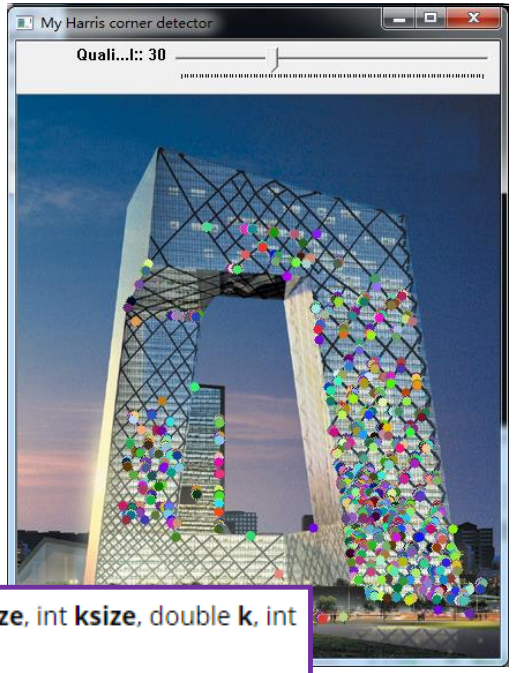
Feature2d Module

- Corner Detection
  - Using OPENCV to detection corners
  - We can detect corners in an image using the corner detectors in OPENCV
  - And can also select some better corners by adjusting quality threshold



C++: void `cornerHarris`(InputArray **src**, OutputArray **dst**, int **blockSize**, int **ksize**, double **k**, int **borderType**=BORDER_DEFAULT)

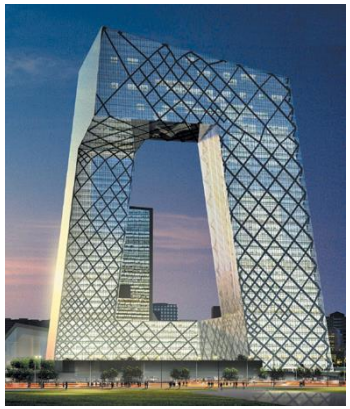
Parameters:
- **src** – Input single-channel 8-bit or floating-point image.
- **dst** – Image to store the minimal eigenvalues. It has the type `CV_32FC1` and the same size as `src`
- **blockSize** – Neighborhood size (see the details on `cornerEigenValsAndVecs()`).
- **ksize** – Aperture parameter for the `Sobel()` operator.
- **borderType** – Pixel extrapolation method. See `borderInterpolate()` .

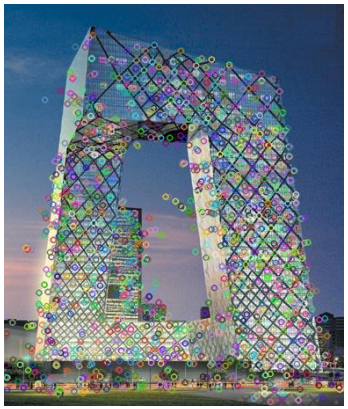
Feature2d Module

- Features detection
  - There are some feature descriptors in OPENCV
    - SIFT, SURF, ……
  - We can detect the key points in an image

Use the DescriptorExtractor interface in order to find the feature vector correspondent to the keypoints. Specifically

- Use SurfDescriptorExtractor and its function compute to perform the required calculations.
- Use a BFMatcher to match the features vector
- Use the function drawMatches to draw the detected matches.

