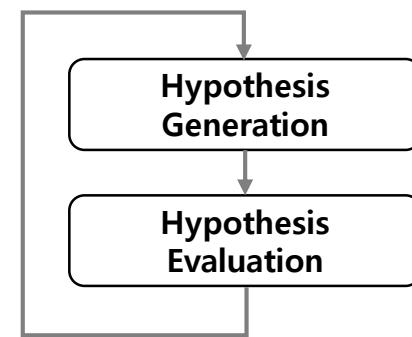
```
def generate_line(pts):
    ...
    ...

def evaluate_line(line, p):
    ...

def fit_line_ransac(data, n_sample, ransac_trial, ransac_threshold):
    best_score = -1
    best_model = None
    for _ in range(ransac_trial):
        # Step 1: Hypothesis generation
        sample = random.choices(data, k=n_sample)
        model = generate_line(sample)

        # Step 2: Hypothesis evaluation
        score = 0
        for p in data:
            error = evaluate_line(model, p)
            if error < ransac_threshold:
                score += 1
        if score > best_score:
            best_score = score
            best_model = model

    return best_model, best_score
```



Outlier Rejection) RANSAC (Random Sample Consensus; 1981)

- Example) Line fitting with RANSAC [line_fitting_ransac.py]

```
if __name__ == '__main__':
    true_line = np.array([2, 3, -14]) / np.sqrt(2*2 + 3*3) # The line model: a*x + b*y + c = 0 (a^2 + b^2 = 1)
    data_range = np.array([-4, 12])
    data_num = 100
    noise_std = 0.2
    outlier_ratio = 0.7

    # Generate noisy points with outliers
    line2y = lambda line, x: (line[0] * x + line[2]) / -line[1] # ax + by + c = 0 -> y = (ax + c) / -b
    y_range = sorted(line2y(true_line, data_range))
    data = []
    for _ in range(data_num):
        x = np.random.uniform(*data_range)
        if np.random.rand() < outlier_ratio:
            y = np.random.uniform(*y_range) # Generate an outlier
        else:
            y = line2y(true_line, x) # Generate an inlier
            x += np.random.normal(scale=noise_std) # Add Gaussian noise
            y += np.random.normal(scale=noise_std)
        data.append((x, y))
    data = np.array(data)

    # Estimate a line using RANSAC
    best_line, best_score = fit_line_ransac(data, 2, 100, 0.3) # log(1 - 0.999) / log(1 - 0.3^2) = 73

    # Estimate a line using OpenCV (for reference)
    # Note: OpenCV's line equation is a*x + b*y + c = 0
```

$$t > \frac{\log(1 - s)}{\log(1 - \gamma^d)} = \frac{\log(1 - 0.999)}{\log(1 - 0.3^2)} = 73$$

Review) Planar Homography

- Example) Planar image stitching [image_stitching.py]

```
# Load two images
img1 = cv.imread('../data/hill01.jpg')
img2 = cv.imread('../data/hill02.jpg')

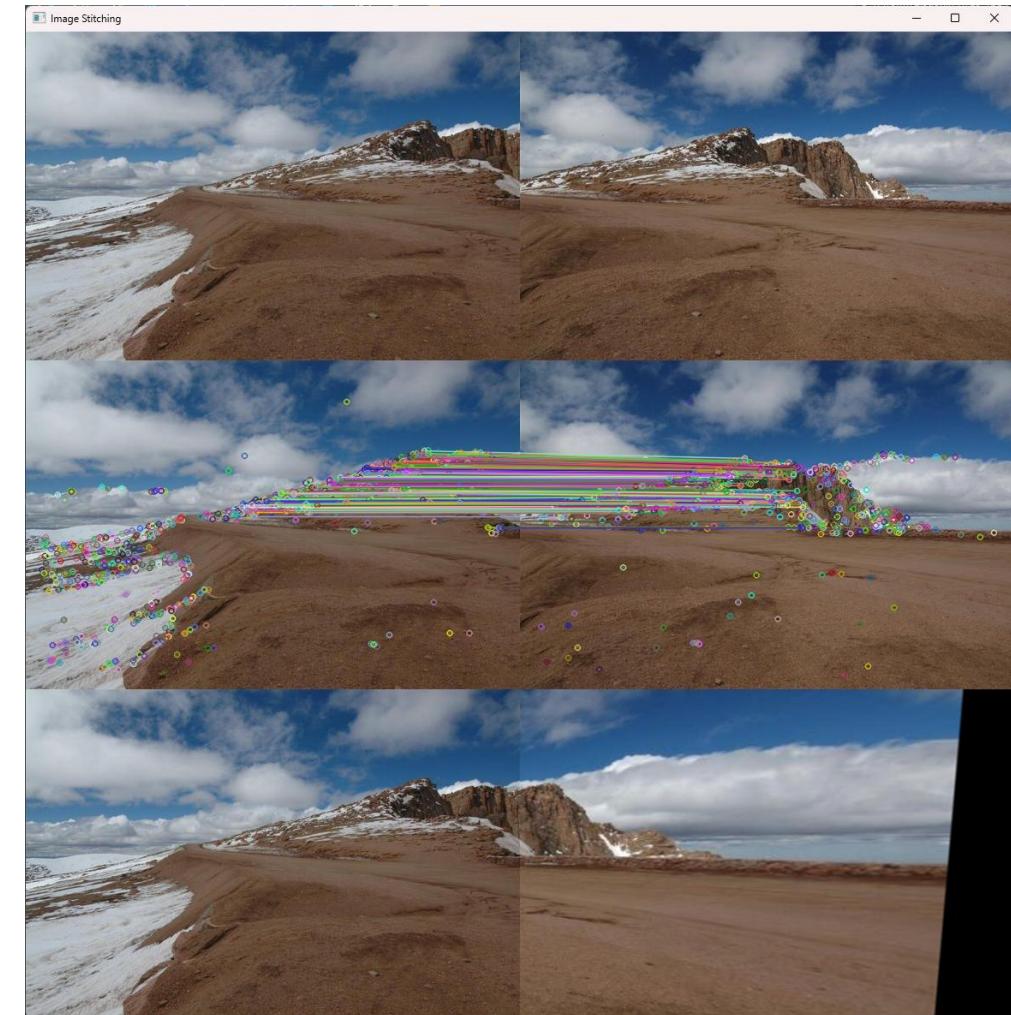
# Retrieve matching points
fdetector = cv.BRISK_create()
keypoints1, descriptors1 = fdetector.detectAndCompute(img1, None)
keypoints2, descriptors2 = fdetector.detectAndCompute(img2, None)

fmatcher = cv.DescriptorMatcher_create('BruteForce-Hamming')
match = fmatcher.match(descriptors1, descriptors2)

# Calculate planar homography and merge them
pts1, pts2 = [], []
for i in range(len(match)):
    pts1.append(keypoints1[match[i].queryIdx].pt)
    pts2.append(keypoints2[match[i].trainIdx].pt)
pts1 = np.array(pts1, dtype=np.float32)
pts2 = np.array(pts2, dtype=np.float32)

H, inlier_mask = cv.findHomography(pts2, pts1, cv.RANSAC)
img_merged = cv.warpPerspective(img2, H, (img1.shape[1]*2, img1.shape[0]))
img_merged[:, :img1.shape[1]] = img1 # Copy

# Show the merged image
img_matched = cv.drawMatches(img1, keypoints1, img2, keypoints2, match, None, None, None,
```



