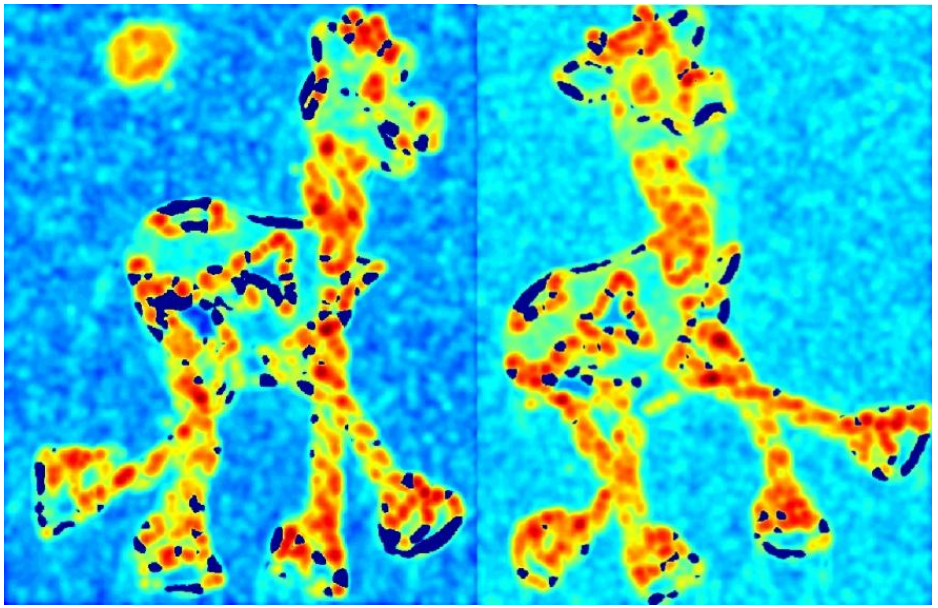
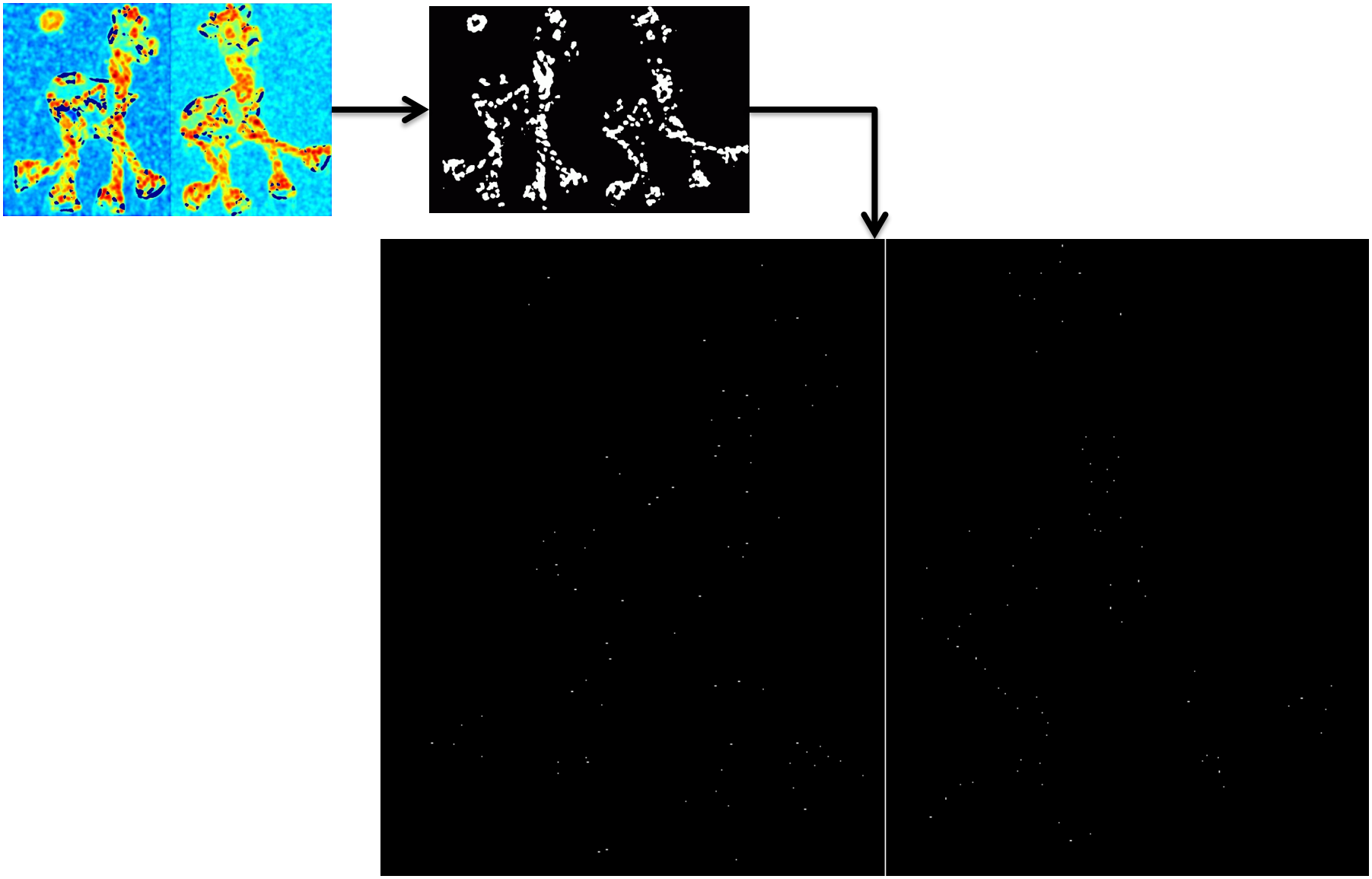
1) Corner Detection

- See Harris Corners Slides
- Links to existing implementations on course website
- Output should be a score for every pixel