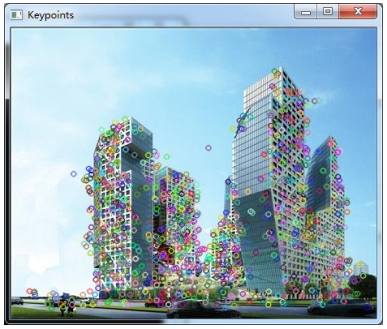


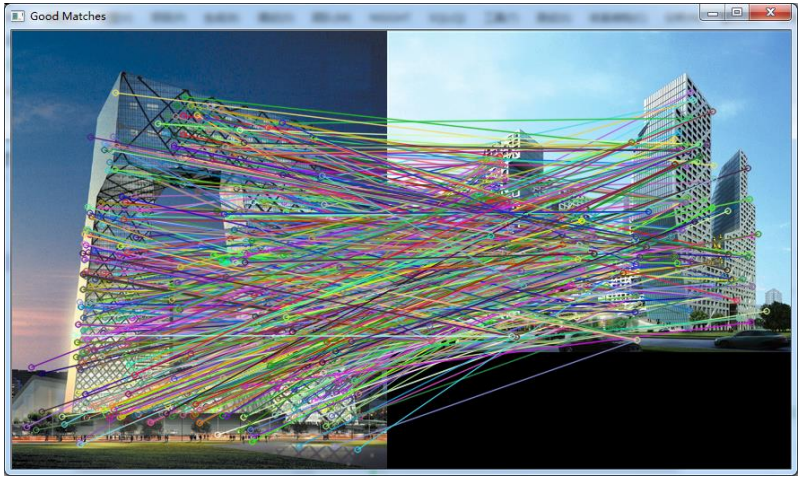


Feature2d Module

- Matching
  - For two images, we can detect k
  - So after we have features of two image, we can know whether they match or whether they are similar

```
BFMatcher matcher(NORM_L2);
std::vector< DMatch > matches;
matcher.match( descriptors_1, descriptors_2, matches );
```





Clustering: group together similar points and represent them with a single token

Clustering: group together similar points and represent them with a single token

Clustering: group together similar points and represent them with a single token

Key Questions:

1) What makes two points/images/patches similar?

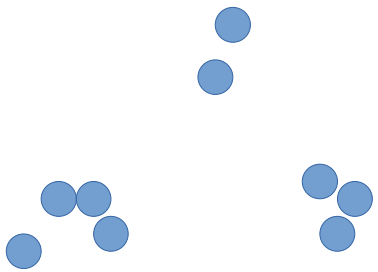
2) How do we determine the grouping from pairwise similarities?

K-means algorithm

$$\underset{S,\mu_i,i=1..K}{\operatorname{argmin}} \sum_{i=1}^{K} \sum_{x \in S_i} ||x - \mu_i||^2$$

We wish to partition the data into K sets S = {S_1, S_2, … S_K} with corresponding centers μ_i

Partition such that variance in each partition is as low as possible

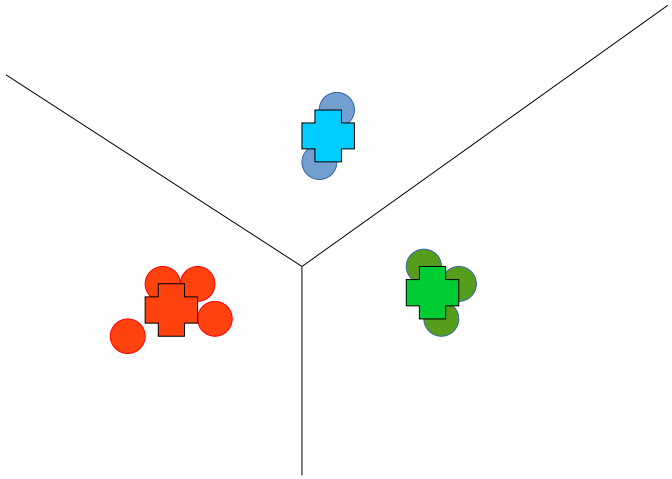


K-means algorithm

$$\underset{S,\mu_i,i=1..K}{\operatorname{argmin}} \sum_{i=1}^{K} \sum_{x \in S_i} ||x - \mu_i||^2$$

We wish to partition the data into K sets S = {S_1, S_2, … S_K} with corresponding centers μ_i

Partition such that variance in each partition is as low as possible



K-means algorithm

1. Randomly select K centers

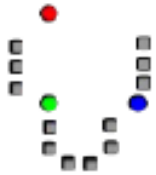
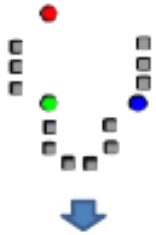


Illustration: http://en.wikipedia.org/wiki/K-means_clustering

K-means algorithm

1. Randomly select K centers



2. Assign each point to nearest center

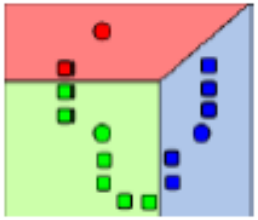
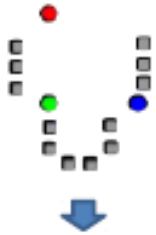