Outlier Rejection) RANSAC (Random Sample Consensus; 1981)

- Example) Planar homography estimation with RANSAC [image_stitching_implement.py]

```
from homography_estimation_implement import getPerspectiveTransform
from image_warping_implement import warpPerspective2

def findHomography(src, dst, n_sample, ransac_trial, ransac_threshold):
    ...

    if __name__ == '__main__':
        # Load two images
        ...

        # Retrieve matching points
        ...

        # Calculate planar homography and merge them
        ...

        H, inlier_mask = findHomography(pts2, pts1, 4, 1000, 2) # log(1 - 0.999) / log(1 - 0.3^4) = 849
        img_merged = warpPerspective2(img2, H, (img1.shape[1]*2, img1.shape[0]))
        img_merged[:, :img1.shape[1]] = img1 # Copy

        # Show the merged image
        img_matched = cv.drawMatches(img1, keypoints1, img2, keypoints2, match, None, None, None,
                                     matchesMask=inlier_mask) # Remove `matchesMask` if you want to show all ...
        merge = np.vstack((np.hstack((img1, img2)), img_matched, img_merged))
        cv.imshow(f'Planar Image Stitching with My RANSAC (score={sum(inlier_mask)})', merge)
        cv.waitKey(0)
        cv.destroyAllWindows()
```

Outlier Rejection) RANSAC (Random Sample Consensus; 1981)

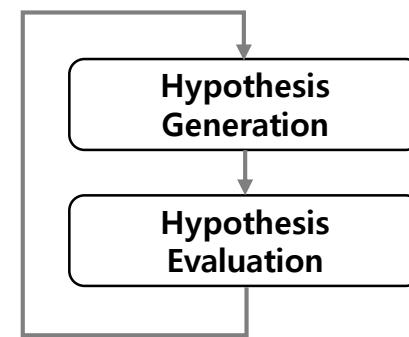
- Example) Homography estimation with RANSAC [image_stitching_implement.py]

```
def evaluate_homography(H, p, q):
    p2q = H @ np.array([[p[0]], [p[1]], [1]])
    p2q /= p2q[-1]
    return np.linalg.norm(p2q[:2].flatten() - q)      Note)  $e(\mathbf{x}, \mathbf{x}') = \|\mathbf{Hx} - \mathbf{x}'\|_2$ 

def findHomography(src, dst, n_sample, ransac_trial, ransac_threshold):
    best_score = -1
    best_model = None
    for _ in range(ransac_trial):
        # Step 1: Hypothesis generation
        sample_idx = random.choices(range(len(src)), k=n_sample)
        model = getPerspectiveTransform(src[sample_idx], dst[sample_idx])

        # Step 2: Hypothesis evaluation
        score = 0
        for (p, q) in zip(src, dst):
            error = evaluate_homography(model, p, q)
            if error < ransac_threshold:
                score += 1
        if score > best_score:
            best_score = score
            best_model = model

    # Generate the best inlier mask
    best_inlier_mask = np.zeros(len(src), dtype=np.uint8)
    for idx, (p, q) in enumerate(zip(src, dst)):
        if abs(evaluate_homography(best_model, p, q)) < ransac_threshold:
            best_inlier_mask[idx] = 1
```



Summary

▪ Feature Points

- Gradient-based: Harris corner, GFTT corner, SIFT, SURF, ...
- Intensity-based: FAST, ...
- DL-based: LIFT, SuperPoint, ...

▪ Feature Descriptors

- Real-valued: SIFT, SURF, ... (DL-based: LIFT, SuperPoint, ...)
- Binary-valued: BRIEF, ORB, ...

▪ Feature Matching

- Distance measures: Euclidean distance, cosine similarity, Hamming distance, ...
- Matching methods: Brute-force search, ANN search, ...
- DL-based: SuperGlue, ...

▪ Feature Tracking

- KLT tracker = GFTT detector + LK optical flow

▪ Outlier Rejection

- RANSAC