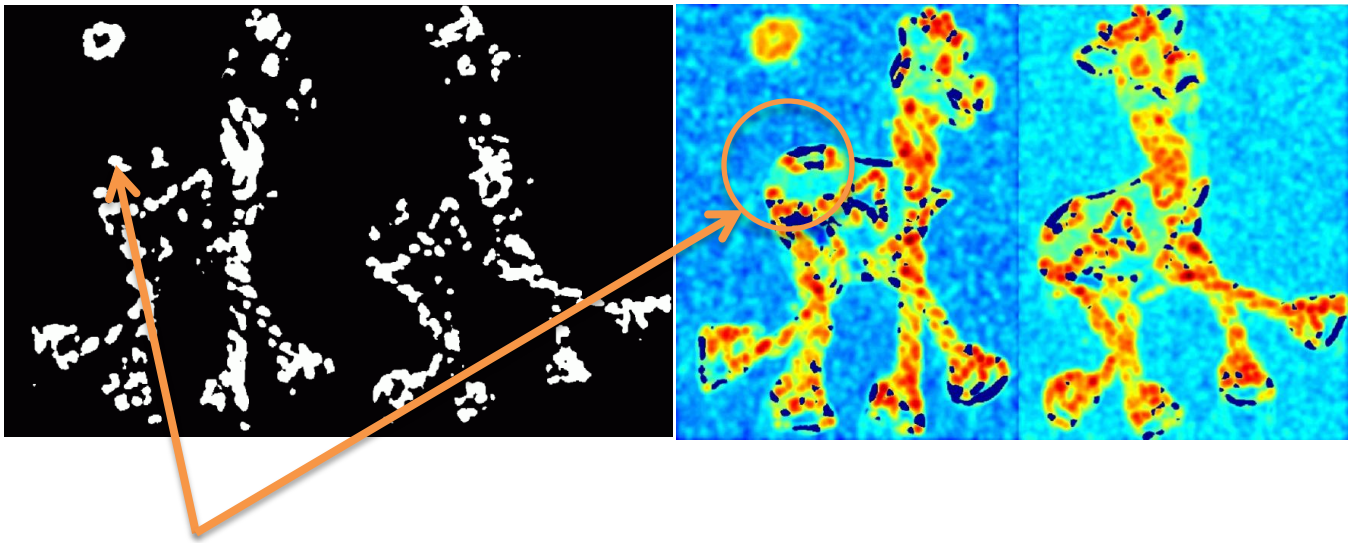


2) Adaptive Non-Maximal Suppression



2) Adaptive Non-Maximal Suppression

- Want a manageable number of corners to work with (Choose a number)
- Want evenly distributed arrangements of points

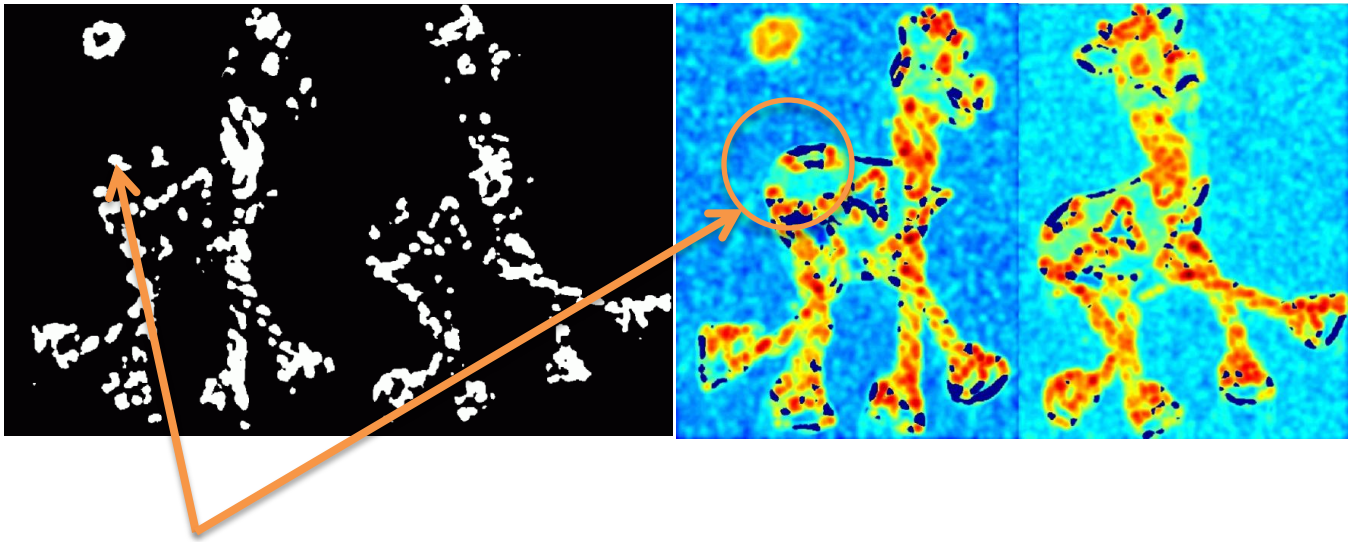


For every pixel **p** with a score greater than the corner threshold:

Check score of points within distance **r**

If **p** is the maximum within radius **r**, mark **p** as a corner

2) Adaptive Non-Maximal Suppression



For every pixel **p** with a score greater than the corner threshold:
Check score of points within distance **r**
If **p** is the maximum within radius **r**, mark **p** as a corner