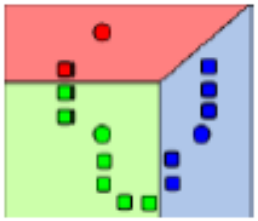
Illustration: http://en.wikipedia.org/wiki/K-means_clustering

K-means algorithm

1. Randomly select K centers



2. Assign each point to nearest center



3. Compute new center (mean) for each cluster

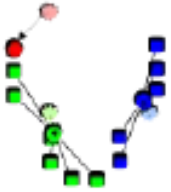


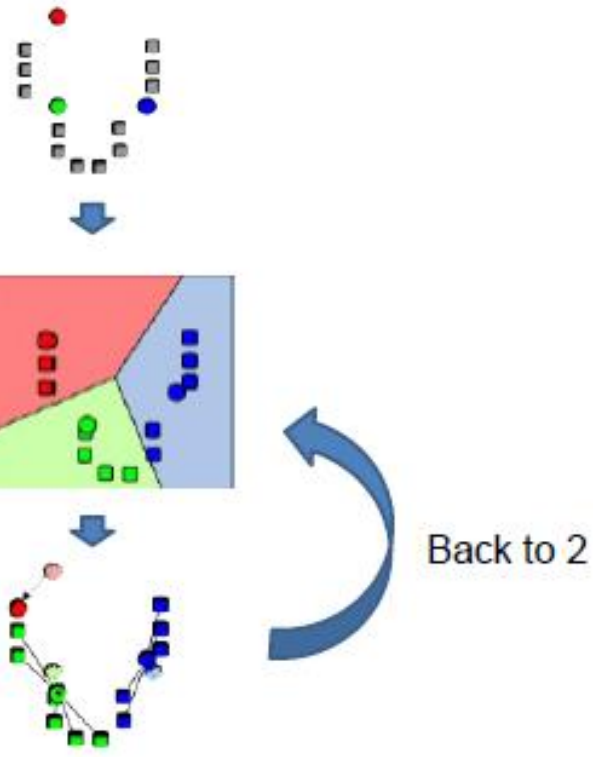
Illustration: http://en.wikipedia.org/wiki/K-means_clustering

K-means algorithm

1. Randomly select K centers

2. Assign each point to nearest center

3. Compute new center (mean) for each cluster



Back to 2

Illustration: http://en.wikipedia.org/wiki/K-means_clustering

K-means algorithm

1. Initialize K centers μ_i (usually randomly)
2. Assign each p $S^t = \underset{S}{\operatorname{argmin}} \sum_{i=1}^{K} \sum_{x \in S_i} ||x - \mu_i||^2$
3. Update cluste $\mu^t = \underset{\mu_i, i=1..K}{\operatorname{argmin}} \sum_{i=1}^{K} \sum_{x \in S_i} ||x - \mu_i||^2$ embers
4. Repeat 2-3 until convergence (t = t+1)

Conclusions: K-means

Good

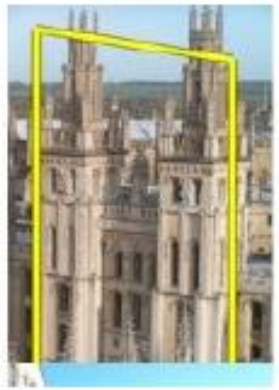
- Finds cluster centers that minimize conditional variance (good representation of data)
- Simple to implement, widespread application

Bad

- Sensitive to starting locations
- Need to choose K
- All clusters have the same parameters (e.g., distance measure is non-adaptive)

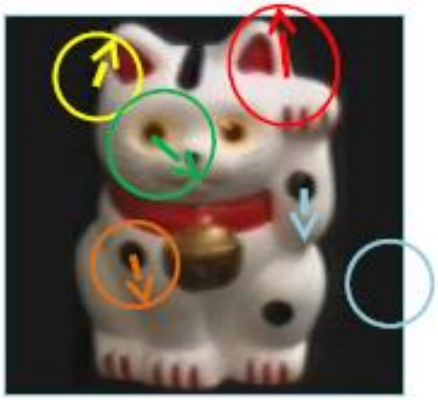
Application of K-means

How to quickly find images in a large database that match a given image region?

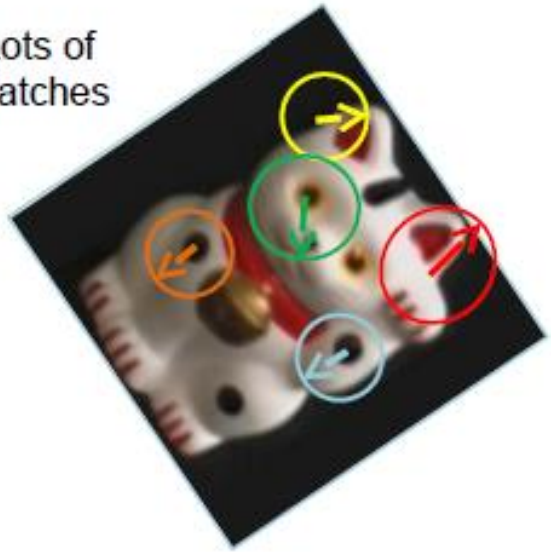


Simple idea

See how many SIFT keypoints are close to SIFT keypoints in each other image



Lots of Matches



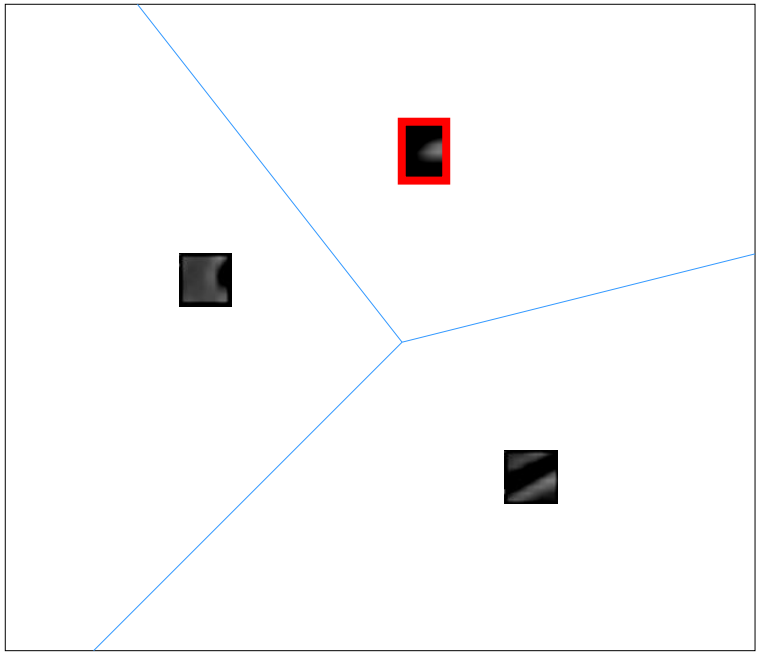
Few or No Matches

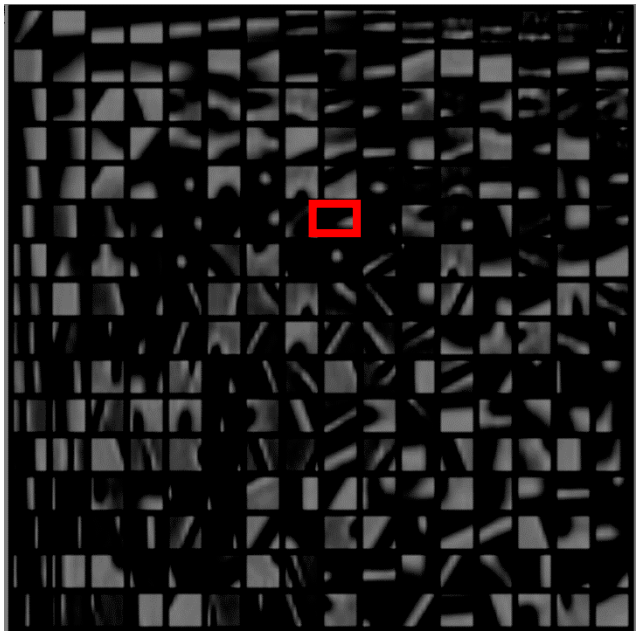
But this will be really, really slow!