---

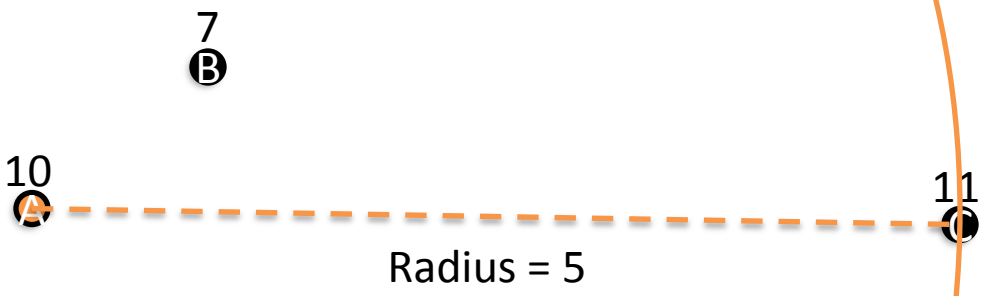
- Vary **r** until the desired number of corners are kept
  - Larger radius will suppress more pixels
  - Smaller radius will keep more

2) Adaptive Non-Maximal Suppression

- Equivalent Approach without Search of **r**
  - For each pixel, find the largest radius in which it would be considered a corner
    - Distance to the closest point with a greater score

2) Adaptive Non-Maximal Suppression

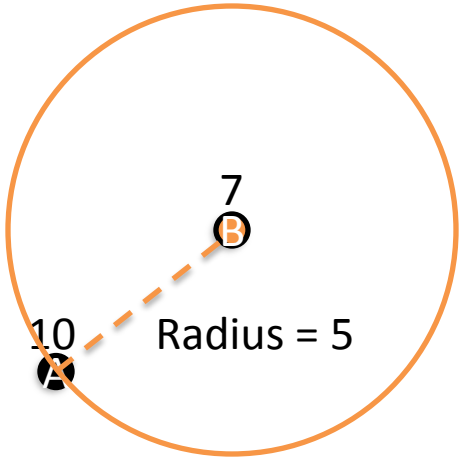
- Equivalent Approach without Search of **r**
  - For each pixel, find the largest radius in which it would be considered a corner
    - Distance to the closest point with a greater score



Point	Maximum Radius
A	5
B	
C	

2) Adaptive Non-Maximal Suppression

- Equivalent Approach without Search of **r**
  - For each pixel, find the largest radius in which it would be considered a corner
    - Distance to the closest point with a greater score



11
C

Point	Maximum Radius
A	5
B	1
C	

2) Adaptive Non-Maximal Suppression

- Equivalent Approach without Search of **r**
  - For each pixel, find the largest radius in which it would be considered a corner
    - Distance to the closest point with a greater score

Point	Maximum Radius
A	5
B	1
C	Infinite

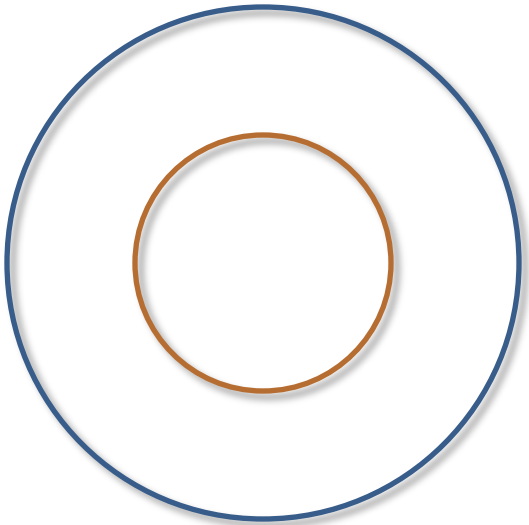
2) Adaptive Non-Maximal Suppression

- Equivalent Approach without Search of **r**
  - For each pixel, find the largest radius in which it would be considered a corner
    - Distance to the closest point with a greater score

Point	Maximum Radius
A	5
B	1
C	Infinite

- Algorithm
  - Sort by radius in decreasing order
  - Keep the top N as corners

- Why?
  - Once a point is marked as a corner, decreasing the radius Will only add corners, but not remove any already on



If the center score is greater than every value inside the blue ring, it must also be greater than every value inside the orange.

2) Adaptive Non-Maximal Suppression

- Let the maximum number of points to keep be 85.
- Let the list at right be an excerpt of the table shown on the previous slide for a larger example. Index represents the row number in the full matrix and the Max Radii have been sorted in decreasing order
- Then we want a radius size such that we keep at most 85.
- Since the 86th point has the same radius as the 85th, however, we can't get exactly 85 points.
- Instead, the desired radius would be 11 and we would get 83 points after suppression.

Index	Maximum Radius
78	20
79	18
80	18
81	13
82	12
83	11
84	10
85	10
86	10
87	7

2) Adaptive Non-Maximal Suppression

[y x rmax] = anms(cimg, max_pts);

cimg = corner strength map

max_pts = number of corners desired

[y,x] = coordinates of corners

rmax = suppression radius used to get max_pts corners

3)Extract Feature Descriptors

- Want an 8x8 patch describing each corner
- Algorithm:
    - 40x40 window around the corner
    - Downsize to 8x8 (blur and then sample every 5^{th} pixel, see image pyramid lecture)
    - Normalize 8x8 patch to mean 0 and standard deviation of 1

- Note in class the lecture discussed using the dominant direction to get rotational invariance. For this project that is not required.

3)Extract Feature Descriptors

[p] = feat_desc(im, y, x);

im = double HxW array (grayscale image) with values in the range 0-255.
y = nx1 vector representing the row coordinates of corners
x = nx1 vector representing the column coordinates of corners