Bag of Visual Words

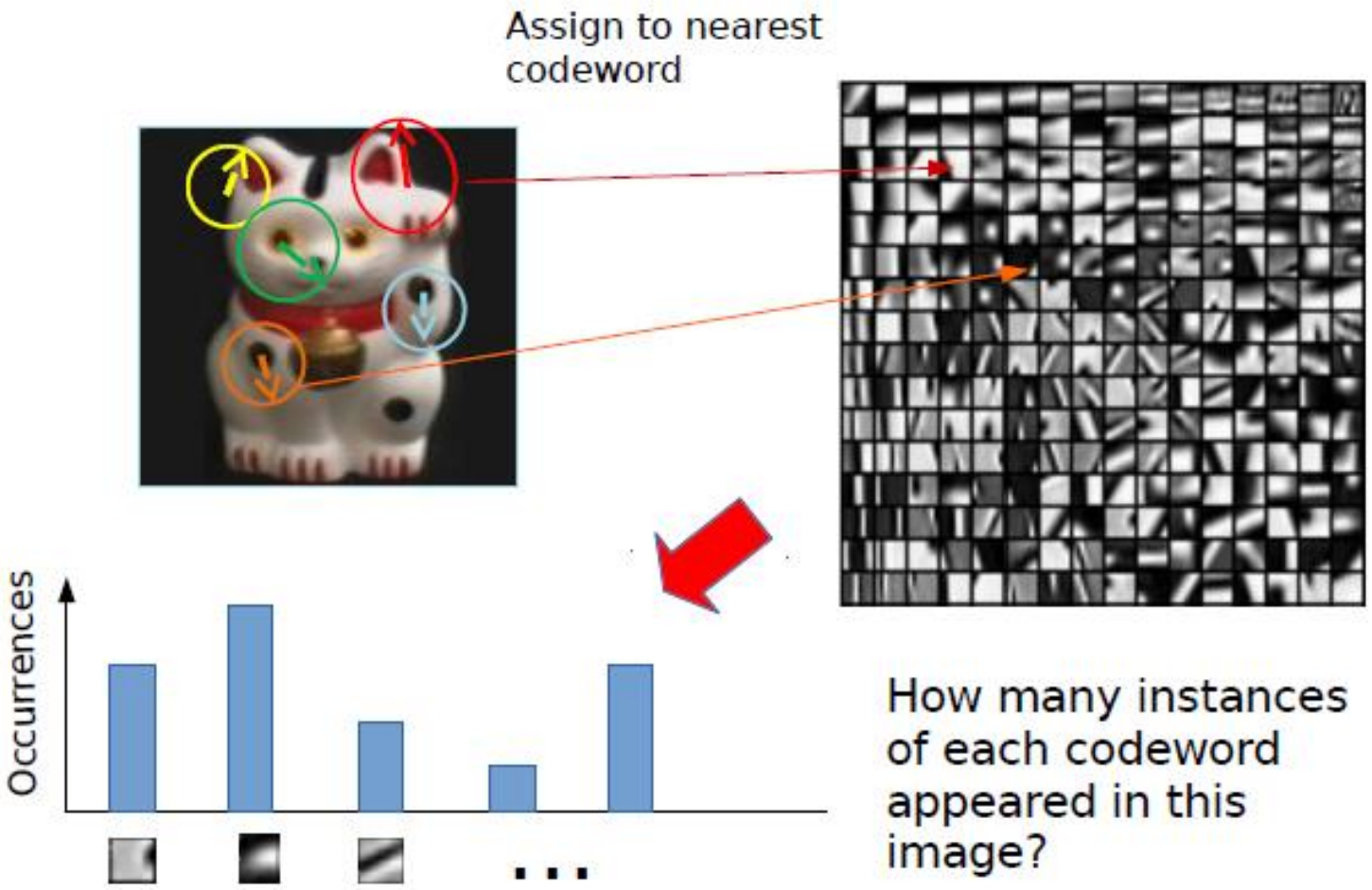
- Cluster the keypoint descriptors into a managable vocabulary size
- Assign each descriptor to a cluster number

Codebook of cluster centers





Bag of Visual Words



Bag of Visual Words

- Each image is represented by a histogram of codeword frequencies
- Similar images should have similar histograms