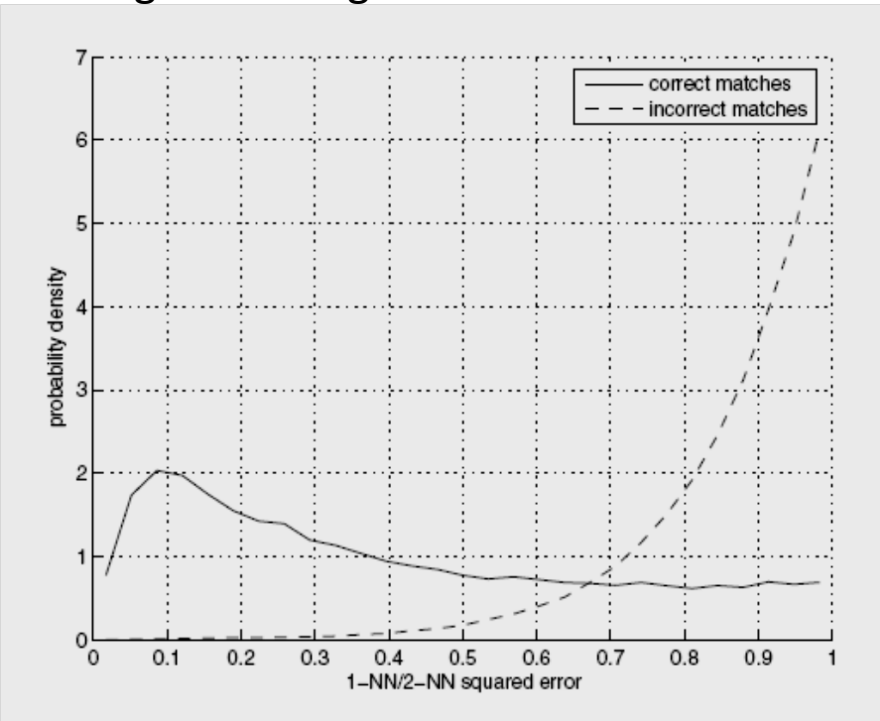
p = 64xn matrix of double values with column i being the 64 dimensional descriptor
(8x8 grid linearized) computed at location (xi,yi) in im

4) Descriptor Matching

• Compute SSD between all pairs of 8x8 descriptors in one image to descriptors in second image

• For each corner, find the 2 nearest neighbors (using SSD from above as the distance) in the second image

• If the ratio (SSD of 1st match/SSD of 2nd match) < thresh
    • Propose the nearest neighbor as a good match



4) Descriptor Matching

[m] = feat_match(p1,p2);

p1 = 64xn1 matrix of double values in the same format as the output from function feat_desc above.

p2 = 64xn2 matrix of double values in the same format as the output from function feat_desc above.

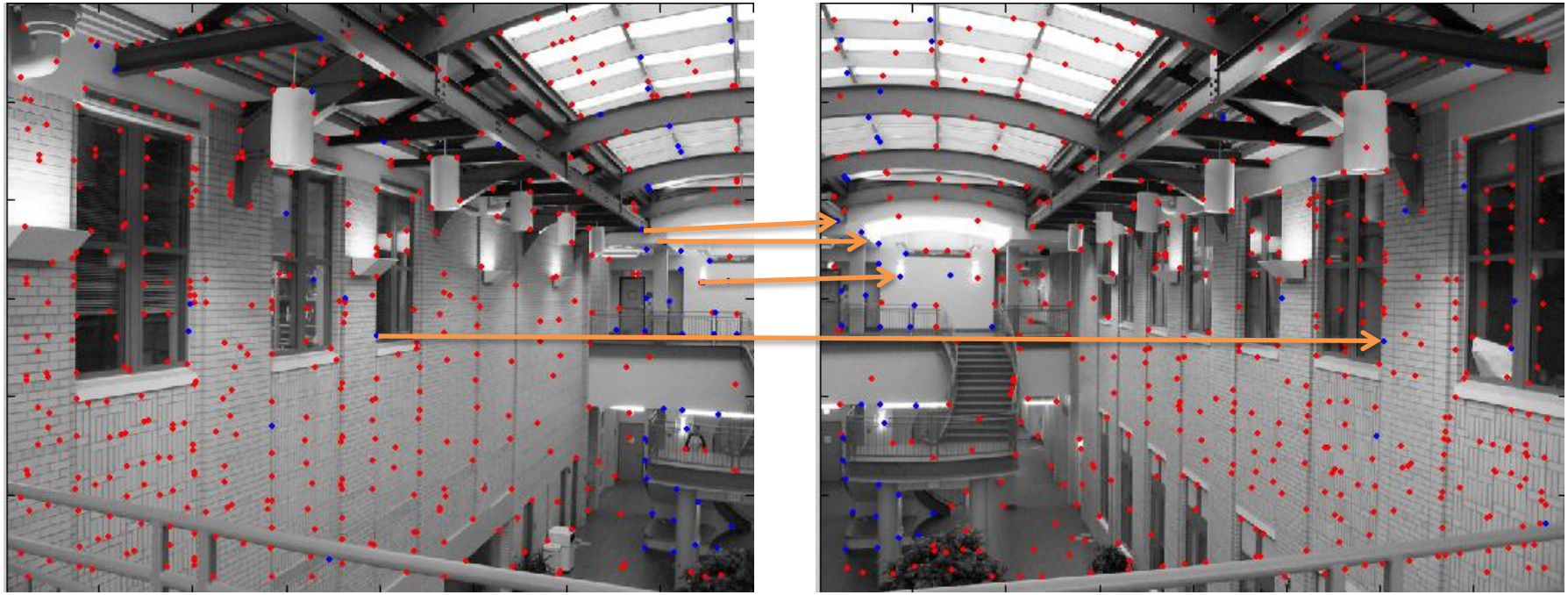
m = n1x1 vector of integers where m(i) points to the index of the descriptor in p2 that matches with the descriptor p1(:,i). If no match is found for feature i, you should put m(i)=-1.

5) RANSAC

- Want to use corresponding corners to calculate a projective transformation
- Don’t want false matches to skew the transform
- RANSAC Algo:
    - Randomly select groups of points to calculate many possible transforms
    - Have all points vote for transforms they agree with
    - Trust the points giving the transform with the most votes
        - Also trust the points that voted for that transform
    - Refine the transform estimate by calculating homography using all of the trusted matches

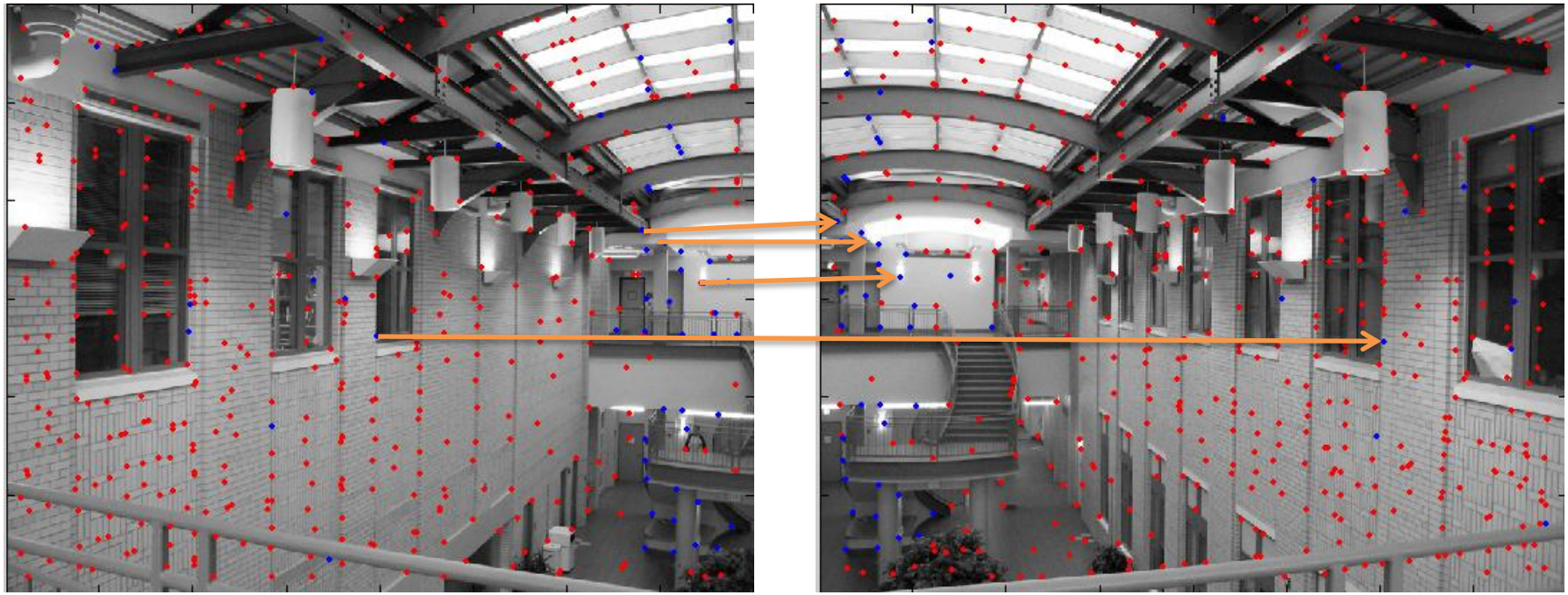
5) RANSAC

Randomly select 4 matches



5) RANSAC

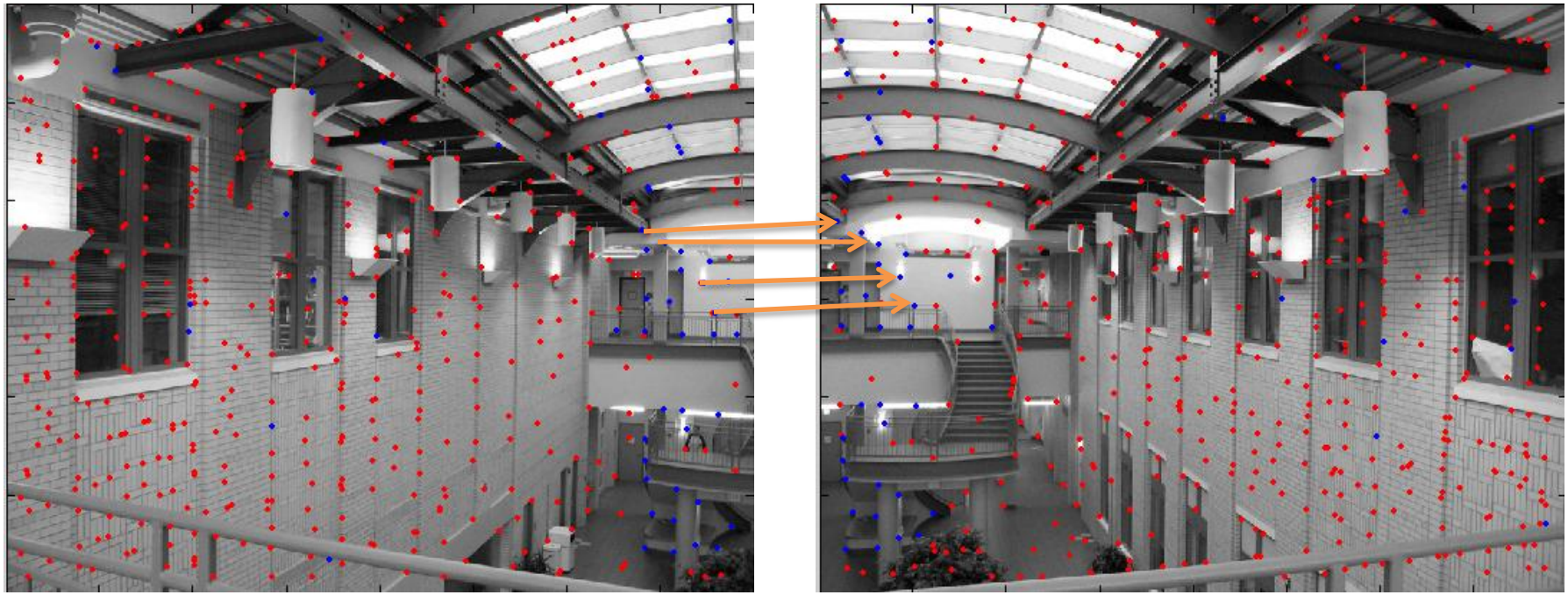
Calculate Homography: est_homography.m
Apply Homography to all points: apply_homography.m
Count Number of points that mapped close to match in the other image



Since these 4 contain an incorrect match, the estimated transform will be wrong. Therefore not many of the points will map to their match

5) RANSAC

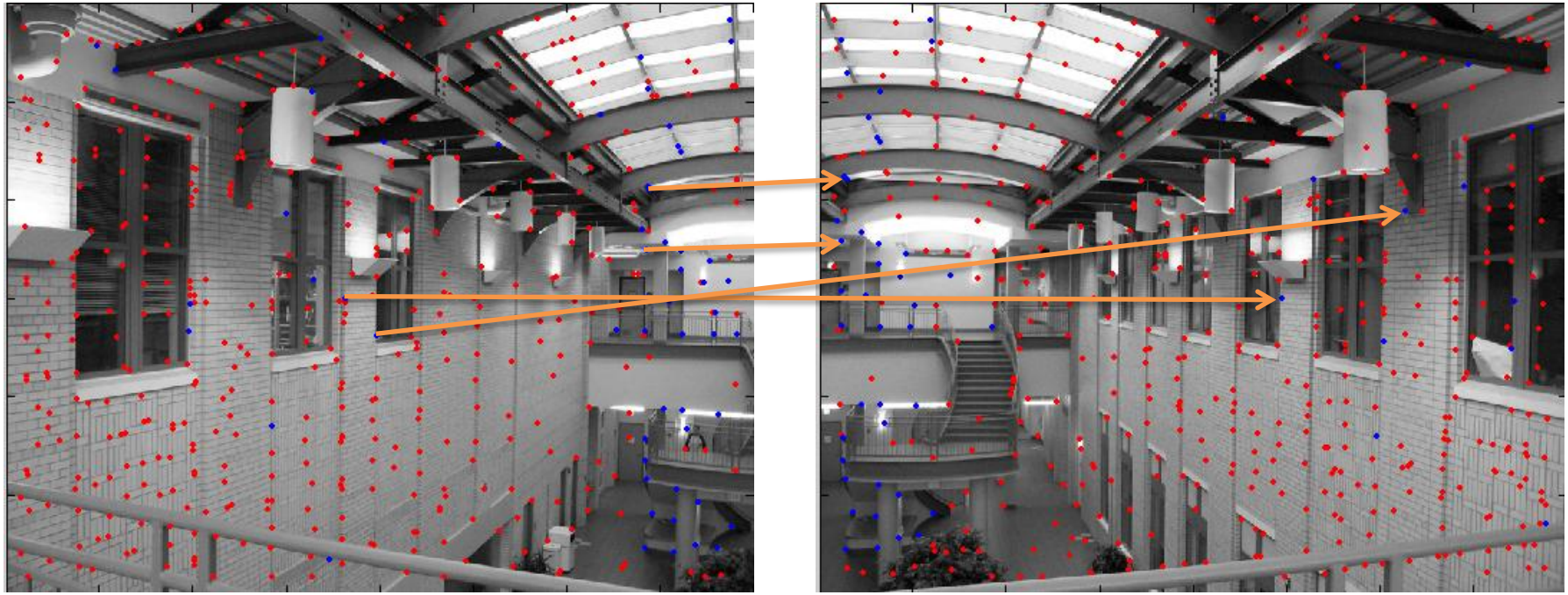
Keep randomly choosing 4 sets until probability at least 1 used all correct matches is high.



Since these 4 are correct matches, the transform generated should explain the location of the other correct matches and get a high vote.

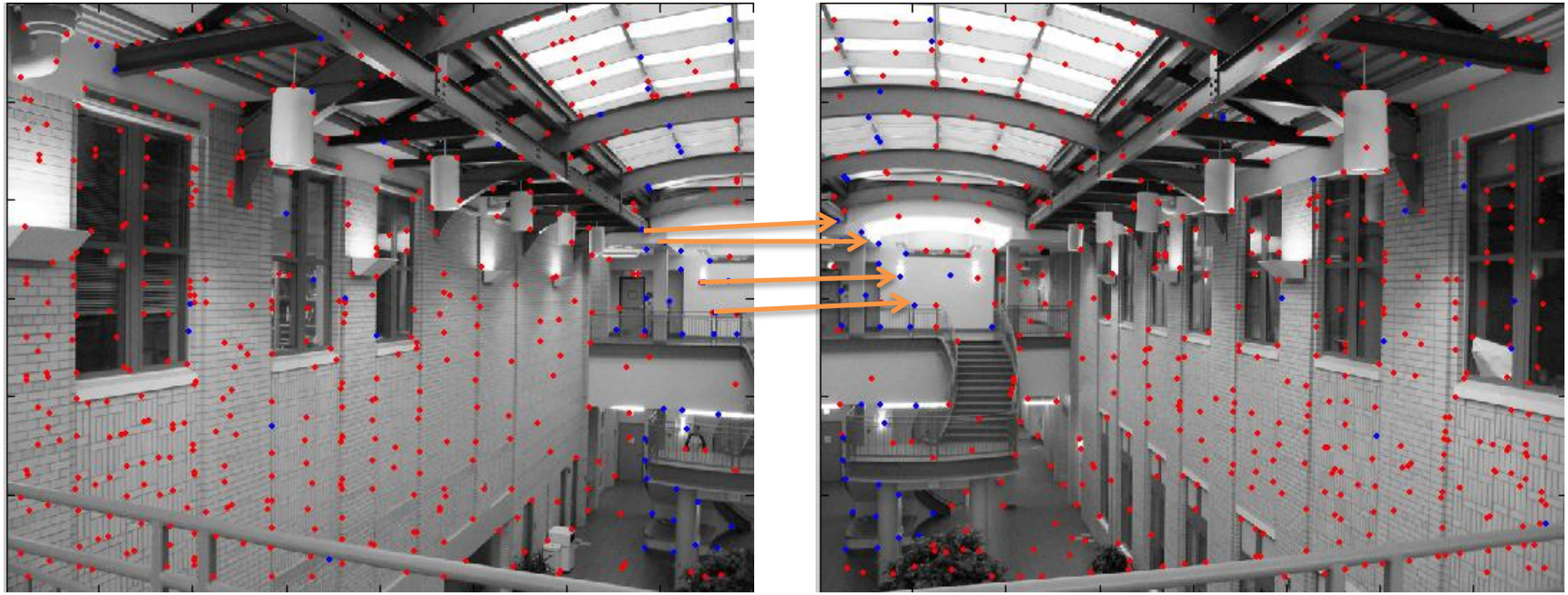
5) RANSAC

Keep randomly choosing 4 sets until probability at least 1 used all correct matches is high.



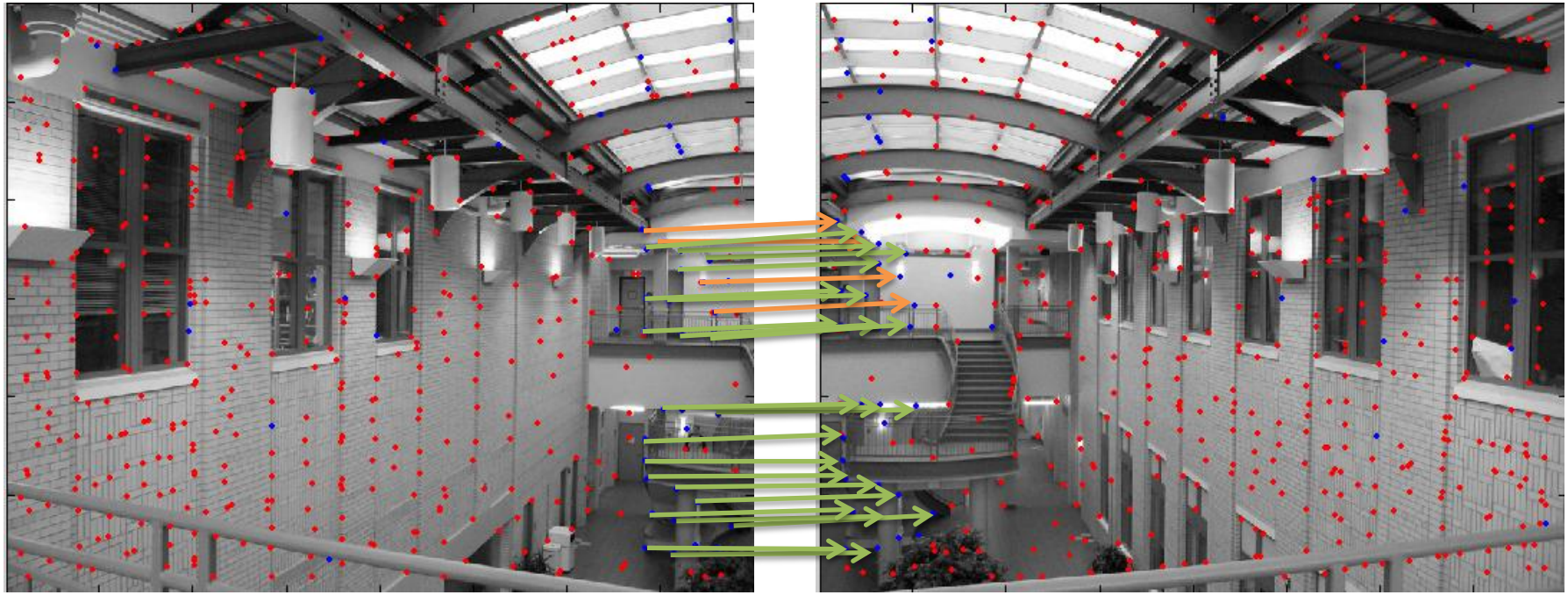
5) RANSAC

Take the highest voted set and find all points that voted for them:



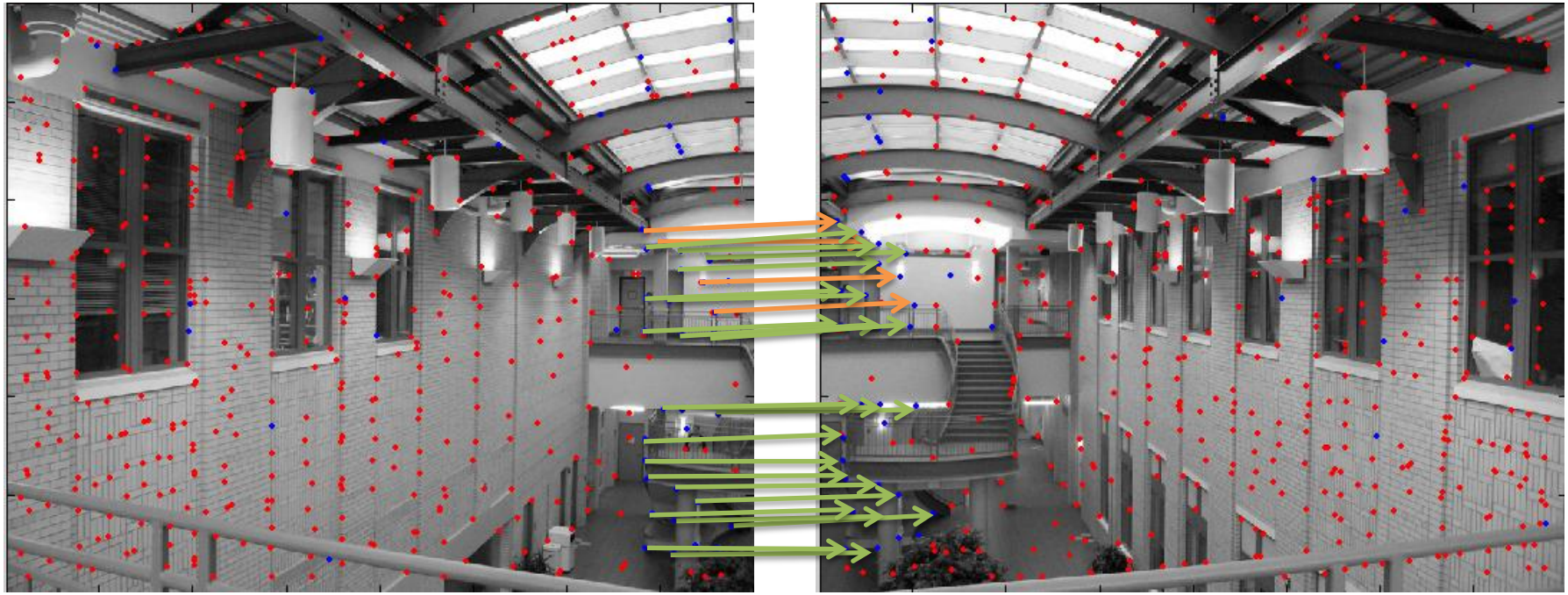
5) RANSAC

Take the highest voted set and find all points that voted for them:



5) RANSAC

Estimate a more accurate homography using all of the correctly matching points
Over constrained => Least Squares Solution (est_homography takes care of this)



5) RANSAC

[H,inlier_ind] = ransac_est_homography(y1, x1, y2, x2, thresh);
y1,x1,y2,x2 =corresponding point coordinate vectors Nx1 such that (x1i,y1i) matches (x2i,y2i) after a preliminary matching. (Blue points in example image)

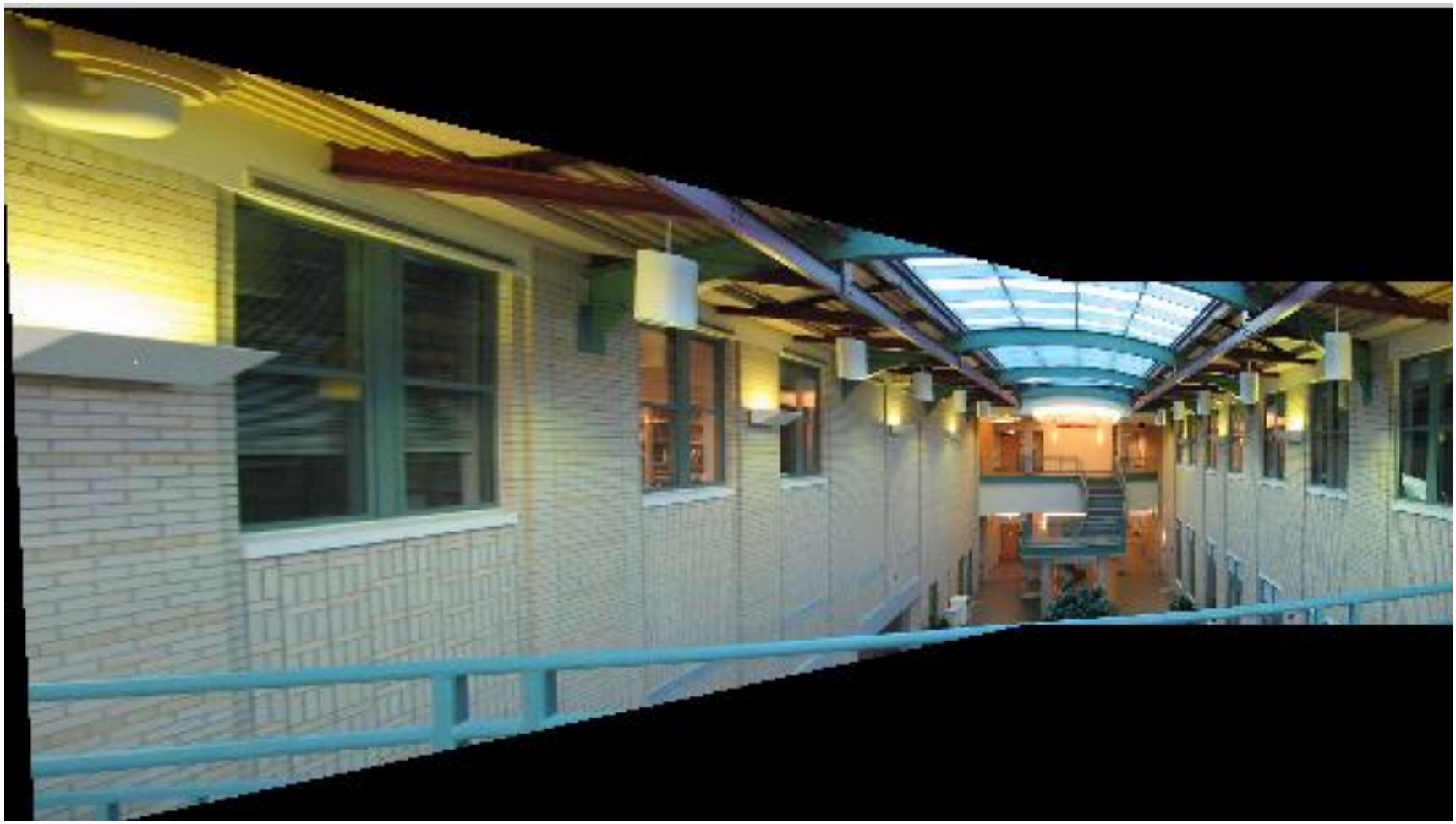
thresh = the threshold on distance used to determine if transformed points agree

H = the 3x3 matrix computed in final step of RANSAC.

inlier_ind = nx1 vector with indices of points in the arrays x1,y1,x2,y2 that were found to be the inliers.

6) Stitch together

- Apply transform to images to map them into the reference image.
- Create a large image that includes all of the transformed points from all of the images
- Optionally blend them together for seamless transitions



6) Stitch together

img_mosaic = mymosaic(img_input);

img_input i= cell array of color images (HxWx3 uint8 values in the range [0,255]).

img_mosaic =output mosaic HxWx3 array of uint8 values in the range [0,255].